Intro to formal languages. Regular grammars. Finite Automata

Author: Plămădeală Maxim

Group: FAF-222

Theory:

A formal language can be considered to be the media or the format used to convey information from a sender entity to the one that receives it. The usual components of a language are:

- The alphabet: Set of valid characters;
- The vocabulary: Set of valid words;
- The grammar: Set of rules/constraints over the lang.

Now these components can be established in an infinite amount of configurations, which actually means that whenever a language is being created, it's components should be selected in a way to make it as appropriate for it's use case as possible. Of course sometimes it is a matter of preference, that's why we ended up with lots of natural/programming/markup languages which might accomplish the same thing.

Objectives:

1. Discover what a language is and what it needs to have in order to be considered a formal one;
2. Provide the initial setup for the evolving project that you will work on during this semester. You can deal with each laboratory work as a separate task or project to demonstrate your understanding of the given themes, but you also can deal with labs as stages of making your own big solution, your own project. Do the following:

   a. Create GitHub repository to deal with storing and updating your project;

   b. Choose a programming language. Pick one that will be easiest for dealing with your tasks, you need to learn how to solve the problem itself, not everything around the problem (like setting up the project, launching it correctly and etc.);

   c. Store reports separately in a way to make verification of your work simpler (duh)
3. According to your variant number, get the grammar definition and do the following:

a. Implement a type/class for your grammar;

b. Add one function that would generate 5 valid strings from the language expressed by your given grammar;

c. Implement some functionality that would convert and object of type Grammar to one of type Finite Automaton;

d. For the Finite Automaton, please add a method that checks if an input string can be obtained via the state transition from it;

Implementation

Grammar Class

The `Grammar` class represents a context-free grammar (CFG), which is a set of recursive rewriting rules used to generate patterns of strings. A CFG consists of a set of non-terminal symbols, terminal symbols, a start symbol, and production rules.

```
class Grammar:
    def __init__(self):
        self.nonterminals = {'S', 'A', 'B', 'C'}
        self.terminals = {'a', 'b', 'c', 'd'}
        self.productions = {
            'S': ['dA'],
            'A': ['d', 'aB'],
            'B': ['bC'],
            'C': ['cA', 'aS']
        }
        self.start_symbol = 'S'
```

In the above code snippet, the `Grammar` class is initialized with non-terminals `S`, `A`, `B`, `C`, terminals `a`, `b`, `c`, `d`, and production rules that define how non-terminals can be transformed into strings of non-terminals and terminals.

FiniteAutomaton Class

The `FiniteAutomaton` class represents a finite automaton (FA), which is a simple machine used to recognize patterns within input taken from some character set (or alphabet).

```
class FiniteAutomaton:
    def __init__(self, states, alphabet, transitions, start_state, final_s
        self.states = states
        self.alphabet = alphabet
        self.transitions = transitions
        self.start_state = start_state
        self.final_states = final_states
```

The `FiniteAutomaton` class is initialized with a set of states, an alphabet, a transition function, a start state, and a set of final states. The transition function dictates how the automaton moves from one state to another based on input symbols.

Conversion from Grammar to Finite Automaton

The `to_finite_automaton` function in the provided code snippet is responsible for converting a given `Grammar` object into a `FiniteAutomaton` object. The conversion process involves the following steps:

1. **Creating States**: For each non-terminal in the grammar, a corresponding state is created in the finite automaton. Additionally, a special 'FINAL' state is created to represent the accepting state of the automaton.
2. **Defining the Alphabet**: The alphabet of the finite automaton is the same as the set of terminal symbols in the grammar.
3. **Building Transitions**: The transitions of the automaton are derived from the production rules of the grammar. For each production rule, a transition is created from the state corresponding to the non-terminal on the left-hand side to the state corresponding to the non-terminal on the right-hand side (if present), labeled with the terminal symbol that precedes it.
    - If a production rule is of the form `A -> a`, where `A` is a non-terminal and `a` is a terminal, a transition is created from the state corresponding to `A` to the 'FINAL' state, labeled with `a`.
    - If a production rule is of the form `A -> aB`, where `A` and `B` are non-terminals and `a` is a terminal, a transition is created from the state corresponding to `A` to the state corresponding to `B`, labeled with `a`.
4. **Setting Start and Final States**: The start state of the automaton is the state corresponding to the start symbol of the grammar. The set of final states contains only the `FINAL` state.

Example

Consider the following grammar production rules:

```
S -> dA
A -> d | aB
B -> bC
C -> cA | aS
```

The conversion process would create states for `S`, `A`, `B`, `C`, and `FINAL`. The transitions would be based on the production rules, such as a transition from state `S` to state `A` labeled with `d`, and a transition from state `A` to `FINAL` labeled with `d`.

Code Snippet

```
def to_finite_automaton(grammar):
    states = grammar.nonterminals.union({'FINAL'})
    alphabet = grammar.terminals
    transitions = {}
    final_states = {'FINAL'}

    for nonterminal, production_list in grammar.productions.items():
        for production in production_list:
            if len(production) == 1:  # A -> a
                terminal = production
                transitions[(nonterminal, terminal)] = {'FINAL'}
            elif len(production) == 2:  # A -> aB
                terminal, next_nonterminal = production
                if (nonterminal, terminal) not in transitions:
                    transitions[(nonterminal, terminal)] = set()
                transitions[(nonterminal, terminal)].add(next_nonterminal)

    start_state = grammar.start_symbol
    return FiniteAutomaton(states, alphabet, transitions, start_state, fina
```

In this code snippet, the `to_finite_automaton` function iterates over the production rules of the grammar and constructs the transition function for the finite automaton. It handles both types of production rules (ending with a terminal or a terminal followed by a non-terminal) and ensures that the transitions are correctly set up to reflect the grammar's structure.

