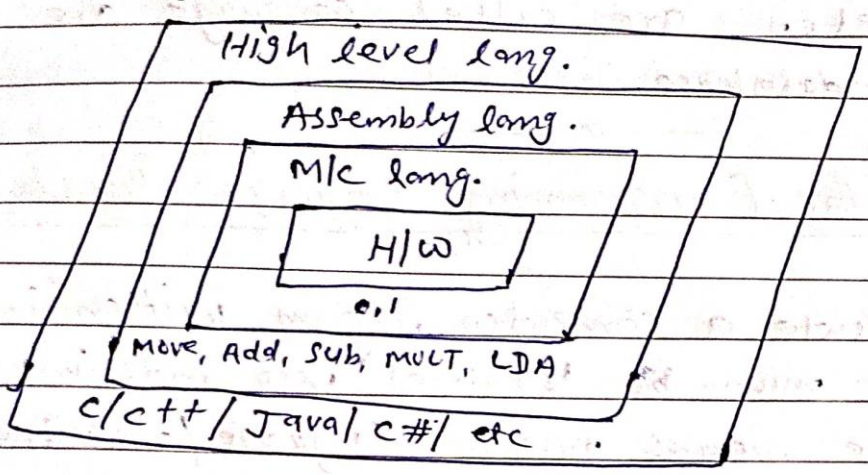
"श्री गणेशायनम : ॐ नमः शिवाय D"

Page No.:
Date:
YOUVA

Unit-I

* Why we study of compiler [Importance of compiler Design] ⇒



⇒ Machine Language ⇒ This language contains only 0's & 1's {0,1} instructions code. It is not required any type of translator. But it's very difficult for programmer to written programming code in this lang.

⇒ Assembly language ⇒ It is similar to machine language {ADD, SUB, MOVE, MULT, LDA} but it's different to machine language (it's only combination of 0's & 1's). It's used new symbols like Add, Move, Sub, MULT, LDA, etc.
It's required Assembler for decode the programming code into machine code.
It's difficult language for programmer But less difficult as machine language. It's required memorisation for complex & large program for more instruction.

⇒ High level language ⇒ Today we use this language on all s/w. means all s/w made in High level language.
It is user friendly language, easly user understandable.
High level language required translator Because translator, translate the High level language into low level language (M/c language)
Example:- C/C++/ Java/ C# etc.

⇒ 4 GL (fourth Generation language) ⇒ 4 GL are closer to Human language. It is

easy to learn the language like :- oracle, vc++, SQL etc. are called language are used to access database.

→

* Translator [Programming language translator] ⇒

To execute a computer program written in High level language must be translated into machine understandable language means machine language / machine code.

" A translator is a program that takes as a input program written in one of the programming language (source language / source code) & produces as a output a program in another language (object / machine language / Target code)

If the source language is High level language such as FORTRAN, C, C++, Java, C# etc & the object language is low level language such as assembly language / machine language.

⇒ Types of translators ⇒ There are mainly three types of programming language translators :-

- Assembler
- Compiler
- Interpreter

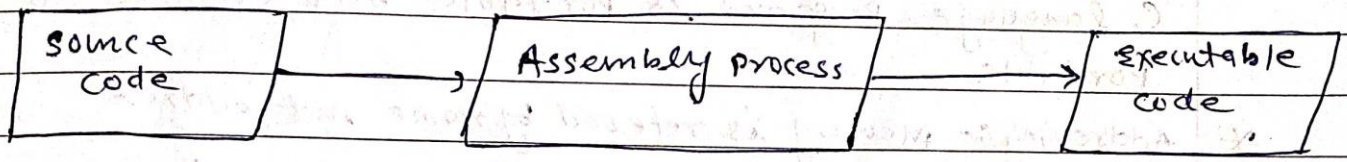
* Assembler ⇒ Assembler is a translator but it translate high level language into Assembly language.

Assembly language means which is basically mnemonics (स्मृति सहायक) code like GO, HALT, JUMP, NOT, MOVE, LDA, ADD, SUB MULT etc code.

It translate machine language into hexa decimal digit code.

Assembler एक program (s/w) होता है जो Translator machine dependent होता है

Assembler convert assembly language into pattern of bits that the computer processor can use to perform it's basic operations. This bit pattern is basically machine language.



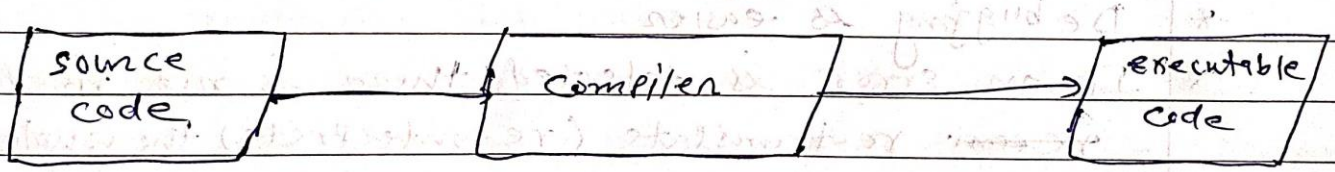
Assembly lang. | Assembler + Assembly lang. | Bit Pattern (or) Machine lang.

Example:- NASM, MASM (it's Microsoft assembler)

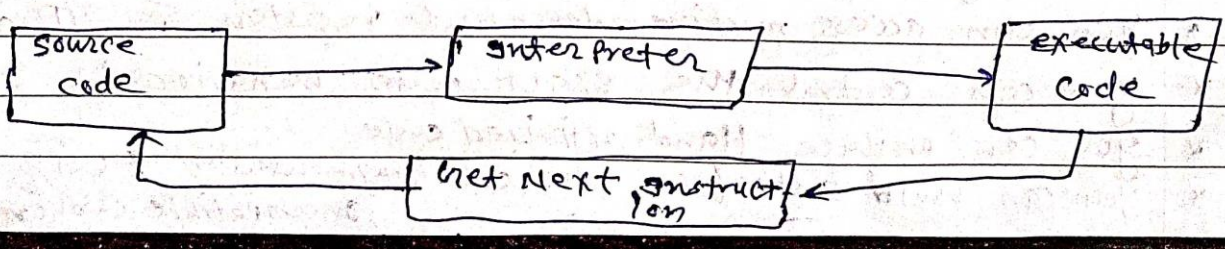
* Compiler ⇒ Compiler is a program that translates a high level language into machine code.

