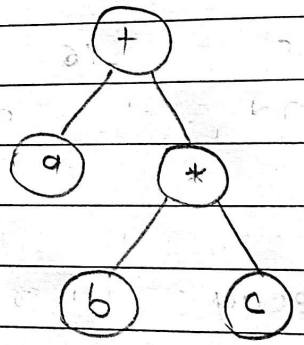
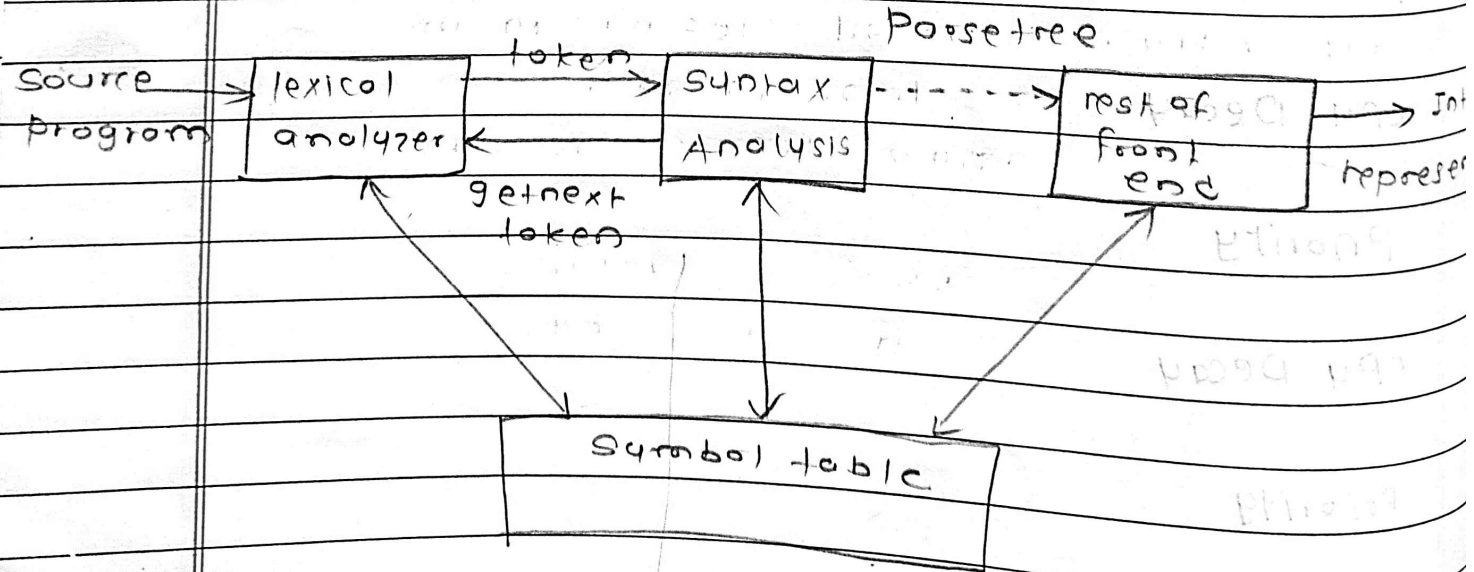
Syntax Analysis.

1) Syntax Analysis is second phase of compilation
2) Syntax Analysis is performed by syntax analyzer
3) Syntax Analyzer also called as parser.
4) Syntax Analysis phase checks grammatical struc & syntactical structure by using context free grammer.
5) Input to syntax Analysis is stream of tokens generated by lexical analysis phase
6) Output of syntax Analysis is Parse tree or a synta tree
7) The main goal is to create a hierarchy of source co & hierarchy shows grammatical structure of source code.
8) Ex - a+b*c



9) Structure of parser



In the above diagram.

Parser takes input from lexical analyzer as form in tokens & for getting next token parser request the lexical analyzer next token.

Parser uses context free grammer & create the parse tree of ~~source code~~ tokens.

From parse tree intermediate code will generated in next some phases.

