2/7/18.

Compiler:

It is a software which converts high level language into low level language.

High Level Language

↓

Pre-processor

↓

Compiler

↓

Assembler

↓

Loader | Linker

↓

Machine Language.

High Level Language: Human readable (C, C++, ...)

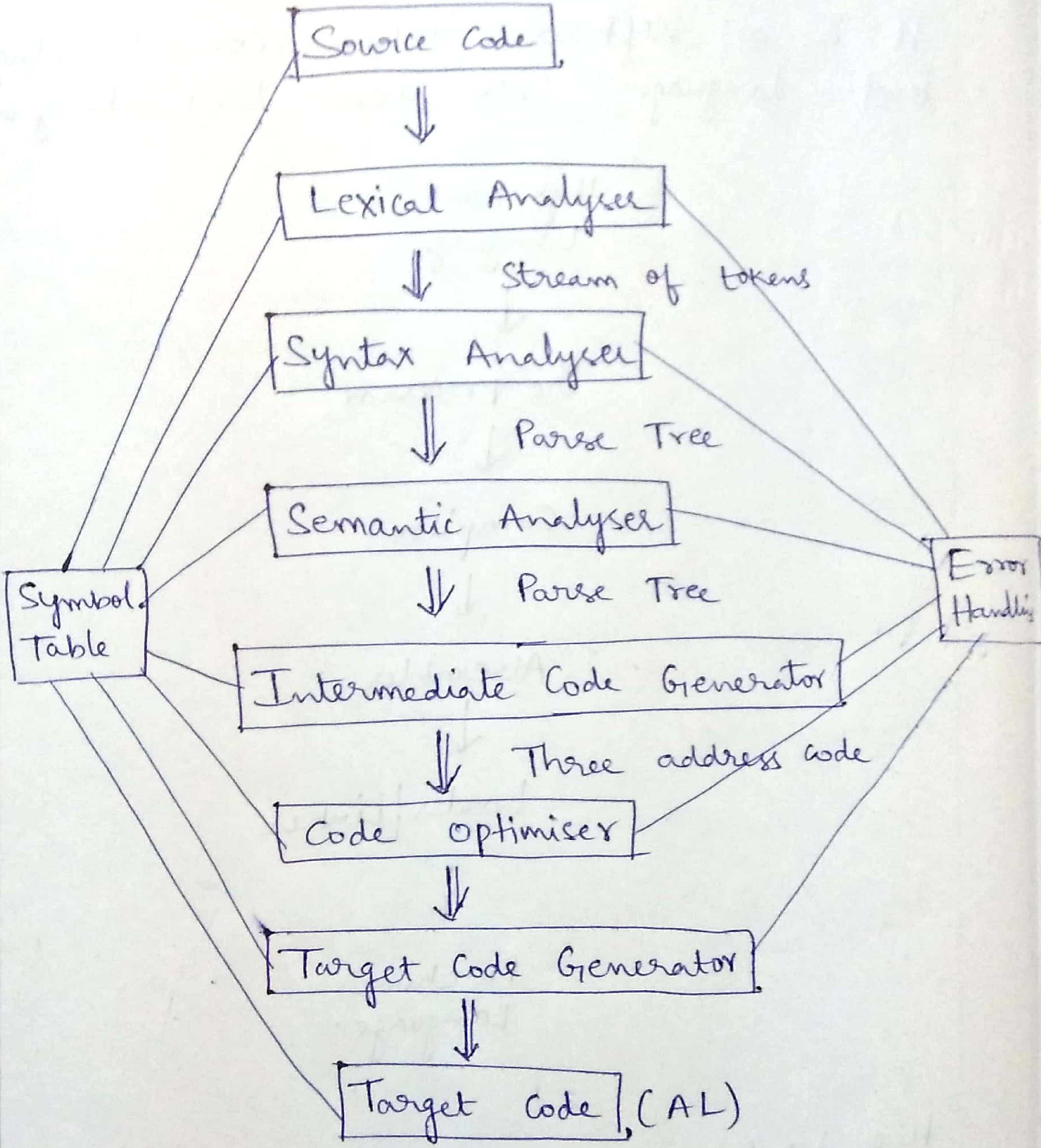
Pre-processor: Removes include files, provides macro expansion. Also sends pure HLL to the compiler as input.

