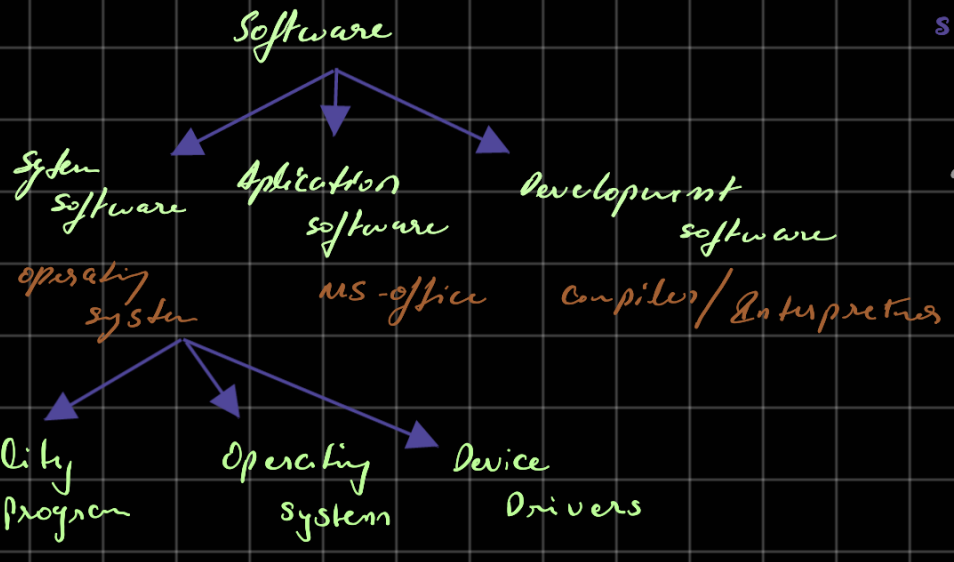
System Program →

System Program
- Compiler

Application Program
- Editor "Pycharm".

System Call →
Method of interaction with operating system via program. Program requests several services, and "O.S" responds by invoking a series of system calls to satisfy the request.



Mnemonic →
Abbreviation for an program.
inc → increase by one.

Kernal →
Is a computer program at the core of computer's "O.S" and has complete control over everything in system.

Allows input/output divices to communicate with programs and other system components.

- System Software →

Operating System →
OS consists of the master system of programs that manages the basic operations of the computer.

Utility Software →
It is a system software that helps to maintain proper and smooth functioning of a computer system.

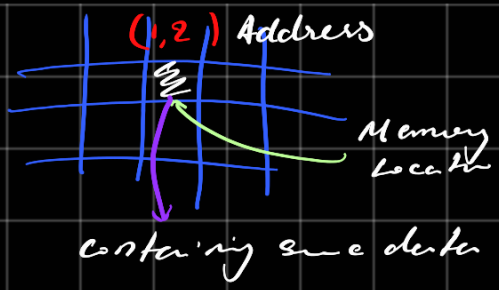
Device Driver →
A software that helps a peripheral device establish communication with computer.

System Software →
Coordinates the activities and function of hardware and programs.

General Machine Structured →

Memory Address Register (MAR) →

Contains the address of memory location.
to be read from.



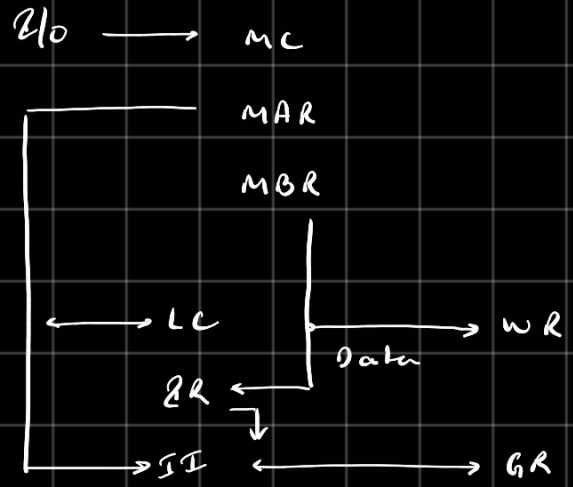
Memory Buffer Register (MBR) →

Contains the copy of memory location whose address is stored in "MAR".

Memory Controller →

Transfer the data of "MBR" to core memory location whose address is stored in "MAR".

General Machine Structure →



IR → Instruction Interpreter
IR → copy the content of LC stored in IR.
LC → Points to the current Instruction.
WR → scratch pad for IR
GR → used by the programmers.

- working Register
- General Register

Assembler →

- Is a low level language.
- Considered as a second-generation language.
- for writing "OS" and desktop applications.

High-Level Language →

- easy to understand, user friendly.
- requires compiler to convert the code, so the system can understand.

  eg → C, C++, Java

Company Top Source Code Management Software →

- **Github** Enables large development teams to collaborate, review, and manage software or application code
  - Offers free trial
  - Team: $4 per user/month
  - Enterprise: $21 per user/month

- **Bitbucket** One-stop solution for versioning, project management, and collaboration across teams of any size
  - Free for small teams with up to 5 members
  - Standard and Premium come at $3 and $6 per user/month, respectively

- **Gitlab** Used for end-to-end project lifecycle with git-based tools: version control, project management, CI/CD
  - Free for individuals
  - Premium and Ultimate editions come at a cost of $19 and $99 per user/month, respectively
- **Team Foundation Server** Enterprise-grade source control management tool that supports integration with most of the existing IDEs
- • Offers free trial
  • Basic Plan: $2 per user/month
  • Azure Pipelines: $15 per parallel job
- **Apache Subversion** Open-source version control system supporting file locking and merge tracking
- • Open source: deployed on premise and is free to use

Levels of System Software →

- System software is a set of programs that handles all the basic internal working of a computer.

Levels of System Software are

1. Operating System

An operating system is system software that controls the working of computer hardware and software. Moreover, it acts as a common connection between the computer hardware and software. In other words, we can also call it an interface between the hardware and the users.

Some important tasks performed by the operating system are:

- Scheduling
- Memory Management
- File Management
- Security
- Protects data and other software from unauthorized access .

2. Language Processors

It is a special type of system software that converts the source code into machine code. The input given has to be in object code only hence, we use language processors. Also, the machine code executes faster as compared to the source code.

- Source Code
- Object Code

- Different Types of Language Processors are:

Assembler	It converts assembly language to machine language.
Interpreter	It is a type of system software that executes the program line by line.
Compiler	It is also a type of system software that executes the whole program at once.

3. Utility Software

- Application Software
- System Utilities

These types of system software are used for the proper and smooth functioning of the computer system. They perform functions like removing outdated files, recover data which is accidentally lost, finding information, arranging data and files in an orderly manner, compress disk drive, install and uninstall programs, etc.

- Different types of utility software are:

1. Antivirus Software They are used to protect the system from viruses. Some examples are Quick Heal, McAfee, etc.

2. Compression Tools They help compress large files. The files can be changed to the original form when we require it. Examples are WinRAR, PeaZip, etc.

3. Disk Management Tools

They are used to manage data on the disks efficiently so that the system performance can enhance. Examples are Disk Cleanup Tool, Backup Utility, etc.

4. **Device Drivers**

- These types of system software are used for the operation of the peripheral devices. Each device connected to the computer has its own driver. These drivers basically contain instructions that tell the operating system how to operate the device.
- Some drivers are pre-installed on the computer while some others are installed when a new device is added. The audio device, video device, scanner, camera, etc. all require a driver. A driver tells the operating system how to use the device.

