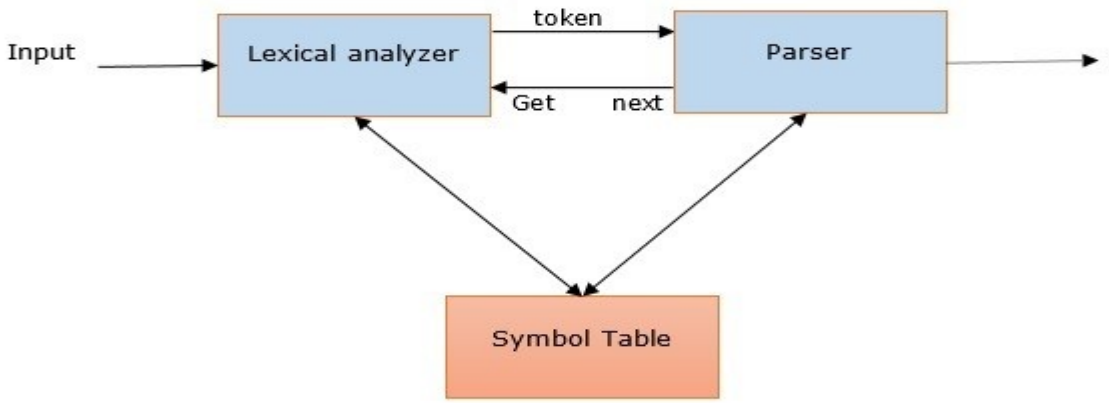
Syntax Analysis

- It is the second phase of the compiler.
- It takes token produced by lexical analyzer and generate a parse tree. In this phase it will check whether the corresponding syntax of the source program is correct or not.
- If the syntax is not correct the error handler will report those errors to the user. If the syntax is correct then it generate a parse tree of the corresponding source program.
- 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. Therefore, this phase uses context-free grammar (CFG)



The role of parser

- The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string 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.
    1. It verifies the structure generated by the tokens based on the grammar.
    2. It constructs the parse tree.
    3. It reports the errors.
    4. It performs error recovery.

Issues :

Parser cannot detect errors such as:

1. Variable re-declaration
2. Variable initialization before use
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

Context free grammar (CFG)

- Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.
- Context free grammar G can be defined by four tuples as:

G= (V, T, P, S)

Where,

- **G** describes the grammar,
- T describes a finite set of terminal symbols. (terminal are the basic symbols and it can't be further derive)
- V describes a finite set of non-terminal symbols (whatever comes in the lefthand sides of a production is terminal symbol)
- P describes a set of production rules (The set of Productions where each production consists of a non-terminal, called the left side followed by an arrow, followed by a string of non-terminals and/or terminals called the right side
- S is the start symbol.
- Example of terminal an non-terminal symbol:

-
- Terminal:{ (,), *, +} Non-terminal :{S,A}

Contd….

- Example: find the terminal and non-terminal symbol

$$
\begin{aligned}
\textit{expr} &\longrightarrow \textit{expr op expr} \\
\textit{expr} &\longrightarrow (\textit{expr}) \\
\textit{expr} &\longrightarrow - \textit{expr} \\
\textit{expr} &\longrightarrow \mathbf{id} \\
\textit{op} &\longrightarrow + \\
\textit{op} &\longrightarrow - \\
\textit{op} &\longrightarrow * \\
\textit{op} &\longrightarrow / \\
\textit{op} &\longrightarrow \uparrow
\end{aligned}
$$

Derivation

- Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing we have to take two decisions. These are as follows:
  - We have to decide the non-terminal which is to be replaced.
  - We have to decide the production rule by which the non-terminal will be replaced.
- We have two options to decide which non-terminal to be replaced with production rule.

1. Leftmost derivation.
2. Rightmost derivation.

Contd…

- **Left most derivation**: In leftmost derivation, at each and every step the leftmost non-terminal is expanded by substituting its corresponding production to derive a string.

$E \longrightarrow E + E \mid E * E \mid id$

Solution:

$w = id + id * id$

$E \xrightarrow[lm]{} E+E$

$E \xrightarrow[lm]{} id+E \quad [E \longrightarrow id]$

$E \xrightarrow[lm]{} id+E * E \quad [E \longrightarrow E * E]$

$E \xrightarrow[lm]{} id + id * E \quad [E \longrightarrow id]$

$E \xrightarrow[lm]{} id + id * id \quad [E \longrightarrow id]$

- **Rightmost Derivation:** In rightmost derivation, at each and every step the rightmost non-terminal is expanded by substituting its corresponding production to derive a string.

$$E \longrightarrow E + E \mid E * E \mid id$$

Solution:

$w = id + id * id$

$E \xrightarrow[rm]{} E+E$

$E \xrightarrow[rm]{} E + E * E \quad [E \longrightarrow E * E]$

$E \xrightarrow[rm]{} E + E * id \quad [E \longrightarrow id]$

$E \xrightarrow[rm]{} E + id * id \quad [E \longrightarrow id]$

$E \xrightarrow[rm]{} id + id * id \quad [E \longrightarrow id]$

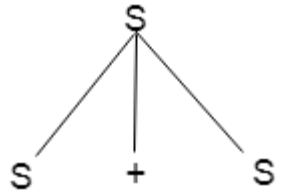
Parse tree

- Parse tree is the pictorial representation of the derivation process. In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.
- While constructing a parse tree, some rule must be follow:
- Root node must be the starting symbol
- All interior nodes have to be non-terminals.
- All leaf nodes have to be terminals.

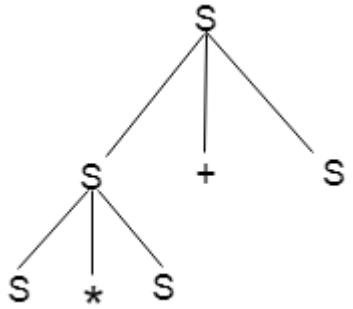
- Example:

- Input:

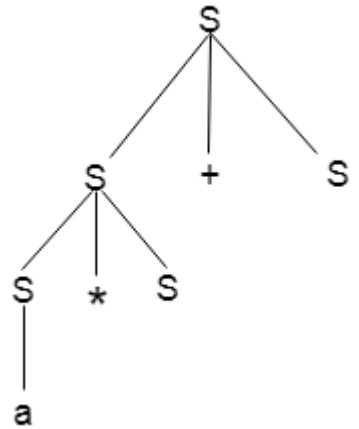
Step 1 :



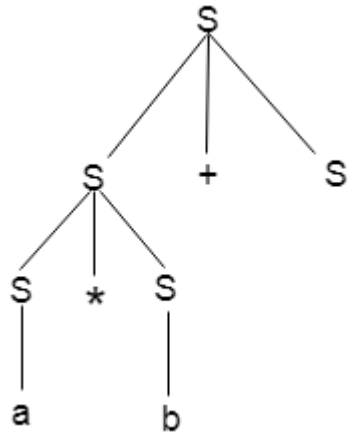
Step 2:



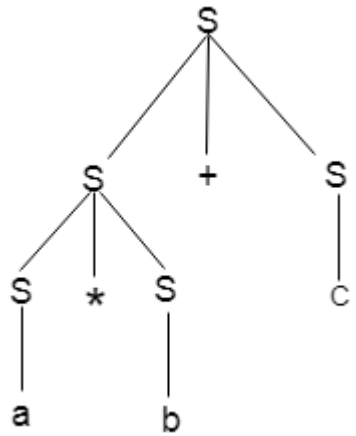
- Step 3:



- Step 4:



- Step 5:



Ambiguity in Grammar

- A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string. If the grammar is not ambiguous, then it is called unambiguous.
- If the grammar has ambiguity, then it is not good for compiler construction. No method can automatically detect and remove the ambiguity, but we can remove ambiguity by re-writing the whole grammar without ambiguity.
- Example: Check whether the given grammar is ambiguous or not?

