Date __/__/__ Page no: __

Compiler :- software that converts a program written in high-level language (source code) to low-level language (object / ML / 0, 1's).

→ type of translator, which takes program written in high-level as inputs and translates it into an equivalent program in low-level such as ML or assembly language.

→ High-level (source program)
→ low-level (object or target)
→ verifies all types of limits, range, errors, etc)
→ Process of translating involves several stages :- lexical analysis, syntax A., semantic A., code generation & optimization.

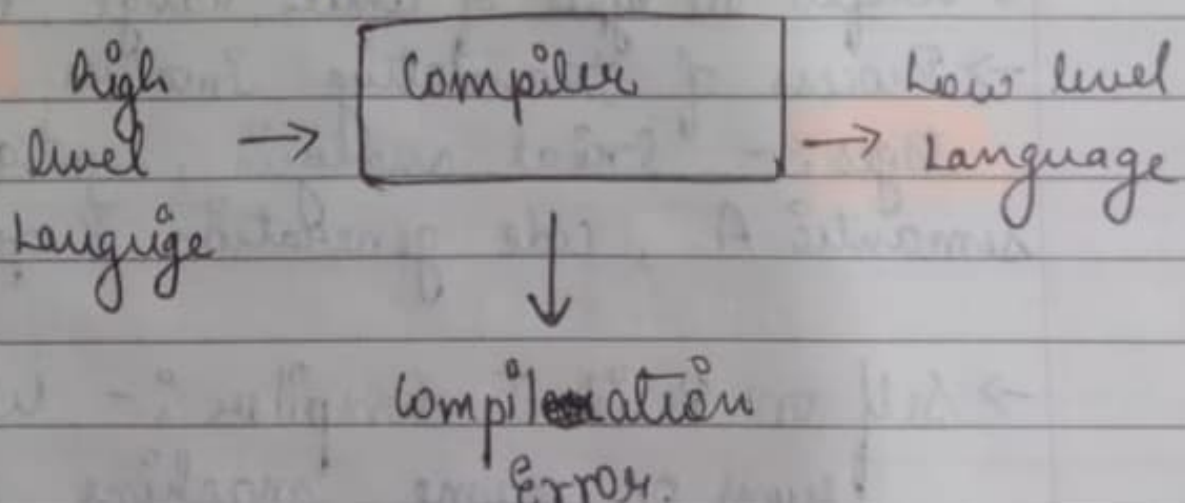
→ Self or Resident Compiler :- When compiler runs on same machine and produces machine code for the same machine on which it is running.
→ Cross Compiler :- Compiler may run on one machine and produces the machine codes for other computer then in that case it is called cross compiler.

Date: / / Page no:

→ High-level Programming language :- It is a lang. that has an abstraction of attributes of computer., more convenient to a user in writing a program.

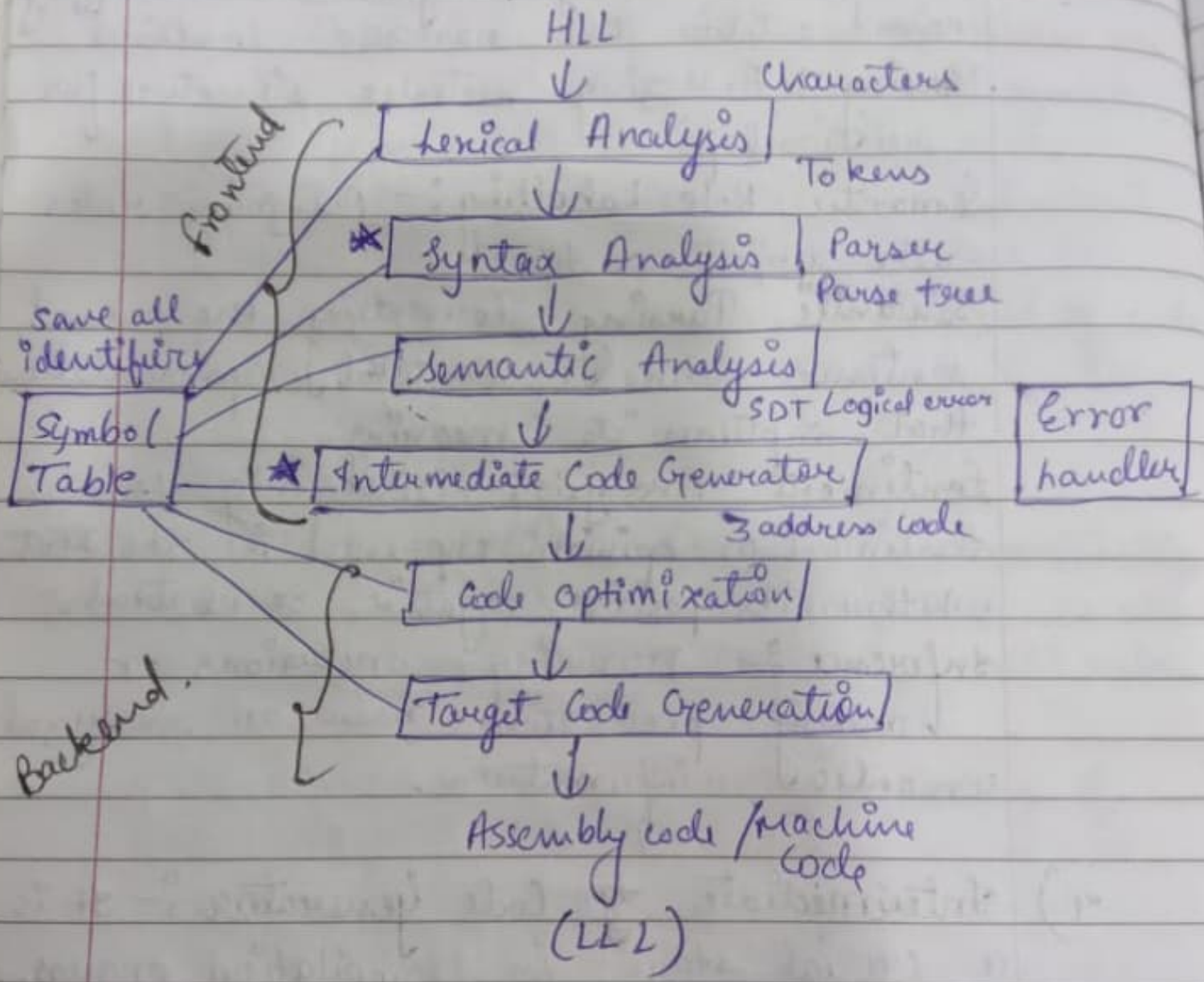
→ Low-level programming language :- It is a language that doesn't require programming ideas & concepts.