Example:-
The turbo Pascal Compiler translates a program written into machine code that can be run on a Personal Computer (PC) Example:- c/c++/ FORTRAN

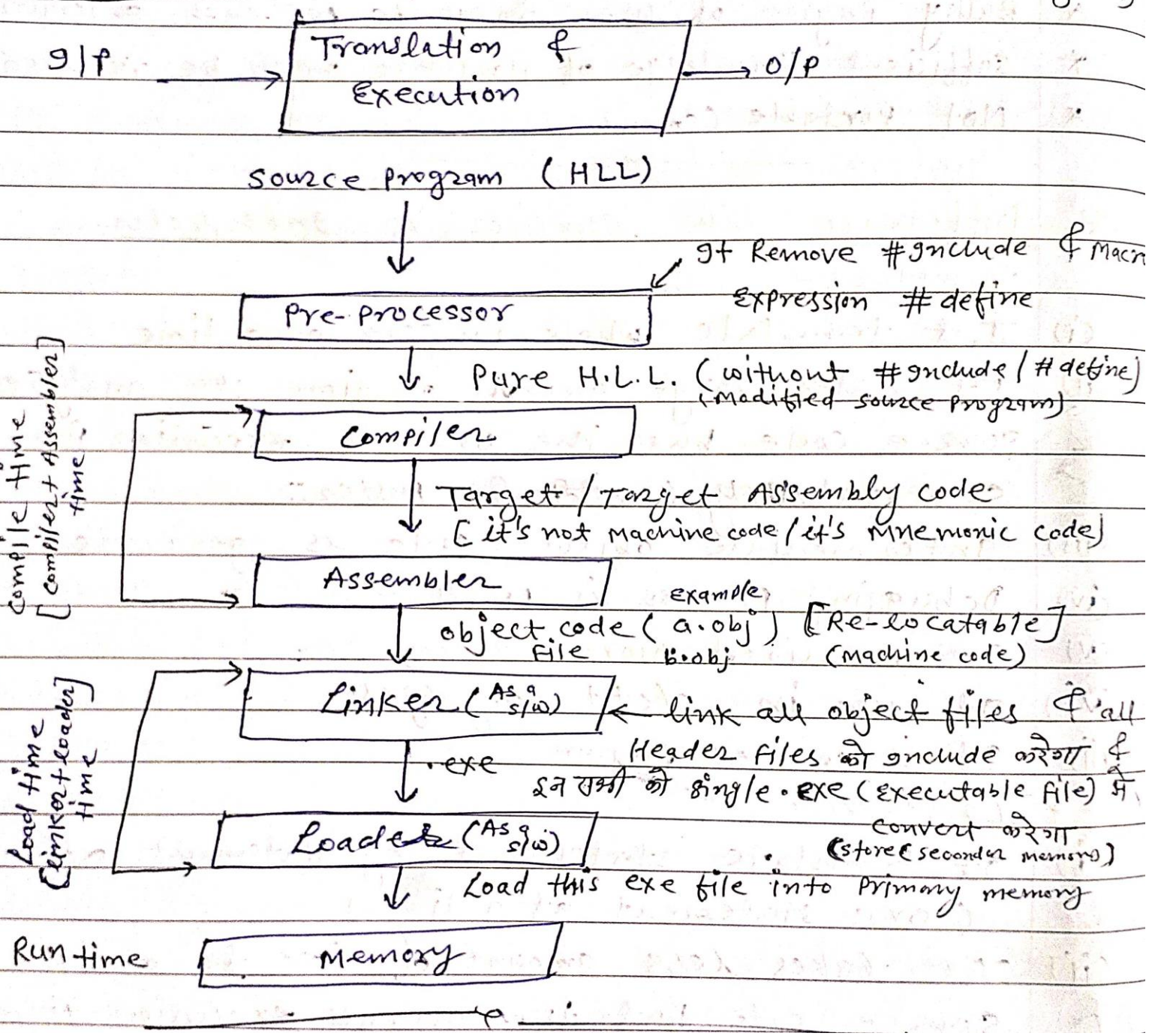
Compiler Platform independent



* Interpreter ⇒ An Interpreter is a another types of translator which convert high level language into machine code.
Interpreter is also a program. Main difference b/w Compiler & Interpreter is that an Interpreter translates one line at a time & then execute it. No object code is produced. So program has to be interpreted each time it to be run. Example:- LISP/Python/BASIC

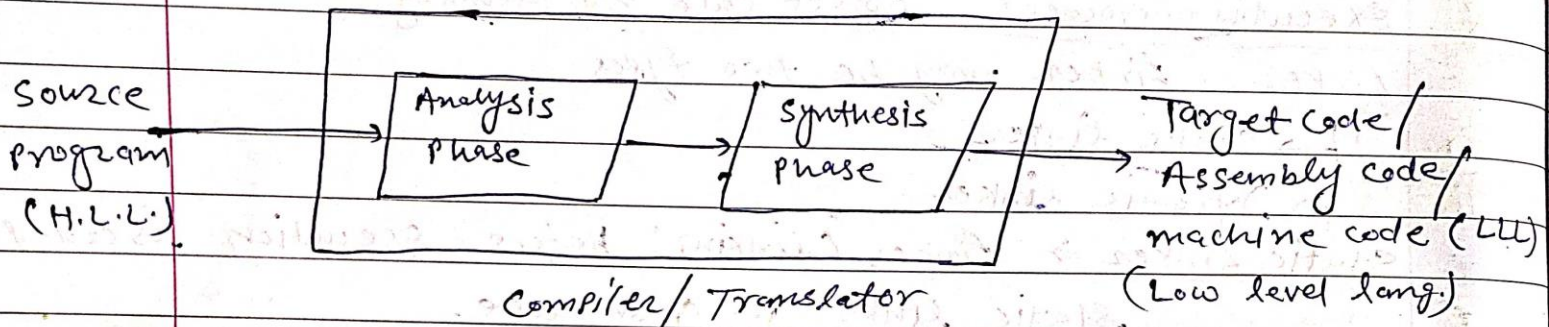


Language processing system & stages of translation & execution of program



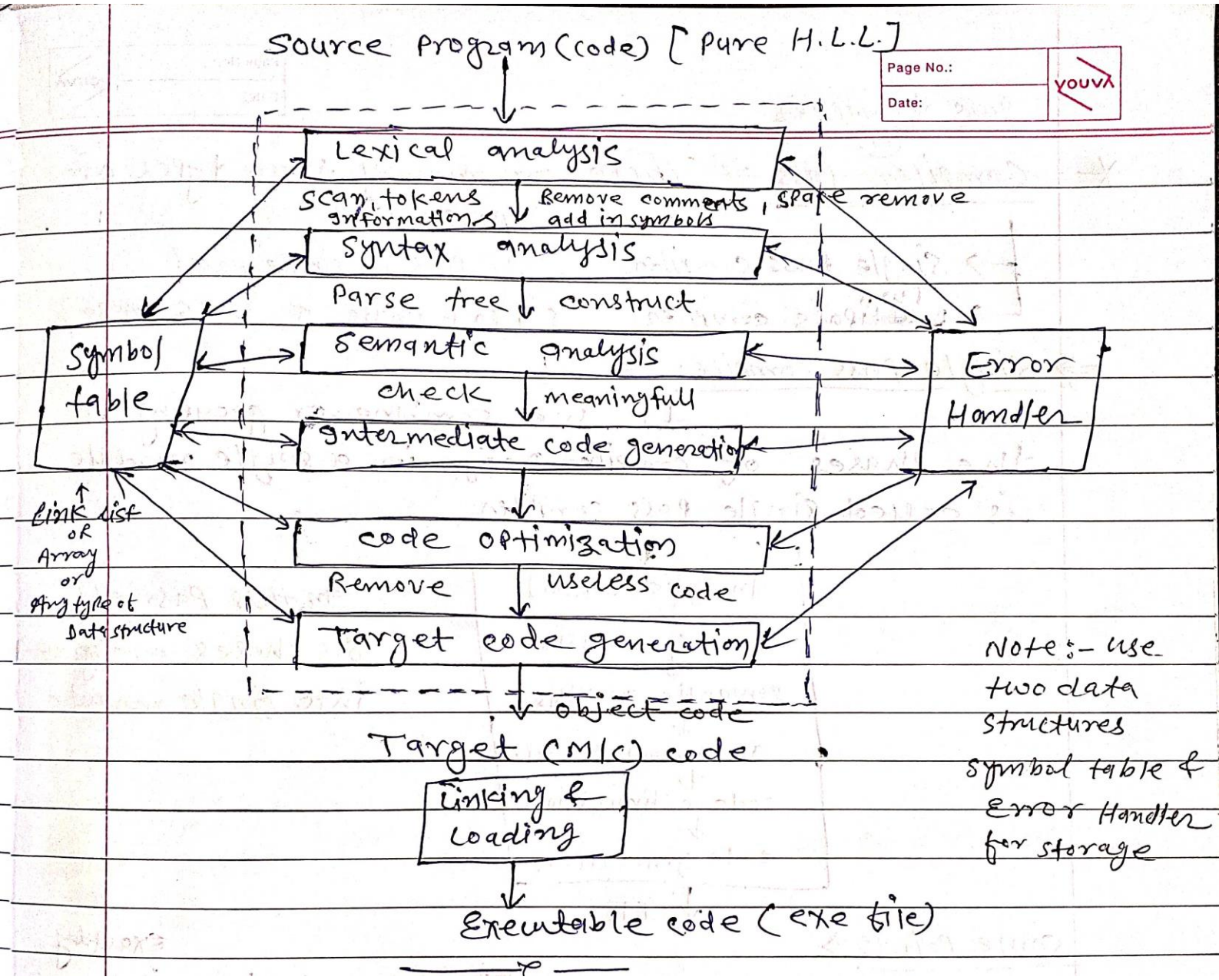
Compiler stages / stages in translation / compiler phases

