# COMPILER DESIGN

# Structure of the Course

- Lectures
  - 30 Hours
- Assignment (30%)
- Final Examination (70 %)
  - 4 Questions

1

# INTRODUCTION

# Computer

- Is a machine that can solve problems by obeying a sequence of instructions that also must be given to it.
- Does not think

# Programming

- In order to attempt to solve any problem on a computer
  - The problem must be analysed into these elementary steps.
  - Instructions must be given to the computer in a language that this particular computer has been constructed to read and obey.
- This is called “**PROGRAMMING**” the computer.

# Computer Programs

- A program is a sequence of instructions for a computer to follow in performing a task.
- These computer programs guide the computer through orderly sets of actions specified by people called **COMPUTER PROGRAMMERS**.

# Computer Programming

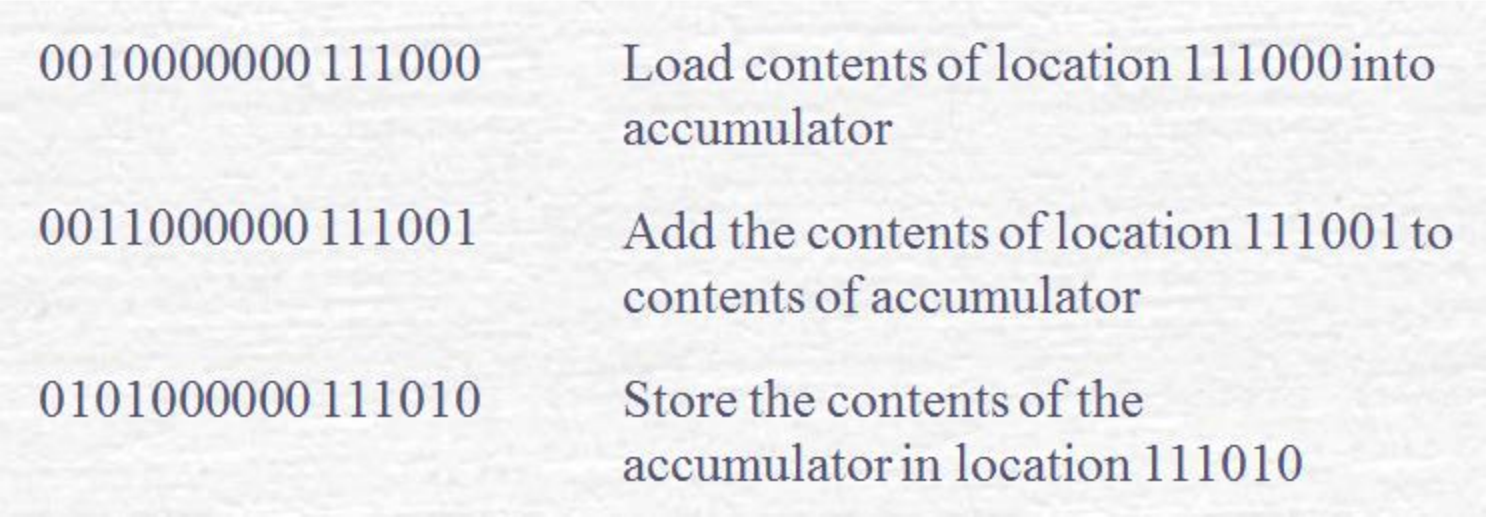
- Requires two skills
  1. You need to develop a plan for solving a particular problem.
  2. You need to learn a computer programming language.
  - C++
  - Java

# Types of Programming Languages

- Programs are written in many different languages
- Three categories of programming languages
  - Machine language
  - Assembly language
  - High-level language

