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

Laboratory work nr. 5
Course: Formal languages and finite automata
Topic: Chomsky Normal Form
Variant-20

Elaborated:
st. gr. FAF-223 Pereteatcu Arina

Verified:
asist. univ. Cretu Dumitru

Chişinău - 2024

Theoretical considerations

Chomsky Normal Form. A grammar where every production is either of the form A → BC or A → c (where A, B, C are arbitrary variables and c an arbitrary symbol).

The key advantage is that in Chomsky Normal Form, every derivation of a string of n letters has exactly 2n − 1 steps. Thus: one can determine if a string is in the language by exhaustive search of all derivations.

The conversion to Chomsky Normal Form has four main steps:

1. Get rid of all ε productions

Identify the nullable variables (those that produce ε).
Go over each production and remove all conceivable subset of variables that can be null.
In the event that P → AxB has both A and B nullable, for instance, include productions P → xB | Ax | x.
Next, remove every production that has an empty

2. Get rid of all productions where RHS is one variable

In a unit production, RHS uses a single symbol.
Think of the manufacturing A → B. Next, add the production A → α for each production B → α.
Continue until the task is completed (do not recreate a previously deleted unit production).

3. Replace every production that is too long by shorter productions

For instance, swap out production A → BCD with A → BE and E → CD.
(In principle, this adds a lot of additional variables, but with caution, variables can be reused.)

4. Move all terminals to productions where RHS is one terminal

Add a substitution variable for each terminal to the right of a non-unit production.
For instance, substitute productions A → BC and B → b for production A → bC.

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

My laboratory work was done in Python language.

Variant 20

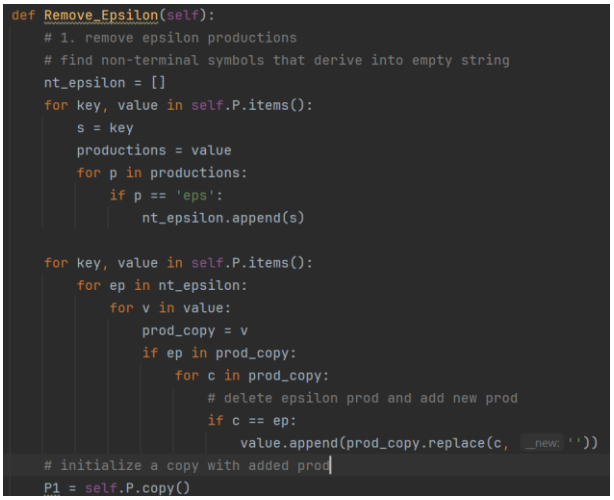
1.Eliminate ε productions.
2.Eliminate any renaming.
3.Eliminate inaccessible symbols.
4. Eliminate the non productive symbols.
5. Obtain the Chomsky Normal Form.

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

P={1. $S \rightarrow aB$
2. $S \rightarrow bA$
3. $S \rightarrow A$
4. $A \rightarrow B$
5. $A \rightarrow Sa$
6. $A \rightarrow bBA$
7. $A \rightarrow b$
8. $B \rightarrow b$
9. $B \rightarrow bS$
10. $B \rightarrow aD$
11. $B \rightarrow \varepsilon$
12. $D \rightarrow AA$
13. $C \rightarrow Ba\}$

Code

First, I defined a class Grammar, which has a dictionary for productions of the grammar, a list of non-terminal symbols and terminal symbols.



```
def Remove_Epsilon(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():
        for ep in nt_epsilon:
            for v in value:
                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, _new: ''))
    # initialize a copy with added prod
    P1 = self.P.copy()
```

Figure1. Remove epsilon

This method “Remove_Epsilon” removes epsilon productions from the grammar.
It starts by identifying non-terminal symbols that have ε productions and stores them in nt_epsilon.
It then iterates through each production in the grammar.
For each production containing a non-terminal symbol with an ε production, it creates new productions by removing the ε symbol from that production. It does this by replacing each occurrence of the ε symbol with an empty string.
After updating the productions, it removes any ε productions from the grammar.
The method prints the updated grammar and returns it.

```python
def Eliminate_Unit_Prod(self):
    # 2. Eliminate any remaining (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
```

Figure2. Eliminate remaining

This method, “Eliminate_Unit_Prod”, is responsible for removing unit productions from the context-free grammar. It iterates through each production in the grammar. For each production, it checks if the length of the production is 1 (meaning it consists of a single non-terminal symbol) and if that symbol is indeed a non-terminal (v in self.V_N).