There are 2 types of parser. ① Universal

① Top Down - Starts building parse from root node to the leaf or child node Top down parsing generates parse tree with or without backtracking

② Bottom up - Starts building parse tree from child or leaf node to root node it consist of types like shift reduce parser etc.

functions

Tasks performed by the parser

① It verifies the structure & grammatical structure of

② It constructs parse tree

③ It reports errors

④ It performs error recovery.

Advantages - Improved code generation
structural validation
Easier semantic Analysis

Disadvantages - Complexity
Reduce performance
limited error recovery.

Shift Reduce Parsing

1) Shift reduce parsing is form of Bottom Up parsing in which a stack holds grammer symbols & input buffer holds rest of input string to be parsed.

2) Handle- Handle always appears at top of the stack.

3) Shift reduce parsing is a process of reducing a string to start symbol of grammer.

A string $\xrightarrow[\text{Reduce to}]{}$ starting symbol

4) Shift reduce parsing always performs two action shift & reduce that's why it is known as shift reduce parsing.

5) We use $ to make bottom of stack & also right end of input.

6) The parser repeats the cycle until it has detected an error or until stack contains the start symbol & input is empty.

STACK	INPUT
$S	$

7) Example id* id

STACK	INPUT	ACTION
$	id1 * id2 $	shift
$id1	* id2 $	reduce by F → id
$F	* id2$	reduce by T → F
$T	* id2 $	shift
$T*	id2 $	shift
$T* id2	$	reduce by F → id
$T* F	$	reduce by T → T*F
$ T	$	reduce by E → T
$ E	$	accept

8) There are 4 main & important possible actions in shift reduce parser
i) shift ii) reduce iii) accept iv) error.

9) Shift action - next input symbol is pushed to stack at top of stack.

10) Reduce action - replaces the handle within a stack with non terminal.

11) accept - Announces successful completion of parsing

12) Error - Discover a syntax error & call an error recovery routine.

13) The use of stack in shift reduce parsing is identified as handle will always eventually at the top of stack & never inside the stack.

Advantages -

① shift reduce parsing is efficient & can handle a wide range of context free grammers.

② It can parse a large variety of programming languages

③ It is capable for handling both left & right recursive grammer.

④ This parser is efficient in terms of memory usage.

Disadvantages

① shift reduce has a limited lookahead it will miss some syntax error.

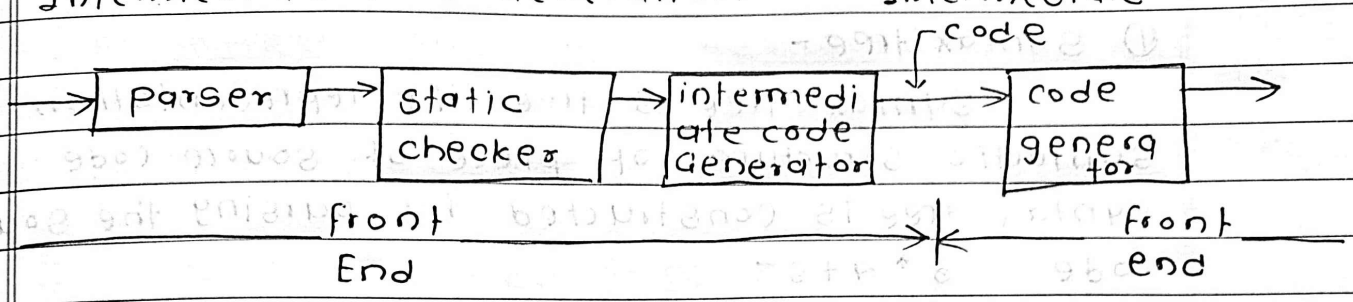
② It may also generate false-positive shift reduce conflicts.

