MODULE - II

SYNTAX ANALYSIS

Review of Context-Free Grammars – Derivation trees and Parse Trees, Ambiguity.

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
AMAL JYOTHI COLLEGE OF ENGINEERING
KANJIRAPPALLY

SYNTAX ANALYSIS

Syntax analysis or parsing is the second phase of a compiler.

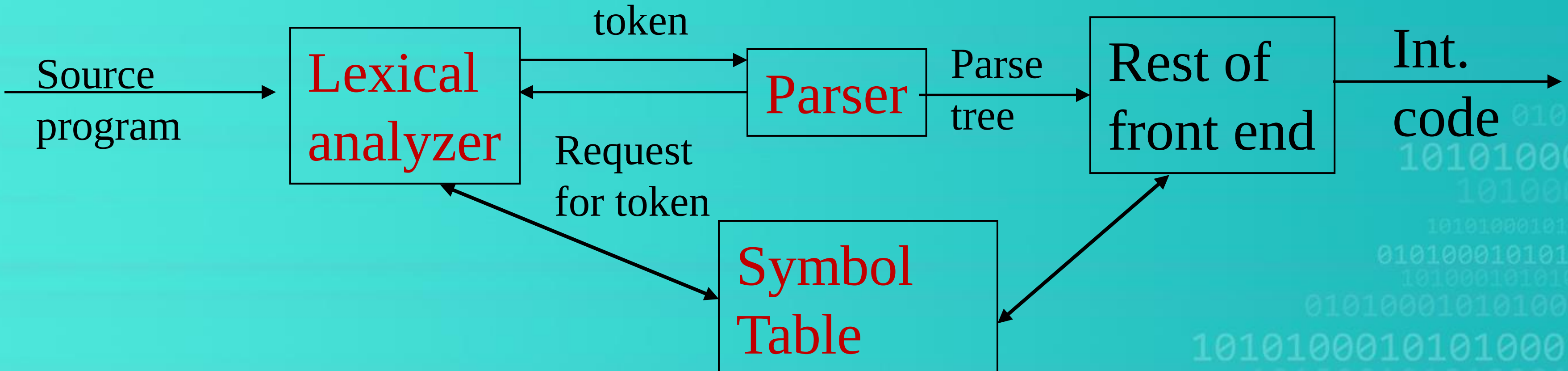
A lexical analyzer can identify tokens with the help of regular expressions and pattern rules.

But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions.

Regular expressions cannot check balancing tokens, such as parenthesis

SYNTAX ANALYSIS

- Syntax analysis is done by the parser.
  - Detects whether the program is written following the grammar rules and reports syntax errors.
  - Produces a parse tree from which intermediate code can be generate d.



SYNTAX ERROR HANDLING

- Good compiler – helps in identifying and locating errors
- Errors maybe:
  - Lexical : misspelling of identifiers, keywords
  - Syntactic: expression with unbalanced parenthesis
  - Semantic: Operator applied to incompatible operands
  - Logical: Infinitely recursive calls

- Much of the error detection & recovery is centered around syntax analysis phase
- Its because, stream of tokens from LA disobeys grammatical rules defining programming language

Use of Grammars

- Syntax of Programming language constructs can be described by CFG

- Grammar - Advantages
  - Precise and easy to understand syntactic specification
  - Automatic construction of efficient parsers
  - Imparts a structure to the programming language
  - Language evolution is easier.

Role of the Parser

- Parser obtains a string of tokens from the lexical analyzer
- Verifies that the string can be generated by the grammar of sou rce language
- Reports Syntax errors / recovers from common errors

ERROR RECOVERY STRATEGIES

- **Panic mode**
  - Discards input symbols until tokens in the synchronizing set are encountered
  - Synchronizing set (delimiters) must be chosen carefully(; or end)
  - Skips input
  - Simple-no infinite loop
  - Adequate when multiple errors in same statement is rare.
- **Phrase level**
  - Local correction, replaces prefix
  - Should not lead to infinite loops
  - First used with top down parsing
  - Difficult when actual error occurred before detection

ERROR RECOVERY STRATEGIES

- **Error productions**
  - Augment grammar with productions that generate erroneous constructs
  - If error production is used by the parser, can generate error diagnostics
- **Global correction**
  - Given x-incorrect input string and grammar G
  - Algorithm will find a parse tree for related string y by identifying minimal sequence of changes needed to transform x to y
  - Costly-time and space

Context Free Grammars :Concepts & Terminology

Context Free Grammars :Concepts & Terminology

Definition:

A Context Free Grammar, CFG, is described by T, NT, S, PR,

- **T: Terminals / tokens of the language**
- **NT: Non-terminals to denote sets of strings generated by the grammar in the language**
- **S: Start symbol, S∈NT, which defines all strings of the language**
- **PR: Production rules to indicate how T and NT are combined to generate valid strings of the language.**

PR: NT → (T | NT)*

CFG

- A context-free grammar consists of terminals, non-terminals, a start symbol, and productions.

Eg : stmt→ **if** expr **then** stmt **else** stmt

1. Terminals are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal”

   if, then, else are **keywords** in the above example.

2. Non-terminals are syntactic variables that denote sets of strings.

   The non-terminals define sets of strings that help to define the language generated by the grammar

   stmt and expr are **non-terminals**

3. In a grammar, one nonterminal is distinguished as the start symbol, and the set of strings it denotes is the language generated by the grammar.

4. The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each production consists of:

   a. A nonterminal called the head or left side of the production; this production defines some of the strings denoted by the head.

   b. The symbol → sometimes : : = has been used in place of the arrow.

   c. A body or right side consisting of zero or more terminals and non terminals.

Notational Conventions

- To avoid always having to state that "these are the terminals," "these a re the non-terminals," and so on, the following notational conventions for grammars will be used.
- These symbols are terminals:

a. Lowercase letters early in the alphabet, such as a, b, c.

b. Operator symbols such as +, *, and so on.

c. Punctuation symbols such as parentheses, comma, and so on.

d. The digits 0, 1, . . . , 9.

e. Boldface strings such as id or if, each of which represents a single terminal symbol.

Notational Conventions - Non-Terminals

- Uppercase letters early in the alphabet, such as *A*, *B*, *C*.
- The letter *S*, which, when it appears, is usually the start symbol.
- Lowercase, italic names such as *expr* or *stmt*.
- When discussing programming constructs, uppercase letters may be used to represent non-terminals for the constructs.
- For example, non-terminals for expressions, terms, and factors are often represented by E, T, and F, respectively.
- Uppercase letters late in the alphabet, such as X, Y, Z, represent grammar symbols; that is, either non-terminals or terminals.

