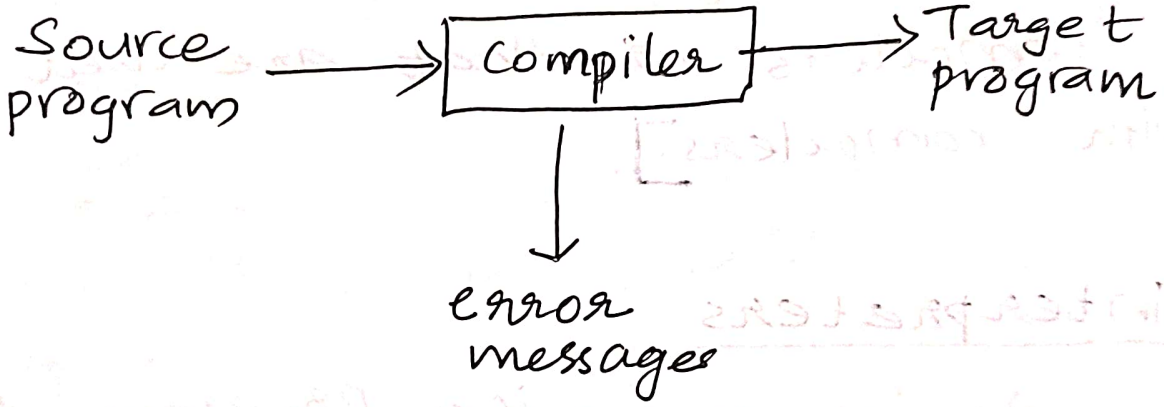
Compiler Design

Module 1

Compiler is a program that translate one language (source programming language) to an equivalent target program.



→ A compiler is basically a software utility; also known as language translators.

→ Source program : It is a high level language. Eg: C, C++, Java

→ object program / target program:
It is machine code, which is written in machine instructions of the compiler on which it is to be executed.

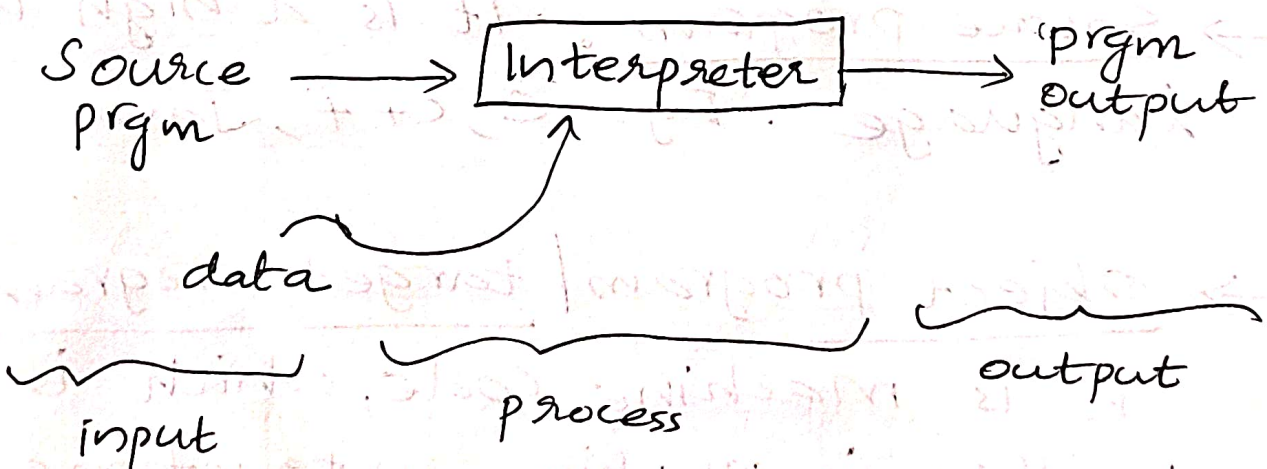
During this translation process, the main function of the compiler is to report any errors in the source program found during translation process.

Compilers related programs or language processing system

[The other programs that are related to compilers or that are used together with compilers.].

1) Interpreters

An interpreter is a program that appears to execute a source program as if it were machine language.



Languages such as BASIC, LISP can be translated using interpreters.
Java also uses interpreter.

phases of interpreter

1) Lexical analysis
2) Syntax analysis
3) Semantic analysis
4) Direct execution

Advantages:

- modification of user program can be easily made.
- may change dynamically (objects)
- Debugging easy
- makes machine independent.

Disadvantages:

- program execution is slow.
- memory consumption is more.

2) Assembler

It is a computer program that translates assembly language code into machine code.

Uses:

Assemblers are used for low level programming, such as writing operating systems, device drivers and embedded systems.

- → single-pass assembler
- → multi-pass assembler

It can convert mnemonics, or symbols and instructions, such as ADD, MUX, DIV, SUB and MOV, into binary code.

3) Preprocessor

A preprocessor may perform the following functions:

→ Produces input to compilers.

→ Macro processing

textual shortcuts will expand

```
#define SQUARE(x) (x) * (x)
int result = SQUARE(5);
```

→ File inclusion
(including header-files into the prgm text)
#Include<stdio.h>

→ Rational preprocessor — these pre-processors augment older languages with more modern flow-of-control and data structuring facilities.

→ Language extensions —

Advantages:
→ code reusability
→ improved readability
→ Platform adaptation.

4) Loaders and Linkers

Loader — It is a program that places programs into memory and prepares them for execution.

Linker — combines object files into a single executable ~~program~~ file. It takes object files created by a compiler or assembler and joins them together.

