MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII

AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software și Automatică

Mihalachi Mihai FAF-221

Report

Laboratory work no.5

of Formal Languages & Finite Automata

Chișinău – 2024

Theory

Chomsky Normal Form (CNF) is a way to represent context-free grammars in a standardized form. In CNF, every production rule in the grammar is of one of two forms:

1. $A \rightarrow BC$, where A, B, and C are non-terminal symbols (symbols that can be replaced with strings of other symbols).
2. $A \rightarrow a$, where A is a non-terminal symbol and a is a terminal symbol (a symbol that cannot be replaced further).

Additionally, the start symbol of the grammar must not appear on the right-hand side of any production rules except for the rule that defines it.

CNF simplifies the process of analyzing and manipulating grammars, making it easier to study their properties and design algorithms for processing them. It's named after the linguist Noam Chomsky, who contributed significantly to the theory of formal languages and grammars.

Chomsky Normal Form finds applications in various areas, including:

- Parsing Algorithms: CNF simplifies the parsing process for context-free grammars. Algorithms such as the CYK (Cocke-Younger-Kasami) algorithm work efficiently with grammars in CNF.
- Compiler Design: CNF is useful in compiler construction, particularly in the syntax analysis phase. Many parser generators, such as YACC and ANTLR, expect grammars to be in CNF.
- Natural Language Processing (NLP): CNF can be applied in NLP tasks like syntactic analysis and parsing of natural language sentences. Grammars in CNF can help identify the syntactic structure of sentences more efficiently.
- Machine Translation: CNF can aid in the development of machine translation systems by providing a structured representation of language rules, facilitating the transformation of sentences from one language to another.
- Formal Language Theory: CNF is essential in the study of formal languages and grammars. Properties of context-free grammars, such as ambiguity and equivalence, are often easier to analyze in CNF.

- Automata Theory: CNF is used in converting context-free grammars into equivalent pushdown automata or other forms, which can then be further analyzed or processed.

Overall, CNF serves as a standardized form for context-free grammars, enabling efficient processing, analysis, and manipulation in various domains within computer science and linguistics.

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 the student who will have unit tests that validate the functionality of the project.
    d. 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:

1. For the implementation of Chomsky Normal Form (CNF) I created a class called Grammar where I insert the grammar and perform the methods related to CNF that are inside the class. The first method is the one we use on every step of converting the grammar to CNF, which is *check_empty()*. This method checks if there exists a state with no production to transition. In that case I remove every reference to the state in every other production. If something was changed in the production, the list in converted to set then back to list to get rid of duplicates. After that I remove the state from the transitions and non-terminal symbols. Here is the code for this method:

```
def check_empty(self):
    states_to_remove = set()
    for key, value in self.P.items():
        if len(value) == 0:
            empty_state = key
            states_to_remove.add(key)
            for state, productions in self.P.items():
                change = False
                for production in productions:
                    if empty_state in production:
                        if len(production) == 1:
                            productions.remove(production)
                        else:
                            productions[productions.index(production)] = production.replace(empty_state, "")
                            change = True
                if change:
                    self.P[state] = list(set(productions))

    for state in states_to_remove:
        if state in self.P.keys():
            self.P.pop(state)
            self.Vn.remove(state)
```

Figure 1: check_empty method in Python

2. The first method I call when converting to CNF is *eliminate_epsilon()*. In this method I search for empty strings inside productions, then I take the state and find every reference of it in other productions, then inside those productions I generate every possible combination of those productions with and without the state with epsilon string in their production. The code for generating combinations is stored in a function called *generate_combinations()*. After that I append the generated combinations into the production and then I call *check_empty()* in case there are empty states. Here is the code snippet for this method:

```
def eliminate_epsilon(self):
    epsilon_state = ''
    for state, productions in self.P.items():
        if '' in productions:
            epsilon_state = state
            productions.remove('')
            for key, value in self.P.items():
                new_productions = []
                for production in value:
                    if epsilon_state in production and len(production) > 1:
                            new_productions += self.generate_combinations(production, epsilon_state)
                value += new_productions
    # print(self.P)
    self.check_empty()
    # print(self.P)
    return self.P
```

