6.6(b)

→ Based on LR Parsing for the following grammar show all steps for the stack, input and Action.

Ans: Given Grammar,

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

Q (6.b)

→ Based on LR Parsing, fore the following grammer show all steps fore stack input and Action. [id+id*id]

step	stack	input	Action.
1		id+id*id $	shift id
2	id	+id*id $	Reduce F→id
3	F	+id*id $	Reduce T→F
4	T	+id*id $	Reduce E→T
5	E	+id*id $	shift +
6	E+	id*id $	shift id
7	E+id	*id $	Reduce F→id
8	E+F	*id $	Reduce T→F
9	E+T	*id $	shift *
10	E+T*	id $	shift id
11	E+T*id	$	Reduce F→id
12	E+T*F	$	Reduce T→T*F
13	E+T	$	Reduce E→E+T
14	E	$	Accepted.

Target: Rules গুলোর মধ্যে Reduce করে।

6.a Describe Algorithm for LR Parser.

Ans: Algorithm:

1. Initialize
   stack ← [$] // $ is the initial state
   Input ← w $ // input string with end marike.

2. Repeat:
   Let s be top state on the stack.
   Let a be the current input symbol.

a) If Action[s, a] = shift t
   — push symbol a and state t onto th stack
   — Advance input to next symbol.

b) If Action[s, a] = reduce A → B.
   — pop 2x|B| from stack (since both symbol & state)
   — let s' be the new top state
   — push A and GOTO [s', A] on the stack.

c) IF ACTION[s,a] = Accept;
— Parsing successful, return Accept

d) IF ACTION = error
— report syntax error.

3. End of Algorithm.

(7) b. given, $S \rightarrow (L) / a$

$L \rightarrow L, S / S$

Step 1: Eliminate left recursion:

The grammer has left recursion in production for L:

$L \rightarrow L, S | S$

Rewrite it as

$L \rightarrow SL'$

$L' \rightarrow , SL' | \in$

so the grammer becomes.

$S \rightarrow (L) | a$

$L \rightarrow SL'$

$L' \rightarrow , SL' | \in$.

Step2: Find First and Follow.

FIRST(S) = {(, a}

FIRST(L) = {(, a}

FIRST(L') = {,, ϵ}

Follow(S) = {$, ;}

Follow(L) = {)}

Follow(L') = {)}

Step3: construct Predictive Parsing table:

Non-Terminal	(	a	)	,	$
S	S → (L)	S → a			
L	L → SL'	L → SL'			
L'			L' → , SL'	L → ϵ	

8.9

Rule	First	Follow
$E \rightarrow TE_R$	(, id	$,),
$E_R \rightarrow +TE_R \mid \epsilon$	+, ϵ	$,)
$T \rightarrow FT_R$	(, id	+, $)
$T_R \rightarrow *FT_R \mid \epsilon$	*, ϵ	+, $,)
$F \rightarrow (E) \mid id$	(, id	*, +, $,),

9

NFA to DFA: subset construction Method.

1. NFA → STT বানাতে হবে।
2. DFA → STT বানাতে হবে। ***

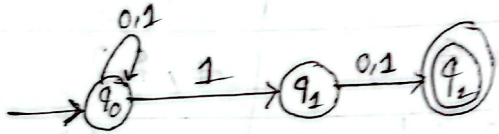
Initial state same হবে।

6.4.a

→ Write down the NFA of all binary strings in which second last bit is 1. Convert the NFA into DFA.

Ans:

Accepted Language: 10, 11, 00010, 1110, ---



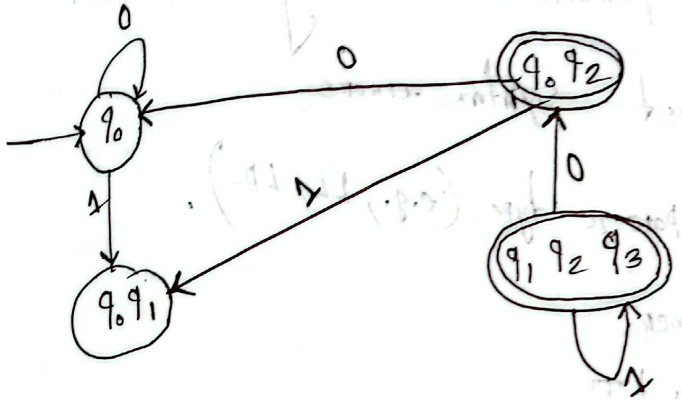
→ Create state transaction table for NFA.

