Ministry of Education and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work No. 5
Discipline: Formal Languages & Finite Automata
Topic: Chomsky Normal Form.

Done by: Barbarov Nadejda st.gr., FAF-221

Checked by:
asist.univ. Crețu Dumitru

Chișinău 2024

Theory

Chomsky Normal Form (CNF), introduced by Noam Chomsky, is a specific way to represent grammars in formal language theory. It's characterized by rules where each production is either of the form A -> BC or A -> a, where A, B, and C are non-terminal symbols, and a is a terminal symbol.

In the context of information theory, CNF plays a crucial role in simplifying and optimizing grammatical structures. By transforming a grammar into CNF, we reduce complexity and redundancy, making it easier to analyze and process languages algorithmically. This transformation aligns with the principles of information theory, as it aims to minimize uncertainty and increase the efficiency of representing linguistic information.

CNF also facilitates more efficient parsing algorithms, such as the CYK algorithm, which take advantage of the regular structure of CNF grammars to achieve faster parsing times. This aligns with the overarching goal of information theory to optimize communication and computation processes.

In summary, Chomsky Normal Form is a key concept in formal language theory that, when applied to grammars, helps streamline their representation and processing, aligning with the principles of information theory to optimize efficiency in language analysis and communication systems.

Objectives:

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

First of all, I defined the class *GrammarTransformer* and some variables that will be used next:

```
class GrammarTransformer:
    Nadejda Barbarov
    def __init__(self):
        self.left, self.right = 0, 1
        self.K, self.V, self.Productions = [], [], []
        self.variablesJar = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R",
                             "S", "T", "U", "W", "X", "Y", "Z"]
```

Figure 1. GrammarTransformer class

The next method, is *isUnitary*, this function checks if a production rule is unitary, meaning it involves a single variable transitioning to another single variable. It evaluates the left and right sides of the rule, ensuring they consist of variables and the right side contains only one variable.

```
def isUnitary(self, rule, variables):
    if rule[self.left] in variables and rule[self.right][0] in variables and len(rule[self.right]) == 1:
        return True
    return False
```

Figure 2. isUnitary function

Next, the *isSimple* function determines whether a production rule is simple, indicating a variable expands to a single terminal symbol. It examines if the left side is a variable, the right side contains a terminal symbol, and that only one symbol is on the right side.

```
def isSimple(self, rule):
    if rule[self.left] in self.V and rule[self.right][0] in self.K and len(rule[self.right]) == 1:
        return True
    return False
```

Figure 3. isSimple function

The function *START* adds a new start symbol 'S0' to the grammar, allowing it to initiate from the first variable. It modifies the list of productions to include the new start symbol, ensuring proper initialization of the grammar. By incorporating 'S0' into the grammar, it ensures a clear starting point for generating valid sentences.

```
def START(self, productions, variables):
    variables.append('S0')
    return [('S0', [variables[0]])] + productions
```

Figure 4. START function

TERM: By transforming the grammar, this function ensures all non-terminal symbols directly produce terminal symbols. It iterates through the productions, replacing terminal symbols with newly assigned variables, and updates the set of productions accordingly. Through this transformation process, it guarantees that terminal symbols are directly derivable from non-terminal symbols.

```
def TERM(self, productions, variables):
    newProductions = []
    dictionary = self.setupDict(productions, variables, terms=self.K)
    for production in productions:
        if self.isSimple(production):
            newProductions.append(production)
        else:
            for term in self.K:
                for index, value in enumerate(production[self.right]):
                    if term == value and term not in dictionary:
                        dictionary[term] = self.variablesJar.pop()
                        self.V.append(dictionary[term])
                        newProductions.append((dictionary[term], [term]))
                        production[self.right][index] = dictionary[term]
                    elif term == value:
                        production[self.right][index] = dictionary[term]
            newProductions.append((production[self.left], production[self.right]))
    return newProductions
```

Figure 5. TERM function

BIN: Transforming the grammar into binary form, this function guarantees each production rule has at most two symbols on the right-hand side. It introduces new variables as needed to split longer productions into binary segments, ensuring compliance with the binary format. By breaking down productions into binary components, it simplifies the grammar structure while maintaining its expressiveness.

```python
def BIN(self, productions, variables):
    result = []
    for production in productions:
        k = len(production[self.right])
        if k <= 2:
            result.append(production)
        else:
            newVar = self.variablesJar.pop(0)
            variables.append(newVar + '1')
            result.append((production[self.left], [production[self.right][0]] + [newVar + '1']))
            for i in range(1, k - 2):
                var, var2 = newVar + str(i), newVar + str(i + 1)
                variables.append(var2)
                result.append((var, [production[self.right][i], var2]))
            result.append((newVar + str(k - 2), production[self.right][k - 2:k]))
    return result
```

