

301AA - Advanced Programming

Lecturer: **Andrea Corradini**

andrea@di.unipi.it

<http://pages.di.unipi.it/corradini/>

Course pages:

<http://pages.di.unipi.it/corradini/Didattica/AP-18/>

AP-03: *Languages and Abstract machines,
Compilation and interpretation schemes*

Outline

- Programming languages and abstract machines
- Implementation of programming languages
- Compilation and interpretation
- Intermediate virtual machines

Definition of Programming Languages

- A PL is defined via **syntax**, **semantics** and **pragmatics**
- The **syntax** is concerned with the form of programs: how expressions, commands, declarations, and other constructs must be arranged to make a well-formed program.
- The **semantics** is concerned with the meaning of (well-formed) programs: how a program may be expected to behave when executed on a computer.
- The **pragmatics** is concerned with the way in which the PL is intended to be used in practice.

Syntax

- Formally defined, but not always easy to find
 - Java: Chapter 19 of Java Language Specification v8
<https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf>
- Lexical Grammar for tokens
 - A *regular* grammar
- Syntactic Grammar for language constructs
 - A *context free* grammar
- Used by the compiler for *scanning* and *parsing*

Semantics

- Usually described precisely, but informally, in natural language.
 - May leave (subtle) ambiguities
- Formal approaches exist, often they are applied to toy languages or to fractions of real languages
 - Denotational [Scott and Strachey 1971]
 - Operational [Plotkin 1981]
 - Axiomatic [Hoare 1969]
- They rarely scale to fully-fledged programming language

(Almost) Complete Semantics of PLs

- Notable exceptions exist:
 - **Pascal** (part), Hoare Logic [C.A.R. Hoare and N. Wirth, ~1970]
 - **Standard ML**, Natural semantics [R. Milner, M. Tofte and R. Harper, ~1990]
 - **C**, Evolving algebras [Y. Gurevich and J. Huggins, 1993]
 - **Java and JVM**, Abstract State Machines [R. Stärk, J. Schmid, E. Börger, 2001]
 - Executable formal semantics using the **K framework** of several languages (C, Java, JavaScript, PHP, Python, Rust,...)

<https://runtimeverification.com/blog/k-framework-an-overview/>

Pragmatics

- Includes coding conventions, guidelines for elegant structuring of code, etc.
- Examples:
 - Java Code Conventions
www.oracle.com/technetwork/java/codeconventions-150003.pdf
 - Google Java Style Guide
<https://google.github.io/styleguide/javaguide.html>
- Also includes the description of the supported *programming paradigms*

Programming Paradigms

A **paradigm** is a style of programming, characterized by a particular selection of key concepts and abstractions

- **Imperative programming:** variables, commands, procedures, ...
- **Object-oriented (OO) programming:** objects, methods, classes, ...
- **Concurrent programming:** processes, communication..
- **Functional programming:** values, expressions, functions, ...
- **Logic programming:** assertions, relations, ...

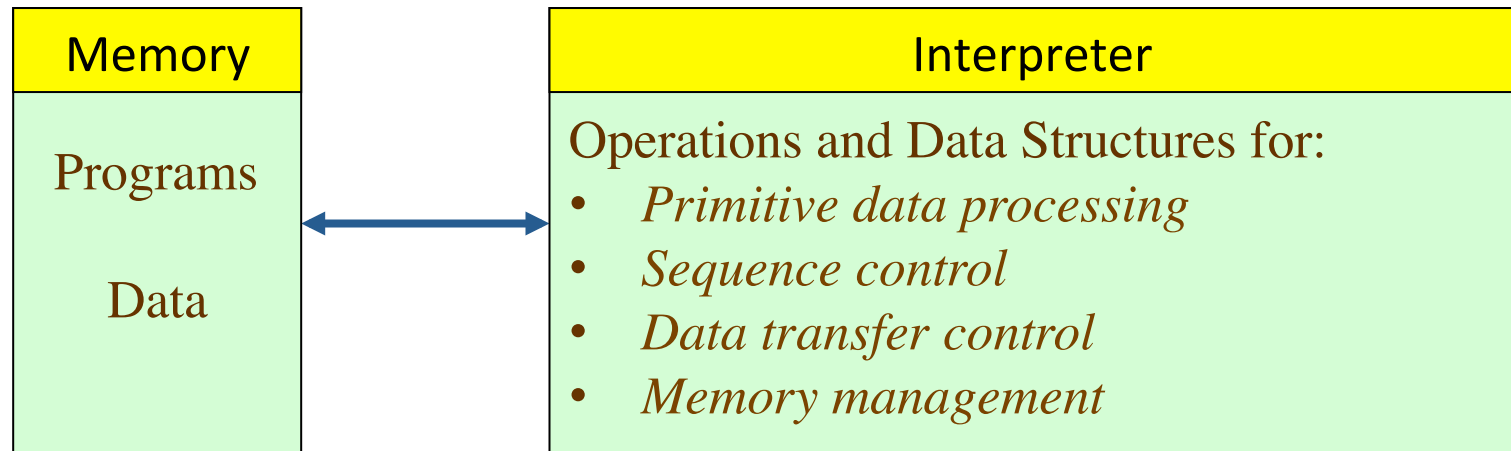
Classification of languages according to paradigms can be misleading

