603AA - Principles of Programming Languages [PLP-2014]

Andrea Corradini
Department of Computer Science, Pisa
Academic Year 2014/15

Admins

- **http://www.di.unipi.it/~andrea/Didattica/PLP-14/**
- 9 CFU/ECTS (3 + 6)
- Replaces previous PLP of 12 CFU [379AA]
- Students enrolled till AY 2013/14 have to integrate the course with a 3 CFU activity
  - To be agreed upon with me
- Office Hours?
- Please, fill in the sheet with required info

Evaluation

- 2 midterms
  - December 18, 2014, at 16:00
  - March or May 2014
- Written proof
- Oral examination
- Homeworks? Project? Seminars?

Course Objectives

- Understand the significance of the design of a programming language and its implementation in a compiler or interpreter
- Enhance the ability to learn new programming languages
- Understand how programs are parsed and translated by a compiler
- Be able to define LL(1), LR(1), and LALR(1) grammars
- Know how to use compiler construction tools, such as generators of scanners and parsers
- Be able, in principle, to implement significant parts of a compiler
- Improve the understanding of general programming concepts and the ability to choose among alternative ways to express things in a particular programming language
- Simulate useful features in languages that lack them
- …

Course Outline (temptative)

- **Abstract Machines and their Languages**
- **Interpretation and Compilation**
- **Structure of a Compiler**
  - **Lexical Analysis and Lex/Flex**
  - **Syntax Analysis and Yacc**
  - **Syntax-Directed Translation**
  - **Static Semantics and Type Checking**
  - **Intermediate Code Generation**
- Programming language concepts and their semantics
  - Names, scopes and bindings
  - Control flow
  - Data types
  - Control abstraction
  - Data abstraction
- Programming paradigms
  - Logic programming
  - Scripting languages
  - Functional programming
  - Object-Oriented programming

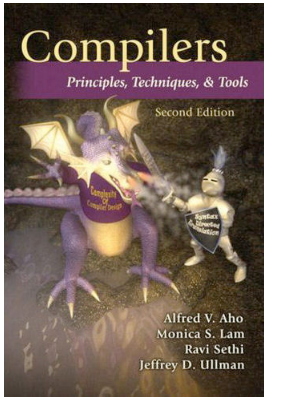
Textbooks

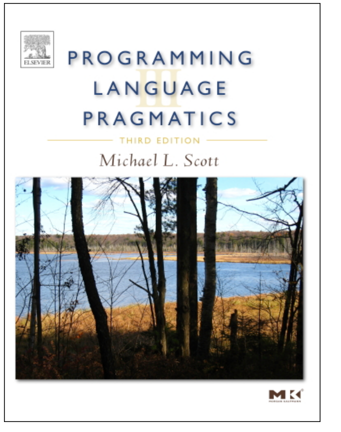
- **[Scott] Programming Language Pragmatics**
  by Michael L. Scott, 3rd edition

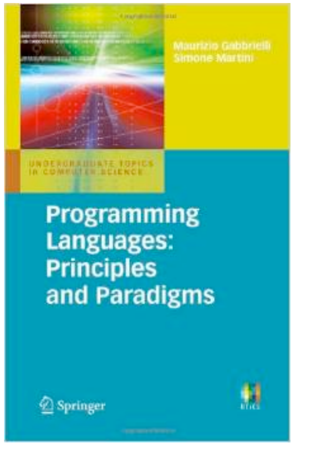
- **[ALSU] Compilers: Principles, Techniques, and Tools**
  by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, 2nd edition

- **[GM] Programming Languages: Principles and Paradigms**
  by Maurizio Gabbrielli and Simone Martini

- + other references







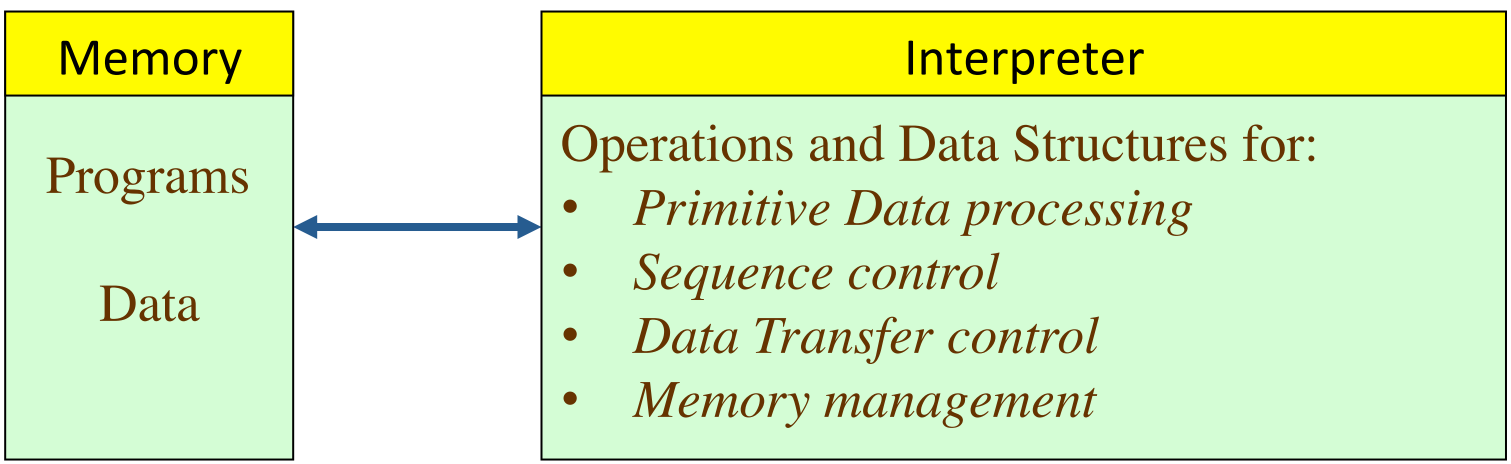
Credits

- Slides freely taken and elaborated from a number of sources:
  - Marco Bellia (DIP)
  - Gianluigi Ferrari (DIP)
  - Robert A. van Engelen (Florida State University)
  - Gholamreza Ghassem-Sani (Sharif University of Technology)

Abstract Machines

Abstract Machine for a Language **L**

- Given a programming language **L**, an **Abstract Machine M_L for L** is *a collection of data structures and algorithms which can perform the storage and execution of programs written in **L***
- An abstraction of the concept of hardware machine
- Structure of an abstract machine:



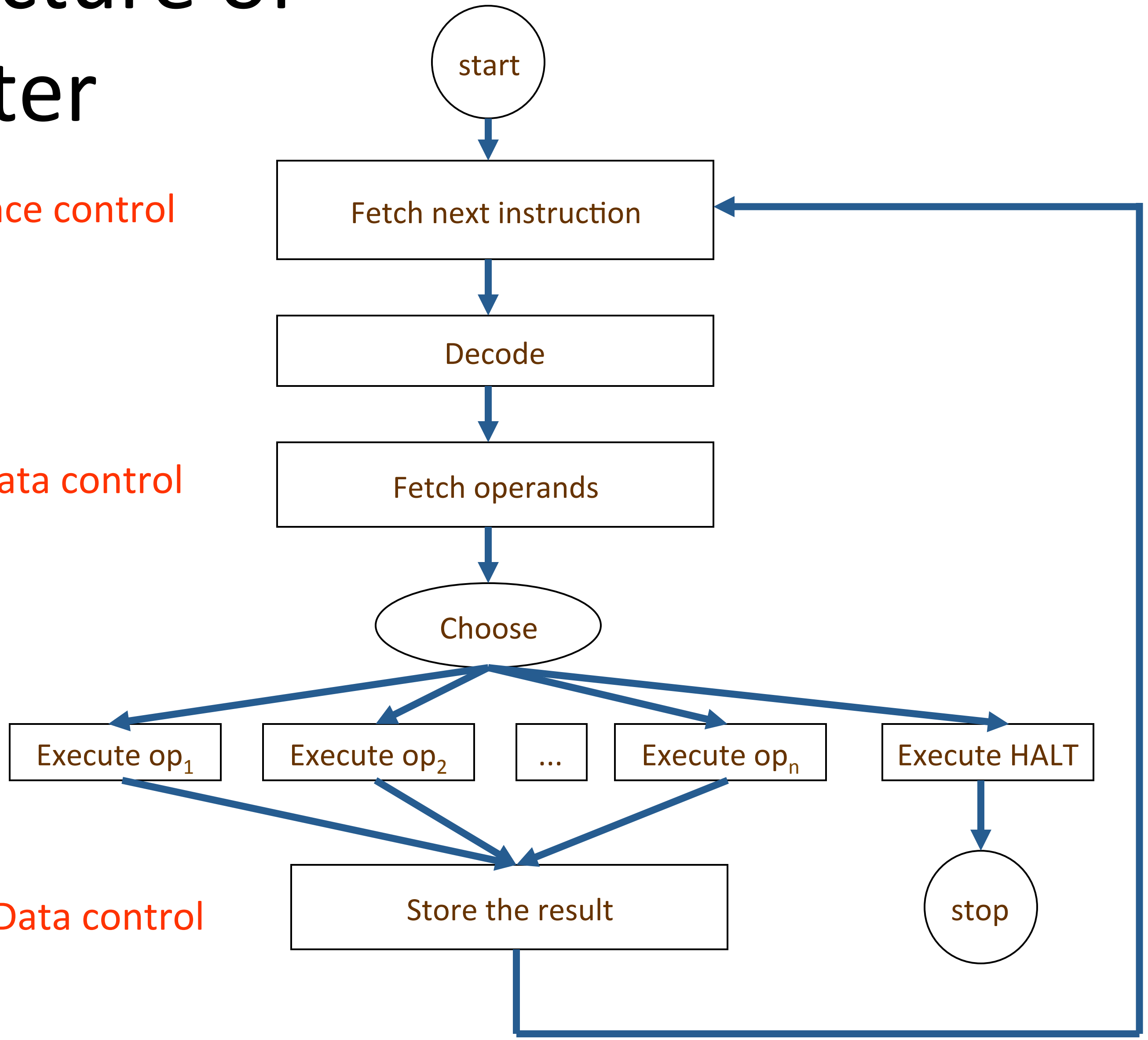
General structure of the Interpreter

Sequence control

Data control

Operations

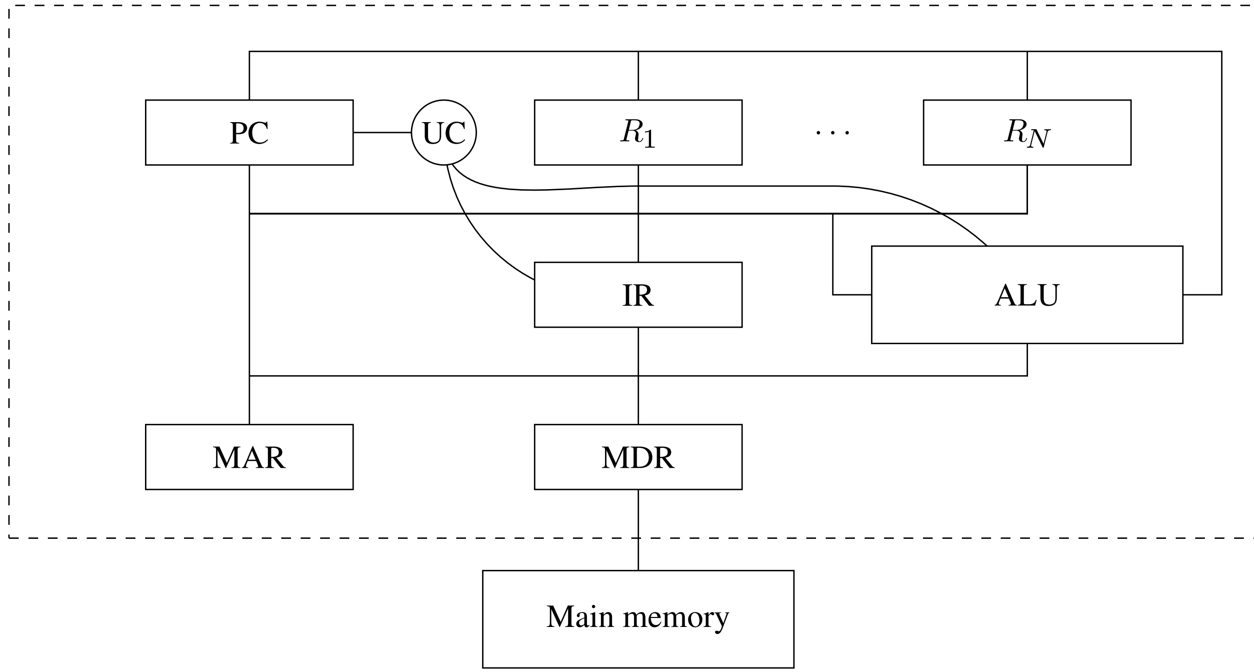
Data control



The Machine Language of an AM

- Given and Abstract machine **M**, the machine language $\mathbf{L_M}$ of **M**
  - includes all programs which can be executed by the interpreter of M
- Programs are particular data on which the interpreter can act
- The components of **M** correspond to components of $\mathbf{L_M}$, eg:
  - Primitive data types
  - Control structures
  - Parameter passing and value return
  - Memory management
- Every Abstract Machine has a unique Machine Language
- A programming language can have several Abstact Machines

