# Course: PARADIGMS AND COMPUTER PROGRAMMING FUNDAMENTALS (PCPF)

**Course Instructor**

Mrinmoyee Mukherjee B.E (Electronics), M.E (EXTC), PhD (Pursuing)

Assistant Professor

Department of Information Technology

St. Francis Institute of Technology

email: mrinmoyeemukherjee@sfit.ac.in

Academic Year: 2023-24 (Odd Semester)

# OUTLINE OF SYLLABUS

| Module | Contents |
|---|---|
| 1 | Introduction to programming paradigms and core language design issues |
| 2 | Imperative Paradigm: Data abstraction in object orientation |
| 3 | Declarative programming paradigm: Functional programming |
| 4 | Declarative programming paradigm: Logic programming |
| 5 | Alternative paradigm: Concurrency |
| 6 | Alternative paradigm: Scripting Languages |

# BOOKS

| Sr. No | Title of Book |
|---|---|
| 1 | Scott M L Programming Language Pragmatics, 3rd Edition, Morgan Kaufmann Publisher, 2009 |
| 2 | Programming Languages: Concepts and constructs, 2nd Edition, Pearson, 1996 |

# TEACHING AND EXAMINATION

- **Teaching Scheme**
  - Theory: 3 hours/week
  - Lab : 2 hour/ week
- **Examination Scheme**
  - IAT1 & IAT2 : 20 marks each
  - End sem exam : 80 marks
  - Oral Exam/Practical : 25 marks
  - Term-work : 25 marks (Related to Lab)
    - Journal : 15 marks
    - Attendance : 05 marks
    - Assignment : 05 marks (Comprises of Assignment Tests and tutorials)

# COURSE OUTCOMES

| | | |
|---|---|---|
| **C305.1** | To understand and compare different programming paradigms | **Chapter 1** |
| **C305.2** | To understand object oriented constructs and use them in programming design | **Chapter 2** |
| **C305.3** | To understand concepts of declarative programming paradigms through functional and logic programming | **Chapter 3** |
| **C305.4** | Design and develop programs based on declarative programming paradigms using functional and logic programming | **Chapter 4** |
| **C305.5** | Understand the role of concurrency in parallel and distributed programming | **Chapter 5** |
| **C305.6** | Understand different application domains for use of scripting language | **Chapter 6** |

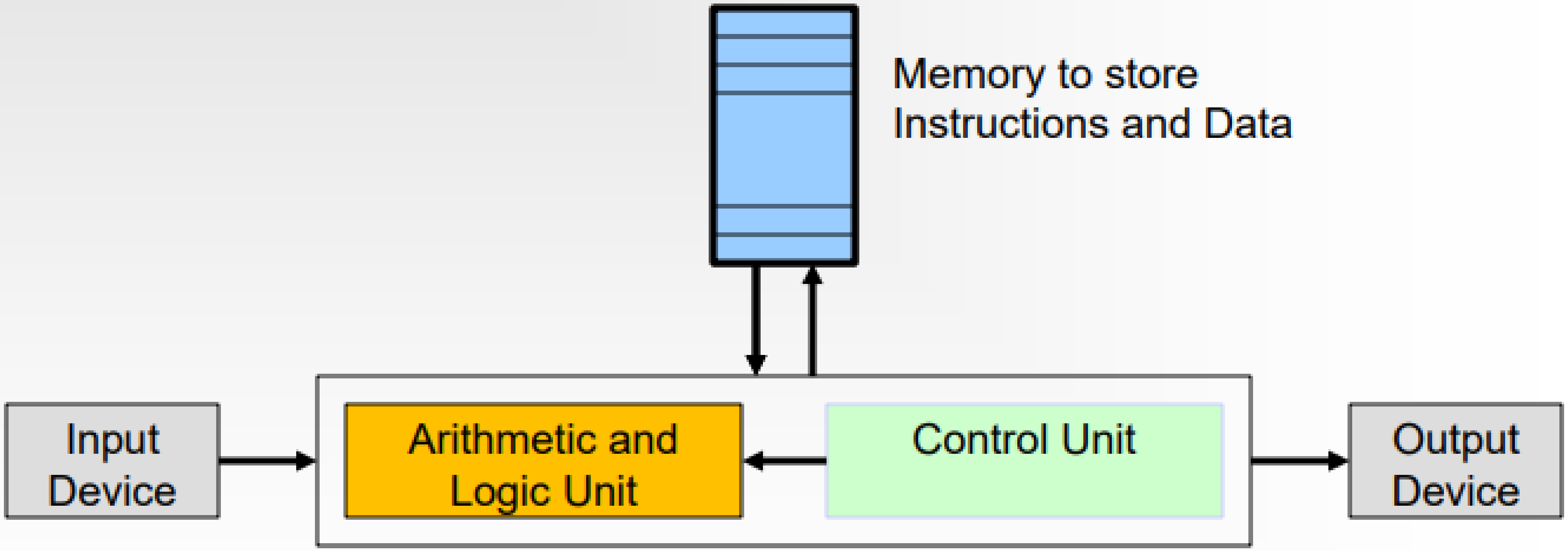
# TENTATIVE PORTION DISTRIBUTION

| | | |
|---|---|---|
| **C305.1** | **Chapter 0, Chapter 1** | **Assignment Test-1 (10 Marks)** |
| **C305.2** | **Chapter 2** | **Internal Assessment Test-1 (20 Marks)** |
| **C305.3** | **Chapter 3** | |
| **C305.4** | **Chapter 4** | **Internal Assessment Test-2 (20 Marks)** |
| **C305.5** | **Chapter 5** | |
| **C305.6** | **Chapter 6** | **Assignment Test-2 (10 Marks)** |

# Unit 0: Prerequisites (Compilation and Interpretation)

# Why Use Programming Language?

- We use von Neumann machines
- We need a programming language to talk to computer.



What happens if we do not use programming languages…….

# Low Level Machine Language

Ex. Light bulb is controlled by a processor running a program in main memory.

