Formal Languages & Finite Automata

Laboratory work 1:

Intro to Formal Languages. Regular Grammars. Finite Automata

Student: Grigoraș Dumitru

Verified: prof., Cojuhari Irina

univ. assist., Crețu Dumitru

Chișinău, 2024

Theory

- *Grammar* - set of rules defining how to form a language. It is consisted of the (V_n,V_t,P,S) tuple.
  - *Non-terminals* (V_n) - uppercase letters, used to derive into other pieces of text.
  - *Terminals* (V_t) - lowercase letters which form the strings of the language.
  - *Productions* (P) - rules on how the non-terminals should be converted into terminals and other non-terminals.
  - *Starting Symbol* (S) - the name speaks for itself; it is the initial symbol from which the language is generated.
- *Finite Automata* - is like a robot that follows a path based on signals, evolving through states until it reaches an end or stops. It consists of a tuple, similar yet different from that of a grammar (Q,Σ,Δ,q_0,F).
  - Q - set of states, equivalent to V_n plus an additional state.
  - Σ - equivalent to V_t.
  - Δ - transition function.
  - q_0 - equivalent to S, the starting state.
  - F - set of final states, a subset of Q.

Objective

- Populate the class Grammar with the variant.
- Implement the generate_string() method in Grammar class.
- Construct the blueprint of Finite Automaton.
- Implement the toFiniteAutomaton() method in Grammar class.
- Implement the stringBelongToLanguage(String string) method in the FiniteAutomata class.
- Understand the basic concepts of Grammar & Finite Automata

Implementation

Grammar

I defined a class Grammar that represents a grammar with productions. It's capable of generating strings based on those productions and classifying the type of grammar it represents (e.g., Type 3, Type 2, etc.).

Here's a breakdown of the code:

Grammar Class:

This class represents a grammar and contains the following elements:
productions: A list of strings representing the production rules of the grammar.

- **Grammar**(): Constructor method that initializes the production rules of the grammar.
- **getProductions**(): A getter method to retrieve the list of production rules.
- **setProductions**(): A setter method to set the list of production rules.
- **generateString**(): A method to generate a string based on the grammar rules.
- **classifyGrammar**(): A method to classify the type of grammar (e.g., Type 3, Type 2, etc.).
- **generateStringHelper**(): A helper method used to generate strings recursively based on the grammar rules.
- **generateString**() Method: This method starts the string generation process by calling
- **generateStringHelper**() with the start symbol 'S' and returns the generated string.
- **classifyGrammar**() Method: This method determines the type of grammar based on certain conditions. It checks whether the grammar is Type 3 (Regular Grammar) or Type 2 (Context-Free Grammar) by analyzing the structure of its production rules..
- **generateStringHelper**() Method: This method recursively generates strings based on the grammar rules. It randomly selects a production rule for a given non-terminal symbol and recursively expands it until terminal symbols are reached.
- Helper Method to Generate String: The provided description introduces a slightly different method to generate strings. It suggests storing non-terminals and terminals separately and using a map to link non-terminals to their corresponding productions. The method recursively generates strings based on these rules.

Finite Automata:

Next I defined a class named FiniteAutomaton that represents a finite automaton. This automaton is constructed based on a given grammar, and it can accept or reject input strings based on its transitions and accepting states.

- **FiniteAutomaton** Class: This class represents a finite automaton and contains the following elements:
    1. states: A set containing all the states of the finite automaton.
    2. alphabet: A set containing the symbols in the input alphabet.
    3. transitions: A map representing the transitions of the finite automaton. Each state maps to another map, where each symbol in the alphabet maps to the state it transitions to.
    4. acceptingStates: A set containing the accepting states of the finite automaton.
    5. startState: A string representing the start state of the finite automaton.
- **Constructors:**
    1. **FiniteAutomaton**(…): Constructs a finite automaton with the provided sets of states, alphabet, transitions, accepting states, and start state.
    2. **FiniteAutomaton**(Grammar grammar): Constructs a finite automaton based on the given grammar. This constructor initializes the sets and maps and constructs the automaton using the grammar's production rules.
- **constructFiniteAutomaton**() Method: This method constructs the finite automaton based on the provided grammar. It initializes the states, alphabet, transitions, and accepting states according to the grammar's production rules.
- **acceptsInput**() Method: This method takes an input string and checks whether the finite automaton accepts it. It simulates the automaton's transitions based on the input symbols and returns true if it ends up in an accepting state after processing the entire input string.
- **Getters**: Getter methods are provided to access the states, alphabet, transitions, accepting states, and start state of the finite automaton.
- **Main Method**: The main method demonstrates the usage of the FiniteAutomaton class. It creates a grammar, constructs a finite automaton based on the grammar, and tests several input strings to see if they are accepted by the automaton.

Results

Variant 14:
VN={S, B, D},
VT={a, b, c, d},
P={
S → aS
S → bB
B → cB
B → d
B → aD
D → aB
D → b
}

1. **String generation:**
   Generated String 1: bab
   Generated String 2: bccaacd
   Generated String 3: abcd
   Generated String 4: aabcd
   Generated String 5: aabcd

2. **Convert Grammar to Finite automaton:**
   States: [B, S, D]
   Alphabet: [a, b, c, d]
   Transitions: {B={a=D, c=B, d=D}, S={a=S, b=B}, D={a=B, b=D}}
   Accepting States: [D]
   Start State: S

3. **Check if input string is valid:**
   String input1 = "aaabd";
   String input2 = "aabccaba";
   String input3 = "abcda";
   String input4 = "abd";
   Input1 accepted? true
   Input2 accepted? false
   Input3 accepted? false
   Input4 accepted? true

Conclusion

To conclude this project, I have successfully applied the theoretical knowledge gained from lectures into a practical coding framework. By constructing a language from the given Grammar and translating it into a Finite Automaton (FA), I have navigated the intricacies of class design and deepened my understanding of Formal Languages and Automata Theory.

This task not only involved developing classes and methods integral to grammars and automata but also emphasized the significance of regular grammars (type 3) during the conversion process to FA. This practical application has solidified my comprehension of these concepts and their relevance in computer science.

The project was a reflective and insightful synthesis of theory and practice, enhancing my problemsolving abilities and reinforcing the concept that complex theoretical constructs can be effectively transformed into working models. It was a testament to the power of applying theoretical foundations to solve practical problems and has contributed to both my academic and personal development.

Bibliography

[1] Formal Languages and Compiler Design Accessed February 14, 2024. https://else.fcim.utm.md/pluginfile.php/110457/mod_resource/content/0/Theme_1.pdf.

[2] Regular Language. Finite Automata. Accessed February 15, 2024. https://drive.google.com/file/d/1rBGyzDN5eWMXTNeUxLxmKsf7tyhHt9Jk/view