# MODULE 3- PART 1

# CONTEXT FREE GRAMMARS

## 3.1 Introduction to Rewrite Systems and Grammars

We'll begin with a very general computational model: Define a *rewrite system,* also called *production system* or a *rule-based system)* to be a list of rules and an algorithm for applying them. Each rule has a left-hand side and a right-hand side. For example, the following could be rewrite-system rules:

*S* → aSb

aS→ ε

aSb → bSabSa

When a rewrite system *R* is invoked on some initial string *w,* it operates as follows:

1. Set *working-string* = *w*.
2. Until told by *R* to halt do:
   - 2.1 Match the lhs of some rule against some part of *working-string*.
   - 2.2 Replace the matched part of *working-string* with the RHS of the rule that was matched.
3. Return *working-string*.

A rewrite system that is used to define a language is called a *grammar.* If *G* is a grammar, let *L(* G) be the language that *G* generates. Like every rewrite system, every grammar contains a list (almost always treated as a set, i.e., as an unordered list) of rules. Also, like every rewrite system, every grammar works with an alphabet, which we can call *V.* In the case of grammars, we will divide *V* into two subsets:

• a ***tenninal alphabet.*** generally called ∑, which contains the symbols that make up the strings in *L(* G), and

• a ***nontermlnal alphabet,*** the elements of which will function as working symbols that will be used while the grammar is operating. These symbols will disappear by the time the grammar finishes its job and generates a string. One final thing is required to specify a grammar. Each grammar has a unique start symbol, often called *S.*

To generate strings in L( G), we invoke *simple-rewrite* (G,S). *Simple-rewrite* will begin with S and will apply the rules of *G,* which can be thought of (given the control algorithm we just described) as licenses to replace one string by another. At each step of one of its derivations, some rule whose left-hand side matches somewhere in *working-string* is selected. The substring that matched is replaced by the rule's right hand side, generating a new value for *working string.*

We will use the symbol ⇒ to indicate steps in a derivation. So, for example suppose that *G* has the start symbol *S* and the rules *S* → aSb | bSa |ε Then a derivation could begin with:

*S* ⇒ aSb ⇒ aaSbb⇒ .....

At each step, it is possible that more than one rule's left-hand side matches the working string. It is also possible that a rule's left-hand side matches the working string in more than one way. In either case, there is a derivation corresponding to each alternative. It is precisely the existence of these choices that enables a grammar to generate more than one string.

Continuing with our example, there are three choices at the next step:

$S \Rightarrow$ aSb $\Rightarrow$ aaSbb $\Rightarrow$ aaaSbbb (using the first rule),

$S \Rightarrow$ aSb $\Rightarrow$ aaSbb $\Rightarrow$ aabSabb (using the second rule).

$S \Rightarrow$ aSb $\Rightarrow$ aaSbb $\Rightarrow$ aabb (using the third rule).

The derivation process may end whenever one of the following things happens:
1. The working string no longer contains any non terminal symbols (including, as a special case. when the working string is ε), or
2. There are nonterminal symbols in the working string but there is no match with the left-hand side of any rule in the grammar. For example, if the working string were *AaBb,* this would happen if the only left-hand side were *C.*

## 3.2 Context Free Grammars and Languages

We now define a C*ontext Free Grammar* (or CFG) to be a grammar in which each rule must:
• have a left-hand side that is a single nonterminal, and
• have a right-hand side.
To simplify the discussion that follows, define an *A* rule, for any nonterminal symbol *A,* to be a rule whose left hand side is *A.*
A derivation will halt whenever no rule's left-hand side matches against *working-string*. At every step, any rule that matches may be chosen.
Context-free grammar rules may have any (possibly empty) sequence of symbols on the right-band side. Because the rule format is more flexible than it is for regular grammars. The rules are more powerful. We will show some examples of languages that can be generated with context-free grammars but that can not be generated with regular ones.
All of the following are allowable context-free grammar rules (assuming appropriate alphabets):
S→ aSb
$S \rightarrow \varepsilon$
$T \rightarrow T$
S→ aSbbT
The following are not allowable context-free grammar rules:
$ST \rightarrow aSb$
a→ aSb
ε→ a
The name for these grammars, "context-free," makes sense because, using these rules, the decision to replace a nonterminal by some other sequence is made without looking at the context in which the non terminal occurs.

**Formal definition of CFG:**
Formally, a context-free grammar *G* is a quadruple ( *V,* ∑, *R,* S). where:

• Vis the rule alphabet, which contains nonterminals (symbols that are used in the grammar but that do not appear in strings in the language) and terminals,
• $\sum$ (the set of terminals) is a subset of *V,*
• *R* (the set of rules) is a finite subset of ( *V*- $\sum$) *X V**, and
• *S* (the start symbol) can be any element of *V* - $\sum$.
Given a grammar G. define $x \Rightarrow_G y$ (abbreviated $\Rightarrow$ when *G* is clear from context) to be the binary relation *derives-in-one-step,* defined so that:

$$\forall x, y \in V^*(x \Rightarrow_G y \text{ iff } x = \alpha A \beta, y = \alpha\gamma\beta, \text{ and there exists a rule } A \rightarrow \gamma \text{ in } R_G)$$

Any sequence of the form $w_0 \Rightarrow_G w_1 \Rightarrow_G w_2 \Rightarrow_G \ldots \Rightarrow_G w_n$ is called a ***derivation*** in *G*. Let $\Rightarrow_G^*$ be the reflexive, transitive closure of $\Rightarrow_G$. We'll call $\Rightarrow_G^*$ the ***derives*** relation.

The *language generated by G,* denoted *L(* G). is {w€$\sum$ * : $S \Rightarrow_G$ * w}. In other words, the language generated by *G* is the set of all strings of terminals that can be derived from *S* using zero or more applications of rules in G. A language *L* is *context-free* iff it is generated by some context-free grammar *G.* The context-free languages (or CFLs) are a proper superset of the regular languages.