Notational Conventions

- Lowercase letters late in the alphabet, chiefly u,v,..., z, represent (possibly empty) strings of terminal.
- Lowercase Greek letters, a, 0, 7 for example, represent (possibly empty) strings of grammar symbols.
- Thus, a generic production can be written as $A \rightarrow a$ where A is the head and a the body.
- A set of productions $A \rightarrow a1$ $A \rightarrow a2$ ... , $A \rightarrow ak$ with a common head.
- A (call them A-productions), may be written $A \rightarrow a1 \mid a2 \mid \ldots ak$
- Unless stated otherwise, the head of the first production is the start symbol

Grammar for simple Arithmetic Expressions

expression -> *expression* + *term*
expression -> *expression* - *term*
expression -> *term*
term -> *term* * *factor*
term -> *term* / *factor*
term -> *factor*
factor -> (*expression*)
factor -> *id*

In this grammar,

- The terminal symbols are id + - * / ( )
- The non-terminal symbols are expression, term and factor, and expression is the start symbol

Using these conventions, the grammar can be rewritten concisely as

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid id$

expression -> *expression* + *term*
expression -> *expression* - *term*
expression -> *term*
term -> *term* * *factor*
term -> *term* / *factor*
term -> *factor*
factor -> (*expression*)
factor -> *id*

The notational conventions tell us that E, T, and F are non-terminals, with E the start symbol. The remaining symbols are terminals.

Derivations

- The construction of a **parse tree** can be made precise by taking a derivational view, in which productions are treated as rewriting rules.
- Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions.
- This derivational view corresponds to the top-down construction of a parse tree.

Derivations

- Production is treated as rewriting rule in which the NT on left is replaced by string on the right side of production
- **EXAMPLE: $E \Rightarrow -E$ (the $\Rightarrow$ means "derives" in one step) using the production rule: $E \rightarrow -E$**
- **EXAMPLE: $E \Rightarrow E\ A\ E \Rightarrow E * E \Rightarrow E * (E)$**
- **DEFINITION: $\Rightarrow$ derives in one step**

  $\overset{+}{\Rightarrow}$ derives in one or more steps

  $\overset{*}{\Rightarrow}$ derives in zero or more steps

EXAMPLE

- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma$ is a production rule
- $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$, $\alpha_1 \stackrel{*}{\Rightarrow} \alpha_n$; $\alpha \stackrel{*}{\Rightarrow} \alpha$ for all α
- If $\alpha \stackrel{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$ then $\alpha \stackrel{*}{\Rightarrow} \gamma$

EXAMPLE

Consider the following grammar, with a single non-terminal *E*, which adds a production to the grammar of **Arithmetic Expressions.**

$$E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$$

- The production E → - E signifies that if E denotes an expression, then – E must also denote an expression.
- placement of a single E by - E will be described by writing E => -E which is read, "E derives - E."
- We can take a single E and repeatedly apply productions in any order to get a sequence of replacements.
- For example, **E => - E => - (E) => - (id)**

The string - (i d + id) is a sentence of grammar because there is a derivation

$E \rightarrow E+E \mid E^* E \mid (E) \mid -E \mid id$

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$

- The strings E, -E, -(E),... , - (id + id) are all sentential forms of this grammar.
- At each step in a derivation, there are two choices to be made.
- We need to choose which nonterminal to replace, and having made this choice, we must pick a production with that nonterminal as head.
- For example, the following alternative derivation of - (i d + id).

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$$

Grammar

list → list + digit | list –digit | digit

digit → 0 | 1 | … | 9

Derive the string **9-5+2** from the grammar

list	→	list + digit
list	→	list – digit + digit
list	→	digit – digit + digit
list	→	9 – digit + digit
list	→	9 – 5 + digit
list	→	**9 – 5 + 2**

Leftmost And Rightmost Derivation of a String

- Leftmost derivation − A leftmost derivation is obtained by applying production to the leftmost variable in each step.
- Rightmost derivation − A rightmost derivation is obtained by applying production to the rightmost variable in each step.

Derivation Order

1. $S \to AB$
2. $A \to aaA$
3. $A \to \lambda$
4. $B \to Bb$
5. $B \to \lambda$

Leftmost derivation:

$$S \overset{1}{\Rightarrow} AB \overset{2}{\Rightarrow} aaAB \overset{3}{\Rightarrow} aaB \overset{4}{\Rightarrow} aaBb \overset{5}{\Rightarrow} aab$$

Rightmost derivation:

$$S \overset{1}{\Rightarrow} AB \overset{4}{\Rightarrow} ABb \overset{5}{\Rightarrow} Ab \overset{2}{\Rightarrow} aaAb \overset{3}{\Rightarrow} aab$$

Example

Let any set of production rules in a CFG be

$X \rightarrow X+X \mid X*X \mid X \mid a$ over an alphabet {a}.

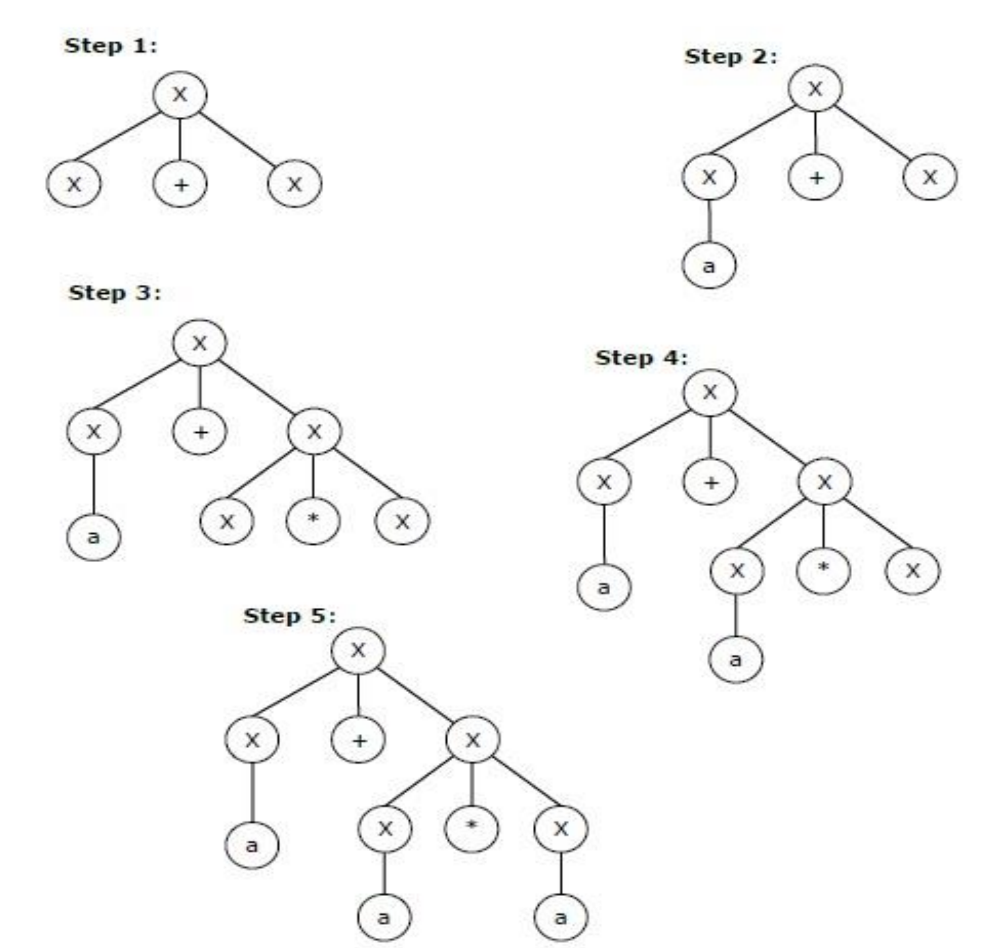
The **leftmost derivation** for the string **"a+a*a"**

