MODULE 9: Software Systems and Tools

Lecture 9.3
Programming Tools

Prepared By:

- Scott F. Midkiff, PhD
- Luiz A. DaSilva, PhD
- Kendall E. Giles, PhD

Electrical and Computer Engineering

Virginia Tech

Lecture 9.3 Objectives

- Describe the basic function of an assembler
- Describe the basic function of a high-level language compiler
- Describe the need for linking and loading
- Differentiate between static linking and dynamic linking at execution time
- Describe the basic function of an interpreter and differentiate it from a compiler

Overview of Programming Tools

- Assemblers – produce machine code from human-readable (mnemonic) assembly language programs
- Compilers – produce assembly programs or machine code from human-readable high-level languages (like C, C++, C#, Java, FORTRAN, etc.)
- Linker – combine different sets of machine code before loading and execution
- Dynamic linker – combine different sets of machine code (dynamic link libraries) at the time of execution
- Interpreters – Process high-level language statements at execution time

Assemblers and Assembly Language

- An assembly language program is a sequence of assembly language instructions
    - Processor instructions that are supported by the processor’s instruction set architecture (ISA)
    - Assembly language directives or pseudo-operations
- Pseudo-ops direct the assembler to perform some action when the program is assembled
    - Executed by the assembler, not by the processor
- The assembler converts the instructions, under the direction of pseudo-operations, to produce machine code for execution on a processor

Compilers and High-Level Languages

- A high-level language (HLL) represents functions at a higher level of abstraction
    - Assembly language corresponds to processor instructions
    - HLL corresponds to programming and functional abstractions
- A compiler converts a high-level language program into an assembly or machine language program for execution on a processor

Six Steps of the Compilation Process

- Six steps of compilation
    - Lexical analysis
    - Syntactical analysis or parsing
    - Semantic analysis
    - Intermediate code generation
    - Code optimization
    - Code generation
- Steps build and share a common symbol table

Compilation Process

- Lexical analysis
  - Identifies the basic symbols of the language to extract tokens (language primitives)
- Syntactical analysis or parsing
  - Identifies the underlying program structure using symbols to create a parse tree

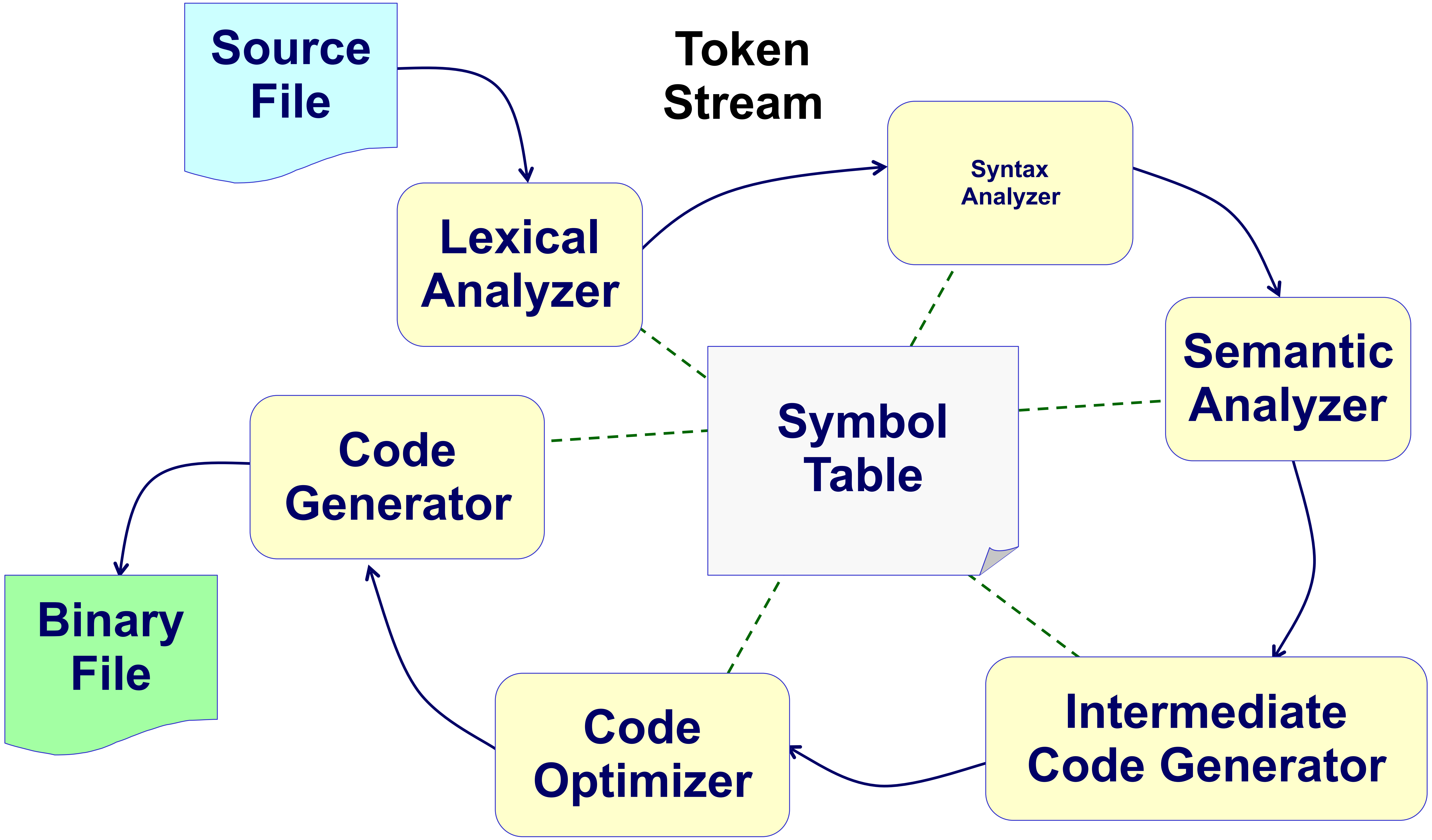
Compilation Process (2)

