Prof. Jingke Li (FAB120-06, lij@pdx.edu), Class: TR 10:00-11:15 @ EB 103; Office Hour: TR 11:20-12:20.

# Exercise 2: Interpreter and Compiler

## (Due Tuesday 10/11/22)

This exercise is to get familiar with Scala's style of programming through the implementation of an interpreter and a compiler for a toy language.

## 1. BoolLang

BoolLang is a toy Boolean-expression language, defined by the following grammar:

```
Expr -> t                 // true
     |  f                 // false
     |  (not Expr)        // negation
     |  (and Expr Expr)   // and
     |  (or Expr Expr)    // or
     |  (xor Expr Expr)   // exclusive or
```

Here is a sample expression written in this language:

```
(xor (or t (not t)) (and f f))
```

BoolLang follows the standard evaluation rule, *not* the short-circuit rule: for any operation, all operands are evaluated first.

Note that BoolLang's syntax is in the form of S-expression (see this week's lecture notes). This is intentional, aiming at simply the lexer and parser, and to have a direct correspondence between a program's external form and its AST form. Our subsequent toy languages will all be in this form.

The file `BoolLang.scala` contains the definitions of BoolLang's internal AST representation,

```
sealed abstract class Expr
case object True extends Expr
case object False extends Expr
case class Not(e:Expr) extends Expr
case class And(l:Expr,r:Expr) extends Expr
case class Or(l:Expr,r:Expr) extends Expr
case class Xor(l:Expr,r:Expr) extends Expr
```

and a parser:

```
def expr: Parser[Expr] = atm | lst
def atm: Parser[Expr] = "t" ^^ {_ => True} | "f" ^^ {_ => False}
def lst: Parser[Expr] =
      "(not" ~> expr <~ ")"      ^^ {e => Not(e)}          |
      "(and" ~> expr~expr <~ ")" ^^ {case l~r => And(l,r)} |
      "(or"  ~> expr~expr <~ ")" ^^ {case l~r => Or(l,r)}  |
      "(xor" ~> expr~expr <~ ")" ^^ {case l~r => Xor(l,r)}
```

Observe the correspondence between BoolLang's grammar and the parser code, and the action attached to each grammar rule.

### Exercises

1. Run the program `BLInterp.scala` interactively with the Scala interpreter:

```
scala> :load BoolLang.scala   // load the source program
scala> import BoolLang._      // import the class to avoid typing long names
scala> And(True,False)        // construct an AST node
res1: BoolLang.And = And(True,False)
scala> BLParse("(or t f)")    // parse a program
res2: BoolLang.Expr = Or(True,False)
```

2. Compile the program, then run it with two different input channels,
   - file input: `linux> scala BoolLang t01.bl`
   - std input: `linux> scala BoolLang -`
     `(or t f)` — end a program with carriage return

## 2. StackM0

A simple stack machine is defined in `StackM0.scala`. It has five instructions:

- `T` — push Boolean value `true` onto the (operand) stack
- `F` — push Boolean value `false` onto the (operand) stack
- `NOT` — pop off an element, negate its value, and push the result back onto the stack
- `AND` — pop off two elements, perform logical-and on their values, and push the result back onto the stack
- `OR` — pop off two elements, perform logical-or on their values, and push the result back onto the stack

Here is a sample program for this machine:

```
T NOT F AND    // !t && f
```

Read the `StackM0.scala` program to understand the instructions' execution.

**Exercises**

1. Write a `StackM0` program for each of the following Boolean expressions. `x` and `y` represent unknown subexpressions; Use `<xc>` and `<yc>` to represent their corresponding code sequence in the SM0 program. $\oplus$ represents exclusive-or.
   (a) `(x && y) || !x`
   (b) `x == y`
   (c) `x` $\oplus$ `y`

   Note that StackM0 does not have matching instructions for `==` and $\oplus$. It will require a bit of creativity to implement these two operations in the above expressions.
2. Test your results by running `StackM0.scala`; randomly assign a value to `x` and `y`.

## 3. An interpreter and a compiler

The files `BLInterp.scala` and `BLComp.scala` contains a half-completed BoolLang interpreter and compiler, respectively. The missing piece in both files is the implementation of the `xor` expression.

**Exercises** Complete the interpreter and compiler implementation.

## 4. Scala's Testing Utility

Scala comes with a testing utility. User define tests in a file. Take a look inside such a file, `TestBL.scala`. A test can take several forms, among them, `assert`, `intercept`. The former is for verifying a result, and the latter is for catching an expected exception:

```
test("parse individual ops") {
  assert(And(True, False) == parse("(and t f)"))
  assert(Or(True, False) == parse("(or t f)"))
}
test("parse exception for expression with 3 arguments") {
  intercept[ParseException] { (parse("(and t t f)")) }
}
```

A test program is compiled as usual, but you need to reference an extra program to run it:

```
linux> scala org.scalatest.run TestBL
```

**Exercises**

1. Compile and run the two test programs, `TestBL.scala` and `TestBLI.scala`.
2. Write a test program, `TestBLC.scala`, for testing the BoolLang compiler. (*Hint:* It should look very similar to `TestBLI.scala`.)

## Submission

Collect your answers to Part 2 questions in a text file, `ex2-notes.txt`. You may add comments about this exercise in there as well. Submit this file, along with three programs, `BLInterp.scala`, `BLComp.scala`, and `TestBLC.scala`, through the "Exercise 2" submission folder on the Canvas class website (under the "Assignments" tab).