State	input 0	1
→ q_0	q_0	(q_0, q_1)
q_1	q_2	q_2
* q_2	ϕ	ϕ

→ Now convert this table for DFA.

State	0	1
→ q_0	q_0	$[q_0 q_1]$
$[q_0 q_1]$	$[q_0 q_2]$	$[q_0 q_1 q_2]$
* $[q_0 q_2]$	$[q_0]$	$[q_0 q_1]$
$[q_0 q_1 q_2]$	$[q_0 q_2]$	$[q_0 q_1 q_2]$

→ state transition diagram:



5.a

Parser: A Parser is a component of a Compiler that takes input in form of a sequence of tokens (produced by the lexical Analyzer) and check it Against the Rules defined by the grammer. Its main task is to determine whether the input syntactically correct and to build a Parse tree or syntax tree that represent the structure of the input.

Role of a grammar in Parse construction:

→ Defines syntax rules of language

→ Guids the parser in constructing parse tree.

→ Helps detect syntax errors.

→ Determine parser type (e.g., LL, LR).

given grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

It has left recursive. Remove it, and then —

$E \rightarrow T E'$

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

$T' \rightarrow T * F$

$T \rightarrow F$

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

Parse table বানানো লাগবে।

(8.0) Design a recursive descent parser for the grammer:

Ans: given grammer:

$E \to E+T \mid T$ [has left recursion]

$T \to T*F \mid F$ [Also has "]

$F \to (E) \mid id.$

step 1: For recursive descent, the grammer must not be left recursive. so we have remove Eliminate All "

- For E:

$E \to TE'$

$E' \to +TE' \mid \epsilon.$

- For T:

$T \to T*F \mid F$

$T \to FT'$

$T' \to *FT' \mid \epsilon.$

Transformed grammer:

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id.$

Steps: Pseudocode for Recursive Descent Parser.

```
void E(){
    T();
    E_Prime();
}
void E_Prime(){
    if(lookahead == '+'){
        match('+');
        T()
        E_Prime();
    }else epsilon
}
void T_Prime(){
    if(lookahed == '*'){
        match('*');
        F();
        T_Prime();
    }
    else epsilon.
```

```
void F() {
    if (lookahead == '(') {
        match('(');
        E();
        match(')');
    } else if (lookahead == 'id') {
        match('id');
    }
    else error();
```

(b) given grammer:

$S \longrightarrow aABe$

$A \longrightarrow Abe \mid b$ (has left recursion, we have to remove it)

$B \longrightarrow d.$

transformed grammer:

$S \longrightarrow aABe$

$A \longrightarrow bA'$

$A' \longrightarrow bcA' \mid \epsilon$

$B \longrightarrow d.$

Non-terminal	First	Follow
S	{a}	{$, }
A	{b}	{d}
A'	{b, ϵ}	{d}
B	{d}	{e}

8.e) What is Recursive Descent parsing? List the Problem faced in designing such parser.

Ans: Is a top down Parsing method where each non-terminal in the grammer implemented as a recursive function. These function process the input and check if it matches the grammer.

Problem in designing Recursive Descent parsers:

1. Left Recursion: cause infinite recursion. must be removed.

2. Backtracking: Required when alternatives begin with same symbol, making parsing inefficient.

3. Ambiguty: Difficult to handle as multiple parse tree are possible.

4. Predictive limitions: works only for grammer suitable for Predictive Parsing (LL(1))

5. Error handling: Manual handling is needed for detecting and recovering from errors.

3.4

Leftmost Derivation

→ Always replaces the left-most non-terminal first

→ Derivation proceeds from Left to Right

→ used in LL Parsers (top-down Parsing)

Ex. given Grammer.

$S \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$

$\Rightarrow$ [ab]

$S \rightarrow AB$
$\rightarrow aB$
$\rightarrow ab.$

Rightmost Derivation

→ Always replaces rightmost non-terminal first.

→ Derivation proceeds from Right to Left.

→ used in LR parser (bootom-up parsing)

Ex.

