LPCC

Unit I : **Introduction To Systems Programming And Assemblers**

- Introduction: **Need of System Software**, **Components of System Software**, Language Processing Activities, Fundamentals of Language Processing.
- Assemblers: Elements of Assembly Language Programming, A simple Assembly Scheme, Pass structure of Assemblers, Design of Two Pass Assembler.

BOOKS

Text Books :

- 1 D. M. Dhamdhere, Systems Programming and Operating Systems, Tata McGrawHill, ISBN 13:978-0-07-463579-7, Second Revised Edition
- 2 Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers Principles, Techniques and Tools, Addison Wesley, ISBN:981–235–885 - 4, Low Price Edition
- 3 John R. Levine, Tony Mason & Doug Brown, “Lex &Yacc”, O’Reilly
- Reference Books :
- 1 J. J. Donovan, Systems Programming, McGraw-Hill, ISBN 13:978-0-07-460482- 3, Indian Edition

What is Software ?

- Software is collection of many programs.
- Two types of software:
  - **System software**
    - These programs assist general use application programs
      - Ex:- Compiler , Assembler etc.
    - Collection of programs that bridge the gap between the level at which users wish to interact with the computer and the level at which computer is capable of operating
  - **Application software**
    - These are the software developed for the specific goal.
    - Application software usually used by end-user
    - It is concerned with the solution of some problem, using the computer as a tool, instead of how computers actually work

System software

- **System software consists of a variety of programs that support the operation of a computer (ex: text editor, compiler, debugger)**
- **One characteristic in which most system software differ from application software is machine dependency**
- **A system software programmer must know the target machine structure**

What is System Software?

- System Software consists of a variety of programs that support the operation of a computer - facilitating execution of programs and use of resources in a computer system
- The software makes it possible for the users to focus on an application or other problem to be solved, without needing to know the details of how the machine works internally
- It forms a software layer which act as intermediary between the user and the computer
- It is a collection of programs that
- Each program in the system software is called a **system program**

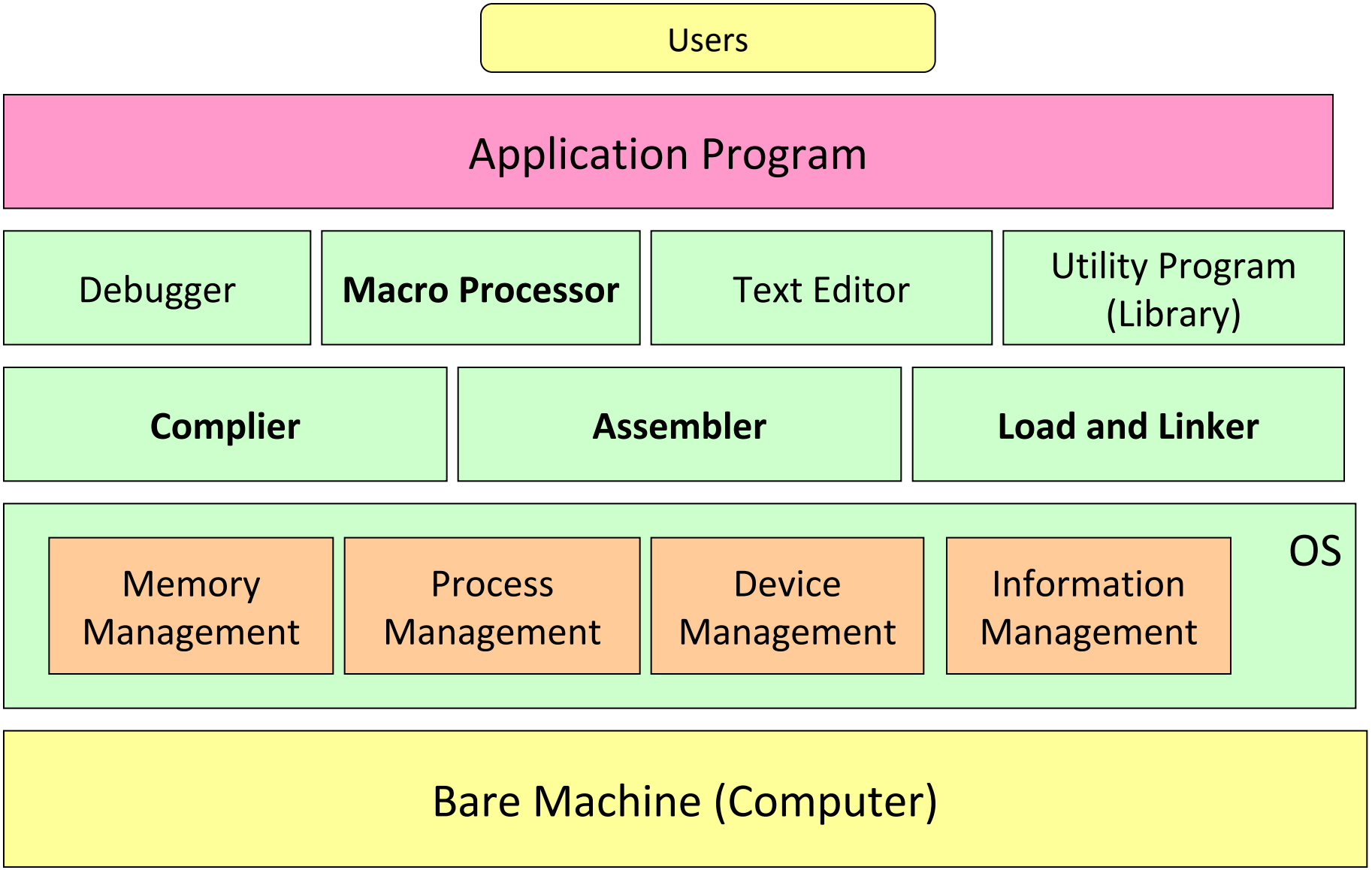
System Software

- **System Programming** is the collection of techniques used in the design of system programs

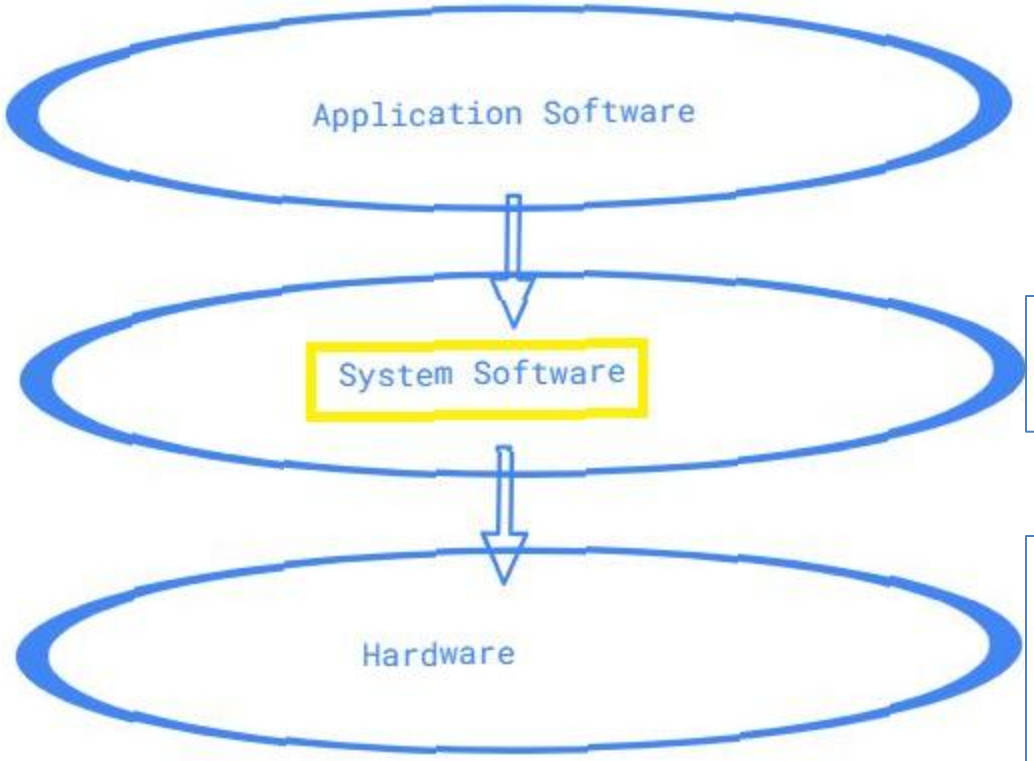
System Software and Machine Architecture