	S Attributed	L Attributed.
①	Associated with synthesized attributes	Associated with both inherited & synthesized.
②	Bottom up parsing is used	Top Down parsing is used.
③	Semantic Rules are defined in bottom up	Semantic Rules are defined in Top down
④	Attributes calculated during reduction	Attributes calculated during parsing.
⑤	Occurs in LR(1) & LALR(1) parsing	Occurs in LL(1) parsing
⑥	Attributes values determined by production rules.	Attribute values determined by semantic Rules.
⑦	Implemented using shift reduce parsing	Implemented using Recursive descent parsing
⑧	Inherited from parents	Synthesized from children.
⑨	No circular Dependencies	Circular dependencies possible
⑩	Simple to implement	Complex to implement

D D M M Y Y Y Y

(11)	less expressive	more expressive
(12)	No need for symbol table	symbol table is required
(13)	suitable for simpler grammer	suitable for complex grammer
(14)	Does not handle left recursion	can handle left recursion

Intermediate Code Generation



① Intermediate code generation is fourth phase compilation.

② The front end translats a source program into an intermediate representation from which back end generates target code

③ parsing, static checking & intermediate code generation are done sequentially

④ Intermediate code is lies betn high level language & machine level language

⑤ Intermediate code generator takes input as syntax tree & generate intermediate code ~~generator~~

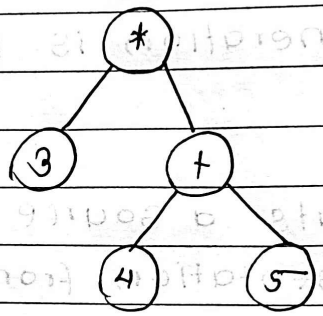
⑥ The compiler cannot generate machine code directly. Therefore first it convert source program into intermediate code

⑦ There are 3 Ways of intermediate representation

1. syntax tree
2. Postfix Notation
3. Three address code.

① Syntax tree -

Syntax tree is tree like representation of syntactic structure of piece of source code Syntax tree is constructed by parsing the source code 3 * 4 + 5



② Postfix Notation - postfix notation are linearized representation of syntax tree

Ex. a b c uminus * b c uminus * + assi

③ Three address code - Three address codes are type of intermediate code which is easy to generate & easily converted into machine code.

General Representation

a = b op c

a, b, c operands

op operator.

Example → d = a * b - c

$t_1 = a * b$

$t_2 = t_1 - c$

d = t_2

Advantages - optimize code
Platform independent
Simplify code

Disadvantages - add extra steps to complication.
Additional memory usage
Increased complexity.

Annotated parse tree

The parse tree containing value of attribute at each node for given input string is called annotated ~~syntax~~ parse tree

It is a data structure used to represent syntactic structure of program with additional information

Additional information contains datatype, attribute & name

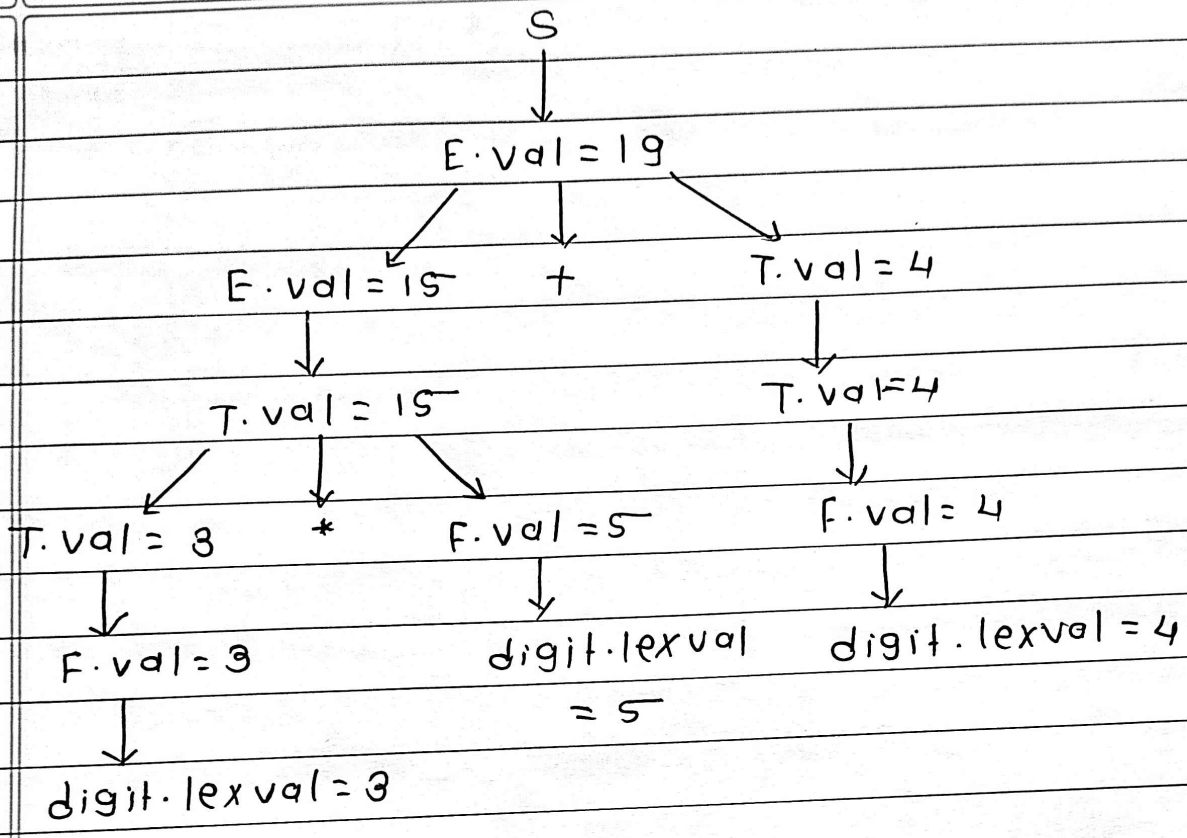
Annotated parse tree is useful at time of compilation because it include both syntax & semantic of program.

