Ministry of Education, Culture and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work *No. 5*

Formal Languages & Finite Automata

Elaborated by:
Gorcea Alexandrina, FAF-223
Verified by: Dumitru Crețu, univ. assist.

Chișinău, 2024

Theoretical considerations:

Chomsky Normal Form (CNF) stands as a significant milestone, named after the pioneering linguist Noam Chomsky. CNF provides a structured and simplified representation of context-free grammars (CFGs), facilitating efficient parsing and analysis in various computational applications. This essay delves into the fundamentals of CNF, its significance, and the process of transforming context-free grammars into this canonical form.

Chomsky Normal Form imposes strict rules on the structure of production rules within a context-free grammar. All production rules adhere to the format A -> BC or A -> a, where A, B, and C represent non-terminal symbols, and "a" signifies a terminal symbol. This standardized form eliminates ambiguity and streamlines parsing algorithms, making it easier to analyze the grammar's generative capacity.

The significance of Chomsky Normal Form transcends theoretical abstraction, finding practical application in various computational domains. In compiler design, CNF simplifies the parsing process, enhancing compiler efficiency and reliability. Natural language processing benefits from CNF's structured representation, aiding in syntactic analysis and grammar induction. Moreover, CNF serves as a foundational concept in computational biology, facilitating the modeling and analysis of biological sequences.

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.

i. The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).

ii. The implemented functionality needs executed and tested.

iii. A **BONUS point** will be given for the student who will have unit tests that validate the functionality of the project.

iv. 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.

Chomsky Normal Form Transformation Process:

1. Elimination of ε-productions: Remove rules producing the empty string, ensuring CNF compliance.
2. Removal of unit productions: Replace unit productions with equivalent rules to satisfy CNF requirements.
3. Introduction of new non-terminals: Introduce new non-terminal symbols as necessary to decompose rules into binary productions.
4. Conversion to binary productions: Transform rules into binary productions, adhering to the A -> BC format.
5. Simplification and optimization: Streamline the grammar by eliminating redundant rules and symbols while preserving language equivalence.

Implementation description:

My code defines a Python class Grammar to manipulate context-free grammars. It includes methods to eliminate ε-productions, unit productions, inaccessible symbols, and non-productive symbols, ultimately transforming the grammar into Chomsky Normal Form.

The **check_empty** function identifies and removes non-terminal symbols (states) that produce no terminal strings. It iterates through the productions, finding empty ones and eliminating them. If an empty production is the sole production of a state, that state is removed. Additionally, it adjusts other productions affected by this removal, updating the grammar accordingly. This process ensures the grammar remains well-defined and avoids potential issues during further transformations, such as those required for achieving Chomsky Normal Form.

```
def check_empty(self):
    states_to_remove = set()
    for key, value in self.P.items():
        if len(value) == 0:
            empty_state = key
            states_to_remove.add(key)
            for state, productions in self.P.items():
                change = False
                for production in productions:
                    if empty_state in production:
                        if len(production) == 1:
                            productions.remove(production)
                        else:
                            productions[productions.index(production)] = production.replace(empty_state, "")
                            change = True
                if change:
                    self.P[state] = list(set(productions))

    for state in states_to_remove:
        if state in self.P.keys():
            self.P.pop(state)
            self.Vn.remove(state)
```

The `**generate_combinations**` method creates new productions by removing occurrences of a given state within a production string. It initializes `**new_productions**` with the original production, then iterates over it. For each occurrence of the state, it generates two new strings: one by removing the first occurrence and another by removing the last occurrence. These strings are added to `**new_productions**`, ensuring no duplicates. Finally, the original production is removed from the list, and the updated list of new productions is returned. This method facilitates the elimination of ε-productions in the grammar transformation process.

Here's a more detailed breakdown of how the method works:

1. **Initialization**: It starts by creating a list **new_productions** containing the original production. This list will hold the new productions generated during the process.
2. **Iteration**: It iterates over each string **c** in the **new_productions** list.