**<u>Recursive and self embedding grammar/rules:</u>**

A grammar is **recursive** if it contains at least one production (rule) of the following forms:

S → wSx, w or x may be empty

S → wTx, T → uSv

Any set of rules that begin at terminal S and derive terminal S and w,x,u,v are elements of V*

- A rule is recursive iff it is $X \rightarrow w_1Yw_2$, where:

  $Y \Rightarrow^* w_3Xw_4$ for some $w_1$, $w_2$, $w_3$, and w in V*.
- A grammar is recursive iff it contains at least one recursive rule.
- Examples: S → (S) S → (T) T → (S)

A grammar is **self-embedding** if it contains at least one production (rule) of the following form:

S → wTx, T → uSv where w,x,u,v are elements of $\sum^+$

Self embedding grammar allows development of non-empty strings on both sides of the embedded non-terminal.

Ex: A non-empty string can be formed on both sides of a non-terminal

- S → aSb
- S → aT, T → Sb

Which is equivalent to S → aSb

- A rule in a grammar G is self-embedding iff it is :

  $X \rightarrow w_1Yw_2$, where $Y \Rightarrow^* w_3Xw_4$ and

  both $w_1w_3$ and $w_4w_2$ are in $\Sigma^+$.
- A grammar is self-embedding iff it contains at least one self-embedding rule.

- Example: $S \rightarrow aSa$ is self-embedding
  $S \rightarrow aS$ is recursive but not self- embedding
  $S \rightarrow aT$
  $T \rightarrow Sa$ is self-embedding

## 3.3 Designing CFG

**EXAMPLE 11.1 The Balanced Parentheses Language**

Consider Bal = $\{w \in \{), (\}^* :$ the parentheses are balanced$\}$. We showed in Example 8.10 that Bal is not regular. But it is context-free because it can be generated by the grammar $G = \{\{S, ), (\}, \{), (\}, R, S)$, where:

$$R = \{S \rightarrow (S) \\ S \rightarrow SS \\ S \rightarrow \varepsilon\}.$$

Some example derivations in $G$:

$$S \Rightarrow (S) \Rightarrow ().$$
$$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow ((\,)S) \Rightarrow ((\,)(S)) \Rightarrow ((\,)(\,)).$$

So, $S \Rightarrow^* ()$ and $S \Rightarrow^* ((\,)(\,))$.

**EXAMPLE 11.2 $A^nB^n$**

Consider $A^nB^n = \{a^nb^n : n \geq 0\}$. We showed in Example 8.8 that $A^nB^n$ is not regular. But it is context-free because it can be generated by the grammar $G = \{\{S, a, b\}, \{a, b\}, R, S\}$, where:

$$R = \{S \rightarrow aSb \\ S \rightarrow \varepsilon\}.$$

**EXAMPLE 11.3 Even Length Palindromes**

Consider PalEven = $\{ww^R : w \in \{a, b\}^*\}$, the language of even-length palindromes of a's and b's. We showed in Example 8.11 that PalEven is not regular. But it is context-free because it can be generated by the grammar $G = \{\{S, a, b\}, \{a, b\}, R, S\}$, where:

$$R = \{S \rightarrow aSa \\ S \rightarrow bSb \\ S \rightarrow \varepsilon\}.$$

**EXAMPLE 11.4 Equal Numbers of a's and b's**

Let $L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$. We showed in Example 8.14 that $L$ is not regular. But it is context-free because it can be generated by the grammar $G = \{\{S, a, b\}, \{a, b\}, R, S\}$, where:

$$R = \{S \rightarrow aSb,\ S \rightarrow bSa,\ S \rightarrow SS,\ S \rightarrow \varepsilon\}.$$

**BNF(Backus Naur form):** A notation for writing practical context-free grammars. The symbol | should be read as "or". Allow a nonterminal symbol to be any sequence of characters surrounded by angle brackets.

Examples of nonterminals:

<program>
<variable>

**Ex 11.5 : BNF for a Java Fragment**

<block> ::= {<stmt-list>} | { }
<stmt-list> ::= <stmt> | <stmt-list> <stmt>
<stmt> ::= <block> | while (<cond>) <stmt> |
if (<cond>) <stmt> |
do <stmt> while (<cond>); |
<assignment-stmt>; |
return | return <expression> |
<method-invocation>;

**Ex 11.6: ENGLISH Grammar CFG:** (NP will derive noun phases, VP will derive verb phases.)

*S* → *NP VP*
*NP* → the *Nominal* | a *Nominal* | *Nominal* |
*ProperNoun* | *NP PP*
*Nominal* → *N* | *Adjs N*
*N* → cat | dogs | bear | girl | chocolate | rifle
*ProperNoun* → Chris | Fluffy
*Adjs* → *Adj Adjs* | *Adj*
*Adj* → young | older | smart
*VP* → *V* | *V NP* | *VP PP*
*V* → like | likes | thinks | shots | smells
*PP* → *Prep NP*
*Prep* → with

**Ex 11.7: unequal a's and b's**

$L = \{a^n b^m : n \neq m\}$

$G = (V, \Sigma, R, S)$, where

$V = \{a, b, S, A, B\}$,

$\Sigma = \{a, b\}$,

$R =$

| | |
|---|---|
| $S \rightarrow A$ | /* more a's than b's |
| $S \rightarrow B$ | /* more b's than a's |
| $A \rightarrow a$ | /* at least one extra a generated |
| $A \rightarrow aA$ | /* any number of a's |
| $A \rightarrow aAb$ | */* equal number of a's and b's* |
| $B \rightarrow b$ | /* at least one extra b generated |
| $B \rightarrow Bb$ | /* any number of b's |
| $B \rightarrow aBb$ | */* equal number of a's and b's* |

**Ex 11.8: L ={w | number of a's > number of b's}**

| | |
|---|---|
| S→ AB | [more 'a' at the end] |
| S → BA | [more 'a' at the begining] |
| S→ ABA | [more 'a' at middle] |
| A→ aAb \| bAa\| AA \| ε | [equal a's and b's][ |
| B→ aB \| a | [one or more a's] |

***<u>NOTE: refer class notes for more CFG examples.</u>***

## 3.4 Simplifying CFG

In this section, we present two algorithms that may be useful for simplifying context free grammars.

