CPSC 3400 Languages and Computation
Winter 2017

Homework 4
123 points
Due: Sunday, March 12 Midnight

Part 1 – Generators in Python

(17 points) Write a Python program (`hw4-1.py`) that does the following:

a. Create the following two functions:

- `genListItems(tupleList)`: Takes a list of tuples and yields each item in the tuple until the list is empty. The tuples can vary in size. In this assignment, tuples will only contain strings but the type of the tuple elements should not matter.

  If the input list is [('Hello', 'World'), (), ('CPSC', '3400', 'is', 'fun')], the function will yield these 6 items in order: 'Hello', 'World', 'CPSC', '3400', 'is', 'fun'.
- `createTupleList(fileName)`: The input parameter is a filename. The file consists of zero or more lines. Each line contains zero or more strings separated by a single space. Each line in the file is represented by a tuple where the words are stored in order. The function returns a list of these line tuples in order (the first tuple in the list is the first line, etc.).
  - Assume the file exists.
  - Assume there are no leading or trailing spaces. Obviously, the newline character should be removed before processing each line.
  - If a line is empty, the corresponding tuple is empty.
  - If the file contains zero lines, return an empty list.

b. The program has the following top-level functionality:

1. Get the name of an input file from the command line (using `sys.argv`). WARNING: Do not prompt the user for a file name.
2. Call `createTupleList` with the file name from step 1 and store the resulting list in the variable `tupleList`.
3. Use a for loop that iterates over the iterable function `genListItems` (called with `tupleList` as a parameter) that creates a dictionary that has strings for keys and the number of times the string occurs as a tuple as values.
4. Print the dictionary created in step 3.
5. Apply `filter` to `tupleList` to create a new list of tuples that does not contain any empty tuples. NOTE: The function passed into `filter` must be a lambda function.
6. Print the list created in step 5.

Example

If the input file contains:

```
char is less than short

short is less than or equal to int

int is less than or equal to long
```

Then, `tupleList` contains the following (note that `tupleList` does not need to be printed out):

```
[('char', 'is', 'less', 'than', 'short'), (), ('short', 'is',
'less', 'than', 'or', 'equal', 'to', 'int'), (), ('int', 'is',
'less', 'than', 'or', 'equal', 'to', 'long')]
```

The dictionary printed out in step 4 should look like this (ordering of keys may differ):

```
{'equal': 2, 'to': 2, 'less': 3, 'int': 2, 'short': 2, 'char':
1, 'is': 3, 'than': 3, 'long': 1, 'or': 2}
```

The list printed out in step 6 should look like this:

```
[('char', 'is', 'less', 'than', 'short'), ('short', 'is',
'less', 'than', 'or', 'equal', 'to', 'int'), ('int', 'is',
'less', 'than', 'or', 'equal', 'to', 'long')]
```

IMPORTANT! It is not sufficient to write a program that produces this output. You must follow the directions listed on the previous page.

Part 2 – Regular Expressions

(36 points, 12 points each) For each of the following items (a-c), create a regular expression that matches each of the item. Note that in ALL cases, the entire string must match.

a. An amount of money that is in the format `$3.56`.
You need to consider the following requirements:

- The dollar sign is required.
- The cents are optional. If the cents are specified, there must be exactly two digits.
- The amount of dollars must not contain leading zeros such `03`, `004`, or `0000`.
- If there are zero dollars, you must specify a single zero such as $0 or $0.32.

b. A string of digits that contain any number of digits but the digits 3, 1, and 4 exactly once. In addition, the 3 must occur earlier in the string than the 1 and the 1 must occur earlier in the string than the 4. An example of an acceptable string is "`098388719400`". Additional notes:

- The string is rejected if it contains anything other than a digits.

c. A Python list (output format) that only consists of one character strings that contain letters (upper or lower case) such as `['h', 'e', 'l', 'l', 'o']`.

If any of the following occur, it is not a match:
- Improperly formed list.
- Any item in the list is not a one-character string.
- Any one-character string in the list does not contain a letter.

Additional notes:
- You may assume that single quotes are used to represent the strings.
- There is exactly one space after every comma. No other spaces are allowed.
- An empty list is considered a match.

