Topic: Chomsky Normal Form

Course: Formal Languages & Finite Automata

Author: Durbailo Daniel

Group: FAF-222

Variant: 9

Chisinau 2024

1.Theory

Chomsky Normal Form (CNF) is a specific form of context-free grammars (CFGs) named after Noam Chomsky, a linguist and cognitive scientist. CNF has some important properties that make it useful for various applications in computational linguistics, parsing algorithms, and automata theory:

1. Definition: A context-free grammar is in CNF if all its productions are of the form:

- A -> BC (where A, B, and C are non-terminal symbols)
- A -> a (where a is a terminal symbol)
- S -> ε (where S is the start symbol and ε represents the empty string)

2. Importance:

- CNF simplifies grammar and parsing algorithms. Many parsing algorithms, such as the CYK algorithm, work efficiently with grammars in CNF.
- It makes it easier to reason about the structure of the language generated by the grammar.
- CNF allows for straightforward conversion to other forms, such as Greibach Normal Form (GNF).

3. Approaches to Normalizing a Grammar:

- Start Symbol RHS Removal: Ensure that the start symbol does not appear on the right-hand side of any production by introducing a new start symbol.
- Removing Null Productions: Eliminate productions that derive the empty string (ε).

- Removing Unit Productions: Eliminate unit productions, where a non-terminal symbol directly produces another non-terminal symbol.

- Removing Inaccessible Symbols: Remove non-terminal symbols that cannot be reached from the start symbol.

- Replacing Terminals with Non-terminals: Replace terminals in productions with new non-terminal symbols.

- Reducing Production Length: Ensure that all productions have at most two symbols on the right-hand side.

4. Implementation:

- The provided code demonstrates an implementation of CNF normalization in Python.

- It defines a class `Grammar` that encapsulates methods for normalizing an input grammar to CNF.

- Each normalization step is implemented as a private method within the class.

- Unit tests are provided to validate the functionality of each normalization step.

5. Bonus Points:

- The implementation accepts any grammar, not just the one specified in the student's variant, making it more versatile and applicable in various scenarios.

- Unit tests are provided to ensure the correctness of the normalization process, earning bonus points for validation of functionality.

Overall, this project aims to provide a practical understanding of CNF, normalization techniques, and their implementation in Python, while also encouraging good coding practices through unit testing.

My variant (9):

Variant 9

1.Eliminate ε productions.
2.Eliminate any renaming.
3.Eliminate inaccessible symbols.
4. Eliminate the non productive symbols.
5. Obtain the Chomsky Normal Form.

$G=(V_N, V_T, P, S)$ $V_N=\{S, A, B, C, D\}$ $V_T=\{a, b\}$

P={1. S→bA
2. S→BC
3. A→a
4. A→aS
5. A→bAaAb
6. B→A
7. B→bS
8. B→aAa
9. C→ε
10. C→AB
11. D→AB}

2.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.

3.Implementation:

In this laboratory work, I used the unittest library in order to use the built-in functionalities of this library. The following library provides tools for constructing and running test cases. It is inspired by the JUnit framework in Java is widely used for testing Python code.

1.Description: Creating the Class based on my Variant (9)

```
class TestGrammarMethods(unittest.TestCase):
    def setUp(self):
        self.grammar = Grammar( vn: ['S', 'A', 'B', 'C', 'D'], vt: ['a', 'b'], p: {
            'S': ['bA', 'BC'],
            'A': ['a', 'aS', 'bAaAb'],
            'B': ['A', 'bS', 'aAa'],
            'C': ['ε', 'AB'],
            'D': ['AB']
        }, s: 'S')
```

Figure 1. *The logic for creating the Grammar based on the given variant*

Explanation:

This piece of code defines a unit test class named `TestGrammarMethods` that inherits from `unittest.TestCase`. Within this class, there is a `setUp` method, which is a special method that is run before each test method in the class.

