Abstract Machines

Programming Language Habitats

Prof. Matt Revelle
CSCI 305

What is an Abstract Machine?

- An ***abstract machine*** is the abstraction of a physical computing machine
- A ***programming language*** $\mathscr{L}$ provides instructions used to construct programs
- A ***program*** written in $\mathscr{L}$ is described in terms of those instructions
- A language $\mathscr{L}$ recognized by $\mathscr{M}_{\mathscr{L}}$ is called the ***machine language*** of $\mathscr{M}_{\mathscr{L}}$

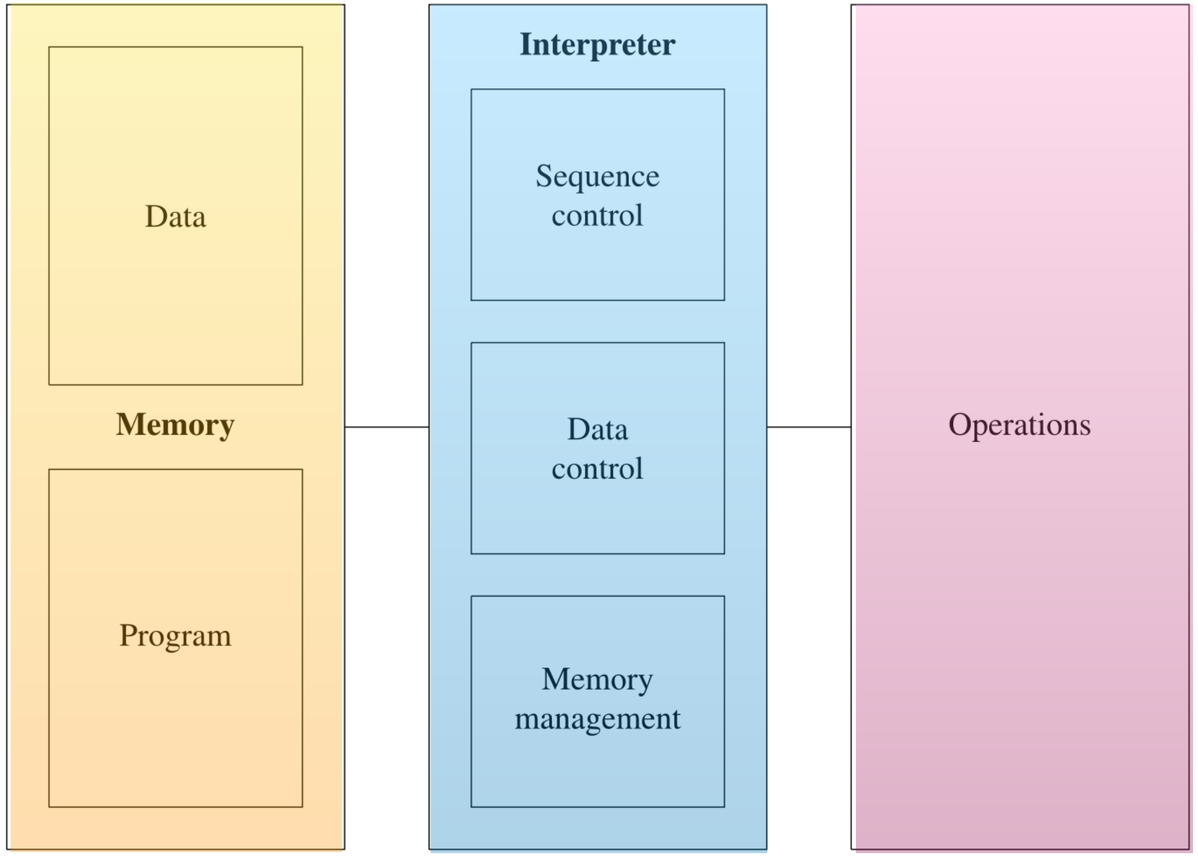
Abstract Machine

Given a programming language $\mathscr{L}$, an *abstract machine* for $\mathscr{L}$ is denoted by $\mathscr{M}_{\mathscr{L}}$.