# Machine Language

- Instructions are all binary

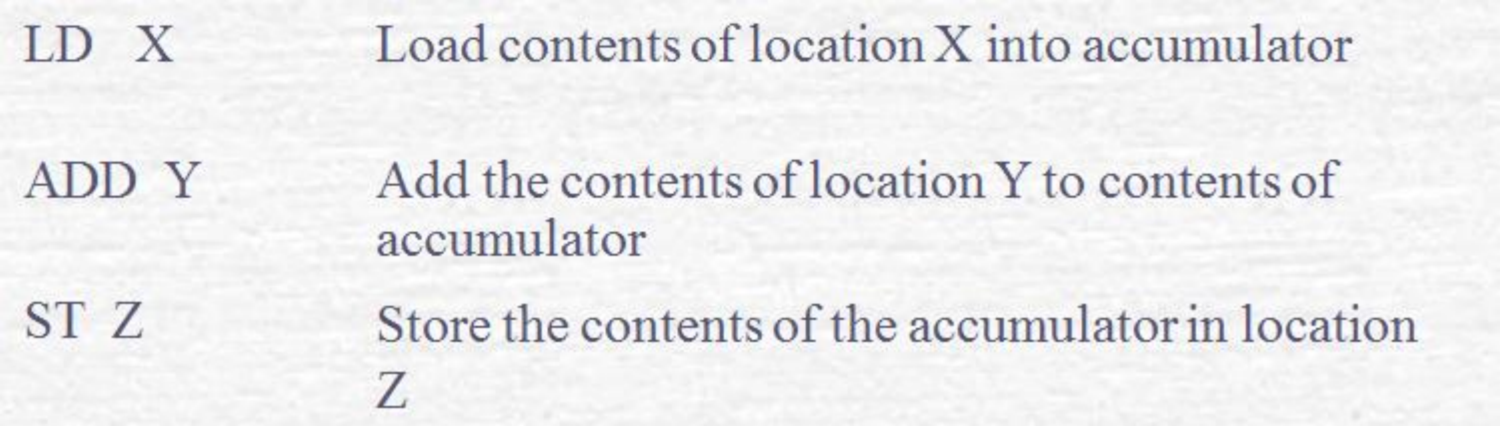


| | |
|---|---|
| 0010000000 111000 | Load contents of location 111000 into accumulator |
| 0011000000 111001 | Add the contents of location 111001 to contents of accumulator |
| 0101000000 111010 | Store the contents of the accumulator in location 111010 |

# Assembly Language

- A programming language that lets programmers write programs at the machine-language level but uses mnemonics representations of operators and symbolic representations of operand addresses.

# Assembly Language (cont..)



| | |
|---|---|
| LD X | Load contents of location X into accumulator |
| ADD Y | Add the contents of location Y to contents of accumulator |
| ST Z | Store the contents of the accumulator in location Z |

Addresses have been replaced by letters

# High-level Language

- Resemble human language in many ways
- Each statement is related to a procedural task and may translate to many machine language instructions.

# Language Translators

- A computer can execute only programs that are in machine language.
- A program in assembly language or in high-level language must be translated into machine language before it can be executed.

# Source Programs

- A program that is written in a high-level language is called source program or source code.
- Source programs are the input to a translator.

# Object Program

- The translated version produced by the compiler is known as the object program or object code.
  - The word code is frequently used to mean a program or a part of a program.

# Assembler

- An assembler translates assembly language programs into machine code.
- The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