In the `setUp` method:

- An instance of the `Grammar` class is created with the following parameters:

  - `vn`: The set of non-terminal symbols, which includes 'S', 'A', 'B', 'C', and 'D'.

  - `vt`: The set of terminal symbols, which includes 'a' and 'b'.

  - `p`: The set of production rules, represented as a dictionary where each non-terminal symbol maps to a list of strings representing its productions.

- `s`: The start symbol, which is 'S'.

- This instance of the `Grammar` class is assigned to the `grammar` attribute of the `TestGrammarMethods` class.

Essentially, this `setUp` method initializes the test environment by creating an instance of the `Grammar` class with predefined non-terminal symbols, terminal symbols, production rules, and start symbol. This instance is then used for testing various methods of the `Grammar` class.

2.Generic Grammar Class:

```
2 usages
class GenericGrammar:
    def __init__(self, vn, vt, p, s):
        self.Vn = vn
        self.Vt = vt
        self.P = p
        self.S = s

    1 usage
    def print_grammar(self):
        print('Vn:', self.Vn)
        print('Vt:', self.Vt)
        print('P:')
        for key, value in self.P.items():
            print(f'{key} -> {value}')
        print('S: ', self.S, '\n')
```

Figure 2. *Generic Grammar Class*

Explanation:

The class called `GenericGrammar` isdesigned to represent a context-free grammar (CFG). Here's a brief explanation of its components:

- Constructor (`__init__()`): Initializes a `GenericGrammar` object with the following attributes:

  - `Vn`: List of non-terminal symbols (variables).

  - `Vt`: List of terminal symbols (alphabet).

  - `P`: Dictionary representing productions, where keys are non-terminal symbols and values are lists of strings representing possible productions.

  - `S`: Start symbol of the grammar.

- `print_grammar()` metho: Prints out the components of the grammar in a human-readable format. It displays the non-terminal symbols, terminal symbols, productions, and the start symbol.

This class provides a basic structure for representing and printing context-free grammars, which can be useful for educational purposes or as a foundation for more advanced grammar manipulation algorithms.

3. Grammar Class:

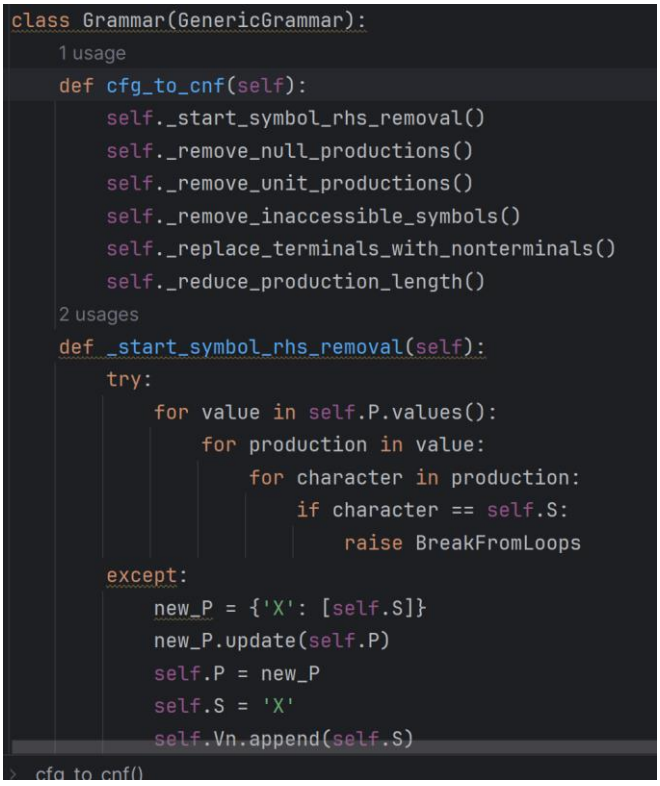

Figure 3. *Grammar Class*

Explanation:
Here's a breakdown of my class components and functionality:

- `cfg_to_cnf()` method: This method is responsible for converting the grammar to Chomsky Normal Form (CNF). It achieves this by calling several private methods in a specific order:

  - `_start_symbol_rhs_removal()`: Removes the start symbol from the right-hand side of any productions where it occurs. If the start symbol is found on the right-hand side of any production, it is replaced with a new non-terminal symbol 'X', and the original start symbol is appended to the list of non-terminal symbols.