$X \rightarrow X+X \rightarrow a+X \rightarrow a + X*X \rightarrow a+a*X \rightarrow a+a*a$

The **rightmost derivation** for the above string **"a+a*a"**

$X \rightarrow X*X \rightarrow X*a \rightarrow X+X*a \rightarrow X+a*a \rightarrow a+a*a$

The stepwise derivation of the above string **"a+a*a"**



X → X+X | X*X |X| a

The **leftmost derivation** for the string "

a+a*a"

X → X+X

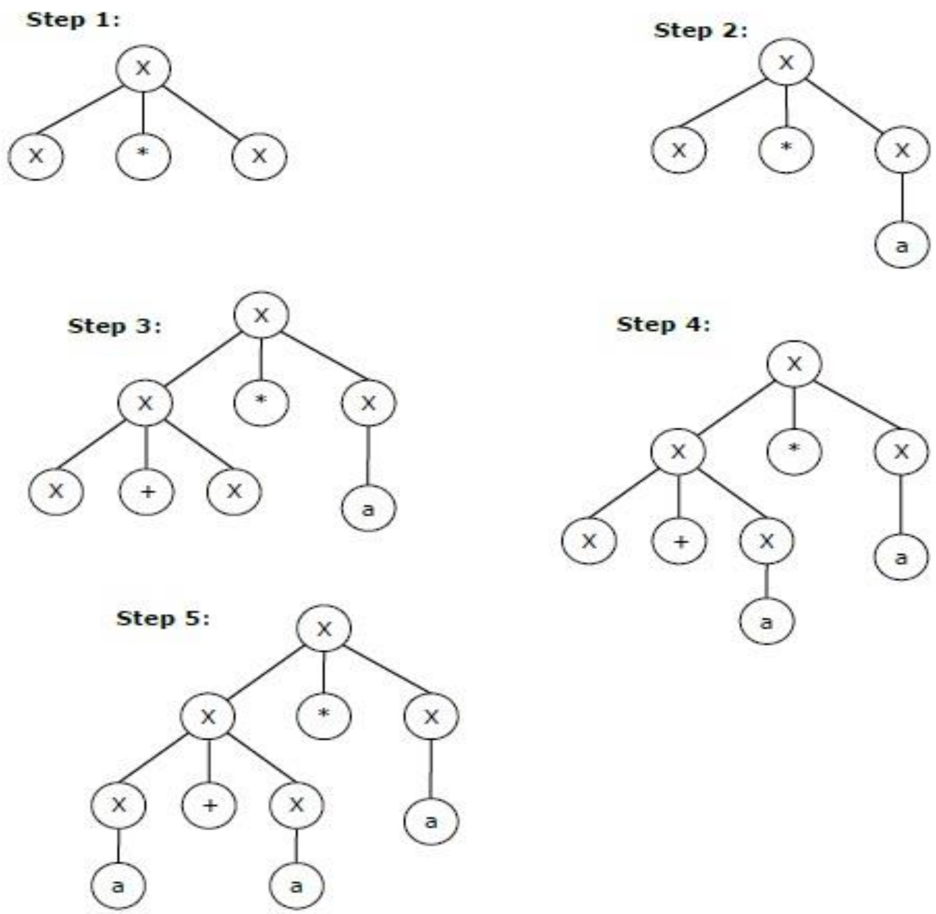
→ a+X

→ a + X*X

→ a+a*X

→ a+a*a

The stepwise derivation of the above string **"a+a*a"**



The **rightmost derivation** for the above string **"a+a*a"**

X → X*X

→ X*a

→ X+X*a

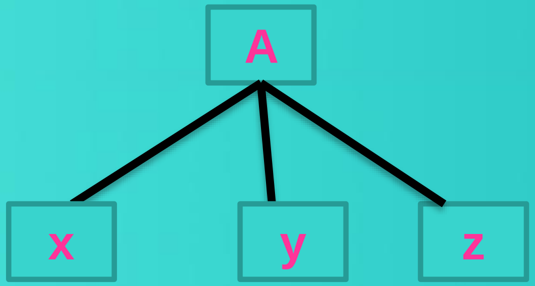
→ X+a*a

→ a+a*a

PARSE TREE

- Parse tree is a hierarchical structure which represents the derivation of the grammar to yield input strings.
- Simply it is the graphical representation of derivations.
- **Root node** of parse tree has the start symbol of the given grammar from where the derivation proceeds.
- Leaves of parse tree are labeled by non-terminals or terminals.
- Each interior node is labeled by some non terminals.

- If **A →xyz** is a production, then the parse tree will have A as interior node whose children are x, y and z from its left to right.

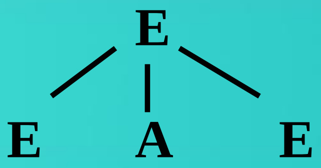


Yield of Parse Tree

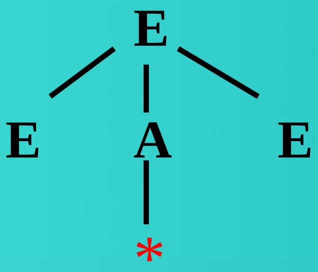
- The leaves of the parse tree are labeled by non-terminals or terminals and read from left to right, they constitute a sentential form, called the yield or frontier of the tree.

Derivations & Parse Tree

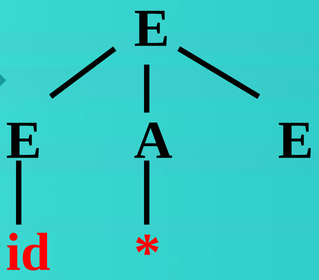
$E \Rightarrow E\ A\ E$ — Parse tree



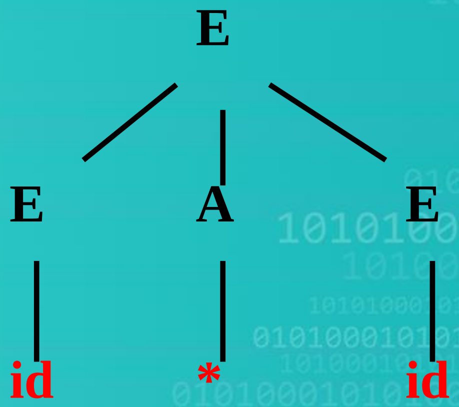
$\Rightarrow E * E$ — Parse tree



$\Rightarrow id * E$ — Parse tree



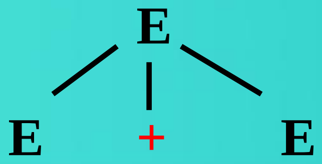
$\Rightarrow id * id$ — Parse tree



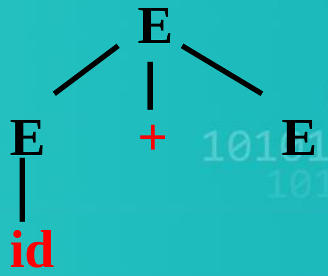
Parse Trees and Derivations

- **Consider the expression grammar:**
  - **$E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$**
- **Leftmost derivations of id + id * id**

