# Introduction to Compiler
# 2CS701 Compiler Construction

Prof Monika Shah

Nirma University

# Outline

- Course overview
- Compiler and Interpreter
- Other types of compiler
- Analysis and Synthesis Model of compilation
- Phases of compiler
- Cousins of compiler
- Other applications of compilation techniques

# Course Overview
# 2CS701 Compiler Construction

- Introduction :

  The course will discuss how a program written in H.L.L.(higher level language) is systematically translated into L.L.L(low level language)

  It also help you to understand various programming constructs and their semantics

- Prerequisites:
  - C/C++ programming skill,
  - Data structure
- Course website : https://lms.nirmauni.ac.in/course/view.php?id=4849

  for : Syllabus, LP/LOP, Handouts, References, Assignment, Forum
- Textbook: "*Compilers: Principles, Techniques, and Tools*" by Aho, Sethi, and Ullman

# Objective & Course Learning Outcomes

**Objective**

To make student understand programming language constructs, and

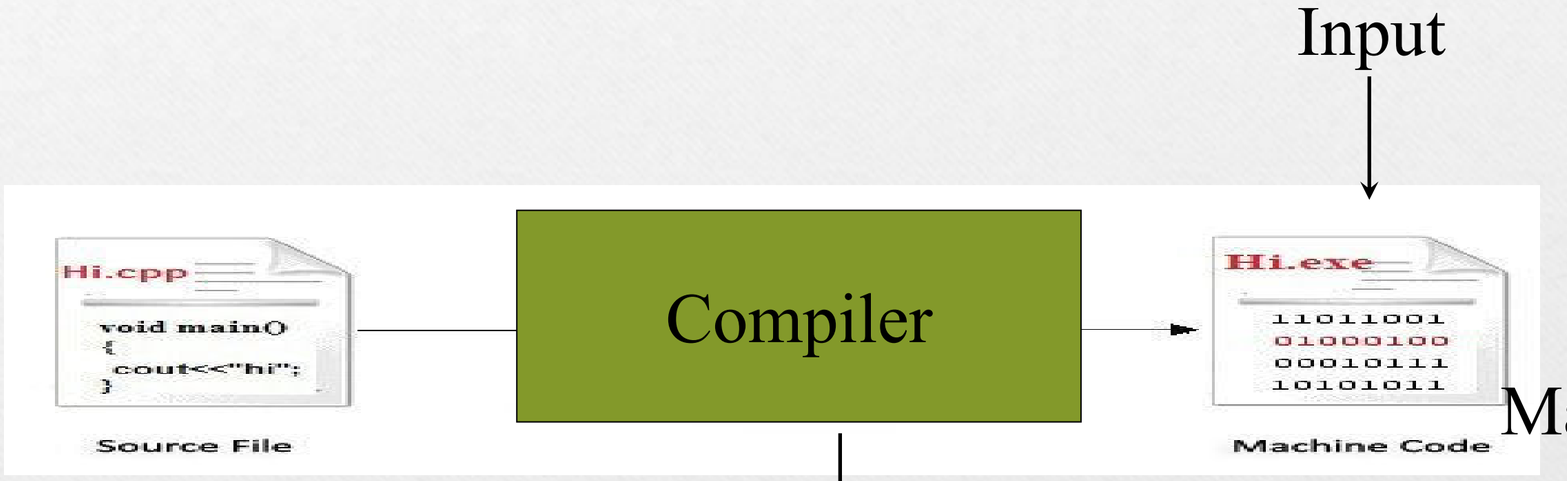
give them hands-on experience with crafting a simple compiler using modern software tools.

**Course Learning Outcomes**

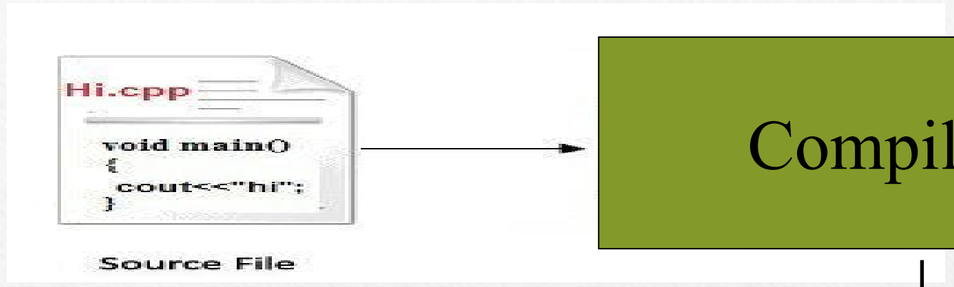
- Summarize the functionalities of various phases of compiler
- Apply language theory concepts to various phases of compiler design
- Identify appropriate optimization technique for compilation process
- Develop a miniature compiler using appropriate compiler design tool

