INTRODUCTION OF SOFTWARE PROGRAMMING

Preprocessor:
A preprocessor is a program that processes its input data to produce output that is used as input to another program.

Editor:
A text editor is a type of computer program that edits plain text. Such programs are known as "notepad" software

Compiler:
A program that translates high level language into machine code

Assembler:
A program that translates ALP into machine code.

Loader:
A program used by operating system to load programs from secondary memory to main so as to be executed.

Linker:
A computer program that takes one or more object files generated by a compiler and combines them into one, executable program.

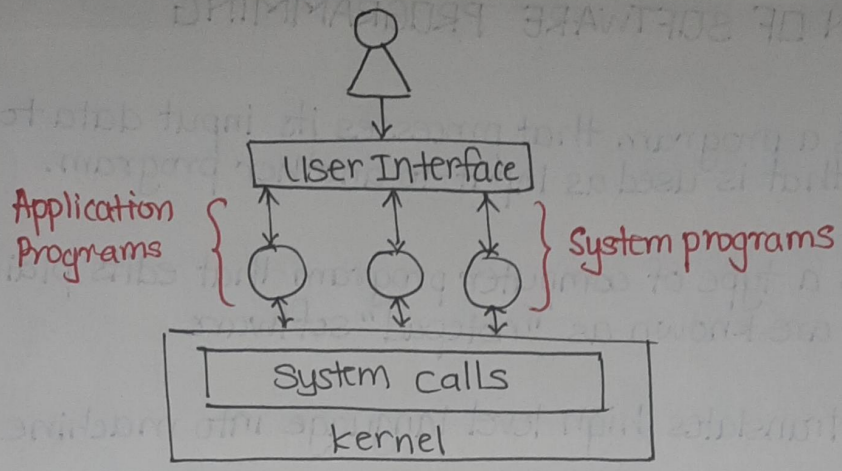
Debugger:
A program which is used to test program or execute program in single step execution.

What is System?
A collection of various components is called a system.

What is Programming?
Art of designing and implementing ideas through code.

What is system programming?
System programming involves development of individual pieces of software that allows the entire system to function as a unit.
System programming involves many layers such as operating system, firmware and development environment.

Need of System programming.
System program provide an environment where programs can be developed and executed.
In simplest sense. system program also provides bridge between UI & system calls.
There are much more complex system program such as compiler



effective performance
effective execution

effective utilization of facilities
make available new, better facilities.

SOFTWARE HIERARCHY

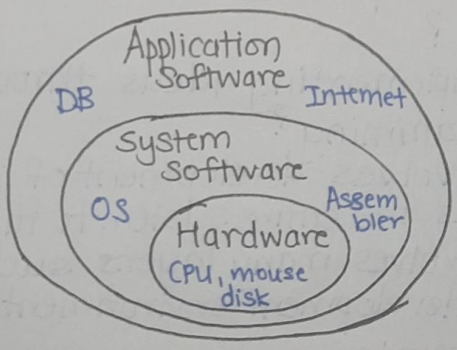
A software hierarchy is combination of product, version and release that represents an item of software in a database or knowledge base. The product is the root of hierarchy

- Software :

1. System software
   - Operating system
   - System support
   - System development

2. Application software.
   - General purpose
   - Application specific.

{These program assist general user application program} {These are software developed for specific use}



Detail	System Software	Application Software
Uses	used for operating computer hardware	used by user to perform specific task.
Installation	When operating system is installed	According to user's requirements.
Interaction	user doesn't interact as it runs in background	user interact with it.
Execution	Can run independently & ~~perform~~ provides platform	Can't run independantly without system software
Purpose	General purpose	specific purpose
Language	Low level language	High level language
Size	Small in size	Large in size
Design	complex to design & implement	Easier to design & implement.
Examples	Compiler, assembler etc	Word processor, web browse

TEXT EDITOR:
A computer program that allows us to create and revise a document
It is used for editing plain text files.
With its help you can write your programs (C++, Java)
Example. Notepad.

LINE EDITOR:
This code editor edits the file line by line
You cannot work on a stream of lines using this editor.

STREAM EDITOR:
In this the file is treated as a continuous flow or sequence of characters instead of line numbers, which means you can type paragraphs.

