MINISTERUL EDUCAŢIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA
UNIVERSITATEA TEHNICĂ A MOLDOVEI
FACULTATEA CALCULATOARE, INFORMATICĂ ȘI MICROELECTRONICĂ

Laboratory work 4:

Chomsky Normal Form.

Course: Formal Languages and Finite Automata

Author: Cucoș Maria

Chișinău, 2024

TABLE OF CONTENTS

Theory 3
Conversion 3
Objectives 3
Implementation Description 4
Testing 11
Conclusion 12

Theory

Chomsky Normal Form represents a grammar where every production is either of the form `A → BC` or `A → c` (where `A, B, C` are variables and `c` a terminal 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.

Conversion

The conversion to Chomsky Normal Form has four main steps:

1. Eliminate ε Productions. Go through all productions, and for each, omit every possible subset of nullable variables. For example, if `P → AxB` with both `A` and `B` nullable, add productions `P → xB | Ax | x`. After this, delete all productions with empty RHS (right hand side).
2. Eliminate Renaming (Unit Productions). A unit production is where RHS has only one symbol. Consider production `A → B`. Then for every production `B →` α, add the production `A →` α. Repeat until done (but don't re-create a unit production already deleted).
3. Eliminate Inaccessible Symbols. Inaccessible symbols cannot be reached from the start symbol. These symbols do not contribute to generating strings in the language described by the grammar and can be safely removed. For example, the symbol A from `A → bC` is inaccessible: `S → B`, `B → aC` and `C → b`. So eliminate the production `A → bC`.
4. Eliminate Non-Productive Symbols. Non-productive symbols are symbols that cannot derive any terminal string. These symbols do not contribute to generating strings in the language described by the grammar and can be safely removed. For example, `A → bA` is non-productive as it goes in a loop so eliminate it.
5. Obtain the Chomsky Normal Form (CNF). The resulting grammar is easier to analyze and has desirable properties for certain algorithms and analyses. Introduce new non-terminal symbols to replace terminals and combinations of terminals and non-terminals in the grammar's production rules, ensuring that all rules adhere to the CNF requirements. Any production rule with a terminal symbol on the RHS needs to be replaced with a non-terminal symbol. For example, if there's a production like `A → a`, where `a` is a terminal symbol, introduce a new non-terminal symbol (let's call it `X`) and replace the production with `A → X`. Then, we add a new production `X → a`. Any production rule with more than two symbols on the RHS or with a combination of terminals and non-terminals needs to be split into multiple rules, each with exactly two symbols on the RHS. For example, if there's a production like `A → BCD`, replace it with `A → BE` and `E → CD`.

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.
   (a) The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).
   (b) The implemented functionality needs executed and tested.
   (c) A BONUS point will be given for unit tests that validate the functionality of the project.
   (d) A BONUS point will be given for the making of the aforementioned function to accept any grammar, not only the one from the self's variant.

Implementation Description

I created a `Grammar` class where multiple methods that represent the conversion to CNF are displayed. Firstly, `__init__(self, non_terminals, terminals, rules, start='S')` method serves as the constructor for the `Grammar` class. It initializes the object with the provided non-terminals, terminals, production rules, and starting symbol.

The `eliminate_epsilon_productions(self)` method eliminates ε productions from the grammar. It iterates over each non-terminal symbol in the grammar and examines its production rules. If a production rule consists of only ε, it marks the non-terminal symbol as nullable.

```
for non_terminal in self.non_terminals:
    for production in self.rules[non_terminal]:
        if production == '':
            nullable.add(non_terminal)
```

Next, the method iterates over the non-terminals again to identify indirect nullable non-terminals - those non-terminals that can derive ε via a sequence of productions involving other nullable non-terminals. It sets up a loop that continues until no more changes occur. Within each iteration of the loop, for each non-terminal that is not already marked as nullable, it checks whether all symbols in its production rules are nullable. If so, it marks the non-terminal as nullable and sets a flag (`changes`) to True to indicate that changes were made.

```
changes = True
while changes:
    changes = False
    for non_terminal in self.non_terminals:
        if non_terminal not in nullable:
```

```
                for production in self.rules[non_terminal]:
                    if all(symbol in nullable for symbol in production):
                        nullable.add(non_terminal)
                        changes = True
                        break
```