- System Software vs Application
  - One characteristic in which most system software differs from application software is machine dependency.
  - System programs are intended to support the operation and use of the computer itself, rather than any particular application.
- Examples of system software
  - Text editor, assembler, compiler, loader or linker, debugger, macro processors, operating system, database management systems, software engineering tools, …

System Software Concept



Need of System Software (*Language Processors*)



Operating System**, Language Processors,** Device Drivers

CPU, Hard disks, CD-ROMs, Keyboard, Mouse, Printer, Camera, Scanners, Fax machines, Video Adapters, Network Cards, Sound cards

System Software Types

- *Operating Systems : (Software that **manages** **hardware, software resources** and **provides common** **services** for computer programs)*

- **Language Processors : (Software that converts Source language into Target language)**

- *Device Drivers : (Software that **Interface** with specific **device** attached to computer)*

Self Study the following topics

- **Debugger**
- **Interpreter**
- **Editor**

Types of Languages

- Binary Code (Machine language) – Uses 0/1
- Assembly Languages - Uses Mnemonics
- High Level Languages (HLLs) - Uses English like words

Language Processors (LP)

- LP = System Software which converts Source code to Machine readable code
  - Source Code (Written in HLLs, Assembly Lang)
  - Object Code / Machine Code - Code in machine readable form

LP Types

- Assemblers – Assembly level code to Machine level code
- Interpreters – HLLs to machine level, but line by line
- Compilers - HLLs to machine level, but in one go

Goals of System Software

- User Convenience

Provide convenient method of using a computer system

- Efficient Use

Ensure efficient use of computer resources

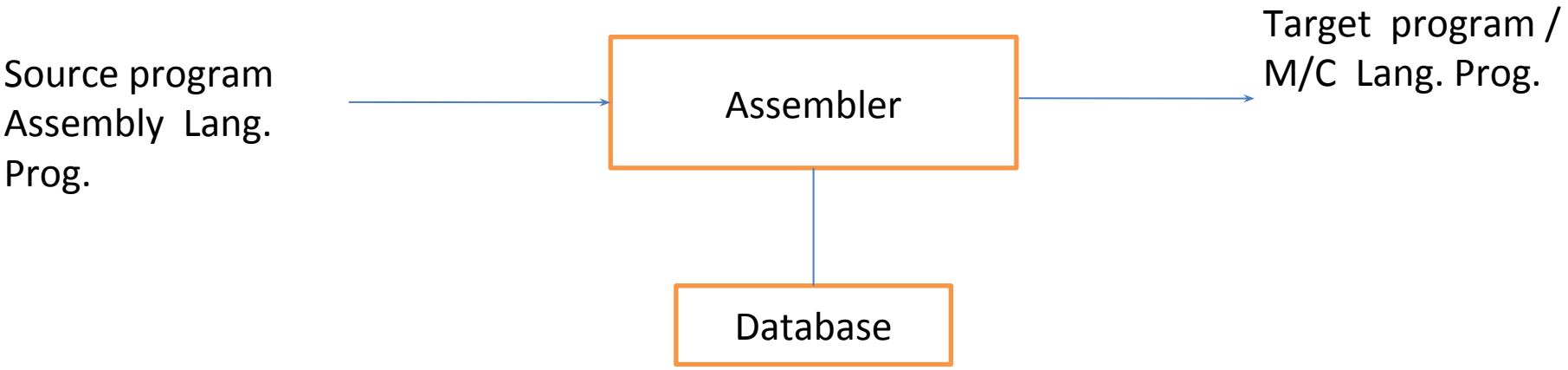
- interfernce

Fill in the Blanks

- Each program in the system software is called a _____
- _____facilitate execution of programs and use of resources in a computer system
- Goals of system software ___, _____

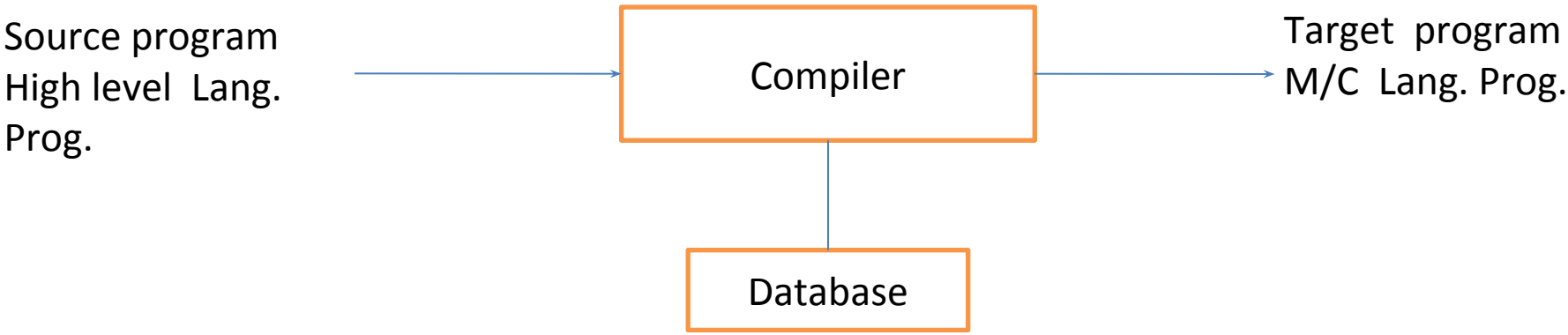
Assembler

- Assembler:-
- These are the system programs which translate the assembly language program into the machine language program.



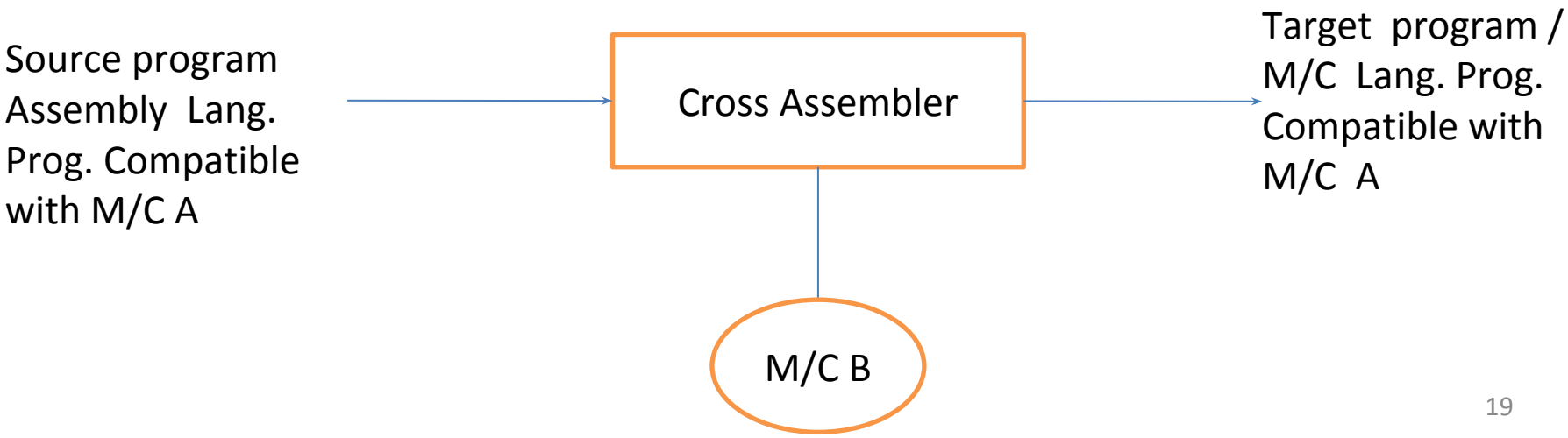
Complier

- Complier:-
- These are the system programs which translate the High level language program into the machine language program.



