Assignment #2

CSE271: Principles of Programming Languages
Section 02 Hyungon Moon

Out: October 5, 2021 (Tue)
Due: Oct 14, 2021 (Thu), 23:59 (KST)

What to submit

Submit your `Hw2.scala` file through the Blackboard.

Info: The directory sturucture of the handout is as follows.

```
sbt/                    - contains the sbt program that you need to test your program.
src/                    - where all your scala source files live.
    main/scala/
        Hw2.scala       - >>>> what you need to edit and submit. <<<<<
        Parser.scala    - parser driver for the languages you will interpret.
    main/antlr4/        - where inputs to the parser generater live. You can ignore this.
    test/scala/
        Hw2Test.scala   - sample tests that we wrote for you.
                          You can edit this to further test your program.
```

Rules

- You must not use the `var`, `for`, or `while` keyword.
- You must not include any additional packages or libraries besides the ones that you already have.

Scala environment

Please refer to the instruction for the first assignment to set up the Scala environment.

Late submission policy

You will get 70% and 50% of the grade if you turn it in within 24 and 48 hours, respectively, after the regular deadline. Please note that an assignment that is re-submitted after the regular deadline will be counted as late even if you have an earlier submission. After the late submission deadline has passed, you won't be able to submit your assignment and will get 0 points for it.

Problems

Problem 1 (30 points)

Implement an interpreter that evaluates an expression into an integer value.

Syntax

$$
\begin{aligned}
E \rightarrow\ & n \in \mathbb{Z} \\
\mid\ & x \in \{'x'\} \\
\mid\ & E + E \\
\mid\ & E - E \\
\mid\ & E * E \\
\mid\ & \texttt{sigma}\ E\ E\ E \\
\mid\ & \texttt{pi}\ E\ E\ E \\
\mid\ & \texttt{pow}\ E\ E
\end{aligned}
\tag{1}
$$

In scala,

```
sealed trait IntExpr
case class IntConst(n: Int) extends IntExpr
case object IntVar extends IntExpr
case class IntAdd(l: IntExpr, r: IntExpr) extends IntExpr
case class IntSub(l: IntExpr, r: IntExpr) extends IntExpr
case class IntMul(l: IntExpr, r: IntExpr) extends IntExpr
case class IntSigma(f: IntExpr, t: IntExpr, b: IntExpr) extends IntExpr
case class IntPi(f: IntExpr, t: IntExpr, b: IntExpr) extends IntExpr
case class IntPow(b: IntExpr, e: IntExpr) extends IntExpr
```

```
sigma 1 10 x     pi 1 10 x
```
(2)

stands for

$$\sum_{x=1}^{10} x \qquad \prod_{x=1}^{10} x \tag{3}$$

Note that an expression can have only one variable, `x`, to represent the index variable in `sigma` and `pi`.

Follow the intuitive semantics for the addition, substraction, multiplication. For `sigma`, `pi` and `pow`, you can also follow the intuitive semantics as follows. Note that `pow` is not defined if the exponent is evaluated to a negative integer.

Semantics

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \rho \vdash E_2 \Rightarrow v_2 \\ [x \mapsto v_1]\rho \vdash E_3 \Rightarrow v_3 \quad \rho \vdash (\texttt{sigma}\ (E_1 + 1)\ E_2\ E_3) + v_3 \Rightarrow v_4}{\rho \vdash \texttt{sigma}\ E_1\ E_2\ E_3 \Rightarrow v_4}\ v_1 \leq v_2$$

$$\frac{\rho \vdash E_1 \to v_1 \quad \rho \vdash E_2 \to v_2}{\rho \vdash \texttt{sigma}\ E_1\ E_2\ E_3 \Rightarrow 0}\ v_1 > v_2$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \rho \vdash E_2 \Rightarrow v_2 \\ [x \mapsto v_1]\rho \vdash E_3 \Rightarrow v_3 \quad \rho \vdash (\texttt{pi}\ (E_1 + 1)\ E_2\ E_3) * v_3 \Rightarrow v_4}{\rho \vdash \texttt{pi}\ E_1\ E_2\ E_3 \Rightarrow v_4}\ v_1 \leq v_2 \tag{4}$$

$$\frac{\rho \vdash E_1 \to v_1 \quad \rho \vdash E_2 \to v_2}{\rho \vdash \texttt{pi}\ E_1\ E_2\ E_3 \Rightarrow 1}\ v_1 > v_2$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \quad \rho \vdash E_2 \Rightarrow v_2 \quad \rho \vdash (\texttt{pow}\ E_1\ (E_2 - 1)) * E_1 \Rightarrow v_3}{\rho \vdash \texttt{pow}\ E_1\ E_2 \Rightarrow v_3}\ v_2 > 0$$

$$\frac{\rho \vdash E_2 \Rightarrow v_2}{\rho \vdash \texttt{pow}\ E_1\ E_2 \Rightarrow 1}\ v_2 = 0$$

In the skeletone, you can find the `IntInterpreter` object whose `apply` method looks like the following. `apply` method also calls the parser and the interpreter for you.:

```
def apply(s: String): Int
```

Your job is to fill out the body of `evalInt` method below.

```
def evalInt(expr: IntExpr, env: Option[Int]): Int
```

Note that not all valid programs under the presented syntax have semantics given an empty environment. Please throw an exception if an interpreter is given such a program.

Problem 2 (40 points)

Implement an interpreter for our LETREC language. Follow the syntax and semantics given in the lecture slides.

The grammar in scala looks like:

```
sealed trait Program
sealed trait Expr extends Program
case class Const(n: Int) extends Expr
case class Var(s: String) extends Expr
case class Add(l: Expr, r: Expr) extends Expr
case class Sub(l: Expr, r: Expr) extends Expr
case class Equal(l: Expr, r: Expr) extends Expr
case class Iszero(c: Expr) extends Expr
case class Ite(c: Expr, t: Expr, f: Expr) extends Expr
case class Let(name: Var, value: Expr, body: Expr) extends Expr
case class Paren(expr: Expr) extends Expr
case class Proc(v: Var, expr: Expr) extends Expr
case class PCall(ftn: Expr, arg: Expr) extends Expr
case class LetRec(fname: Var, aname: Var, fbody: Expr, ibody: Expr)
extends Expr
```

The set of values can be defined like:

```
sealed trait Val
case class IntVal(n: Int) extends Val
case class BoolVal(b: Boolean) extends Val
case class ProcVal(v: Var, expr: Expr, env: Env) extends Val
case class RecProcVal(fv: Var, av: Var, body: Expr, expr: Expr, env: Env)
extends Val
```

Now you can fill out the following method in `LetRecInterpreter` object.

```
def eval(env: Env, expr: Expr): Val
```

Note that not all valid programs under the presented syntax will have semantics given an empty environment. Please throw an exception if an interpreter is given such a program.

Problem 3 (30 points)

Implement a function that builds a program into a string. Use the same `case class` definitions and implement `apply` method of `LetRecToString` object.

Note that not all valid programs under the presented syntax will have semantics given an empty environment. Please throw an exception if an interpreter is given such a program.

```
def apply(expr: Expr): String
```