Compilers are mainly two phases
- Analysis phase
- Synthesis phase



* **Analysis Phase** ⇒ This phase is used for source program recognization phase. It consists of three phases :-
  - Lexical analysis
  - Syntax analysis
  - Semantic analysis

* **Synthesis phase** ⇒ This phase is used for executable object program recognization phase. It consists of three phases :-
  - Intermediate code generation
  - code optimization
  - Target code generation


Source Program (code) [Pure H.L.L.]
Lexical analysis
Scan, tokens information
Remove comments, add in symbols
Space remove
Syntax analysis
Parse tree construct
Symbol table
Semantic analysis
Error Handler
check meaningfull
Intermediate code generation
Link list or Array or any type of Data structure
code optimization
Remove useless code
Target code generation
object code
Target (M/c) code
Linking & Loading
Executable code (exe file)
Note :- use two data structures symbol table & Error Handler for storage

* Analysis Phase ⇒ This phase is used for source program recognization phase. It consists of three phases :-

- Lexical analysis
- Syntax analysis
- Semantic analysis

* Synthesis Phase ⇒ This phase is used for executable object program recognization phase. It consists of three phases :-

- Intermediate code generation
- code optimization
- Target code generation

(1) Lexical analysis ⇒ The initial phase of any Translator/compiler is a lexical analysis phase. It's main task is scanning & generating tokens & remove comments & variable information stored in symbol table. remove spaces

Example :- int a, b ;

Delimiters

int → Keyword; a, b → Variables

1	a	int
2	b	int

Remove comments like /* ... */ // (OR)

Page No.:
Date:

Example :- a + b * 60 ;

Variables, Constant value, Delimiters, operators

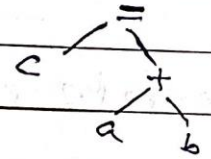
Note :- INT लिखने पर error generate होगा क्योंकि C lang. is not allow this type of INT, C allows only int

(2) Syntax Analysis ⇒ This is the second phase of compiler it construct Parse tree of tokens (lexical analysis tokens)

Parser construct Parse tree

Example :- c = a + b

Tokens are generate so Parse tree construct



Ex:- a + 1b
~~lexical not generate~~ error because tokens are generate But syntax analysis generate error because not construct Parse tree

Example :-

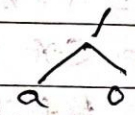
```
while (1)
{
Printf ("Hello");
}
```

This program lexically are correct because tokens are generate but syntax error are generate because syntax Mistake while (1) & ~~Hello~~ Printf ("Hello");

(3) Semantic analysis ⇒ It is the central Phase of compiler. It check the meaning full statement & maintenance symbol table, error detection & correction. It check meaning full tree

Example :- a / 0

Parse tree ⇒



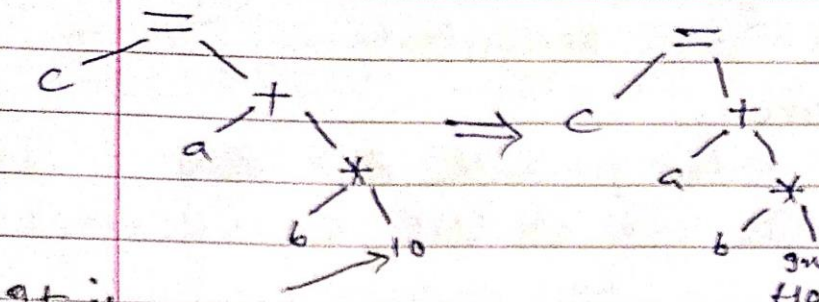
Lexically are correct because it generates token, syntaxically is correct because it generate Parse tree But it is not correct semantically because it's not meaning full Value of a/0 [किसी भी number का divide zero, C language में allow नहीं करता है]

Example :- float a, b, c;
c = a + b * 10;

⇒

Symbol table

1	a	Float
2	b	Float
3	c	Float



It is an integer value to generate error so 10 is convert into float value 10.0

Logically correct because token are generate & syntaxically is correct because syntax are construct parse tree But semantically is not correct because

Note:- अगर कोई syntax logically, syntaxically & semantically are correct तो बाकी के आगे बचे हुए Phases में Error आने के chance बहुत कम हो जाता है।

---

(4) Intermediate code generation ⇒ It generate the intermediate code. It divide the complex statement into smallest sub statement using TAC (three address code) technique [Maximum three address code]

Example:- $R = a + b + c * d$

$t_1 = c * d$, $t_2 = b + t_1$, $t_3 = a + t_2$, $R = t_3$

(5) Code optimization (it's optional phase) ⇒

It remove the unusuable code which not affected on actual result. It's optional phase

Example:- In above Phase Example

$R = a + b + c * d$

$t_1 = c * d$
$t_2 = b + t_1$
$t_3 = a + t_2$, $R = t_3$

optimize ⟹

$t_1 = c * d$
$t_2 = b + t_1$
$R = a + t_2$

(6) Target code generation ⇒ It is the last phase of any compiler (translator) (OR (Assembly code)) It's generate target code (mnemonic code) M/c code

Example:- Move a, R1

Pass of compiler

Date:

(or)

* Compiler pass ⇒ There are mainly two types of compiler pass :-
  - → Single pass compiler {All pass in single pass}
  - → Two pass compiler {1 to 4 phase & 5 & 6 phase}