Implementation of a Programming Language L

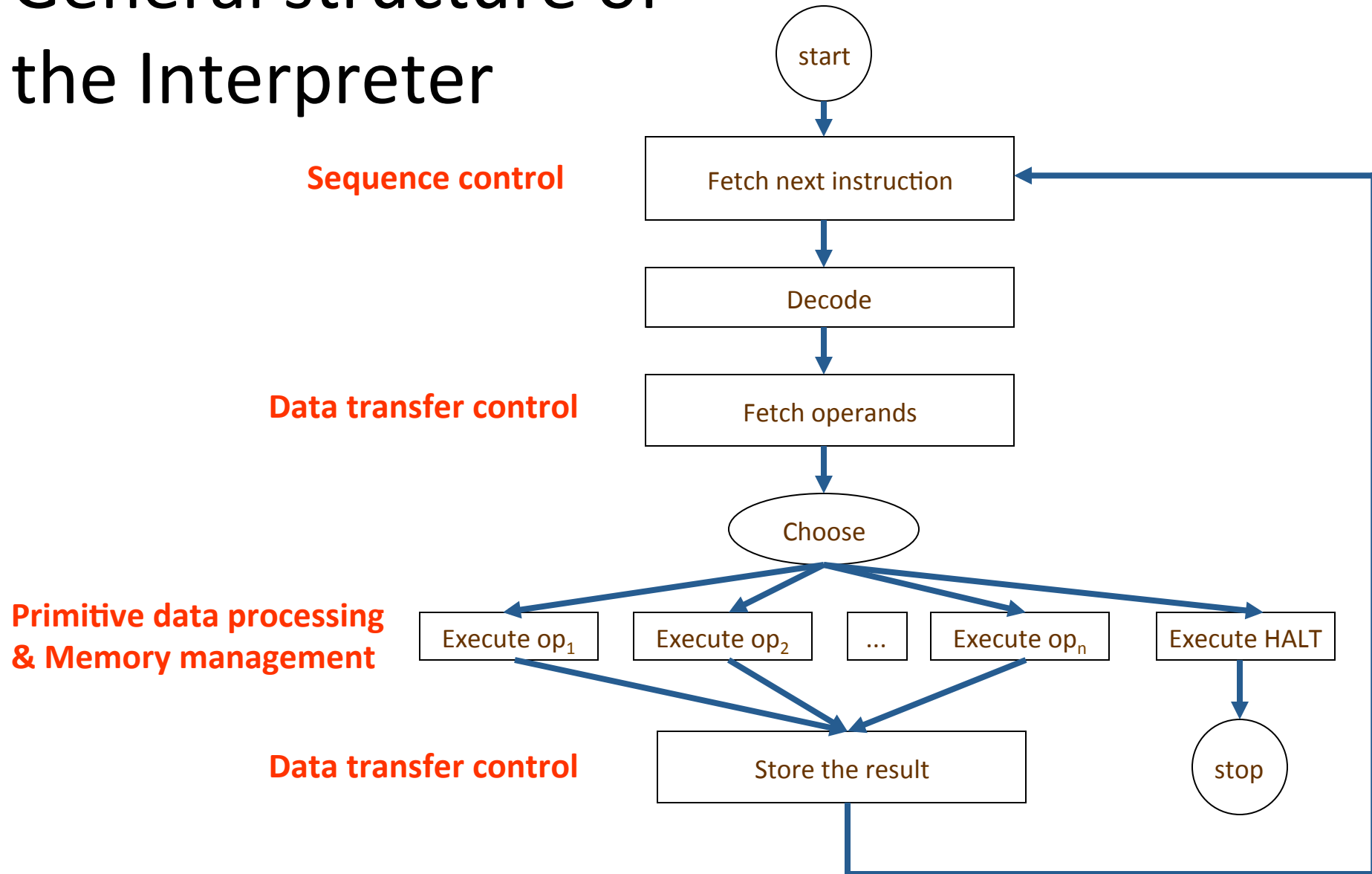
- Programs written in L must be executable
- Every language L implicitly defines an ***Abstract Machine M_L*** having L as machine language
- Implementing M_L on an existing host machine M_O (via ***compilation, interpretation*** or both) makes programs written in L executable

Programming Languages and Abstract Machines

- 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:



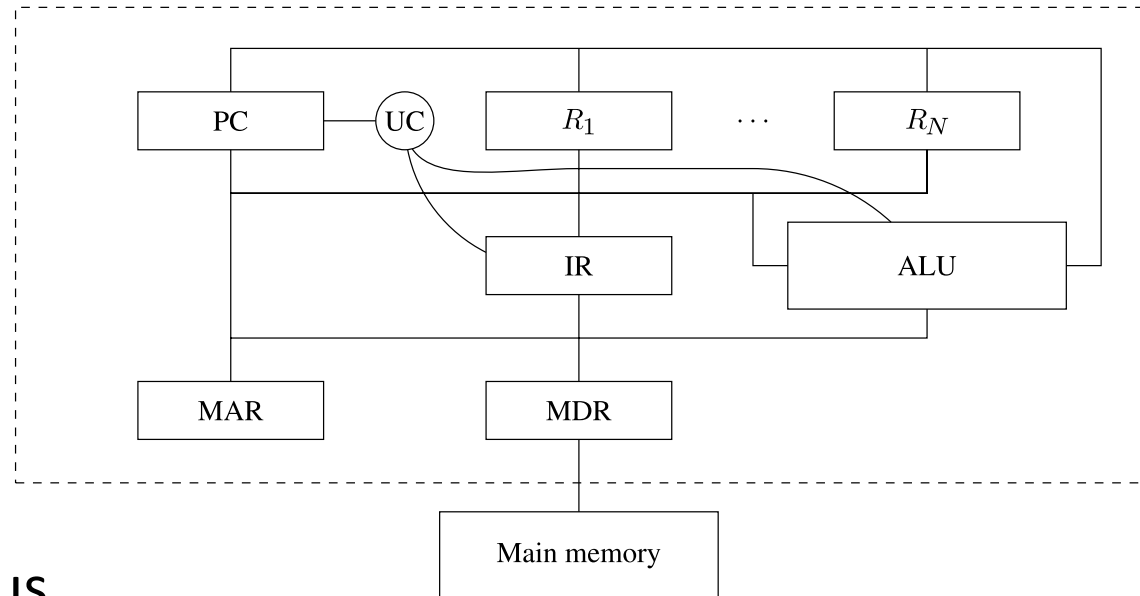
General structure of the Interpreter



The Machine Language of an AM

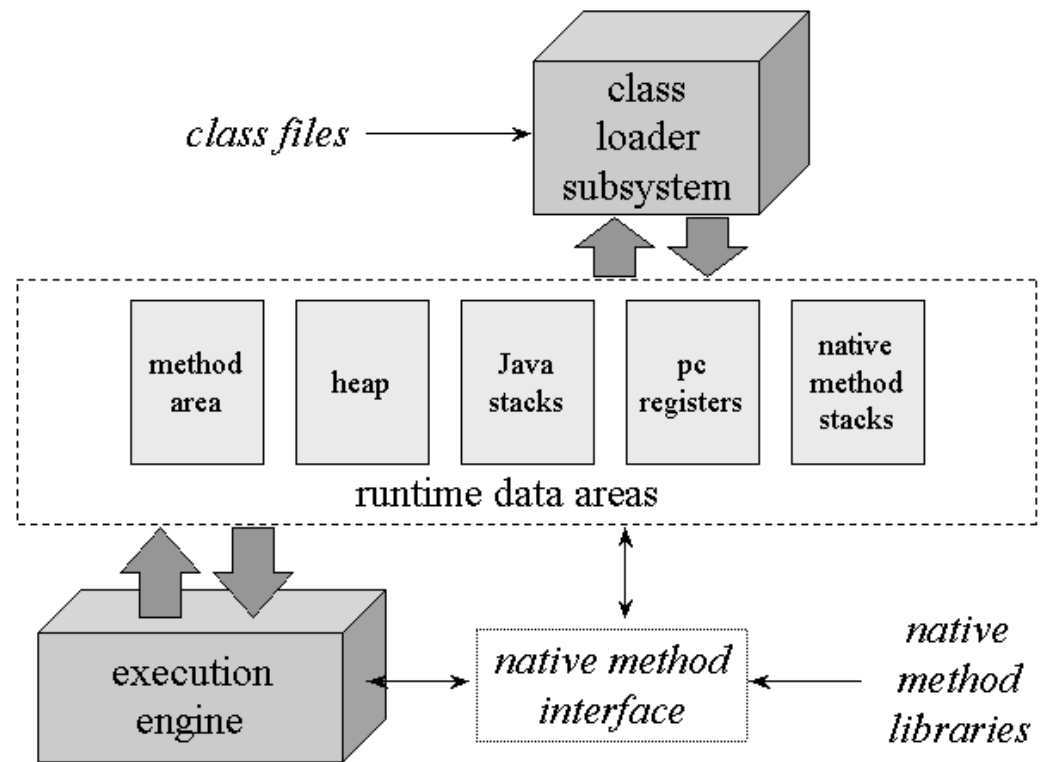
- Viceversa, each Abstract machine **M** defines a language L_M including all programs which can be executed by the interpreter of **M**
- Programs are particular data on which the interpreter can act
- The components of **M** correspond to components of L_M , eg:
 - Primitive data types
 - Control structures
 - Parameter passing and value return
 - Memory management

