Laboratory Work No. 5

Course: Formal Languages & Finite Automata

Student: Iordan Liviu, Group: FAF-223

Topic: Chomsky Normal Form

Objectives

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

Introduction

Chomsky Normal Form: in formal language theory, a context-free grammar, G, is said to be in Chomsky normal form (first described by Noam Chomsky) if all of its production rules are of the form:

$A \to BC$, or

$A \to a$, or

$S \to \varepsilon$,

Implementation description

The laboratory work on finite languages and automata encompasses a set of Python scripts designed to explore fundamental concepts in formal language theory and automata theory. The codebase consists of two main modules: grammar.py and main.py. In grammar.py, the Grammar class is implemented, which serves as a versatile tool for defining, manipulating, and analyzing grammars. This class allows users to specify sets of non-terminal (V_n) and terminal (V_t) symbols, production rules (P), and a start symbol (S), either interactively through user input or directly through initialization parameters. The Grammar class provides various methods for checking and transforming grammars, such as identifying grammar types (e.g., regular, context-free) based on production rules and symbol usage. Noteworthy operations include eliminating epsilon-productions, unit-productions, unproductive symbols, and converting grammars to Chomsky Normal Form (CNF) where applicable. Additionally, the main.py script demonstrates the usage of the Grammar class by defining a sample grammar, printing its components, checking its grammar type, and converting it to CNF. This comprehensive approach to formal language analysis facilitates learning and experimentation with theoretical concepts in computational linguistics and theoretical computer science.

Also, to better see how well the code works, I have implemented some test cases using the unittest library in python.Here is a breakdown of the test cases there were implemented:

Test Case: test_epsilon_elimination

- This test case evaluates the elimination of epsilon productions ("\u03B5") from a grammar.
- It includes two test scenarios:
    - Test Case 1: Grammar with epsilon production.
    - Test Case 2: Grammar without epsilon production.

Test Case: test_inaccessible_elimination

- This test case checks the removal of inaccessible symbols from a grammar.
- It includes two scenarios:
    - Test Case 1: Grammar with inaccessible symbols.
    - Test Case 2: Grammar without inaccessible symbols.

Test Case: test_unit_production_elimination

- This test case focuses on the elimination of unit productions from a grammar.
- It contains two sub-tests:
    - Test Case 1: Grammar with unit productions.
    - Test Case 2: Grammar without unit productions.

Test Case: test_chomsky_normal_form_conversion

- This test case verifies the conversion of a grammar to Chomsky Normal Form (CNF).
- It includes two test scenarios:
    - Test Case 1: Grammar suitable for CNF conversion.
    - Test Case 2: Grammar not suitable for CNF conversion.

I have incorporated an automatic grammar type-checking feature into the implementation. This feature assigns a grammar type during the construction of a `Grammar` object. The assigned type is then utilized within the algorithm to determine if the grammar is a type 2 - Context-Free Grammar, which is necessary for conversion into Chomsky Normal Form. This enhancement involved introducing a new variable in the constructor, which is optional as it will always be initialized during grammar construction.

```python
class Grammar:
    def __init__(self, V_n=None, V_t=None, P=None, S=None, type=None):
        self.type_grammar = type
        if V_n is None or V_t is None or P is None or S is None:
            self.create_grammar()
        else:
            self.V_n = V_n
            self.V_t = V_t
            self.P = P
            for v in self.P.values():
                for derive in v:
                    if derive == "epsilon":
                        v.remove(derive)
                        v.add("\u03B5")
            self.S = S
            if type is None:
                self.check_type_grammar()
```

To accomplish the task, I have followed the model provided on our LFA else course about Chomsky Normal Form examples:

- Step 1 - Eliminate ε-productions:

a) $N_\varepsilon = \varnothing$

b) for the production A→ ε $N_\varepsilon = \varnothing \cup \{A\}$

$N_\varepsilon = \{A\}$

- Step 2 - Eliminate unit-productions:

The production that has the form X→Y, X and Y are nonterminal, is called renaming.

