Topic: Regular expressions

Course: Formal Languages & Finite Automata

Author: Grosu Renat

Theory:

Chomsky Normal Form (CNF) is a significant concept in theoretical computer science, particularly in the study of formal grammars within the context of computational linguistics and the development of algorithms for parsing natural languages. It is a way of simplifying the grammar of a context-free language so that it can be more easily analyzed and processed by algorithms, such as those used in parsing and in the construction of parse trees. The theory of CNF is essential for understanding and implementing algorithms that handle context-free languages, which are more complex than regular languages.

Context-Free Grammars: Chomsky Normal Form is applicable to context-free grammars (CFGs), which are used to define context-free languages. These grammars consist of a set of production rules that describe how strings in the language can be generated from a start symbol by replacing symbols with groups of other symbols. CFGs are more powerful than regular grammars and can describe languages capable of nesting structures, such as the balanced parentheses problem.

Form of Rules in CNF: A grammar is in Chomsky Normal Form if all of its production rules are of the form:

$A \rightarrow BC$

$A \rightarrow a$

$S \rightarrow \epsilon$ (where S is the start symbol and ϵ represents the empty string, allowed only if the empty string is part of the language)

Here, A, B, and C are non-terminal symbols, and a is a terminal symbol. This restriction simplifies the structure of the grammar without reducing its power to express any context-free language.

Simplification of Parsing: Transforming a CFG into CNF can simplify the design of parsing algorithms, such as the CYK algorithm, which determines whether a given string can be generated by a grammar. A CNF-based grammar ensures that every production expands in a uniform way, facilitating a more straightforward and systematic approach to parsing.

Conversion to CNF: Converting an arbitrary CFG to CNF involves several steps:

- Removing useless symbols (symbols that do not generate any string of terminal symbols or are unreachable from the start symbol).
- Eliminating ε-productions (productions that generate an empty string) except for the start symbol.
- Eliminating unit productions (productions where a single non-terminal generates another non-terminal).
- Ensuring that all remaining productions meet the strict form requirements of CNF.

Algorithmic Implications: The transformation of grammars into CNF is central in theoretical computer science because it impacts the efficiency and feasibility of algorithms for language recognition and parsing. This form allows algorithms to run in polynomial time, which is crucial for applications in compilers and natural language processing systems.

Applications:

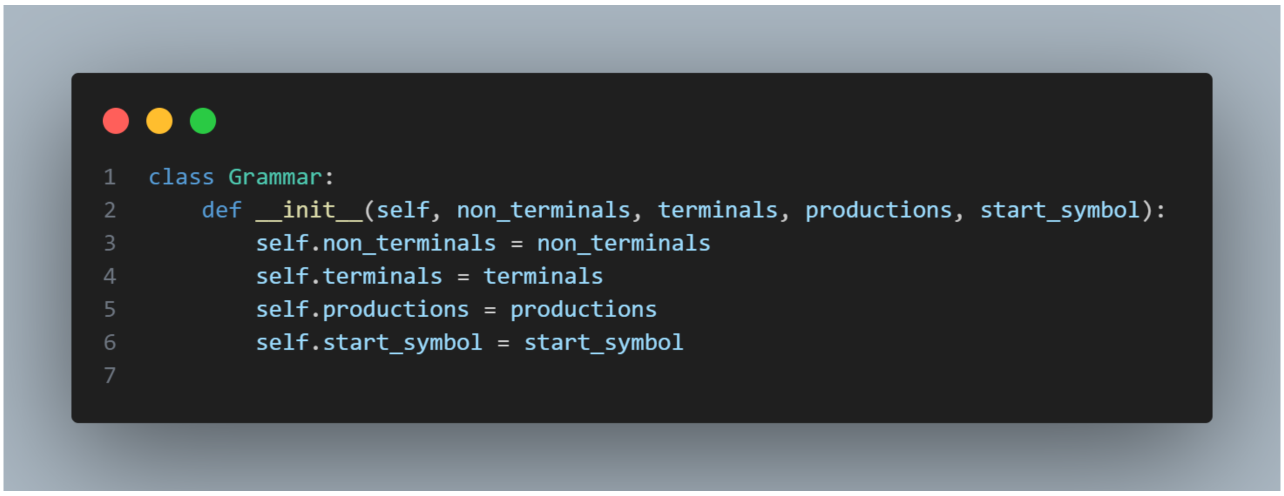
- Parsing Algorithms: CNF is primarily used in parsing algorithms, like the CYK algorithm, which can efficiently parse strings and construct parse trees for context-free languages when the grammar is in CNF.
- Compiler Design: In compiler construction, transforming the grammar of a programming language into CNF can facilitate the development of efficient syntax analysis components.
- Natural Language Processing: CNF assists in the structural analysis of natural languages, aiding in tasks such as parsing complex sentence structures, which are common in human languages.
- Educational Tools: CNF is also used in educational contexts to teach the fundamental concepts of grammar transformations and their implications for language processing.

In summary, Chomsky Normal Form is a fundamental concept in the field of formal languages and automata theory. It plays a critical role in simplifying context-free grammars to enhance the efficiency of parsing algorithms, making it a cornerstone in the areas of computational linguistics, compiler construction, and broader applications in computer science.

