Chomsky Normal Form

Course: Formal Languages & Finite Automata

Author: Dimbitchi Sergiu

Theory

Chomsky Normal Form (CNF) is a specific way to simplify the structure of context-free grammars, making them easier to analyze and manipulate, particularly in algorithms related to parsing and language recognition. Developed by Noam Chomsky, this form is essential for several computational processes in the field of computer science, especially in the theory of formal languages. In CNF, each rule of the grammar adheres to a strict pattern, which helps in the standardization of parsing techniques such as the CYK algorithm.

A context-free grammar is in Chomsky Normal Form if all its production rules satisfy one of two conditions: either the rule produces a single terminal symbol, or it produces two non-terminal symbols. This restriction greatly simplifies the process of generating parse trees since it limits the complexity of the rules that can be applied at each step. Specifically, no production rule can directly produce an empty string, except for the rule that allows the start symbol to produce an empty string if necessary.

This characteristic is particularly valuable in parsing algorithms, which rely on predictable patterns of grammar production. By reducing each rule to one of these two forms, algorithms can operate more efficiently and with greater predictability. For example, the CYK algorithm, which determines whether a given string can be generated by a certain grammar, benefits from the grammar being in CNF because it can systematically apply these simple rules to analyze the string.

To convert any context-free grammar into Chomsky Normal Form, several steps are typically followed. Initially, useless symbols, which do not appear in any derivation of a terminal string, are removed. Subsequently, unit productions, where a non-terminal directly leads to another non-terminal, are eliminated. Finally, any rules producing more than two non-terminals or a single non-terminal alongside terminals are rearranged into binary productions, maintaining the two forms permissible under CNF.

Despite its strengths in simplifying grammar for computational uses, Chomsky Normal Form is not always practical for human understanding or for representing natural language grammar in a way that reflects human language use. Its application is mainly theoretical and computational, focused on efficiency and systematic processing in computer algorithms, rather than linguistic nuance. Nevertheless, the form is a cornerstone in the theory of formal languages, demonstrating the deep interconnections between linguistics and computer science.

Objectives:

- Learn about Chomsky Normal Form (CNF) [1].
- Get familiar with the approaches of normalizing a grammar.
- Implement a method for normalizing an input grammar by the rules of CNF.

1. The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).
2. The implemented functionality needs executed and tested.
3. A BONUS point will be given for the student who will have unit tests that validate the functionality of the project.
4. 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:

```
def remove_null_productions(self):
    null_prods = {key for key, values in self.P.items() if "ε" in values}
    if null_prods:
        # Remove ε productions
        for key in null_prods:
            self.P[key].remove("ε")
        # Add new productions by removing nullable symbols
        for key, values in list(self.P.items()):
            new_values = []
            for production in values:
                for null_prod in null_prods:
                    if null_prod in production:
                        new_values.extend(
                            self.create_new_productions(production, null_prod)
                        )
            self.P[key].extend(
                prod for prod in new_values if prod not in self.P[key]
            )
```

remove_null_productions(self) : This function is tasked with removing null (epsilon) productions from the grammar. It first identifies any productions that can produce an epsilon and then iteratively removes these productions and updates other productions that might be affected by the removal.

```python
class Grammar(BaseGrammar):
    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()

    def start_symbol_rhs_removal(self):
        if any(
            self.S in production for value in self.P.values() for production in value
        ):
            self.P = {"X": [self.S]} | self.P
            self.S = "X"
            self.Vn.append(self.S)

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

cfg_to_cnf(self): This function is the primary method for converting a context-free grammar to Chomsky Normal Form. It calls a series of helper functions in a specific sequence to transform the grammar, addressing the start symbol on the right-hand side, null productions, unit productions, inaccessible symbols, and ensuring that terminal and nonterminal symbols are properly handled.

start_symbol_rhs_removal(self): This function addresses the issue of the start symbol appearing on the right-hand side (RHS) of any production. If the start symbol is found on the RHS, it introduces a new start symbol and updates the production rules and nonterminal symbols accordingly.

create_new_productions(self, production, character): It generates new productions by breaking down a production rule whenever a specific character (typically representing a null production or a unit production) is encountered. This helps in eliminating these specific types of productions from the grammar.

```
def remove_unit_productions(self):
    for key, value in self.P.items():
        for production in value:
            if key == production:
                self.P[key].remove(production)
    changes = True
    while changes:
        changes = False
        for key, value in self.P.items():
            for production in value:
                if production in self.Vn:
                    changes = True
                    self.P[key].remove(production)
                    for prod in self.P[production]:
                        if prod not in self.P[key]:
                            self.P[key].append(prod)
```

remove_unit_productions(self): This function eliminates unit productions (productions where a nonterminal leads directly to another nonterminal) from the grammar. It iterates through the productions, identifies unit productions, and replaces them with non-unit equivalents until no unit productions remain.

```
def remove_inaccessible_symbols(self):
    accessible = set([self.S])
    queue = [self.S]

    while queue:
        current = queue.pop(0)
        for production in self.P.get(current, []):
            for symbol in production:
                if symbol in self.Vn and symbol not in accessible:
                    accessible.add(symbol)
                    queue.append(symbol)

    self.Vn = [nt for nt in self.Vn if nt in accessible]
    for nt in list(self.P.keys()):
        if nt not in accessible:
            del self.P[nt]
```

remove_inaccessible_symbols(self): This function removes symbols that are not accessible from the start symbol. It uses a breadth-first search approach to identify all symbols that are reachable from the start symbol and then removes any nonterminals that are not in this set from the grammar.

