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

Laboratory work 5:
Formal Languages and Finite Automata

Elaborated:
st. gr. FAF-232 Istrati Ștefan

Verified:
asist. univ. Crețu Dumitru

Chişinău - 2025

Theory:

Context-Free Grammars (CFGs) are a class of formal grammars that are widely used to describe the syntax of programming languages, natural languages, and other formal languages. A context-free grammar consists of a set of production rules, where each rule maps a non-terminal symbol to a string of symbols, which can either be non-terminal or terminal. The non-terminal symbols define the structure of the language, while terminal symbols represent the actual content.

One common task when working with CFGs is to transform them into a more standardized or simplified form. One such form is Chomsky Normal Form (CNF), a grammar format that simplifies the structure of CFGs. Normalizing a CFG into CNF has important theoretical and practical implications, particularly in the fields of parsing and automata theory. CNF is particularly useful because it ensures that the grammar's production rules are restricted to a limited form, making it easier to analyze and apply algorithms like the CYK (Cocke-Younger-Kasami) algorithm for parsing.

What is Chomsky Normal Form (CNF)?

A context-free grammar is in Chomsky Normal Form if it satisfies the following conditions:

1. **Production rules must be of two forms:**
    - $A \rightarrow BC$ (where A, B, and C are non-terminal symbols, and B and C are not the start symbol).
    - $A \rightarrow a$ (where A is a non-terminal symbol and 'a' is a terminal symbol).
2. **No ε-productions (empty string productions)** are allowed, except for the start symbol under specific conditions (if the language includes the empty string).
3. **No unit productions** are allowed. A unit production is one where a non-terminal symbol directly maps to another non-terminal symbol (e.g., $A \rightarrow B$).
4. **No useless symbols** are allowed, meaning that every non-terminal symbol must be reachable from the start symbol, and must eventually derive some terminal string.

The transformation process of a CFG into CNF involves several steps, each of which addresses different aspects of the grammar's structure. These steps ensure that the grammar adheres to the form required by CNF.

Steps to Convert a Grammar to CNF

1. **Eliminate ε-productions**: An ε-production is a production of the form $A \rightarrow \varepsilon$, where A is a non-terminal and ε represents the empty string. If a CFG includes ε-productions, we must remove them while ensuring that the language generated by the grammar remains unchanged. This can be done by finding nullable non-terminals and adjusting the productions accordingly to account for the presence of the empty string.
2. **Eliminate unit productions**: A unit production is a production of the form $A \rightarrow B$, where both A and B are non-terminals. These productions must be removed by substituting the right-hand side of $A \rightarrow B$ with all the productions that B can derive. This ensures that no production directly maps from one non-terminal to another.
3. **Eliminate useless symbols**: Useless symbols are non-terminals that cannot derive any terminal string or are unreachable from the start symbol. We remove such symbols to simplify the grammar.
4. **Convert remaining productions into binary form**: In CNF, each production rule must have at most two non-terminals on the right-hand side. If a production rule has more than two non-terminals, we introduce new intermediate non-terminals to break down the rule into binary productions. For example, a rule like $A \rightarrow XYZ$ can be transformed into two rules: $A \rightarrow XY$ and $X \rightarrow Z$.
5. **Ensure that terminal symbols appear in the appropriate form**: In CNF, terminal symbols must appear alone on the right-hand side of production rules (i.e., $A \rightarrow a$). If a production has a combination of a terminal and a non-terminal (e.g., $A \rightarrow aB$), we must introduce a new non-terminal for the terminal and rewrite the production as $A \rightarrow XB$ and $X \rightarrow a$.

Objectives:

1. Learn about Chomsky Normal Form (CNF) [1].
2. Get familiar with the approaches of normalizing a grammar.
3. Implement a method for normalizing an input grammar by the rules of CNF.

- The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).
- The implemented functionality needs executed and tested.
- Also, another BONUS point would be given if the student will make the aforementioned function to accept any grammar, not only the one from the student's variant.

Implementation:

1. __init__(self, variables, terminals, start_symbol, productions)

- **Purpose**: Initializes a CFG object with the given components.
- **Parameters**:
  - variables: A set of non-terminal symbols (e.g., {'S', 'A', 'B'}).
  - terminals: A set of terminal symbols (e.g., {'a', 'b'}).
  - start_symbol: The start symbol of the grammar (e.g., 'S').
  - productions: A dictionary where keys are non-terminal symbols, and values are lists of production rules (right-hand sides).

```
def __init__(self, variables, terminals, start_symbol, productions):
    self.V = set(variables)
    self.T = set(terminals)
    self.S = start_symbol
    self.P = defaultdict(list)
    for lhs, rhs_list in productions.items():
        for rhs in rhs_list:
            self.P[lhs].append(tuple(rhs) if isinstance(rhs, str) else tuple(rhs))
```