⇒ Single pass compiler ⇒

If we combine or grouping all the phases of compiler Design in a single module

Example :- Float a, b, c;

a = b + c * 60

① Lexical analysis

↓

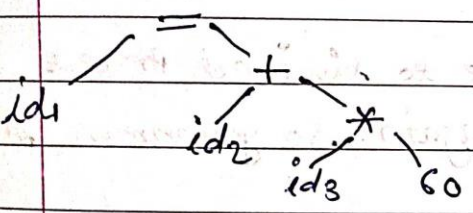
$id_1 = id_2 + id_3 * 60$

id_1 op_1 id_2 op_2 id_3 op_3 constant

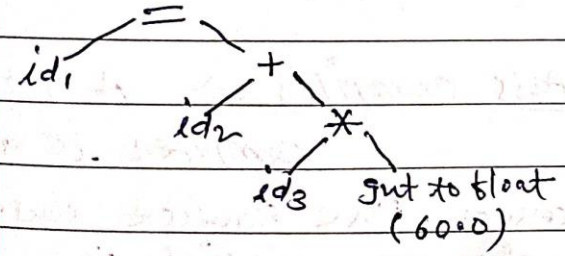
Total 7 tokens generates

② Syntax Analysis

↓



③ semantic analysis



④ Intermediate code generator

↓

$t_1 = 60.0$, $t_2 = id_3 * t_1$,

$t_3 = id_2 + t_2$

$id_1 = t_3$

⑤ Code optimization

↓

$t_1 = id_3 *$ int to float (60.0)

$id_1 = id_2 + t_1$

⑥ Target code generator

↓

MOVE id_3, R_2

MULT 60.0, R_2

MOVE id_2, R_1

ADD R_2, R_1

MOVE R_1, id_1

→ We can not optimize very well due to limited process

→ We can not backup & process. It again so grammar should be limited or simplified.

→ Command interpreters such as bash|sh|tcsh can be consider

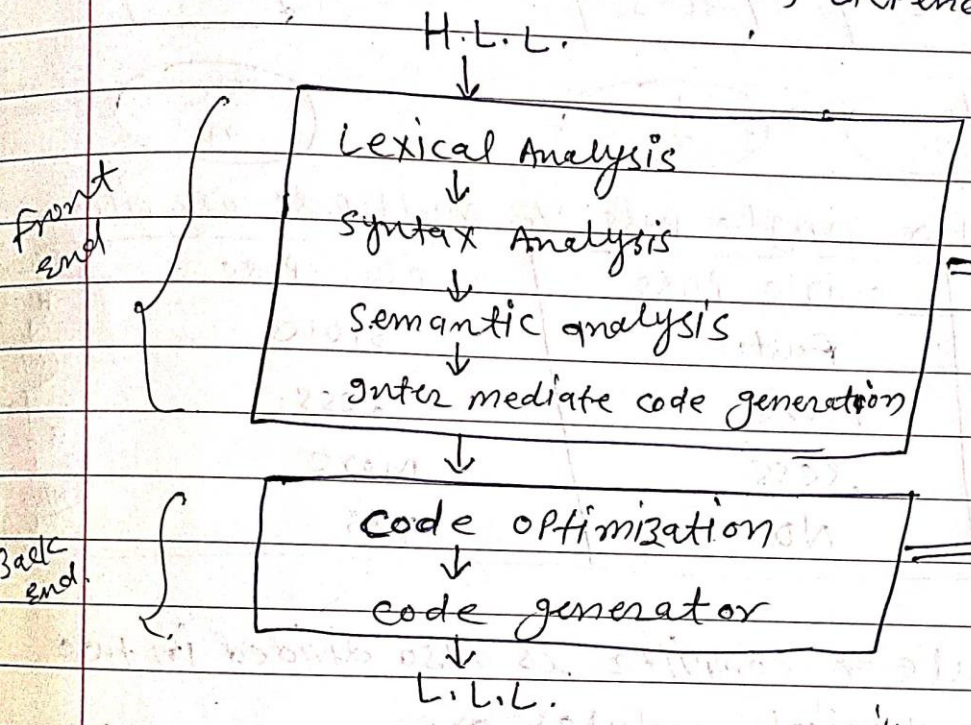
—+—

⇒ Two pass compiler / multipass compiler ⇒ A two pass compiler is a type of compiler that ~~Passes~~ processes the source code or abstract syntax tree of a program multiple time

Date:

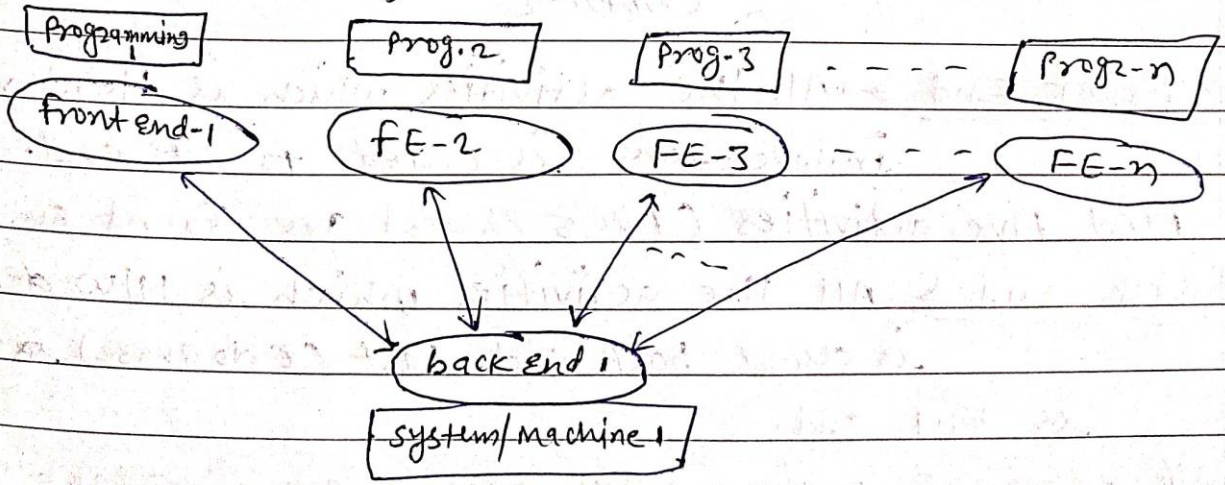
In multipass compiler we divide phases into two pass as:-

- First Pass
  - front end
  - Analytic Part
  - Platform Independent
- Second Pass
  - Back End
  - synthesis Part
  - Platform dependent



