CS 3100, Fall 2018, Assignment 3
150 pts total (will be scaled to 30)

Note: Note: We provide TA names so that you know whom to approach for grading-related matters for each question. As for seeking help, *any* TA will help you with *any* of the questions.

In doing all these questions, you may want to keep a copy of our book's Appendix B in front of you (in hardcopy or PDF format). This lists *all* of Jove's functions that you can search down and employ in your work.

NOTE: In general, whenever you finish an exercise using Jove, please do not merely have a code-cell run and produce an answer. *Always write a few sentences (ideally in bulletted form) of comments before/after each code cell that contains a result that you produced.*

Submission instructions: which files?

- For all "Written" parts, hand-write neatly or typeset.
- For all "Jove code modification" parts, do the modification of the ipynb files and run all cells. Then submit on Canvas. The files have to be renamed to the following:
  - `J3_Unid_Somewhat_Chatty_Parser.ipynb` (fill UNID)
  - `J3_Unid_Calculator.ipynb` (fill UNID)
- You'll be modifying two Python files; they are
  - `Def_RE2NFA.py`
  - `Def_md2mc.py`

  You must document the modifications inside of these files (put a Python comment flagging the changes). Then submit the modified Python files also via Canvas, named as follows:
  - `J3_Unid_Def_RE2NFA.py`
  - `J3_Unid_Def_md2mc.py`
- **So, in total, submit five files:**
  - Two ipynb
  - Two Python
  - The PDF of your answers as another file `A3_Unid.pdf`

1. **75 points:** (Maryam, Harshitha, and Arnab)

   (a) **25 points:** (Maryam) Clearly explain how we proved the consistency and completeness of the CFG

   ```
   S-> '' | a S b S | b S a S
   ```

   in Section 11.4 through the "hill/valley plot." Your answer must present the following details:

   - How you can generate the first 8 strings (in numeric order) of equal number of `a`'s and `b`'s using this CFG (draw parse-trees for each such string).
   - How consistency was argued (in three well-chosen sentences)
   - How completeness was argued (in three well-chosen sentences)

Answer:

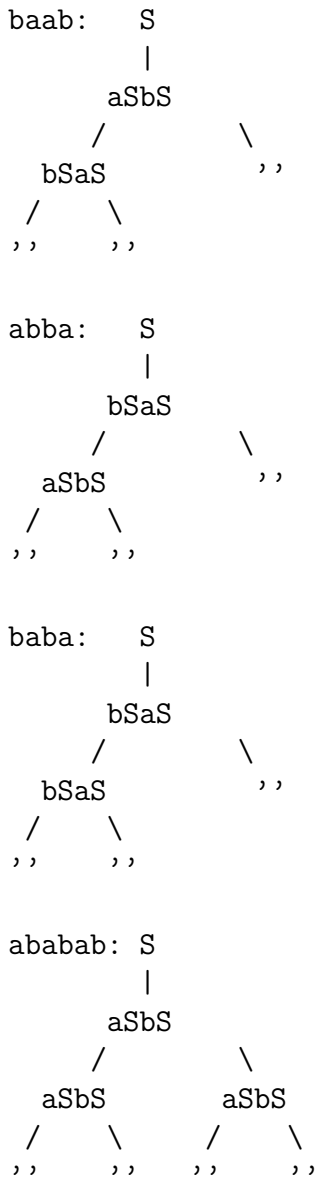
First 8 strings: ", ab, ba, abab, baab, abba, baba, ababab

```
'' : S
     |
    ''
```

```
ab: S
    |
   aSbS
  /    \
 ''    ''
```

```
ba: S
    |
   bSaS
  /    \
 ''    ''
```

```
abab:   S
        |
      aSbS
     /    \
  aSbS     ''
 /    \
''    ''
```



The book argued consistency by proving all right-hand sides of production are consistent. " is always consistent, and aSbS/bSaS are if S is assumed to be consistent. Because everything on the right-hand side of S is consistent, then S must be consistent.