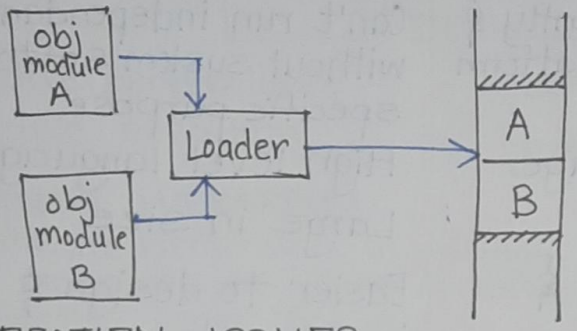
SCREEN EDITOR:
In this type of editor, users are able to see the cursor on the screen and make a copy cut paste actions easily.
Very easy to use mouse pointer.

WORD PROCESSOR:
Overcoming the limitations of screen editor, it allows one to use some format to insert images, files, videos, use font size, style features. It majorly focuses on Natural language

LOADER

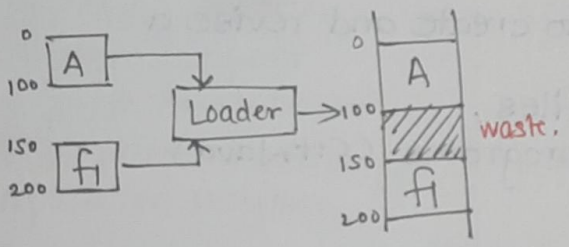
- ALLOCATION : Loader allocates space for program in main memory
- LINKING : Combines two or more seperate objects program.
- RELOCATION : Modifies object program so that it can be loaded at an address diffrent from originally specified.
- LOADING : Allocates memory location and brings object program into memory for execution.

GENERAL LOADING SCHEME

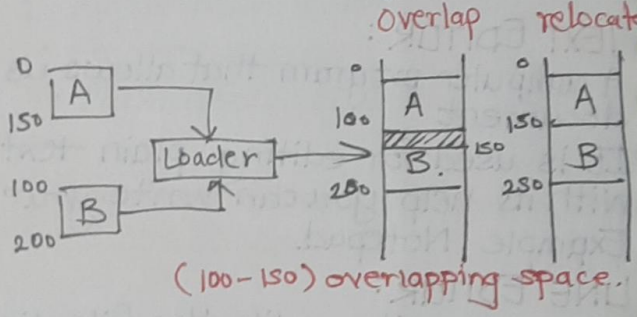


RELOCATION ISSUES.

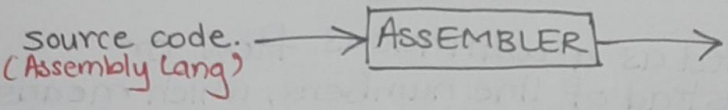
Case 1: Lots of wasted space.



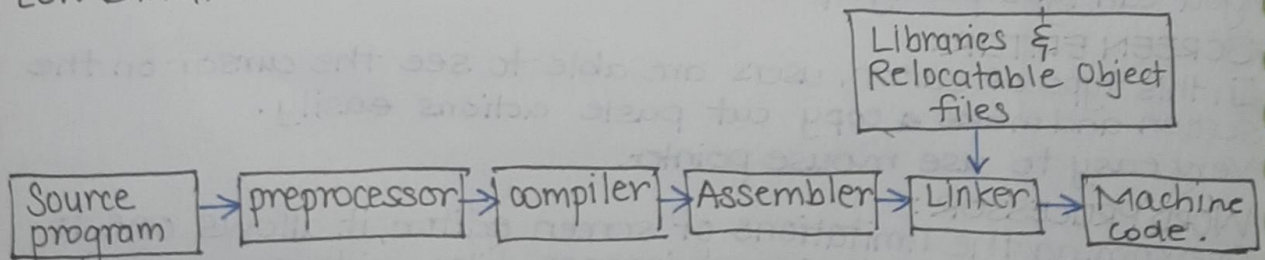
Case 2: Overlapping.



ASSEMBLER



FLOW DIAGRAM :

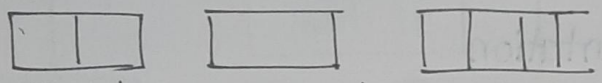


DEBUGGER:
for testing & debugging programs
facilities like: • Setting Breakpoint • Displaying values of variable.

MACHINE INSTRUCTION IN ALP

[Label:] Mnemonic [Operand operand] [;comments]

MACHINE INSTRUCTION FORMAT



opcode | register/ operands | memory operands.

ALP terms.