⇒ We can solve two basic problems with multipass compiler ⇒

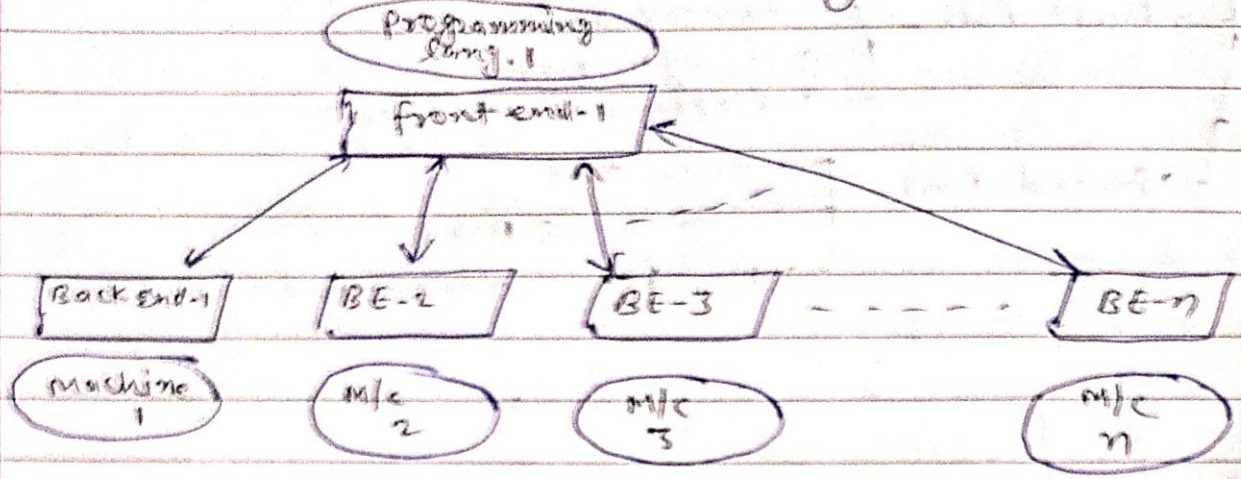
① If we want to design a compiler for different programming language for same machine. In this case, each programming language is required of front end / First pass for each of them & only one back end / second pass



② If we want to design a compiler for same programming long. for different machine/system. In this case we make different back end for different machine/system & make only one

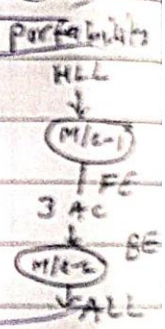
ALL → Assembly level lang
3AC = Three Address code
FE = Front end, BE => back End

front end for same programming languages as :-



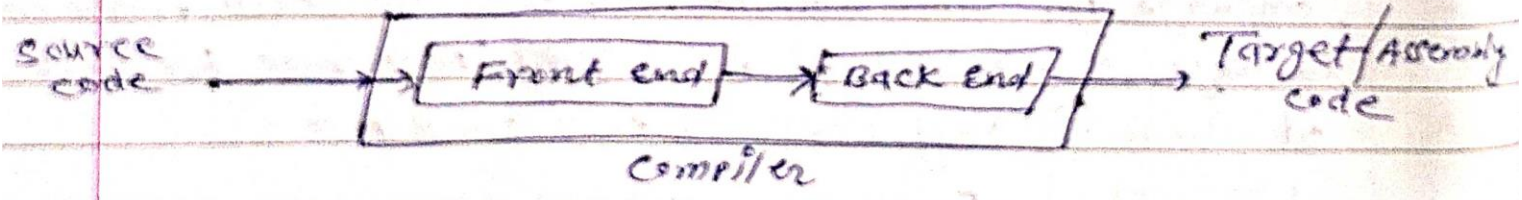
⇒ Differences b/w single pass Vs multipass compiler ⇒

Parameters	single pass	multipass
speed	Fast	slow
memory	more	less
time	less	more
portability	NO	yes



* Compiler Module ⇒ compiler is also divided in two physical modules are

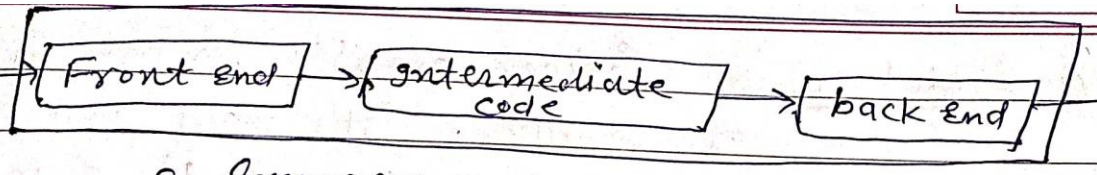
→ Front End [1 to 4 Phases]
→ Back End [5 & 6 Phases]



→ Front End ⇒ All the activities which is Hardware independent is called front end.
First five activities (1 to 5 Phases) are front end.

→ Back End ⇒ All the activities which is H/w dependent is called back end. Last (6 No. Phase) activity is back end.

All different company Intel, IBM, wipro, microtake etc has same kind of front end but they have different back End.



अगर सभी language के intermediate code same हो तो उसके back end भी same होंगे और front end अलग-अलग तरह (types) के हो सकते हैं इससे compiler construction पर effect पड़ेगा [compiler construction के efforts कम हो जाते हैं]

* **Tokens** ⇒ Tokens are terminal symbols of the source language.

Examples:- Identifiers, numbers, keywords, punctuations, symbols etc

Identifier:- { Total, count, sum, main ---}
Constants:- { 10, 25, 50, 5, ----}
Numbers:- { 0, 1, 2, 3, ----}
Keywords:- { int, while, if, for, ----}
Punctuations:- { !, ^, :, ;, ; : ?}
Operators:- { +, -, *, /, %. }
Symbols:- { (), {}, [], @, $, *, ---}

index = 2*count + 17;

Lexemes	tokens
index	identifier
=	operator
2, 17	constant
*	operator
count	identifier
+	operator
;	punctuations OR delimiters

⇒ Comments, space, tabs, blank, newline these are not a token

* **Pattern** ⇒ Pattern is a rule describing all these lexemes that can represent a particular token in a source language.

Identifier rules:- start with alphabet or _, followed by any alphanumeric character.

* **Lexemes**:- Lexemes are matched against the pattern

A specific instance of a token

count = count + temp;

Tokens:- id, operator, punctuation (;)

lexeme ;

* Advantage / Disadvantage of compiler ⇒

Advantage :- * Fast in execution
* No Bulky program. (means simple program)
* No Hardware knowledge required (register, capacitor)
* Portable (means s/w run on different machine)
C language program is portable but compiler is not portable
* Addressing :- Memory is refered by name not address
* Object executable code produced by a compiler can be distributed / executable without having (without compiler)
* object code can be used without re-compilation.

Disadvantages :-
* Debugging a program is ~~must~~ much harder
* when a error is found then whole program has to be re-compiled. * more memory required.

* Advantage / Disadvantage of ~~compil~~ interpreter ⇒

Advantages :-
* Good for coding error in program
* Debugging is easier
* If an error is detected there is not need to ~~re-com~~ re-translate (re-interprete) the whole program.
* It use less memory because few lines (statements) has to be into the memory & no object code generate.

Disadvantages :-
* Slow :- speed is the biggest disadvantage
* No object code is produced so translation has to be done every time when ~~ever~~ ever program is run.
* when program is run then interpreter must be present.

