German Jordanian University الجامعة الألمانية الأردنية

CS419 Compiler Construction

Lecture 1

Slides Originally by Dr. Robert A. Engelen
Author of the Textbook

Edited by: Dr. Ismail Hababeh and Dr. Mohammad Daoud
German Jordanian University

Course Information

- **Course Number: CS419**
- **Course Name: Compiler Construction**
- **Teaching Method: Face-to-Face**

Course Outline

- Ch. 1: Introduction
- Ch. 2: A simple syntax-directed translator
- Ch. 3: Lexical analysis
- Ch. 4: Syntax analysis
- Ch. 5: Syntax-directed translation
- Ch. 6: Intermediate code generation
- Ch. 7: Run-time environments
- Ch. 8: Code generation
- Ch. 9: Code optimization

Course Requirement

- **Prerequisites:** Algorithms and Data Structures (CS 222 & CS223)
- **Textbook:** "Compilers: Principles, Techniques, and Tools" by A.V. Aho, M. S. Lam, R. Sethi, and J.D. Ullman

Course Overview

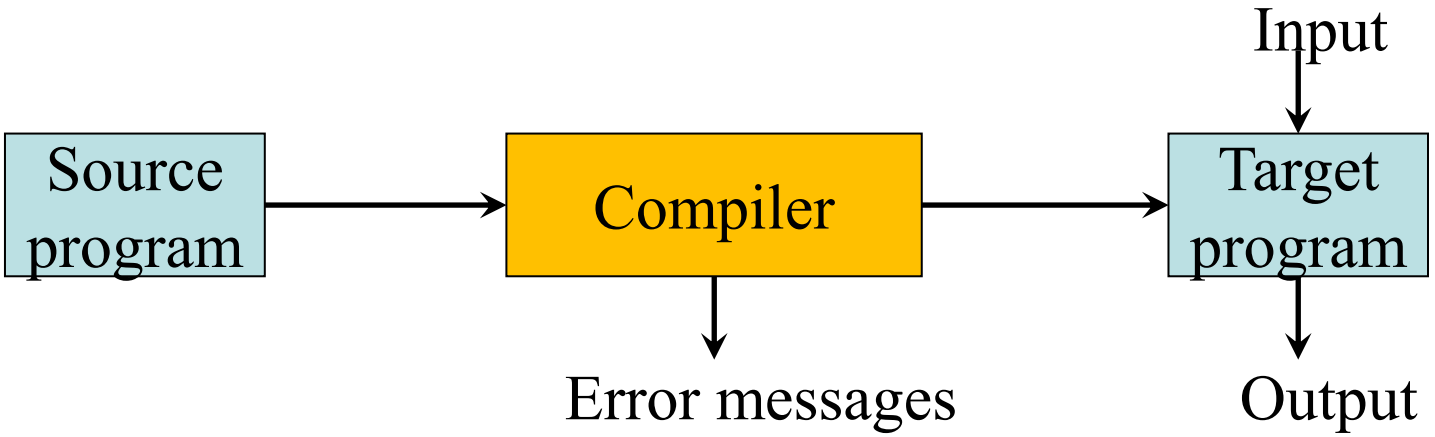
- Understand the design and construction principles of programming languages' compilers
- Learn new concepts, techniques, and tools that can be applied to build compilers
- Be familiar with compiler analysis and optimization techniques

Compilers and Interpreters

- Both compilers and interpreters are language processors
- The target program produced by a compiler is faster than an interpreter
- An interpreter provides better error diagnostics than a compiler (executes the source code)

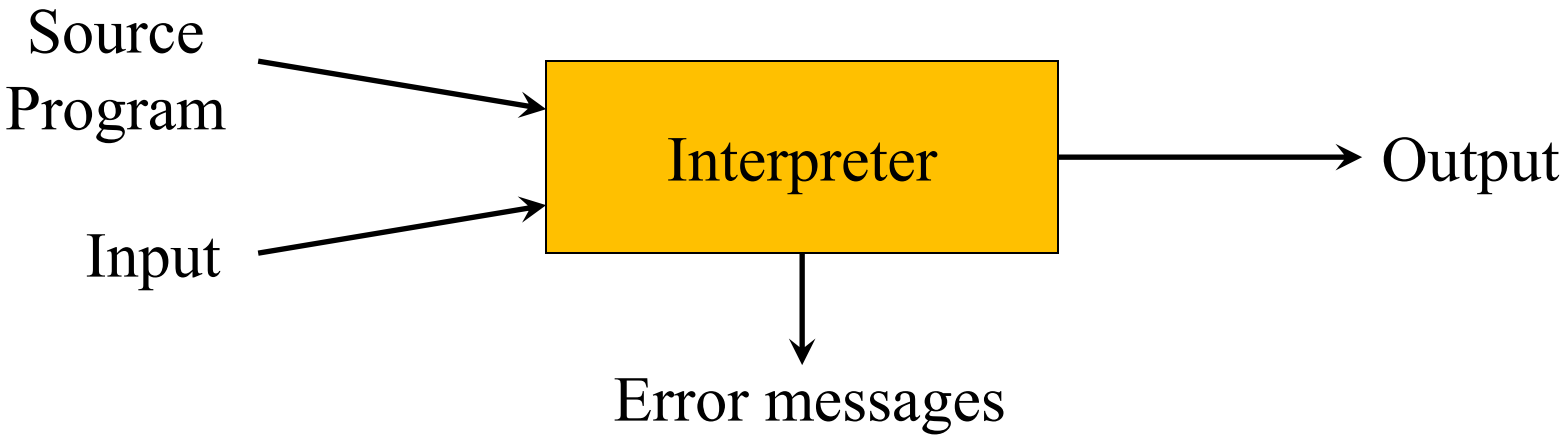
Compilers

- **Compiler**
  - A software (program) that translates a program in the *source* language into a semantically equivalent program in the *target* language (machine language)
  - Reports the source program errors that are detected during translation