Translators

A translator is a program that takes as input a program written in one lang. and produces ~~as~~ ~~output~~ a program as output in another language.

It performs the error-detection.

Editors

Source programs are accepted by the compiler which is written using any editor to produce a standard file like ASCII file. IDE includes compilers along with editors and other programs (Integrated Development Env. - IDE)

Example for compilers

C compilers
C++ compilers
Python compilers
Java compilers
Ada compilers
ALGOL "
Pascal
PL/1 compilers

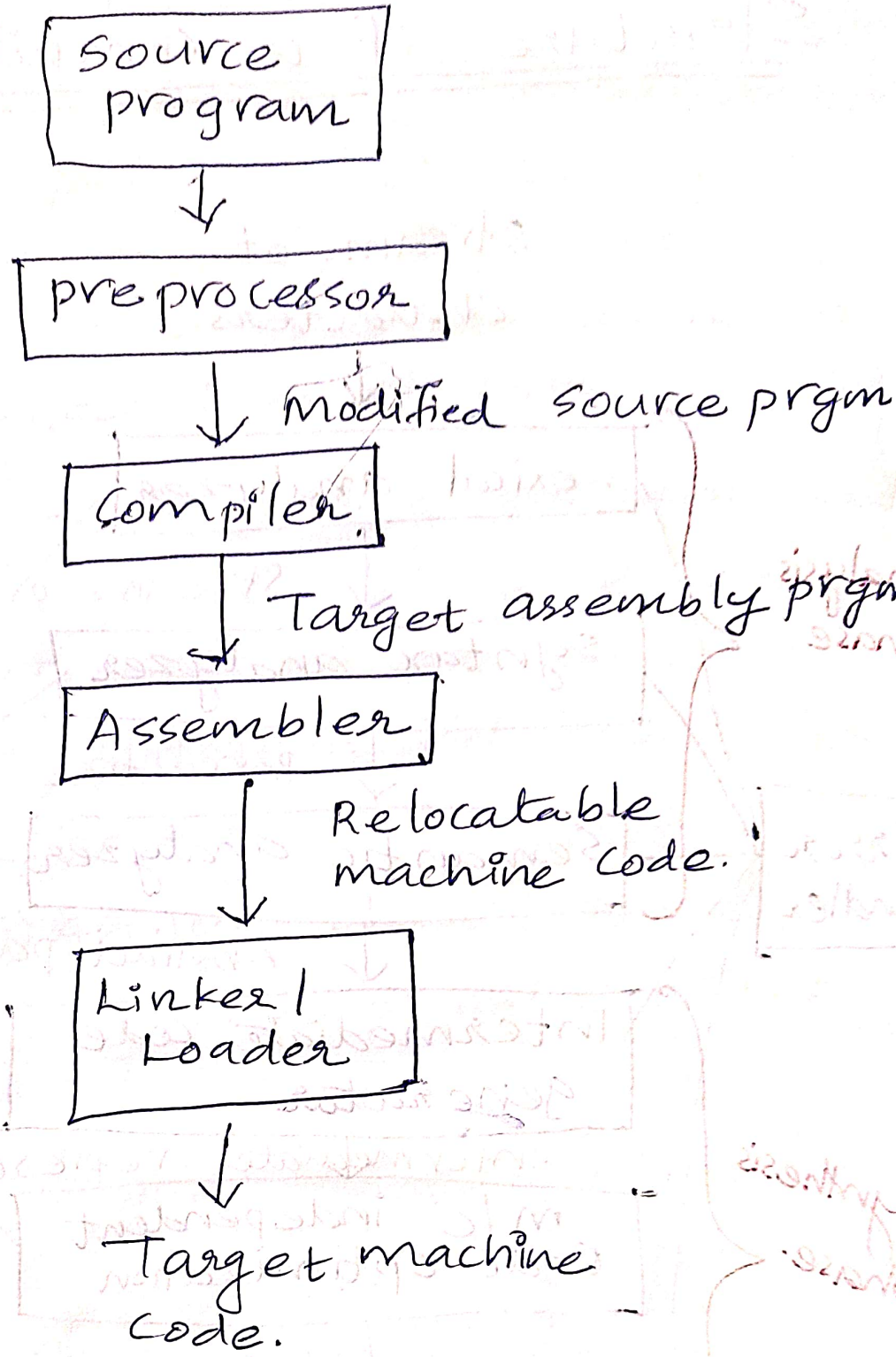


Fig: A language processing system.

Major data structures in a compiler.

A compiler should be capable of compiling a program in time proportional to the size of the program, $O(n)$ time, where n is the measure of program size.

1) Tokens

When a scanner collects characters into a token, its a category, like a keyword or identifier, representing units of meaning.

2) The syntax tree

3) The symbol table

The symbol table is a data structure keeps information associated with identifiers, functions, variables, constants and data types.

This will interacts with almost every phase of the compiler. A standard data structure for this purpose is the hash table, and tree structures are also using for this. Sometimes several tables are used and maintained in a list or stack.