Assembler Design →

- Scanning
- Parsing (Syntax Analyser)
- Forward Referencing
- Symbol Table
- Machine Code Generation

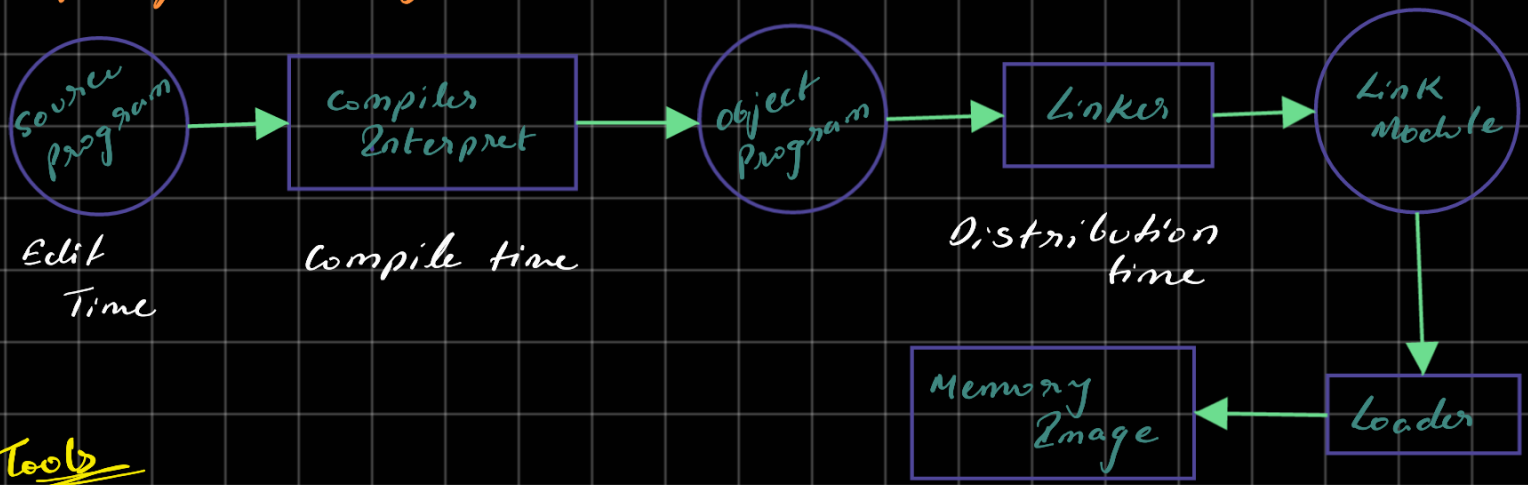
Operating System functions →

- User interface and input/output management →
  - User interface : allows individuals to access and command the computer system.
  - Command-based user interface : requires that text commands to be given to the computer to perform basic activities.
  - Graphical user interface : uses icons and menus displayed on screen to send commands to the computer system.
- Hardware independence →
  - Application Software : allows applications to make use of the operating system.
- Memory Management →
- Processes Tasks →
  - Multitasking
  - Time-sharing
  - Scalability
- File Mangement →
- Security →

.py → translator → .obj → .exe

Source Program — Compiler — object Program — executable form.

Life Cycle of Source Program →



Tools

Source program ex →

1. GitHub
2. Team foundation server
3. Apache subversion

Application Software

open for all

1. Horizontal application software (General Purpose)
2. Vertical application software (Particular / personal perpose)

System Software Development

1. Requirement Analysis and Resource planing / finalizing.
2. Design and Prototyping.
   - Algorithm
   - Flow chart / DFD
3. Software Development / coding / Implementing
4. Testing / Deployment of software
5. Maintainence

Testing { α - developer end, β - user end }

Recent Trends in Software Development

1. Progressive Web-Application (PWA)
2. Dev Ops (Development Operations)
3. Cyber security
4. Cloud Native Apps
5. IOB (Internet Of Behaviour)

Assembler →

Converts assembly language to machine level language.

- Pass 1 (one pass assembler)
- Pass 2 (2-pass assembler)

HLL → Human language
ALL → symbol
MLL → Machine

Elements of Assembly language →

The various elements that are possible are →

① Labels — symbols (Names)
② Opcodes / operands — (logical / Mathematical)
③ Directives — (Assembler Directives) EQU & SET
④ Comments

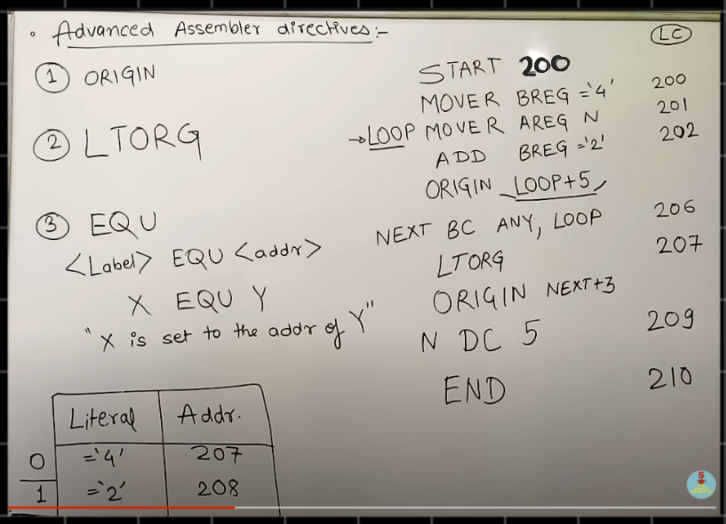
Symbols written in capital letters. ex → = EQU
→ SET

Assembly language should start with any of these.
Should have space, commas and end with ';'.

Advanced Assembler Directives ⟶

Directive →
are the commands of assembly language and has no influence on the operation of microcontroller.

- **Origin (xyz)**
  address will be assigned from xyz onwards.



- **LTORG**
  Literals in the literal table will only be assigned address after LTORG only.

Literal	Address
⇒ '4'	–
⇒ '2'	–

} before LTORG is called.

after →

Literal	Address
⇒ '4'	206
⇒ '2'	207

- **EQU**
  X EQU Y
  Label/variable X is set to the address of Y.

Data Structures in Pass-1

I. Assembly prg.
II. M/C opcode Table (MOT)
III. Symbol table:

Symbol	addr

IV. Literal table:

Literal	addr

V. Pool Table:-

Mnemonic	class	m/c code	length
STOP	IS	00	1
ADD	IS	01	1
SUB	IS	02	1
MULTI	IS	03	1
MOVER	IS	04	1
MOVEM	IS	05	1
COMB	IS	06	1
BC	IS	07	1
DIV	IS	08	1
READ	IS	09	1
PRINT	IS	10	1
END	AD	02	–
START	AD	01	–
ORIGIN	AD	03	–
EQU	AD	04	–
LTORG	AD	05	–
DS	DL	01	–
DC	DL	02	1
A – C	RG	01 – 03	–
EQ, LT, GT LE, GE, NE	CC	01 – 06	

Design of Assembler →

1. Specify the problem
2. Specify the Data structure / Define format of data structure
3. Specify algorithm
4. Look for modularity

↳ Repeat 1-4 until problem is solved.

1. Pass 1

① Defines symbols — ST (Symbol Table)
literals — LT (Literal Table)
MOT (Machine operation)
LC (Location Counter)

literals → same value and name in ALL.

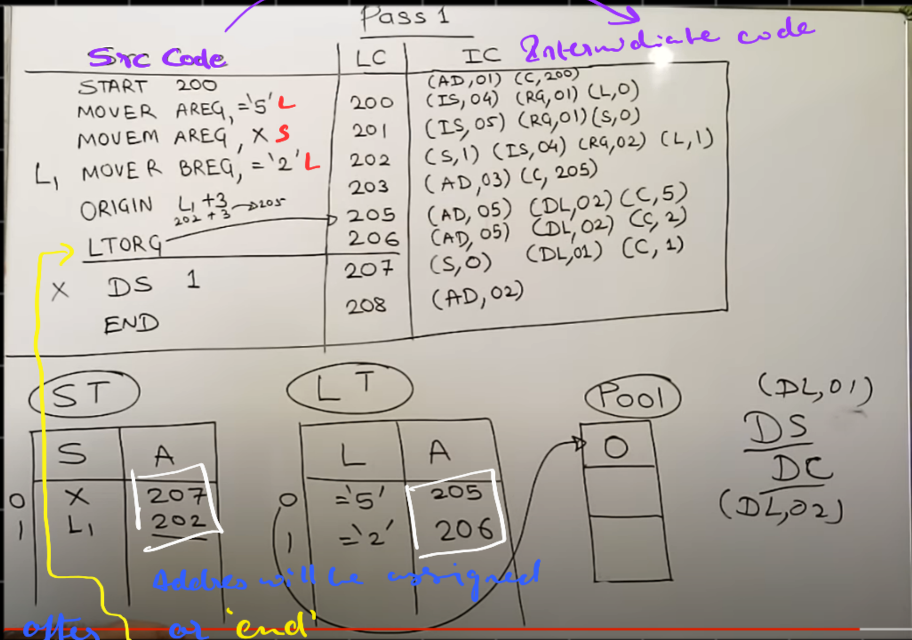
2. Pass 2

Translation of Object code
Put values of symbols

Pass 1 OS	Pass 2 OS
Source Program	Copy of file from Pass 1
LC	LC
MOT	MOT
POT (Pseudo Operation Table)	POT
ST	ST
LC	BT (Base Table)
File passed to Pass 2 →	

Pass-1 ⟶

1) Symol Table / Literal Table created, but no address will be provided until until "LTORG" or "end" keywords.



