Prof. Jingke Li (lij@pdx.edu), Classes: TR 1100-1240 @CIN 90; TA: Katherine Philip (`kphilip@pdx.edu`).
Office Hours: JL: TR 1000-1100 (in-person @FAB 120-06), KP: W 1300-1500 (in-person @FAB Fishbowl)

Assignment 2 Relational Expressions and Simple Statements

(Due Thursday 11/6/25)

This assignment continues the work of Exercises 3 and 4. You will implement several interpreters, for relational expressions and for simple statements. This assignment carries a total of 20 points.

1. [*8 points*] Interpreters for Relational Expressions

Consider the following relational expression language, `RelEx`:

```
relex -> relex rop atom
      |  atom
atom  -> "(" relex ")"
      |  NUM
rop   -> "<"|"<="|">"|">="|"=="|"!="
```

All operators of this language belong to the same precedence level and all are left associative. As shown by the grammar, `RelEx` allows relational expressions to be chained up, like Python and C. Note that the given grammar already encodes the left-associativity and is unambiguous.

Your task is to implement two interpreters in Lark, one follows C semantics and one Python semantics. Both interpreters take the form of an `Eval()` class, with a collection of interpretation methods, one for each type of AST nodes. A starting version of the program is given in `relex.py`. Before implementing the two `Eval()` classes, you need to encode `RelEx`'s grammar in Lark, and attach AST nodes to the productions. Note that the grammar is already in the right form, so there is no need to change it.

The details for the two interpreters are given below.

1. [*3*] **EvalC() — An Interpreter with C Semantics.** With C semantics, a chained-up relational expressions is interpreted similarly to a chained-up arithmetic or logical one. E.g. `1<2<3` is interpreted as equivalent to `(1<2)<3`. Note that there is no Boolean type in C; the result of a relational comparison is either 1 (representing true) or 0 (representing false), which makes it possible to have the above treatment for chained-up relational exprssions. This interpreter should return an integer value of either 1 or 0.

2. [*5*] **EvalP() — An Interpreter with Python Semantics.** With Python semantics, a chained-up relational expression is interpreted quite differently from an arithmetic or logical expression. Each middle operand is replicated and used in two separate operations. E.g. `1<2<3` is interpreted as equivalent to `(1<2) and (2<3)`.

   Note that unlike precedence and associativity, this replication semantics cannot be handled by the grammar re-writing scheme. Rather, it needs to be manually handled in the interpreter. Specifically, when evaluating a binary operation `x rop y`, the interpreter needs to check if `x` is a subexression of the same binary-op form, say `a op b`. If so, it needs to extract the second operand `b` out from `x`. With that, it can then correctly implement `x rop y` by evaluating the expression `x and (b rop y)`. Using `1<2<3` as an example, `x` in this case is `1<2` and `y` is `3`. We need to extract `2` out from `x`, and then return the value of `(1<2) and (2<3)`.

   Hint: In Lark, an AST node's children can be accessed through the `.children` attribute (which returns a list). For instance, if `x` is an AST node representing a binary-op `a op b`, then `x.children[0]`, `x.children[1]`, and `x.children[2]`, will allow you to access the three components, `a`, `op`, and `b`.

   Note that, unlike `EvalC()`, this interpreter should return a Boolean result.

2. [*8 points*] An Interpreter for a Simple Statement Language

Consider the following simple statement language, `Stmt`:

```
stmt -> ID "=" expr
      | "if" "(" expr ")" stmt ["else" stmt]
      | "while" "(" expr ")" stmt
      | "print" "(" expr ")"
      | "{" stmt (";" stmt)* "}"

expr -> expr "+" term
      | expr "-" term
      | term
term -> term "*" atom
      | term "/" atom
      | atom
atom: "(" expr ")"
      | ID
      | NUM
```

This language has four statements, assignment, if, while, and print. It also has a construct (a pair of curly braces) representing a statement block, which may appear anywhere a statement is expected. All statements have embedded expressions, which are constructed from integer constants, variables, and (only) arithmetic operations. Every variable is implicitly initialized to 0 at the beginning of program execution. There is no nested scopes, the same variable name always refers to the same variable. This language follows the common convention in handling the dangling-else case, i.e. a dangling-else is matched with the closest if statement. A program in this language is either a single statement or a list of statements in the form of a statement block.

The semantics of the statements and the block construct is given below.

- Executing `x = e` evaluates `e`, assigns the resulting value into variable `x`.
- Executing `if (e) s1 [else s2]` evaluates expression `e`; if the result is non-zero, then statement `s1` is executed; otherwise if the else-clause exists, statement `s2` is executed.
- Executing `while (e) s` evaluates expression `e`; if the result is non-zero, statement `s` is executed and the entire `while` statement is executed again; otherwise the execution of the `while` is complete.
- Executing `print (e)` evaluates expression `e` and prints out the resulting value.
- Executing `{s1; s2; ... sn}` executes statements `s1,s2,...,sn` in that order.

Your task is to implement an interpreter using Lark for this language. Your interpreter should follow the given semantics of the statements. Note that the above grammar is already in the right form for Lark parser implementation. But you need to pay special attention to the dangling-else case (Cf. Exercise 4). Also note that even though there is only one scope in a program of this language, you still need to implement a simple environment to keep track of variables' values.

A starting version of the program is given in `stmt.py`.

3. [*2 points*] Two Programs in this Statement Language

Write the following two programs in `Stmt`, and then use your interpreter to test them:

1. `sum.st` – It computes the sum of 1 to 100, and prints out the result.
2. `skip.st` – It starts with two assignments, `n = NUM1` and `k = NUM2`, where `NUM1` $\geq$ `NUM2`, and prints out the values from `n` down to 1, but skips `k`. For instance, if `n = 4` and `k = 3`, the program will print out three values, 4, 2, and 1. You should implement this program with `while` and `if` statements, rather than hard-wiring a list of `print` statements.

4. [*2 points*] Summary Report

Write a short summary covering your experience with this assignment, including the following:

- Status of your programs. Do they successfully run on all tests you conducted? If not, describe the remaining issues as clearly as possible.
- Experience and lessons. What issues did you encounter? How did you resolve them?

Save your summary in a text or pdf file; call it `report.[txt|pdf]`.

Submission

Zip the four program files, `relex.py`, `stmt.py`, `sum.st`, `skip.st`, along with your summary report, into a single zip file `assign2sol.zip`, and upload it through the upload it through the "File Upload" tab in the "Assignment 2" folder. (You need to press the "Start Assignment" button to see the submission options.) *Important:* Keep a copy of your submission file in your folder, and do not touch it. In case there is a glitch in Canvas submission system, the time-stamp on this file will serve as a proof of your original submission time.