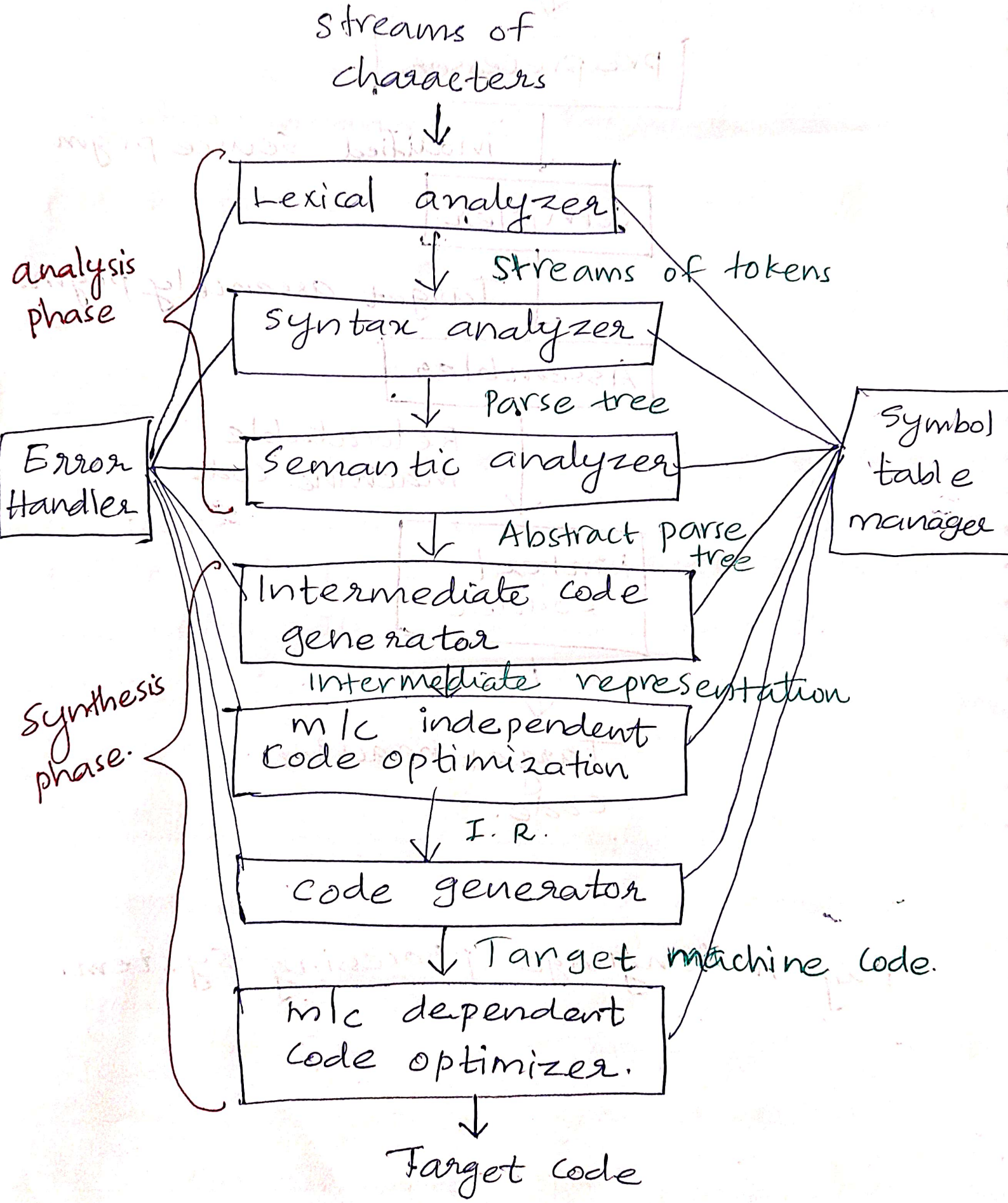
4) Literal table

Quick insertion and lookup are essential to this, which stores constants and strings used in a program. This will not allow deletions.

~~→ This is important blog~~

→ It allows reuse of constants and strings, so as to reduce the use of memory.

Structure of a compiler.



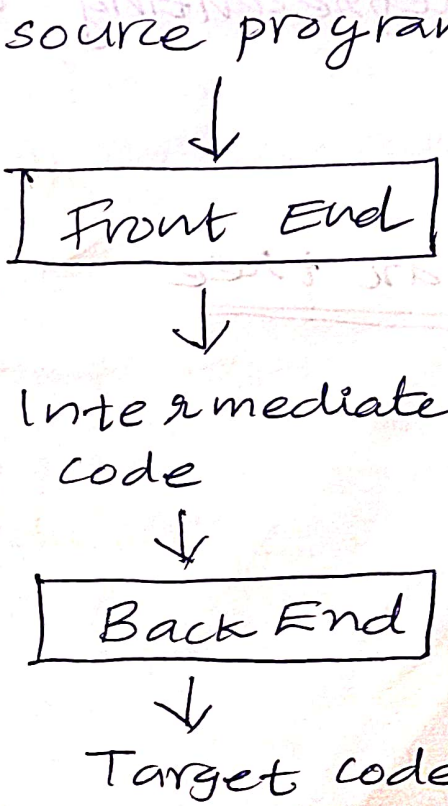
The compiler accepts the pre-processed file as input and translates it into an equivalent assembly language. compiler is a huge program that can consists of 10,000 - 1,000,000 lines of code.

As the compiler is a huge program, its difficult to understand the entire compilation process. so the process is divided into different PHASES.

Mainly two phases

i) Analysis phase / FRONT END

ii) Synthesis phase / BACK END



① Lexical Analysis

→ 1st phase of compiler. called Lexical - analysis or scanning.

The lexical analyzer reads the stream of characters making up the source prgm and groups the characters into meaningful sequences called lexemes.

For each lexeme, the lexical analyzer produces as o/p a token of the form <token_name, attribute value>

it passes on to next phase.

token_name — It is an abstract symbol that is used during syntax analysis.

attribute value — It points to an entry in the symbol table for this token.

Input for LA → the program.

output from LA → Tokens.

