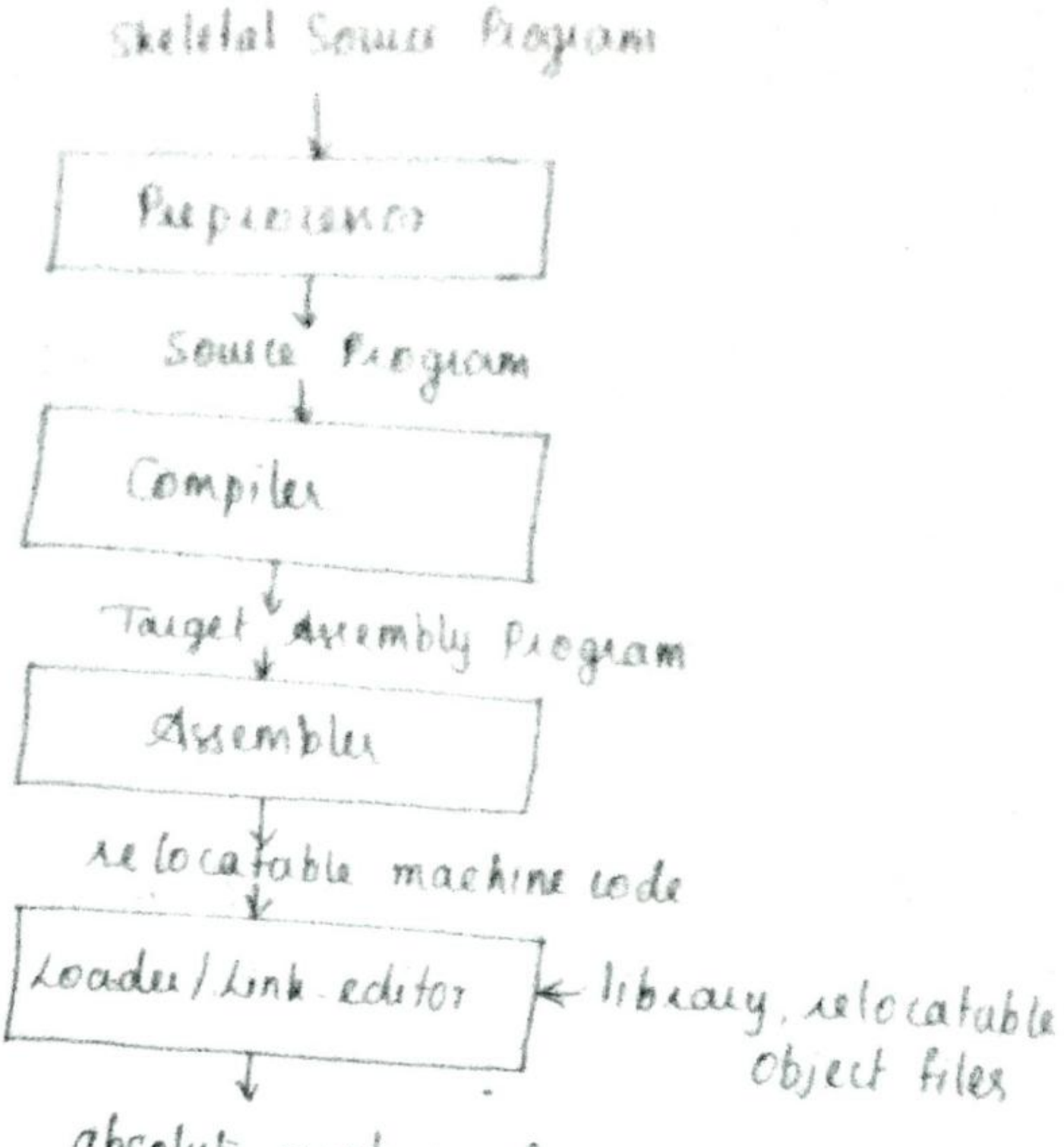
III

LANGUAGE PROCESSOR

(or)

COUSINS OF COMPILR

* In addition to compiler, several other programs may be required to create an executable target program.
* They are
  1. Preprocessor
  2. Assembler
  3. Link-Editor / Loader.



