Prof. Jingke Li (lij@pdx.edu), Classes: TR 1400-1550 @OND 218, Office Hours: TR 1300-1400 (in-person @FAB 120-06). TAs: Nicholas Coltharp (`coltharp`), M 2-4pm @Fishbowl; Laura Israel (`laisrael`), W 10-11 @Fishbowl, F 1-2pm (Zoom: 761-457-3865)

Assignment 1 Parsers and Interpreters

(Due Thursday 10/24/24)

This assignment continues the work of Exercises 1 and 2. You should complete those exercises before starting on this assignment. This assignment carries a total of 20 points.

Learning Objectives

Upon successful completion, students will be able to:

- Use Lark to generate simple parsers,
- Perform simple tasks over an AST, and
- Use an environment for managing binding information.

1. [*10 points*] An Interpreter for a Boolean-Expression Language

The following grammar is for a Python-style Boolean expression language, which we call `BoolEx`:

```
orex  -> orex "or" andex
      |  andex
andex -> andex "and" atom
      |  atom
atom  -> "not" atom
      |  "(" orex ")"
      |  "True"
      |  "False"
```

Your tasks are to implement a parser using the Lark parer-generator, along with several actions on the AST. A starting version of the program is available in `boolex.py`. You are to add code to complete the program.

1. [*2*] Convert the above grammar to a Lark specification, and implement a parser for the language. Name corresponding AST nodes for the three operations and the two literal values. As an example, here is a possible run of this program (showing suggested names for the AST nodes):

   ```
   linux> python boolex.py
   Enter a bool expr: (True or not False) and True
   andop
     orop
       truev
       notop
         falsev
     truev
   ```

   Since we have not studied grammars yet, you should not modify the given grammar; just port it as is over to Lark.

2. [*3*] Add an interpreter component to the program in the form of an `Eval()` class. It traverses the AST, and evaluates the corresponding expression to a Boolean value. We assume this `BoolEx` language follows the standard "short-circuit" evaluation semantics (which is also the semantics of Python).

3. [*2*] In parallel to the `Eval()` action, define a second action over the AST to convert it to a linear list form. Call this action, `toList()`. In this list form, each operation node is turned into a prefix list, with the operator appearing as the first element (as a string), and each operand as a separate element. The operands

can themselves be nested lists. The two value nodes appear as their literal form, `'True'` and `'False'`. Here is a sample output for the expression in the above example:

```
tree.toList() = ['and', ['or', 'True', ['not', 'False']], 'True']
```

4. [*3*] The list form of the AST shown above contains lots of quotes, hence are not easy to read. In this part, you are to write a function `strForm(lst)` to convert such a list into a simpler string form, in which all quotes and commas are dropped, and square brackets get turned into parentheses. Here is a sample output for the above list:

   ```
   strForm() = (and (or True (not False)) True)
   ```

   Warning: This is not a trivial function to implement. You need to find a way to deal with the nested lists.

2. [*8 points*] An Interpreter for a Let-Expression Language

The second language, `LetEx`, has been discussed in class. Here is its grammar:

```
expr0 -> "let" ID "=" expr0 "in" expr0
      |  expr
expr  -> expr "+" term
      |  expr "-" term
      |  term
term  -> term "*" atom
      |  term "/" atom
      |  atom
atom  -> "(" expr0 ")"
      | ID
      | NUM
```

1. [*2*] Do the same as you did for `BoolEx`: Convert the grammar to a Lark specification, name corresponding AST nodes, and implement a parser for the language. here is a possible run of this program:

   ```
   linux> python boolex.py
   Enter a let expr: let x=1 in x+1
   let
     x
     num   1
     add
       var x
       num 1
   ```

   A starting version of the parser program is available in `letex.py`.

2. [*2*] Port the implementation code for an environment from Thursday's lecture notes (pages 20-21) into this parser program. You may want to do some testing to verify it is working, before moving on.

3. [*4*] Again, this part is similar to what you did for `BoolEx`: Add an interpreter component to the program in the form of an `Eval()` class. It traverses the AST, and evaluates a let expression into a value. However, there is a new situation, dealing with variables. Specifically, you need to write interpretation code for both the `var` (variable) and `let` nodes. The base for these code are their semantics, which are shown below:

   - In interpreting a variable, look up its value from the current environment; return that value
   - In interpreting a `let(x, exp, body)` node, follow the steps:
     - evaluate the `let x` binding's expression (to a `val`)
     - store the binding `(x, val)` into the current environment
     - evaluate the `let` construct's body expression (to a `result`)
     - remove the binding `(x, val)` from the environment
     - return `result`

3. [*2 points*] Testing and Summary Report

With the read-eval-print loop (REPL) in the `main` function, you can run each program with multiple interactive inputs (and terminates it with a Control-D). You may also run batch tests, i.e. placing test expressions in a file and piping it into the parser program.

Two sample test files are provided in `testboolex` and `testletex`. Here is sample usage:

```
linux> python boolex.py < testboolex
Enter a bool expr: (True or not False) and True
andop
  orop
    truev
    notop
      falsev
  truev
tree.Eval() = True
tree.toList() = ['and', ['or', 'True', ['not', 'False']], 'True']
strForm() = (and (or True (not False)) True)
...
```

Feel free to add new tests as you see fit to these files, and use them to test your programs.

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 three files, `boolex.py`, `letex.py`, and `report.[txt|pdf]`, to a single file, `assign1sol.zip`, and uploaded it through the "File Upload" tab in the "Assignment 1" 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 any glitch in the Canvas submission system, the time-stamp on this file will serve as a proof of your original submission time.