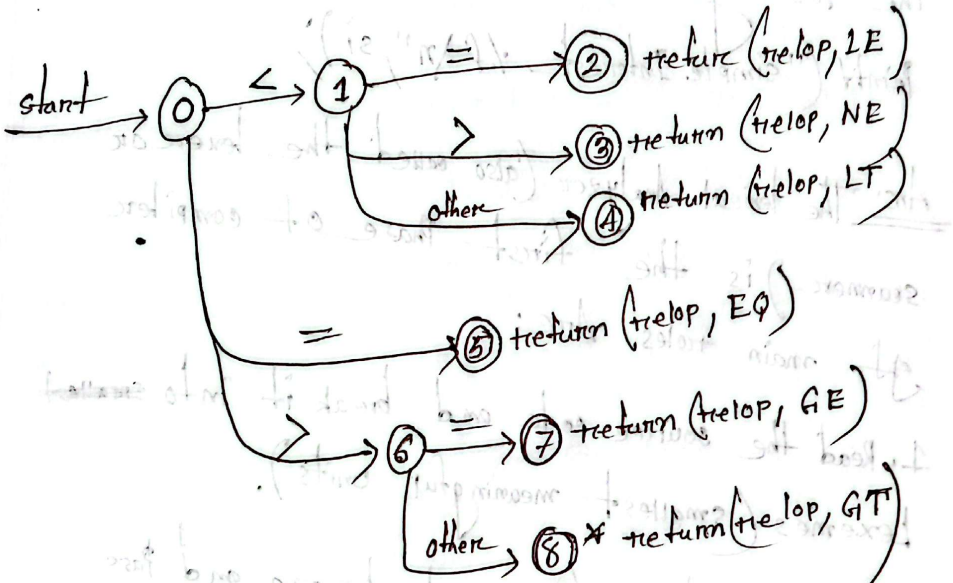
$S \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$

string: ab

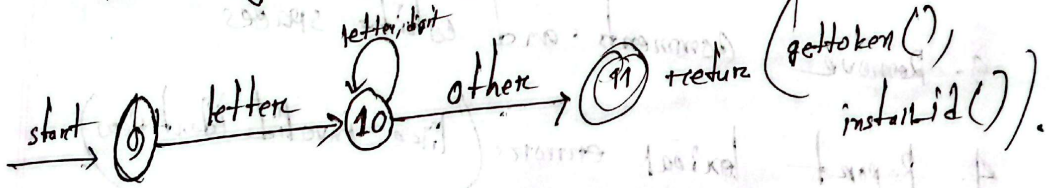
$S \rightarrow AB$
$\rightarrow Ab$
$\rightarrow ab.$

(c) Draw the transition diagram for the regular definition

relop () $<|<=|=|<>|>|>=$



id → letter (letter | digit)*



(2.a) State the role of the lexical Analyzer. Identify the lexemes and their corresponding tokens in the following statement:

printf("simple Interest = %f\n", si);

Ans: The lexical Analyzer (also called the lexer or scanner) is the first phase of compiler. Its main roles are:

1. Read the source code and break it into ~~smallest~~ Lexemes (smallest meaningful units).
2. Generate token for each Lexeme and pass them into parser.
3. Remove comments and white spaces.
4. Report lexical errors (like invalid identifiers).

Lexeme	Token
printf	Identifier
(	LEFT_PAREN
"simple interest = "	STRING_LITERAL.
,	comma
I	IDENTIFIER.
si	
)	Right-Paren
;	SEMICOLON.

(b) Discuss various phases of Compiler and trace the Program segment $a = b * c * 50$ for all Phase.

Ans A compiler translates source code into machine code in multiple well-defined Phases. Lets discuss the phases and trace the segment;

~~a = b * c~~ · $a = b + c * 50$.

1. Lexical Analysis:

→ Breaks the input into tokens

→ Lexemes & tokens:

Lexeme	token
a	IDENTIFIER
=	ASSIGN_OP
b	IDENTIFIER
+	plus_OP
a	Identifier.
*	Multi'OP
50	Number
;	SEMICOLON.

2. Syntax Analysis: Builds a Parse tree based on grammer Rules.

assignment → ID = expression

expression → ID + term.

term → ID * Number.

3. Semantic Analysis: Checks meaningful use of variable and operation

→ Are variable declared.

→ Are type compitable (e.g., 50 is valid for int/float)

4 Intermediate code generation: Converts code into three-Address code

$t_1 = c * 50$

$t_2 = b + t_1$

$a = t_2$

5. code optimization Improve Performance without changing behaviour

→ if c is constant c*50 can be Precalculated.

