CS 3100, Spring 2020, Assignment 7 – 100 pts total – Due Dates on Syllabus

1 Overview of Assignment-7

This assignment is designed to help you see the practical side of context-free parsing. Given that the files are a bit spread out, I'm making this description in the PDF document a bit longer. Please as usual find the places to fill your answers within `u1234567_asg7_Prob1234.ipynb`.

This notebook is kept under `13_Linearity_Amb/` Also, submit the fully executed notebooks that are included in support of this assignment (mentioned below in each subsection); the TAs may only selectively glance at the work there. *Details of how to answer each question are outlined in the main submission file*

`u1234567_asg7_Prob1234.ipynb`.

2 Details

There are four graded problems, with relevant files in these directories. We will describe each of the files and the questions associated.

Important: This assignment is more about reading code and describing what you've read concisely. It isn't as much work compared to prior assignments—except perhaps the volume of information you need to go through. Please ask questions if you are lost. Ideally you should be coding some of these files...but time does not permit me to assign coding exercises. Hopefully you'll be able to write similar mini-parsers in later projects.

2.1 Linearity and Ambiguity (AR)

File `CH11_Linearity_Ambiguity.ipynb` in `13_Linearity_Amb/` gives you an actual implementation of these ideas:

- Gives you an NFA
- This NFA is converted to a purely right-linear CFG documented at the top of `pda_if_sndlast_then_1`.
- This CFG is converted into the PDA `pda_if_sndlast_then_1`. which is run via `explore_pda('1001010', pda_if_sndlast_then_1)`.
- Then we show that a purely left-linear CFG can also be obtained, by reversing the original CFG.
- We encode this CFG using the PDA `pda_rev`.

QUESTION-1a (5 points): Once you make sure that the `iso_dfa` check succeeds (you are able to enter an RE that matches in cell 4), answer this:

why (do you think) that the call

`explore_pda('0101001', pda_rev, STKMAX=9)`

in cell 21 (below "Reversed PDA works, but needs STKMAX = ..") required a huge stack depth (of 9 for me) compared to the shorter stacks that were needed by `pda_if_sndlast_then_1`? Answer in a few sentences.

Indicate these in your answer: (1) The RE in cell 4. (2) The largest STKMAX below 9 where it did not finish (3) Does this behavior seem to be connected to the type of linearity in this grammar compared to the type of linearity in

`pda_if_sndlast_then_1` ?

QUESTION-1b (5 points): Considering the logic of these conversions, in a sentence or two explain why `explore_pda('0101001', pda_rev, STKMAX=9)` must result in an accepting run.

Answer this way: Indicate the language of `pda_rev` as a regular expression, and show that 0101001 is in the language of this RE.

QUESTION-1c (6 points): What is meant by an inherently ambiguous language? Why did the PDA `pda_inh_amb` generate two parses for `abc`? Answer by filling the prompts.

QUESTION-1d (4 points): How many parses will it generate for `aabbbccc`, and why? Answer without running the PDA.

2.2 Chatty Parser (SV)

In file File `CH11_Linearity_Ambiguity.ipynb` in `13_Linearity_Amb/`, I include a "chatty parser" by importing

`Def_md2mc_chatty`.

The idea is to give you a clue about how the command `md2mc` works. When you run

```
DO_pda_inh_amb = dotObj_pda(pda_inh_amb, FuseEdges=True)
```

you can see lots of printouts (showing how md2mc works inside). Your task is to answer the following questions on these printouts **plus** looking inside `Def_md2mc_chatty.py` which is in `Jove/jove`.

QUESTION-2a (5 points): Locating the printout, can you determine how markdown comments (the one that you start with `!!`) are processed by `md2mc`? Does it get handled by the lexer? Does it ever reach the parser? (Two sentences.)

QUESTION-2b (20 points): The parser CFG rules are under the `p_` functions (ignore the `p_error` rule). PLY does not care what you name these functions. For instance, the rules are present within the comment field of `p_you_are_hosed`, `p_dfa_md`, etc., all the way up to `p_one_label3` and `p_error`. Provide a neat listing of **all** the CFG rules that govern `md2mc` in its processing. There are 17 rules. **Add exactly one sentence describing each rule.**

2.3 Calculator (LT)

File `14_Calculator/Calculator_For_Asg7.ipynb` contains a calculator where we wish to add the SUCC (successor) operator. Note that the calculator prompt loop that you get is exited with an "END" (else it throws an error); then you can come to later cells in this notebook. This calculator code nearly works, but it throws an error on things like `!5`. It basically ignores `!` and moves on, which is dangerous (for the intended meaning). *Note that* `!` *is not factorial; I had to find a symbol I could use that did not clash with others in use. Hence I chose a prefixed successor via* `!`.

QUESTION-3a (9 points): Add all the indicated code items to make this output happen, should you insert it after **Your output must resemble...**.

```
calcparser.parse("3 * !!3 + !!3 * !!!3", lexer=calclexer)
```

Your answer: List all the changes you made to the code and details, including tokenization (how the SUCC token is generated), associativity, precedence with respect to unary negation, and semantics-handling.

QUESTION-3b (16 points):

For the test strings

```
test_strings = ['2+3', '2+!3', '3 + 3 * !!3 + !!3 * !!!3', 'x=3', 'y=4', 'z=x+y', 'z', 'z=x+!y', 'z']
```

Obtain the actual output generated by your code. Justify all the calculations that occured.

2.4 Derivatives (XL)

File

`15_Derivatives/CH10_Derivatives.ipynb`

presents derivative-based pattern matching.

You will be given help during lectures to answer these questions. Take good notes and then answer these questions.

QUESTION-4a (8 points): Explain why `c* & !b` is nullable. Explain how it got parsed. Then state all the rules involved.

QUESTION-4b (8 points): Explain why `c* & b*` is nullable. Explain how it got parsed. Then state all the rules involved.

QUESTION-4c (9 points): Explain how we can conclude that the RE `(a+bc+def+bd)*` matches string `bc`. Show the full derivation.