- Semantic analysis
    - Associates names with variables and specific memory locations
    - Identifies the data types of variables
    - Ensures correct association of data types and operations
- Intermediate code generation
    - Associates program statements with a sequence of assembly or machine language instructions

Compilation Process (3)

- Code optimization
  - Matches best instructions to operations
  - Optimizes uses of registers
  - Removes unnecessary code and variables
- Code generation
  - Create assembly language or (more often) machine code for linking and execution

Compilation Process (4)



CHECK POINT

As a checkpoint of your understanding, please pause the video and make sure you can do the following:

- Describe the basic function of an assembler
- Describe the basic function of a high-level language compiler

If you have any difficulties, please review the lecture video before continuing.

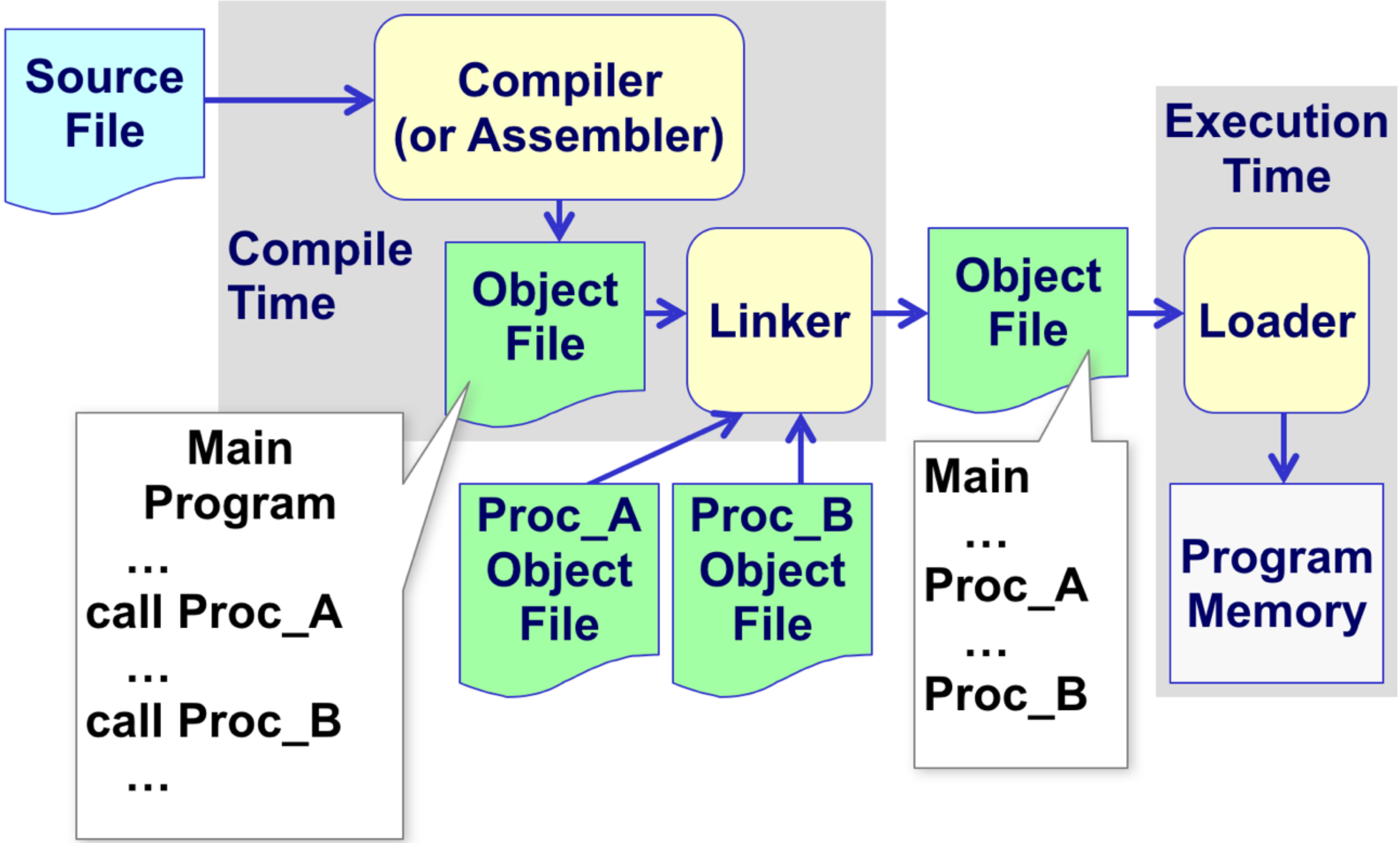
Linker

- A “linker” combines separately assembled or compiled “object modules” into a single program
    - Creates a “load module”
    - Load module loaded by a “loader”
- Functions of a linker
    - Resolve addresses that are external to a module
    - Relocate modules to merge them
    - Determine starting symbol of a load module
    - Link memory segments if multiple segments defined

Loader

- Most programs can be dynamically loaded when ready for execution
  - Absolute memory addresses not known until loaded
  - Addresses are “relocatable” in memory, except for I/O addresses and other special locations
- Loader functions
  - Locate program – map relative relocatable addresses to absolute addresses or initializing a base register
  - Load program module into memory
  - Initialize registers

