K. J. Somaiya College of Engineering, Mumbai-77

(Autonomous College Affiliated to University of Mumbai)

Division: B
Roll No.: 1914078, 1914092, 1914094

IA : II

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

TITLE: Automata Development

AIM: To Automate the conversion of Context Free Grammar to Greibach Normal Form

Title: Automate CFG to GNF

Theme: Conversion of Grammar to Normal Form

Used Tools and Technologies: Python, VS Code Editor

Expected OUTCOME of Experiment:
The outcome of the experiment is to automate the process of converting Context Free Grammar into a normal form i.e Greibach Normal Form to remove unnecessary ambiguity.
CO1: Design mathematical models of computation
CO2: Comprehend significance

Books/ Journals/ Websites referred:

https://en.wikipedia.org/wiki/Context-free_grammar
https://www.javatpoint.com/context-free-grammar
https://brilliant.org/wiki/context-free-grammars/
https://brilliant.org/wiki/context-free-grammars/\
https://www.geeksforgeeks.org/converting-context-free-grammar-chomsky-normal-form/
https://www.geeksforgeeks.org/converting-context-free-grammar-greibach-normal-form/
https://liyanxu.blog/2019/03/02/context-free-grammar-chomsky-normal-form-pushdown-automaton/

Problem Definition:
To understand the problem better, let us see the problem by a set of question and answers to better understand the topic as well as our reasoning for conversion.

What is CFG(Context Free Grammar)?

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.
Context-free grammars can generate context-free languages. They do this by taking a set of variables which are defined recursively, in terms of one another, by a set of production rules.
A context-free grammar can be described by a four-element tuple (V,Σ,R,S), where

- V: V is a finite set of variables (which are non-terminal);
- Σ: Σ is a finite set (disjoint from V) of terminal symbols;
- R: R is a set of production rules where each production rule maps a variable to a string $s \in (V \cup \Sigma)*$;
- S (which is in V) which is a start symbol.

Production rules are of the form
$A \rightarrow \alpha$
with A a single nonterminal symbol, and α a string of terminals and/or nonterminals (α can be empty).
A formal grammar is "context free" if its production rules can be applied regardless of the context of a nonterminal. No matter which symbols surround it, the single nonterminal on the left hand side can always be replaced by the right hand side.
eg)
L= {wcw^R | w € (a, b)*}
Production rules:
S → aSa
S → bSb
S → c

Now check that abbcbba string can be derived from the given CFG.
S ⇒ aSa
S ⇒ abSba
S ⇒ abbSbba
S ⇒ abbcbba

What is the need for conversion to other forms of grammar?

The sequence of substitution is called **derivation**, and this chain of substitution can be represented by **parse tree**.
eg)
Consider a production like: S→ S+S | SxS | (S) | a
We can make **a+a x a** via two methods like:-
S ⇒ S
S ⇒ S+S
S ⇒ S + (S x S)
S ⇒ a + (a x a)
S ⇒ a + a x a
OR
S ⇒ S
S ⇒ S x S
S ⇒ (S + S) x S
S ⇒ (a + a) x a
S ⇒ a + a x a

Ambiguity is a problem during parsing/derivation. Sometimes a grammar can generate the same string in several different ways; such a string will have several different parse trees and thus several different meanings. If a grammar generates the same string in several different ways, we say that the grammar is **ambiguous**.
Converting to Normal Forms like GNF puts some constraints on the grammar rules while preserving the same language. The benefit is that if a grammar is in GNF, then we can avoid the ambiguity problem during parsing.

What is GNF(Greibach Normal Form)?
GNF stands for Greibach normal form. A CFG(context free grammar) is in GNF(Greibach normal form) if all the production rules satisfy one of the following conditions:

1. A start symbol (S) generating ε. For example, S → ε.
2. A non-terminal generating a terminal. For example, A → a.
3. A non-terminal generating a terminal which is followed by any number of non-terminals. For example, S → aASB.

eg) Consider two Grammar:-
G1 = {S → aAB | aB, A → aA| a, B → bB | b}
G2 = {S → aAB | aB, A → aA | ε, B → bB | ε}

The production rules of Grammar G1 satisfy the rules specified for GNF, so the grammar G1 is in GNF.

However, the production rule of Grammar G2 does not satisfy the rules specified for GNF as A → ε and B → ε contains ε (only start symbol can generate ε). So the grammar G2 is not in GNF.

