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

Laboratory work 5:
Chomsky Normal Form

Elaborated:

st. gr. FAF-221 Reabciuc Daria-Brianna

Verified:

asist. univ. Crețu Dumitru

Chisinau 2024

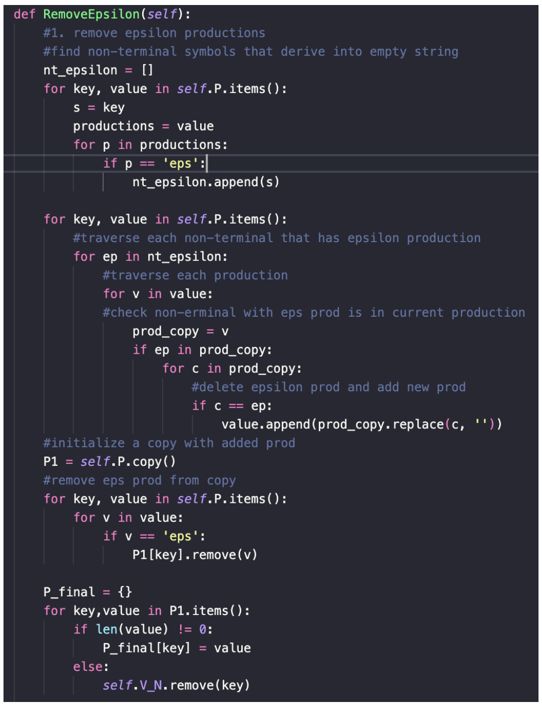
Objectives

Learn about Chomsky Normal Form (CNF) and get familiar with the approaches of normalizing a grammar. Implement a method for normalizing an input grammar by the rules of CNF, and ensure that this implementation is encapsulated in a method with an appropriate signature, ideally within an appropriate class or type. The implemented functionality needs to be executed and tested. A bonus point will be given for the student who has unit tests that validate the functionality of the project. Additionally, another bonus point will be awarded if the student makes the aforementioned function capable of accepting any grammar, not only the one from the student's variant.

Chomsky Normal Form (CNF) is essential for representing context-free grammars in computational linguistics and parsing algorithms, named after Noam Chomsky. CNF involves formatting production rules in specific ways: either as $A \rightarrow BC$ or $A \rightarrow a$, where A, B, and C are non-terminals and a is a terminal. The rule $A \rightarrow \varepsilon$ is permitted only if the start symbol appears on the right-hand side of a production. This standardization is crucial for simplifying grammars by eliminating unproductive rules and unreachable symbols, thereby facilitating efficient parsing algorithms such as the CYK algorithm and supporting theoretical analyses of context-free languages. The conversion process to CNF includes removing ε-productions, unit productions, and simplifying complex productions by introducing new non-terminals as necessary. This streamlined structure makes CNF advantageous for analyzing and processing grammars in various linguistic and computational tasks.

Implementation

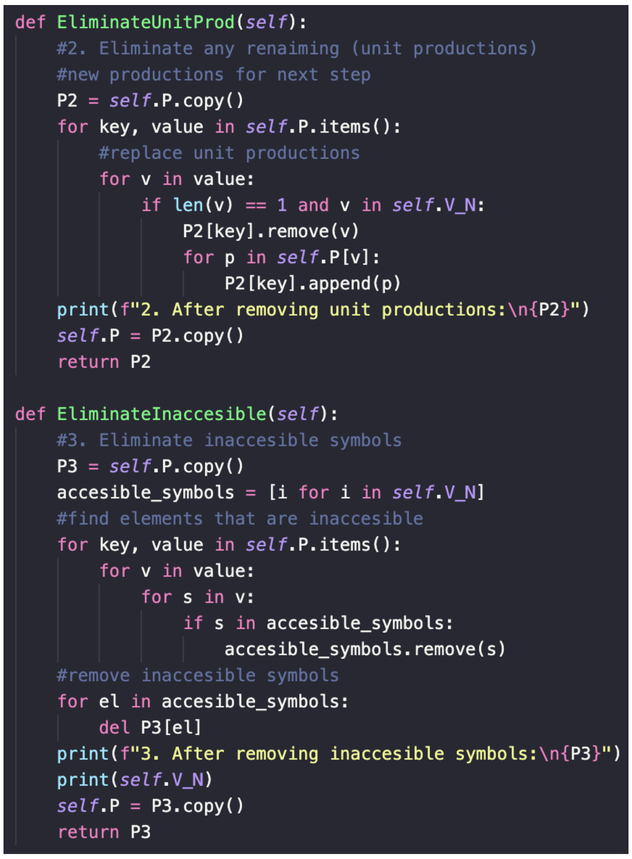
For the implementation, I opted for Python due to its familiarity. Initially, I established a class with a constructor for the grammar, which also includes methods for converting to Chomsky Normal Form step by step. These methods include RemoveEpsilon, EliminateUnit, EliminateInaccessible, EliminateUnproductive, 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():
        for v in value:
            if v == 'eps':
                P1[key].remove(v)

