MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA

TECHNICAL UNIVERSITY OF MOLDOVA

FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Laboratory work 5:
Chomsky Normal Form

Course: Formal Languages & Finite Automata

Author: Bostan Victor, FAF-222

Chișinău, 2024

THEORY

Theory of Chomsky Normal Form (CNF)

Chomsky Normal Form is a stringent way of expressing a Context-Free Grammar (CFG) that facilitates both the theoretical study and practical parsing of languages. A grammar in CNF is particularly powerful for parsing algorithms like the CYK algorithm. In CNF, every production rule adheres to one of the following forms: A production is binary (two non-terminals on the right-hand side), unary (a single terminal on the right-hand side), or allows the start symbol to produce the empty string directly. The conversion to CNF is essential in computational linguistics, automata theory, and algorithm design for language parsing.

Conversion from CFG to CNF

1. Start Symbol Removal: A new start symbol is introduced to prevent the original start symbol from appearing on the right side of any production, thus avoiding potential recursion issues that could violate CNF structure.
2. Null Production Removal: All productions that generate an empty string (ε) are eliminated. This involves expanding the grammar to account for scenarios where these null productions might have been used, by creating additional productions that omit the nullable non-terminals.
3. Unit Production Removal: Productions where a non-terminal leads directly to another non-terminal (unit productions) are removed. This is done by substituting the unit productions with the set of productions that the target non-terminal can generate.
4. Terminals in Mixed Productions: For productions containing both terminals and non-terminals, the terminals are replaced with new non-terminals which then produce these terminals. This ensures that terminals only appear in unary productions.
5. Long Production Reduction: Productions with more than two symbols are systematically shortened by introducing new non-terminals that help decompose the original production into a series of binary productions.

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

For this laboratory we are given a context free grammar and are given the task to create a code implementation which would convert it into chomsky normal form. But I decided to go beyond my variant and create an implementation which would accept any context free grammar and return the correct conversion to CNF. Anyway, here is my variant, Variant 1:

Variant 1

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

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

P={
1. S→aB
2. S→AC
3. A→a
4. A→ASC
5. A→BC
6. A→aD
7. B→b
8. B→bS
9. C→ε
10. C→BA
11. E→aB
12. D→abC}

So, let's get to the actual implementation. The first thing worth to mention is that all the conversion logic is encapsulated in the Grammar class, which extends from the GenericGrammar class that I defined back in the second laboratory. In the GenericGrammar class i defined the constructor of the grammar and a print grammar function:

```
3   class GenericGrammar:
4       def __init__(self, vn, vt, p, s):
5           self.Vn = vn
6           self.Vt = vt
7           self.P = p
8           self.S = s
9
10      def print_grammar(self):
11          print('Vn:', self.Vn)
12          print('Vt:', self.Vt)
13          print('P:')
14          for key, value in self.P.items():
15              print(f'{key} -> {value}')
16          print('S: ', self.S, '\n')
17
```

Now, here is the Grammar class and the main method that I created for this laboratory *cfg_to_cnf* which is basically a method that calls all the other methods (steps for the conversion defined in the theory part) necessary to convert CFG to CNF:

```
class Grammar(GenericGrammar):
    def cfg_to_cnf(self):
        self.start_symbol_rhs_removal()
        self.remove_null_productions()
        self.remove_unit_productions()
        self.remove_inaccessible_symbols()
        self.replace_terminals_with_nonterminals()
        self.reduce_production_length()
```

Basically, after all these methods are called our grammar will be converted to CNF. Now, let's showcase each step or method. The first one is *start_symbol_rhs_removal*:

```
    def start_symbol_rhs_removal(self):
        try:
            for value in self.P.values():
                for production in value:
                    for character in production:
                        if character == self.S:
                            raise BreakFromLoops
        except:
            new_P = {'X': [self.S]}
            new_P.update(self.P)
            self.P = new_P
            self.S = 'X'
            self.Vn.append(self.S)
```