  E → E + E

  E → E - E

  E → id
- String: id + id - id

Contd….

- First Leftmost derivation

E → E + E

→ id + E

→ id + E - E

→ id + id - E

→ id + id- id

- Second Leftmost derivation

E → E - E

→ E + E - E

→ id + E - E

→ id + id - E

→ id + id - id

- Since there are two leftmost derivation for a single string "id + id - id", the grammar G is ambiguous.

How to remove ambiguity?

We can remove ambiguity solely on the basis of the following two properties –

I. **Precedence –** If different operators are used, we will consider the precedence of the operators. The three important characteristics are :
   - ✓ The level at which the production is present denotes the priority of the operator used.
   - ✓ The production at **higher levels** will have **operators with less priority**. In the parse tree, the nodes which are at top levels or close to the root node will contain the lower priority operators.
   - ✓ The production at **lower levels** will have operators with **higher priority**. In the parse tree, the nodes which are at lower levels or close to the leaf nodes will contain the higher priority operators.

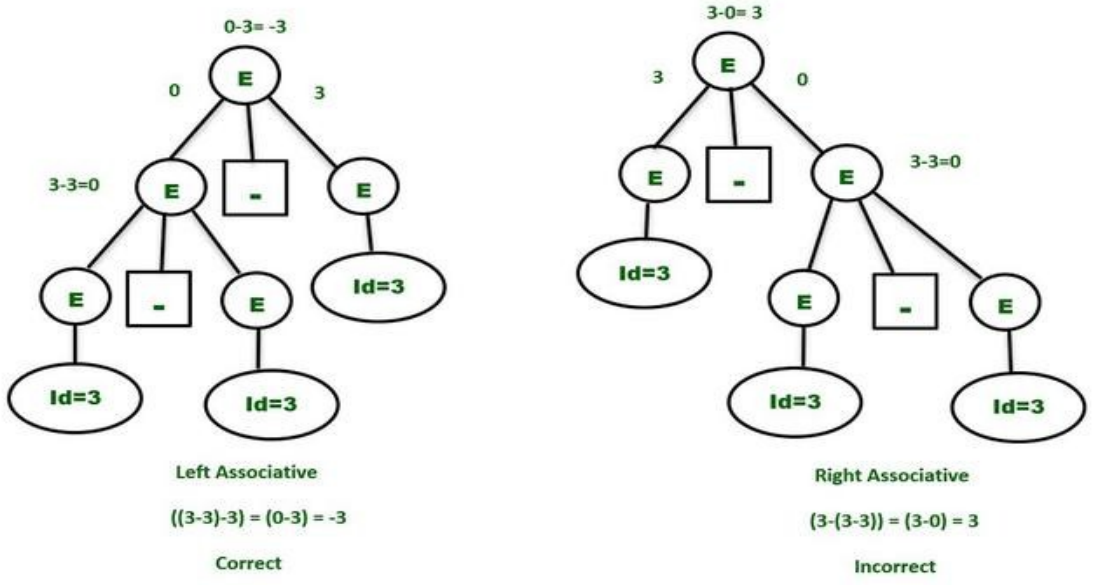
II. **Associativity –**If the same precedence operators are in production, then we will have to consider the associativity.
   - ✓ If the associativity is left to right, then we have to prompt a left recursion in the production. The parse tree will also be left recursive and grow on the left side. +, -, *, / are left associative operators.
   - ✓ If the associativity is right to left, then we have to prompt the right recursion in the productions. The parse tree will also be right recursive and grow on the right side. ^ is a right associative operator.

Example:

- Consider the ambiguous grammar ***E -> E-E | id***
- Say, we want to derive the string id-id-id. Let's consider a single value of id=3 to get more insights. The result should be :

3-3-3 =-3

- Since the same priority operators, we need to consider associativity which is left to right.



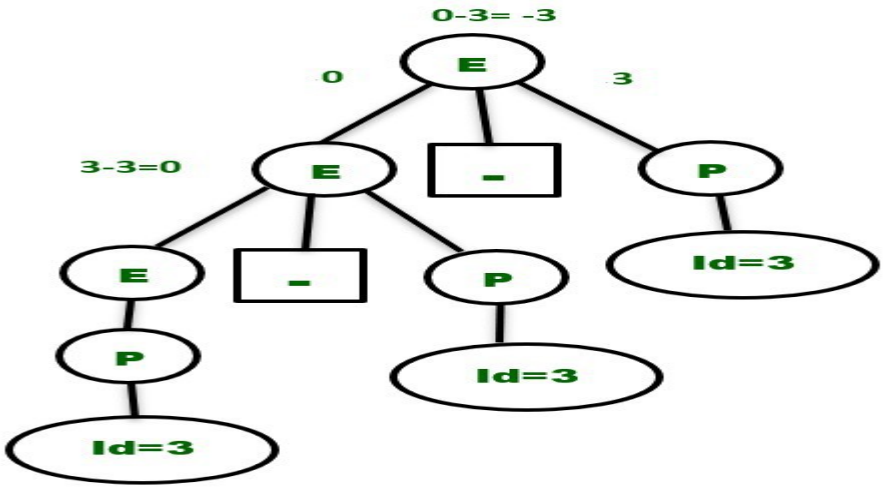
Contd….

- So, to make the above grammar unambiguous, simply make the grammar Left Recursive and we need to place a random non-terminal say P in place of the right non-terminal. The grammar becomes :

  E -> E – P | P

  P -> id

- The above grammar is now unambiguous and will contain only one Parse Tree for the above expression as shown below –



Example:

- Consider the grammar shown below, which has two different operators :

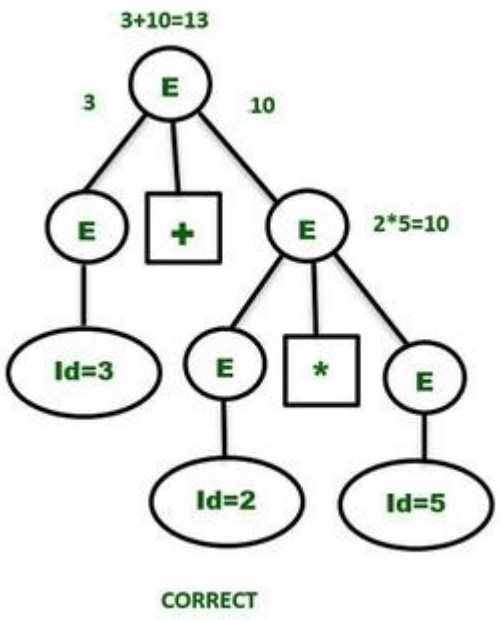
  E -> E + E | E * E | id

- Clearly, the above grammar is ambiguous as we can draw two parse trees for the string “id+id*id” as shown below. Consider the expression :

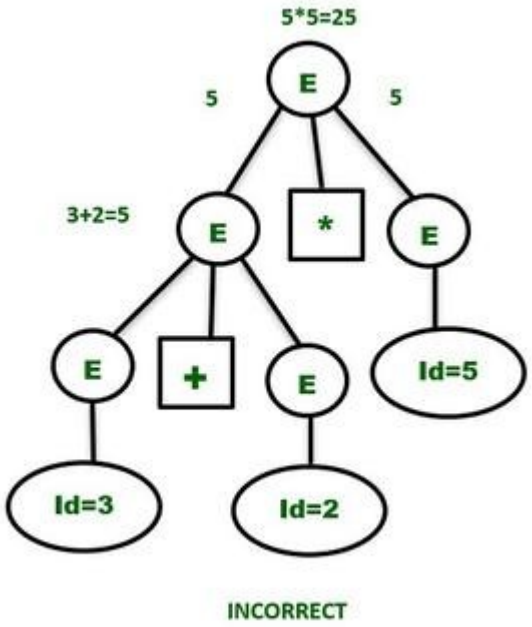
  3 + 2 * 5 // “*” has more priority than “+”

- The correct answer is : (3+(2*5))=13

Contd…



CORRECT



INCORRECT

Contd….

