CS 81, Spring 2019
Problem Set 9: Parsing and Turing Machines
Due: Tuesday, April 16 at 11:59 PM

Please read Handout 6 on Chomsky Normal Form before embarking on this assignment.

Challenge 1: Parsing! [50 Points]

In this problem, you will implement the CYK parsing algorithm for context-free grammars. Start by downloading the starter file `parse.py` and modify it as follows:

Part 1: Producing Parse Trees. The `cyk(rules, variable, string, memo)` function takes as input the rules of a grammar in Chomsky Normal Form (CNF), a variable in the grammar, a string to parse, and a memo dictionary which is initially `{}` (an empty dictionary) and returns `True` if the string can be generated from the grammar starting from the given `variable` and returns `False` otherwise. Your task is to modify this code so that rather than simply returning a Boolean, it returns a tuple: If the string is not in the language it returns `(False, None)`. But, if the string is in the language, it returns a tuple of the form `(True, ParseTree)` where `ParseTree` is a parse tree for that string and grammar. The format of a parse tree is one of the following:

1. `(Root, Left, Right)` where `Root` is a variable with a rule of the form `Root -> AB` and `Left` and `Right` are parse trees rooted at `A` and `B`.
2. `(Root, x)` where `Root` is a variable with a rule of the form `Root -> x` where `x` is a terminal.

For example, here is a grammar in Chomsky Normal Form for the language $L = \{0^i 1^i \mid i \geq 1\}$:

$$\begin{aligned} S &\rightarrow CB \mid AB \\ C &\rightarrow AS \\ A &\rightarrow 0 \\ B &\rightarrow 1 \end{aligned}$$

Our Python representation would look like this:

```
V1 = ["S", "A", "B", "C"]
Sigma1 = ["0", "1"]
R1 = {
    "S" : [ ["C", "B"], ["A", "B"] ],
    "C" : [ ["A", "S"] ],
```

```
    "A" : [["0"]],
    "B" : [["1"]]
}
S1 = "S"
grammar1 = (V1, Sigma1, R1, S1)
```

Note that the grammar is a 4-tuple comprising a list of variables (each of which is a string of length one), a list of terminals (each of which is a string), a rules dictionary in which there is a rule that associates each variable with a list of lists, each of which contains either two variables or one terminal.

Now, when you run your `cyk` function on the grammar rules in Chomsky Normal form, a string, and an empty dictionary (which serves as the memo for memoization), you'll get this back:

```
>>> cyk(R1, "S", "01", {})
(True, ('S', ('A', '0'), ('B', '1')))
>>> cyk(R1, "S", "001", {})
(False, None)
>>> cyk(R1, "S", "0011", {})
(True, ('S', ('C', ('A', '0'), ('S', ('A', '0'), ('B', '1'))), ('B', '1')))
```

Note: In total, you will add or change only around five lines of code!

Part 2: Converting a Grammar to CNF! The next step is to write a function called called `cnf(grammar)` that takes a context-free grammar as input and returns an equivalent grammar in Chomsky Normal Form. You should assume that ϵ does not appear in the grammar at all. Here's an example of the conversion working on the grammar from Handout 6 (this grammar is called `grammar3` in the provided `parse.py` starter code):

```
V = ["S", "A", "B"]
Sigma = ["a", "b"]
R = {"S": [["A", "S", "A"], ["a", "B"], ["A"]],
     "A": [["B"], ["S"]],
     "B": [["b"]]}
S = "S"
grammar = (V, Sigma, R, S)

>>> cnf(grammar)
(['S', 'A', 'B', 'Ta', 'Tb', 'C0', 'C1'],
 ['a', 'b'],
 {'Tb': [['b']],
 'A': [['b'], ['Ta', 'B'], ['A', 'C1']],
```

```
 'C1': [['S', 'A']],
 'Ta': [['a']],
 'C0': [['S', 'A']],
 'S': [['Ta', 'B'], ['b'], ['A', 'C0']],
 'B': [['b']]},
'S')
```

Note: Divide this task into smaller helper functions as suggested in the Handout. In total, our sample solution is about 70 lines of code.