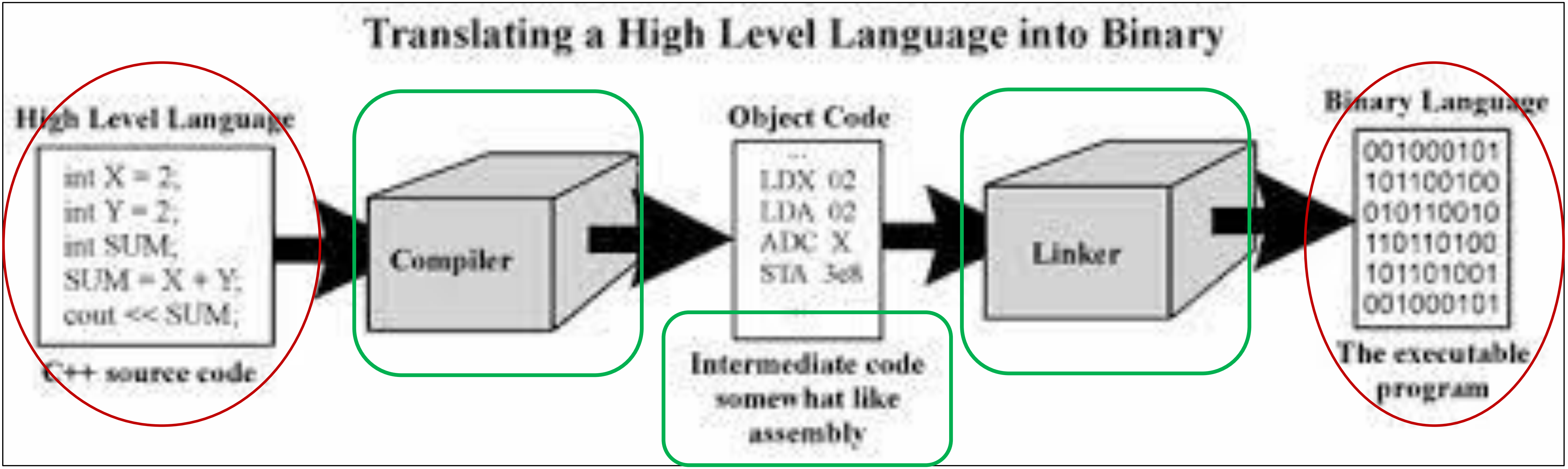
| Machine Instruction | Machine Operation |
|---|---|
| 00000000 | Stop Program |
| 00000001 | Turn bulb fully on |
| 00000010 | Turn bulb fully off |
| 00000100 | Dim bulb by 10% |
| 00001000 | Brighten bulb by 10% |
| 00010000 | If bulb is fully on, skip over next instruction |
| 00100000 | If bulb is fully off, skip over next instruction |
| 01000000 | Go to start of program (address 0) |

- Machine language or machine code is the basic low level programming language of computers.
- It is a sequence of binary digits (bits, 0s and 1s) comprehended only by computers but not by humans.
- This machine code directly controls a processor, causing it to add, compare, move data from one place to another, etc.

# High Level Programming Language

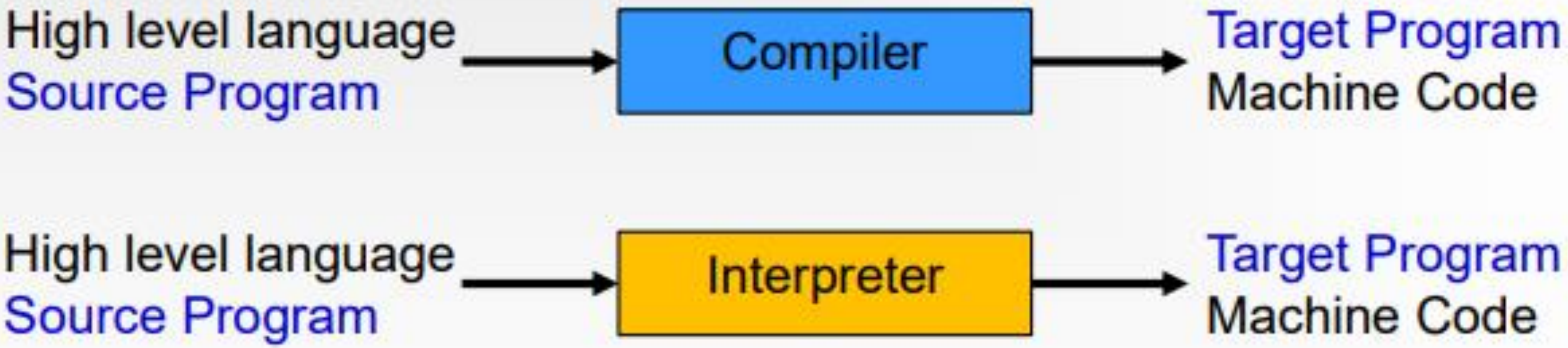
- Machine language is processor specific and difficult to understand by humans, hence high level, English like programming languages such as C, C++, java were invented.
- High level programming language is not understood by machine, hence we need to use another program like compiler or interpreter which will convert high level program code into machine code.

Mrinmoyee Mukherjee


Translating a High Level Language into Binary
High Level Language
int X = 2;
int Y = 2;
int SUM;
SUM = X + Y;
cout << SUM;
C++ source code
Compiler
Object Code
LDX 02
LDA 02
ADC X
STA 3e8
Intermediate code somewhat like assembly
Linker
Binary Language
001000101
101100100
010110010
110110100
101101001
001000101
The executable program

# Compiler and Interpreter

- To write larger programs which are independent of computer architecture, English like high level programming languages are invented.
- Computer does not understand these languages, a special system program called compiler or interpreter is required to convert programs in these languages, to machine code



# Compiler

- The compiler is itself a machine language program that is understandable by OS.
- The compiler translates the high-level source program into an equivalent target program (machine language) and then goes away.
- In later time, the user tells the operating system to run the target program.
- Advantages: fast, gives better performance
- Disadvantages: complex and not flexible.
- Fortran and C use compilers.

# Interpreter

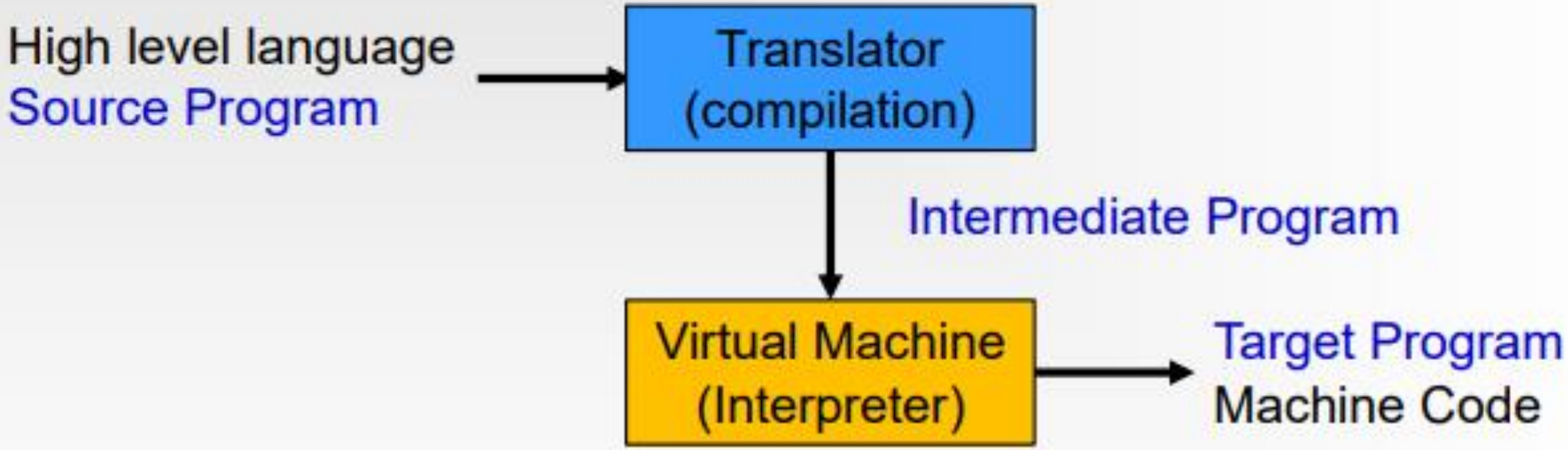
- Unlike a compiler, an interpreter stays around for the execution of the application.
- The interpreter implements a virtual machine whose “machine language” is the high-level programming language.
- The interpreter reads statements in that language more or less one at a time, executing them as it goes along.
- Simple, Greater flexibility, better diagnosis, but slow.
- Lisp, Prolog and other scripting languages like Perl, Tcl, Python, and Ruby etc. use interpreter.