# Compilers and Interpreters

- "*Compilation*"
  - Translation of a program written in a source language into a semantically equivalent program written in a target language



# Compilers and Assembler




Program in
Source Language

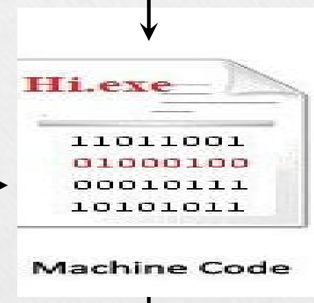
Compiler

Error messages

Hi.asm
(Assembly code)

Assembler

Input

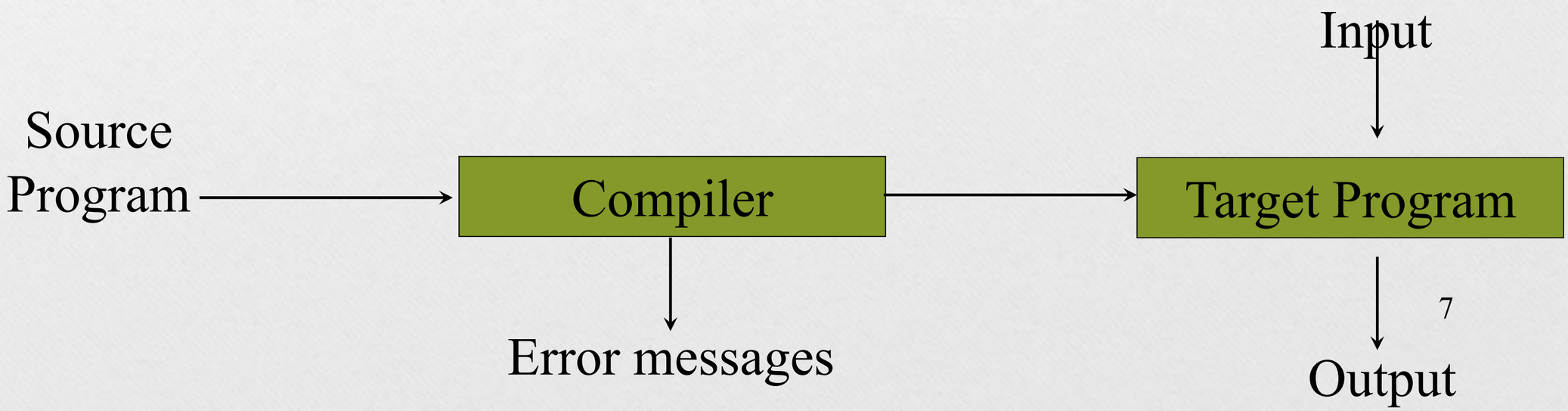



Output

# Compiler

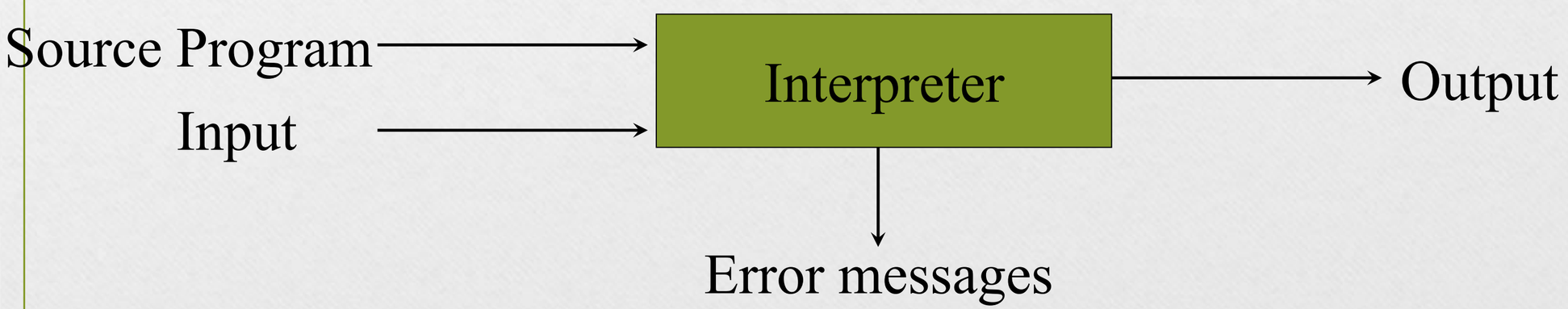
- Translation of a program written in a source language into a semantically equivalent program written in a binary/machine dependent language

| Compiler | Source Language | Target Language |
|---|---|---|
| Gcc | C | Binary / Machine Language |
| G++ | C++ | Binary / Machine Language |
| Javac | Java | Byte Code |