Figure 6. BIN function

DEL: In the elimination of epsilon productions, this function recursively removes empty productions from the grammar. It iteratively updates the set of production rules by seeking and destroying epsilon transitions, ensuring the resulting grammar is epsilon-free. Through this iterative process, it systematically eliminates epsilon transitions, refining the grammar to a more concise form.

```python
def DEL(self, productions):
    newSet = []
    outlaws, productions = self.seekAndDestroy(target='e', productions=productions)
    for outlaw in outlaws:
        for production in productions + [e for e in newSet if e not in productions]:
            if outlaw in production[self.right]:
                newSet = newSet + [e for e in self.rewrite(outlaw, production) if e not in newSet]
    return newSet + ([productions[i] for i in range(len(productions)) if productions[i] not in newSet])
```

Figure 7. DEL function

The *unit_routine* function finds and fixes unit productions. It looks at each rule to see if it's a unit production. If it is, it collects it. Then, it finds rules that can replace these units, making the grammar bigger.

```python
def unit_routine(self, rules, variables):
    unitaries, result = [], []
    for aRule in rules:
        if self.isUnitary(aRule, variables):
            unitaries.append((aRule[self.left], aRule[self.right][0]))
        else:
            result.append(aRule)
    for uni in unitaries:
        for rule in rules:
            if uni[self.right] == rule[self.left] and uni[self.left] != rule[self.left]:
                result.append((uni[self.left], rule[self.right]))
    return result
```

Figure 8. unit_routine function

A simpler function, *union*, just combines two lists. It makes sure there are no repeating things in the new list.

```python
def union(self, lst1, lst2):
    final_list = list(set().union(lst1, lst2))
    return final_list
```

Figure 9. union function

loadModel reads a grammar model from a file. It pulls out important stuff like variables, terminals, and productions. Then, it cleans up this info and puts it in the right spots.

```python
def loadModel(self, modelPath):
    file = open(modelPath).read()
    K = (file.split("Variables:\n")[0].replace("Terminals:\n", "").replace("\n", ""))
    V = (file.split("Variables:\n")[1].split("Productions:\n")[0].replace("Variables:\n", "").replace("\n", ""))
    P = (file.split("Productions:\n")[1])
    self.K = self.cleanAlphabet(K)
    self.V = self.cleanAlphabet(V)
    self.Productions = self.cleanProduction(P)
```

Figure 10. loadModel function

cleanProduction gets the rules from the file. It separates the left side from the right side. It also splits the right side into separate terms if needed.

```python
def cleanProduction(self, expression):
    result = []
    rawRulse = expression.replace('\n', '').split(';')
    for rule in rawRulse:
        leftSide = rule.split(' -> ')[0].replace(' ', '')
        rightTerms = rule.split(' -> ')[1].split(' | ')
        for term in rightTerms:
            result.append((leftSide, term.split(' ')))
    return result
```

Figure 11. cleanProductions function

cleanAlphabet cleans up the letters in the grammar. It makes sure there aren't extra spaces and makes a list of them.

```python
def cleanAlphabet(self, expression):
    return expression.replace('  ', ' ').split(' ')
```

Figure 12. cleanAlphabet function

seekAndDestroy finds and gets rid of rules with a certain symbol. It checks if the symbol is by itself or the only thing on the right side.

```python
def seekAndDestroy(self, target, productions):
    trash, erased = [], []
    for production in productions:
        if target in production[self.right] and len(production[self.right]) == 1:
            trash.append(production[self.left])
        else:
            erased.append(production)
    return trash, erased
```

Figure 13. seekAndDestroy function

setupDict makes a list that connects symbols with their meanings. It uses the rules and symbols in the grammar to make this list.

```
def setupDict(self, productions, variables, terms):
    result = {}
    for production in productions:
        if production[self.left] in variables and production[self.right][0] in terms and len(
                production[self.right]) == 1:
            result[production[self.right][0]] = production[self.left]
    return result
```

Figure 14. setupDict function

rewrite makes new rules by changing one symbol with others. It looks at where the symbol is and makes different rules without it.

```
def rewrite(self, target, production):
    result = []
    positions = [i for i, x in enumerate(production[self.right]) if x == target]
    for i in range(len(positions) + 1):
        for element in list(itertools.combinations(positions, i)):
            tadan = [production[self.right][i] for i in range(len(production[self.right])) if i not in element]
            if tadan != []:
                result.append((production[self.left], tadan))
    return result
```