Tokens are like the words of a natural language; each token is a sequence of characters that represents a unit of information in the source program.

Examples

keywords : Eg: if, while

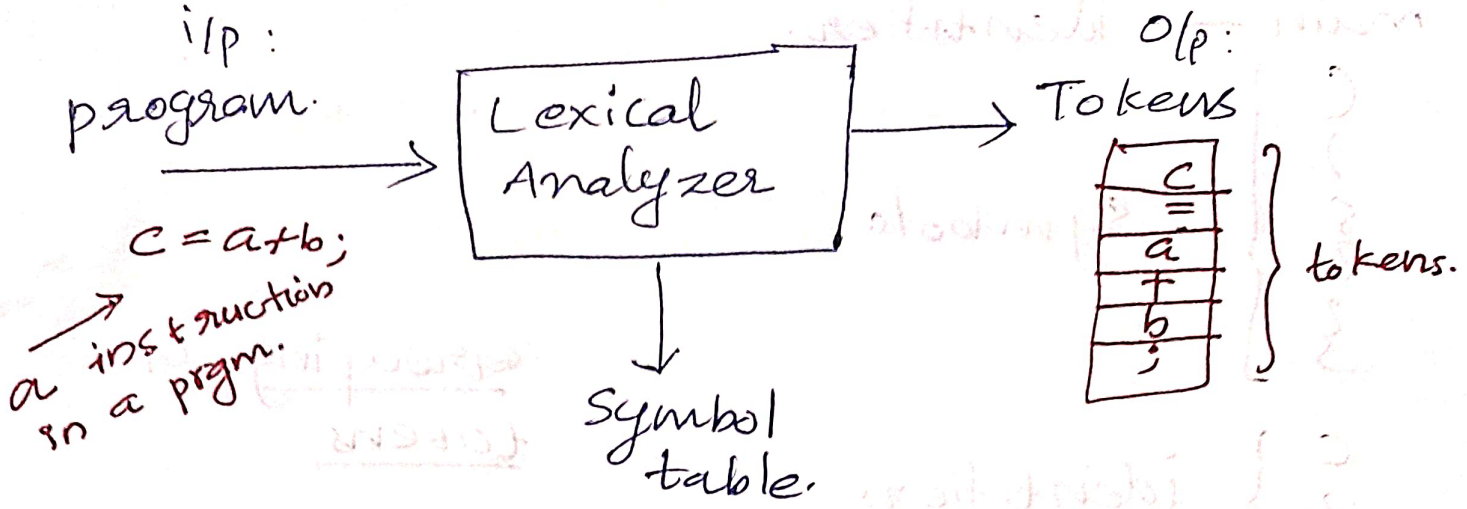
identifiers : Eg: (user defined strings)

special symbols : Eg: +, *, >=, <>

operators
seperators.
symbols
(), {}

since the task performed by the scanner is a special type of pattern matching, we need to study methods of pattern specification and recognition as they apply to the scanning process.

For that we're using regular expressions and finite automata.



Eg: for tokens.

1) int a, b;

int | a | , | b | ;
1 2 3 4 5

5 tokens.

Lexemes → int
a
b
,
;

2) int | main | ( ) |
{ | int | a | , | b | ; |
c | = | a | + | b | ; | /* addition */ |
}

this'll be eliminated.

int — keyword

main — identifier.

( ) { } — symbols.

c a b — identifiers.

+ = — operators

, ; — seperator symbols.

lexemes — tokens

Grouping of tokens.

3) ~~Do~~ If min(i,j)

If/ min/(/i/,/j/)/ — 7 tokens.

Role of LA

- Eliminate comments, whitespaces, unessential symbols.
- Program/instruction will be divided into tokens.
- Grouping of tokens will be done.
- Each and every token will be Validated.
- After that, a list of tokens will be generated.
- Use of symbol table, and error handlers.

Valid tokens

keywords
Identifiers
operators
Seperators
and special symbols.

Invalid tokens

spaces
Macros
Escape sequences.

- It counts line numbers of source prgm.
- It provide error msgs with corresponding line and column numbers.

Compiler construction tools

1) **Scanner tool**

i/p : program.
o/p : tokens.

Eg: LEX tool

2) **Parser tool**

i/p : tokens
o/p : parse tree.

Eg: YACC tool (Yet Another Compiler Compiler)

3) **Syntax Directed Transition Engine**

i/p : Parse tree
o/p : Intermediate code.

4) **Data flow analysis engine**

i/p : Intermediate code
o/p : optimized code

5) **Code generator**

i/p : optimized code
o/p : machine code

6) **Compiler construction tool kit.**

— combination of the above tools.

Specification of tokens.

Token ⟶ Regular expression
↳ language
↳ set of strings
↳ Set of alphabets
↓
Symbols.

Symbols — a, b, 0, 1 etc.

Alphabets —
i) 0, 1 — binary
ii) a .... z — small letters.
iii) A — Z — uppercase letters
iv) ASCII characters.

Strings — aabb
0101 etc.
{ prefix
suffix
substring

language — Set of strings generated from alphabets.

Operations — concatenation
intersection
→ product of both set.

Kleen closure
+ve closure

Regular Expression

The concept here is 'pattern matching'

Eg:-
For identifiers,

$$[\_ / a\text{-}z / A\text{-}Z]\ [a\text{-}z / A\text{-}Z / 0\text{-}9 / \_]^*$$

↓ start with underscore

↓ combination of letters, no.s and underscore.

So the tokens must satisfy the above r.e, then only it can be grouped as an identifier. Otherwise it will generate error.