- Seperates the Symbols, Mnemonic opcode and operand fields.
- Buid symbol table
- Perform LC (Location Counter) processing.
- Construct Intermediate Representation

LTORG → Literal Table origin.

assign address for Literal's in Literal table

DS → Data Storage

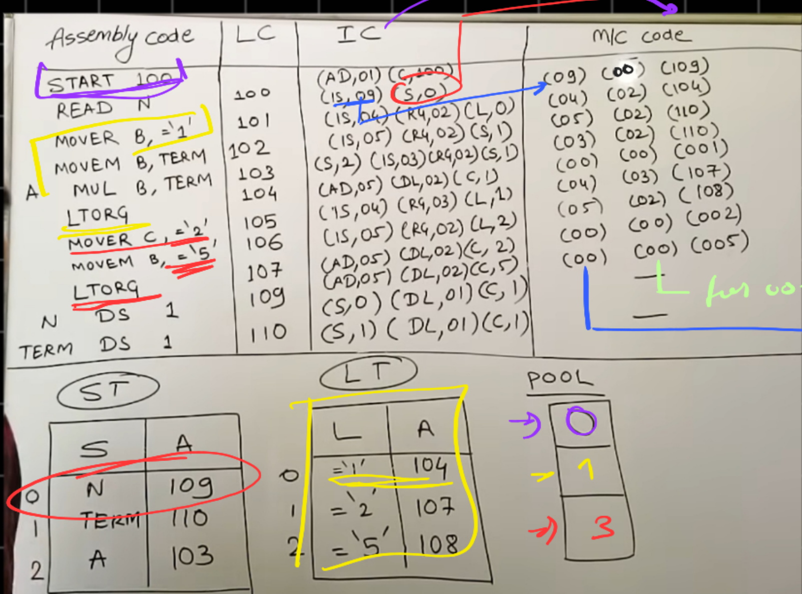
assign address for symbol's in symbol table.

Data Structures →

- OPTAB (operation table)
- SYMTAB (symbol table)
- LIT TAB (Literal table)
- POOL TAB (Pool table)
- MOT (Machine Operation Table)
- POT (Pseudo Operation Table)
- LC (Location Counter)

Pass – 2 →

Pass 1 → machine code



In 'ST' at index '0'.

(IS, 09) (S, 0)

(09) (00) (109)

Register code

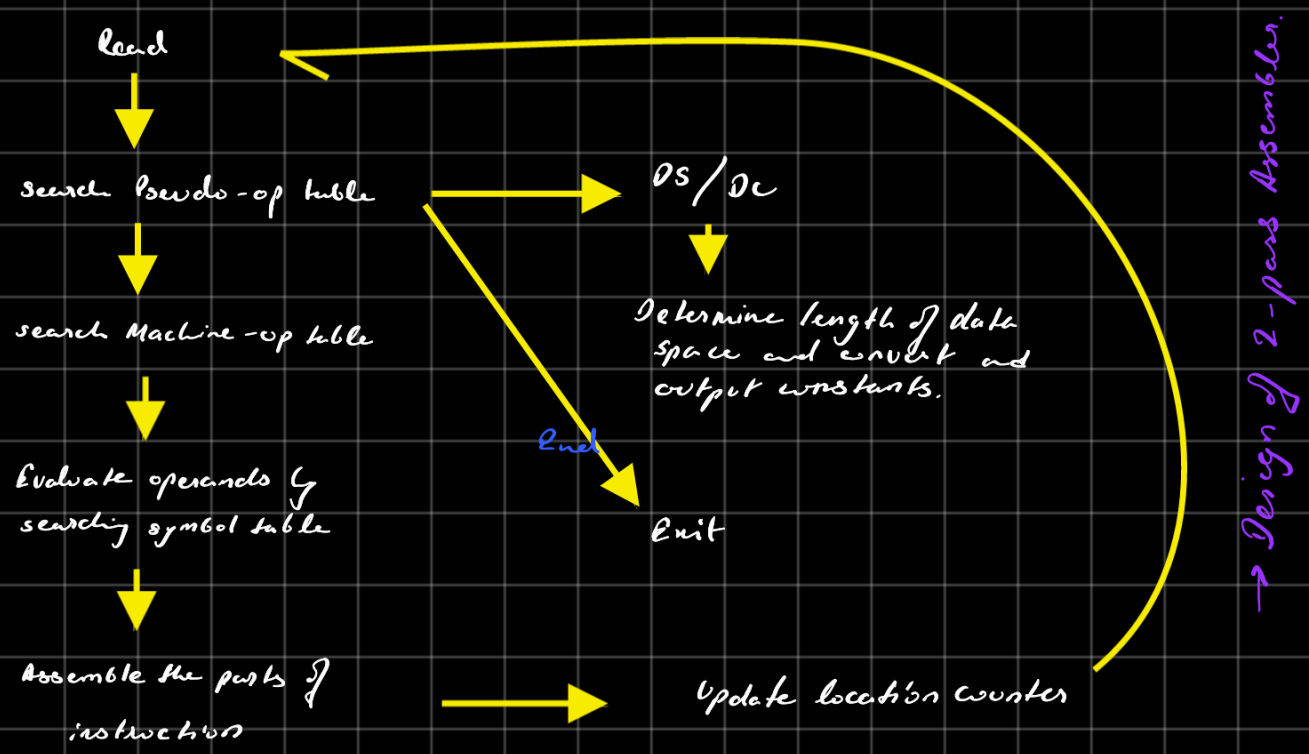
00 – non register

01 – A register

: – :

for use of Registers value of 1st tupple

- 2-pass system is to address the problem of "forwarding references" — references to variables or subroutines that have not yet been encountered when parsing the source code. A strict 1-pass scanner cannot assemble source code which contains forward references. Pass 1 of the assembler scans the source, determining the size and address of all data and instructions; then pass 2 scans the source again, outputting the binary object code.



Working of pass-2

(Symbol Table)

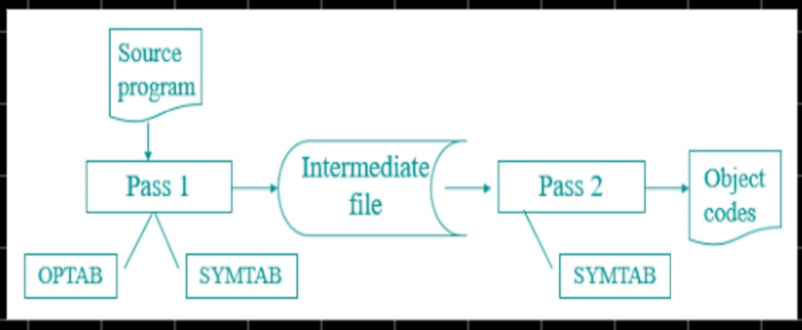
- Generates machine code by converting symbolic-machine-opcode into their respective bit configuration.
- It stores all machine-opcodes in MOT table, with symbolic code, their length and their bit configuration.

  It also process pseudo-ops and store them in POT table (pseudo-op table).

Data Structures →

- MOT table (machine opcode table)
- POT table (pseudo opcode table)
- Base table (storing values of base registers)
- LC (Location Counter)
- ST (Symbol Table)

Design of 2-pass Assembler →



One-pass Assembler →

- The assembler reads the source file once.
- During single pass, the assembler handles both label definitions and assembly.
- Only problem is future symbols.

Two-pass Assembler →

- Performs two passes over the source file.
- In "first pass" looks for label definitions.
- All labels are collected, assigned values are placed in symbol table.
- In "second pass", instruction are read again and assembled, using the symbol table.

One pass And two pass Assembler Implementation →

Label	Opcode	Operands
John	START	200
	MOVER	R_1 = '3'
200	MOVEM	R_1 X
201		
L1	MOVER	R_2 = '2'
202	LTORG	
203		
X	DS	1
204		
	END	
205		

Declaration Statement

Forward reference for 'X'

	LT	Address
1	'3'	203
2	'2'	—

	ST	Address
1	X	204
2	L1	202

Single pass 'Assembler for 8086' →

Labels — Reserved Words & Identifier

Assembler Directive — case Insensitive

Operands — letters, _, #, @

Addressing Modes →

1. Register — ADD A,B
2. Immediate — ADD 5,6
3. Memory — ADD 0014,0024
   - ① Direct
   - ② Indirect

opcode	operand
ADD	6,5

Data Structure →

① Opcode template

② Operand template

Multipass Assembler →

```
ALPHA   EQU   BETA
BETA    EQU   GAMMA
GAMMA   RESW  1
```

- Alpha and Beta cannot be defined in "pass 1."
- While doing multipass processing, DELTA is defined in pass 1, BETA in pass 2, Alpha in pass 3.

