301AA - Advanced Programming

Lecturer: **Andrea Corradini**

andrea@di.unipi.it

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

***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.

A PROGRAM THAT IS NOT WELL FORMED (WRONG SYNTAX) HAS NO MEANING!

Syntax (+)

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

THE TERMINAL SYMBOLS ARE THE ASCII CHARS

THE TERMINALS SYMBOLS ARE THE TOKENS

Semantics

AMBIGUITIES IN THE SEMANTICS ⇒ WE CAN'T PREDICT THE BEHAVIOUR OF A GIVEN PROGRAM

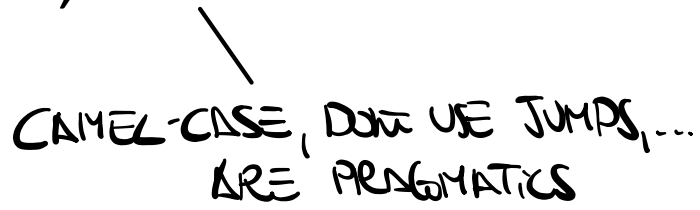
- Usually described precisely, but informally, in natural language.
  - May leave (subtle) ambiguities
- Formal approaches exist, often they are applied to toy languages or to fractions of real languages
  - Denotational [Scott and Strachey 1971]
  - Operational [Plotkin 1981]
  - Axiomatic [Hoare 1969]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

Pragmatics

- Includes coding conventions, guidelines for elegant structuring of code, etc.
- Examples:
  - Java Code Conventions

    http://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*



DON'T CREATE PARTITION OF LANGUAGES! IT'S JUST THE KEY INGREDIENTS OF A PL

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, higher-order functions, …
- **Logic programming**: assertions, relations, …

Classification of languages according to paradigms can be misleading

Implementation of a Programming Language *L*

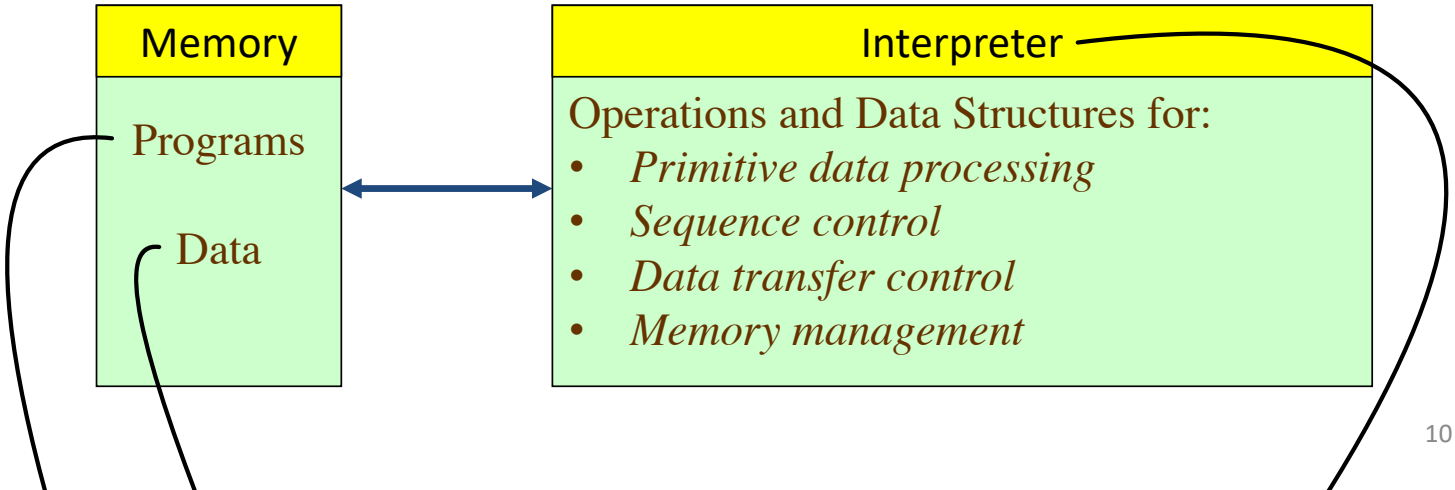
- Programs written in ***L*** must be executable
- Every language ***L*** implicitly defines an ***Abstract Machine*** M_L having ***L*** as machine language
- Implementing M_L on an existing host machine M_O (via ***compilation***, ***interpretation*** or both) makes programs written in ***L*** executable

YOU HAVE A SYNTAX, A SEMANTIC AND THE PRAGMATICS, NOW YOU HAVE TO MAKE EXECUTABLE THE PROGRAMS WRITTEN IN L AND THE EXECUTION HAS TO RESPECT THE GIVEN SEMANTICS

