MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

GLIGA DANIELA, FAF-223

Report

Laboratory work nr.5 Chomsky Normal Form

of Formal Languages and Finite Automata

Checked by:

Dumitru Crețu, *university assistant*

FCIM, UTM

Chișinău – 2024

1. Theory:

A useful form for dealing with context free grammars is the Chomksy normal form. This is a particular form of writing a CFG which is useful for understanding CFGs and for proving things about them. It also makes the parse tree for derivations using this form of the CFG a binary tree. And as a CS major, I know you really love binary trees! So what is Chomsky normal form? 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 where S is the start variable, for technical reasons. Note that this means that we allow S as one of many possible rules. Okay, so if this is the Chomsky normal form what is it good for? Well as a rst fact, note that parse trees for a derivation using such a grammar will be a binary tree. Thats nice. It will help us down the road. Okay, so if it might be good for something, we can ask the natural question: is it possible to convert an arbitrary CFG into an equivalent grammar which is of the Chomsky normal form? The answer, it turns out, is yes. Lets see how such a conversion would proceed.

Step 1: Eliminate start symbol from the RHS. If the start symbol T is at the right-hand side of any production, create a new production as:

1. S1 → S

Step 2: In the grammar, remove the null, unit and useless productions. You can refer to the Simplification of CFG.

Step 3: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, production S → aA can be decomposed as:

1. S → RA
2. R → a

Step 4: Eliminate RHS with more than two non-terminals. For example, S → ASB can be decomposed as:

1. S → RS
2. R → AS

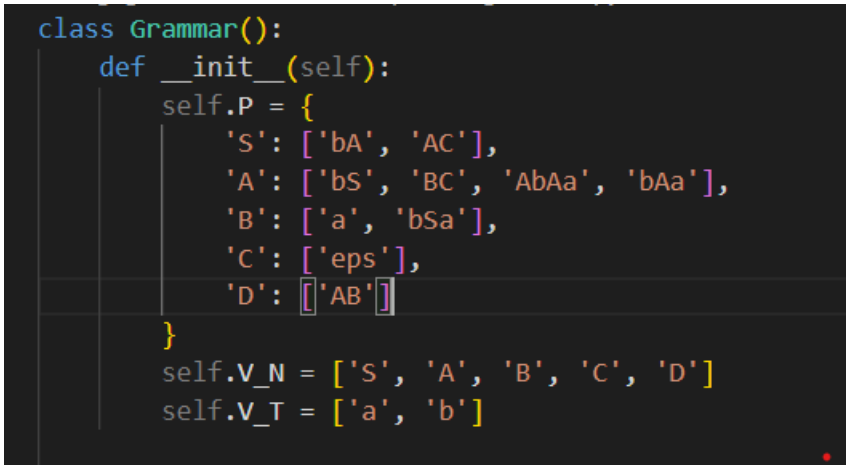
There are special forms for CFGs such as Chom sky Normal Form, where every production has the form A BC or A c. The algorithm to convert to this form involves (1) determin ing all nullable variables and getting rid of all-productions, (2) getting rid of all variable unit productions, (3) breaking up long productions, and (4) moving terminals to unit productions.

2. Objectives:

1. Learn about Chomsky Normal Form (CNF) [1].
2. Get familiar with the approaches of normalizing a grammar.
3. 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 executed and tested.
4. A BONUS point will be given for the student who will have unit tests that validate the functionality of the project.
5. 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.

3. Implementation description:

The code defines a class Grammar that performs various transformations on a context-free grammar (CFG) to convert it into Chomsky Normal Form (CNF). Let's break down each method:
init(self) method initializes the CFG with production rules (self.P), non-terminal symbols (self.V_N), and terminal symbols (self.V_T).



```
class Grammar():
    def __init__(self):
        self.P = {
            'S': ['bA', 'AC'],
            'A': ['bS', 'BC', 'AbAa', 'bAa'],
            'B': ['a', 'bSa'],
            'C': ['eps'],
            'D': ['AB']
        }
        self.V_N = ['S', 'A', 'B', 'C', 'D']
        self.V_T = ['a', 'b']
```

Remove_Epsilon(self) method removes epsilon productions from the CFG. Identify Epsilon-Producing Non-terminals Initialize an empty list nt_epsilon to store non-terminals that produce epsilon. Iterate through each key-value pair in self.P, where keys represent non-terminals and values represent their productions. If a production contains epsilon ('eps'), add the corresponding non-terminal to nt_epsilon.
Remove Epsilon Productions