- The “+” having the least priority has to be at the upper level and has to wait for the result produced by the “*” operator which is at the lower level. So, the first parse tree is the correct one and gives the same result as expected.
- The unambiguous grammar will contain the productions having the highest priority operator (“*” in the example) at the lower level and vice versa. The associativity of both the operators are Left to Right. So, the unambiguous grammar has to be left recursive. The grammar will be :

E -> E + P // + is at higher level and left associative

E -> P

P -> P * Q // * is at lower level and left associative

P -> Q

Q -> id

(or)

E -> E + P | P

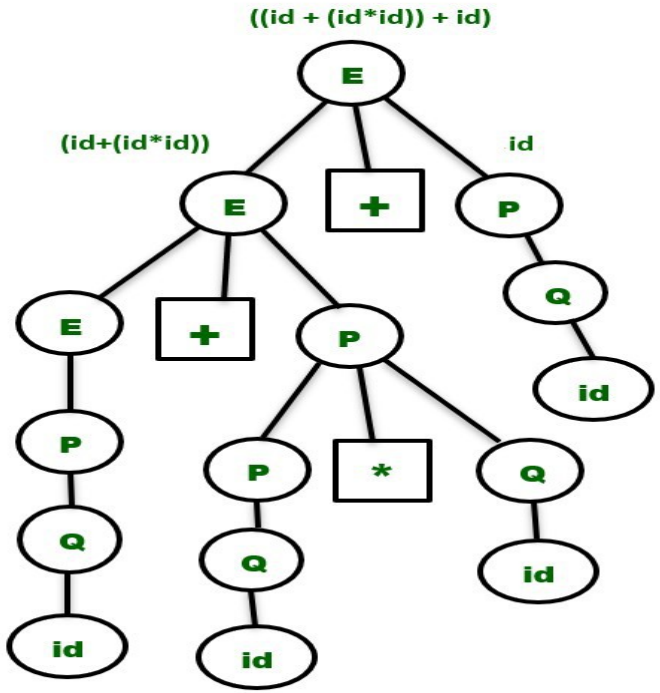
P -> P * Q | Q

Q -> id

E is used for doing addition operations and **P** is used to perform multiplication operations. They are independent and will maintain the precedence order in the parse tree.

Contd…

- The parse tree for the string **” id+id*id+id ”** will be –



Elimination of left recursion

- **Left recursion**: A grammar in the form G = (V, T, S, P) is said to be in left recursive form if it has the production rules of the form

$$A \rightarrow A\,\alpha \,|\beta$$

- **Problem with Left Recursion**: If a left recursion is present in any grammar then, during parsing in the syntax analysis part of compilation, there is a chance that the grammar will create an infinite loop. This is because, at every time of production of grammar, A will produce another A without checking any condition.
- Left recursive grammar is not suitable for Top down parsers.
- This is because it makes the parser enter into an infinite loop.
- To avoid this situation, it is converted into its equivalent right recursive grammar.
- This is done by eliminating left recursion from the left recursive grammar.

Contd…

$$A \rightarrow A\,\alpha \mid \beta.$$

- The above Grammar is left recursive because the left of production is occurring at a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with

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

Example:

- Consider the Left Recursion from the Grammar.

  E → E + T|T

  T → T * F|F

  F → (E)|id

- Eliminate immediate left recursion from the Grammar.
- Here, E → E + T|T is left recursive

  A = E, α = +T, β = T

  A → A α |β is changed to A → βA′ and A′ → α A′|ε in order to eliminate the left recursion.

  A → βA′ means E → TE′

  A′ → α A′|ε means E′ → +TE′|ε
- And the next production is T → T * F|F with A → Aα|β
  - A = T, α = * F, β = F
  - A → β A′ means T → FT′
  - A → α A′|ε means T′ →* FT′|ε

Production F → (E)|id does not have any left recursion

Contd…

- After eliminating left recursion, we get the following grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Example 2:

Indirect left-recursion

$S \rightarrow Aa|b$

$A \rightarrow Ac|Sd|\in$

Left factoring

- Left factoring is used to convert a left-factored grammar into an equivalent grammar to remove the uncertainty for the top-down parser. In left factoring, we separate the common prefixes from the production rule.
- The following algorithm is used to perform left factoring in the grammar-
- Suppose the grammar is in the form:

- Where **A** is a non-terminal and **α** is the common prefix.
- We will separate those productions with a common prefix and then add a new production rule in which the new non-terminal we introduced will derive those productions with a common prefix.

Contd….

S → iEtS / iEtSeS / a

E → b

- Perform left factoring for the above grammar
- The left factored grammar is-

S → iEtSS’ / a

S’ → eS / ∈

E → b

Do left factoring in the following grammar-

- A → aAB / aBc / aAc
- **Step-01:**

  A → aA’

  A’ → AB / Bc / Ac
- Again, this is a grammar with common prefixes.

  A’ → AD / Bc

  D → B / c
- This is a left factored grammar.

FIRST AND FOLLOW

- **Rules for calculating First:**

1. If then First(A)= {a}
2. If , then First (A) = {}
3. If then First (A) = First (B) , if First (B) doesn’t contains

   if First (B) contain then First (A) = First (B) First (C)

- **Rules for calculating Follow:**

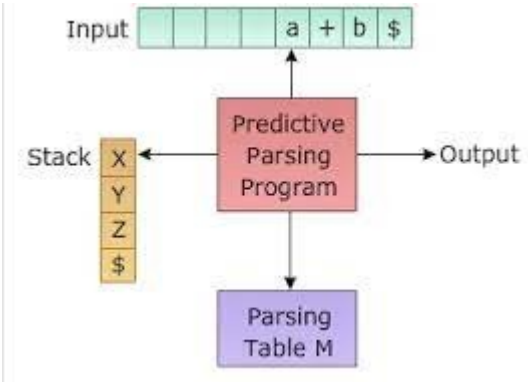
1. If ‘S’ is a start symbol then, Follow (S) = { $ }.
2. If , then Follow (B) = First ( ) , if First ( ) doesn’t contain
3. If , then Follow (B) = Follow (A)
4. If , where , then Follow(B) = { First(β) – ∈ } ∪ Follow(A)

