Prof. Jingke Li (FAB120-06, lij@pdx.edu), Class: TR 12:00-13:15 @ KMC 185; Office Hour: TR 11:00-11:55.

Assignment 2: S-Expressions, Interpreter, and Compiler

(Due Thursday 10/11/18)

This week's assignment introduces the s-expression format that we will use to describe program ASTs throughout the course, and asks you to investigate an interpreter and a compiler for a very simple expression language.

Unzip the file `assign2.zip`, and you'll see this handout file, `assign2.pdf`, and several program files.

1. S-Expressions

We will use s-expressions as an intermediate representation of programs when parsing, based on a library `SExprLibrary`. There are two main functions in this library:

- The `SExprReader.read` function parses an s-expression in textual notation and returns an `SExpr` object representing the s-expression. This `read` function is defined using Scala parser combinators. You don't need to know exactly how it works, but you may find it interesting to try to make some sense out of it.
- The `SExprPrinter.print` function is the inverse of the `read` function. If `SExprReader.read(s)` succeeds and returns a result `r`, then `SExprPrinter.print(r)` should print out a string that is equivalent to `s`. (The two may differ in terms of white spaces.)

Exercises

1. Invoke the Scala interpreter shell, and load the file `SExprLibrary.scala`. Try to call the `read` and `print` functions in the REPL shell. Here is a sample run:

```
scala> :load SExprLibrary.scala
...
scala> import SExprLibrary._
...
scala> SExprReader.read("(1 2 3)")
res0: SExprLibrary.SExpr = (1 2 3)
scala> SExprReader.read("(a b c)")
res1: SExprLibrary.SExpr = (a b c)
scala> SExprPrinter.print(res0)
res2: String = (1 2 3)
scala> SExprPrinter.print(res1)
res3: String = (a b c)
```

   Also try the `SExpr` constructors:

```
scala> SNum(1)
res4: SExprLibrary.SNum = 1
scala> SSym("x")
res5: SExprLibrary.SSym = x
scala> SString("Hello!")
res6: SExprLibrary.SString = "Hello!"
```

   Now, try to use the `SList` constructor to create an s-expression list object.

2. Scala comes with a testing utility. To use it, first copy the file `scalatest-app_2.12-3.0.5.jar` from D2L (under the "Content/General" tab) into your home directory, and check that the environment variable `HOME` is set to pointing to that directory.

   User define tests in a file. Take a look inside such a file, `TestSExpr.scala`. A test can take several forms, among them, `assert` and `assertResult`. The two forms have similar expressive ability:

   ```
   test("reading a number 123") {
     assert(SNum(123) == SExprReader.read("123"))
   }
   test("reading a number 456") {
     assertResult(SNum(456))(SExprReader.read("456"))
   }
   ```

   Use the `Makefile` to compile the `TestSExpr` program, and the `run` script to run it:

   ```
   linux> make testsexpr
   linux> ./run TestSExpr
   ```

   Note that the `make` target uses all lower-case letters. You may want to peek inside the `Makefile` and `run` files to see what's in there.

   Observe the printout. Now add more tests into the test file to cover the other `SExpr` constructors, for both reading and printing.

2. EL0

EL0 is a simple expression language, defined by the following AST grammar:

```
Expr -> Int
     |  (+ Expr Expr)
     |  (- Expr Expr)
     |  (* Expr Expr)
     |  (/ Expr Expr)
     |  (% Expr Expr)
```

The file `EL0.scala` provides an implementation, along with two interface functions, `parse` and `print`. Note that the `parse` function takes an optional `debug` argument: if this is greater than 0, some potentially useful debug information is printed out. (If not specified, the argument defaults to 0.)

Exercises

1. Read the content of `EL0.scala` to get familiar with the AST representation. Use the scala interpreter to try constructing a few EL0 ASTs.
2. Look inside the file `TestEL0.scala`. Notice a new form of test, an exception interception:

   ```
   test("parse exception for s-expression with 3 arguments") {
     intercept[ParseException] { (parse("(+ 1 2 3)")) }
   }
   ```

   Compile and run the tests, and observe the printouts.

3. Machine0

A simple stack machine is defined in `Machine0.scala`. It has the following instructions:

`Const n` — push integer `n` onto the (operand) stack
`Pop` — pop off an element from the stack

`Swap` — swap top two elements of the stack

`Plus` — pop off two elements, add their values, and push the sum back onto the stack

`Times` — pop off two elements, multiply their values, and push the product back onto the stack

`Divrem` — pop off two elements, divide their values, and push both the quotient and the remainder back onto the stack

Exercises

1. Read the content of `Machine0.scala` to get familiar with the program representation and the instructions' execution.
2. Write a `Machine0` program for each of the following expressions:
   (a) `1 + (3 - 2)`
   (b) `(2 * -3) - (5 / 3)`
   (c) `(-2 / 3) * 3 + (-2 % 3)`

   Note that Machine0 does not have a subtraction instruction. It will require a bit of creativity to implement subtraction operations in the first two expressions. Also note that `Divrem` pushes two result items onto the stack.
3. What value should each of the above expressions evaluate to? Following the style of `TestSExpr.scala` and `TestEL0.scala`, write a `TestMachine0.scala` program with the above tests as its content. Compile and run the program; make sure all tests pass.

4. EL0 Interpreter

The file `Interp0.scala` contains a skeleton of an EL0 interpreter. Note that the `process` function has a `debug` flag, just like the the `parse` function in `EL0.scala`.

Exercises

1. Complete the interpreter implementation for the language EL0. You will need to add new clauses to the `interp` function.

   Be careful to implement `%` correctly on non-positive arguments: the equation

   ```
   (a / b) * b + (a % b) = a
   ```

   should hold whenever `b` is non-zero, and a division by zero exception should be thrown otherwise.
2. The file `TestInterp0.scala` contains a small set of tests. Add new tests to cover a few more complicated EL0 expressions. Make sure your interpreter passes all tests.

5. EL0 Compiler

The file `Comp0.scala` contains a skeleton of an EL0 compiler. It first compiles an EL0 program into Machine0's instructions, and then executes those instructions. Note that the `debug` flag in the `process` function now has two useful non-zero values: setting it to 1 will print out the original parsed expression and the compiled code; setting it to 2 will also print a trace of the machine execution steps.

Exercises

1. Complete the compiler implementation. Once again, you will need to add new clauses to the `compile` function.

2. Adapt the program `TestInterp0.scala` to a test program for the compiler, `TestComp0.scala`. Since both the interpreter and the compiler has a driver function with the same name, `process`, the adaptation should be easy.

   Test your compiler with the test program, and make sure it passes all tests.

6. Submission and Grading

For this assignment, you need to submit five program files, `TestMachine0.scala`, `Interp0.scala`, `Comp0.scala`, and `TestComp0.scala`. Include your full name in each of the program files. Zip them into a single file, and submit it through the "Assignment 2" submission folder on the D2L class website (under the "Activities/Assignments" tab). Keep your original files untouched in case there is a need to show their timestamp.

Grading will be based on program correctness, mostly through testing.