- Location counter (LC): points to next instruction.
- Literals : constant values
- Procedures: methods/functions

ALP statements

Imperative Statements

00 STOP
01 ADD
02 SUB
03 MUL
04 MOVER
05 MOVEM
06 COMP
07 BC
08 DIV
09 READ
10 PRINT

Declarative statement

00 DS
01 DC

Assembler Directive.

00 START
01 END
Advanced. { 02 LTORG, 03 ORIGIN, 04 EQU

Registers.

01 AREG
02 BREG
03 CREG

2 PASS ASSEMBLER

Requires two scans of program to generate machine code

Design :

Task performed by passes of two pass assembler are as-follows

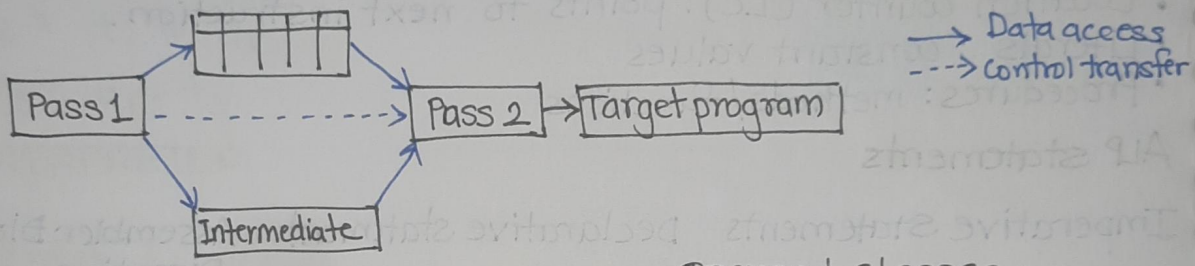
Pass 1: (Generates Intermediate code)

1. Seperate the symbol, mnemonic opcode & operand field
2. Build the Symbol table
3. Perform LC processing
4. Construct Intermediate representation

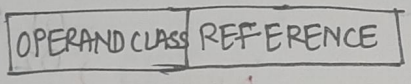
Pass2: (Generates Machine code)

1. Synthesize the target program.

TWO PASS ASSEMBLER



INTERMEDIATE CODE



Operand classes :

C - constant L - Literal
S - symbol RG - Register
cc - conditional code.

PASS I ALGORITHM

LC = 0;