Create a copy P1 of the original production rules self.P. Iterate through each key-value pair in self.P. For each non-terminal ep in nt_epsilon and each production v in the current non-terminal's productions: Create a copy prod_copy of the production v.
If the epsilon-producing non-terminal ep is found in prod_copy, replace ep with an empty string in prod_copy. Add the modified production prod_copy to the list of productions for the current non-terminal.
Remove Non-Essential Epsilon Productions
Iterate through each key-value pair in self.P again. If a non-terminal (key) is in nt_epsilon and has less than 2 productions (meaning it can derive epsilon only), remove it from P1. For each production v of the non-terminal: If v contains 'eps', remove 'eps' from its productions in P1.
Update and Return
Update self.P with the modified production rules P1. Print the modified production rules after epsilon removal. Return the modified production rules P1.

```python
def Remove_Epsilon(self):
    nt_epsilon = []
    for key, value in self.P.items():
        s = key
        productions = value
        for p in productions:
            if p == 'eps':
                nt_epsilon.append(s)

    P1 = self.P.copy()
    for key, value in self.P.items():
        for ep in nt_epsilon:
            for v in value:
                prod_copy = v
                if ep in prod_copy:
                    for c in prod_copy:
                        if c == ep:
                            value.append(prod_copy.replace(c, ''))

    for key, value in self.P.items():
        if key in nt_epsilon and len(value) < 2:
            del P1[key]
        else:
            for v in value:
                if v == 'eps':
                    P1[key].remove(v)

    print(f"1. After removing epsilon productions:\n{P1}")
    self.P = P1.copy()
    return P1
```

Eliminate_Unit_Prod method works in the following way:
Create a copy P2 of the original production rules self.P. Iterate Through Productions
Iterate through each key-value pair in self.P, where keys represent non-terminals and values represent their productions. For each production v of the current non-terminal:

Check if the length of the production v is 1 (len(v) == 1) and if it is a non-terminal (v in self.V_N). If a unit production is found (i.e., a single non-terminal production): Remove this unit production from the productions of the current non-terminal in P2. Add the productions of the unit non-terminal (self.P[v]) to the productions of the current non-terminal in P2. Update self.P with the modified production rules P2. Print the modified production rules after eliminating unit productions. Return the modified production rules P2

```
def Eliminate_Unit_Prod(self):
    P2 = self.P.copy()
    for key, value in self.P.items():
        for v in value:
            if len(v) == 1 and v in self.V_N:
                P2[key].remove(v)
                for p in self.P[v]:
                    P2[key].append(p)
    print(f"2. After removing unit productions:\n{P2}")
    self.P = P2.copy()
    return P2
```

Eliminate_Inaccesible_Symbols method works as following:
Create a copy P2 of the original production rules self.P. Iterate Through Productions Iterate through each key-value pair in self.P, where keys represent non-terminals and values represent their productions. For each production v of the current non-terminal: Check if the length of the production v is 1 (len(v) == 1) and if it is a non-terminal (v in self.V_N). If a unit production is found (i.e., a single non-terminal production): Remove this unit production from the productions of the current non-terminal in P2. Add the productions of the unit non-terminal (self.P[v]) to the productions of the current non-terminal in P2. Update self.P with the modified production rules P2.Print the modified production rules after eliminating unit productions. Return the modified production rules P2.

```
def Eliminate_Inaccesible_Symbols(self):
    P3 = self.P.copy()
    accesible_symbols = self.V_N
    for key, value in self.P.items():
        for v in value:
            for s in v:
                if s in accesible_symbols:
                    accesible_symbols.remove(s)

    for el in accesible_symbols:
        del P3[el]
    print(f"3. After removing inaccessible symbols:\n{P3}")
    self.P = P3.copy()
    return P3
```

The Remove_Nonproductive method identifies nonproductive symbols by counting terminal productions for each non-terminal in the grammar. Nonproductive non-terminals are then removed along with their productions and references in other productions. Additionally, unreachable non-terminals are identified and removed to ensure the resulting grammar contains only productive and reachable symbols. The method iterates through the productions and symbols to update the grammar P4 accordingly. After the removal process, the modified grammar P4 is printed and returned as the final result.

```
def Remove_Nonproductive(self):
    P4 = self.P.copy()
    for key, value in self.P.items():
        count = 0
        for v in value:
            if len(v) == 1 and v in self.V_T:
                count += 1
        if count == 0 and key not in self.V_T:
            del P4[key]
            for k, v in self.P.items():
                for e in v:
                    if k == key:
                        break
                    else:
                        if key in e:
                            P4[k].remove(e)

    for key, value in self.P.items():
        for v in value:
            for c in v:
                if c.isupper() and c not in P4.keys() and c != key:
                    P4[key].remove(v)
                    break

    print(f"4. After removing non-productive symbols:\n{P4}")
    self.P = P4.copy()
    return P4
```

