Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică
Departamentul Inginerie software și automatică

Laboratory work 1: Formal Languages and Finite Automata

Elaborated:
st. gr. FAF-232 Istrati Ștefan

Verified:
asist. univ. Crețu Dumitru

Chişinău - 2025

Theory:

A formal grammar is a set of rules used to generate strings in a specific language. It consists of non-terminal symbols (which can be replaced), terminal symbols (which appear in the final string), production rules (which define how symbols are replaced), and a start symbol.

A finite automaton (FA) is a computational model used to recognize patterns in strings. It consists of states, transitions, a start state, and accepting states. After converting a grammar into an FA, we can verify if a given string belongs to the language defined by the grammar.

Objectives:

Implement a class to represent a grammar.

Generate five valid strings based on the grammar rules.

Convert the grammar into a finite automaton.

Check if a given string can be derived using the finite automaton.

Implementation:

This implementation defines two classes: Grammar and FA (Finite Automaton). The Grammar class represents a formal grammar with production rules, and the FA class converts this grammar into a finite automaton for string validation. The program includes methods for generating strings, checking derivations, and simulating state transitions.

Class: Grammar

Attributes:

- Vn: List of non-terminal symbols.
- Vt: List of terminal symbols.
- P: String representation of production rules.
- S: Start symbol.
- P_dictionary: Dictionary representation of production rules.

Methods:

__init__(self, Vn, Vt, P, S)

Initializes the grammar by parsing the production rules into a dictionary format.

generate_string(self)

Generates a string by applying production rules from the start symbol until only terminal symbols remain. It also stores the transformation steps.

Returns:

- The generated string.
- The list of transformation steps.

generate_5_strings(self)

Generates five unique strings using generate_string(), ensuring no duplicates. It also prints the transformation steps for each string.

Returns:

- A set containing five unique strings.

reverse_dictionary(self)

Creates and returns the reverse mapping of production rules, swapping keys and values.

Returns:

- A dictionary where each terminal maps to possible non-terminal replacements.

check_string(self, string)

Attempts to derive the given string back to the start symbol using reversed production rules.

Returns:

- A list of derivation steps if a valid derivation is found.
- A message indicating failure if no valid derivation exists.

Class: FA (Finite Automaton)

Attributes:

- Vn: List of non-terminal states.
- Vt: List of terminal symbols.
- P: String representation of production rules.
- S: Start state.
- P_dictionary: Dictionary representation of production rules.

Methods:

__init__(self, Vn, Vt, P, S)

Initializes the finite automaton and parses production rules into a dictionary.

convert_grammar_to_fa(self)

Converts the given grammar into an equivalent finite automaton representation.

Returns:

- A set of states.
- A dictionary of state transitions.
- A set of accepting states.
- The start state.

check_string_via_transition(self, string)

Simulates the finite automaton’s state transitions to verify if a given string is accepted.

Returns:

- A list of steps detailing the transition process.
- A message indicating whether the string was accepted or rejected.

Main Execution

1. A Grammar object is created and initialized with non-terminals, terminals, production rules, and a start symbol.

```
Vn = ["S", "A", "B"]
Vt = ["a", "b", "c", "d"]
P = "S->bS|dA, A->aA|dB|b, B->cB|a"
S = "S"
grammar = Grammar(Vn, Vt, P, S)
```

2. Five unique strings are generated and displayed.

```
The steps taken to generate bdb:  ['S -> bS', 'bS -> bdA', 'bdA -> bdb']
The steps taken to generate bbdada:  ['S -> bS', 'bS -> bbS', 'bbS -> bbdA', 'bbdA -> bbdaA', 'bbdaA -> bbdadB', 'bbdadB -> bbdada']
The steps taken to generate bbdda:  ['S -> bS', 'bS -> bbS', 'bbS -> bbdA', 'bbdA -> bbddB', 'bbddB -> bbdda']
The steps taken to generate bbbbdda:  ['S -> bS', 'bS -> bbS', 'bbS -> bbbS', 'bbbS -> bbbbS', 'bbbbS -> bbbbdA', 'bbbbdA -> bbbbddB', 'bbbbddB -> bbbbdda']
The steps taken to generate ddca:  ['S -> dA', 'dA -> ddB', 'ddB -> ddcB', 'ddcB -> ddca']
```

3. The validity of a sample string is checked using grammar rules.

```
Checking if the string bdb was obtained via the finite set of production of rules from the Grammar:
['bdb <- Adb', 'bdb <- bdA', 'Adb <- AdA', 'bdA <- AdA', 'bdA <- bS', 'AdA <- AS', 'AdA <- AS', 'bS <- S', 'Valid derivation: S -> bdb']
```

4. A FA object is created from the grammar.
5. The finite automaton's states, transitions, and accepting states are displayed.

```
Converting the object of type Grammar to type Finite Automaton
States:  {'S', 'A', 'B'}
Tansitions:  {('S', 'b'): 'S', ('S', 'd'): 'A', ('A', 'a'): 'A', ('A', 'd'): 'B', ('B', 'c'): 'B'}
Terminals:  {'B', 'A'}
Start state:  S
```

6. The same sample string is checked using finite automaton transitions.

```
Checking if the string 'bdb' can be obtained via the state transition from FA:
['Start at S', "Read 'b', move from S to S", "Read 'd', move from S to A", "Read 'b', no transition exists. String rejected."]
```

This implementation allows for both forward generation of strings from grammar rules and backward verification using finite automaton states.

Conclusion:

This implementation successfully demonstrates the relationship between formal grammars and finite automata. By generating strings and verifying their validity using both approaches, we can see how grammars define languages and how finite automata recognize them. The ability to convert a grammar into an automaton highlights the fundamental connections between these two models in formal language theory. Future improvements could include optimizing the derivation process and extending the model to support more complex grammar types.