    P_final = {}
    for key,value in P1.items():
        if len(value) != 0:
            P_final[key] = value
        else:
            self.V_N.remove(key)
```

Fig. 1. Method to remove epsilon production

The method begins by iterating through each production rule in the grammar (self.P) to identify any that include the epsilon symbol 'eps', which denotes productions that can derive the empty string. It compiles a list called nt_epsilon to keep track of non-terminal symbols associated with epsilon productions. A second iteration through each production rule then occurs to locate and modify productions that involve these non-terminal symbols, effectively eliminating epsilon productions. This involves creating a modified version of each relevant production by omitting the non-terminal symbols that derive epsilon. A copy of the original productions is made (P1 = self.P.copy()). The method also processes each non-terminal in the grammar, removing those that solely result in epsilon productions without contributing to other productions, and it cleans up epsilon from the rules as well. The updated productions are printed and set as the current grammar's productions in self.P. Finally, it returns these revised 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 = [i for i in 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}")
    print(self.V_N)
    self.P = P3.copy()
    return P3
```

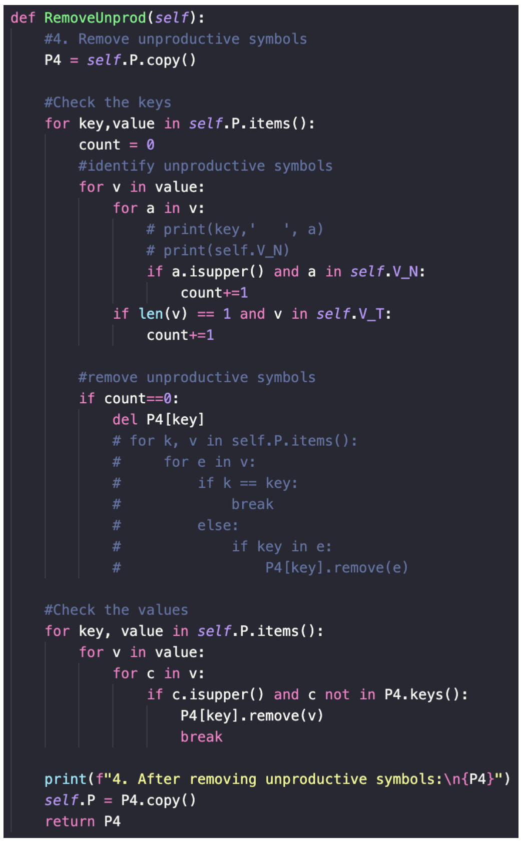
Fig. 2. Method to remove production and inaccessible elements

EliminateUnitProd Method:

This method is designed to remove unit productions from a grammar. Unit productions are rules where one non-terminal symbol directly generates another non-terminal symbol without any intermediate symbols. The process begins by duplicating the original set of production rules (P2 = self.P.copy()). It then examines each production rule (for key, value in self.P.items()). If a rule consists solely of one non-terminal symbol (if len(v) == 1 and v in self.V_N), the method substitutes the unit production with the productions associated with the non-terminal it points to. After modifying the productions to exclude unit productions, it prints the revised set of productions. The updated productions are then assigned back to self.P, and the modified set is returned.