If the above condition is met, it removes the unit production (the single non-terminal symbol) from the list of productions for that non-terminal.

Then, it adds the productions of the non-terminal (found in self.P[v]) to the current non-terminal's production list.

This process effectively replaces the unit production with the productions of the non-terminal.

```python
def Eliminate_Inaccesible_Symbols(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
```

Figure3. Eliminate inaccessible symbols

This code starts by making a copy of the original productions dictionary self.P and assigns it to P3. Similar to previous methods, this copy is made to avoid modifying the original grammar until all changes are complete. It initializes accessible_symbols to contain all non-terminal symbols self.V_N. This list initially assumes that all non-terminals are accessible.

Then, it iterates through each production in the grammar.

For each production, it iterates through each symbol in the production.

If a symbol from the production is found in accessible_symbols, it removes it. This step is to update accessible_symbols to contain only the symbols that are actually accessible. After identifying all inaccessible symbols, it removes them from the productions dictionary P3.

It iterates through each inaccessible symbol (el) and deletes its entry from P3.

```python
def Remove_Nonproductive(self):
    # 4. Remove non-productive symbols
    P4 = self.P.copy()

    # Check the keys
    for key, value in self.P.items():
        count = 0
        # identify non-productive symbols
        for v in value:
            if len(v) == 1 and v in self.V_T:
                count += 1
        # remove non-productive symbols
        if count == 0 and key not in self.V_T:
            del P4[key]
            for k, v in self.P.items():
                for e in v:
                    if k == key:
                        break
                    else:
                        if key in e:
                            P4[k].remove(e)
```

Figure4.Remove non-productive symbols

This code iterates through each non-terminal symbol (key) in the grammar.
For each non-terminal, it counts the number of productions that consist only of terminal symbols (self.V_T). If a non-terminal has no such productions and is not a terminal symbol itself, it is considered non-productive and is removed from the grammar. If a non-productive symbol is found, it deletes its entry from the productions dictionary P4. It also removes any productions that contain the non-productive symbol. It then iterates through each production in the grammar.
For each production, it checks if it contains any non-terminal symbols (c.isupper()) that are not present in the keys of P4 and are not the current non-terminal being checked (c != key).
If such non-terminal symbols are found, the production is removed from the list of productions for the current non-terminal.

```python
def Chomsky_Normal_Form(self):
    # 5. Obtain CNF
    P5 = self.P.copy()
    temp = {}
    vocabulary = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N
                  '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:
            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
```

Figure5.Chomsky Form

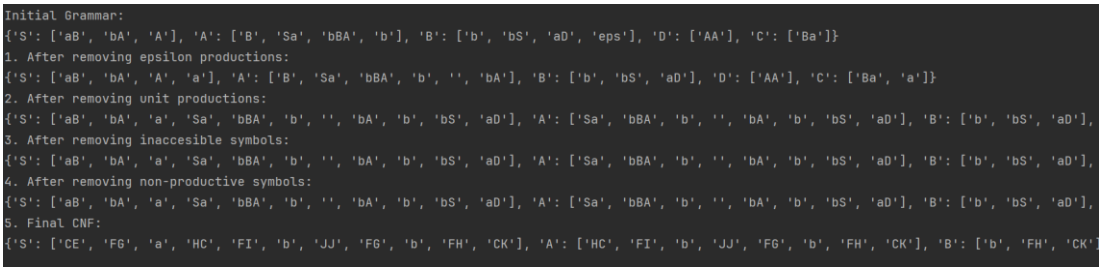
This code defines a list of free symbols vocabulary that are not present in the original grammar.
It iterates through each production in the grammar. For each production, it splits it into two parts: left and right. It checks if the production satisfies CNF criteria:
If the production consists of a single terminal symbol or two non-terminal symbols, it continues to the next production. If the production doesn't satisfy CNF criteria, it generates new symbols for left and right halves: If the left or right half is already assigned a new symbol in temp, it uses that symbol. Otherwise, it pops a symbol from the free_symbols list.
It replaces the original production with the new symbols in P5.