After identifying all nullable non-terminals, the method eliminates ε productions from the grammar. It creates a new dictionary called `new_rules` to store the updated production rules without ε productions. For each non-terminal in the original grammar, it iterates over its production rules. If a production rule is not an ε production, it adds it to the list of new productions. If it is an ε production, it expands it into all possible combinations of symbols, taking into account the nullable non-terminals. Finally, it removes any duplicate productions and updates the grammar's rules with the new set of productions.

```
    new_rules = {}
    for non_terminal in self.rules:
        new_prods = []
        for production in self.rules[non_terminal]:
            if production != '':
                new_prods.extend(
                    self._expand_nullable_prod(production, nullable))
        new_rules[non_terminal] = list(set(new_prods))
    self.rules = new_rules
```

`_expand_nullable_prod` is a helper method of the `eliminate_epsilon_productions` method to expand a production by replacing nullable non-terminals with ε. The method begins by initializing a list called `expansions` with a single empty string `''`. This list will store all possible expansions of the given production. For each symbol, it creates a new list called `new_expansions` to store the expanded productions resulting from the current symbol. If the symbol is nullable, it iterates over each expansion in the `expansions` list and creates two new expansions: one with the nullable symbol appended and one without it. This accounts for the possibility of either including or excluding the nullable symbol in the expansion. If the symbol is not nullable, it simply appends the symbol to each expansion in the `expansions` list. After processing all symbols in the production, the expansions list is updated to contain the new expansions. Finally, the method returns a list containing only the non-empty expansions from the expansions list.

```
    expansions = ['']
    for symbol in production:
        new_expansions = []
```

```
            if symbol in nullable:
                for expansion in expansions:
                    new_expansions.append(expansion + symbol)
                    new_expansions.append(expansion)
            else:
                for expansion in expansions:
                    new_expansions.append(expansion + symbol)
            expansions = new_expansions
        return [expansion for expansion in expansions if expansion]
```

The `eliminate_renaming` method is responsible for eliminating renaming productions (unit productions) from the grammar. The method initializes a boolean variable `changes` that tracks whether any changes have been made to the grammar during the current iteration of the loop. The method enters a `while` loop that continues as long as changes are being made to the grammar. The method iterates over each non-terminal symbol in the grammar. For each non-terminal, the method identifies unit productions, it filters them out from the list of productions associated with the current non-terminal. For each identified unit production, the method retrieves the productions of the corresponding non-terminal symbol, it adds them to the list of productions for the current non-terminal, effectively replacing the unit production. After adding the new productions, the method removes any duplicates from the list of productions for the current non-terminal. Once all unit productions for a non-terminal have been processed, the method filters out these unit productions from the list of productions.

```
        changes = True
        while changes:
            changes = False
            for non_terminal in self.non_terminals:
                unit_productions = [
                    p for p in self.rules[non_terminal] if p in self.non_terminals]
                for unit in unit_productions:
                    new_productions = self.rules[unit]
                    if new_productions:
                        self.rules[non_terminal].extend(new_productions)
                        self.rules[non_terminal].remove(unit)
                        self.rules[non_terminal] = list(
                            set(self.rules[non_terminal]))
```

```
                changes = True

        self.rules[non_terminal] = [
            p for p in self.rules[non_terminal] if p not in self.non_terminals]
```

The `eliminate_inaccessible_symbols` method starts by initializing a set called `accessible` with the starting symbol of the grammar `self.start`. This set stores symbols that are reachable from the starting symbol. The boolean variable `changes` tracks whether any changes have been made to the grammar during the current iteration of the loop. The method creates a copy of the original grammar rules `self.rules` and stores them in the `old_rules` dictionary. This copy will be used to update the grammar after removing inaccessible symbols.

```
    accessible = {self.start}
    changes = True
    old_rules = self.rules.copy()
```