EliminateInaccessible Method:

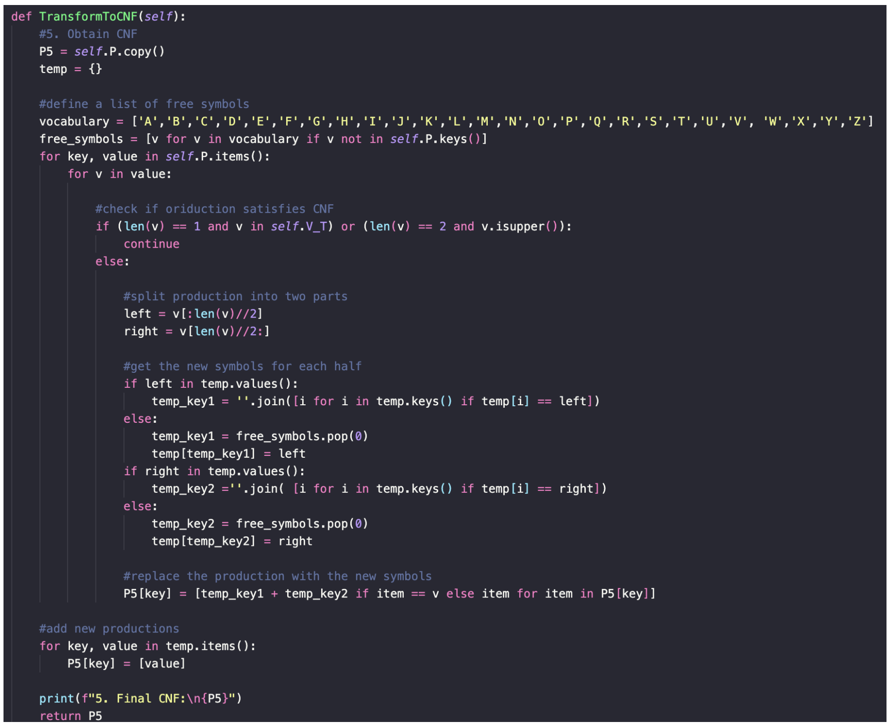
This method targets the elimination of inaccessible symbols from a grammar. Inaccessible symbols are non-terminal symbols that cannot be derived from the starting symbol. Initially, the method duplicates the original productions (P3 = self.P.copy()). It then sets up a set named accessible_symbols including all non-terminal symbols (self.V_N). It iterates through each production rule in the grammar, identifying and removing symbols that are accessible (for key, value in self.P.items()). Subsequently, it deletes the inaccessible symbols from the production rules. The method outputs the updated production rules after the removal of inaccessible symbols. Finally, it updates self.P with the new set of productions and returns this set.



```python
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:
            for a in v:
                # print(key,'   ', a)
                # print(self.V_N)
                if a.isupper() and a in self.V_N:
                    count+=1
            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
```

Fig. 3. Method To remove unproductive symbols

This method initiates by creating a copy of the original production rules (P4 = self.P.copy()). It then processes each production rule within the grammar by iterating through every key-value pair (for key, value in self.P.items()). For each key, it evaluates if any of the production rules consist solely of a single non-terminal symbol that also qualifies as a terminal symbol (len(v) == 1 and v in self.V_T). If no such rules are found for a key, that key is deemed unproductive. Upon identifying an unproductive key (where count equals 0), it is removed from P4. Moreover, any occurrences of this unproductive key within other non-terminal symbols' productions are also eliminated. The method then revisits each key-value pair in the grammar. During this second pass, for each production rule, it checks every character (for c in v) within the production. If a character is an uppercase letter (signifying a non-terminal symbol) and is absent from P4 (indicating it is unproductive), the entire production associated with that character is removed from P4. After completing these modifications, it displays the revised set of productions. Finally, it updates self.P with these new productions and returns the modified production set.



```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
```