Cross Assembler:-

- Cross Assembler:-
- These are the system programs which will automatically translate the Assembly Language program compatible with M/C A, in to the machine language program compatible with M/C A, but the underlying M/C is M/C B



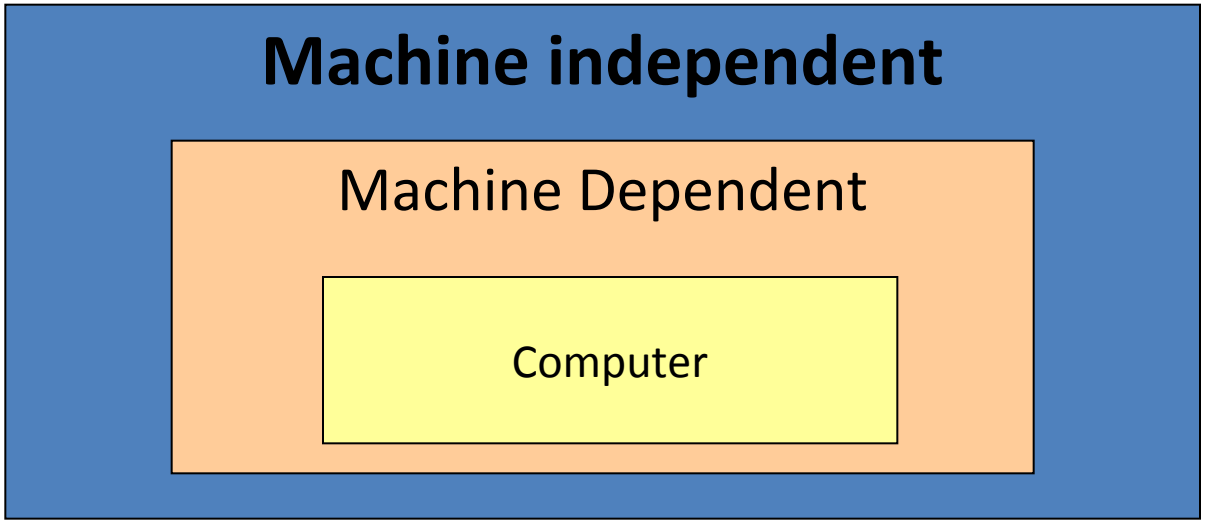
System Software

- Text editor
  - To create and modify the program
- Compiler and assembler
  - Translate these programs into machine language
- Loader or linker
  - The resulting machine program was loaded into memory and prepared for execution
- Debugger
  - To help detect errors in the program

System Software and Machine Architecture

- Machine dependent
  - Instruction Set, Instruction Format, Addressing Mode, Assembly language …
- Machine independent

General design logic/strategy, Two passes assembler…



System Software and Machine Architecture

- Machine dependent system software
  - System programs are to support the operation and use of the target computer.
  - The difference between different machine
    - Machine code
    - Instruction formats
    - Addressing mode
    - Registers
- Machine independent system software
  - General design and logic is basically the same:
    - Code optimization
    - General design and logic of an assembler

Programming Languages

- **Machine language:**
- **It is computer's native language having a sequence of zeroes and ones (binary). Different computers understand different sequences. Thus, hard for humans to understand: e.g.0101001...**

- **Assembly language:**
- **It uses mnemonics for machine language. In this each instruction**
- **is minimal but still hard for humans to understand:**
- **e.g. ADD AH, BL**

- **High-level languages:**
- **FORTRAN, Pascal, BASIC, C, C++, Java, etc.**
  Each instruction composed of many low-level instructions, closer to English. It is easier to read and understand:
  e.g. hypot = sqrt(opp*opp + adj * adj);

Intro to Assemblers

- Binary code is what a computer can run
- An assembler language is just a **thin syntactic layer** on top of its binary code

- Each architecture (computer) requires different binary code
- Which is why the assembler languages also differ

…contd..

- Assemblers (programs) convert Assembly language code to Machine Code

- All assemblers are somewhat similar in concept, but they **differ greatly** between **different architectures**.
- Code for ARM processor will not run on Intel c processor Learning one will certainly help in learning others, but it will still require lots of work.

Fill in the Blanks

- _____ is used to create and modify the program.
- _____is used to detect errors in the program.
- ____ , computer program which translate the program from high level language to machine language.
- _____, computer program which translate the program from assembly language to machine language.

We know…

- Source Program – Assembly Language
- Object Program - From assembler

  - Contains translated instructions and data values from the source program
- Executable Code - From Linker
- Loader - Loads the executable code to the specified memory locations and code gets executed.

Fill in the Blanks

- _____ is a program which load programs from a secondary to main memory so as to be executed
- _____ is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file

Machine and Assembly Languages

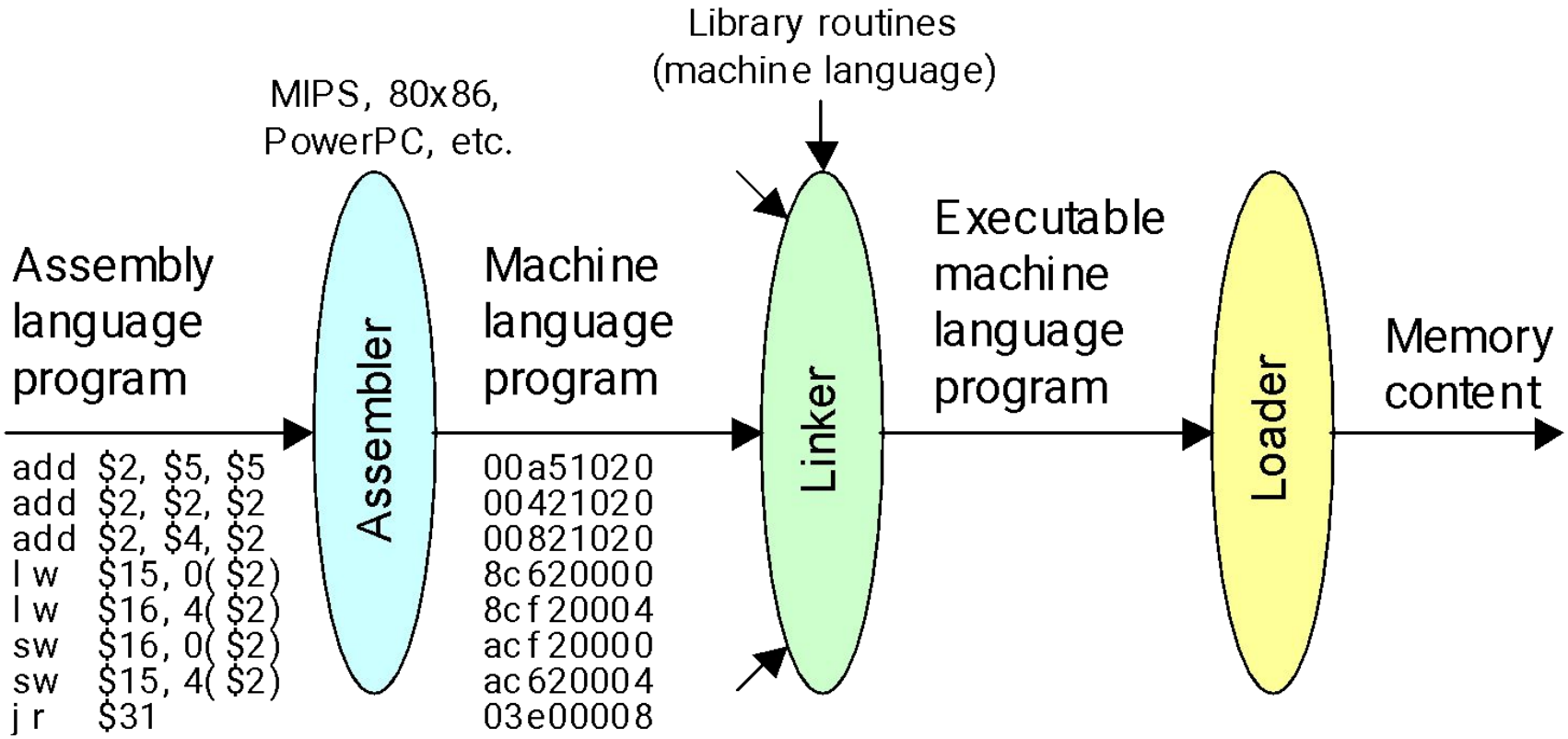
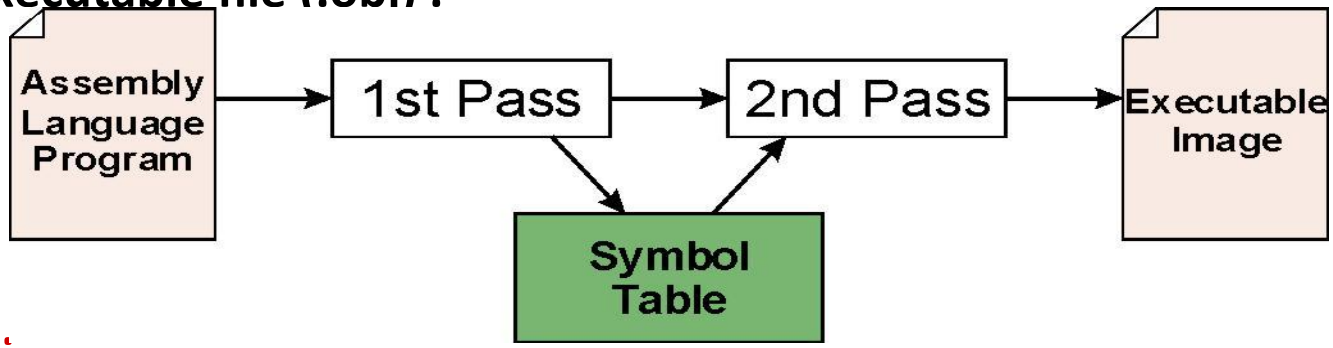


