Context-Free Grammar (CFG)

CFG stands for context-free grammar. It is a formal grammar which is used to generate all possible patterns of strings in a given formal language. Context-free grammar G can be defined by four tuples as:

1. G = (V, T, P, S)

Where,

G is the grammar, which consists of a set of the production rule. It is used to generate the string of a language.

T is the final set of a terminal symbol. It is denoted by lower case letters.

V is the final set of a non-terminal symbol. It is denoted by capital letters.

P is a set of production rules, which is used for replacing non-terminals symbols(on the left side of the production) in a string with other terminal or non-terminal symbols(on the right side of the production).

S is the start symbol which is used to derive the string. We can derive the string by repeatedly replacing a non-terminal by the right-hand side of the production until all non-terminal have been replaced by terminal symbols.

Example 1:

Construct the CFG for the language having any number of a's over the set ∑= {a}.

Solution:

As we know the regular expression for the above language is

1. r.e. = a*

Production rule for the Regular expression is as follows:

1. S → aS   rule 1
2. S → ε   rule 2

Now if we want to derive a string "aaaaaa", we can start with start symbols.

1. S
2. aS
3. aaS   rule 1
4. aaaS   rule 1
5. aaaaS   rule 1

6. aaaaaS rule 1
7. aaaaaaS rule 1
8. aaaaaaε rule 2
9. aaaaaa

The r.e. = a* can generate a set of string {ε, a, aa, aaa,.....}. We can have a null string because S is a start symbol and rule 2 gives S → ε.

Example 2:

Construct a CFG for the regular expression (0+1)*

Solution:

{ ε ,0,1,01,10,11,00,010,..

The CFG can be given by,

Production rule (P):

1. S → 0S | 1S
2. S → ε

The rules are in the combination of 0's and 1's with the start symbol. Since (0+1)* indicates {ε, 0, 1, 01, 10, 00, 11, ...}. In this set, ε is a string, so in the rule, we can set the rule S → ε.

Example 3:

Construct a CFG for a language L = {wcwR | where w € (a, b)*}.

Solution:

The string that can be generated for a given language is {aacaa, bcb, abcba, bacab, abbcbba, ...}

The grammar could be:

1. S → aSa rule 1
2. S → bSb rule 2
3. S → c rule 3

Now if we want to derive a string "abbcbba", we can start with start symbols.

1. S → aSa
2. S → abSba from rule 2
3. S → abbSbba from rule 2

4. S → abbcbba   from rule 3

Thus any of this kind of string can be derived from the given production rules.

Example 4:

Construct a CFG for the language L = anb2n where n>=1.

Solution:

The string that can be generated for a given language is {abb, aabbbb, aaabbbbbb....}.

The grammar could be:

5. S → aSbb | abb

Now if we want to derive a string "aabbbb", we can start with start symbols.

S → aSbb

S → aabbbb

Example 5:

A CFG for the regular language corresponding to the RE $00*11*$.

Solution:

The language is the concatenation of two languages: all strings of zeroes with all strings of ones.

The grammar could be:

S → CD

C → 0C | 0

D → 1D | 1

BACKUS NAUR FORM(BNF) :

It stands for Backus-Naur Form one notation used to write grammar.

It is a formal, mathematical way to specify context-free grammars

Example:

BNF for number--

<number> ::= <digit> | <number> <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

WHERE

““::=”means “is defined as” (somevariants use “:=”instead)
“|” means “or”
Angle brackets mean a Non Terminal
Symbols without angle brackets are Terminals

<expression> ::= <expression> + <term>
| <expression> - <term>
| <term>
<term> ::= <term> *<factor>
| <term> /<factor>
| <factor>
<factor> ::= (<expression>)
|<primary>
| <variable>
| <number>

EXAMPLE 2:
There is the production for any grammar as follows:
S → aSa
S → bSb
S → c
In BNF, we can represent above grammar as follows:

S → aSa| bSb| c

EXTANDED BACKUS NAUR FORM(EBNF) :

EBNF is a few simple extensions to BNF which make expressing grammars more convenient

- “*****” The Kleene Star): means 0 or more occurrences
- “**+**” (The Kleene Cross): means 1 or more occurrences
- “[…]”:means 0 or 1 occurrences
- ()Use of parentheses for grouping
- {}* used to show arbitary sequence
- If you have a rule such as:

Examples :

- <real no> ::= <integer part>.<fraction part> | .<fraction part>

  You can replace it with:

  <real no> ::= [<integer part>].<fraction part>

- If you have a rule such as:

  <signed integer> ::= + <integer> | - <integer>

  <integer> ::= <digit> | <integer> <digit>

  You can replace it with:

  <signed integer> ::=[+|-]<digit>{ <digit>}*

  <signed integer> ::=[+|-]{ <digit>}+

- If you have a rule such as:
  <exp> ::= <term>
  | <exp> + <term>
  | <exp> - <term>