ABSTRACTION OF HW MACHINES

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: = MEMORY + INTERPRETER



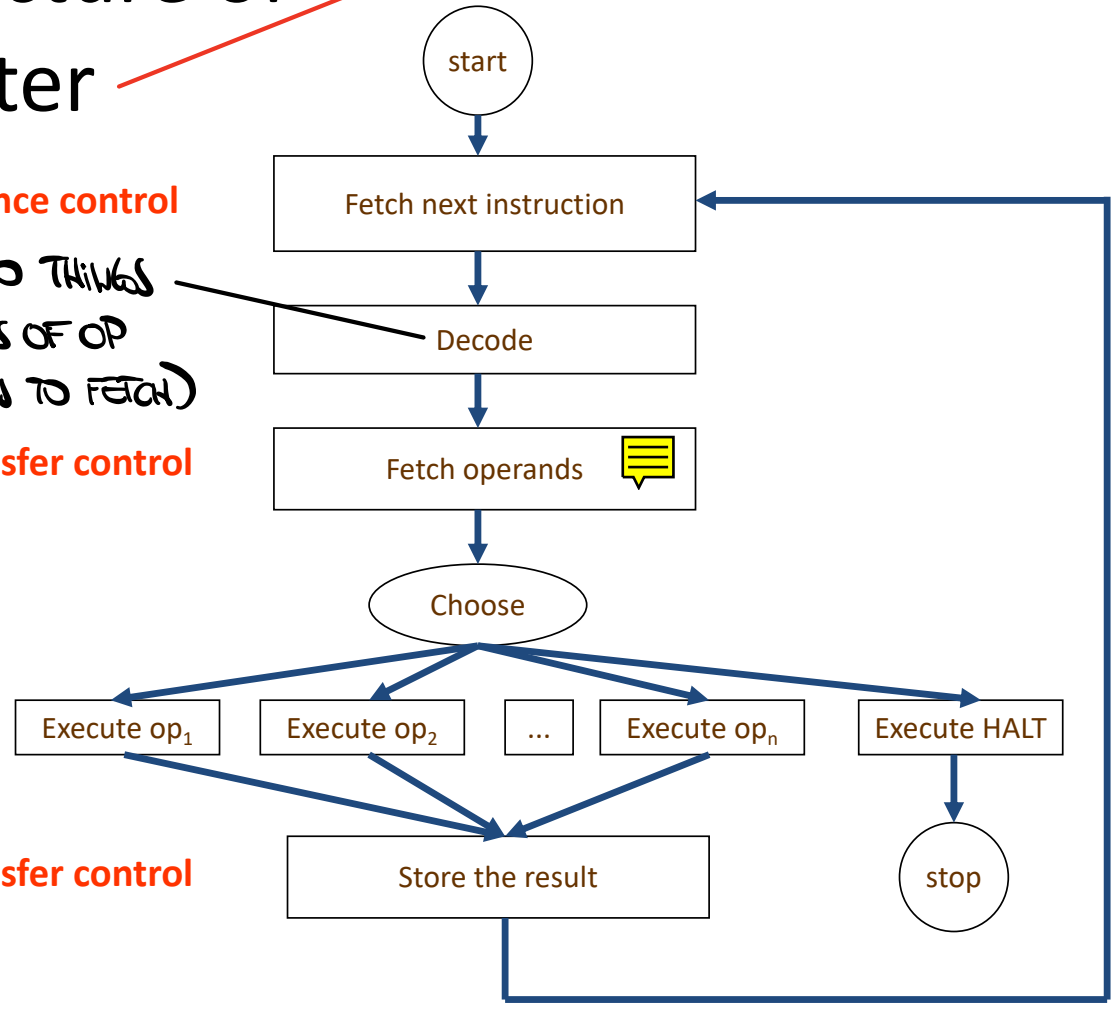
THE DATA OF THE PROGRAM (INPUT, WORKING DATA, ...)

AT THIS LEVEL OF ABSTRACTION A PROGRAM ITSELF IS A DATA TO BE MEMORIZED IN THE MEMORY

- PRIMITIVE DATA PROCESS: EXECUTES EASY PRIMITIVE OPERATIONS (THAT ARE ARCHITECTURALLY REASONABLE)
- SEQUENCE CONTROL: HANDLES CONDITIONAL JUMPS, RETURNS,.. BASICALLY IT HANDLES THE EXECUTION FLOW OF P
- DATA TRANSFER CONTROL: HANDLES PARAMETER PASSING AND VALUES RETURNS
- MEMORY MANAGEMENT: HOW TO STORE AND WHERE TO STORE DATA. BRING IN REGISTERS DATA WHEN NEEDED + HANDLES DEALLOCATION

General structure of the Interpreter

FETCH-DECODE-EXECUTE LOOP



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
- Components of ***M*** correspond to components of L_M:
  - Primitive data processing ➔ Primitive data types
  - Sequence control ➔ Control structures
  - Data transfer control ➔ Parameter passing and value return
  - Memory management ➔ Memory management

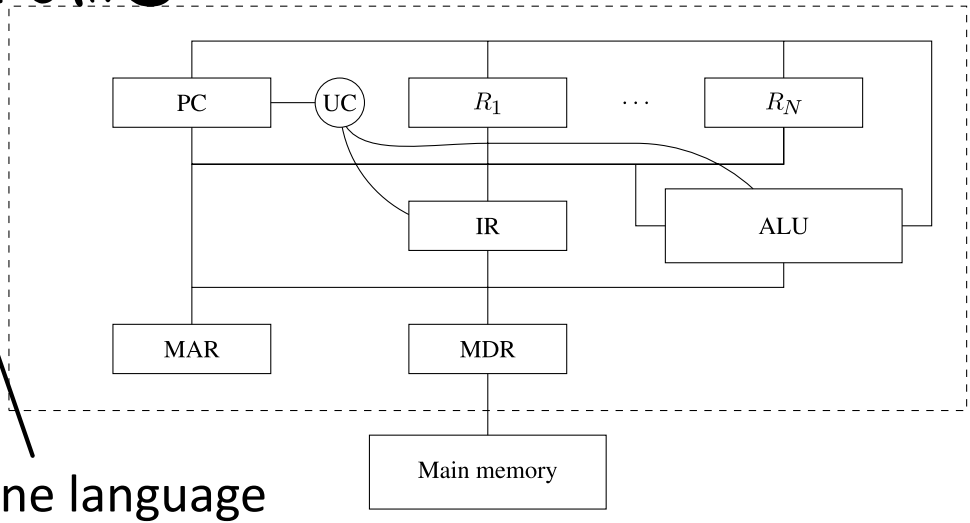
IF M AND THE LANGUAGE ARE ON THE SAME LEVEL THEN THERE A 1-TO-1 CORRISPONDENCE

- GIVEN THE PRIMITIVE DATA PROCESSING (PRIMITIVE OPERATIONS) WE DERIVE THE PRIMITIVE DATA TYPES OF THE LANGUAGE
- GIVEN THE SEQUENCE CONTROL WE GET WHICH BRANCH CONSTRUCTS OUR LANGUAGE SUPPORT
- GIVEN THE DATA TRANFER CONTROL WE GET WHICH KIND OF PARAMETERS PASSING THE LANGUAGE USES AND WHICH METHOD OF RETURN THE FUNCTIONS OF L_M USE

- ...