Advantage / Disadvantage of Assembler ⇒

Advantages :-
* you can access machine dependent registers & I/O devices
* you can control the exact code behavior in critical sect
* you can produce Hand-optimized code
* you can built interfaces b/w code fragmentation & using incompatible conventions (

Disadvantages:-

* Bulky program of user to write for each instruction
* Sufficient Knowledge of Hardware must be required.
* Not Portable (small)

---

* Difference b/w Compiler Vs Interpreter

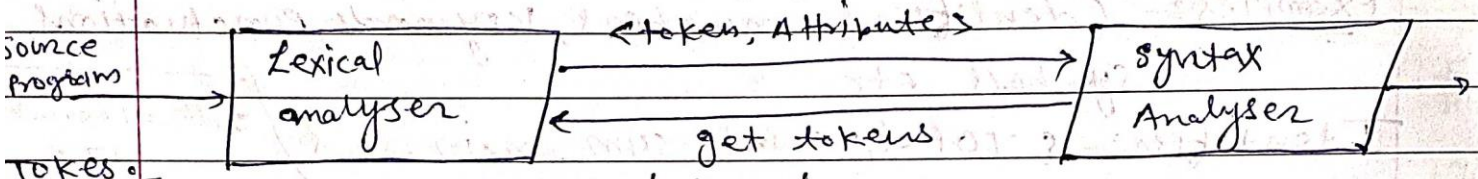
Compiler:-

(i) It translate whole program at a time
(ii) It takes large amount of time to analyse the source code but the overall execution time is comparatively (अपेक्षाकृत) faster.
(iii) Intermediate object code is generate
(iv) Debugging process is Hard.
(v) It required more memory.
(vi) It use in c/c++ languages.
(vii) It is complex program

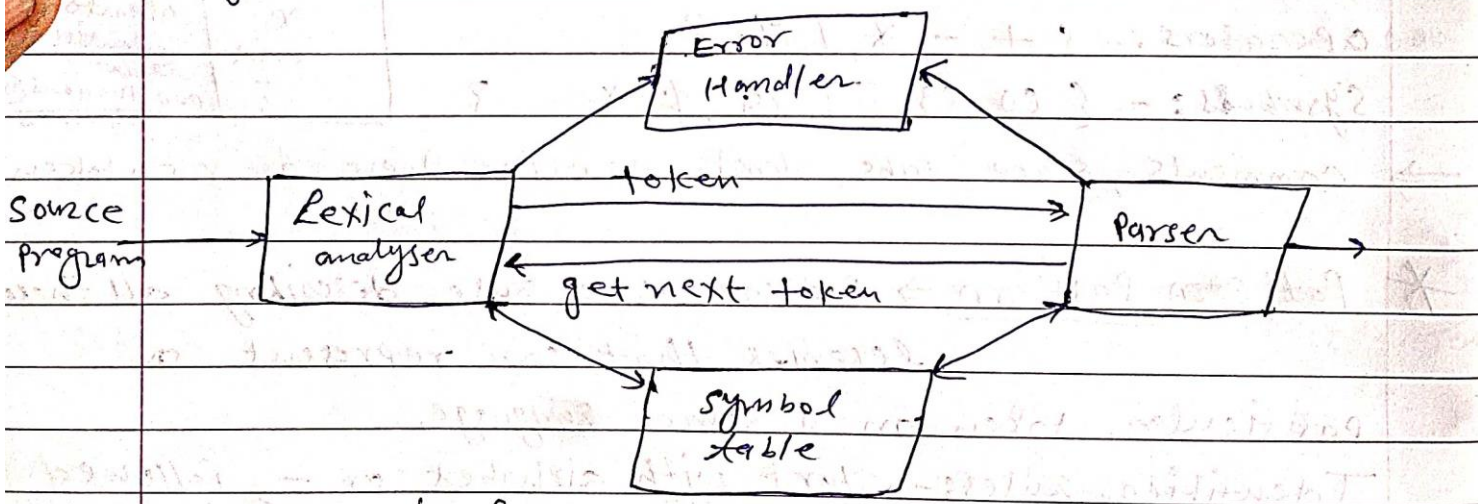
Interpreter:-

(i) It translate statement by statement at a time (one statement at a time)
(ii) It takes less amount of time to analyse the source code but the overall execution time is sl
(iii) No intermediate object code is generate
(iv) Debugging process is easy
(v) It required less memory.
(vi) It use in Python/Ruby languages.
(vii) It is easy program.

* Lexical analyser ⇒ Lexical analyser is the first phase of compiler. It's main task is to read (scan) the input symbols (characters) & produce as output a sequence of tokens that the parser uses for syntax analysis.
Some times, Lexical analysers are divided into a cascade of two phases :- The first phase is called "scanning" & the second phase is called "Lexical analysis".



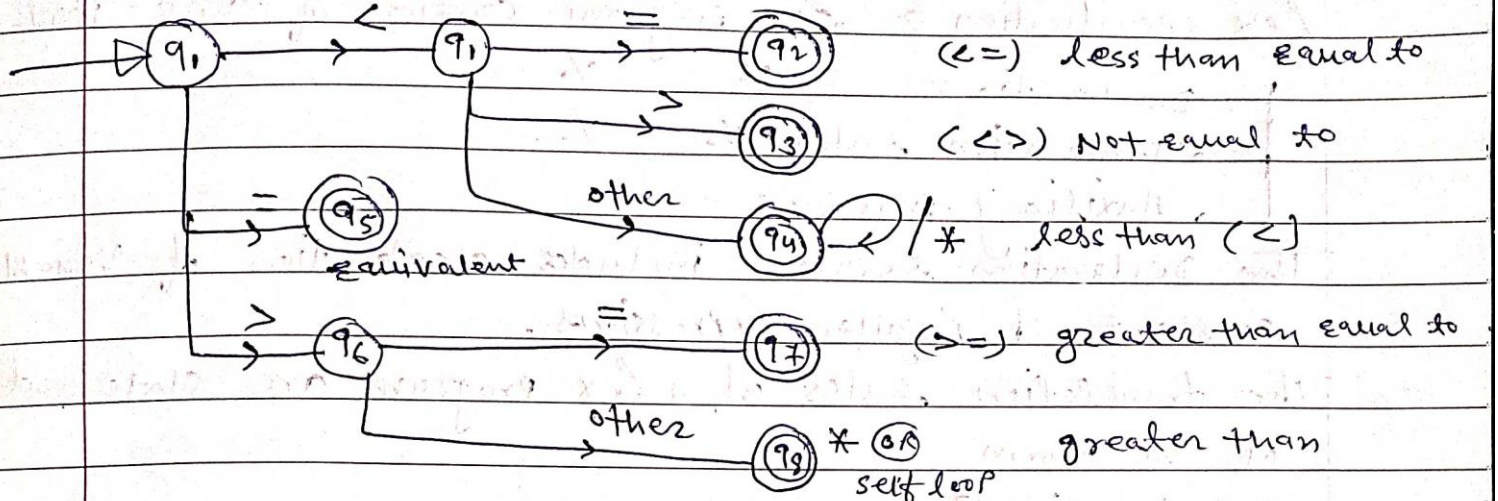
Tokes :-
operators :- +|-|*|/|mod|%
keywords :- If|while|do|then|for
Letters :- a|b|c|---z|A|B|C|---|Z
Digits :- 0|1|2|3|---|9