3. **State Presence Check**: For each string **c**, it checks if the given state exists within it (**state in c**).
4. **Multiple Occurrences Handling**: If the state occurs more than once in the string (**sum(char == state for char in c) > 1**), it generates two new strings:

- One string (**str**) is created by removing the last occurrence of the state (**c[:c.rfind(state)] + c[c.rfind(state) + 1:]**).
- The other string is created by removing the first occurrence of the state (**c[:c.index(state)] + c[c.index(state) + 1:]**).

5. **Avoiding Duplicates**: It ensures that the generated strings are not already present in the **new_productions** list to avoid duplicates.
6. **Removal of Original Production**: After generating new productions, the original production is removed from the list.
7. **Return**: Finally, the list of new productions, excluding the original one, is returned.

```
def generate_combinations(self, production, state):
    new_productions = [production]
    for c in new_productions:
        if state in c:
            if sum(char == state for char in c) > 1:
                str = c[:c.rfind(state)] + c[c.rfind(state) + 1:]
                if str not in new_productions:
                    new_productions.append(str)
            str = c[:c.index(state)] + c[c.index(state) + 1:]
            if str not in new_productions:
                new_productions.append(str)
    new_productions.remove(production)
    return new_productions
```

The **eliminate_epsilon** method aims to remove ε-productions from a given context-free grammar. An ε-production is a production rule that derives the empty string ('). The method iterates over each state and its associated productions. If an ε-production is found for a state, it removes it and then generates new productions to replace occurrences of the epsilon state in other production rules.

1. **Identification of Epsilon State**: It initializes **epsilon_state** as an empty string and iterates over each state and its associated productions. If an ε-production is found (i.e., an empty string) within the productions of a state, the **epsilon_state** variable is updated to that state, and the empty string production is removed.
2. **Generation of New Productions**: After removing the ε-production, it iterates over all productions again to identify where the epsilon state occurs. For each non-empty production containing the epsilon state, it calls the **generate_combinations** method to generate new productions by removing occurrences of the epsilon state within the production.

3. **Updating Productions**: The new productions generated in the previous step are added to the original productions associated with the state.
4. **Checking for Empty States**: After eliminating ε-productions, it calls the **check_empty** method to remove any states that no longer have productions.
5. **Return**: Finally, it returns the updated productions.

```
def eliminate_epsilon(self):
    epsilon_state = ''
    for state, productions in self.P.items():
        if '' in productions:
            epsilon_state = state
            productions.remove('')
            for key, value in self.P.items():
                new_productions = []
                for production in value:
                    if epsilon_state in production and len(production) > 1:
                            new_productions += self.generate_combinations(production, epsilon_state)
                value += new_productions
    # print(self.P)
    self.check_empty()
    # print(self.P)
    return self.P
```

The **eliminate_unit** method serves to eliminate unit productions, which are production rules in a context-free grammar where a single non-terminal symbol directly produces another non-terminal symbol. It begins by identifying unit productions within the grammar and storing them in a dictionary. These unit productions are then removed from their respective states, and the associated non-terminal symbols they produce are incorporated into the original states' production lists. This process ensures that the grammar remains equivalent while eliminating unit productions.

After handling unit productions, the method checks for any states that have become empty due to the removal of unit productions and removes them to maintain a valid grammar structure. Finally, the method returns the updated productions. Through these steps, **eliminate_unit** effectively transforms the grammar to a form where unit productions are eliminated, facilitating further transformations or analysis of the grammar.

```
def eliminate_unit(self):
    unit_productions = {}
    for state, productions in self.P.items():
        for production in productions:
            if len(production) == 1 and production[0].isupper():
                productions.remove(production)
                unit_productions.setdefault(state, []).append(production)

    if not bool(unit_productions):
        return self.P
```

```
    for state in self.P.keys():
        if state in unit_productions.keys():
            for unit in unit_productions[state]:
                self.P[state].extend(self.P[unit])
                self.P[state] = list(set(self.P[state]))

    self.check_empty()
    # print(self.P)
    return self.P
```

