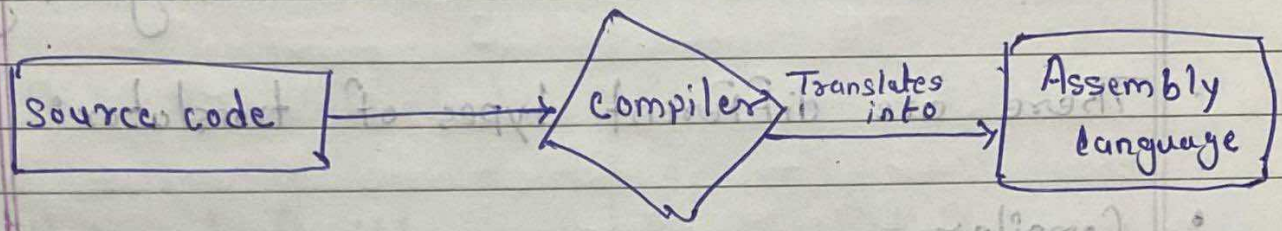
(1) Define compiler, Interpreter and translator?

Compiler:

Compiler is a software that translate source code from a high-level programming language to a low level language.



Source code ⟶ Compiler ⟶ Machine Code

↓

Output

Interpreter:

An interpreter is a program that directly executes the instructions in a high-level language, without converting it into machine code.

Source code ⟶ Interpreter ⟶ Output

Translator :

It is a software program that takes input a program written in some language and produce output in another language.

Source code ⟶ Translator ⟶ Target language

There are different types of translator:

- Compiler
- Interpreter
- Assembler
- Macro Assembler
- Preprocessor
- High level translator
- De compiler & Deassembler

(2) Difference between Compiler & Interpreter ?

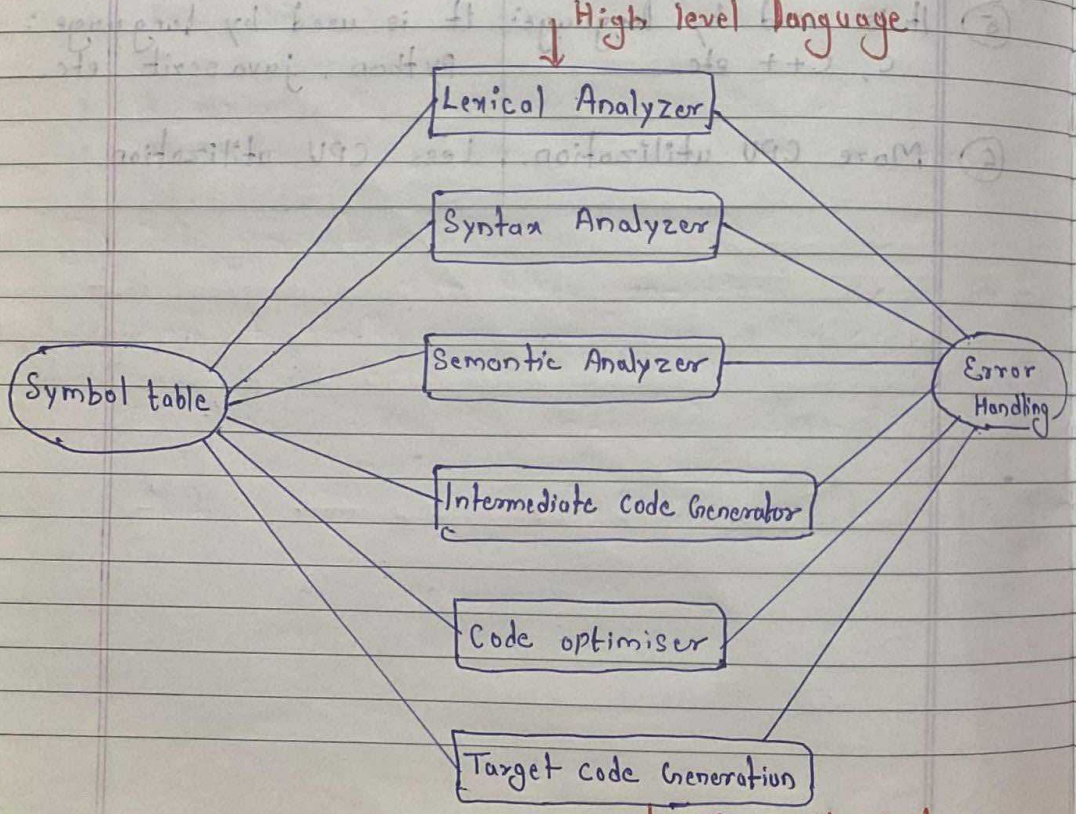
	Compiler	Interpreter
(1)	It translate program in a single run.	It translate program line by line.
(2)	It consumes less time.	It consumes more time.
(3)	It is more efficient.	It is less efficient.
(4)	It is not flexible.	It is flexible.
(5)	It is used by language: C, C++ etc.	It is used by language: python, javascrit etc.
(6)	More CPU utilization.	Less CPU utilization.