# Interpreters

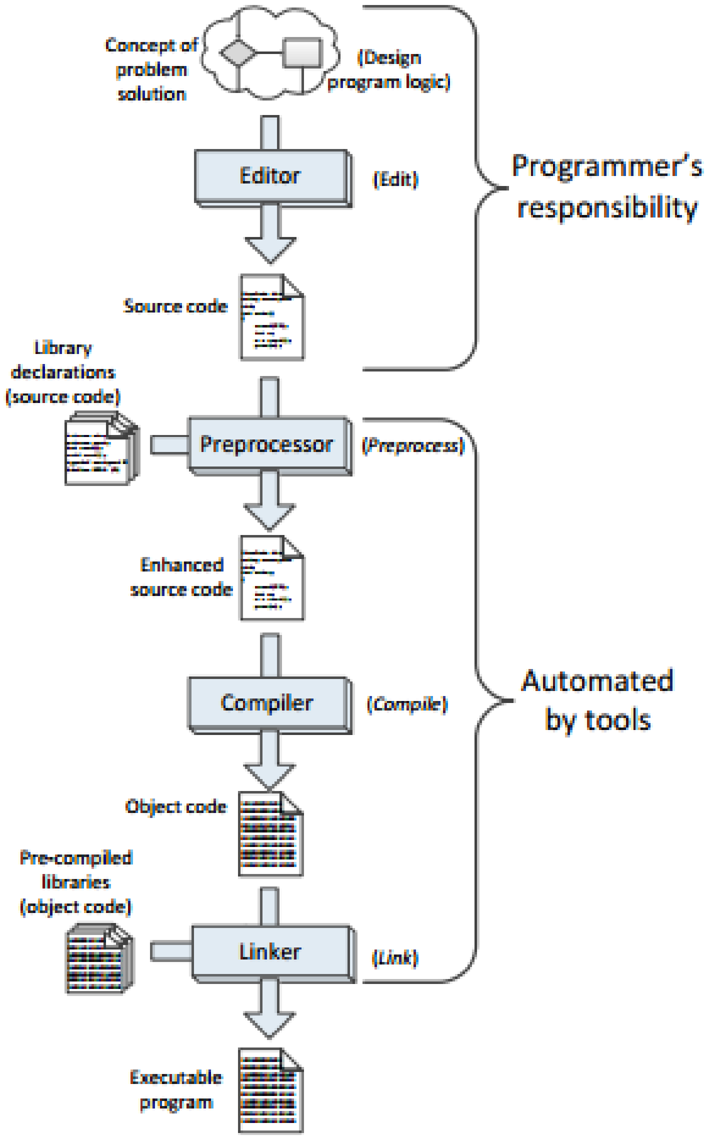
- An interpreter, like a compiler, translates high-level language into low-level machine language.
- The difference lies in the way they read the source code or input.
- A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes.
- In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it; whereas a compiler reads the whole program even if it encounters several errors.

# Compiler

- A program that translates a program written in a high-level language into a program written in the machine language which the computer can directly understand and execute.

# Simple view of running a program

The complete set of build tools for C++ includes a preprocessor, compiler, and linker.



# Preprocessor

- Adds to or modifies the contents of the source file before the compiler begins processing the code.
- Use the services of the preprocessor mainly to #include information about library routines our programs use.

# Linker

- Combines the compiler-generated machine code with precompiled library code or compiled code from other sources to make a complete executable program.
- Most compiled C++ code is incapable of running by itself and needs some additional machine code to make a complete executable program.
- The missing machine code has been precompiled and stored in a repository of code called a library.
- A program called a **LINKER** combines the programmer’s compiled code and the library code to make a complete program.

# Loader

- Loader is a part of operating system and is responsible for loading executable files into memory and execute them.
- It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