$G = (\{S, A, B, C, D, a, b\}, \{a, b\}, R, S)$, where

$R =$

$\{ S \rightarrow AB \mid AC$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow aA$

$C \rightarrow bCa$

$D \rightarrow AB \}$

*G* contains two useless variables: *C* is Useless because it is not able to generate any strings in ∑•. (Every time a rule is applied to a *c.* a new C is added.) *D* is useless because it is unreachable via any derivation from *S.* So any rules that mention either *C* or *D* can be removed from *G* without changing the language that is generated. We present two algorithms, one to find and remove variables like *C* that are unproductive, and one to find and remove variables like *D* that arc unreachable.

Given a grammar $G = ( V, \sum , R, S )$, we define *removeunproductive(G)* to create a new grammar *G'*. where $L$ ( G') = $L$ (G) and G' does not contain any unproductive symbols. Rather than trying to find the

unproductive symbols directly. *removeunproductive(G* )will find and mark all the productive ones. Any that are left unmarked at the end are unproductive.

Initially, all terminal symbols will be marked as productive since each of them generates a terminal string (itself). A nonterminal symbol will be marked as productive when it is discovered that there is at least one way to rewrite it as a sequence productive symbols. So *removeunproductive(G* )effectively moves backwards from terminals marking nonterminal along the way.

***removeunproductive(G: CFG) =***

1. ***G′= G.***
2. ***Mark every nonterminal symbol in G′as unproductive.***
3. ***Mark every terminal symbol in G′as productive.***
4. ***Until one entire pass has been made without any new symbol being marked do:***
   ***For each rule X → α in R do:***
   ***If every symbol in α has been marked as productive and X has not yet been marked as productive then:***
   ***Mark X as productive.***
5. ***Remove from G′every unproductive symbol.***
6. ***Remove from G′every rule that contains an unproductive symbol.***
7. ***Return G′.***

*removeunproductive(G)* must halt because there is only some finite number of nonterminals that can be marked as productive. So the maximum number of times it can execute step 4 is |V- ∑ |. Clearly *L* (G') is a subset of *L* (G) since *G′* can produce no derivations that *G* could not have produced. And *L* (G') = *L* (G) because the only derivations that *G* can perform but *G′* cannot are those that do not end with a terminal string.

Notice that it is possible that *S* is unproductive. This will happen precisely in case *L* (G) = Φ.

Next we will define an algorithm for getting rid of unreachable symbols like *D* in the grammar we presented above. Given a grammar *G* -= ( *V,* ∑, *R.* S), we define *removtunreachable(G)* to create a new grammar G' where *L* (G') = *L* (G) and *G′* does not contain any unreachable nonterminal symbols. What *removeunreachable* does is to move forward from *S,* marking reachable symbols along the way.

***removeunreachable(G: CFG) =***

1. ***G′= G.***
2. ***Mark S as reachable.***
3. ***Mark every other nonterminal symbol as unreachable.***
4. ***Until one entire pass has been made without any new symbol being marked do:***
   ***For each rule X → αAβ (where A ∈ V - Σ) in R do:***
   ***If X has been marked as reachable and A has not then:***
   ***Mark A as reachable.***
5. ***Remove from G′every unreachable symbol.***
6. ***Remove from G′every rule with an unreachable symbol on the left-hand side.***

*Return G′.Removeunreachable* must halt because there is only some finite number of nonterminals that can be marked as reachable. So the maximum number of times it can execute step 4 is |V- ∑ |. Clearly *L* (G') is a subset of *L* (G) since G' can produce no derivations that *G* could not have produced. And *L* ( G') = *L* (G) because every derivation that can be produced by G can also be produced by G'.

***<u>NOTE: Refer class notes for the problems on simplification of CFG.</u>***

## 3.5 Proving that a grammar is correct

Given some language *L* and a grammar *G,* can we actually prove that *G* is correct (i.e., that it generates exactly the strings in L)? To do so, we need to prove two things:

1. *G* generates only strings in *L,* and

z. *G* generates all the strings in *L.*

The most straightforward way to do step 1 is to imagine the process by which *G* generates a string as the following loop (a version of *simple-rewrite,* using *st* in place of *working-string* ):

***1. st = S.***

***2. Until no nonterminals are left in st do:***
***Apply some rule in R to st.***

***3. Output st.***

Then we construct a loop invariant *I* and show that:

• *I* is true when the loop begins.

• *I* is maintained at each step through the loop (i.e., by each rule application), and

• $I \wedge$ (st contains only terminal symbols) → *st* € *L.*

Step 2 is generally done by induction on the length of the generated strings.

**Example: Prove that *G* generates only strings in *L* = {$A^nB^n$ = {$a^nb^n$ : $n \geq 0$}}. *G* = ({*S*, a, b}, {a, b}, *R*, *S*),**

$R = \{ \quad S \rightarrow a\,S\,b$

$S \rightarrow \varepsilon \}.$

Let I = ($\#_a(st) = \#_b(st)) \wedge (st \in a^*(S \cup \varepsilon)\, b^*)$.