Figure 2: eliminate_epsilon method in Python

3. The next method we call after eliminating epsilon productions is *eliminate_unit()*. In this method, I check every production if they have a unit production, which in our case will be a string containing one uppercase letter. If there is one, we add it into a separate dictionary with the state where the unit production is as key and the unit production as value, and since there can be more than one unit production, the value is an array. After that we remove the unit production inside the state. If there are no unit productions we simply return as it is, otherwise we add inside the state we found unit productions the productions for each unit production inside the state. As usual we remove the duplicates and after we're done we check for empty states. Here is the code for this method:

```
def eliminate_unit(self):
    unit_productions = {}
    for state, productions in self.P.items():
        for production in productions:
            if len(production) == 1 and production[0].isupper():
                productions.remove(production)
                unit_productions.setdefault(state, []).append(production)

    if not bool(unit_productions):
        return self.P

    for state in self.P.keys():
        if state in unit_productions.keys():
            for unit in unit_productions[state]:
                self.P[state].extend(self.P[unit])
                self.P[state] = list(set(self.P[state]))

    self.check_empty()
    # print(self.P)
    return self.P
```

Figure 3: eliminate_unit method in Python

4. The next method we call after eliminating unit productions is *eliminate_inaccessible()*. In this method, I check if every state is called in productions that are not their own by iterating through every state starting with the start symbol. We have a list of symbols that we have to check and if they are reached, we remove them from that list. After this, if there exists any inaccessible symbols we eliminate them from the dictionary and non-terminal symbols. The method for checking empty states is called. Here is the code for this method:

```
def eliminate_inaccessible(self):
    reachable_symbols = set()
    pending_symbols = [list(self.P.keys())[0]]

    while pending_symbols:
        symbol = pending_symbols.pop(0)
        reachable_symbols.add(symbol)
        for production in self.P[symbol]:
            for char in production:
                if char.isupper() and char not in reachable_symbols:
                    pending_symbols.append(char)

    unreachable_symbols = set(self.P.keys()) - reachable_symbols
    for symbol in unreachable_symbols:
        del self.P[symbol]
        self.Vn.remove(symbol)

    for symbol, productions in self.P.items():
        self.P[symbol] = [prod for prod in productions if all(char not in unreachable_symbols for char in prod)]
    # print(self.P)
    self.check_empty()
```

Figure 4: eliminate_inaccessible method in Python

5. In CNF after inaccessible symbols we check for non-productive symbols. In our class there is a method called *eliminate_non_productive()*. In this method, I check if every state is productive, and by that I mean if a state is not infinitely recursive, but instead can transition into a terminal symbol. I do that by cheking if the state has a terminal symbol as a separate string inside its productions. If not then we first check if there is only one production and it has its state inside it, which means it's non-productive and we must remove it. If it's not, then we check if all of the non-terminal symbols inside the productions are productive. We store the productive symbols if we find them and we use a while to iterate thorugh the transitions until there is no change to the set of productive symbols. After that we check if there are as many non-terminal symbols as productive symbols. If so, then we leave it as it is, otherwise we check which state is not productive then we empty the array so that when we will call *check_emtpy()*, it will automatically remove it with no need to write more code. The method for checking empty states is called. Here is the code for this method:

```
def eliminate_non_productive(self):
    productive_symbols = set()
    changed = True
    while changed:
        changed = False
        for symbol, productions in self.P.items():
            if symbol in productive_symbols:
                continue
            if any(len(production) == 1 and production.islower() for production in self.P[symbol]):
                productive_symbols.update(symbol)
                changed = True
                continue
            non_terminals = set()
            if len(self.P[symbol]) == 1 and symbol in self.P[symbol][0]:
                continue
            for prod in self.P[symbol]:
                non_terminals.update(char for strings in prod for char in strings if char.isupper() and char != symbol)
            if all(non_terminal in productive_symbols for non_terminal in non_terminals):
                productive_symbols.update(symbol)
                changed = True

    if len(productive_symbols) == len(self.Vn):
        return self.P
    else:
        for symbol in self.Vn:
            if symbol not in productive_symbols:
                self.P[symbol] = []
    self.check_empty()
    return self.P
```