→ Reuse expression if possible.

6. Target code generation: Translates IR to Assembly code

```
MOV R1, c
MUL R1, 50
MOV R2 b
ADD R2, R1
MOV a, R2.
```

7. Code Linking & ~~Libs~~ Assembly:

Final phase: resolve Addresses, links, libraries and create executable code.

a<b: shift
a>b: reduce
a=b: match Parenthesis

(c) Using Operator Precedence relations, Parse the string.

id + (id * id)

→ Define A Grammer:

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

id → সবচেয়ে বেশি
$ → সবচেয়ে কম

→ Operator Precedence Table:

Left side	id	+	*	$
id	=	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	=

* Terminal এর আগে compare করতে হবে।

stack	Input	Action.
$	id + id * id $	shift
$id	+id * id $	id > +; reduce
$E	+id * id $	$ < +; shift
$E+	id * id $	+ < id; shift
$E+id	* id $	id > *; reduce.
$E+E	* id $	$ < *; shift
$E+E*	id $	* < id; shift
$E+E*id	$	id > $; reduce
$E+E*E	$	* > $ reduce
$E+E	$	+ > $ reduce
$E	$	Accepted.

1.a **Compiler:** A Compiler is a Program that translates source code written in High Level Language into machine code or Intermediate cod.

Analysis & synthesis Model of a compiler:

(Front-end) → **Analysis Phase:** Breaks source code into parts and build intermediate representation.

Phases:

1. Lexical Analysis: Converts input into tokens
2. syntax Analysis: Build Parse tree.
3. Semantic Analysis: checks meaning and type.
4. Intermediate code generation: Produce IC.

Synthesis phase (Back-end): Converts IC intermediate representation to target code.

Phases:

1. code optimization: Improve code efficiency.
2. code generation: Generates machine code
3. Assembly and Linking: Combine all and Produce executable.

(b) given, $x = y * z + y * (-z)$

Generate Three-Address code:

$t_1 = y * z$

$t_2 = -z$

$t_3 = y * t_2$

$t_4 = t_1 + t_3$

$x = t_4$.

Index	operation	Argument 1	Argument 2
0	*	y	z
1	uminus	z	
2	*	y	(1)
3	+	(0)	(2)
4	=	x	(3)

(c) **Code optimization:** is the Process of improving the Intermediate code to make Program run faster, use less memory without changing its behaviour.

→ Types of optimization.

1. **Machine-Independent optimization:** Applied before knowing the target machine.

Ex: → constant Folding: $x = 2+3 \rightarrow x = 5$.

→ Dead code elimination.

→ common sub-expression elimination.

2. Machine Dependent optimization:

→ Applied after selecting the target machine
→ Tailors code to specific Architecture (eg, registers, instruction sets).

Example:
→ Instruction scheduling
→ Register Allocation
→ peephole optimization.

(4d) Differentiate Tokens, Patterns, Lexeme;

Token: A category/class of lexical units (e.g. ID, Number ...)

Lexeme: Actual string of character in the source code. (e.g., x, 123, if)

Pattern: Rule/regular expression that describes the structure of lexemes for a token.

Example: In the statement:

int x = 10;

Lexeme: int, x, =, 10;

token: keyword, identifier, number.

pattern: Identifier → letter (letter | digit)*

Number → digit+.

6.a **LL(1) grammer:** An LL(1) grammer is type of context free grammer that can be parsed by a top down parser using:

L: Left to right scanning of the input

L: Left-most derivation

1: using one lookhead symbol (token) to make parsing decisions.

Given grammer:

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

$E \rightarrow b$

Both 1st 2 Production of S start with iES.

