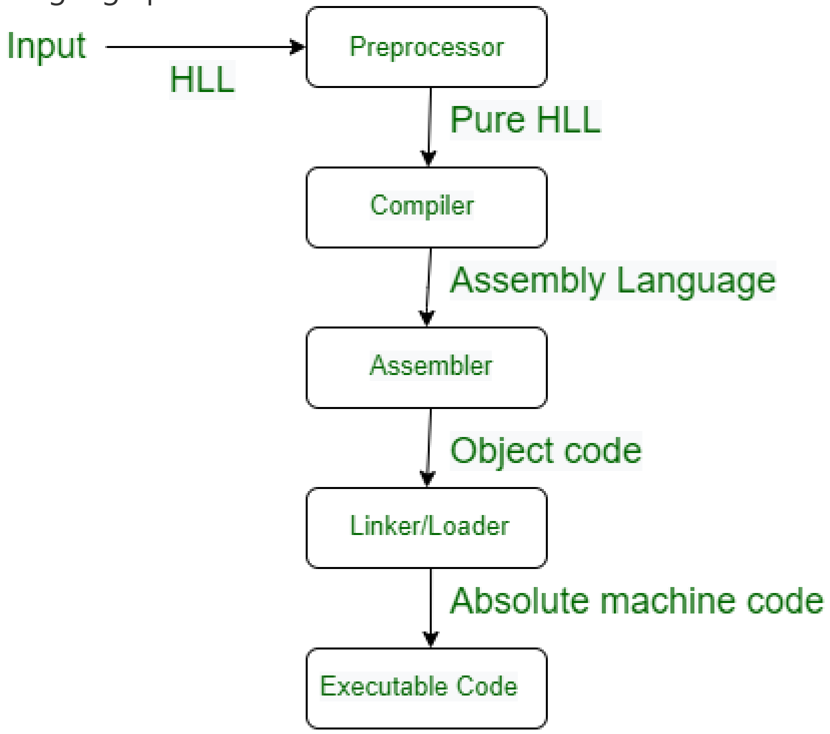
Compiler Design IMP Questions

UNIT-1

1. **Explain Language processing system in detail.**

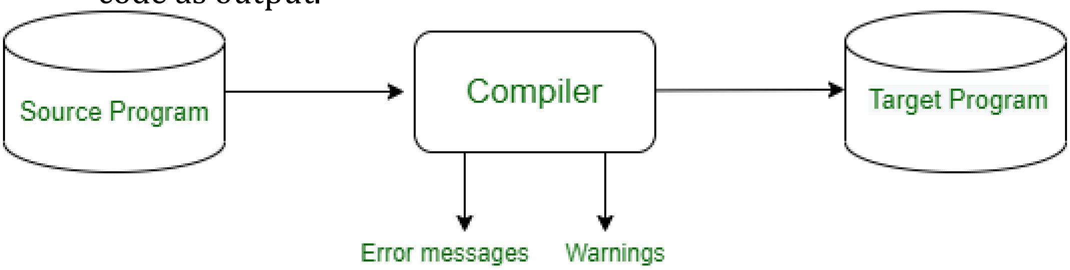
   A language processor is a special type of software program that has the potential to translate the program codes into machine codes. Languages such as COBOL and Fortran have language processors, which are generally used to perform tasks like processing source code to object code. A specific description of syntax, lexicon, and semantics of a high-level language is required to design a language processor.



Components of Language processing system :
Preprocessor –

- The preprocessor includes all header files. The preprocessor is also known as a macro evaluator, processing is optional that is if any language that does not support *#include* and *macros* processing is not required.

  Compiler –
  The compiler takes the modified code as input and produces the target code as output.



Input-Output

ASSEMBLER:

The assembler takes the target code as input and produces real locatable machine code as output.

Linker –

A linker or link editor is a program that takes a collection of objects (created by assemblers and compilers) and combines them into an executable program.

LOADER:

The loader keeps the linked program in the main memory.

- **Executablecode** –
  It is the low level and machine specific code and machine can easily understand. Once the job of linker and loader is done then object code finally converted it into the executable code.

2)Explain the various phases(structure) of a compiler in detail with an example.