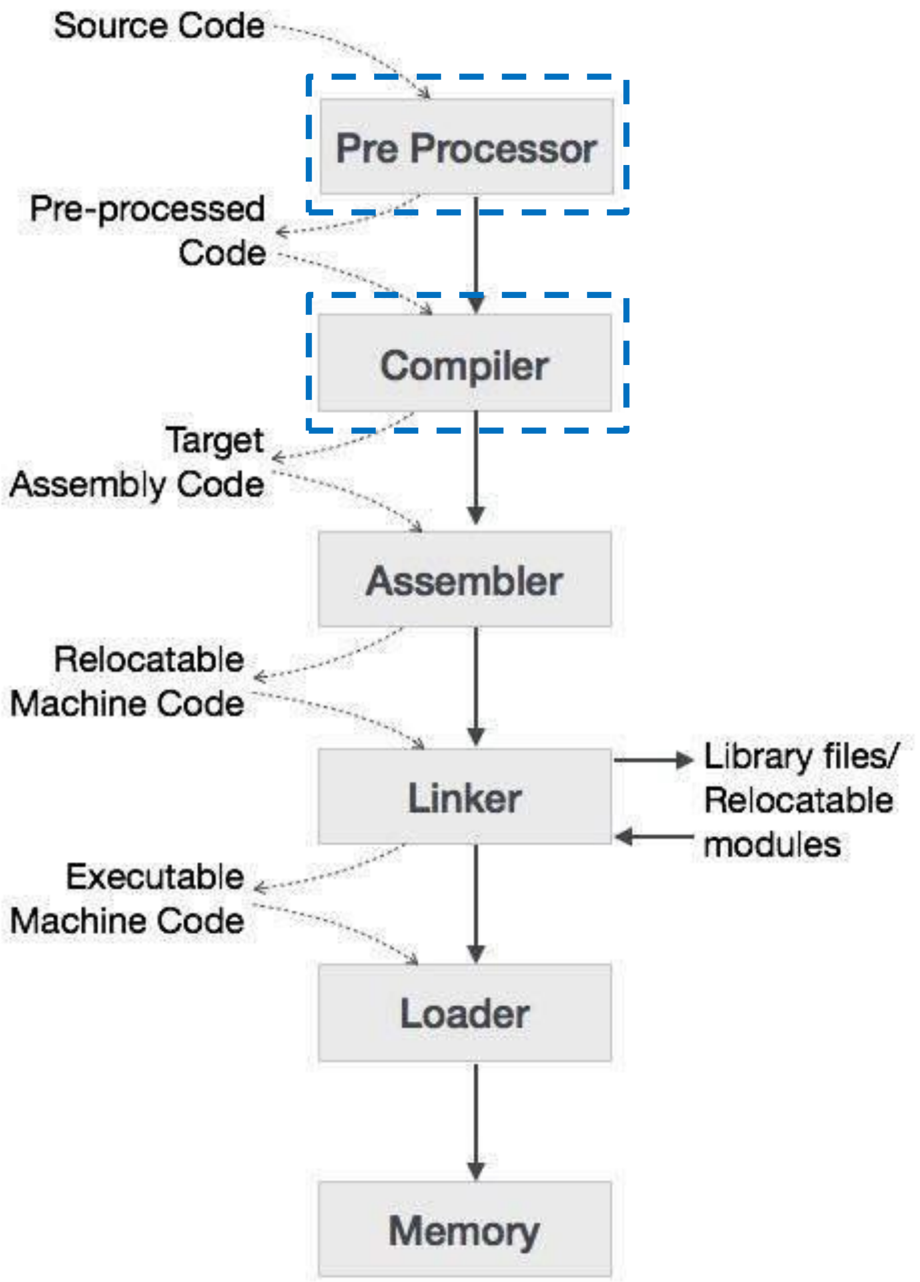
High level language / Source Program → Interpreter → Target Program / Machine Code

# Compiler and Interpreter

- Some languages use both compiler and Interpreter for converting source program to first intermediate program and then to Target Program.