Features - High level specification
Hides implementation details.
Explicit order of evaluation.

Production	Semantic Rules
S → E return	print (E.val)
E → E + T	E.val = E.val * T.val
E → T	E.val = T.val
T → T * F	T.val = T.val + F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit

Where E = Represents Expression
T = term
F = Factor
digit = numeric value.

Annotated parse tree.



5. Code Optimization.

Q. 1 What are sources of optimization?

→ ① Code optimization is a process of eliminating redundancies from program & improving performance of program.

② It improves code by making it consume less resources & deliever high speed. (Reduce space & time)

Following are the code optimization transformation

① Elimination of common subexpression
② Copy propagation
③ Dead code elimination
④ Code Motion
⑤ Induction Variables & Reduction in strength.

1) Elimination of common subexpression.

An occurance of Expression E is called as common sub-expression if

① E was previously computed
② & values of variable in E have not changed by previous computation.

We can avoid recomputing expression E if we can use previously computed value.

Example.

a = b+c		a = b+c
d = a-d	⟶	d = a-d
c = b+c		c = a

The common subexpression b+c is eliminated in 3rd step as it computed already in a & value of expressions are not changed.

2) Copy Propagation

Copy propagation means eliminate a statement with new transformation.

DATE / /

Assignment in form of f := g is called as copy statements or copies for short.
The idea behind the copy propagation is transformation is to use g for f, whenever possible after the copy statement.
Copy propagation means use one variable instead of another.
Example.

```
x = a // copy stmt.      x = a
y = x + b      ⟶        y = a + b
z = x + c                z = a + c
```

Here variable x is replaced / eliminated by a

3) Dead code elimination
Dead code is nothing but a useless code of program.
A variable is useful in the program if its value is used in program frequently otherwise it is an dead code.
A main idea behind this is dead code - means the statements that computes values that never get used.
programmer is unlikely to introduce the dead code intentionally, it may appears as the result of previous transformations.

Example -

```
i = 0;
if (i = 1) {
    a = b + 5
}
```

Here if statement is dead code because this condition will never get satisfied.

4) Code Motion

- Code motion is loop optimization technique
- Code Motion moves the code outside of the loop.
- In loops especially in inner loops, programs tend to stend the bulk of their time.
- The running time of program may be improved if number of instructions isn inner loop is decreased.

Example

```
while(i<10)                    x = y+2;
{                              while(i<10)
  x = y+2;          ⟶          {
  i = i+1;                       i = i+1;
}                              }
```

the expression x=y+2 is loop invariant expression computation it is not based on loop. every time x value will be same & no of statement will reduced.