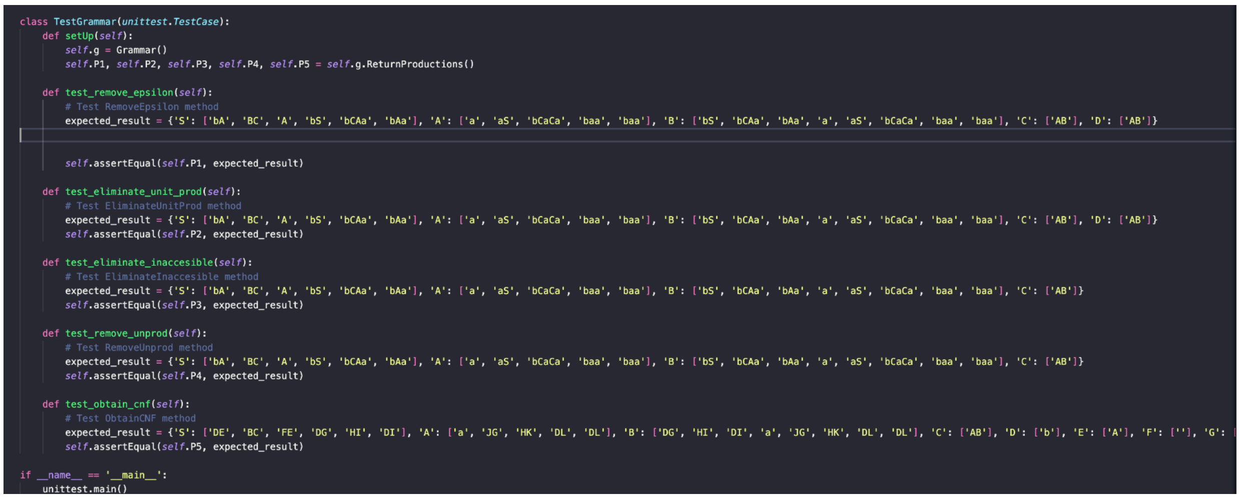
Fig. 4 Method to construct CNF

This method starts by duplicating the original set of production rules (P5 = self.P.copy()). It also sets up an empty dictionary called temp, which is used to store temporary symbols introduced during the transformation to Chomsky Normal Form (CNF). Additionally, it defines a list of available symbols (vocabulary) and selects those not currently used as keys in the grammar (free_symbols). The method iterates through each production rule in the grammar (for key, value in self.P.items()). For each rule, it checks whether the production already conforms to CNF:

If the production is either a single terminal symbol or a pair of non-terminal symbols, it skips to the next rule. Otherwise, the production is divided into two parts (left and right).

New symbols are created for each part if they are not already noted in the temp dictionary. The original production is then replaced by a concatenation of these new symbols. These mappings from new symbols to their respective halves are recorded in the temp dictionary. The method continues by iterating through each entry in the temp dictionary, adding new production rules to P5 corresponding to each half of the original productions. Once all transformations are complete, the final CNF is printed.

The updated set of productions in CNF is returned.

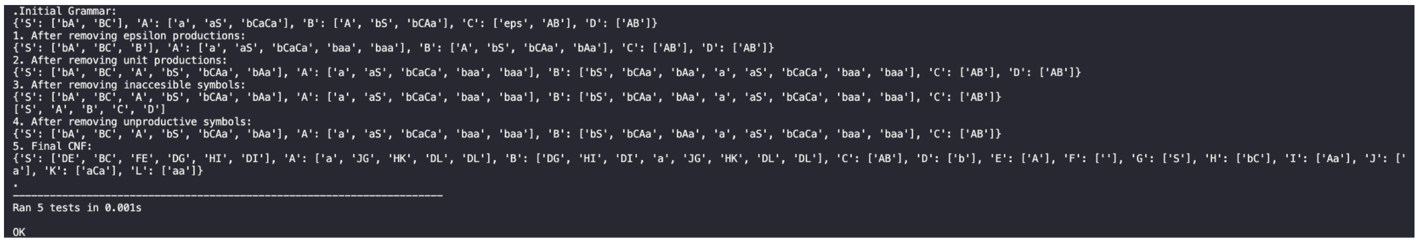


```python
class TestGrammar(unittest.TestCase):
    def setUp(self):
        self.g = Grammar()
        self.P1, self.P2, self.P3, self.P4, self.P5 = self.g.ReturnProductions()

    def test_remove_epsilon(self):
        # Test RemoveEpsilon method
        expected_result = {'S': ['bA', 'BC', 'A', 'bS', 'bCAa', 'bAa'], 'A': ['a', 'aS', 'bCaCa', 'baa', 'baa'], 'B': ['bS', 'bCAa', 'bAa', 'a', 'aS', 'bCaCa', 'baa', 'baa'], 'C': ['AB'], 'D': ['AB']}


        self.assertEqual(self.P1, expected_result)

    def test_eliminate_unit_prod(self):
        # Test EliminateUnitProd method
        expected_result = {'S': ['bA', 'BC', 'A', 'bS', 'bCAa', 'bAa'], 'A': ['a', 'aS', 'bCaCa', 'baa', 'baa'], 'B': ['bS', 'bCAa', 'bAa', 'a', 'aS', 'bCaCa', 'baa', 'baa'], 'C': ['AB'], 'D': ['AB']}
        self.assertEqual(self.P2, expected_result)

    def test_eliminate_inaccesible(self):
        # Test EliminateInaccesible method
        expected_result = {'S': ['bA', 'BC', 'A', 'bS', 'bCAa', 'bAa'], 'A': ['a', 'aS', 'bCaCa', 'baa', 'baa'], 'B': ['bS', 'bCAa', 'bAa', 'a', 'aS', 'bCaCa', 'baa', 'baa'], 'C': ['AB']}
        self.assertEqual(self.P3, expected_result)

    def test_remove_unprod(self):
        # Test RemoveUnprod method
        expected_result = {'S': ['bA', 'BC', 'A', 'bS', 'bCAa', 'bAa'], 'A': ['a', 'aS', 'bCaCa', 'baa', 'baa'], 'B': ['bS', 'bCAa', 'bAa', 'a', 'aS', 'bCaCa', 'baa', 'baa'], 'C': ['AB']}
        self.assertEqual(self.P4, expected_result)

    def test_obtain_cnf(self):
        # Test ObtainCNF method
        expected_result = {'S': ['DE', 'BC', 'FE', 'DG', 'HI', 'DI'], 'A': ['a', 'JG', 'HK', 'DL', 'DL'], 'B': ['DG', 'HI', 'DI', 'a', 'JG', 'HK', 'DL', 'DL'], 'C': ['AB'], 'D': ['b'], 'E': ['A'], 'F': [''], 'G': [
        self.assertEqual(self.P5, expected_result)

if __name__ == '__main__':
    unittest.main()
```