Compiler Phases

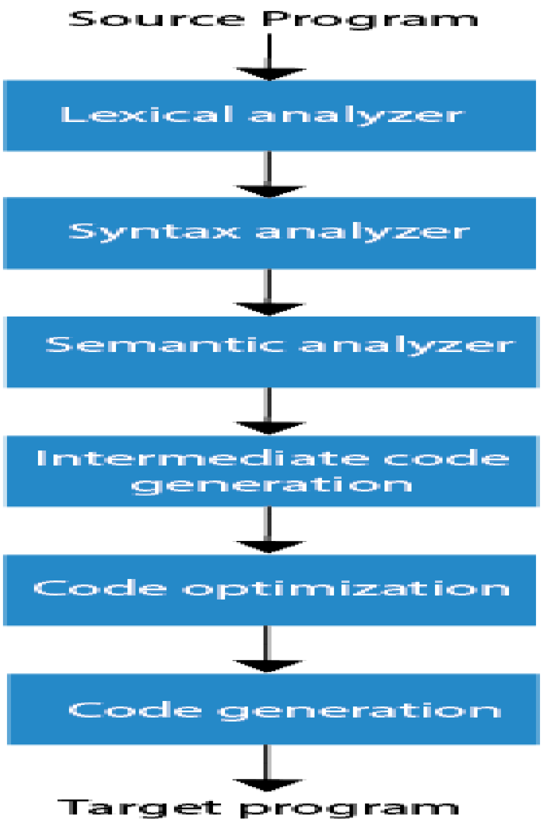
The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. Each phase takes input from its previous stage.

There are the various phases of compiler:

Fig: phases of compiler

Lexical Analysis:

Lexical analyzer phase is the first phase of compilation process. It takes source code as input. It reads the source program one character at a time and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens.



Syntax Analysis

Syntax analysis is the second phase of compilation process. It takes tokens as input and generates a parse tree as output. In syntax analysis phase, the parser checks that the expression made by the tokens is syntactically correct or not.

Semantic Analysis

Semantic analysis is the third phase of compilation process. It checks whether the parse tree follows the rules of language. Semantic analyzer keeps track of identifiers, their types and expressions. The output of semantic analysis phase is the annotated tree syntax.

Intermediate Code Generation

In the intermediate code generation, compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language. The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