The method enters a `while` loop that continues until no changes are made to the accessible set during an iteration. For each symbol in the `accessible` set, the method examines the productions associated with that symbol in the grammar rules. It iterates over each symbol in the productions and checks if the symbol is a non-terminal and not already in the accessible set. If such a symbol is found, it is added to the `accessible` set, and the `changes` flag is set to True to indicate that changes have occurred. Once all accessible symbols have been identified, the method updates the non-terminals of the grammar to include only the accessible symbols. It also updates the grammar rules `self.rules` to include only the rules associated with the accessible symbols.

```
    while changes:
        changes = False
        for non_terminal in accessible.copy():
            for production in self.rules[non_terminal]:
                for symbol in production:
                    if symbol in self.non_terminals and symbol not in accessible:
                        accessible.add(symbol)
                        changes = True

    self.non_terminals = list(accessible)
    self.rules = {nt: old_rules[nt] for nt in accessible}
```

The `eliminate_non_productive_symbols` method starts by initializing a set called `productive` with the starting symbol of the grammar `self.start`. This set stores symbols that lead to terminal strings. Inside the `while` loop, for each symbol in the grammar's non-terminals, the method checks if it is not already in the productive set. If the symbol is not productive, it checks each production associated with that symbol. It verifies whether every symbol in the production is either a terminal or a productive non-terminal. If all symbols in a production are productive, the symbol is added to the `productive` set.

```
    productive = {self.start}
    changes = True
    while changes:
        changes = False
        for non_terminal in self.non_terminals:
            if non_terminal not in productive:
                for production in self.rules[non_terminal]:
                    if all(symbol in self.terminals or symbol in productive
                        for symbol in production):
                        productive.add(non_terminal)
                        changes = True
                        break

    self.non_terminals = list(productive)
```

Once all productive symbols have been identified, the method updates the non-terminals of the grammar to include only the productive symbols. It also updates the grammar rules `self.rules` to include only the rules associated with the productive symbols.

```
    updated_rules = {}
    for nt in productive:
        productive_rules = []
        for production in self.rules[nt]:
            if all(symbol in self.terminals or symbol
                in productive for symbol in production):
                productive_rules.append(production)

        updated_rules[nt] = productive_rules
    self.rules = updated_rules
```

The method `_create_new_non_terminal` generates a new non-terminal symbol that is not already present in the grammar's set of non-terminals `self.non_terminals`. A string containing a set of characters is used as non-terminal symbols. It includes digits and uppercase letters from the Latin alphabet. The method first attempts to find a single unused letter from the alphabet. It iterates over each letter in the `alphabet` string. If a letter is not already in `self.non_terminals`, it means it's available for use. In this case, the method appends the letter to `self.non_terminals` and returns it.

```
alphabet = '0123456789EFGHIJKLMNOPQRTUVWXYZ'
for letter in alphabet:
    if letter not in self.non_terminals:
        self.non_terminals.append(letter)
        return letter
```

If all single letters are already used, the method combines letters with numbers. It iterates over each letter in the alphabet string and combines it with numbers ranging from 0 to 99. For each combination, if the resulting symbol is not already in `self.non_terminals`, it appends the symbol to `self.non_terminals` and returns it. If all possible combinations of letters and numbers are exhausted without finding an unused symbol, the method raises a `ValueError` indicating that all possible non-terminal symbols have been exhausted.

```
for letter in alphabet:
    for num in range(100):
        new_symbol = f'{letter}{num}'
        if new_symbol not in self.non_terminals:
            self.non_terminals.append(new_symbol)
            return new_symbol
raise ValueError("Exhausted all possible non-terminal symbols.")
```

The `to_cnf` method is responsible for converting the grammar to Chomsky Normal Form (CNF). It calls the methods `eliminate_epsilon_productions`, `eliminate_renaming` etc to perform these transformations. These steps ensure that the grammar is prepared for conversion to CNF.

```
self.eliminate_epsilon_productions()
self.eliminate_renaming()
self.eliminate_inaccessible_symbols()
self.eliminate_non_productive_symbols()
```