Figure 15. rewrite function

dict2Set turns a list into pairs. It takes each thing in the list and pairs it with something else.

```
def dict2Set(self, dictionary):
    result = []
    for key in dictionary:
        result.append((dictionary[key], key))
    return result
```

Figure 16. dict2Set function

pprintRules just prints out the rules in a nice way. It puts each rule on its own line.

```
def pprintRules(self, rules):
    for rule in rules:
        tot = ""
        for term in rule[self.right]:
            tot = tot + " " + term
        print(rule[self.left] + " -> " + tot)
```

Figure 17. pprintRules function

prettyForm makes the rules look nice and neat. It groups rules with the same starting symbol together.

```
def prettyForm(self, rules):
    dictionary = {}
    for rule in rules:
        if rule[self.left] in dictionary:
            dictionary[rule[self.left]] += ' | ' + ' '.join(rule[self.right])
        else:
            dictionary[rule[self.left]] = ' '.join(rule[self.right])
    result = ""
    for key in dictionary:
        result += key + " -> " + dictionary[key] + "\n"
    return result
```

Figure 18. prettyForm function

The *UNIT* function keeps going until it can't fix any more unit rules. It uses the *unit_routine* to do this over and over again.

```
def UNIT(self, productions, variables):
    i = 0
    result = self.unit_routine(productions, variables)
    tmp = self.unit_routine(result, variables)
    while result != tmp and i < 1000:
        result = self.unit_routine(tmp, variables)
        tmp = self.unit_routine(result, variables)
        i += 1
    return result
```

Figure 19. UNIT function

Finally, *transform_grammar* does all the work to change the grammar. It reads the grammar from a file, fixes it up with different functions, and then makes it look nice and tidy before giving it back.

```
def transform_grammar(self, modelPath):
    self.loadModel(modelPath)

    self.Productions = self.START(self.Productions, variables=self.V)
    self.Productions = self.TERM(self.Productions, variables=self.V)
    self.Productions = self.BIN(self.Productions, variables=self.V)
    self.Productions = self.DEL(self.Productions)
    self.Productions = self.UNIT(self.Productions, variables=self.V)

    return self.prettyForm(self.Productions)
```

Figure 20. transform_grammar function

Next, there's a test class named *TestGrammarTransformer* that's using the unittest framework to test the *transform_grammar* function of the GrammarTransformer class.

First, the *setUp* method initializes a *GrammarTransformer* instance before each test to ensure a clean environment.

```
class TestGrammarTransformer(unittest.TestCase):
    def setUp(self):
        self.transformer = GrammarTransformer()
```

The *test_transform_grammar* method verifies if the *transform_grammar* function correctly transforms a grammar model stored in a file named 'model.txt'. It defines the input model path and the expected output after transformation. Then, it calls the *transform_grammar* function and checks if the output matches the expected result.

```
def test_transform_grammar(self):
    # Define input model path
    model_path = 'model.txt'
```

Similarly, the *test_transform_grammar_model2* method tests the transformation of a different grammar model stored in 'model2.txt'. It sets up a different input model path and expected output, then runs the transformation function and checks the result.

Finally, the block of code starting with *if __name__ == '__main__'*: ensures that the tests run when the script is executed directly. It initializes the GrammarTransformer, checks if a model path is provided as a command-line argument, then transforms the grammar using the specified or default model path. After that, it runs the unit tests using unittest.main() to ensure the correctness of the transform_grammar function for different input grammar models.

Conclusions / Screenshots / Results

In conclusion, this lab taught me about Chomsky Normal Form (CNF) and how to standardize a grammar. By learning and implementing CNF rules, I've gained skills in transforming grammars to a common format. Putting this into a structured method or class makes it easier to use again. Testing what I've done ensures it works well.

```
S -> B A | b | B S
A -> A S | b | Z D | B A1 | B A | b | B S | b | B S
A1 -> A A2 | A B | b | B S
A2 -> A B | b | B S
C -> A B | b | B S
Z -> a
B -> b | B S
D -> B B
S0 -> B A | b | B S
```

Figure 21. Answer

Also, adding tests to check if everything works as expected shows I'm careful about quality. Making the method able to handle any grammar, not just one type, makes it more useful in different situations. Overall, this lab helped me learn grammar rules and how to apply them practically, making me better at problem-solving in programming.

```
Launching unittests with arguments python -m unittest C:\U

Ran 1 test in 0.003s

OK

Process finished with exit code 0

Tests passed: 1                                        21:57
```

Figure 22. Test results

References

1. **Wikipedia.** https://en.wikipedia.org/wiki/Chomsky_normal_form
2. **Github.**https://github.com/filpatterson/DSL_laboratory_works/blob/master/3_LexerScanner/task.md
3. **Github.** https://github.com/nadea-b/LFA