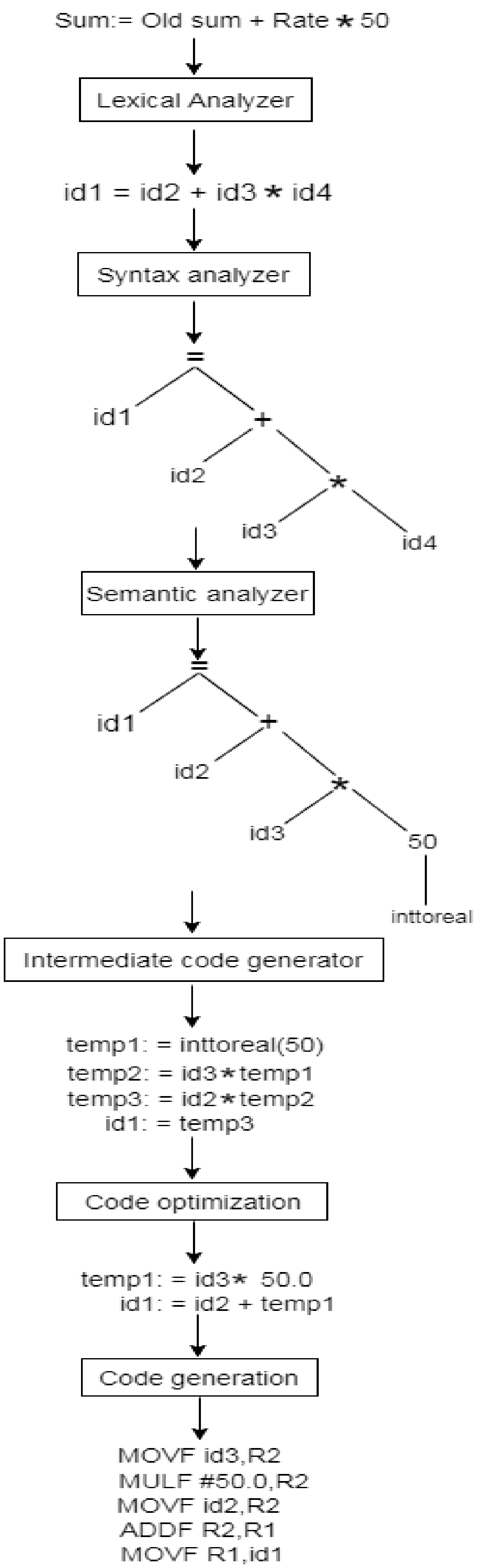
Code Optimization

Code optimization is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

Code Generation

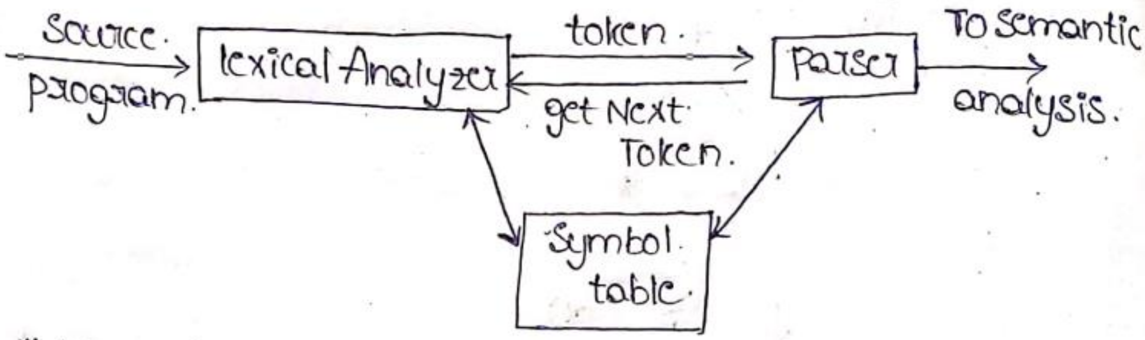
Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language. Code generator translates the intermediate code into the machine code of the specified computer.

Example:


Sum:= Old sum + Rate ∗ 50
Lexical Analyzer
id1 = id2 + id3 ∗ id4
Syntax analyzer
=
id1
+
id2
*
id3
id4
Semantic analyzer
=
id1
+
id2
*
id3
50
inttoreal
Intermediate code generator
temp1: = inttoreal(50)
temp2: = id3∗temp1
temp3: = id2∗temp2
id1: = temp3
Code optimization
temp1: = id3∗ 50.0
id1: = id2 + temp1
Code generation
MOVF id3,R2
MULF #50.0,R2
MOVF id2,R2
ADDF R2,R1
MOVF R1,id1

3)Explain Role of Lexical Analyzer

Role of lexical Analyzer :-



* Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.

* The main role of lexical analyzer is whenever the parser wants a token it will send 'get Nex Token' command to the lexical analyzer.

* After receiving 'get Next Token' command, the lexical Analyzer reads the source program characters until it identifi-es the next Token.

4)Differences b/w Lexical analyzer and Parser.

lexical analysis	VS parsing
* output (tokens)	* output (parse tree)
* Also recognized as Scanner	* Also recognized as parser.
* Buffering Techniques is used.	* parsing Techniques is used. (Topdown (or) bottom up)
* Design for finite automata. & regular expressions	* Design for context free grammars
* LEX Tools.	* YACC Tools.

5)Explain Token, Lexeme, Pattern with an example.

Token: It is a sequence of characters having a collection meaning. It is generated when lexeme is matched against pattern.

pattern: The rule associated with each set of strings is called pattern.

(or)

A pattern is a rule describing the set of lexemes that can represent a particular token. in the small program.

lexeme:

A lexeme is a sequence of characters in the source program that matches the pattern for a token

EXAMPLE:

Examples:

Token	pattern	lexeme
const	const	const
if	if	if
relation	< or <= or >(or)<>(or) >= or >	<, <=, =, <>, >, >=
id	letter followed by letters and digits.	Pi, count, D2
num.	any numeric constant	3.1416, 0, 6.02E23.
literal.	any characters b/w " and " except "	" core dumped "

