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 will 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 based on parser combinators for the WHILE language. Be careful that the parser takes as input a stream of token generated by the tokenizer from the previous courswork. Your parser should be able to handle the WHILE programs in Figures 1 and 2.

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 **??**.

Question 3 (marked with 2%)

Implement an interpreter for the WHILE language you designed and parsed above.

```
abstract class Stmt
abstract class AExp
abstract class BExp

type Block = List[Stmt]

case object Skip extends Stmt
case class If(a: BExp, bl1: Block, bl2: Block) extends Stmt
case class While(b: BExp, bl: Block) extends Stmt
case class Assign(s: String, a: AExp) extends Stmt

case class Var(s: String) extends AExp
case class Num(i: Int) extends AExp
case class Aop(o: String, a1: AExp, a2: AExp) extends AExp

case object True extends BExp
case object False extends BExp
case class Bop(o: String, a1: AExp, a2: AExp) extends BExp
```

Figure 1: The datatype for parse trees in Scala.

```
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.