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

    new *
    def test_remove_epsilon(self):
        # Test Remove Epsilon method
        expected_result = {'S': ['aB', 'bA', 'a', 'Sa', 'bBA', 'b', '', 'bA', 'b', 'bS', 'aD'], 'A': ['Sa', 'bB
        self.assertEqual(self.P1, expected_result)

    new *
    def test_eliminate_unit_prod(self):
        # Test Eliminate Unit Prod method
        expected_result = {'S': ['aB', 'bA', 'a', 'Sa', 'bBA', 'b', '', 'bA', 'b', 'bS', 'aD'], 'A': ['Sa', 'bBA
        self.assertEqual(self.P2, expected_result)

    new *
    def test_eliminate_inaccesible(self):
        # Test Eliminate Inaccesible Symbols method
        expected_result = {'S': ['aB', 'bA', 'a', 'Sa', 'bBA', 'b', '', 'bA', 'b', 'bS', 'aD'], 'A': ['Sa', 'bBA
```

Figure5.Unit Test

This code contains a unit test case for the Grammar class to ensure that the methods for transforming a context-free grammar into Chomsky Normal Form (CNF) are working correctly. Each test method asserts that the transformed productions match the expected results

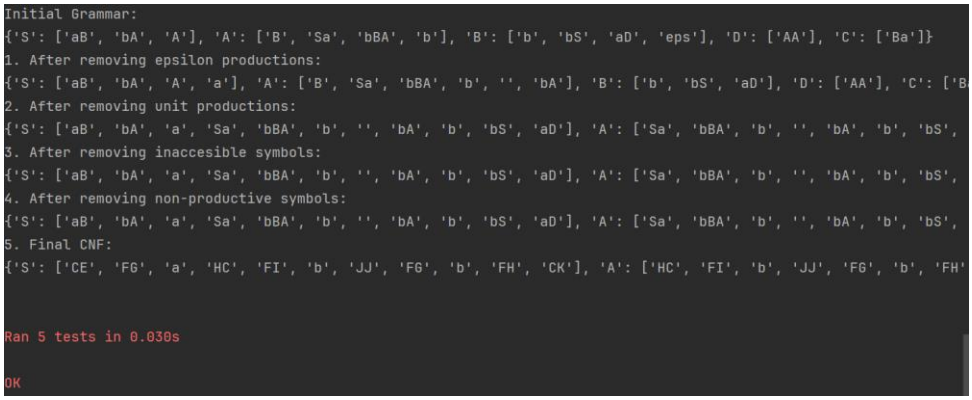
Outputs