Figure 5: eliminate_inaccessible method in Python

6. Finally, we reached the method that converts Context-Free Grammar into CNF. The method we use for conversion is called *to_cnf()*. In this method we call every method mentioned previously. After that we iterate through each state and replace the terminal symbols that are not in a string by themselves with a non-terminal symbol which we create. The non-terminal symbol letter is created by iterating through each of the uppercase letters in the alphabet then finding one that is not used as a non-terminal symbol. Then we replace the terminal symbol in the production with its non-terminal representative. Also the non-terminal symbol is added into a dictionary called new_strings, where we add the new transitions so that later we append it into the grammar's dictionary. Here is the code snippet for this part of the method:

```
for prod in productions:
    if len(prod) != 1 and not(len(prod) == 2 and prod[0].isupper() and prod[1].isupper()):
        for n in range(len(prod)):
            if prod[n].islower():
                assigned_key = next((key for key, array in new_productions.items() if prod[n] in array), None)
                if assigned_key is None:
                    while symbols[0] in self.Vn:
                        symbols = symbols.replace(symbols[0], _new: '')
                    symbol = symbols[0]
                    self.Vn.add(symbol)
                    new_productions[symbol] = list(prod[n])
                else:
                    symbol = assigned_key
                prod = prod[:n] + symbol + prod[n + 1:]
```

Figure 6: for loop that replaces terminal symbols with non-terminal

After that we use a while loop to make the production's length equal to 2 non-terminals. We do that by adding a new state with the first 2 symbols inside the production and we repeat it until the length of the production we iterate with is 2. The same thing as previously, we add the new states in the new_strings dictionary. Here is the code snippet:

```
while len(prod) > 2:
    assigned_key = next((key for key, array in new_productions.items() if prod[0] + prod[1] in array), None)
    if assigned_key is None:
        while symbols[0] in self.Vn:
            symbols = symbols.replace(symbols[0], _new: '')
        symbol = symbols[0]
        self.Vn.add(symbol)
        str = prod[0] + prod[1]
        new_productions[symbol] = [str]
    else:
        symbol = assigned_key
    prod = prod.replace(prod[0] + prod[1], symbol)
```

Figure 7: for loop that replaces terminal symbols with non-terminal

At the finale we just add the new transitions in the grammar's transitions.

Conclusion

In conclusion, establishing a laboratory focused on creating a code to convert Context-Free Grammars (CFGs) to Chomsky Normal Form (CNF) offers valuable insights into the foundational principles of formal languages and grammars. Through this endeavor, students and researchers gain a deeper understanding of the structure and transformational properties of grammars, honing their skills in algorithm design, parsing techniques, and language processing.

The significance of CNF lies in its role as a standardized representation that simplifies the analysis and manipulation of context-free grammars. By converting grammars into CNF, we streamline various computational tasks, including parsing algorithms, compiler design, natural language processing, machine translation, formal language theory, and automata theory. This standardized form not only facilitates efficient processing but also enhances the clarity and precision of grammar specifications, making it easier to reason about language structures and properties.

Thus, this laboratory not only equips learners with practical programming skills but also instills a deeper appreciation for the importance of formal language theory in diverse fields of computer science and linguistics. By mastering the conversion process and understanding the significance of CNF, students are better prepared to tackle complex language-related challenges and contribute to advancements in computational linguistics and theoretical computer science.