Fig. 5. Unit Testing

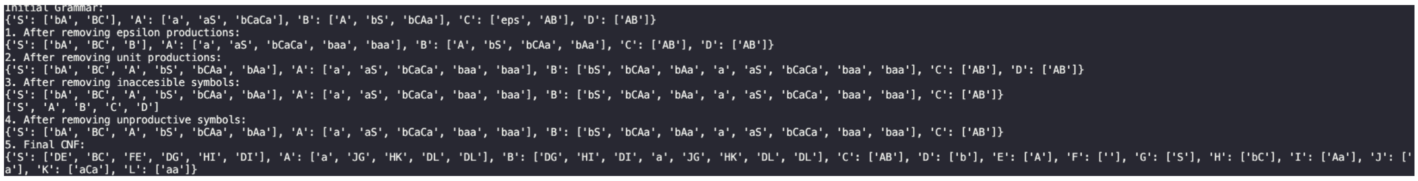
The TestGrammar class is structured to evaluate the operations of a class called Grammar. Within this test class, the setUp method initializes by creating a new instance of the Grammar class and retrieves the set of production rules using the ReturnProductions method. The class then includes separate test methods for each specific transformation process within the Grammar class. These methods are named test_remove_epsilon, test_eliminate_unit_prod, test_eliminate_inaccessible, test_remove_unprod, and test_obtain_cnf, each designed to test a different aspect of the Grammar class's functionality in processing grammar transformations.

Output



```
.Initial Grammar:
{'S': ['bA', 'BC'], 'A': ['a', 'aS', 'bCaCa'], 'B': ['A', 'bS', 'bCAa'], 'C': ['eps', 'AB'], 'D': ['AB']}
1. After removing epsilon productions:
{'S': ['bA', 'BC', 'B'], 'A': ['a', 'aS', 'bCaCa', 'baa', 'baa'], 'B': ['A', 'bS', 'bCAa', 'bAa'], 'C': ['AB'], 'D': ['AB']}
2. After removing unit productions:
{'S': ['bA', 'BC', 'A', 'bS', 'bCAa', 'bAa'], 'A': ['a', 'aS', 'bCaCa', 'baa', 'baa'], 'B': ['bS', 'bCAa', 'bAa', 'a', 'aS', 'bCaCa', 'baa', 'baa'], 'C': ['AB'], 'D': ['AB']}
3. After removing inaccesible symbols:
{'S': ['bA', 'BC', 'A', 'bS', 'bCAa', 'bAa'], 'A': ['a', 'aS', 'bCaCa', 'baa', 'baa'], 'B': ['bS', 'bCAa', 'bAa', 'a', 'aS', 'bCaCa', 'baa', 'baa'], 'C': ['AB']}
['S', 'A', 'B', 'C', 'D']
4. After removing unproductive symbols:
{'S': ['bA', 'BC', 'A', 'bS', 'bCAa', 'bAa'], 'A': ['a', 'aS', 'bCaCa', 'baa', 'baa'], 'B': ['bS', 'bCAa', 'bAa', 'a', 'aS', 'bCaCa', 'baa', 'baa'], 'C': ['AB']}
5. Final CNF:
{'S': ['DE', 'BC', 'FE', 'DG', 'HI', 'DI'], 'A': ['a', 'JG', 'HK', 'DL', 'DL'], 'B': ['DG', 'HI', 'DI', 'a', 'JG', 'HK', 'DL', 'DL'], 'C': ['AB'], 'D': ['b'], 'E': ['A'], 'F': [''], 'G': ['S'], 'H': ['bC'], 'I': ['Aa'], 'J': ['
a'], 'K': ['aCa'], 'L': ['aa']}
.
----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
```