1 Preprocessor:

* The task of collecting all the modules of the source program is done by preprocessors
* Preprocessors produce input to compilers
* They perform the following functions

1. Macroprocessing:

A preprocessor may allow user to define macros that are short hands for longer constructs

2. File Inclusion:

A preprocessor includes the header files into the program text.

3 Rational Preprocessors

These processors augment the older languages with more modern flow-of-control statements and data structuring facilities

4 Language Extensions:

The processors attempt to add capabilities to the language by what amounts to built-in macros

For eg) Equel is a data base query language embedded in C. Statements beginning with ## are taken by the preprocessor to be data base access statements.

* Macro processor deal with two kinds of statemen
  1. Macro definition
  2 Macro use.

1 Macro definition:

* Definitions are normally indicated by some unique character or keyword like define or macro
* They consist of a name for the macro and a body forming its definition.
* Macro processors permits formal parameters in their definition.

2 Macro use:

* The macro use consists of naming the macro and supplying actual parameters. i.e) values for its formal parameter

Assembler

- Compilers produce assembly code that is passed to an assembler for further processing
- Some compilers perform the job of the assembler producing relocatable machine code that can be passed directly to the loader / link-editor.

* Assembly code is a mnemonic version of machine code in which names are used instead of binary codes.
* A typical sequence of assembly instruction might be

```
MOV  a   R1
ADD  #2  R1
MOV  R1  b
```

This code moves the contents of the address a into a register R1, then adds the constant 2 to it and finally stores the result in the location named by b.

* Thus it computs

$$b = a + 2$$

Two pass Assembly:

* The simplest form of assembler makes two passes over the input
* A pass consists of reading an input file once

* In the first pass all the identifiers that denote storage locations are found and stored in a symbol table

* For eg)

IDENTIFIER	ADDRESS
a	0
b	4

⇒ An assembler symbol table with identifiers of

```
MOV a R1
ADD #2 R1
MOV R1 b
```

* In the second pass, the assembler scan the input again. This time it translates each operation code into the sequence of bits representing that operation in machine language and it translates each identifier representing a location into the address given for that identifier in the symbol table.

* The output of the second pass is relocatable machine code. That it can be loaded starting at any location L in memory

  i.e) if L is added to all addresses in the code then all the references will be correct

* The following is machine code into which the assembly instructions might be translated

```
0001   01   00   00000000*
0011   01   10   00000010
0010   01   00   00000100*
```

* The first four bits are the instruction code

0001 – load
0010 – store
0011 – add

* By load and store we mean moves from memory into a register and vice versa respectively.
* The next two bits designate a register and 01 refers to register 1.
* The two bits after that represent a "tag" represent the addressing mode

00 → ordinary addressing mode
10 → Immediate addressing mode

* The last eight bits refers to a memory address

3. Loaders and Link-Editors:

* Loaders performs the two functions of of loading and link-editing.
* The process of loading consists of taking relocatable machine code, altering the relocatable addresses and placing the altered instructions and data in memory at proper locations
* The link-editor allows us to make a single program from several files of relocatable machine code

* If the files are to be used together, there may be some external references in which the code of one file refers to a location in another file

* This reference may be to a
  - → data location defined in one file and used in another
  - → it may be to the entry point of a procedur that appears in the code for one file and is called from another file

* Suppose that the address space containing the data is to be loaded starting at location L. The presence of * means that L must be added to the address of the instruction

* Thus if L = 00001111 i.e) 15 then a and b would be at locations 15 and 19 respectivel and the absolute machine code would appea as

```
0001  01  00  00001111
0011  01  10  00000010
0010  01  00  00010011
```

* If file loaded with this instruction refered to be then the reference would be replaced by A plus the offset by which the data locations in file were relocated.