The Chomsky_Normal_Form method transforms the given CFG into Chomsky Normal Form (CNF), where every production is either of the form A -> BC (where A, B, and C are non-terminals) or A -> a (where a is a terminal). It creates a copy P5 of the original production rules self.P to work with without modifying the original grammar. It initializes a dictionary temp to store temporary symbols and a list free_symbols containing unused non-terminal symbols from the given vocabulary. It iterates through each production in the grammar to analyze and modify productions

that are not in CNF. For productions that are already in CNF (either single terminals or pairs of non-terminals), it skips further processing. For productions that are not in CNF (i.e., more than two symbols or a mix of terminals and non-terminals), it splits the production into two parts: left and right. It checks if the left part of the production has been encountered before in temp, using it as a temporary key, or assigns a new temporary key from free_symbols if it's a new left part. It repeats the process for the right part of the production, assigning a temporary key if necessary. It updates the production in P5 with the temporary keys, ensuring that each production follows the CNF structure. After processing all productions, it updates P5 with the productions represented by the temporary keys in temp. Finally, it prints the grammar P5 in CNF and returns it as the modified grammar ready for further use in CNF-related algorithms or analysis.

4. Screen printing of program output:

The provided output demonstrates the stepwise transformations applied to the initial grammar to achieve its Chomsky Normal Form (CNF).

```
Initial Grammar:
{'S': ['bA', 'AC'], 'A': ['bS', 'BC', 'AbAa', 'bAa'], 'B': ['a', 'bSa'], 'C': ['eps'], 'D': ['AB']}
1. After removing epsilon productions:
{'S': ['bA', 'AC', 'A'], 'A': ['bS', 'BC', 'AbAa', 'bAa', 'B'], 'B': ['a', 'bSa'], 'D': ['AB']}
2. After removing unit productions:
{'S': ['bA', 'AC', 'bS', 'BC', 'AbAa', 'bAa', 'a', 'bSa'], 'A': ['bS', 'BC', 'AbAa', 'bAa', 'a', 'bSa'], 'B': ['a', 'bSa'], 'D': ['AB']}
3. After removing inaccessible symbols:
{'S': ['bA', 'AC', 'bS', 'BC', 'AbAa', 'bAa', 'a', 'bSa'], 'A': ['bS', 'BC', 'AbAa', 'bAa', 'a', 'bSa'], 'B': ['a', 'bSa']}
4. After removing non-productive symbols:
{'S': ['bA', 'bS', 'AbAa', 'bAa', 'a', 'bSa'], 'A': ['bS', 'AbAa', 'bAa', 'a', 'bSa'], 'B': ['a', 'bSa']}
5. Final CNF:
{'S': ['CD', 'CE', 'FG', 'CG', 'a', 'CH'], 'A': ['CE', 'FG', 'CG', 'a', 'CH'], 'B': ['a', 'CH'], 'C': ['b'], 'D': ['A'], 'E': ['S'], 'F': ['Ab'], 'G': ['Aa'], 'H':
['Sa']}
```

It shows the output of each method implemented which represent the rules of obtaining a Chomsky Normal Form from Regular Grammar.

5. Conclusions:

In conclusion, during this laboratory work, I've gained valuable insights into formal language theory and computational linguistics. This project allowed me to understand the intricate details of context-free grammars (CFGs) and their transformations, such as epsilon removal, unit production elimination, and Chomsky Normal Form conversion. Through implementing these transformations in code, I honed my programming skills, especially in data structures and algorithms relevant to parsing and grammar analysis. Working on the CFGConverter deepened my understanding of nonterminals, terminals, and production rules in CFGs, essential concepts in language processing and

compiler design. I faced challenges in handling various grammar structures and edge cases, which improved my problem-solving abilities and attention to detail. Collaborating with peers during code reviews and discussions enriched my learning experience and exposed me to diverse perspectives on CFG manipulation. The iterative process of testing different grammars against the converter reinforced the importance of unit testing and validation in software development. Overall, this laboratory work not only enhanced my technical proficiency but also fostered a deeper appreciation for the theoretical foundations underlying formal languages and grammars in computational linguistics.