Coursework 3

This coursework is worth 10% and is due on 2 December at 16:00. You are asked to implement a parser for the WHILE language and also an interpreter. 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. You should use the lexer from the previous coursework for the parser. Please package everything(!) in a zip-file that creates a directory with the name `YournameYourFamilyname` on my end.

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!

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 stream, or 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 Figures 2 – 5. In addition give the parse tree according to your grammar for the statement:

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

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`. Programs you should be able to run are shown in Figures 2 – 5.

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?

```
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.

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

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.

```
// prints out prime numbers from 2 to 100

end := 100;
n := 2;
while (n < end) do {
  f := 2;
  tmp := 0;
  while ((f < n / 2 + 1) && (tmp == 0)) do {
    if ((n / f) * f == n) then  { tmp := 1 } else { skip };
    f := f + 1
  };
  if (tmp == 0) then { write(n); write("\n") } else { skip };
  n  := n + 1
}
```

Figure 4: Prime number program.

```
// Collatz series
//
// needs writing of strings and numbers; comments

bnd := 1;
while bnd < 101 do {
  write bnd;
  write ": ";
  n := bnd;
  cnt := 0;

  while n > 1 do {
    write n;
    write ",";

    if n % 2 == 0
    then n := n / 2
    else n := 3 * n+1;

    cnt := cnt + 1
  };

  write " => ";
  write cnt;
  write "\n";
  bnd := bnd + 1
}
```

Figure 5: Collatz series program.