This is the motivation for using a multipass assembler.

Variants of Assembler →

1. GNU Assembler
2. 45 xxxx Cross Assemblers
3. Amsterdam Compiler Kit (ACK)
4. single target Assembler

Macro →

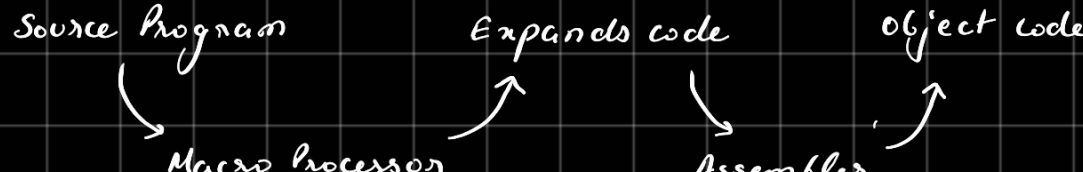
Macro represents a commonly used group of statements in the source programming language.

Macro Processor →

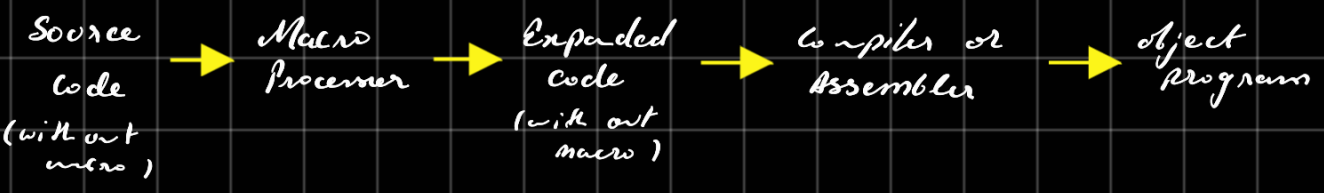
Replaces each macro instruction with the corresponding group of source language statement called "expanding macro".

Label	opcode	operand
	MACRO	&X, &Y
	DIFF	&X, &Y
	DA	&X
	MOV	B, A
	LDA	&Y
	ADD	B
	MEND	

Each parameter starts with &.



Label	Opcode	Operand
SWAP	MACRO	&x, &y
	LDA	&x
	LDX	&y
STORE	MACRO	&x, &y
	STA	&y
	STA	&x
	MEND	
	MEND	



Design of Macro Assembler →

pass 1 →

- Macro definition processing
- SYMTAB construction

pass - 2 →

- Macro expansion
- Memory allocation & LC processing.
- Processing of Literals
- Intermediate code Generation.

pass - 3 →

- Target code Generation.

Data structures used →

1. DEFTAB
2. NAMTAB
3. ARGTAB

Difference Between Macro Processor and Macro Assembler:

Macro Assembler	Macro Processor
1)It performs Expansion as well as generate object code	It only Performs Expansion
2) Expansion is performed during Design Analysis	It may have its own pass structure
3)It doesn't require separate preprocessor and Assembler	It require separate preprocessor and Assembler
4)From user point of view it is very simple	The user have to run preprocessor first and then assembler to generate object code . So its complicated
Less Time Consuming	More time consuming
Complex Logic	Comparatively easy Logic
Data Structure Used : All data structures used by Macro processor and SYTAB , OPTAB , LITAB etc	Data Structure used : MNT , MDT , PNTAB , EVTAB, SSNTAB, KPDTAB,SSTAB
It has three phases : Macro Definition Processing , Macro Expansion and simple Assembling	It has two Phases : Macro Processing and Macro Expansion

Flow of execution →
- sequential
- conditional
- looping

Macro Expansion → (MEC)

Algo while there is no MEND

a) If normal seq statement

i) Expand statement

ii) MEC = MEC + 1

else

MEC = Address of next statement to be expanded

Exit.

Advanced Macro facilities →

1. Expansion time variables

Expansion time sequencing symbols	Expansion time statement
SS	AIF, AGO, ANOP

labels
↓
SS
Sum MACRO

Nested Macro →

Label	opcode	Operand
SWAP	MACRO	&X, &Y
	LDA	&X
	LDX	&Y
STORE	MACRO	&X, &Y
	STA	&Y
	STX	&X
	MEND	
	MEND	

Flow in control in Macro's →

① Sequential
② Conditional
③ Looping

MEC (Macro Execution Count)
+1 with every single macro seen.

Advanced Macro facilites →

1. Expansion time sequencing symbols

   <SS>

2. Expansion time statements

   AIF

   AGO

   ANOP (No operation)

AIF (Expression) <SS>

AGO <SS>

<SS> ANOP

```
Test    MACRO    &X, &Y, &Z
        AIF (&X EQU 5).only
        MOVER 3 &.
```

Pass 1

- Macro definition
- Symbol table constant

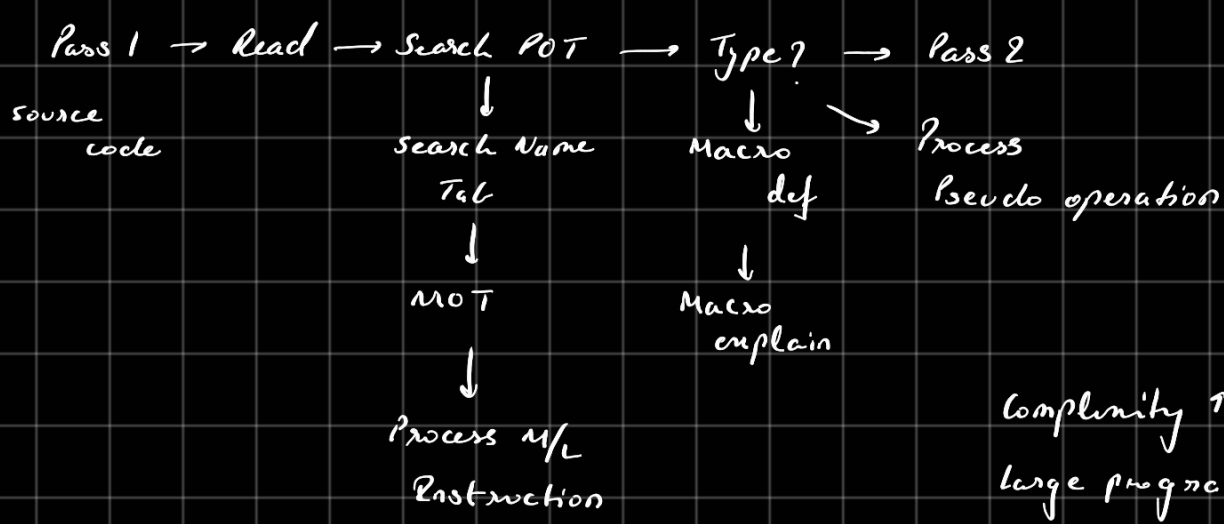
Pass 2

- Macro expansion
- Process symbols/literals
- Intermediate code generation

Pass 3

- Object code generation

Macro Assembler flowchart



Complinity ↑ ⇒ disadvantages when macro, large program

Basic tasks of Macroprocessor

- Macro definition
- Macro Invocation
- Macro expansion

Basic tasks of Macro processor

- Name Tab
- Definition Tab
- ARG Table

- Definition of Macro
- Move the macro
- Expansion (substitution)

```
label   opcode    operand
SUM     MACRO     &A, &B
        LDA       &A
        MOV       B
        LDA       B
        ADD       B
        MEND
```

Design issues of Macroprocessor

1. Flexible Data Structures
2. Attributes / Parameters Names shouldn't be duplicated
3. Defaut Argument / Numeric value at Macro definition time.
4. Comments should be clearly specified.

