Converting Grammar to Finite Automaton

Course: Formal Languages & Finite Automata

Author: Ernu Catalina

Theory:

In formal language theory, a grammar is a set of rules used to generate strings in a language, while a finite automaton is a mathematical model of computation. Converting a grammar to a finite automaton involves representing the grammar's rules as states and transitions in the automaton. Regular grammars, which have specific restrictions on their production rules, can be easily converted to finite automata, facilitating further analysis and manipulation.

Objectives:

- Understand the relationship between grammars and finite automata.
- Learn how to convert a grammar to a finite automaton.
- Implement the conversion process programmatically.

Grammar Class:

Purpose: Represents a formal grammar with non-terminals, terminals, and production rules.

Functionality:
- __init__: Initializes the grammar with provided non-terminals, terminals, and production rules.
- __str__: Returns a string representation of the grammar, including its non-terminals, terminals, and production rules.
- classify_chomsky: Classifies the grammar based on the Chomsky hierarchy, determining whether it is regular, context-free, context-sensitive, or unrestricted.

Code Snippet:
```
class Grammar:
    def __init__(self, non_terminals, terminals, productions):
        self.non_terminals = non_terminals
        self.terminals = terminals
        self.productions = productions

    def __str__(self):
```

```
        productions_str = "\n".join([f"{non_terminal} -> {' | '.join(productions)}" for non_terminal, productions in self.productions.items()])
        return f"Non-terminals: {self.non_terminals}\nTerminals: {self.terminals}\nProductions:\n{productions_str}"

    def classify_chomsky(self):
        # Check if the grammar is regular
        regular = all(len(production) <= 2 and (len(production) == 1 or production[0] in self.non_terminals)
                      for productions in self.productions.values() for production in productions)

        # Check if the grammar is context-free
        context_free = all(len(production) == 1 for productions in self.productions.values() for production in productions)

        # Check if the grammar is context-sensitive
        context_sensitive = not regular and not context_free

        # If none of the above are True, the grammar is unrestricted
        if not any([regular, context_free, context_sensitive]):
            return "Type 0 : Unrestricted"
        elif regular:
            return "Type 3 : Regular"
        elif context_free:
            return "Type 2 : Context-Free"
        elif context_sensitive:
            return "Type 1 : Context-Sensitive"
```

FiniteAutomaton Class:

Purpose: Represents a finite automaton with states, alphabet, transitions, initial state, and accepting states.

Functionality:

- __init__: Initializes the finite automaton with empty sets for states, alphabet, transitions, initial state, and accepting states.
- convert_from_grammar: Converts a given grammar to a finite automaton by mapping grammar symbols to automaton states and transitions.
- __str__: Returns a string representation of the finite automaton, including its states, alphabet, and transition functions.
- check_string: Checks whether a given input string is accepted by the finite automaton by following its transitions from the initial state.

Code Snippet:

```
class FiniteAutomaton:
    def __init__(self):
        self.states = set()
        self.alphabet = set()
        self.transitions = {}
        self.initial_state = None
        self.accepting_states = set()

    def convert_from_grammar(self, grammar):
        for non_terminal in grammar.non_terminals:
            self.states.add(non_terminal)
        for terminal in grammar.terminals:
            self.alphabet.add(terminal)

        for non_terminal, productions in grammar.productions.items():
            for production in productions:
                if len(production) == 1:  # Singleton production
                    self.transitions.setdefault(non_terminal,
{}).setdefault(production, 'ε')
                else:
                    self.transitions.setdefault(non_terminal,
{}).setdefault(production[0], production[1])

        self.initial_state = 'S'
        self.accepting_states = grammar.terminals

    def __str__(self):
        transitions_str = "\n".join([f"{state}: {transitions}" for
state, transitions in self.transitions.items()])
        return f"States: {self.states}\nAlphabet:
{self.alphabet}\nTransitions:\n{transitions_str}"

    def check_string(self, input_string):
        current_state = self.initial_state
        for symbol in input_string:
            if symbol not in self.transitions.get(current_state, {}):
                return False
            current_state =
self.transitions[current_state].get(symbol)
            if current_state is None:
                return False
        return current_state in self.accepting_states
```

Conversion Functions:

Purpose: Convert the finite automata to regular grammar.

