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

Formal Languages and Finite Automata

Laboratory work 1:
Regular Grammar

Elaborated:
st. gr. FAF-221 Reabciuc Daria-Brianna

Verified:
asist. univ. Cretu Dumitru
prof. univ. Cojuhari Irina

Chişinău - 2024

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

```
class Grammar:
    def __init__(self):
        self.VN = {'S', 'A', 'B', 'C'}
        self.VT = {'a', 'b', 'c', 'd'}
        self.P = {
            'S': ['dA'],
            'A': ['aB', 'b'],
            'B': ['bC', 'd'],
            'C': ['cB', 'aA']
        }
```

This code defines a Python class named **Grammar** that represents a specific type of grammar in computer science, often used in language processing or compiler design.

- **Non-terminal symbols (VN):** These are placeholders that can be replaced with sequences of other symbols (both terminal and non-terminal) according to the production rules. In this grammar, 'S', 'A', 'B', and 'C' are non-terminal symbols.
- **Terminal symbols (VT):** These are the actual characters or symbols that appear in the strings generated by the grammar. Here, 'a', 'b', 'c', and 'd' are terminal symbols.
- **Production rules (P):** These rules define how non-terminal symbols can be transformed into other symbols (either terminal or non-terminal). For example, 'S' can be replaced with 'dA'.

```
def generate_valid_strings(self, symbol='S', depth=0):
    if depth > 10 or symbol not in self.VN:  # Limit depth to prevent infinite recursion
        return [symbol]
    results = []
    for production in self.P[symbol]:
        partial_results = ['']
        for char in production:
            new_partial_results = []
            for pr in partial_results:
                for expansion in self.generate_valid_strings(char, depth + 1):
                    new_partial_results.append(pr + expansion)
            partial_results = new_partial_results
        results.extend(partial_results)
    return results
```

This method attempts to generate all valid strings starting from a given symbol (by default, 'S') up to a certain depth to avoid infinite recursion. The depth is limited to 10 levels. It works by recursively expanding each non-terminal symbol according to the production rules until it reaches terminal symbols or the depth limit. The method accumulates and returns all possible.

```
def generate_five_valid_strings(self):
    valid_strings = self.generate_valid_strings()
    return valid_strings[:5]
```

This method uses **generate_valid_strings** to create a list of valid strings starting from the default symbol 'S'. It then returns the first five strings from this list, providing a sample of the possible strings generated by the grammar.

```python
class FiniteAutomaton:
    def __init__(self):
        self.states = set()
        self.alphabet = set()
        self.transition_function = {}
        self.start_state = None
        self.accept_states = set()

    def add_state(self, state):
        self.states.add(state)

    def add_transition(self, from_state, input_char, to_state):
        if from_state not in self.transition_function:
            self.transition_function[from_state] = {}
        self.transition_function[from_state][input_char] = to_state

    def set_start_state(self, state):
        self.start_state = state

    def add_accept_state(self, state):
        self.accept_states.add(state)

    def check_string(self, input_string):
        current_state = self.start_state
        for char in input_string:
            if char in self.transition_function.get(current_state, {}):
                current_state = self.transition_function[current_state][char]
            else:
                return False
        return current_state in self.accept_states
```

Imagine you're playing a video game where you need to go through various levels (states) by pressing the right buttons (characters). The FiniteAutomaton class is like the game itself, where:

- Each level is a state.
- The buttons you press are characters from the alphabet.
- The rules on when to press which button to get to the next level are the transition functions.
- The starting level is the start state.
- The levels where you win the game are the accept states.

```python
def grammar_to_automaton(grammar):
    fa = FiniteAutomaton()
    fa.set_start_state('start')
    fa.add_accept_state('accept')

    for symbol in grammar.VN:
        fa.add_state(symbol)
        if symbol == 'S':  # Assuming 'S' is always the start symbol
            fa.set_start_state(symbol)

    for left, productions in grammar.P.items():
        for production in productions:
            if production.islower():  # Assuming lowercase are terminal symbols
                fa.add_transition(left, production, 'accept')
            else:
                for char in production:
                    if char.isupper():  # Assuming uppercase are non-terminal symbols
                        fa.add_transition(left, char, char)
                    else:
                        fa.add_state(char + '_mid')
                        fa.add_transition(left, char, char + '_mid')
                        fa.add_transition(char + '_mid', '', 'accept')

    return fa
```

