CSE211: Compiler Design

Homework 1: Parsing Overview
Due Oct. 22, 2020

Oct. 8, 2020

- Read through instructions fully before beginning assignment; the testing scripts described later can help you while developing!
- These assignments are done with Python 3; you'll need to explicitly call `python3` on the server unless you switch it in your environment. The default `python` command defaults to version 2.7.

1 Parsing Regular Expressions with Derivatives

This problem should introduce you to parsing using a Python implementation of Lex and Yacc (PLY). We 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 (?).

1.1 Preliminaries

1. Read through the first 7 pages of [1] to refresh yourself on parsing REs with derivatives
2. Find the assignment skeleton at `/home/tsorensen/public/homework1/part1`. Make a copy over to some homework1/part1 directory within your home directory. Make sure to include the subdirectory for PLY and the tester files. e.g. run the following:

   ```
   cp -r /home/tsorensen/public/homework1/part1/* ./
   ```
3. Your assignment is constrained to `skeleton.py`. Read through this code and understand the structure of the RE tree structure, the tokenizer (lex) and the parser (yacc).

1.2 Technical work

1. Implement the nullability operation for the union operator on line 115. You will see a comment and a `raise NotImplementedError`.

- *hint: look at the implementation of the nullability operation for the concatenation operator just above. Use the* `mk` *functions to build a new RE to return. Recursion is your friend!*

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 a similar precedence as the Kleene star (*) operator. I do not mind if it is higher or lower, but it should not be higher than parenthesises, or lower than concatenation. Ideally it would be the same precedent as the Kleene star, evaluated with right-associativity, but we did not cover that in class. If you read the PLY documentation you can try it out!
   - There are several ways to implement the operator. I suggest you have the parser return a union regular expression that is equivalent to the definition of the optional operator (figuring out this union 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 that approach.

1.3 Evaluation

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.

1.4 Submission

Please create a directory in your home directory called: `submissions`. Within that directory, create a directory called `homework1_part1`. Move your `skeleton.py` into that directory. Do not make this directory readable by anybody else. I have admin privileges that will allow me to copy from there.

1.5 References

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

2 Parsing a Simple Programming Language

Using your experiences with PLY from question 1, finish implementing a parser for a simple c-like programming language. You do not need to do anything with the language except for parsing. Given this, your Yacc production rules should only just contain `pass` (I have provided examples). Like in part 1, I have provided a skeleton for you. Please use this skeleton because it is set up in a way that I can automatically test.

2.1 Preliminaries

1. Review the tokenizer and parser from part 1 to understand how to implement tokens and production rules in PLY.

2. Review how to write production rules that take precedence into account. Consider the operators for the regular expression in part 1. This topic is also covered in class slides.

3. Find the assignment skeleton at `/home/tsorensen/public/homework1/part2`. Make a copy over to some homework1/part2 directory within your home directory. Make sure to include the subdirectory for PLY and the tester files. e.g. run the following:

   ```
   cp -r /home/tsorensen/public/homework1/part2/* ./
   ```

4. Your assignment is constrained to `skeleton.py`. Read through this code and to understand the structure and what you will be implementing.

2.2 Technical work

1. Implement the tokens around line 22. Most of these will be a single character. The notable exception is that the `NUMBER` token should be able to match floating point numbers. I have implemented the ID and keyword tokenizing for you.

2. Impelement the remaining production rules. I have provided the top level rule as a statement list, defined the different types of statements, and implemented the declaration statements. The remaining statements for you to implement are:

   - an assignment statement of the form: `ID ASSIGNMENT expression`. You will need to also implement the expression rules which should match arithmetic expressions consisting of `{(),*,+,<}`, in that precedence. Using our method discussed in class, this will require 5 levels of expression parsing. You can also use fewer if you explicitly define precedence as described in the PLY documentation. Examples of assignment statements are:

     ```
     x = 5
     tmp = (5+6) * 5.6 < 0 + z
     ```

- an if statement of the form:
  `IF OPEN_PAREN expression CLOSE_PAREN OPEN_BRACE statement_list CLOSE_BRACE`.
  You should be able to implement this with one production rule. An example of an if statement is:
  `if (x < 5) {x = x + 1;}`
- a for statement. This statement should mirror the C for statement. It starts with the keyword `for`, followed by parenthesis a list of: an assignment statement, an expression, and an assignment expression. There is no semicolon required after the last assignment expression. After this list, there is a list of statements enclosed in braces. An example is:
  `for (x = 0; x < 5; x = x + 1) {z = z + 1;}`

2.3 Evaluation

1. Test your implementation. You can run your script standalone, i.e. running: `python skeleton.py` and it will run the test program at the bottom of the file.
2. You can test your implementation using my testing script `tester.py`. It will run many programs and report errors. Simply run it as `python tester.py`.

2.4 Submission

Please create a directory in your home directory called: `submissions`. Within that directory, create a directory called `homework1_part2`. Move your `skeleton.py` into that directory. Do not make this directory readable by anybody else. I have admin privileges that will allow me to copy from there.