Regular Grammars

Course: Formal Languages & Finite Automata

Author: Dimbitchi Sergiu

Theory

1. Context-Free Grammars :
I think of CFGs like a set of rules for building sentences. You have a starting point (like a sentence you want to create), and you follow certain rules to put words together. These rules tell you how to combine different parts (like nouns, verbs, etc.) to make a valid sentence. It's kind of like playing with building blocks, where each block represents a word or a group of words, and the rules show you how to stack them together.

2. Finite Automata:
Imagine you have a little machine that reads symbols (like letters or numbers) one at a time and moves between different states based on what it reads. It's like a mini-game where you move through different levels by pressing buttons according to the symbols you see. The machine has a starting point, some rules for moving between states, and maybe some special places you want to reach (like winning points).

3. Relationship between CFGs and Finite Automata:
These CFGs and finite automata are like two sides of the same coin. CFGs are good at describing how to build sentences or strings, while finite automata are good at recognizing patterns in those strings. It's like having a recipe book (CFG) that tells you how to make different dishes, and a taste tester (finite automaton) that checks if a dish matches a certain recipe. They work together to understand and create languages in a computer-friendly way.

So, in this project, I made some rules for creating arithmetic expressions and then used those rules to make some valid expressions. Converting these rules into a machine that can recognize valid expressions is a bit trickier and involves more advanced concepts, but it's all about understanding how languages are built and understood by computers.

Objectives:

- Implement a type/class for your grammar
- Add one function that would generate 5 valid strings from the language expressed by your given grammar
- Implement some functionality that would convert and object of type Grammar to one of type Finite Automaton
- 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

```
class Grammar:
    def __init__(self, variables, terminals, productions, start_variable):
        self.variables = set(variables)
        self.terminals = set(terminals)
        self.productions = productions
        self.start_variable = start_variable

    def generate_string(self):
        def generate_from(variable):
            production = random.choice(self.productions[variable])
            return ''.join(generate_from(sym) if sym in self.variables else sym for sym in production)

        return generate_from(self.start_variable)

    def generate_strings(self, n):
        return [self.generate_string() for _ in range(n)]

    def to_finite_automaton(self):

        states = self.variables.union({'F'})
        alphabet = self.terminals
        transition_function = {}
        accept_states = set()

        for variable, expansions in self.productions.items():
            for expansion in expansions:
                if expansion != 'e':
                    if len(expansion) == 2 and expansion[1] in self.variables:
                        transition_function[(variable, expansion[0])] = expansion[1]
                    else:
                        transition_function[(variable, expansion[0])] = 'F'
                        accept_states.add('F')
                else:
                    accept_states.add(variable)

        return FiniteAutomaton(states, alphabet, transition_function, self.start_variable, accept_states)
```

In this Grammar class we have the next methods:
1. “generate_string” – this method generates a single word based on the grammar rules

2. “to_finite_automaton” – this method works by first setting up the states for the Finite Automaton, including all the variables from the Grammar along with a special final state 'F' denoting the end of a string. Next, it defines the alphabet of the Finite Automaton using the terminals from the Grammar, which are the basic symbols that don't change. Then, it creates a transition function that determines how to move between states based on the symbols read, mapping each production rule in the Grammar to transitions in the Finite Automaton. Certain states are marked as accept states, indicating where the string generation process can stop and the generated string is considered valid. Finally, a Finite Automaton object is constructed using the defined states, alphabet, transition function, start state, and accept states, enabling it to recognize strings generated by the Grammar by following transitions and reaching accept states.

```
class FiniteAutomaton:
    def __init__(self, states, alphabet, transition_function, start_state, accept_states):
        self.states = states
        self.alphabet = alphabet
        self.transition_function = transition_function
        self.start_state = start_state
        self.accept_states = accept_states

    def check_string(self, input_string):
        current_state = self.start_state

        for symbol in input_string:
            if (current_state, symbol) in self.transition_function:
                current_state = self.transition_function[(current_state, symbol)]
            else:
                return False

        return current_state in self.accept_states
```