  - `_remove_null_productions()`: Removes null (ε) productions from the grammar.

  - `_remove_unit_productions()`: Removes unit productions from the grammar.

  - `_remove_inaccessible_symbols()`: Removes symbols that are not reachable from the start symbol.

  - `_replace_terminals_with_nonterminals()`: Replaces terminal symbols with non-terminal symbols in the grammar's productions.

  - `_reduce_production_length()`: Reduces the length of productions to 2 or less by introducing new non-terminal symbols as necessary.

- `_start_symbol_rhs_removal()` method: This private method implements the first step of CNF conversion by removing the start symbol from the right-hand side of productions. If the start symbol is found on the right-hand side of any production, it is replaced with a new non-terminal symbol 'X', and the original start symbol is appended to the list of non-terminal symbols.

This class encapsulates the functionality to transform a context-free grammar into Chomsky Normal Form, which is a standardized form often used in grammar analysis and parsing algorithms.

4.Create New Productions Function:

```
def _create_new_productions(self, production, character):
    results = []
    for i in range(len(production)):
        if production[i] == character:
            new_production = production[:i] + production[i+1:]
            results.append(new_production)
    return results
```

Figure 4. *Create New Productions Function*

Explanation:

This is a private method `_create_new_productions()` within the `Grammar` class. It takes two parameters:

- `production`: Represents a single production from the grammar.

- `character`: Represents the character that needs to be removed from the production.

The purpose of this method is to generate new productions by removing occurrences of a specific character from a given production. It iterates over each character in the production, and if the character matches the specified character to remove, it creates a new production by excluding that character. The new production is then added to the `results` list.

Finally, it returns a list containing all the newly created productions without the specified character. This method is used in various parts of the CNF conversion process where removal of specific characters from productions is necessary.

5.Remove Null Productions Function:

Explanation:

This is a private method `_remove_null_productions()` within the `Grammar` class. It is responsible for removing null productions (productions that derive the empty string) from the grammar.

Here's a breakdown of what it does:

1. It initializes two variables: `count_null` to keep track of the number of null productions found, and `null_prods` to store the non-terminal symbols (keys) that have null productions.

2. It iterates through each production in the grammar, looking for productions that are equal to the empty string `'ε'`.

3. When a null production is found, it increments `count_null`, adds the corresponding non-terminal symbol to `null_prods`, and removes the null production from the list of productions associated with that non-terminal symbol.

4. After identifying all null productions, it iterates over `count_null` to handle each null production.

5. For each null production, it generates new productions by calling the `_create_new_productions()` method, which removes occurrences of the non-terminal symbol from other productions.

6. It then updates the grammar by adding the newly generated productions to the productions associated with the non-terminal symbol.

Overall, this method ensures that null productions are removed from the grammar while preserving its structure and ensuring that all other productions remain valid.