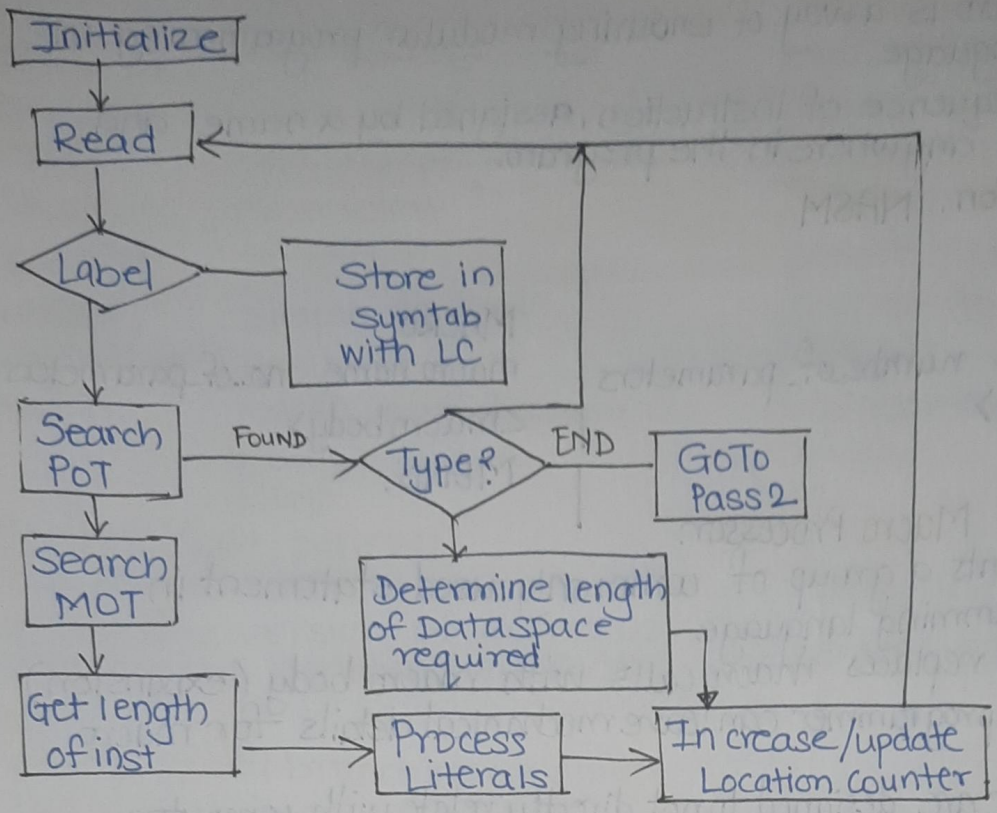
{

1. Read in line of Assembly code
2. If there is Label then insert it with LC in symbol table
3. If Assembler directive then process it
4. If executable instruction then generate machine code
   If instruction contains symbol then enter the same in Backpatch list

}

Process the Backpatch list

FLOWDIAGRAM



PASS II ASSEMBLER

1. Code_Area_Address = Address of Code Area.
2. For each address in LC[]
   { a If IS
      i) Read LC
      ii) Get opcode
      iii) Get operand/Literal/symbol from Table
      iv) Assemble inst in machin code_buffer
      v) Move content of this buffer in Code_Area ~~Address~~ at LC+ code_area_add
   b If DC
      i) Read LC
      ii) Assemble constant in machine code buffer
      iii) Move contents of this buffer in Code Area at LC+ code_area_address
   }
3. Write code area in output file.

UNIT 2

MACRO PROCESSOR & COMPILER

Writing a macro is a way of ensuring modular programming in Assembly Language

A macro is sequence of instruction, assigned by a name and could be used anywhere in the program.

Macro definition. NASM

Syntax

%macro macro_name numbe_of_parameters <macro body> %end macro	MACRO macro_name no_of_parameters <macro body> MEND.

Features of Macro Processor:

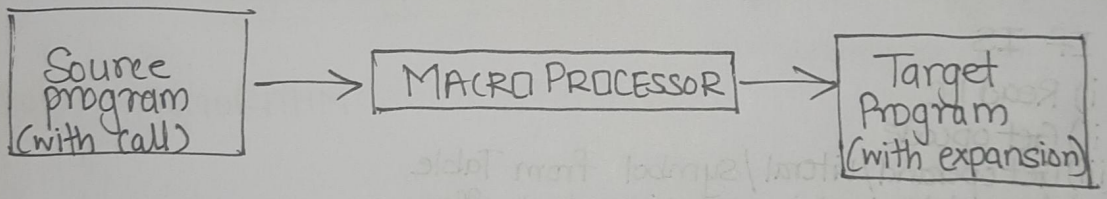
Macro represents a group of commonly used statement in source programming language

Macro process replaces macro calls with macro body (expansion)

Using Macro, programmer can leave mechanical details for macro processor

Macroprocesor are designed to not directly relate with computer architecture on which it runs

Macro processor involves definition, invocation and expansion



Macros are typically defined at the start of program

A macro call is writing macro name with actual parameters

FORMAL PARAMETER A parameter preceeding with '&' typically used in MACRO definition.

ACTUAL PARAMETER: The parameters which are actually passed during macro call

They will replace respective formal parameter in expansion

DETAIL	MACRO	SUBROUTINE
CALL	within program itself	outside program aswell
SPACE	More space required	Less space required.
Execution	Faster speed	slower speed.
Labels	Cant handle	Can handle
Executed by	Assembler	Hardware.
Code size	increases	doesnt increase
Usage.	Simple to write	Complex to write.
Understand.	Easy	Difficult.
Syntax	MACRO name [parameter] <body> MEND	SUBROUTINE name[parameter] <body> END

CONDITIONAL MACRO

The MACRO which can be used to generate multiple executable version of same code

They can be achieved with the help of AIG & AGO statement

AIG

used to specify branching condition.

Syntax: AIF (condition) Label

If true executes Label else continues

AGO

Provides unconditional branching

The statements are directives to macroprocessor & do not appear in expansion.