Figure : Steps in transforming an assembly language program to an executable program residing in memory.

Assembly Process

- **Convert assembly language file (.asm) into an executable file (.obj) .**



- **First Pass:**
  - **scan program file**
  - **find all labels and calculate the corresponding addresses; this is called the *symbol table***
- **Second Pass:**
  - **convert instructions to machine language, using information from symbol table**

```
;CLEAR SCREEN USING BIOS
CLR: MOV AX,0600H        ;SCROLL SCREEN
     MOV BH,30           ;COLOUR
     MOV CX,0000         ;FROM
     MOV DX,184FH        ;TO 24,79
     INT 10H             ;CALL BIOS;
;INPUTTING OF A STRING
KEY: MOV AH,0AH          ;INPUT REQUEST
     LEA DX,BUFFER       ;POINT TO BUFFER WHERE STRING STORED
     INT 21H             ;CALL DOS
     RET                 ;RETURN FROM SUBROUTINE TO MAIN PROGRAM;
; DISPLAY STRING TO SCREEN
SCR: MOV AH,09           ;DISPLAY REQUEST
     LEA DX,STRING       ;POINT TO STRING
     INT 21H             ;CALL DOS
     RET                 ;RETURN FROM THIS SUBROUTINE;
```

Assembly code

Assembler

```
0001010010110101010101010101010100010
1110110101010101010101110010100010110
0010100101010010111101011101011101010
1001010010110101010101010101010110110
0110100100110010111101011101010100010
0001000101011101010101000101010111010
1010100101010010101110101110101110101
0001010010110101010101010101010100010
```

Object code

D. Assembler

1. Assembly language programming
2. Simple assembly scheme,
3. Pass structure of assembler,
4. Design of two pass assembler

Introduction to Assembly Language

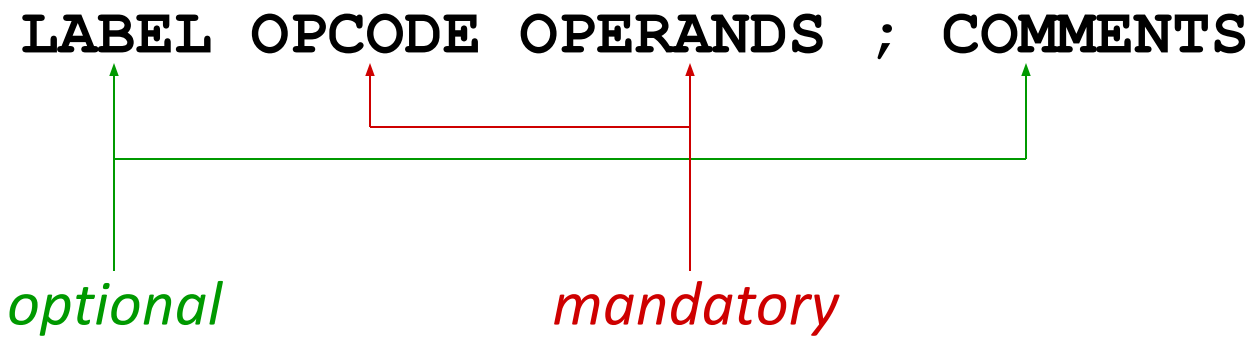
Elements

- Mnemonic operation code (Mnemonic opcode)
- Symbolic operands
- Data declarations

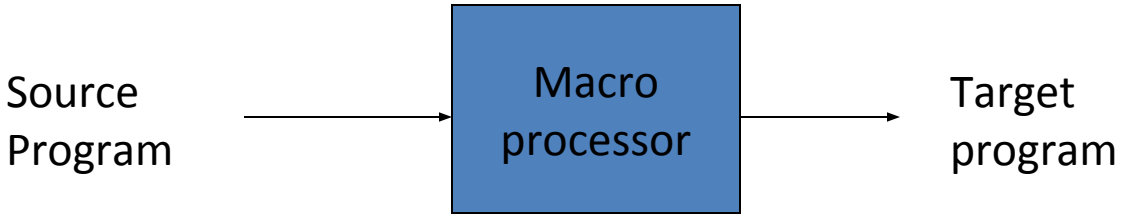
Assembly language is a machine dependent, low level programming language which is specific to a certain computer system (or a family of computer system)

Assembly Language Syntax

- **Each line of a program is one of the following:**
  - **an instruction**
  - **an assembler directive (or pseudo-op)**
  - **a comment**
- **Whitespace (between symbols) and case are ignored.**
- **Comments (beginning with ";") are also ignored.**
- **An instruction has the following format:**



- Macro : a abbreviation for a group of instructions.
- Acts as a preprocessor to assembler
- Source prg : assembly program with macros
- Target prg : assembly program without macros



Data structures

- Macro Name Table (MNT)
- Macro Definition Table (MDT)

Macro Name Table (MNT)

Name of macro	No. of parameters	Starting Index (row)	End Index (row)
SAMPLE1	0	1	2
SAMPLE2	0	3	4
SAMPLE3	0	5	6

Macro Definition Table (MDT)

1	**LOAD A**
2	**ADD B**
3	**LOAD X**
4	**SUB Y**
5	**LOAD P**
6	**DIV Q**

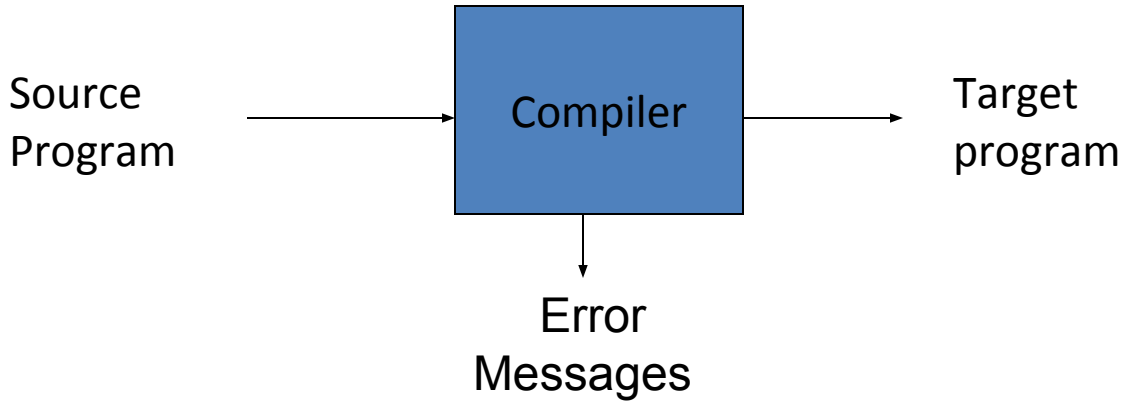
Modified MNT & MDT

Name of macro	No. of parameters	Starting Index
SAMPLE1	0	1
SAMPLE2	0	4
SAMPLE3	0	7