Interpreters

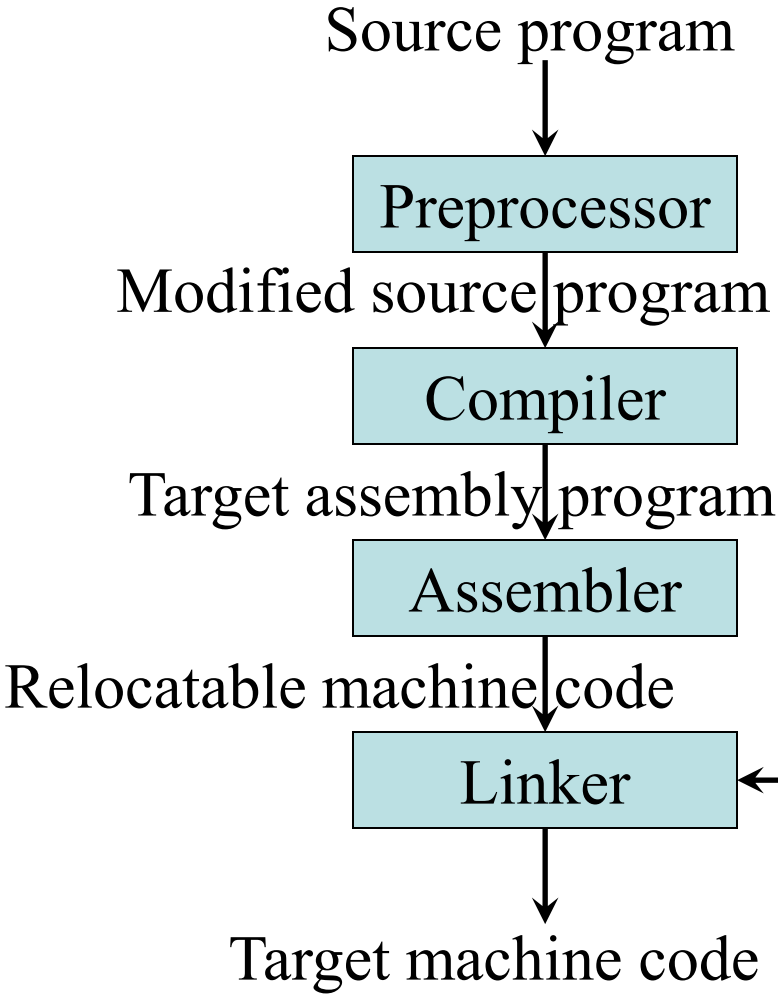
- Interpreter
  - Directly performs the operations implied by the source program without producing a target program



Preprocessors, Compilers, Assemblers, and Linkers

- If the source program is divided into modules stored in separate files, the **preprocessor** collects the source program
- The **compiler** translates the source program to produce an assembly-language program
- The assembly-language program processed by an **assembler** to produce relocatable machine code
- The **linker** links the relocatable machine code with other object files and library files to produce a code that runs on the machine

Preprocessors, Compilers, Assemblers, and Linkers



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

Libraries and
Relocatable Object Files

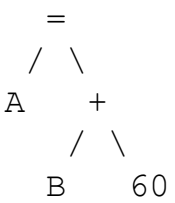
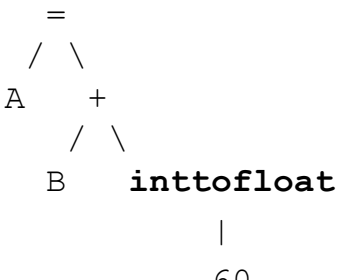
The Analysis-Synthesis Model of Compilation

- There are two main parts of compilation:
  - *Analysis* breaks up the source program into basic pieces with grammatical structure. This structure is then represented as a tree.
  - *Synthesis* takes the tree structure and translates its operations into the target program

The Phases of a Compiler

- The compilation process operates as a sequence of phases
- Each compilation phase transforms one form of the source program to another representation form

The Phases of a Compiler

Phase	Output	Sample
Programming	Source string	**A=B+60**
Lexical analysis (scanning)	Token string	**'A', '=', 'B', '+', '60'** And *symbol table* for identifiers
Syntax analysis (parsing) (creates a tree representation that depicts the grammatical structure of the token string)	Syntax tree (each interior node represents an operation and the children of the node represent the arguments of the operation)	
Semantic analysis (checks the source program for semantic consistency with the language definition: type checking and type correction)	Syntax tree	
Intermediate code generation (generates a machine-like representation of the source program. This representation should be easy to translate into the target machine)	Three-address code	**t1 = inttofloat(60)** **t2 = B + t1** **A = t2**

The Phases of a Compiler …Continued

Phase	Output	Sample
Code optimization (improves the intermediate code to make it faster, shorter , etc)	Three-address code, quads, or RTL	`t1 = inttofloat(60)` `A = B + t1`

Code generation
(maps the intermediate code into the target language)
	Assembly code	`MOVF #60.0,r1`
		`ADDF r1,r2`
		`MOVF r2,A`

Peephole optimizer	Assembly code	`ADDF #60.0,r2` `MOVF r2,A`