- **If X is Grammar Symbol, then First (X) will be −**
- If X is a terminal symbol, then FIRST(X) = {X}
- If X → ε, then FIRST(X) = {ε}
- If X is non-terminal & X → a α, then FIRST (X) = {a}
- If X → Y_1, Y_2, Y_3, then FIRST (X) will be
- (a) If Y is terminal, then
- FIRST (X) = FIRST (Y_1, Y_2, Y_3) = {Y_1}
- (b) If Y_1 is Non-terminal and
- If Y_1 does not derive to an empty string i.e., If FIRST (Y_1) does not contain ε then, FIRST (X) = FIRST (Y_1, Y_2, Y_3) = FIRST(Y_1)
- (c) If FIRST (Y_1) contains ε, then.
- FIRST (X) = FIRST (Y_1, Y_2, Y_3) = FIRST(Y_1) − {ε} ∪ FIRST(Y_2, Y_3)
- Similarly, FIRST (Y_2, Y_3) = {Y_2}, If Y_2 is terminal otherwise if Y_2 is Non-terminal then
- FIRST (Y_2, Y_3) = FIRST (Y_2), if FIRST (Y_2) does not contain ε.
- If FIRST (Y_2) contain ε, then
- FIRST (Y_2, Y_3) = FIRST (Y_2) − {ε} ∪ FIRST (Y_3)
- Similarly, this method will be repeated for further Grammar symbols, i.e., for Y_4, Y_5, Y_6 … . Y_K.

- **Computation of FOLLOW**
- **Follow (A) is defined as the collection of terminal symbols that occur directly to the right of A.**
  - FOLLOW(A) = {a|S ⇒* αAaβ where α, β can be any strings}
- **Rules to find FOLLOW**
- If S is the start symbol, FOLLOW (S) ={$}
- If production is of form A → α B β, β ≠ ε.
- (a) If FIRST (β) does not contain ε then, FOLLOW (B) = {FIRST (β)}
  - Or
- (b) If FIRST (β) contains ε (i. e. , β ⇒* ε), then
- FOLLOW (B) = FIRST (β) − {ε} ∪ FOLLOW (A)
- ∵ when β derives ε, then terminal after A will follow B.
- If production is of form A → αB, then Follow (B) ={FOLLOW (A)}.

LL(1) parser (or) predictive parser (or) non-recursive descent parser

- **LL(1) Parsing:** Here the 1st **L** represents that the scanning of the Input will be done from the Left to Right manner and the second **L** shows that in this parsing technique, we are going to use the Left most Derivation Tree. And finally, the **1** represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.



- **Essential conditions to check first are as follows:**

1. The grammar is free from left recursion.
2. The grammar should not be ambiguous.
3. The grammar has to be left factored in so that the grammar is deterministic grammar.

- **Algorithm to construct LL(1) Parsing Table:**
- **Step 1:** First check all the essential conditions mentioned above and go to step 2.
- **Step 2:** Calculate First() and Follow() for all non-terminals.

1. **First():** If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.
2. Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.

- **Step 3:** For each production A –> α. (A tends to alpha)

1. Find First(α) and for each terminal in First(α), make entry A –> α in the table.
2. If First(α) contains ε (epsilon) as terminal, then find the Follow(A) and for each terminal in Follow(A), make entry A –> ε in the table.
3. If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> ε in the table for the $. To construct the parsing table, we have two functions:

- In the table, rows will contain the Non-Terminals and the column will contain the Terminal Symbols. All the **Null Productions** of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

Example of LL(1) Parser: Example 1

S → aABb

A → c | €

B → d | €

Step: 1: No left recursion in the grammar, hence no modification required.

Step 2: Calculation of First Set

First(S) = {a}

First(A) = {c, €}

First(B) = {d, €}

Example of LL(1) Parser

S → aABb

A → c | €

B → d | €

Step 3: Calculation of Follow Set

Follow(S) = {$}

Follow(A) = First(Bb) = First(B) = {d, €)

Since it contains €, continue FIRST rule.

First(Bb) = First(B) - € U First (b) = {d, b}

Follow(A) = {d,b}

Follow(B)= {b}

Example of LL(1) Parser

S → aABb ➔ First(aABb)= a

A → c | € ➔ First(c)=c and First(€) = € (Use follow)

B → d | € ➔ First(d)=d and First(€) = € (Use follow)

Step 4: Parsing Table

Grammar is LL(1)

Example String: acdb$** **H/w string: adb$

	a	b	c	d	$
S	S→aABb				
A		A→€	A→c	A→€	
B		B→€		B→d	

Example of LL(1) Parser

Example String: acdb$

	a	b	c	d	$
S	S→aABb				
A		A→€	A→c	A→€	
B		B→€		B→d	

Stack	Input	Action
$	acdb$	Push S into Stack
$S	acdb$	S→aABb
$bBAa	acdb$	Pop a
$bBA	cdb$	A→c
$bBc	cdb$	Pop c
$bB	db$	B→d
$bd	db$	Pop d
$b	b$	Pop b
$	$	Accept

Recursive descent parser

- Recursive-descent parsing is one of the simplest parsing techniques that is used in practice. Recursive-descent parsers are also called *top-down* parsers, since they construct the parse tree top down (rather than bottom up).
- The basic idea of recursive-descent parsing is to associate each non-terminal with a procedure. The goal of each such procedure is to read a sequence of input characters that can be generated by the corresponding non-terminal, and return a pointer to the root of the parse tree for the non-terminal. The structure of the procedure is dictated by the productions for the corresponding non-terminal.
- The procedure attempts to "match" the right hand side of some production for a non-terminal.
  - To match a terminal symbol, the procedure compares the terminal symbol to the input; if they agree, then the procedure is successful, and it consumes the terminal symbol in the input (that is, moves the input cursor over one symbol).
  - To match a non-terminal symbol, the procedure simply calls the corresponding procedure for that non-terminal symbol (which may be a recursive call, hence the name of the technique).

Backtracking

- In Top-Down Parsing with Backtracking, Parser will attempt multiple rules or production to identify the match for input string by backtracking at every step of derivation. So, if the applied production does not give the input string as needed, or it does not match with the needed string, then it can undo that shift.
- Example1 − Consider the Grammar

  S → a A d

  A → b c | b
- Make parse tree for the string a bd. Also, show parse Tree when Backtracking is required when the wrong alternative is chosen.
- **Solution**

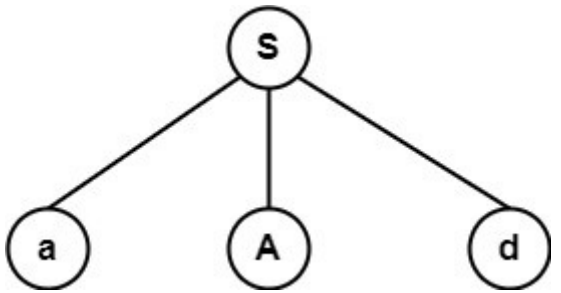
  The derivation for the string abd will be −

  S ⇒ a A d ⇒ abd (Required String)

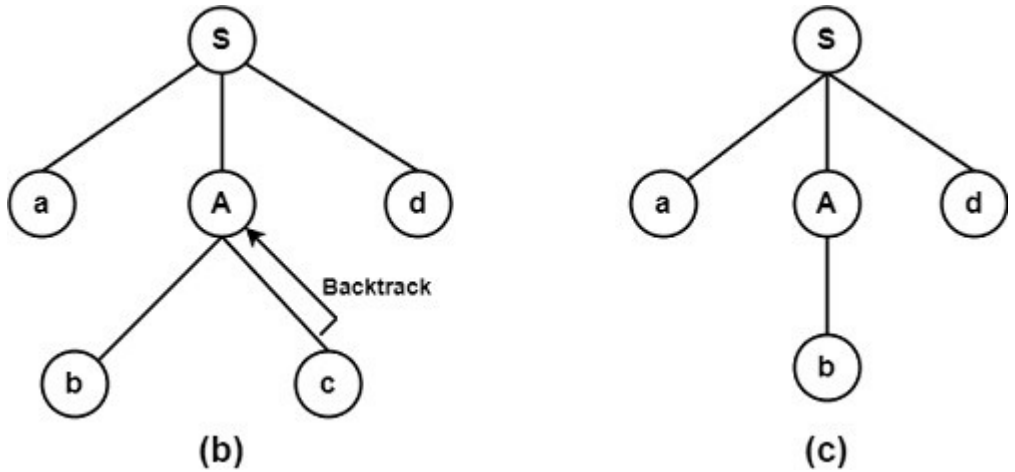
  If bc is substituted in place of non-terminal A then the string obtained will be abcd.