Any set of data structures and algorithms which can perform the storage and execution of written programs in the language $\mathscr{L}$ is an $\mathscr{M}_{\mathscr{L}}$.

What is an Abstract Machine?

- A generic abstract machine $\mathscr{M}_{\mathscr{L}}$ is composed of a ***store*** and an ***interpreter***
- The store contains programs and data
- The interpreter executes the instructions contained in programs
- Standard ***operations*** common to all interpreters



Operations

- Four categories of operations:
  1. Processing primitive data
  2. Control flow
  3. Data transfer
  4. Memory management

Basic arithmetic for integers and floating point numbers.

Boolean logic.

Keep track of the next instruction to execute.

Support for conditional execution.

Move data between memory and registers.

What data should be available for instructions.

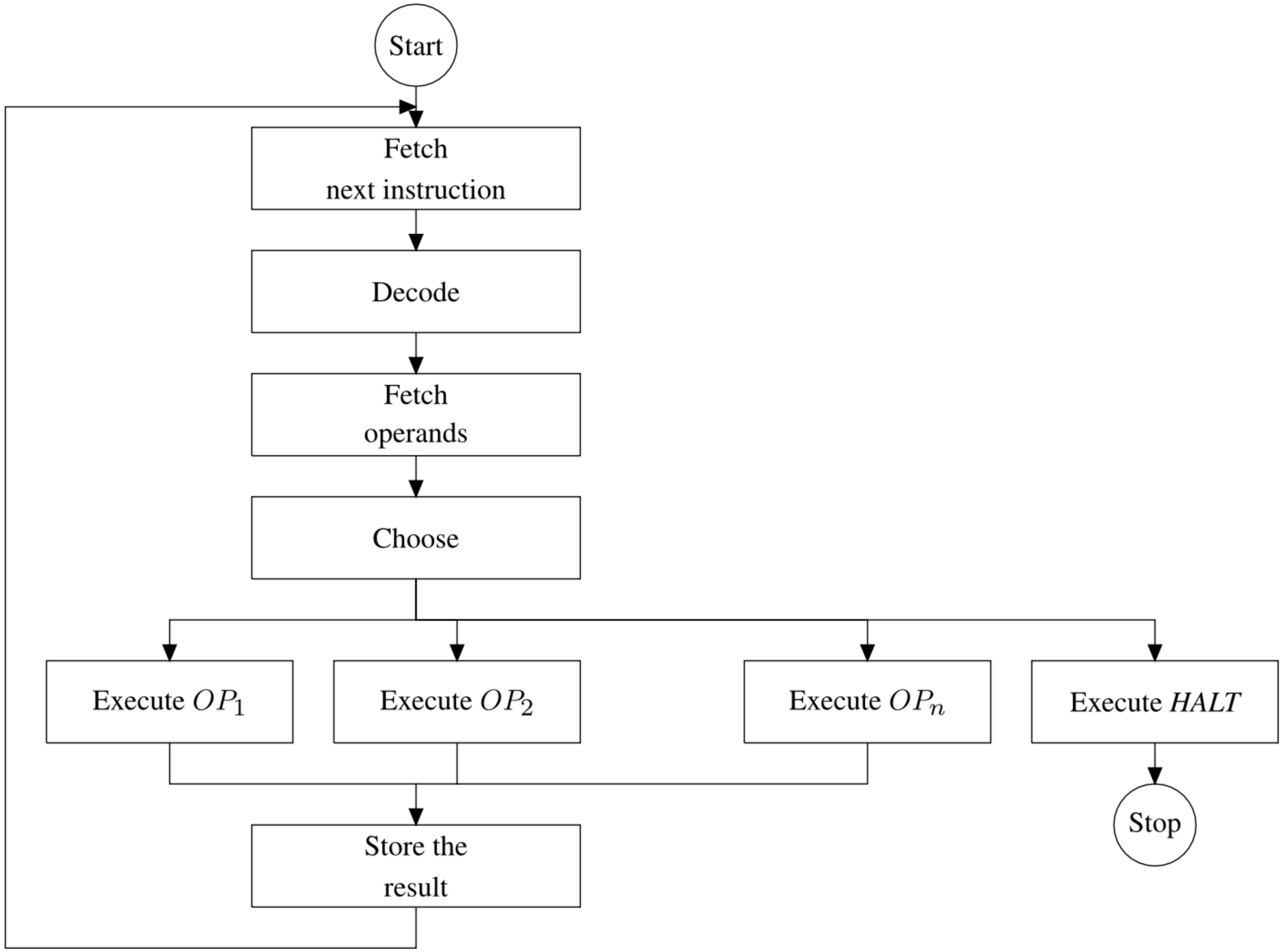
Store programs and associated data in memory.

