Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Course: Formal Languages & Finite Automata

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

Elaborated:

st. gr. FAF-221 Clepa Rodion

Verified:

asist. univ. Dumitru Crețu

Chişinău – 2024

Objectives

Understand what an automaton is and what it can be used for.

Continuing the work in the same repository and the same project, the following need to be added:

Provide a function in your grammar type/class that could classify the grammar based on Chomsky hierarchy.

For this you can use the variant from the previous lab.

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 NDFA to a DFA.

Implementation

```
def check_chomsky(self):
        if self.is_regular() is True:
            return "Type 3"
        elif self.is_context_free() is True:
            return "Type 2"
        elif self.is_context_sensitive() is True:
            return "Type 1"
        else:
            return "Type 0"
```

check_chomsky method:

- This method determines the Chomsky type of grammar.
- It checks each type in order (Type 3, Type 2, Type 1, and finally Type 0).
- If a type check (is_regular, is_context_free, is_context_sensitive) returns True, it returns the corresponding Chomsky type as a string.
- If none of the type checks pass, it returns "Type 0" indicating an unrestricted grammar.

```
def is_regular(self):
        for i in self.P:
            if len(self.P[i])>2:
```

```
                return False
            for production in self.P[i]:
                for nonTerm in self.VN:
                    symbols = production.replace(nonTerm, "")
                if symbols == "":
                    return False
        return True
```

is_regular method:

- It iterates over each non-terminal in grammar.
- For each non-terminal, it checks the number of productions associated with it. If there are more than two productions, it returns False because regular grammars can have at most one production per non-terminal.
- After removal process the production becomes an empty string (**symbols == ""**), it means the production only contained non-terminals, violating the regular grammar structure. In this case, it returns False.

```
def is_context_free(self):
        for i in self.P:
            non_terminal = i
            for j in self.VN:
                non_terminal = non_terminal.replace(j, "")
            if non_terminal != "":
                return False
        return True
```

is_context_free method:

- It iterates over each non-terminal in grammar.
- For each non-terminal, it removes occurrences of all non-terminals from it (non_terminal = **non_terminal.replace(j, ""))**. After this process, if the resulting string is not empty (**non_terminal != ""**), it means the non-terminal had terminals or other symbols besides the non-terminals. In this case, it returns False because context-free grammars expect non-terminals on the left-hand side only.

```
def is_context_sensitive(self):
        for non_terminal in self.P:
            for production in self.P[non_terminal]:
                if len(non_terminal) > len(production):
                    return False
        return True
```

is_context_sensitive method:

- It iterates over each non-terminal in grammar.
- For each production associated with the current non-terminal, it compares the length of the non-terminal to the length of the production. If the non-terminal is longer than the production, it returns False because context-sensitive grammars expect productions to be at least as long as the non-terminal.

```
def determine_DFA_or_NDFA(self):
        for i in self.transitions:
            for symbol in self.transitions[i]:
                if len(self.transitions[i][symbol])>1:
                    return "NDFA"
        return "DFA"
```

It iterates over each state in the FA

For each state, it iterates over the symbols in the alphabet

For each symbol, it checks the number of transitions associated with that state and symbol. If there is more than one transition (**len(self.transitions[i][symbol]) > 1**), it means the FA has a non-deterministic choice at that state for that symbol. In this case, it immediately returns "NDFA"

```
class Grammar():
    def __init__(self, finite):
        self.VN = finite.listStates
        self.VT = finite.alphabet
        self.S = finite.startState
        self.P = {}
```

```
        trns = finite.transitions
        for sender in trns:
            if sender not in self.P:
                self.P[sender] = []
            for symbol in trns[sender]:
                for destinator in trns[sender][symbol]:
                    if destinator == finite.finalState:
                        terminal = symbol
                    else:
                        terminal = symbol + destinator

                    self.P[sender].append(terminal)
```

It iterates over the transitions of the Finite Automaton (finite.transitions), where each state (sender) is associated with a dictionary of symbols and destination states.

For each sender state, it checks if it's already in the set of non-terminals (VN). If not, it adds it.

It then iterates over the symbols in the transitions of the current sender state.

For each symbol, it iterates over the destination states associated with that symbol.

It appends the determined terminal to the list of productions for the sender state in the **P** dictionary.

```
def convert_NDFA_to_DFA(self):
        state_table = {}
        newState = [(self.startState, )]
        alreadyMet = [self.finalState]
        while newState:
            current_state = newState[0]
            if current_state not in state_table:
                state_table[current_state] = {}
```

```
            trnsFnct = {}
            trnsFnct[current_state] = {}
            for symbol in self.alphabet:
                for state in current_state:
                    # In case if state doesnt have any transitions
                    if state not in self.transitions:
                        if symbol not in state_table[current_state]:
                            state_table[current_state][symbol] = ""
                        else:
                            # To skip state without transitions
                            if state in self.transitions:
                                state_table[current_state][symbol] =
tuple(sorted(set(

tuple(state_table[current_state][symbol]) +
tuple(self.transitions[state][symbol])
                                    )))
                        continue


                    if symbol not in self.transitions[state]:
                        if symbol not in state_table[current_state]:
                            state_table[current_state][symbol] = ""
                    else:
                        if symbol not in state_table[current_state]:
                            state_table[current_state][symbol] =
tuple(sorted(self.transitions[state][symbol]))
                        else:
```

```
                            state_table[current_state][symbol] =
tuple(sorted(set(
                                    tuple(state_table[current_state][symbol])
+ tuple(self.transitions[state][symbol])
                                    )))


            alreadyMet.append(newState[0])
            newState.pop(0)
            # Append new states un met state
            for passState in state_table:
                for symbol in state_table[passState]:
                    combState = state_table[passState][symbol]
                    if combState not in alreadyMet and combState !="":
                        newState.append(combState)


        self.transitions = state_table
        print(self.transitions)
```

state_table: A dictionary to store DFA states and their transitions.

newState: A list containing the initial state of the DFA, initialized with the start state of the NDFA.

alreadyMet: A list to keep track of states that have been processed.

For each symbol in the alphabet, it constructs the transition function for the current state.

It considers each individual state within the current state.

If a state has no transitions for a symbol, it handles this case appropriately.

It constructs the combined state resulting from transitions and ensures uniqueness and order.

It updates the **state_table** with the transition function for the current state.

It marks the current **state** as processed by adding it to the **alreadyMet** list.

It removes the current **state** from **newState**.

It explores new states by looking at the transitions in the **state_table**.

For each state in the table, it considers its transitions and appends unprocessed states to **newState**.

Conclusions

In conclusion, the laboratory work focused on formal languages and finite automata covered several key objectives. The tasks involved understanding the concept of automata, their applications, and the conversion processes between different types. The work was conducted in the context of the Chomsky Hierarchy, encompassing regular grammars, context-free grammars, and context-sensitive grammars.