To prove completeness, we start by assuming all strings of length $\leq$ n are consistent, and consider a string s of length n+2. Using a hill/valley plot to visualize s, then s either will or won't cross the x axis. Either way, there will be a first letter which departs the line from the x axis, and eventually one of the other letter will bring the line back to the x axis; is this a and b pair is removed from s, then you are left with a string of length n that must be

consistent. Therefore we know our CFG is complete, and no SS production is needed.

(b) **25 points:** (Harshitha) Grammar ambiguity is discussed in Section 11.5. There we also discuss how to disambiguate grammars. An example of a disambiguated grammar is shown here.

```
Grammar G2:
E -> E + T | T
T -> T * F | F
F -> 1 | 2 | 3 | 4 | ~F | (E)
```

Describe how this grammar helps parse the following expressions unambiguously when in fact the ambiguous grammar

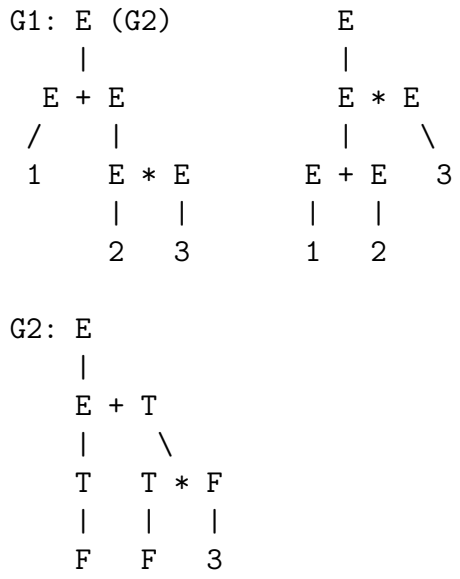
```
Grammar G1:
E -> 1 | 2 | 3 | 4 | ~E | E + E | E * E | ( E )
```

produces multiple parse-trees for each of these expressions. Specifically show the multiple parses for these expressions using G1, and identify which of these parses G2 arrives at (draw the various parses produced by G1, the unique parse by G2, and a sentence or two for each of these cases):

- `1+2*3`
- `~1 * ~2 + ~3`
- `~1 * (2 + 3) + 4`

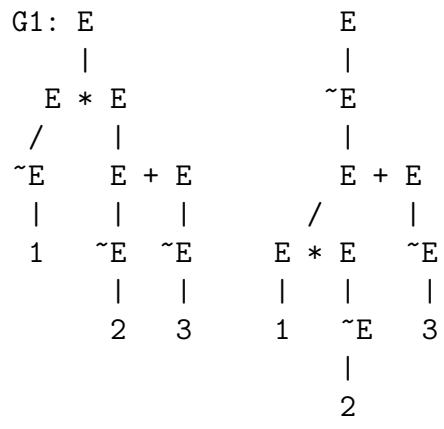
Answer:

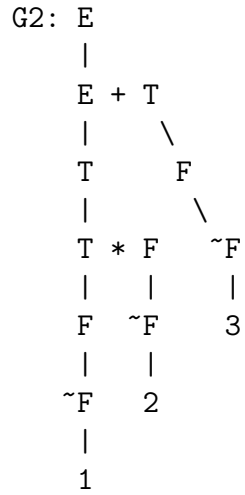
- `1+2*3`



G1 produces 2 parse trees for this expression, the first one being the most similar to the tree produced by G2. Because of this G1 does not establish an order of operations.

