Laboratory Work Nr.5

Chomsky Normal Form

Course: Formal Languages & Finite Automata

Author: Tofan Liviu

Group: FAF-223

Chişinău – 2024

Theory

In formal language theory, a context-free grammar, G, is said to be in **Chomsky normal form** (first described by Noam Chomsky) if all of its production rules are of the form:

$A \rightarrow BC$,   or

$A \rightarrow a$,   or

$S \rightarrow \varepsilon$,

Where A, B, and C are nonterminal symbols, the letter *a* is a terminal symbol (a symbol that represents a constant value), *S* is the start symbol, and *ε* denotes the empty string. Also, neither B nor C may be the start symbol, and the third production rule can only appear if ε is in L(G), the language produced by the context-free grammar G. Every grammar in Chomsky normal form is context-free, and conversely, every context-free grammar can be transformed into an equivalent one which is in Chomsky normal form and has a size no larger than the square of the original grammar's size.

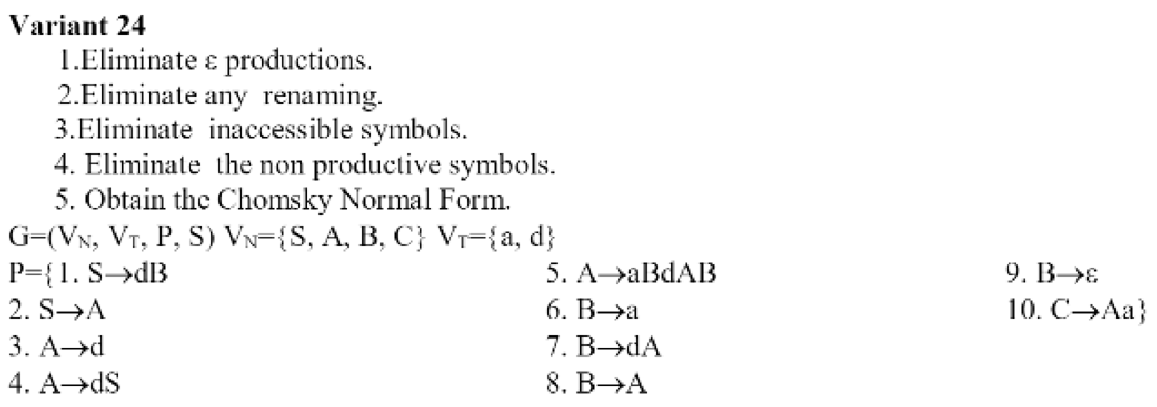
To convert a grammar to Chomsky normal form, a sequence of simple transformations is applied in a certain order; this is described in most textbooks on automata theory. The presentation here follows Hopcroft, Ullman (1979), but is adapted to use the transformation names from Lange, Leiß (2009). Each of the following transformations establishes one of the properties required for Chomsky normal form.

Objectives

1. Learn about Chomsky Normal Form (CNF)
2. Get familiar with the approaches of normalizing a grammar
3. Implement a method for normalizing an input grammar by the rules of CNF:
    a. The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).
    b. The implemented functionality needs executed and tested.
    c. A BONUS point will be given for the student who will have unit tests that validate the functionality of the project.
    d. 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

Variant 24:



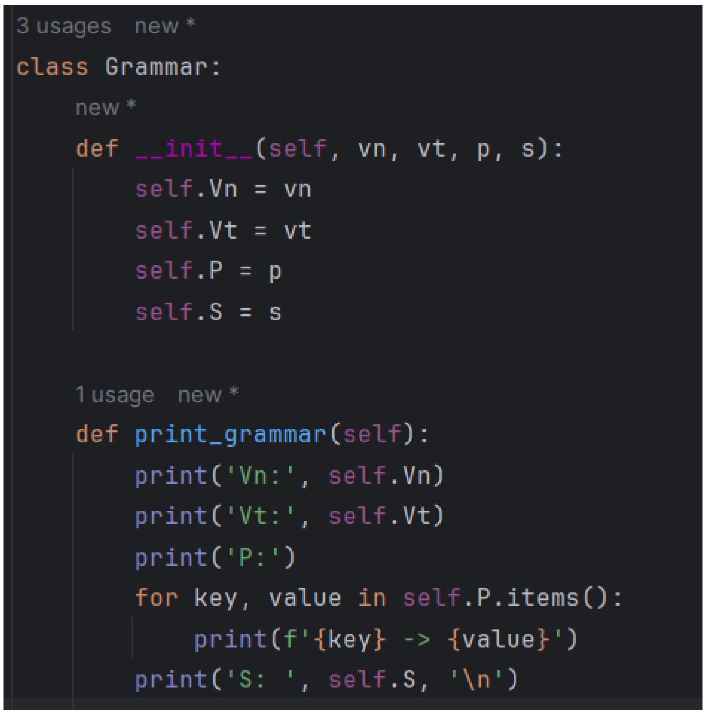
Variant 24

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

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

P={ 1. S→dB
2. S→A
3. A→d
4. A→dS
5. A→aBdAB
6. B→a
7. B→dA
8. B→A
9. B→ε
10. C→Aa}

First I created class Grammar, that represents a context-free grammar. It is initialized with parameters Vn (non-terminal symbols), Vt (terminal symbols), P (production rules), and S (start symbol). The class includes a method print_grammar() to print out the details of the grammar in a readable format. Overall, it encapsulates the structure and behavior of a context-free grammar for further manipulation.

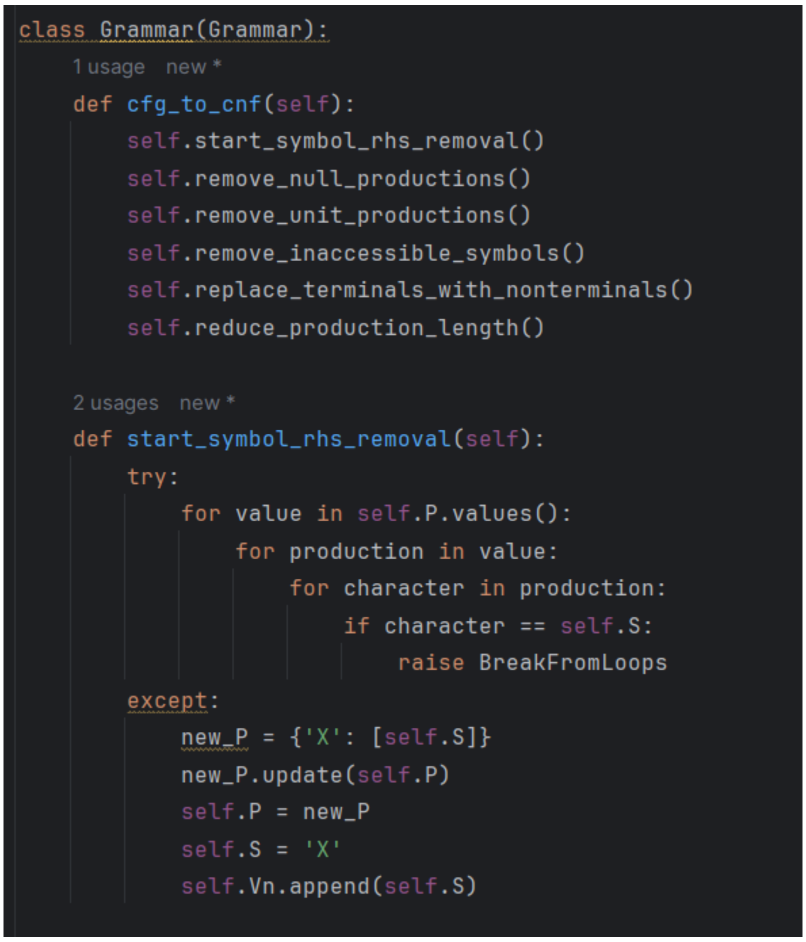


```
3 usages   new *
class Grammar:
    new *
    def __init__(self, vn, vt, p, s):
        self.Vn = vn
        self.Vt = vt
        self.P = p
        self.S = s

    1 usage   new *
    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')
```