For digit ⇒ $[0\text{-}9]^+$
for letter ⇒ $(a\text{-}z / A\text{-}Z)^+$
for whitespace ⇒ (newline / blank / tab / comment)$^+$

Recognition of tokens.

→ Transition table
→ Transition diagram.

Transition table

Same symbol table can also call transition table.

stamnt ⟶ if expr then stmt
if expr then stmt else stm
ε

expr ⟶ term relop term

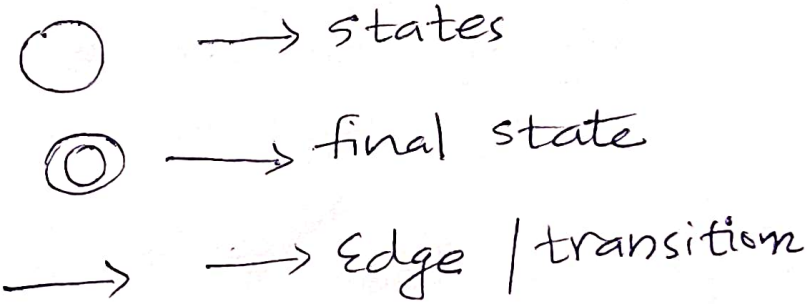
term ⟶ id
number

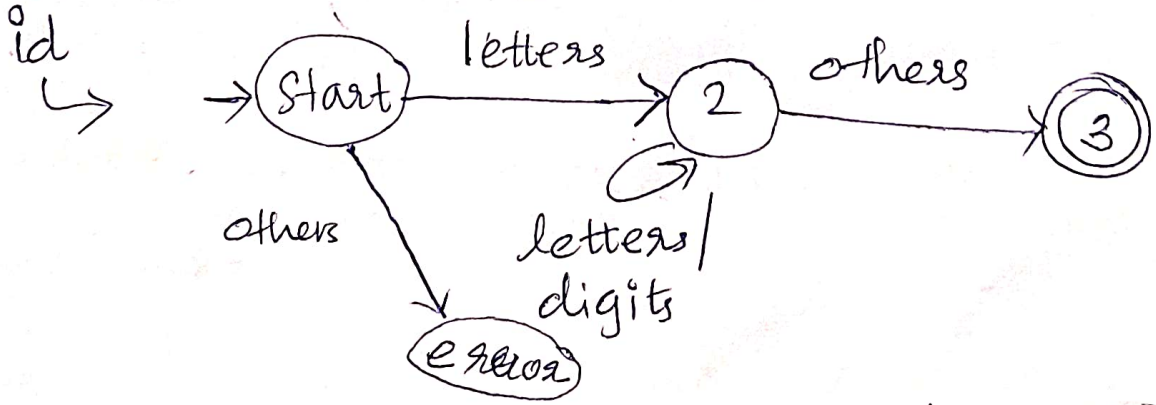
first, eliminate all the white spaces.
Then will coinsider the lexemes.

LEXEME	TOKEN
if	keyword
then	keyword
else	keyword.
<	operators.
>	
<=	
>=	
=	
<>	

Likewise we can represent tokens using a transition table.

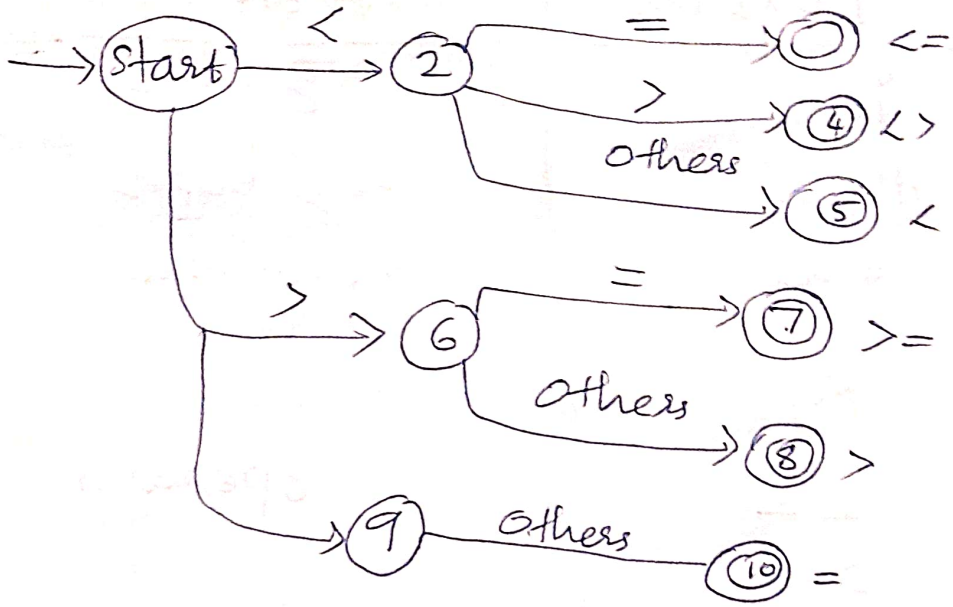
Transition diagram





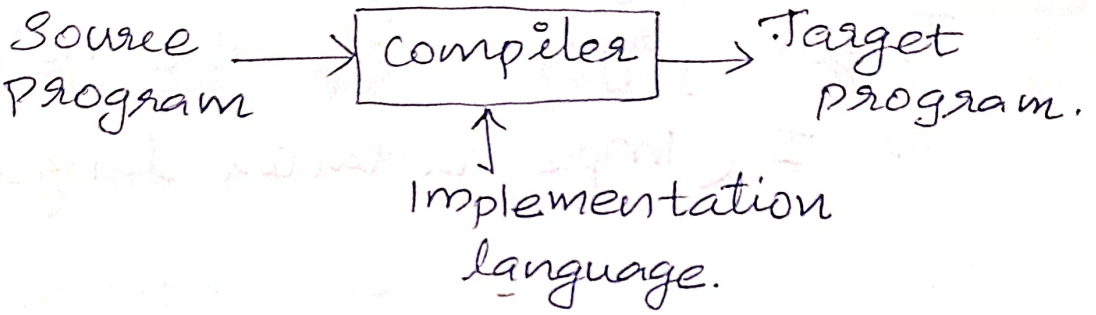
$id \rightarrow (letter)\ (letter \mid digit^*)^+$

Rel.op



Bootstrapping

- It will be used in designing of compiler.
- compiler development.