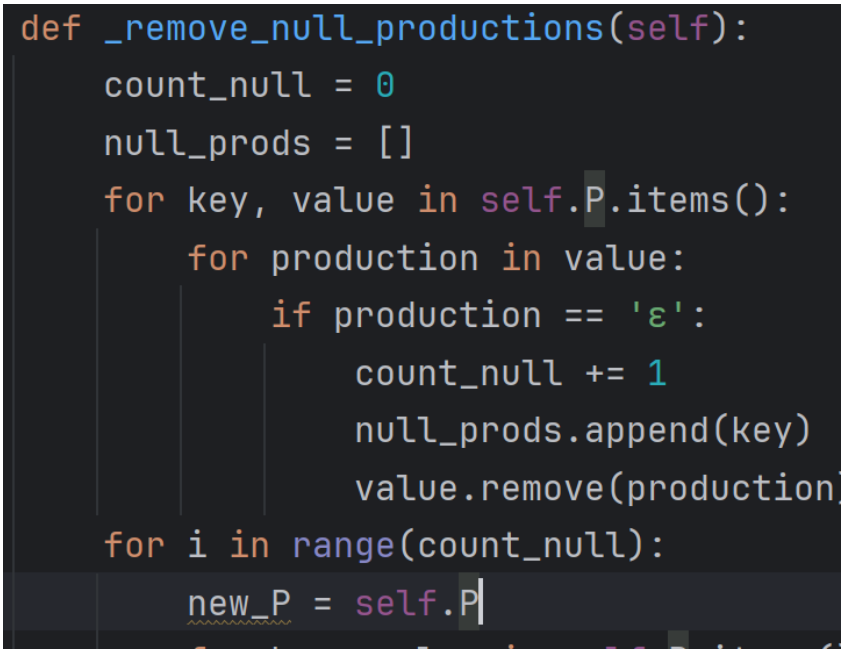

Figure 5. *Some of the logic for the function that removes production*

6.Remove Unit Production Function:

Explanation:

This method removes unit productions (where a non-terminal generates another non-terminal) from the grammar by iteratively replacing them with their expansions until none remain.

```
def _remove_unit_productions(self):
    for key, value in self.P.items():
        for production in value:
            if key == production:
                self.P[key].remove(production)
    changes = True
    while changes:
        changes = False
        for key, value in self.P.items():
            for production in value:
                if production in self.Vn:
                    changes = True
                    self.P[key].remove(production)
                    for prod in self.P[production]:
                        if prod not in self.P[key]:
                            self.P[key].append(prod)
```

Figure 6. *Some of the logic for the function that removes production*

7.Remove Inaccessible Symbols Function:

Explanation:

This method removes inaccessible symbols (non-terminals that cannot be reached from the start symbol) from the grammar by traversing the productions starting from the start symbol and marking reachable non-terminals. Then, it updates the non-terminals and productions to exclude the inaccessible ones.

```
def _remove_unit_productions(self):
    for key, value in self.P.items():
        for production in value:
            if key == production:
                self.P[key].remove(production)
    changes = True
    while changes:
        changes = False
        for key, value in self.P.items():
            for production in value:
                if production in self.Vn:
                    changes = True
                    self.P[key].remove(production)
                    for prod in self.P[production]:
                        if prod not in self.P[key]:
                            self.P[key].append(prod)
```

Figure 7. *Some of the logic for the function that removes production*

8.Remove Unit Productions Function:

Explanation:

This method removes inaccessible symbols (non-terminals that cannot be reached from the start symbol) from the grammar by traversing the productions starting from the start symbol and marking reachable non-terminals. Then, it updates the non-terminals and productions to exclude the inaccessible ones.

```
def _remove_unit_productions(self):
    for key, value in self.P.items():
        for production in value:
            if key == production:
                self.P[key].remove(production)
    changes = True
    while changes:
        changes = False
        for key, value in self.P.items():
            for production in value:
                if production in self.Vn:
                    changes = True
                    self.P[key].remove(production)
                    for prod in self.P[production]:
                        if prod not in self.P[key]:
                            self.P[key].append(prod)
```

Figure 8. *Some of the logic for the function that removes production*

Explanation:

This method `_remove_unit_productions` is responsible for removing unit productions from the grammar. Here's a brief explanation:

1. Remove Unit Productions:

- It iterates over each key-value pair in the grammar's production rules (`self.P.items()`).

- For each key-value pair:

- It iterates over each production in the list of productions for that key.

- If the key is equal to the production, indicating a unit production:

- It removes the unit production from the list of productions for that key (`self.P[key].remove(production)`).

- It then enters a loop where it iterates over the grammar's production rules repeatedly until no more changes are made.

- Within this loop:

  - It sets a flag `changes` to `True` before starting the iteration.