After that, it initializes a dictionary `rhs_to_non_terminal` to keep track of replacements of RHS symbols with new non-terminal symbols. The list of old non-terminals is copied from the grammar rules and

an empty dictionary `new_rules` is initialized to store the updated rules after the conversion. It then iterates through each non-terminal in the grammar rules. For each production of a non-terminal, if the production has more than 2 symbols, it extracts the first two symbols `first_two_symbols`. If the `first_two_symbols` combination is not already present in `rhs_to_non_terminal`, it creates a new non-terminal symbol using `_create_new_non_terminal` method and adds it to `rhs_to_non_terminal`. The program replaces the first two symbols with the new non-terminal symbol in the production, adds the modified production to the `new_rules` dictionary under the corresponding non-terminal.

```
    for non_terminal in list(self.rules):
        new_rules[non_terminal] = set()
        for production in self.rules[non_terminal]:
            while len(production) > 2:
                first_two_symbols = production[:2]
                if first_two_symbols in rhs_to_non_terminal:
                    new_non_terminal = rhs_to_non_terminal[first_two_symbols]
                else:
                    new_non_terminal = self._create_new_non_terminal()
                    new_rules[new_non_terminal] = {first_two_symbols}
                    rhs_to_non_terminal[first_two_symbols] = new_non_terminal

                production = new_non_terminal + production[2:]
            new_rules[non_terminal].add(production)
```

The mixed productions where one symbol is a terminal and the other is a non-terminal are handled by replacing the terminal with a new non-terminal symbol. It then reorders the rules to match the original order of non-terminals plus any new non-terminals created during the conversion. The CNF rules stored in `self.rules` are returned.

```
    for non_terminal, productions in list(new_rules.items()):
        temp_productions = productions.copy()
        for production in temp_productions:
            if len(production) == 2 and any(symbol in self.terminals for symbol
                in production):
                new_production = []
                for symbol in production:
                    if symbol in self.terminals:
                        if symbol in rhs_to_non_terminal:
```

```
                        new_non_terminal = rhs_to_non_terminal[symbol]
                    else:
                        new_non_terminal = self._create_new_non_terminal()
                        new_rules[new_non_terminal] = {symbol}
                        rhs_to_non_terminal[symbol] = new_non_terminal
                    new_production.append(new_non_terminal)
                else:
                    new_production.append(symbol)
            productions.remove(production)
            productions.add(''.join(new_production))

    self.rules = {nt: new_rules[nt] for nt in old_non_terminals +
                  list(set(new_rules) - set(old_non_terminals))}
    return self.rules
```

The following piece of code is an example of how to use the Grammar class to convert a given context-free grammar to Chomsky Normal Form. The example is the 8 variant given by the teacher.

It is created an instance of the `Grammar` class with the defined non-terminals, terminals, production rules, and start symbol. It is called the `to_cnf` method of the `Grammar` instance to convert the grammar to Chomsky Normal Form.

```
VN = {'S', 'A', 'B', 'C'}
VT = {'a', 'd'}
P = {'S': ['dB', 'A'], 'A': ['d', 'dS', 'aAdAB'], 'B': ['a', 'aS', 'A', ''],
    'C': ['Aa']}
S = 'S'
grammar = Grammar(VN, VT, P, S)
cnf_rules = grammar.to_cnf(print_steps=False)
print("Chomsky Normal Form (CNF):")
print(cnf_rules)
```

Testing

By executing the `Grammar` class, we get the following output. `0, 1, 2, 3, 4` are introduced non-terminal symbols during the conversion process. They are used to represent combinations of terminal symbols.

```
Chomsky Normal Form (CNF):
{'S': {'d', '3B', '3S', '2B', '1A'}, 'A': {'d', '3S', '2B', '1A'},
'B': {'d', 'a', '3S', '2B', '1A', '4S'}, '3': {'d'}, '0': {'4A'},
'2': {'1A'}, '4': {'a'}, '1': {'03'}}
```

Conclusion

In this report, I got to achieve several objectives related to understanding Chomsky Normal Form, implementing a method for normalizing a grammar according to CNF rules, and ensuring the functionality works correctly. This project was an opportunity to learn about CNF, a specific form of context-free grammars that simplifies parsing and analysis in various computational linguistics and natural language processing tasks.

Through the project, I became familiar with different approaches to normalizing a grammar, including eliminating epsilon productions, removing renaming productions, handling inaccessible symbols, and eliminating non-productive symbols. I implemented a method to normalize an input grammar according to the rules of CNF. To ensure the correctness of the implementation, I had to test if the CNF conversion method correctly transformed grammars into CNF, maintaining the language's equivalence before and after normalization. Additionally, I extended the normalization method to accept any grammar, not just a specific variant. This makes the implementation more generalizable and adaptable to different grammar structures and input formats.