Coursework 3

This coursework is worth 5% and is due on 28 November at 16:00. You are asked to implement a parser for the WHILE language and also an interpreter. You should use the lexer from the previous coursework for the parser.

Question 1 (marked with 1%)

Design a grammar for the WHILE language and give the grammar rules. The main categories of non-terminal should be:

- arithmetic expressions (with the operations from the previous coursework, such as `+`, `*` and so on)
- boolean expressions (such as `<`, `!=` and so on)
- single statements (such as `skip`, assignments, `if`s, `while`-loops and so on)
- compound statements separated by semicolons
- blocks which are enclosed in curly parentheses

Question 2 (marked with 2%)

You should implement a parser for the WHILE language using parser combinators. Be careful that the parser takes as input a stream, or list, of tokens generated by the tokenizer from the previous coursework. Your parser should be able to handle the WHILE programs in Figures 2 and 3. In addition give the parse tree for the statement:

```
if (a < b) then skip else a := a * b + 1
```

A (possibly incomplete) datatype for parse trees in Scala would look as in Figure 1.

Question 3 (marked with 2%)

Implement an interpreter for the WHILE language you designed and parsed in Question 1 and 2. This interpreter should take as input a parse tree. However be careful because programs contain variables and variable assignments. This means you need to maintain a kind of memory, or environment, where you can look up a value of a variable and also store a new value if it is assigned. Therefore an evaluation function (interpreter) needs to look roughly as follows

```
eval_stmt(stmt, env)
```

where `stmt` corresponds to the parse tree of the program and `env` is an environment acting as a store for variable values. Consider the Fibonacci program in Figure 2. At the beginning of the program this store will be empty, but needs

```
1  abstract class Stmt
2  abstract class AExp
3  abstract class BExp
4
5  type Block = List[Stmt]
6
7  case object Skip extends Stmt
8  case class If(a: BExp, bl1: Block, bl2: Block) extends Stmt
9  case class While(b: BExp, bl: Block) extends Stmt
10 case class Assign(s: String, a: AExp) extends Stmt
11
12 case class Var(s: String) extends AExp
13 case class Num(i: Int) extends AExp
14 case class Aop(o: String, a1: AExp, a2: AExp) extends AExp
15
16 case object True extends BExp
17 case object False extends BExp
18 case class Bop(o: String, a1: AExp, a2: AExp) extends BExp
```

Figure 1: The datatype for parse trees in Scala.

to be extended in line 3 and 4 where the variables `minus1` and `minus2` are assigned values. These values need to be reassigned in lines 7 and 8. The program should be interpreted according to straightforward rules: for example an if-statement will "run" the if-branch if the boolean evaluates to `true`, otherwise the else-branch. Loops should be run as long as the boolean is `true`.

Give some time measurements for your interpreter and the loop program in Figure 3. For example how long does your interpreter take when `start` is initialised with 100, 500 and so on. How far can you scale this value if you are willing to wait, say 1 Minute.

```
write "Fib";
read n;
minus1 := 0;
minus2 := 1;
while n > 0 do {
       temp := minus2;
       minus2 := minus1 + minus2;
       minus1 := temp;
       n := n - 1
};
write "Result";
write minus2
```

Figure 2: Fibonacci program in the WHILE language.

```
start := 1000;
x := start;
y := start;
z := start;
while 0 < x do {
 while 0 < y do {
  while 0 < z do { z := z - 1 };
  z := start;
  y := y - 1
 };
 y := start;
 x := x - 1
}
```

Figure 3: The three-nested-loops program in the WHILE language. Usually used for timing measurements.