Example

```
Macro
eval1  &A, &B, &C
AIF(&A EQ &B) NEXT
LOAD &A
SUB  &B
ADD  &C
AGO .FIN
NEXT LOAD &C
FIN MEND.
```

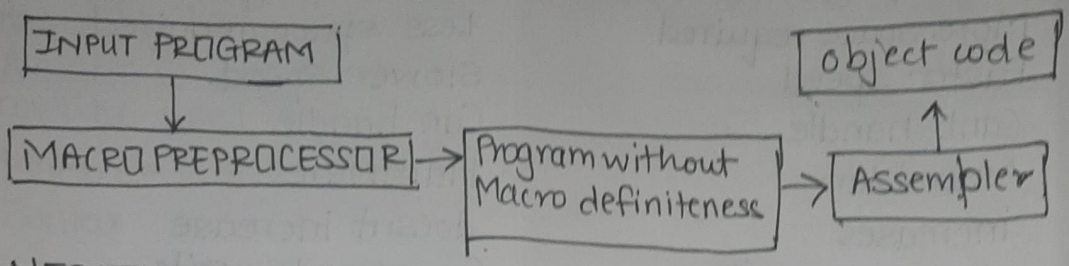
"Labels with '.' do not appear in expansion"

EVAL1 X,Y,Z ——— LOADX, SUB Y, ADD Z

EVAL1 X,X,Y ——— LOAD Y

} Same macro expands differently.

MACRO PROCESSOR.



NESTED MACRO.

A macro can be described within a macro body
This concept is used to define similar macros.
When outer macro is called then the inner macro comes into existence.
And thus inner macro can be called.

```
MACRO
DEFINE &SUB
MACRO
&SUB &y
MOVER AREG,&y
ADD AREG,='5'
MOVEM AREG,&y
MEND
MEND.
```

Here inner macro has same name as the outer macro's parameter.

MACRO PROCESSOR

PASS I :

Scan all macro definition one by one
a) Enter its name in MNT (Macroname Table)
b) Store definition in MDT (Macro definition Table)
c) Add details to MNT indication whether definition found in MDT
d) Prepare Argument list array.

PASS II:

Examine all statements for MACRO CALLS : For each call.
a) Locate macro name in MNT
b) Establish correspondance between Formal & Actual parameter
c) Obtain info about definition from MNT abt MDT locations.
d) Expand the macro call by picking up model statements.