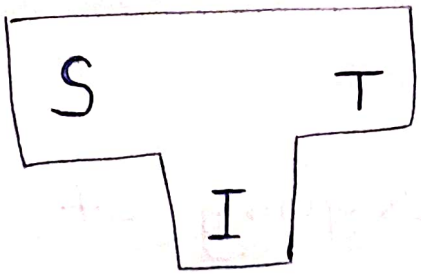
[compiler is designed using any programming language called implementation lang.]

- self-Hosting compiler. (a compiler that can compile its own source code).

The technique used to create a self-hosting compiler is called bootstrapping.

Compiler design is categorized with 3 languages-

- source language (S)
- Target language (T)
- Implementation language (I)

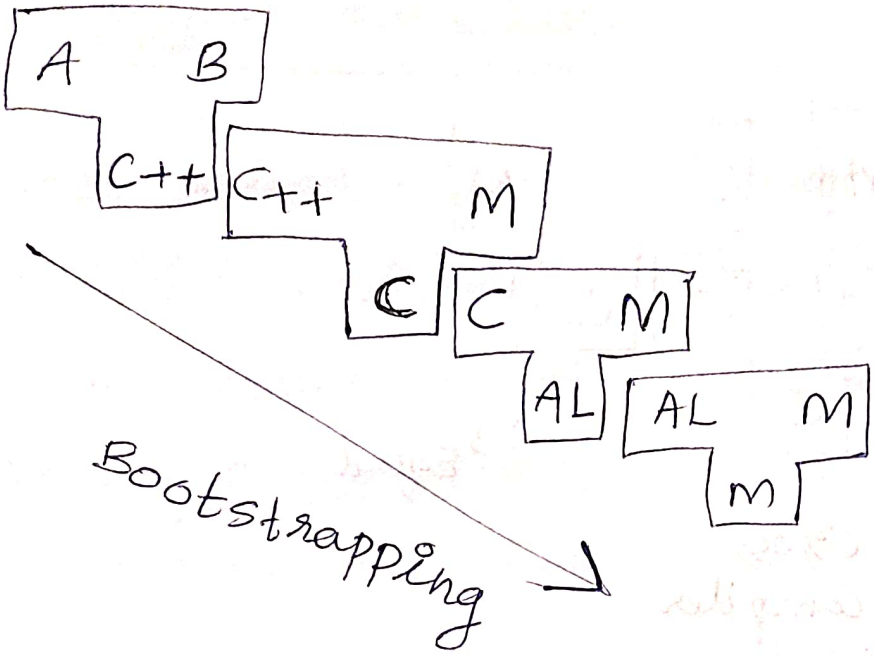


T- diagram.

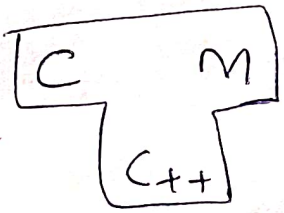
S_I^T source prgm converted to Target prgm with the help of I (Implementation language)

Eg:-

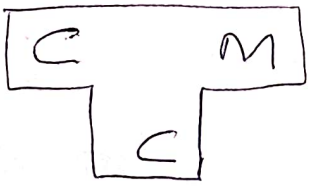
C program — source prgm
C compiler — Implem. language
Machine code — Target prgm.



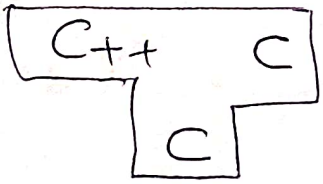
- Bootstrapping is used to create CROSS-COMPILER. (Cross-compiler is one that runs on one machine and produces a code for another machine).
  ⇒ It is capable of creating code for a platform other than the one on which the compiler is running.