2. eliminate_epsilon(self)

- **Purpose**: Eliminates ε-productions (productions that derive the empty string).
- **Process**:
  - First, it identifies nullable non-terminals (non-terminals that can derive ε).
  - Then, it generates all subsets of productions where nullable symbols are replaced with ε.
  - It adds the resulting new productions to the grammar, ensuring that the grammar still generates the same language.

```
def eliminate_epsilon(self):
    nullable = set()
    changed = True
    while changed:
        changed = False
        for A in self.V:
            if A not in nullable:
                for prod in self.P[A]:
                    if all(sym in nullable or sym == '' for sym in prod):
                        nullable.add(A)
                        changed = True

    new_P = defaultdict(list)
    for A in self.P:
        for prod in self.P[A]:
            subsets = list(itertools.product(*[[s, ''] if s in nullable else [s] for s in prod]))
            for alt in subsets:
                new_rhs = tuple(s for s in alt if s != '')
                if new_rhs != () or A == self.S:
                    if new_rhs not in new_P[A]:
                        new_P[A].append(new_rhs)
    self.P = new_P
```

3. eliminate_renaming(self)

- **Purpose**: Removes unit productions (productions where a non-terminal directly leads to another non-terminal).
- **Process**:
    - Identifies unit pairs (pairs of non-terminals where one can directly derive the other).
    - Replaces unit productions with the original productions of the non-terminal.
    - Iterates over the grammar to remove all unit productions.

```
def eliminate_renaming(self):
    unit_pairs = set()
    for A in self.V:
        unit_pairs.add((A, A))

    changed = True
    while changed:
        changed = False
        for A in self.V:
            for prod in self.P[A]:
                if len(prod) == 1 and prod[0] in self.V:
                    B = prod[0]
                    for C in self.V:
                        if (B, C) in unit_pairs and (A, C) not in unit_pairs:
                            unit_pairs.add((A, C))
                            changed = True

    new_P = defaultdict(list)
    for A in self.V:
        for (X, Y) in unit_pairs:
            if X == A:
                for prod in self.P[Y]:
                    if len(prod) != 1 or prod[0] not in self.V:
                        if prod not in new_P[A]:
                            new_P[A].append(prod)
    self.P = new_P
```

4. eliminate_inaccessible(self)

- **Purpose**: Removes inaccessible non-terminals (non-terminals that are not reachable from the start symbol).
- **Process**:
    - Starts with the start symbol and iteratively adds all reachable non-terminals based on the current productions.
    - Removes any non-terminals that cannot be reached.
    - Adjusts the production rules to reflect the removal of inaccessible non-terminals.

```
def eliminate_inaccessible(self):
    reachable = {self.S}
    changed = True
    while changed:
        changed = False
        for A in list(reachable):
            for prod in self.P[A]:
                for sym in prod:
                    if sym in self.V and sym not in reachable:
                        reachable.add(sym)
                        changed = True
    self.V = reachable
    self.P = {A: self.P[A] for A in self.V}
```

5. eliminate_non_productive(self)

- **Purpose**: Removes non-productive non-terminals (non-terminals that cannot generate terminal strings).
- **Process**:
    - Starts by marking all terminal symbols as productive.
    - Iteratively adds non-terminals that can eventually derive terminal strings (productive symbols).
    - Removes any non-terminal that cannot derive a terminal string.

```
def eliminate_non_productive(self):
    productive = set()
    changed = True
    while changed:
        changed = False
        for A in self.V:
            for prod in self.P[A]:
                if all(sym in self.T or sym in productive for sym in prod):
                    if A not in productive:
                        productive.add(A)
                        changed = True
    self.V = self.V & productive
    new_P = defaultdict(list)
    for A in self.V:
        for prod in self.P[A]:
            if all(sym in self.T or sym in self.V for sym in prod):
                new_P[A].append(prod)
    self.P = new_P
```

6. to_cnf(self)