Ex: Python,





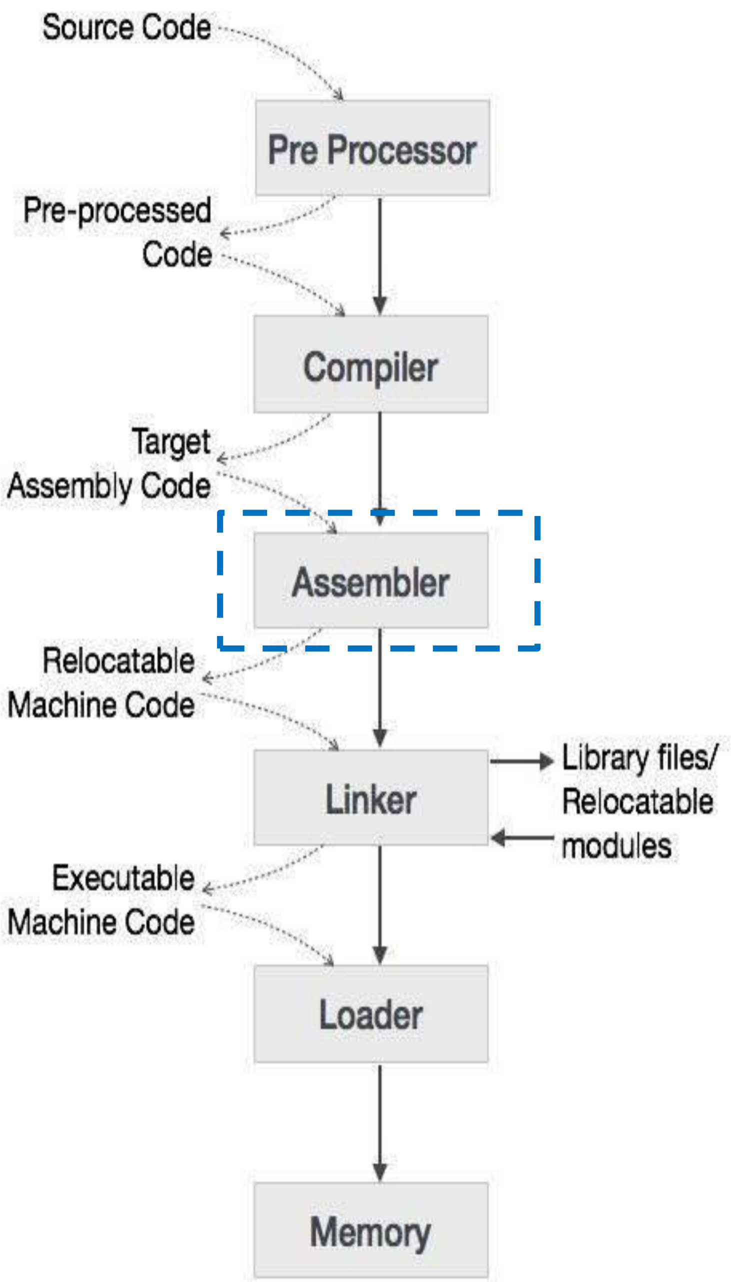

Pre-processor: Will remove all lines that include pre-processor directives and create a new file. This is called file inclusion

<#include stdio.h> [In C]
Using namespace [In C++]
#ifdef constant [In Python]

The compiler will take the Pure HLL and convert to Assembly Language Program (ALP)

***Does every language have its own compiler ????***

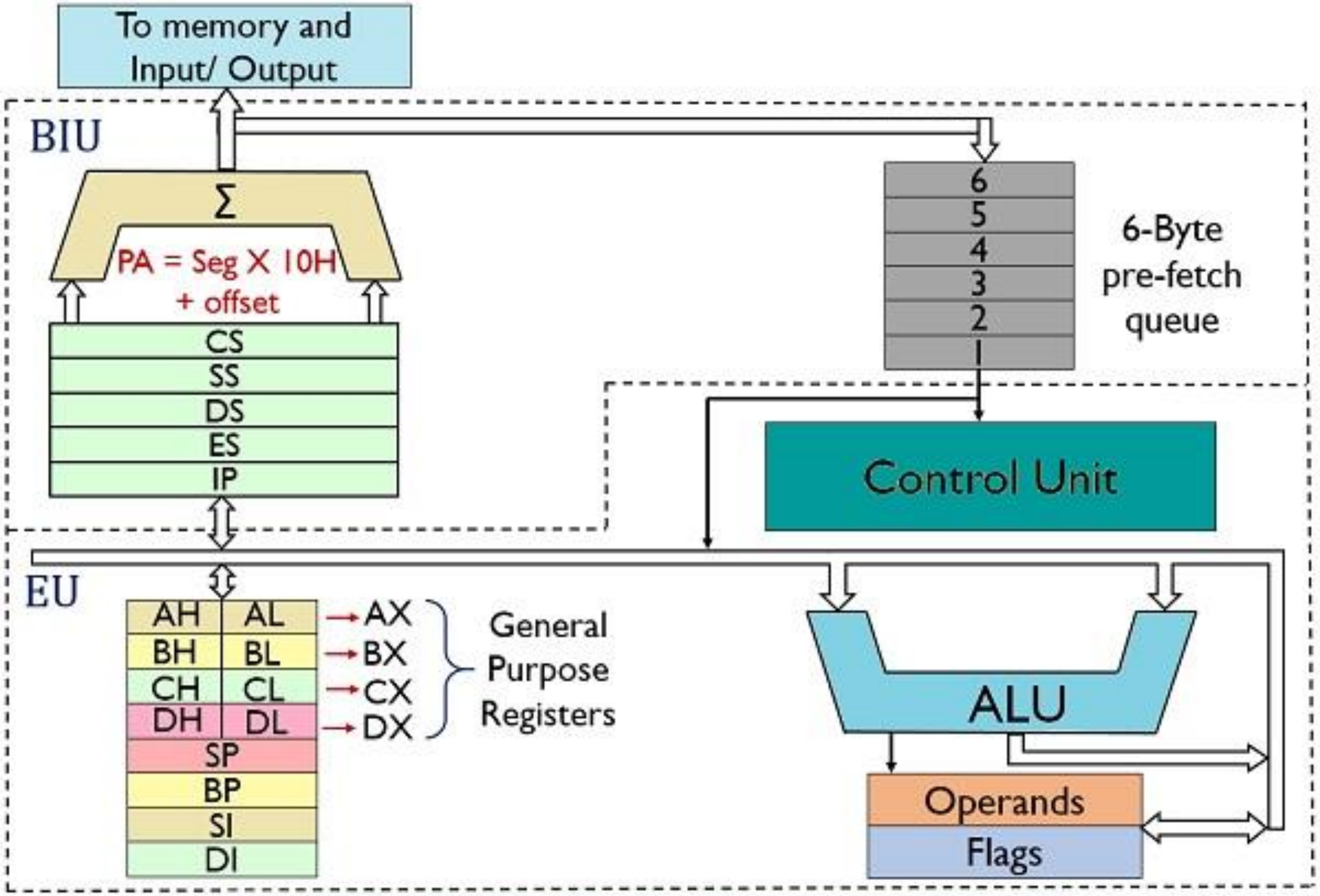
C compilers, C++ compilers, Java Compilers, Python compilers