An example: the Hardware Machine

OF ABSTRACT MACHINE

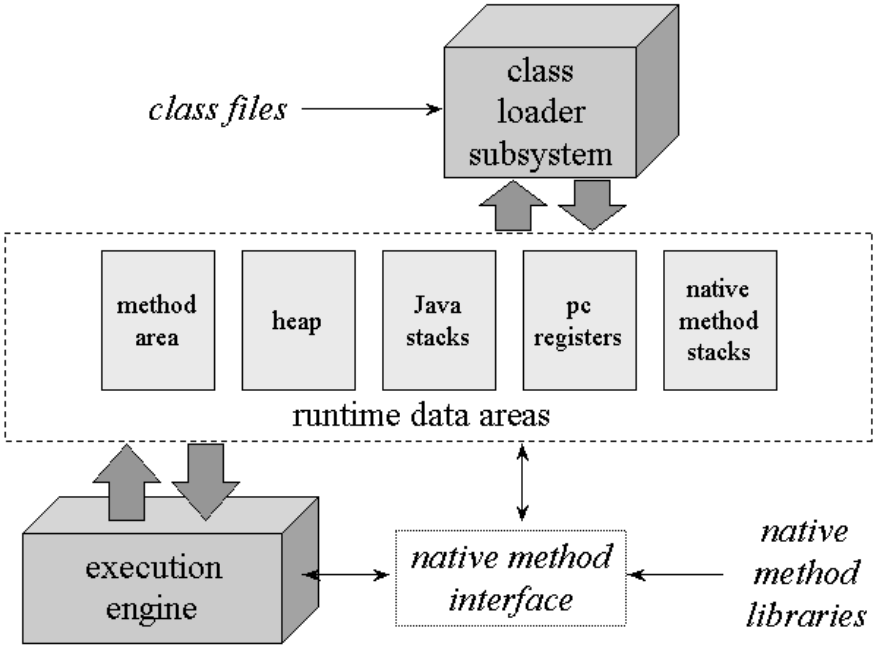


- Language: Machine language
- Memory: Registers + RAM (+ cache)
- Interpreter: fetch, decode, execute loop
- Operations and Data Structures for:
  - Primitive data processing
  - Sequence control
  - Data transfer control
  - Memory management