Overall, it a complex process that involves multiple stages.



Phases of Compiler.

Date: / / Page no:



lexemes are group of characters which has some meaning.

Date: / / Page no:

Lexical Analysis :- input - stream of characters
output :- stream of tokens.

Ex :- $y = 2 * x$ (2 : constant, * : operator, = : assignment, y : identifier)

→ remove comments, whitespaces
→ scanner (scans characters from left to right)
→ Finite automata, DFA, NFA
→ Tokenization [lexer, Tokenizer, Scanner]

Tokens
- identifier : x, y, z
- separator : , ; {} () []
- keywords : int, if, float
- operators : =, +, -, <, >
- constants (literals) : True, False, 2, 20
- special character : @, #, $

→ give error messages
- Exceeding length
- Unmatched string
- illegal character

Ex :- `int main ( ) { /* Find Max of a & b */ }`

(int : 1, main : 2, (: 3,) : 4, { : 5, /* Find Max of a & b */ : 5, } : 6)

`Printf ( " i = %d, &i = %x " , i , &i ) ;`

(Printf : 1, (: 2, " i = %d, &i = %x " : 3, , : 4, i : 5, , : 6, & : 7, i : 8,) : 9, ; : 10)

Date: / / Page no:

Compiler :- software that converts a program written in high-level language (source code) to low-level language (object / ML / 0, 1's).

→ type of translator, which takes program written in high-level as inputs and translates it into an equivalent program in low-level such as ML or assembly language.

→ high-level (source program)
→ low-level (object or target)
→ verifies all types of limits, range, errors, etc.
→ Process of translating involves several stages :- lexical analysis, syntax A., semantic A., code generation & optimization.

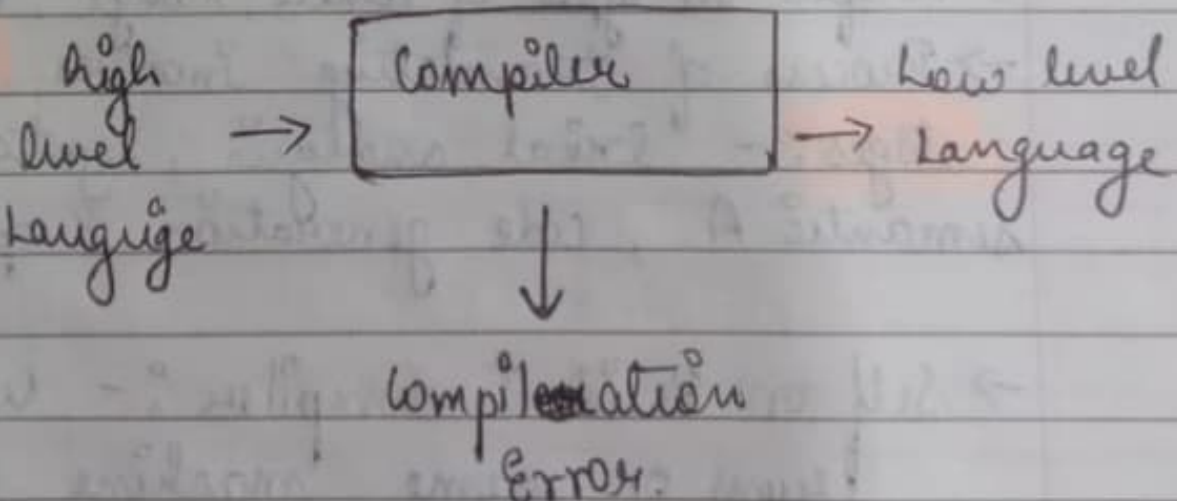
→ Self or Resident Compiler :- When compiler runs on same machine and produces machine code for the same machine on which it is running.
→ Cross Compiler :- Compiler may run on one machine and produces the machine codes for other computer then in that case it is called cross compiler.