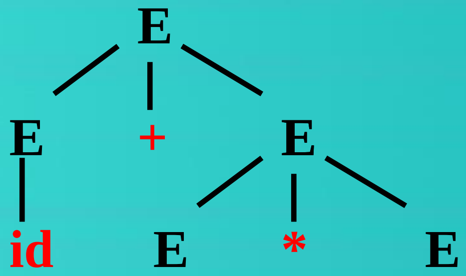
$E \Rightarrow E + E$



$E + E \Rightarrow id + E$

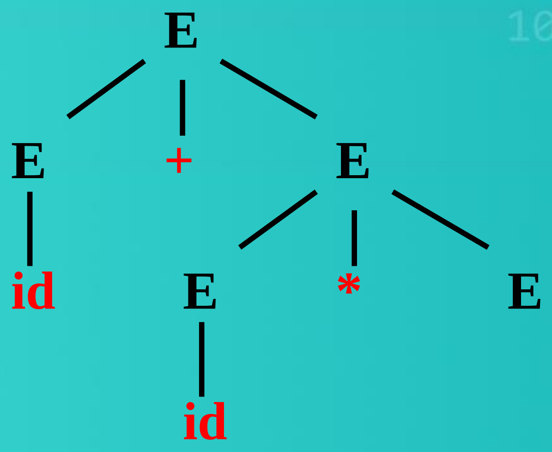


$id + E \Rightarrow id + E * E$

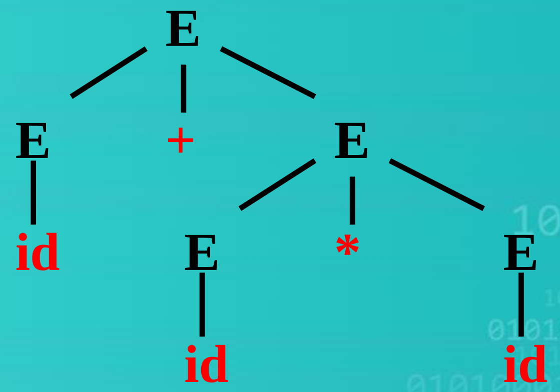


Parse Tree & Derivations – cont….

id + E * E ⇒ id + id * E



id + id * E ⇒ id + id * id

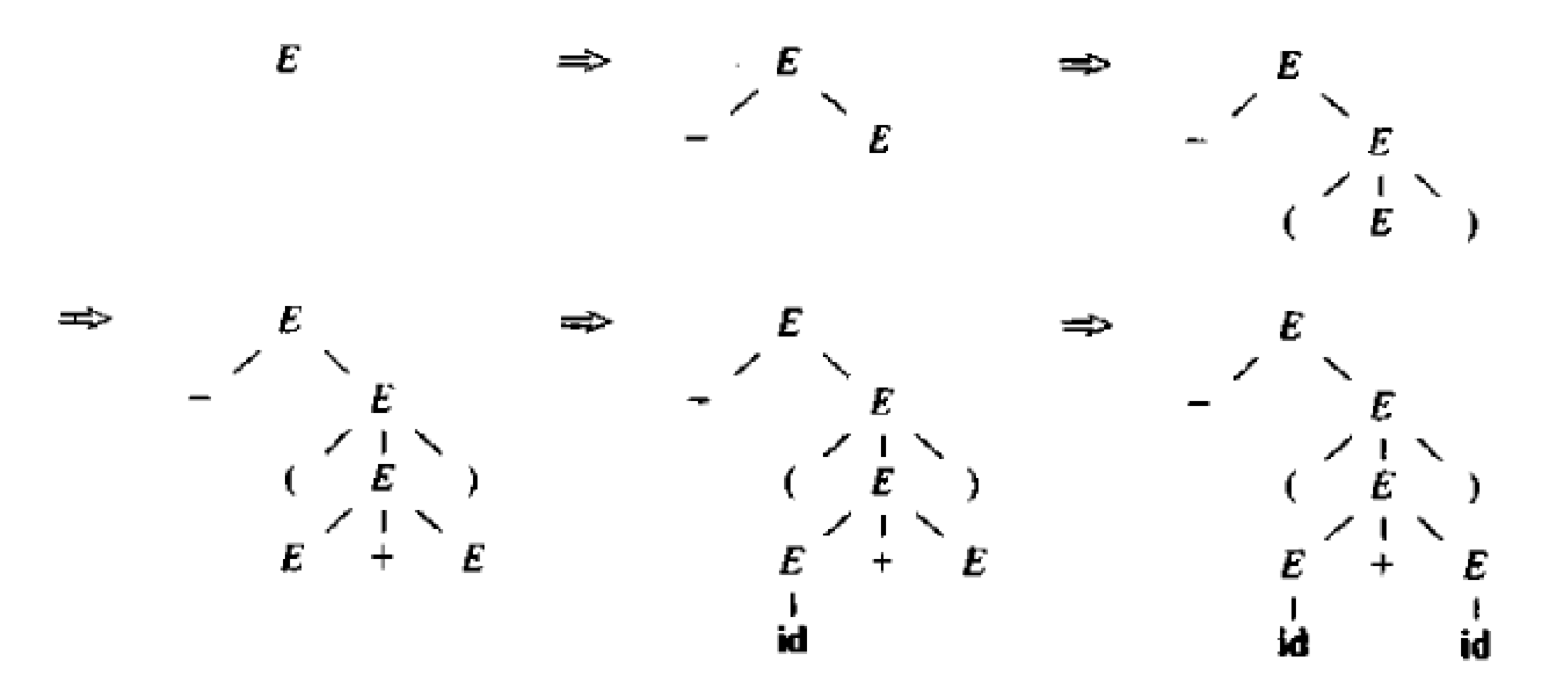


Draw a parse tree for –(id + id)

Grammar :

E→ E+E | E*E |(E) | -E | id

Parse tree for –(id + id)



Ambiguous Grammar

- An ambiguous grammar is one that produces **more than one** leftmost or more than one rightmost derivation for the same sentence.
- For most parsers, it is desirable that the **grammar be made unambiguous**, for if it is not, we cannot uniquely determine which parse tree to select for a sentence.
- In other cases, it is convenient to use carefully chosen ambiguous grammars, together with disambiguating rules that "throw away" undesirable parse trees, leaving only one tree for each sentence.