ACHIEVED BY PROGRAM COUNTER MANIPULATION

USING LOAD/STORE IN RIGHT REGISTERS THAT WILL BE USED BY THE CALLED FUNCTION AND STORE THE RESULT IN THE RIGHT LOCATION

The Java Virtual Machine



- Language: bytecode
- Memory Heap+Stack+Permanent
- Interpreter

The Java Virtual Machine

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)
```

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

LATER ON

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, … ✱ SEE BELOW
- Abstract machine **M** can be implemented over a **host machine** $\mathbf{M_o}$, which we assume to be already implemented
- The components of **M** are realized using *data structures* and *algorithms* implemented in the machine language of $\mathbf{M_o}$
- Two main cases:
  - The interpreter of **M** coincides with the interpreter of $\mathbf{M_o}$
    - **M** is an **extension** of $\mathbf{M_o}$
    - other components of the machines can differ
  - The interpreter of **M** is different from the interpreter of $\mathbf{M_o}$
    - **M** is **interpreted** over $\mathbf{M_o}$
    - other components of the machines may coincide

* THEORETICALLY IS POSSIBLE TO IMPLEMENT THE AM DIRECTLY ON HARDWARE
  - OFC IT IS VERY DIFFICULT SINCE THE AM AND THE HW COULD BE VERY DISTANT FROM EACH OTHER IN THE ABSTRACTION STANDPOINT
  - ALSO A SMALL CHANGE TO THE AM WOULD REQUIRE THE REDESIGN AND THE REPRODUCTION OF THE PHYSICAL HARDWARE (THE ARCHITECTURE OF THE CPU)
    - THIS IS WHY HW THAT INTERPRETER BYTECODE, WHILE BEING VERY EFFICIENT IN THE COMPUTATION, ARE NOT PURSUED (A BRAND NEW ARCHITECTURE EVERY VERSION OF JAVA IS NOT DOABLE, BOTH TECHNICALLY AND ECONOMICALLY)

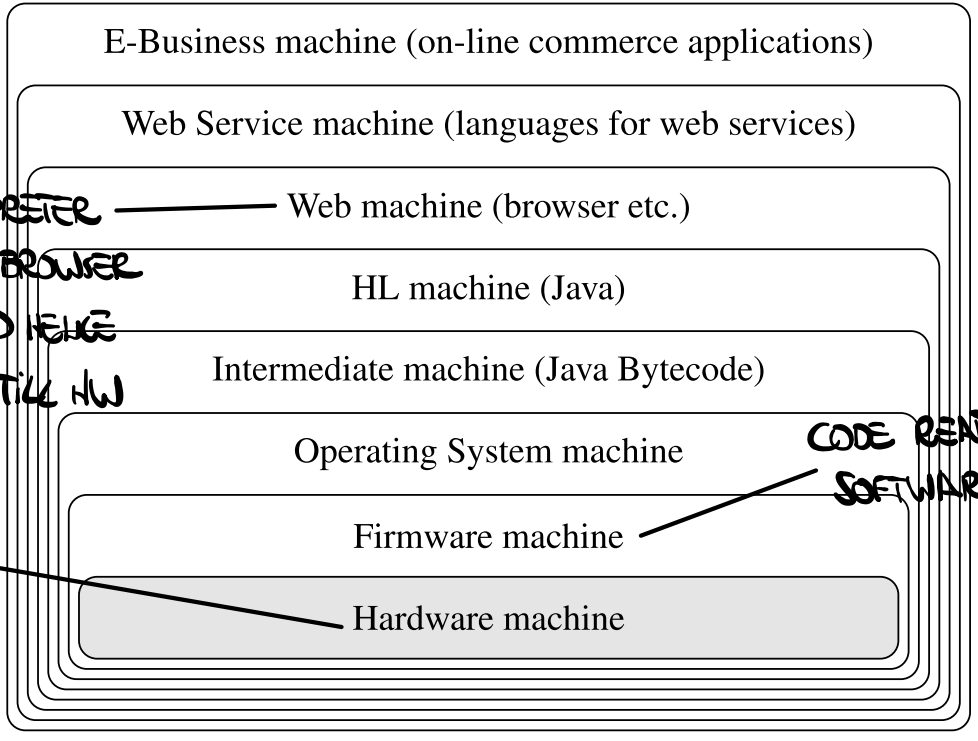