An example: the Hardware Machine



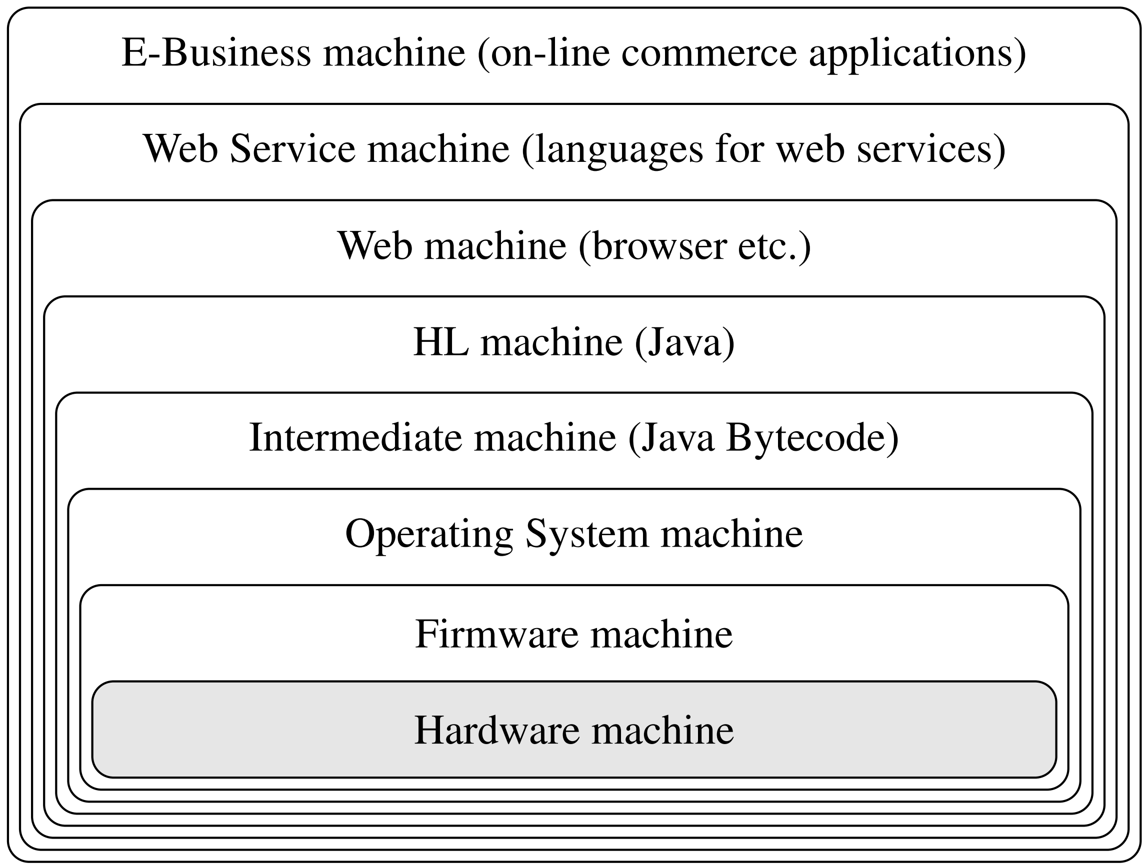
- The language?
- The memory?
- The interpreter?
- Operations and Data Structures for:
    - Primitive Data processing?
    - Sequence control?
    - Data Transfer control?
    - Memory management?

Implementing an Abstract Machine

- Each abstract machine can be implemented in **hardware** or in **firmware**, but if it is high-level this is not convenient in general
- An abstract machine **M** can be implemented over a **host machine M_O**, which we assume is already implemented
- The components of **M** are realized using data structures and algorithms implemented in the machine language of M_O
- Two main cases:
  - The interpreter of **M** coincides with the interpreter of M_O
    - **M** is an **extension** of M_O
    - other components of the machines can differ
  - The interpreter of **M** is different from the interpreter of M_O
    - **M** is **interpreted** over M_O
    - other components of the machines may coincide

Hierarchies of Abstract Machines

- Implementation of an AM with another can be iterated, leading to a hierarchy (onion skin model)
- Example:



Implementing a Programming Language

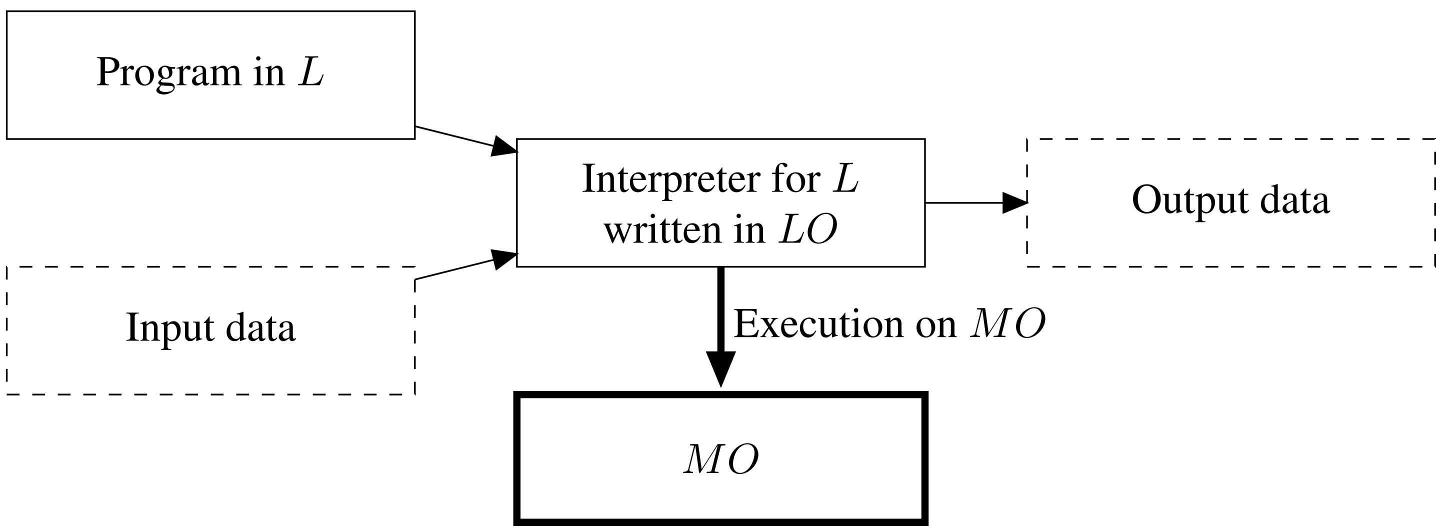
- **L** high level programming language
- **M_L** abstract machine for **L**
- **M_O** host machine
- **Pure Interpretation**
  - **M_L** is interpreted over **M_O**
  - Not very efficient, mainly because of the interpreter (fetch-decode phases)
- **Pure Compilation**
  - Programs written in **L** are translated into equivalent programs written in **L_O**, the machine language of **M_O**
  - The translated programs can be executed directly on **M_O**
    - **M_L** is not realized at all
  - Execution more efficient, but the produced code is larger
- Two limit cases that almost never exist in reality