(13) Explain the different phases of compiler with an example ?

A compiler is a software program that converts the high-level source code written in a programming language into low-level machine code that can be executed by the computer hardware.

The process of converting the source code into machine code involves several phases or stages, which are known as the phases of compiler.

The various phases of compiler are:



(i) Lexical Analyzer :

- The first phase of a compiler is lexical analyzer, also known as scanning.
- This phase reads the source code and breaks it into a stream of tokens, using a tool called LEX tool.
- It removes white spaces, comments, tabs.

(ii) Syntax Analyzer :

- The second phase of a compiler is Syntax Analyzer, also known as parsing.
- It takes the tokens one by one and uses CFG to construct the parse tree.
- It takes input as tokens & check their syntatic correctness.

(iii) Semantic Analyzer :

- Verifies the parse tree, wheather it is meaningful or not.
- It checks wheather the parse tree follows the rules of language.
- It keep track of identifier, their types and expression.

(iv) Intermediate code Generator:

- It generates the intermediate representation between the source code and machine code.
- The intermediate code is generated in such a way that you can easily translated into the machine code.

(v) Code optimization:

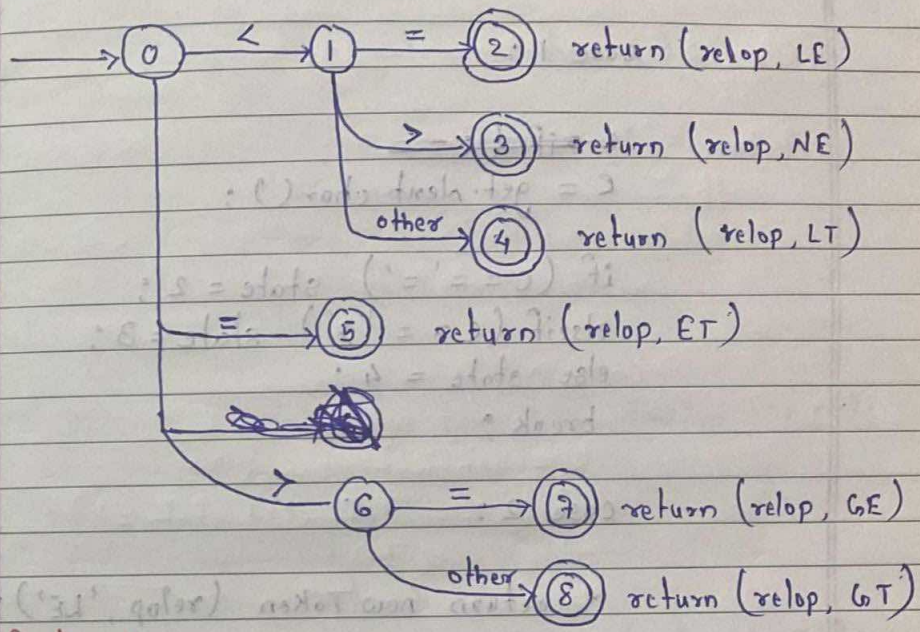
- It is used to improve the intermediate code so that the output of the program could run faster and takes less space.
- It removes the unnecessary line of code in order to speedup the program execution.

(vi) Code generator:

- It takes optimized code as input and it finally generated assembly code.
- This phase takes the optimized intermediate code and generates the actual machine code that can be executed by the target hardware.

(4) Draw the transition diagram of
a. identifier b. Relational operator

- Identifier Relational operator
(<, <=, >, >=, =, =<>)

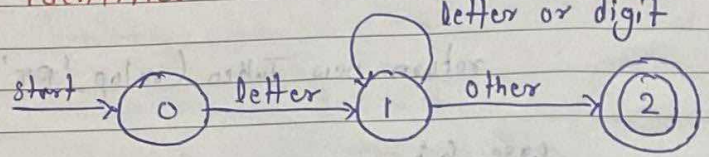


Code:

```
token get Relop () {
// ...
    while (yes) {
        switch state {

        case 0 :
            c = getNext char ();
```

```
if (c == '<') state = 1;
else if (c == '=') state = 5;
elseif (c == '>') state = 6;
else fail ()
break;

case 1:

~~if (c ==~~
c = get next char();

if (c == '=') state = 2;
elseif (c == '>') state = 3;
else state = 4;
break;

case 2:

return new Token (relop, 'LE');

case 3:

return new Token (relop, 'NE');

case 4:

retract ();
return new Token (relop, 'LT');

case 5:

return new Token (relop, 'ET');

case 6:

c = get next char ();

if (c == '=') state = 7;
else state = 8;
break;

case 7:

return new Token (relop, 'GE');

case 8:

retract ();
return new Token (relop, 'GT');

default:
state = 0;
break;

}
}
// ...
}
```

