Coursework 3

This coursework is worth 10% and is due on 2nd January at 16:00. You are asked to implement a parser for the WHILE language and also an interpreter. The parser needs to use parser combinators. 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 a mark of 0% will be awarded. If you use Scala in your code, a good place to start is the file `comb1.sc` and `comb2.sc` uploaded to KEATS. Feel free to use the "hack" explained during the lectures. This might make your grammar simpler. However, make sure you understand the code involved in the "hack" because if you just do "mix-and-match" you will receive strange errors. The main function that will be tested is called `eval` and `Stmts.parse_all`. The latter expects a list of tokens as input and generates an AST. The former expects an AST and "runs" the program. The marks will be distributed such that 6 marks are given for the correct grammar (and parsers); 4 marks for the correct `eval` function. You should use the lexer from CW2 for the parser - you potentially need to make additions for CW2.

⚠ Disclaimer

It should be understood that the work you submit represents your own effort. You have not copied from anyone else. An exception is the Scala code I showed during the lectures or uploaded to KEATS, which you can both use. You can also use your own code from the CW 1 and CW 2. But do not be tempted to ask Github Copilot for help or do any other shenanigans like this!

⚠ Syntax Error in Template File cw03.sc

Apologies, there is a small syntax error in the template file where a variable needs to be called `tks` instead of `tk`. The code in question is at the end of `cw03.sc` and should be like this (see lines 5, 6 and 8):

```
@main
def test(file: String) = {
  val contents = os.read(os.pwd / "examples" / file)
  println(s"Lex $file: ")
  val tks = tokenise(contents)
  println(tks.mkString(","))
  println(s"Parse $file: ")
  val ast = Stmts.parse_all(tks).head
  println(ast)
  println(s"Eval $file: ")
  println(eval(ast))
}
```

Question 1

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

- arithmetic expressions (with the operations from the previous coursework, that is +, -, *, / and %)

- boolean expressions (with the operations ==, <, >, >=, <=, !=, &&, ||, true and false)

- single statements (that is skip, assignments, ifs, while-loops, read and write)

- compound statements separated by semicolons

- blocks which are enclosed in curly parentheses

Make sure the grammar is not left-recursive.

Question 2

You should implement a parser for the WHILE language using parser combinators. Be careful that the parser takes as input a list of *tokens* generated by the tokenizer from the previous coursework. For this you might want to filter out whitespaces and comments. Your parser should be able to handle the WHILE programs in the examples directory. The output of the parser is an abstract syntax tree (AST). A (possibly incomplete) datatype for ASTs of the WHILE language is shown in Figure 1.

Question 3

Implement an interpreter for the WHILE language you designed and parsed in Question 1 and 2. This interpreter should take as input an AST. 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 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. Note also that

```scala
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 Read(s: String) extends Stmt
case class WriteVar(s: String) extends Stmt
case class WriteStr(s: String) extends Stmt
                      // for printing variables and strings

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
case class Lop(o: String, b1: BExp, b2: BExp) extends BExp
                      // logical operations: and, or
```

Figure 1: The datatype for abstract syntax trees in Scala.

some programs contain a read-statement, which means you need to read and integer from the commandline and store the value in the corresponding variable. Programs you should be able to run are given in the `examples` directory. The output of the `primes.while` should look as follows:

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
Map(end -> 100, n -> 100, f -> 4, tmp -> 1)
```

Figure 2: Sample output for the file `primes.while`.