An example: the Hardware Machine



- Language: obvious
- Memory: Registers + RAM (+ cache)
- Interpreter: fetch, decode, execute loop
- Operations and Data Structures for:
 - Primitive data processing
 - Sequence control
 - Data transfer control
 - Memory management

The Java Virtual Machine



- Language: bytecode
- Memory Heap+Stack+Permanent
- Interpreter
- Operations and Data Structures for:
 - Primitive data processing
 - Sequence control
 - Data transfer control
 - Memory management

The Java Virtual Machine

- Language: bytecode
- Memory Heap+Stack+Permanent
- Interpreter
- Operations and Data Structures for:
 - Primitive data processing
 - Sequence control
 - Data transfer control
 - Memory management

The core of a JVM interpreter is basically this:

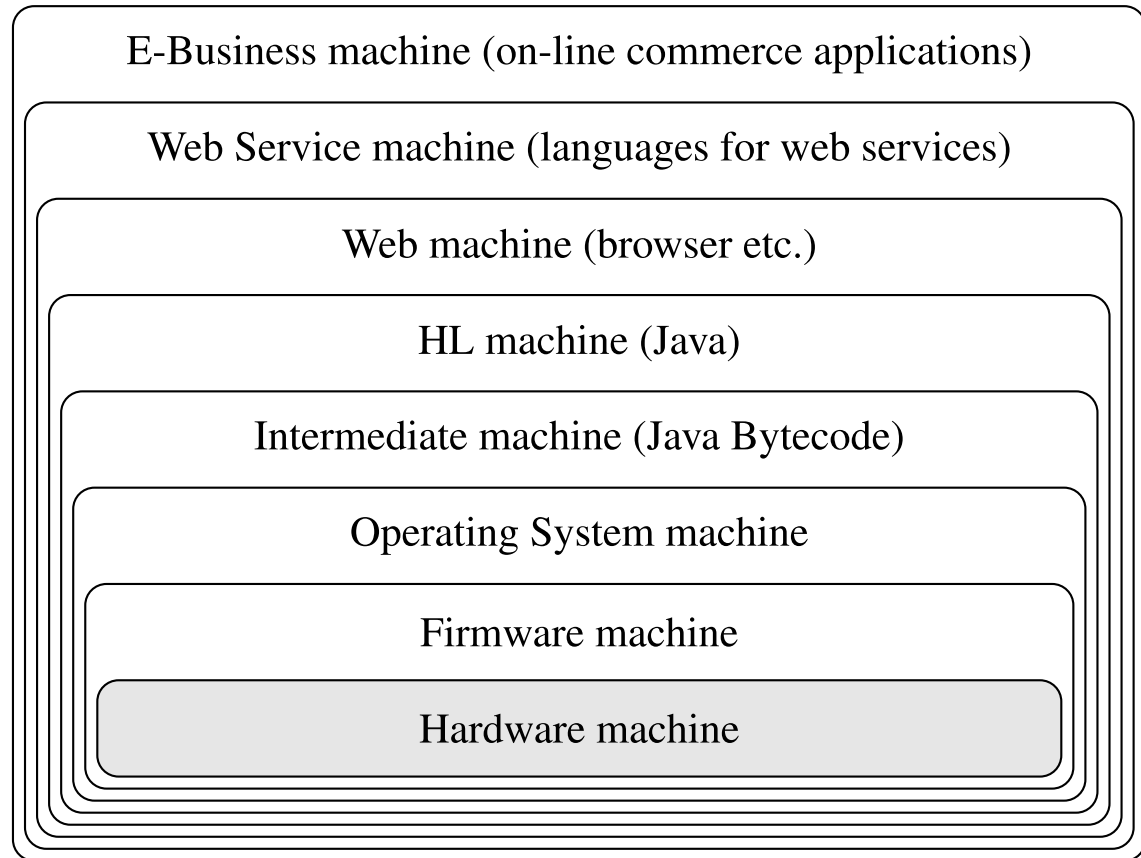
```
do {  
    byte opcode = fetch an opcode;  
    switch (opcode) {  
        case opCode1 :  
            fetch operands for opCode1;  
            execute action for opCode1;  
            break;  
        case opCode2 :  
            fetch operands for opCode2;  
            execute action for opCode2;  
            break;  
        case ...  
    } while (more to do)
```

Implementing an Abstract Machine

- Each abstract machine can be implemented in **hardware** or in **firmware**, but if high-level this is not convenient in general
 - Exception: Java Processors, ...
- Abstract machine **M** can be implemented over a **host machine** **M₀**, which we assume to be already implemented
- The components of **M** are realized using data structures and algorithms implemented in the machine language of **M₀**
- Two main cases:
 - The interpreter of **M** coincides with the interpreter of **M₀**
 - **M** is an **extension** of **M₀**
 - other components of the machines can differ
 - The interpreter of **M** is different from the interpreter of **M₀**
 - **M** is **interpreted** over **M₀**
 - other components of the machines may coincide