For this part, copy and modify **hw4-2-py** file that is provided with this assignment.

In the **hw4-2.py** file, you will notice three `re.compile` statements for variables a-c. These variables refer to items a-c respectively. Simply fill in the regular expression for each variable. Patterns that are too long to fit on one line can be split into multiple lines like this:

```
x = re.compile(r"dog|"
               r"cat|"
               r"cow|"
               r"duck")
```

In addition, you need to provide a test suite for each regular expression. The second part of the file has two sections for each item: a section for tests that match and a section for tests that do not match. Each section will consist of lines like this:

```
print(a.search(r"$3.56"))
```

Each section has one line to get you started. Add additional lines, using a different test string.

A portion of your grade will be based on the thoroughness of your test suite.

Part 3 – Substitutions

(22 points) Write a Python program (`hw4-3.py`) that has two command-line parameters: an input file name (first parameter) and an output file name (second parameter). The program reads in the input file and writes the same file to the output file except that all phone numbers are converted to a standard format.

In particular, any time there is a phone number in one of these formats:

a. `(555)123-4567`
b. `(555)-123-4567`
c. `(555) 123-4567`
d. `555-123-4567`
e. `(555)123.4567`
f. `(555).123.4567`
g. `(555) 123.4567`
h. `555.123.4567`

Replace it with this format:

`(555)123-4567`

Additional notes:

- Format *a* is the same as the intended format. It is permissible, but not required, to ignore phone numbers in this format since they are already in the correct format.
- In formats *c* and *g*, there is exactly one space after the).
- The phone number must be preceded by a space and followed by space unless it is at the beginning of the line (then a space is not needed before the phone number) and/or at the end of the line (then a space is not needed after the phone number).
- You may use multiple regular expressions if desired.
- Be sure to test your program with a variety of different phone number formats. Also include strings those are close, but do not match, one of the formats.

Part 4 – DFAs

Create DFAs for the following language specifications. The DFA must be expressed as files that can be used as inputs for the DFA simulator provided with this assignment (i.e., `dfa.py`) and described below.

1. (8 points, filename: `dfa-1.txt`) All strings on Σ = {A, B, C} that contain either (or both) of these substrings: BB and ACA.

2. (8 points, filename: `dfa-2.txt`) All strings on Σ = {X, Y} that contain at least two Xs anywhere in the string but no more than four.

3. (12 points, filename: `dfa-3.txt`) All strings on Σ = {A, B, C, D} that adhere to this Python regular expression: `^((AB)*(AC|BD)+|A?BC?D)$`

DFA Simulator

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

`python3 dfa.py` <*DFA file*> <*strings file*>

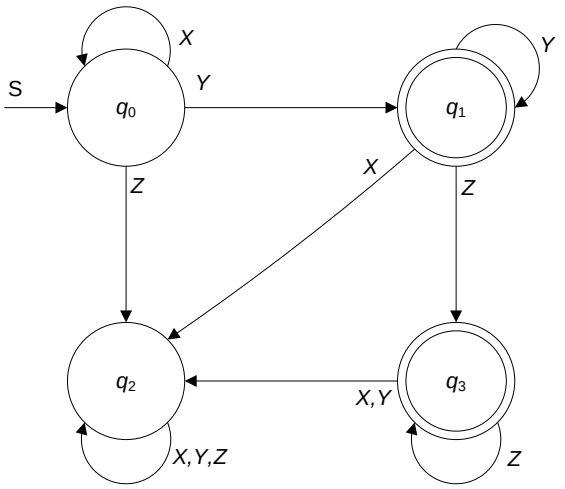
A sample DFA file is provided with this assignment, `sample-dfa.txt` and `sample-tests.txt` is a string file for testing. The *DFA file* describes the DFA using the following format:

- The first line will contain one or more characters that define the input alphabet.
    - Each character in the input alphabet is separated by a single space.
    - Each character in the input alphabet is a unique *single* letter or digit.
- All subsequent lines will refer to a different state in DFA. The second line in the file (the first line after the input alphabet definition line) will be state 0. The third line in the file will be state 1, the fourth line in the file will be state 2, and so forth.
    - The first character in the line will either be a '+' (accepting state) or '–' (rejecting state).
    - Then there will be *n* nonnegative integers where *n* is the number of characters in the input alphabet. Each integer refers to the destination state for a transition for the corresponding input character.
    - The corresponding input character is based on the order of the states and the order of the input characters on the first line of the file. If the input alphabet line is "D O G". The first integer the destination state for input D, the second is for input O, the third is for input G.
    - The integers are separated by a single space. There is also a single space between the first character (+ or –) and the first integer.
- State 0 is the starting state.

