Programming Assignment

Automata Theory Monsoon 2023, IIIT Hyderabad

August 21, 2023

Total Points: 100 Points
Deadline: 4 September 2023

General Instructions: Read the input and output formats very carefully since the evaluation will be **scripted**. The goal is not to stress test your code but to see if you have understood the underlying concepts used in the implementation. There will be an evaluation/viva after the submission, and a part of the grade would depend on it.
Language constraints: The submission must be in Python. You will have to read/write from both a file and stdout. For any further use of other libraries please refer to one of the TAs for approval. Usage of any other language will lead to no evaluation.

---

1 Probabilistic Auto-complete

1. [50 points] **Writing your own auto-complete**
   Link to boilerplate

   You are given a file containing a large text corpus that contains valid English words. You have to use this file to generate a Probabilistic Automaton(PFSA) that will work as a letter-level auto-complete. The states of PFSA will represent a pre-fix of arbitrary length for an English word. You will also write a program to take such a PFSA and generate random words based on the distribution.

1.1 Rules on PFSA

1.1.1 States

Each state in your PFSA will represent a sequence of letters. The PFSA must not contain any dead states. Due to the probabilistic nature of PFSA, a dead state is a state whose incoming probability is zero. It is possible that your word is both a pre-fix and a valid Engligh word. For example, Auto. To distinguish the a pre-fix and a valid English word, you will mark the words that are in the text corpus with a * at the end.

1.1.2 Transitions

The transitions will be positively weighted and directed. Following the axioms of probability, the sum of outgoing probabilities from each state must add up to one. The incoming probabilities need not add up to one.

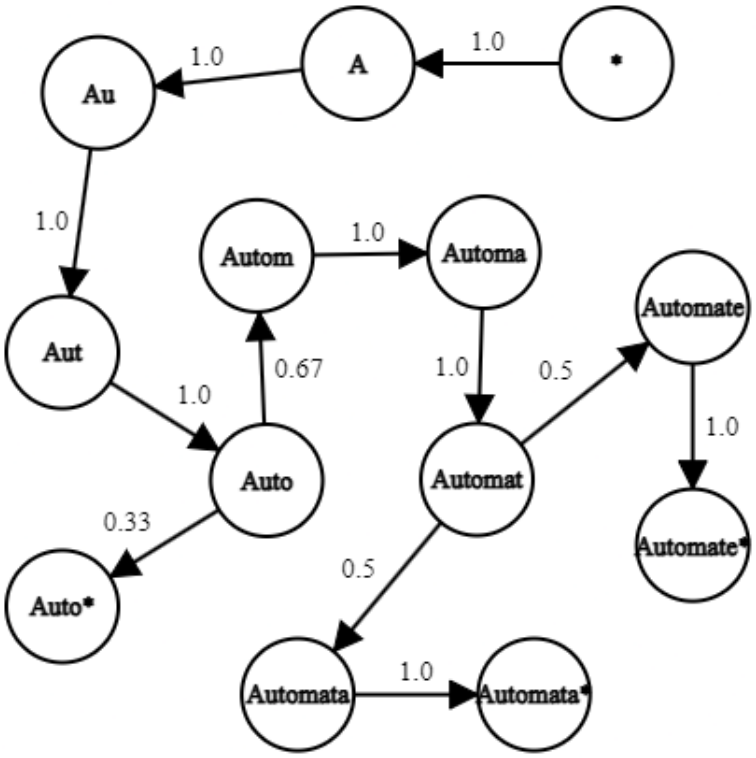

Figure 1: Example two-prefix PFSA

1.1.3 Initial State

The initial state will be represented using a *. It will have edges to nodes with only one letter, the weights will be based on the the text corpus.

1.1.4 End States

The difference between a state representing a valid English word from the corpus and a state representing a pre-text was established earlier. Naturally, any state that represents a valid English word will be an end state. In figure 1, there are three states that are marked with a *.

1.2 Example

Let us assume that the text corpus contains only three words, Auto Automata Automate. Your PFSA will contain all the prefixes of the two words with the transition probabilities. Refer figure 1.

1.3 I/O format

You will be provided with a boilerplate that will perform most of the I/O for you.

1.4 JSON format

Here is an example JSON format, this JSON does not represent a complete answer and must be referred only for understanding the format.

Listing 1: JSON example

```
{
  "*": {
    "A": "1.0"
  },
  "A": {
    "Au": "1.0"
  },
  "Auto": {
    "Auto*": "0.33",
    "Autom": "0.67"
  }
}
```

You will be provided with the necessary boilerplate code for converting a Python dictionary to a JSON and storing it in the file. You must ensure that your dictionary is in the correct format.

