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

Laboratory work 1:
Intro to formal languages. Regular grammars. Finite Automata.

Elaborated: st. gr. FAF-223 Friptu Ludmila

Verified: asist. univ. Cretu Dumitru

THEORY

1. Formal Languages

Formal languages are a foundational concept in computer science and linguistics, referring to sets of strings of symbols that are constrained by specific rules or grammars. These symbols can be anything from characters and numbers to more abstract elements, and the rules define how these symbols can be combined to form valid strings within the language.

The study of formal languages involves understanding how these languages can be classified, manipulated, and applied in various domains. Formal languages are categorized into different types based on the complexity of their rules, ranging from regular and context-free to context-sensitive and recursively enumerable languages. Each category has its own set of properties and is associated with specific types of computational models that can process them, such as finite automata, pushdown automata, and Turing machines.

One of the primary applications of formal languages is in the design and analysis of programming languages and compilers. They are also crucial in the development of various algorithms for parsing and understanding natural languages, thus bridging the gap between human communication and machine processing.

2. Regular grammars

Regular grammars are a specific type of formal grammar that generates regular languages. These grammars are defined by a set of rules or productions that describe how strings in the language can be formed from the alphabet of the language. Regular grammars are pivotal in the field of formal languages and automata theory because they provide the foundation for understanding more complex language classes and for the practical implementation of lexical analyzers, which are components of compilers.

A regular grammar consists of:

1. A finite set of non-terminal symbols, which are placeholders for patterns of symbols that can be replaced according to the grammar's rules.
2. A finite set of terminal symbols, which form the actual content of the language and appear in the strings generated by the grammar. Terminal symbols are the alphabet of the language.
3. A finite set of production rules, each of which specifies how a non-terminal symbol can be replaced by a combination of terminals and possibly one non-terminal. The productions in regular grammars are restricted to two types:

- Right-linear rules, where the non-terminal, if present, appears at the right end of the production (e.g., A → aB or A → a).

- Left-linear rules, where the non-terminal, if present, appears at the left end of the production (e.g., A → Ba or A → a).

4. A designated start symbol, which is a non-terminal from which generation of strings in the language begins.

The significance of regular grammars lies in their correspondence to finite automata. Specifically, for every regular grammar, there exists a finite automaton (deterministic or nondeterministic) that recognizes the same language, and vice versa. This equivalence establishes the basis for understanding and designing lexical analyzers and parsers, which are crucial in the processing of programming languages.

3. Finite Automata

Finite Automata are a fundamental concept in the theory of computation and formal languages, serving as a mathematical model for machines with a finite number of states. Finite automata are used to represent and analyze the operation of systems that have a limited and discrete number of possible configurations. These models are particularly powerful for designing and understanding computer algorithms, programming languages, and various types of software and hardware systems.

A finite automaton consists of the following components:

1. **States:** A finite set of conditions or configurations, one of which the automaton is always in. There is a designated start state and one or more accept (or final) states.

2. **Alphabet:** A finite set of symbols that the automaton can read as input. These symbols drive transitions between states.

3. **Transitions:** A set of rules that describe how the automaton moves from one state to another based on the input symbol it reads. Each transition is a mapping from a state and an input symbol to a new state.

4. **Start State:** The state in which the automaton begins operation.

5. **Accept States:** A subset of states that are designated as accept or final states. If the automaton ends in one of these states after processing an input string, the string is considered to be accepted by the automaton.

Finite automata are classified into two main types:

- **Deterministic Finite Automata**: In a DFA, for every state and input symbol, there is exactly one transition to the next state. This means that the automaton's behavior is completely determined by its current state and the input symbol it reads.

- **Nondeterministic Finite Automata**: An NFA allows for multiple possible transitions for a given state and input symbol, including transitions without consuming any input symbols (ε-transitions). For every NFA, there exists an equivalent DFA that recognizes the same language.

Finite automata are used to recognize regular languages, which are the simplest class of languages in the Chomsky hierarchy. The significance of finite automata lies in their ability to model simple computational processes and patterns of behavior. They are used in various applications, including the design of digital circuits, the development of compilers and lexical analyzers, network protocols, and the analysis of DNA sequences.

OBJECTIVES

Variant 11

VN={S, B, D} VT={a, b, c}

P={ S → aB S → bB B → bD D → b D → aD B → cB B → aS }

1. Discover what a language is and what it needs to have in order to be considered a formal one;
2. Implement a type/class for your grammar;
3. Add one function that would generate 5 valid strings from the language expressed by the given grammar;
4. Implement some functionality that would convert and object of type Grammar to one of type Finite Automaton;
5. For the Finite Automaton, add a method that checks if an input string can be obtained via the state transition from it;

IMPLEMENTATION

The first thing I created was the ***Grammar*** Class, which represents a context-free grammar. This class encapsulates the structure and rules of a CFG, allowing for the generation of strings that are part of the language defined by the grammar.When an instance of the ***Grammar*** class is created, it initializes with predefined sets of non-terminal symbols (VN), terminal symbols (VT), and production rules (P). The start symbol (S) is also defined.