The *strings file* contains one or more strings that will be executed in the DFA. Each line will be interpreted as a test string. Each string will be processed by the DFA and the simulator will print out the final state along with whether the string is accepted or rejected. Additional notes:

- The lines in the string file must only contain characters from the DFA alphabet. If a letter outside the alphabet is found, a `keyError` exception will abort the script.
- You will need to create your own test scripts but there is no requirement to hand them in.

Example DFA:



Input file for DFA above (available as `sample-dfa.txt`):

```
X Y Z
- 0 1 2
+ 2 1 3
- 2 2 2
+ 2 2 3
```

Sample strings file for above DFA (available as `sample-tests.txt`):

```
X
Y
Z
XY
XYZ
XXXYYYZZZ
XYZYX
YZZZZ
XXX
YYY
ZZZ
```

Output from simulator:

```
String: X Final state: 0 REJECTED
String: Y Final state: 1 ACCEPTED
String: Z Final state: 2 REJECTED
String: XY Final state: 1 ACCEPTED
String: XYZ Final state: 3 ACCEPTED
String: XXXYYYZZZ Final state: 3 ACCEPTED
String: XYZYX Final state: 2 REJECTED
String: YZZZZ Final state: 3 ACCEPTED
String: XXX Final state: 0 REJECTED
String: YYY Final state: 1 ACCEPTED
String: ZZZ Final state: 2 REJECTED
```

Part 5 – Context-Free Grammars

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

1. (8 points, filename: `cfg-1.txt`) The set of all strings in the set { $x \in \{0, 1\}^*$ | the number of 0s in x is either exactly one more or one less than the number of 1s }. $T = \{0, 1\}$.

2. (12 points, filename: `cfg-2.txt`) The set of all strings consisting of an open bracket '[' followed by a list of positive integers (without leading zeroes) separated by commas, followed by a closing bracket ']'. T = { `[ ] , 0 1 2 3 4 5 6 7 8 9` }.

Notes:

- Spaces are not permitted anywhere in the string and are not part of *T*.
- You may not use a token to represent a positive integer. Note that the set of terminals *T* includes the digits 0-9.

Grammar File Format

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

- Nonterminals can either be single letters or longer strings (with no spaces). If you use longer strings, you must 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 the shorthand notation using `|`.
- The start symbol must either be `S` or `Start`.
- You may use `E`, `Empty`, or `''` to represent the empty string.

Example using single letter nonterminals with T = {`a`, `b`}:

```
S -> aS
S -> X
X -> bX
X -> ''
```

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

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

Unlike Parts 4, there is no Python simulator for testing your grammars. You may find the tool JFLAP (www.jflap.org) useful for testing. The tool requires you to enter your grammar manually and has stricter rules with respect to the grammar. The use of this tool is completely optional. Also, the instructor has very limited experience with JFLAP so support associated with JFLAP will be extremely limited.

Grading

The assignment will be graded in accordance with the programming assignment expectation handout. Grading will be based on correctness. Large deductions will be given if several tests fail. In addition:

- Assignments that do not adhere to the coding requirements described in this document (especially in Part 1) will lose points.
- A portion of your grade in Part 2 is developing a thorough test suite for the regular expressions.
- Parts 4 be graded using the simulator using a variety of test strings.
- Part 5 will be graded manually.

Submitting your Assignment

You need to submit the following files in one single zipped file through Canvas:

1. `hw4-1.py`
2. `hw4-2.py`
3. `hw4-3.py`
4. `dfa-1.txt`
5. `dfa-2.txt`
6. `dfa-3.txt`
7. `cfg-1.txt`
8. `cfg-2.txt`

Please be sure to keep the same file names or grading you assignment will fail. Only the last assignment submitted before the due date and time will be graded. ***Late submissions are not accepted and result in a zero.***