The **eliminate_inaccessible** method is designed to remove symbols (states) in a context-free grammar that cannot be reached from the start symbol. It initializes two sets: **reachable_symbols**, which stores symbols that are reachable from the start symbol, and **pending_symbols**, which initially contains the start symbol. The method then iterates through the pending symbols, popping one symbol at a time and adding it to the set of reachable symbols. For each symbol popped, it explores its productions and adds any unseen non-terminal symbols to the pending symbols list.

Once the traversal is complete, the method identifies unreachable symbols by finding the set difference between all symbols in the grammar and the reachable symbols. Unreachable symbols are removed from the grammar along with their associated productions, and the method updates the grammar's production rules to remove any references to unreachable symbols. Finally, it checks for and removes any states that have become empty due to the elimination of unreachable symbols, ensuring the grammar remains well-defined. This method effectively prunes the grammar to include only symbols that are reachable from the start symbol, facilitating further analysis or transformations.

```
def eliminate_inaccessible(self):
    reachable_symbols = set()
    pending_symbols = [list(self.P.keys())[0]]

    while pending_symbols:
        symbol = pending_symbols.pop(0)
        reachable_symbols.add(symbol)
        for production in self.P[symbol]:
            for char in production:
                if char.isupper() and char not in reachable_symbols:
                    pending_symbols.append(char)

    unreachable_symbols = set(self.P.keys()) - reachable_symbols
    for symbol in unreachable_symbols:
        del self.P[symbol]
        self.Vn.remove(symbol)

    for symbol, productions in self.P.items():
        self.P[symbol] = [prod for prod in productions if all(char not in unreachable_symbols for char in prod)]
    # print(self.P)
    self.check_empty()
```

The **eliminate_non_productive** method is designed to remove non-productive symbols from a context-free grammar. Non-productive symbols are those that cannot generate any terminal strings.

The method works in the following way:

1. It initializes a set called **productive_symbols** to store symbols that are determined to be productive, and a boolean variable **changed** to track whether any changes were made during the iteration process.
2. The method iterates over each symbol in the grammar's productions. For each symbol, it checks if it's already marked as productive. If it is, the iteration continues to the next symbol. Otherwise, it examines the symbol's productions. If any production consists of a single lowercase string (indicating it's a terminal), the symbol is marked as productive, and the iteration continues to the next symbol.
3. If none of the productions are terminal strings, the method checks if all non-terminal symbols referenced in the productions are already marked as productive. If they are, the symbol is marked as productive, indicating that it can generate terminal strings.
4. After each iteration over the symbols, the method checks if all symbols in the grammar are now marked as productive. If they are, indicating that all symbols can generate terminal strings, the method returns the grammar unchanged. Otherwise, it removes non-productive symbols from the grammar by setting their production rules to empty lists.
5. It calls the **check_empty** method to remove any states that have become empty due to the elimination of non-productive symbols.
6. Finally, it returns the updated grammar.

```
def eliminate_non_productive(self):
    productive_symbols = set()
    changed = True
    while changed:
        changed = False
        for symbol, productions in self.P.items():
            if symbol in productive_symbols:
                continue
            if any(len(production) == 1 and production.islower() for production in self.P[symbol]):
                productive_symbols.update(symbol)
                changed = True
                continue
            non_terminals = set()
            if len(self.P[symbol]) == 1 and symbol in self.P[symbol][0]:
                continue
            for prod in self.P[symbol]:
                non_terminals.update(char for strings in prod for char in strings if char.isupper() and char != symbol)
            if all(non_terminal in productive_symbols for non_terminal in non_terminals):
                productive_symbols.update(symbol)
                changed = True
```

```
    if len(productive_symbols) == len(self.Vn):
        return self.P
    else:
        for symbol in self.Vn:
            if symbol not in productive_symbols:
                self.P[symbol] = []
    self.check_empty()
    return self.P
```