Alternative Parse Trees

- Consider the sentence id + id * id
- It has two leftmost derivations

Derivation 1

$$
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow id + E * E \\
&\Rightarrow id + id * E \\
&\Rightarrow id + id * id
\end{aligned}
$$

Derivation 2

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow id + E \\
&\Rightarrow id + E * E \\
&\Rightarrow id + id * E \\
&\Rightarrow id + id * id
\end{aligned}
$$

LEFT -Derivation 1

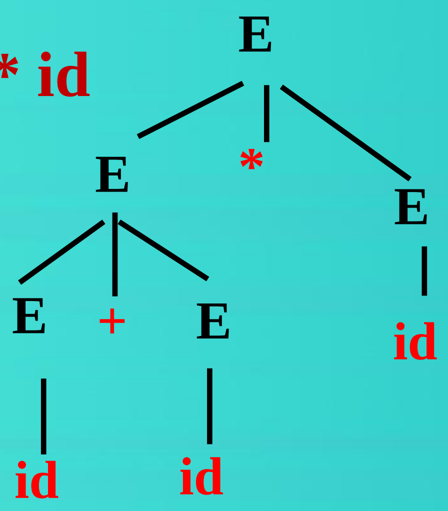
$E \Rightarrow E * E$

$\Rightarrow E + E * E$

$\Rightarrow id + E * E$

$\Rightarrow id + id * E$

$\Rightarrow id + id * id$



LEFT - Derivation 2

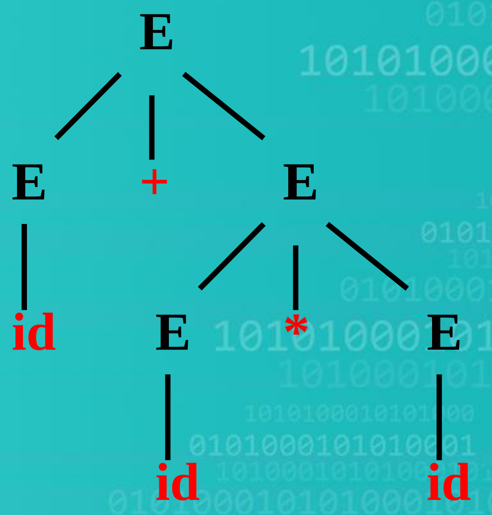
$E \Rightarrow E + E$

$\Rightarrow id + E$

$\Rightarrow id + E * E$

$\Rightarrow id + id * E$

$\Rightarrow id + id * id$



Resolving Problems: Ambiguous Grammars

Consider the following grammar segment:

stmt* → if *expr* then *stmt

| if *expr* then *stmt* else *stmt*

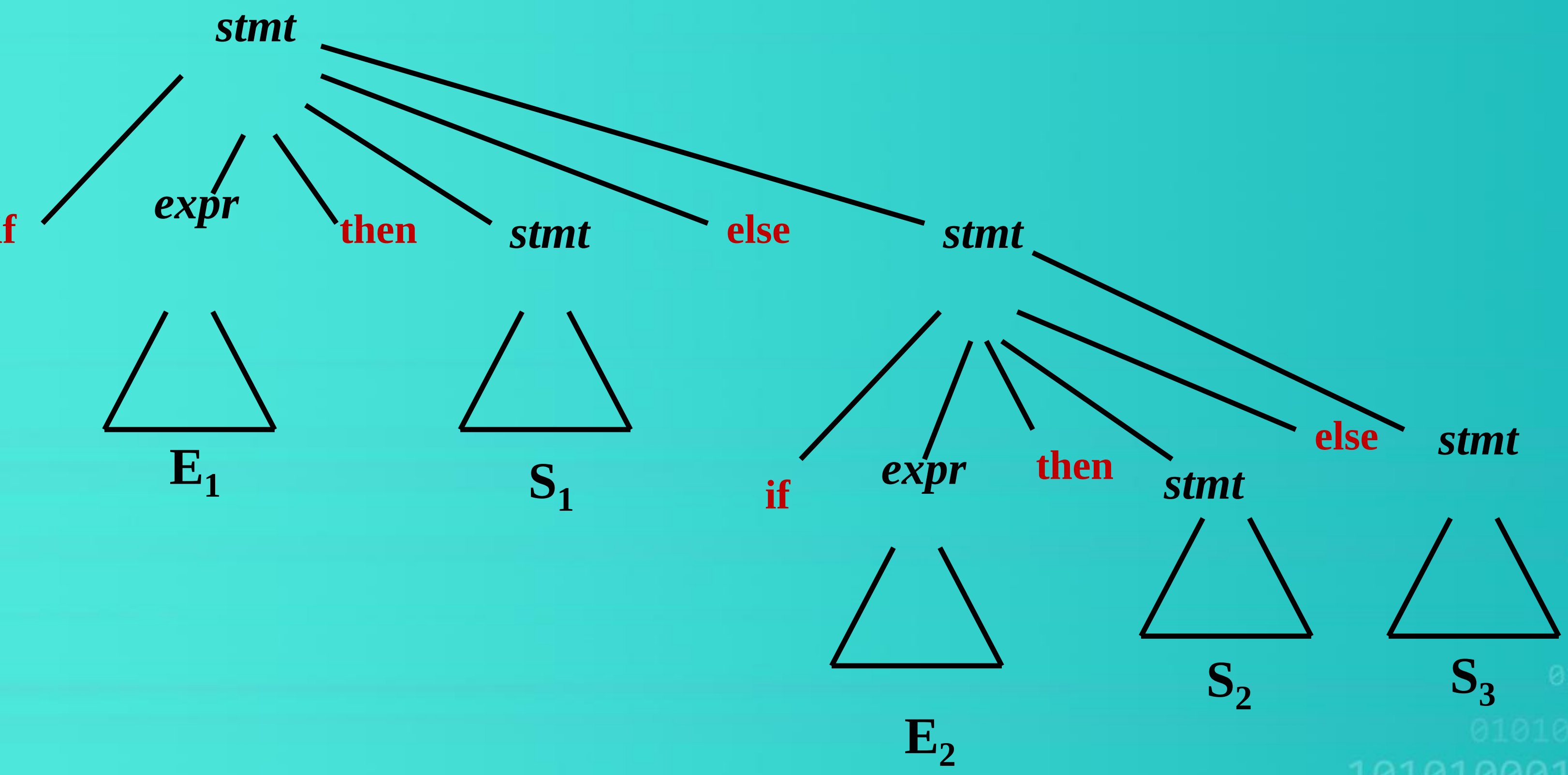
| other (any other statement)