Source

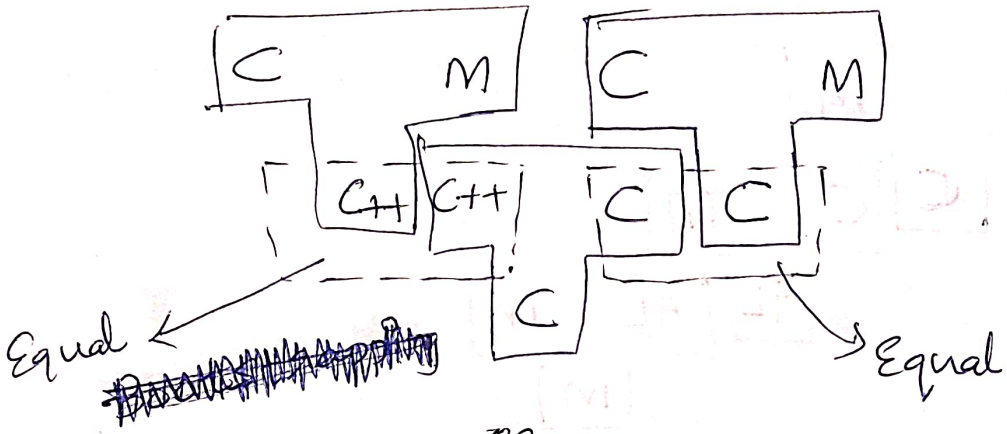


Target



Cross Compiler

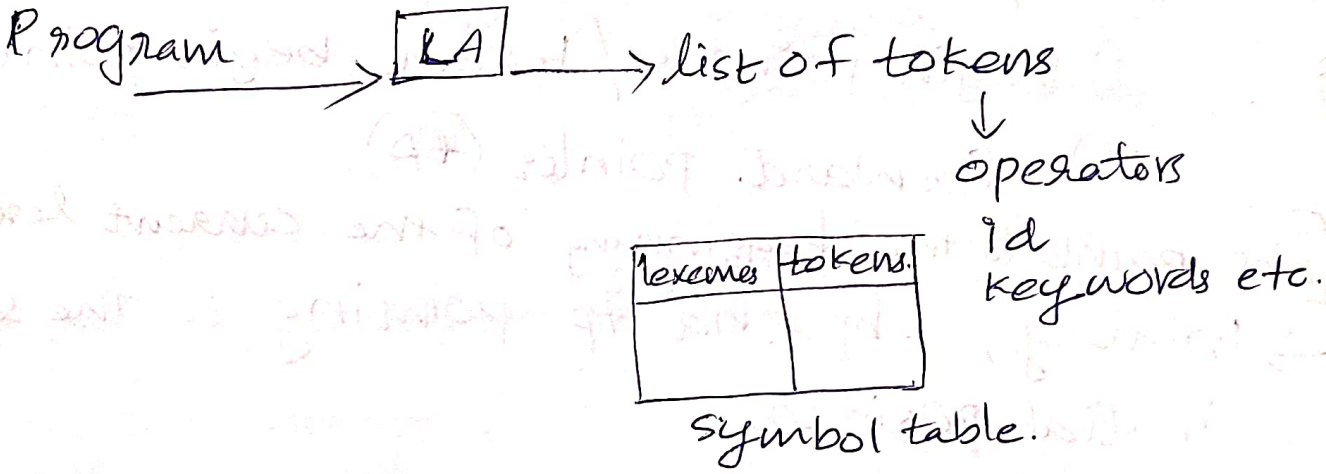
Combining



Cross Compiler

Input Buffering

Input Buffering is a technique that improves performance by reading and storing parts of the source code in memory before processing.



The program is converted into tokens by lexical analyser with the help of input buffering.

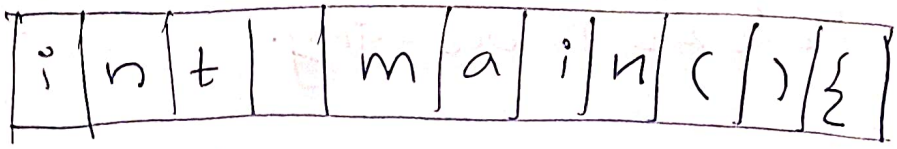
Eg:-

```
int main()
{
    int a, b;
    a = a + b;
}
```

consider this piece of code.
The program will first store in the

secondary memory. Then copied into main memory.
It reads every character from main memory.

a memory



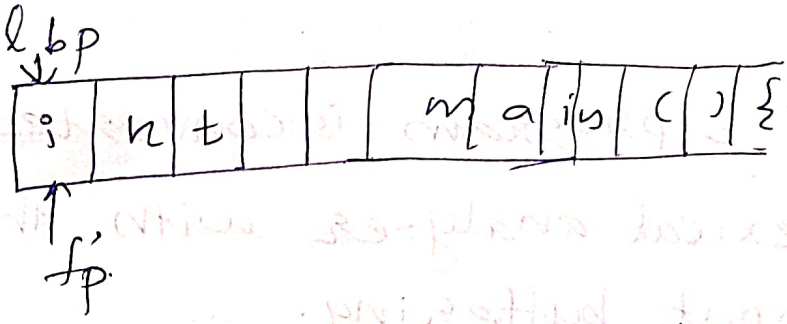
Here, we've two pointers.

i) begin pointer / lexeme begin pointer (lbp)

ii) forward pointer (fp).

[lbp points to the begginning of the current lexeme.]

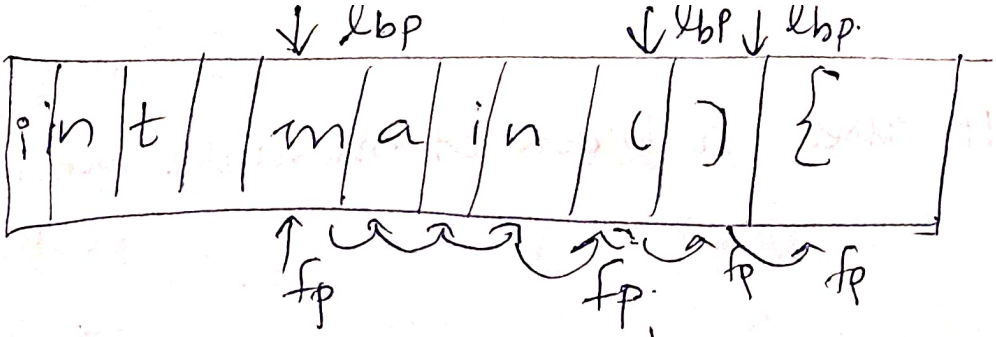
→ Initially, lbp and fp pointing to the same initial position.