The **to_ChomskyNormalForm** method aims to transform a given context-free grammar into Chomsky Normal Form (CNF) by applying a series of grammar transformations. Here's how it accomplishes this:

1. It first eliminates ε-productions and unit productions using the **eliminate_epsilon** and **eliminate_unit** methods, respectively. This ensures that the grammar does not contain empty productions or productions with a single non-terminal symbol.
2. It then removes symbols (states) that are not reachable from the start symbol, ensuring that only reachable symbols are considered in the CNF transformation.
3. Next, it eliminates non-productive symbols, ensuring that only symbols capable of generating terminal strings are retained.
4. After eliminating non-productive symbols, it performs another round of elimination for ε-productions and unit productions to adjust the grammar further.
5. It converts the remaining productions into Chomsky Normal Form by replacing productions with more than two symbols. Each such production is decomposed into smaller productions, ensuring that each production either contains two non-terminals or a single terminal symbol.
6. During the CNF conversion, new non-terminal symbols are introduced as needed. These symbols are assigned from the set 'ABCDEFGHIJKLMNOPQRSTUVWXYZ', ensuring they are distinct from existing symbols in the grammar.
7. Finally, it updates the grammar with the new productions generated during the CNF conversion and returns the transformed grammar.

```
def to_ChomskyNormalForm(self):
    self.eliminate_epsilon()
    self.eliminate_unit()
    self.eliminate_inaccessible()
    self.eliminate_non_productive()
    self.eliminate_epsilon()
    self.eliminate_unit()
    # print(self.P)
    new_productions = {}
    symbols = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

```
for state, productions in self.P.items():
    new_strings = []
    for prod in productions:
        if len(prod) != 1 and not(len(prod) == 2 and prod[0].isupper() and prod[1].isupper()):
            for n in range(len(prod)):
                if prod[n].islower():
                    assigned_key = next((key for key, array in new_productions.items() if prod[n] in array), None)
                    if assigned_key is None:
                        while symbols[0] in self.Vn:
                            symbols = symbols.replace(symbols[0], '')
                        symbol = symbols[0]
                        self.Vn.add(symbol)
                        new_productions[symbol] = list(prod[n])
                    else:
                        symbol = assigned_key
                    prod = prod[:n] + symbol + prod[n + 1:]
```

```
            while len(prod) > 2:
                assigned_key = next((key for key, array in new_productions.items() if prod[0] + prod[1] in array), None)
                if assigned_key is None:
                    while symbols[0] in self.Vn:
                        symbols = symbols.replace(symbols[0], '')
                    symbol = symbols[0]
                    self.Vn.add(symbol)
                    str = prod[0] + prod[1]
                    new_productions[symbol] = [str]
                else:
                    symbol = assigned_key
                prod = prod.replace(prod[0] + prod[1], symbol)
        new_strings.append(prod)
    self.P[state] = new_strings

self.P.update(new_productions)
return self.P
```

The **print_grammar** method facilitates the display of the context-free grammar's productions in a readable format. It iterates through the grammar's key-value pairs, where each key represents a non-terminal symbol, and each associated value holds a list of productions for that symbol. For each non-terminal symbol, it prints the symbol followed by an arrow "->", and then proceeds to print each production. If there are multiple productions for a symbol, they are separated by the pipe symbol "|". This method organizes the grammar's productions in a structured manner, aiding comprehension and analysis of the grammar's structure and rules.