Implementation description:

1) Grammar Class

Class Structure and Initialization

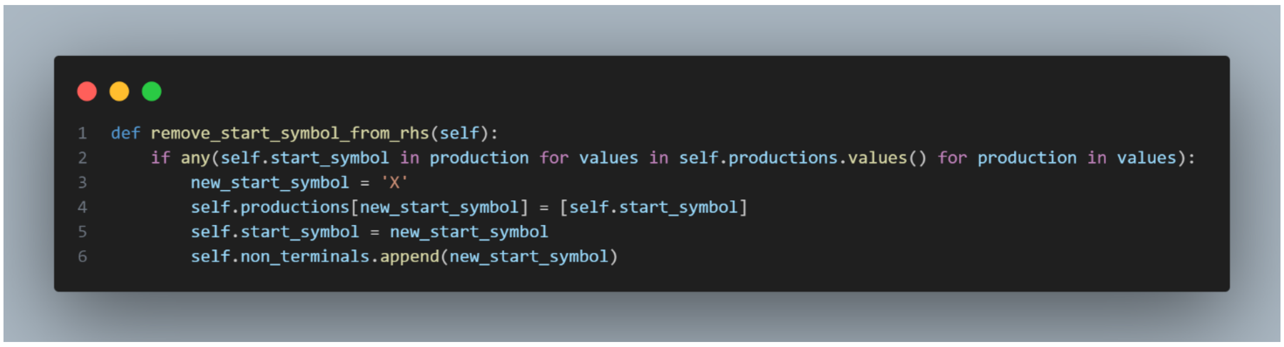


```python
class Grammar:
    def __init__(self, non_terminals, terminals, productions, start_symbol):
        self.non_terminals = non_terminals
        self.terminals = terminals
        self.productions = productions
        self.start_symbol = start_symbol

```

Class Definition: The Grammar class is designed to manage and manipulate grammars represented by non-terminals, terminals, production rules, and a start symbol.

Constructor: Initializes the grammar with sets of non-terminals, terminals, production rules, and the start symbol.

Start Symbol Removal from RHS Method:



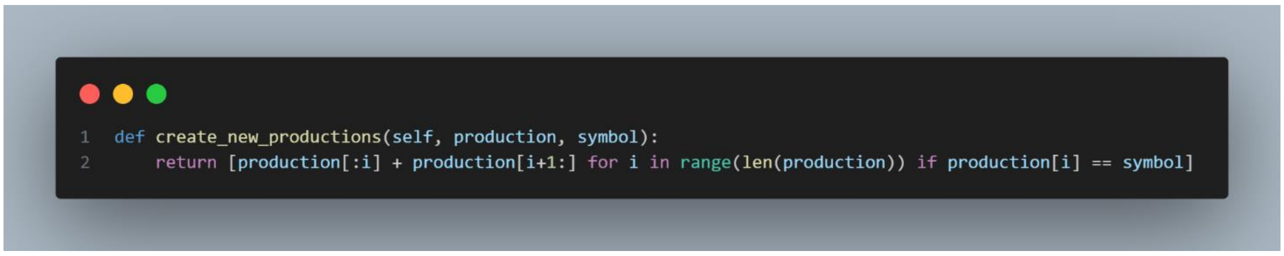
```python
def remove_start_symbol_from_rhs(self):
    if any(self.start_symbol in production for values in self.productions.values() for production in values):
        new_start_symbol = 'X'
        self.productions[new_start_symbol] = [self.start_symbol]
        self.start_symbol = new_start_symbol
        self.non_terminals.append(new_start_symbol)
```

This method is designed to ensure that the start symbol of the grammar does not appear on the right-hand side (RHS) of any production rules, which is a requirement for CNF. If the start symbol does appear on the RHS, the method proceeds as follows:

- It introduces a new start symbol (e.g., 'X').
- It adds a new production rule where this new start symbol produces the old start symbol.
- It updates the grammar's start symbol to the new one.

- It appends the new start symbol to the list of non-terminals.

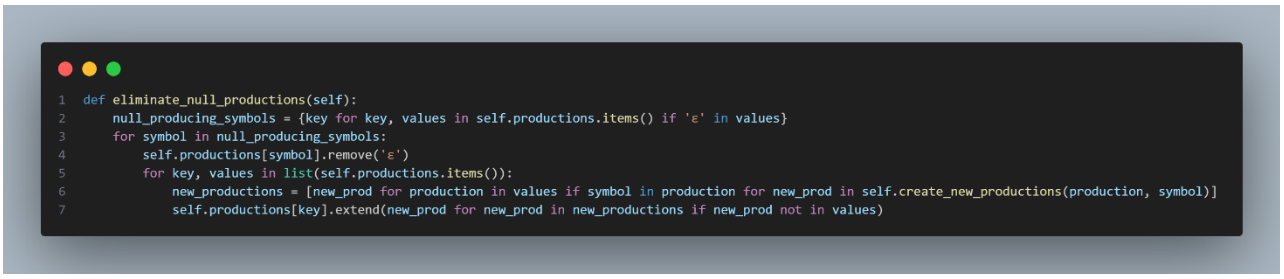
Create New Productions Method:



```python
def create_new_productions(self, production, symbol):
    return [production[:i] + production[i+1:] for i in range(len(production)) if production[i] == symbol]
```

This method is a helper function used during the elimination of null productions. It takes a production and a symbol as arguments and returns a list of new productions, each created by removing one occurrence of the symbol from the original production. This method is critical when dealing with nullable symbols in the grammar.

Null Productions Elimination Method:

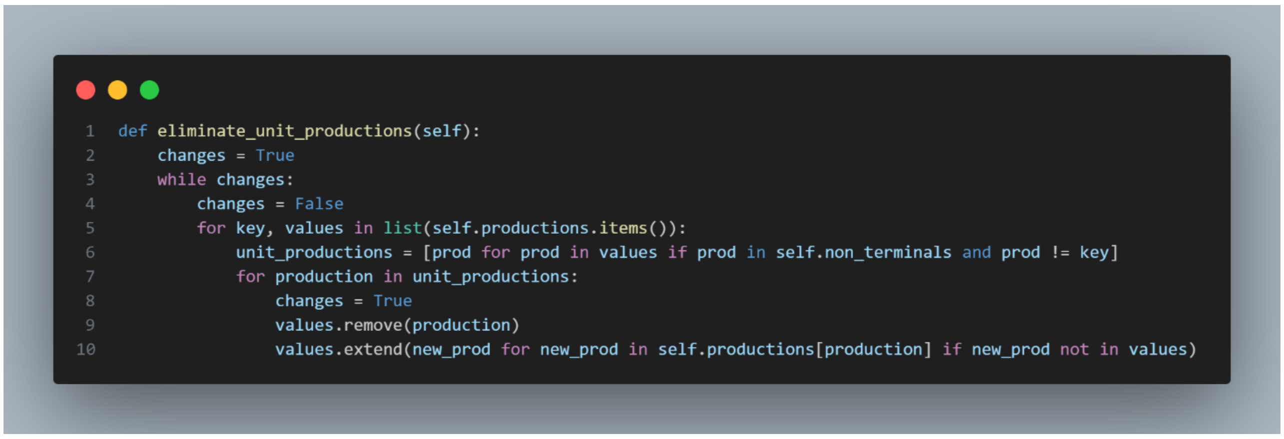


```python
def eliminate_null_productions(self):
    null_producing_symbols = {key for key, values in self.productions.items() if 'ε' in values}
    for symbol in null_producing_symbols:
        self.productions[symbol].remove('ε')
        for key, values in list(self.productions.items()):
            new_productions = [new_prod for production in values if symbol in production for new_prod in self.create_new_productions(production, symbol)]
            self.productions[key].extend(new_prod for new_prod in new_productions if new_prod not in values)
```