Functionality:

- fa_to_regular_grammar: Converts a finite automaton to a regular grammar by mapping automaton states to non-terminals and transitions to production rules.
- is_deterministic: Checks if a given finite automaton is deterministic, i.e., each state has at most one transition for each symbol in the alphabet.

Code Snippet:

```
def fa_to_regular_grammar(fa):
    non_terminals = set()
    terminals = fa.alphabet
    productions = {}

    # Add non-terminals
    for state in fa.states:
        non_terminal = f'N_{state}'
        non_terminals.add(non_terminal)
        productions[non_terminal] = []

    # Add productions
    for state in fa.transitions:
        for symbol, next_state in fa.transitions[state].items():
            if next_state in fa.accepting_states:
                productions[f'N_{state}'].append(symbol)
            else:
                productions[f'N_{state}'].append(symbol +
f'N_{next_state}')

    return Grammar(non_terminals, terminals, productions)


def is_deterministic(fa):
    for state in fa.transitions:
        for symbol, next_state in fa.transitions[state].items():
            if isinstance(next_state, set):
                return False
    return True
```

Main Function:

Purpose: The main function serves as the entry point for the program, demonstrating the usage of the implemented classes and functions.

Functionality:

- Defines a grammar with non-terminals, terminals, and production rules.
- Generates and prints valid strings based on the grammar.
- Converts the grammar to a finite automaton.
- Converts the finite automaton to a regular grammar.
- Checks if the finite automaton is deterministic.
- Optionally, draw the finite automaton graphically.
- Classifies the grammar based on the Chomsky hierarchy.

Code Snippet:

```
if __name__ == "__main__":
    # Grammar definition
    non_terminals = {'S', 'B', 'L'}
    terminals = {'a', 'b', 'c'}
    productions = {
        'S': ['aB'],
        'B': ['bB', 'cL'],
        'L': ['cL', 'aS', 'b']
    }
    grammar = Grammar(non_terminals, terminals, productions)

    # Generating and printing valid strings
    valid_strings = generate_strings(grammar, 5)
    print("Valid Strings:")
    for string in valid_strings:
        print(string)

    # Convert Grammar to Finite Automaton
    fa = FiniteAutomaton()
    fa.convert_from_grammar(grammar)
    print("\nFinite Automaton:")
    print(fa)

    # Convert FA to regular grammar
    rg = fa_to_regular_grammar(fa)
    print("\nRegular Grammar:")
    print(rg)

    # Determine if FA is deterministic
    if is_deterministic(fa):
        print("\nThe Finite Automaton is deterministic.")
    else:
        print("\nThe Finite Automaton is non-deterministic.")
```

```
# Draw FA graphically
draw_fa_graph(fa)

# Classify Grammar based on Chomsky Hierarchy
grammar_classification = grammar.classify_chomsky()
print(f"Grammar Classification: {grammar_classification}")
```

Conclusions

In this laboratory, we learned about the relationship between grammars and finite automata. We implemented the conversion of a grammar to a finite automaton and vice versa. Additionally, we explored the classification of grammars based on the Chomsky hierarchy, providing insights into the expressive power of different types of grammars. Overall, the laboratory provided valuable hands-on experience in formal language theory and finite automata.

References

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). Introduction to Automata Theory, Languages, and Computation. Pearson Education India.

Sipser, M. (2012). Introduction to the Theory of Computation. Cengage Learning.

Chomsky, N. (1956). Three models for the description of language. IRE Transactions on Information Theory, 2(3), 113-124.