CSE211: Compiler Design

Homework 1: Parsing Overview
Assigned Oct. 4, 2021
Due Oct. 18, 2021

- This assignment has two parts: part 1 implements a simple interpreter and part 2 deals with parsing regular expressions with derivatives. The first part is worth 60% of the homework grade and the second part is worth 40% of the grade.
- All required software has been pushed to the class docker. Please do a `docker pull` command as described on the class webpage to get required software.
- Skeletons for this assignment can be found in two places: either in a zip from class webpage at: https://sorensenucsc.github.io/CSE211-fa2021/assignment_data/homework1_code.zip, or in the docker image with base path `assignment_data/homework1_code`. I suggest you copy the skeleton directory to a working directory.
- 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 you do! Ask if you have questions about docker! Any questions about docker are allowed in the class slack. Conversely, please help out your classmates if you know an answer.
- Please read the instructions in their entirety before asking questions.
- There are likely typos in the homework (writeup and code)! Please let me know if you see any. This is still a new course and I am evolving the homework constantly. Along these lines, if you have ideas for the homework, please let me know!

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 `test.py`. I have provided the start 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 an integer 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 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, which will eventually be returned. Values must be appended in the order that they are printed to in the input string.

The scope statement creates a new lexical scope. The end of the scope statement removes the lexical scope, as discussed in class. I have provided a skeleton class of a `SymbolTable`. I suggest implementing it as a stack of dictionaries and mapping variable IDs to their value.

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

The operands in expressions can be 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. The operands can additionally be variables.

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 (in an expression) without being assigned (or if it was assigned outside of the current scope), you should raise a `SymbolTableException`.

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

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

1.1.4 Testing

I have provided a testing file `test.py`. There are a few easy test cases in the file. Your solution must pass these tests as a starting point, i.e. there should be no assertion failures.

Please add *three* additional tests in each category. Feel free to add more. If you have questions about the format, please ask.

1.2 Submission

Please zip up the directory containing `skeleton.py` and `test.py` the `ply` directory. Submit the zipped directory to Canvas. I should be able to run `python3 test.py` and not see any assertion errors on the docker image.

1.3 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 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 3 tests in each category in `test.py`.

2 Parsing regular expressions with derivatives

The required code for this part of the assignment can be found in `homework1_code/part2`. In this part of the homework, you will be parsing regular expressions 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 characters (upper-case, lower-case, and numbers). The operators are concatenation (.), union (|), and Kleene star (*). Your job is to implement several missing functions and add a new RE operator: the optional operator (?).

Your assignment is constrained to `skeleton.py`, `tester.py`, and `tester_optional.py`. Read through this code and understand the structure of the RE tree structure, the tokenizer (lex) and the parser (yacc).

Your tasks are as follows:

1. Implement the nullability operation for the union operator on line 115. You will see a comment and a `raise NotImplementedError`.
2. Implement the derivative operation for the Kleene star operator around line 163. You will see the `raise NotImplementedError`

- *hint: look at the definition in [1] Use the functions provided to create a new regular expression.*

3. Implement the derivative operation for the concatenation operator around line 170. You will see the `raise NotImplementedError`
   - *hint: remember the nullability function returns a regular expression.*
4. Add in support for the unary *optional* regular expression operator (?). This operator matches zero or one instances of the sub-expression. For example: the regular expression `f.l.o.w.e.r.s?` matches the strings `{flower, flowers}`. The regular expression `e.x.c.i.t.e.(m.e.n.t)?` matches the strings `{excite, excitement}`. You should do this in steps:
   - Add a token for '?'
   - Parse the operation. It should be at the same precedence as the Kleene star (*) operator and evaluated with right-associativity.
   - There are several ways to implement the operator. I suggest you have the parser return a regular expression that is equivalent to the definition of the optional operator (figuring out this regular expression is up to you!). The harder route would be to make a new optional operator in the RE tree, and implement both the nullability and derivative function for the optional operator. *I would not recommend 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 will run many strings and REs and report errors. Simply run it as `python tester.py` This script tests only the concatenation, union, star and parenthesis operations. The `tester_optional.py` script additionally tests the optional (?) operator.

As part of your assignment, please add 3 tests to `tester.py` and 3 tests to `tester_optional.py`.

2.2 Submission

Please zip up your directory containing `skeleton.py`, `tester.py`, `tester_optional.py`, and `PLY`. Submit this zip file to Canvas under "Homework 1 Part 2".

2.3 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.
- 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 3 tests to both of `tester.py` and `texter_optional.py`.

2.4 References

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