Matching then and else

Eg: **if** E1 **then** S1 **else if** E2 **then** S2 **else** S3

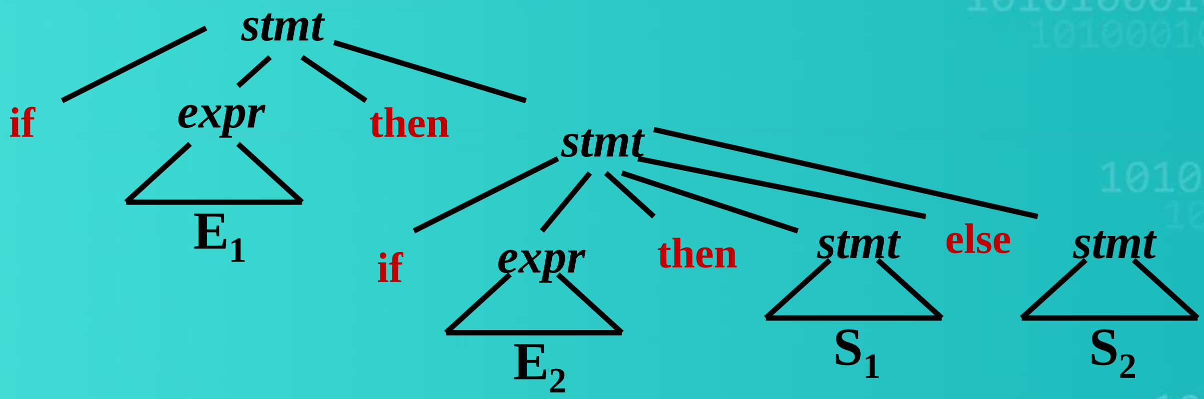
Let’s consider a simple parse tree:

Eg: **if** E1 **then** S1 **else if** E2 **then** S2 **else** S3

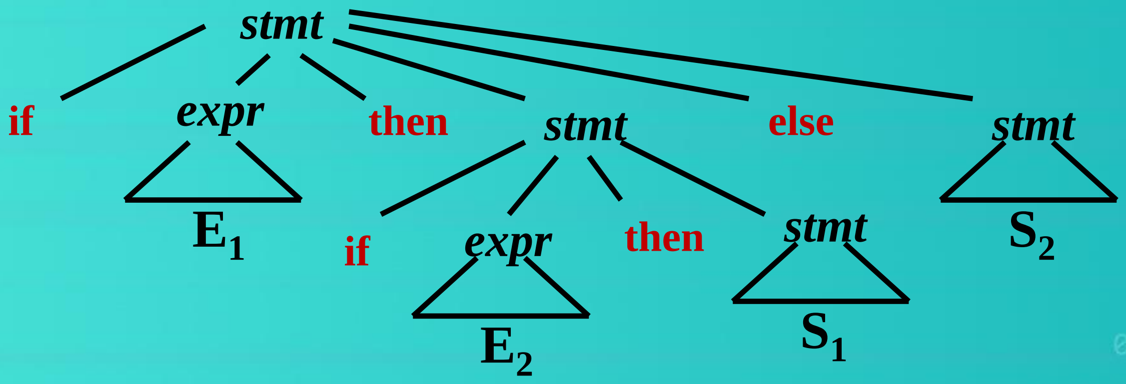


Parse Trees for Example

Form 1:



Form 2:



Eg: **if** E1 **then** S1 **else if** E2 **then** S2 **else** S3

Removing Ambiguity

Take Original Grammar:

stmt → **if** ***expr*** **then** ***stmt***

| **if** ***expr*** **then** ***stmt*** **else** ***stmt***

| **other (any other statement)**

Rule: Match each **else** with the closest previous unmatched **then**.

Revise to remove ambiguity: Statement between then and else must be matched

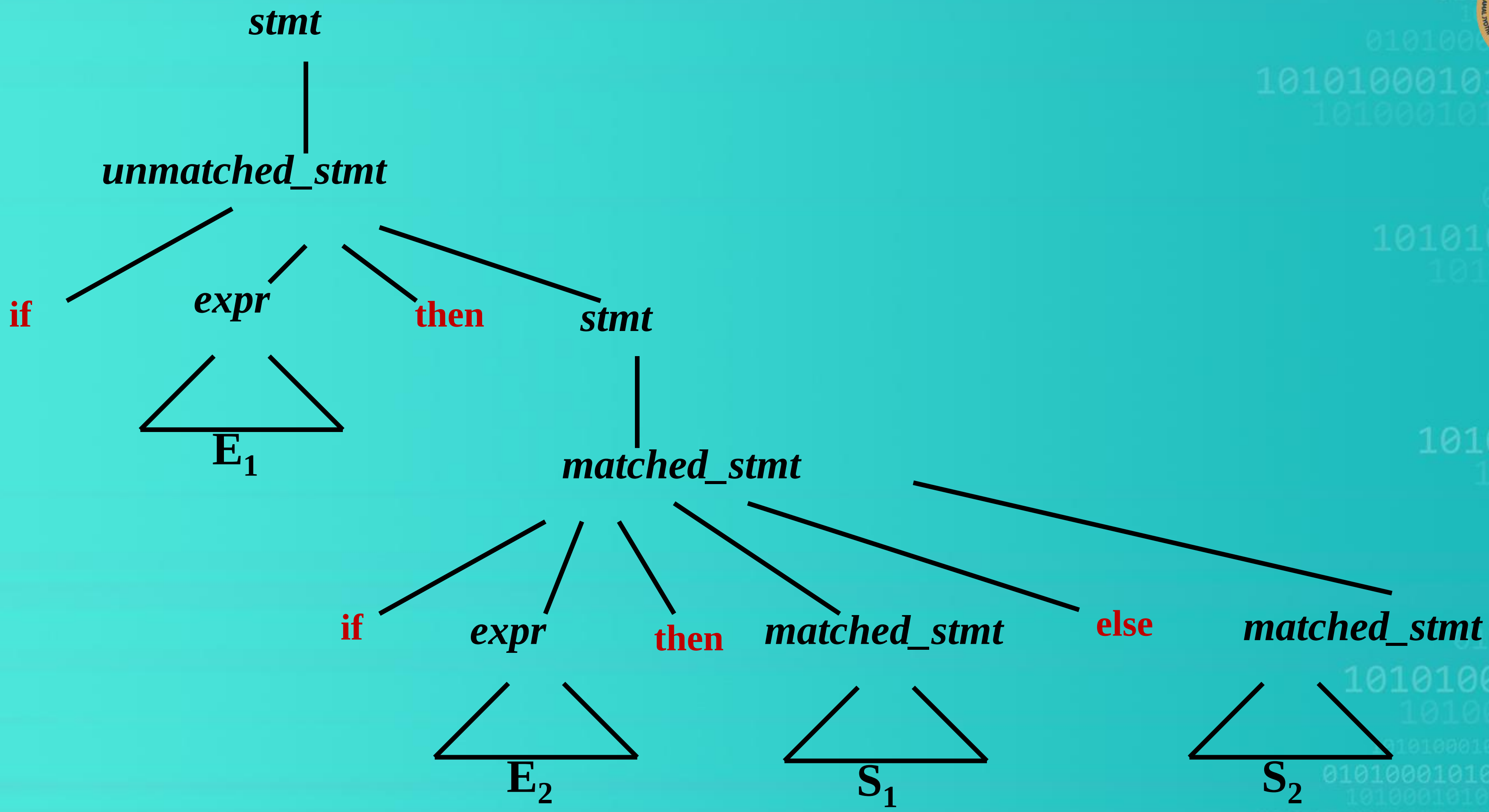
Removing Ambiguity

stmt → *matched_stmt* | *unmatched_stmt*

matched_stmt → **if** *expr* **then** *matched_stmt* **else** *matched_stmt* | **other**

