Ministerul Educaţiei şi Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică şi Microelectronică

Laboratory work nr. 5
Course: Formal languages and finite automata
Topic: Chomsky Normal Form

Elaborated:
st. gr. FAF-221 Berco Andrei

Verified:
asist. univ. Cretu Dumitru

Chişinău - 2024

Theory

Chomsky Normal Form (CNF) is a way of representing context-free grammars (CFGs) in a specific form, named after the renowned linguist and cognitive scientist Noam Chomsky. It has several important properties that make it useful for theoretical analysis and practical applications in areas such as natural language processing and parsing algorithms. Here's a breakdown of the key aspects of CNF:

1. Formal Definition: In Chomsky Normal Form, every production rule of the grammar is in one of two forms:
   - A → BC
   - A → a

   where A, B, and C are non-terminal symbols, and a is a terminal symbol. The production A → ε (where ε represents the empty string) is allowed only if the start symbol appears on the right-hand side of a production.
2. Non-terminal Usage: In CNF, each non-terminal symbol (except for the start symbol, which can produce ε) must derive at least one string of terminal symbols.
3. Simplification: The removal of useless symbols and unproductive rules is often done before converting a grammar to CNF. Useless symbols are those that cannot be reached from the start symbol, and unproductive rules are those that cannot derive any terminal string.
4. Advantages:
   - CNF simplifies the structure of the grammar, making it easier to analyze and process.
   - It facilitates certain parsing algorithms, such as the CYK (Cocke-Younger-Kasami) algorithm, which operates efficiently on grammars in CNF.
   - CNF helps in proving various properties of context-free languages, such as closure properties.
5. Conversion: Any context-free grammar can be converted to an equivalent grammar in Chomsky Normal Form. The process typically involves several steps, including:
   - Eliminating ε-productions
   - Eliminating unit productions (productions of the form A → B)
   - Eliminating productions with more than two non-terminals on the right-hand side
   - Introducing new non-terminals as necessary

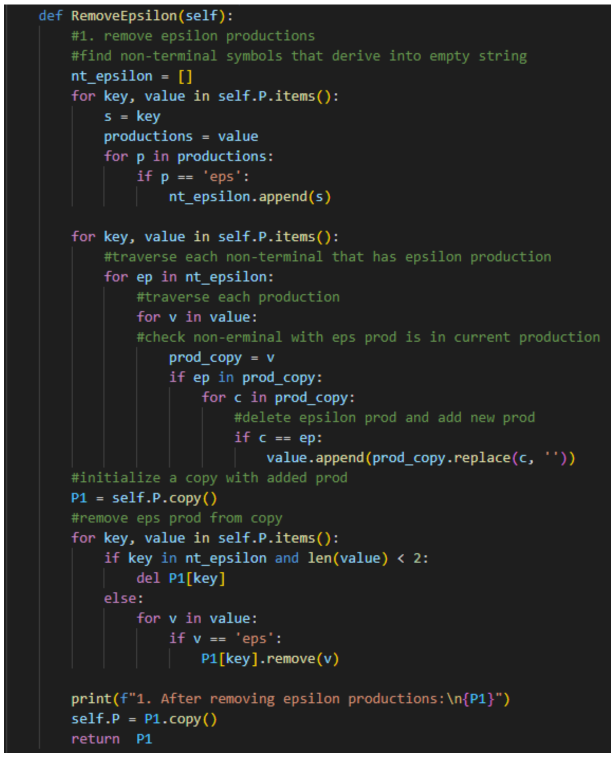
Objectives:

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.
4. The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).
5. The implemented functionality needs executed and tested.
6. A BONUS point will be given for the student who will have unit tests that validate the functionality of the project.
7. 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 Description

For implementation I chose to use Python, because it is a familiar language.

