CSE211: Compiler Design

Homework 1: Parsing Overview
Assigned Oct. 9, 2023 (by midnight)
Due Oct. 25, 2023 (by midnight)

- This assignment has two parts: part 1 implements a simple interpreter and part 2 deals constructs an AST and parses regular expressions using the *parsing with derivitives* approach. Part 1 is worth 40% of the total for this homework while part 2 is worth 60% of the total for this homework.
- You are free to develop locally or using the class docker. This assignment requires only Python3 and a command line. However, the interface must match exactly what is specified otherwise you may lose points. If you develop in the docker, please review the docker setup page on the class webpage and make sure you understand what data persists and what does not in a docker environment; I do not want you to lose any work. Please ask if you have questions about docker! Any questions about docker are allowed as public posts in the class piazza. Conversely, please help out your classmates if you know an answer.
- There are likely typos in the homework (writeup and code)! We are redoing some of these assignments for the pair programming we are doing this quarter. Along these lines, if you have ideas for the homework, please let me know!
- This homework is designed to be done as a pair. I encourage you to utilize pair programming techniques, that is: both people must be present for any work on the homework. If this is on zoom, then a screen should be shared at all times. Only one computer should be working on the homework at a time. Please take this chance to help each other understand the homework; explain the concepts you are coding and make sure you both understand what is going on completely before moving on.
- If you have any issues with your partner, please let us know ASAP.
- Do not work with anyone apart from your partner on this homework (except for what is explicitly mentioned above).

1 Interpreting a simple language

This part of the homework will have you implementing an interpreter for a simple programming language. The required code for this part of the assignment can be found in `homework1_code/part1`. Unless otherwise approved, please use PLY to implement the interpreter. Please limit your work to two files, `skeleton.py` and `my_tests.py`. I have provided the outline of an implementation for you. Your job is to finish the implementation.

You are free to change any part of the file except for the function: `parse_string`. This function must take as input a string. It must parse (and interpret) the string according the language specification provided below. It must throw the exceptions as specified below, or return a list of values, as specified below. Feel free to consult the PLY documentation or class slides as much as you want. If you copy something directly, please mention it in a comment.

1.1 Language Specification

This language is an extension of the calculator language we showed in class. The language is augmented with variables (IDs) that can store a number value. The language additionally has a print statement, and braces for variable scoping.

1.1.1 Statements

This simple language consists of a list of statements. There are three types of statements:

- An assignment statement, which consists of an ID, followed by the '=' symbol, followed by a mathematical expression, and concluding with a semicolon. For example: `x = 1 + 1;`
- A print statement, which consists of the keyword `print`, followed by an opening parenthesis '(', followed by an ID, followed by a closing parenthesis ')' and concluding with a semicolon.
- a scope statement, which consists of an opening brace '{', a statement list, and a closing brace '}'.

The assignment statement creates a variable with the identifier ID. The result of the expression is stored in the variable. The ID consists of alphabetical characters (i.e. only letters). They can be upper or lower case.

The print statement records the value in the variable argument. It appends the value to a global list (`to_print`), which will eventually be returned. Values must be appended in the order that they are printed to.

The scope statement creates a new lexical scope. The end of the scope statement removes the lexical scope. I have provided a skeleton class of a `SymbolTable`. You will need to implement this class.

1.1.2 Expressions

The expressions in this language can be addition, subtraction, multiplication, division, and exponentiation, given by the following symbols: (+,-,*,/,^). [1]. You must also allow the use of parenthesis.

You must enforce precedence of the operators as discussed in class using production rules. You must also enforce the left (or right) associativity as discussed in class. You are **NOT** allowed to use PLY's `precedence` keyword.

The operands in expressions are positive floating point numbers. That is, a number with an optional decimal point (.). You do not need to parse scientific notation. You should not have leading or trailing zeros. You may have a zero immediately before the decimal point (.). The operands can additionally be variables. All values should be interpretted as floating point values internally.

[1] the last symbol is the carrot symbol. Latex doesn't like it

1.1.3 Exceptions

There are two exceptions you must raise when different errors are encountered. I have provided both exception declarations at the top of the skeleton file.