Unit - 2

Compiler

- Converts the entire source code into exec. at a time
- Execution Time of the code is faster.
- Generates Intermidiate Object code.
- ex → C, C++

Interpreter

- runs the code line by line.
- execution time of code is slow.
- No Object code is generated.
- ex → Python, Perl

Assembler

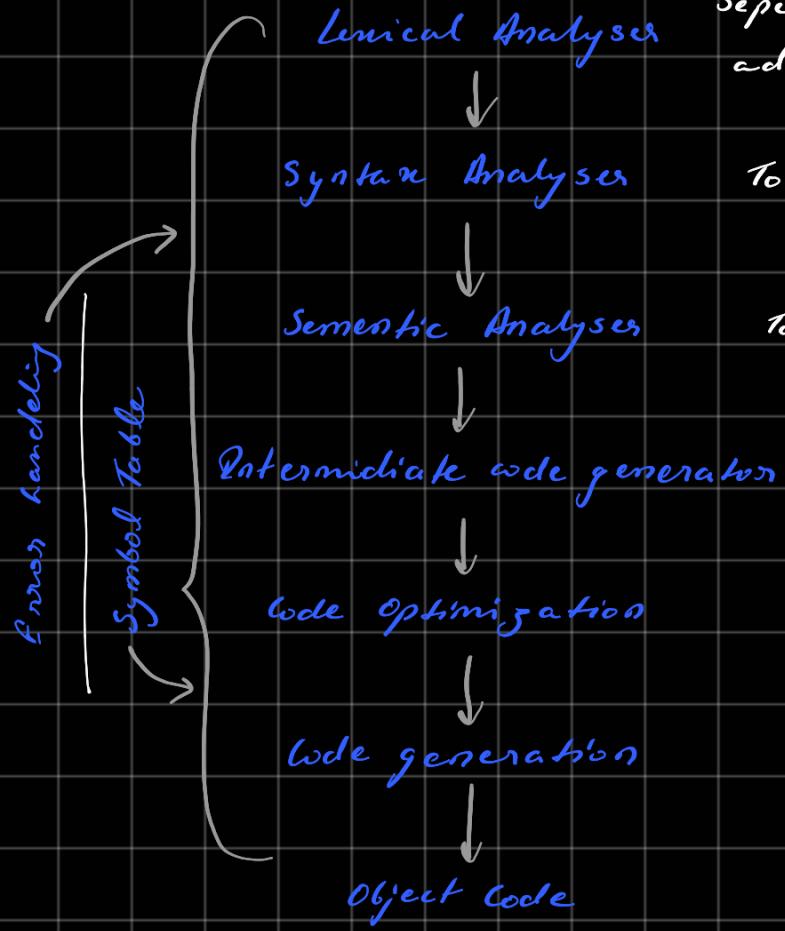
- LLL to MLL
- One to one
- translate entire program before running
- Requirement of memory less
- Used only once to create .exe file.

Interpreter

- HLL to MLL
- One to many
- translate entire program line by line.
- Requirement of memory more.
- Used every time when the program is running.

Subroutines → function is known as subroutine.

Phases in Compiler →



Seperate / divide the program into tokens, and stored in different tables, keywords...

To check the syntax in the program.

To check if the code is logically correct. also if the parsed tree is making any sence.

white code / board in java.

Reducing space & Time complexity. Of two types machine dependent / independent.

$c = a + b \times 5$

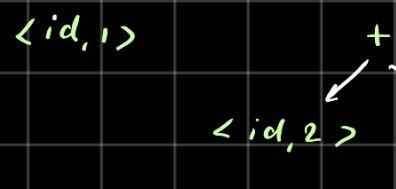
↓

Lexical Analysis

$\langle id,1 \rangle \langle = \rangle \langle id,2 \rangle \langle + \rangle \langle id,3 \rangle \langle * \rangle \langle 5 \rangle$

↓

Syntax Analysis



<id, 3> inttofloat
|
5

↓

Intermidiate code Generator

$t_1 = inttofloat(5)$
$t_2 = id_3 * t_1$
$t_3 = id_2 + t_2$
$id_1 = t_3$

↓

Code optimizer

$t_1 = id_3 * 5$
$id_1 = id_2 + t_1$

↓

Code generator

LDF R_2, id_3
MULF R_2, #5.0
LDF R_1, id_2
ADDF R_1, R_2
STF id_1, R_1

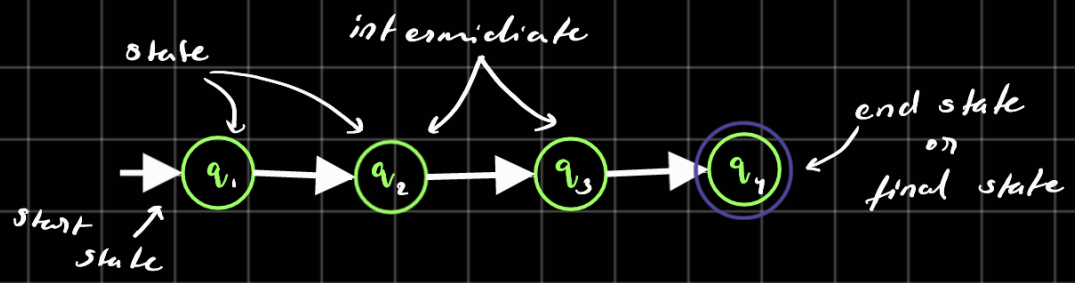
Grammer and finite Automata

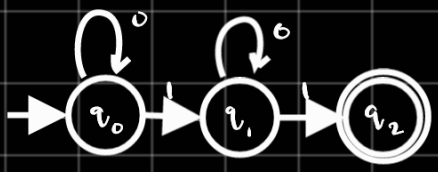
Grammer = { V, T, S, P } $\Sigma \Leftrightarrow T$

V → Variable capital letters
T → Terminal small letters, symbols
S → start
P → Production

Finite Automata (FA) is the simplest machine to recognize pattern.
A finite Automata consists of the following →

Q	finite set of states.
Σ	set of input symbols
q	Initial state
F	set of final states
δ	Transition function





$\delta(q_0, 0) \rightarrow q_0$

$\delta(q_0, 1) \rightarrow q_1$

$\delta(q_1, 0) \rightarrow q_1$

$\delta(q_1, 1) \rightarrow q_2$

$\delta = Q \times \Sigma \rightarrow Q$

$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{0, 1\}$

$q = q_0$

$F = \{q_2\}$

Finite Automata

DFA

Determinate Finite Automata

transition will lead to one exact state.

NFA

Non Deterministic Finite Automata

to any different state

Null or ε

DFA

$Q \times \Sigma \to Q$

NFA

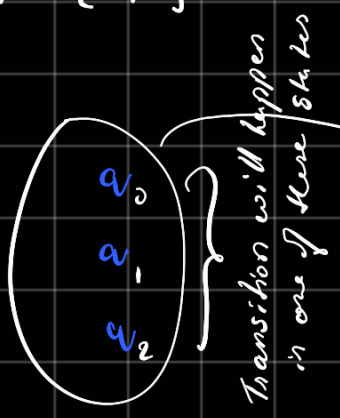
$Q \times (\Sigma \cup \varepsilon) \to 2^Q$ } ϕ

$Q \Rightarrow (q_0, q_1, q_2) \quad \Sigma = \{0, 1\}$

DFA $(Q, \Sigma, q, F, \delta)$

$(q_0, q_1, q_2) \times \{0, 1\}$

q_0	0
q_0	1
q_1	0
q_1	1
q_2	0
q_2	1



NFA $(Q, \Sigma, q, F, \delta)$

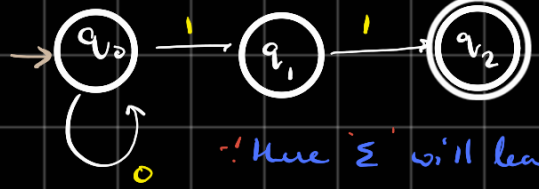
$(q_0, q_1, q_2) \times \{\phi, 0, 1\}$

q_0	q_0, q_1
q_1	q_0, q_2
q_2	q_1, q_2
	q_0, q_1, q_2
	+ ϕ

$\phi, q_0, q_1, q_2, q_0q_1, q_0q_2, q_1q_2, q_0q_1q_2$

Transaction will happen in any one of these states.

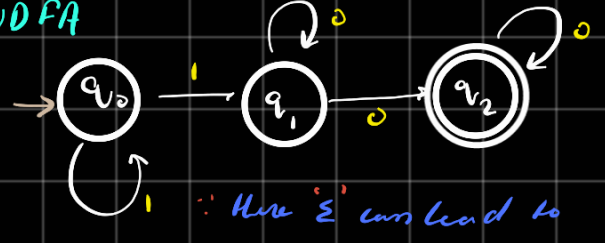
DFA



∴ Here Σ will lead from one state to another.