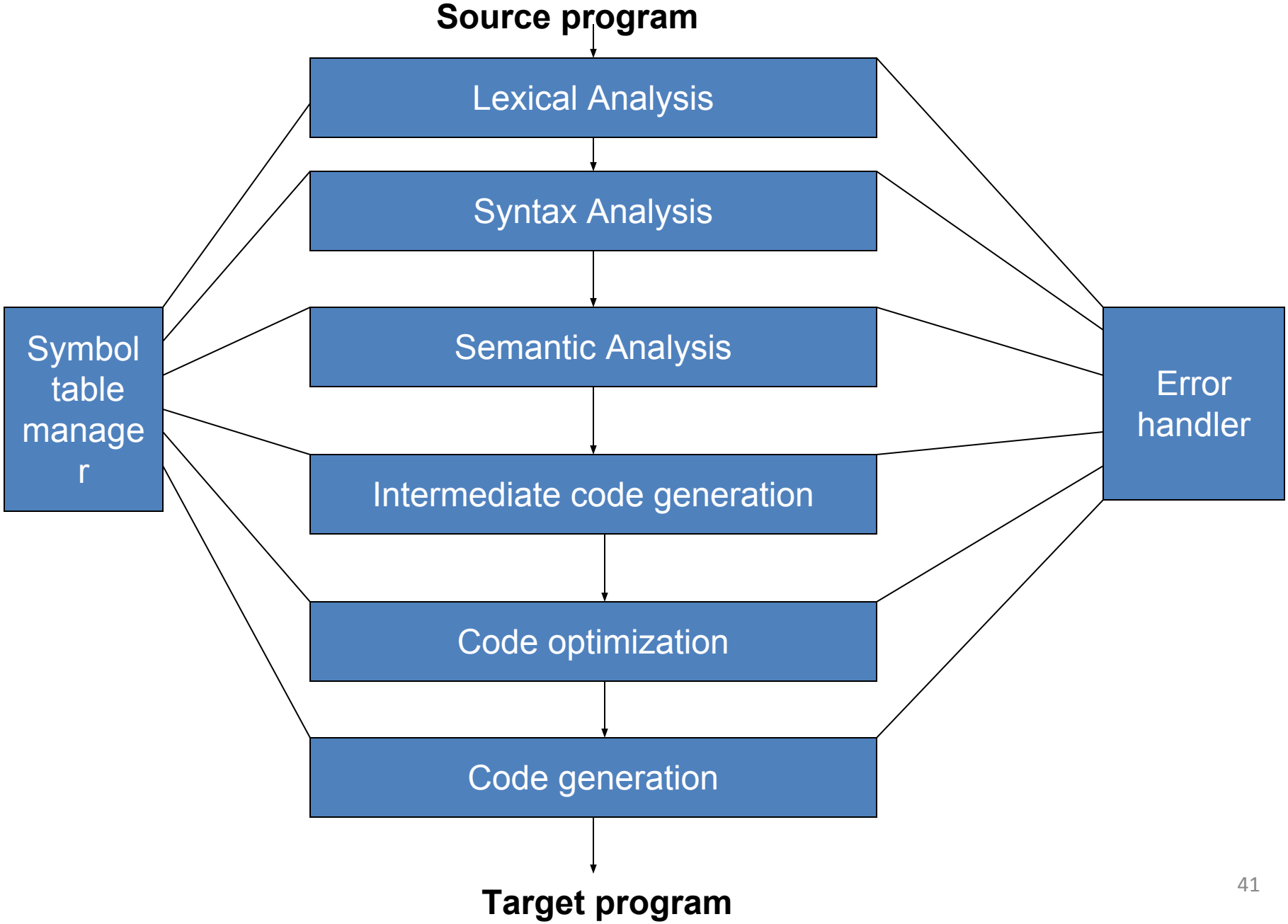
1	**LOAD A**
2	**ADD B**
3	MEND
4	**LOAD X**
5	**SUB Y**
6	MEND
7	**LOAD P**
8	**DIV Q**
9	MEND

Introduction-Compiler

- Translator for conversion of HLL to machine language
- Source prg : High Level Language program (e.g. sample.c)
- Target prg : machine language program (e.g. sample.obj)



The phases of a compiler



Lexical Analyzer:

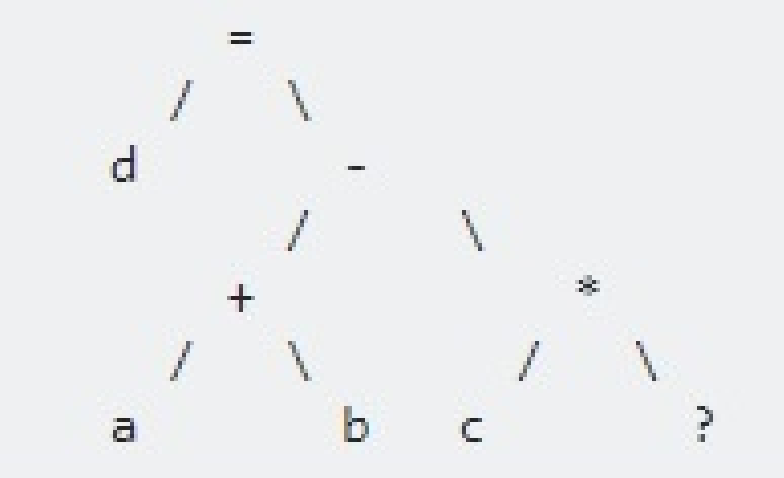
- It combines characters in the source file, to form a "TOKEN". A token is a set of characters that does not have 'space', 'tab' and 'new line'.
- It is also called "TOKENIZER"
- It also removes the comments, generates symbol table and relocation table entries

Syntax Analyzer:

- This unit check for the syntax in the code. For ex:
- {
- int a; int b; int c; int d;
- d = a + b - c * ;
- }
- This unit checks this internally by generating the parser tree

Parse Tree

Therefore this unit is also called PARSER



Semantic Analyzer:

This unit checks the meaning in the statements.

For ex

{

- Int i;
- Int *p;
- P=i;

}

The above code generates the error “Assignment of incompatible type”

Code Optimization

- This unit optimizes the code in following forms:

I) Dead code elimination

II) Sub code elimination

III) Loop optimization

Dead code elimination:

```
{
Int a= 10;
If(a>5)
{.....}
Else
{
}
```

- compiler knows the value of 'a' at compile time, therefore it also knows that the if condition is always true. Hence it eliminates the else part in the code

- **Sub code elimination**:

```
Int a, b, c;
Int x,y;
{
int a, b, c; int x, y;
/* … */
x = a + b;
y = a + b + c;
 /* … */
}
```

```
{
int a, b, c;
 int x, y;
/* ... */
x = a + b;
 y = x + c; // a + b is replaced by x
/* ... */
 }
```

- **Loop optimization:**

For ex:

```
{ int a;
for (i = 0; i < 1000; i++ )
{ /* … */
a = 10;
 /* … */
} }
```