* write lexeme, token, and pattern for the following code

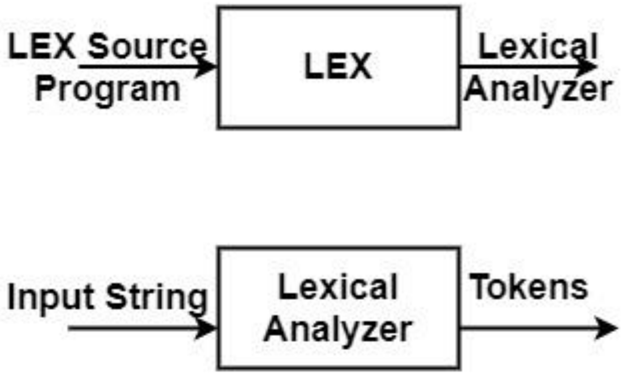
```
void Swap(int i, int j)
{
    int t;
    t=i;
    i=j;
    j=t;
}
```

lexeme.	Token	pattern.
void	keyword.	It is predefined word.
Swap.	identifiers.	
(	operator	
int	keyword.	
i	identifier.	
,	operator	
)	operator	
{	operator	
int	keyword.	
t	identifier	
;	operator	
t	identifier.	
=	operator	
i	identifier	
j	identifier	

Scanned with CamScanner

6. **Explain Lex Tool in detail with example.**

It is a tool or software which automatically generates a lexical analyzer (finite Automata). It takes as its input a LEX source program and produces lexical Analyzer as its output. Lexical Analyzer will convert the input string entered by the user into tokens as its output.



LEX Tool

... sequence of tokens

Structure of lex program:

Lex program is separated into three section.

→ declaration Selection.

%%

→ translation rules → pattern {action}

%%

→ auxiliary functions

Declaration Section:-

* Declarations of ordinary C variables, constants and libraries.
* The declarations are surrounded by

```
%{
-----
%}
```

Example :-

```
%{
    #include<math.h>
    # include <stdio.h>
    int count = 0;
%}
```

Translation Rules:-

The form of rules are:

```
%{ pattern {action }
    P1 {action1}
    P2 {action2}
    
    Pn {action n}
%}
```

4) write a program to count a n.o of lines & n.o of ~~words~~ words.

```
%{
#include<stdio.h>
%} int numwords=0, numlines=0;
%%
\n          {numlines++;}
[\t \n]     {numwords++;}
%%
int main()
{
 yylex();
 printf("%d\t%d\n", numwords, numlines);
}
```

o/p
2 6

Test:
This is a line
compiler design

UNIT-2

1. **What is meant by left most derivation (LMD) and right most derivation (RMD) and parse tree. Discuss with an example.**

1. Leftmost Derivation:

In the leftmost derivation, the input is scanned and replaced with the production rule from left to right. So in leftmost derivation, we read the input string from left to right.

Example:

Production rules:

1. E = E + E
2. E = E - E
3. E = a | b

To show:

1. a - b + a

The leftmost derivation is:

1. E = E + E
2. E = E - E + E

3. E = a - E + E
4. E = a - b + E
5. E = a - b + a

2. Rightmost Derivation:

In rightmost derivation, the input is scanned and replaced with the production rule from right to left. So in rightmost derivation, we read the input string from right to left.

Example

Production rules:

1. E = E + E
2. E = E - E
3. E = a | b

To show:

1. a - b + a

The rightmost derivation is:

1. E = E - E
2. E = E - E + E
3. E = E - E + a
4. E = E - b + a
5. E = a - b + a

Parse tree

- Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.

The parse tree follows these points:

- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

Example:

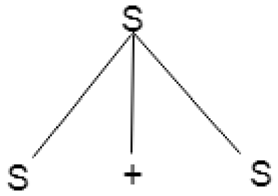
Production rules:

1. T= T + T | T * T
2. T = a|b|c

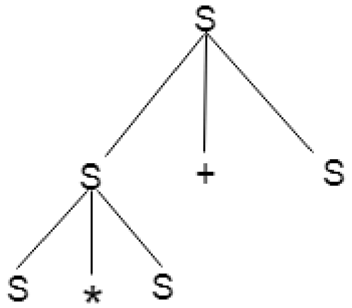
Input:

a*b+c

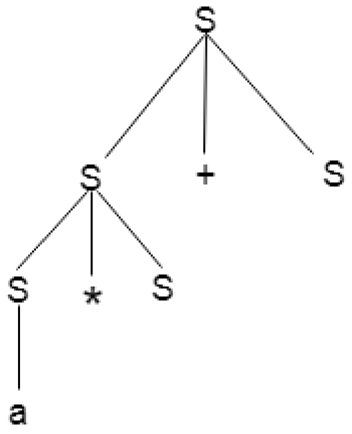
Step 1:



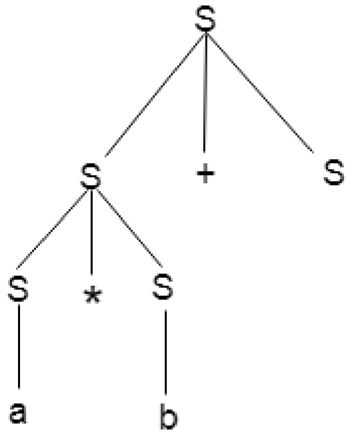
Step 2:



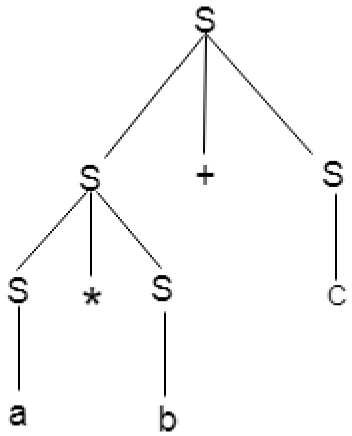
Step 3:



Step 4:



Step 5:



2)What is ambiguous grammar? Give an example.

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.

EXAMPLE:

Check whether the given grammar G is ambiguous or not.

1. E → E + E
2. E → E - E
3. E → id

Solution:

From the above grammar String "id + id - id" can be derived in 2 ways:

First Leftmost derivation

1. E → E + E
2. → id + E

3. → id + E - E
4. → id + id - E
5. → id + id- id

Second Leftmost derivation

1. E → E - E
2. → E + E - E
3. → id + E - E
4. → id + id - E
5. → id + id - id

3)Eliminate Left Recursion, Left Factoring.

Left Recursion:-

A grammar is said to be left recursive, if the grammar is of the form

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

$\downarrow$ eliminate immediate left recursion

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$ an equivalent grammar

In general,

$A \rightarrow A\alpha_1 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \cdots \mid \beta_n$

$\downarrow$ Immediate left recursion (elimination)

$A \rightarrow \beta_1 A' \mid \cdots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$

an equivalent grammar.

Immediate left recursion Example:- (8)

1. $E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow id \mid (E)$

↓ eliminate immediate left recursion.

$E \rightarrow E+T \mid T$
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow T*F \mid F$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$

Required Grammar is

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

2. $A \rightarrow Aa \mid bd$
$A \rightarrow bdA'$
$A' \rightarrow aA' \mid \epsilon$

3. $S \rightarrow (L) \mid a$
$L \rightarrow L, S \mid S$ $\Rightarrow$ $S \rightarrow (L) \mid a$
$L \rightarrow SL'$
$L' \rightarrow ,SL' \mid \epsilon$

4. $S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$
⇓
$S \rightarrow Aa \mid b$
$A \rightarrow bdA' \mid A'$
$A' \rightarrow cA' \mid adA' \mid \epsilon$

Left - Factoring :- (9)

- A grammar is said to be left factoring, if there is a production.

$$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$$

After eliminating Left Factoring

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

- In general,

$$A \rightarrow \alpha \beta_1 \mid \dots \mid \alpha \beta_n \mid \gamma_1 \mid \dots \mid \gamma_m$$

convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \dots \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid \dots \mid \beta_n$$

Problem-02:

Do left factoring in the following grammar-

$$A \rightarrow aAB \;/\; aBc \;/\; aAc$$

Solution-

Step-01:

$$A \rightarrow aA'$$

$$A' \rightarrow AB \;/\; Bc \;/\; Ac$$

Again, this is a grammar with common prefixes.

Step-02:

$$A \rightarrow aA'$$

$$A' \rightarrow AD \;/\; Bc$$

$$D \rightarrow B \;/\; c$$

This is a left factored grammar.

4)Recursive Descent parser.

Recursive Descent Parsing:-

- A parser that uses collection of recursive procedures for parsing the given input string is called Recursive Descent parser.
- Each non-terminal corresponds to a procedure.
- Execution begins with the procedure for start symbol.

Example – Write down the algorithm using Recursive procedures to implement the following Grammar.