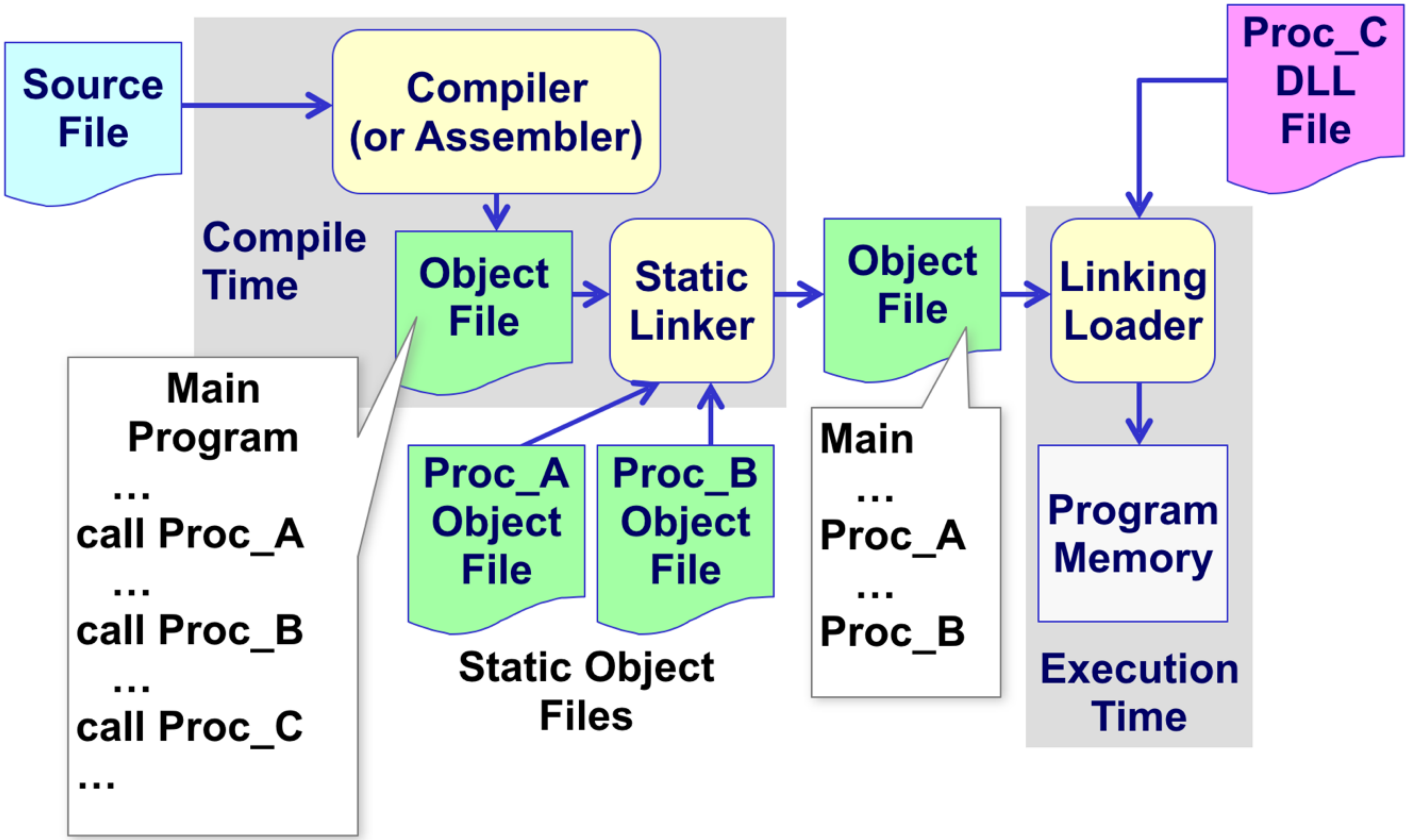
Static Versus Dynamic Linking



Loader

- Traditional static linkers link object modules at the time of compilation
  - Object files are linked to create the executable
  - Executable has binary code for main program and libraries … so it is large
  - Library code fixed at compile time … so changing the library requires re-linking the program
- Dynamic link libraries allow component to be linked at run time
  - Library code that is available at run time (DLL file) is linked with the program

