Marwadi University
Marwadi Chandarana Group
NAAC A+

CD : COMPILER DESIGN
Parsing

Department of CE

Unit no : 3
Parsing
(01CE0714)

Prof. Shilpa Singhal

Outline :

- Role of parser
- Parse tree
- Classification of grammar
- Derivation and Reduction
- Ambiguous grammar
- Left Recursion
- Left Factoring
- Top-down Bottom-up parsing
- LR Parsers – LR(0), SLR, CLR , LALR

Marwadi University
Marwadi Chandarana Group
NAAC A+

Department of CE

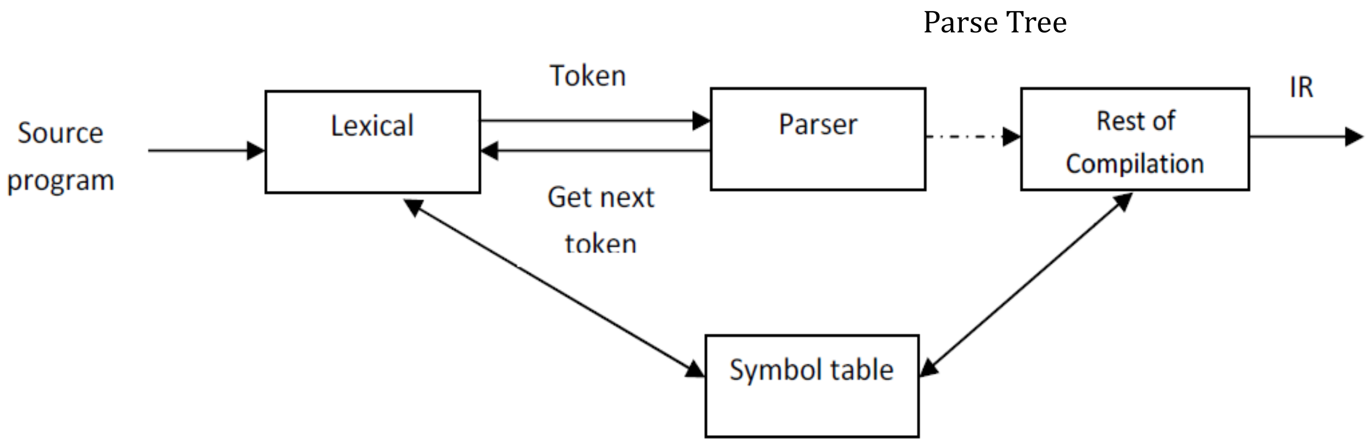
Unit no : 3
Parsing
(01CE0714)

Prof. Shilpa Singhal

Role of Parser

- In our compiler model, the parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language.
- It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

Scanner – Parser Interaction



- For well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

Syntax Error Handling

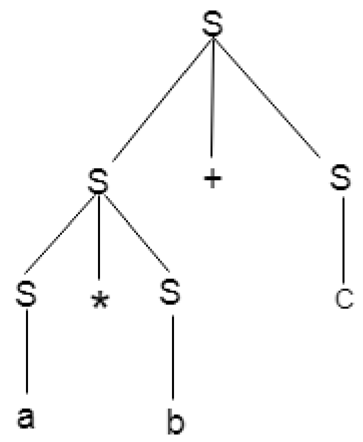
- If a compiler had to process only correct programs, its design and implementation would be greatly simplified.
- But programmers frequently write incorrect programs, and a good compiler should assist the programmer in identifying and locating errors.
- We know that programs can contain errors at many different levels. For example, errors can be
- **Lexical** : Such a misspelling an identifier, keyword, or operator
- **Syntactic** : Such as arithmetic expression with unbalanced parenthesis
- **Semantic** : Such as an operator applied to incompatible operand
- **Logical** : Such as infinitely recursive call

Syntax Error Handling

- The error handler in a parser has simple-to-state goals :
- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect sub sequent errors.
- It should not significantly slow down the processing of correct programs.

