# Topic: Chomsky Normal Form

# Course: Formal Languages & Finite Automata

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

**Variant 2**

1.Eliminate ε productions.
2.Eliminate any renaming.
3.Eliminate inaccessible symbols.
4. Eliminate the non productive symbols.
5. Obtain the Chomsky Normal Form.

$G=(V_N, V_T, P, S)$ $V_N=\{S, A, B, C, D\}$ $V_T=\{a, b\}$

P={1. S→aB
2. S→bA
3. A→B
4. A→b
5. A→aD
6. A→ AS
7. A→bAAB
8. A→ ε
9. B→b
10. B→bS
11. C→AB
12. D→BB}

# Laboratory work:

## Theoretical notes:

Chomsky Normal Form is a grammar where every production is either of the form A → BC or A → c (where A, B, C are arbitrary variables and c an arbitrary symbol).

Example:

S → AS | a

A → SA | b

The key advantage is that in Chomsky Normal Form, every derivation of a string of n letters has exactly 2n − 1 steps. Thus: one can determine if a string is in the language by exhaustive search of all derivations.

The conversion to Chomsky Normal Form has four main steps: 1. Get rid of all ε productions. 2. Get rid of all productions where RHS is one variable. 3. Replace every production that is too long by shorter productions. 4. Move all terminals to productions where RHS is one terminal.

1) Eliminate ε Productions

Determine the nullable variables (those that generate ε) (algorithm given earlier). Go through all productions, and for each, omit every possible subset of nullable variables. For example, if P → AxB with both A and B nullable, add productions P → xB | Ax | x.

After this, delete all productions with empty RHS

2) Eliminate Variable Unit Productions

A unit production is where RHS has only one symbol. Consider production A → B. Then for every production B → α, add the production A → α. Repeat until done (but don't re-create a unit production already deleted).

3) Replace Long Productions by Shorter Ones

For example, if have production A → BCD, then replace it with A → BE and E → CD.

4) Move Terminals to Unit Productions

For every terminal on the right of a non-unit production, add a substitute variable. For example, replace production A → bC with productions A → BC and B → b.

## Why we need it:

Consider a scenario where a compiler is processing a program written in a programming language. During compilation, the compiler relies on a context-free grammar to understand the structure and syntax of the program. A well-structured grammar greatly facilitates the compiler's task of determining the appropriate rules to apply, leading to faster and more efficient compilation. Therefore, in some cases, we might like to transform a context-free grammar into a normal form; that is, to modify the grammar in such a way that each rule follows a canonical form or template.

The main benefit of converting a grammar into Chomsky normal form comes in how we can represent and store derivations of words in memory. In CNF, every production rule either generates two nonterminal symbols or one terminal symbol. Consequently, parse trees resulting from these rules exhibit a consistent branching factor of either 2 or 1. This fact allows us to use efficient data structures for representing binary trees in memory, as well as to apply ecient algorithms to process parse trees and derivations.

## Implementation

The following implementation of the code is accepting any grammar, not only from Variant 2. For exaplanation I will use Variant 2:

**Step 1:**

For the expressions provided, there needs to be specified the implementation of the rules it needs to follow that I specified previously in theory. For example, here is the bit of code of what the program will do if it encounters long rules in the expression :

```
        for i in range(len(values)):
            if len(values[i]) > 2:
                for j in range(0, len(values[i]) - 2):
                    if j == 0:
                        rules[key][i] = rules[key][i][0] + let[0]
                        new_key = copy.deepcopy(let[0])
                    else:
                        voc.append(let[0])
```

Here, the function first makes a deep copy of the original grammar rules **(rules)** because it will modify them during the process, and it's essential to preserve the original state. It then iterates over each rule in the grammar. For each rule, the function checks if the length of the right-hand side (the value) is greater than 2, indicating a large rule. If a rule is identified as large, the function breaks it down into smaller rules that adhere to the canonical form. Specifically: A -> BCD gives 1) A-> BE (if E is the first "free" letter from letters pool) and 2) E-> CD.

It then updates the original grammar **(rules)** to reflect the new smaller rules. If **E** is the first available letter, it replaces the original large rule **A -> BCD** ith **A->BE**. Additionally, it updates the **let** list to remove the letter that was used (**E** in this case) and appends it to the **voc** list, indicating that it's now part of the vocabulary.

Finally, the function returns the updated vocabulary **(voc)** , which now includes the new symbols introduced during the process of breaking down large rules.

This is the output this function will give:

```
Rules after large rules removal
S->aB
S->bA
A->B
A->b
A->aD
A->AS
A->bE
A->e
B->b
B->bS
C->AB
D->BB
```

### Step 2

For the next step into obtaining CNF, we get rid of empty rules. The function iterates over each rule in the grammar, checking if any of the rules contain an empty string ('ε' or 'E') on the right-hand side.