This method addresses a specific requirement of Chomsky Normal Form (CNF): the start symbol should not appear on the right-hand side (RHS) of any production. This is to prevent possible recursion issues and to keep the grammar in line with CNF conventions. The method iterates through the production rules of the grammar, and upon detecting the start symbol on the RHS, it introduces a new start symbol 'X'. This new symbol is then assigned a production rule that simply produces the original start symbol. The grammar's production set and the list of non-terminals are updated accordingly to include this new start symbol and its rule. It is also important to note that i used a try except block with a raise, in order to break from all three for loops when a start symbol is found in rhs. BreakFromLoops doesn't actually output anything as it is an empty class I created, it is just used to break from the three loops at the same time without the need to use *break* three times.

```
    def remove_null_productions(self):
        count_null = 0
        null_prods = []
        for key, value in self.P.items():
            for production in value:
                if production == 'ε':
                    count_null += 1
                    null_prods.append(key)
                    value.remove(production)
        for i in range(count_null):
            new_P = self.P
            for key, value in self.P.items():
                for production in value:
                    for character in production:
                        if character == null_prods[i]:
                            new_productions = self.create_new_productions(production, null_prods[i])
                            for j in new_productions:
                                if j not in new_P[key]:
                                    new_P[key].append(j)
                            break
            self.P = new_P
```

The `remove_null_productions` method is crucial for conforming to the Chomsky Normal Form, which does not permit ε-productions (productions that generate an empty string) except for the start symbol producing ε directly. This method systematically identifies and eliminates ε-productions from the grammar.

First, the method traverses all production rules, tracking each non-terminal that leads to ε and removing such ε-productions from the grammar. Then, it addresses the indirect impact of these removed ε-productions. For each production that contains a non-terminal leading to ε, new variants of the production are generated by the `create_new_productions` method. These new variants represent all possible strings that can be generated by omitting the nullable non-terminal. This step is critical because removing an ε-production may require compensatory modifications elsewhere in the grammar to maintain the same language generation capability.

For instance, if we have a production `A → aBc` and `B` can produce ε, we must add a new production `A → ac` to preserve the language.

```
    def create_new_productions(self, production, character):
        results = []
        for i in range(len(production)):
            if production[i] == character:
                new_production = production[:i] + production[i+1:]
                results.append(new_production)
        return results
```

The `create_new_productions` function facilitates this by creating new productions without the nullable non-terminal, ensuring that every combination is considered. This meticulous approach allows the grammar to maintain its generative capacity without relying on ε-productions, aligning the grammar with the restrictions of CNF. Now the next step is removing unit productions:

```
60      def remove_unit_productions(self):
61          for key, value in self.P.items():
62              for production in value:
63                  if key == production:
64                      self.P[key].remove(production)
65          changes = True
66          while changes:
67              changes = False
68              for key, value in self.P.items():
69                  for production in value:
70                      if production in self.Vn:
71                          changes = True
72                          self.P[key].remove(production)
73                          for prod in self.P[production]:
74                              if prod not in self.P[key]:
75                                  self.P[key].append(prod)
```

The `remove_unit_productions` method is a critical step in the process of converting a Context-Free Grammar (CFG) into Chomsky Normal Form (CNF). Unit productions are rules where a non-terminal symbol produces another single non-terminal symbol, like A → B. These are not allowed in CNF except when the production is the start symbol producing a single non-terminal.

The method first scans all productions in the grammar and directly removes any self-referencing unit productions, i.e., productions where a non-terminal produces itself, such as A → A . It then iteratively processes the remaining unit productions. For each unit production found, it replaces the unit production with the productions of the non-terminal it points to. This substitution ensures that the language generated by the grammar remains unchanged while aligning it with the CNF structure, which dictates that any production must either lead to exactly two non-terminals, one terminal, or be the start symbol producing an empty string.