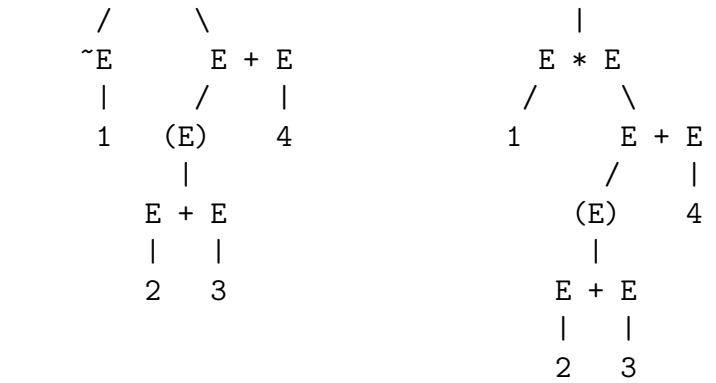
- `~1 * ~2 + ~3`

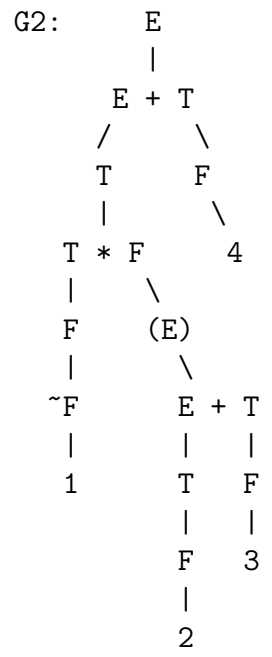




Shown are two very different parse trees produced by G1, with the first one being more like the tree produced by G2. Because the unary minus can be applied at almost any 'depth' many different trees are possible for this expression. Additionally, the multiplication and addition can be done in any order.

- `~1 * (2 + 3) + 4`





Again due to the unary minus and ambiguity in the order of addition and multiplication many different trees are possible from G1. The first one is most like the tree produced by G1

(c) **25 points:** (Arnab) Study the derivatives chapter (Chapter 10) and answer these questions with respect to two examples from Section 10.2.1:

i. (15 pts) How was $ac \in L(!(ab))$ concluded in the affirmative? Go through the table "Precondition, Derivation, Explanation" given in Chapter 10. Mention the rules fired (also mentioned when we discussed the example $ac \in L(!(ab))$). How were these rules implemented in Figure 10.3? Write you answer as follows:

- Write one sentence on the first rule invoked when handling $ac \in L(!(ab))$, mentioning what that rule does. Then write another sentence explaining what the code in Figure 10.3 does. Finally write one sentence conveying

whether we are ready for a nullability test at the end of this rule or not, and why.
- Then write another sentence for the second rule picked, etc.

ii. (10 pts) How was $ab \in L(!(ab))$ concluded in the negative? Write an answer similar to what you wrote for the earlier example above.

(a) How $ac \in L(!(ab))$ was concluded in the affirmative
- The first rule invoked is $!E \xrightarrow{c} !E_{1c}$, which does $!(ab) \xrightarrow{a} !b$. The code for this rule would need to take a regular expression, negate it, feed it the given letter, then negate it again. We are not ready for a nullability test after this rule because there is still a character in the regex.
- The second rule invoked is $c \xrightarrow{d} \emptyset$, which does $!b \xrightarrow{c} !\emptyset$. The code for this rule would need to take a regular expression and an input, feed the input to the regex, then return " or the empty set as necessary. We are ready for a nullability test after this; we know that $\emptyset$ is not nullable, so $!\emptyset$ must be nullable.

(b) How $ab \in L(!(ab))$ was concluded in the negative
- The first rule used is $!E \xrightarrow{c} !E_{1c}$, which does $!(ab) \xrightarrow{a} !b$. There is still a character in the regex, so we are not yet ready for a nullability test.
- The second rule used is $c \xrightarrow{c} \varepsilon$, which does $!b \xrightarrow{c} !\varepsilon$. We are ready for a nullability test after this rule; because ε is nullable, then $!\varepsilon$ must not be.

2. **40 points:** (Alex) You are given a calculator program that you can run and test. This is in file `J3_Unid_Calculator.ipynb` on the Jove's git (when you pull, you'll get an `asg3/ directory` plus all the support files). Here are your tasks:

(a) (5 points, Alex) Write down the CFG implemented by the calculator. You can glean it by going through the contents of `J3_Unid_Calculator.ipynb` (the CFG rules sit as comments within the `p_` functions of PLY). Write the CFG rules as follows:

```
S -> A B C
A -> a A | epsilon
...etc...
```

Use the token names found inside PLY's productions.