1. *Grammar Class initialization and print_grammar() function*

The Grammar class inherits from itself (overridden), suggesting an extension or refinement of its functionality. The cfg_to_cnf() method, within this class, is used to convert a context-free grammar (CFG) into Chomsky Normal Form (CNF). It orchestrates several transformations, starting with the removal of the start symbol from right-hand sides of productions. This ensures adherence to CNF rules, enabling efficient parsing and analysis.



```python
class Grammar(Grammar):
    1 usage   new *
    def cfg_to_cnf(self):
        self.start_symbol_rhs_removal()
        self.remove_null_productions()
        self.remove_unit_productions()
        self.remove_inaccessible_symbols()
        self.replace_terminals_with_nonterminals()
        self.reduce_production_length()

    2 usages   new *
    def start_symbol_rhs_removal(self):
        try:
            for value in self.P.values():
                for production in value:
                    for character in production:
                        if character == self.S:
                            raise BreakFromLoops
        except:
            new_P = {'X': [self.S]}
            new_P.update(self.P)
            self.P = new_P
            self.S = 'X'
            self.Vn.append(self.S)
```

2. *Overridden Grammar Class initialization and cfg_to_cnf() function*

After I have defined multiple function inside grammar to convert from Context Free Gammar to Chomsky normal form. Below I wrote functions explication shortly:

```python
def start_symbol_rhs_removal(self):
```

The `start_symbol_rhs_removal()` function in the program ensures that the start symbol doesn't appear on the right-hand side of any production rules. If the start symbol is found in such a position, it gets replaced with a new non-terminal symbol 'X', and the grammar's attributes are adjusted accordingly to maintain consistency and adhere to Chomsky Normal Form (CNF) rules.

```
def create_new_productions(self, production, character):
```

The `create_new_productions()` method generates new productions by removing occurrences of a specified character from a given production. It iterates through each character in the production, and if it matches the specified character, it constructs a new production by excluding that character. The resulting list contains all possible productions formed by removing the specified character from different positions within the original production.

```
def remove_null_productions(self):
```

The `remove_null_productions()` method eliminates null productions from the grammar. It iterates through the productions, detecting any occurrences of the null symbol 'ε'. For each null production found, it removes 'ε' from the production and updates the grammar accordingly. Then, it generates new productions to replace instances where the null symbol appears, ensuring that all affected non-terminal symbols are correctly accounted for in the updated grammar.

```
 def remove_unit_productions(self):
```

The `remove_unit_productions()` method targets the removal of unit productions from the grammar. It initially eliminates any unit productions where a non-terminal symbol directly produces itself. Then, it iterates through the grammar, identifying and replacing indirect unit productions with their corresponding non-unit productions. This process continues iteratively until no further changes are detected.

By systematically addressing unit productions, this method ensures that the resulting grammar is free from such redundant structures, contributing to its clarity and compliance with standard grammar forms.

```
def remove_inaccessible_symbols(self):
```

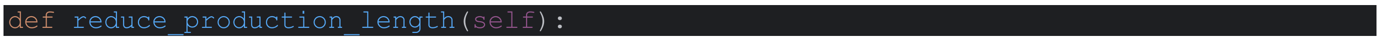
The `remove_inaccessible_symbols()` method eliminates inaccessible symbols from the grammar. Starting from the start symbol, it traverses through productions to identify reachable symbols, using a breadth-first search approach. Once all reachable symbols are determined, it updates the set of non-terminal symbols (`Vn`) and production rules (`P`) to exclude inaccessible symbols.

By systematically pruning inaccessible symbols, this method ensures that the resulting grammar only contains symbols that can be reached from the start symbol, enhancing the efficiency and clarity of subsequent grammatical operations.

```
def replace_terminals_with_nonterminals(self):
```

The `replace_terminals_with_nonterminals()` method substitutes terminal symbols with new non-terminal symbols in the grammar. It iterates through the production rules, replacing terminal symbols with corresponding non-terminal symbols. Each terminal symbol is mapped to a unique non-terminal symbol, and the productions are updated accordingly.

By performing this transformation, the method ensures that all productions consist of non-terminal symbols only, simplifying subsequent grammar manipulation and analysis.