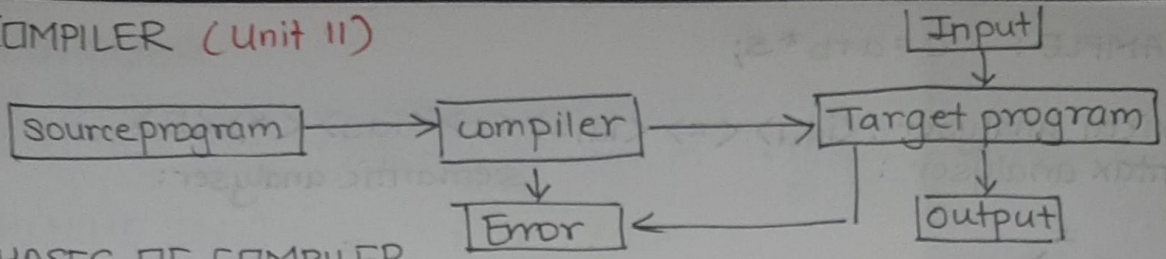
MNT

Ind	Name	MDT index

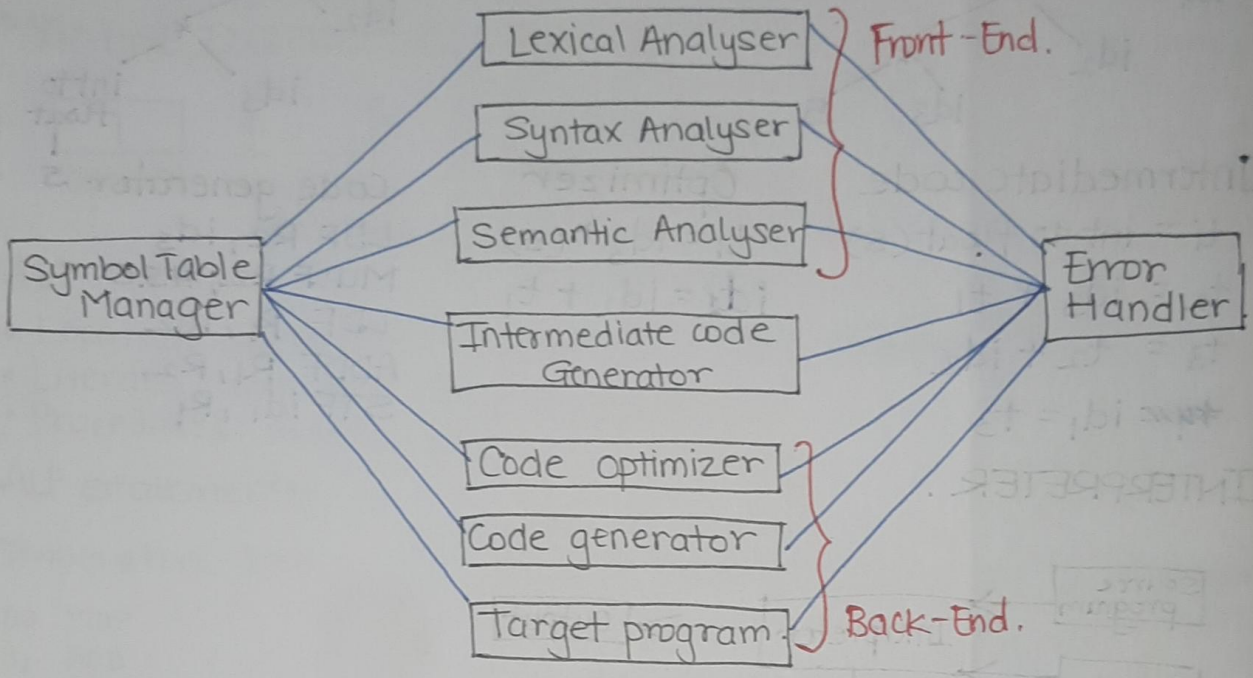
MDT

Definition	Index

COMPILER (Unit II)



PHASES OF COMPILER

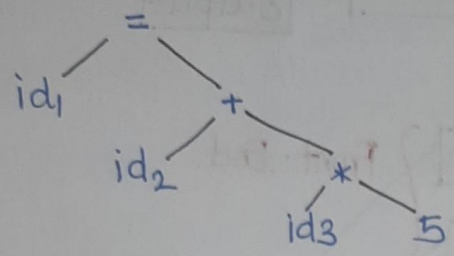


1. Lexical Analyser: Take input program one char at time and converts into meaningful lexemes in form of tokens.
2. Syntax Analyser: Takes tokens as input and generates parse tree as output.
   The parse checks the expression made is syntactically correct or not.
3. Semantic Analyser: checks whether parse tree follows rules of language.
   keeps track of identifiers. Output is annoted tree syntax.
4. Intermediate code generator: converts source program into intermediate code. It is between high level & Machine lang.
5. Code optimizer: optimizes the intermediate code so it can run faster and take less space.
6. Code generator: It take intermediate code & maps it to the target machine language.

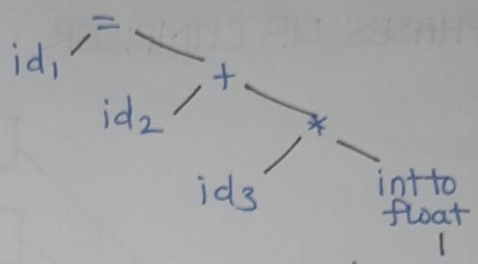
EXAMPLE: c = a+b*5;

lexical analyser : <id,1> <=> <id,2><+><id3><*><5>

Syntax analyser :



semantic analyser:



Intermediate code

t_1 = int to float(5)

t_2 = id_3 * t_1

t_3 = t_2 + id_2

id_1 = t_3

Optimizer

t_1 = id_3 * (5)

id_1 = id_2 + t_1

Code generator

LDF R2, id_3

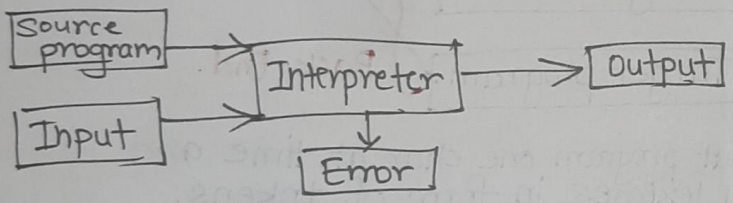
MULF R2; #5.0

LDF R1, id_2

ADDF R1, R2

STF id_1, R1

INTERPRETER.



DETAIL	COMPILER	Interpreter
Input program	Takes entire program	Takes single instruction.
Intermediate code	Generated	Not Generated
Condition Control statement	Executed faster	Executed slower
Memory	More required	Less required.
Error	Displayed after checking entire program	Displayed for every instruction.
Execution	Need not be compiled every time	Need to be converted from high to low every time.
Example	C, C++, JAVA	Python, BASIC