$q_0 \xrightarrow{1} q_1$ or $q_0 \overset{0}{\circlearrowright}$

NDFA



∴ Here Σ can lead to multiple states or no state. or null state.

$q_0 \xrightarrow{1} q_1$ & $q_0 \xrightarrow{0} q_0$

Compiler Bootstrapping

The process of writing a compiler in the source programming language that it intends to compile.

This type of technique is known as "self hosting compiler".

Lexical Analyser

- Regular Language (Natural language)

- Stashing of the code.
- Breaks the source code into token, pattern, Lexeme.

– Token

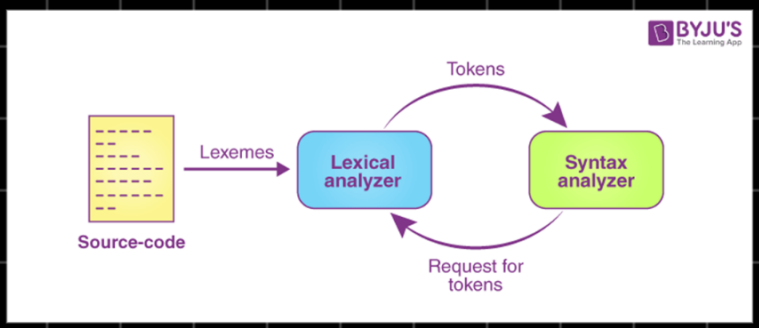
Smallest individual unit in side a code.

– Pattern

Description used by Token

– Lexeme

Sequence of characters in the source code.



Syntax Analyzer – Content free grammer

$(NUT)^{+} \rightarrow 1, +\infty$, can't be null ie, 1, 2, ...

$(NUT)^{*} \rightarrow 0, +\infty$, can be null ie, 0, 1, 2...

System Analyzer →

- Creates syntactic structure of the given source program
- Syntactic structure is a parses tree.

System of a programing is described by a content-free grammar (CFG)

we will use BNK (Bakus Naur form) notation in the description of CFG.

Provides a precise specification of programing language.

- The syntax Analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammer.

Two types of parsers →

1. Top-Down Parser
2. Bottom-Up Parser

} scans the input from left to right

Efficient top-down, bottom-up parsers can be implemented only for sub-classes of context free grammers.

Scanner and Parser →

- LL for top-down parsing
- LR for bottom-up parsing

Top down - Parser

$S = A + B \quad -(1)$

$S = id + id$

$= A + id$ left approach

$= A + B$

→ is recursive in nature.

CFG consists of a finite set of grammer rules in a quadruple (N, T, P, S)

N → set of non-terminal symbols. / V → Variable also

T → set of terminals, $N \cap T = Null$

P → set of rules / Production Rules (Substitution Rules)

S → start symbol

- Let any set of production rules in a CFG be
- X → X+X | X*X |X| a
  The **leftmost derivation** for the string **"a+a*a"** may be –
- X → X+X → a+X → a + X*X → a+a*X → a+a*a
- The **rightmost derivation** for the above string **"a+a*a"** may be –
- X → X*X → X*a → X+X*a → X+a*a → a+a*a

- Ambiguous Grammer →

Memory	op	arg1	arg2
(0)	+	c	d
(1)	-	-b	-
(2)	*	(0)	(1)
(3)	=	(2)	

$a = -b * c + d$

$r_1 = -b.$

Intermediate

If a context free Grammer G has more than one derivation tree for some string $\omega \in L(G)$.

Intermidiate Code Generation →

During the translation of a source program into the object code for target machine, a compiler may generate a middle-level language code known as intermediate text.

- It should be easy to produce and translate.
- It is a sort of universal assembly language.

1) Quadriples
2) Triples
3) Indirect tuples

$a = b + c * d$

↓

$r_1 = c * d$

$r_2 = b + r_1$

$a = r_2$

Op	arg1	arg2	result
*	c	d	r1
+	b	r1	r2
+	r2		a

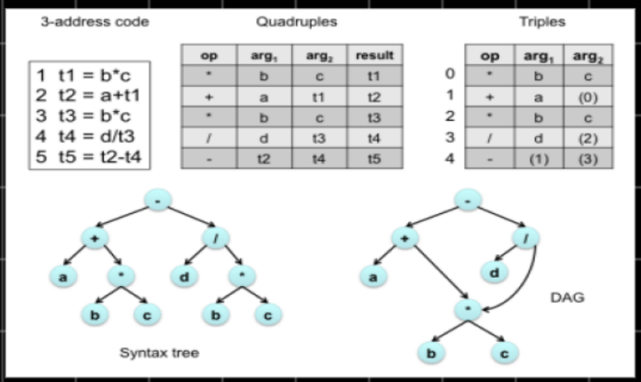
Example: a := -b * c + d

t_1 := -b t_2 := c + d M t_3 := t_1 * t_2 a := t_3

Three address code is as follows:

	Operator	Source 1	Source 2
(0)	uminus	b	-
(1)	+	c	d
(2)	*	(0)	(1)
(3)	:=	(2)	-

- Example: The three-address code for a+b*c-d/(b*c) is below
- t1 = b*c
- t2 = a+t1
- t3 = b*c
- t4 = d/t3
- t5 = t2-t4



Code Optimization techniques

1) Machine dependent optimization
- After the generation of target code and when the target code is transformed according to machine architecture.
- uses CPU registers and may have absolute memory location.

2) Machine independent optimization
- After the compiler takes intermidiate code and transform it such that it does not
- uses CPU registers and absolute memory location
- relative

3) Induction analysis
4) Loop optimization
5) Strength reduction
6) Deadlock Elimination
7) Partial - Deadlock

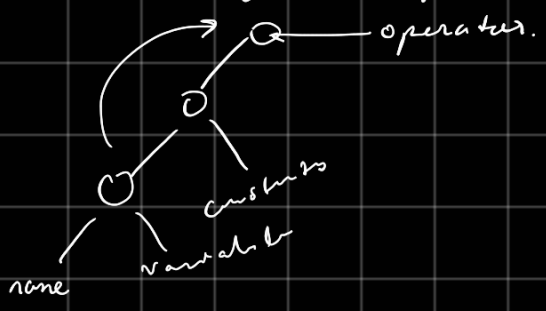
Peephole →

reduction in source code.

Code Generation →

1. Target language
2. Selection of Instructions
3. RR
4. Register allocation
5. ordering of instructions

DAG (Directed Acyclic Graph)



Descriptors →

The code generator has to keep the track record of registers and addresses.

Register descriptors →

To tell code generators about the register

Address descriptors →

To keep the track of values of identifiers address location.

getReg :- check register and
function values

LEX (Lexical Analyser Tool)

YACC (Yet Another Code Code)

Backus Naur Form

$E \rightarrow id$

$E \rightarrow E * E$

$E \rightarrow E + E$

.x + y * z	shift
x . + y * z	Reduce
E . + y * z	shift
E + . y * z	shift
E + y . * z	Reduce
E + E . * z	shift
E + E * . z	shift
E + E * z .	Reduce
E + E * E .	Reduce (Emit *)
E + E .	Reduce (Emit +)
E .	accept

Not doing 'E+E' → E because '*' has higher precedence than '+'

Recursive Descent Parsing → (Top-Down)

- Input is read from left to right
-

LL(K) K → number of look ahead parser

Predictive Parser →

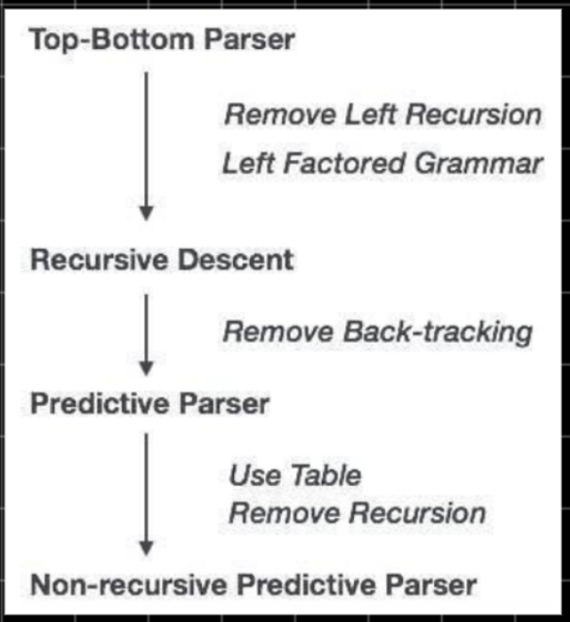
Top - down

LL(K)

left to right

K lookahead symbol.

left most derivation



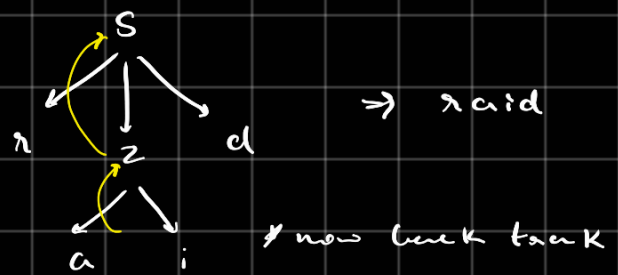
Back Tracking → Top down

S → aXd | aZd

X → oa | ea

Z → ai

S → read



⇒ aaid

✗ now back track

⇒ read

Unit - 3 → (Loaders and Linkers)

Function's for Computer Program Execution →

- Allocation
- Linking
- Relocation
- Loading

loader →
- Compile and Go loader
- General loader scheme
- Absolute Loader

Loader →

Loader is utility program responsible for allocation, relocation, linking and loading.

