

MINISTERUL EDUCAȚIEI 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.

Course: Formal Languages and Finite Automata

Author: Cucos Maria

Chișinău, 2024

TABLE OF CONTENTS

Theory	3
Formal Languages	3
Regular Grammars	3
Finite Automata	3
Objectives	4
Implementation Description	4
The Grammar class	4
The FiniteAutomaton class	6
Other implementations	7
Conclusion	8

Theory

Formal Languages

A formal language is a set of strings of symbols drawn from some alphabet. These languages are used to describe the syntax of programming languages, regular expressions, and many other areas. An alphabet is a finite set of symbols used to form strings. Strings are sequences of symbols from an alphabet.

In computer science, this formal language is essential for the definition of computer programs and the expression of algorithmic problems. Formal Languages are classified into different levels based on the Chomsky hierarchy. These levels include Recursively enumerable languages (type 0), Context-free languages (type 1), Context-sensitive languages (type 2) and Regular languages (type 3). These languages all have different sets of rules for construction and provide different levels of expressibility. Furthermore, formal language theory provides systematic ways to determine whether a given string adheres to the rules of a language, which is fundamental for the creation of software like compilers or interpreters.

Regular Grammars

Regular grammars are formal systems used to describe regular languages. Components of Regular Grammars are terminals, non-terminals, productions and the start symbol. Regular Expressions are a concise way to describe regular languages using algebraic expressions. Moreover, regular grammars and regular expressions are equivalent in terms of expressive power.

Finite Automata

Finite Automata are abstract computational devices used to recognize patterns within strings. It's characterised by limited memory and the potential to change from one state to another when triggered by external inputs. Types of Finite Automata:

- Deterministic Finite Automata (DFA)
- Nondeterministic Finite Automata (NFA)

Components of Finite Automata:

- States - represent different configurations of the machine.
- Transitions - define the movement from one state to another based on input symbols.
- Accepting states - indicate whether a particular input string is accepted or rejected.

Acceptance Criteria is when a string is accepted if the automaton reaches an accepting state after processing the entire input string. Finite automata can recognize exactly the class of regular languages.

Deterministic Finite Automata

Deterministic Finite Automata (DFA) is a type of Finite Automata where for each state and input symbol, there exists one and only one transition. This essentially means that a DFA cannot have multiple paths for the same input from any given state or an undefined path. DFAs function on a finite set of input

symbols and from each state for every input symbol, the automaton deterministically transits to a next state. This brings about deterministic computation, allowing DFAs to process regular languages, which are the simplest form of formal languages in computer science. A DFA accepts an input string if and only if the DFA ends in an accepting (or final) state after processing the entire string.

Non-Deterministic Finite Automata

Non-Deterministic Finite Automata (NFA) is a variation of Finite Automata in which one or more specific condition transitions are not necessarily defined for all states, or there may be several uniquely defined transitions for the same state and input symbol. The ace that a NFA holds over a DFA is its ability to transition to multiple next states from a particular state for the same input symbol. Alternatively, an NFA can choose to completely neglect an input symbol from a state, leading it to a null transition. This allows more flexibility in modelling real-world computational problems. NFAs recognise the same class of languages as DFAs, known as regular languages, though sometimes with a simpler and more intuitive structure.

Objectives

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

Implementation Description

In this project, a grammar defined by a set of non-terminals (VN), terminals (VT), and production rules (P) was implemented to generate strings. Additionally, the grammar was converted into a Finite Automaton (FA) to determine the acceptability of generated and pre-defined strings.

The Grammar class

The Grammar class encompasses the structure and functionalities of the given context-free grammar.

```
def __init__(self):  
    self.VN = {'S', 'D', 'E', 'J'}  
    self.VT = {'a', 'b', 'c', 'd', 'e'}
```

```

self.P = {
    'S': ['aD'],
    'D': ['dE', 'bJ', 'aE'],
    'J': ['cS'],
    'E': ['e', 'aE']
}

```

The `generate_string` method recursively generates strings based on the grammar rules until a maximum length is reached.

```

def generate_string(self, symbol=None, length=0, max_length=15):
    # If no symbol is provided, start from the initial state
    if symbol is None:
        symbol = 'S'

    if length > max_length:
        return ''

    # If the current symbol is a terminal, it returns it
    if symbol in self.VT:
        return symbol

    # Otherwise, it generates a string for each production of the current symbol
    production = random.choice(self.P.get(symbol, []))
    generated_string = ''
    for s in production:
        generated_string += self.generate_string(s, length + 1, max_length)
    return generated_string