THERE IS AN HIERARCHY OF MACHINES WHERE THE i+1-TH MACHINE IS INTERPRETED BY THE i-TH MACHINE. THE RECURSION ENDS WHEN WE REACH THE LOWEST MACHINE, WHICH IS THE ACTUAL PROCESSOR

- MORE HIGH YOU ARE AND MORE YOU ARE HUMAN FRIENDLY BUT HW-UNFRIENDLY AND VICEVERSA

IDEALLY YOU DON'T HAVE TO JUMP LEVELS BUT SOMETIMES, FOR EFFICIENCY REASONS, YOU DO
• EXAMPLE: THE BROWSER MAKE A SYSTEM CALL

Hierarchies of Abstract Machines

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



THE BROWSER IS AN INTERPRETER FOR JAVASCRIPT. ALSO THE BROWSER MAY BE WRITTEN IN JAVA AND HENCE EXECUTED ON THE JVM, ... TILL HW

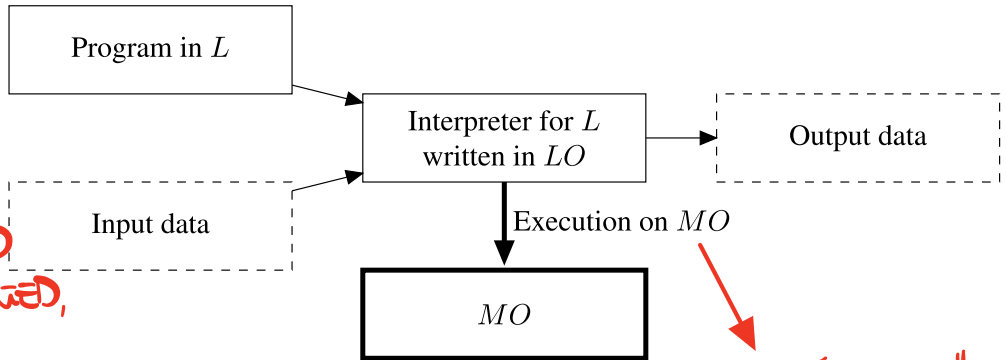
AND-OR-NOT LOGIC PORTS

CODE READ ONLY SOFTWARE

Implementing a Programming Language

- **L** high level programming language
- $\mathbf{M_L}$ abstract machine for **L**
- $\mathbf{M_O}$ host machine (WHICH WE ASSUME IMPLEMENTED AND EXECUTABLE)
- **Pure Interpretation**
  - $\mathbf{M_L}$ is interpreted over $\mathbf{M_O}$
  - Not very efficient, mainly because of the interpreter (fetch-decode phases)

THERE ARE NOT OPTIMIZATION



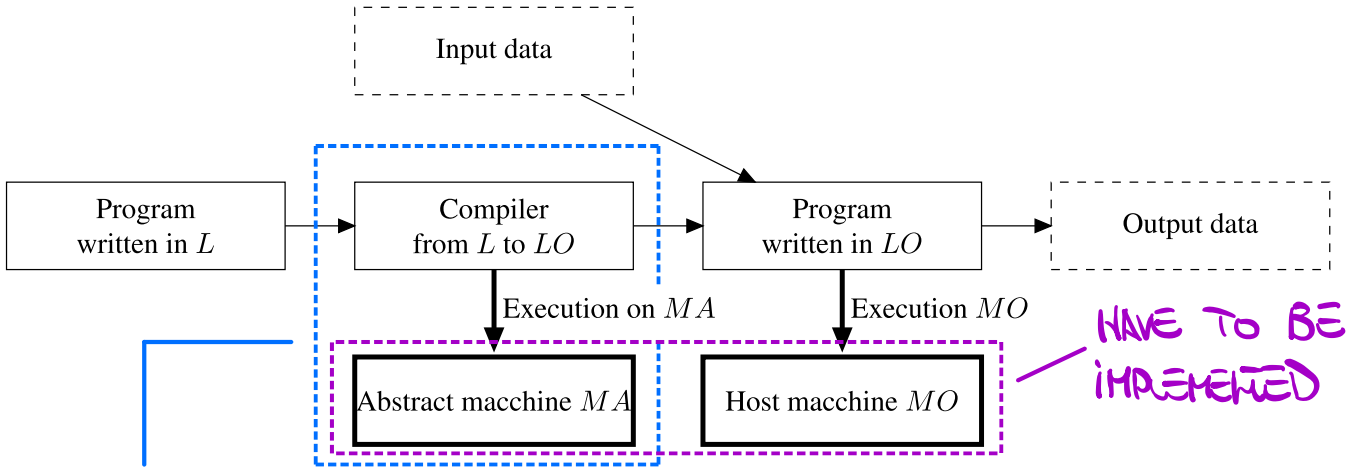
THE CODE is READ ONLY WHEN EXECUTED, EVERYTHING is AT RUNTIME

SORT OF "EMULATION" OF M_L

Implementing a Programming Language

THE "EXECUTION" HAS "2 PHASES":
a) STATIC PHASE: COMPILE THE CODE
b) DYNAMIC PHASE: EXECUTE THE COMPILED CODE

- **Pure Compilation**
  - Programs written in **L** are *translated* into equivalent programs written in $\mathbf{L_O}$, the machine language of $\mathbf{M_O}$
  - The translated programs can be executed directly on $\mathbf{M_O}$
    - $\mathbf{M_L}$ is not realized at all // WE TRASLATE L TO L_O THAT RUNS ON M_O!