Support dynamic allocation of memory.

Interpreter

Execution Cycle

- Fetch next instruction from memory
- Fetch operands
- Execute the instruction
- Store the result

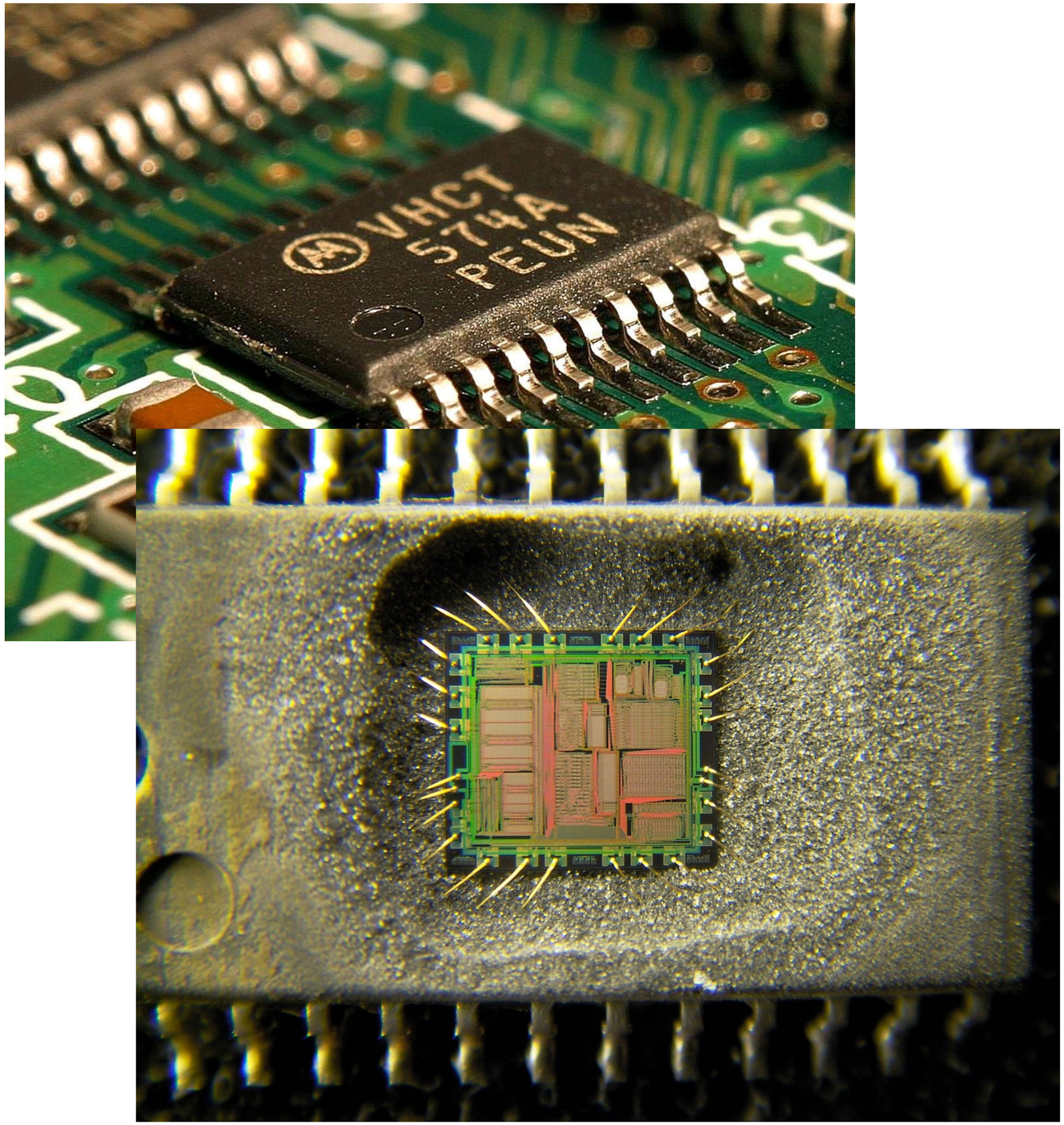


Hardware Machine

- Implemented using logic circuits and electronic components
- Typically supports simple operations