  S ⇒ a A d ⇒ abcd (Wrong Alternative)

  Figure (a) represents S → aAd



- Figure (b) represents an expansion of tree with production A → bc which gives string abcd which does not match with string abd. So, it backtracks & chooses another alternative, i.e., A → b in figure (c) which matches with abd.



Bottom up parsing

- A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).
- Start at the leaves and grow toward root
- We can think of the process as reducing the input string to the start
- symbol
- At each reduction step a particular substring matching the right-side of a
- production is replaced by the symbol on the left-side of the production
- Bottom-up parsers handle a large class of grammars

Bottom-Up Parsing

- Bottom-up parsing corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

$$
\begin{array}{lcll}
E & \rightarrow & E + T \mid T & \\
T & \rightarrow & T * F \mid F & (4.1) \\
F & \rightarrow & (E) \mid \mathbf{id} &
\end{array}
$$

- It is convenient to describe parsing as the process of building parse trees, although a front end may in fact carry out a translation directly without building an explicit tree.

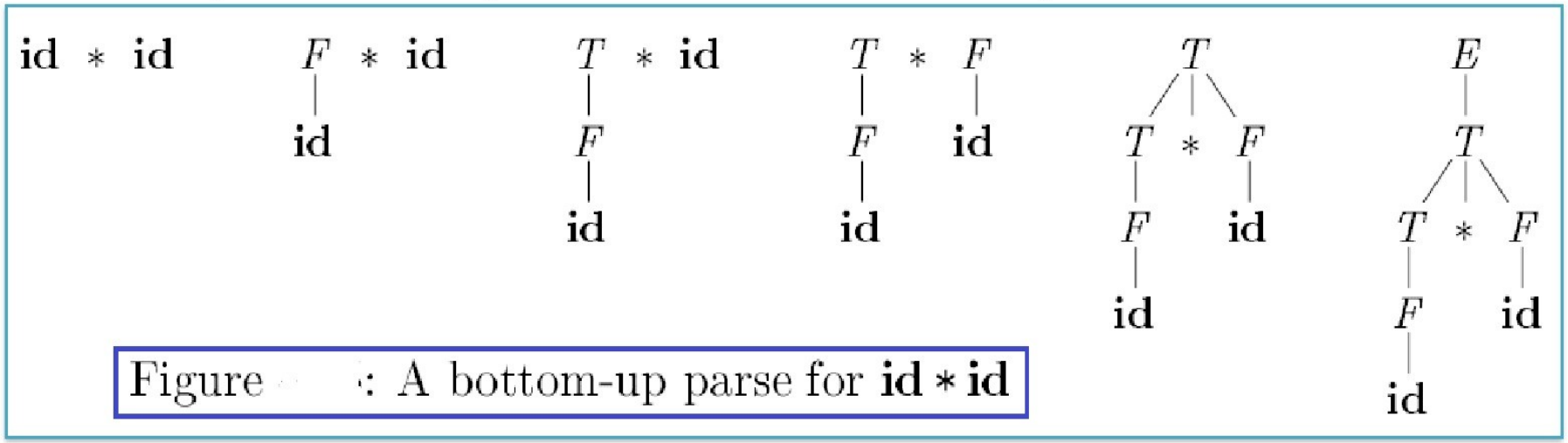


Figure : A bottom-up parse for **id * id**

Reductions

- We can think of bottom-up parsing as the process of "reducing" a string ***w*** to the start symbol of the grammar.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array} \qquad (4.1)
$$

- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

$$\mathbf{id} * \mathbf{id}, \quad F * \mathbf{id}, \quad T * \mathbf{id}, \quad T * F, \quad T, \quad E$$

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

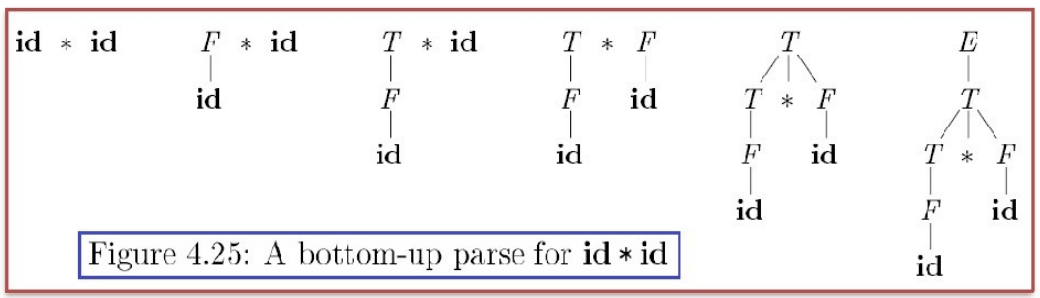


Figure 4.25: A bottom-up parse for $\mathbf{id} * \mathbf{id}$

HANDLE PRUNING

- A "handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation
- The handles during the parse of $\mathbf{id_1} * \mathbf{id_2}$

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

- The leftmost substring that matches the body of some production need not be a handle

Shift reduce parser

- Shift Reduce Parser is a type of Bottom-Up Parser. It generates the Parse Tree from Leaves to the Root. In Shift Reduce Parser, the input string will be reduced to the starting symbol. This reduction can be produced by handling the rightmost derivation in reverse, i.e., from starting symbol to the input string.
- Shift Reduce Parser requires two Data Structures
  - Input Buffer
  - Stack
- **Working-**
- Initially, shift-reduce parser is present in the following configuration where-
  - Stack contains only the $ symbol.
  - Input buffer contains the input string with $ at its end.
  - Moving the input symbols on the top of the stack.
  - Until a handle β appears on the top of the stack.
  -

- The parser keeps on repeating this cycle until-
  - An error is detected.
  - Or stack is left with only the start symbol and the input buffer becomes empty.

- After achieving this configuration,
  - The parser stops / halts.
  - It reports the successful completion of parsing.

- **Possible Actions-**
- A shift-reduce parser can possibly make the following four actions-

1. Shift-

- In a shift action,
- The next symbol is shifted onto the top of the stack.

2. Reduce-

- In a reduce action,
- The handle appearing on the stack top is replaced with the appropriate non-terminal symbol.

3. Accept-

- In an accept action,
- The parser reports the successful completion of parsing.

4. Error-

- In this state,
- The parser becomes confused and is not able to make any decision.
- It can neither perform shift action nor reduce action nor accept action.

Rules To Remember

- **Rule 1:** If the priority of the incoming operator is higher than the operator's priority at the top of the stack, then we perform the shift action.
- **Rule 2:** If the priority of the incoming operator is equal to or less than the operator's priority at the top of the stack, then we perform the reduce action.