Algorithm:

Steps to Convert CFG to GNF are as following:-

1. Take in Context Free Grammar as input.
2. Convert the grammar into CNF.
   If the given grammar is not in CNF, convert it into CNF. Which can be done as:
   a. Eliminate start symbol from RHS.
      If start symbol S is at the RHS of any production in the grammar, create a new production as:
      S0->S
      where S0 is the new start symbol.
   b. Eliminate null productions.
   c. Eliminate unit productions.
   d. Eliminate useless productions.
   e. Eliminate terminals from RHS if they exist with other terminals or non-terminals. e.g,; production rule X->xY can be decomposed as:
      X->ZY
      Z->x
   f. Eliminate RHS with more than two non-terminals.
      e.g,; production rule X->XYZ can be decomposed as:
      X->PZ
      P->XY
3. If in the grammar there exists left recursion, eliminate it.
4. In the grammar, convert the given production rule into GNF form.
5. Give output

Implementation details:

Code:-

```
import copy

# This function is used to print the productions
def print_productions(productions):
    for i in productions:
        print(i + "->" + "|".join(productions[i]))


# This function converts given CFG to CNF
def CFG_to_CNF(dict):
    # Step 1.1 check if S is on RHS of any production
    # If it is, make new production S'->S
    flag = False
    for i in dict:
        for j in dict[i]:
            if "S" in j:
                flag = True
    if flag == True:
        dict["S'"] = ["S"]

    print("\nProductions after Step 1.1:")
    print_productions(dict)

    # Step 1.2 eliminating null productions
    for i in dict:
        if "e" in dict[i]:
            dict[i].pop(dict[i].index("e"))
            for j in dict:
                for k in dict[j]:
                    if i in k:
                        new_list = []
                        for elem in dict[i]:
                            new_str = k[: k.index(i)] + elem +
k[k.index(i) + 1 :]
                            new_list.append(new_str)
                        dict[j] = dict[j] + new_list
                        dict[j].pop(dict[j].index(k))
    print("\nProductions after Step 1.2 (Null Production Removal):")
    print_productions(dict)

    # Making useful variables
```

```
singleUnit = [
    "A",
    "B",
    "C",
    "D",
    "E",
    "F",
    "G",
    "H",
    "I",
    "J",
    "K",
    "L",
    "M",
    "N",
    "O",
    "P",
    "Q",
    "R",
    "S",
    "T",
    "U",
    "V",
    "W",
    "X",
    "Y",
    "Z",
]
singleTerminal = [
    "a",
    "b",
    "c",
    "d",
    "e",
    "f",
    "g",
    "h",
    "i",
    "j",
    "k",
    "l",
    "m",
    "n",
    "o",
```

```
        "p",
        "q",
        "r",
        "s",
        "t",
        "u",
        "v",
        "w",
        "x",
        "y",
        "z",
    ]
    variables = list(dict.keys())
    tempList = []
    for i in dict:
        for j in dict[i]:
            tempList = tempList + list(j)
    terminals = [item for item in list(set(tempList)) if item not in
variables]
    newUnits = [item for item in singleUnit if item not in variables]
    newTerminals = [item for item in singleTerminal if item not in
terminals]

    # Step 1.3 eliminating unit productions
    for k in range(len(dict)):
        for i in dict:
            for j in dict[i]:
                if len(j) == 1 and j in singleUnit:
                    dict[i].pop(dict[i].index(j))
                    dict[i] = dict[i] + dict[j]

    print("\nProductions after Step 1.3 (Unit Production Removal):")
    print_productions(dict)

    # Step 1.4 Eliminating terminal-variable and terminal-terminal pairs
    new_dict = {}
    for i in dict:
        if len(dict[i]) == 1 and dict[i][0] in terminals:
            new_dict[dict[i][0]] = i
    for i in dict:
        new_prods = []
        for j in dict[i]:
            if len(j) >= 2:
                flag = [item for item in list(j) if item in terminals]
```

```
                if len(flag) != 0:
                    for k in flag:
                        if k not in new_dict:
                            new_dict[k] = newUnits[0]
                            newUnits.pop(0)
                    newList = []
                    for l in list(j):
                        if l in terminals:
                            newList.append(new_dict[l])
                        else:
                            newList.append(l)
                    new_prods.append("".join(newList))
                else:
                    new_prods.append(j)
            else:
                new_prods.append(j)
        dict[i] = new_prods

    for terminal in new_dict:
        dict[new_dict[terminal]] = [terminal]

    print(
        "\nProductions after Step 1.4 (Eliminating terminal-variable and
terminal-terminal pairs):"
    )
    print_productions(dict)

    # Step 1.5 getting productions to have only 2 variables
    new_dict = {}
    for i in dict:
        new_prods = []