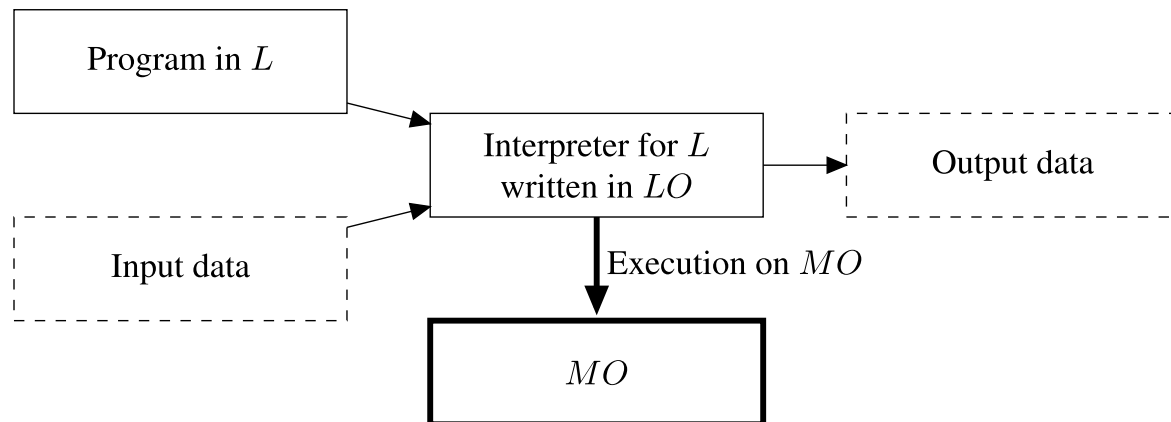
Hierarchies of Abstract Machines

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



Implementing a Programming Language

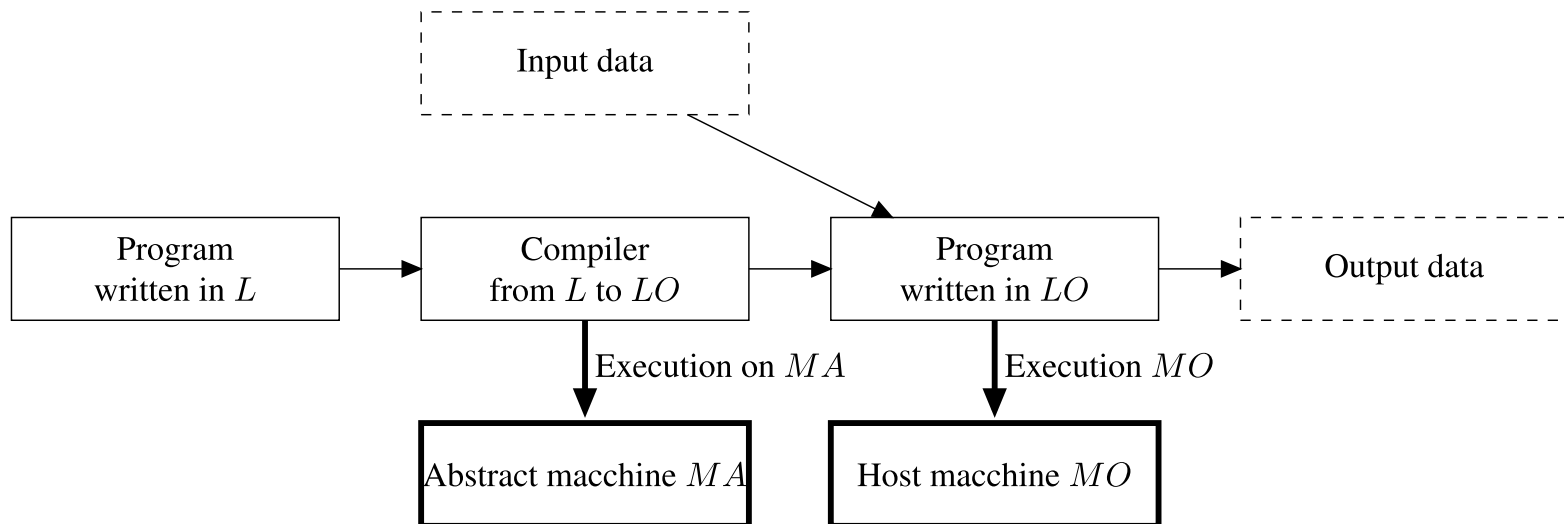
- **L** high level programming language
- **M_L** abstract machine for **L**
- **M_O** host machine
- **Pure Interpretation**
 - **M_L** is interpreted over **M_O**
 - Not very efficient, mainly because of the interpreter (fetch-decode phases)



Implementing a Programming Language

- **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

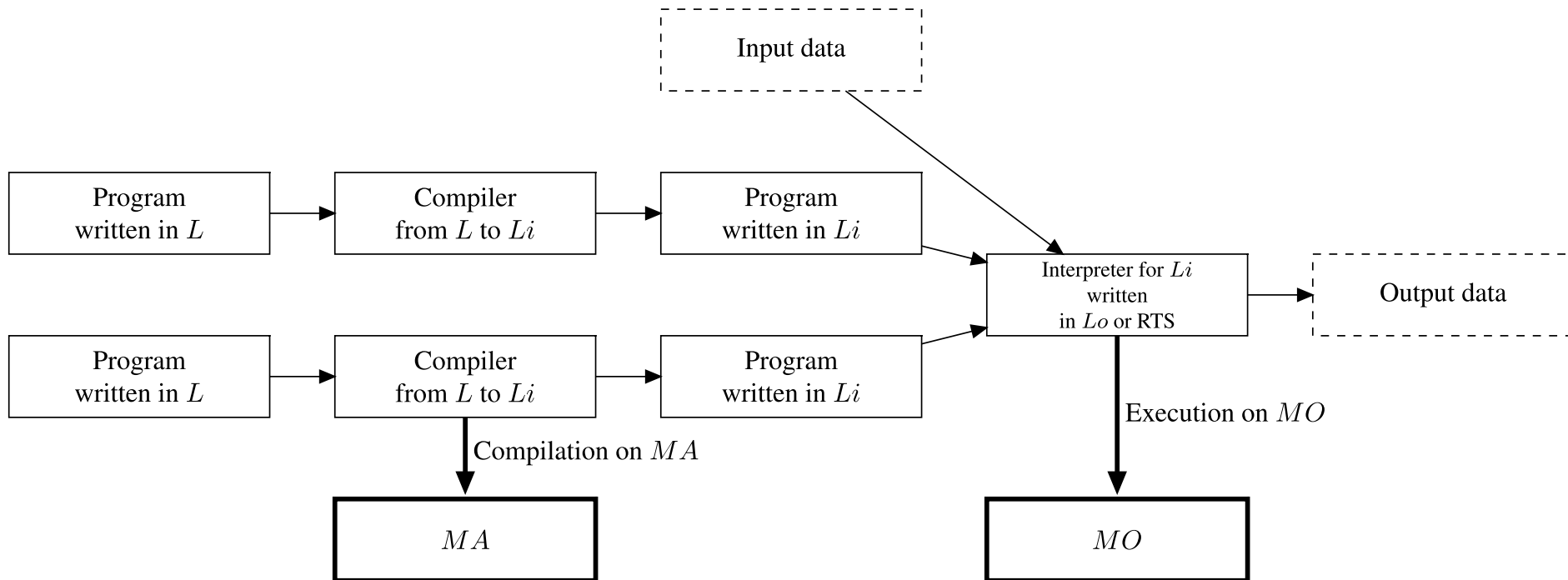
Compilation versus Interpretation

- 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** M_I with language L_I
 - A program in L is compiled to a program in L_I
 - The program in L_I is executed by an interpreter for M_I

