Intro to formal languages. Regular grammars. Finite Automata.

Course: Formal Languages & Finite Automata

Author: Ungureanu Vlad

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 description

Part 1: Grammar Definition and Grammar Class

```
import random
```

```
# Define the grammar rules
VN = {"S", "A", "B"}  # Non-terminal symbols
VT = {"a", "b", "c"}  # Terminal symbols
P = {  # Production rules
    "S": ["aA", "bB"],
    "A": ["bS", "cA", "aB"],
    "B": ["aB", "b"],
}

# Define a class for the grammar
class Grammar:
    def __init__(self, vn, vt, p):
        self.vn = vn  # Set of non-terminal symbols
        self.vt = vt  # Set of terminal symbols
        self.p = p    # Dictionary of production rules

    def generate_string(self, symbol):
        """Generate a string based on the grammar rules starting from a given symbol."""
        if symbol in self.vt:  # If the symbol is terminal, return it
            return symbol
        else:
            options = self.p[symbol]  # Get production rules for the symbol
            chosen_option = random.choice(options)  # Choose a random production rule
            generated_string = ""
            for char in chosen_option:
                generated_string += self.generate_string(char)  # Recursively generate stri
            return generated_string

# Create an instance of the Grammar class
grammar = Grammar(VN, VT, P)

# Generate and print 5 valid strings
for _ in range(5):
    generated_string = grammar.generate_string("S")  # Start generating from the start symbo
    print(generated_string)
```

In this part, we define the context-free grammar with non-terminal symbols (`VN`), terminal symbols (`VT`), and production rules (`P`). We also define a class `Grammar` to work with the grammar, including a method `generate_string` to generate strings based on the grammar rules.

Part 2: Grammar to Finite Automaton Conversion

```
def grammar_to_finite_automaton(grammar):
    """Convert the grammar to a finite automaton (FA)."""
    states = grammar.vn  # States of the FA are the non-terminal symbols of the grammar
```

```python
    alphabet = grammar.vt  # Alphabet of the FA is the terminal symbols of the grammar
    start_state = "S"  # Start state of the FA is the start symbol "S" of the grammar
    accept_states = {
        state for state, rules in grammar.p.items() if any(r in rules for r in grammar.vt)
    }  # Accept states are states that have production rules with terminal symbols

    transitions = {}
    for state, rules in grammar.p.items():
        for rule in rules:
            if len(rule) == 2:  # Production X -> aY
                transitions[(state, rule[0])] = rule[1]  # Add transition rule to the FA
            elif len(rule) == 1:
                if rule[0] in grammar.vt:  # Simple terminal case
                    transitions[(state, rule[0])] = rule[0]  # Add transition rule to the F
                else:  # Special Case for new transitions like 'A'->'bS'
                    transitions[(state, rule[0])] = next(iter(grammar.p[rule[0]]))[0]  # Ad

    return FiniteAutomaton(states, alphabet, transitions, start_state, accept_states)

# Define the Finite Automaton class
class FiniteAutomaton:
    def __init__(self, states, alphabet, transitions, start_state, accept_states):
        self.states = states  # Set of states
        self.alphabet = alphabet  # Alphabet
        self.transitions = transitions  # Transition rules
        self.start_state = start_state  # Start state
        self.accept_states = accept_states  # Accept states

    def accepts(self, string):
        """Check if the FA accepts the given string."""
        current_state = self.start_state
        for char in string:
            if (current_state, char) not in self.transitions:
                return False  # No transition possible for this character
            current_state = self.transitions[(current_state, char)]  # Transition to the ne
        return current_state in self.accept_states  # Check if the final state is an accept
```

In this part, we define a function `grammar_to_finite_automaton` that converts the grammar into a finite automaton (FA). The FA has states, alphabet, transitions, start state, and accept states. We also define the `FiniteAutomaton` class to work with FAs, including a method `accepts` to check if a given string is accepted by the FA. The function works by creating a state for each non-terminal symbol in the grammar, and an additional state for the start symbol. The function then adds transitions between the states according to the production rules in the grammar.

For example, the production rule “S -> aA” would create a transition from the start state to the state “A” labeled with the symbol “a”.

The code then creates a FiniteAutomaton object from the grammar and uses it to test whether or not a few strings are accepted by the grammar.

Part 3: Testing Strings with the Finite Automaton

```
# Test strings
test_strings = ["aabbc", "ac", "abba", "acaaaabba", "aab"]
for string in test_strings:
    fa = grammar_to_finite_automaton(grammar)  # Convert the grammar to FA for each test st
    if fa.accepts(string):
        print(f"String '{string}' is accepted by the FA")
    else:
        print(f"String '{string}' is not accepted by the FA")
```

The code then creates a FiniteAutomaton object from the grammar and uses it to test whether or not a few strings are accepted by the grammar. In this part, we test several strings using the FA generated from the grammar. For each test string, we convert the grammar to an FA and check if the FA accepts the string, printing the result.

Conclusions and Results

This project was a great introduction to the world of formal languages! Here’s what we achieved:

- **Grammar Rules:** We clearly defined the “recipe” for creating valid sentences in our language. This involved symbols and production rules.
- **Building a Language Generator:** We wrote code that uses our grammar recipe to make random, correct sentences.
- **The Machine Translator:** We created a program that turns our grammar into a special machine (a finite automaton). This machine is like a language checker.
- **Testing Time:** We fed some test sentences to our machine to see if it recognized them as part of our language.

What’s Next?

- **Bigger Challenges:** We could try this with more complex languages.
- **Faster Machine:** Can we make our language-checking machine work even faster?
- **Cool Projects:** The things we learned here could help us build parts of compilers (for programming languages) or tools to understand everyday language better.

Overall, this project gave me a strong foundation in how to define languages and build tools to check them. This is a cool starting point for exploring other language-related projects!