- Identifier



Code :

```
token get identifier () {
    //...
    while (yes) {
      switch (state) {

        case 0:

          c = get Next char ();
          if (c == 'letter') state = 1);
          else fail ();
          break;

        case 1:

          retract ();
          return to new token (identifier, 'letter');

        case 2;

          c = get Next char ();
          if (c == 'letter' or 'digit') state = 1);
          else state = 2;
          break;

        case 3:

          retract ();
          return new token (identifier, 'letter'
                                    or 'digit');

        default:

          State = 0;
          break;
        }
      }
    //...
}
```

(5) Consider the following program

```
Main {}
{
  int x, y, z;
  z = x + y;
}
```

List down all the lexemes, tokens and the attributes of the tokens at the end of lexical analysis of the above program.

Leximes	Tokens	Attributes
Main	keyword	
(	open parenthesis	
)	close parenthesis	
{	open curlly bracket	
int	keyword	
x	id	<id 1>
,	comma	
y	id	<id 2>
,	comma	
z	id	<id 3>
;	semicolon	
=	assignment operator	
+	sum	
}	close curly bracket	

(6) Difference between single parse and multi parse compiler ?

	Single Parse	Multi Parse
(1)	It reads the code only once and translates it at a similar time.	It reads the code multiple times, each time changing it into numerouss forms.
(2)	They are faster.	They are slower than single parse.
(3)	Called a narrow compiler.	Called a wide compiler.
(4)	There is no code optimization.	There is code optimization.
(5)	There is no intermediate code generator.	There is a intermediate code generator.
(6)	It takes minimum time to compile.	It takes more time to compile.
(7)	Memory consumption is less	Memory consumtion is more.
(8)	Has a limited scope.	Has a great scope.
(9)	Used in Pascal & C	Used in Java.

(7) Eliminate left recursion from the follow grammar

$E \longrightarrow E+T \mid T$
$T \longrightarrow T*F \mid F$
$F \longrightarrow (E) \mid id$

$E \longrightarrow E+T \mid T$

$A \longrightarrow E \qquad \alpha = +T \qquad \beta = T$

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

$T \longrightarrow T*F \mid F$

$A \longrightarrow T \qquad \alpha = *F \qquad \beta = F$

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

Recursive grammer after eleminating left recursion

$E \longrightarrow TE'$
$E' \longrightarrow +TE' \mid \varepsilon$
$T \longrightarrow FT'$
$T' \longrightarrow *FT' \mid \varepsilon$
$F \longrightarrow (E) \mid id$

(8) Eliminate left recursion from the follow grammer

$A \longrightarrow ABd \mid Aa \mid a$
$B \longrightarrow Be \mid b$

$A \longrightarrow ABd \mid Aa \mid a$

$A = A \qquad \alpha = Bd \mid a \qquad \beta = a$

$A \longrightarrow aA'$
$A' \longrightarrow BdA' \mid \varepsilon$
$A' \longrightarrow aA' \mid \varepsilon$
$A' \longrightarrow BdA' \mid aA' \mid \varepsilon$

$B \longrightarrow Be \mid b$

$A = B \qquad \alpha = e \qquad \beta = b$

$B \longrightarrow bB'$
$B' \longrightarrow eB' \mid \varepsilon$

Recursive grammer after eleminating left recursion

⑨ Write the rules to find first and follow. Also calculate first and follow for the following grammer

$E \longrightarrow TA$
$A \longrightarrow +TA \mid \epsilon$
$T \longrightarrow FB$
$B \longrightarrow *FB \mid \epsilon$
$F \longrightarrow (E) \mid id$

- **Rule for first ( )**

① If X is a terminal then first (x) is just X!

② If there is a production $X \longrightarrow \epsilon$ then add ϵ to first (x)

③ If there is a production $X \longrightarrow Y_1, Y_2 \dots Y_k$ then add first $(Y_1, Y_2 \dots Y_k)$ to first (x).

④ First $(Y_1, Y_2 \dots Y_k)$ to first (x) is either.
- first (Y_1) (if first (Y_1) doesn't contains ϵ).
- OR (if first (Y_1) doesn't contains ϵ) then first $(Y_1 Y_2 \dots Y_k)$ is everything in first (Y_1)
- If first (Y_1) first (Y_2) .... first (Y_k) all contains ϵ then add ϵ to first $(Y_1, Y_2 \dots Y_k)$ as well.