5) Induction Variable Elimination & Reduction in strength. (Induction variable is loop based variable i,j)

- Another important optimization is to find induction variable in loop & optimize their computation.

A variable x is said to be induction variable if there is positive or negative constant c such that each time x is assigned & its value is increased by c [The transformation of replacing a complex operation with simpler one is known as strength reduction. For ex. multiplication is replaced by addition.

Example.

```
i = 1                          t = 4;
while(i<10)                    while(t<40)
{                      ⟶       {
  t = i*4;                       t = t+4;
  i = i+1;                     }
}
```

DATE / /

In this above example i is induction variable loop will execute 9 times & value of t increased by 4 & multiplication operation t = i * 4 is replace with t = t+4 & induction variable is removed.

Q.2 What is peephole Optimization ?

→ ① peephole optimization is an optimization tech. performed on a small set of instructions / progra

② the small set is known as the peephole or window

③ Peephole optimization involves changing the small set of instructions to an equivalent set that has better performance.

④ simple but yet effective & improves the target code

⑤ Peephole optimization It is done by examining a sliding window of target ~~code~~ instructions & replace them with peephole shorter or faster sequence.

⑥ It can be done directly after intermediate code generation to improve intermediate Representation.

characteristics of peephole optimization-

1. Eliminate Redundant loads & stores.
2. Eliminate Unreachable code
3. Flow of control optimization
4. Algebric Simplification & Reduction in strength
5. Use of Machine Idioms.

① Eliminate Redundant loads & stores.

In target program we can delete store instruction because whenever it is executed, the first instruction will ensure that value of a is already loaded in Register Ro

```
LD   Ro, a   --- ①
ST   a, Ro   ... ②
```

DATE / /

if store instruction had a label, we could not remove sure that first instruction is always executed before second. So we could not remove store instruction. So we can put them in same basic block.

② Eliminate Unreachable Code -

Another opportunity for peephole optimization is removal of unreachable instruction. An unlabeled instruction immediately following an unconditional jump may be removed [Peephole optimization also used to eliminate jumps over jumps]

Example

```
        if debug==1 goto L1
        goto L2
    L1: print debug information
    L2:
```

debug is replaced with 0 then & performing constant propagation code will be

```
        if 0!=1 goto L2
        print debug information
    L2:
```

③ Flow of control optimization

Intermediate code generation algorithm frequently produce jumps to jump, jumps to conditional jumps or conditional jumps to jumps. These unnessary jumps can be eliminated

Example

```
    goto L1                    goto L2
    ....                       ....
    L1: goto L2      ⟶        L1: goto L2
    ....                       ....
    L2: a==b                   L2: a=b
```

There are two jumps & we removed one jump & also we can optimize the code by removing label L1.

If we see the instruction sequence
LD R0, a
ST a, R0
in a target program, we can delete the store instruction because whenever it is executed, the rst instruction will ensure that the value of a has already been loaded into register R0. Note that if the store instruction had a label, we could not be sure that the rst instruction is always executed before the second, so we could not remove the store instruction. Put another way, the two instructions have to be in the same basic block for this transformation to be safe.
Redundant loads and stores of this nature would not be generated by the simple code generation algorithm of the previous section. However, a naivecode generation algorithm like the one in Section 8.1.3 would generate redundant sequences such as these

DATE / /

④ Algebric specification & Reduction in strength.

algebric identities can also used by peephole optimizer to eliminate three address statement

Ex. $x = x + 0 \longrightarrow x = x * 1$

also reduction in strength transformation can be applied & replace complex / expensive operation by simple / cheaper operation

Ex.

complex / expensive	x^2	$= x * x$	simple / cheaper
	$2 * x$	$= x + x$	
	$x * 2$	$= x << 1$	
	$x / 2$	$= x >> 2$	

⑤ Use of Machine Idioms.

Target machine have hlw instructions to implement certain operation efficiently.

Detecting which operation should perform the use of the instructions can reduce execution time

Some machines have auto-increment & auto-decrement addressing mode.

Ex.

auto increment $i = i + 1 \longrightarrow$ INC i

Decrement $i = i - 1 \longrightarrow$ DEC i