Pure Interpretation

- Program ***P*** in ***L*** as a partial function on ***D***:

  $$\mathscr{P}^{\mathscr{L}} : \mathscr{D} \rightarrow \mathscr{D}$$

- Set of programs in **L**: $\mathscr{P}rog^{\mathscr{L}}$



- The interpreter defines a function

$$\mathscr{I}_{\mathscr{L}}^{\mathscr{L}o} : (\mathscr{P}rog^{\mathscr{L}} \times \mathscr{D}) \rightarrow \mathscr{D} \quad \text{such that } \mathscr{I}_{\mathscr{L}}^{\mathscr{L}o}(\mathscr{P}^{\mathscr{L}}, Input) = \mathscr{P}^{\mathscr{L}}(Input)$$

Pure [*cross*] Compilation

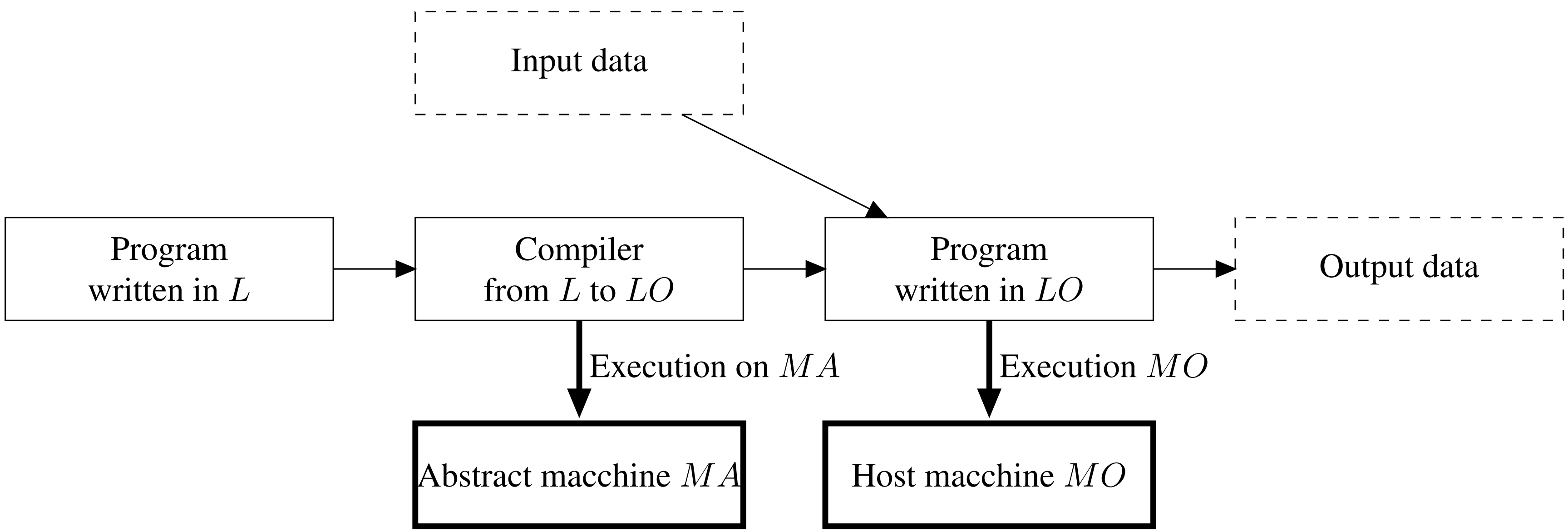
A compiler from ***L*** to ***LO*** defines a function

$$\mathscr{C}_{\mathscr{L},\mathscr{L}o} : \mathscr{P}rog^{\mathscr{L}} \to \mathscr{P}rog^{\mathscr{L}o}$$

such that if

$$\mathscr{C}_{\mathscr{L},\mathscr{L}o}(\mathscr{P}^{\mathscr{L}}) = \mathscr{P}c^{\mathscr{L}o},$$

then for every *Input* we have $\mathscr{P}^{\mathscr{L}}(Input) = \mathscr{P}c^{\mathscr{L}o}(Input)$



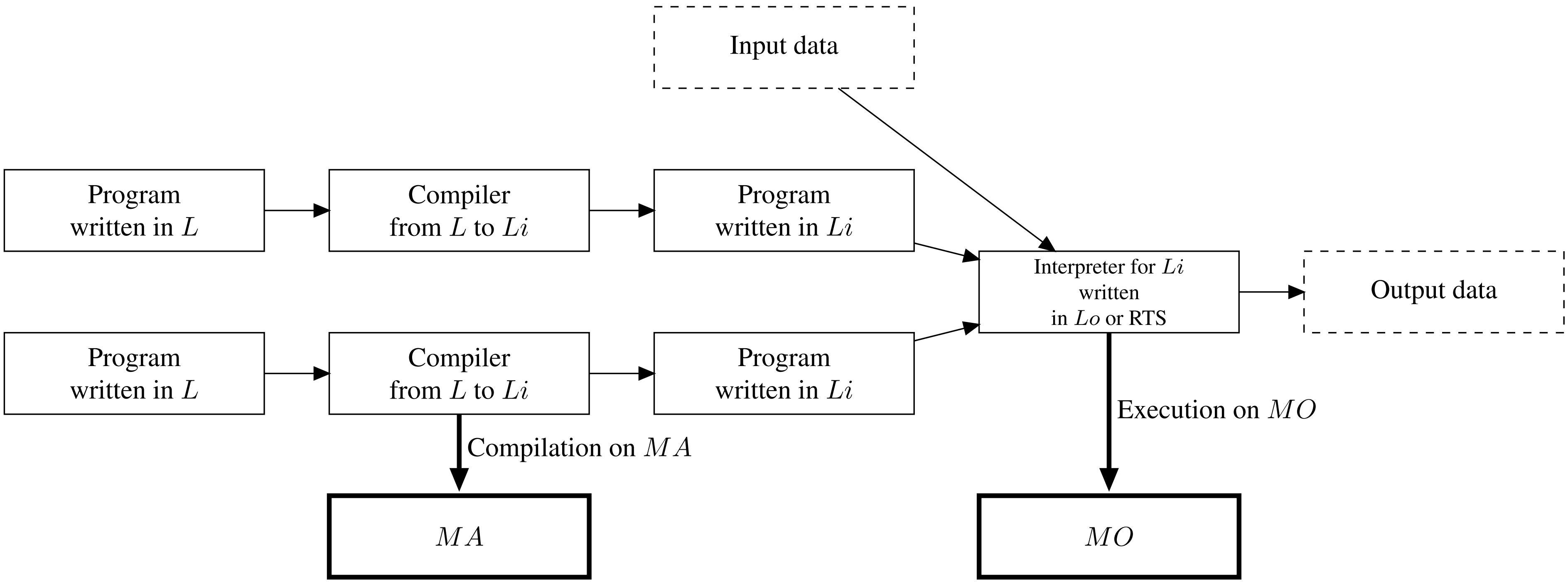
Compilers versus Interpreters

- Compilers efficiently fix decisions that can be taken at compile time to avoid to generate code that makes this decision at run time
  - Type checking at compile time vs. runtime
  - Static allocation
  - Static linking
  - Code optimization
- Compilation leads to better performance in general
  - Allocation of variables without variable lookup at run time
  - Aggressive code optimization to exploit hardware features
- Interpretation facilitates interactive debugging and testing
  - Interpretation leads to better diagnostics of a programming problem
  - Procedures can be invoked from command line by a user
  - Variable values can be inspected and modified by a user