```
def print_grammar(self):
    print("P = {")
    for key, value in self.P.items():
        print("\t", key, "->", end=" ")
        for prod in value:
            if value.index(prod) != len(value) - 1:
                print(prod, end=" | ")
            else:
                print(prod)
    print("}")
```

The unit tests provided in the script aim to verify the correctness of the Chomsky Normal Form (CNF) transformation implemented in the **ChomskyNormalForm** module by comparing the output of the transformation against predefined variants of grammars stored in the **Variants** module.

Here is a example:

```python
class UnitTestCNF(unittest.TestCase):
    def test_ChomskyNormalForm_variant1(self):
        grammar_to_ChomskyNormalForm_test = ChomskyNormalForm.Grammar(Variants.Vn1, Variants.Vt1, Variants.P1).to_ChomskyNormalForm()
        dict_to_verify = {'S': ['AC', 'EC', 'BC', 'AS', 'a', 'FB', 'FD'],
                          'A': ['GS', 'EC', 'b', 'AS', 'BC', 'a', 'FD'],
                          'B': ['b', 'GS'],
                          'C': ['BA'],
                          'D': ['HC', 'FG'],
                          'E': ['AS'],
                          'F': ['a'],
                          'G': ['b'],
                          'H': ['FG']}
        for key in grammar_to_ChomskyNormalForm_test.keys():
            grammar_to_ChomskyNormalForm_test[key] = set(grammar_to_ChomskyNormalForm_test[key])
            dict_to_verify[key] = set(dict_to_verify[key])
        self.assertDictEqual(grammar_to_ChomskyNormalForm_test, dict_to_verify)
```

The tests are written using the **unittest** framework. Each test case defines a specific variant of a context-free grammar and applies the CNF transformation to it, expecting the resulting transformed grammar to match a predefined expected output. The test cases cover different scenarios and grammatical structures, ensuring the robustness and accuracy of the CNF transformation algorithm. This systematic approach helps ensure the correctness and reliability of the implemented transformation process.

Results (for variant 12):

```
P = {
     S -> a | BX | b | CX
     A -> a | BX | b | CX
     X -> X | a | BX | b | CX
     B -> a
     C -> b
}
```

Conclusion:

This laboratory was an engaging exploration of fundamental concepts in formal language theory and algorithmic transformations. The primary objective was to develop a Python module, `ChomskyNormalForm`, capable of converting context-free grammars into CNF while ensuring accuracy through comprehensive unit testing. This exercise provided invaluable insights into the structure of context-free grammars and the rigorous processes involved in their manipulation.

The implementation of the CNF conversion algorithm involved several crucial steps. First, epsilon (ε) productions and unit productions were eliminated to streamline the grammar's structure

and ensure adherence to CNF rules. Next, inaccessible and non-productive symbols were identified and removed, further refining the grammar. The CNF conversion itself required decomposing productions into smaller, more manageable units, typically consisting of two non-terminal symbols or a single terminal symbol. New non-terminal symbols were introduced as needed to accommodate this transformation.

Throughout the development process, unit tests played a pivotal role in verifying the correctness of the CNF transformation algorithm. These tests were meticulously designed to cover various grammatical structures and scenarios, ensuring robustness and reliability. By comparing the output of the transformation against predefined grammar variants stored in the `Variants` module, the unit tests effectively validated the implementation's accuracy and consistency.

Beyond technical proficiency, the laboratory exercise fostered critical thinking skills and problem-solving abilities. It required students to analyze grammar structures, devise transformation strategies, and translate these strategies into executable code. Additionally, the exercise underscored the importance of systematic testing in software development, emphasizing the significance of verifying functionality across diverse inputs and edge cases.

In conclusion, the laboratory exercise on CNF transformation provided a comprehensive learning experience encompassing theoretical concepts, algorithmic implementation, and software testing methodologies. It equipped students with a deeper understanding of context-free grammars, CNF rules, and the practical challenges associated with grammar manipulation. Moreover, it instilled valuable skills in Python programming, algorithm design, and test-driven development, preparing students for real-world applications in computational linguistics, compiler design, and related fields.