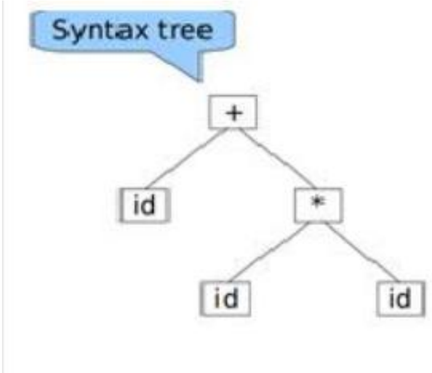
Parse tree

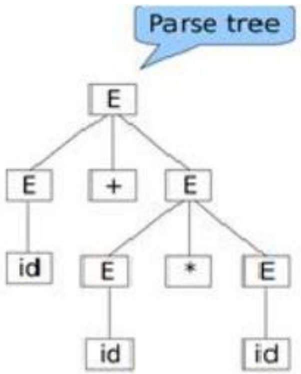
- Parse tree is graphical representation of symbol. Symbol can be terminal as well as non-terminal.
- The root of parse tree is start symbol of the string.
- Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.
- Example:-



Parse tree V/S Syntax tree

Syntax tree	Parse tree
Interior nodes are operator, leaves are operands	Interior nodes are non-terminal, leaves are terminal.
When representing program in a tree structure, usually syntax tree is used.	Rarely constructed as data structure.
Represents the abstract syntax of a program.	Represents the concrete syntax of a program.
Grammar:- E→E*E | E+E | id Program: a+b*c	Grammar:- E→E*E | E+E | id Program: a+b*c





Classification of Grammar

- Grammars are classified on the basis of production they use (Chomsky, 1963).
- Given below are class of grammar where each class has its own characteristics and limitations.

1. **Type-0 Grammar:- Recursively Enumerable Grammar**
   - These grammars are known as phrase structure grammars. Their productions are of the form,
   - $\alpha = \beta$, where both α and β are terminal and non-terminal symbols.
   - This type of grammar is not relevant to Specifications of programming languages.

2. **Type-1 Grammar:- Context Sensitive Grammar**
   - These Grammars have rules of the form $\alpha A \beta \rightarrow \alpha \Upsilon \beta$ with A nonterminal and α, β, Υ strings of terminal and nonterminal symbols. The string α and β may be empty but Υ must be nonempty.
   - Eg:- AB->CDB

     Ab->Cdb

     A->b

Classification of Grammar

3. **Type-2 Grammar:- Context Free Grammar**
   - These are defined by the rules of the form A → ϒ, with A nonterminal and ϒ a sting of terminal and nonterminal Symbols. These grammar can be applied independent of its context so it is Context free Grammar (CFG). CFGs are ideally suited for programming language specification.
   - Eg:- A → aBc

4. **Type-3 Grammar:- Regular Grammar**
   - It restrict its rule of single nonterminal on the left hand side and a right-hand side consisting of a single terminal, possibly followed by a single nonterminal. The rule S → ϵ is also allowed if S does not appear on the right side of any rule.
   - Eg:- A → ϵ

     A → a

     A → aB

Derivation

- Let production P_1 of grammar G be of the form

  $P_1 : A ::= \alpha$

  and let β be a string such that $\beta = \gamma A \theta$, then replacement of A by α in string β constitutes a derivation according to production P_1.

- Example

  <Sentence> ::= <Noun Phrase><Verb Phrase>

  <Noun Phrase> ::= <Article> <Noun>

  <Verb Phrase> ::= <Verb><Noun Phrase>

  <Article> ::= a | an | the

  <Noun> ::= boy | apple

  <Verb> ::= ate

Derivation

- The following strings are ***<u>sentential form</u>***.

  <Sentence>

  <Noun Phrase> <Verb Phrase>

  the boy <Verb Phrase>

  the boy <verb> <Noun Phrase>

  the boy ate <Noun Phrase>

  the boy ate an apple

Derivation

- The process of deriving string is called Derivation and graphical representation of derivation is called derivation tree or parse tree.
- Derivation is a sequence of a production rules, to get the input string.
- During parsing we take two decisions:

1) Deciding the non terminal which is to be replaced.
2) Deciding the production rule by which non terminal will be replaced.