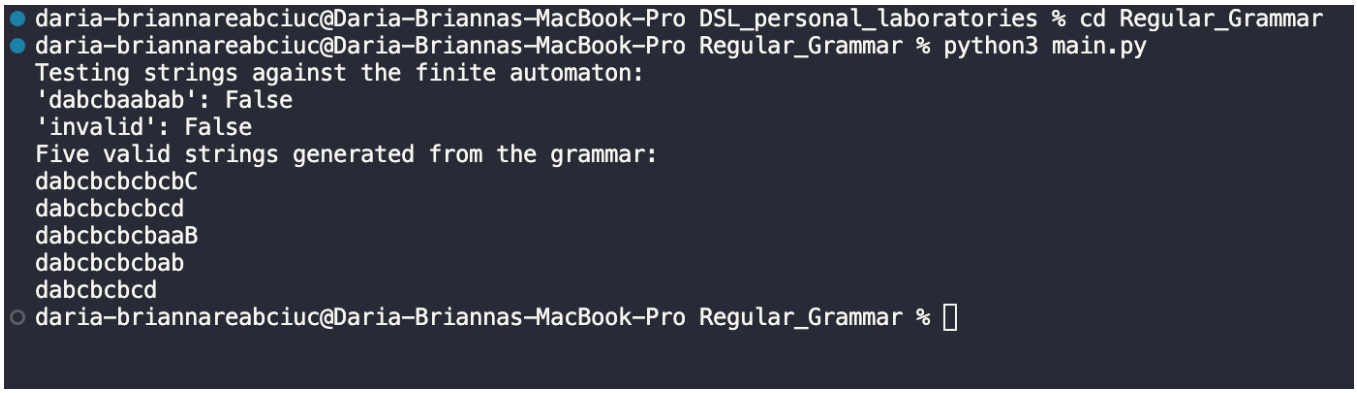
The grammar_to_automaton function is like designing the game based on a story (grammar). It sets up the levels (states), tells you the starting point, and defines which levels make you win. It also carefully designs each level so that you need to follow the story's plot (grammar rules) to win the game.

```python
grammar = Grammar()

# Check if a string can be obtained via the state transition from the automaton
fa = grammar_to_automaton(grammar)

# Test the FA with a valid string and an invalid string
test_strings = ['dabcbaabab', 'invalid']
results = {ts: fa.check_string(ts) for ts in test_strings}
print("Testing strings against the finite automaton:")
for ts in test_strings:
    result = fa.check_string(ts)
    print(f"'{ts}': {result}")

five_valid_strings = grammar.generate_five_valid_strings()

# Print the 5 valid strings
print("Five valid strings generated from the grammar:")
for string in five_valid_strings:
    print(string)
```

First, we create our rulebook (**grammar**). Then, we use a magic spell (**grammar_to_automaton**) that turns our rulebook into a smart checker (**fa**) that can automatically tell us if words follow our rules. We use our smart checker (**fa**) to test these words. It looks at each letter in the words and checks if the way those letters are put together matches any of the rules in our rulebook. Next, we use another magic trick from our rulebook (**grammar.generate_five_valid_strings()**) to come up with five words that definitely follow the rules. It's like asking our rulebook to give us examples of secret codes that are guaranteed to be right.

Results

These are the results that we get, as we can see if we test the words which does not match our grammar we get a false affirmation, also we have generated five valid strings that matches the rules in our rulebook.



CONCLUSION

We understood the theoretical computer science by dissecting code that models both grammars and finite automatons. Grammars, as we've seen, are sets of rules that define how strings (or words) can be constructed from given symbols. These grammars are crucial in understanding the syntax of languages, whether programming or natural languages. The **Grammar** class we explored provided a structured way to represent these rules and generate valid strings that adhere to the specified grammar, showcasing the foundational principles of language processing and compilation in computer science.

Transitioning to finite automatons, state machines that can recognize patterns or validate strings against defined criteria. The **FiniteAutomaton** class illustrated how such machines are constructed, including states, transitions, and acceptance conditions, offering a practical glimpse into automata theory. This theory underpins much of computational linguistics, compiler design, and even some algorithms in computer science.

The transformation of grammar into a finite automaton through the **grammar_to_automaton** function bridged the conceptual gap between static rules and dynamic state processing. This transformation demonstrates the underlying unity of concepts in theoretical computer science, where seemingly abstract rules find concrete application in machines capable of recognizing complex patterns.

Finally, by testing the finite automaton with both valid and invalid strings and generating valid strings from the grammar, we practically applied these theoretical concepts. This application not only

solidified our understanding but also highlighted the practical importance of these concepts in designing systems that process or recognize languages.

In conclusion, our journey from understanding grammar and finite automata to applying these concepts in code exemplifies the beauty and power of computer science. It combines abstract theoretical foundations with practical applications, revealing the intricate ways in which we can model, process, and understand the complexities of language and patterns.