# Interpreters

- Translation of an instruction of a program written in a source language into a semantically equivalent instruction written in a target (machine) language
- Execute translated line with given input
- E.g. Debugger , command line interpreter , Python Interpreter, Perl Interpreter, PHP interpreter



# Types of compilers

- **Basic Ahead of Time (AOT) Compilers:** translate entire code into machine code before execution i.e. **GCC for C,C++, Fortran,**

- **Just-In-Time Compiler :** Translating code at runtime. **i.e. HotSpot Java Virtual Machine (JVM)**

- **Incremental Compiler :** Recompile only the modified parts of a program. E.g. **Eclipse Java Development Tools (JDT)**

- **Interpreting Compilers :** combine compilation and interpretation, translating high-level code into intermediate code, which is then interpreted. E.g.
  - Matlab - Interpreter with JIT Compilation
  - Python (Cpython) - Interpreter with Bytecode Compilation

# Types of compilers

- **Cross Compilers:** They produce an executable machine code for a platform but, this platform is not the one on which the compiler is running.

  **Applications:**
  - to separate build environment from target environment.
  - Often use for Embedded system design.
  - To support multiple operating systems or several versions of operating systems

  **Examples:**
  - a compiler that runs on a PC but **generates code that runs on an Android smartphone** is a cross compiler.
  - **Android NDK (Native Development Kit).** The Android NDK includes tools like **clang** and **gcc** that allow you to compile C and C++ code for Android devices

# Types of compilers

- **Source to source compiler / Transpiler / Transcompiler :** takes the source code of a program written in a programming language as its input and produces the equivalent source code in the same or a different programming language.
- Applications :
  - **An automatic parallelizing compiler** will frequently take in a HLL program as an input and then transform the code and annotate it with parallel code (e.g., OpenMP)
  - **translating legacy code to use the next version** of the underlying programming language or an API that breaks backward compatibility

| Compiler | Source Language | Target Language |
|---|---|---|
| Cfront | C++ | C |
| HPHPc | PHP | C++ |
| JSSweet | Java | Typescript |

# Types of compilers

- **Bootstrap Compilers.** These compilers are written in a programming language that they have to compile.
  - Can compile its own source code
  - E.g. GCC, LLVM

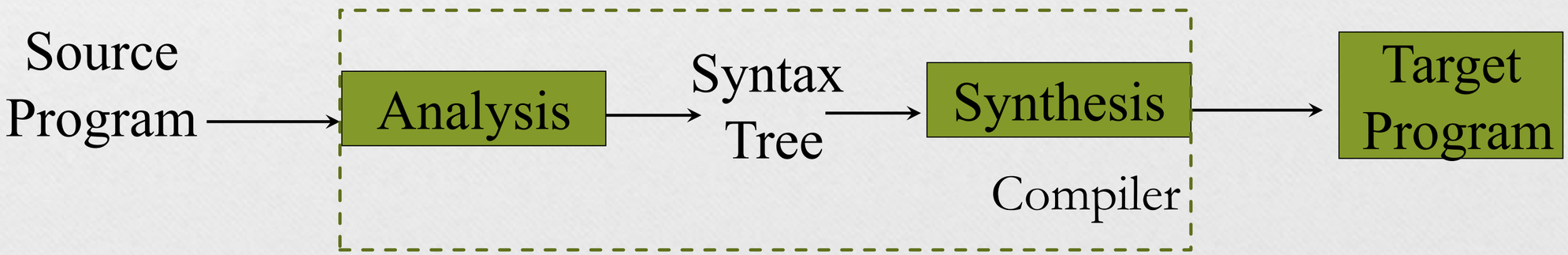
- **Decompiler** : translates an executable file to a high-level source file which can be recompiled successfully.

# Self Evaluation

- Is JavaC Compiler or Interpreter?
- Is compiler faster than Interpreter? How?
- What is need of Decompiler ?