  - Execution more efficient, but the produced code is larger ☠ SEE BELOW



THE COMPILER IS A PROGRAM THAT MAYBE WRITTEN IN A THIRD LANGUAGE L_A EXECUTABLE ON A (ABSTRACT) MACHINE M_A: CROSS-COMPILATION, COMPILATION AND EXECUTION ON DIFFERENT MACHINES

WHENEVER POSSIBLE THE DECISIONS ARE TAKEN IN THE STATIC PHASE (AT COMPILE TIME), BEFORE EXECUTING THE PROBLEM.

WE DON'T CHECK STUFF AT RUNTIME (TYPE CORRECTNESS, CONTROL FLOW, MEMORY ALLOCATION...)

- WE HAVE TO CHECK STUFF THAT CAN BE DONE ONLY AT RUNTIME (EX: USER INPUT...)

- COMPILATION IS DONE ONCE, INTERPRETATION IS DONE AT EVERY EXECUTION

* IT IS MORE EFFICIENT THX TO THE (CODE) OPTIMIZATION THAT WE DO AT COMPILE TIME
  - THE SIZE OF CODE EXPLODES THO
    - GENERALLY SPEAKING MORE L AND L_0 ARE DISTANT (IN ABSTRACTION) AND MORE INSTRUCTION OF L_0 WILL BE NEEDED TO PERFORM AN ISTRUCTION OF L
      - AN ISTRUCTION LIKE MALLOC IS PERFORMED WITH HUNDREDS OF LINES OF MACHINE CODE

PURE COMPILATION AND PURE INTERPRETATION ARE LIMIT CASES AND ARE ALMOST NEVER USED

Compilation versus Interpretation

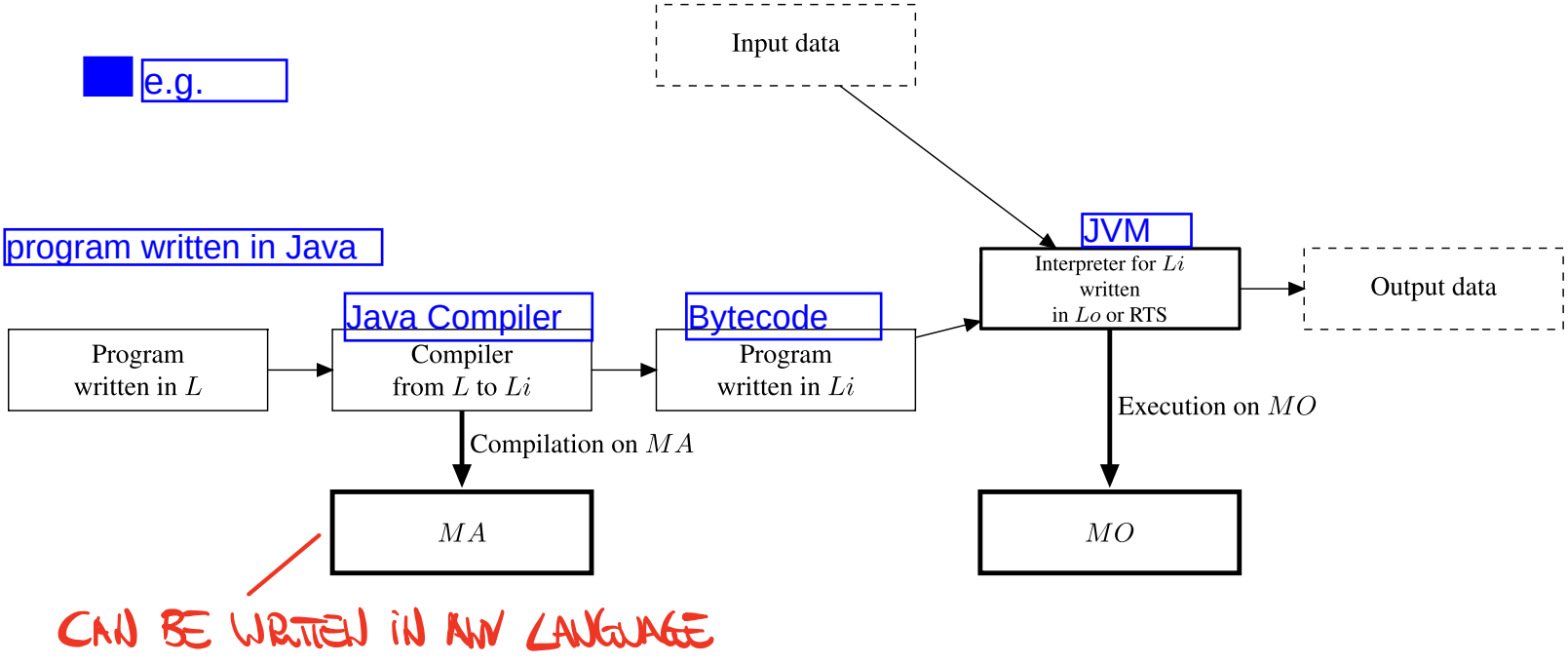
- Compilers efficiently fix decisions that can be taken at compile time to avoid to generate code that makes this decision at run time
  - Type checking at compile time vs. runtime — TYPE CHECKING CAN BE DONE BOTH AT COMPILE AND RUN TIME - JAVA DO BOTH
  - Static allocation — KNOWN WHERE VARS ARE ALLOCATED NO LOOKUP IN THE SYMBOL TABLE
  - Static linking — HANDLES WAY MORE EFFICIENTLY THE LIBRARIES
  - Code optimization — DEAD CODE ELIMINATION, ...
- **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

KIND OF STEP-BY-STEP EXECUTION, HENCE STEP-BY-STEP DEBUG IS GRANTED