if 'a' is local and not used in the loop, then it can be optimized as follows

- { int a;
- a = 10;
- for (i = 0; i < 1000; i++)
- {
- /* ... */
- } }

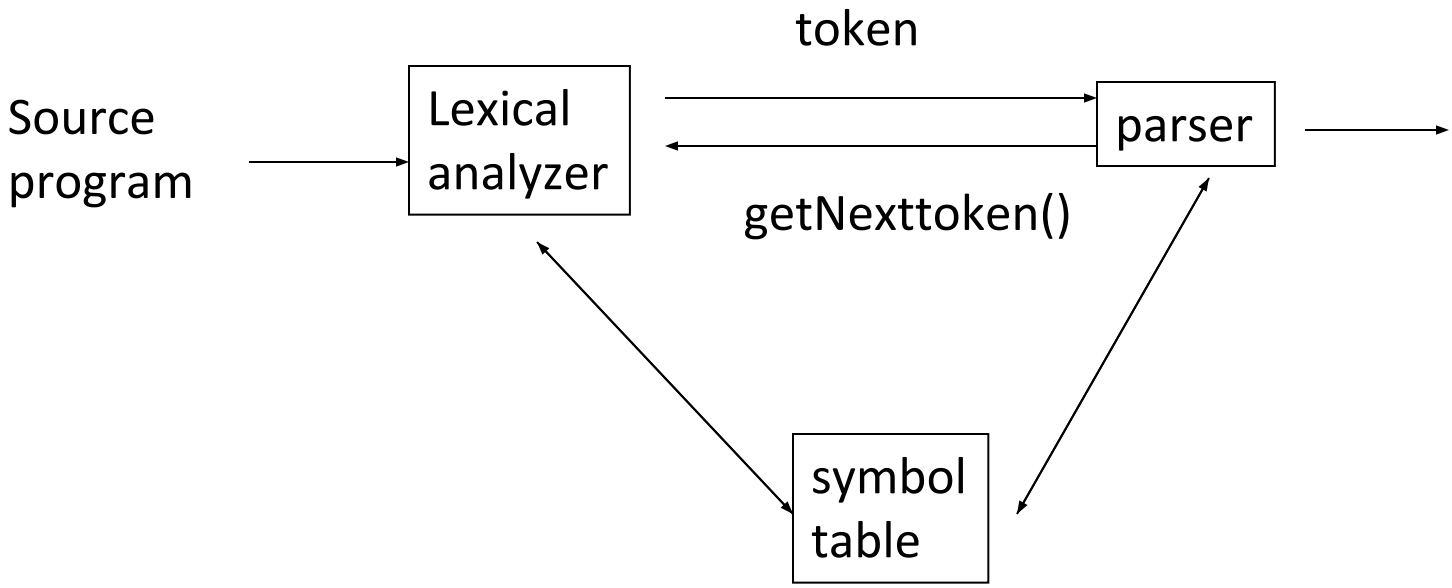
Code Generation

- It takes the optimized representation of the intermediate code and maps it to the target machine language
- It translates the intermediate code into a sequence of (generally) re-locatable machine code
- Sequence of instructions of machine code performs the task as the intermediate code would do

Symbol Table

- It is a data-structure maintained throughout all the phases of a compiler
- All the identifier's names along with their types are stored here
- The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it
- The symbol table is also used for scope management

Interaction of Lexical analyzer with parser



- Some terminology:
  - ***Token***: a group of characters having a collective meaning. A *lexeme* is a particular instant of a token.
    - E.g. token: identifier, lexeme: pi, etc.
  - ***pattern***: the rule describing how a token can be formed.
    - E.g: identifier: ([a-z]|[A-Z]) ([a-z]|[A-Z]|[0-9])*

- Lexical analyzer does not have to be an individual phase.
- But having a separate phase simplifies the design and improves the efficiency and portability.

Specification of tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
    - Letter_(letter_ | digit)*
- Each regular expression is a pattern specifying the form of strings

Regular expressions

- ε is a regular expression, L(ε) = {ε}
- If a is a symbol in Σ then a is a regular expression, L(a) = {a}
- (r) | (s) is a regular expression denoting the language L(r) $\cup$ L(s)
- (r)(s) is a regular expression denoting the language L(r)L(s)
- (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denting L(r)

Automatic construction of lexical analyzer using LEX

Lex

- generates C code for the lexical analyzer (scanner)
- Token patterns specified by regular expressions

Lex

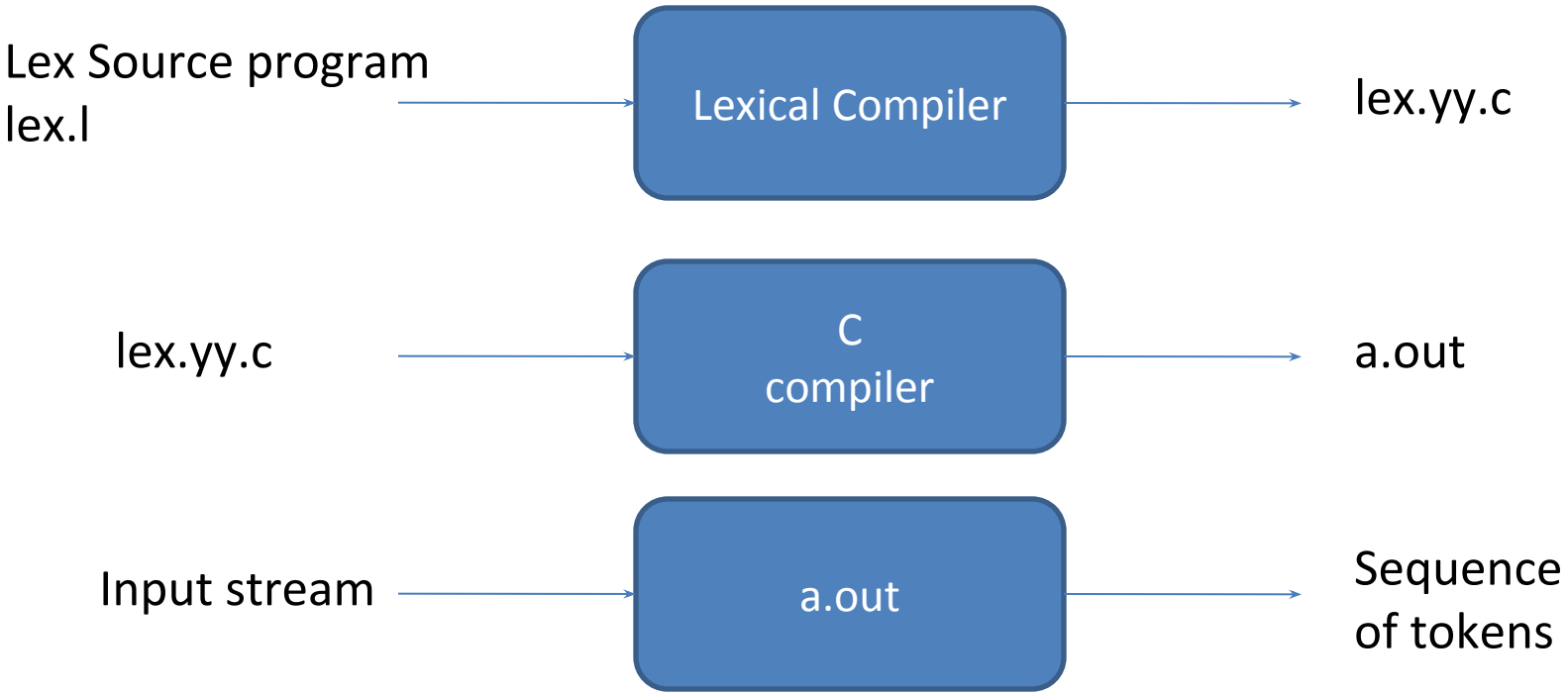
lex.yy.c

- Lex Source file → Lex compiler→

C compiler→ a.out

Input stream → Scanner→ Sequence of tokens

Lexical Analyzer Generator - Lex



Lex program structure

… definitions …

%%

… rules …

%%

… subroutines …

Three parts to Lex

1. Declarations

It has global C and Lex declaration

- Regular expression definitions of tokens

```
%{ This is a sample Lex program written by....%}
digit   -->   [0-9]
number  -- > {digit} +
```

2. Transition Rules

pattern

- Regular Expression +Action when matched

```
{number} { printf("The number is %s\n", yytext); }
junk     { printf("Junk is not a valid input!\n"); }
quit     { return 0; }
```

3. Auxilliary Procedures

- Written into the C program…..
- int main() is required

%% separates the three parts

LEX program

```
%{/*recognition of Verb*/%}
%%
[/t  ]+  ;
is I
    am I
    are I
    were I
    was I
    be  {printf("%s: is a verb\nn, yytext); }
[a-zA-Z]+  {printf("%s:is not a verb\n", yytext);}

    %%
Main()
{
yylex();
}
```

Assignment 3

- **Write a program to implement a lexical analyzer for parts of speech.**

INPUT:

Ram ran quickly

OUTPUT:

Noun: Ram
Verb: ran
Adverb: quickly

```
%{
/* Part of speech program */
%}
%%
Ram|
SAM       { printf(“%s, is a Noun”,yytext); }
ran |
walk     { printf(“%s, is a verb”,yytext); }
quickly |
Slowly      { printf(“%s, is a adverb”,yytext); }
%%
Main()
{
        yylex();
}
```

Example

```
% {
% }

delim       [ \t\n]
ws      {delim}+
letter      [A-Za-z]
digit       [0-9]
id      {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

%%

{ws}        { /* no action and no return */ }
if      {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = install_id(); return(ID);}
{number}    {yylval = install_num();return(NUMBER);}
…
%%
```