**Now we prove:**

- ***I* is true when *st* = *S*: In this case, $\#_a(st) = \#_b(st)) = 0$ and *st* is of the correct form.**
- **If *I* is true before a rule fires, then it is true after the rule fires: To prove this, we consider the rules one at a time and show that each of them preserves *I*. Rule (1) adds one a and one b to *st*, so it does not change the difference between the number of a's and the number of b's. Further, it adds the a to the left of *S* and the b to the right of *S*, so if the form constraint was satisfied before applying the rule it still is afterwards. Rule (2) adds nothing so it does not change either the number of a's or b's or their locations.**
- **If *I* is true and *st* contains only terminal symbols, then $st \in A^nB^n$: In this case, *st* possesses the three properties required of all strings in $A^nB^n$: They are composed only of a's and b's, ($\#_a(st) = \#_b(st)$), and all a's come before all b's.**

**Next we show that every string *w* in $A^nB^n$ can be generated by *G*: Every string in $A^nB^n$ is of even length, so we will prove the claim only for strings of even length. The proof is by induction on $|w|$:**

- **Base case: If $|w| = 0$, then $w = \varepsilon$, which can be generated by applying rule (2) to *S*.**
- **Prove: If every string in $A^nB^n$ of length *k*, where *k* is even, can be generated by *G*, then every string in $A^nB^n$ of length $k + 2$ can also be generated. Notice that, for any even *k*, there is exactly one string in $A^nB^n$ of length *k* : $a^{k/2}b^{k/2}$. There is also only one string of length $k + 2$, namely $aa^{k/2}b^{k/2}b$, that can be generated by first applying rule (1) to produce aSb, and then applying to *S* whatever rule sequence generated $a^{k/2}b^{k/2}$. By the induction hypothesis, such a sequence must exist.**

Example 2: prove that the following grammar is correct.

**$L = \{w \in \{a, b\}^*: \#_a(w) = \#_b(w)\}$**
$G = \{\{S, a, b\}, \{a, b\}, R, S\}$, where:

$R = \{$ $S \rightarrow aSb$ (1)
$S \rightarrow bSa$ (2)
$S \rightarrow SS$ (3)
$S \rightarrow \varepsilon$ $\}$. (4)

● Prove that $G$ generates only strings in $L$:
Let $\Delta(w) = \#_a(w) - \#_b(w)$.
Let $I = st \in \{a, b, S\}^* \wedge \Delta(st) = 0$.

I is true when $st$ = S: In this case, $\#_4$ (st) = #b(st)) = 0. So A(st) = 0.
• If $I$ is true before a rule fires. then it is true after the rule fires: The only symbols that can be added by any rule are a. b. and $S$. Rules (1) and (2) each add one a and one b to $st$, so neither of them changes Δ (st). Rules (3) and (4) add neither a's nor b's to the working string, so Δ (st) does not change.
• If $I$ is true and $st$ contains only terminal symbols, then $st$ e L: In this case, $st$ possesses the two properties required of all strings in $L$. They are composed only of a's and b's and Δ (st) = 0.

● Prove that $G$ generates all the strings in $L$:
Base case:
Induction step: if every string of length $k$ can be generated, then every string $w$ of length $k$+2 can be.
$w$ is one of: a$x$b, b$x$a, a$x$a, or b$x$b.
Suppose $w$ is a$x$b or b$x$a: Apply rule (1) or (2), then whatever sequence generates $x$.
Suppose $w$ is a$x$a or b$x$b:
Suppose $w$ is a$x$a: $|w| \geq 4$. We show that $w = vy$, where $v$ and $y$ are in $L$, $2 \leq |v| \leq k$, and $2 \leq |y| \leq k$.
If that is so, then $G$ can generate $w$ by first applying rule (3) to produce $SS$, and then generating $v$ from the first $S$ and $y$ from the second $S$. By the induction hypothesis, it must be possible for it to do that since both $v$ and $y$ have length $\leq k$.
Suppose $w$ is a$x$a: we show that $w = vy$, where $v$ and $y$ are in $L$, $2 \leq |v| \leq k$, and $2 \leq |y| \leq k$.
Build up $w$ one character at a time. After one character, we have a. $\Delta(a) = 1$. Since $w \in L$, $\Delta(w) = 0$. So $\Delta(ax) = -1$. The value of Δ changes by exactly 1 each time a symbol is added to a string. Since Δ is positive when only a single character has been added and becomes negative by the time the string a$x$ has been built, it must at some point before then have been 0. Let $v$ be the shortest nonempty prefix of $w$ to have a value of 0 for Δ. Since $v$ is nonempty and only even length strings can have Δ equal to 0, $2 \leq |v|$. Since Δ became 0 sometime before $w$ became a$x$, $v$ must be at least two characters shorter than $w$, so $|v| \leq k$. Since $\Delta(v) = 0$, $v \in L$. Since $w = vy$, we know bounds on the length of $y$: $2 \leq |y| \leq k$. Since $\Delta(w) = 0$ and $\Delta(v) = 0$, $\Delta(y)$ must also be 0 and so $y \in L$.

## 3.6 Derivations and parse trees

**<u>Derivations</u>:** The process of obtaining strings from the start symbol by applying rules is called derivation. There are two types of derivations:

- A ***left-most derivation(LMD)*** is one in which at each step the leftmost non terminal in the working string is chosen for expansion.
- A ***right-most derivation(RMD)*** is one in which at each step the rightmost non terminal in the working string is chosen for expansion.

***NOTE: Refer class notes for the examples.***

**Parse trees:** imposes a grammatical structure to the grammar.

A parse tree, derived by a grammar $G = (V, \Sigma, R, S)$, is a rooted, ordered tree in which:

- Every leaf node is labeled with an element of $\Sigma \cup \{\varepsilon\}$,
- The root node is labeled $S$,
- Every other node is labeled with some element of: $V - \Sigma$, and
- If $m$ is a nonleaf node labeled $X$ and the children of $m$ are labeled $x_1, x_2, \ldots, x_n$, then $R$ contains the rule $X \rightarrow x_1 x_2 \ldots x_n$.

Because parse trees matter, it makes sense, given a grammar $G$, to distinguish between:

- $G$'s ***weak generative capacity***, defined to be the set of strings, $L(G)$, that $G$ generates, and
- $G$'s ***strong generative capacity***, defined to be the set of parse trees that $G$ generates.

## 11.7 Ambiguity

Sometimes a grammar may produce more than one parse tree for some (or all) of the strings it generates. When this happens, we say that the grammar is ambiguous. More precisely, a grammar $G$ is *ambiguous* iff there is at least one string in L( G) for which $G$ produces more than one parse tree.