Dynamic Linking



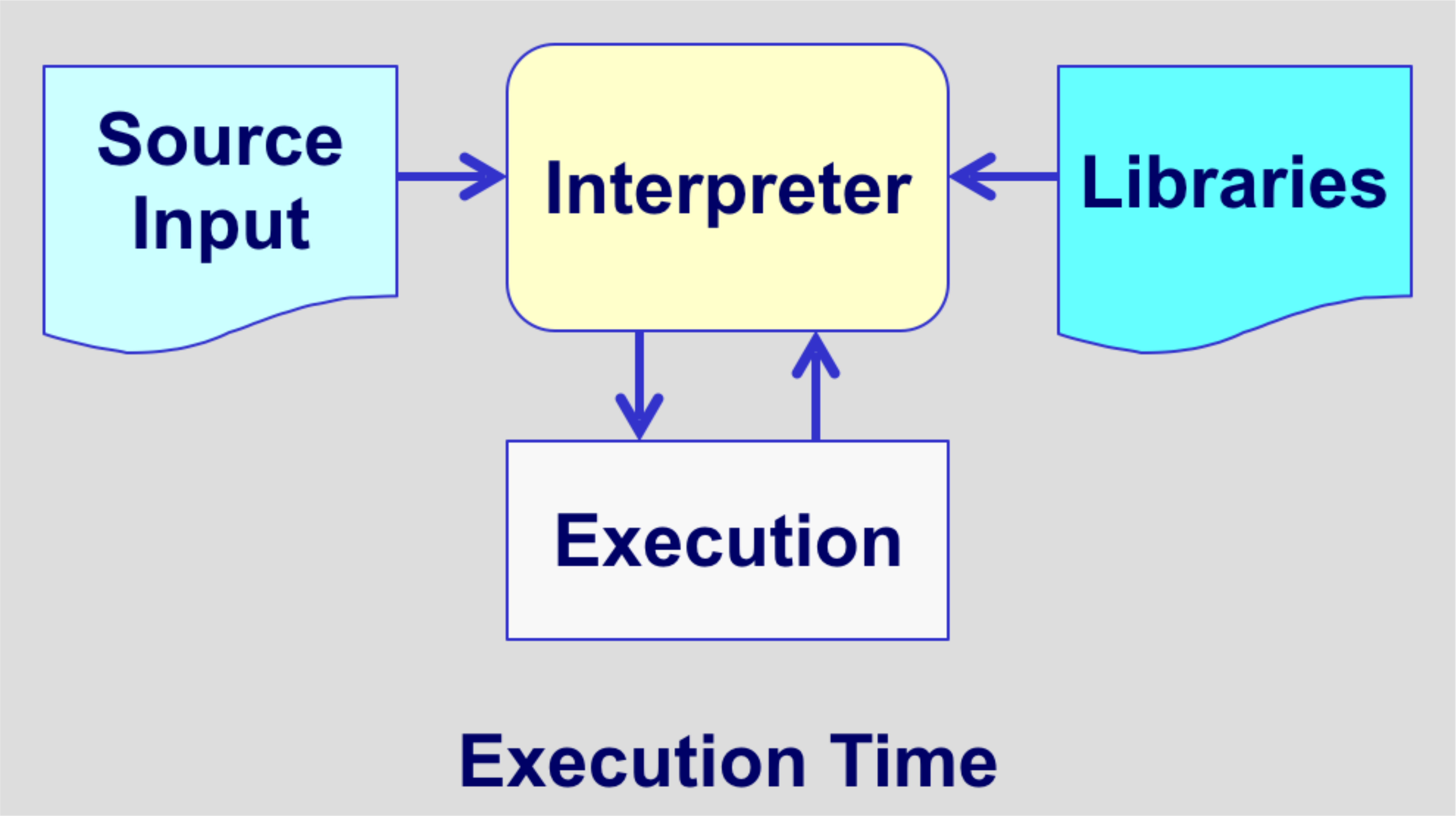
Interpreters

- Interpreted languages, like compiled languages, typically represent algorithmic or functional abstractions
- Unlike compiled languages, interpreted language statements are processed and then executed immediately
- Interpreters often are integrated with an interactive environment for command input, graphical output, etc.
- Tend to be slow, but are easy to use and allow easy debugging (immediate results)

Interpreters (2)

- Examples
  - BASIC
  - PYTHON
  - R
- Variations
  - Some languages can be interpreted (for debugging) or compiled (for efficiency)
  - Just-in-time (JIT) compilers are a compromise

Interpreters (3)



CHECK POINT

As a checkpoint of your understanding, please pause the video and make sure you can do the following:

- Describe the need for linking and loading
- Differentiate between static linking and dynamic linking at execution time
- Describe the basic function of an interpreter and differentiate it from a compiler

If you have any difficulties, please review the lecture video before continuing.

Summary

- Assemblers convert assembly language programs into binary machine or object code
  - Relatively simple one-to-one translation
- Compilers convert high-level language programs into assembly language or binary machine or object code
  - Relatively complex one-to-many or many-to-one translation
- Six steps for compilation: (1) lexical analysis; (2) syntactical analysis or parsing; (3) semantic analysis; (4) intermediate code generation; (5) code optimization; and (6) code generation

Summary (2)

- Linkers link multiple object files to create a binary file that is ready to load and execute
- A loader locates a program in memory for execution
- Linking may be…
    - Static – linker creates a static executable file
    - Dynamic – linking loader can add dynamic link library code execution time
- Interpreters process (“compile”) and execute statements in an integrated and interactive manner

MODULE 9: Software Systems and Tools

Lecture 9.3
Programming Tools

Prepared By:

- Scott F. Midkiff, PhD
- Luiz A. DaSilva, PhD
- Kendall E. Giles, PhD

Electrical and Computer Engineering
Virginia Tech