unmatched_stmt → **if** *expr* **then** *stmt*

| **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*



Eg: **if** E1 **then** S1 **else if** E2 **then** S2 **else** S3

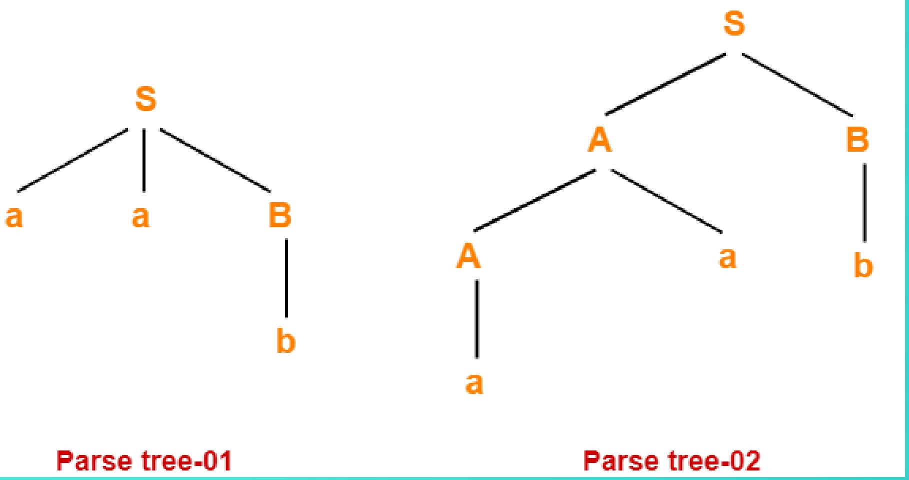
Check whether the given grammar is ambiguous or not-

$$S \rightarrow AB \mid aaB$$
$$A \rightarrow a \mid Aa$$
$$B \rightarrow b$$

Let us consider a string w generated by the given grammar

w = aab

Now, let us draw parse trees for this string w.



Since two different parse trees exist for string w, therefore the given grammar is ambiguous.

Solve

Consider the grammar-

S → A1B

A → 0A / ∈

B → 0B / 1B / ∈

For the string w = 00101, find-

- ✓ Leftmost derivation
- ✓ Rightmost derivation
- ✓ Parse Tree

And check whether the grammar is ambiguous or not

FOR MORE PROBLEMS VISIT :
https://www.gatevidyalay.com/grammar-ambiguity-ambiguous-grammar/

1. Leftmost Derivation-

$S \to A1B$

$\to 0A1B$ (Using $A \to 0A$)

$\to 00A1B$ (Using $A \to 0A$)

$\to 001B$ (Using $A \to \in$)

$\to 0010B$ (Using $B \to 0B$)

$\to 00101B$ (Using $B \to 1B$)

$\to 00101$ (Using $B \to \in$)

2. Rightmost Derivation-

$S \to A1B$

$\to A10B$ (Using $B \to 0B$)

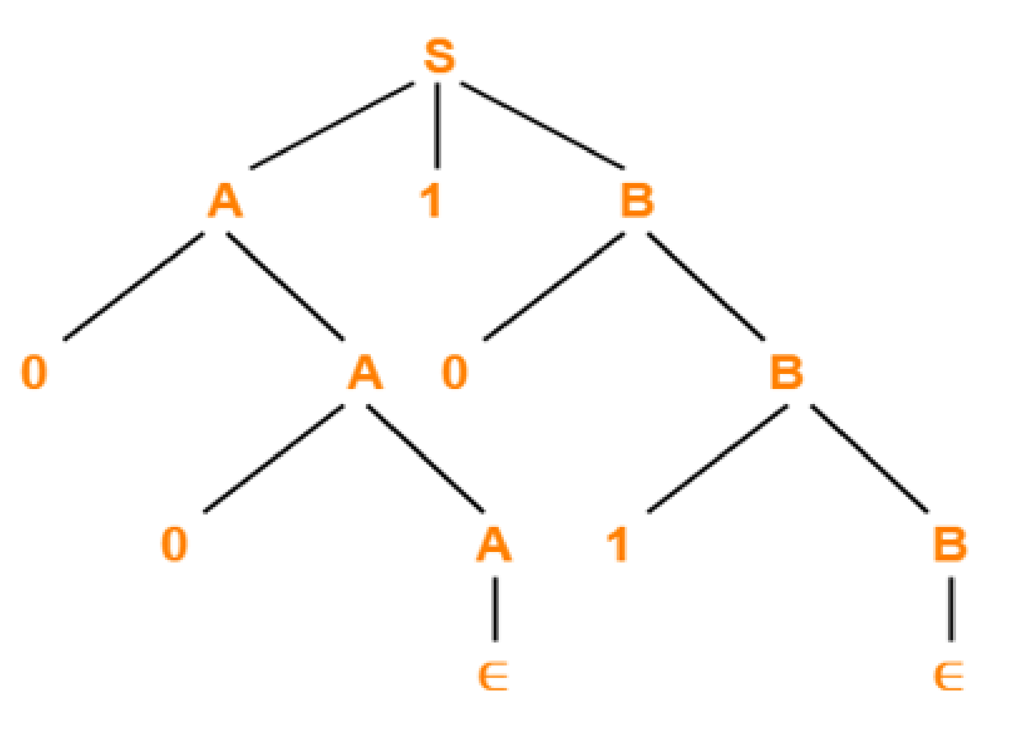
$\to A101B$ (Using $B \to 1B$)

$\to A101$ (Using $B \to \in$)

$\to 0A101$ (Using $A \to 0A$)

$\to 00A101$ (Using $A \to 0A$)

$\to 00101$ (Using $A \to \in$)



Parse Tree

Whether we consider the leftmost derivation or rightmost derivation, we get the above parse tree.
Thus the given grammar is **unambiguous.**

Left Recursion

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
AMAL JYOTHI COLLEGE OF ENGINEERING
KANJIRAPPALLY

Left Recursion

- A grammar is said to be left –recursive if it has a non-terminal A such that there is a derivation **A →Aα**, for some string α.
- The generation is **left-recursive** if the *leftmost symbol on the right side is equivalent to the nonterminal on the left side*.