# The Analysis-Synthesis Model of Compilation

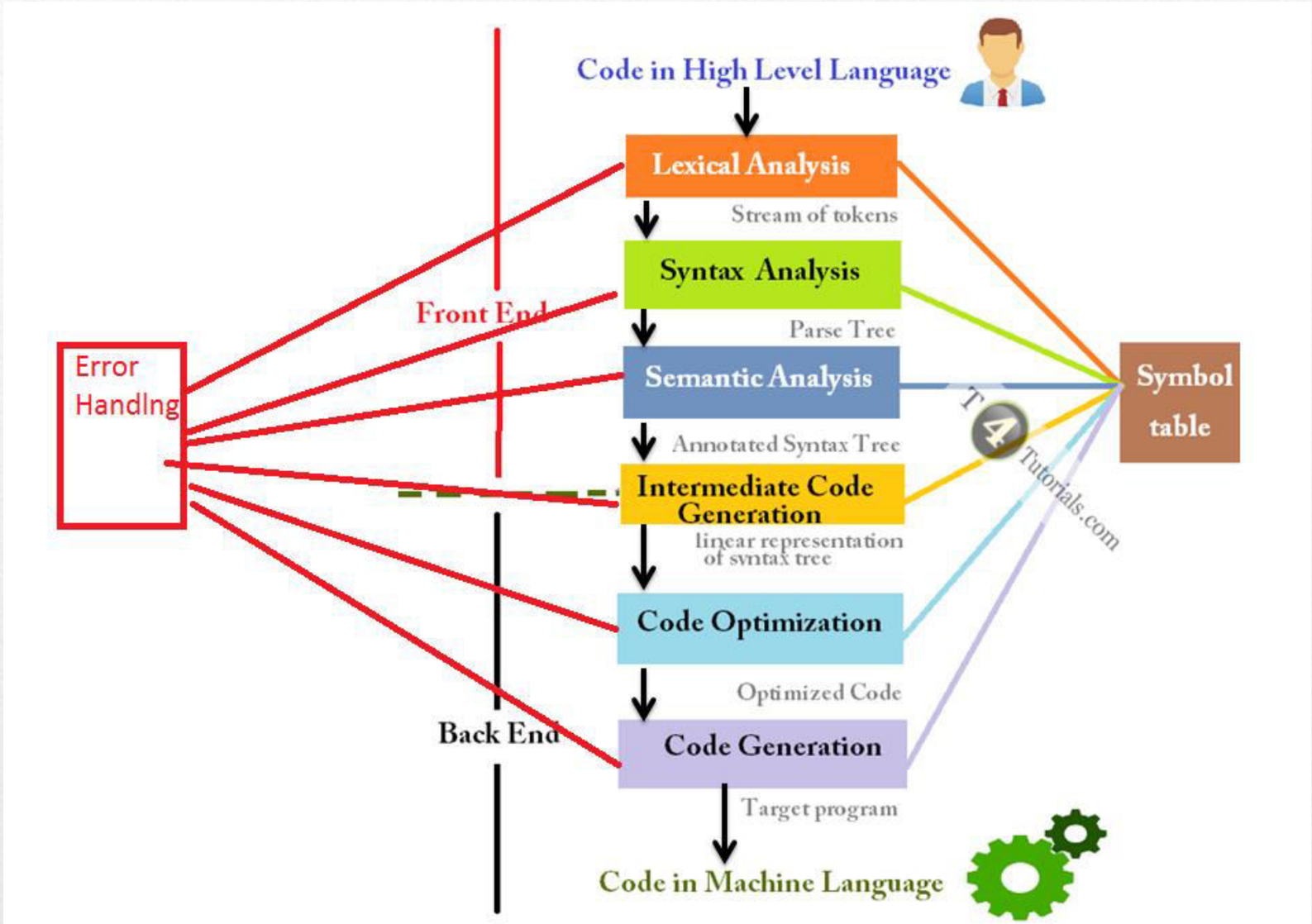
- There are two parts to compilation:
  - ***Analysis***
    - Understand source program as per program language used in source program
    - determines the operations implied by the source program which are recorded in a tree structure
  - ***Synthesis***
    - takes the tree structure and translates the operations there in target language and compile it into the target program



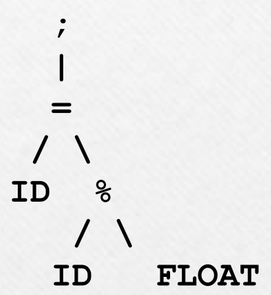
# Other Tools that Use the Analysis-Synthesis Model