Compiler: Converts the pure HLL to assembly lang. and sends as input.

Assembler: Converts assembly language to pure LLL.

Loader | Linker: Creates and loads addresses.

Phases of Compiler:



Lexical Analyser: Divides the expression into categories of tokens like identifiers, operators, - - and sends the tokens into Syntax Analyser as input.

Syntax Analyser: It will follow the grammar and converts to parse tree or constructing parse tree.

Semantic Analyser: Verifies whether the expression is meaningful or valid or not.

Intermediate Code Generator: It converts the expression into atmost 3-address code and sends as input to code optimiser.

Code optimiser: It tries to reduce the code for efficient use.

Target Code generator: It converts the input into assembly language and sends to assembler.

NOTE 1) Syntax analyser is the heart of compiler.

2) The first four phases are common for any compiler.

phases
- Front end (Phases 1,2,3,4)
- Back end (Phases 5,6)

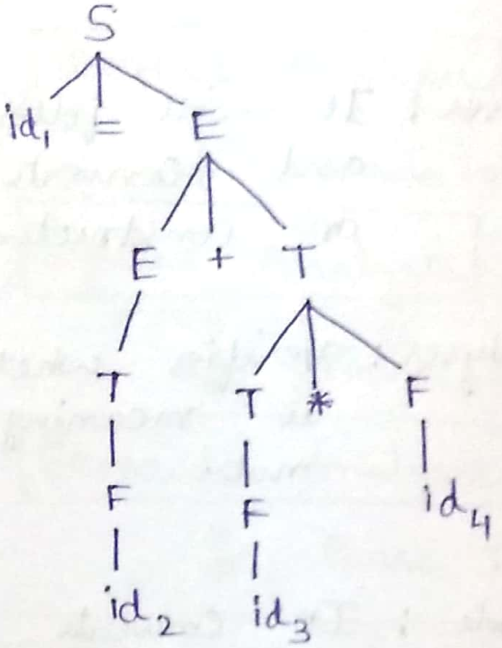
Ex: $x = a + b * c$.

$\Downarrow$

$id_1 \; op_1 \; id_2 \; op_2 \; id_3 \; op_3 \; id_4$

$id_1 = 'x'$ $id_3 = 'b'$
$op_1 = '='$ $id_4 = 'c'$
$id_2 = 'a'$ $op_3 = '*'$
$op_2 = '+'$

$\Downarrow$



Grammar :

$S \rightarrow id_1 \; op_1 \; E;$
$E \rightarrow (E \; op_2 \; T) | T$
$T \rightarrow (T \; op_3 \; F) | F$
$F \rightarrow id$

S = Statement
E = Expression
T = Term
F = Factor
id = identifier

$\Downarrow$

$id_1 = id_2 + id_3 * id_4$ (Meaningful)

$\Downarrow$

$T_1 = id_3 \; op_3 \; id_4$
$T_2 = id_2 + T_1$
$id_1 = T_2$.

$\Downarrow$

$T_1 = id_3 * id_4$
$id_1 = id_2 + T_1$

$\Downarrow$

mul R_2, R_3
add R_1, R_2

$id_2 = R_1$
$id_3 = R_2$
$id_4 = R_3$.

Rules to convert ambiguous → Unambiguous:

* Associativity rule
* Precedence rule

Grammar:

Grammar consists of four parts. They are:

(i) Set of tokens
(ii) set of non-terminals
(iii) Set of productions
(iv) Start symbol.

For example: $S \rightarrow PQ$
$P \rightarrow p$
$Q \rightarrow q$

In the above example; tokens = p, q
non-terminals = S, P, Q
productions : $S \rightarrow PQ$, $P \rightarrow p$
$Q \rightarrow q$
Start symbol = S.