This FiniteAutomaton class is defined with an initialization method that takes in parameters such as states, alphabet, transition function, start state, and accept states, storing them as attributes. The “check_string” method is implemented to determine whether a given input string is accepted by the finite automaton. It iterates through each symbol in the input string, updating the current state based on the transition function. If the final state after processing the entire string is one of the accept states, the method returns True, indicating that the input string is accepted; otherwise, it returns False.

```python
# a. Creating the grammar object
variables = ['S', 'I', 'J', 'K']
terminals = ['a', 'b', 'c', 'e', 'n', 'f', 'm']
productions = {
    'S': ['cI'],
    'I': ['bJ', 'fI'],
    'J': ['nJ', 'cS', 'eK'],
    'K': ['nK', 'e', 'm']
}
start_variable = 'S'
grammar = Grammar(variables, terminals, productions, start_variable)

# b. Generating 5 valid strings from the grammar
valid_strings = grammar.generate_strings(5)

# c. Converting the grammar to a finite automaton
finite_automaton = grammar.to_finite_automaton()

# d. Checking if certain strings can be obtained from the finite automaton
print()
test_strings = input("Input the word to check ")

print ("5 VALID STRINGS :")
for i in valid_strings:
    print(i)
print()
print(f"The word {test_strings} is {finite_automaton.check_string(test_strings)}")
```

a. A Grammar object is created using specified variables, terminals, productions, and a start variable. These elements define the rules of a language.

b. The Grammar object generates 5 valid strings based on its rules.

c. The Grammar object is converted into a Finite Automaton. This involves setting up states, defining transitions, and identifying accept states.

d. The program prompts the user to input a word. It then checks if the given word can be obtained from the Finite Automaton. The result of this check is printed to the screen, indicating whether the word is accepted by the language described by the Finite Automaton.

Conclusions / Screenshots / Results

Output of the code :

```
5 VALID STRINGS :
cfbccfbccbee
cbnnenm
cbenm
cfffbccbccbnccbnnnennm
cbnem

The word cbnec is rejected
```

This laboratory work provided a practical exploration of fundamental concepts in formal language theory, focusing on context-free grammars and finite automata. By implementing Python classes to represent grammars and automata, I gained hands-on experience in translating abstract theoretical concepts into concrete computational structures.

I started by defining a CFG capable of generating strings according to specified rules. Using Python's object-oriented features, I designed a Grammar class to encapsulate variables, terminals, productions, and start variables. Through recursive techniques, I implemented methods to systematically generate valid strings from the grammar, demonstrating the power and versatility of CFGs in language generation.

Transitioning to finite automata, I converted CFGs into automata, symbolizing grammatical rules as state transitions. Within the “FiniteAutomaton” class, I defined states, alphabets, transition functions, start states, and accept states to represent an automaton capable of recognizing strings generated by the grammar. This conversion process illustrated the computational equivalence between CFGs and finite automata, a crucial concept in formal language theory.

In practical applications, I generated valid strings from the grammar and validated the conversion process by checking if arbitrary strings could be recognized by the resulting finite automaton. This hands-on experimentation reinforced my understanding of formal language theory principles and equipped me with practical tools for language generation and recognition.

Overall, this laboratory work provided valuable insights into the theoretical foundations of computation and their practical applications in software engineering. It underscored the importance of computational models in understanding and manipulating language constructs, with implications for compiler construction, natural language processing, and artificial intelligence. Through this hands-on exploration, I gained a deeper understanding of formal language theory concepts and their relevance to software engineering practice.

References

1. https://www.geeksforgeeks.org/introduction-of-finite-automata/

2. https://www.youtube.com/watch?v=58N2N7zJGrQ&list=PLBlnK6fEyqRgp46KUv4ZY69yXmpwKOIev

3. http://www.gcekjr.ac.in/pdf/lectures/2020/3174I_5th%20Semester_Computer%20Science%20And%20Engineering.pdf

4. Theory lessons from else