Ministerul Educaţiei, Culturii şi Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Departamentul Ingineria Software şi Automatică

REPORT

Laboratory work 5
Course: Formal Languages & Finite Automata
Theme: Chomsky Normal Form

Elaborated: Latcovschi Cătălin,
FAF-221

Verified: asist.univ.
Dumitru Crețu

Chișinău 2024

Overview

A CFG is in Chomsky normal form when every rule is of the form A → BC and A → a, where a is a terminal, and A, B, and C are variables. Further B and C are not the start variable. Additionally we permit the rule S → ε where S is the start variable, for technical reasons.

Objectives

1. Learn about Chomsky Normal Form (CNF) [1].
2. Get familiar with the approaches of normalizing grammar.
3. Implement a method for normalizing an input grammar by the rules of CNF.
4. The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).
5. The implemented functionality needs to be executed and tested.
6. A BONUS point will be given for the student who will have unit tests that validate the functionality of the project.
7. 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

1. **remove_duplicates method:**

```
    def remove_duplicates(self, array):
        unique_strings = set()
        result = []

        for string in array:
            if string not in unique_strings:
                result.append(string)
                unique_strings.add(string)

        return result
```

This function, remove_duplicates, eliminates duplicate strings from an input array. It initializes an empty set to store unique strings and an empty list for the final result. By iterating through the array, it checks if each string is already in the set. If not, it appends it to the result list and adds it to the set. Finally, it returns the list containing only unique strings. This approach efficiently identifies and

removes duplicates using a set, ensuring that the resulting list contains distinct elements from the original array.

2. **generate_strings_with_letter_removed method:**

```
def generate_strings_with_letter_removed(self, s, letter):
    strings = []
    count_b = s.count('B')
    for i in range(0, len(s)):
        if s[i] == letter:
            string = s[0:i] + s[i + 1:]
            strings.append(string)
            strings += self.generate_strings_with_letter_removed(string, letter)
    return strings
```

This function, `generate_strings_with_letter_removed`, recursively generates strings by removing a specified letter from the input string `s`. It iterates through each character in `s`, removing instances of the specified letter and appending the resulting strings to a list. Recursively, it continues this process with the updated strings. Finally, it returns the list of generated strings.

3. **remove_empty_string method:**

```
def remove_empty_string(self):
    empty_symbols = []
    for symbol in self.productions:
        for transition in self.productions[symbol]:
            if transition == "":
                if symbol not in empty_symbols:
                    empty_symbols.append(symbol)
        self.productions[symbol] = list(filter(lambda x: len(x) > 0, self.productions[symbol]))

    for empty_symbol in empty_symbols:
        for symbol in self.productions:
            transitions_to_change = []
            for transition in self.productions[symbol]:
                if empty_symbol in transition:
                    transitions_to_change.append(transition)

            append_to_productions = []
            for trns in transitions_to_change:
                append_to_productions += self.generate_strings_with_letter_removed(trns, empty_symbol)

            self.productions[symbol] += append_to_productions
            self.productions[symbol] = self.remove_duplicates(self.productions[symbol])
```

This method, `remove_empty_string`, removes empty strings from the productions dictionary. It first identifies symbols with empty transitions and removes them. Then, it iterates through symbols containing transitions involving those empty symbols, generating strings by removing them. These generated strings are appended to the respective symbol's transitions, ensuring no empty strings

remain. Lastly, it removes any duplicates to maintain unique transitions. This process efficiently cleanses the productions of empty strings, preserving the integrity of the grammar.

4. **remove_inaccessible method:**

```
def remove_inaccessible(self):
    reachable_symbols = [self.start_symbol]
    while True:
        symbol = reachable_symbols[len(reachable_symbols) - 1]
        found_something = False
        for trns in self.productions[symbol]:
            for i in trns:
                if i in self.variables:
                    if i not in reachable_symbols:
                        found_something = True
                        reachable_symbols.append(i)
        if not found_something:
            break
    unreachable_symbols = []
    for i in self.variables:
        if i not in reachable_symbols:
            unreachable_symbols.append(i)
    for i in self.variables:
        if len(self.productions[i]) == 0 and i not in unreachable_symbols:
            unreachable_symbols.append(i)
    for symbol in self.productions:
        for idx in range(0, len(self.productions[symbol])):
            for i in unreachable_symbols:
                self.productions[symbol][idx] = self.productions[symbol][idx].replace(i, "")
    for i in unreachable_symbols:
        del self.productions[i]
        self.variables.remove(i)
```

This method, `remove_inaccessible`, eliminates unreachable symbols and productions from a context-free grammar. Initially, it identifies reachable symbols from the start symbol by traversing the productions. Then, it identifies unreachable symbols by comparing all symbols against the reachable ones. Afterwards, it removes unreachable symbols from the productions, replacing any occurrences in existing productions. Finally, it deletes unreachable symbols from the productions dictionary and updates the list of variables accordingly, ensuring the grammar remains valid. This process effectively prunes the grammar, removing inaccessible components.