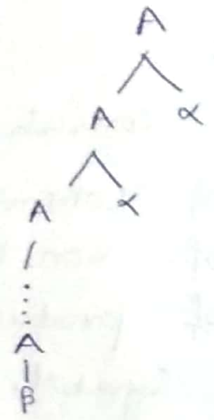
Types of Grammar:

- Grammar
  - → Based on ambiguity
    - → Ambigious
    - → Unambiguous
  - → Based on recursion
    - → Left Recursion
    - → Right Recursion
  - → Based on determinism
    - → Non-Deterministic
    - → Deterministic

Grammar based on recursion:

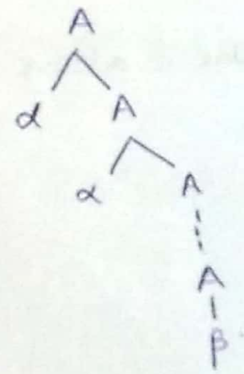
Left Recursion: Left most of RHS is equal to LHS in a given production

Ex: $A \rightarrow A\alpha / \beta$ $(\beta\alpha^*)$



Right Recursion: Right most of RHS is equal to LHS in a given production

Ex: $A \rightarrow \alpha A / \beta$ $(\alpha^*\beta)$.

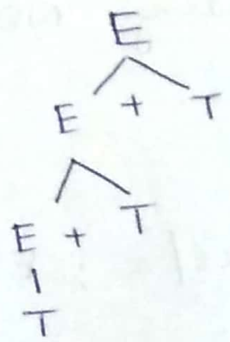


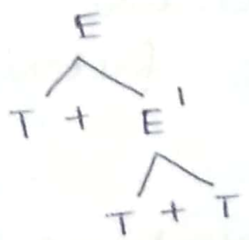
In the left recursion, the string derived is $\beta\alpha^*$ which is not understandable by compiler since the string doesn't end and compiler doesn't know when to stop. Hence we need to remove the infinite loop and there is a need of change in grammar. The grammar can be changed as

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

By using the changed grammar, we converted the left recursion to right recursion and in right recursion there is no scope of infinite loop. Hence, there is an end for a given string.

Ex: Convert $E \rightarrow E+T/T$ into Right recursion.





$\therefore$ RR grammar: $E \rightarrow TtE'$

$E' \rightarrow \epsilon / T+E'$

Ex: Convert $S \rightarrow SOSIS/01$ into right recursion

$S \rightarrow 01S'$

$S' \rightarrow \epsilon / OSISS'$

Non-Deterministic → Deterministic:

Let $A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3$ be an ND grammar.
Now, Converted D grammar looks like:

$$A \to \alpha A'$$
$$A' \to \beta_1 | \beta_2 | \beta_3 .$$

The procedure of converting ND → D is called as left factoring.

Ex: Convert $S \to iEtS \,|\, iEtSeS \,|\, a$; $E \to b$ into Deter.

$$S \to iEtS S' \,|\, a$$
$$S' \to eS \,|\, \epsilon$$
$$E \to b$$

i → if
E → Expression
t → then
S → statement
e → else
b → boolean.

For a given grammar to draw a parse tree, the below conditions must be satisfied

(i) Grammar is unambiguous
(ii) Grammar is right recursive
(iii) Grammar is deterministic.

NOTE: Applying left factoring doesnt remove the ambiguity in the grammar.

Ex: Convert $S \rightarrow aSSbS / aSaSb / abb / b$ into deterministic

$S \rightarrow aSS' / abb / b$

$S' \rightarrow SbS / aSb$

$\Downarrow$

$S \rightarrow aS' / b$

$S' \rightarrow SSbS / SaSb / bb$

$\Downarrow$

$S \rightarrow aS' / b$

$S' \rightarrow SS'' / bb$

$S'' \rightarrow SbS / aSb$

Ex: Convert $S \rightarrow bSSaaS / bSSaSb / bSb / a$ into D-form

$S \rightarrow bS' / a$

$S' \rightarrow SS'' /$

$S'' \rightarrow SaS''' / b$

$S''' \rightarrow aS / Sb$

(or)

$S \rightarrow bSS' / a$

$S' \rightarrow SaS'' / b$

$S'' \rightarrow aS / Sb.$

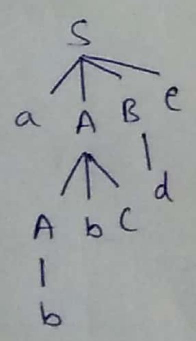
Parser Hierarchy:

- Parser
  - Top-Down Parser
    - with full Back tracking → Brute Force Method
    - without Back tracking
      - Recursive Descent
      - Non-Recursive Descent [LL(1)]
  - Bottom-Up parser
    - Operator precedence Parser (Allows Ambiguous) grammar.
    - LR-Parser
      - LR(0)
      - SLR(1)
      - LALR(1)
      - CLR(1)