# COMPILER DESIGN – ARCHITECTURE

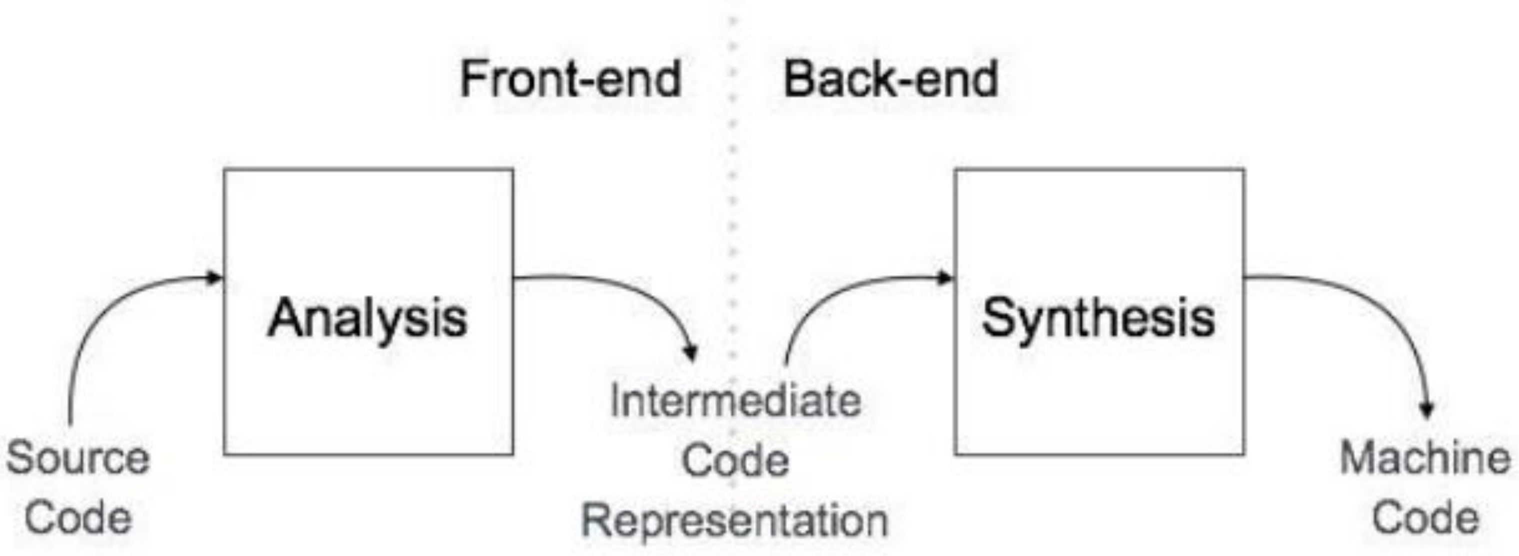
# COMPILER DESIGN – ARCHITECTURE

- A compiler can have many phases and passes.
- **Pass** : A pass refers to the traversal of a compiler through the entire program.
- **Phase** : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage.

- A pass can have more than one phase.

# COMPILER DESIGN – ARCHITECTURE

- A compiler can broadly be divided into two phases based on the way they compile.



# Analysis Phase

- The front-end of the compiler.
- The **analysis** phase of the compiler reads the source program, divides it into core parts, and then checks for lexical, grammar, and syntax errors.
- The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

