CS46 Homework 7

This homework is due at 11:59pm on **Saturday, February 5**. This is an 8-point homework. Note the unusual deadline.

For this homework, you will work with a partner. It's ok to discuss approaches at a high level with other students, your discussions should be just with your partner. The only exception to this rule is work you've done with another student *while in lab*. In this case, note who you've worked with and what parts were solved during lab. Your partnership's write-up and code is your own: do not share it, and do not read other teams' write-ups. If you use any out-of-class references (anything except class notes, the textbook, or asking Lila), then you **must** cite these in your post-homework survey. Please refer to the course webpage or directly ask any questions you have about this policy.

The main **learning goal** of this homework is to work with and think about Turing machines, and to implement one algorithm related to context-free languages.

Part 1. Your solution to this part should be written using LaTeX and submitted using **github** as a **.tex** file. Write clear and unambiguous implementation-level Turing machine descriptions. Your explanation of *why* and *how* a Turing machine works should be separate from your description of that Turing machine. Give the Turing machine description first, then explain it separately.

Another Turing machine extension. Let's consider an extension which gives a Turing machine tape which is infinite in both directions. We keep everything else the same, and specify that the input string is given on a tape which is blank everywhere else, with the read/write head on the first character of the input. So the starting configuration on input w in state q_0 looks like:

$$\cdots \sqcup \sqcup q_0 w \sqcup \sqcup \cdots$$

Notice that now the tape head can *always* move left, no matter where it is.

Prove that Turing machines with doubly infinite tape are no more powerful than standard Turing machines (that is, with singly-infinite tape).

Part 2. Your solution to this part should be written in python and submitted in your github repository as a file called **cnf.py**.

Recognizing context-free languages. Now that we're talking about computability and the Church-Turing thesis, it seems reasonable to ask how these concepts align with our actual usage of computers as they exist *in the real world*.

Recall that on lab 4 you implemented a program which checked whether a given NFA would accept a given input string. This involved converting the nondeterminism into a deterministic version, since the computers we work with everyday are deterministic. For **context-free languages**, the connection is not obvious: how can we implement the nondeterminism of a PDA? How do we know which generating rules to follow in a grammar? The good news: there is an algorithm to do this for grammars in Chomsky normal form, and by theorem 2.9, every grammar can be converted to Chomsky normal form! Reread Sipser pages 108-111 before you start this part.

Parsing Chomsky normal form: the algorithm. Given a grammar G in Chomsky Normal Form, and an input string $w = w_1 w_2 \cdots w_n$, we can design a polynomial time algorithm[1] to determine if $w \in L(G)$. The basic idea is to compute for all substrings $x = w_i \cdots w_j$, $i \leq j$ of w

[1] Don't worry if this terminology "polynomial time algorithm" is new to you, we will discuss it more in-depth in a few weeks.

if there is some rule in G that generates x. Define $V_{i,i+s}$ to be the set of all variables $V \in G$ that that can generate x. G can generate w if $V_{1,n}$ contains the start symbol $S \in V$.

Our algorithm for parsing strings computes the sets $V_{i,j}$ for $1 \leq i \leq j \leq n$. The algorithm proceeds in a series of rounds, where in each round, it considers substrings of length s, where $1 \leq s \leq n$.

In the first round, the algorithm sets $V_{i,i}$ to $\{A \in V \mid A \to w_i \text{ is a rule in grammar } G\}$.

For substrings longer than 1, the algorithm checks if the substring $x = x_i \cdots x_{i+s-1}$ can be broken into two smaller pieces at some index k, so we have $y = x_i \cdots x_k$ and $z = x_{k+1} \cdots x_{i+s-1}$ such that there is a rule $A \to BC$ in the grammar where $B \in V_{i,k}$ and $C \in V_{k+1,i+s-1}$. If such a rule exists, we add A to $V_{i,i+s-1}$. After finishing all rounds, we just need to check if $S \in V_{1,n}$.

A pseudocode summary is below. (For those who are interested, this is a rephrasing of the dynamic programming algorithm given on Sipser page 291.)

CANGENERATE(G,w)

```
1  for i = 1 to n
2       add A to V[i,i] if there is a rule A → w_i
3  for s = 1 to n − 1
4       for i = 1 to n − s
5            for k = i to i + s − 1
6                 if there is a rule A → BC where B ∈ V[i,k] and C ∈ V[k+1,i+s]
7                      add A to V[i,i+s]
8  if S ∈ V[1,n]
9       return TRUE
10 else return FALSE
```

Write a program that determines if a string w is accepted by a grammar G given in Chomsky normal form. Your program should take a CNF grammar and file containing test strings as input and determine if each string can be generated by the grammar. Fill in the required parts of the provided starter code in `cnf.py`.

The grammar format starts with a line containing the symbols of the alphabet separated by spaces, a blank line, and then the grammar rules. The first rule lists the start symbol on the left-hand side. The rule format is `V : A B` to indicate $V \to AB$. Note that variables are separated by spaces (so `A1` is a single variable, not two variables). The empty string ε is represented as `E`.

Three example grammars are included in your repository. The first, `grammar1.txt`, is a Chomsky normal form grammar generating our favorite $\{a^n b^n\}$. It goes with the test string file `string1.txt`. Sample output below:

```
$ more grammar1.txt
a b

S: E
S: A1 B
S: A B
A1: A S
A: a
B: b
```

```
$ python cnf.py grammar1.txt string1.txt
aabb: True
abab: False
a: False
aaaabbbb: True
ab: True
```

The second, `grammar2.txt` is a Chomsky normal form grammar generating expressions of balanced parentheses. Its test string file is `string2.txt`.

```
$ more grammar2.txt
( )

S: E
S: A1 B
S: A B
S: S S
A1: A S
A: (
B: )

$ python cnf.py grammar2.txt string2.txt
(): True
E: True
()(): True
((())): True
)(: False
((((): False
```

The third, `grammar3.txt` is a Chomsky normal form grammar generating $\{ww^R \mid w \in \{c, d\}^*\}$.

```
$ more grammar3.txt
c d

S: E
S: C C1
S: D D1
S: C C
S: D D
C1: S C
D1: S D
C: c
D: d
$ python cnf.py grammar3.txt string3.txt
ccccc: False
cccc: True
```

```
E: True
cdcd: False
ccddddddcc: True
ccddcc: True
ddcdd: False
dd: True
```