- **Consider the following grammar-**
  - **E → E – E**
  - **E → E x E**
  - **E → id**
- **Parse the input string id – id x id using a shift-reduce parser.**
- **<u>Solution-</u>**
- The priority order is: id > x > –

Stack	Input Buffer	Parsing Action
$	id – id x id $	Shift
$ id	– id x id $	Reduce E → id
$ E	– id x id $	Shift
$ E –	id x id $	Shift
$ E – id	x id $	Reduce E → id
$ E – E	x id $	Shift
$ E – E x	id $	Shift
$ E – E x id	$	Reduce E → id
$ E – E x E	$	Reduce E → E x E
$ E – E	$	Reduce E → E – E
$ E	$	Accept

Operator precedence parsing

- Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.
- A grammar is said to be operator precedence grammar if it has two properties:
  - No R.H.S. of any production has a $\in$.
  - No two non-terminals are adjacent.
- There are the three operator precedence relations:
- a $\gtrdot$ b means that terminal "a" has the higher precedence than terminal "b".
- a $\lessdot$ b means that terminal "a" has the lower precedence than terminal "b".
- a $\doteq$ b means that the terminal "a" and "b" both have same precedence.

$$E \rightarrow EAE \mid (E) \mid -E \mid id$$

$$A \rightarrow + \mid - \mid x \mid / \mid \hat{} $$

Operator Precedence Grammar

$$E \rightarrow E + E \mid E - E \mid E \times E \mid E / E \mid E \hat{} E \mid (E) \mid -E \mid id$$

Operator Precedence Grammar

Precedence table:

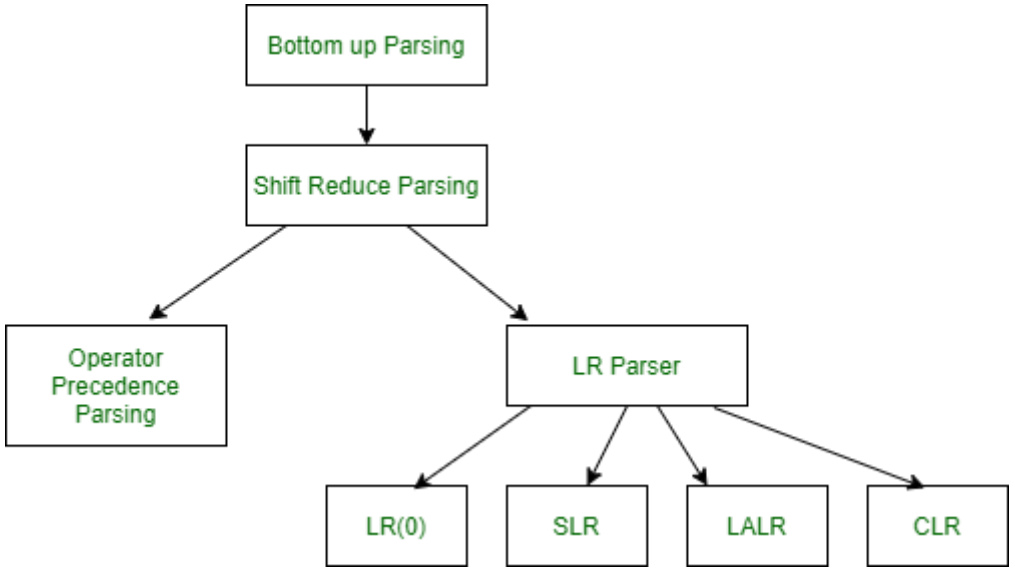
	+	*	(	)	id	$
+	⋗	⋖	⋖	⋗	⋖	⋗
*	⋗	⋗	⋖	⋗	⋖	⋗
(	⋖	⋖	⋖	≐	⋖	X
)	⋗	⋗	X	⋗	X	⋗
id	⋗	⋗	X	⋗	X	⋗
$	⋖	⋖	⋖	X	⋖	X

Contd….

- Step 1 : Check operator grammar or not.
- Step2 : Construct operator precedence relation table.
- Step 3 : Parse the given input string.
- Step 4 : Generate parse tree

LR parsing table

- LR parser is a bottom-up parser for context-free grammar that is very generally used by computer programming language compiler and other associated tools. LR parser reads their input from left to right and produces a right-most derivation. It is called a Bottom-up parser because it attempts to reduce the top-level grammar productions by building up from the leaves. LR parsers are the most powerful parser of all deterministic parsers in practice.



- **Description of LR parser :**
  The term parser LR(k) parser, here the L refers to the left-to-right scanning, R refers to the rightmost derivation in reverse and k refers to the number of unconsumed “look ahead” input symbols that are used in making parser decisions. Typically, k is 1 and is often omitted. A context-free grammar is called LR (k) if the LR (k) parser exists for it. This first reduces the sequence of tokens to the left. But when we read from above, the derivation order first extends to non-terminal.

1. The stack is empty, and we are looking to reduce the rule by S’ → S$.
2. Using a “.” in the rule represents how many of the rules are already on the stack.
3. A dotted item, or simply, the item is a production rule with a dot indicating how much RHS has so far been recognized. Closing an item is used to see what production rules can be used to expand the current structure. It is calculated as follows:

- **Rules for LR parser :**
  The rules of LR parser as follows.

1. The first item from the given grammar rules adds itself as the first closed set.
2. If an object is present in the closure of the form A → α. β. γ, where the next symbol after the symbol is non-terminal, add the symbol’s production rules where the dot precedes the first item.
3. Repeat steps (B) and (C) for new items added under (B).

- **LR parser algorithm :**
  LR Parsing algorithm is the same for all the parser, but the parsing table is different for each parser. It consists following components as follows.

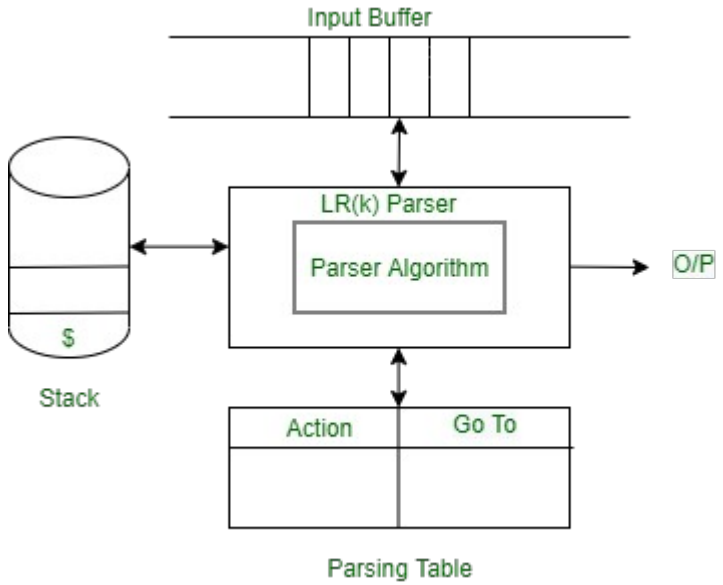
1. Input Buffer –
It contains the given string, and it ends with a $ symbol.

2. Stack –
The combination of state symbol and current input symbol is used to refer to the parsing table in order to take the parsing decisions.

- **Parsing Table** :
  Parsing table is divided into two parts- Action table and Go-To table. The **action table** gives a grammar rule to implement the given current state and current terminal in the input stream. There are four cases used in action table as follows.