- **Purpose**: Converts the grammar to Chomsky Normal Form (CNF).
- **Process**:
    - First, it processes terminal symbols that appear in productions with more than one symbol (since CNF only allows terminal symbols to appear in productions of the form A → a).
    - Then, it handles productions with more than two symbols on the right-hand side by introducing new variables to break them down into binary productions (productions with two non-terminals).
    - Finally, it ensures that all productions are in CNF form.

```python
def to_cnf(self):
    new_P = defaultdict(list)
    new_vars = {}
    counter = 1

    def get_var_for_terminal(t):
        if t not in new_vars:
            new_var = f"T{counter + len(new_vars)}"
            while new_var in self.V:
                new_var = f"T{counter + len(new_vars)}"
            new_vars[t] = new_var
            self.V.add(new_var)
            new_P[new_var].append((t,))
        return new_vars[t]

    for A in self.P:
        for prod in self.P[A]:
            if len(prod) == 1 and prod[0] in self.T:
                new_P[A].append(prod)
            else:
                new_rhs = []
                for sym in prod:
                    if sym in self.T:
                        new_rhs.append(get_var_for_terminal(sym))
                    else:
                        new_rhs.append(sym)

                while len(new_rhs) > 2:
                    B = f"X{counter}"
                    counter += 1
                    self.V.add(B)
                    new_P[B].append((new_rhs[0], new_rhs[1]))
                    new_rhs = [B] + new_rhs[2:]
                new_P[A].append(tuple(new_rhs))

    for t, v in new_vars.items():
        new_P[v] = [(t,)]

    self.P = new_P
```

7. normalize(self)

- **Purpose**: Performs all the necessary transformations to convert the CFG to CNF.
- **Process**:
    - It calls all the previous methods (eliminate_epsilon, eliminate_renaming, eliminate_inaccessible, eliminate_non_productive, and to_cnf) in sequence.

```python
def normalize(self):
    self.eliminate_epsilon()
    self.eliminate_renaming()
    self.eliminate_inaccessible()
    self.eliminate_non_productive()
    self.to_cnf()
```

8. print_grammar(self)

- **Purpose**: Prints the current state of the grammar.
- **Process**:
    - It prints the variables (non-terminals), terminals, start symbol, and all the production rules of the grammar.

```python
def print_grammar(self):
    print(f"Variables: {sorted(self.V)}")
    print(f"Terminals: {sorted(self.T)}")
    print(f"Start symbol: {self.S}")
    print("Productions:")
    for A in sorted(self.P):
        for prod in self.P[A]:
            print(f"  {A} → {''.join(prod)}")
```

9. get_var_for_terminal(t)

- **Purpose**: Generates a new variable for a terminal symbol if it has not already been assigned one.
- **Process**:
    - It checks if a terminal symbol has already been mapped to a new variable. If not, it generates a unique variable (e.g., T1, T2), adds it to the grammar, and creates a production rule for the terminal (e.g., T1 → a).

The grammar is then printed before and after the normalization process. The normalization applies the steps to eliminate ε-productions, unit productions, inaccessible symbols, non-productive symbols, and finally converts the grammar into CNF.

```
Before normalization:
Variables: ['A', 'B', 'C', 'D', 'S']
Terminals: ['a', 'b']
Start symbol: S
Productions:
  A -> aSab
  A -> BS
  A -> aA
  A -> b
  B -> BA
  B -> ababB
  B -> b
  B ->
  C -> AS
  S -> abAB
```

```
After normalization to CNF:
Variables: ['A', 'B', 'S', 'T1', 'T2', 'X1', 'X10', 'X11', 'X12', 'X13', 'X14', 'X15', 'X16', 'X17', 'X18', 'X2', 'X3', 'X4', 'X5', 'X6', 'X7', 'X8', 'X9']
Terminals: ['a', 'b']
Start symbol: S
Productions:
```

```
Productions:
  A -> X12T2
  A -> BS
  A -> T1A
  A -> b
  A -> X14B
  A -> X15A
  B -> BA
  B -> X3B
  B -> X5T2
  B -> b
  B -> X7T2
  B -> BS
  B -> T1A
  B -> X9B
  B -> X10A
  S -> X17B
  S -> X18A
  T1 -> a
  T2 -> b
  X1 -> T1T2
  X10 -> T1T2
  X11 -> T1S
  X12 -> X11T1
  X13 -> T1T2
  X14 -> X13A
  X15 -> T1T2
```

```
X14 -> X13A
X15 -> T1T2
X16 -> T1T2
X17 -> X16A
X18 -> T1T2
X2 -> X1T1
X3 -> X2T2
X4 -> T1T2
X5 -> X4T1
X5 -> X4T1
X6 -> T1S
X7 -> X6T1
X8 -> T1T2
X9 -> X8A
X5 -> X4T1
X6 -> T1S
X5 -> X4T1
X6 -> T1S
X7 -> X6T1
X5 -> X4T1
X5 -> X4T1
X6 -> T1S
X7 -> X6T1
X8 -> T1T2
X9 -> X8A
```

Conclusion:

Through this lab, I learned how to transform context-free grammars into Chomsky Normal Form (CNF), gaining an understanding of the steps required for this process, such as eliminating ε-productions and removing inaccessible symbols. I also became familiar with the different approaches to grammar normalization and how to implement them programmatically. By encapsulating the normalization logic in a method, I improved my ability to write clean, modular code. Additionally, I learned how to extend my solution to work with any grammar, not just a specific variant, which enhanced the flexibility and generality of my implementation.