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

Laboratory work 2:
Determinism in Finite Automata. Conversion from NFA 2 DFA. Chomsky Hierarchy.

Elaborated: st. gr. FAF-223 Friptu Ludmila

Verified: asist. univ. Cretu Dumitru

THEORY

1. The Chomsky Hierarchy

The Chomsky hierarchy is a classification of formal languages based on their generative grammars. Developed by Noam Chomsky in the 1950s, this hierarchy divides languages into four levels, each with increasing expressive power. These levels are:

1. Type 0 (Recursively Enumerable Languages): The most general class, generated by unrestricted grammars. These languages can be recognized by a Turing machine, but not all can be decided (meaning that there is not always a Turing machine that can always halt and state whether a given string belongs to the language or not).

2. Type 1 (Context-Sensitive Languages): Generated by context-sensitive grammars, these languages can be recognized by a linear bounded automaton (a Turing machine with a tape length linearly bounded by the input size). Context-sensitive languages are more restricted than recursively enumerable languages and include all the languages that can be generated by grammars where the productions have contexts around the symbols being replaced.

3. Type 2 (Context-Free Languages): These languages are generated by context-free grammars, which are used to describe the syntax of programming languages and can be recognized by pushdown automata. The productions in a context-free grammar allow a single non-terminal symbol to be replaced by a string of terminal and/or non-terminal symbols, without any context considerations.

4. Type 3 (Regular Languages): The simplest class, generated by regular grammars, and recognizable by finite automata. Regular languages are those that can be generated by grammars that have productions with a single non-terminal symbol being replaced by a single terminal symbol, possibly followed by a single non-terminal symbol.

2. NFAs and DFAs

NFAs (Nondeterministic Finite Automata) and DFAs (Deterministic Finite Automata) are both types of finite automata used in the study of computer science and formal language theory. They are models of computation that define how inputs are processed and accepted or rejected based on the state transitions of the automaton. Despite serving similar purposes, NFAs and DFAs have distinct characteristics:

Deterministic Finite Automata (DFAs)

- Deterministic: In a DFA, for a given state and input symbol, the next possible state is uniquely determined. There is exactly one transition for every symbol of the alphabet in each state.
- Structure: A DFA consists of a finite set of states, a finite set of input symbols, a transition function, a start state, and a set of accept states.
- Computation: The DFA reads an input string symbol by symbol, and based on the current state and the input symbol, it moves to the next state according to its transition function. If the input ends in an accept state, the input is accepted; otherwise, it is rejected.

Nondeterministic Finite Automata (NFAs)

- Nondeterministic: In an NFA, for a given state and input symbol, there can be several possible next states. An NFA can choose any of these next moves; in other words, it can "guess" the right path to take. Additionally, NFAs can have ε-transitions.
- Structure: Similar to DFAs, NFAs are defined by a finite set of states, a finite set of input symbols, transition functions , a start state, and accept states.
- Computation: NFAs can move to several new states from a given state and symbol, or even without consuming a symbol. If any sequence of transitions leads to an accept state by the end of the input, the input is accepted.
- Equivalence: Despite their nondeterminism, NFAs are equivalent to DFAs in terms of the languages they can recognize—every NFA has an equivalent DFA that recognizes the same language, although the resulting DFA may have exponentially more states.

OBJECTIVES

PART I

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. Understand what an automaton is and what it can be used for.
2. Continuing the work in the same repository and the same project, the following need to be added:

a. Provide a function in your grammar type/class that could classify the grammar based on Chomsky hierarchy.
b. For this you can use the variant from the previous lab.

IMPLEMENTATION

In order to classify my grammar according to the Chomsky hierarchy, I implemented a new method called: *classify_grammar*. It checks if the grammar is regular, context-free, or context-sensitive, and returns a string indicating the type of the grammar.

```
def classify_grammar(self):
    if self.is_regular():
        return "Type 3 (Regular)"
    elif self.is_context_free():
        return "Type 2 (Context-Free)"
    elif self.is_context_sensitive():
        return "Type 1 (Context-Sensitive)"
    else:
        return "Type 0 (Recursively Enumerable)"
```