The eliminate_null_productions method identifies and eliminates ε-productions (productions that generate the empty string 'ε'). This is how it operates:

- It first identifies all null-producing symbols (non-terminals that can produce 'ε').
- It removes the 'ε' production from those symbols.
- It then iterates over all productions in the grammar.
- For each production that contains a null-producing symbol, it uses the create_new_productions method to generate all possible combinations of the production with and without the nullable symbol.
- It then extends the existing productions with these new combinations, avoiding the inclusion of duplicates.

Unit Productions Elimination Method:

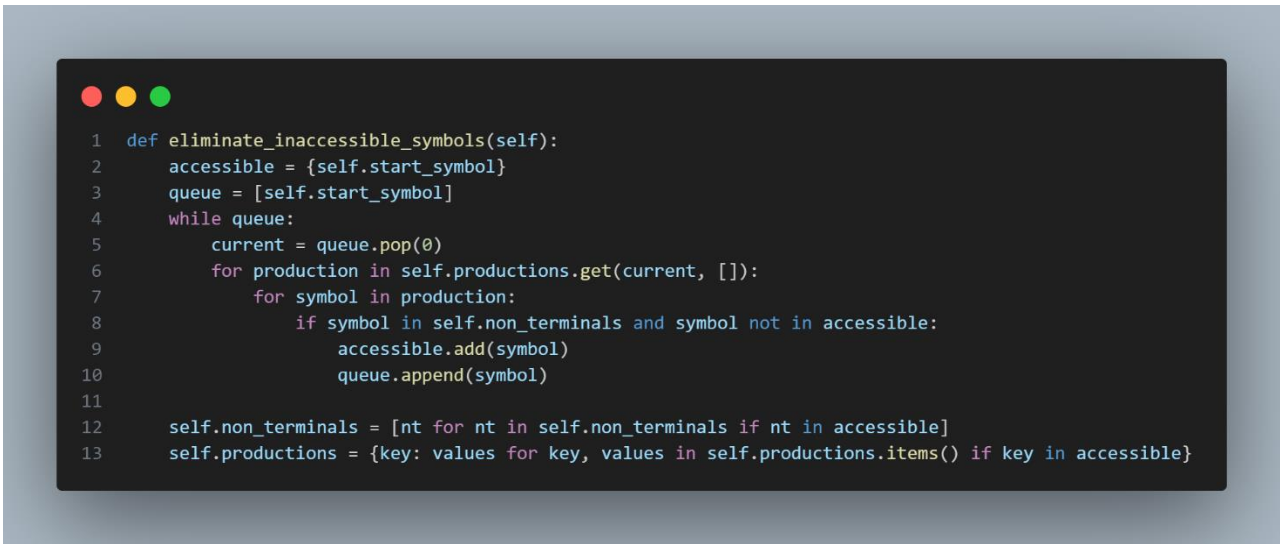


```python
def eliminate_unit_productions(self):
    changes = True
    while changes:
        changes = False
        for key, values in list(self.productions.items()):
            unit_productions = [prod for prod in values if prod in self.non_terminals and prod != key]
            for production in unit_productions:
                changes = True
                values.remove(production)
                values.extend(new_prod for new_prod in self.productions[production] if new_prod not in values)
```

Unit productions are productions where a non-terminal leads to a single non-terminal (e.g., A ⟶ B). The eliminate_unit_productions method removes these by:

- Initializing a flag to track changes called changes and setting it to True.
- It enters a loop that continues as long as changes are being made.
- Within the loop, it resets the changes flag to False.
- It then iterates over all production rules.
- For each rule, it identifies unit productions.
- For each unit production, it removes the unit production from the rule and extends the rule with the productions of the non-terminal it pointed to.
- If any changes are made (i.e., if unit productions are found and replaced), the changes flag is set to True to continue the loop.

Inaccessible Symbols Elimination Method:



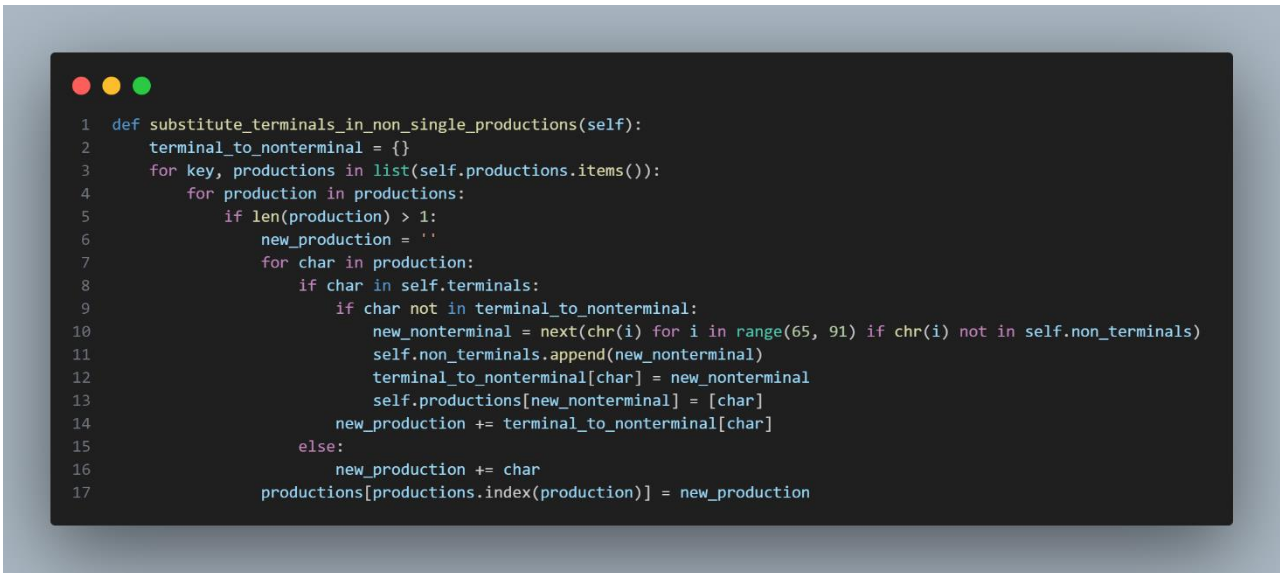
```python
def eliminate_inaccessible_symbols(self):
    accessible = {self.start_symbol}
    queue = [self.start_symbol]
    while queue:
        current = queue.pop(0)
        for production in self.productions.get(current, []):
            for symbol in production:
                if symbol in self.non_terminals and symbol not in accessible:
                    accessible.add(symbol)
                    queue.append(symbol)

    self.non_terminals = [nt for nt in self.non_terminals if nt in accessible]
    self.productions = {key: values for key, values in self.productions.items() if key in accessible}
```

This function removes symbols that are not accessible from the start symbol. Here's how it works:

- Initialize a set accessible with the start symbol. This set will contain all symbols that can be reached from the start symbol.
- Initialize a queue with the start symbol. This queue will be used for a breadth-first search of the grammar.
- Use a while loop to iterate until the queue is empty.
- In each iteration, remove the first symbol from the queue (current).
- For each production of the current symbol, iterate through each symbol in the production.
- If the symbol is a non-terminal and not yet in accessible, add it to accessible and append it to the queue for further exploration.
- After the while loop, filter the non_terminals list and productions dictionary to only include those that are accessible.

This ensures that the grammar only includes symbols that are reachable and can contribute to deriving strings in the language.

Terminals in Non-Single Productions Substitution Method:

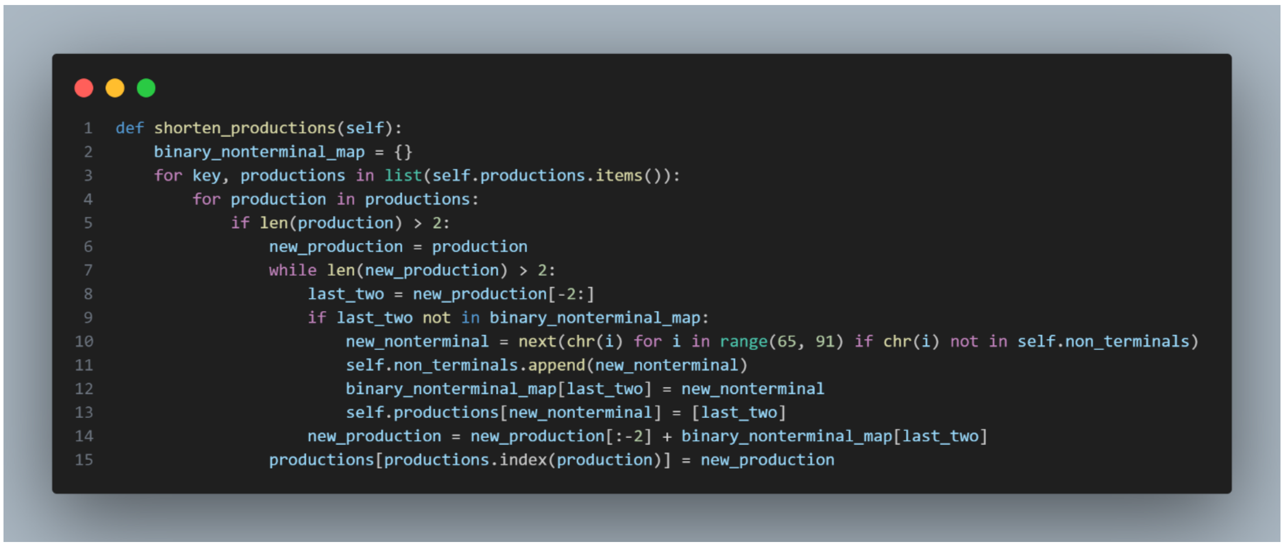


```python
def substitute_terminals_in_non_single_productions(self):
    terminal_to_nonterminal = {}
    for key, productions in list(self.productions.items()):
        for production in productions:
            if len(production) > 1:
                new_production = ''
                for char in production:
                    if char in self.terminals:
                        if char not in terminal_to_nonterminal:
                            new_nonterminal = next(chr(i) for i in range(65, 91) if chr(i) not in self.non_terminals)
                            self.non_terminals.append(new_nonterminal)
                            terminal_to_nonterminal[char] = new_nonterminal
                            self.productions[new_nonterminal] = [char]
                        new_production += terminal_to_nonterminal[char]
                    else:
                        new_production += char
                productions[productions.index(production)] = new_production
```

This method's purpose is to ensure that terminals appear only in productions where they are the sole right-hand side symbol or in binary productions with one non-terminal, following the CNF rules. Here's the step-by-step:

- Create a mapping dictionary terminal_to_nonterminal to track which new non-terminals have been introduced for terminals.
- Iterate over all productions in the grammar.
- For each production that has more than one symbol, construct a new production string (new_production).
- For each character in the production, if the character is a terminal, check if it already has an associated non-terminal.
- If not, create a new non-terminal, update the grammar, and add it to the mapping dictionary.
- Replace the terminal with the new non-terminal in new_production.
- If the character is not a terminal, just append it to new_production.
- Update the original production with new_production in the productions dictionary.