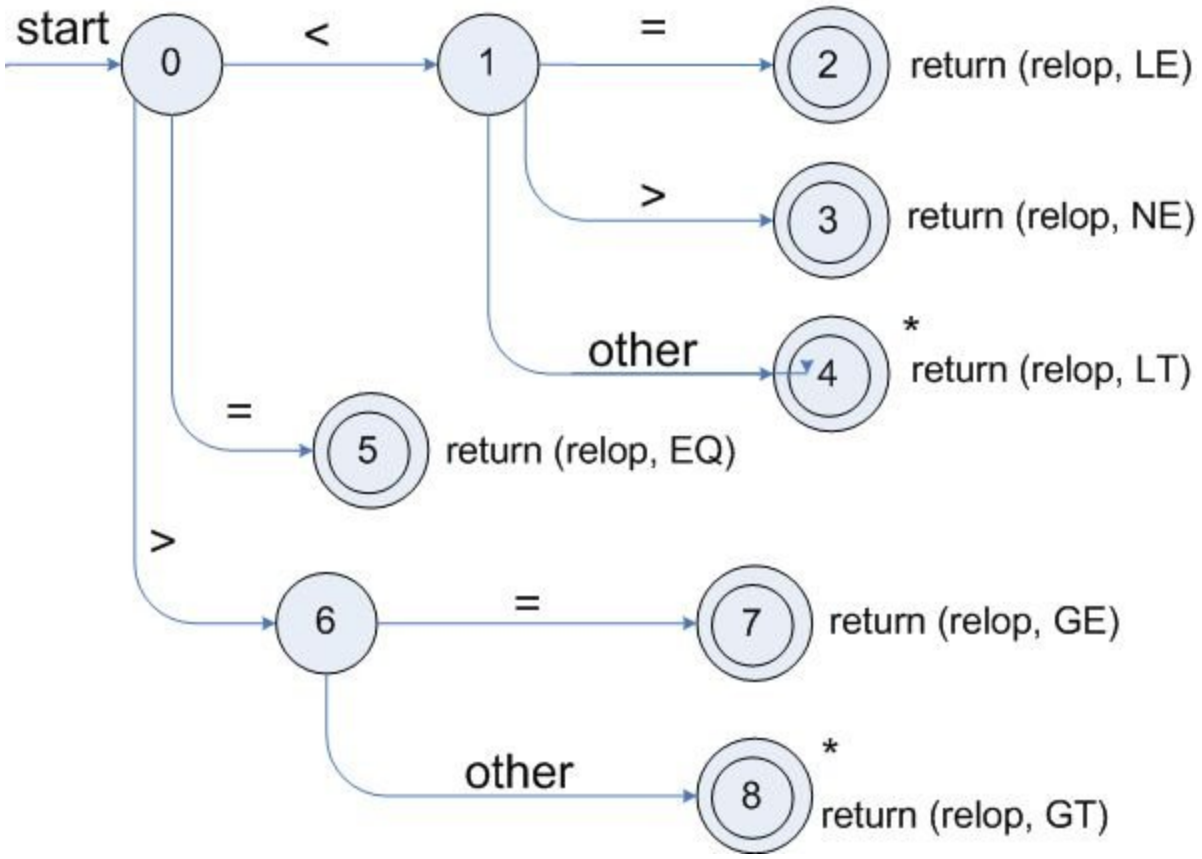
- **Available variables**
  - **`Yylval –value associated with token`**
  - **`yytext`** (null terminated string)
  - **`yyleng`** (length of the matching string)
  - **`yyin`** : the file handle
    - **`yyin = fopen(args[0], "r")`**
- **Available functions**
  - **`yylex()`** (the primary function generated)-starts the analysis
  - **`input()`** - Returns the next character from the input
  - **`int main(int argc, char *argv[])`**
    - Calls yylex to perform the lexical analysis
    - **`Int yywrap(void) wrapup, return 1 if done, 0 if not done`**

Regular Expression in Lex

- A- matches A
- (abc) –matches abc
- [abc]- matches a, b or c
- [0-9] - matches any digit
- [0-9]+ - matches any integer

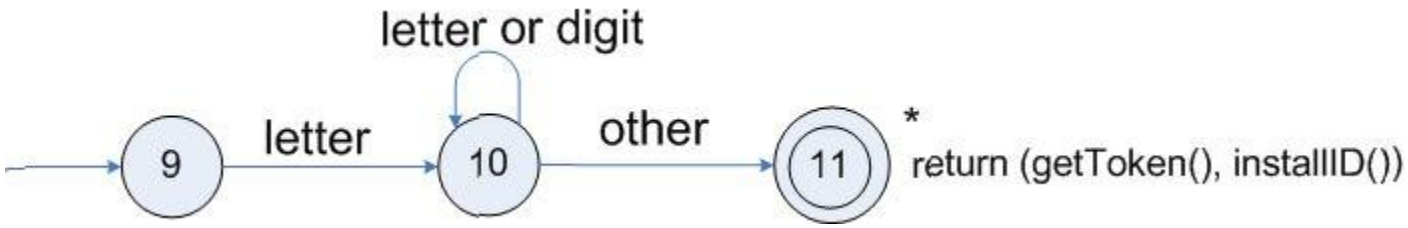
Transition diagrams

- Transition diagram for relop



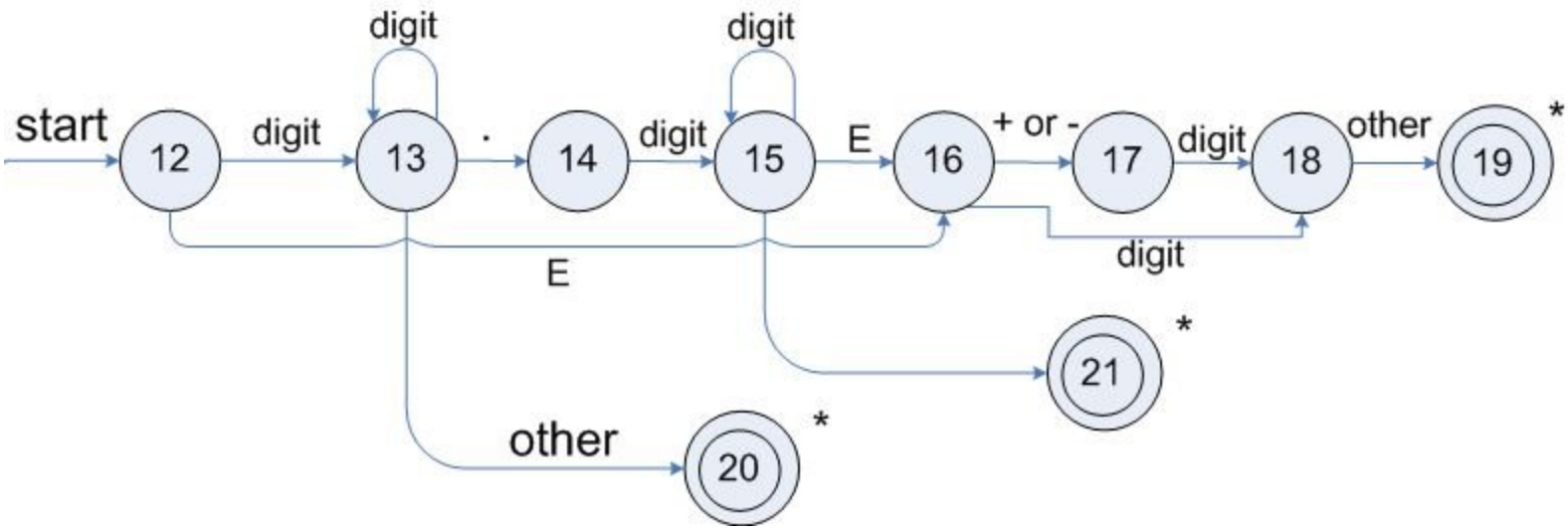
Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers



Transition diagrams (cont.)