  – `ADD reg1, reg2`

- CPU executing instructions can be viewed as an interpreter



Implementation of an Abstract Machine

- Three options:
  - Implementation in hardware
    - Build custom hardware for a language
  - Simulation using software
    - Write an interpreter
  - Emulation using firmware
    - Use a general-purpose CPU

Pros
- Fast

Cons
- Complicated
- Expensive
- Cannot be modified

Pros
- Flexible

Cons
- Slow

Pros
- Reasonably Fast
- Reasonably Flexible

Cons
- Not as fast or as flexible as other options

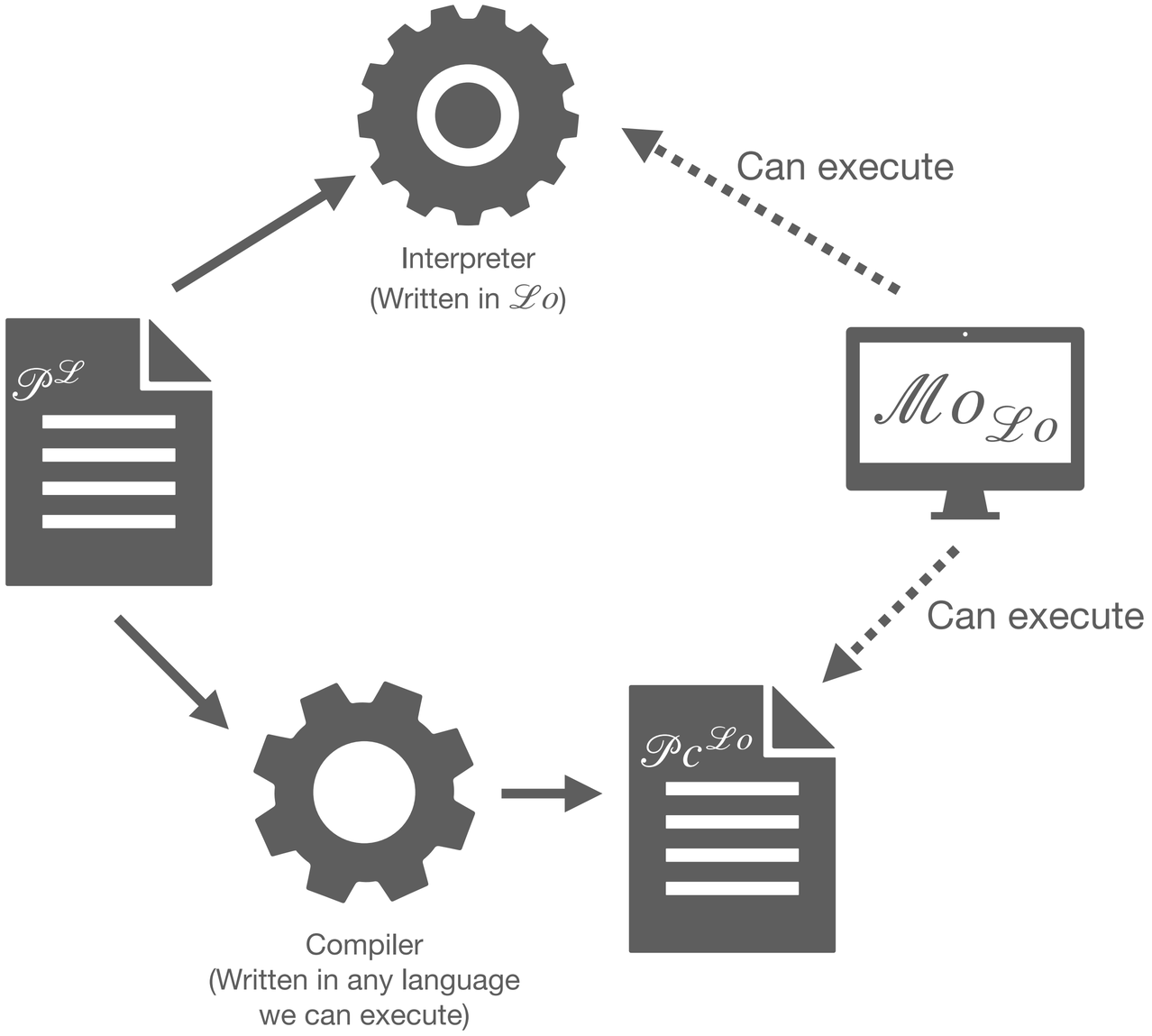
Programs