- **VN**: In my grammar, `S`, `B`, and `D` are non-terminal symbols.
- **VT**: In my case, `a`, `b`, and `c` are terminal symbols.
- **Production rules**: My grammar includes rules like `S → aB`, allowing `S` to be replaced with `aB`, and so on.
- **Start symbol** : The symbol from which string generation begins according to the grammar's production rules.

```
class Grammar:
    def __init__(self):
        self.VN = {'S', 'B', 'D'}
        self.VT = {'a', 'b', 'c'}
        self.P = {
            'S': ['aB', 'bB'],
            'B': ['bD', 'cB', 'aS'],
            'D': ['b', 'aD']
        }
        self.S = 'S'
```

Figure 1 Grammar Class

Next, in my class I implemented the *generate_strin*g method which is meant to generate a string that is valid within the language defined by the context-free grammar. This method encapsulates a recursive approach to expand non-terminal symbols based on the production rules defined in the grammar until a string consisting only of terminal symbols is produced. The *generate_string* function itself doesn't take any parameters. It serves as a wrapper around the inner function, *expand*, and initiates the recursive expansion process by calling *expand* with the start symbol. On the other hand, *expand* is a recursive function. It takes a single argument, *symbol*, which can be either a terminal or a non-terminal symbol. If *symbol* is in **VT**, *expand* returns the symbol itself. If *symbol* is in **VN**, the method selects a random production rule for this symbol from **P**. For the selected production rule, expand is called recursively for each symbol in the production. This process continues until all symbols are terminal.

```
def generate_string(self):
    def expand(symbol):
        if symbol in self.VT:
            return symbol
        elif symbol in self.VN:
            production = random.choice(self.P[symbol])
            return ''.join(expand(s) for s in production)
        return ''

    return expand(self.S)
```

Figure 2 Generate String Method

Next, in my class I implemented the to_finite_automaton method that converts the CFG into a finite automaton. This involves translating the production rules of the CFG into state transitions of the FA. The method iterates over each non-terminal symbol in the grammar and its associated productions. These production rules define how non-terminal symbols can be expanded or transformed. If a production rule consists of exactly two symbols, it's assumed to be of the form A -> aB, where A is the current non-terminal being processed, a is a terminal symbol, and B is another non-terminal symbol. In this case: input_char is set to the first symbol of the production, next_state is set to the second symbol of the production .

A transition is added to the finite automaton's transitions dictionary, mapping from the current state (non_terminal, corresponding to A) and input character (input_char, corresponding to a) to a set containing the next_state (B). This transition represents the automaton moving from state A to state B upon reading the symbol a.

If a production rule consists of a single symbol (terminal or non-terminal), two cases are considered:

- If the single symbol is a terminal (input_char in self.VT), a transition is added that moves from the current non-terminal state to an 'end' state, indicating that reading this terminal symbol leads to a potential accept state in the context of this automaton.
- If the single symbol is a non-terminal, the logic here intends for a transition that doesn't consume any input (an epsilon transition).

```python
def to_finite_automaton(self):
    fa = FiniteAutomaton()
    for non_terminal, productions in self.P.items():
        for production in productions:
            if len(production) == 2:  # Assuming productions like A -> aB
                input_char = production[0]
                next_state = production[1]
                fa.transitions[(non_terminal, input_char)] = {next_state}
            elif len(production) == 1:  # Assuming productions like A -> a or A -> B
                input_char = production
                # Assuming 'end' state for terminal transitions
                if input_char in self.VT:
                    fa.transitions[(non_terminal, input_char)] = {'end'}
                else:
                    fa.transitions[(non_terminal, '')] = {input_char}
    return fa
```

Figure 3 Finite Automaton method

Next, I implemented the ***FiniteAutomaton*** class. This class defines an FA with states, an alphabet, transitions between states based on input symbols, a start state, and accept states. The FiniteAutomaton class defines a specific FA with predefined behavior based on its transitions. The automaton starts in the *start_state* (S). As the automaton reads symbols from an input string, it follows the transitions defined in transitions to move between states. Each transition is determined by the current state and the input symbol read at each step. After reading the entire input string, if the automaton is in an *accept_states* (end), the input is accepted, otherwise, it is rejected.

```python
class FiniteAutomaton:
    def __init__(self):
        self.states = {'S', 'B', 'D', 'end'}
        self.alphabet = {'a', 'b', 'c'}
        self.transitions = {
            ('S', 'a'): {'B'},
            ('S', 'b'): {'B'},
            ('B', 'b'): {'D'},
            ('B', 'c'): {'B'},
            ('B', 'a'): {'S'},
            ('D', 'b'): {'end'},
            ('D', 'a'): {'D'}
        }
        self.start_state = 'S'
        self.accept_states = {'end'}
```