The renaming from P' are: S→B, S→C, D→B

$R_S = \{S\},\ R_B = \{B\},\ R_C = \{C\},\ R_D = \{D\}$

- Step 3 - Eliminate unproductive-productions:

$$PROD(G) = \{A \mid A \in V_N, \exists A \Rightarrow v, v \in V_T\}$$
$$NEPROD(G) = V_N \setminus PROD(G)$$
$$V_N = \{S, A, B, C, D\}$$
$$PROD(G) = \{B, S, A, D\}$$
$$NEPROD(G) = \{S, A, B, C, D\} \setminus \{B, S, A, D\} = \{C\}$$

- Step 4 - Eliminate inaccessible symbols

Initial $ACCES(G) = \{S\}$

$$ACCES(G) = \{x \mid \exists S \Rightarrow \alpha x \beta\}$$
$$INACCES(G) = (V_N \cup V_T) \setminus ACCES(G)$$
$$ACCES(G) = \{S, A, b, a, B\}$$
$$V_N = \{S, A, B, D\} \qquad V_T = \{a, b\}$$
$$INACCES(G) = \{S, A, B, D, a, b\} \setminus \{S, A, b, a, B\} = \{D\}$$

- Step 5 - Perform conversion to Chomsky Normal Form:

A grammar in the Chomsky Normal Form is a grammar of rules that has a form A→ BC, D→i, where A,B,C,D $\in V_N$ and i $\in V_T$

Code Explanation

1. Convert to Chomsky Normal Form function:

The `convert_to_Chomsky_Normal_Form` method is designed to transform a grammar into Chomsky Normal Form (CNF), specifically for grammars identified as type 2 - Context-Free Grammars (CFGs). The method proceeds with several stages of transformation. First, it eliminates epsilon (\u03B5) productions from the grammar by invoking `eliminate_epsilon_productions`. Next, it eliminates unit productions using `eliminate_unit_productions`. Then, it removes unproductive symbols and inaccessible symbols using `eliminate_unproductive_symbols` and `eliminate_inaccessible_symbols`, respectively. After these initial transformations, the method performs further conversions to achieve CNF. It constructs new sets of terminal and non-terminal symbols, initializes a new start symbol, and transforms productions into tuples. The method iteratively applies additional transformations to the grammar until no new rules are generated. Throughout this process, detailed logging and printing of rule modifications are provided for clarity. Finally, the resulting CNF grammar is displayed, and a new `Grammar` object with the converted grammar is returned. If the input grammar is not a CFG (type 2), the method prints an error message indicating the impossibility of conversion and returns `None`.

```python
def convert_to_Chomsky_Normal_Form(self):

   if self.type_grammar == 2:

       print("\nPerforming Conversion to Chomsky Normal Form")

       print("\nEliminating the \u03B5-productions")

       new_P = self.eliminate_epsilon_productions()

       print("\nEliminating the unit-productions...")

       new_P = self.eliminate_unit_productions(new_P)

       print("Eliminating the unproductive Symbols...")

       new_P, new_V_n = self.eliminate_unproductive_symbols(new_P)

       print("\nEliminating the inaccessible symbols...")

       new_P, new_V_n = self.eliminate_inaccessible_symbols(new_P, new_V_n)
```

```
        print("\nConverting to Chomsky Normal Form...")

        print("CFG IN CHOSMKY NORMAL FORM:")

         print("S =", new_S)

         print("V_t =", new_V_t)

        new_V_n = set(new_P.keys())

         print("V_n =", new_V_n)

         print("P = {")

        for (k, v) in new_P.items():

            print("  " + k, "->", v)

          print("}")

         return Grammar(V_n=new_V_n, V_t=new_V_t, P=new_P, S=new_S,
type=self.type_grammar)

        else:

        print("Grammar is not of type 2! Can't convert to Chomsky Normal Form!")

        return None
```