During this process, the method maintains a flag to track whether any changes have been made in the current iteration. If at least one unit production was replaced, it continues for another iteration to ensure that no indirect unit productions are left unresolved. For instance, if we remove a unit production A → B and B itself had a production B → C, we must also ensure that A now produces whatever C can produce. This looping continues until no further unit productions are detected, at which point the grammar will have no unit productions left, satisfying another important criterion for CNF.

```
77      def remove_inaccessible_symbols(self):
78          accessible = set([self.S])
79          queue = [self.S]
80
81          while queue:
82              current = queue.pop(0)
83              for production in self.P.get(current, []):
84                  for symbol in production:
85                      if symbol in self.Vn and symbol not in accessible:
86                          accessible.add(symbol)
87                          queue.append(symbol)
88
89          self.Vn = [nt for nt in self.Vn if nt in accessible]
90          for nt in list(self.P.keys()):
91              if nt not in accessible:
92                  del self.P[nt]
```

The `remove_inaccessible_symbols` method is an essential step toward refining a Context-Free Grammar (CFG) into Chomsky Normal Form (CNF). It eliminates non-terminal symbols that are not accessible from the start symbol, which means they can never appear in any derivation of a string in the language defined by the grammar.

The method employs a breadth-first search (BFS) strategy starting from the start symbol. It uses a queue to explore all production rules reachable from the start symbol and accumulates accessible symbols in a set. Each time it processes a symbol from the queue, it examines all production rules where that symbol is on the left-hand side and adds the symbols on the right-hand side to the queue if they haven't been encountered before. This way, it effectively traverses the "graph" of production rules, discovering all symbols that contribute to the language generation.

After this traversal, the grammar is pruned to contain only the non-terminals that are accessible. It removes any non-terminal symbols that do not have a path from the start symbol and any production rules that contain these inaccessible symbols. By doing so, the method ensures that the grammar is left with only the functional "core" necessary to produce the language, with all superfluous or unreachable symbols discarded. This step does not alter the language defined by the grammar but simplifies the grammar by removing extraneous elements, facilitating the subsequent steps of the CNF conversion and making the grammar more efficient for parsing algorithms.

```
    def replace_terminals_with_nonterminals(self):
        def new_nonterminal(existing):
            for char in (chr(i) for i in range(65, 91)):
                if char not in existing:
                    return char
            raise ValueError("Ran out of single-letter nonterminal symbols!")

        terminal_to_nonterminal = {}
        new_P = {}
        for key, productions in self.P.items():
            new_productions = []
            for prod in productions:
                if len(prod) > 1:
                    new_prod = ''
                    for char in prod:
                        if char in self.Vt:
                            if char not in terminal_to_nonterminal:
                                new_nt = new_nonterminal(self.Vn)
                                self.Vn.append(new_nt)
                                terminal_to_nonterminal[char] = new_nt
                                new_P[new_nt] = [char]
                            new_prod += terminal_to_nonterminal[char]
                        else:
                            new_prod += char
                    new_productions.append(new_prod)
                else:
                    new_productions.append(prod)
            new_P[key] = new_productions
        self.P.update(new_P)
```

The `replace_terminals_with_nonterminals` method within the conversion process to Chomsky Normal Form (CNF) addresses a specific constraint of CNF: if a production rule has more than one symbol on the right-hand side (RHS), then all symbols must be non-terminals. Therefore, this method systematically replaces terminals in such production rules with new non-terminal symbols which, in turn, produce the original terminals.

The method begins by iterating through all production rules in the given grammar. When it encounters a production with a length greater than one that includes terminals, it proceeds to substitute each terminal with a corresponding non-terminal. To ensure that each terminal has a unique and previously unused non-terminal, the method employs a helper function `new_nonterminal` that generates new non-terminal symbols based on the ASCII uppercase letters, avoiding any symbols already in use.

For each terminal found, the method checks if it has already been replaced by a new non-terminal. If not, a new non-terminal is created, added to the grammar's list of non-terminals, and mapped to produce the terminal. Then, the original terminal in the production rule is replaced with this new non-terminal.

Once all necessary replacements are made, the grammar's set of production rules is updated to reflect these changes. This substitution does not alter the language generated by the grammar; it merely ensures that all production rules conform to the structural requirements of CNF. Consequently, after this method has been applied, the grammar will only have production rules that either consist of exactly one terminal or sequences of non-terminals, moving a significant step closer to being fully in Chomsky Normal Form.

```
125     def reduce_production_length(self):
126         def new_nonterminal(existing):
127             for char in (chr(i) for i in range(65, 91)):
128                 if char not in existing:
129                     return char
130             raise ValueError("Ran out of single-letter nonterminal symbols!")
131 
132         existing_binaries = {}
133         new_productions_dict = {}
134         for key, productions in list(self.P.items()):
135             new_productions = []
136             for production in productions:
137                 if len(production) > 2:
138                     while len(production) > 2:
139                         last_two = production[-2:]
140                         if last_two not in existing_binaries:
141                             new_nt = new_nonterminal(set(self.Vn) | set(new_productions_dict.keys()))
142                             self.Vn.append(new_nt)
143                             new_productions_dict[new_nt] = [last_two]
144                             existing_binaries[last_two] = new_nt
145                         production = production[:-2] + existing_binaries[last_two]
146                     new_productions.append(production)
147                 else:
148                     new_productions.append(production)
149             self.P[key] = new_productions
150         self.P.update(new_productions_dict)
```