For this we are having:

1) Left most derivation

2) Right most derivation

Left Derivation

- A derivation of a string S in a grammar G is a left most derivation if at **every step the left most non terminal is replaced.**

Example:

- Production:

S→ S + S

S→ S * S

S→ id

- String:- id+id*id

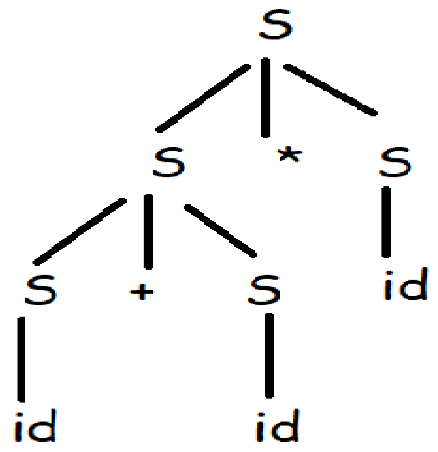
S→ S * S

S→S + S * S

S→ id + S * S

S→ id + id * S

S→ id + id * id



Right Derivation

- A derivation of a string S in a grammar G is a right most derivation if **at every step the Right most non terminal is replaced.**

Example:

- Production:

S→ S + S

S→ S * S

S→ id

- String:- id+id*id

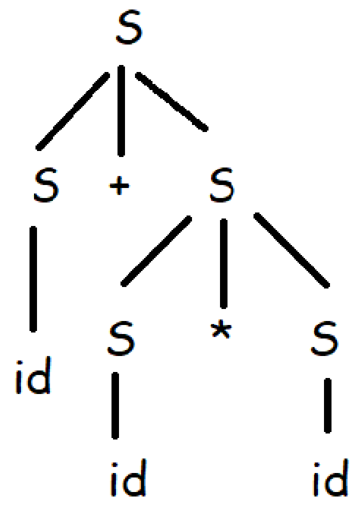
S→ S + S

S→S + S * S

S→ S + S * id

S→ S + id * id

S→ id + id * id



Left Derivation and Right Derivation

Derive the string "abb" for leftmost derivation and rightmost derivation using a CFG given by,

$S \rightarrow AB \mid \varepsilon$

$A \rightarrow aB$

$B \rightarrow Sb$

Leftmost derivation:

S

AB

$\boxed{aB}\ B$

$a\ \boxed{Sb}\ B$

$a\ \boxed{\varepsilon}\ bB$

$ab\ \boxed{Sb}$

$ab\ \boxed{\varepsilon}\ b$

abb

Rightmost derivation:

S

AB

$A\ \boxed{Sb}$

$A\ \boxed{\varepsilon}\ b$

$\boxed{aB}\ b$

$a\ \boxed{Sb}\ b$

$a\ \boxed{\varepsilon}\ bb$

abb

Left Derivation and Right Derivation

1.Derive the string "aabbabba" for leftmost derivation and rightmost derivation using a CFG given by,

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA$

$B \rightarrow b \mid bS \mid aBB$

2.Derive the string "00101" for leftmost derivation and rightmost derivation using a CFG given by,

$S \rightarrow A1B$

$A \rightarrow 0A \mid \varepsilon$

$B \rightarrow 0B \mid 1B \mid \varepsilon$

Left Derivation and Right Derivation

Soution.1

Leftmost derivation:		**Rightmost derivation:**	
S		S	
aB	$S \longrightarrow aB$	aB	$S \longrightarrow aB$
aaBB	$B \longrightarrow aBB$	aaBB	$B \longrightarrow aBB$
aabB	$B \longrightarrow b$	aaBbS	$B \longrightarrow bS$
aabbS	$B \longrightarrow bS$	aaBbbA	$S \longrightarrow bA$
aabbaB	$S \longrightarrow aB$	aaBbba	$A \longrightarrow a$
aabbabS	$B \longrightarrow bS$	aabSbba	$B \longrightarrow bS$
aabbabbA	$S \longrightarrow bA$	aabbAbba	$S \longrightarrow bA$
aabbabba	$A \longrightarrow a$	aabbabba	$A \longrightarrow a$

Left Derivation and Right Derivation

Soution.2

Leftmost derivation:	**Rightmost derivation:**
S	S
A1B	A1B
0A1B	A10B
00A1B	A101B
001B	A101
0010B	0A101
00101B	00A101
00101	00101

Reduction

Let production P_1 of grammar G be of the form

$P_1 : A ::= \alpha$

and let σ be a string such that $\sigma = \gamma\ \alpha\ \theta$, then replacement of α by A in string σ constitutes a reduction according to production P_1.