Fig. 6. Output from unit test



```
Initial Grammar:
{'S': ['bA', 'BC'], 'A': ['a', 'aS', 'bCaCa'], 'B': ['A', 'bS', 'bCAa'], 'C': ['eps', 'AB'], 'D': ['AB']}
1. After removing epsilon productions:
{'S': ['bA', 'BC', 'B'], 'A': ['a', 'aS', 'bCaCa', 'baa', 'baa'], 'B': ['A', 'bS', 'bCAa', 'bAa'], 'C': ['AB'], 'D': ['AB']}
2. After removing unit productions:
{'S': ['bA', 'BC', 'A', 'bS', 'bCAa', 'bAa'], 'A': ['a', 'aS', 'bCaCa', 'baa', 'baa'], 'B': ['bS', 'bCAa', 'bAa', 'a', 'aS', 'bCaCa', 'baa', 'baa'], 'C': ['AB'], 'D': ['AB']}
3. After removing inaccesible symbols:
{'S': ['bA', 'BC', 'A', 'bS', 'bCAa', 'bAa'], 'A': ['a', 'aS', 'bCaCa', 'baa', 'baa'], 'B': ['bS', 'bCAa', 'bAa', 'a', 'aS', 'bCaCa', 'baa', 'baa'], 'C': ['AB']}
['S', 'A', 'B', 'C', 'D']
4. After removing unproductive symbols:
{'S': ['bA', 'BC', 'A', 'bS', 'bCAa', 'bAa'], 'A': ['a', 'aS', 'bCaCa', 'baa', 'baa'], 'B': ['bS', 'bCAa', 'bAa', 'a', 'aS', 'bCaCa', 'baa', 'baa'], 'C': ['AB']}
5. Final CNF:
{'S': ['DE', 'BC', 'FE', 'DG', 'HI', 'DI'], 'A': ['a', 'JG', 'HK', 'DL', 'DL'], 'B': ['DG', 'HI', 'DI', 'a', 'JG', 'HK', 'DL', 'DL'], 'C': ['AB'], 'D': ['b'], 'E': ['A'], 'F': [''], 'G': ['S'], 'H': ['bC'], 'I': ['Aa'], 'J': ['
a'], 'K': ['aCa'], 'L': ['aa']}
```

Fig. 7. Output for converted grammar

Conclusion

Context-free grammars (CFGs) and Chomsky Normal Form (CNF) are fundamental concepts in the study of formal languages, used to articulate the syntax rules of languages through specific production rules. Transforming a CFG into CNF involves multiple crucial steps such as eliminating epsilon productions, removing unit productions, and purging inaccessible and unproductive symbols. These transformations aim to streamline the grammar for better analysis and computational handling.

Chomsky Normal Form is particularly valuable because it supports efficient parsing algorithms like the CYK algorithm and aids in the simplification of proofs concerning language properties. By standardizing the grammar's structure, the transformation process enhances both theoretical studies and practical applications in fields like natural language processing and parsing technologies.

The Python code provided illustrates the meticulous process of converting a CFG to CNF, with each transformation step methodically tested through unit tests. These tests are essential for verifying that each step of the transformation is executed correctly, thereby ensuring the reliability of the code.

In essence, mastering the conversion of CFGs to CNF is pivotal for those studying formal language theory and computational linguistics. It lays the groundwork for deeper investigations into language attributes, parsing techniques, and language processing tools. Furthermore, these practical implementations are not only educational but also highly applicable in various computational domains.

https://github.com/dariabrianna/DSL_personal_laboratories/tree/main/Chomsky_Normal_Form