→ Functions of Lexical analyser ⇒
(i) Remove white space ⇒ White space of blank's tabs & new lines etc. appear b/w tokens they are detected & dropped by Lexical analyser.
(ii) Remove comments :- Like // or /* --- */ इनके साथ (comment) में जो भी लिखा होगा वो भी remove होगा
(iii) Handling of constants :- ~~int white, for, if~~, 2, 0, 5, 100 etc
(iv) Handling of operators :- +|-|*|/|%|mod
(v) Recognization of identifiers & keywords :-
count, sum, main ← If, else, while, out for

→ Recognization of tokens by lexical analyser ⇒

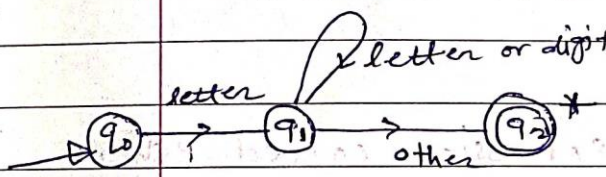
(i) Recognization of relation operators ⇒



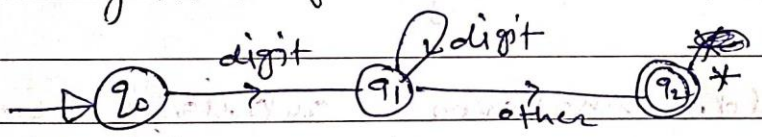
Operators relations :-

$<$, $>$, $<=$, $>=$, $<>$, $=$

(ii) Recognization of identifiers & keywords :-

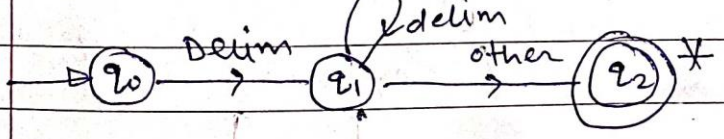


(iii) Recognition of number constants ⇒

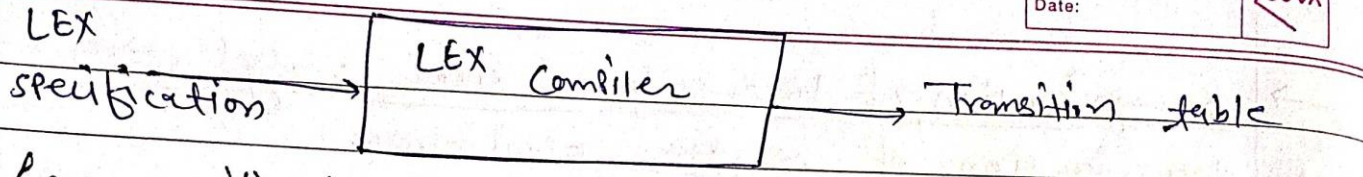


(iv) Recognization of white space ⇒

Delimiter or Delim = blank / tab / newline



* Lex (LEX) Concept ⇒ Lex is a compiler writting tools which facilitated writting of lexical analyser & hence a compiler. There are several tools that have been built for constructing a lexical analyser. LEX has been widely used to specify lexical analysers for variety of languages. This tool is reffered to LEX compiler & input specification is called Lex Language.



Lex specification ⇒ Lex program consists of three parts :-

- Declaration % %
- Translation rules % %
- Auxiliary procedures

The declaration section includes declaration of variables, constant & Regular expressions.

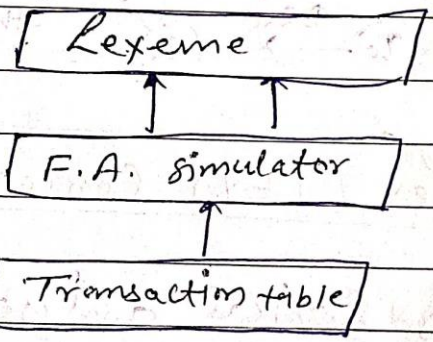
The translation rules of a Lex program are statements of the form

P_1 { action-1 }
P_2 { action-2 }
P_3 { action-3 }
:
P_n { action-n }

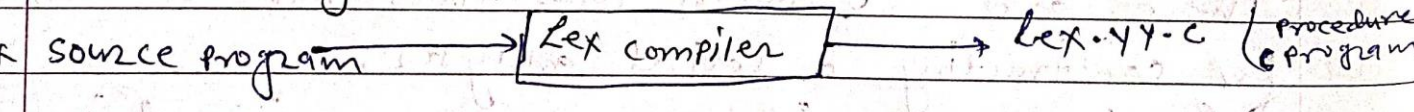
where each P is a regular expression called pattern & each action is called program that is to be execute &

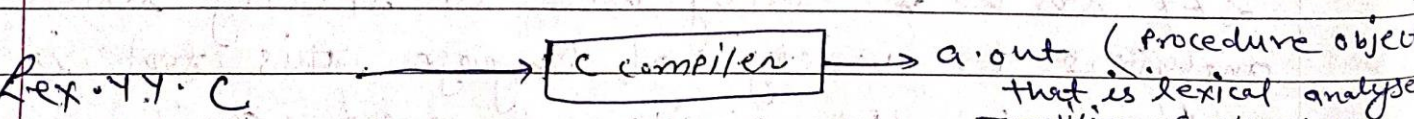
Third section holds whatever auxiliary procedures are needed by the actions. These procedures can be ~~compiled~~ compiled separately & located with lexical analyser

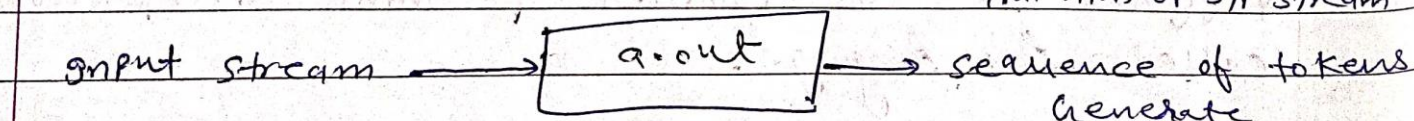
Model of Lex compiler ⇒



Lex is usually



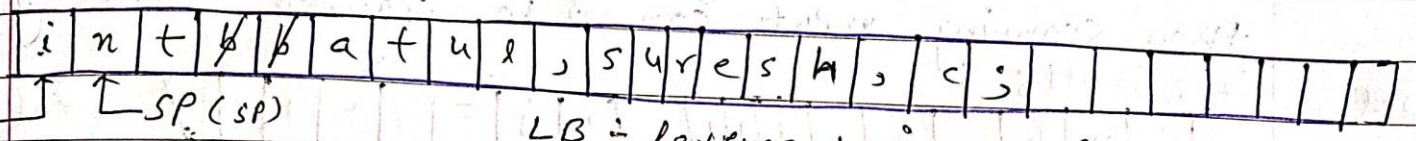




* Input Buffer / I/P Buffering ⇒ Input buffer is done for increasing efficiency of the compiler.