- **Allocation →**

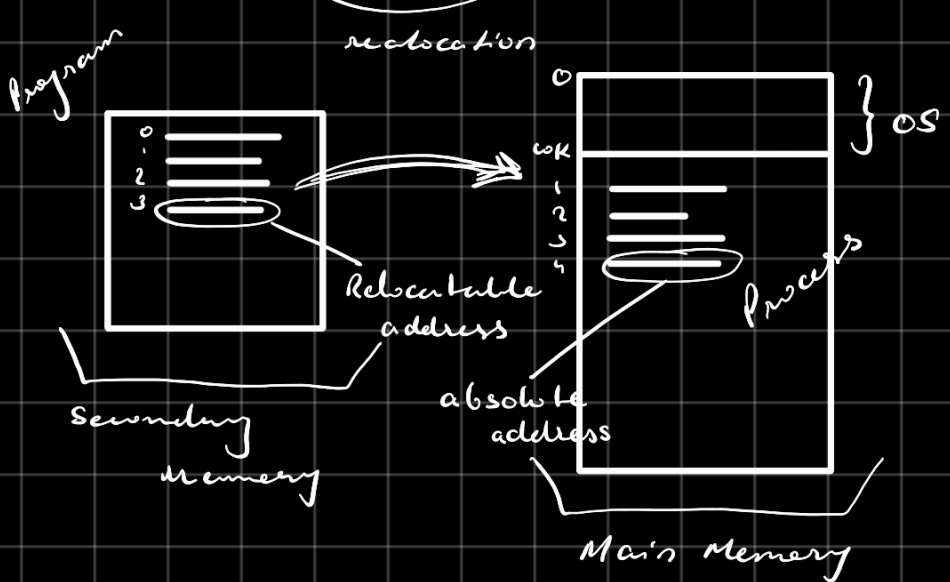
allocating the space for program in memory, by calculating the size of program

- **Relocation →**

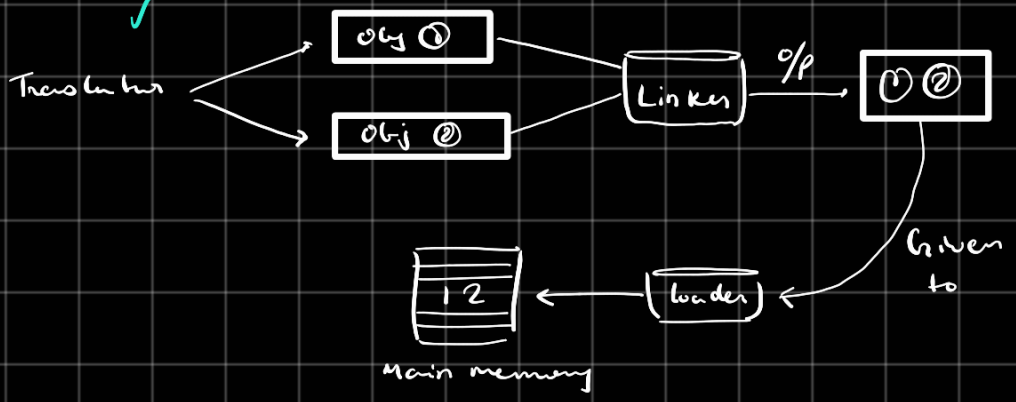
Program in secondary memory any relocatable address goes as process in main memory with absolute memory, this process is call relocation.

in main memory — is main memory

2000 — 6000 ← can be different

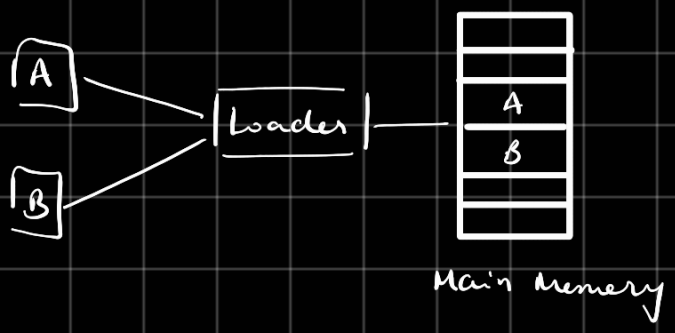


- Linking →



Linker links a number of files to merge and make a single program.

- Loader →



Loader pick the translated file from secondary and load it in main memory for execution.

Loader Schemes →

1. Compile and Go Loader
2. General Loader Scheme
3. Absolute Loader

- Compile and Go Loader →

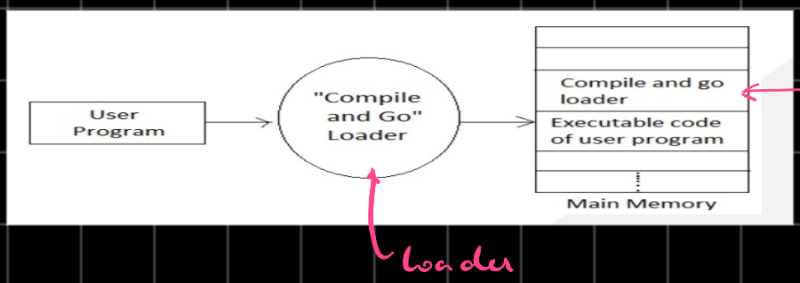
The instruction do read line by line, its machine code do obtained and it do directly put in the main memory.

- Assembler runs in one part of the memory
- Assembled machine instructions and data do directly put into assigned memory locations.

Advantages →

- Simple to implement.

Because assembler is placed at one part of the memory and loader simply loads assembled machine instructions into the memory.



Disadvantage →

- Assembler occupies space in memory.
- No obj is created
- Cannot handel multiple program in multiple languages.

A.C → A.obj → A.exe

Direct to .exe

so we need to compile again and again.

- General Loader Scheme →

The source program is converted to object program by assembler

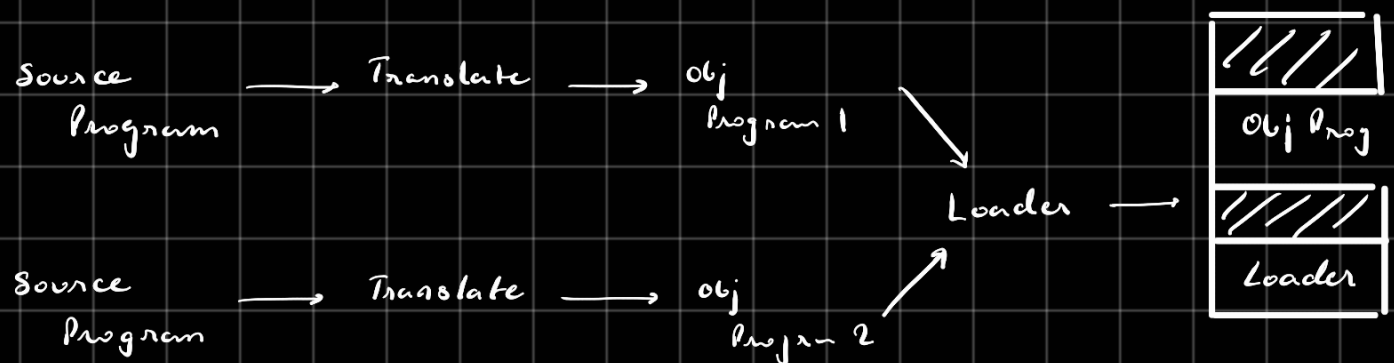
- loader occupies some portion of main memory.
- Loader accepts obj and puts machine instruction and data at assigned memory.