Productions Shortening Method:

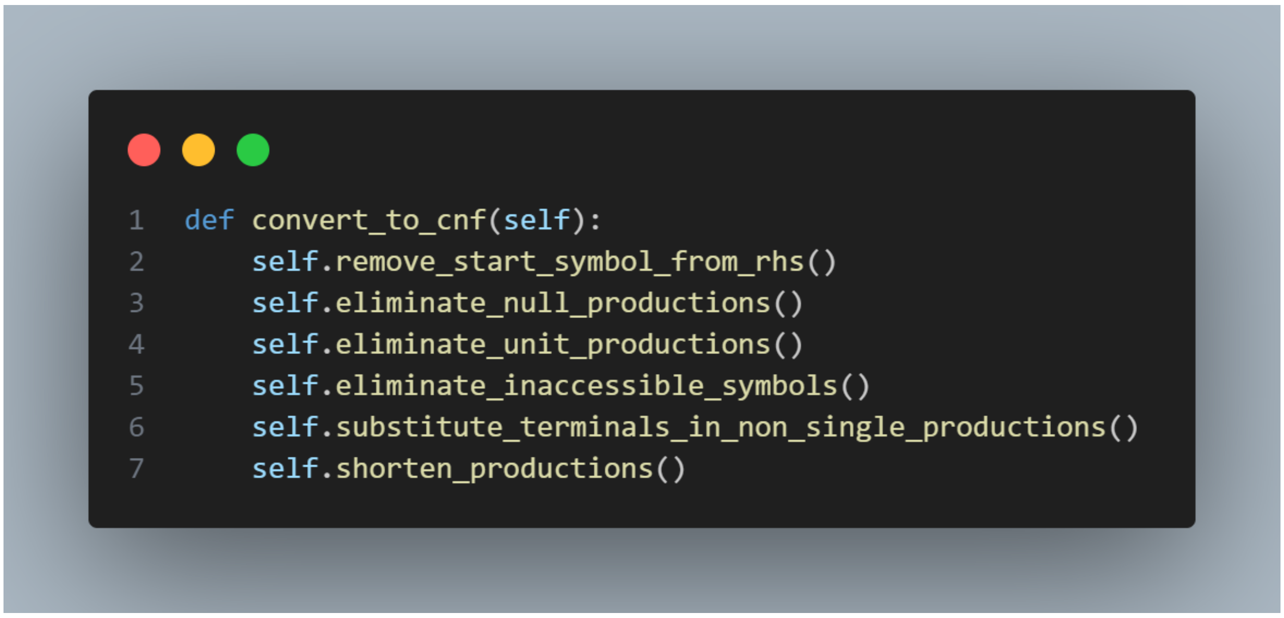


```python
def shorten_productions(self):
    binary_nonterminal_map = {}
    for key, productions in list(self.productions.items()):
        for production in productions:
            if len(production) > 2:
                new_production = production
                while len(new_production) > 2:
                    last_two = new_production[-2:]
                    if last_two not in binary_nonterminal_map:
                        new_nonterminal = next(chr(i) for i in range(65, 91) if chr(i) not in self.non_terminals)
                        self.non_terminals.append(new_nonterminal)
                        binary_nonterminal_map[last_two] = new_nonterminal
                        self.productions[new_nonterminal] = [last_two]
                    new_production = new_production[:-2] + binary_nonterminal_map[last_two]
                productions[productions.index(production)] = new_production
```

This method's purpose is to transform productions that are longer than two symbols into a series of binary productions, which is a requirement for Chomsky Normal Form (CNF). Here's how it works:

- binary_nonterminal_map is a dictionary to keep track of new non-terminals created during the shortening process.
- The method iterates over each production for each non-terminal.
- If the length of a production is more than two symbols, it must be shortened. The method enters a while loop to deal with this.
- Inside the loop, it checks the last two symbols of the current production (last_two).
- If this pair of symbols does not already have a corresponding non-terminal, it creates a new non-terminal and updates the binary_nonterminal_map with this new pairing.
- This new non-terminal is added to the grammar, and its production is set to the last two symbols of the current production.
- The current production is then updated to replace the last two symbols with the new non-terminal.
- This process repeats until all productions are binary (length of two symbols).

CFG to CNF Method:



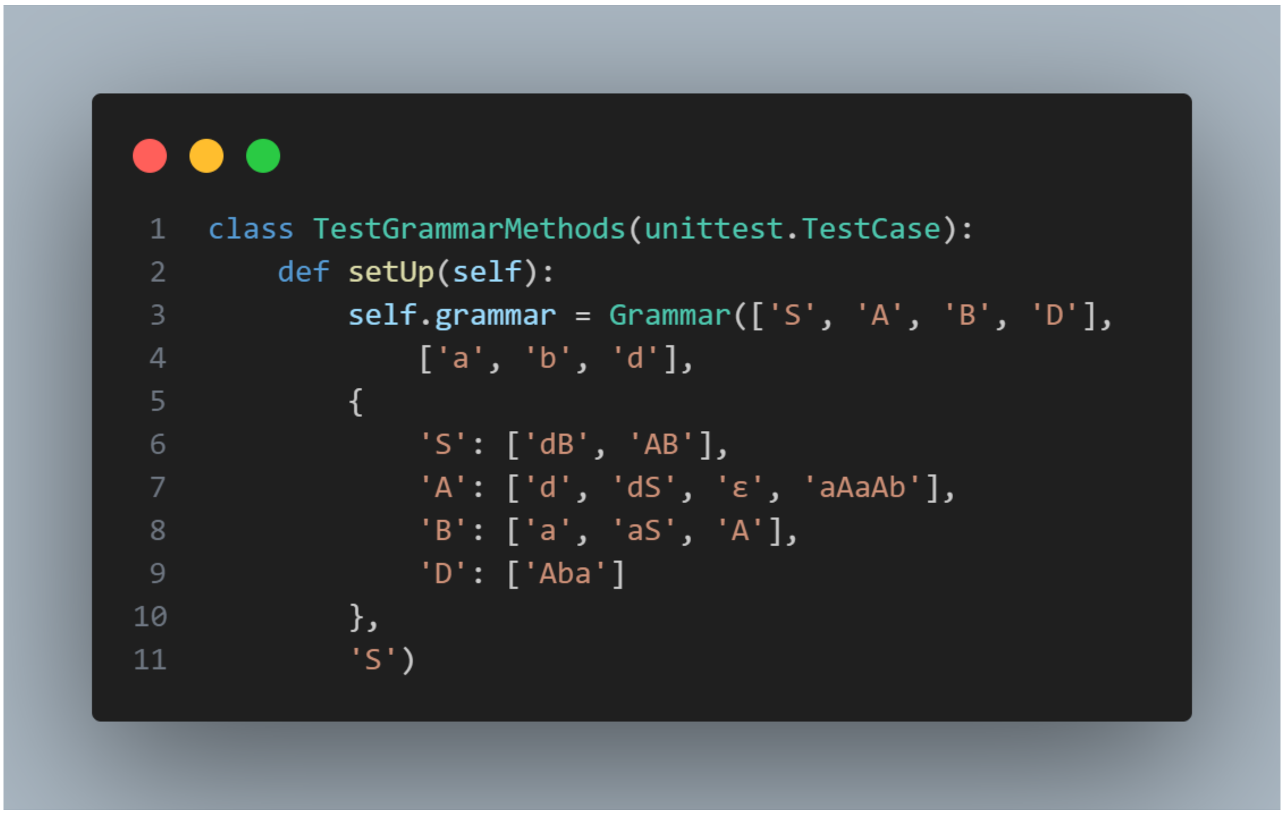
```python
def convert_to_cnf(self):
    self.remove_start_symbol_from_rhs()
    self.eliminate_null_productions()
    self.eliminate_unit_productions()
    self.eliminate_inaccessible_symbols()
    self.substitute_terminals_in_non_single_productions()
    self.shorten_productions()
```