- *Editors* (syntax highlighting)
- *Pretty printers* (e.g. Doxygen)
- *Static checkers* (e.g. Lint and Splint)
- *Interpreters*
- *Text formatters* (e.g. TeX and LaTeX)
- *Silicon compilers* (e.g. VHDL)
- *Query interpreters/compilers* (Databases)
- Circuit design from K-map

# The Phases of a Compiler



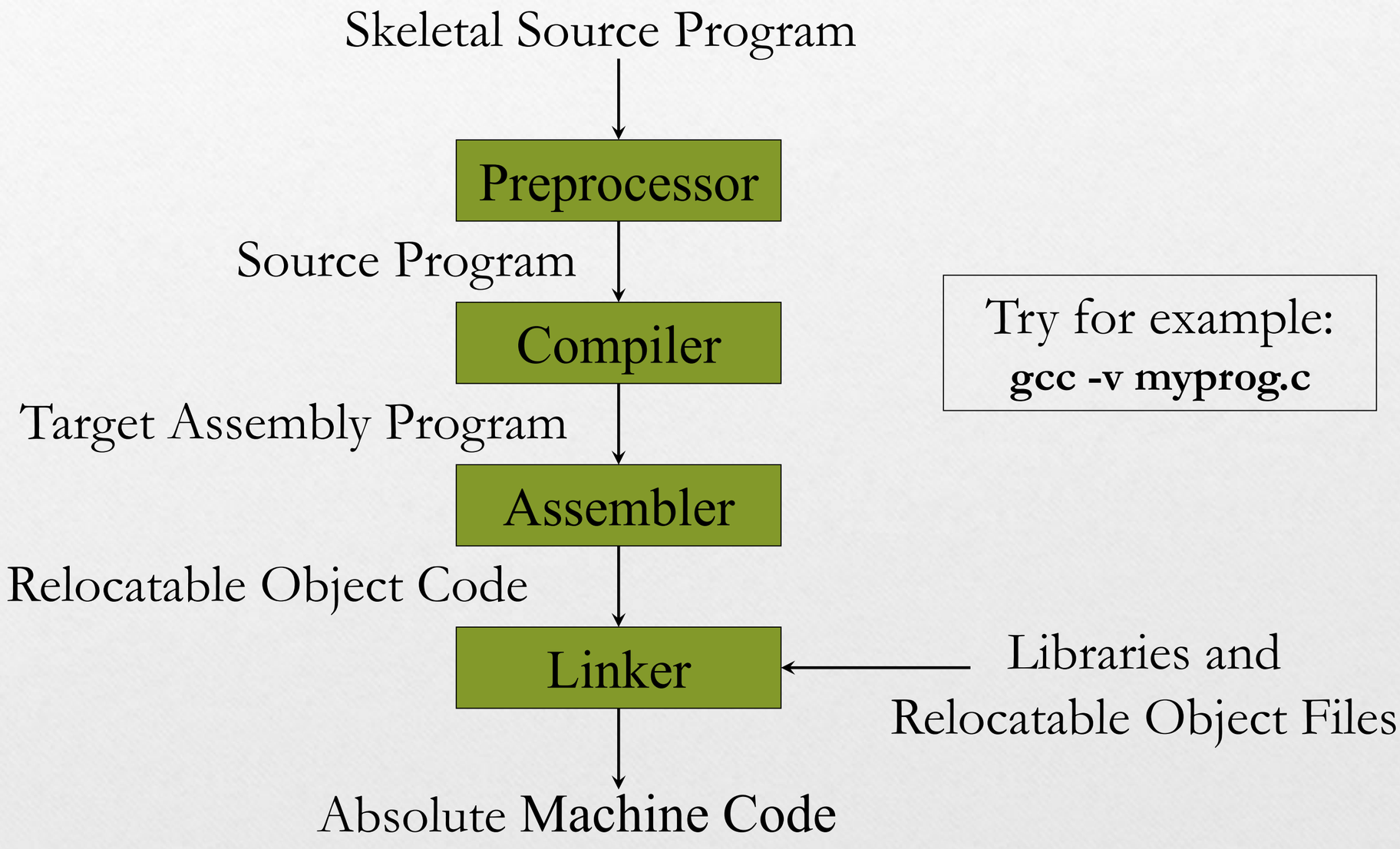
# The Phases of a Compiler

| Phase | Output | Sample |
|---|---|---|
| *Programmer (source code producer)* | Source string | **`A=A/5.2;`** |
| *Scanner* (performs *lexical analysis*) | Token string | **ID '=' ID '%' FLOAT ';'**<br>And *symbol table* with names |
| *Parser* (performs *syntax analysis* based on the grammar of the programming language) | Parse tree or abstract syntax tree<br>Or Syntax Error |  |
| *Semantic analyzer* (type checking, etc) | Annotated parse tree or abstract syntax tree | **`Error:'%' operator should have both operand integer.`**<br>**`ID '=' ID '%' fp2Int(FLOAT)`** |
| *Intermediate code generator* | Three-address code, quads, or RTL | **`fp2int 5.2 t1`**<br>**`% A t1 t2`**<br>**`:= t2 A`** |
| *Optimizer* | Three-address code, quads, or RTL | **`fp2int 5.2 t1`**<br>**`% A t1 A`** |
| *Code generator* | Assembly code | **`MOVF #5.2,r1`**<br>**`ADDF2 r1,r2`**<br>**`MOVF r2,A`** |
| *Peephole optimizer* | Assembly code | **`ADDF2 #5.2,r2`**<br>**`MOVF r2,A`** |

# Cousins of compiler

## playing crucial role in compilation process

Preprocessors, Compilers, Assemblers, and Linkers



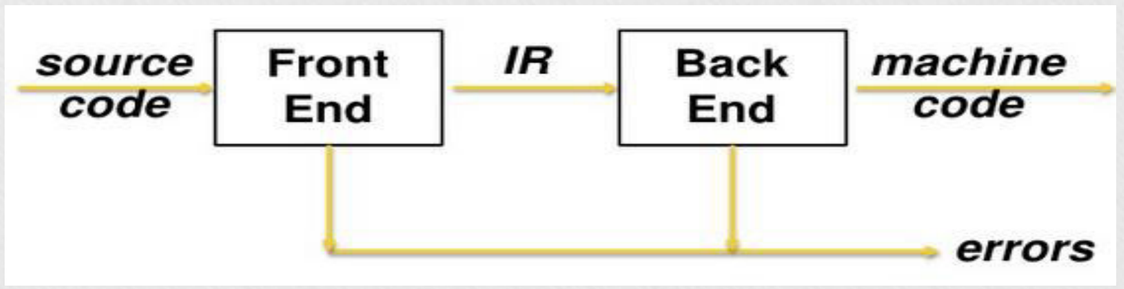

Try for example:
**gcc -v myprog.c**

# Self Evaluation

1. Which compiler phase is optional ? Why?
2. Which compiler phase concept can be apply in text editor for spell check?
3. Which compiler phase concept can be apply in text editor for grammar check?
4. “Compiler can generate assembly code as output”. State True/False. Justify