Answer:

```
E - Expression, S - Statement
  S -> NAME EQUALS E | E
```

```
E -> E PLUS E | E MINUS E | E TIMES E | E DIVIDE E
   | MINUS E | LPAREN E RPAREN
   | NUMBER | NAME
```

(b) (15 points, Alex) Run this file and test one instance of all the calculator inputs. This means, if the calculator has a `x=33` command (store `33` into register `x`), then exercise that. If the calculator supports an expression `33+x-(-33)`, then test that. If the calculator supports an expression `-33+33`, test that. Likewise, cover the grammar with at least a dozen expressions that walks through the CFG of this calculator. You most certainly can study-up `https://www.dabeaz.com/ply/` and `https://www.dabeaz.com/ply/example.html` which is a full tutorial (some code adaptations have been made) from the latter.

Output:

```
calc > y=20
calc > y
20
calc > y+y
40
calc > y*3
60
calc > x = 4
calc > y/x
5.0
calc > (33 + (-(x*(-y+5)/2)/y))
34.5
calc > -x+y
16
calc > 2(3)
Syntax error at '('
calc > 5
5
calc > 1+2-3/4*5
-0.75
calc > 1+(2-(3/(4*5)))
2.85
calc > -(5/x)
-1.25
calc > -5/x
-1.25
calc > 5/-x
```

`-1.25`

These expressions together use every possible transformation of E, and aim to prove that my depiction of the grammar is correct.

(c) (20 points, Alex) Add the "SUCC" command (for "successor") represented by `!33` which must of course return `34`. See the code for suggestions on how to add this in. Test that it works. You must add the correct grammar rules also to make this work. Test it thoroughly, showing a dozen expressions as above, but now with "SUCC" (`!`) used. Here are examples of expressions that must pass (more are in the `.ipynb` file given to you):

```
calc > 2+3
5
calc > 2+ !3
Illegal character '!'
(The above would work, yielding 6 as soon as you add the ''SUCC'' command)
calc > x=3
calc > y=4
calc > z=x+y
z
7
calc > z=x+!y
Illegal character '!'
(The above would work, yielding 8 as soon as you add the ''SUCC'' command)
```

Output:

```
calc >  2+3
5
calc >  2+!3
6
calc >  x=3
calc >  y=4
calc >  z=x+y
calc >  z
7
calc >  z=x+!y
calc >  z
8
calc >  1*!8
9
calc >  !(5)
6
```

```
calc >  !-7
-6
calc >  -!7
-8
calc >  !-(7 / 3)
-1.3333333333333335
calc >  1+!(!(2-3)*4/!5)
2.3333333333333335
calc >  !(-((-4)))
5
```

These expression fully test the associative properties of the SUCC command

3. **45 points:** (Paridhi and Paul)

   You are given a file `J3_Unid_Somewhat_Chatty_Parser.ipynb`. Run it, and you can see that for `re2nfa` it now prints some of the internal actions. It also does that for `md2mc`.

   This Jove file is only somewhat chatty. The goal is to make it very chatty and informative. For this, your tasks are to modify `J3_Unid_Somewhat_Chatty_Parser.ipynb` and after that, run everything, and if all works as per the specification below, turning it into the "garrulous" version of this file.[1]

   (a) (5 points, Paul) First, before you begin your task of making the parser chattier, write down the CFG implemented by the RE to NFA converter. You can glean it by going through the contents of `Def_RE2NFA.py` (the CFG rules sit as comments within the `p_` functions of PLY). Write the CFG rules as follows:

   ```
   S -> A B C
   A -> a A | epsilon
   ...etc...
   ```

   Use the token names found inside PLY's productions.

   Answer:

   ```
   E -> E PLUS C | C
   C -> C O | O
   O -> O STAR | (E) | EPS | STR
   ```

[1] Garrulous means "very chatty."