If an empty rule is found, it is added to a list e_list to keep track of which symbols can derive empty strings.:

```
def empty(rules, voc):


    # list with keys of empty rules
    e_list = []
```

For each key (non-terminal symbol) in the grammar, if it has a production rule that directly derives an empty string, that empty string is removed from the rule. If a key (non-terminal symbol) ends up with no production rules after removing the empty string, it is removed from the grammar.

After removing empty strings from the rules, the function iterates over the grammar again to identify rules of the form A -> BC or A -> CB, where one symbol (either B or C) is in the e_list and the other is in the vocabulary (voc). If such a rule is found, it ensures that the corresponding non-terminal symbol (A) can also derive the other symbol. For example, if B can derive an empty string and C is in the vocabulary, then A should also be able to derive C.

```
for i in range(len(values)):
```

```
            if len(values[i]) == 2:
               if values[i][0] in e_list and key != values[i][1]:
                     rules.setdefault(key, []).append(values[i][1])
                if values[i][1] in e_list and key != values[i][0]:
                     if values[i][0] != values[i][1]:
                         rules.setdefault(key, []).append(values[i][0])
```

This is the output this function will give:

```
Rules after empty rules removal
S->aB
S->bA
S->b
A->B
A->b
A->aD
A->AS
A->bE
A->S
B->b
B->bS
C->AB
C->B
D->BB
```

**Step 3:**

The next function is responsible for handling short rules in a grammar. Short rules are rules where a non-terminal symbol directly derives only one terminal or non-terminal symbol.

The function creates a dictionary D where each non-terminal symbol (key) is initially mapped to itself. This dictionary represents the set of symbols that each non-terminal symbol can directly derive.

The values of the dictionary D are transformed from strings to lists. This transformation is done to facilitate the insertion of additional values if needed.For each non-terminal symbol A in the vocabulary (voc), the function checks every production rule in the grammar to find rules of the form B -> C, where B is in D(A) (meaning A can derive B) and C is not yet in D(A). If such a rule is found, the function adds C to the set of symbols derivable from A (D(A)). After identifying and updating the set of symbols derivable from each non-terminal symbol in D, the function calls another function short1(rules, D) to perform further processing on the rules.Finally, the function returns the updated grammar (rules) and the

updated dictionary D, which now represents the set of symbols that each non-terminal symbol can directly derive.

```
for letter in voc:
        for key in rules:
            if key in D[letter]:
                values = rules[key]
                for i in range(len(values)):
                    if len(values[i]) == 1 and values[i] not in D[letter]:
                        D.setdefault(letter, []).append(values[i])
```

For the next step, we further process the grammar. This code iterates over each rule in the grammar, and if any rule has a length of 1 on the right-hand side (i.e., it's a short rule), it removes that rule from the grammar. Additionally, if a non-terminal symbol ends up with no production rules after removing short rules, it is removed from the grammar.For each rule in the grammar, it searches for rules of the form A -> BC, where B and C are non-terminal symbols. It then replaces each such rule with multiple rules of the form A -> B'C', where B' and C' are symbols derivable from B and C respectively according to the dictionary D.

```
for key in new_dict:
        values = new_dict[key]
        for i in range(len(values)):
            if len(values[i]) == 1:
                rules[key].remove(values[i])
        if len(rules[key]) == 0:
            rules.pop(key, None)

    for key in rules:
        values = rules[key]
        for i in range(len(values)):
```

```
            for j in D[values[i][0]]:
                for k in D[values[i][1]]:
                    if j + k not in values:
                        rules.setdefault(key, []).append(j + k)
```

**Output for these steps:**

```
Rules after short rules removal
S->aB
S->bA
S->ab
S->bB
S->bb
S->bS
A->aD
A->AS
A->bE
A->Ab
A->BS
A->Bb
A->bS
A->bb
A->SS
A->Sb
B->bS
B->bb
C->AB
C->Ab
C->BB
C->Bb
C->bB
C->bb
C->SB
C->Sb
D->BB
D->Bb
D->bB
D->bb
```

**The input: (works for other variants too)**

```
Rule #1S aB
Rule #2S bA
Rule #3A B
Rule #4A b
Rule #5A aD
Rule #6A AS
Rule #7A bAAB
Rule #8A e
Rule #9B b
Rule #10B bS
Rule #11C AB
Rule #12D BB
```

## Conclusion:

This laboratory demonstrated a systematic approach to processing context-free grammars (CFGs). CNF reduces ambiguity in grammars, leading to more deterministic and efficient parsing. Each function of the task played a crucial role in transforming the grammar into a more structured form - handling various aspects of grammar manipulation, including addressing large rules, empty rules, and short rules, conducive to efficient parsing and language processing. The task showcased the importance of grammar manipulation techniques in language processing systems, as it is used in natural language processing, algorithm parsing, compiler design, grammar optimization etc. By systematically addressing various aspects of grammar refinement, the functions contributed to creating a more standardized and efficient representation of the language grammar, laying the groundwork for effective parsing and language analysis.