Step	String
0	the boy ate an apple
1	<Article> boy ate an apple
2	<Article> <Noun> ate an apple
3	<Article> <Noun> <Verb> an apple
4	<Article> <Noun> <Verb> <Article> apple
5	<Article> <Noun> <Verb> <Article> <Noun>
6	<Noun Phrase> <Verb> <Article> <Noun>
7	<Noun Phrase> <Verb> <Noun Phrase>
8	<Noun Phrase> <Verb Phrase>
9	<Sentence>

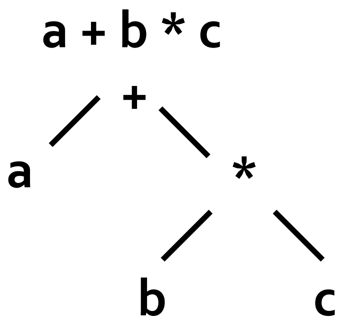
Ambiguous Grammar

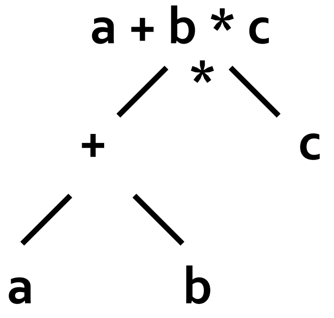
- A CFG is said to be **ambiguous** if there exists more than one derivation tree for the given input string i.e., more than one LeftMost Derivation Tree (LMDT) or RightMost Derivation Tree (RMDT).
- It implies the possibility of different interpretation of a source string.
- Existence of ambiguity at the level of the syntactic structure of a string would mean that more than one parse tree can be built for the string. So string can have more than one meaning associated with it.

Ambiguous Grammar

E → Id| E + E | E * E

Id → a | b | c



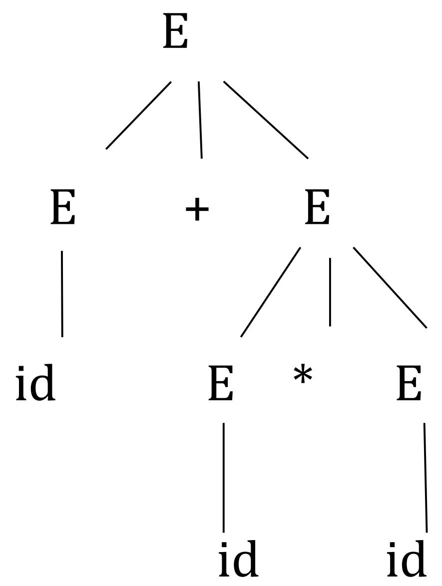


Both tree have same string : a + b * c

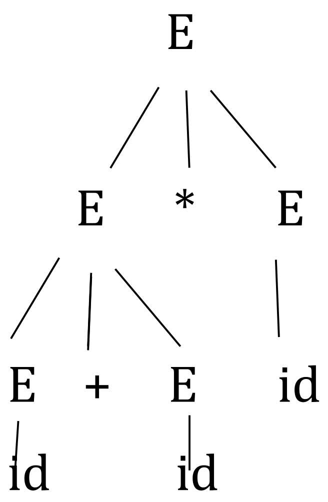
Ambiguous Grammar

$E \rightarrow E + E \mid E * E \mid id$

By parse tree:-



Parse tree-1



Parse tree-2

Ambiguous Grammar Example:-

Prove that given grammar is ambiguous grammar:

E→ a | Ea | bEE | EEb | EbE

Ans:-

Assume string baaab

E→ bEE

baE

baEEb

baaEb

baaab

Left derivation-1

OR

E→ EEb

bEEEb

baEEb

baaEb

baaab

Left derivation-2

Exercise: Ambiguous Grammar

Check whether following grammars are ambiguous or not:

1. $S \rightarrow aS \mid Sa \mid \epsilon$ (string: aaaa)
2. $S \rightarrow aSbS \mid bSaS \mid \epsilon$ (string: abab)
3. $S \rightarrow SS+ \mid SS^* \mid a$ (string: aa+a*)