Compilation + Interpretation

- All implementations of programming languages use both. At least:
  - Compilation (= translation) from external to internal representation
  - Interpretation for I/O operations (runtime support)
- Can be modeled by identifying an *Intermediate Abstract Machine* $\boldsymbol{M_I}$ *with language* $\boldsymbol{L_I}$
  - A program in $\boldsymbol{L}$ is compiled to a program in $\boldsymbol{L_I}$
  - The program in $\boldsymbol{L_I}$ is executed by an interpreter for $\boldsymbol{M_I}$

Compilation + Interpretation with Intermediate Abstract Machine



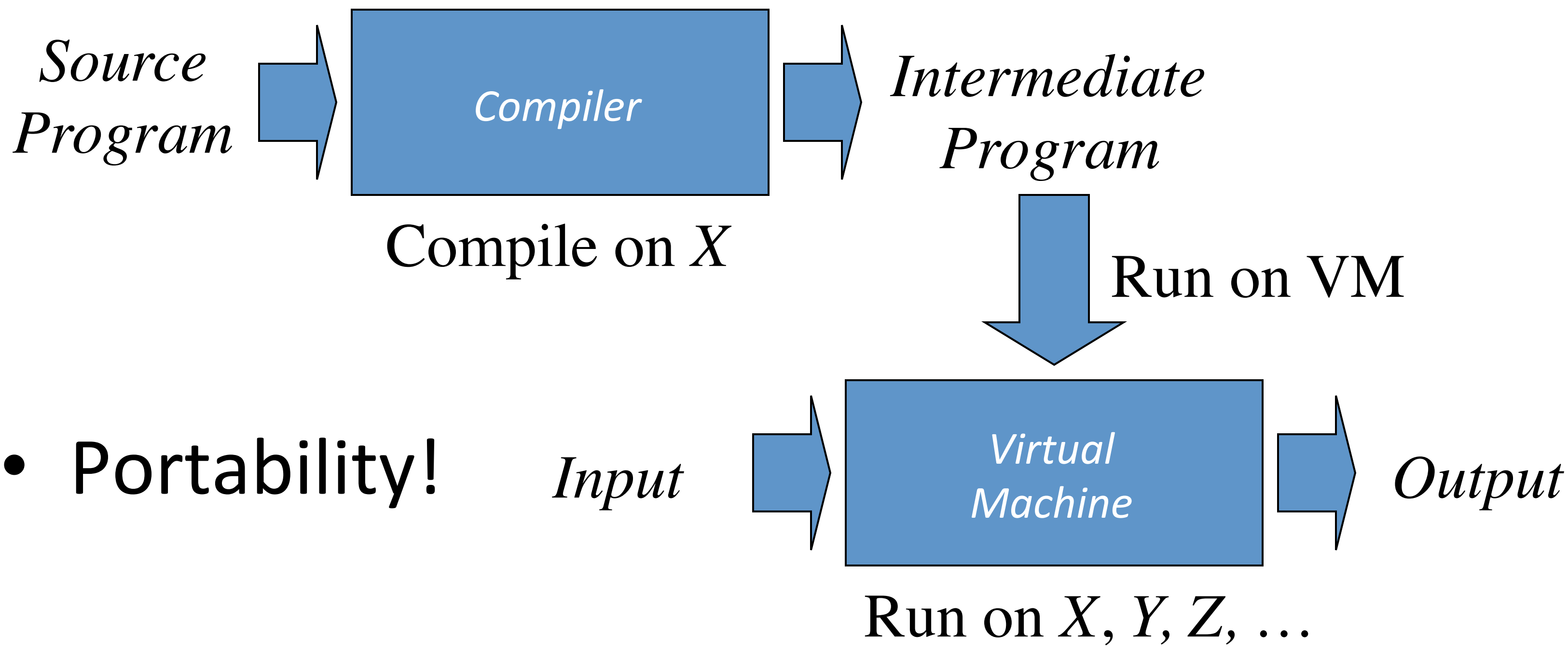
- The “pure” schemes as limit cases
- Let us sketch some typical implementation schemes…

Virtual Machines as Intermediate Abstract Machines

- Several language implementations adopt a compilation + interpretation schema, where the Intermediate Abstract Machine is called Virtual Machine
- Adopted by Pascal, Java, Smalltalk-80, C#, functional and logic languages, and some scripting languages
  - Pascal compilers generate P-code that can be interpreted or compiled into object code
  - Java compilers generate bytecode that is interpreted by the Java virtual machine (JVM)
  - The JVM may translate bytecode into machine code by just-in-time (JIT) compilation