E → TE'

$E' \rightarrow +TE'$

$T \rightarrow FT'$

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

$F \rightarrow (E) | id$

Solution

Procedure E ( )

{

T ( );

E′( );

}

$E \rightarrow TE'$

Procedure E′( )

{

If input symbol ='+' then

$E \rightarrow + TE'$

advance( );

T ( );

E′( );

}

```
Procedure T()
{
        F ( );
        T'();
}
```

$T \rightarrow F\,T'$

```
Procedure T'()
{
        If input symbol ='*' then
        advance( );
        F ( );
        T'();
}
```

$T' \rightarrow * \; FT'$

```
Procedure F ( )
{
        If input symbol ='id' then
        advance ( );
        else if input-symbol ='(' then
        advance ( );
        E ( );
        If input-symbol = ')'
        advance ( );
        else error ( );
        else error ( );
}
```

$F \rightarrow id$

$F \rightarrow (E)$

6. **Write the rules for calculating FIRST and FOLLOW**

Rules for calculating FIRST

- If x is a terminal symbol then $FIRST(x) = \{x\}$
- If $X \rightarrow \epsilon$ is a production then $FIRST(x) = \{\epsilon\}$
- If x is a non-terminal symbol and $x \rightarrow Y_1 Y_2 Y_3$ is a production.

$FIRST(X) = \{Y_1\}$ if Y_1 is a terminal.

otherwise

$FIRST(X) = FIRST\{Y_1 Y_2 Y_3\} = FIRST(Y_1)$, if $FIRST(Y_1)$ doesn't contains ϵ.

Scanned with CamScanner

If $FIRST(Y_1)$ contains ϵ then

$FIRST(X) = FIRST\{Y_1 Y_2 Y_3\} = FIRST(Y_1) - \{\epsilon\}$

$\cup \; FIRST(Y_2 Y_3)$.

$FIRST(Y_2 Y_3)$ is computed in similar manner.

Rules for calculating FOLLOW (for non-terminals)

- If S is the start symbol then add \$ to FOLLOW(S)
- If $A \rightarrow \alpha B \beta$ is a production then

$FOLLOW(B) = FIRST(\beta)$ if $FIRST(B)$ doesn't contains ϵ.

if $FIRST(\beta)$ contains ϵ then add $FOLLOW(A)$ to $FOLLOW(B)$ i.e., $FOLLOW(B) =$

$FIRST(\beta) - \{\epsilon\} \cup FOLLOW(A)$.

- If $A \rightarrow \alpha B$ is a production

$FOLLOW(B) = FOLLOW(A)$.

We apply these rules until nothing more can be added to any follow set.

7. Problems on Predictive parsing table or LL(1) parsing table.

Constructing Predictive Parsing Table -- (23)

Algorithm:-

1. For each production $A \rightarrow \alpha$ do steps 2, 3 and 4
2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to M[A, a].
3. If ϵ in FIRST(α), add $A \rightarrow \alpha$ to M[A, b] where b is the terminal in FOLLOW(A).
4. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, \$].

- All other undefined entries of the parsing table are error entries.

2. Construct Predictive Parsing Table for the following grammar.

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b$$

Sol: After removing left-factoring

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow \epsilon \mid eS$$
$$E \rightarrow b$$

Non-terminals - $\{S, S', E\}$

Terminals - $\{i, t, a, e, b\}$

Calculating First Function:-

$$\text{FIRST}(S) = \text{FIRST}(iEtSS') \cup \text{First}(a)$$
$$= \{i\} \cup \{a\}$$
$$= \{a, i\}$$

$$\text{FIRST}(S') = \text{FIRST}(eS) \cup \text{FIRST}(\epsilon)$$
$$= \{e, \epsilon\}$$

$$\text{FIRST}(E) = \text{FIRST}(b)$$
$$= \{b\}$$

Calculating Follow Function :-

29

(i) FOLLOW(S) = {$}

Using $S' \rightarrow eS$

= FOLLOW(S')

= {e, $}

Using $S \rightarrow iEtSS'$

= FIRST(S')

= {e, ε} − {ε} ∪ FOLLOW(S)

= {e, $}

(ii) Using $S \rightarrow iEtSS'$

FOLLOW(S') = FOLLOW(S)

= {e, $}

(iii) Using $S \rightarrow iEtSS'$

FOLLOW(E) = FIRST(tSS')

= {t}

Constructing Predictive Parsing Table :-

not LL(1) because single entries

	i	t	a	e	b	$
S	$S \rightarrow iEtSS'$		$S \rightarrow a$			
S'				$S' \rightarrow \epsilon$ $S' \rightarrow eS$		$S' \rightarrow \epsilon$
E					$E \rightarrow b$	

$S \rightarrow iEtSS'$ FIRST(iEtSS') = {i}

→ $S \rightarrow iEtSS'$ into M[S, i]

$S \rightarrow a$ FIRST(a) = {a} → $S \rightarrow a$ into M[S, a]