First of all, I define a class with constructor for grammar. Also, it will have methods for step-by-step obtaining the CNF, like RemoveEpsilon, EliminateUnit, EliminateInaccesible, EliminateUnprod and TransformToCNF

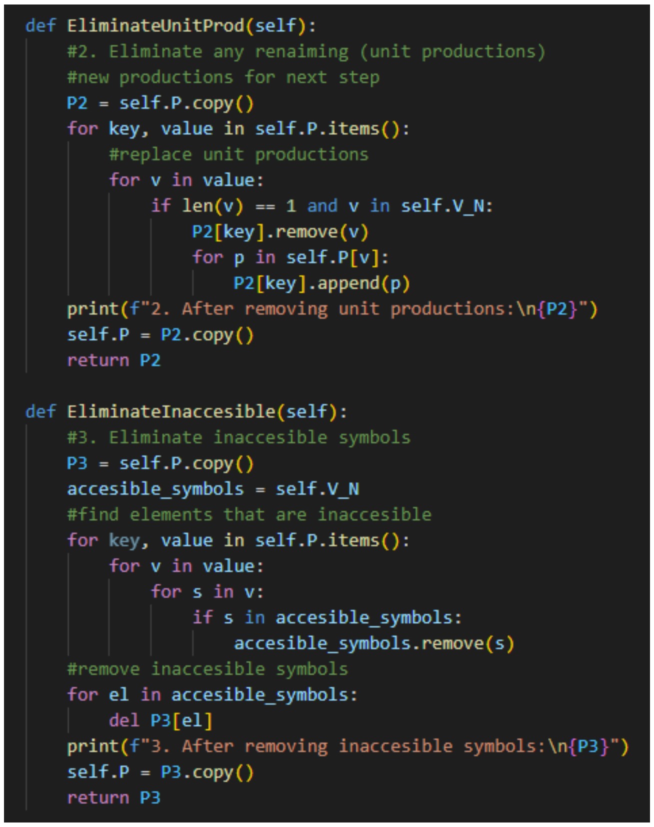


```python
def RemoveEpsilon(self):
    #1. remove epsilon productions
    #find non-terminal symbols that derive into empty string
    nt_epsilon = []
    for key, value in self.P.items():
        s = key
        productions = value
        for p in productions:
            if p == 'eps':
                nt_epsilon.append(s)

    for key, value in self.P.items():
        #traverse each non-terminal that has epsilon production
        for ep in nt_epsilon:
            #traverse each production
            for v in value:
            #check non-erminal with eps prod is in current production
                prod_copy = v
                if ep in prod_copy:
                    for c in prod_copy:
                        #delete epsilon prod and add new prod
                        if c == ep:
                            value.append(prod_copy.replace(c, ''))
    #initialize a copy with added prod
    P1 = self.P.copy()
    #remove eps prod from copy
    for key, value in self.P.items():
        if key in nt_epsilon and len(value) < 2:
            del P1[key]
        else:
            for v in value:
                if v == 'eps':
                    P1[key].remove(v)

    print(f"1. After removing epsilon productions:\n{P1}")
    self.P = P1.copy()
    return  P1
```

Figure 1. Method to remove epsilon prod

It first iterates over each production rule in the grammar (self.P) and checks if any of them contain the epsilon symbol 'eps'. Epsilon productions are those that can derive the empty string.

It maintains a list called nt_epsilon, which stores the non-terminal symbols that have epsilon productions. It then iterates over each production rule again and checks if any of them contain non-terminal symbols that derive epsilon. For each production containing such non-terminal symbols, it creates a modified version where all occurrences of the non-terminal symbol are removed, effectively removing epsilon productions. It creates a copy of the original productions (P1 = self.P.copy()). It iterates over each non-terminal symbol in the grammar and removes it if it only had epsilon productions and no other productions. It also removes epsilon from any production rules. It prints the updated productions after removing epsilon productions. It sets self.P to the updated productions. It returns the updated productions.



```python
def EliminateUnitProd(self):
    #2. Eliminate any renaiming (unit productions)
    #new productions for next step
    P2 = self.P.copy()
    for key, value in self.P.items():
        #replace unit productions
        for v in value:
            if len(v) == 1 and v in self.V_N:
                P2[key].remove(v)
                for p in self.P[v]:
                    P2[key].append(p)
    print(f"2. After removing unit productions:\n{P2}")
    self.P = P2.copy()
    return P2

def EliminateInaccesible(self):
    #3. Eliminate inaccesible symbols
    P3 = self.P.copy()
    accesible_symbols = self.V_N
    #find elements that are inaccesible
    for key, value in self.P.items():
        for v in value:
            for s in v:
                if s in accesible_symbols:
                    accesible_symbols.remove(s)
    #remove inaccesible symbols
    for el in accesible_symbols:
        del P3[el]
    print(f"3. After removing inaccesible symbols:\n{P3}")
    self.P = P3.copy()
    return P3
```