String Generation and Validation

The `generate_strings` method in the `Grammar` class produces valid strings by recursively applying the grammar's production rules. Starting with the start symbol, the method expands it by randomly selecting one of the applicable productions until a string of terminal symbols is formed. This process ensures that each generated string adheres to the grammar's structure.

```
def generate_strings(self, count=5):
    strings = []
    while len(strings) < count:
        string = self._generate_from_symbol(self.start_symbol)
        if string not in strings:
            strings.append(string)
    return strings
```

```
def _generate_from_symbol(self, symbol):
    if symbol in self.terminals:
        return symbol
    else:
        production = self.productions[symbol]
        chosen_production = random.choice(production)
    return ''.join(self._generate_from_symbol(sym) for sym in chosen_produ
```

The `accepts` method in the `FiniteAutomaton` class determines if a string is recognized by the automaton, which corresponds to being a valid string in the language. It uses a recursive approach to simulate the automaton's state transitions based on the input string. If the automaton reaches a final state after consuming the entire string, the string is accepted; otherwise, it is rejected.

```
def accepts(self, string):
        def _accepts(state, remaining_string):
            if not remaining_string:
                return state in self.final_states
            current_char = remaining_string[0]
            if (state, current_char) in self.transitions:
                for next_state in self.transitions[(state, current_char)]:
                    if _accepts(next_state, remaining_string[1:]):
                        return True
            return False

        return _accepts(self.start_state, string)
```

Results and Conclusions

Generated strings:

```
dabcd
dabadd
dabadabadabadd
dabadabcabadabcd
dd
```

As seen, the system is generating the strings correctly.

Finite Automaton:

```
States: {'B', 'C', 'FINAL', 'A', 'S'}
Alphabet: {'d', 'b', 'c', 'a'}
Transitions: {
    'S' ('d': A),
    'A' ('d': FINAL),
    'A' ('a': B),
    'B' ('b': C),
    'C' ('c': A),
    'C' ('a': S),
}
Start State: S
Final States: {'FINAL'}
```

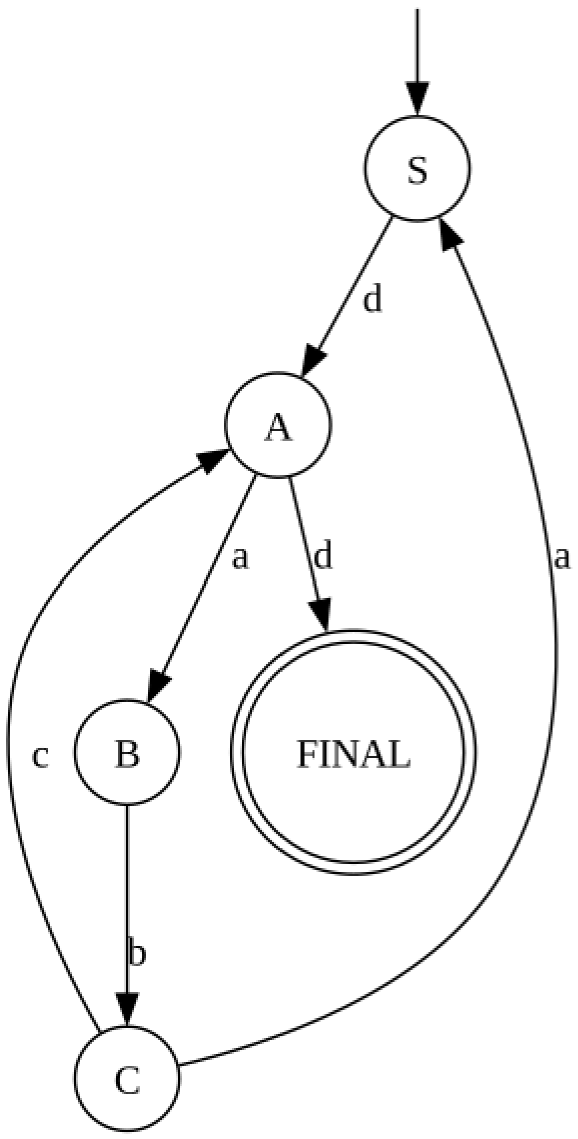
Based on the `Grammar` structure, I could generate a Finite Automaton with certain `States` and `Alphabet`. As seen, the `accepts` function is based on the `Transitions` objects, that "guides" on how the string should be structured (language rules).

String Validation:

```
dd (belongs to the language)
dabca (doesn't)
bbca (doesn't)
acac (doesn't)
bad (doesn't)
```

The generated strings are all valid strings that can be derived from the grammar. The finite automaton correctly accepts the string `dd` as it is a valid string in the language and rejects the other strings, which are not valid according to the grammar's production rules.

Graph view:



With the `graphviz` library, I made a script to generate the Finite Automata transitions.

In conclusion, the lab gave the basic knowledge on working with DSL and regular grammar.

References

1. COJUHARI Irina, Duca Ludmila, Fiodorv Ion. "Formal Languages and Finite Automata: Guide for practical lessons". Technical University of Moldova