Compilation and Execution on Virtual Machines

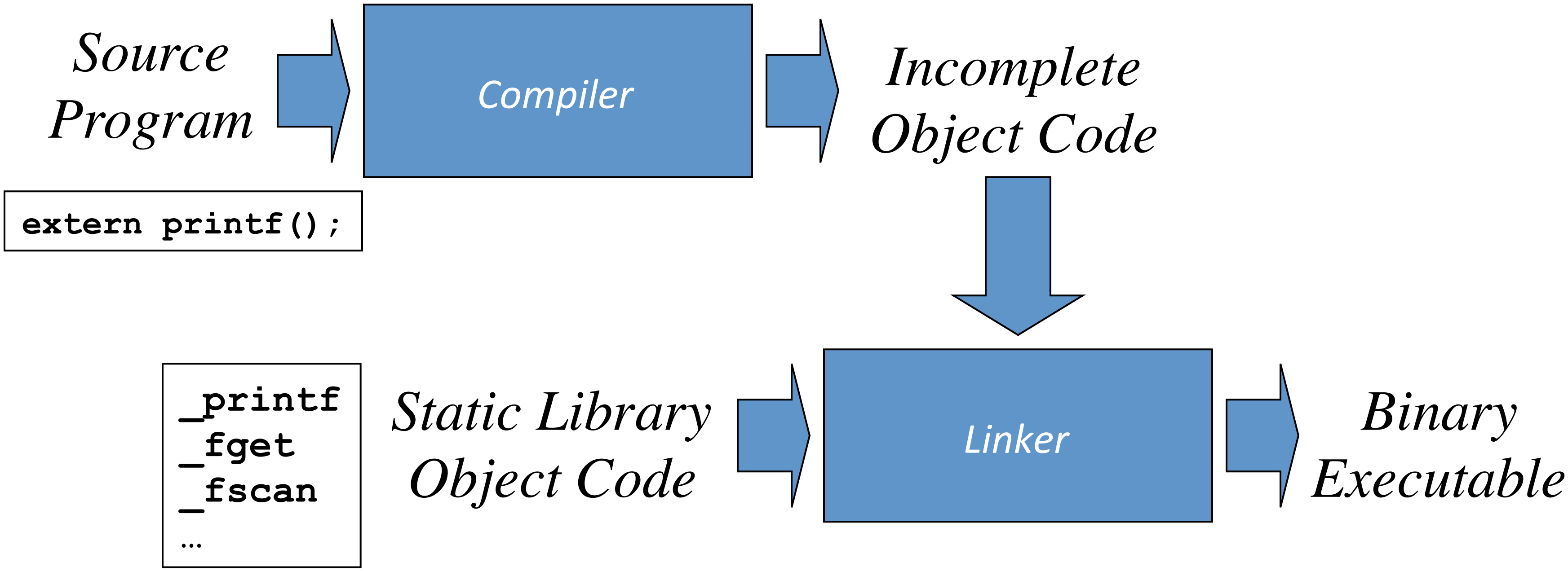
- Compiler generates intermediate program
- Virtual machine interprets the intermediate program



- Portability!

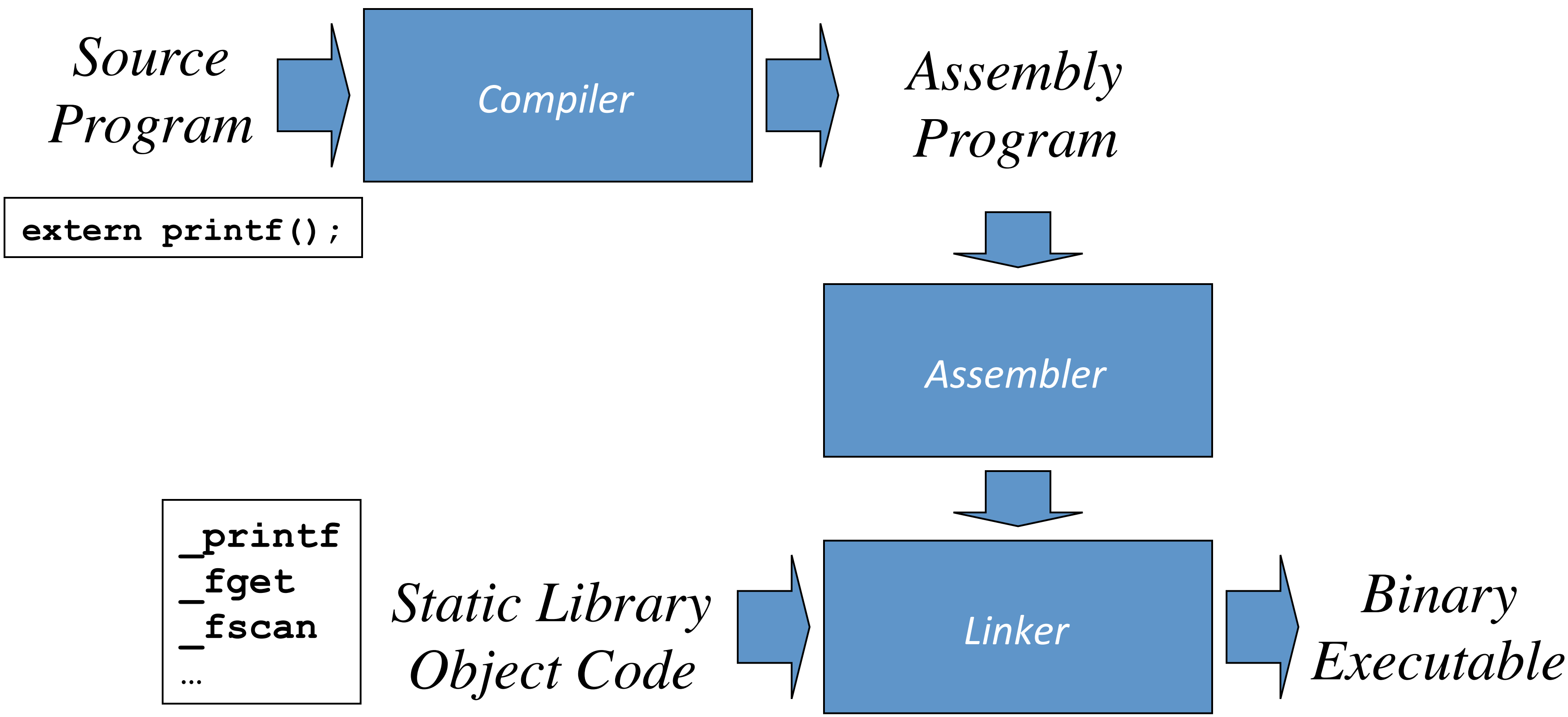
Pure Compilation and Static Linking

- Adopted by the typical Fortran systems
- Library routines are separately linked (merged) with the object code of the program



