Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work nr. 1
Course: Formal languages and finite automata
Topic: Intro to formal languages. Regular grammars. Finite automata
Variant-20

Elaborated:
st. gr. FAF-223 Pereteatcu Arina

Verified:
asist. univ. Cretu Dumitru

Chişinău - 2024

Theoretical considerations

A formal language is defined as a collection of strings, or symbol sequences. The collection of symbols that make up the alphabet is what makes up the strings. This formal language is crucial to computer science because it allows algorithmic problems to be expressed and computer programs to be defined. Programmers can specify exactly what they want the machine to accomplish by using programming language syntax, which is defined based on formal languages. Formal language theory also offers methodical approaches to ascertain whether a given string conforms to a language's rules, which is essential for developing software such as interpreters and compilers.

A finite or infinite collection of strings over a finite number of symbols is the definition of a formal language in computer science. An 'alphabet' is a finite set of symbols. The formal language is made up of the ordered strings that are produced with this alphabet according to the established grammar rules.

Alphabet: In the scope of formal languages, an alphabet, often denoted by the Greek letter Σ, is simply a finite set of distinct symbols.

String: A string is a finite sequence of symbols selected from an alphabet. It is notable that the order of symbols matters in a string. An empty string, denoted often as lambda, is a string with zero symbols.

Grammar: Grammar is a set of formal rules that governs the combination of symbols to compose strings in a formal language. The structural nature of these string-producing rules is intrinsically tied into the classification of formal languages: regular, context-free, context-sensitive, and recursively enumerable.

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
3. Create GitHub repository to deal with storing and updating your project
4. Choose a programming language
5. Store reports separately in a way to make verification of your work simpler
6. Implement a type/class for your grammar

7. Add one function that would generate 5 valid strings from the language expressed by your given grammar
8. Implement functionality to convert an object of type Grammar to one of type Finite Automaton
9. Add a method to the Finite Automaton that checks if an input string can be obtained via state transitions from it

Implementation Description

My laboratory work was done in Python language. Variant 20:

VN={S, A, B, C},

VT={a, b, c, d},

P={

$S \rightarrow dA$

$A \rightarrow d$

$A \rightarrow aB$

$B \rightarrow bC$

$C \rightarrow cA$

$C \rightarrow aS$

```python
import random
class Grammar:
    def __init__(self):
        self.S = 'S'
        self.VN = ['S', 'A', 'B', 'C']
        self.VT = ['a', 'b', 'c', 'd']
        self.P = {
            'S': ['dA'],
            'A': ['d', 'aB'],
            'B': ['bC'],
            'C': ['cA', 'aS']
        }
        self.generated_strings = set()
```

First I declared a class for my Grammar. Initialization contains the start symbol S, non-temrinal symbols S,A,B,C and terminal symbols a,b,c,d , and a set of production rules, where each non-terminal symbol maps to a list of strings representing its possible expansions. Also, I have a set used to keep track of generated strings to ensure uniqueness.

```python
def generateString(self):
    def generateFromSymbol(symbol):
        if symbol in self.VT:
            return symbol
        else:
            production = random.choice(self.P[symbol])
            return ''.join(generateFromSymbol(s) for s in production)

    generated_word = generateFromSymbol(self.S)
    while generated_word in self.generated_strings:  # Ensure uniqueness
        generated_word = generateFromSymbol(self.S)
    self.generated_strings.add(generated_word)  # Add to the set of generated
strings
    return generated_word

def generate5Words(self):
    for _ in range(5):
        w = self.generateString()
        print("Generated string:", w)

def toFiniteAutomaton(self):
    return FiniteAutomaton()
```

I implemented mehods, such as generateString, which generates a random string based on grammar rules defined in P. It uses a nested function to recursively generate strings by expanding non-terminal symbols. The generated string is checked for uniqueness against the set of previously generated strings to ensure each generated string is unique. The method generate5Words generates and prints 5 random strings. It iterates 5 times and calls generateString.