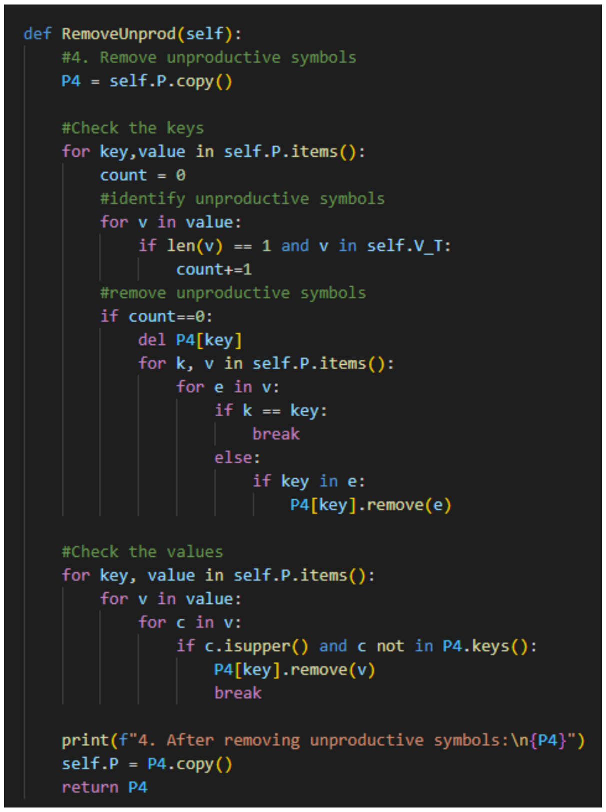
Figure 2. Methods to eliminate unit prod. and inaccessible symbols

EliminateUnitProd Method:

This method aims to remove unit productions, which are production rules where a non-terminal directly produces another single non-terminal. It starts by creating a copy of the original productions (P2 = self.P.copy()). It iterates over each production rule in the grammar (for key, value in self.P.items()). For each production rule, if the length of the rule is 1 and it is a non-terminal (if len(v) == 1 and v in self.V_N), it replaces the unit production with the productions of the non-terminal it produces. It prints the updated productions after removing unit productions. Finally, it sets self.P to the updated productions and returns them.

EliminateInaccesible Method:

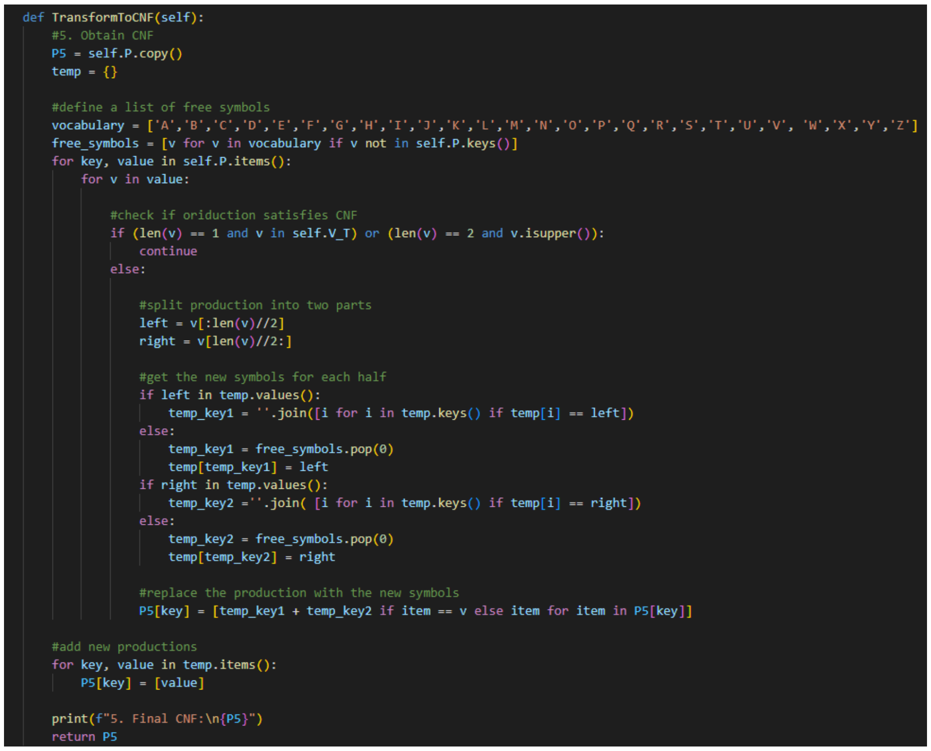
This method aims to remove inaccessible symbols, which are non-terminals that cannot be reached from the start symbol. It starts by creating a copy of the original productions (P3 = self.P.copy()). It initializes a set accessible_symbols with all non-terminal symbols (self.V_N). It iterates over each production rule in the grammar and removes symbols that are accessible (for key, value in self.P.items()). Then, it removes inaccessible symbols from the productions. It prints the updated productions after removing inaccessible symbols. Finally, it sets self.P to the updated productions and returns them.



```
def RemoveUnprod(self):
    #4. Remove unproductive symbols
    P4 = self.P.copy()

    #Check the keys
    for key,value in self.P.items():
        count = 0
        #identify unproductive symbols
        for v in value:
            if len(v) == 1 and v in self.V_T:
                count+=1
        #remove unproductive symbols
        if count==0:
            del P4[key]
            for k, v in self.P.items():
                for e in v:
                    if k == key:
                        break
                    else:
                        if key in e:
                            P4[key].remove(e)

    #Check the values
    for key, value in self.P.items():
        for v in value:
            for c in v:
                if c.isupper() and c not in P4.keys():
                    P4[key].remove(v)
                    break

    print(f"4. After removing unproductive symbols:\n{P4}")
    self.P = P4.copy()
    return P4
```

Figure 3. Method to remove unproductive symbols

It creates a copy of the original productions (P4 = self.P.copy()). It iterates over each key-value pair in the grammar (for key,value in self.P.items()). For each key, it counts the number of single non-terminal symbols that are also terminal symbols (len(v) == 1 and v in self.V_T). If there are no such symbols, it considers the key unproductive. If a key is found to be unproductive (count equals 0), it deletes that key from P4. Additionally, it removes any productions that contain the unproductive key from other non-terminal symbols' productions. It iterates over each key-value pair in the grammar again. For each value (production rule), it iterates over each character (for c in v) in the production. If a character is an uppercase letter (indicating a non-terminal symbol) and it is not a key in P4 (indicating it's unproductive), it removes the entire production from the corresponding key in P4. It prints the updated productions after removing unproductive symbols. It sets self.P to the updated productions and returns them.



```python
def TransformToCNF(self):
    #5. Obtain CNF
    P5 = self.P.copy()
    temp = {}

    #define a list of free symbols
    vocabulary = ['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V', 'W','X','Y','Z']
    free_symbols = [v for v in vocabulary if v not in self.P.keys()]
    for key, value in self.P.items():
        for v in value:

            #check if oriduction satisfies CNF
            if (len(v) == 1 and v in self.V_T) or (len(v) == 2 and v.isupper()):
                continue
            else:

                #split production into two parts
                left = v[:len(v)//2]
                right = v[len(v)//2:]

                #get the new symbols for each half
                if left in temp.values():
                    temp_key1 = ''.join([i for i in temp.keys() if temp[i] == left])
                else:
                    temp_key1 = free_symbols.pop(0)
                    temp[temp_key1] = left
                if right in temp.values():
                    temp_key2 =''.join( [i for i in temp.keys() if temp[i] == right])
                else:
                    temp_key2 = free_symbols.pop(0)
                    temp[temp_key2] = right

                #replace the production with the new symbols
                P5[key] = [temp_key1 + temp_key2 if item == v else item for item in P5[key]]

    #add new productions
    for key, value in temp.items():
        P5[key] = [value]

    print(f"5. Final CNF:\n{P5}")
    return P5
```