5. **eliminate_unit_productions method:**

```python
def eliminate_unit_productions(self):
    while True:
        found_something = False
        for symbol in self.productions:
            for idx in range(0, len(self.productions[symbol])):
                trns = self.productions[symbol][idx]
                if len(trns) == 1 and trns in self.variables:
                    found_something = True
                    self.productions[symbol].pop(idx)
                    self.productions[symbol] += self.productions[trns]
        if not found_something:
            break
```

This method, `eliminate_unit_productions`, removes unit productions from a context-free grammar. It iterates through each symbol in the productions dictionary, checking for unit productions where a single non-terminal symbol generates another non-terminal symbol. If found, it replaces the unit production with the productions of the generated non-terminal symbol. This process continues until no more unit productions can be found. By eliminating unit productions, the method simplifies the grammar, making it easier to work with and analyze.

6. convert_to_chomsky_normal_form method:

```python
def convert_to_chomsky_normal_form(self):
    lambda_symbol = "ε"
    new_nonterminals = {}
    k = 1
    for symbol in self.productions:
        for idx in range(0, len(self.productions[symbol])):
            trns = self.productions[symbol][idx]
            if len(trns) >= 2:
                if len(trns) == 2:
                    cnt_nonterm = 0
                    for i in trns:
                        if i in self.variables:
                            cnt_nonterm += 1
                    if cnt_nonterm == 2:
                        continue

                rest = self.productions[symbol][idx][1:]
                nonterm_found = ""
                for i in new_nonterminals:
                    if rest == new_nonterminals[i]:
                        nonterm_found = i
                        break
                if nonterm_found == "":
                    new_nonterminals[lambda_symbol + str(k)] = rest
                    nonterm_found = lambda_symbol + str(k)
                    k += 1
                self.productions[symbol][idx] = self.productions[symbol][idx].replace(rest, nonterm_found)
```

```
                if self.productions[symbol][idx][0] in self.terminals:
                    term = self.productions[symbol][idx][0]
                    nonterm_found = ""
                    for i in new_nonterminals:
                        if term == new_nonterminals[i]:
                            nonterm_found = i
                            break
                    if nonterm_found == "":
                        new_nonterminals[lambda_symbol + str(k)] = term
                        nonterm_found = lambda_symbol + str(k)
                        k += 1
                    self.productions[symbol][idx] = self.productions[symbol][idx].replace(term, nonterm_found)

    self.productions.update(new_nonterminals)
```

This method, `convert_to_chomsky_normal_form`, transforms a context-free grammar into Chomsky Normal Form (CNF). It introduces new non-terminal symbols to replace productions with more than two symbols.

First, it initializes a lambda symbol and a dictionary for new non-terminals. It then iterates through each production, identifying productions with more than two symbols. For such productions, it introduces new non-terminals to replace subsequences of symbols.

It ensures that each production contains either two non-terminals or a non-terminal followed by a terminal. It updates the productions accordingly, replacing symbols with newly introduced non-terminals as needed. Finally, it updates the productions dictionary with the new non-terminals.

This method effectively converts the grammar into CNF, facilitating further analysis and manipulation according to Chomsky Normal Form standards.

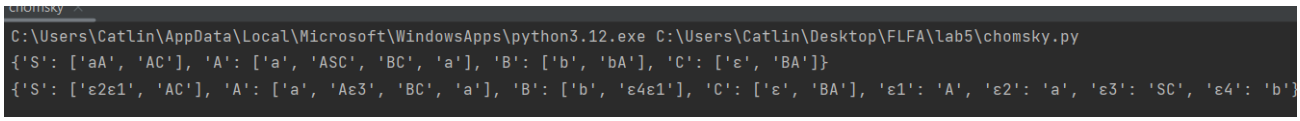
Results

```
variables = {"S", "A", "B", "C", "D", "E"}
terminals = {"a", "b"}
start_symbol = "S"

productions = {
    "S": ["aA", "AC"],
    "A": ["a", "ASC", "BC", "aD"],
    "B": ["b", "bA"],
    "C": ["ε", "BA"],
    "D": ["abC"],
    "E": ["aB"]
}

grammar = Grammar(variables, terminals, productions, start_symbol)

grammar.transform_to_chomsky_normal_form()
```



Conclusions

In conclusion, the laboratory work on converting context-free grammars into Chomsky Normal Form (CNF) provided a comprehensive understanding of formal languages and finite automata concepts. The implementation covered various essential methods for grammar normalization, including removing duplicates, eliminating unit productions, removing inaccessible symbols, removing empty strings, and finally, converting to CNF.

Each method was meticulously designed and implemented to fulfill specific objectives, contributing to the overall goal of achieving a grammar in CNF. Notably, the iterative approach employed in methods like `remove_inaccessible` and `eliminate_unit_productions` ensured thorough pruning of the grammar, removing unnecessary components while preserving its structure and validity.

The implementation showcased a blend of fundamental algorithms, recursion, and data manipulation techniques, reflecting a deep understanding of the underlying theoretical concepts. Moreover, the optional inclusion of unit tests demonstrated a commitment to quality assurance, ensuring the functionality's reliability and robustness across different scenarios.

By successfully completing this laboratory work, valuable insights were gained into grammar normalization techniques, paving the way for further exploration and application in fields like natural language processing, compiler construction, and formal language theory. Overall, the experience enhanced problem-solving skills and reinforced theoretical knowledge, contributing to a well-rounded understanding of formal languages and automata.