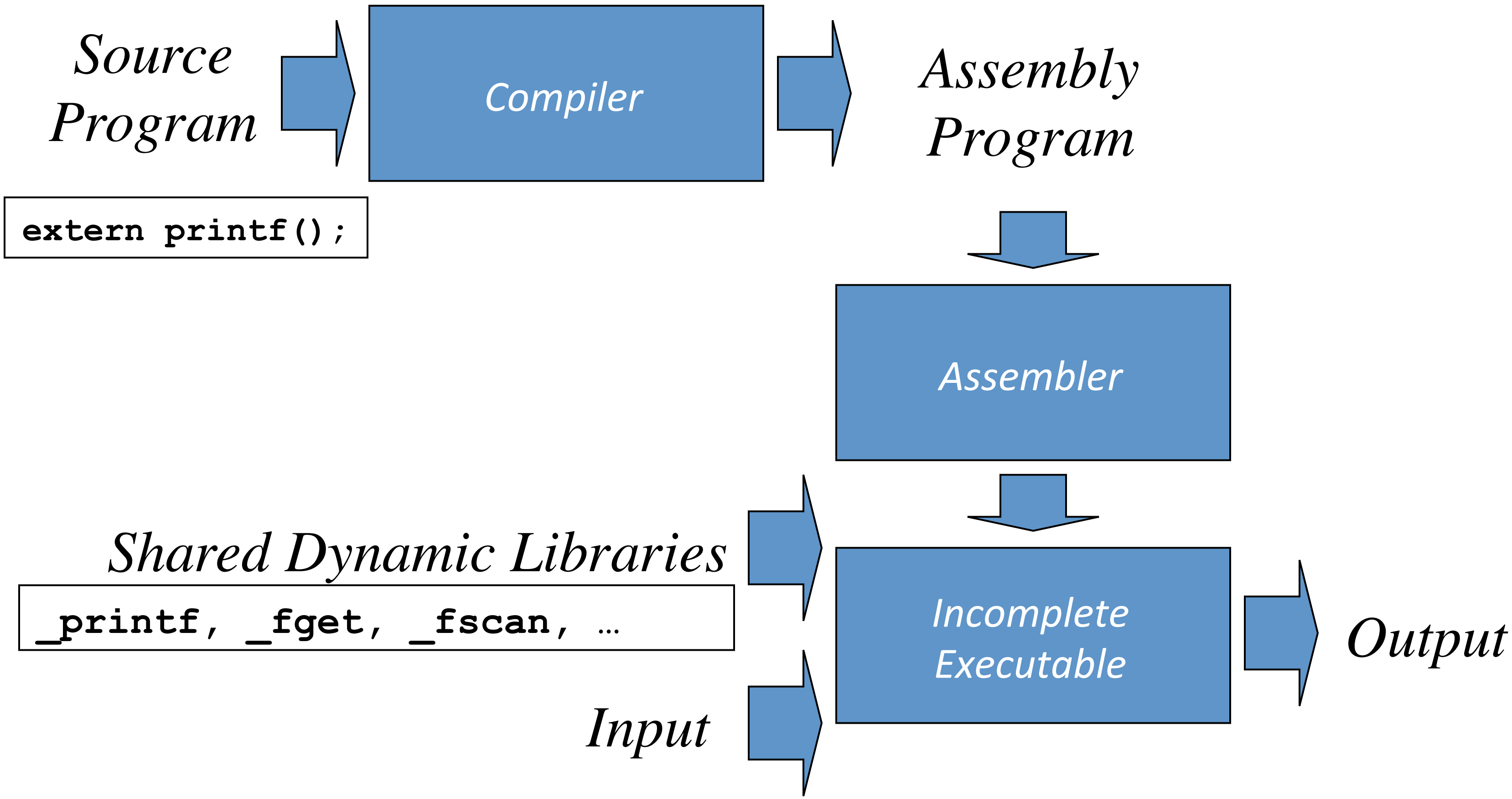
Compilation, Assembly, and Static Linking

- Facilitates debugging of the compiler



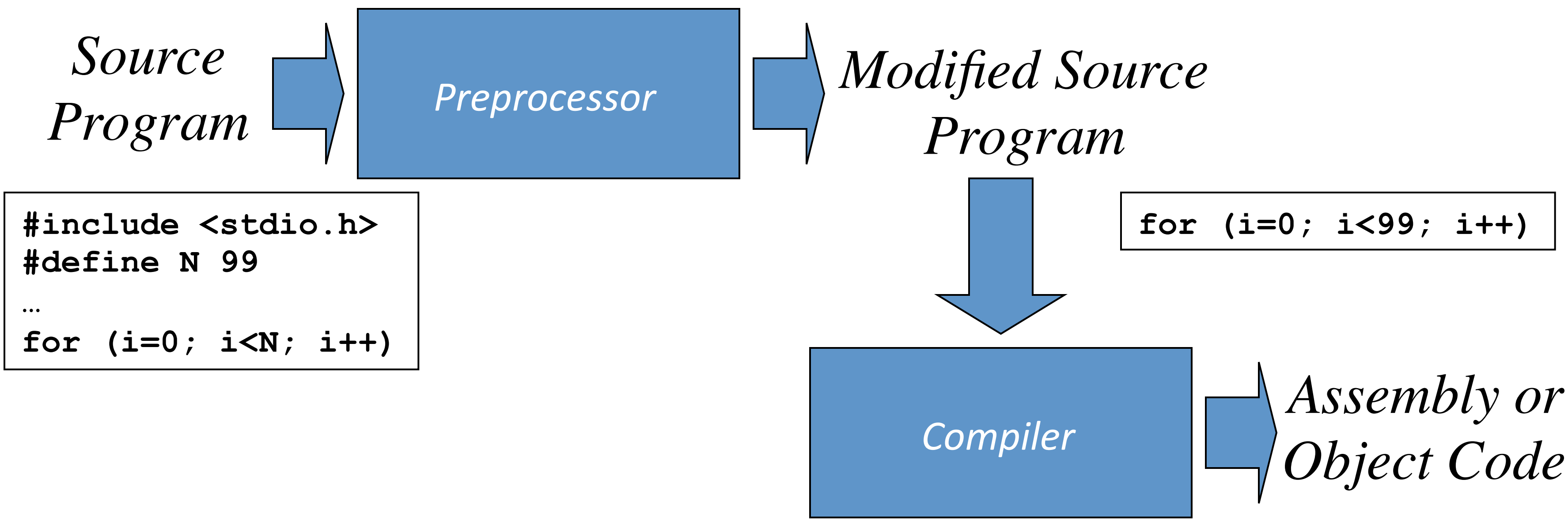
Compilation, Assembly, and Dynamic Linking

- Dynamic libraries (DLL, .so, .dylib) are linked at run-time by the OS (via stubs in the executable)