Figure 4 Finite Automaton Class

Next, I implemented the *string_belongs_to_language* method in the **FiniteAutomaton** class, which I created to determine whether a given input string is accepted by the finite automaton. The method starts by initializing *current_states* with the start state of the automaton. The method then iterates over each character in the *input_string*. For each character, the automaton will transition to new states based on its current state and the transitions defined for the current input symbol. For every character in the input string, the method initializes an empty set *next_states*. This set will collect all the states the automaton can transition into based on its current state and the current input symbol. For each state in *current_states*, it checks if there is a transition defined for the current state and character pair (state, char). If such a transition exists, it updates *next_states* with the states that can be reached from the current state on reading the character. This is done using *next_states.update* (self.transitions[(state, char)]), which adds all the next possible states to *next_states*. After processing the current character for all current states, *current_states* is updated to be *next_states*, preparing it for the next iteration. After the loop finishes, the method checks if any of the current states is an accept state. This is done by checking if there is an intersection between *current_states* and *accept_states*. If the intersection is not empty (len(current_states.intersection(self.accept_states)) > 0), it means the automaton ended in at least one accept state, and the input string is accepted, otherwise, the string is rejected.

```
def string_belongs_to_language(self, input_string):
    current_states = {self.start_state}
    for char in input_string:
        next_states = set()
        for state in current_states:
            if (state, char) in self.transitions:
                next_states.update(self.transitions[(state, char)])
        current_states = next_states
    return len(current_states.intersection(self.accept_states)) > 0
```

Figure 5 string_belongs_to_language method

Results:

```
Generated strings from the grammar:
bcbaab
bcbb
bababaabaab
bbab
baabab
```

```
Generated strings from the grammar:
accbb
babbb
acccbaaaab
abab
acbab
```

Figures 6, 7 Results for the possible generated strings

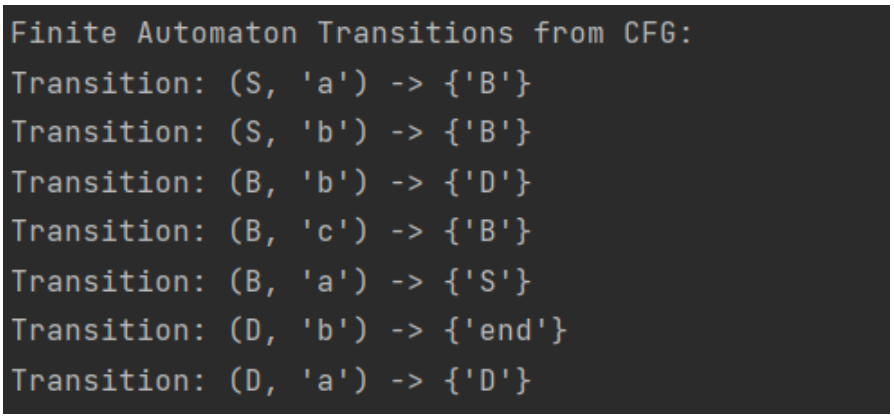

Figure 8 Finite Automaton Transitions

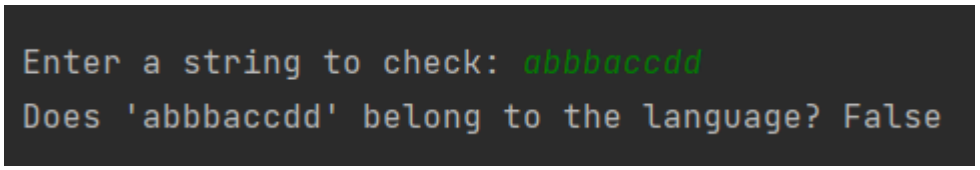

Figure 9 Checking a string

Conclusion

In concluding this laboratory, I have explored and implemented foundational concepts of formal languages and automata theory, focusing on the construction and manipulation of grammars and finite automata. Through the development of the **Grammar** and **FiniteAutomaton** classes, I have gained a practical understanding of how grammatical rules and automaton transitions are applied to generate and recognize strings within specific languages.

The **Grammar** class allowed me to define a CFG with a set of VN and VT symbols, along with production rules. By implementing a method to generate strings from this grammar, I experienced firsthand the process of recursive symbol expansion, which is central to how CFGs produce valid strings. Transitioning to the **FiniteAutomaton** class, I explored the representation of languages through state transitions, starting states, and accepting states.

An important part of this laboratory was attempting to conceptually convert a CFG to an FA. This exercise served as a valuable lesson in the hierarchy of formal languages and the Chomsky hierarchy.

Overall, this laboratory has significantly enhanced my understanding of the theoretical foundation and practical applications of formal languages and automata theory. It has underscored the importance of these concepts in computer science, from compiler construction to the analysis of algorithms and beyond.