1. Shift Action- In shift action the present terminal is removed from the input stream and the state ***n*** is pushed onto the stack, and it becomes the new present state.
2. Reduce Action- The number ***m*** is written to the output stream.
3. The symbol ***m*** mentioned in the left-hand side of rule ***m*** says that state is removed from the stack.
4. The symbol ***m*** mentioned in the left-hand side of rule **m** says that a new state is looked up in the goto table and made the new current state by pushing it onto the stack.

- **LR parser diagram :**



Syntax directed Definition

- Parser uses a CFG(Context-free-Grammar) to validate the input string and produce output for the next phase of the compiler. Output could be either a parse tree or syntax tree. Now to interleave semantic analysis with the syntax analysis phase of the compiler, we use Syntax Directed definition.

Semantic Analysis

- In syntax directed definition, along with the grammar it can identify some informal notations and these notations are known as semantic rules.

- After implementing the Semantic Analysis, the source program is modified to an intermediate form.
- There is some information that is required by the Intermediate code generation phase to convert the semantically checked parse tree into intermediate code. But this information or attributes of variables cannot be represented alone by Context- Free Grammar.
- So, some semantic actions or rules have to be attached with Context-Free grammar which helps the intermediate code generation phase to generate intermediate code.
- So, Attaching attributes to the variables of the context Free Grammar and defining semantic rules (meaning) of each production of grammar is called **Syntax Directed Definition**.
- It is a kind of notation in which each production of Context-Free Grammar is related with a set of semantic rules or actions, and each grammar symbol is related to a set of Attributes.
- Attributes may be number, strings, references, datatypes etc.

Production	Semantic Rule

Types of Attributes

1. **Synthesized Attributes :** If a node takes value from its children then it is synthesized attribute. A Synthesized attribute is an attribute of the non-terminal on the left-hand side of a production. Synthesized attributes represent information that is being passed up the parse tree. The attribute can take value only from its children (Variables in the RHS of the production). The non-terminal concerned must be in the head (LHS) of production. For e.g. let's say A -> BC is a production of a grammar, and A's attribute is dependent on B's attributes or C's attributes then it will be synthesized attribute.
2. **Inherited Attributes:** If a nodes takes its value from its parent or sibling. An attribute of a nonterminal on the right-hand side of a production is called an inherited attribute. The attribute can take value either from its parent or from its siblings (variables in the LHS or RHS of the production). The non-terminal concerned must be in the body (RHS) of production. For example, let's say A -> BC is a production of a grammar and B's attribute is dependent on A's attributes or C's attributes then it will be inherited attribute because A is a parent here, and C is a sibling.

Types of syntax directed definition

I. S-attributed SDD :

- If an SDD uses only synthesized attributes, it is called as S-attributed SDD.
- S-attributed SDDs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
- Semantic actions are placed in rightmost place of RHS.

II. L-attributed SDD:

- If an SDD uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings or parents node only, it is called as L-attributed SDD.
- Attributes in L-attributed SDDs are evaluated by depth-first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS.
- Example : S->ABC, Here attribute B can only obtain its value either from the parent – S or its left sibling A but It can’t inherit from its right sibling C. Same goes for A & C – A can only get its value from its parent & C can get its value from S, A, & B as well because C is the rightmost attribute in the given production.

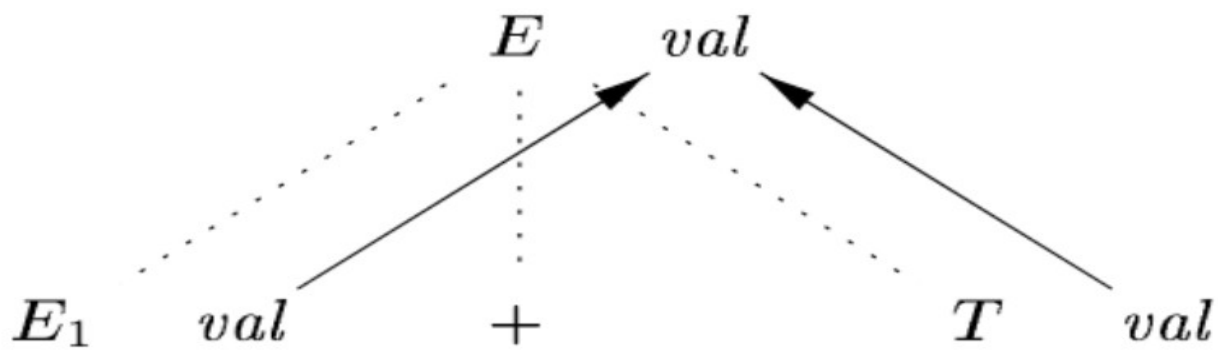
Dependency graph

- It represent the flow of information among the attributes in a parse tree.
- Dependency graph are useful for determining the evaluation order of attributes in parse tree.
- While an annotated parse tree shows the values of attributes, a dependency graph determines how those values can be computed.
- These are used to show the flow of information among attribute instances within a parse tree.
- In the graph, an edge from one attribute instance to another means that the value of the first attribute is required to compute the value of the second.
- Edges are used to express constraints that are implied by the language semantic rules.
- For each node in the parse tree, for example a node X, the dependency graph will have a node associated with X.
- If a semantic rule is associated with a production P defines the value of a synthesized attribute A.b in terms of the value of X.c, then the dependency graph is said to have an edge from X.c to A.b. In more detail, at every node N labeled A where this production P is applied, we create an edge to attribute b at N from attribute c at the child of N that corresponds to this instance of symbol X in the body of the production.
- If a semantic rule associated with a production P defines a value of an inherited attribute B.c in terms of the value X.a, the dependency graph contains an edge from X.a to B.c. For each node N that is labeled B and corresponds to an occurrence of B in the body of the production P, we create an edge to and attribute c at N from the attribute a at node M that corresponds to this occurrence of X. Keep in mind that M should be either the parent or a sibling.

An example:
We have the production;
$E \rightarrow E_1 + T$

And the semantic rule;
$E.val = E_1.val + T.val$.

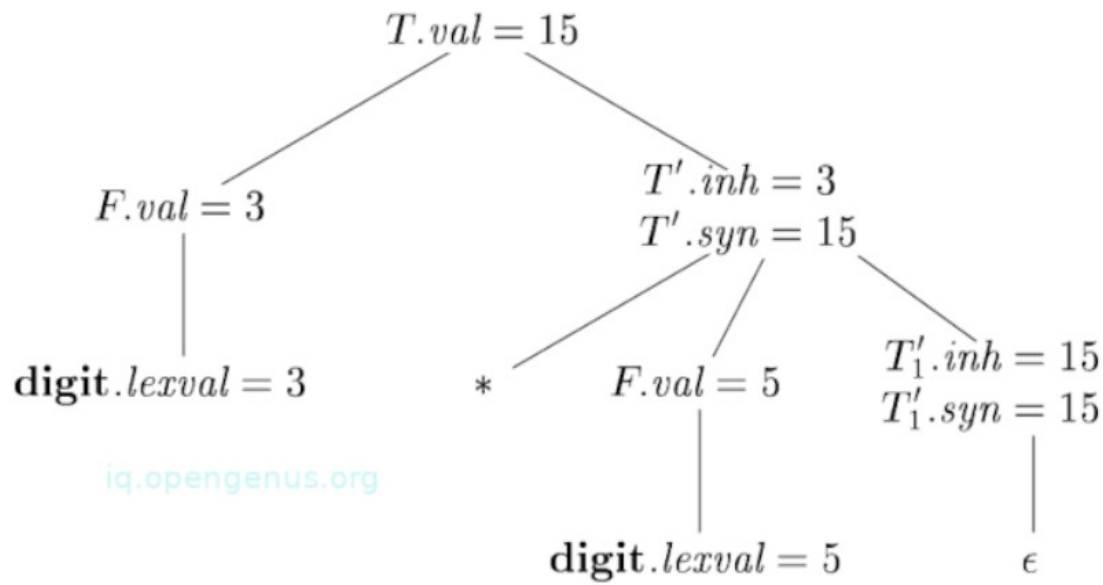
For every node *N* that is labeled *E*, that has children corresponding to the body of the production, the synthesized attribute val at *N* is computed using values of val at the two children *E* and *T*.
Therefore a section of the dependency graph for every parse tree where this production is used will look like the image below.