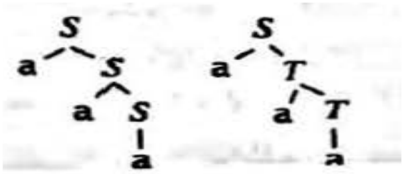
**Procedure to show that a grammar is ambiguous:**

- For the given grammar, select a string.
- Obtain the string by applying LMD(or RMD) twice.
- Construct parse trees for both the derivations.
- Compare the parse trees and conclude that it is ambiguous if they are different.

***Note: refer class notes for the problems on ambiguity.***

### 3.7.1 Regular Expressions and Grammars Can Be Ambiguous.

| **Regular Expression** | **Regular Grammar** |
|---|---|
| $(a \cup b)^*a\,(a \cup b)^*$ | $S \rightarrow a$ |
| | $S \rightarrow bS$ |
| choose a from $(a \cup b)$ | $S \rightarrow aS$ |
| choose a from $(a \cup b)$ | $S \rightarrow aT$ |
| choose a | $T \rightarrow a$ |
| | $T \rightarrow b$ |
| choose a | $T \rightarrow aT$ |
| choose a from $(a \cup b)$ | $T \rightarrow bT$ |
| choose a from $(a \cup b)$ | |




### 3.7.2 Inherent ambiguity

In many cases, when confronted with an ambiguous grammar *G,* it is possible to construct a new grammar G' that generates *L(* G) and that has less (or no) ambiguity. Unfortunately, it is not always possible to do this. There exist context-free languages for which no unambiguous grammar exists. We call such languages ***inherently ambiguous.***

Ex 1: Write inherently ambiguous grammar for L = $\{a^n b^n c^m : n, m \geq 0\} \cup \{a^n b^m c^m : n, m \geq 0\}$.

$S \rightarrow S_1 \mid S_2$

$S_1 \rightarrow S_1 c \mid A$ /* Generate all strings in $\{a^n b^n c^m\}$.

$A \rightarrow aAb \mid \varepsilon$

$S_2 \rightarrow aS_2 \mid B$ /* Generate all strings in $\{a^n b^m c^m\}$.

$B \rightarrow bBc \mid \varepsilon$

Consider any string of the form $a^n b^n c^n$, (ex: aabbcc), get 2 derivations and different parse trees.

*L* is inherently ambiguous.

Ex 2: Write inherently ambiguous grammar for L = $\{a^n b^n c^m d^m : n, m \geq 1\} \cup \{a^n b^m c^m d^n : n, m \geq 1\}$.

S→ AB | C

A→ aAb |ab

B→ cBd |cd

C→ aCd |aDd

D→ bDc |bc

### 3.7.3 Techniques for reducing ambiguity

Three grammar structures that often lead to ambiguity:

l. *s.* rules like S→ ε

2. Rules like *S*→ *SS* or *E*→ *E* + *E.* In other words recursive rules whose right· hand sides are symmetric and contain at least two copies of the nonterminal on the left-hand side.

3. Rule sets that lead to ambiguous attachment of optional postfixes.

**1. Eliminating ε-rules:**

**Nullable variable**: A variable *X* is *nullable* iff either:

(1) there is a rule $X \rightarrow \varepsilon$, or

(2) there is a rule $X \rightarrow PQR\ldots$ and *P*, *Q*, *R*, …
are all nullable.

So compute *N*, the set of nullable variables, as follows:

1. Set *N* to the set of variables that satisfy (1).
2. Until an entire pass is made without adding anything to *N* do
   Evaluate all other variables with respect to (2).

If any variable satisfies (2) and is not in *N*, insert it.

**Modifiable variable**: a rule is *modifiable* iff it is of the form:

$P \rightarrow \alpha Q \beta$, for some nullable $Q$.

**A General Technique for Getting Rid of ε-Rules :**

**removeEps(G: CFG) =**

1. **Let G′ = G.**
2. **Find the set N of nullable variables in G′.**
3. **Repeat until G′ contains no modifiable rules that haven't been processed:**

   **Given the rule $P \rightarrow \alpha Q \beta$, where $Q \in N$, add the rule $P \rightarrow \alpha\beta$, if it is not already present and if $\alpha\beta \neq \varepsilon$ and if $P \neq \alpha\beta$.**
4. **Delete from G′ all rules of the form $X \rightarrow \varepsilon$.**
5. **Return G′.**

**Therefore, L(G′) = L(G) – {ε}.**

**Example:** S→ aTa

*T→ ABC*

A→ aA| C

*B→ Bb* | C

C→ c | ε}.

On input *G, removeEps* behaves as follows: Step 2 finds the set *N* of nullable variables by initially setting *N* to { C}. On its first pass through step 2.2 it adds *A* and *B* to N. On the next pass. it adds *T* (since now *A, B.* and Care all in N). On the next pass, no new elements are found, so step 2 halts with *N* = {C. *A, B,* T}. Step 3 adds the following new rules to G':

*S→* aa /* Since T is nullable.

*T→ BC* /* Since *A* is nullable.

*T-→ AC* /* Since *D* is nullable.

*T→ AB* /* Since C is nullable.

*T →* C /* From *T→ BC,* since *B* is nullable. Or from *T→ AC.*

*T→ B* /* From *T→ BC.* since C is nullable. Or from *T→ AB.*

*T→ A* /* FromT→ *AC.* since C is nullable. Or from *T→ AB*

*A →* a /* Since *A* is nullable.

*B →* b /* Since *D* is nullable.

C→ c

Finally, step 4 deletes the rule C→ε.

***What If ε ∈ L?***

Sometimes L(G) contains ε and it is important to retain it. To handle this case, we present the following algorithm which constructs a new grammar *G",* such that *L* ( G") = *L* (G). If L( G) contains ε, then G" will contain a single ε-rule that can be thought of as being "quarantined".Its sole job is to generate the string ε. It can have no interaction with the other rules of the grammar.