Then I implemented a method called *is_regular()* within the Grammar class. The purpose of this method is to determine whether the grammar is a regular one. The method iterates through each production rule in the grammar. The production rules are stored in the dictionary self.P, where each key-value pair corresponds to a non-terminal symbol (lhs for left-hand side) and its possible expansions (productions). For each non-terminal symbol, it checks if the length of the lhs is greater than 1. In a regular grammar, the left-hand side of a production rule must be a single non-terminal symbol. If any lhs has more than one symbol, the method returns False. For each production associated with a non-terminal symbol, the method checks the format of the production to ensure it conforms to one of the allowed patterns for regular grammars. If the production is a single terminal symbol (production.islower()), it's a valid regular grammar production. The method continues to the next production without taking any action. If the production consists of two symbols where the first is a terminal (production[0].islower()) and the second is a non-terminal (production[1].isupper()), it's considered a right-linear production. Similarly, if the production consists of two symbols where the first is a non-terminal (production[0].isupper()) and the second is a terminal (production[1].islower()), it's considered a left-linear production. This pattern is also valid for regular grammars, and the method continues to the next production. If a production doesn't match any of the allowed patterns, the method concludes that the grammar is not regular and returns False, otherwise, returns True.

```
def is_regular(self):
    for lhs, productions in self.P.items():
        if len(lhs) > 1:
            return False
        for production in productions:
            if len(production) == 1 and production.islower():  # Terminal symbol
                continue
            elif len(production) == 2 and production[0].islower() and production[1].isupper():  # aB form
                continue
            elif len(production) == 2 and production[1].islower() and production[
                0].isupper():  # Ba form (for left-linear)
                continue
            else:
                return False
    return True
```

Next, I implemented the *is_context_free* method designed to determine if a given grammar is a context-free grammar. Each production rule in a CFG has a single non-terminal symbol on its left-hand side (LHS) and a string of terminal and/or non-terminal symbols on its right-hand side (RHS). For a grammar to be considered context-free, each production rule must have exactly one non-terminal symbol on its LHS. This is what the condition *len(lhs) != 1* is checking for. If the length of *lhs* is not exactly one, it means there's either no symbol or more than one symbol on the left side of a production. Non-terminal symbols are typically represented by uppercase letters, while terminal symbols are represented by lowercase letters, symbols, or numbers. The check *not lhs.isupper()* verifies that the LHS symbol is uppercase. If it's not, the grammar does not adhere to the convention for CFGs, suggesting it might not be context-free. The method iterates over all production rules (*self.P.items()*), where *self.P* is presumably a dictionary representing the production rules of the grammar with non-terminal symbols as keys and their corresponding production lists as values (*productions)*.

```
def is_context_free(self):
    print("Checking if context free...")
    for lhs, productions in self.P.items():
        if len(lhs) != 1 or not lhs.isupper():
            return False
    return True
```

Next I implemented the *is_context_sensitive* method. Context-sensitive grammars are more general than context-free grammars and allow rules where the length of the string on the left-hand side of a production rule can be less than or equal to the length of the string on the right-hand side. The variable *s_on_right* is initially set to False. This variable tracks whether the start symbol (S) appears on the right-hand side of any production rule. The method iterates through all production rules in *self.P.items()*, where *self.P* represents the production rules of the grammar. *left* is the left-hand side of a production rule, and *productions* is a list of right-hand side possibilities for that rule. If the start symbol S is found in the left side of any production rule, *s_on_right* is set to True. Then, the code explicitly checks for a production

rule where the start symbol S produces the empty string. According to context-sensitive grammar rules, S -> ε is allowed if S does not appear on the RHS of any production rule. The condition *if left == 'S' and production == '' and not s_on_right* checks for this specific case and skips the usual length comparison for this rule. The core condition that defines a context-sensitive grammar is checked with *if len(left) > len(production):*. This line checks if the length of the LHS of a production rule is greater than the length of its RHS, which would violate the condition for a grammar to be context-sensitive. If such a rule is found, the method returns False, indicating the grammar is not context-sensitive.

```
def is_context_sensitive(self):
    print("Checking if context sensitive...")
    s_on_right = False
    for left, productions in self.P.items():
        if 'S' in left:
            s_on_right = True
        for production in productions:
            # Check for S -> epsilon
            if left == 'S' and production == '' and not s_on_right:
                continue
            if len(left) > len(production):
                return False
    return True
```

Results:

```
Type 3 (Regular)
Generated strings from the grammar:
acaabb
bbb
accaabaaaab
babaabb
baabb
```