Compilation + Interpretation with Intermediate Abstract Machine



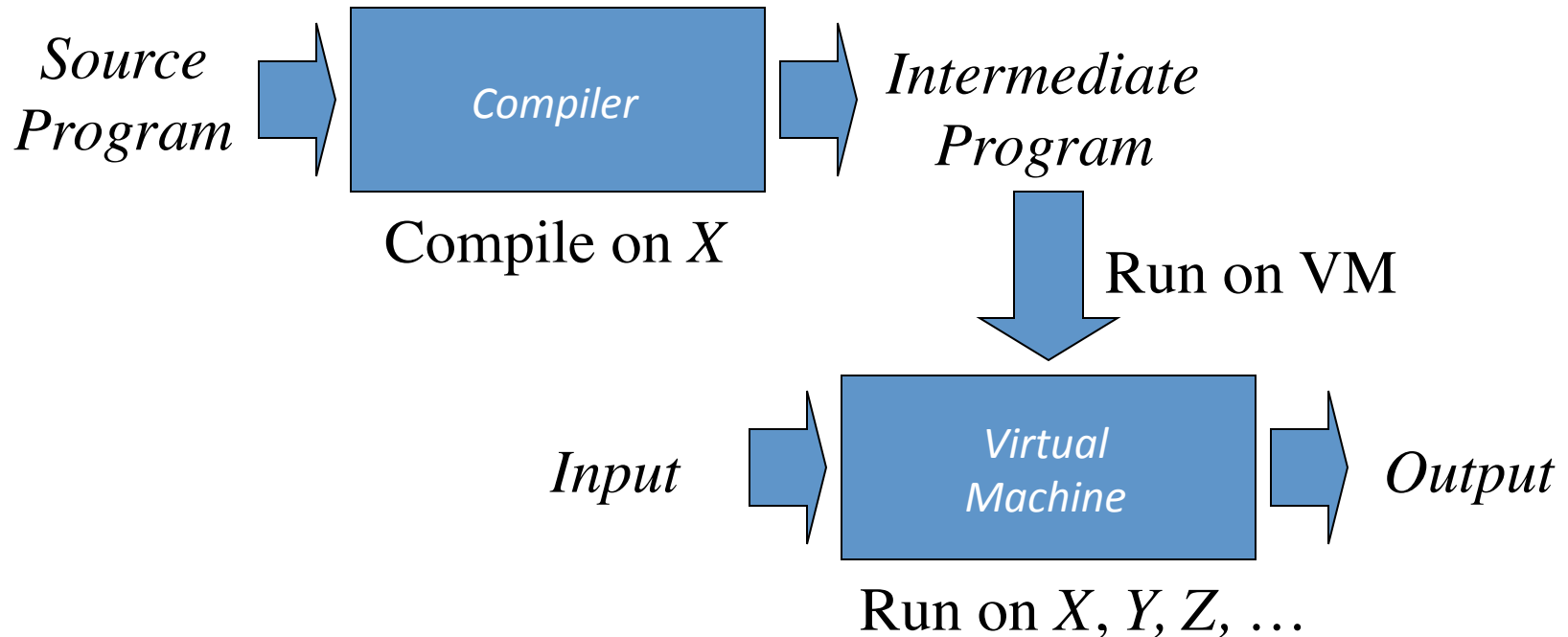
- The “pure” schemes as limit cases

Virtual Machines as Intermediate Abstract Machines

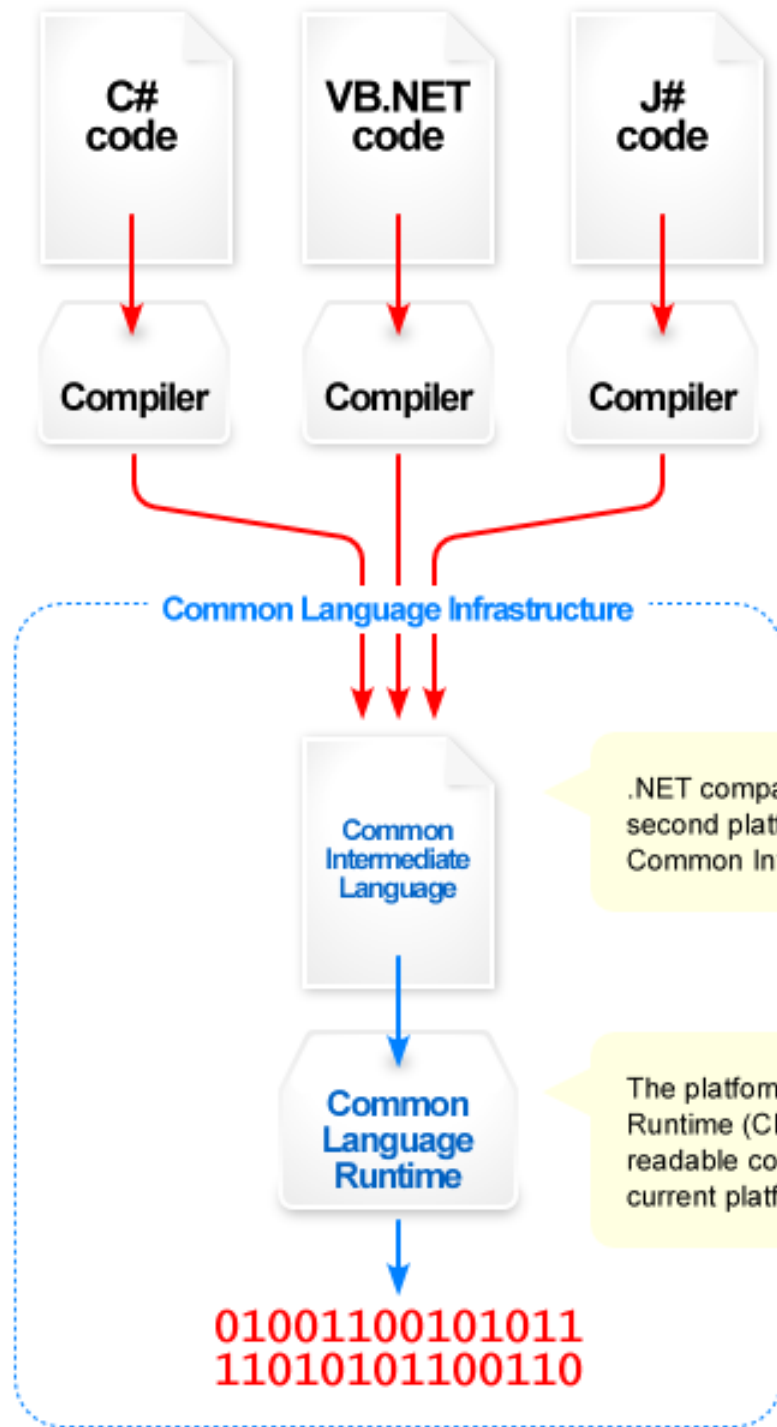
- Several language implementations adopt a compilation + interpretation schema, where the Intermediate Abstract Machine is called **Virtual Machine**
- Adopted by Pascal, Java, Smalltalk-80, C#, functional and logic languages, and some scripting languages
 - Pascal compilers generate **P-code** that 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