Part 3: Putting it all together! There's anothing to do here! We've provide the `parse` function that calls the `cnf` and `cyk` functions to parse input. Test it on the grammars provided at the top of starter code file. Add one more grammar to that file called `grammar4` and include a comment to explain what that grammar encodes. Test your parser on that grammar too! Here's what things should look like:

```
>>> parse(grammar1, "0011")
(True, ('S', ('C', ('A', '0'), ('S', ('A', '0'), ('B', '1'))), ('B', '1')))
>>> parse(grammar2, "(())()")
(True, ('S', ('S', ('T(', '('), ('C0', ('S', ('T(', '('), ('T)', ')')), ('T)', ')'))),
 ('S', ('T(', '('), ('T)', ')')))
```

Challenge 2: Turing Machines in Prolog! [25 Points]

Given your long history with creating computational artifacts from pure logic predicates, you will now attempt to create a Turing machine simulator in Prolog. Given that a Turing machine consists of a finite state controller (DFA) and an input tape, this will not be too different from when you created PDAs in Prolog. You will be given a starter file `TMaccepts.pl`, which defines the shape of the accepts predicate you will need to define. From the file, you'll see

```
accepts(Q, LeftTapeReversed, RightTape) :-
```

which defines the general form of the predicate. The `Q` denotes the current state of your DFA and `LeftTapeReversed` is a list containing the contents on the tape to the left of the reading head (not including it), but in reversed order (which will make it easier to push things to your left). `RightTape` is a list containing the contents of the tape from the reading head, and extending to the right, up to but not including the first blank. The lists should not contain blanks (they should be implicit), so part of the assignment is to figure out a good way to handle and represent them implicitly. To recap, your input tape only will ever have a finite number of non-blank symbols on it, so we can split the tape into two parts at the read head, with the left side of the tape being stored in `LeftTapeReversed`

in reversed order (from right to left, i.e., from the tape head moving out towards the left). The first symbol of `LeftTapeReversed` should be the symbol immediately to the left of the tape head, and the last symbol in the list `LeftTapeReversed` should be the non-blank symbol furthest left from the tape head (namely, the symbol at the beginning of our input).

As an example, if your string is $w = 000111222$ (encoded as `[0,0,0,1,1,1,2,2,2]` on the tape) and your language is $L = \{0^i 1^i 2^i | i \geq 0\}$, your `accepts` predicate should accept beginning at a start state `q0` and given that tape (assuming a correctly coded Turing machine), namely

```
?- accepts(q0, [], [0,0,0,1,1,1,2,2,2]).
true.
```

Warning: This and the next problem are beta-tests, so this is experimental! There could be something I'm not thinking of that could prevent this whole scheme from working, though I'm fairly confident it can be done. If for some reason you come up against a road-block that you feel would prevent either of these problems from actually being completable, please let me know ASAP! Begin early and let me know of any pitfalls! Good luck!

Challenge 3: Designing a Turing machine for $0^i 1^i 2^i$ [25 Points]

Create a Turing machine in Prolog by defining the predicates for its transitions and accepting state, similar to what we did for NFAs and PDAs. Notice, our Turing machines have DFAs as finite state controllers, so you'll need to define outgoing transitions from each state for all possible symbols you could encounter. The alphabet Σ will be $\{0, 1, 2\}$ with $\Gamma = \Sigma \cup \{\text{␣}\}$. Make sure you test your TM using your accepts predicate!

Transitions will have the form

```
transition(Q, Symbol, NewQ, NewSymbol, NextMove)
```

where

- `Q` = current state
- `Symbol` = current symbol under the read head
- `NewQ` = new state
- `NewSymbol` = new symbol to write at the current position (before moving, overwriting `Symbol`)
- `NextMove` = which direction to move the tape head, wither `L` (left), `S` (stay), or `R` (right).

Some strategies we used in class may come in handy when deciding how you want to design your machine, and the idea of what you want to do shouldn't be too tricky. Put yourself in the place of the finite state controller, with the tape laid out before you, and decide how you could check whether the input has the form $0^i1^i2^i$ for $i \geq 0$. Notice, the empty string should be accepted by your machine since it is in the language, so part of this challenge will be to figure out how to handle that particular case, using a DFA.

Also take into account that you'll need to halt computation whenever you move into your accept state or your reject state. Consider ways to short-circuit Prolog's recursion in those cases (so you don't end up with endless recursion).