- **First Function**
  - First (E) = { c, id }
  - First (A) = { +, ε }
  - First (T) = { c, id }
  - First (B) = { *, ε }
  - First (F) = { c, id }

- **Follow function**
  - Follow (E) = { $,) }
  - Follow (A) = { $,) }
  - Follow (T) = { +, $,) }
  - Follow (B) = { +, $,) }
  - Follow (F) = { *, +, $,) }

(10) Calculate the first and follow

S ⟶ aBDh
B ⟶ cC
C ⟶ bC / ϵ
D ⟶ EF
E ⟶ g / ϵ
F ⟶ f / ϵ

- First function

- First (S) = {a}
- First (B) = {c}
- First (C) = {b, ϵ}
- First (D) = {g, f, ϵ}
- First (E) = {g, ϵ}
- First (F) = {F, ϵ}

- Follow function

- Follow (S) = {$}
- Follow (B) = {g, f, h}
- Follow (C) = {g, f, h}
- Follow (D) = {h}
- Follow (E) = {g, h}
- Follow (F) = {h}

(11) Consider the following grammer

S ⟶ AA
A ⟶ aA
A ⟶ b

Construct the LALR parsing table.

Step (1): Design of LR[1] parser

(1) Argumented grammer

S' ⟶ .S — (0)
S ⟶ .AA — (1)
A ⟶ .aA — (2)
A ⟶ .b — (3)

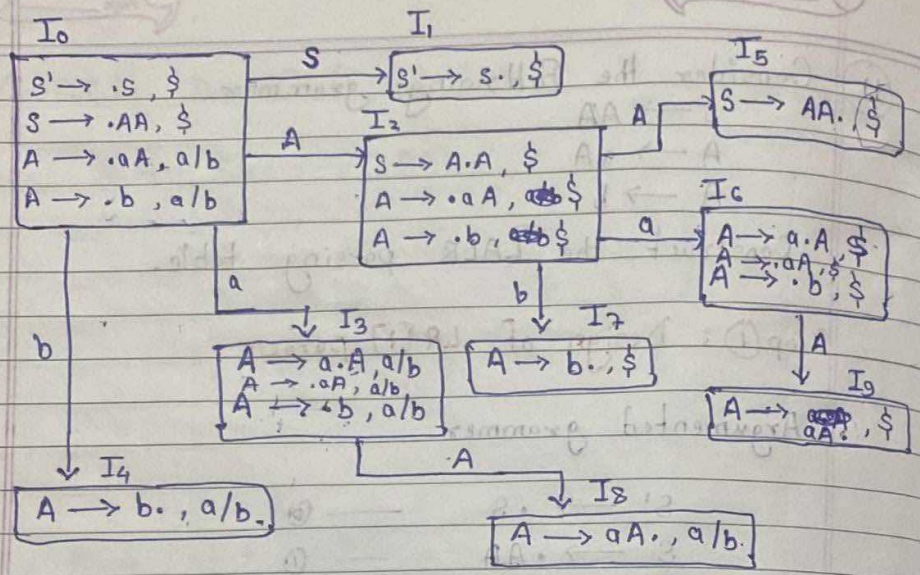
(2) Calculation of first set

first (A) = {a, b}
first (S) = {a, b}

(3) Transition diagram

S' ⟶ .S, $
S ⟶ .AA, $
A ⟶ .aA, a/b
B ⟶ .b, a/b

P.T.O



(4) LR(1) Parsing table

	Action			Goto	
State	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_6	S_7			5
I_3					8
I_4	R_3	R_3			
I_5			R_1		
I_6			..		9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

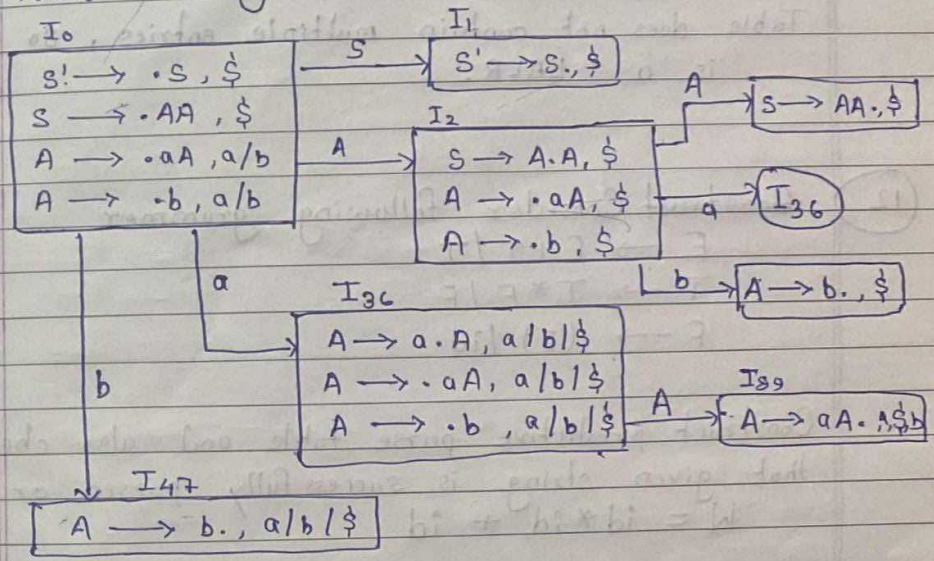
Step (2) : Design of LALR

I_{36} = A → a.A , a/b/$
A → .b , a/b/$
A → .aA , a/b/$

I_{47} = A → b. , $/a/b

I_{89} = A → aA. , $/a/b

- Transition Diagram :



LALR parsing table:

State	Action			GOTO	
	a	b	$	S	A
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	R_3	R_3	R_3		
I_{89}	R_2	R_2	R_2		
I_5			R_1		

Table does not contain multiple entries, so it is a LALR.