Figure 4. Method to construct CNF

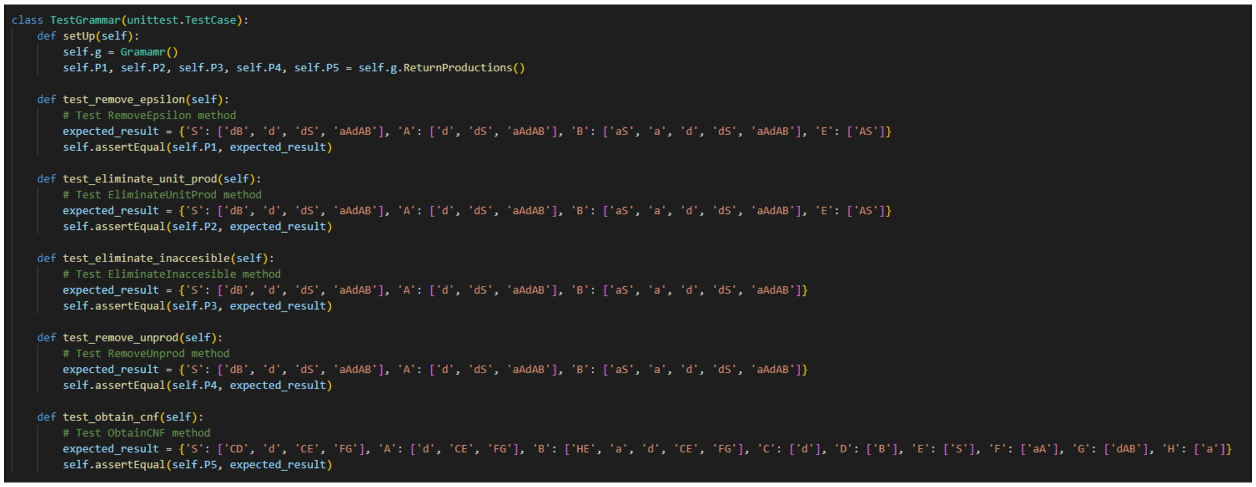
It creates a copy of the original productions (P5 = self.P.copy()). It initializes an empty dictionary temp to keep track of temporary symbols introduced during the transformation.

It defines a list of free symbols (vocabulary) and identifies symbols that are not already used as keys in the grammar (free_symbols). It iterates over each production rule in the grammar (for key, value in self.P.items()). For each production, it checks if it already satisfies CNF:

If the production consists of a single terminal symbol or two non-terminal symbols, it continues to the next production. Otherwise, it splits the production into two halves (left and right).

It then generates new symbols for each half if they are not already present in temp. It replaces the original production with the concatenation of the new symbols. It stores the mappings of new symbols to their corresponding halves in the temp dictionary. It iterates over the items in the temp dictionary. For each new symbol introduced, it adds a production rule to P5 with the corresponding half of the original production. It prints the final CNF after the transformation.

It returns the updated productions in CNF.