The `reduce_production_length` method is the final step in the process of converting a Context-Free Grammar (CFG) into Chomsky Normal Form (CNF). This method ensures that all production rules in the grammar conform to one of the key stipulations of CNF: each production rule is either a single terminal or consists of exactly two non-terminals.

The functionality of this method is to systematically reduce the length of any production rules that exceed two symbols on the right-hand side (RHS). Here's how it accomplishes this:

1. Initialization: The method first sets up a helper function `new_nonterminal` to generate new non-terminal symbols that have not been used in the grammar so far. It also initializes two dictionaries: `existing_binaries` to track binary combinations of symbols that have already been replaced by a non-terminal, and `new_productions_dict` to store new production rules created during the process.
2. Production Rule Evaluation: The method iterates through each production rule in the grammar. If a production rule has more than two symbols on the RHS, it enters a loop to reduce this length.

3. Binary Reduction Process: Inside the loop, the method focuses on the last two symbols of the current production. If this pair of symbols hasn't been replaced by a non-terminal in previous iterations, a new non-terminal is created specifically for this pair, and a new production rule (non-terminal → last two symbols) is added to the grammar. The new non-terminal then replaces the pair in the original production rule.
4. Update Production Rule: This replacement shortens the original production by one symbol at a time. This process repeats until the length of the production is reduced to two symbols, ensuring it adheres to the CNF format.
5. Grammar Update: After processing all productions of a non-terminal, the updated list of productions (including newly created binary productions) replaces the old list in the grammar's production set.
6. Final Integration: At the end of the method, any new production rules stored in `new_productions_dict` are merged into the main production dictionary of the grammar, ensuring that all production rules are accessible and usable in the grammar.

This method is critical because it finalizes the structure of the grammar ensuring that it strictly follows the CNF requirements.

Testing and Results

To validate the functionality and correctness of the implementation that converts a Context-Free Grammar (CFG) into Chomsky Normal Form (CNF), a comprehensive set of unit tests was developed and executed using Python's `unittest` framework. These tests play a pivotal role in ensuring that each step of the conversion process is performed accurately, and that the overall system behaves as expected under various scenarios. Here's a concise overview of the testing strategy:

Setup for Testing

```
153  class TestGrammarMethods(unittest.TestCase):
154      def setUp(self):
155          self.grammar = Grammar(['S', 'A', 'B', 'C', 'D', 'E'], ['a', 'b'], {
156              'S': ['aB', 'AC'],
157              'A': ['a', 'ASC', 'BC', 'aD'],
158              'B': ['b', 'bS'],
159              'C': ['ε', 'BA'],
160              'E': ['aB'],
161              'D': ['abC']
162          }, 'S')
```

Each test begins with the instantiation of the `Grammar` class within the `setUp` method. This method pre-configures a grammar that is then used across multiple test cases, allowing for repeated application of the transformation methods on a consistent initial state. The pre-configured grammar includes a mix of production types to rigorously test the conversion logic under diverse conditions.

Testing Steps and Validation

Start Symbol Removal:

```
164     def test_start_symbol_rhs_removal(self):
165         self.grammar.start_symbol_rhs_removal()
166         for prod in self.grammar.P.values():
167             self.assertNotIn(self.grammar.S, prod)
168
```

`test_start_symbol_rhs_removal` checks if the start symbol has been effectively removed from the RHS of all productions, ensuring it does not appear in any production rule except in its initial production.