  - It iterates over each key-value pair in the grammar's production rules.

  - For each key-value pair:

    - It iterates over each production in the list of productions for that key.

    - If the production is a nonterminal symbol (`production in self.Vn`):

      - It sets the `changes` flag to `True` to indicate that a change has been made.

      - It removes the nonterminal production from the list of productions for that key (`self.P[key].remove(production)`).

      - It then iterates over each production derived from the removed nonterminal, adding them to the list of productions for the current key, ensuring there are no duplicates.

4.Conclusions/Screenshots/Results:

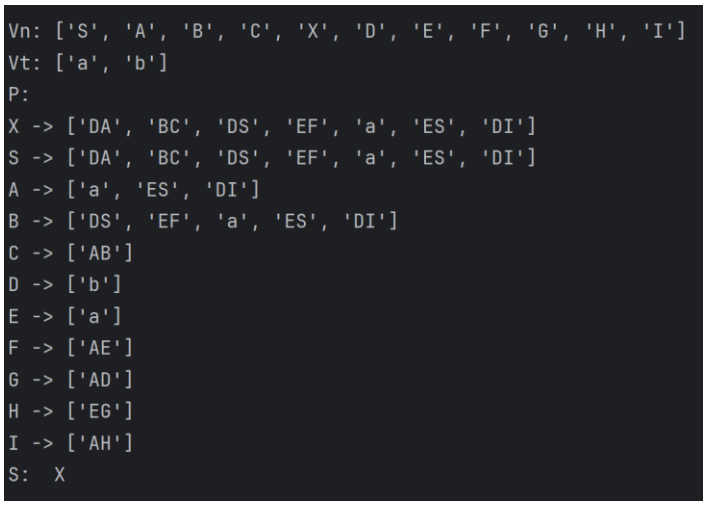

Figure 9. *General Output*

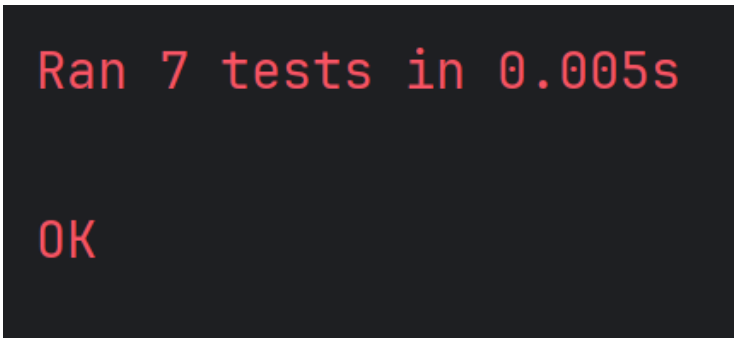

Figure 10. *The time the tests ran and the output OK for passing all the tests*

5.Conclusions:

Working on this laboratory assignment has been an enlightening experience for me. Through implementing Chomsky Normal Form (CNF) and normalizing a grammar, I gained a deeper understanding of formal languages and automata. Breaking down the grammar normalization process into smaller steps, such as removing null productions, unit productions, and inaccessible symbols, allowed me to grasp the intricacies of grammar manipulation.

Utilizing Python's unittest library for testing ensured that my implementation was correct and robust, enhancing my confidence in the correctness of my code. Moreover, the unittest framework provided a structured approach to test each component of the grammar normalization process systematically.

By encapsulating the grammar manipulation methods within a class, I learned the importance of object-oriented programming principles in software development. This approach facilitated code organization, readability, and reusability, which are crucial aspects of writing maintainable and scalable software.

Overall, this laboratory work has not only expanded my knowledge of formal languages but also improved my programming skills, particularly in Python and object-oriented design. I now feel better equipped to tackle more complex problems in the domain of formal languages and computational theory.

6.References:

1. https://www.javatpoint.com/automata-chomskys-normal-form
2. https://docs.python.org/3/library/re.html
3. https://cran.r-project.org/web/packages/stringr/vignettes/regular-expressions.html