NOTE: Bottoum-up parser is also called as SR-parser (Shift-Reduced parser)

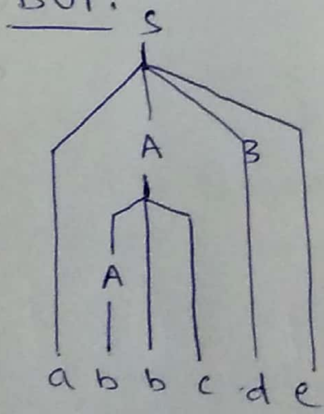
Ex: Consider the grammar $S \to aABe$, $A \to Abc|b$, $B \to d$. Construct parse trees using TDP, BUP

TDP:



$\Rightarrow$ abbcde.

BUP:



NOTE: TDP follows left most derivation
BUP follows reverse of right most derivation.

To draw a parse tree, we use two methods.

(i) First( )

(ii) Follow ( ).

First method is used to find the first terminal of a string. In a given production the left most terminal in yield side is the first of a production. If a non-terminal is present at left then select the first of non-terminal as first of production.

Follow method is used to find the follow terminal of a string. It finds the terminal following another terminal. The follow can be found in two methods. The first is to just blindly finding the follow by going through the productions. The second way is to find the first of the remaining string. i.e; Converting follow to first.

<u>Finding follow ( ):</u>

Step-1: where verify whether the production side of a production is present. If yes, then proceed through the second way. Else Keep "\$".

NOTE ⊛ First ( ) Can have ∈ while follow( ) Cant

Consider the productions below:

$S \rightarrow ABCD$

$A \rightarrow b / \epsilon$

$B \rightarrow c$

$C \rightarrow d / \epsilon$

$D \rightarrow e / \epsilon$.

Now, for the above productions;

$first(S) = \{b, c\}$

$first(A) = \{b, \epsilon\}$

$first(B) = \{c\}$

$first(C) = \{d, \epsilon\}$

$first(D) = \{e, \epsilon\}$

Now,

$follow(S) = \{\$\}$

$follow(A) = \{c\}$

$follow(B) = \{d, e, \$\}$

$follow(C) = \{e, \$\}$

$follow(D) = \{\$\}$.

<u>Explanation</u>:

$follow(B) = first(CD)$

$= first(CD) = d \cup first(D) = d \cup e \cup first(S)$

$= d \cup e \cup \$$

$= \{d, e, \$\}$.

Scanned by CamScanner

Ex: Consider the productions below:

$$S \rightarrow Bb \mid Cd$$
$$B \rightarrow aB \mid \epsilon$$
$$C \rightarrow cC \mid \epsilon$$

$first(S) = \{a, b, c, d\}$ $\quad$ $follow(S) = \{\$\}$

$first(B) = \{a, \epsilon\}$ $\quad$ $follow(B) = \{b\}$

$first(C) = \{c, \epsilon\}$ $\quad$ $follow(C) = \{d\}$

Ex: Consider the productions below:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \epsilon$$
$$F \rightarrow id \mid (E)$$

$first(E) = \{id, (\}$ $\quad$ $follow(E) = \{\$,)\}$

$first(E') = \{+, \epsilon\}$ $\quad$ $follow(E') = \{\$,)\}$

$first(T) = \{id, (\}$ $\quad$ $follow(T) = \{+, \$,)\}$

$first(T') = \{*, \epsilon\}$ $\quad$ $follow(T') = \{+, \$,)\}$

$first(F) = \{id, (\}$ $\quad$ $follow(F) = \{*, +, \$,)\}$

Constructing parse tree (Table):

Consider the above productions;

$E \rightarrow TE'$

$E' \rightarrow +TE' / \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \epsilon$

$F \rightarrow id / (E)$

Step-1: Draw a table with rows and Columns where no. of rows are equal to no. of variables and columns are the terminals. (Exclude ϵ)

Step-2: Select a variable from a row and choosing from first() of the variable write the production respectively i.e; for E; first (E) = {id, (} i.e; mention $E \rightarrow TE'$ in the column id, (and so on.