***atmostoneEps(G: cfg) =***

***1. G″= removeEps(G).***

***2. If $S_G$ is nullable then /* i. e., $\varepsilon \in L(G)$***

***2.1 Create in G″a new start symbol S*.***

***2.2 Add to $R_{G''}$ the two rules:***

$$S^* \rightarrow \varepsilon$$

$$S^* \rightarrow S_G.$$

***3. Return G″.***

*Ex: Remove ε- rule from balanced parentheses grammar S → SS | (S) |ε*

$S^* \rightarrow \varepsilon$

$S^* \rightarrow S$

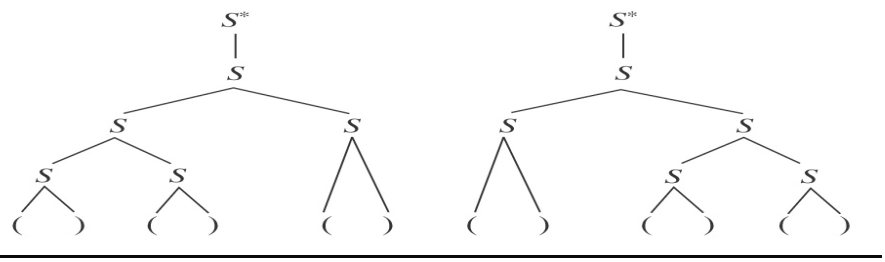
$S \rightarrow SS$

$S \rightarrow (S)$

$S \rightarrow ()$

**2. Eliminating symmetric rules:**

The new grammar that we just built for Bal is better than our original one. But it is still ambiguous. The string()()() has two parses shown in Figure. The problem now is the rule *S→ SS,* which must be applied *n* - 1 times to generate a sequence of *n* balanced parentheses substrings. But, at each time after the first, there is a choice of which existing *S* to split.




The solution to this problem is to rewrite the grammar so that there is no longer a choice. We replace the rule *S*--+ *SS* with one of the following rules:

$S \rightarrow SS_1$ /* *force branching to the left*

$S \rightarrow S_1S$ /* *force branching to the right*

Then we add the rules→ S1 and replace the rules *S→ (S)* and *S→* ( )with the rules *S1 →* (S) and S→ ( ).What we have done is to change the grammar so that branching can occur only in one direction. Every *S* that is generated can branch, but no S1 can. When all the branching has happened, *S* rewrites to S1 and the rest of the derivation can occur.

So one unambiguous grammar for Bals *G* = { {S, ), (}, {), (}, *R,* S), where:

$S^* \rightarrow \varepsilon$ $S \rightarrow SS_1$

$S^* \rightarrow S$ $S \rightarrow S_1$

$S_1 \rightarrow (S)$

$S_1 \rightarrow ( )$

There exists single parse tree for the string ( )( )( )

**3. Ambiguous Attachment**

Third source of ambiguity that we will consider arises when constructs with optional fragments are nested. Probably the most often described instance of this kind of ambiguity is known as the *dangling else problem.* Suppose that we define a programming language with an if statement that can have either of the following forms:

<stmt> ::= if <cond> then <stmt>

<stmt> ::= if <cond> then <stmt> e1se <stmt>

In other words. the e1se clause is optional. Then the following statement with just a single e1se clause has two parses:

if *cond1* then if *cond2* then *stmt1* e1se *smt2*

In the first parse, the single e1se clause goes with the first if. (So it attaches high in the parse tree.) In the second parse, the single e1se clause goes with the second if. (In this case it attaches lower in the parse tree.)

Ex: dangling else problem in Java:

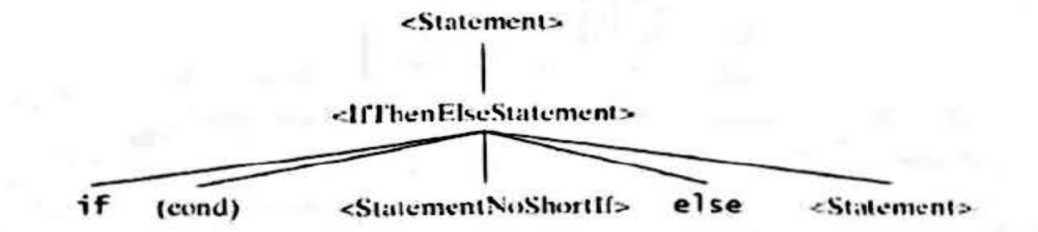
<Statement> ::= <IfThenStatement> | <IfThenElseStatement> |
<IfThenElseStatementNoShortIf>

<StatementNoShortIf> ::= <block> |
<IfThenElseStatementNoShortIf> | …

<IfThenStatement> ::= if ( <Expression> ) <Statement>

<IfThenElseStatement> ::= if ( <Expression> )
<StatementNoShortIf> else <Statement>

<IfThenElseStatementNoShortIf> ::=
if ( <Expression> ) <StatementNoShortIf>
else <StatementNoShortIf>




***<u>Note: Refer class notes for the problems on elimination of ε-rules.</u>***

## 3.8 Normal forms

Let *C* be any set of data objects. For example, *C* might be the set of context-free grammars. Or it could be the set of syntactically valid logical expressions or a set of database queries. We will say that a set *F* is a *normal form* for C iff it possesses the following two properties:

• For every element *c* of C, except possibly a finite set of special cases, there exists some element *f* of *F* such that *f* is equivalent to *c* with respect to some set of tasks.

• *F* is simpler than the original form in which the elements of *C* are written. By "simpler" we mean that at least some tasks are easier to perform on elements *of F* than they would be on elements of C.

**3.8.1 normal forms for grammars:**

We will define the following two useful normal forms for context-free grammars:

• *Chomsky Normal Form:* In a Chomsky normal form grammar $G = (V, \sum R, S)$,vall rules have one of the following two forms:

• $X \rightarrow a$, where a €∑, or

• $X \rightarrow BC$, where $B$ and Care elements of $V$ -∑

• *Greibuch Normal form:* In a Grcibach normal form grammar G = $(V, \sum R, S)$, all rules have the following form:

• X→ αβ(where α€∑ and β € ( V-∑) *

**3.8.2 Converting to a normal form:**

Algorithms to convert grammars into normal forms generally begin with n grammar G and then operate in a series of steps as follows:

1. Apply some transformation to G to get rid of undesirable property 1. Show that the language generated by $G$ is unchanged.
2. Apply another transformation to $G$ get rid of undesirable property 2. Show that the language generated by G is unchanged *and* that undesirable property 1 has not been reintroduced.
3. Continue until the grammar is in the desired form.