(12) ~~Construct~~ Consider following grammer

$E \longrightarrow E+T \mid T$

$T \longrightarrow T*F \mid F$

$F \longrightarrow (E) \mid id$

Construct predictive parse table and also check that given string is successfully parsed or not.

$W = id * id + id$

Step ①: Eleminate left recursive

$E \longrightarrow E+T \mid T$

$A = E \quad \alpha = +T \quad \beta = T$

$E \longrightarrow TE'$

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

$T \longrightarrow T*F \mid F$

$A = T \quad \alpha = *F \quad \beta = F$

$T = FE'$

$T' = *FE' \mid \epsilon$

$E \longrightarrow TE'$

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

$T \longrightarrow FT'$

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

$F \longrightarrow (E) \mid id$

Step ②: First ( ) and Follow ( )

	First ( )	Follow ( )
$E \longrightarrow TE'$	{(, id}	{$,)}
$E' \longrightarrow +TE' \mid \epsilon$	{+, ε}	{$,)}
$T \longrightarrow FT'$	{(, id}	{+, $,)}
$T' \longrightarrow *FT' \mid \epsilon$	{*, ε}	{+, $,)}
$F \longrightarrow (E) \mid id$	{(, id}	{*, +, $,)}

Step (3): Predictive Parsing table

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

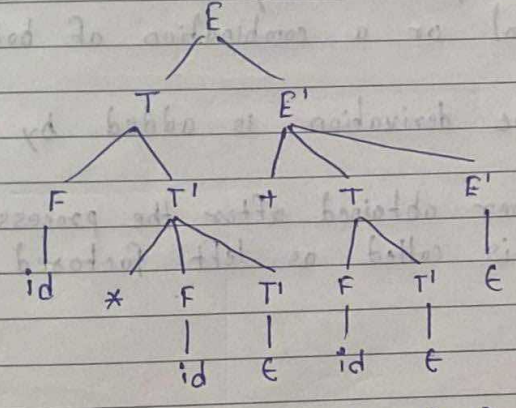
- Parser input string

$w = id * id + id$

Stack	Input	Output
$\$E$	id * id + id $	$E \to TE'$
$\$E'T$	id * id + id $	$T \to FT'$
$\$E'T'F$	id * id + id $	$F \to id$
$\$E'T'id$		
$\$E'T'$	* id + id $	$T' \to *FT'$
$\$E'T'F*$		
$\$E'T'F$	id + id $	$F \to id$
$\$E'T'id$		
$\$E'T'$	+ id $	$T' \to \epsilon$
$\$E'$	+ id $	$E' \to +TE'$
$\$E'T+$		
$\$E'T$	id $	$T \to FT'$
$\$E'T'F$	id $	$F \to id$

Stack	Input	Output
$\$E'T'id$		
$\$E'T'$	$	$T' \to \epsilon$
$\$E'$	$	$E' \to \epsilon$
$	$	

- Parse tree



⇒ Hence Input string derived form ~~stat~~ start symbol.

(13) Define left factoring and eliminate left factoring from the following grammer (if there is any)

$S \longrightarrow ietSeS \mid ietS \mid a$

$E \longrightarrow t$

Eliminating Left factoring

$S \longrightarrow ietSS' \mid a$

$S' \longrightarrow eS \mid \epsilon$

$E \longrightarrow t$

Left factoring :

Left factoring is a process by which the grammer with common pre-fixes is transformed to make it useful for top down parsers.

- We make one production for each common prefixes.
- The common prefixes may be a terminal or a non-terminal or a combination of both.
- Rest of the derivation is added by new production.

The grammer obtained after the process of left factoring is called as left factored Grammer.

(14) Explain various code optimization techniques.

(1) Constant folding:

Constant folding is a technique used to simplify expressions that involve constant values. This technique involves evaluating constant expressions at compile time, rather than at runtime.

(2) Common subexpression elimination:

The optimizations techniques identifies expressions that are computed multiple times within a program and replace them with a single computation.

(3) Dead code elimination:

This technique removes code that has no effect on the output of a program.

This includes unused variables, unreachable code, and expressions that are always false.

(4) Strength reduction:

This optimization technique replaces expensive operations with less expensive ones.

For ex: replacing multiplication with addition or shifting.

(5) Loop optimization:

This technique optimize loops by minimize the no. of instructions executed inside a loop.

⑥ Register allocation:

This technique assigns the variables of a program to registers on the CPU. This reduces the number of memory accesses and improve performance.

(15) Explain various issues that occur during code generation.