Step-3: If "ϵ" appears in the first, then go to the follow and mention the production yielding "ϵ" at corresponding follow columns. for Ex: $E' \rightarrow +TE' / \epsilon$. Mention the same production excluding null for the first() columns and the production $E' \rightarrow \epsilon$ for follow()

Parse Table:

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

Now, since the table is constructed, now parse tree has to be constructed for which we follow some algorithms, to derive string.

LL(1) Algorithm:

L → Scan string from left to right
L → Use left most derivation
1 → look ahead = 1 i.e; symbols used for decision.

LL(1) algorithm has 3 phases.

(i) Input Buffer
(ii) LL(1) parsing table
(iii) LL(1) parser.

Also there exists a stack with stack symbol as $. Whenever a production is added into stack, left most element has to be placed at top of stack.

LL(1) algorithm doesn't work when the entry in each cell is > 1.

Ex: Consider the production $S \rightarrow (S) / \epsilon$ and (())$ as string for drawing parse tree.

$$S \rightarrow (S) / \epsilon$$

$$\text{First}(S) = \{ (, \epsilon \}$$

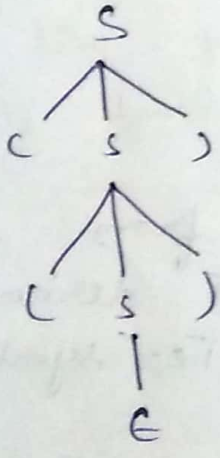
$$\text{Follow}(S) = \{ \$,) \}.$$

Parse table:

	(	)	$
S	$S \rightarrow (S)$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$.

Stack: | $\cancel{\$}$ | $\cancel{\$}$ | $\cancel{)}$ | $\cancel{\$}$ | $\cancel{(}$ | $\cancel{)}$ | $\cancel{\$}$ | $\cancel{(}$ |

Parse tree:



∴ Tree is drawn and the string is accepted by given grammar.

Recursive Descent parser:

Recursive i.e; for every variable, we write a function. For example, consider the grammar

$$E \rightarrow iE'$$
$$E' \rightarrow +iE' / \epsilon$$

Here Variables are E, E'. So, we write functions.

```
E()
{
    if(l=='i')
    {
        match('i');
        E'();
    }
}
```

```
E'()
{
    if(l=='+')
    {
        match('+');
        match('i');
        E'();
    }
    else
        return;
}
```

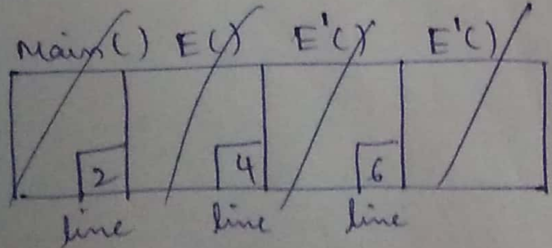
NOTE: Here l = look ahead, global variable; l = getchar()

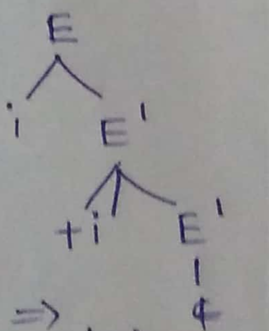
```
match(char t)
{
    if(l==t)
        l=getchar();
    else
        printf("Error");
}.
```

```
main()
{
    E();
    if(l=='$')
        printf("Success");
}.
```

Ex: i+i$

Stack is provided by OS





⇒ i+i

∴ Success

Operator precedence parser:

Consider the grammar $E \rightarrow E+E / E*E / id$

The above grammar can also be written as

$E \rightarrow EAE / id$

$A \rightarrow + / *$

From the above two, both represents same grammar and but first one is operator grammar second is not.

NOTE: While considering operating precedence parser, then the grammar must be operator grammar i.e; no two variables are adjacent to each other.

Ex: $E+E$, $E*E$, EaE are operator grammars.

Before parsing a table in this parser, we need to draw a table called operator relational table wherein table contains all the elements as rows and columns.

	id	+	*	$
id	–	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	–

Here, identifier is given highest preference and $ the least preference. Also, both the identifiers can't be compared. * has highest precedence in (+, *)

(+, +) => } Left Associativity i.e; row element
(*, *) => } is given highest precedence.