2. Eliminate Epsilon Productions functions:

The method `eliminate_epsilon_productions` is designed to handle the removal of ε-productions (productions that derive ε) from a given grammar represented by a set of production rules (`self.P`). The function initializes an empty dictionary `new_P` to store the modified production rules without ε-productions and a set `set_nullable_symbols` to keep track of symbols that can derive ε. It iterates over each production rule, checking if any production is ε (`"\u03B5"`), and adds the corresponding left-hand side (LHS) symbol to `set_nullable_symbols`. It then enters a loop to iteratively identify all symbols that can derive ε transitively by examining the existing set of nullable symbols. For each nullable symbol identified, it processes the associated productions, generating new productions by systematically replacing occurrences of the nullable symbol with empty strings. The resulting modified productions are then updated in `new_P`. Finally, the method prints the set of nullable symbols and the resulting production rules without ε-productions. The modified production rules (`new_P`) are returned as the output of this method.

```
Def eliminate_epsilon_productions(self)
    new_P = {}
    # Declare empty set that will hold symbols from LHS that derive into ε
    set_nullable_symbols = set()
    # Iterate over all the production Rules
    for (LHS, RHS) in self.P.items():
        # Iterate over all the RHS possible derivations
        for production in RHS:
            # If derivation is ε, then add this symbol to the set from above
            if production == "\u03B5":
                set_nullable_symbols.add(LHS)
            else:
                if LHS not in new_P:
                    new_P[LHS] = {production}
                else:
                    new_P[LHS].add(production)
```

3. Eliminate Unit Productions function:

The `eliminate_unit_productions` method in the Grammar class aims to remove unit productions from a given set of production rules (`new_P`). It iterates through the production rules until no unit productions are left. During each iteration, it checks if there are any unit productions (single non-terminal on the right-hand side) and replaces them with the corresponding productions they derive. This process is repeated until no further unit productions are found. After completing the iterations, it calculates the difference between the original and modified production rules to highlight any changes made. If there are differences (indicating unit productions were removed), it prints the updated set of production rules. Otherwise, it confirms that no unit productions were removed and displays the unchanged set of production rules. The final result is the modified set of production rules (`new_P`) after eliminating unit productions.

```
def eliminate_unit_productions(self, new_P):

   prev_P = new_P.copy()

   copy_p = new_P.copy()

   has_unit_productions = True

   iteration = 0

   while has_unit_productions:

       iteration += 1

       print(f"Iteration {iteration}:")



       has_unit_productions = False

       nr_unit_production = 0



       for (LHS, RHS) in new_P.items():



           RHS_copy = RHS.copy()   # Create a copy of the RHS set

           for production in RHS_copy:

               if len(production) == 1 and production.isupper():

                   nr_unit_production += 1

                   print(f"Old Productions: {LHS} -> {RHS}")

                   print(f"Unit Production {nr_unit_production}: {LHS} ->
{production}")

                   new_derivations = copy_p[LHS].union(new_P[production])

                   new_derivations.remove(production)

                   print(f"New Productions: {LHS} -> {new_derivations}")
```

```
                    copy_p[LHS] = new_derivations

            new_P = copy_p

        for (LHS, RHS) in new_P.items():

            for production in RHS:

                if len(production) == 1 and production.isupper():

                    has_unit_productions = True
```