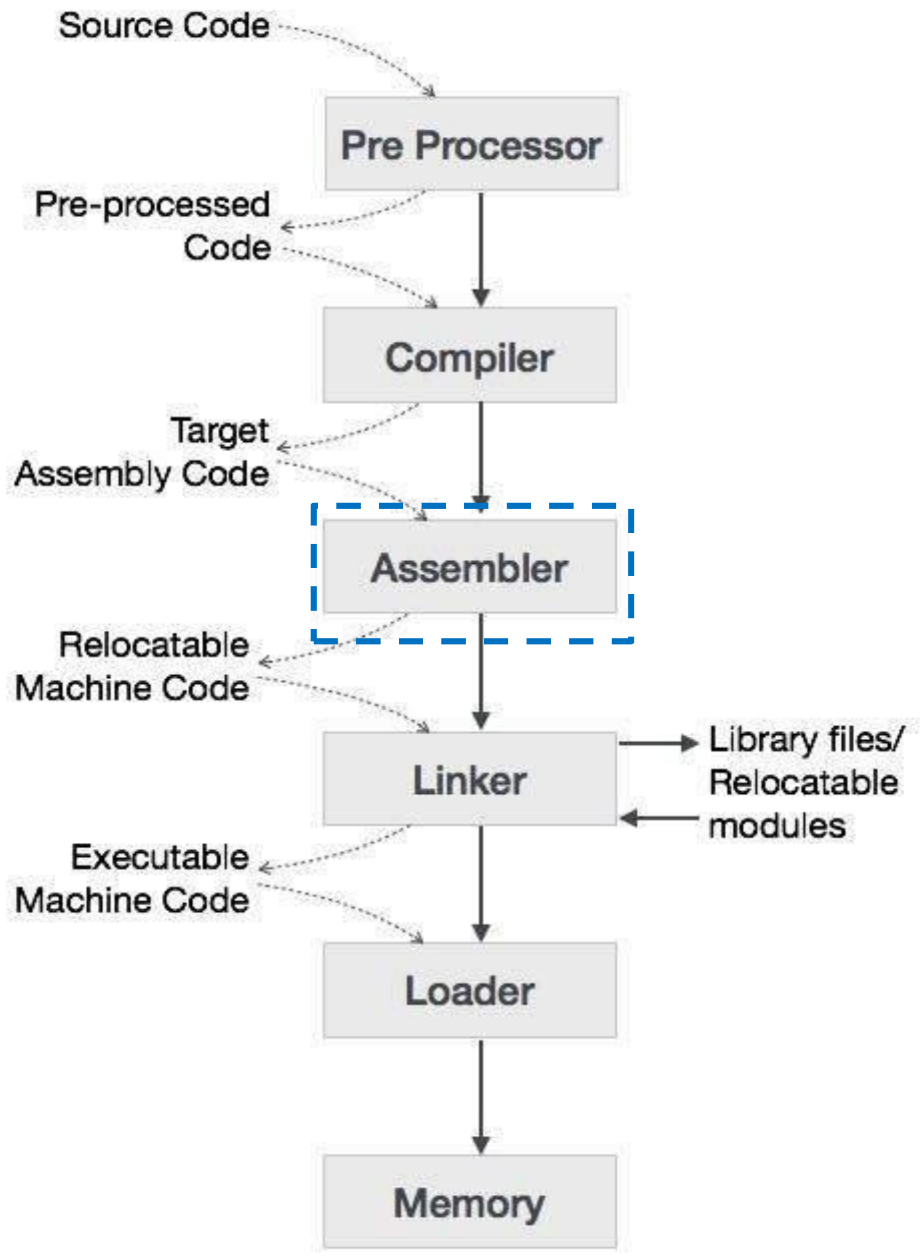

An Assembler is a system software which converts an ALP to its equivalent object code.