① Instruction selection:

The first step of code generation is to select appropriate instructions for each operation in the intermediate representation.

② Register allocation:

The compiler needs to allocate registers efficiently to reduce meomory access and enhance performance.

③ Address calculation:

When accessing arrays and structures, the compiler needs to culculate the memory address of each element based on the data type, index and base address.

④ Control flow:

The compiler must generate instructions to handle control flow constructs like if-else, switch case, loops and function calls.

⑤ Code optimization:

The compiler performs various code optimization technique to improve code quality, performance and reduce memory usage. Some common optimization techniques are: Dead code, Constant folding, Loop optimization.

⑥ Memory manggement:

The compiler needs to mangge the memory used by the programs, including stack, heap and static memory.

⑦ Error handling:

The compiler needs to detect and report error during code generation, such as syntax error, semantic errors and runtime errors.

(16) Basic Blocks

It is a straight line code sequence that has no branches in and out branches ~~except~~ except to the entry and at the end respectively.

Basic block is a set of statements that always executes one after other, in a sequence. There are four types of basic blocks are:

① Standard block
② Simplified block
③ Trade block
④ Tailoring block

(17) For the given grammer

$E \rightarrow E_1 + T$

$E \rightarrow E_1 - T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow T_1 / F$

$T \rightarrow F$

$F \rightarrow (E)$

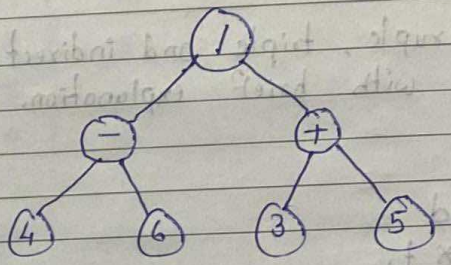
$F \rightarrow num$

Give the semantic rule to represent syntax directed definition for arithmetic expression and draw annotated parse tree for $4-6/3+5$.

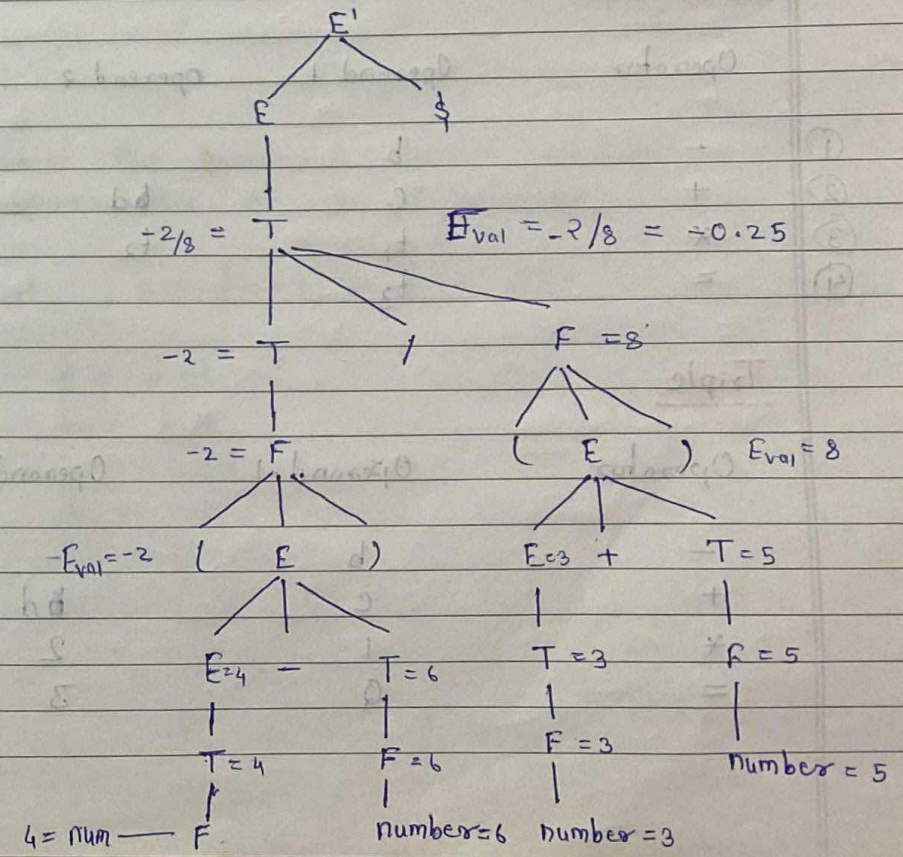
Production Rule	Semantic Rule
$E' \rightarrow E\ \$$	$E'.val = print(E_{val})$
$E \rightarrow E_1 + T$	$E_{val} = E_{val} + T_{val}$
$E \rightarrow E_1 - T$	$E_{val} = E_{val} - T_{val}$
$E \rightarrow T$	$E_{val} = T_{val}$
$T \rightarrow T_1 * F$	$T_{val} = T_{val} * F_{val}$
$T_1 \rightarrow T_1 / F$	$T_{val} = T_{val} / F_{val}$
$T \rightarrow F$	$T_{val} = F_{val}$
$F \rightarrow (E)$	$F_{val} = E_{val}$
$F \rightarrow num$	$F_{val} =$ number value

Given, $4-6/3+5$

Postfix Notation = $46-35+/$



Annotated parse tree



(18) For the given arithmatic expression,

$$Q = -b * (c+d)$$

give the quadruple, triple and indirect triple representation with brief explanation.