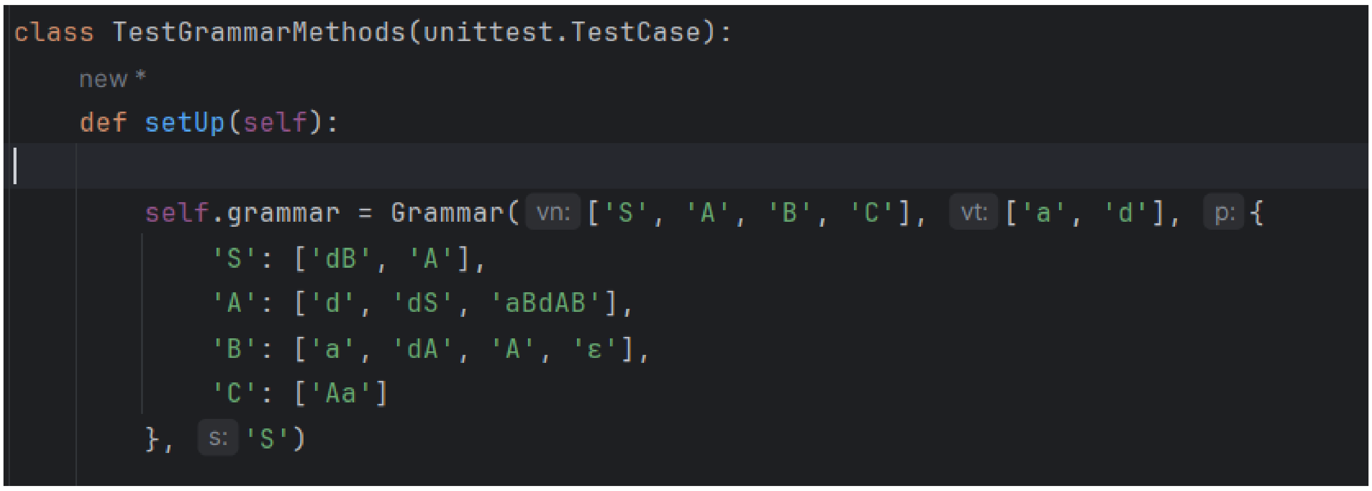


```
def reduce_production_length(self):
```

The `reduce_production_length()` method aims to reduce the length of productions in the grammar by introducing new non-terminal symbols for binary productions. It iterates through each production rule, and if a production has a length greater than 2, it replaces pairs of adjacent symbols with a new non-terminal symbol. This process continues until all productions are of length 2 or less.

By converting longer productions into binary productions, this method simplifies the grammar structure, which can enhance parsing efficiency and facilitate further grammatical analysis.

After creating a separate class for tests to ensure the code works correctly, several tests were defined to validate different methods of the `Grammar` class:



```
class TestGrammarMethods(unittest.TestCase):
    new *
    def setUp(self):

        self.grammar = Grammar( vn: ['S', 'A', 'B', 'C'],  vt: ['a', 'd'],  p: {
            'S': ['dB', 'A'],
            'A': ['d', 'dS', 'aBdAB'],
            'B': ['a', 'dA', 'A', 'ε'],
            'C': ['Aa']
        },  s: 'S')
```

3. *TestGrammarMethods Class initialization*

1. `test_start_symbol_rhs_removal`: This test checks if the start symbol is removed from the right-hand side of any production rules after applying the `start_symbol_rhs_removal()` method.

2. `test_remove_null_productions`: This test verifies that null productions ('ε') are removed from the grammar after executing the `remove_null_productions()` method.

3. `test_remove_unit_productions`: This test ensures that unit productions are eliminated from the grammar by confirming that no production consists of only one non-terminal symbol after invoking the `remove_unit_productions()` method.