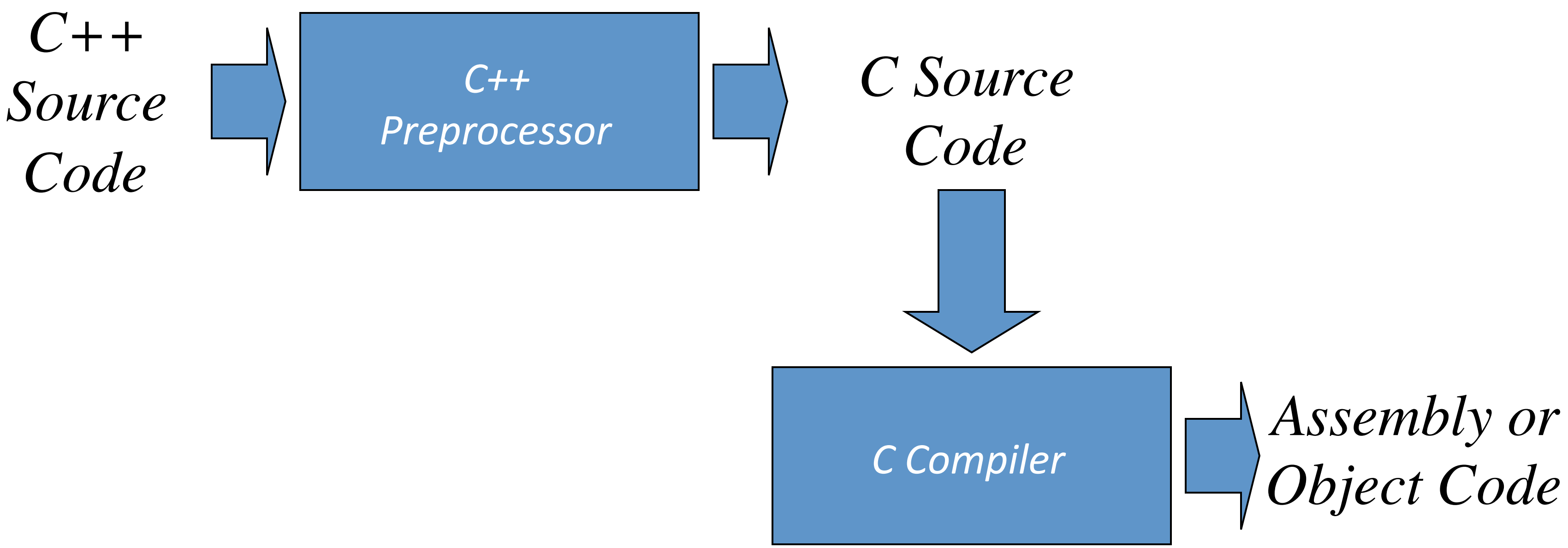
Preprocessing

- Most C and C++ compilers use a preprocessor to import header files and expand macros



The CPP Preprocessor

- Early C++ compilers used the CPP preprocessor to generated C code for compilation



Compilers

The Analysis-Synthesis Model of Compilation

- Compilers translate programs written in a language into equivalent programs in another language
- There are two parts to compilation:
  - **Analysis** determines the operations implied by the source program which are recorded in a tree structure
  - **Synthesis** takes the tree structure and translates the operations therein into the target program

Other Tools that Use the Analysis-Synthesis Model

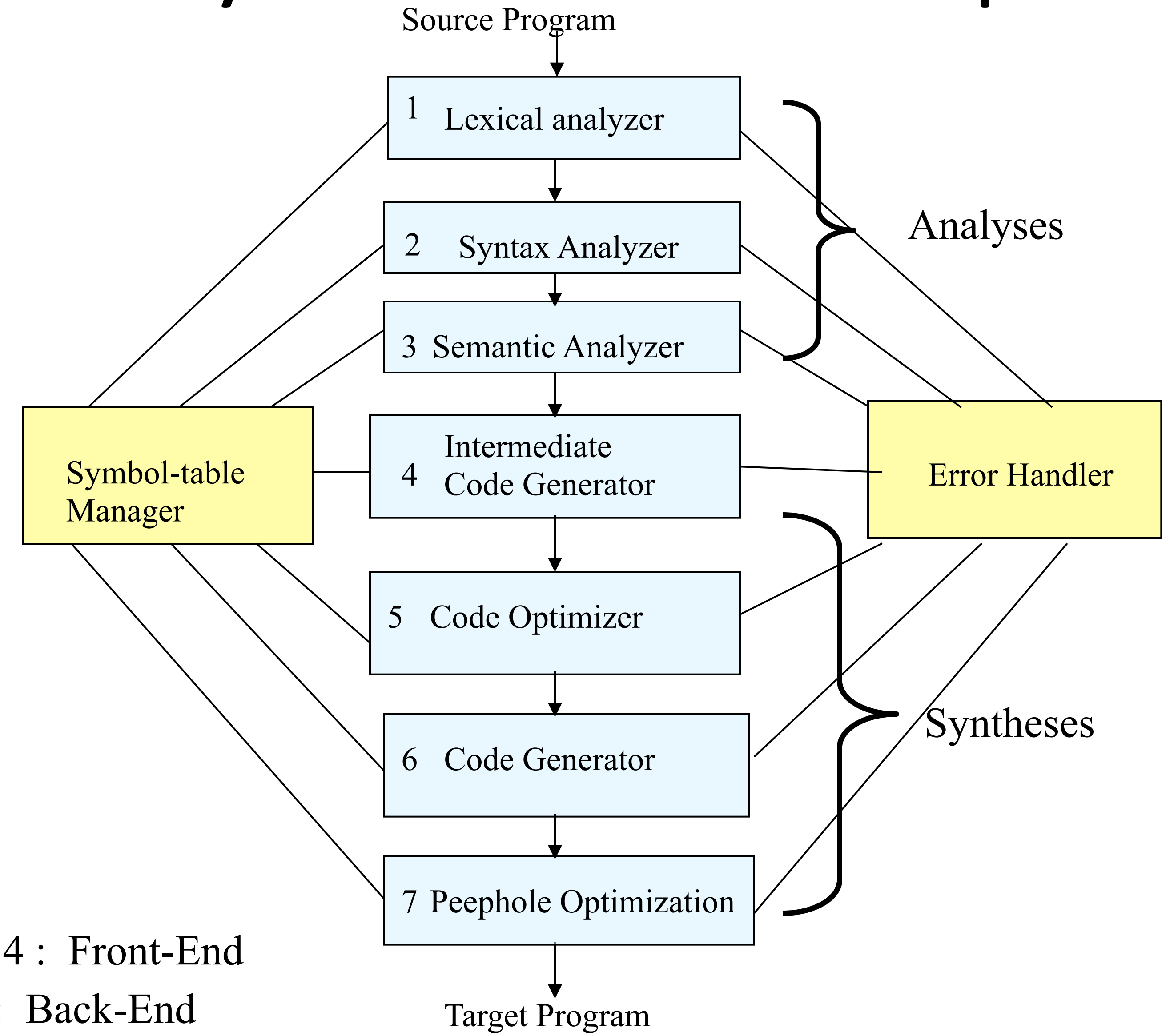
- Editors (syntax highlighting)
- Pretty printers (e.g. Doxygen)
- Static checkers (e.g. Lint and Splint)
- Interpreters
- Text formatters (e.g. TeX and LaTeX)
- Silicon compilers (e.g. VHDL)
- Query interpreters/compilers (Databases)

Several compilation techniques are used in other kinds of systems

Compilation Phases and Passes

- Compilation of a program proceeds through a fixed series of phases
- A **pass** is one phase or a sequence of phases that starts from a representation of the program and produces another representation of it
- Passes can be serialized, phases not necessarily
  - Pascal, FORTRAN, C languages designed for one-pass compilation, which explains the need for function prototypes
  - Single-pass compilers need less memory to operate
  - Java and ADA are multi-pass

The Many Phases of a Compiler



1, 2, 3, 4 : Front-End

5, 6, 7 : Back-End