Date: / / Page no:

→ High-level Programming language :- It is a lang. that has an abstraction of attributes of computer., more convenient to a user in writing a program.

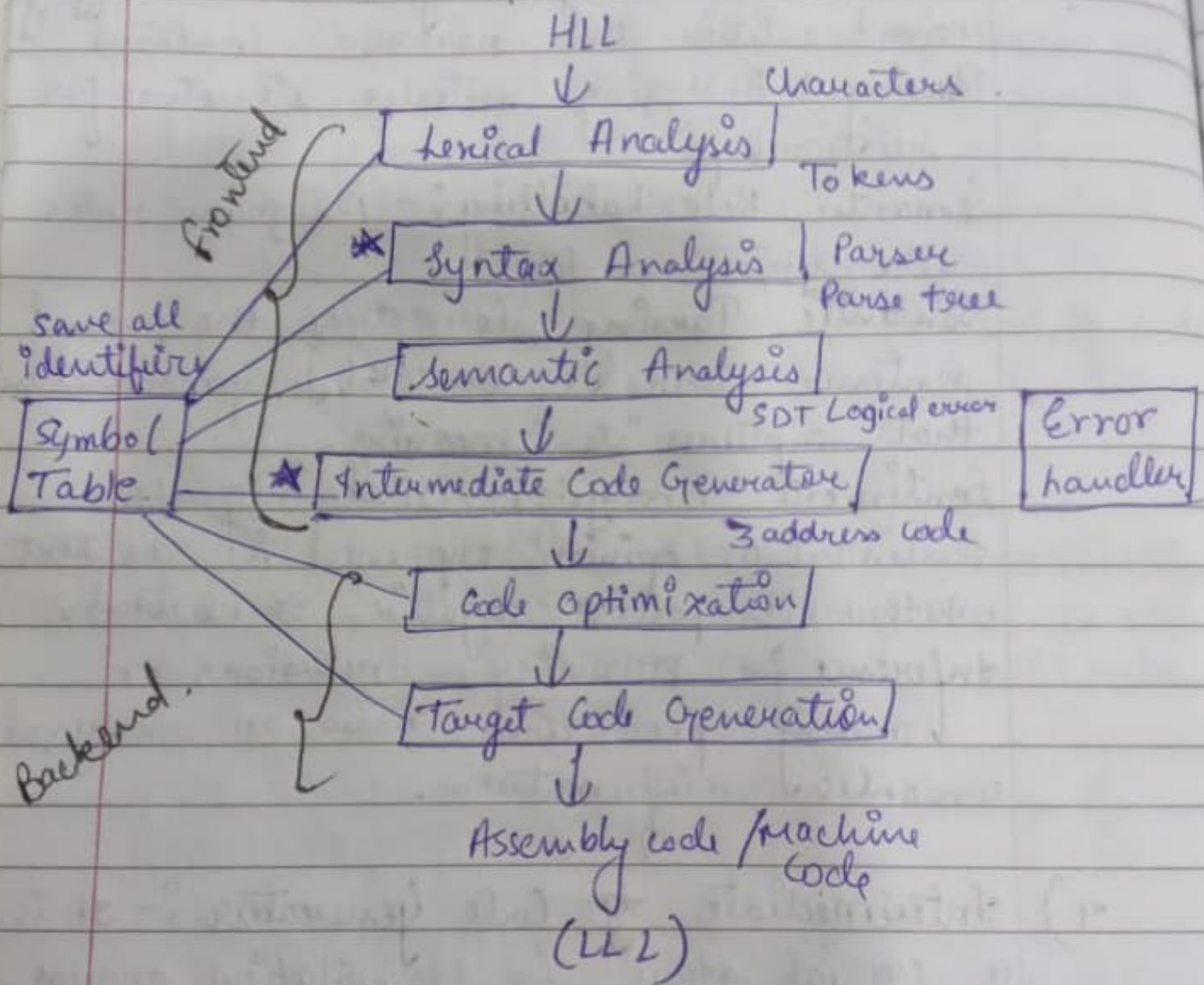
→ Low-level programming language :- It is a language that doesn't require programming ideas & concepts.

Overall, it a complex process that involves multiple stages.



Phases of Compiler.

Date: / / Page no:



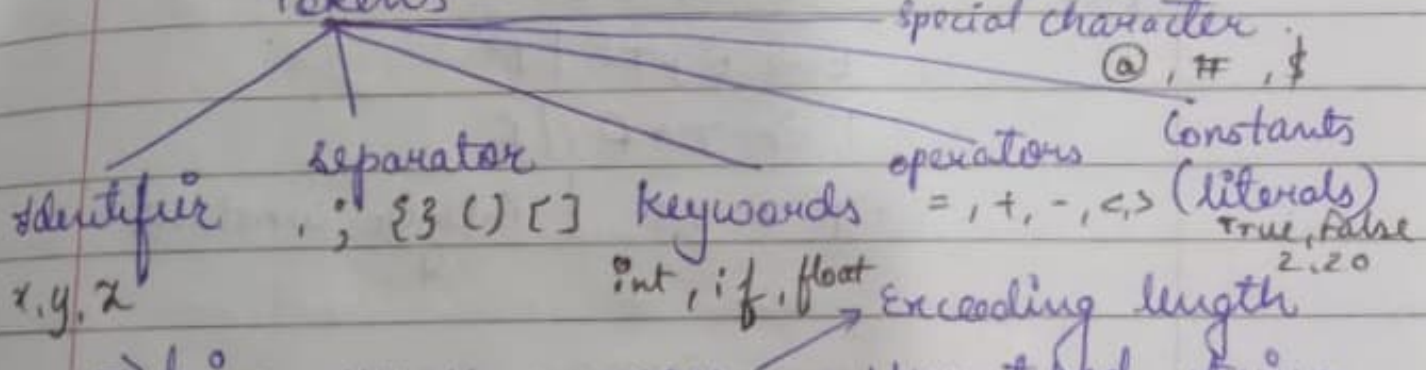
lexemes are group of characters which has some meaning.

Date: / / Page no:

Lexical Analysis :- input - stream of characters
output :- stream of tokens.

Ex :- $y = 2 * x$ (constants, operator, assignment, identifier)

→ remove comments, whitespaces
→ scanner (scans characters from left to right)
→ Finite automata, DFA, NFA
→ Tokenization [lexer, Tokenizer, Scanner]



→ give error messages → Exceeding length
→ Unmatched string
→ illegal character

Ex :- int main ( )
{
/* Find Max of a & b */ ;X
}

Printf ("i = %d, &i = %x", i, &i);

Date: / / Page no:

2) Syntax Analyzer.

→ 'Parser'

→ output of lexical analyzer is its input.

→ checks for syntax error

→ does by constructing parse tree of all tokens.

→ CFG, the parse tree should be according to the rules of source code grammar.

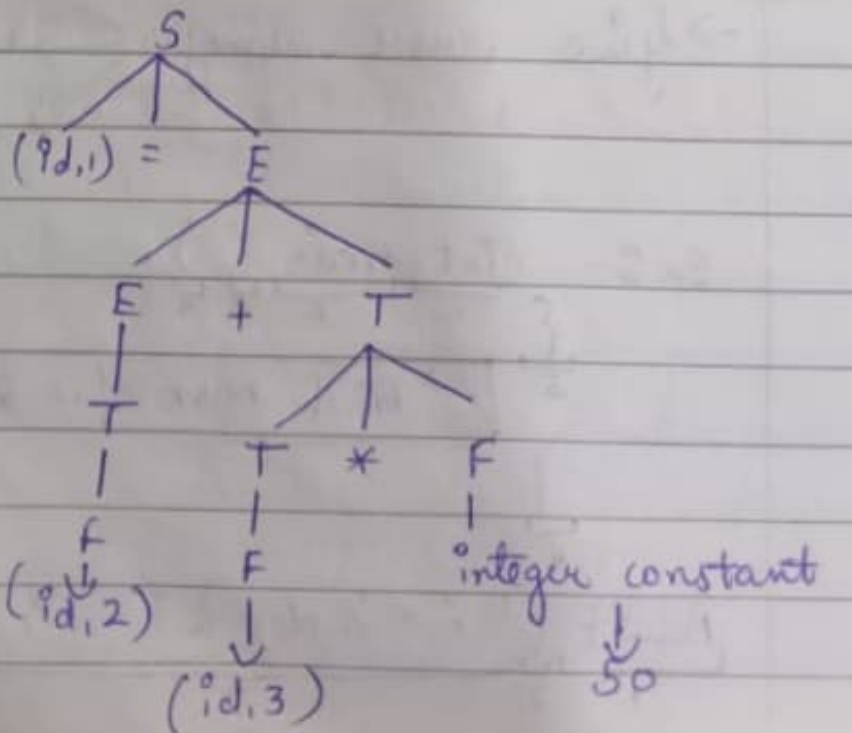
Ex:-

$S \rightarrow id = E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * f \mid f$