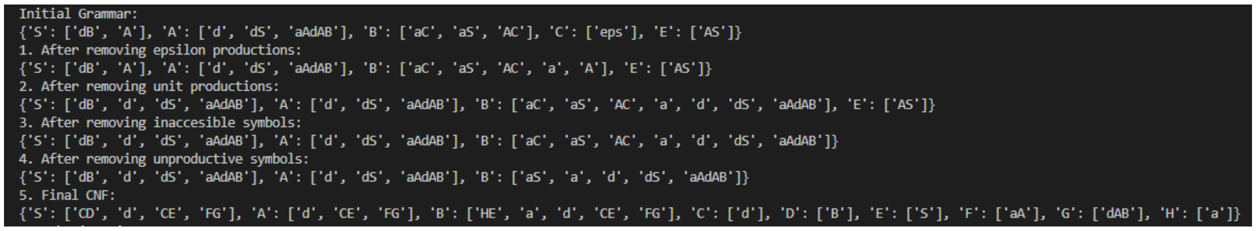


```python
class TestGrammar(unittest.TestCase):
    def setUp(self):
        self.g = Gramamr()
        self.P1, self.P2, self.P3, self.P4, self.P5 = self.g.ReturnProductions()

    def test_remove_epsilon(self):
        # Test RemoveEpsilon method
        expected_result = {'S': ['dB', 'd', 'dS', 'aAdAB'], 'A': ['d', 'dS', 'aAdAB'], 'B': ['aS', 'a', 'd', 'dS', 'aAdAB'], 'E': ['AS']}
        self.assertEqual(self.P1, expected_result)

    def test_eliminate_unit_prod(self):
        # Test EliminateUnitProd method
        expected_result = {'S': ['dB', 'd', 'dS', 'aAdAB'], 'A': ['d', 'dS', 'aAdAB'], 'B': ['aS', 'a', 'd', 'dS', 'aAdAB'], 'E': ['AS']}
        self.assertEqual(self.P2, expected_result)

    def test_eliminate_inaccesible(self):
        # Test EliminateInaccesible method
        expected_result = {'S': ['dB', 'd', 'dS', 'aAdAB'], 'A': ['d', 'dS', 'aAdAB'], 'B': ['aS', 'a', 'd', 'dS', 'aAdAB']}
        self.assertEqual(self.P3, expected_result)

    def test_remove_unprod(self):
        # Test RemoveUnprod method
        expected_result = {'S': ['dB', 'd', 'dS', 'aAdAB'], 'A': ['d', 'dS', 'aAdAB'], 'B': ['aS', 'a', 'd', 'dS', 'aAdAB']}
        self.assertEqual(self.P4, expected_result)

    def test_obtain_cnf(self):
        # Test ObtainCNF method
        expected_result = {'S': ['CD', 'd', 'CE', 'FG'], 'A': ['d', 'CE', 'FG'], 'B': ['HE', 'a', 'd', 'CE', 'FG'], 'C': ['d'], 'D': ['B'], 'E': ['S'], 'F': ['aA'], 'G': ['dAB'], 'H': ['a']}
        self.assertEqual(self.P5, expected_result)
```

Figure 5. Unit Test

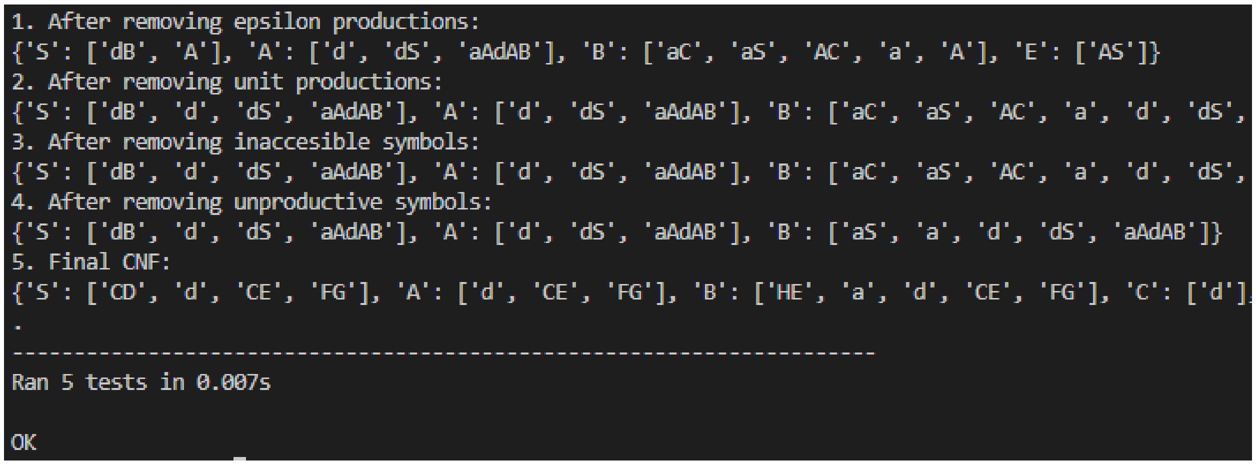
This is a unit test class named TestGrammar, which is designed to test the functionality of a class named Grammar. The setUp method is used to set up the initial conditions for the tests by initializing an instance of the Grammar class and obtaining the productions through the ReturnProductions method. Then, there are individual test methods for each transformation step: test_remove_epsilon, test_eliminate_unit_prod, test_eliminate_inaccesible, test_remove_unprod, and test_obtain_cnf.