1.5 Sampling from PFSA

Once you complete the above tasks, you will have a program that will take in a valid text corpus and store the corresponding PFSA in a JSON file. Now you write a program that will take this JSON file and randomly sample words.

You will be provided with the boilerplate for reading and converting the JSON file for converting it to a Python dictionary. You will also be provided with the boilerplate to store a string as a text file.

1.5.1 Validation

The validity of your file will be checked by calculating the frequencies of each word, and ensuring that it is close to the expected distribution.

Submission details

A separate portal for Question 1 will be opened. You must submit a zip file with two Python programs. The program names must be pfsa.py for part I and generator.py for part II. The name of the zip file will be roll_number_AT1.zip.

1.6 Grading Scheme

a) **Correctness of the Program (30 points)**

b) **Viva Component (20 points)**

2 Compiler with tokenization and CFG parsing

1. [50 points] **Syntactic Analysis**
   Link to the boilerplate

   You are tasked with building a compiler for a simple programming language. The language supports the following token types: identifiers, keywords, integers, floating-point numbers, and symbols. **NOTE: You have been provided a boilerplate code which already parses the program. You are expected to implement a function that takes as parameters the list of tokens and performs syntactic analysis**

   Tokenization: Your compiler should perform tokenization (lexical analysis) of the source code using Finite State Automata (FSAs) for each token type. The compiler should be able to recognize and classify the tokens as identifiers, keywords, integers, floating-point numbers, and symbols present in the source code.

   Token Hierarchy: If the source code contains any keywords, identifiers, or numbers that match the same patterns, prioritize the matching based on token hierarchy. For example, if 'if' is a valid identifier, but it is also a keyword, it should be recognized as a keyword.

   Syntactic Analysis: After tokenizing, the compiler should also be able to perform some syntactical analysis according to the grammar provided in the question.

2.1 Rules for Syntactic Analysis

1. S $\rightarrow$ statement
2. statement $\rightarrow$ if (A)| (statement)(statement) | y
3. y $\in$ statement alphabets [$\Sigma_{statement}$]
4. A $\rightarrow$ (cond)(statement) | (cond)(statement)(else)(statement)
5. cond $\rightarrow$ (x)(op1)(x) | x
6. op1 $\rightarrow$ + | - | * | / | ^ | < | > | =
7. x $\rightarrow \mathbb{R}$ | cond

NOTE: $\Sigma_{statement}$ **is all valid tokens except 'if' and 'else'.**
The brackets in the above grammar are only for reference, you can implement the compiler without the brackets

2.2 Grading Scheme

a) **Correctness of the Program (30 points)**

b) **Viva Component (20 points)**

2.3 I/O format

When the program is run, it should ask for an input statement

2.3.1 Input

It is a single line of text

Output

2.3.2 No Error

If there are no errors (Syntactical or Lexical), then on each line, output
Token Type: <token type>, Token value <token value>

Token Type ∈ Identifier, Keyword, Integer, Float, Symbol
Token Value is the token itself

2.3.3 Lexical Error

Raise the error and give a brief description about the error
Eg: ValueError: Invalid identifier: 2xi
Identifier can't start with a number

2.3.4 Syntactic Error

When the compiler detects a code which does not follow the mentioned grammar, raise an error and give a brief description about it

2.4 Sample Test Cases

Input	Output

`if 2+xi > 0 print 2.0 else print -1;`	`Token Type: KEYWORD, Token Value: if`
	`Token Type: INTEGER, Token Value: 2`
	`Token Type: SYMBOL, Token Value: +`
	`Token Type: IDENTIFIER, Token Value: xi`
	`Token Type: SYMBOL, Token Value: >`
	`Token Type: INTEGER, Token Value: 0`
	`Token Type: KEYWORD, Token Value: print`
	`Token Type: FLOAT, Token Value: 2.0`
	`Token Type: KEYWORD, Token Value: else`
	`Token Type: KEYWORD, Token Value: print`
	`Token Type: SYMBOL, Token Value: -`
	`Token Type: INTEGER, Token Value: 1`
	`Token Type: SYMBOL, Token Value: ;`

Input	Output
`if 2xi > 0 print 2.0 else print -1;`	`ValueError: Invalid identifier: 2xi` `Identifier can't start with digits`

Input	Output
`else print 2.0 if 2>0 print 11;`	`SyntaxError: 'else' occurs before 'if'`

2.5 Submission Format

a) Submit the Python code file

b) Add a brief README explaining the program and assumptions made (if any)