(b) **15 points:** (Paridhi) Now modify the somewhat chatty version of the file `Def_RE2NFA_chatty.py` in the `jove/` directory to make it become chattier, printing details about all the interesting internal things that the `re2nfa` command does. This way, you "see the parser actions" from outside.

The connection between `J3_Unid_Somewhat_Chatty_Parser.ipynb` and `Def_RE2NFA_chatty.py` is this: the former doees an import of the latter. Thus by making the latter (Python file) print more, you get to see these chattier effects when you run `J3_Unid_Somewhat_Chatty_Parser.ipynb`.

Of course don't make gobs and gobs of info come out; but producing 8-10 useful lines of information for each `re2nfa` command is a good rule of the thumb.

(c) **15 points:** (Paridhi) Also in about six neat bulleted steps, explain how the `re2nfa` algorithm actually parsed the given RE and produced an NFA. Specifically, walk through how it ended-up parsing the RE `(a* + ''*)*`. Mention the grammar rules involved and how the NFA was formed. Specifically, for each of the

- Type `dotObj_nfa(re2nfa("(a*+''*)*"))` to see that not only did many relevant rules get fired, but you also saw the correct NFA appearing. Confirm this with a sentence in your answer (one bullet).
- Then Type the individual sub-expressions, i.e., these, and make sure that the correct parsing rules are being fired (one bullet each):
  ```
  dotObj_nfa(re2nfa("''"))
  dotObj_nfa(re2nfa("''*"))
  dotObj_nfa(re2nfa("a*"))
  dotObj_nfa(re2nfa("(a*+''*)"))
  dotObj_nfa(re2nfa("(a*+''*)*"))
  ```

Answer:

- Firing `dotObj_nfa(re2nfa("(a*+''*)*"))` triggered all the correct token detections and NFA creation methods, as well as creating a correct NFA.
- `"''"` detected only the epsilon symbol.
- `"''*"` detected the epsilon then applied DFA star to it.
- `"a*"` detected the a and star'd it.
- `"(a*+''*)"` star'd the a and epsilon, plus-connected those NFA, then detected the parens.
- `"(a*+''*)*"` did the same as the previous sub-expression, then star'd the resulting NFA

(d) **10 points:** (Paul) As a similar exercise to Part 3b, modify the chatty version of the file `Def_md2mc_chatty.py` in the `jove/` directory to make it garrulous (very chatty), printing details about all the interesting internal things that the `md2mc` command does. Make these changes so that the DFA, NFA, and PDA actions of

`md2mc` become garrulous. This way, you "see the parser actions" from outside.

After the aforesaid modification to the Python file `Def_md2mc_chatty.py`, if you run `J3_Unid_Somewhat_Chatty_Parser.ipynb`, then each `md2mc` command becomes chattier.

The grading will depend on how informative the garrulous sessions are, and how much useful information is brought out. Of course don't make gobs and gobs of info come out; but producing 8-10 lines for each `md2mc` command is a good rule of the thumb. Specifically, write your answers in terms of five bullets covering these.

You should introduce print commands to discover these answers automatically (and in the bullets, mention the print commands that helped you figure out things).

- How is the "DFA" keyword parsed?
- How are the things you write in one line of a DFA description parsed?
- How are comments (that start with `!!`) handled?
- How is one line of a "PDA" description parsed?
- How are *all* the lines after the "PDA" keyword parsed?

Answer:

- The DFA keyword is parsed by a CFG with transformation
  `md -> DFA lines | NFA lines | PDA lines | TM lines`
- A single line of the DFA is parsed by getting the initial state and final state, then determining the delta. The print statements that helped here were the ones in p_one_line and p_ID_or_EPS_or_B.
- Comments are detected then ignored, as seen in t_COMMENT
- PDA lines are parsed by getting the initial and final states, as well as modifications to the stack. This is described in form_delta.
- Each line after the PDA keyword is broken off as a single line from a group of all lines. The line is then parsed as necessary. Print statements in p_lines1 and p_lines1 helped me figure this out.