Left Recursion

- In leftmost derivation by scanning the input from left to right, grammars of the form **$A \rightarrow A\ x$** may cause endless recursion.
- Such grammars are called **left-recursive** and they must be transformed if we want to use a top-down parser.
- Example:

$E \rightarrow Ea \mid E+b \mid c$

Algorithm

- Assign an ordering from A1,…….An to the non terminal of the grammar;
- For i = 1 to n do

begin

for j=1 to i-1 do

begin

replace each production of the form $A_i \rightarrow A_i\Upsilon$

by the productions $A_i \rightarrow \delta_1\Upsilon \mid \delta_2\Upsilon \mid \ldots\ldots \mid \delta_k\Upsilon$

where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots\ldots\ldots\ldots \mid \delta_k$ are all current Aj production.

end

eliminate the intermediate left recursion among A_i productions.

end

Left Recursion

- There are three types of left recursion:

  direct ($A \to A\ x$)

  indirect ($A \to B\ C,\ B \to A$)

  hidden ($A \to B\ A,\ B \to \varepsilon$)

To eliminate direct left recursion replace

$$\mathbf{A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n}$$

with

$$\mathbf{A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'}$$
$$\mathbf{A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon}$$

Example

1. $E \rightarrow E + T \mid T$

   $T \rightarrow T * F \mid F$

   $F \rightarrow (E) \mid id$

Ans.

$$A \rightarrow A\alpha \mid \beta$$

Replace with,

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Example

1. $A \rightarrow Aad \mid Afg \mid b$

Ans:-

Remove left recursion

$A \rightarrow bA'$

$A' \rightarrow adA' \mid fgA' \mid \varepsilon$

2. $A \rightarrow Acd \mid Ab \mid jk$

$B \rightarrow Bh \mid n$

Ans :-

Remove left recursion

$A \rightarrow jkA'$

$A' \rightarrow cdA' \mid bA' \mid \varepsilon$

$B \rightarrow nB'$

$B' \rightarrow hB' \mid \varepsilon$

Example

3. $E \rightarrow Aa \mid b$

   $A \rightarrow Ac \mid Ed \mid \varepsilon$

Ans:-

Replace E,

$E \rightarrow Aa \mid b$

$A \rightarrow Ac \mid \mathit{Aad} \mid \mathit{bd} \mid \varepsilon$

Remove left recursion

$E \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Left Factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
- Consider,
  - $S \rightarrow$ **if** E **then** S **else** S | **if** E **then** S
  - Which of the two productions should we use to expand non-terminal S <u>when the next token is **if**</u>?
  - We can solve this problem by factoring out the common part in these rules. This way, we are postponing the decision about which rule to choose until we have more information (namely, whether there is an **else** or not).
  - This is called **left factoring**

Algorithm

- For each non terminal A find the longest prefix **α** common to two or more of its alternative.
- If **α!=E,** i.e, there is a non trivial common prefix, replace all the A productions **A→ αβ1 | αβ2 | ….. | αβn | ϒ ,**

where ϒ represents all the alternative which do not starts with α by,

A→ αA' | ϒ

A'→ β1 | β2 |…… | βn

Here, A' is new non terminal, repeatedly apply this transformation until no two alternatives for a non-terminal have a common prefix.

Left Factoring

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_n \mid \gamma$

becomes

$A \rightarrow \alpha\ A'' \mid \gamma$

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

Left Factoring Example

E -> T+E | T

T -> V*T | V

V-> id

Ans.

E→ TE’
E’ → +E | ε

T→ VT’

T’ → *T | ε

V → id

Left Factoring Example

1. S → cdLk | cdk | cd

L→ mn | ε

Ans.

S→ cdS'

S' → Lk | k | ε

L→ mn | ε

2. E → iEtE | iEtEeE | a

A→b

Ans.

E → iEtEE' | a

E' → ε | eE

A → b

Left Factoring Example

3. A → xByA | xByAzA | a

Ans.

A→ xByAA'|a

A' → ε| zA

4. A → aAB | aA| a

Ans.

A→ aA'

A' → AB| A | ε

A' → AA''|ε

A''→ B|ε