$t_1 = -b$
$t_2 = c+d$
$t_3 = t_1 * t_2$
$Q = t_3$

Quadruple

	Operator	Operand 1	Operand 2	Result
(1)	-	b		t_1
(2)	+	c	d	t_2
(3)	*	t_1	t_2	t_3
(4)	=	t_3		Q

Triple

	Operator	Operand 1	Operand 2
1	-	b	
2	+	c	d
3	*	1	2
4	=	Q	3

Indirect

	Statement
1	10
2	11
3	12
4	13

	Operator	Operand 1	Operand 2
(1)	-	b	
(2)	+	c	d
(3)	*	10	11
(4)	=	Q	12

(19) Give the quadruples, triples and indirect triple representation of the statements:

$$x = (a+b) * + - c/d.$$

$t_1 = a+b$
$t_2 = t_1 *$
$t_3 = c/d$
$t_4 = -t_3$
$t_5 = t_2 + t_4$
$x = t_5$

Quadruples

	Operators	Operand 1	Operand 2	Result
①	+	a	b	t_1
②	*	t_1	t_2	t_2
③	/	c	d	t_3
④	−	t_3		t_4
⑤	+	t_2	t_4	t_5
⑥	=	t_5		x

Triple

	Operators	Operand 1	Operand 2
	+	a	b
	*	1	−
	/	c	d
	−	3	−
	+	2	4
	=	x	5

Indirect triples

	Statements
1	14
2	12
3	13
4	14
5	15

	Operators	Operand 1	Operand 2
①	+	a	b
②	*	11	−
③	/	c	d
④	−	13	−
⑤	+	12	14
⑥	=	x	15

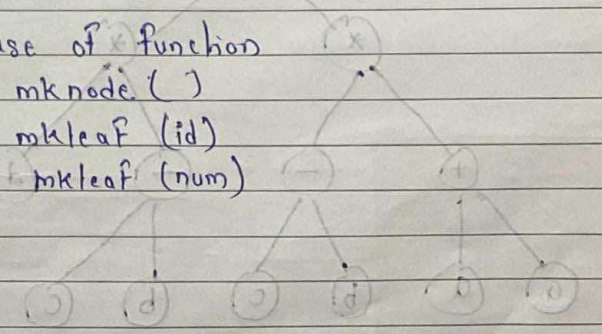
(20) Explain in details the construction of syntax tree for a binary infix arthemetic expression:

$$a + a * (b - c) + (b - c) * d$$

step(1): Convert the expression from infix to postfix

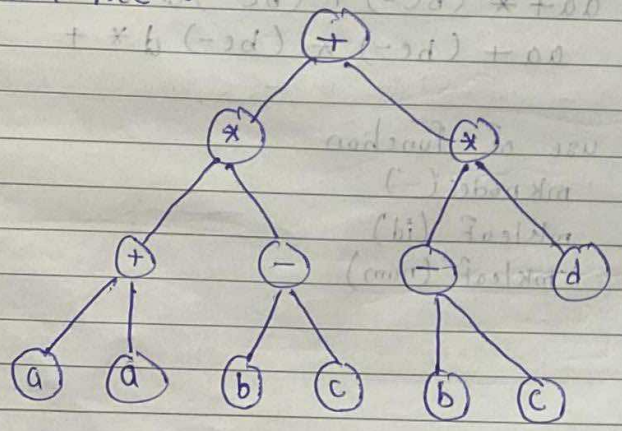
$$a\,a + * (b\,c -) + (b\,c -)\,d *$$
$$a\,a + (b\,c -) * (b\,c -)\,d * +$$

step(2): Make use of function
mknode ( )
mkleaf (id)
mkleaf (num)



Symbol	Operation (table in symbolic)
a	P_1 = mkleaf (id, pointer to id entry)
a	P_2 = mkleaf (id, pointer to id entry)
+	P_3 = mknode (+, P_1, P_2)
b	P_4 = mkleaf (id, pointer to id entry)
c	P_5 = mkleaf (id, pointer to id entry)
-	P_6 = mknode (-, P_4, P_5)
*	P_7 = mknode (*, P_3, P_6)
b	P_8 = mkleaf (id, pointer to id entry)
c	P_9 = mkleaf (id, pointer to id entry)
-	P_{10} = mknode (-, P_8, P_9)
d	P_{11} = mkleaf (id, pointer to id entry)
*	P_{12} = mknode (*, P_{10}, P_{11})
+	P_{13} = mknode (+, P_7, P_{12})