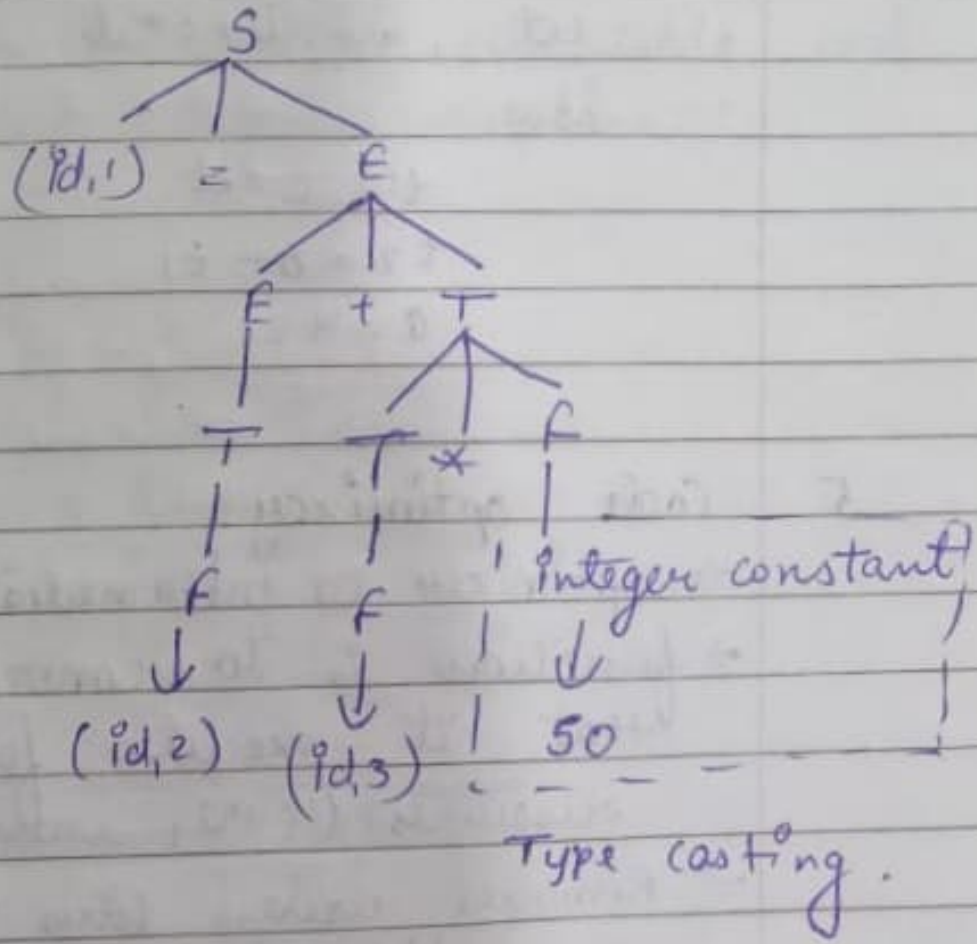
$f \rightarrow id \mid$ integer constant



Date __/__/__ Page no.__

3) Semantic Analysis :-

→ verifies parse tree of the syntax Analyzer.
→ checks validity of code in terms of PL like data types, declaration, etc.
→ produces verified parse tree, we called it annotated parse tree.
→ also performs flow checking, Type checking etc.



Date / / Page no.

4. Intermediate Code Generation.

→ this code is neither high level language nor machine. It is in intermediate form.

→ converted in machine L. but, last two phases are platform dependent.

→ Intermediate code is same for all the compilers. further, we generate the machine code according to platform.

→ Ex → 3 address code.

Let, $a = b + c * d$

now,

$t1 = c * d$

$t2 = b + t1$

$a = t2$

5. Code optimizer.

→ optimizer on intermediate code.

→ function is to convert the code so that it executes faster using fewer resources (CPU, memory)

→ Removes useless lines of code & rearranges the code

→ meaning of source code remains same.

Ex → $t1 = c * d$
$a = b + t1$

6. Target Code Generator :-
→ final phase, converts optimized intermediate code into the machine code.
→ machine code which is produced is relocatable.
→ produces the final executable code for the target machine.

Ex :-

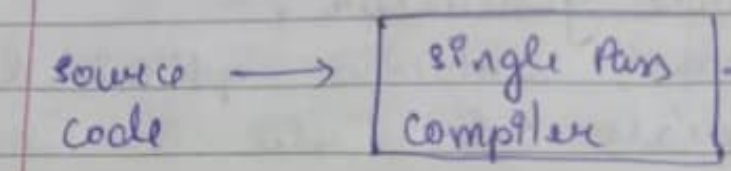
```
MOVE    R2, #d2
MUL     R2, 50
MOVE    R1, #d1
ADD     R1, R2
MOVE    #d, R1
```

← Processing move.

Types of Compiler.

Date: / / Page no.

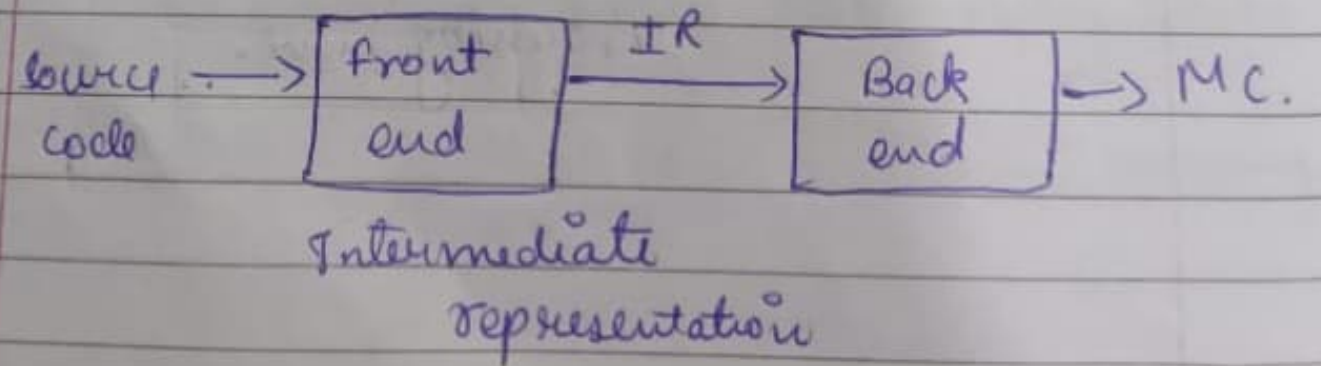
1) Single-Pass Compiler :- combines all the compiler phases in a single module.