        for j in dict[i]:
            if len(j) > 2:
                for x in range(len(j) - 2):
                    new_str = j[len(j) - 2 :]
                    if new_str not in new_dict:
                        new_dict[new_str] = newUnits[0]
                        newUnits.pop(0)
                    j = j[: len(j) - 2] + new_dict[new_str]
                new_prods.append(j)
            else:
                new_prods.append(j)
        dict[i] = new_prods
```

```
    for prod in new_dict:
        dict[new_dict[prod]] = [prod]

    print(
        "\nProductions after Step 1.5 (getting productions to have only 2
variables):"
    )
    print_productions(dict)
    return dict


# This function converts given CNF to GNF
def CNF_to_GNF(productions):
    # Changing names of non terminal symbols to be in a sequence
    temp = {}
    temp_productions = {}
    count = 65
    for i in productions:
        if i not in temp:
            key = temp[i] = chr(count)
            count += 1
        else:
            key = temp[i]
        value = []
        for j in productions[i]:
            temp_string = ""
            for k in j:
                if k.islower():
                    temp_string += k
                elif k in temp:
                    temp_string += temp[k]
                else:
                    temp_string += chr(count)
                    count += 1
                    temp[k] = temp_string[-1]
            value.append(temp_string)
        temp_productions[key] = value
    productions = temp_productions
    print(
        "\nProduction after changing the names of non terminal symbols to
be sequential:"
    )
    print_productions(productions)
```

```
    # Replacing the less than variables
    temp_productions = {}
    for i in productions:
        key = i
        value = []
        for j in productions[i]:
            if not j[0].islower():
                flag = False
                for index, k in enumerate(j):
                    if k.isupper() and k < key:
                        flag = True
                        if "k" in temp_productions:
                            value2 = temp_productions[k]
                        else:
                            value2 = productions[k]
                        for l in value2:
                            value.append(j[:index] + l + j[index + 1 :])
                        break
                if not flag:
                    value.append(j)
            else:
                value.append(j)
        temp_productions[key] = value
    productions = temp_productions
    print("\nProduction after updating the less than variables:")
    print_productions(productions)

    # Removing left recursion if there are any
    temp_productions = {}
    count = 90
    for i in productions:
        key = i
        value = []
        for j in productions[i]:
            if not j[0].islower() and j[0] == key:
                value2 = [j[1:] + chr(count), j[1:]]
                temp_productions[chr(count)] = value2
                value2 = copy.deepcopy(value)
                for k in value2:
                    value.append(k + chr(count))
                count -= 1
            else:
                value.append(j)
        temp_productions[key] = value
```

```
    productions = temp_productions
    print("\nProduction after removing left recursions:")
    print_productions(productions)

    # Replacing variables with their terminal value
    temp_productions = {}
    running = True
    while running:
        running = False
        for i in productions:
            key = i
            value = []
            for j in productions[i]:
                if not j[0].islower():
                    running = True
                    value2 = productions[j[0]]
                    for k in value2:
                        value.append(k + j[1:])
                else:
                    value.append(j)
            temp_productions[key] = value
        productions = temp_productions
        temp_productions = {}
    print(
        "\nProduction after replacing variables with their respective
values to achieve GNF:"
    )
    print_productions(productions)


# Printing instructions for user to provide the CFG in a particular
format
print("1. You can use '->', '|' and 'e' for inputing the CFG.")
print("2. Do not enter spaces.")
print("3. Enter every production rule on a new line.")
print("4. Enter '*' at the end when all production rules have been
entered.")
print("Provide the production rules for the CFG below:")

# Taking CFG input from the user
productions = {}
take_input = True
while take_input:
    temp = input()
```

```
    if temp == "*":
        take_input = False
    else:
        temp2 = temp.split("->")
        productions[temp2[0]] = temp2[1].split("|")

# Converting CFG to CNF
print("\n\nConverting CFG to CNF:-")
productions = CFG_to_CNF(productions)

# Converting CNF to GNF
print("\n\nConverting CNF to GNF:-")
CNF_to_GNF(productions)
```