5. Does compiler recognize semantic error and reports it?
6. What is difference between code optimization phase before and after code generation?

# The Grouping of Phases

- Compiler *passes:*
  - Single pass:
    - Read source code once
    - Read one part, process all phases, read next part
    - Does not look code previously processed
    - Require everything to be defined before. Else Use Backpatch
    - Require large memory
- Multi pass: Every pass results new representation and input to next pass
  - Compiler *front* and *back ends*:
  - Front end: *analysis* (*machine independent*)
  - Back end: *synthesis* (*machine dependent*)



# Other Applications of techniques used in compiler design

- Lexical Analyzer → text editors, information retrieval system, and pattern recognition programs. E.g. pretty printers apply stylist formatting to source code, markup like text using indenting styles, coloring token classes
- Syntax Analyzer → query processing system such as SQL, K-map to circuit design
- Syntax Analyzer + semantic analyzer → Equation solver
- Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems.

# Other Application of Lexical Analyzer
# E.g. Pretty-printing : Formatting coding

```
int foo(int k){if(k<1||k>2)
{printf("out of range\n");
printf(“Requires a value of 1 or 2\n");}
else{printf("Switching\n");}}
```

GNU Indent Program

```
int foo(int k)
{
  if(k<1||k>2)
    {
      printf("out of range\n");
      printf(“Requires a value of 1 or 2\n");
    }
  else
    {
      printf("Switching\n");
    }
}
```

# Self Evaluation

- What are advantages and disadvantages of single pass compiler and multi-pass compiler?
- Why is it preferred to keep front-end phases and back-end phases into different pass?
- Find at-least 3 applications of compiler techniques other than compiler