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



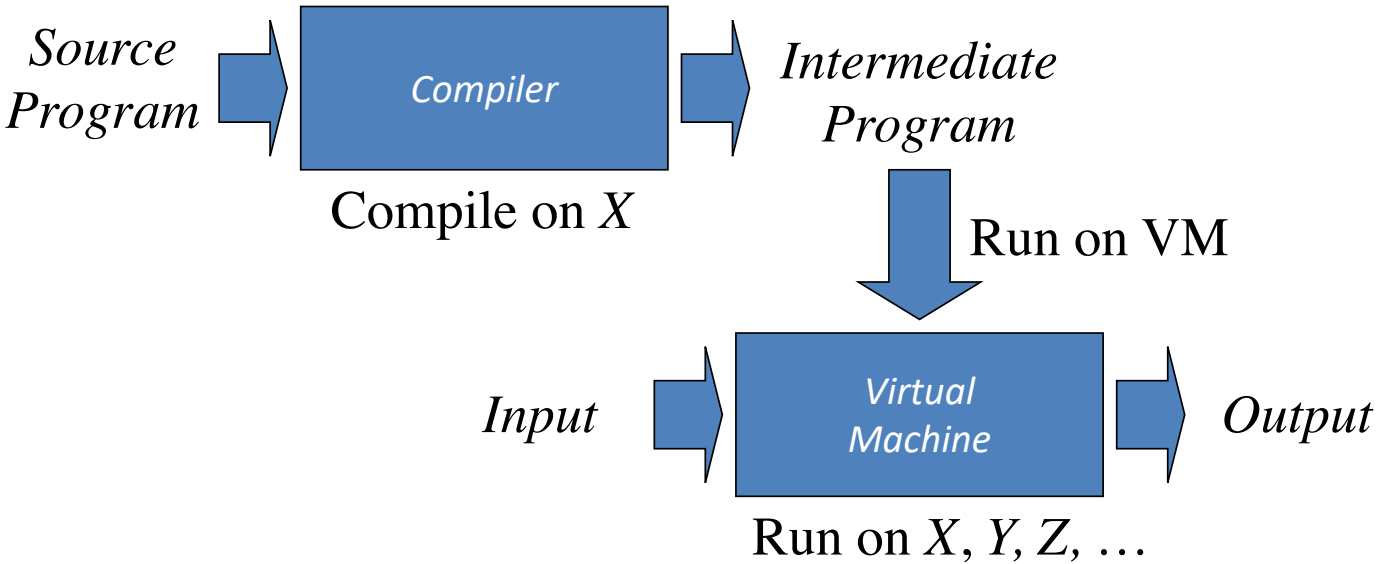
Virtual Machines as Intermediate Abstract Machines

- 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

HAVING A COMPILED INTERMEDIATE CODE IS GOOD FOR PORTABILITY AND FOR INTELLECTUAL PROPERTY (DON'T SHARE THE REAL CODE)

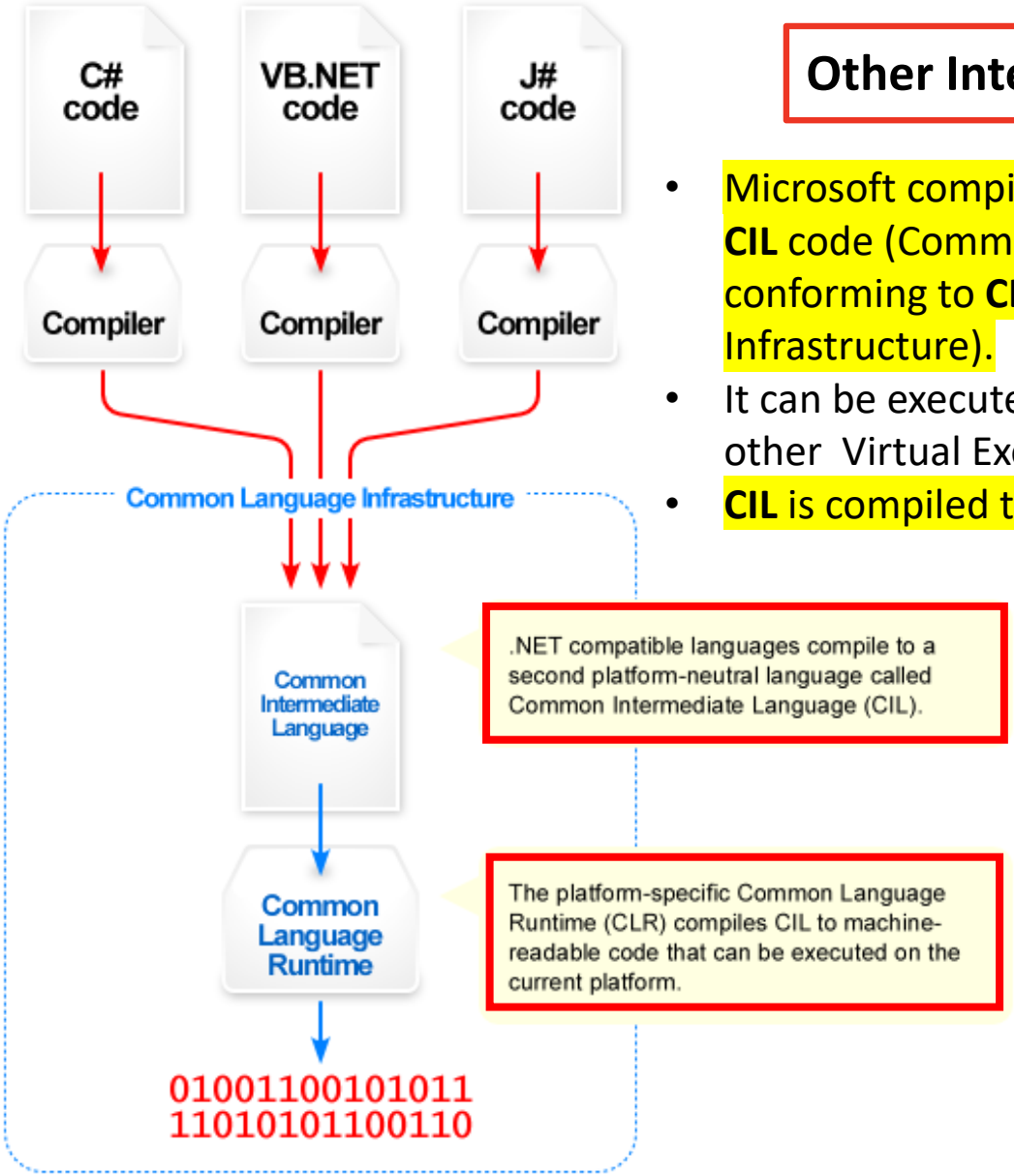
Compilation and Execution on Virtual Machines

- 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** , **.NET Core**, or other Virtual Execution Systems (like **Mono**)
- **CIL** is compiled to the target machine



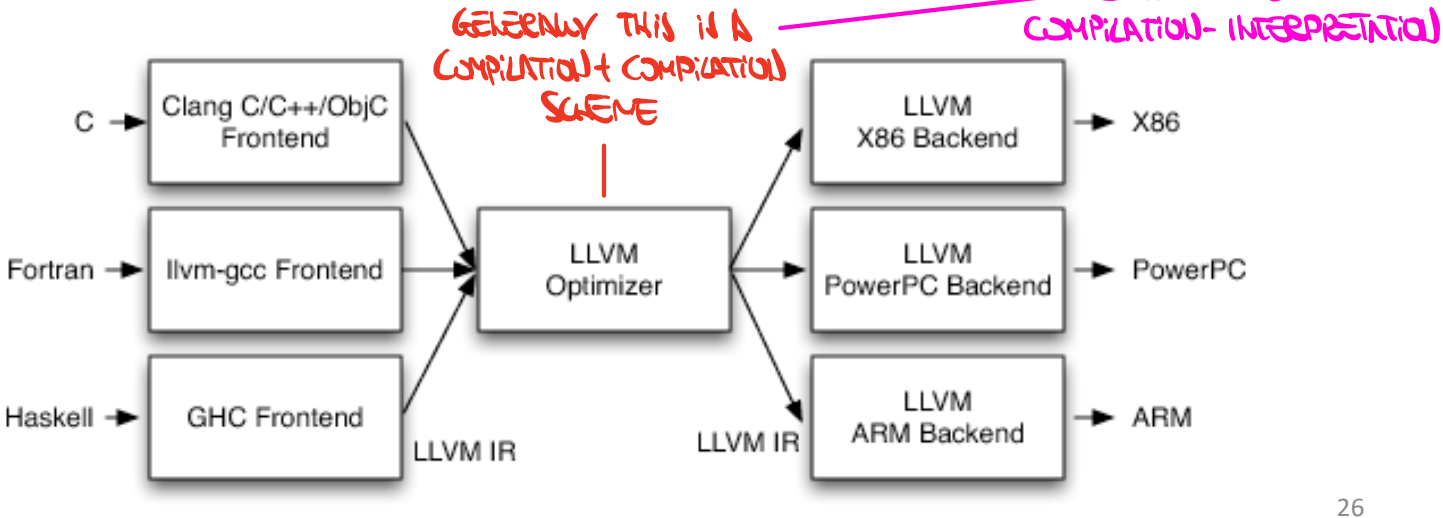
- .NET IS DESIGNED TO HAVE A (DIFFERENT) COMPILER FOR EACH LANGUAGE OF THE .NET FAMILY
  - EACH COMPILER CONVERTS THE LANGUAGE (C#, F#,...) IN THE SAME INTERMEDIATE CODE CALLED COMMON INTERMEDIATE LANGUAGE (CIL)
    - THE CIL CODE IS THEN TAKEN BY THE COMMON LANGUAGE INFRASTRUCTURE (CLI) AND (TO GAIN MAX EFFICIENCY) IT IS TRANSLATED TO ASSEMBLY (THERE IS NO INTERPRETATION IN .NET!)
      - SINCE CIL IS VERY DOCUMENTED AND STANDARDIZED TO IMPLEMENT A BRAND NEW PROGRAMMING LANGUAGE IT IS ENOUGH TO DEVELOP THE COMPILER FROM L TO CIL
        - THE CLI WILL TAKE CARE OF THE REST

- FRONT ENDS (SCANNER, PARSER) FOR A LOT OF KNOWN LANGUAGE THAT ARE TRANSLATED TO THE LLVM INTERMEDIATE REPRESENTATION
- BACK-END FOR DIFFERENT ARCHITECTURES THAT TRANSLATE LLVM IR (OPTIMIZED) TO MACHINE CODE

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)

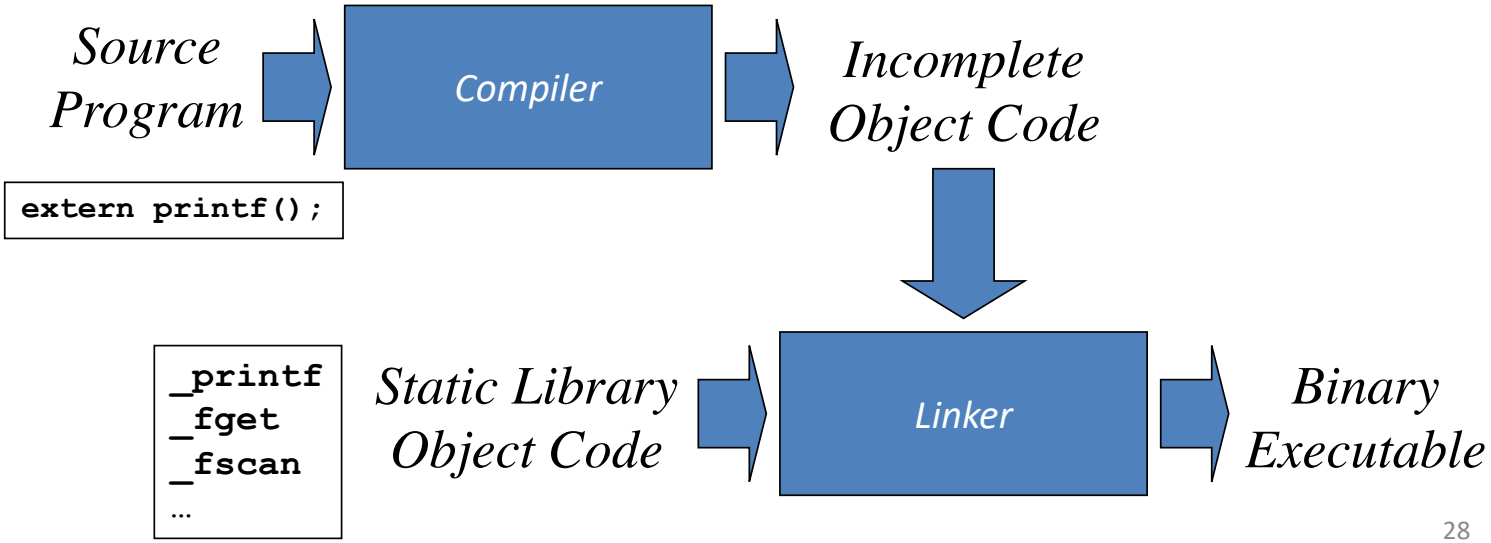
- **Portability**: Compile the Java source, distribute the bytecode and execute on any platform equipped with JVM