Other Intermediate Machines

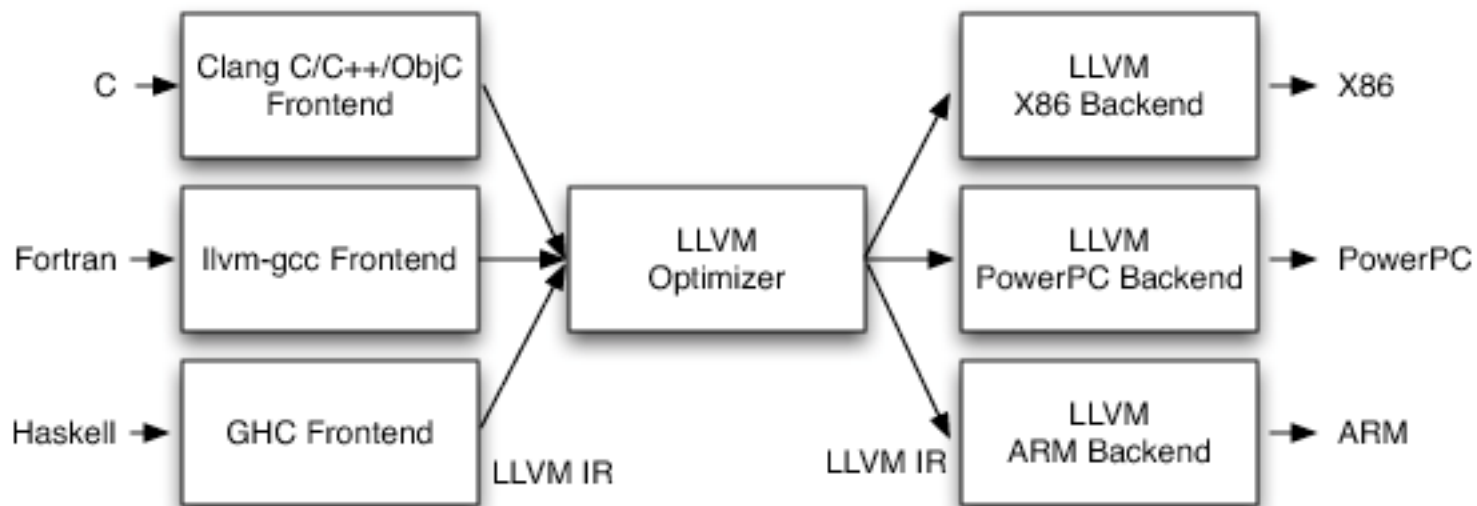


- Microsoft compilers for C#, F#, ... generate **CIL** code (Common Intermediate Language) conforming to **CLI** (Common Language Infrastructure).
- It can be executed in **.NET** or other Virtual Execution Systems (like **Mono**)
- **CIL** is compiled to the target machine

Other Intermediate Machines

LLVM is a **compiler infrastructure** designed as a set of reusable libraries with well-defined interfaces:

- Implemented in C++
- Several front-ends
- Several back-ends
- First release: 2003
- The LLVM IR (Intermediate representation) can also be interpreted
- LLVM IR much lower-level than Java bytecodes or CIL

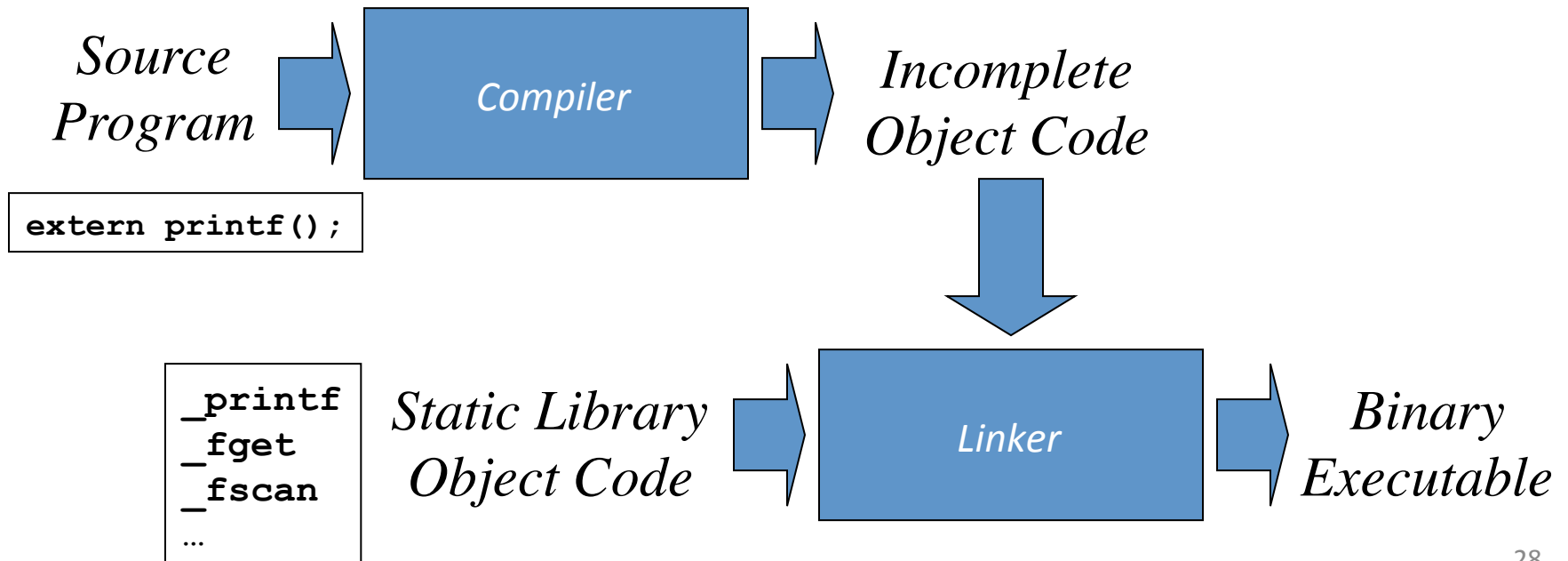


Advantages of intermediate abstract machine (examples for JVM)

- **Portability**: Compile the Java source, distribute the bytecode and execute on any platform equipped with JVM
- **Interoperability**: for a new language **L**, just provide a compiler to JVM bytecode; then it could exploit Java libraries
 - By design in Microsoft CLI
 - De facto for several languages on JVM