**THEOREM 11.3 Rule Substitution**

**Theorem:** Let $G = (V, \Sigma, R, S)$ be a context-free grammar that contains a rule $r$ of the form $X \rightarrow \alpha Y \beta$, where $\alpha$ and $\beta$ are elements of $V^*$ and $Y \in (V - \Sigma)$. Let $Y \rightarrow \gamma_1 | \gamma_2 | \ldots | \gamma_n$ be all of $G$'s rules whose left-hand side is $Y$. And let $G'$ be the result of removing from $R$ the rule $r$ and replacing it by the rules $X \rightarrow \alpha\gamma_1\beta, X \rightarrow \alpha\gamma_2\beta, \ldots, X \rightarrow \alpha\gamma_n\beta$. Then $L(G') = L(G)$.

**Proof:** We first show that every string in $L(G)$ is also in $L(G')$: Suppose that $w$ is in $L(G)$. If $G$ can derive $w$ without using rule $r$, then $G'$ can do so in exactly the same way. If $G$ can derive $w$ using rule $r$, then one of its derivations has the following form, for some value of $k$ between 1 and $n$:

$$S \Rightarrow \ldots \Rightarrow \delta X \phi \Rightarrow \delta \alpha Y \beta \phi \Rightarrow \delta \alpha \gamma_k \beta \phi \Rightarrow \ldots \Rightarrow w.$$

Then $G'$ can derive $w$ with the derivation:

$$S \Rightarrow \ldots \Rightarrow \delta X \phi \Rightarrow \quad \delta \alpha \gamma_k \beta \phi \Rightarrow \ldots \Rightarrow w.$$

Next we show that only strings in $L(G)$ can be in $L(G')$. This must be so because the action of every new rule $X \rightarrow \alpha\gamma_k\beta$ could have been performed in $G$ by applying the rule $X \rightarrow \alpha Y \beta$ and then the rule $Y \rightarrow \gamma_k$.

**3.8.3 Converting to Chomsky Normal Form**

There exists a straightforward four-step algorithm that converts a grammar $G = ( V. \sum , R. S)$ into a new grammar $Gc$ such that $Gc$ is in Chomsky normal form and $L$ (Gc) = $L$ (G)- {ε}. Define:

***converttotoChomsky(* G: CFG) =**

1. **Let *Gc* be the result of removing from *G* all ε-rules, using the algorithm remo*veEps( )***
2. **Let *Gc* be the result of removing from *Gc* all unit productions (rules of the form *A* →B) using the algorithm *removeUnits* defined below. It is important to remove ε-rules first, before applying *removeUnits( ).* Once this step has been completed, all rules whose right-hand sides have length 1 are in Chomsky normal form (i.e . they are composed of a single terminal symbol).**
3. **Let *Gc* be the result of removing from *Gc* all rules whose right-hand sides have length greater than 1 and include a terminal (e.g : *A* → *aB* or *A* → BaC). This step is simple and can be performed by the algorithm *remove Mixed* given below. Once this step has been completed all rules whose right-hand sides have length 1 or 2 are in Chomsky normal form.**
4. **Let *Gc* be the result of removing from *Gc* all rules whose right-hand sides have length greater than 2 (e.g : *A* → *BCD£).* This step too is simple. It can be performed by the algorithm *remove Long* given below.**
5. **Return *Gc*.**

A ***unit production*** is a rule whose right-hand side consists of a single nonterminal symbol. The job of *remove Units* is to remove all unit productions and to replace them by a set of other rules that accomplish the job previously done by the unit productions. So, for example, suppose that we start with a grammar *G* that contains the following rules:

*S* → *XY*
*X* → *A*
*A* → *B* | *a*
*B* → b

Once we get rid of unit productions, it will no longer be possible for *X* to become *A* (and then B) and thus to go on to generate a or b. So *X* will need the ability to go directly to a and b, without any intermediate steps. We can define *removeUnits( )* as follows:

***removeUnits(G:* CFG) =**

**1. Let G' =G.**
**2. Until no unit productions remaining G' do:**
 **2.1. Choose some unit production *X* → *Y*.**
 **2.2. Remove it from G'.**
 **2.3. Consider only rules that still remain in G'. For every rule *Y* →*β* where *β* € *V**, do:**
 **Add to *G'* the rule *X* →*β* unless that is a rule that has already been removed once.**
**3. Return G'.**

EXAMPLE : Removing Unit Productions:

S→ XY
*X* →*A*
*A* →*B* | a
B→b
*Y* →*T*
*T* →*Y* | c

The order in which *removeUnits* chooses unit productions to remove doesn't matter. We'll consider one order it could choose:

Remove $X \rightarrow A$. Since $A \rightarrow B$ | a, add $X \rightarrow B$ | a.

Remove $X \rightarrow B$. Add $X \rightarrow$ b.

Remove $Y \rightarrow T$. Add $Y \rightarrow Y$ | c. Notice that we've added $Y \rightarrow Y$, which is useless, but it will be removed later;

Remove Y $\rightarrow Y$. Consider adding Y$\rightarrow T$, but don't since it has previously been removed

Remove $A \rightarrow B$. Add $A \rightarrow$b.

Remove $T \rightarrow Y$. Add $T \rightarrow$ c, but with no effect since it was already present.

At this point, the rules of *G* are:

$S \rightarrow XY$

A$\rightarrow$a | b

B$\rightarrow$ b

$T \rightarrow c$

X$\rightarrow$a | b

$Y \rightarrow c$

No unit productions remain, so *remove Units* halts.