A two buffer input scheme is useful when look ahead on the input is ~~increasing~~ necessary to identify tokens

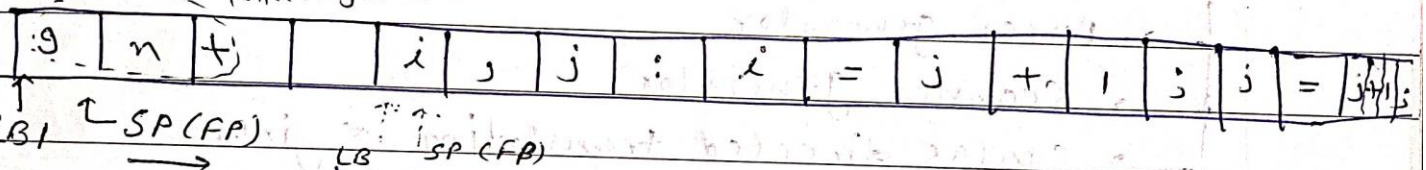
Example:- int, atul, suresh, c;



LB ↑ SP (SP) ⟶

LB ≡ lexeme begin, SP = Search Pointer or forward pointer

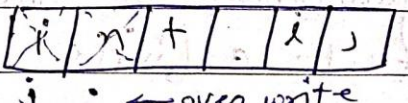
The Lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin pointer (LB) & forward to keep track of the pointer of the input scanned.



Initial both pointers point to the first character of ~~the~~ the input symbol (I/P string) & forward pointer moves ahead to search for the end of lexeme. As soon as the blank symbol (blank space) is encountered, it indicates end of lexeme.

⇒ One (Single) Buffer scheme ⇒ In this scheme only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very large (long) then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled that makes overwriting the first of lexeme.

int i,j;

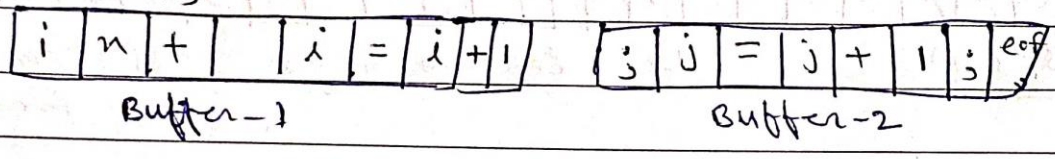
int i, j; ⟹ 

j ; ← over write

⇒ Two buffer (double ~~buffer~~ buffer) scheme ⇒ To over come the problem of one buffer scheme, in this method two

buffers are used to store the input string.
The first buffer & second buffer are scanned alternately when end of the current buffer is reached the other buffer is filled.
The only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely

int i=i+1; j=j+1;



* Construction tools for compiler ⇒ There are different types of tools used in compiler constructions are following as

- Parser generator
- Scanner generator
- Syntax directed translation engines
- Automatic code generators
- Data flow engines.

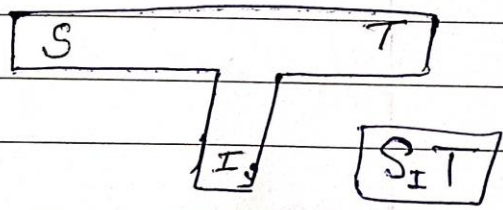
* Boot strapping (OR) Bootstrap compiler ⇒

Compiler is a complex enough program that we would like to write in a friendlier language than assembly language. Boot strapping is used to create compilers & to move them from one machine to another by modifying the backend. It represented by as "T" diagram.

Bootstrapping is widely used in the compilation development. It is used to produce a self hosting compiler. self Hosting compiler means it is a type of compiler that can compile it's own source code

A compiler can be characterized by three languages :-

- Source language (S)
- Target language (T)
- Implementation lang. (I)



Bootstrapping is the process by which simple language is used to translate more complicated programs, which in turn may handle an even more complicated program soon...

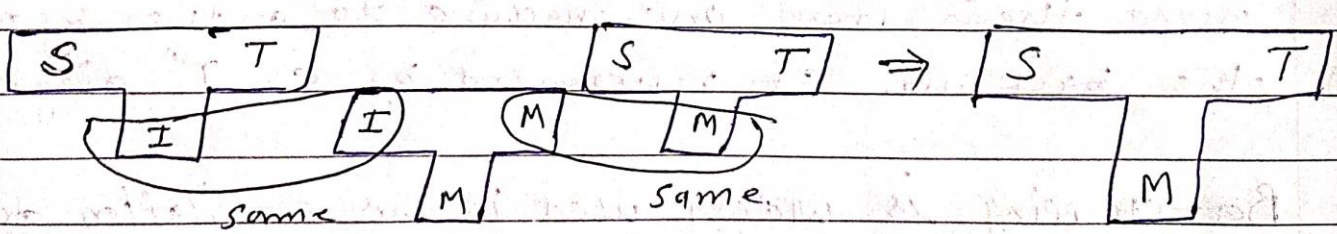
* Cross compiler ⇒ A compiler may run on one machine & produce target code for another machine such a compiler called cross compiler

Example ⇒ suppose, we write a cross compiler for a new language $\mathcal{L}$ in implementation language S to generate code for a machine N that we create $\mathcal{L}_S N$ If existing compiler for S runs on machine M & generates code for M it characterises by $S_M M$ If $\mathcal{L}_S N$ is run through $S_M M$ then we get a compiler $L_M N$

$$\mathcal{L}_S N + S_M M = \mathcal{L}_M N$$

(i)

(ii) $S_I T + T_M M \Rightarrow S_M T$



* Symbol table ⇒ Symbol table is an important data structure Created & maintained by compiler in order to keep track of semantics of variable. It stores information.

→ It is built in lexical & syntax analysis phases.

→ The information is collected by analysis phase & it used by synthesis phase.

→ It is used by ~~various phases of~~ compiler to achieve compile time efficiency.

→ It used by various phases of compiler :-

* Lexical analysis :- create new table, token entries in this table
* Syntax analysis :- Add informations regarding attribute type, scope, dimension, line of reference, use. etc. in table.
* semantic analysis :- uses variables information in the table to check for semantics (type checking)
* Intermediate code generation :- Refers symbol table for knowing how much & what type of runtime is allocated & table helps in adding temporary variable information.
* Code optimization :- uses information present in table for machine dependent optimization.
* Target code generation :- Generates code by using address information of identifiers present in the table.

⇒ Symbol table entries :- Each entry in symbol table is associated with attributes that support compiler in different phases.

Items stored in symbol table :-

Variable name & constants, procedure & function names, literal constants & strings, compiler generated temporaries, labels in source languages.

Information used by compiler from symbol table :-

Data type & name, Declaring procedures, offset in storage, If structure or record then pointer to structure table, Parameters, number & type etc.

Operation of Symbol table :- According to data structure like; list, linked, arrays, Hash table, Binary search tree

Error Handling in compiler Design ⇒ The tasks of the error handling process are detect each error, report it to the user & then make some recover strategy & implement them to handle Error During this whole process. An error is the blank entries in the symbol table.

Types of error :- There are mainly two types (source) of error :- Run time error & compile time error

→ Run time error :- Run time error is an error which takes place during the execution of a program. types :- Lack of sufficient memory, run an application, logical errors occurs when executed code does not produce the expected result.

→ Compile time error :- compile time errors rises at compile time before execution of the program.

Compile time errors :-

→ Lexical :- misspellings of identifiers

→ Syntactical :- missing semi colon or unbalance parenthesis

→ Semantical :- incompatible value, assignment or type mismatch b/w operand & operators

→ Logical :- code not reachable, ∞ loop,

Error recovery methods :-

Panic mode recovery :- (1++2)+3

Phase level recovery :-

Error Productions

Global correction