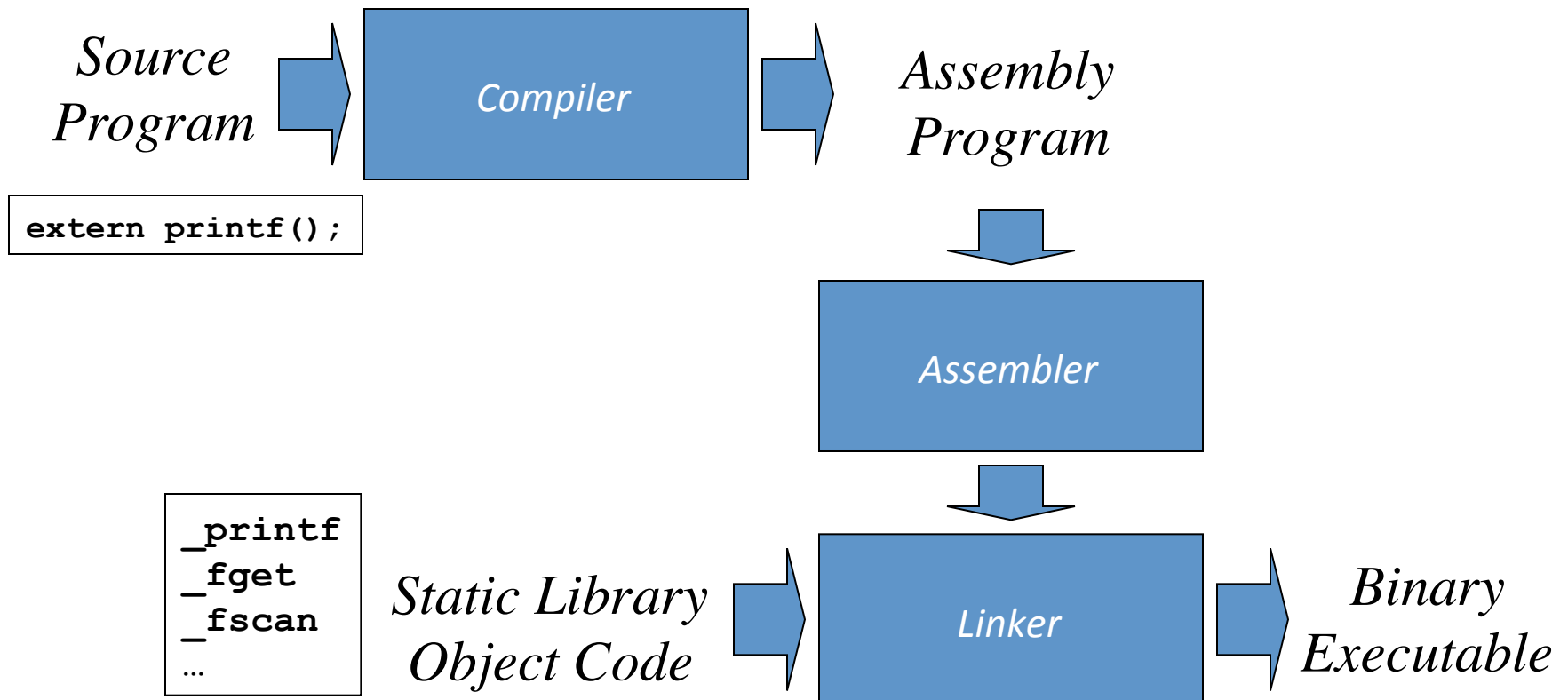
Other Compilation Schemes

- **Pure Compilation and Static Linking**
- Adopted by the typical Fortran systems
- Library routines are separately linked (merged) with the object code of the program



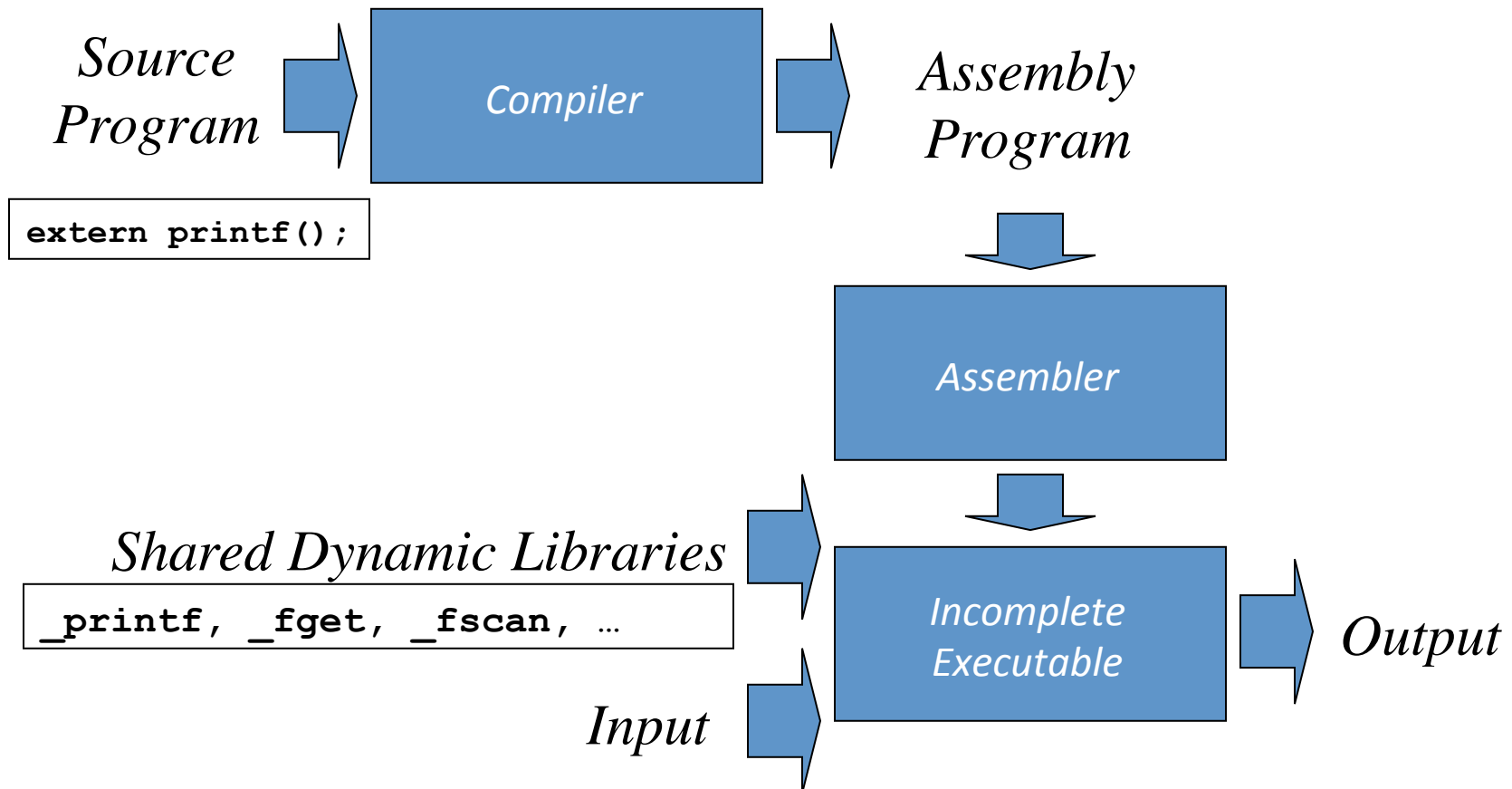
Compilation, Assembly, and Static Linking