Output(s):

```
PS C:\Users\Admin> python -u "c:\Users\Admin\Downloads\TOC_IA2_-_cfg_to_gnf.py"
1. You can use '->', '|' and 'e' for inputing the CFG.
2. Do not enter spaces.
3. Enter every production rule on a new line.
4. Enter '*' at the end when all production rules have been entered.
Provide the production rules for the CFG below:
S->A|bA|e
A->B|aS
B->b|e
*


Converting CFG to CNF:-

Productions after Step 1.1:
S->A|bA|e
A->B|aS
B->b|e
S'->S

Productions after Step 1.2 (Null Production Removal):
S->A|bA
A->aA|abA|b
B->b
S'->A|bA

Productions after Step 1.3 (Unit Production Removal):
S->bA|aA|abA|b
A->aA|abA|b
B->b
S'->bA|aA|abA|b

Productions after Step 1.4 (Eliminating terminal-variable and terminal-terminal pairs):
S->BA|CA|CBA|b
A->CA|CBA|b
B->b
S'->BA|CA|CBA|b
C->a

Productions after Step 1.5 (getting productions to have only 2 variables):
S->BA|CA|CD|b
A->CA|CD|b
B->b
S'->BA|CA|CD|b
C->a
D->BA
```

```
Converting CNF to GNF:-

Production after changing the names of non terminal symbols to be sequential:
A->BC|DC|DE|b
C->DC|DE|b
B->b
F->BC|DC|DE|b
D->a
E->BC

Production after updating the less than variables:
A->BC|DC|DE|b
C->DC|DE|b
B->b
F->bC|aC|aE|b
D->a
E->bC

Production after removing left recursions:
A->BC|DC|DE|b
C->DC|DE|b
B->b
F->bC|aC|aE|b
D->a
E->bC

Production after replacing variables with their respective values to achieve GNF:
A->bC|aC|aE|b
C->aC|aE|b
B->b
F->bC|aC|aE|b
D->a
E->bC
```

```
PS C:\Users\Admin> python -u "c:\Users\Admin\Downloads\TOC_IA2_-_cfg_to_gnf.py"
1. You can use '->', '|' and 'e' for inputing the CFG.
2. Do not enter spaces.
3. Enter every production rule on a new line.
4. Enter '*' at the end when all production rules have been entered.
Provide the production rules for the CFG below:
S->CA|BB
B->b|SB
C->b
A->a
*


Converting CFG to CNF:-

Productions after Step 1.1:
S->CA|BB
B->b|SB
C->b
A->a
S'->S

Productions after Step 1.2 (Null Production Removal):
S->CA|BB
B->b|SB
C->b
A->a
S'->S

Productions after Step 1.3 (Unit Production Removal):
S->CA|BB
B->b|SB
C->b
A->a
S'->CA|BB

Productions after Step 1.4 (Eliminating terminal-variable and terminal-terminal pairs):
S->CA|BB
B->b|SB
C->b
A->a
S'->CA|BB

Productions after Step 1.5 (getting productions to have only 2 variables):
S->CA|BB
B->b|SB
C->b
A->a
S'->CA|BB
```

```
Converting CNF to GNF:-

Production after changing the names of non terminal symbols to be sequential:
A->BC|DD
D->b|AD
B->b
C->a
E->BC|DD

Production after updating the less than variables:
A->BC|DD
D->b|BCD|DDD
B->b
C->a
E->bC|bD|ADD

Production after removing left recursions:
A->BC|DD
Z->DDZ|DD
D->b|BCD|bZ|BCDZ
B->b
C->a
E->bC|bD|ADD

Production after replacing variables with their respective values to achieve GNF:
A->bC|bD|bCDD|bZD|bCDZD
Z->bDZ|bCDDZ|bZDZ|bCDZDZ|bD|bCDD|bZD|bCDZD
D->b|bCD|bZ|bCDZ
B->b
C->a
E->bC|bD|bCDD|bDDD|bCDDDD|bZDDD|bCDZDDD
```

Conclusion: We have successfully converted CFG to its GNF normal Form. Hence, removing ambiguity from it and making it better for use.