Advantages →

- Smaller than Assembler
- More memory is available to the user.

Disadvantages →

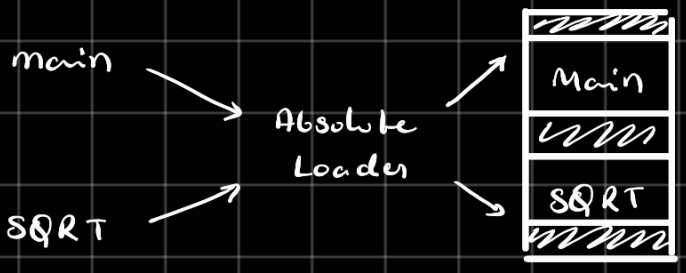
- Some portion of memory is occupied by the loader.



- Absolute Loader
  - Relocated object files are created.
  - Accepts these files and place them at specified location.
  - No reallocation.
  - Addres interdependent → with OS -

Advantages →
- Simple to implement
- Programmer has to specify the address to assembler.
- Process of execution.

Disadvantage →
- It is necessary for the programmer to know the memory management.



1) Allocation ⟷ program / Assembler
2) Linking ⟷ program / Assembler
3) Resolution ⟷ assembler
4) Loading ⟷ loader

- Relocating loader
  - Replacing symbols or names with actual usable address.
  - Like, binding, loading and allocation but to be done explicitly and manually by the programer.

main function

SQU ← sub-routine.

Editor →

- For performing functions on text file.
- Functions such as →
  - copy
  - paste
  - replacing
  - deleting
  - saveing

Need and purpose →

- For faulty document
- Transfering of data

Types of Editor →

- Line editor →
  - One line at a time.

- streams editor →
  - Can write a paragraph.

- Screen editor →
  - User can interact visually with the help of mouse pointer. eg :- Notepad

- Word Processor →
  - Allowed the insertion of image, files, video, fontstyle

- Structure Editor →
  - For writing and editing of code.

    eg :- Netbeans IDE

- Full screen Editors →
  - Is a full size text editor.
  - Only entire screen can be seen.
  - Word count a feature.

    eg:- Focus writer

- Multiple window Editors →
  - Working on more than one file at a time..
  - Cut paste from one file to another.

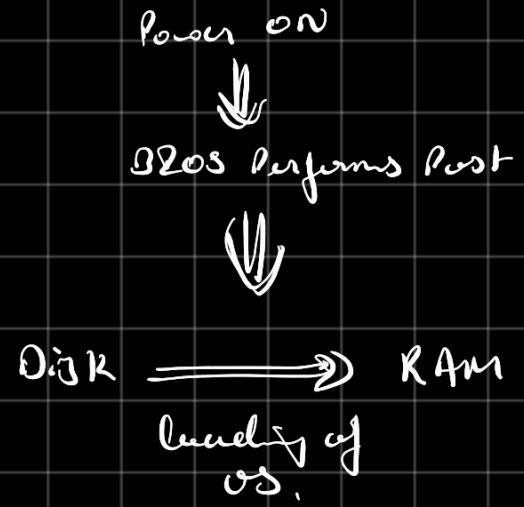
Booting →

Bootstraping →
loading kernal in main memory.

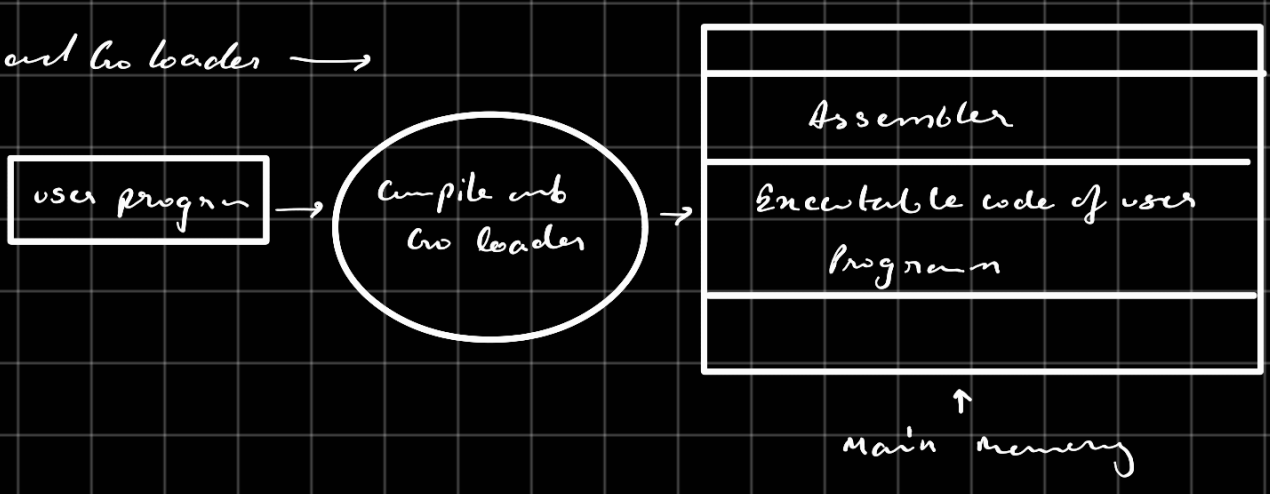
Dual booting →
loading 2 operating system

warm Booting →
Starting system

Cold Booting →
Computer Restart, due to power cut.



Compile and Go loader →



General loader scheme →

Absolute loader →

↓ improved

Relocation loader →

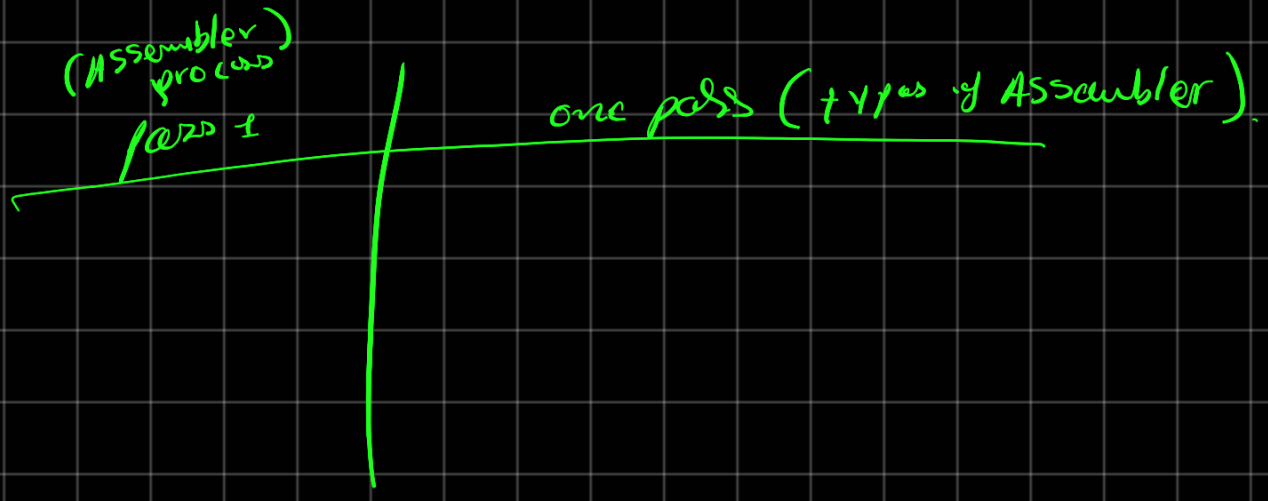
ex → Binary Symbolic subroutine (BSS)

Linking loader →

Data structures used → ESTAB (External Symbol Table)
PROGADDR (Program load address)
CSADDR (Control section address)

MS-DOS linker →

Difference between Linker and loader →



Unit 1

Definition / short note / block diagram

Assembler passes – detailed description / flow chart / short note

Macro, Macro process – Passes, Macro assembler

Unit 2

compiler all phases complete description

Passing types – all important – with 1 implementation example

Design finite automata with regular expression

Code optimization

Unit - 3

Booting and its types

Operating system architecture and functions

Loading types - all

Case studies - Vi, MS-DOS

JDBC
with MySQL
Driver → Database
Callable.

RMI ⟵ Rest ⟶

E