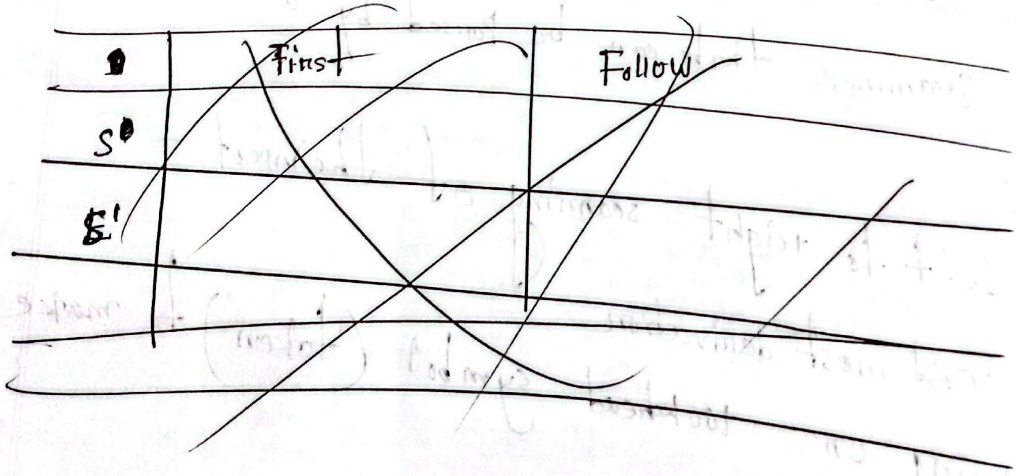
→ Left factoring of the grammer:

$S \rightarrow iEtSS'$

$S' \rightarrow \epsilon \mid eS$

$S \rightarrow a$

$E \rightarrow b.$



	First	Follow
S	{i, a}	{$, e}
S'	{e, ϵ}	{$, e}
E	{b}	{t}

Parsing Table:

Non terminal	i	a	b	e	t	$
S	S→iEtSS'	S→a				
S'				S→eS S→eS		
E			E→b			

This grammer is not LL(1) because the Predictive Parse table contains multiple entries in the same cell (S', e) causing a First / Follow conflict.

In LL(1) grammer, each must have at most one Production, which is not satisfied here.

Grammer:

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$E' \rightarrow \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT'$

$T' \rightarrow \epsilon$

$F \rightarrow (E)$

$F \rightarrow id.$

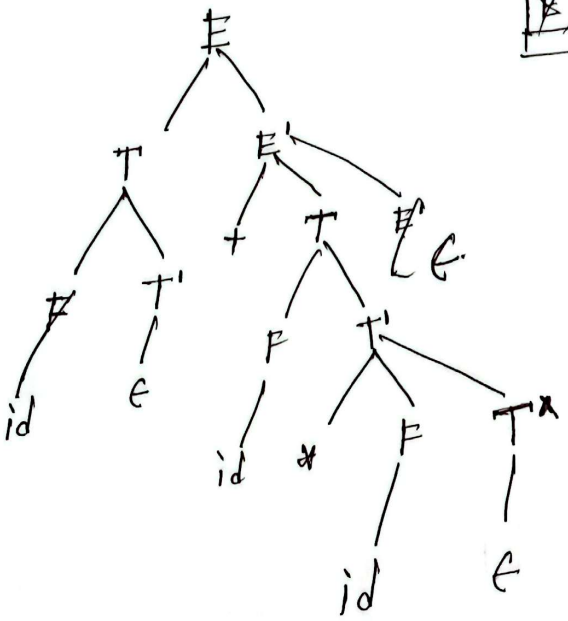
First(E):-

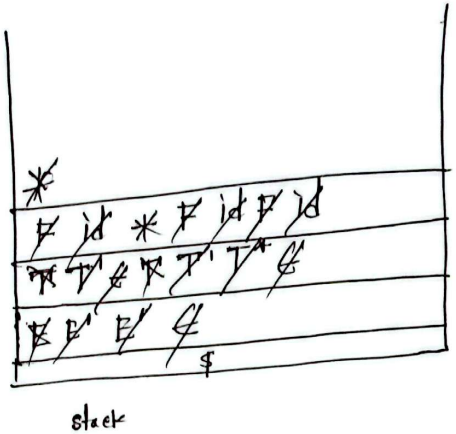
5.4b

First	follow	Non-terminal	+	*	(	)	id	$
(, id	$),	E			$E \rightarrow TE'$		$E \rightarrow TE'$	
$+\epsilon$	$)	E'	$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$		$E \rightarrow \epsilon$
(, id	+, $),	T			$T \rightarrow FT'$		$T \rightarrow FT'$	
$* \epsilon$	+ $,)	T'	$T' \rightarrow \epsilon'$	$T' \rightarrow *FT'$	*	$T' \rightarrow \epsilon'$		$T \rightarrow \epsilon'$
(, id,	*, +, $)	F			$F \rightarrow (E)$		$F \rightarrow id$	

ming: id + id * id $
X X X X X

Parse Tree:





stack