Homework 8

1. Write a program in the WHILE-language that calculates the factorial function.

```
write "factorial: ";
read n;
minus1 := 1;
while n > 0 do {
minus1 := minus1 * n;
n := n - 1
};
write "Result: ";
write minus1 ;
write "\n"
```

2. What optimisations could a compiler perform when compiling a WHILE-program?

   - peephole optimisations (more specific instructions)
   - common sub-expression elimination
   - constant folding / constant propagation (that is calculate the result of $3 + 4$ already during compilation)
   - tail-recursion optimisation cannot be applied to the WHILE language because there are no recursive functions

3. What is the main difference between the Java assembler (as processed by Jasmin) and Java Byte Code?

   The main difference is that the j-files have symbols for places where to jump, while class files have this resolved to concrete addresses (or relative jumps). That is what the assembler has to generate.

4. Remember symbolic labels in the Jasmin-assembler are meant to be used for jumps (like in loops or if-conditions). Assume you generated a Jasmin-file with some redundant labels, that is some labels are not used in your code for any jumps. For example `L_begin` and `L_end` are not used in the following code-snippet:

```
L_begin:
ldc 1
ldc 2
ldc 3
imul
ldc 4
ldc 3
isub
iadd
iadd
L_end:
```

Do these redundant labels affect the size of the generated JVM-code? (Hint: What are the labels translated to by the Jasmin-assembler?).

> The answer is no. The reason is that assemblers calculate for labels either relative or explicit adresses, which are then used in the JVM-byte-code. Relative addresses are like "jump 10 bytes forward" or "12 bytes backward". So additional labels do not increase the size of the generated code.

5. Consider the following Scala snippet. Are the two functions `is_even` and `is_odd` tail-recursive?

```
def is_even(n: Int) : Boolean = {
    if (n == 0) true else is_odd(n - 1)
}

def is_odd(n: Int) : Boolean = {
    if (n == 0) false
    else if (n == 1) true else is_even(n - 1)
}
```

Do they cause stack-overflows when compiled to the JVM (for example by Scala)?

> Scala cannot generate jumps in between different methods (to which functions are compiled to). So cannot eliminate the tail-calls. Haskell for example can do this because it compiles the code in a big "blob" inside a main-method (similar to the WHILE language).

6. Explain what is meant by the terms lazy evaluation and eager evaluation.

> Lazy evaluation only evaluates expressions when they are needed and if they are needed twice, the results will be re-used. Eager evaluation immediately evaluates expressions, for example if they are arguments to function calls or allocated to variables.

7. **(Optional)** This question is for you to provide regular feedback to me: for example what were the most interesting, least interesting, or confusing parts in this lecture? Any problems with my Scala code? Please feel free to share any other questions or concerns. Also, all my material is ~~crap~~ imperfect. If you have any suggestions for improvement, I am very grateful to hear.

   If *you* want to share anything (code, videos, links), you are encouraged to do so. Just drop me an email or send a message to the Forum.