Screenshots



```
Initial Grammar:
{'S': ['dB', 'A'], 'A': ['d', 'dS', 'aAdAB'], 'B': ['aC', 'aS', 'AC'], 'C': ['eps'], 'E': ['AS']}
1. After removing epsilon productions:
{'S': ['dB', 'A'], 'A': ['d', 'dS', 'aAdAB'], 'B': ['aC', 'aS', 'AC', 'a', 'A'], 'E': ['AS']}
2. After removing unit productions:
{'S': ['dB', 'd', 'dS', 'aAdAB'], 'A': ['d', 'dS', 'aAdAB'], 'B': ['aC', 'aS', 'AC', 'a', 'd', 'dS', 'aAdAB'], 'E': ['AS']}
3. After removing inaccesible symbols:
{'S': ['dB', 'd', 'dS', 'aAdAB'], 'A': ['d', 'dS', 'aAdAB'], 'B': ['aC', 'aS', 'AC', 'a', 'd', 'dS', 'aAdAB']}
4. After removing unproductive symbols:
{'S': ['dB', 'd', 'dS', 'aAdAB'], 'A': ['d', 'dS', 'aAdAB'], 'B': ['aS', 'a', 'd', 'dS', 'aAdAB']}
5. Final CNF:
{'S': ['CD', 'd', 'CE', 'FG'], 'A': ['d', 'CE', 'FG'], 'B': ['HE', 'a', 'd', 'CE', 'FG'], 'C': ['d'], 'D': ['B'], 'E': ['S'], 'F': ['aA'], 'G': ['dAB'], 'H': ['a']}
```

Figure 6. Output for converted grammar



```
1. After removing epsilon productions:
{'S': ['dB', 'A'], 'A': ['d', 'dS', 'aAdAB'], 'B': ['aC', 'aS', 'AC', 'a', 'A'], 'E': ['AS']}
2. After removing unit productions:
{'S': ['dB', 'd', 'dS', 'aAdAB'], 'A': ['d', 'dS', 'aAdAB'], 'B': ['aC', 'aS', 'AC', 'a', 'd', 'dS',
3. After removing inaccessible symbols:
{'S': ['dB', 'd', 'dS', 'aAdAB'], 'A': ['d', 'dS', 'aAdAB'], 'B': ['aC', 'aS', 'AC', 'a', 'd', 'dS',
4. After removing unproductive symbols:
{'S': ['dB', 'd', 'dS', 'aAdAB'], 'A': ['d', 'dS', 'aAdAB'], 'B': ['aS', 'a', 'd', 'dS', 'aAdAB']}
5. Final CNF:
{'S': ['CD', 'd', 'CE', 'FG'], 'A': ['d', 'CE', 'FG'], 'B': ['HE', 'a', 'd', 'CE', 'FG'], 'C': ['d'],
.
----------------------------------------------------------------------
Ran 5 tests in 0.007s

OK
```

Figure 7. Output of Unit Test

Conclusions

Context-free grammars (CFGs) and Chomsky Normal Form (CNF) are crucial concepts in formal language theory, providing a structured way to describe the syntax of languages through production rules. The process of transforming a CFG into CNF involves several steps, including removing epsilon productions, eliminating unit productions, and eliminating inaccessible and unproductive symbols. These steps aim to simplify the grammar's structure and make it easier to analyze and process.

Chomsky Normal Form offers several advantages, including facilitating parsing algorithms like the CYK algorithm and simplifying proofs of language properties. The conversion process ensures that the resulting grammar adheres to a standardized form, which aids in theoretical analysis and practical applications in areas such as natural language processing and parsing algorithms.

The provided Python implementations demonstrate the step-by-step transformation of a CFG into CNF, showcasing each transformation method's functionality through unit tests. These tests ensure that each transformation method produces the expected output, validating the correctness of the implementation.

In conclusion, understanding and implementing the conversion of CFGs to CNF is essential for formal language theory and computational linguistics. It provides a foundation for further exploration into language properties, parsing algorithms, and language processing applications. Additionally, the practical implementations serve as valuable tools for educational purposes and real-world applications in various computational fields.