- **Interoperability**: for a new language **L**, just provide a compiler to JVM bytecode; then it could exploit Java libraries
  - *By design* in Microsoft CLI
  - *De facto* for several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

NOT BY DESIGN BUT THEN LANGUAGES HAVE BEEN DESIGNED TO BE COMPILED IN BYTECODE (FOR EXAMPLE SCALA)

NOT ONLY A COMPILED PROGRAM DOES RUN ON ANOTHER PROCESSOR BUT THE SAME CODE COMPILED ON DIFFERENT ARCHITECTURES MAY PRODUCE DIFFERENT RESULTS SINCE THE COMPILER ITSELF CAN BE ARCHITECTURE DEPENDANT! WITH INTERMEDIATE REP THE PORTABILITY EXPLODES

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



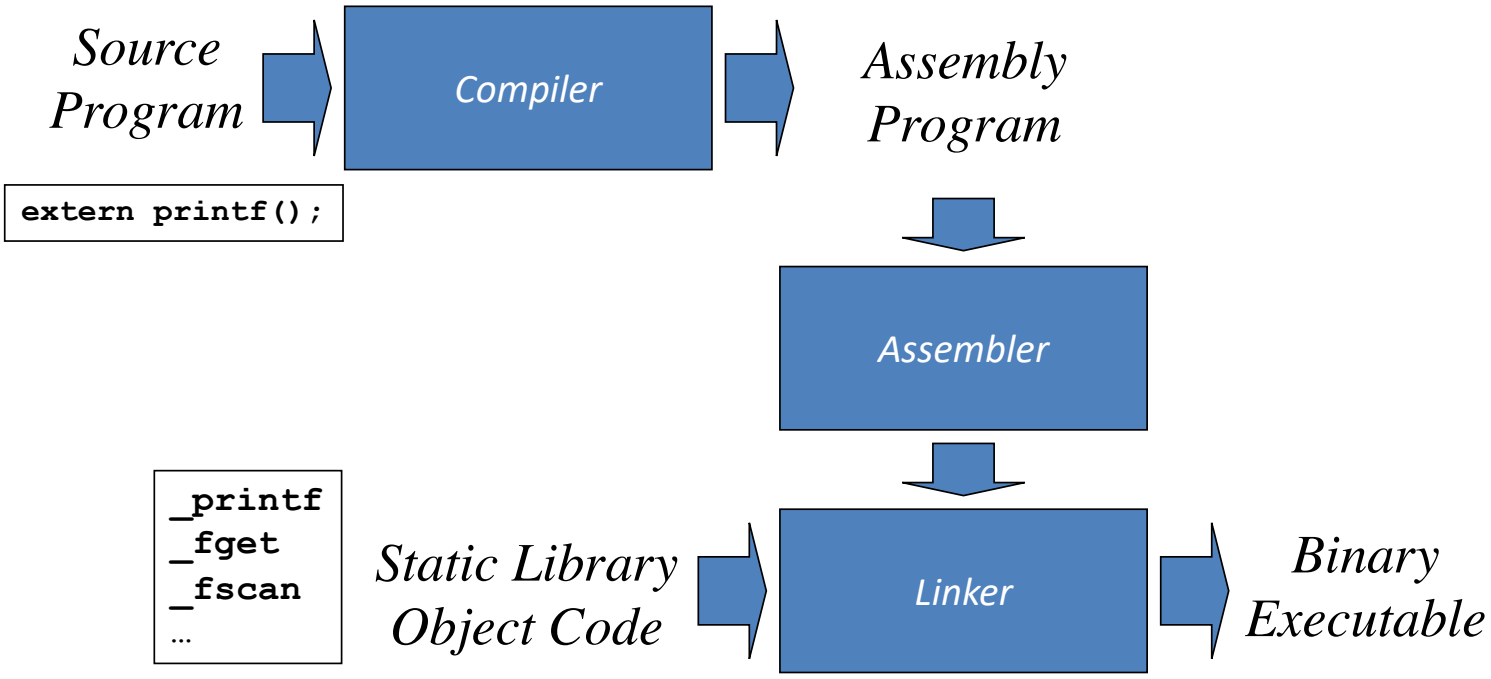
STATIC LINKING IN PURE COMPILATION HAVE TO LINK THE NAME OF THE FUNCION TO THE CORRESPONDING ASSEMBLY CODE TO EXECUTE THE CODE (LINK NAME TO THE ADDRESSES TO JUMP, THIS IS DONE IN COMPILATION PHASE)

EXACTLY AS BUT COMPILE TO ASSEMBLY FACILITATES THE DEBUG (ASSEMBLY IS MORE MORE READABLE THAN MACHINE CODE)

- THEN ASSEMBLER TRANSLATES THE ASSEMBLY TO THE BINARY

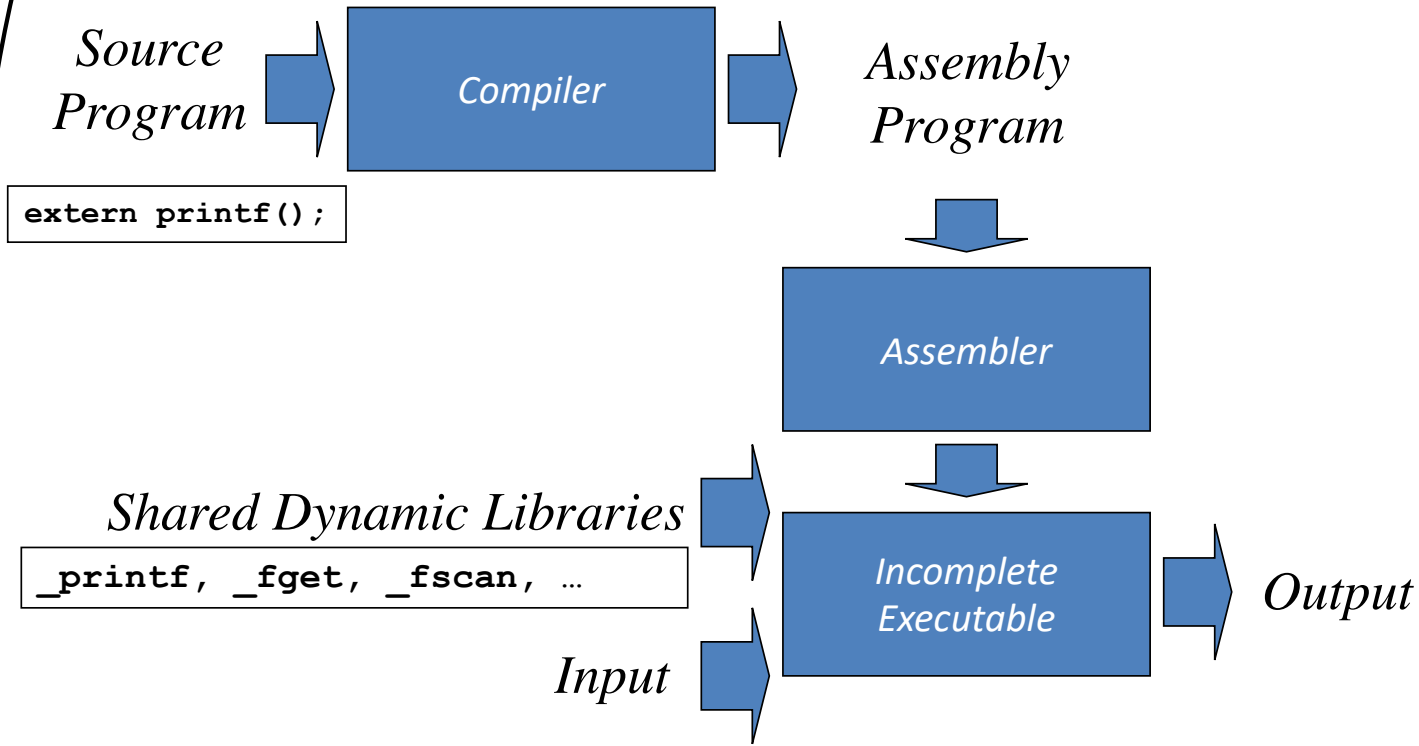
Compilation, Assembly, and Static Linking

- 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)



THIS IS HOW THINGS ARE DONE CURRENTLY

- THE COMPILED CODE IS NOT COMPLETE (HAS "HOLES"), MEANING THAT WE DO NOT HAVE INSERTED THE (STATIC) BODIES OF SOME FUNCTION. WE WILL JUMP TO IT IF NEEDED DURING THE EXECUTION
  - THIS WAY IN THE COMPILED THERE WILL NOT BE THE BODY OF PRINTF IN THERE AND SO ON

SIDE NOTE: REGISTER MACHINE (A KIND OF EXECUTION MODEL)

- THE CPU CAN ONLY USE DATA THAT ARE CONTAINED IN THE REGISTERS
  - PROGRAM COUNTER, THE REGISTERS ARE GIVEN AS INPUT TO THE ALU AND THE RESULT IS PUSHED IN A REGISTER. CLASSIC RISC ARCHITECTURE