PART II

According to your variant number (by universal convention it is register ID), get the finite automaton definition and do the following tasks:
a. Implement conversion of a finite automaton to a regular grammar.
b. Determine whether your FA is deterministic or non-deterministic.
c. Implement some functionality that would convert an NFA to a DFA.
d. Represent the finite automaton graphically (Optional, and can be considered as a *bonus point*):

- You can use external libraries, tools or APIs to generate the figures/diagrams.
- Your program needs to gather and send the data about the automaton and the lib/tool/API return the visual representation**.**

$Q = \{q0,q1,q2,q3\}$, $\sum = \{a,b,c\}$, $F = \{q3\}$,

$\delta(q0,a) = q1$, $\delta(q1,b) = q2$, $\delta(q2,c) = q0$, $\delta(q1,a) = q3$, $\delta(q0,b) = q2$, $\delta(q2,c) = q3$.

IMPLEMENTATION

In order to convert the finite automaton to a regular grammar I defined my FiniteAutomaton class.

```
class FiniteAutomaton:
    def __init__(self):
        self.states = {'q0', 'q1', 'q2', 'q3'}
        self.alphabet = {'a', 'b', 'c'}
        self.transitions = {
            'q0': {'a': 'q1', 'b': 'q2'},
            'q1': {'a': 'q3', 'b': 'q2'},
            'q2': {'c': ['q0', 'q3']},
        }
        self.start_state = 'q0'
        self.accept_states = {'q3'}
```

Then I implemented the *to_regular_grammar* method that converts a finite automaton into regular grammar. The method starts by creating an empty dictionary named *grammar*, which

will be populated with the grammar's production rules. The outer loop iterates through each state in *self.states*. For every state, it initializes an empty list named *rules* that will store the production rules originating from that state. For each state, the method accesses its transition rules from *self.transitions*. Since the automaton could be nondeterministic, its transitions might lead to multiple states for a single input symbol. The code ensures that *destinations* is always a list, even if there's only a single destination state. For each destination state that a given symbol leads to, the method checks if the destination is an accepting state (*destination in self.accept_states*). If it is, the symbol itself is added as a production rule because reaching an accept state implies the end of a string in the language. If the destination is not an accept state, a production rule is created that consists of the input symbol followed by the destination state (*f'{symbol}{destination}*). After processing all transitions for a state, the list of *rules* is assigned to the *grammar* dictionary under the key corresponding to that state. This creates a set of production rules for each nonterminal state in the grammar. Finally, the method returns the *grammar* dictionary.

```
def to_regular_grammar(self):
    grammar = {}
    for state in self.states:
        rules = []
        for symbol, destinations in self.transitions.get(state, {}).items():
            if not isinstance(destinations, list):
                destinations = [destinations]
            for destination in destinations:
                if destination in self.accept_states:
                    rules.append(symbol)
                else:
                    rules.append(f'{symbol}{destination}')
        grammar[state] = rules
    return grammar
```

Next, in order to check if my grammar is deterministic or not, I implemented the method *is_deterministic* that checks whether a given finite automaton is deterministic. The method starts by iterating through all transition rules of the FA, which are stored in *self.transitions*. For each state's transitions, it iterates through the destinations for each input symbol. The destinations are the values in the inner dictionary of *transitions*. The method checks if *destinations* is a list and if it contains more than one element using *if isinstance(destinations, list) and len(destinations) > 1:* This condition is true if, for a given input symbol from a given state, there are multiple next states. If such a condition is found, the method immediately returns False, indicating that the FA is not deterministic, otherwise it is True.

```python
def is_deterministic(self):
    for transitions in self.transitions.values():
        for destinations in transitions.values():
            if isinstance(destinations, list) and len(destinations) > 1:
                return False  # Nondeterministic if any symbol leads to more than one state
    return True
```

Results:

```
q2 --> cq0|c
q0 --> aq1|bq2
q1 --> a|bq2
Is deterministic: False
```

FROM NFA TO DFA

First I defined my NFA class:

```python
class NDFA:
    def __init__(self):
        self.states = {'q0', 'q1', 'q2', 'q3'}
        self.input_symbols = {'a', 'b', 'c'}
        self.transitions = {
            'q0': {'a': {'q1'}, 'b': {'q2'}},
            'q1': {'a': {'q3'}, 'b': {'q2'}},
            'q2': {'c': {'q0', 'q3'}},
            'q3': {}
        }
        self.start_state = 'q0'
        self.accept_states = {'q3'}
```