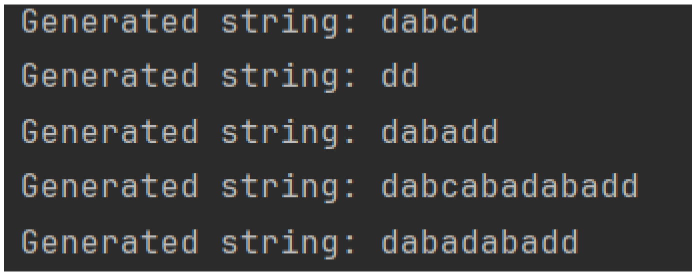
```
class FiniteAutomaton:
    def __init__(self):
        self.states = {'S', 'A', 'B', 'C'}  # Set of states
        self.alphabet = {'a', 'b', 'c', 'd'}  # Set of input symbols
        self.transitions = {
            ('S', 'd'): 'A',
            ('A', 'd'): 'A',
            ('A', 'a'): 'B',
            ('B', 'b'): 'C',
            ('C', 'c'): 'A',
            ('C', 'a'): 'S'
        }  # Dictionary representing transition function
        self.initial_state = 'S'  # Initial state
        self.accepting_states = {'S', 'A'}  # Set of accepting states
```

Here I implemented a class Finite Automaton. Self.states represent the set of states in the finite automaton S,A,B,C. Self.alphabet represents the set of input symbols or the alphabet of the finite automaton a,b,c,d. Self.transitions represents the transition function of the finite automaton. It is a dictionary where each key-value pair represents a transition from one state to another on consuming an input symbol.Self.initial_state represents the intial state of the finite automaton. The self.accepting_states represents the set of accepting states of the finite automaton.

```
def accepts_string(self, input_string):
    current_state = self.initial_state
    for symbol in input_string:
         if (current_state, symbol) in self.transitions:
            current_state = self.transitions[(current_state, symbol)]
         else:
             return False
    return current_state in self.accepting_states

grammar = Grammar()

grammar.generate5Words()


finiteAutomaton = grammar.toFiniteAutomaton()
```

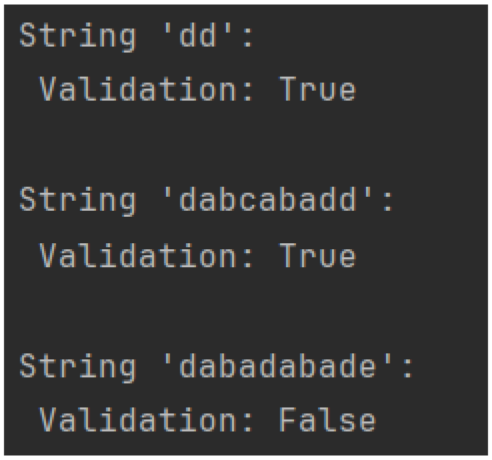
Method accept_string checks whether a given string is accepted by the finite automaton according to the rules. The variable current_state keeps track of the current state of the finite automaton during the processing of the input string. The method iterates through each symbol in the input_string, for each symbol it checks if there is a transition defined trom the current_state on consuming the symbol. If a transition is found it updates, the current_state to the state obtained from the transitions. If no transition is found for the current state and symbol, it means that the input string is not valid according to the finite automaton's rules, so it returns false.

Outputs



```
Generated string: dabcd
Generated string: dd
Generated string: dabadd
Generated string: dabcabadabadd
Generated string: dabadabadd
```

Figure1. Output 5 strings



```
String 'dd':
 Validation: True

String 'dabcabadd':
 Validation: True

String 'dabadabade':
 Validation: False
```

Figure2.Validate strings

Conclusions

In conclusion, this laboratory provided an exploration of formal languages and their significance in computer science. First, we studied the basic ideas behind formal languages, such as grammars, strings, and alphabets. A formal language, which is essential for addressing algorithmic issues and constructing computer programs, is defined as a set of strings produced by rules specified in a grammar. In order to generate valid strings from the stated language, we built a class for representing grammars.
Also, to bridge the gap between automata theory and formal language theory, we built capabilities to translate a grammar object into a finite automaton. We implemented a grammar from our own variant and converted it to finite automaton. We used terminal and non-terminals and production rules.