If a variable is used without having been assigned earlier, then raise a `SymbolTableException`. If a variable is assigned, then it can only be used in its current scope, or any scope contained in the current scope. Any uses outside of this should raise a `SymbolTableException`.

If there is a lexer or parser error, you should raise a `ParsingException`.

1.1.4 Testing

I have provided a testing harness `tester.py`. It takes in a python test file and runs the provided tests. I have provided some tests in `provided_tests.py`. Please look over that file for examples of the test input and for examples of what programs should return. Your solution must pass these tests as a starting point. To run the tester, simply run:

```
python tester.py provided_tests
```

Please add *five* additional tests in the file `my_tests.py`. Please provide each test a name that includes your cruzid as a prefix. Include a short comment for each test to describe what parts of the interpreter you are testing; try to achieve some notion of coverage in your test cases. Feel free to add more than five tests. If you have questions about the format, please ask.

1.2 Grading

You will be tested on several components:

- Correctness: I will run a suite of tests on your interpreter. If the right answers are produced, you will receive all points in this category. You must avoid ambiguity through production rules. You are not allowed to use PLY's built-in `precedence` functionality.
- Unambiguous grammar: You may lose points if your grammar is ambiguous. Make sure that PLY does not give any shift/reduce warnings to get full points here.
- Comments and clarity: Please document your code. You don't need tons of comments, but your code should be readable.
- Tests: If you have added 5 tests `my_test.py` and if you can justify some notion of coverage with your tests.

2 Parsing regular expressions with derivatives

The template for this part of the assignment can be found in `homework1_code/part2`. In this part of the homework, you will be matching regular expressions (REs) using the "Parsing with Derivatives" method, detailed in [1]. This approach treats REs as a tree structure and recursively generates new regular expressions. We will use Lex and Yacc to parse the regular expression, creating an RE tree. We will then use derivatives to match strings to the RE.

The REs we will be considering consist of single characters (upper-case, lower-case, and digits). The operators are concatenation (.), union (|), Kleene star (*) and the optional operator (?). Note that concatenation here is an actual operator instead of implicit. You should also support parenthesis.

Your assignment is constrained to `skeleton.py`, and `my_tests.py`. Please do not change the `match_regex` function. It takes in a regular expression and a string and returns true or false depending on if the RE matches the string. It should go without saying that you cannot use a regular expression library in your implementation (outside of your PLY tokens).

Your tasks are as follows:

- Write a PLY grammar (tokens and production rules) to parse regular expressions as defined above. For now you can simply write `pass` as production actions. In this part of the assignment you can use the PLY's `precedence` variable if you'd like.
- Write Python objects to represent a RE tree (i.e., an AST). I suggest having an object for leaf nodes (single characters), and then intermediate nodes to represent the RE operators. Concat and Union will have two children; star and optional will have 1 child.
- Revisit your production actions to produce an RE AST. For example, the production rule that matches a single character should return an AST leaf. The production rule that matches an operator, e.g. concat, should return an AST node where the children are two REs that are being concatenated.
- Following the reference, and the slides, implement the `nullable` function over an RE AST.
- Following the reference, and the slides, implement the `derivative_re` function over an RE AST.
- for both of the above functions, you will need to think of how to implement the optional operator. There are several possible ways to do this.

2.1 Testing

1. Test your implementation. You can run your script standalone, i.e. running: `python skeleton.py` and it will run the test RE and strings at the bottom of the file.
2. You can test your implementation using my testing script `tester.py`. It takes in a python test file in the same way as part 1 of the homework.
3. Similar to part 1, please add 5 more tests to `my_tests.py` and briefly explain how they provide coverage of your implementation.

2.2 Grading

You will be graded on several components:

- Correctness: I will run a suite of tests on your RE matcher and check for the expected outcomes.

- Comments and clarity: Please document your code. You don't need tons of comments, but your code should be readable.
- Tests: If you have added 5 tests to both of `my_tests.py`.

3 Submission

Please zip up the `HW1` directory containing all your work and submit this zip file to Canvas under Homework 1.

3.1 References

[1] Scott Owens, John Reppy, Aaron Turon. "Regular-expression derivatives reexamined". https://www.ccs.neu.edu/home/turon/re-deriv.pdf