4. Eliminate Unproductive Symbols function:

The method `eliminate_unproductive_symbols` within the `Grammar` class is responsible for identifying and removing unproductive symbols from the grammar's productions. It initializes a set `productive_symbols_set` to track productive non-terminal symbols and a dictionary `productive_productions` to store their corresponding productions. The algorithm iteratively checks for productive symbols by inspecting each production rule in `new_P`. If a production consists of terminal symbols only, the left-hand side (LHS) symbol is deemed productive and added to `productive_symbols_set`. For non-terminal productions, the method verifies if all symbols are already productive; if so, the LHS symbol is marked as productive. After each iteration, the updated set of productive symbols and productions is displayed. The process terminates once no new productive productions are found. Finally, the method computes the difference between the original productions (`new_P`) and the productive productions, printing the results. If unproductive symbols were removed, it displays the modified set of productions; otherwise, it confirms that no changes were made. The method concludes by returning `productive_productions` and `productive_symbols_set`. This process effectively refines the grammar by eliminating unproductive symbols and their associated productions.

```
def eliminate_unproductive_symbols(self, new_P):

    productive_symbols_set = set()

    productive_productions = {}
```

```python
    iteration = 0
    has_unproductive_symbols = True

    print("Finding Productive Symbols:")
    while has_unproductive_symbols:
        iteration += 1
        print(f"Iteration {iteration}:")
        prev_productive_productions = productive_productions.copy()
        for (LHS, RHS) in new_P.items():
            for production in RHS:
                if len(production) == 1:
                    if production in self.V_t:
                        if LHS not in productive_symbols_set:
                            print(f"{LHS} -> {production}")
                        productive_symbols_set.add(LHS)
                        if LHS not in productive_productions:
                            productive_productions[LHS] = {production}
                        else:
                            productive_productions[LHS].add(production)
                else:
                    is_productive = True
                    for symbol in production:
                        if symbol.isupper():
                            # if is_productive:
                            if symbol not in productive_symbols_set:
                                is_productive = False
                    if is_productive:
                        print(f"{LHS} -> {production}")
                        productive_symbols_set.add(LHS)
                        if LHS not in productive_productions:
                            productive_productions[LHS] = {production}
                        else:
                            productive_productions[LHS].add(production)
```

5. Eliminate Inaccessible Symbols function:

The `eliminate_inaccessible_symbols` method serves to identify and remove inaccessible symbols from a given set of production rules (`new_P`) and a set of non-terminal symbols (`new_V_n`). Initially, the method creates a copy of `new_P` to preserve the original production rules. It then iterates over each production rule and checks which symbols are accessible starting from the start symbol (`self.S`). Symbols that are reachable from the start symbol are added to the `accessible_symbols_set`. The method then calculates `inaccessible_symbols_set` by taking the difference between `new_V_n` and `accessible_symbols_set`, representing symbols that are not reachable. Next, the method removes these inaccessible symbols from `copy_P` (the copy of `new_P`). It further examines the changes made to `new_P` by calculating the difference between `prev_P` (original `new_P`) and the modified `new_P`, storing removed symbols and associated productions in `removed_symbols_with_productions`. Finally, the method prints details about the changes made and returns the modified `new_P` and `new_V_n`. This method effectively ensures that only reachable symbols and associated productions are retained within the grammar, reflecting a practical step in grammar optimization and analysis.

```
def eliminate_inaccessible_symbols(self, new_P, new_V_n):
    prev_P = new_P.copy()
    copy_P = new_P.copy()
    accessible_symbols_set = set()
    for (LHS, RHS) in new_P.items():
```

```python
            for production in RHS:
                if LHS == self.S:
                    accessible_symbols_set.add(LHS)
                for symbol in production:
                    if symbol.isupper() and symbol != LHS:
                        accessible_symbols_set.add(symbol)

        inaccessible_symbols_set = new_V_n.difference(accessible_symbols_set)
        print("Set of Accessible Symbols =", accessible_symbols_set)
        print("Set of Inaccessible Symbols =", inaccessible_symbols_set)
        for inaccessible_symbol in inaccessible_symbols_set:
            try:
                copy_P.pop(inaccessible_symbol)
            except KeyError:
                continue

        new_V_n = accessible_symbols_set
        new_P = copy_P

        # Calculate the difference between new_P and productive_productions
        removed_symbols_with_productions = {}
        for symbol in prev_P:
            if symbol not in new_P:
                removed_symbols_with_productions[symbol] = prev_P[symbol]

        removed_symbols = set()
        for symbol in prev_P:
```

```
        if symbol not in new_P:

            removed_symbols.add(symbol)
```

