Ministerul Educaţiei, Culturii și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Departamentul Ingineria Software și Automatică

REPORT

Laboratory work 1
Course: Formal Languages & Finite Automata
Theme: Intro to formal languages. Regular grammars. Finite Automata

Elaborated: Latcovschi Cătălin, FAF-221

Verified: asist.univ. Dumitru Crețu

Chișinău 2024

Overview

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

Discover what a language is and what it needs to have in order to be considered a formal one;

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)

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

Implementation - Grammar Class:

```
class Grammar:
    def __init__(self, variables, terminals, productions, start_symbol):
        """Constructor method for the Grammar class."""
        self.variables = variables
        self.terminals = terminals
        self.productions = productions
        self.start_symbol = start_symbol

    def generate_string(self):
        """Generates a string by randomly expanding the start symbol based on the grammar's productions."""

        def expand(symbol):
            """Recursively expands a symbol based on the grammar's productions."""
            if symbol in self.terminals:
                return symbol
            if symbol not in self.productions:
```

```
            return ''
        prod = random.choice(self.productions[symbol])
        return ''.join(expand(sym) for sym in prod)

    return expand(self.start_symbol)
```

The Grammar class has a constructor (**__init__**) method that initializes the grammar components - variables, terminals, productions, and the start symbol. The generate_string method utilizes a recursive expand function to generate a string by randomly expanding the start symbol based on the grammar's productions. It checks whether a symbol is a terminal or non-terminal and selects a random production to generate the string.

Implementation - FiniteAutomaton Class:

```
class FiniteAutomaton:
    def __init__(self, states, alphabet, transition_function, start_state, accept_states):
        """Constructor method for the FiniteAutomaton class."""
        self.states = states
        self.alphabet = alphabet
        self.transition_function = transition_function
        self.start_state = start_state
        self.accept_states = accept_states

    def string_belongs_to_language(self, input_string):
        """Checks if the given string is accepted by the automaton."""
        current_state = self.start_state
```

```
        for symbol in input_string:
            if symbol not in self.alphabet:
                return False  # Reject strings with symbols not in the automaton's alphabet
            if symbol in self.transition_function.get(current_state, {}):
                current_state = self.transition_function[current_state][symbol]
            else:
                return False
        return current_state in self.accept_states
```

The FiniteAutomaton class has a constructor (**__init__**) method that initializes the components of a deterministic finite automaton (DFA) - states, alphabet, transition function, start state, and accept states. The class also has a string_belongs_to_language method that checks if a given string is accepted by the automaton. It iterates through the input string, following transitions based on the transition function, and returns True if the final state is in the set of accept states.

Test cases:

```
# Initialize grammar and finite automaton with given components
variables = ['S', 'A', 'B', 'C']
terminals = ['a', 'b', 'c', 'd']
productions = {
    'S': ['dA'],
    'A': ['aB', 'bA'],
    'B': ['bC', 'aB', 'd'],
    'C': ['cB']
}
start_symbol = 'S'
my_grammar = Grammar(variables, terminals, productions, start_symbol)

# Generate and print valid strings from the grammar
```

```
print("A list of valid strings: ")

for _ in range(10):

   print(my_grammar.generate_string())



# Define the finite automaton's components

states = {'S', 'A', 'B', 'C'}

alphabet = {'a', 'b', 'c', 'd'}

transition_function = {

   'S': {'d': 'A'},

   'A': {'a': 'B', 'b': 'A'},

   'B': {'b': 'C', 'a': 'B', 'd': 'D'},

   'C': {'c': 'B'}

}

start_state = 'S'

accept_states = {'D'}



# Initialize finite automaton

fa = FiniteAutomaton(states, alphabet, transition_function, start_state,
accept_states)



# Test and print whether each string is accepted by the automaton

test_strings = [

   'dad',

   'dbc',

   'dac',

   'dab',

   'daad',

   'dbb',

   'daabb',

   'daab',
```

```
    'dd',
    'aaa',
    'dcc',
    'aaaa',
    'bbbb',
    'ddd',
    'dadc',
    'daabc',
    'dabcd',
    'daacb',
    'dacd',
    'dbca'
]


# Test and print whether each string is accepted by the automaton
for string in test_strings:
    if fa.string_belongs_to_language(string):
        print(f"The string '{string}' is accepted by the automaton.")
    else:
        print(f"The string '{string}' is not accepted by the automaton.")
```

Results

```
A list of valid strings:
dad
dad
dad
daad
dbad
dbabcbcaad
dabcad
daad
dbbaad
daad
```

```
The string 'dad' is accepted by the automaton.
The string 'dbc' is not accepted by the automaton.
The string 'dac' is not accepted by the automaton.
The string 'dab' is not accepted by the automaton.
The string 'daad' is accepted by the automaton.
The string 'dbb' is not accepted by the automaton.
The string 'daabb' is not accepted by the automaton.
The string 'daab' is not accepted by the automaton.
The string 'dd' is not accepted by the automaton.
The string 'aaa' is not accepted by the automaton.
The string 'dcc' is not accepted by the automaton.
The string 'aaaa' is not accepted by the automaton.
The string 'bbbb' is not accepted by the automaton.
The string 'ddd' is not accepted by the automaton.
The string 'dadc' is not accepted by the automaton.
The string 'daabc' is not accepted by the automaton.
The string 'dabcd' is accepted by the automaton.
The string 'daacb' is not accepted by the automaton.
The string 'dacd' is not accepted by the automaton.
The string 'dbca' is not accepted by the automaton.
```

Conclusions

In this laboratory work, I explored the fundamentals of formal languages, grammars, and finite automata, essential concepts in the study of computer science and language processing. The key takeaways include:

Understanding Formal Languages: I delved into the structure of formal languages, learning about alphabets, vocabularies, and grammars, which are crucial for defining and understanding both programming and natural languages.

Practical Implementation: By developing a Grammar class and a FiniteAutomaton class, I applied theoretical concepts practically, gaining insights into how languages are processed and recognized by computational systems.

Insight into Computational Theory: The lab provided a clear example of how theoretical models like grammars and automata are implemented in code, offering a solid foundation for understanding more complex computational theories.

Skill Development: The hands-on approach enhanced our problem-solving and debugging skills, essential for software development and computational problem-solving.

Overall, this lab was a valuable exercise in bridging the gap between theoretical computer science concepts and their practical applications, setting a strong foundation for future studies and projects in the field of formal languages and automata theory.