- Facilitates debugging of the compiler



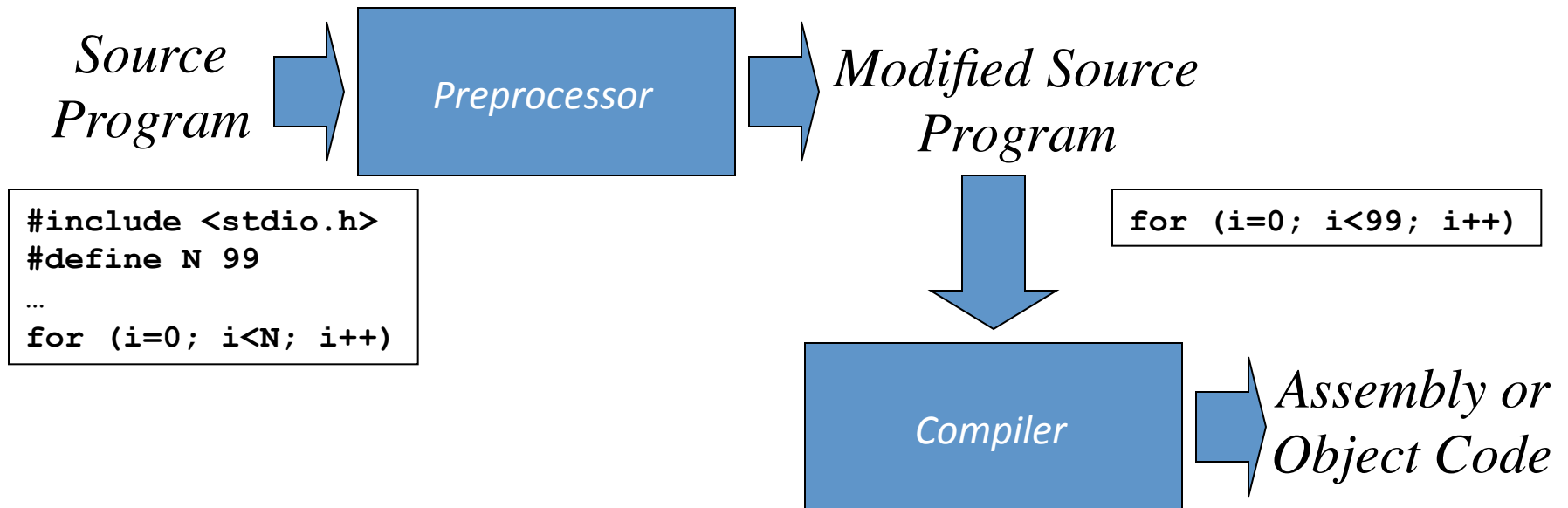
Compilation, Assembly, and Dynamic Linking

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



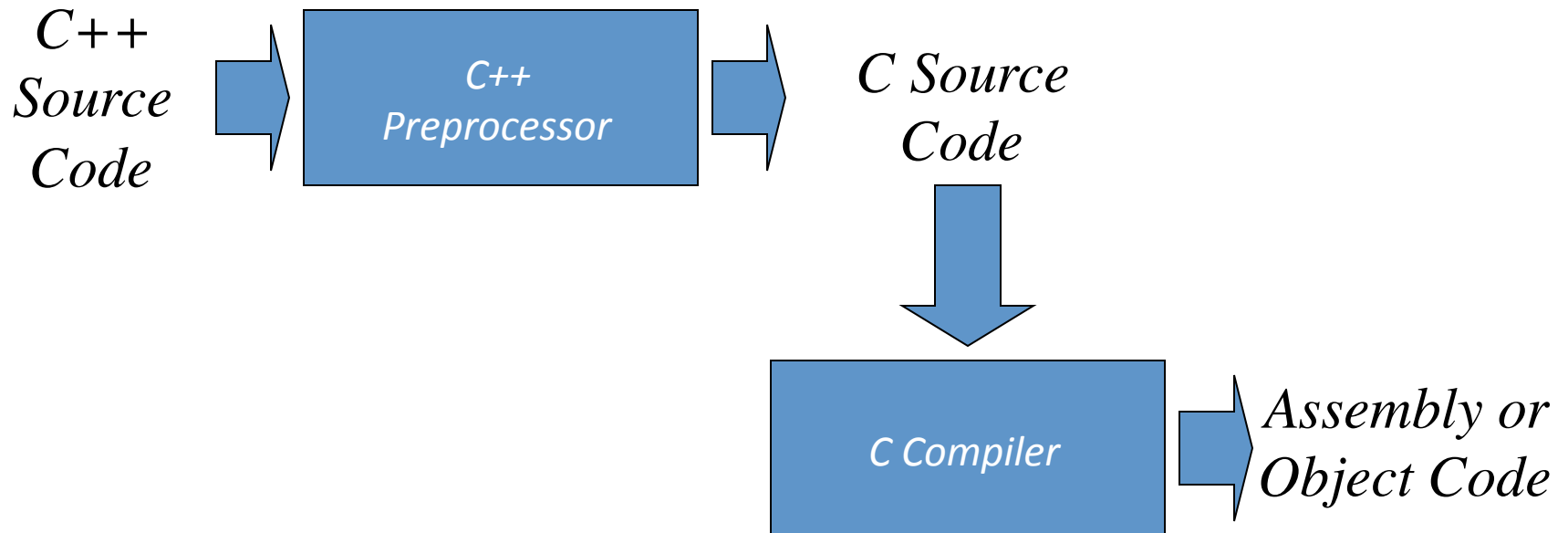
Preprocessing

- 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 generate C code for compilation



Summary: Languages and Abstract Machines

Compilation and interpretation schemes

- **Reading:** Ch. 1 of *Programming Languages: Principles and Paradigms* by M. Gabbrielli and S. Martini
- Syntax, Semantics and Pragmatics of PLs
- Programming languages and Abstract Machines
- Interpretation vs. Compilation vs. Mixed
- Examples of Virtual Machines
- Examples of Compilation Schemes
- → Next: **Runtime Support** and the **JVM**