$S' \rightarrow \epsilon$ FIRST(ε) = {ε} − {ε} ∪ FOLLOW(S') = {e, $} → $S' \rightarrow \epsilon$ into M[S', e] and M[S', $]

31

$S' \rightarrow eS$ FIRST(eS) = {e} → $S' \rightarrow eS$ into M[S', e]

$E \rightarrow b$ FIRST(b) = {b} → $E \rightarrow b$ into M[E, b]

UNIT-3

1. **Compare Top down parsing and Bottom up parsing with example.**

Top Down Parsing	Bottom Up Parsing.
1. Process starts with root	1. Process starts with leaves.
2. It starts with starting symbol of the grammar.	2. It ends with starting symbol of the grammar.
3. This parsing technique uses Left Most Derivation.	3. This parsing technique uses Right Most derivation.
4. It is categorise by recursive descent parser and predictive parser.	4. It is categorise by Operator precedence parser and shift reduce parser.
5. It is not accepting Ambiguous Grammar.	5. It accepts ~~Ambiguous~~ grammar.
6. It is less powerful when compared to bottom-up parser.	6. It is more powerful as compare to top down parser.
7. It is simple to produce parser.	7. It is difficult to produce parser.
8. It uses LL(1) grammar to perform parsing.	8. It uses SLR, CLR, LALR grammar to perform parsing.
9. Error detection is weak.	9. Error detection is strong.

EXAMPLE(TOP-DOWN):

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

Sol: $\omega = id + id * id$

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

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

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

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

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

Example(BOTTOM -UP):

Example:

$S \rightarrow aABb$ input string: aaabb

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

$a\,a\,\underline{a}\,b\,b$

$a\,\underline{a\,A}\,b\,b$ $[A \rightarrow a]$

$a\,A\,\underline{b}\,b$ $[A \rightarrow aA]$

$a\,A\,B\,b$ $[B \rightarrow b]$

S $[S \rightarrow aABb]$.

Thus the start symbol S is obtained.

String is accepted.

2. **Explain handle and handle pruning**.

Handle and Handle Pruning:-

- A Handle is a substring of the string that matches the right side of a production rule.
  - But not every substring matches the right side of a production rule is handle.

- Handle Pruning is reducing the handle with non-terminal of the left side of the production. A right-most derivation in reverse can be obtained by handle pruning.

Example:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid Id$

Right-Most Sentential Form	Reducing Production
id + id * id	F → id
F + id * id	T → F
T + id * id	E → T
E + id * id	F → id
E + F * id	T → F
E + T * id	F → id
E + T * F	T → T * F
E + T	E → E + T
E	

Handles are red and underelined in the right-sentential forms.

Right most derivation of id+id*id

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

3. Explain shift reduce parsing with an example

Shift Reduce Parsing:-

- There are four possible actions (operations) of a shift reduce parser.

1. Shift: In shift action, parser push input symbols onto the stack.
2. Reduce: In reduce action, the parser replace the handle with non-terminal.
3. Accept: In this action, parser announces successful completion of parsing.
4. Error: In this action, parser identifies syntax errors.

Stack implementation of Shift Reduce Parser

- It contains stack and input buffer.
- Stack contains Grammar symbols and input buffer is used to store the string to be parsed.
- Initial stack contains empty and output buffer contains input string is marked by the end-marker $.

Stack	input
$	w$

1. Shift reduce parser push zero or more input symbols onto the stack until a handle on top of the stack.
2. Once there is a handle on the top of stack then parser replace handle with non terminal.
3. The parser repeats this process until stack top contains start symbol and Input buffer is empty. or until it has detected an error.

Stack	input
$S	$

2. Consider the following Grammar

$S \to TL;$

$T \to int \mid float$

$L \to L, id \mid id$

Parse the input sting int id, id;

Stack	Input	Action
$	int id,id;$	shift int
$int	id,id;$	reduce by $T \to int$
$T	id,id;$	shift id
$T id	,id;$	reduce by $L \to id$
$T L	,id;$	shift ,
$T L,	id;$	shift id
$TL,id	;$	reduce $L \to L,id$
$ TL	;$	shift ;
$ TL;	$	reduce $S \to TL;$
$ S	$	String accepted.

$S \to TL;$

$\Rightarrow T\, L, id;$

$\Rightarrow T\, id, id;$

$\Rightarrow int\ id, id;$