Null Productions Elimination:

```
169     def test_remove_null_productions(self):
170         self.grammar.remove_null_productions()
171         for prods in self.grammar.P.values():
172             self.assertNotIn('ε', prods)
```

`test_remove_null_productions` validates that all ε-productions are successfully removed from the grammar, except where allowed by CNF rules. The test confirms that no production rule ends up producing an empty string erroneously.

Unit Productions Elimination:

```
174     def test_remove_unit_productions(self):
175         self.grammar.remove_unit_productions()
176         for key, prods in self.grammar.P.items():
177             for prod in prods:
178                 self.assertFalse(len(prod) == 1 and prod.isupper())
179
```

`test_remove_unit_productions` ensures that no unit productions (productions where a nonterminal produces a single nonterminal) remain in the grammar, as these are not permitted in CNF unless explicitly required.

Accessibility of Symbols:

```
180     def test_remove_inaccessible_symbols(self):
181         self.grammar.Vn.append('Z')
182         self.grammar.remove_inaccessible_symbols()
183         self.assertNotIn('Z', self.grammar.Vn)
184
```

`test_remove_inaccessible_symbols` asserts that all nonterminals that are not reachable from the start symbol are removed. This test is crucial for verifying that the grammar does not contain superfluous symbols which do not contribute to the language generation.

Terminal Replacement:

```
185     def test_replace_terminals_with_nonterminals(self):
186         self.grammar.replace_terminals_with_nonterminals()
187         for prods in self.grammar.P.values():
188             for prod in prods:
189                 if len(prod) > 1:
190                     self.assertTrue(all(char in self.grammar.Vn for char in prod))
191
```

`test_replace_terminals_with_nonterminals` checks that in any production containing both terminals and non-terminals, terminals are correctly replaced by new non-terminals which produce the terminals. This step ensures that the resulting grammar adheres to the structure required by CNF.

Production Length Reduction:

```
192     def test_reduce_production_length(self):
193         self.grammar.replace_terminals_with_nonterminals()
194         self.grammar.reduce_production_length()
195         for prods in self.grammar.P.values():
196             for prod in prods:
197                 self.assertTrue(len(prod) <= 2)
```

`test_reduce_production_length` confirms that all productions are reduced to the correct length, adhering to CNF standards where each production is either two non-terminals or a single terminal.

Output Verification:

```
199     def test_grammar_print(self):
200         self.grammar.cfg_to_cnf()
201         self.grammar.print_grammar()
202
```

In addition to the functional tests, the `test_grammar_print` method calls the `print_grammar` function after converting the initial CFG to CNF. This output is used to manually verify that the transformed grammar visually conforms to CNF expectations and provides a quick, human-readable way to inspect the results of the transformation process.

So, here are the results for my variant:

```
Vn: ['S', 'A', 'B', 'C', 'D', 'X', 'E', 'F', 'G', 'H']
Vt: ['a', 'b']
P:
X -> ['EB', 'AC', 'a', 'AG', 'BC', 'ED', 'AS', 'b', 'FS']
S -> ['EB', 'AC', 'a', 'AG', 'BC', 'ED', 'AS', 'b', 'FS']
A -> ['a', 'AG', 'BC', 'ED', 'AS', 'b', 'FS']
B -> ['b', 'FS']
C -> ['BA']
D -> ['EH', 'EF']
E -> ['a']
F -> ['b']
G -> ['SC']
H -> ['FC']
S:  X

.......
----------------------------------------------------------------------
Ran 7 tests in 0.002s

OK
```

CONCLUSION

This laboratory work provided an in-depth exploration into converting Context-Free Grammars (CFG) into Chomsky Normal Form (CNF), highlighting the methodical approach required to standardize grammars for computational use. Through the implementation of various transformation functions—such as removing the start symbol from production rules, eliminating null and unit productions, replacing terminals in mixed contexts, and reducing production lengths—I gained a comprehensive understanding of CNF's structural rules and its significance in parsing and automata theory. Despite the complexity of these transformations, the systematic procedure and robust testing ensured a thorough comprehension and effective application, reinforcing CNF's essential role in simplifying grammars for algorithmic processing. This lab not only enhanced my theoretical knowledge but also honed my practical skills in grammar manipulation, preparing me for advanced tasks in language processing and computational linguistics.