  Ex: **exp** → **exp** + **term.**

- A **grammar** that contains a production having **left recursion** is called as a **Left-Recursive Grammar**.
- Similarly, if the rightmost symbol on the right side is equal to left side is called **Right -Recursion.**

Direct Left Recursion:

- If we write it as a function, $A \rightarrow A\alpha \mid \beta$

```
A()
{
    A()
    some α.....
}
```

A() is going to call A() , it will result in recursion.

LR: $A \rightarrow A\alpha \mid \beta$

Why Eliminate Left Recursion?

Consider an example: **E -> E+T | T**

- The above example will go in an infinite loop because the **function E** keeps calling itself which causes a problem for a parser to go in an infinite loop which is a never-ending process
- Eg : **S → Sa / ∈** - Left Recursive Grammar
- Left recursion is considered to be a problematic situation for Top down parsers.
- Therefore, left recursion has to be eliminated from the grammar.

Elimination of Left Recursion

- Left recursion is eliminated by converting the grammar into a right recursive grammar.
- If we have the left-recursive pair of productions-

$A \rightarrow A\alpha \mid \beta$ (Left Recursive Grammar)

where β does not begin with an A.

Then, we can eliminate left recursion by replacing the pair of productions with-

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \in$ (Right Recursive Grammar)

This right recursive grammar functions same as left recursive grammar.

Left recursive grammar: $A \rightarrow A\alpha \mid \beta$

Replace with following productions:

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

Consider an example: **E -> E+T | T**

In the grammar, E can be replaced with A,
+T can be replaced with α and
T with β

Thus after removing LR,

E→TE'

E'→+TE' |∊

For example:

$E \rightarrow E + T \mid T \longrightarrow \begin{cases} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \epsilon \end{cases}$

$T \rightarrow T * F \mid F \longrightarrow \begin{cases} T \rightarrow FT' \\ T' \rightarrow *FT' \mid \epsilon \end{cases}$

$F \rightarrow (E) \mid id \longrightarrow F \rightarrow (E) \mid id$

LR: $A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

EXAMPLE

Consider the following grammar and eliminate left recursion-

$$A \rightarrow ABd \mid Aa \mid a$$
$$B \rightarrow Be \mid b$$

LR: $A \rightarrow A\alpha \mid \beta$

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

Solution-

The grammar after eliminating left recursion is-

$$A \rightarrow aA'$$
$$A' \rightarrow BdA' \,/\, aA' \,/\, \epsilon$$
$$B \rightarrow bB'$$
$$B' \rightarrow eB' \,/\, \epsilon$$

EXAMPLE

Consider the following grammar and eliminate left recursion

$$S \rightarrow (L) \mid a$$

$$L \rightarrow L\,,S \mid S$$

LR: $A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

Solution-

The grammar after eliminating left recursion

$$S \rightarrow (L) \mid a$$

$$L \rightarrow SL'$$

$$L' \rightarrow ,SL' \mid \epsilon$$

Eliminating Immediate Left Recursion

Immediate left recursion occurs , if there is a Production of the form:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Where no β_i begins with A.

Replace it with

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \in$$

Consider the following grammar and eliminate left recursion.

$$S \rightarrow \underbrace{Sab}_{\alpha_1} \mid \underbrace{Scd}_{\alpha_2} \mid \underbrace{Sef}_{\alpha_3} \mid \underbrace{g}_{\beta_1} \mid \underbrace{h}_{\beta_2}$$

Apply the rule:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \in$$

$$S \rightarrow g\,S' \mid h\,S'$$

$$S' \rightarrow abS' \mid cdS' \mid efS'$$

Indirect Left Recursion:

A grammar is said to have **indirect left recursion** if, starting from any symbol of the grammar, it is possible to derive a string whose head is that symbol.

For example,

A --> **B**r

B --> **C**d

C --> **A**t

Where A, B, C are non-terminals and r, d, t are terminals.

Here, starting with A, we can derive A again on substituting C to B and B to A.

Algorithm to remove Indirect Recursion

Consider an example :

A1 --> A2 A3
A2 --> A3 A1 | b
A3 --> A1 A1 | a

Where A1, A2, A3 are non terminals and a, b are terminals.

Step 1: Identify the productions which can cause **indirect left recursion**.

A3 --> A1 A1 | a

Step 2: Substitute this production at the place the terminal is present in any other production.

substitute **A1–> A2 A3** in production of A3. ie, **A3 –> A2 A3 A1.**

Now in this production substitute A2–> A3 A1 / b and then replace this by,

A3 --> A3 A1 A3 A1 / b A3 A1

Now the new production is converted in form of direct left recursion , solve this by direct left recursion method.

A3 --> A3 A1 A3 A1 | b A3 A1 | a

Eliminating direct left recursion in the above,

LR: $A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \in$

A3 --> a | b A3 A1 | aA' | b A3 A1A'
A' --> A1 A3 A1 | A1 A3 A1A'

The resulting grammar is then:

A1 --> A2 A3
A2 --> A3 A1 | b
A3 --> a | b A3 A1 | aA' | b A3 A1A'
A' --> A1 A3 A1 | A1 A3 A1A'

Consider the following grammar and eliminate left recursion-

$$X \to XSb \,/\, Sa \,/\, b$$

$$S \to Sb \,/\, Xa \,/\, a$$

Solution-

This is a case of indirect left recursion.

Step-01:

First let us eliminate left recursion from $X \to XSb \,/\, Sa \,/\, b$

Eliminating left recursion from here, we get-

$$\mathbf{X \to SaX' \,/\, bX'}$$

$$\mathbf{X' \to SbX' \,/\, \in}$$

Now, given grammar becomes

$$X \rightarrow SaX' / bX'$$
$$X' \rightarrow SbX' / \in$$
$$S \rightarrow Sb / Xa / a$$

Step-02:

Substituting the productions of X in **S → Xa,** we get the following grammar

$$X \rightarrow SaX' / bX'$$
$$X' \rightarrow SbX' / \in$$
$$S \rightarrow Sb / SaX'a / bX'a / a$$

Step-03:

Now, eliminating left recursion from the productions of S, we get the following grammar

$$\mathbf{X \rightarrow SaX' / bX'}$$
$$\mathbf{X' \rightarrow SbX' / \in}$$
$$\mathbf{S \rightarrow bX'aS' / aS'}$$
$$\mathbf{S' \rightarrow bS' / aX'aS' / \in}$$

Consider the following grammar and eliminate left recursion

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \in$$

Solution-

This is a case of **indirect left recursion**.

Step-01:

First let us eliminate left recursion from **$S \rightarrow Aa \mid b$**

This is already free from left recursion.

Step-02:

Substituting the productions of S in **$A \rightarrow Sd$**, we get the following grammar-

$$\mathbf{S \rightarrow Aa \mid b}$$

$$\mathbf{A \rightarrow Ac \mid Aad \mid bd \mid \in}$$

Now , $A \rightarrow Ac \mid Aad \mid bd \mid \in$ is in Immediate Left Recursion.

Apply the Rule :

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \in$$

Step-03:

After eliminating left recursion from the productions of A, we get:

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid adA' \mid \in$$

Thank you