→ fp move towards right and read each character one by one.

→ whenever fp finds a space, lbp move forward and points to next character.

lexeme	token
int	keyword



here it stops. and main will stored to symbol table.

lexeme	token
int	keyword
main	keyword
(	special symbol
)	special symbol
{	special symbol

It uses a block of memory buffer.

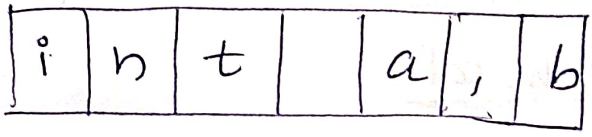
- One buffer
- Two buffer

One buffer

Suppose the size of the input buffer be 6,

from the piece of code,

int a,b;

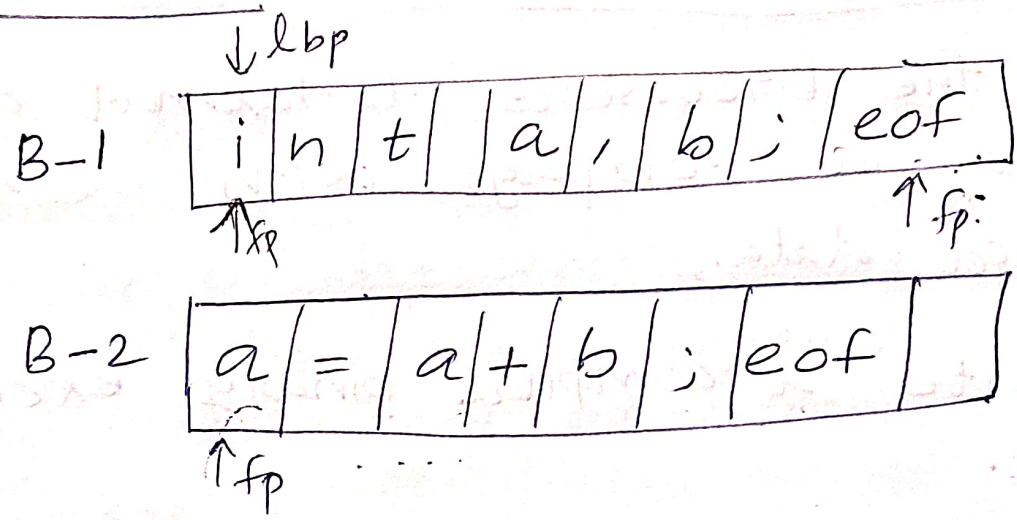


; is remaining.

So i will replace or overwrite with that.

→ So there might exist some sort of inconsistancy, we can use Two buffers.

Two Buffers



eof – represents end of file, indicates the end of the buffer.

The other name of eof is 'Sentinel character'

```
switch (*forward++)
{       case 'eof':
          if (forward is at end of 1st buffer
          {
              forward = beginning of 2nd buffer;
              refill 2nd buffer;
          }
    else if (forward is at end of 2nd buffer)
          {
              forward = begining of 1st buffer;
              refill 1st buffer;
          }
          else
              break;
```

Advantages of LA

i) Helps the browsers to format and display a webpage with the help of parsed data.

ii) Create a compiled binary executable code.

iii) Helps to create a more efficient and specialised processor for the task.

Disadvantages of LA

→ much time required to read the source code and partition it into tokens.

→ It requires much effort to doing and develop the lexer and its token description.

→ It requires additional runtime overhead to generate the lexer table and construct the tokens.

Grouping phases of Compiler.

Phases — Units / stages that program runs during compilation.

.. — 6 phases.

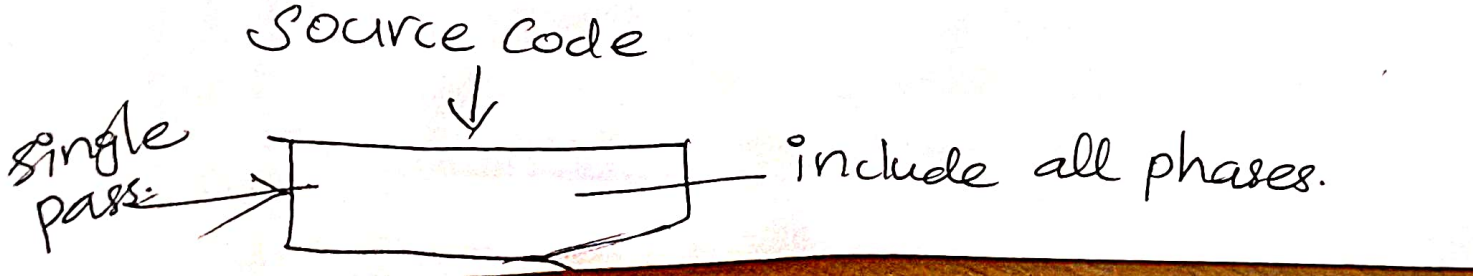
{LA
SA
sem. A.
Interm. code - gen.
COpt
TCG.}

Passes — How many times a ~~compiler~~ program / source code runs or compiles.

- single pass compiler / one pass compiler
- two pass compiler / multi pass "

(i) Single pass Compiler

All 6 phases will be grouped into single category.



(ii) Multi or two pass compiler

All phases will be grouped into two categories or 2 passes.

1st pass ⟹ Front end / Analysis phase
- depends upon source code.
- Independent upon Target code.

2nd pass ⟹ Back end / Synthesis phase
- depends upon target code
- Independent upon source code.