- Transition diagram for unsigned numbers



Assembly language statements

- Imperative statements
  - Indicate an action to be performed during the execution.
- Declarative statements
  - [label] DS <constant>
  - [label] DC <value>
    - A DS 1
    - B DS 200
    - ONE DC ‘1’
- Assembler directives
  - START <constant>
  - END [<operand spec >]
  -

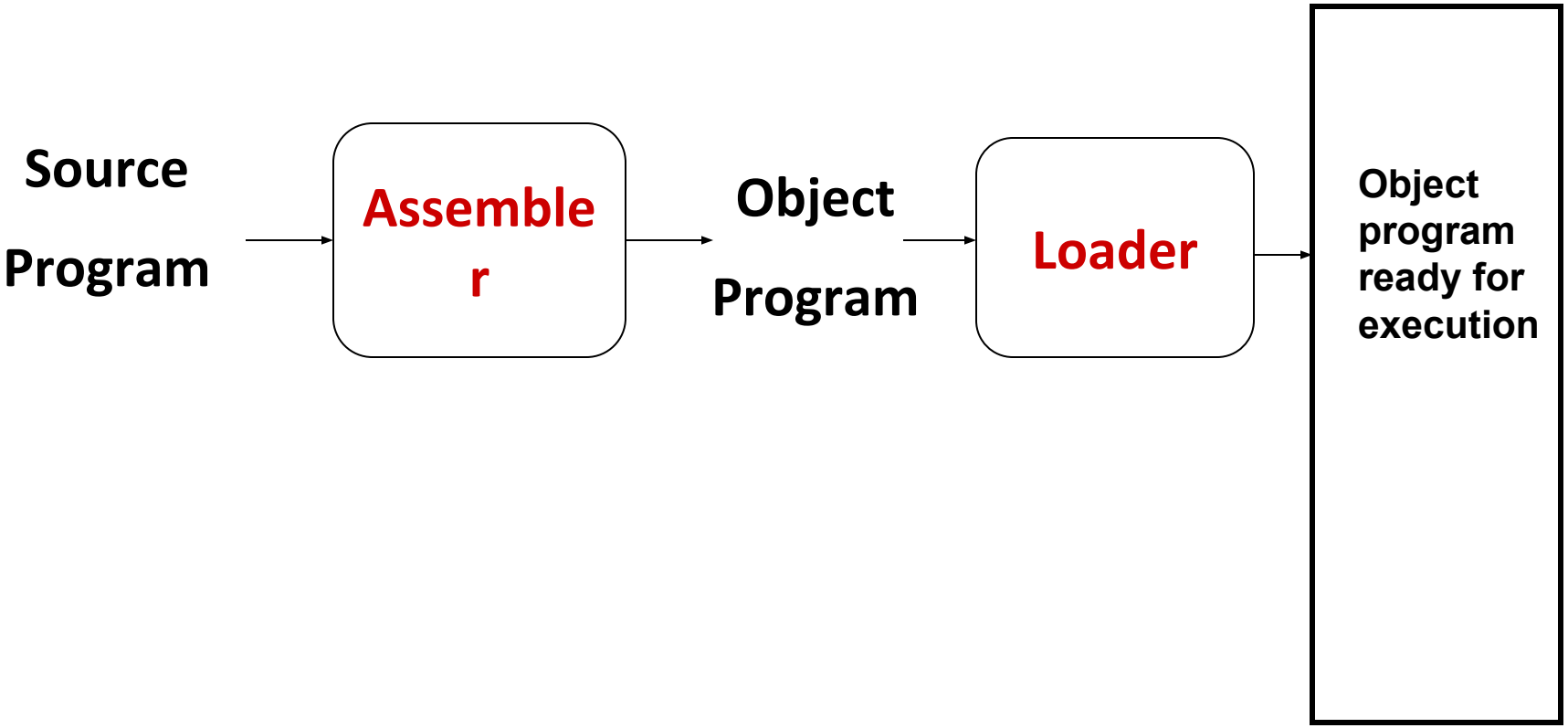
A. LOADERS AND LINKERS

- A loader is a program which load programs from a secondary to main memory so as to be executed.
- Loader is a program which accepts the object program decks, prepare these programs for execution by the computer, and initiates the execution.

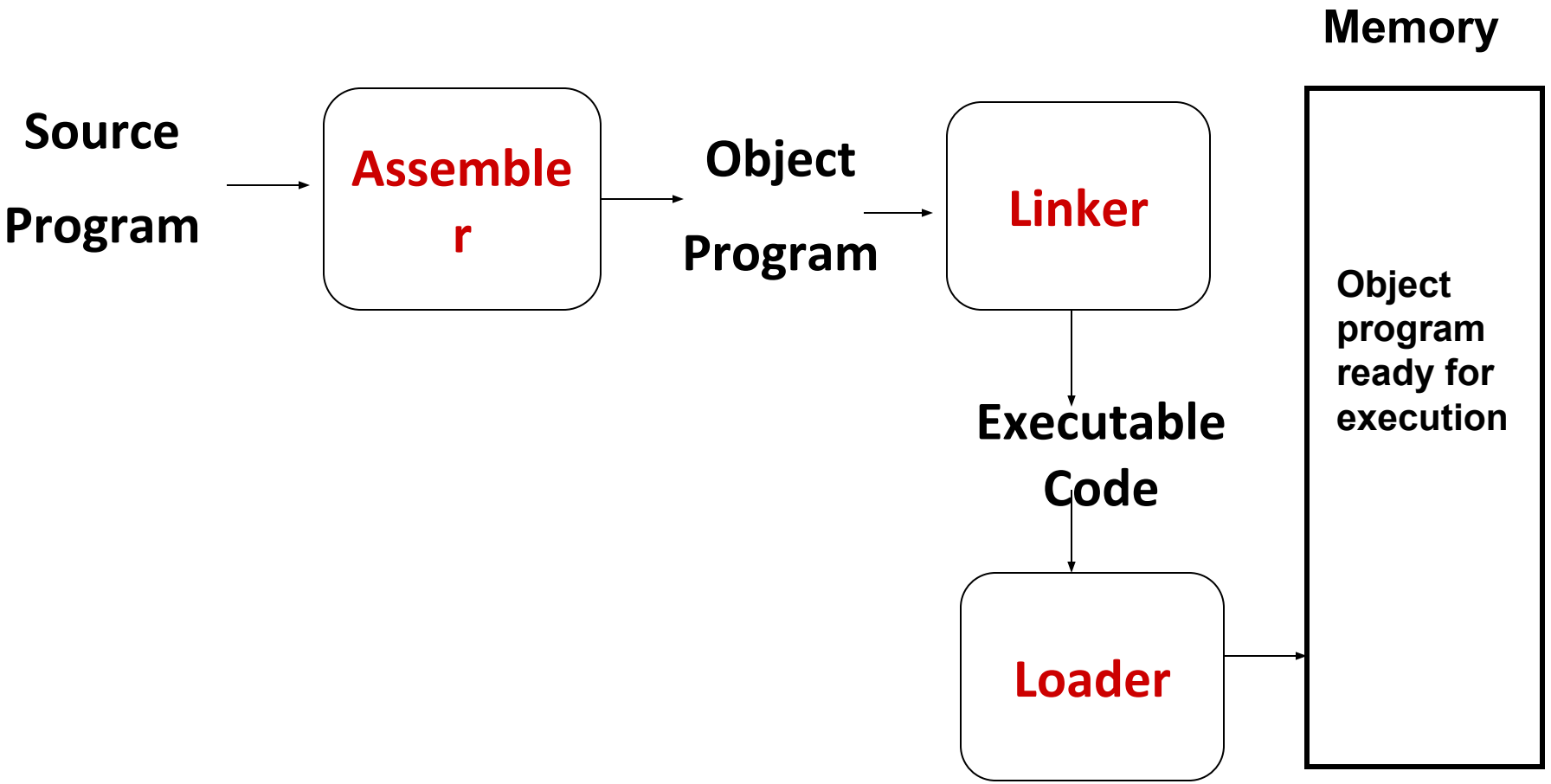
Linker is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file

- Many programming languages allow you to write different pieces of code, called *modules,* separately.
- This simplifies the programming task because you can break a large program into small, more manageable pieces.
- Eventually, though, you need to put all the modules together.
- This is the job of the linker
- a linker also replaces symbolic addresses with real addresses

Role of a Loader



Role of a Loader and Linker



SELF STUDY

Advantage, disadvantage, comparisons between

Compilers, Interpreters, Debugger, Editors,

Thank You