→ Compilers read source code once & generate the target code in a single pass through the source code.
→ generally faster but may lack certain optimizations.

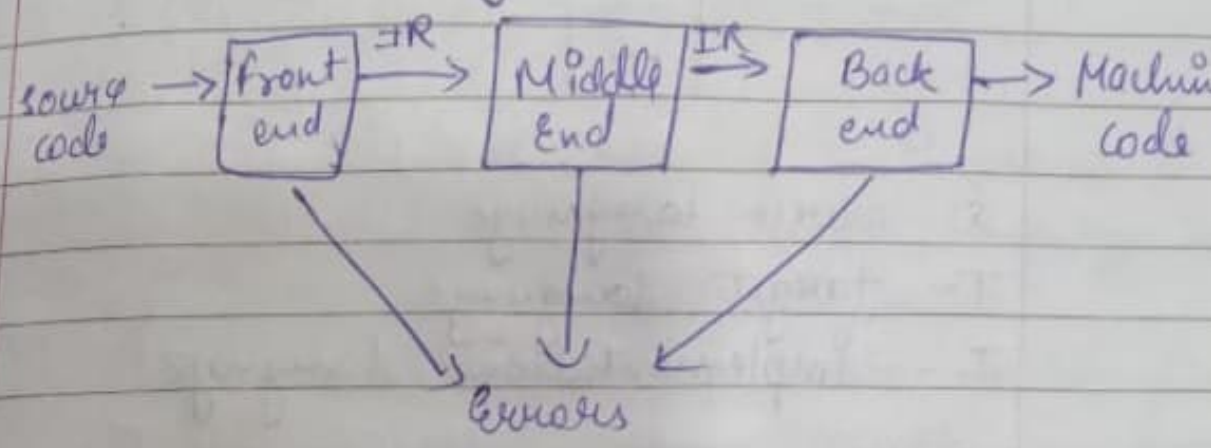
2) Two-Pass Compiler :- processes the source code into machine code in two passes.

→ requires two passes to scan the source code & performs translation.



Date: / / Page no:

3) Multipass Compiler :- processes source code into multiple passes.
→ huge program is broken into various small programs & all the programs runs simultaneously.



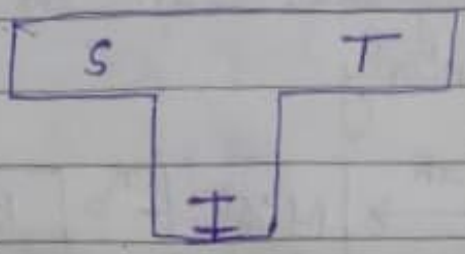
- produce more efficient code but may take longer to compile.

4) Cross Compiler :- A cross compiler is a tool that translates source code from one programming language to another, generating executable code that runs on a different platform than the one it's running on. For example, it could compile code on a Windows computer to run on a Linux server. It is used in bootstrapping, for microcontrollers, for embedded

Date: / / Page no:

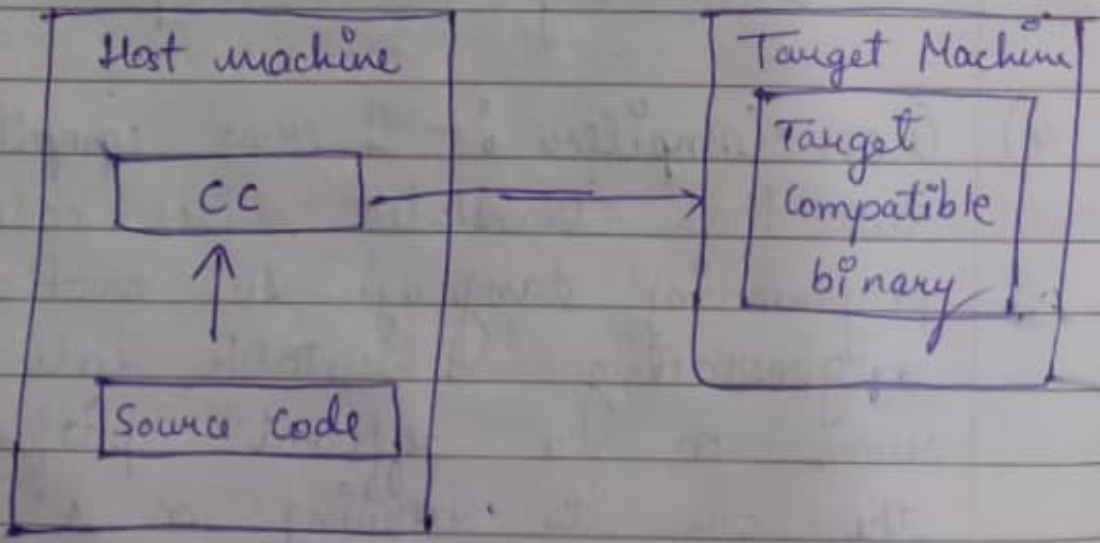
computers, ∮, etc.

T-Diagram for CC



S - source language
T - target language
I - Implementation language.

CC operation



Date: / / Page no:

for example:- a compiler that runs on windows PC but generates code that runs on Android smartphones is a cross compiler.