The convert_to_cnf method is the main function that orchestrates the transformation of the entire grammar into CNF. It does so by calling each of the transformation methods in the required order:

- remove_start_symbol_from_rhs to ensure the start symbol doesn't appear on the right side of any production.
- eliminate_null_productions to remove ε-productions, with adjustments made for other productions that may depend on these.
- eliminate_unit_productions to replace unit productions with the actual productions of the non-terminals they point to.
- eliminate_inaccessible_symbols to remove any non-terminals and productions that cannot be reached from the start symbol.
- substitute_terminals_in_non_single_productions to ensure terminals only appear in productions of length two.
- shorten_productions to reduce all productions to binary form.

2) Test Grammar Class

setUp Method:

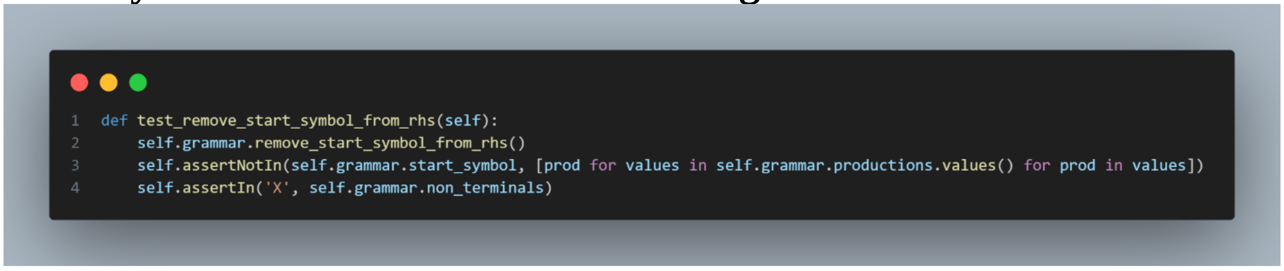


```python
class TestGrammarMethods(unittest.TestCase):
    def setUp(self):
        self.grammar = Grammar(['S', 'A', 'B', 'D'],
            ['a', 'b', 'd'],
        {
            'S': ['dB', 'AB'],
            'A': ['d', 'dS', 'ε', 'aAaAb'],
            'B': ['a', 'aS', 'A'],
            'D': ['Aba']
        },
        'S')
```

This method sets up the conditions for the tests:

- It initializes an instance of the Grammar class with specific non-terminals ('S', 'A', 'B', 'D'), terminals ('a', 'b', 'd'), and a set of productions for each non-terminal.
- The 'S' symbol is designated as the start symbol of the grammar.

Start Symbol Removal from RHS Testing Method:



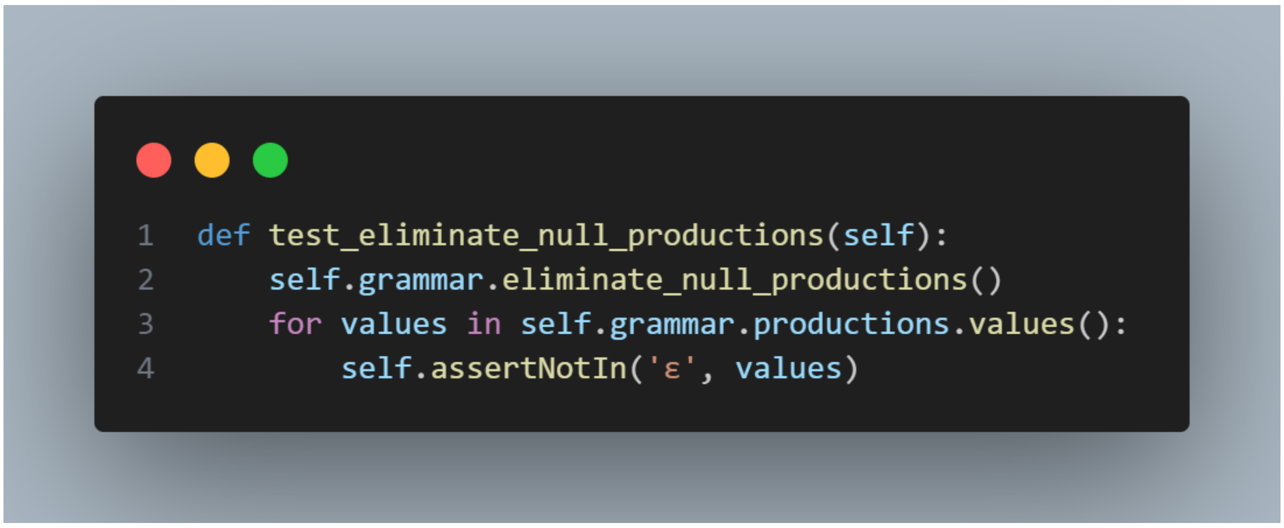
```python
def test_remove_start_symbol_from_rhs(self):
    self.grammar.remove_start_symbol_from_rhs()
    self.assertNotIn(self.grammar.start_symbol, [prod for values in self.grammar.productions.values() for prod in values])
    self.assertIn('X', self.grammar.non_terminals)
```

