Part 1 - Context-Free Grammars

Create context-free grammars for the following languages. The grammar must be specified as a text file in the format below.

A. (Filename: cfgA.txt) Create a context-free grammar for the language that accepts all strings in the alphabet $T = \{0, 1\}$ where the number of 0s in x is divisible by 3, or the length of x is divisible by 3, or both.
B. (Filename: cfgB.txt) Create a context-free grammar for the language that accepts all strings in the alphabet $T = \{x, y, z\}$ that is equivalent to the regular expression: `^(xz)*(zy?|xx)+$`

Grammar File Format

The grammar file must be specified using the notation used in class. Some rules:

- Nonterminals can either be single letters or longer strings (with no spaces). If you use longer strings, you much separate the elements on the right-hand side of the production with spaces.
- The terminal alphabet, T, is provided with each problem. Anything that is not a terminal is assumed to be a nonterminal.
- The assignment requires the grammar to be a context-free grammar. This means that the left-hand side of each production only consists of a single nonterminal.
- You may use shorthand notation using |.
- The start symbol must either be *S* or *Start*.
- You may use *E*, *Empty*, or ***ε*** (*Unicode* U+03B5) to represent the empty string.

Example file using single letter nonterminals and shorthand notation with $T = \{a, b\}$.

```
S -> aS | X
X -> bX | E
```

Example file using longer string nonterminals and shorthand notation with $T = \{(,), +, *, 0, 1\}$:

```
Start -> BoolExpr
BoolExpr -> BoolExpr BoolOp BoolExpr | ( BoolExpr ) | BoolVal
BoolOp -> + | *
BoolVal -> 0 | 1
```

Unfortunately, there is no Python simulator for testing you grammars like the DFA in the last assignment and the Turing machines in Part 2. Your grammars will be graded manually.

Part 2 - Turing Machines

Create Turing machines based on the following specifications. The Turing machine must be expressed as files that can be used as inputs for the Turing machine simulator described below.

A. (Filename tmA.txt) Design a Turing machine on the input alphabet $\{d, e, f\}$ that accepts strings represented using this Python regular expression that uses back references: `r'^([de]*)f\1$'` For this machine, the final tape output does not matter. You will need to modify the tape in order to complete this exercise.

B. (Filename tmB.txt) Design a Turing machine on the input alphabet $\{x, y, z\}$ that removes all *z* characters from the input such that there are no gaps. If the input string is *xzzyxzy*, the output should be *xyxy*. For this machine, it does not matter if the string is accepted or rejected. Hint: The final string does not need to reside on the same part of the tape where it started.

Turing Machine Simulator

To run the simulator, the tm.py file needs to be in the current directory:

```
python3 tm.py tm_file input_string mode
```

The *mode* is optional (defaults to 3): 1 (step, interactively), 2 (step, non-interactive), 3 (display final state and output only).

For example,

```
python3 tm.py tmB.txt xzzyxzy 1
```

In a Turing machine, states are represented by nonnegative integers. The tape alphabet consists of any desired characters. The character `B` represents a blank character.

In the file, the first line contains two integers separated by a comma:

initial state, accepting state

Each subsequent line of the file refers to a different transition represented by a comma-separated list:

current state, current input, next state, new symbol, direction

For instance, the line `0,x,1,y,R` means that if the Turing machine is currently in state 0 with the head reading an *x* it will transition to state 1, overwrite the current tape cell with a *y*, and then move the head right one cell.

Notes:

- The five fields within a line are separated by commas. Spaces in the line are removed and ignored. You may have an optional comment at the end of the line that starts with a `#` character (like in Python).
- The symbol `B` represents a blank character on the tape.
- The direction can be either `L` or `R`.
- Only one accepting state (noted on the first line) is permitted. As with Turing machines, you cannot have transitions out of the accepting state.
- The same input pair (current state, current input) cannot appear multiple times in the file.
- The simulator halts when there is no transition for the current input pair. Just like real Turing machines, it is possible for the simulator to go into an infinite loop.

- After the required first line, lines that start with `#` are ignored and can be used for comments. You can also have blank lines anywhere in the file--these are ignored.
- You can use any set of nonnegative state numbers - they don't have to be consecutive. For example, you could have states 1, 2, 17, 99, and 1000. This is often useful to keep track of what you're currently doing by having number ranges to remember your current phase in your algorithm.
- If you do something wrong (such as type in the filename incorrectly), you will likely get an uncaught Python exception rather than a clear error message.

When running the simulator using interactive mode (choice 1), you will see the initial state of the machine. It will look something like this:

```
xyyzzz
^
0
```

The first line is the tape. The initial blanks on both sides of the input are not displayed until necessary. The subsequent lines show where the head is pointing to and the current state. Press Enter to advance the Turing machine one step. Continue to press Enter until the simulator halts. You can also press Ctrl-C to stop execution, necessary in infinite loops.

The Turing machine we used in lecture as an example (the one that finds two consecutive 1s) would have the following file, sample-tm.txt):

```
# Sample Turing machine
# accepts any binary string that has two consecutive 1's
# output is the input where the left-most pair of 1s is replaced by 0's

# start in q0, accept in q3
0, 3

# Transition function, δ
0, 0, 0, 0, R  # in q0 see a 0 then stay in q0 but move right
0, 1, 1, 1, R  # in q0 see a 1, move to q1 and the right
1, 0, 0, 0, R  # in q1 see a 0, so go back to start
1, 1, 2, 0, L  # in q1 see (a second) 1, so go to q2, replace 1 on tape with 0, move left
2, 1, 3, 0, R  # in q2, finish up by replacing the first 1 with a 0 and go to accepting state q3
```

Try this:

```
$ python3 tm.py tm-sample.txt 11011 2
11011
^
0

11011
 ^
1

10011
^
2

00011
 ^
3
```

```
output: 00011
state:  3
accepted
```

Provided Files

Download the **zip file** **(https://seattleu.instructure.com/courses/1602042/files/67685017/download?download_frd=1)** that contains:

- tm.py
- tm-sample.txt

Submission

Hand in ***four*** files:

1. cfgA.txt
2. cfgB.txt
3. tmA.txt
4. tmB.txt

HW7 Rubric

Criteria	Ratings			Pts
Context Free Grammar 1	**10 to >9.0 pts** **Correct - full credit**	**9 to >0.0 pts** **Partial Credit**	**0 pts** **No credit**	10 pts
Context Free Grammar 2	**10 to >9.0 pts** **Correct - full credit**	**9 to >0.0 pts** **Partial Credit**	**0 pts** **No credit**	10 pts
Turing Machine 1	**15 to >14.0 pts** **Passes all tests**	**14 to >0.0 pts** **Fails some tests**	**0 pts** **Fails too many tests**	15 pts
Turing Machine 2	**15 to >14.0 pts** **Passes all tests**	**14 to >0.0 pts** **Fails some tests**	**0 pts** **Fails too many tests**	15 pts
				Total Points: 50