Syntax tree :



(21) Write the three address code for the following program fragment :

```
while (p > q and r > s) do
    if p = 1 then q = q + 1

else
    while (p <= s) do
    p = p + 3
```

```
0   if p > q goto 2
1   goto 14
2   if r > s goto 4
3   goto 14
4   p = 1 goto 6
5   goto 9
6   t1 = q + 1
7   q = t1
8   goto 0
9   p <= s goto 11
10  goto 0
11  t2 = p + 3
12  p = t2
13  goto 9
14  END
```

(22) Construct CLR table

$S \longrightarrow AaAb \mid BbBa$
$A \longrightarrow \epsilon$
$B \longrightarrow \epsilon$

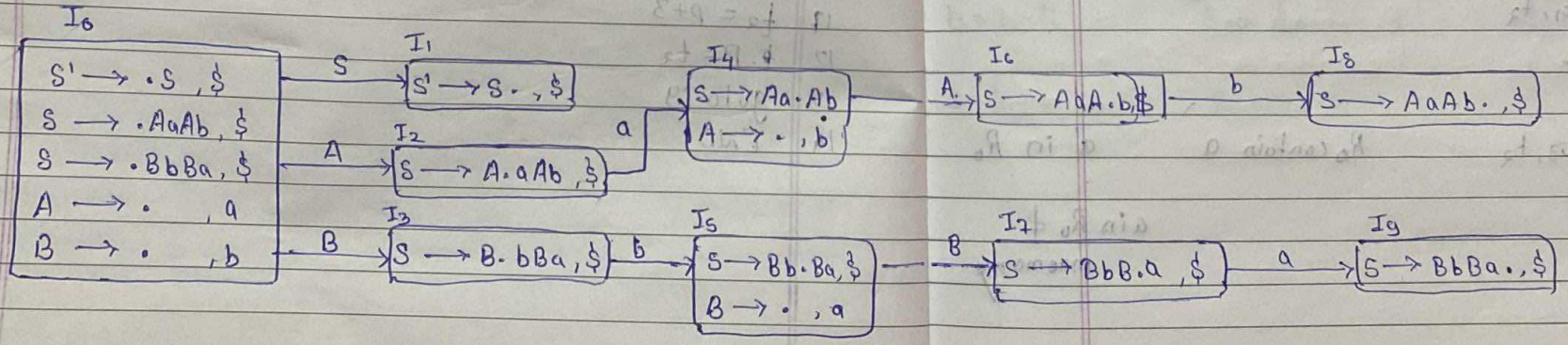
Step (1) : Argumented grammer

$S' \longrightarrow .S$	(0)
$S \longrightarrow .AaAb$	(1)
$S \longrightarrow .BbBa$	(2)
$A \longrightarrow .$	(3)
$B \longrightarrow .$	(4)

Step (2) Calculation first set

First (S) = {a, b}
(A) = {ϵ}
(B) = {ϵ}

Step (3): Transition Diagram :



• CLR Parsing table

State	a	b	$	S	A	B
I_0	R_3	R_4		1	2	3
I_1			Accept			
I_2	S_4					
I_3		S_5				
I_4		R_3			6	
I_5	R_4					7
I_6		S_8				
I_7	S_9					
I_8			R_1			
I_9			R_2			

(23) Generate code sequence for:

$$a = x-y + x-z + x-z$$

Three address code

$t_1 = x-y$
$t_2 = x-z$
$t_3 = t_1+t_2$
$a = t_3+t_2$

Statement	Code Generated	Register Discriptor	Address Discriptor
$t_1 = x-y$	Mov x, R_0 Sub y, R_0	R_0 contain t_1	t_1 in R_0
$t_2 = x-z$	Mov x, R_1 Sub z, R_1	R_0 contain t_1 R_1 contain t_2	t_2 in R_1
$t_3 = t_1+t_2$	Add t_1, t_2	R_0 contain t_1 R_1 contain t_3	t_3 in R_1
$a = t_3+t_2$	Add t_3, t_2	R_0 contain a	a in R_0 & m/o

(24) Apply code algorithm to:

$$a = (p+q) - (r+s) - t$$

Three address code

$t_1 = p+q$
$t_2 = r+s$
$t_3 = t_2-t_0$
$a = t_1-t_3$

Statement	Code Generated	Register Discriptor	Address Discriptor
$t_1 = p+q$	Mov p, R_0 Add q, R_0	R_0 contains t_1	t_1 in R_0
$t_2 = r+s$	Mov r, R_1 Add s, R_1	R_0 contains t_1 R_1 contain t_2	t_2 in R_1
$t_3 = t_2-t_1$	Sub t_2, t_1	R_0 contain t_1 R_0 contain t_3	t_3 in R_0
$a = t_1-t_3$	Sub t_1, t_3	R_0 contain a	a in R_0 & m/o