```

The method `convert_to_fa` outlines the process of converting the grammar into a Finite Automaton by defining transitions based on the grammar production rules. Additionally, it clarifies the purpose of the final state and how it is used in the conversion process.

```

def convert_to_fa(self):
    # Initializes the transitions dictionary and the final state
    transitions = {}
    final_state = 'DEAD'

```

```

# Initializes transitions for each non-terminal
for non_terminal in self.VN:
    transitions[non_terminal] = {}

# Adds transitions for each production
for non_terminal, productions in self.P.items():
    for production in productions:
        # If the length of the current production is 1 and it's a terminal, add
        if len(production) == 1 and production[0] in self.VT:
            transitions[non_terminal][production] = final_state
        else:
            # Otherwise, add a transition to the new state defined by the prod
            transition, new_state = production[0], production[1]
            transitions[non_terminal][transition] = new_state

return FiniteAutomaton(
    states=self.VN.union({final_state}),
    alphabet=self.VT,
    transitions=transitions,
    initial_state='S',
    accept_states={final_state}
)

```

The FiniteAutomaton class

The FiniteAutomaton class represents a finite automaton with states, transitions, and acceptance states. It checks whether a given input string can be obtained via state transitions from the initial state to an accept state. After a Finite Automaton object is initialized, the accepts method checks whether the input string is accepted by the Finite Automaton.

```

def accepts(self, input_string):
    current_state = self.initial_state

    for symbol in input_string:
        # Check if the current state has a transition defined for the current symbol
        if current_state in self.transitions and symbol in self.transitions[current_state]:

```

```

        current_state = self.transitions[current_state][symbol]
    else:
        # If there's no transition defined, the string cannot be obtained by s
        return False
    if symbol not in self.alphabet:
        # If the symbol is not part of the alphabet, the string cannot be obta
        return False

    # Check if the final state after processing all symbols is one of the accept s
    return current_state in self.accept_states

```

Other implementations

Testing functionalities ensure the correctness of the implemented grammar and Finite Automaton by generating and validating strings against defined criteria. Additionally, it outlines the testing of pre-defined strings and the printing of their acceptance/rejection status.

```

# Testing the functionalities

grammar = Grammar()
automaton = grammar.convert_to_fa()

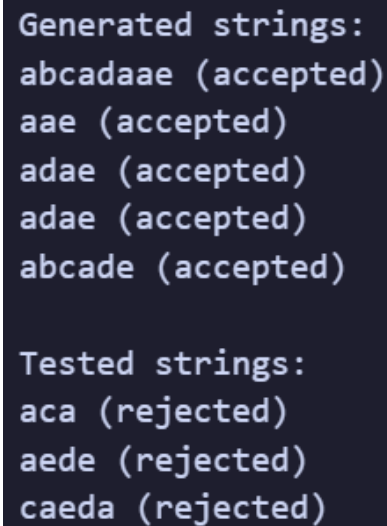
print("Generated strings:")
for _ in range(5):
    gen_string = grammar.generate_string(max_length=15)
    if automaton.accepts(gen_string):
        print(f"{gen_string} (accepted)")
    else:
        print(f"{gen_string} (rejected)")

print("\nTested strings:")
tested_strings = ['aca', 'aede', 'caeda']
for string in tested_strings:
    if automaton.accepts(string):
        print(f"{string} (accepted)")
    else:
        print(f"{string} (rejected)")

```

Conclusion

By running this program, we see a list of strings that have been generated and tested. Each string is marked as 'accepted', indicating that the `generate_string` method from the `Grammar` class is producing strings that are valid within the language defined by the grammar. *Figure 1* also shows several strings that were tested for acceptance by the automaton: 'aca', 'aede', 'caeda'. These strings are marked as 'rejected', which means they do not match the language rules defined by the finite automaton.



```
Generated strings:
abcadaae (accepted)
aae (accepted)
adae (accepted)
adae (accepted)
abcade (accepted)

Tested strings:
aca (rejected)
aede (rejected)
caeda (rejected)
```

Figure 1. Example of an output

This implementation demonstrates the transformation of a context-free grammar into a Finite Automaton, enabling the generation and validation of strings according to specified grammar rules. The project underscores the versatility and practicality of formal language theory in computational linguistics and language processing applications.