# Synthesis Phase

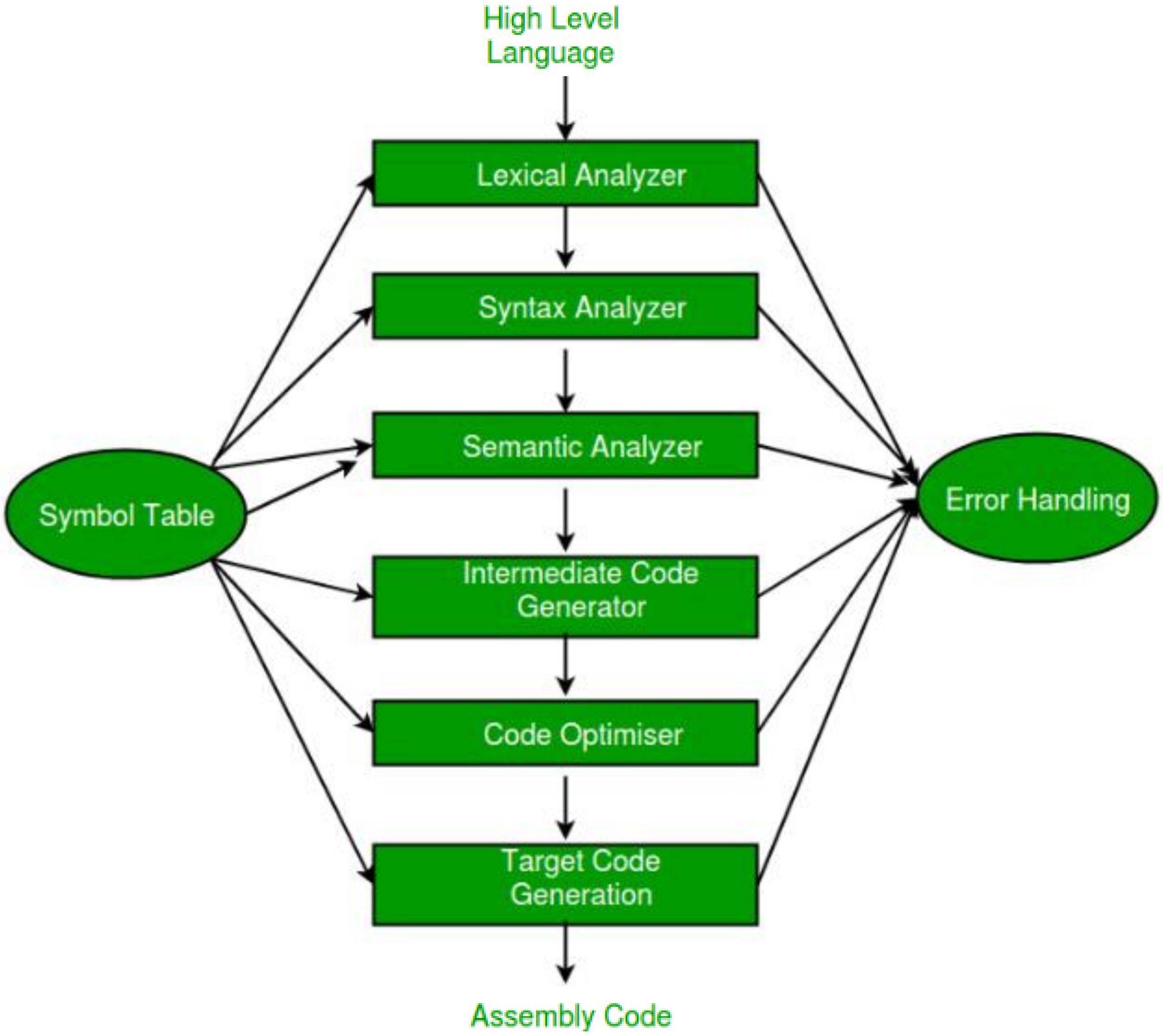
- The back-end of the compiler, the **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table.

# PHASES OF COMPILER

# PHASES OF COMPILER

- The compilation process is a sequence of various phases.
- Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.

# PHASES OF COMPILER



# Analysis Phase

- An intermediate representation is created from the give source code :
  - Lexical Analyzer - Divides the program into “tokens”
  - Syntax Analyzer - Recognizes “sentences” in the program using syntax of language
  - Semantic Analyzer - Checks static semantics of each construct

# Synthesis Phase

- Equivalent target program is created from the intermediate representation. It has three parts :
    - Intermediate Code Generator - Generates “abstract” code
    - Code Optimizer - Optimizes the abstract code
    - Code Generator - Translates abstract intermediate code into specific machine instructions

# Lexical Analysis

- The first phase of scanner works as a text scanner.
- This phase scans the source code as a stream of characters and converts it into meaningful lexemes.
- Lexical analyzer represents these lexemes in the form of tokens as:

**<token-name, attribute-value>**

# Syntax Analysis

- The next phase is called the syntax analysis or **parsing**.
- It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree).
- In this phase, token arrangements are checked against the source code grammar, i.e., the parser checks if the expression made by the tokens is syntactically correct.

# Semantic Analysis

- Semantic analysis checks whether the parse tree constructed follows the rules of language.

- For example:
  - Assignment of values is between compatible data types, and adding string to an integer.
  - Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not, etc.

- The semantic analyzer produces an annotated syntax tree as an output.

# Intermediate Code Generation

- After semantic analysis, the compiler generates an intermediate code of the source code for the target machine.
- It represents a program for some abstract machine.
- It is in between the high-level language and the machine language.
- This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

# Code Optimization

- The next phase does code optimization of the intermediate code.
- Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

# Code Generation

- In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language.
- The code generator 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 identifiers' 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.

# Next Week……

- LEXICAL ANALYSIS