Then, my DFA class:

```
class DFA:
    def __init__(self):
        self.states = set()
        self.input_symbols = set()
        self.transitions = {}
        self.start_state = None
        self.accept_states = set()
```

The actual conversion from NFA to DFA is made with the help of the *from_ndfa* method in the DFA class. The method begins by copying the input symbols from the NFA. It initializes *unprocessed_states* with a frozenset containing only the start state of the NFA. This frozenset represents a state in the DFA, where each state is a set of NFA states. The start state of the DFA is set to this initial frozenset. The method enters a loop that continues until there are no unprocessed states left. Each iteration processes a current state (a set of NFA states). For each input symbol, the method calculates the next state as a frozenset of all possible states that can be reached from any state in *current_state* under this input symbol. This calculation is based on the NFA's transition function. If the calculated *next_state* is not empty, it is added to the DFA's transition function. The key for this entry is a tuple (current_state, input_symbol), and the value is *next_state*. *next_state* is also added to *unprocessed_states* if it hasn't been processed yet. After all states have been processed, the method identifies the accept states for the DFA. This is determined by checking if the intersection of *state* and *ndfa.accept_states* is not empty.

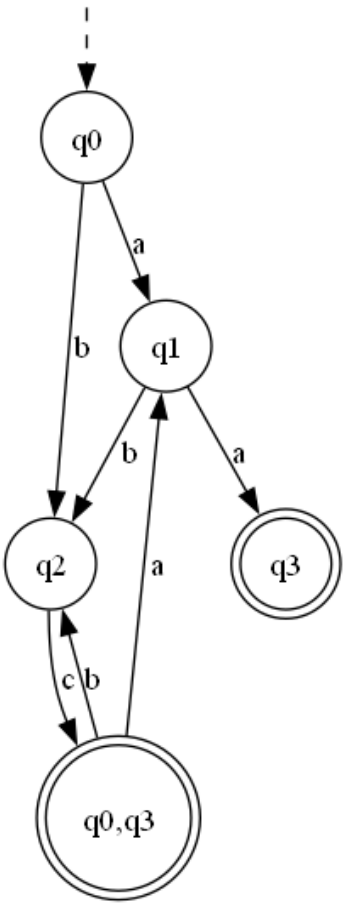
```
def from_ndfa(self, ndfa):
    self.input_symbols = ndfa.input_symbols
    unprocessed_states = [frozenset([ndfa.start_state])]
    self.start_state = frozenset([ndfa.start_state])

    while unprocessed_states:
        current_state = unprocessed_states.pop()
        if not current_state in self.states:
            self.states.add(current_state)
            for input_symbol in ndfa.input_symbols:
                next_state = frozenset(
                    [s for state in current_state for s in ndfa.transitions.get(state, {}).get(input_symbol, [])])
                if next_state:
                    self.transitions[current_state, input_symbol] = next_state
                    unprocessed_states.append(next_state)

    self.accept_states = {state for state in self.states if ndfa.accept_states.intersection(state)}
```

Results:

I represented the DFA graphically using GraphViz, which is an open source graph visualization software.



Conclusion

In this lab, I have explored the fundamental concepts of the Chomsky hierarchy and the process of converting nondeterministic finite automata to deterministic finite automata. Through theoretical exploration and practical implementation, I've gained insights into the structural differences and computational implications of various classes of formal languages and automata within the Chomsky hierarchy, specifically focusing on the transition from nondeterminism to determinism in the context of finite automata. My study emphasized the practical significance of regular languages and finite automata, illustrating how these concepts underpin many computational processes and algorithms.

The conversion of NFAs to DFAs is pivotal in computational theory and applications. While NFAs provide a more flexible and intuitive way to design finite automata, DFAs are crucial for implementation due to their deterministic nature, which ensures a unique computation path for any given input string. The subset construction algorithm, demonstrated in this lab, is a systematic method for achieving this conversion, showcasing how a potentially exponential increase in the number of states can result from the need to maintain determinism.

In conclusion, the study of the Chomsky hierarchy and the NFA to DFA conversion process illuminates the intricate balance between expressiveness and computational efficiency in the design of computational models. This lab has provided me with a deeper understanding of the theoretical foundations of computer science, while also equipping me with practical skills and insights that are applicable in various domains, from software development to the analysis of complex systems.