$$x = a + b * c$$

Eg: mul R1, R2    a….>R0, b….>R1, c…>R2
add R0,R2
mov R2,X

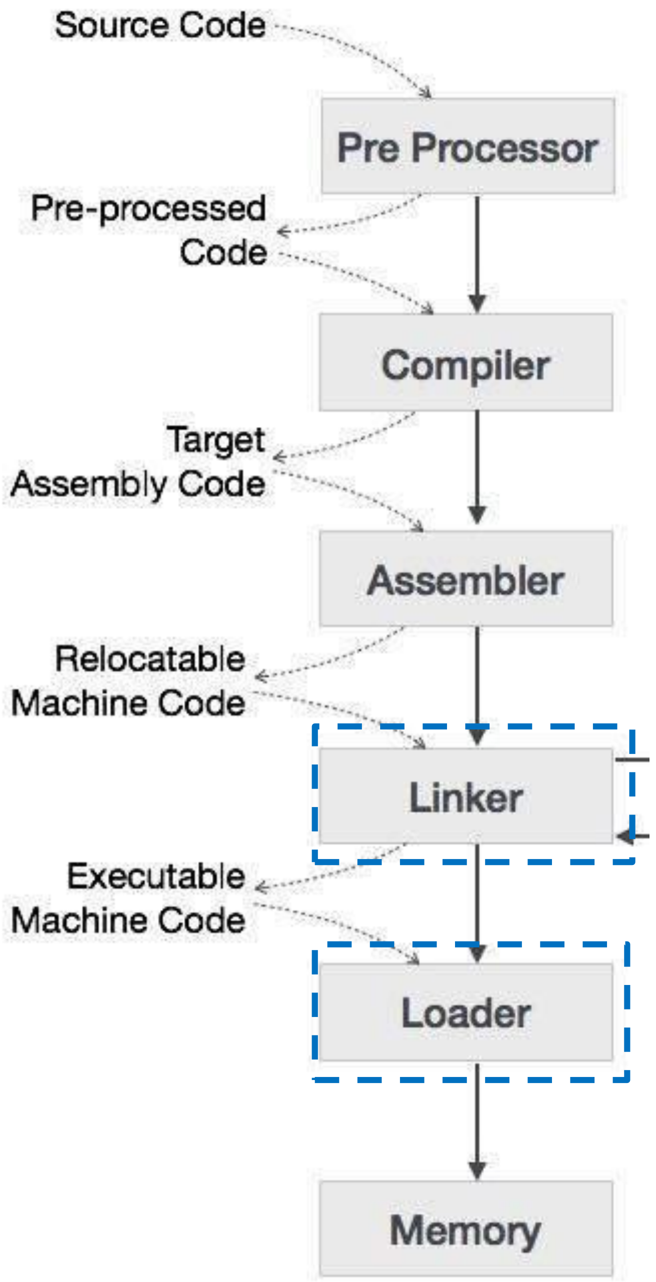


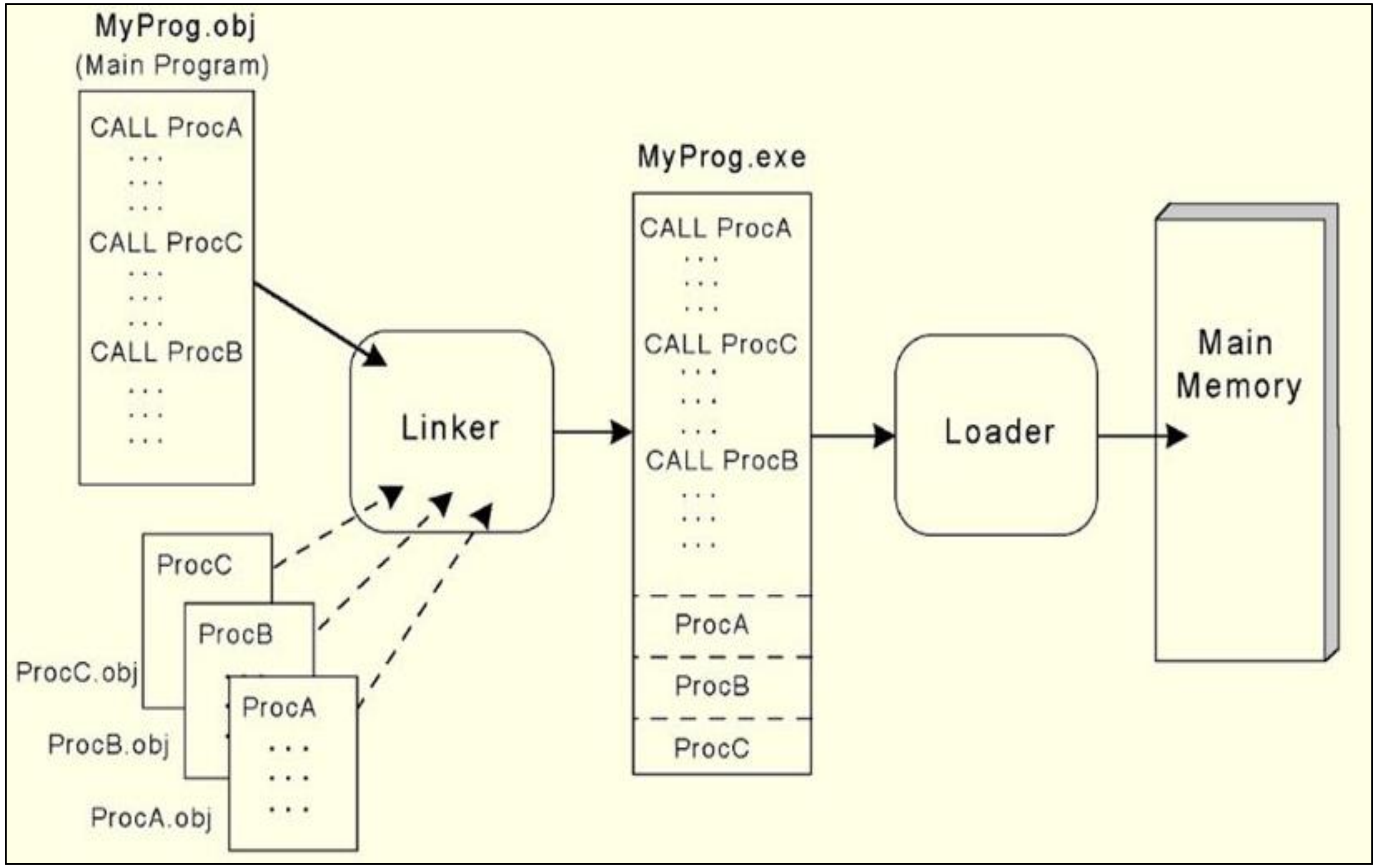

Block Diagram of 8086 Microprocessor




- The assembler converts the ALP to machine code (in terms of 0s and 1s. The machine codes are of two types –*The relocatable machine code and absolute machine code*
- The output of most assemblers is a stream of relocatable binary code
- In relocatable codes the operand addresses are relative to where the operating system chooses to load the program
- When relocatable code is loaded for execution, special registers provide the base addressing
- Addresses specified within the program are interpreted as offset from base addresses

-



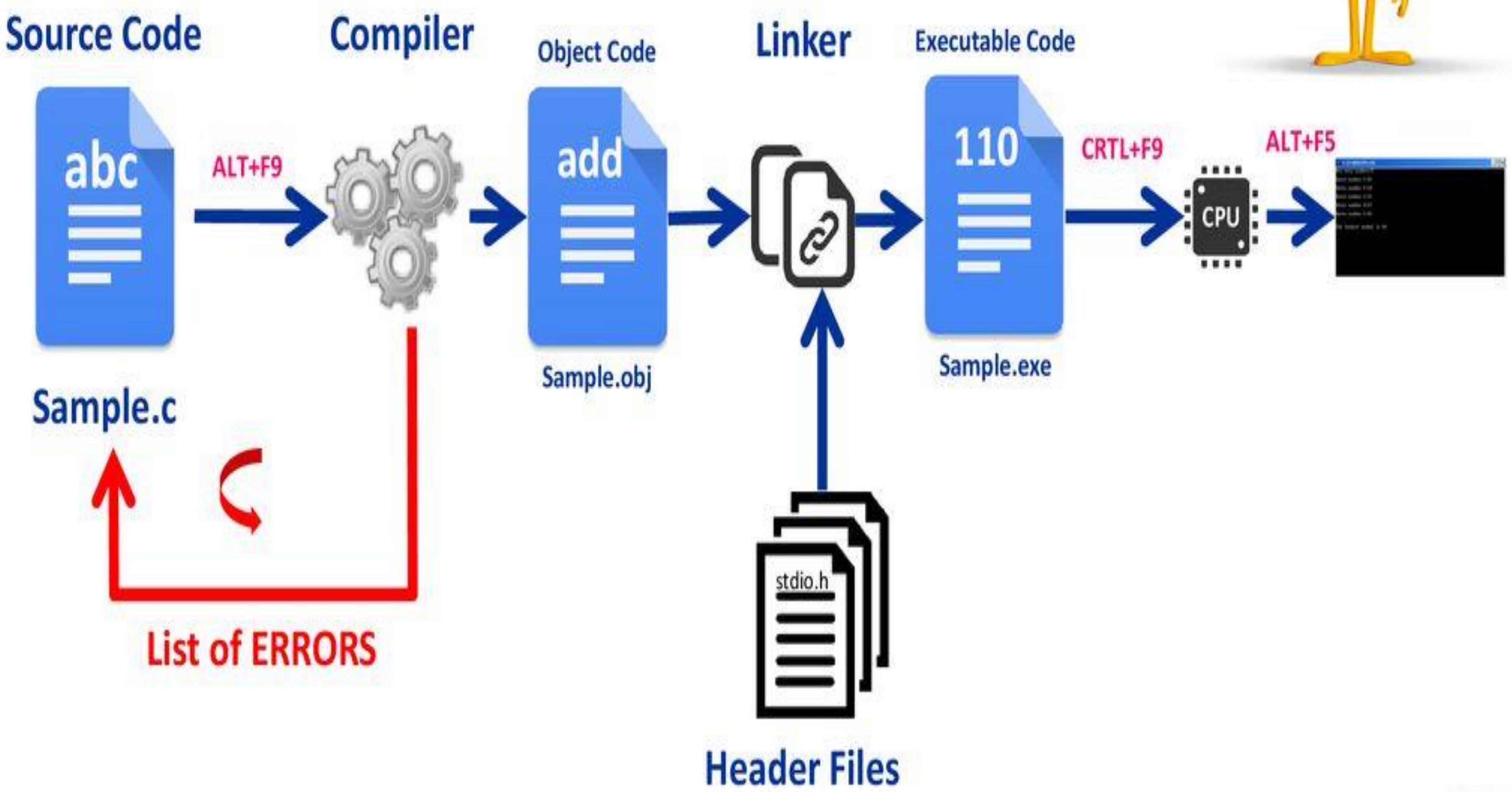




## LINKER and LOADER

- Constructs single executable program from multiple object code file compiled at different times
- Program can be subdivided into components and parcelled out to different developers
- Loader loads binary files that have been linked