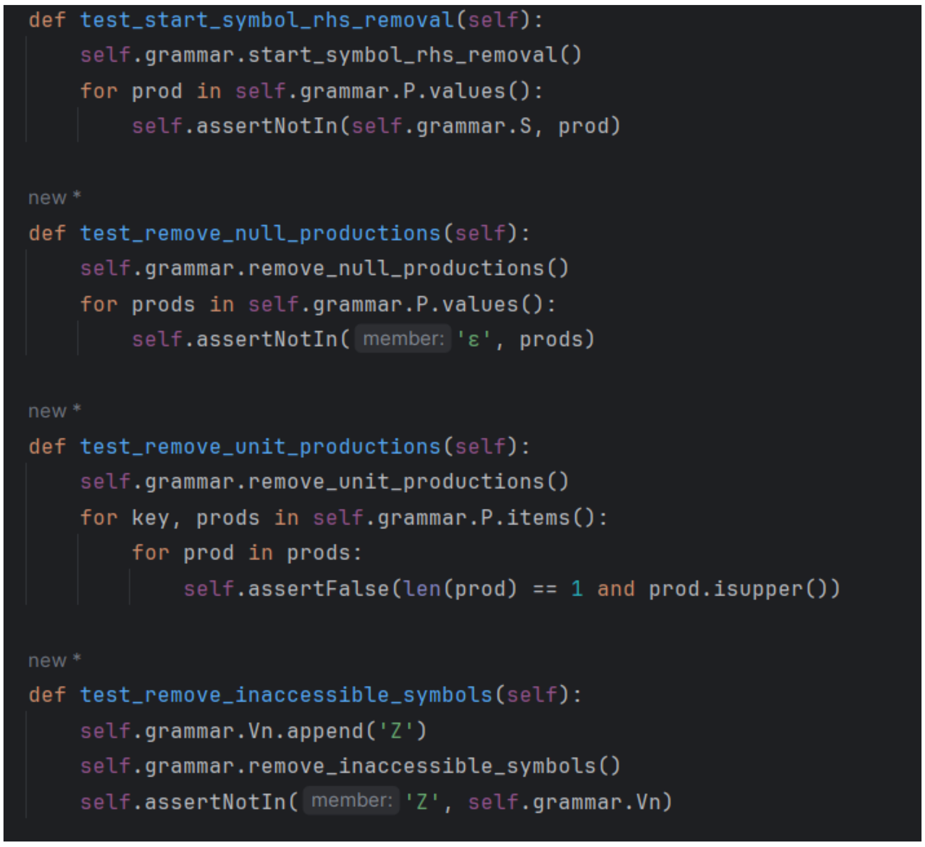
4. `test_remove_inaccessible_symbols`: This test validates the removal of inaccessible symbols from the grammar, ensuring that any symbol that cannot be reached from the start symbol is excluded from the grammar.

5. `test_replace_terminals_with_nonterminals`: This test checks if terminal symbols are replaced with non-terminal symbols in the grammar after executing the `replace_terminals_with_nonterminals()` method.

6. `test_reduce_production_length`: This test confirms that the production length is reduced to at most 2 symbols per production after applying the `reduce_production_length()` method.

7. `test_grammar_print`: This test ensures that the grammar is correctly converted to Chomsky Normal Form (CNF) and prints out the resulting grammar.

By defining these tests, the functionality and correctness of the `Grammar` class methods are thoroughly verified, contributing to the reliability and robustness of the codebase.



```
def test_start_symbol_rhs_removal(self):
    self.grammar.start_symbol_rhs_removal()
    for prod in self.grammar.P.values():
        self.assertNotIn(self.grammar.S, prod)

new *
def test_remove_null_productions(self):
    self.grammar.remove_null_productions()
    for prods in self.grammar.P.values():
        self.assertNotIn( member: 'ε', prods)

new *
def test_remove_unit_productions(self):
    self.grammar.remove_unit_productions()
    for key, prods in self.grammar.P.items():
        for prod in prods:
            self.assertFalse(len(prod) == 1 and prod.isupper())

new *
def test_remove_inaccessible_symbols(self):
    self.grammar.Vn.append('Z')
    self.grammar.remove_inaccessible_symbols()
    self.assertNotIn( member: 'Z', self.grammar.Vn)
```

3. *Some Methods using to TEST work*

Conclusion

This laboratory work was dedicated to the development and testing of methods for transforming context-free grammars (CFGs) into Chomsky Normal Form (CNF). The code encompasses a wide range of transformations necessary for this conversion process, including the removal of null productions, unit productions, and inaccessible symbols, as well as the replacement of terminals with non-terminals, and reduction of production length to at most 2 symbols per production.

By implementing these transformations, the code ensures that the resulting grammar adheres to CNF standards, which facilitates subsequent parsing and grammatical analysis tasks. Each method has been carefully designed and tested using the `unittest` framework, verifying its functionality and correctness under various scenarios.

Overall, this laboratory work exemplifies a structured and systematic approach to CFG manipulation and testing. The code's general design and well-structured implementation make it adaptable to handle different grammar variants effectively. It serves as a robust tool for processing and analyzing CFGs, contributing to the advancement of grammatical studies and computational linguistics.