Scanned by CamScanner

<u>Ex:</u> E → E+E / E*E / id

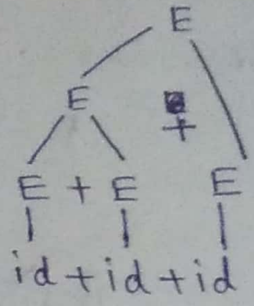
id + id * id $

<u>NOTE:</u> The top of the stack is less than or equ to look ahead i.e; push the look ahead and shift the cursor towards right. (Vice ve

Input	Stack	Relation
id + id * id $	$	($ < id)
id + id * id $	$ | id	(id > +)
id + id * id $	$ | ~~id~~	($ < +)
id + id * id $	$ | +	(+ < id)
id + id * id $	$ | + | id	(id > *)
id + id * id $	$ | + | ~~id~~	(+ < *)
id + id * id $	$ | + | *	(* < id)
id + id * id $	$ | + | * | id	(id > $)
id + id * id $	$ | + | * | ~~id~~	(* > $)
id + id * id $	$ | + | ~~*~~ | ~~id~~	(+ > $)
id + id * id $	$ | ~~+~~ | ~~*~~ | ~~id~~	($ $)

Finally parsed grammar is E+(E*E)

Consider id + id + id $.



$	~~id~~	~~*~~	~~id~~	~~*~~	~~id~~

$\therefore$ (E+E)+E

Disadvantages in ORT:

(i) Size / Memory

To overcome this disadvantage, we create an another table called operator function table using operator relational table.

Operator Function Table:

Step-1: Assign a variables f, g to row and column respectively.

Step-2: Mention all identifiers along with their corresponding variables in the form of a vertical line to map.

Step-3: Consider the first identifier and verify to all the columns and find the relation (>/<) from the ORT.

If relation => then $f \rightarrow g$

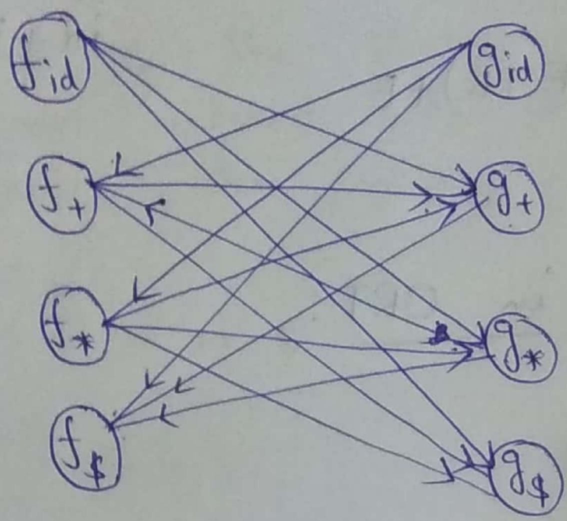
If relation =< then $f \leftarrow g$

Step-4: Mention the longest path for nodes. i.e; starting nodes which also gives for all other nodes and create OFT.

Scanned by CamScanner

For the above example,

$E \to E+E \,/\, E*E \,/\, id.$



longest path:

$f_{id} \to g_{*} \to f_{+} \to g_{+} \to f_{\$}$

$g_{id} \to f_{*} \to g_{*} \to f_{+} \to g_{+} \to f_{\$}$

Operator Function Table: (Max. Count)

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Space Complexity: ORT : $O(n^2)$
OFT : $O(2n)$

Ex: $P \rightarrow SR/S$

$R \rightarrow bSR/bs$

$S \rightarrow WbS/W$

$L \rightarrow id$

$W \rightarrow L*W/L$

$L \rightarrow$ Letter
$P \rightarrow$ paragraph
$S \rightarrow$ Sentence
$R \rightarrow$ Recursive Sentences
$b \rightarrow$ blank
$W \rightarrow$ word
$id \rightarrow$ identifier

The above grammar is not an operator grammar because two variables stay adjacent to each other i.e; SR. Now, we need to convert to operator grammar.

$P \rightarrow SbSR/Sbs/S$

$$\left.\begin{array}{l} P \rightarrow SbP/Sbs/S \\ S \rightarrow Wbs/W \\ L \rightarrow id \\ W \rightarrow L*W/L \end{array}\right\} \Rightarrow \text{operator grammar}$$