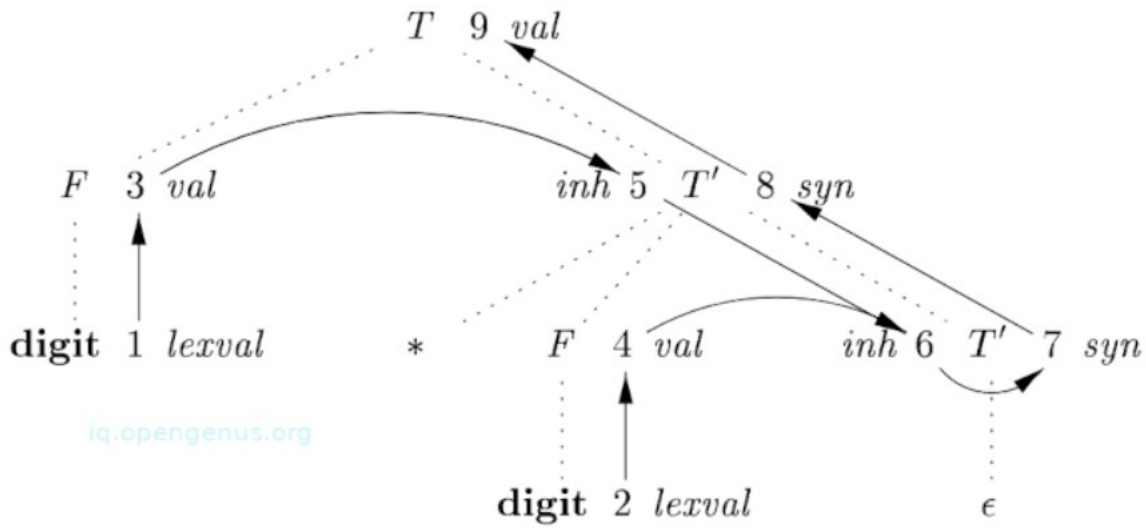
As a common convention we use dotted lines to represent the edges of the parse tree and solid lines to represent edges of the dependency graph.

An example:

We have the annotated parse tree for *3 * 5*.



Now its complete dependency graph.



Numbers *1-9* represent nodes of the graph and correspond to attributes in the annotated parse tree for *3 * 5*.

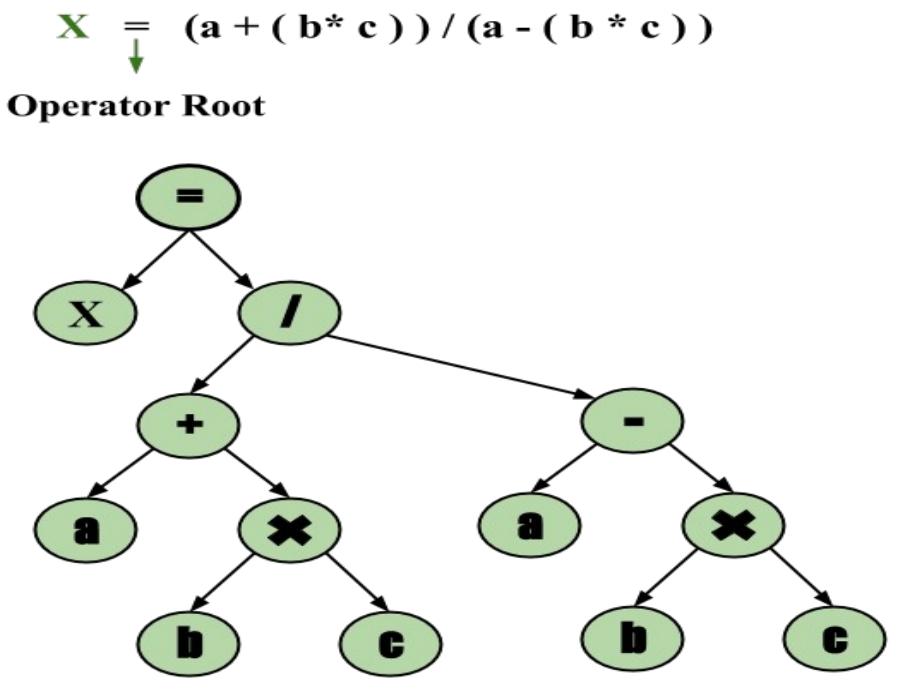
Nodes *1* and *2* represent the attribute *lexval* that is associated with the two leaves that are labeled *digit*.
Nodes *3* and *4* represent the attribute *val* that is associated with nodes labeled *F*.
Edges to node *3* from *1* and those from node *4* from *2* result from the semantic rule defining *F.val* in terms of *digit.lexval*.
As a matter of fact, *F.val* is equal to *digit.lexval* although the edges represent dependence and not equality.
5 and *6*, represent the inherited attribute T'.*inh* that is associated with every occurrence of the nonterminal T'.
Edge(5, 3) is a result of the rule T'.*inh* = *F.val* that defines T'.*inh* at the right child of the root from F.val at the left child.
Edges from node 5 to 6 for T'.*inh* and from node 4 for *F.val* since these value are multiplied to evaluate the attribute *inh* at node 6.

Nodes *7* and *8* represent a synthesized attribute *syn* associated with the occurrences of T'.
Edge from *6* to *7* is a result of the semantic rule 3 associated with production 3 from the SDD.

Intermediate code generator

- Intermediate codes are machine-independent codes, but they are close to machine instructions. The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.
- Forms of intermediate code :
  - Syntax tree or abstract syntax tree
  - Postfix notation
  - Three address code

- **Syntax Tree** Syntax tree is nothing more than condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.



Postfix notation

- In postfix notation, the operator comes after an operand, i.e., the operator follows an operand.
- Example
- Postfix Notation for the expression (a+b) * (c+d) is ab + cd +*
- Postfix Notation for the expression (a*b) - (c+d) is ab* + cd + -

Three address code

- The characteristics of Three Address instructions are-
- They are generated by the compiler for implementing Code Optimization.
- They use maximum three addresses to represent any statement.
- They are implemented as a record with the address fields.
- General Form-
- In general, Three Address instructions are represented as-
- a = b op c
- Here,
- a, b and c are the operands.
- Operands may be constants, names, or compiler generated temporaries.
- op represents the operator.

- **Example:** The three address code for the expression

  a + b * c + d

  T 1 = b * c

  T 2 = a + T 1

  T 3 = T 2 + d

- Where T 1 , T 2 , T 3 are temporary variables.