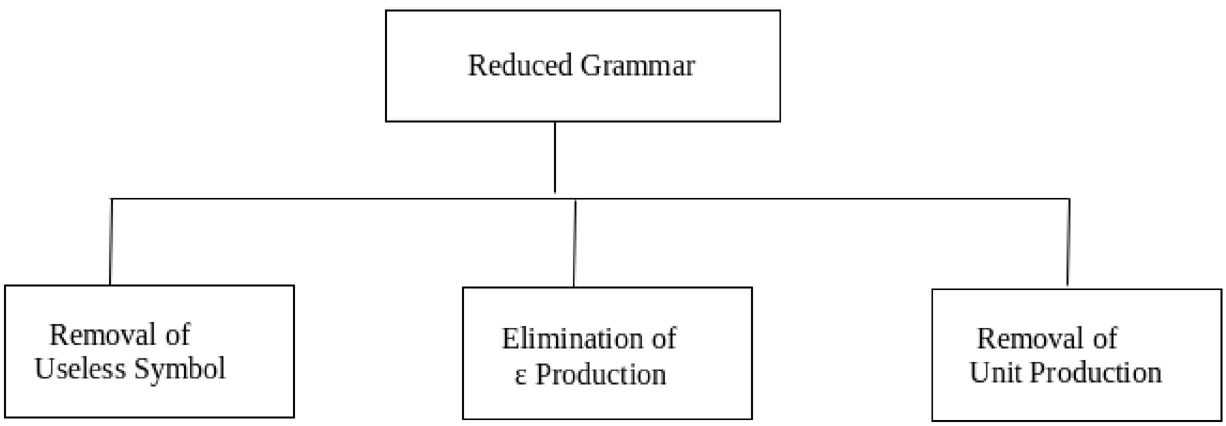
You can replace it with:

<exp> ::= <term>[(+|-)<term>]

SIMPLICATION OF CFG

As we have seen, various languages can efficiently be represented by a context-free grammar. All the grammar are not always optimized that means the grammar may consist of some extra symbols(non-terminal). Having extra symbols, unnecessary increase the length of grammar. Simplification of grammar means reduction of grammar by removing useless symbols. The properties of reduced grammar are given below:

1. Each variable (i.e. non-terminal) and each terminal of G appears in the derivation of some word in L.
2. There should not be any production as X → Y where X and Y are non-terminal.
3. If ε is not in the language L then there need not to be the production X → ε.



Removal of Useless Symbols

A symbol can be useless if it does not appear on the right-hand side of the production rule and does not take part in the derivation of any string. That symbol is known as a useless symbol. Similarly, a variable can be useless if it does not take part in the derivation of any string. That variable is known as a useless variable.

For Example:

1. T → aaB | abA | aaT
2. A → aA
3. B → ab | b
4. C → ad

In the above example, the variable 'C' will never occur in the derivation of any string, so the production C → ad is useless. So we will eliminate it, and the other productions are written in such a way that variable C can never reach from the starting variable 'T'.

Production A → aA is also useless because there is no way to terminate it. If it never terminates, then it can never produce a string. Hence this production can never take part in any derivation.

To remove this useless production A → aA, we will first find all the variables which will never lead to a terminal string such as variable 'A'. Then we will remove all the productions in which the variable 'B' occurs.

Elimination of ε Production

The productions of type S → ε are called ε productions. These type of productions can only be removed from those grammars that do not generate ε.

Step 1: First find out all nullable non-terminal variable which derives ε.

Step 2: For each production A → a, construct all production A → x, where x is obtained from a by removing one or more non-terminal from step 1.

Step 3: Now combine the result of step 2 with the original production and remove ε productions.

Example:

Remove the production from the following CFG by preserving the meaning of it.

1. S → XYX
2. X → 0X | ε
3. Y → 1Y | ε

Solution:

Now, while removing ε production, we are deleting the rule X → ε and Y → ε. To preserve the meaning of CFG we are actually placing ε at the right-hand side whenever X and Y have appeared.

Let us take

1. S → XYX

If the first X at right-hand side is ε. Then

1. S → YX

Similarly if the last X in R.H.S. = ε. Then

1. S → XY

If Y = ε then

1. S → XX

If Y and X are ε then,

1. S → X

If both X are replaced by ε

1. S → Y

Now,

1. S → XY | YX | XX | X | Y

Now let us consider

1. X → 0X

If we place ε at right-hand side for X then,

1. X → 0
2. X → 0X | 0

Similarly Y → 1Y | 1

Collectively we can rewrite the CFG with removed ε production as

1. S → XY | YX | XX | X | Y
2. X → 0X | 0
3. Y → 1Y | 1

Removing Unit Productions

The unit productions are the productions in which one non-terminal gives another non-terminal. Use the following steps to remove unit production:

Step 1: To remove X → Y, add production X → a to the grammar rule whenever Y → a occurs in the grammar.

Step 2: Now delete X → Y from the grammar.

Step 3: Repeat step 1 and step 2 until all unit productions are removed.

For example:

1. S → 0A | 1B | C
2. A → 0S | 00
3. B → 1 | A

4. C → 01

Solution:

S → C is a unit production. But while removing S → C we have to consider what C gives. So, we can add a rule to S.

1. S → 0A | 1B | 01

Similarly, B → A is also a unit production so we can modify it as

1. B → 1 | 0S | 00

Thus finally we can write CFG without unit production as

1. S → 0A | 1B | 01
2. A → 0S | 00
3. B → 1 | 0S | 00
4. C → 01