Testing

Choosing to write tests for software components, like the grammar class, is crucial for ensuring the correctness, reliability, and robustness of the implemented functionality. Tests provide a systematic way to verify the behavior of methods and classes under various scenarios, helping to catch bugs and regressions early in the development cycle. By defining test cases that cover different aspects of the grammar manipulation operations—such as eliminating epsilon productions, inaccessible symbols, and unit productions, as well as converting to Chomsky Normal Form—the test suite validates that the grammar class functions as expected across diverse input configurations. This proactive approach to testing not only aids in identifying and rectifying potential issues but also enhances maintainability by serving as living documentation of the grammar class's intended behavior.

```
import unittest

from grammar import Grammar


class TestGrammarMethods(unittest.TestCase):


    def test_epsilon_elimination(self):

        # Test epsilon production elimination

        # Define grammar with epsilon production

        # Assert expected changes


    def test_inaccessible_elimination(self):

        # Test inaccessible symbol elimination
        # Define grammar with inaccessible symbols

        # Assert removal of inaccessible symbols


    def test_unit_production_elimination(self):

        # Test unit production elimination

        # Define grammar with unit productions

        # Assert removal of unit productions


    def test_chomsky_normal_form_conversion(self):

        # Test conversion to Chomsky Normal Form

        # Define grammar for conversion

        # Assert successful conversion


if __name__ == '__main__':

    unittest.main()
```

The provided TestGrammarMethods file encapsulates a suite of test cases designed to evaluate the behavior of methods within the Grammar class across various scenarios. The test class inherits from unittest.TestCase to utilize the testing framework provided by Python's unittest module. Each test method corresponds to a specific functionality of the Grammar class:

- *test_epsilon_elimination*: This test case evaluates the elimination of epsilon productions from the grammar. It defines two scenarios—a grammar with epsilon productions and a grammar without epsilon productions—and verifies that after applying the elimination, epsilon productions are successfully removed and the structure of the grammar is appropriately modified.

- *test_inaccessible_elimination*: This test focuses on eliminating inaccessible symbols from the grammar. It tests two scenarios—one with inaccessible symbols and another without—and confirms that inaccessible symbols are correctly identified and removed.
- *test_unit_production_elimination*: This test case checks the elimination of unit productions. It examines a grammar with and without unit productions and validates that unit productions are eliminated as expected.
- *test_chomsky_normal_form_conversion*: This test suite assesses the conversion of a grammar into Chomsky Normal Form (CNF). It includes a grammar suitable for conversion and one not suitable for conversion. The tests confirm that the conversion method appropriately transforms the grammar into CNF and handles cases where conversion is not feasible by returning None.

Each test case uses assertions (self.assertNotIn, self.assertEqual, self.assertIsNone) to validate specific conditions and expected outcomes, ensuring that the Grammar methods perform correctly under different scenarios. The setUp and tearDown methods are used to print separator lines before and after each test case for better readability of the test output. By covering multiple cases and potential edge scenarios, this test suite enhances the reliability and correctness of the Grammar class implementation. Running unittest.main() executes the test suite, providing feedback on the success or failure of each test case based on defined assertions.

Conclusions:

In this laboratory work, we delved into the concept of Chomsky Normal Form (CNF) within the realm of Formal Languages & Finite Automata. Our objectives were comprehensive, aiming to not only understand CNF but also implement and test a method for normalizing grammars to this form.

Through this project, we achieved several key milestones:

- Understanding CNF: We gained a solid understanding of what constitutes Chomsky Normal Form, its significance in formal language theory, and its role in simplifying grammars.
- Normalization Approaches: We explored various approaches and algorithms for converting grammars to CNF, encompassing the elimination of epsilon productions, unit productions, and other transformations.
- Implementation: The core of our work involved implementing a method capable of normalizing an input grammar to CNF. This was encapsulated within a suitable class structure, aligning with good programming practices.
- Testing: Acknowledging the importance of reliability and correctness, we incorporated unit tests to validate our CNF normalization method. This not only ensured functionality but also demonstrated proficiency in software testing.
- Repository and Documentation: This laboratory work was organized in a public repository on GitHub, complete with a detailed Markdown report outlining the objectives, implementation details, and evaluation criteria. This allowed for transparent evaluation and easy access to our work.

In conclusion, this laboratory work provided a practical and insightful journey into the world of Chomsky Normal Form, reinforcing fundamental concepts of formal language theory while honing our programming and testing skills. The structured approach to implementation and documentation reflects our commitment to excellence in academic endeavors and software engineering practices.