***removeMixed(G):***

***removeMixed* (G: CFG) =**

**1. LetG' =G.**

**2. Create a new nonterminal $T_a$, for each terminal *a* in $\sum$.**

**3. Modify each rule in G' whose right-hand side has length greater than 1 and that contains a terminal symbol by substituting $T_a$, for each occurrence of the terminal *a*.**

**4. Add to G', for each $T_a$, the rule $T \rightarrow a$.**

**5. Return *G'*.**

EXAMPLE: Removing Mixed Productions from

A$\rightarrow$a

$A \rightarrow aB$

$A \rightarrow BaC$

$A \rightarrow BbC$

The result of applying *remove Mixed* to the above grammar produces:

A$\rightarrow$a

$A \rightarrow T_aB$

$A \rightarrow BT_aC$

$A \rightarrow BT_bC$

$T_a \rightarrow a$

$T_b \rightarrow$ b

*removeLong* (*G*: CFG) =

1. Let $G' = G$.
2. For each $G'$ rule $r^k$ of the form $A \rightarrow N_1N_2N_3N_4 \ldots N_n$, $n > 2$, create new non-terminals $M^k_2, M^k_3, \ldots M^k_{n-1}$.
3. In $G'$, replace $r^k$ with the rule $A \rightarrow N_1M^k_2$.
4. To $G'$, add the rules $M^k_2 \rightarrow N_2M^k_3$, $M^k_3 \rightarrow N_3M^k_4, \ldots M^k_{n-1} \rightarrow N_{n-1}N_n$.
5. Return $G'$.

EXAMPLE: Removing Rules with Long Right-hand Sides

The result of applying *remove Long* to the single rule grammar *A →BCDEF* is the grammar with rules:

A→BM2

*M2 →CM3*

*M3 →DM4*

M4→EF

**Problem: convert the following grammar into CNF.**

S→aACa

*A →B* | a

*B →C* | c

C→Cc | ε

**Step 1: Apply *removeEps( ), N={C, A, B}***

Grammar becomes

S→.aAca | aAa | aCa | aa

A→B | a

*B→ C* | c

C→cC | c

**Step 2: Next we apply *remove Units:***

Remove *A →B.* Add *A→* C | c.

Remove *B →*C. Add *B →*cC (and *B →*c, but it was already there).

Remove A→ C. Add A→cC (and A→ c, but it was already there).

So *remove Units* returns the rule set:

S→aAca | aAa| aca | aa

A→a | c | cC

*B →c* | cC

C → cC | c

**Step 3: Next we apply *removeMixed,* which returns the rule set:**

$S \rightarrow T_aACT_a, \mid T_a,AT_a, \mid T_aCT_a \mid T_aT_a$

A→ a | c| $T_cC$

$S \rightarrow c \mid T_cC$

$C \rightarrow T_c \mid c$

$T_a \rightarrow a$

$T_c \rightarrow c$

**Step 4: Finally, we apply *remove Long,* which returns the rule set:**

$S \rightarrow T_aS_1$ $S \rightarrow T_aS_3$ $S \rightarrow T_aS_4$ $S \rightarrow T_aT_a$

$S_1 \rightarrow AS_2$ $S_3 \rightarrow AT_a$ $S_4 \rightarrow CT_a$

$S_2 \rightarrow CT_a$

$A \rightarrow a \mid c \mid T_cC$

$B \rightarrow c \mid T_cC$

$C \rightarrow T_cC \mid c$

$T_a \rightarrow a$

$T_c \rightarrow c$

***<u>Note: Refer class notes for problems on CNF.</u>***

**Proving that a Grammar is Unambiguous**

While it is undecidable in *general* whether a grammar is ambiguous or unambiguous. It may be possible to prove that a *particular* grammar is either ambiguous or unambiguous.

A grammar *G* canbhe shown to be ambiguous by exhibiting single string for which *G* produces two parse trees.

EXAMPLE: The Final Balanced Parens Grammar is Unambiguous

We return to the final grammar G:

$S^* \rightarrow \varepsilon$ (1)

$S^* \rightarrow S$ (2)

$S \rightarrow SS_1$ (3)

$S \rightarrow S_1$ (4)

$S_1 \rightarrow (S)$ (5)

$S_1 \rightarrow ( )$ (6)

We prove that $G$ is unambiguous. Given the leftmost derivation of any string $w$ in $L(G)$, there is, at each step of the derivation, a unique symbol, which we'll call $X$, that is the leftmost nonterminal in the working string. Whatever $X$ is, it must be expanded by the next rule application, so the only rules that may be applied next are those with $X$ on the left-hand side. There are three nonterminals in $G$. We show, for each of them, that the rules that expand them never compete in the leftmost derivation of a particular string $w$. We do the two easy cases first:

- $S^*$: The only place that $S^*$ may occur in a derivation is at the beginning. If $w = \varepsilon$, then rule (1) is the only one that can be applied. If $w \neq \varepsilon$, then rule (2) is the only one that can be applied.
- $S_1$: If the next two characters to be derived are (), $S_1$ must expand by rule (6). Otherwise, it must expand by rule (5).

In order discuss $S$, we first define, for any matched set of parentheses $m$, the ***siblings*** of $m$ to be the smallest set that includes any matched set $p$ adjacent, on the right, to $m$ and all of $p$'s siblings. So, for example, consider the string:

$$\underbrace{\left(\underset{1}{()}\,\underset{2}{()}\right)}_{5}\,\underset{3}{()}\,\underset{4}{()}$$

The set () labeled 1 has a single sibling, 2. The set (()()) labeled 5 has two siblings, 3 and 4. Now we can consider $S$. We observe that:

- $S$ must generate a string in Bal and so it must generate a matched set, possibly with siblings.
- So the first terminal character in any string that $S$ generates is (. Call the string that starts with that ( and ends with the ) that matches it, $s$.
- The only thing that $S_1$ can generate is a single matched set of parentheses that has no siblings.
- Let $n$ be the number of siblings of $s$. In order to generate those siblings, $S$ must expand by rule (3) exactly $n$ times (producing $n$ copies of $S_1$) before it expands by rule (4) to produce a single $S_1$, which will produce $s$. So, at every step in a derivation, let $p$ be the number of occurrences of $S_1$ to the right of $S$. If $p < n$, $S$ must expand by rule (3). If $p = n$, $S$ must expand by rule (4).

**********