```
Initial Grammar:
{'S': ['aB', 'bA', 'A'], 'A': ['B', 'Sa', 'bBA', 'b'], 'B': ['b', 'bS', 'aD', 'eps'], 'D': ['AA'], 'C': ['Ba']}
1. After removing epsilon productions:
{'S': ['aB', 'bA', 'A', 'a'], 'A': ['B', 'Sa', 'bBA', 'b', '', 'bA'], 'B': ['b', 'bS', 'aD'], 'D': ['AA'], 'C': ['Ba', 'a']}
2. After removing unit productions:
{'S': ['aB', 'bA', 'a', 'Sa', 'bBA', 'b', '', 'bA', 'b', 'bS', 'aD'], 'A': ['Sa', 'bBA', 'b', '', 'bA', 'b', 'bS', 'aD'], 'B': ['b', 'bS', 'aD'],
3. After removing inaccesible symbols:
{'S': ['aB', 'bA', 'a', 'Sa', 'bBA', 'b', '', 'bA', 'b', 'bS', 'aD'], 'A': ['Sa', 'bBA', 'b', '', 'bA', 'b', 'bS', 'aD'], 'B': ['b', 'bS', 'aD'],
4. After removing non-productive symbols:
{'S': ['aB', 'bA', 'a', 'Sa', 'bBA', 'b', '', 'bA', 'b', 'bS', 'aD'], 'A': ['Sa', 'bBA', 'b', '', 'bA', 'b', 'bS', 'aD'], 'B': ['b', 'bS', 'aD'],
5. Final CNF:
{'S': ['CE', 'FG', 'a', 'HC', 'FI', 'b', 'JJ', 'FG', 'b', 'FH', 'CK'], 'A': ['HC', 'FI', 'b', 'JJ', 'FG', 'b', 'FH', 'CK'], 'B': ['b', 'FH', 'CK']
```

Figure6.Result of code

This output represents the transformations applied to the initial grammar to convert it into Chomsky Normal Form (CNF), as performed by the Grammar class methods.



```
Initial Grammar:
{'S': ['aB', 'bA', 'A'], 'A': ['B', 'Sa', 'bBA', 'b'], 'B': ['b', 'bS', 'aD', 'eps'], 'D': ['AA'], 'C': ['Ba']}
1. After removing epsilon productions:
{'S': ['aB', 'bA', 'A', 'a'], 'A': ['B', 'Sa', 'bBA', 'b', '', 'bA'], 'B': ['b', 'bS', 'aD'], 'D': ['AA'], 'C': ['Ba']}
2. After removing unit productions:
{'S': ['aB', 'bA', 'a', 'Sa', 'bBA', 'b', '', 'bA', 'b', 'bS', 'aD'], 'A': ['Sa', 'bBA', 'b', '', 'bA', 'b', 'bS',
3. After removing inaccesible symbols:
{'S': ['aB', 'bA', 'a', 'Sa', 'bBA', 'b', '', 'bA', 'b', 'bS', 'aD'], 'A': ['Sa', 'bBA', 'b', '', 'bA', 'b', 'bS',
4. After removing non-productive symbols:
{'S': ['aB', 'bA', 'a', 'Sa', 'bBA', 'b', '', 'bA', 'b', 'bS', 'aD'], 'A': ['Sa', 'bBA', 'b', '', 'bA', 'b', 'bS',
5. Final CNF:
{'S': ['CE', 'FG', 'a', 'HC', 'FI', 'b', 'JJ', 'FG', 'b', 'FH', 'CK'], 'A': ['HC', 'FI', 'b', 'JJ', 'FG', 'b', 'FH'


Ran 5 tests in 0.030s

OK
```

Figure6.Result of unit tests

This output shows the runned tests on the grammar to ensure the correctness. The ‘ok’ message at the end indicates that all tests passed succesfully.

Conclusions

To sum up, in this laboratory work, we explored the idea of Chomsky Normal Form (CNF) and its importance in formal language theory in this lab work. A grammar defined by CNF has production rules that can be expressed in two ways: $A \rightarrow BC$ or $A \rightarrow c$, where c is a terminal symbol and A, B, and C are non-terminal symbols. One of the main benefits of CNF is its regularity: precisely $2n - 1$ steps are needed for each derivation of a string made up of n letters. This feature makes it possible to search through all possible combinations to see if a given text belongs in the language. By following procedures of Chomsky Normal Form, we convert grammars into CNF, which makes parsing and using them in computational tasks simpler. The grammar normalization approaches and their applications in language processing and recognition were clarified by this laboratory work. Learning Chomsky Normal Form gives us the basic tools we need to analyze and process formal languages, which improves our knowledge of computational linguistics and opens the door to the creation of more effective language processing systems and algorithms.