This test checks the remove_start_symbol_from_rhs method of the Grammar class by asserting two conditions:

- The start symbol 'S' should not be on the right-hand side of any productions after the method is executed.

- A new non-terminal 'X' should be added to the non-terminals list of the grammar.

Null Production Elimination Testing Method:

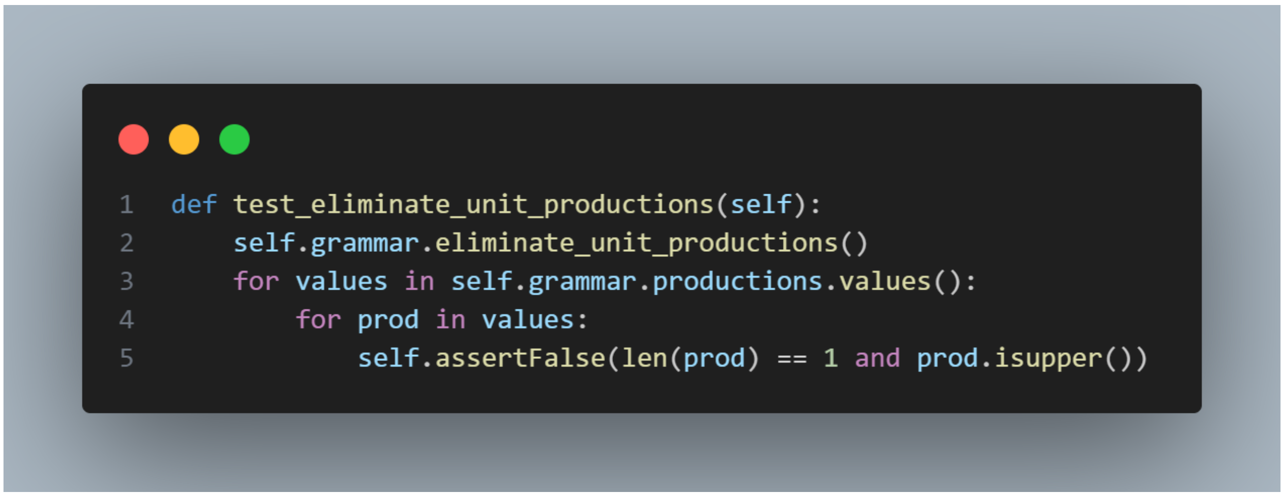


```python
def test_eliminate_null_productions(self):
    self.grammar.eliminate_null_productions()
    for values in self.grammar.productions.values():
        self.assertNotIn('ε', values)
```

This test verifies the functionality of the eliminate_null_productions method by ensuring:

- After the method is executed, the null character 'ε' does not appear in any of the productions, implying that all null productions have been successfully removed.

Unit Production Elimination Testing Method:

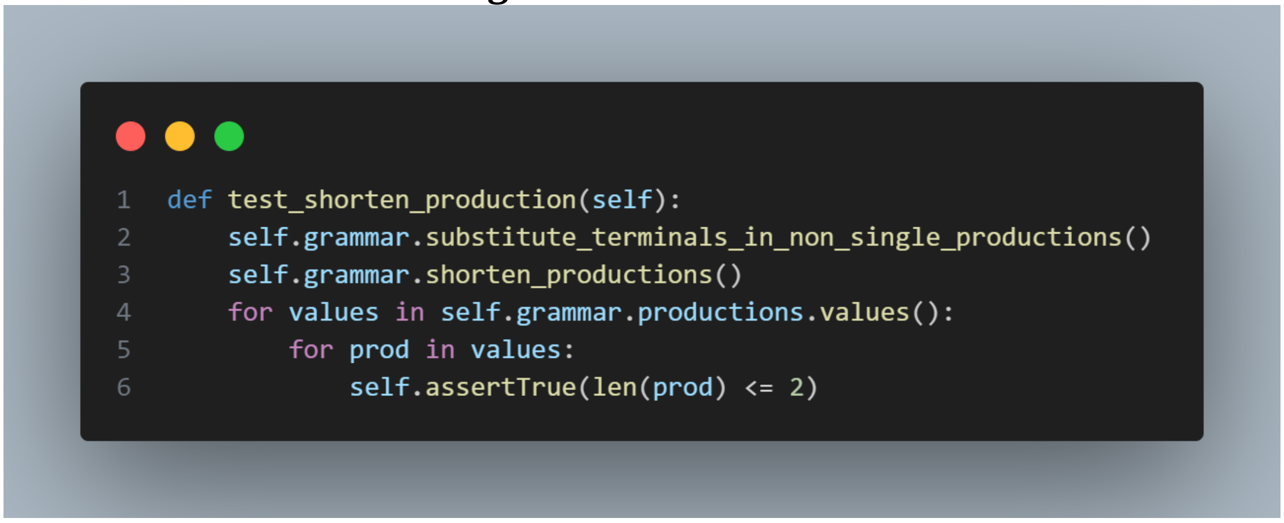


```python
def test_eliminate_unit_productions(self):
    self.grammar.eliminate_unit_productions()
    for values in self.grammar.productions.values():
        for prod in values:
            self.assertFalse(len(prod) == 1 and prod.isupper())
```

The test checks the eliminate_unit_productions method by asserting that:

- After running the method, there are no unit productions left, meaning no production rule should consist of a single non-terminal symbol only.

