## Exercise 2: Interpreter for Boolean Expressions

In Exercise 7 of Chapter 4, you specified a scripting language for Boolean expressions and described its processing by an attributed grammar. Use Coco/R now to implement an interpreter that processes and executes programs of this scripting language. No code should be generated, but the instructions should be executed immediately during parsing.

As a parser specification, you can use the attributed grammar developed in Exercise 7 of Chapter 4 in Coco/R notation. In the scanner specification, you only need to declare the terminal symbol ident. The symbol table holding the names and values of the variables should be managed as a hash table that is declared in the global declarations of the compiler specification. Write a main program that creates a scanner and a parser, and then starts the parser (and thus the interpreter). The name of the file with the script program should be passed to the main program as a command line argument.

Use Coco/R to generate a scanner and a parser and compile them together with the main program to get the executable interpreter.

### Solution

```
import java.util.HashMap;

COMPILER BoolScript

//---------------------- global declarations -----------------------------
  HashMap<String, Boolean> values = new HashMap<String, Boolean>();  // symbol table

//--------------------- scanner specification ---------------------------
CHARACTERS
  letter = 'A'..'Z' + 'a'..'z'.

TOKENS
  ident = letter {letter}.

IGNORE '\r' + '\n' + '\t'

//---------------------- parser specification ---------------------------
PRODUCTIONS

BoolScript = { Statement }.

Statement                     (. boolean val; .)
=  ident                      (. String name = t.val; .)
   "="
   Expr <out val>             (. values.put(name, val); .)
  ";"
|  "print"
   Expr <out val>             (. System.out.println(val); .)
   ";".

Expr <out boolean val>        (. boolean val2; .)
=  Term <out val>
   {  "||" Term <out val2>    (. val = val || val2; .)
   }.

Term <out boolean val>        (. boolean val2; .)
=  Factor<out val>
   {  "&&" Factor <out val2>  (. val = val && val2; .)
   }.
```

```
Factor <out boolean val>
=                               (. val = false; .)
(  "true"                       (. val = true; .)
|  "false"                      (. val = false; .)
|  ident                        (. Boolean c = values.get(t.val);
                                   if (c != null) val = c; else { SemErr("undeclared name"); val = false; } .)
|  "(" Expr <out val> ")"
|  "!" Factor <out val>         (. val = ! val; .)
).

END BoolScript.
```

The names used in the scripting language as well as their Boolean values are stored in a hash table values, which serves as a *symbol table* and is declared in the global semantic declarations.

The scanner specification declares the terminal symbol ident as a sequence of letters. It also specifies that, in addition to blanks, line endings and tabs should be ignored.

The parser specification is an attributed grammar. Expr, Term and Factor perform the calculation of a Boolean expression. The value of Boolean variables is taken from the symbol table. An assignment statement enters the Boolean value of the Expr on the right-hand side under the name specified on the left-hand side into the symbol table. The print statement outputs the value of a Boolean expression to the console.

We still need a main program that gets the name of the script file as a command line argument, generates a scanner and a parser and starts the syntax analysis (and thus the interpretation):

```
class BoolScript {
   public static void main (String[] arg) {
      Scanner scanner = new Scanner(arg[0]);
      Parser parser = new Parser(scanner);
      parser.Parse();
      System.out.println(parser.errors.count + " errors detected");
   }
}
```

We process the compiler specification with Coco/R:

```
java -jar Coco.jar BoolScript.atg
```

thus generating a scanner and a parser. Then we compile all Java files

```
javac Scanner.java Parser.java BoolScript.java
```

and invoke the interpreter:

```
java BoolScript script.txt
```

The file script.txt contains the script program, which could look as follows:

```
big = true;
small = ! big;
ready = false;
print (big || small) && ready || big && ! ready;
```

The output in this case is

```
true
```