# C Program Execution Process

**Can we have a quick review of the processes**



# Compiler Stages

- Scanner (lexical analysis)
- Symbol table construction
- Parser (syntax analysis)
- Sematic analysis and intermediate code generation
- Target code generation
- Machine-specific code improvement (optional)

- Lexical Analysis:
  - Comments and unnecessary white spaces are removed.
  - Keywords, constants and identifiers are replaced by 'tokens', which are symbolic strings to identify what the elements are.
  - Helps to identify token into the symbol table
  - Removes white spaces and comments from the source program
  - Correlates error messages with the source program
  - Helps you to expands the macros if it is found in the source program
  - Read input characters from the source program
- Symbol Table Construction:
  - A table stores the names and addresses of all variables, constants and arrays.


- Scanner (lexical analysis)
- Symbol table construction
- Parser (syntax analysis)
- Sematic analysis and intermediate code generation
- Target code generation
- Machine-specific code improvement (optional)

Scanner (lexical analysis)

Symbol table construction

Parser (syntax analysis)

Sematic analysis and intermediate code generation

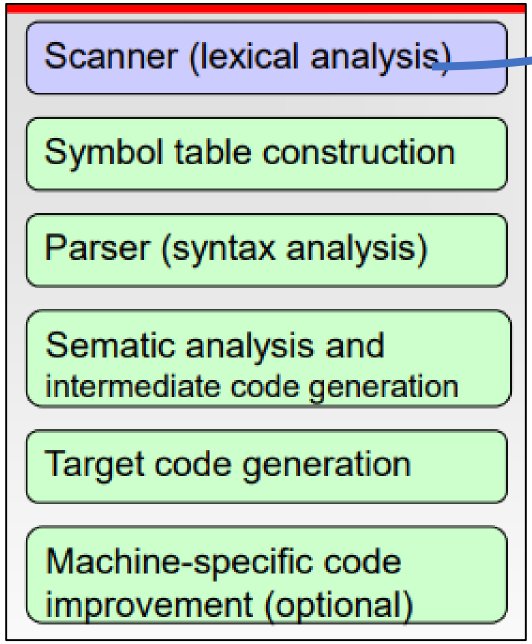
Target code generation

Machine-specific code improvement (optional)

# Lexical Analysis Example

- A line of code such as user_name = “Alex" is tokenised as follows:
- Assign the token identifier to user_name
- Assign the token operator to =
- Assign the token literal to “Alex"

| Lexeme | Token | Pattern |
|---|---|---|
| user_name | Identifier | Letter followed by digits or letters |
| = | Operator | = |
| “Alex" | Literal | Any string between a pair of single or double quotes |




# Lexical Analysis Example

**Ex-2:**

c = a + b;

| Lexeme | Token |
|---|---|
| c | identifier |
| = | operator |
| a | identifier |
| + | operator |
| b | identifier |

Mrinmoyee Mukherjee

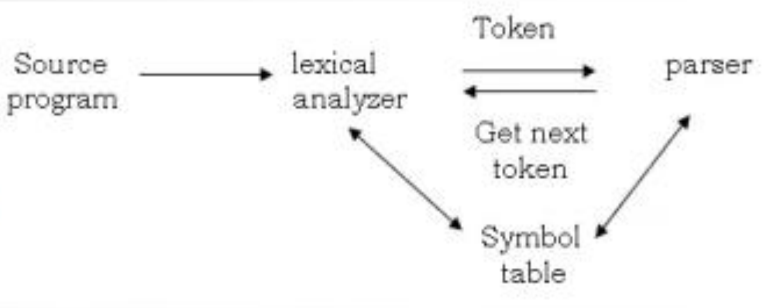
- Scanner (lexical analysis)
- Symbol table construction
- Parser (syntax analysis)
- Sematic analysis and intermediate code generation
- Target code generation
- Machine-specific code improvement (optional)

# Symbol Table Generation Example

**Ex-1:**

```
PROCEDURE print_discount_price(price)
    discount = price * 0.8
    PRINT("The discounted price is £" + STR(discount))
ENDPROCEDURE

price = INPUT("Enter the price: ")
fl_price = FLOAT(price)
print_discount_price(fl_price)
```




| Token | Data type | Scope |
|---|---|---|
| print_discount_price | procedure | global |
| price | float | print_discount_price |
| discount | float | print_discount_price |
| price | string | global |
| fl_price | float | global |

- Scanner (lexical analysis)
- Symbol table construction
- Parser (syntax analysis)
- Sematic analysis and intermediate code generation
- Target code generation
- Machine-specific code improvement (optional)

# Compiler Stages

- Scanner (lexical analysis)
- Symbol table construction
- Parser (syntax analysis)
- Sematic analysis and intermediate code generation
- Target code generation
- Machine-specific code improvement (optional)

- Syntax Analysis:
  - Tokens are checked to see if they match the syntax of the programming language.
  - If syntax errors are found, error messages are produced.
  - E.g. in C
    - Missing Parenthesis ()
    - Printing the value of variable without declaring it
    - Missing semicolon like this:

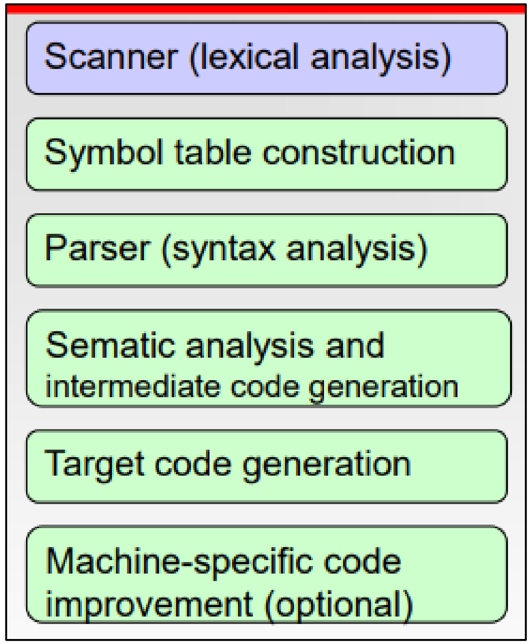
- Scanner (lexical analysis)
- Symbol table construction
- Parser (syntax analysis)
- Sematic analysis and intermediate code generation
- Target code generation
- Machine-specific code improvement (optional)