Shorten Production Testing Method:



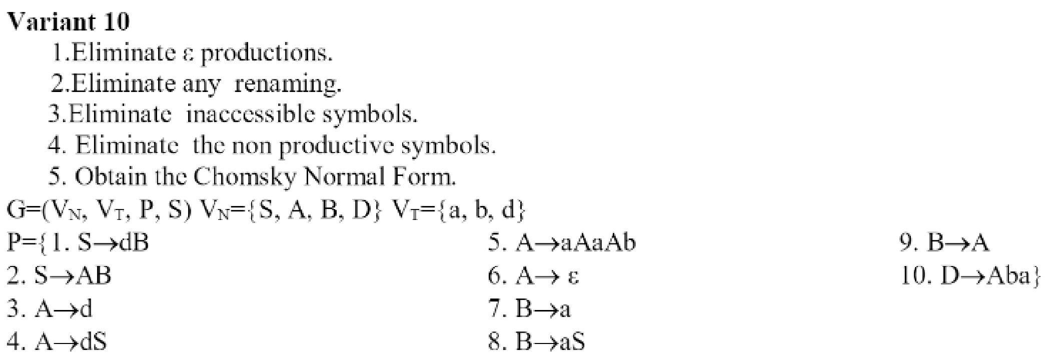
```python
def test_shorten_production(self):
    self.grammar.substitute_terminals_in_non_single_productions()
    self.grammar.shorten_productions()
    for values in self.grammar.productions.values():
        for prod in values:
            self.assertTrue(len(prod) <= 2)
```

This test ensures the shorten_productions method works as expected by:

- Confirming that after the method's execution, all production rules have been reduced to at most two symbols, which aligns with the requirement for CNF where each production rule must have either two non-terminals or a single terminal on the right-hand side.

Conclusions/Screenshots/Results:

My variant:



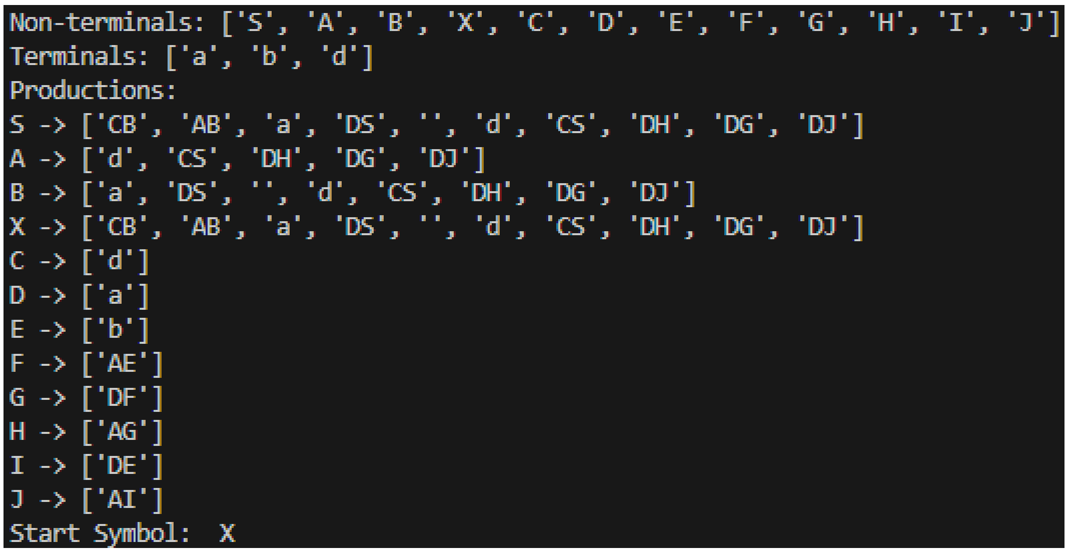
Variant 10

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

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

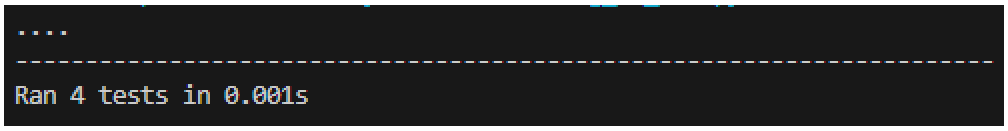
P={1. S→dB	5. A→aAaAb	9. B→A
2. S→AB	6. A→ ε	10. D→Aba}
3. A→d	7. B→a	
4. A→dS	8. B→aS	

This is the output of the CFG to CNF function for the example from the 10th variant:



```
Non-terminals: ['S', 'A', 'B', 'X', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
Terminals: ['a', 'b', 'd']
Productions:
S -> ['CB', 'AB', 'a', 'DS', '', 'd', 'CS', 'DH', 'DG', 'DJ']
A -> ['d', 'CS', 'DH', 'DG', 'DJ']
B -> ['a', 'DS', '', 'd', 'CS', 'DH', 'DG', 'DJ']
X -> ['CB', 'AB', 'a', 'DS', '', 'd', 'CS', 'DH', 'DG', 'DJ']
C -> ['d']
D -> ['a']
E -> ['b']
F -> ['AE']
G -> ['DF']
H -> ['AG']
I -> ['DE']
J -> ['AI']
Start Symbol:  X
```

This is the output of the unit tests made in the TestGrammarMethods class:



```
....
----------------------------------------------------------------------
Ran 4 tests in 0.001s
```

In this lab, I got hands-on experience with Chomsky Normal Form, and it was eye-opening to see how such a theoretical concept is essential for parsing and working with grammars. Turning abstract rules into a practical form that a computer can work with wasn't just educational—it was also a lot of fun.

I learned that attention to detail is crucial when transforming grammars, especially with tricky parts like null and unit productions. Running tests to make sure each step was done right showed me how important it is to check your work as you go.

Overall, this lab was super useful. It made me better at understanding and applying the theories behind programming languages and gave me a real sense of achievement when I saw the grammar transformations working correctly.

References:

https://introcs.cs.princeton.edu/java/51language/

https://www.geeksforgeeks.org/introduction-of-finite-automata/