```
def replace_terminals_with_nonterminals(self):
    def new_nonterminal():
        available = set(chr(i) for i in range(65, 91)) - set(self.Vn)
        if not available:
            raise ValueError("Ran out of single-letter nonterminal symbols!")
        return min(available)

    terminal_to_nonterminal = {}
    new_P = {key: [] for key in self.P}

    for key, productions in self.P.items():
        for prod in productions:
            new_prod = ""
            for char in prod:
                if char in self.Vt and len(prod) > 1:
                    if char not in terminal_to_nonterminal:
                        new_nt = new_nonterminal()
                        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_P[key].append(new_prod)

    self.P = new_P
```

replace_terminals_with_nonterminals(self): This function is designed to replace terminals in the productions with nonterminals when the production contains more than one symbol. It ensures that each terminal is only found in productions where it is the only symbol on the right-hand side, a requirement for CNF.

```
def reduce_production_length(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!")

    existing_binaries = {}
    new_productions_dict = {}
    for key, productions in list(self.P.items()):
        new_productions = []
        for production in productions:
            if len(production) > 2:
                while len(production) > 2:
                    last_two = production[-2:]
                    if last_two not in existing_binaries:
                        new_nt = new_nonterminal(
                            set(self.Vn) | set(new_productions_dict.keys())
                        )
                        self.Vn.append(new_nt)
                        new_productions_dict[new_nt] = [last_two]
                        existing_binaries[last_two] = new_nt
                    production = production[:-2] + existing_binaries[last_two]
                new_productions.append(production)
            else:
                new_productions.append(production)
        self.P[key] = new_productions
    self.P.update(new_productions_dict)
```

reduce_production_length(self): The purpose of this function is to ensure that each production rule has a right-hand side consisting of exactly two nonterminals or a single terminal, as required by CNF. It systematically breaks down longer productions into binary productions to meet this criterion.

The output :

```
Current Grammar:
Nonterminals (Vn): ['S', 'A', 'B', 'C', 'E']
Terminals (Vt): ['a', 'b', 'c']
Productions (P):
S -> ['aB', 'AC']
A -> ['a', 'ASC', 'BC']
B -> ['b', 'bS']
C -> ['ε', 'BA']
E -> ['bB']
Start symbol (S): S


Current Grammar:
Nonterminals (Vn): ['S', 'A', 'B', 'C', 'X', 'D', 'E', 'F']
Terminals (Vt): ['a', 'b', 'c']
Productions (P):
X -> ['DB', 'AC', 'a', 'AF', 'BC', 'AS', 'b', 'ES']
S -> ['DB', 'AC', 'a', 'AF', 'BC', 'AS', 'b', 'ES']
A -> ['a', 'AF', 'BC', 'AS', 'b', 'ES']
B -> ['b', 'ES']
C -> ['BA']
D -> ['a']
E -> ['b']
F -> ['SC']
Start symbol (S): X
```

Conclucions :

The process of transforming a context-free grammar into Chomsky Normal Form is an intricate task that involves a series of detailed steps, each tailored to streamline and standardize the grammar for computational analysis. The methodology laid out in the implementation description effectively tackles the core aspects of grammar normalization by focusing on the removal of epsilon and unit productions, the elimination of inaccessible symbols, and the proper restructuring of production rules to meet the stringent requirements of CNF. This conversion is crucial for enabling efficient parsing algorithms, such as the CYK algorithm, which depend on the predictable structure that CNF provides.

The comprehensive approach to addressing the start symbol appearing on the right-hand side ensures that the grammar remains consistent and avoids recursive ambiguities. By introducing a new start symbol and updating production rules accordingly, the grammar maintains its integrity while adapting to the CNF requirements. The steps for removing null productions and unit productions are particularly vital, as they eliminate redundancies and simplify the grammar, making it easier to manage and more practical for algorithmic applications.

Moreover, the methodology for replacing terminals with nonterminals in complex productions and reducing production lengths to adhere to the two nonterminals or a single terminal format is a testament to the meticulous attention to detail required in such transformations. These steps not only help in achieving the formal structure demanded by CNF but also enhance the grammar's usability in computational contexts.

The encapsulation of these functions within a class structure and the sequential execution of these methods demonstrate a robust approach to grammar normalization. The possibility of extending this functionality to accept any grammar highlights the flexibility and adaptability of the implementation. Additionally, the emphasis on unit testing underscores the importance of reliability and correctness in software development, particularly when dealing with the foundational elements of language processing technologies.

Overall, this work not only showcases the technical proficiency required to manipulate and convert grammatical structures but also highlights the intersection of theoretical computer science with practical application, providing a valuable tool for researchers and developers in the field of computational linguistics.

References

1. https://www.geeksforgeeks.org/introduction-of-lexical-analysis/

2. https://medium.com/@pythonmembers.club/building-a-lexer-in-python-a-tutorial-3b6de161fe84

3. https://www.youtube.com/watch?v=BI3K-ME3L74

4. Theory lessons from else