Uses:- 1) separate the build environment from target environment.

5) Bootstrapping :- A process in which simple language is used to translate more complicated program which in turn may handle for more complicated program.

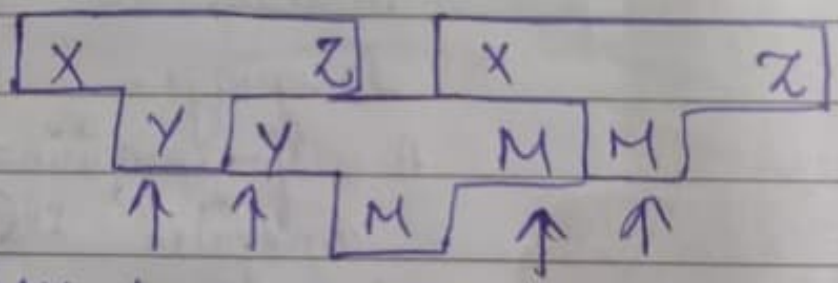
→ It is a self-compiling compiler, written in its source language.

A self-compiling compiler compiles its source code.

This compiled compiler can compile everything else & its future versions as well.

Date: / / Page no:

Ex:- Suppose we want to write a cross compiler for new language X.
The implementation language of this compiler is Y and the target code being generated is in language Z.
i.e. we create X_YZ.
Now, if existing compiler Y [i.e. compiler written in language Y] runs on machine M & generate code for M, then it is denoted as Y_MM.
Now, if we run X_YZ & Y_MM then we get a compiler X_MZ. That means a compiler for source language X that generates target L.(X) & runs on machine M.



These two languages must be same.

Date / / Page no.

Single Pass	Multi Pass
→ Reading source code once & in one go.	→ Makes multiple passes through the source code.
→ Faster	→ Slower
→ less memory usage	→ more
→ Limited error checking	→ Comprehensive error checking
→ Limited optimization	→ Allows for more advanced optimization
→ Suitable & good for simple languages & quick compilation.	→ for complex L.
→ Simpler architecture	→ more complex.
→ Easy to implement	→ require more careful design & implementation

★ One complete scan of source ~~code~~ language is called Pass.

Date: / / Page no:

Features of Good Compilers

→ less time
→ less amount of memory space
→ compile only modified code segment
→ while handling HW interrupts the good compiler interact with OS.

Error Detection :- Whole Compilation & phases story.
+ the errors are reported in the form of messages.

Symbol Table :- store identifiers also store information about attributes of identifiers
- its type, scope, size, storage, etc
- also stores info. about the subroutines.
- data structure
→ compilation phases story.

Compiler	Interpreter
→ Converts entire program into machine code.	→ executes source code line by line.
→ o/p - generates executable files.	→ No separate executable file is generated.
→ Generally faster execution as code is pre-compiled.	→ slower as code is interpreted.
→ All errors are detected before execution.	→ Errors are detected at runtime.
→ less memory	→ more due to runtime environment
→ requires re-compilation for different platforms.	→ Usually portable across platforms.
→ Longer Development time due to compilation process.	→ Faster development

$A \to A\alpha \mid \beta$ so

$\to A \to \beta A'$

$\to A' \to \alpha A' \mid \varepsilon$

Top down parser does not accept LR

Date / / Page no.

LL(1)

- Left: Left right Scan input symbol
- Left most derivation
- No. of lookahead symbol to take a decision.

Q.) $E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid id$

$A = E$

$\alpha = +T$

$\beta = T$

Now,

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

$A = T$

$\alpha = *F$

$\beta = F$

$F(E) = \{ (, id \}$

$F(E') = \{ +, \varepsilon \}$

$F(T) = \{ (, id \}$

$F(T') = \{ *, \varepsilon \}$

$F(F) = \{ (, id \}$

$FO(E) = \{ \$,) \}$

$FO(E') = \{ \$,) \}$

$FO(T) = \{ +, \$,) \}$

$FO(T') = \{ +, \$,) \}$

$FO(F) = \{ *, +, \$,) \}$

Parsing Table.

Date: / / Page no:

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

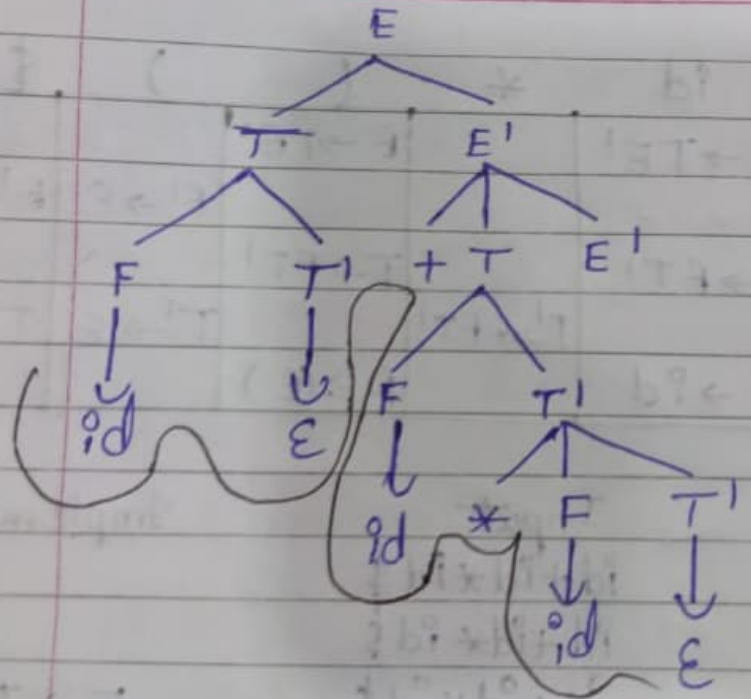
id + id * id

Stack	Input	Implement.
$	id + id * id $	
E $	id + id * id $	
TE' $	id + id * id $	$E \to TE'$
FT'E' $	id + id * id $	$T \to FT'$
id T'E' $	id + id * id $	$F \to id$
T'E' $	+ id * id $	$T' \to \varepsilon$
E' $	+ id * id $	$E' \to +TE'$
+TE' $	+ id * id $	
TE' $	id * id $	$T \to FT'$
FT'E' $	id * id $	$F \to id$
id T'E' $	id * id $	$T' \to FT'$
* FT'E' $	* id $	$F \to id$
id T'E' $	id $	$T' \to \varepsilon$
E' $	$	$E' \to \varepsilon$
$	$	

$ = $

Parse Tree

Date: / / Page no:



id + id * id

R - L = Follow. S → { $ }

R - R = first. (E) X

E → TE'

E' → +TE' | ε.

T → FT'

T' → *FT' | ε.

F → (E) | id.

- Right side.

E = ? . E' = ?

F (E) = { (, id }

F (E') = { +, ε }

F (T) = { (, id }

F (T') = { *, ε }

F (F) = { (, id }

FO (E) = { $,) }

FO (E') = { $,) }

FO (T) = { +, $,) }

FO (T') = { +, $,) }

FO (F) = { *, +, $,) }

$S \rightarrow (A)a$ $S \rightarrow \{a, a\}$

Date: / / Page no:

$S \rightarrow \{\$\}$

$S \rightarrow \{\varepsilon\}\ X$

R – R → First

R – L → follow.

Date: / / Page no:

Q $S \rightarrow ACB \mid CbB \mid Ba$
$A \rightarrow da \mid BC$
$B \rightarrow g \mid \varepsilon$
$C \rightarrow h \mid \varepsilon$

$f(S) \rightarrow \{ d, g, h, \varepsilon, b, a \}$
$f(A) \rightarrow \{ d, g, h, \varepsilon \}$
$f(B) \rightarrow \{ g, \varepsilon \}$
$f(C) \rightarrow \{ h, \varepsilon \}$

$fo(S) \rightarrow \{ \$ \}$
$FO(A) \rightarrow \{ h, g, \$ \}$
$fo(B) \rightarrow \{ \$, a, h, g \}$
$fo(C) \rightarrow \{ g, \$, b, h \}$

Date: / / Page no:

$S \rightarrow aBDh$
$B \rightarrow cC$
$C \rightarrow bC / \varepsilon$
$D \rightarrow EF$
$E \rightarrow g / \varepsilon$
$F \rightarrow f / \varepsilon$

$F(S) \rightarrow \{a\}$
$F(B) \rightarrow \{c\}$
$F(C) \rightarrow \{b, \varepsilon\}$
$F(D) \rightarrow \{g, f, \varepsilon\}$
$F(E) \rightarrow \{g, \varepsilon\}$
$F(F) \rightarrow \{f, \varepsilon\}$

$Fo(S) \rightarrow \{\$\}$
$Fo(B) \rightarrow \{g, f, h\}$
$Fo(C) \rightarrow \{g, f, h\}$
$Fo(D) \rightarrow \{h\}$
$Fo(E) \rightarrow \{f, g, h\}$
$Fo(F) \rightarrow \{h\}$

Ambiguous
Top-down
without backtracking

Q $S \to AaAb \mid BbBa$ $F(S) = \{a, b, \varepsilon\}$

$A \to \varepsilon$ $f(A) = \{\varepsilon\}$ $Fo(A) \to \{a, b\}$

$B \to \varepsilon$ $f(B) = \{\varepsilon\}$ $Fo(B) \to \{b, a\}$

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

Q $E \to TE'$ $F(E) \to \{id, (\}$

$E' \to +TE' \mid \varepsilon$ $F(E') \to \{+, \varepsilon\}$

$T \to FT'$ $F(T) \to \{id, (\}$

$T' \to *FT' \mid \varepsilon$ $F(T') \to \{*, \varepsilon\}$

$F \to id \mid (E)$ $F(F) \to \{id, (\}$

If start symbol then follow

$Fo(E) \to \{\$,)\}$

$Fo(E') \to \{\$,)\}$

$Fo(T) \to \{+, \$,)\}$

$Fo(T') \to \{+, \$,)\}$

$Fo(F) \to \{*, +, \$,)\}$

$Fo(E) \to \{\$,)\}$

$Fo(E') \to \{\$,)\}$

$Fo(T) \to \{+, \$,)\}$

$Fo(T') \to \{+, \$,)\}$

$Fo(F) \to \{*, +, \$,)\}$