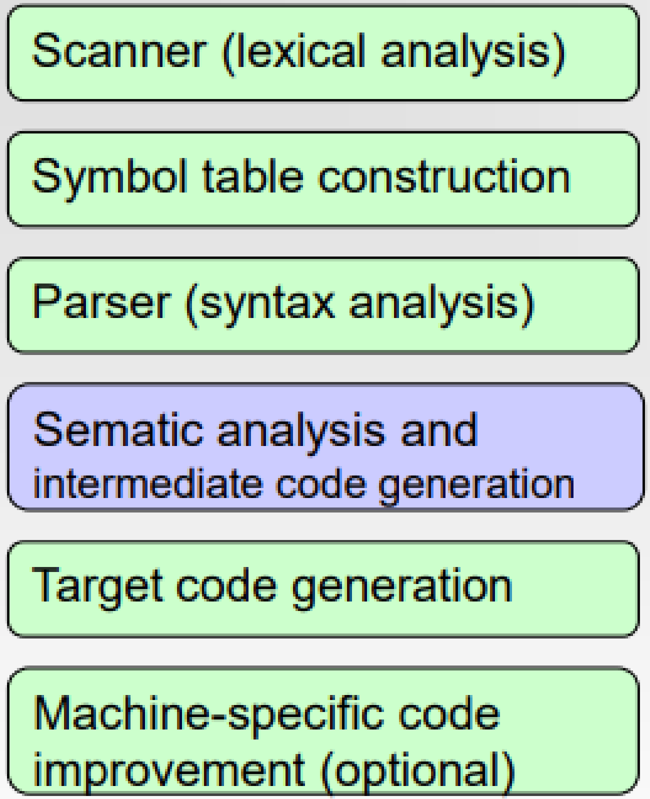
# Syntax Analysis Example

- The compiler checks that the tokens are in the correct order and that they follow the rules of the programming language being used.

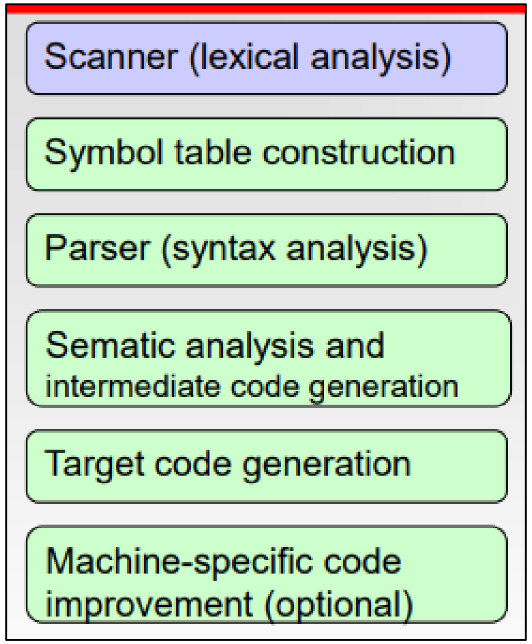
- Syntax rules differ between languages:

- if a > b: is syntactically correct in Python
- if a > b is not syntactically correct in Python — the rule for using a colon is broken
- if (a == b) is syntactically correct in Java
- if (a = b) is not syntactically correct in Java — a single equals sign is not a conditional operator




- Sematic Analysis:
  - Variables are checked to make sure they have been correctly declared and contain the correct data type.
  - Operations are checked to ensure that they are appropriate for the type of variable being used.
  - Types of semantic errors
    - Data type declaration mismatch
    - Trying to access an array index that is out of bounds
    - Variables being used before they have been declared
    - The wrong number of parameters being passed to a subroutine
    - Trying to access a variable that is out of scope
    - Attempting to use a reserved word from the language (eg print) as a variable name




# Semantic Analysis

- Semantic analysis can be used to determine whether the code is valid within a given context.
- **Example-1** of C++

```
int first_number = 8;
int second_number = "6";
```

- Above statements generate correct tokens, but second statement gives sematic error when the program is compiled.
- This is because declaring a string value for an identifier that has been declared to be an integer is not valid.
- **Example-2** of Python

```
fruit = ["apple", "peach", "pear"]
print(fruit[4])
```

- The second statement would produce a semantic error because within the context of the fruit list that has been defined, index 4 is out of bounds.

# Compiler Stages

- Scanner (lexical analysis)
- Symbol table construction
- Parser (syntax analysis)
- Sematic analysis and intermediate code generation
- Target code generation
- Machine-specific code improvement (optional)


- Scanner (lexical analysis)
- Symbol table construction
- Parser (syntax analysis)
- Sematic analysis and intermediate code generation


- Machine-independent code improvement (optional)


- Target code generation
- Machine-specific code improvement (optional)


- Target Code Generation:
  - Machine code is generated in this stage.
- Code Improvement or Optimization:
  - Code optimization makes the program more efficient so it runs faster and uses fewer resources.
  - Goal is to transform a program into a new version that computes the same result more efficiently—more quickly or using less memory, or both.

- Scanner (lexical analysis)
- Symbol table construction
- Parser (syntax analysis)
- Sematic analysis and intermediate code generation
- Target code generation
- Machine-specific code improvement (optional)

- **Example-1:**

```
FOR number FROM 1 TO 12
   multiplier = 9
   result = number * multiplier
   PRINT(result)
NEXT I
```

- This code is inefficient because the variable multiplier is assigned the value 9, within the loop, so this assignment happens twelve times. The value assigned to this variable never changes, so the code would be optimised by moving this statement before the loop.

- **Example-2:**

```
FUNCTION double_it(num)
   result = num * 2
   RETURN result
   PRINT("The number doubled is " + STR(result))
END FUNCTION
```

- The print statement in this example is known as dead code because it can never be executed.

# References


1. Alfred Aho, Jeffrey Ullman, “ Principles of compiler design”, Addison-Wesley (Chapter 1,2)
2. Ravi Sethi, “ Programming Languages-Concepts and constructs”, Pearson Education (Chapter 1 and 2)

Web Resources

1. NPTEL Lecture on “Principles of Compiler Design”, Lecture 01, Y.N Srikant, IISc Bangalore

(https://www.youtube.com/watch?v=yxnbvS2t_QA&list=PLbMVogVj5nJQNjkHZgwuAlfQ9tzmQDxjA)


Thank you!
seb seb
seg seb
seb seb