- A program written in $\mathscr{L}$ can be seen as a ***partial function*** for generic data $\mathscr{D}$:
  - $\mathscr{P}^{\mathscr{L}} : \mathscr{D} \to \mathscr{D}$
  - i.e., $\mathscr{P}^{\mathscr{L}}(\texttt{Input}) = \texttt{Output}$
- The superscript $^{\mathscr{L}}$ indicates that a program (e.g., $\mathscr{P}^{\mathscr{L}}$) is written in language $\mathscr{L}$
- The subscript $_{\mathscr{L}}$ indicates a construct (e.g., $\mathscr{M}_{\mathscr{L}}$) is used for a language $\mathscr{L}$

Implementation

The Ideal Case

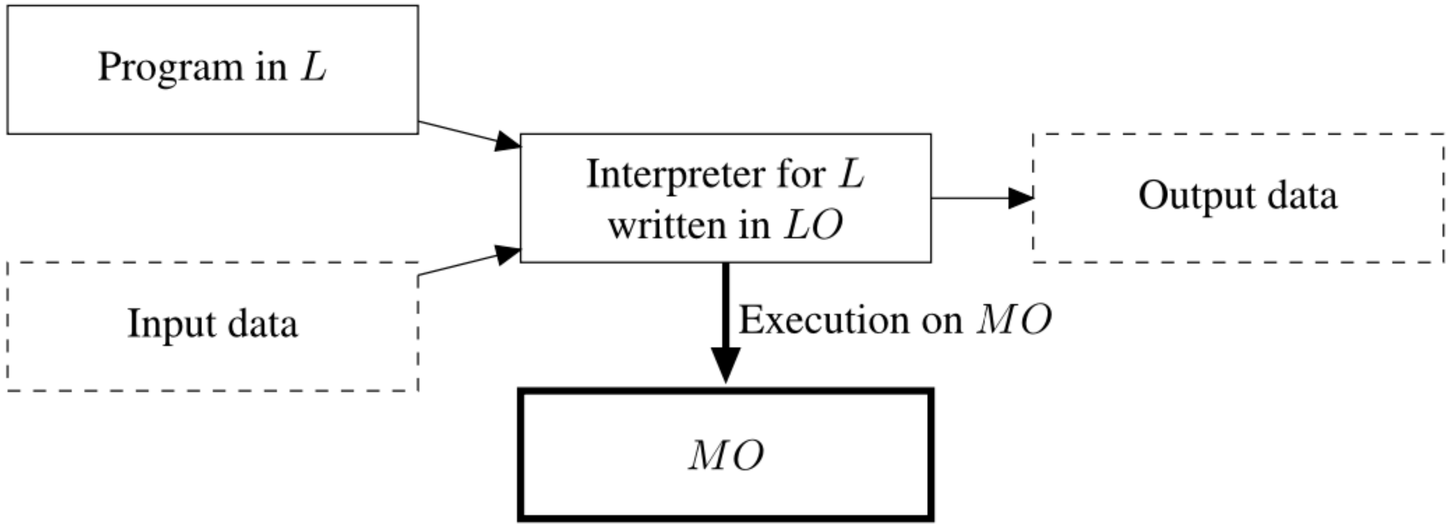
- Consider a generic programming language $\mathscr{L}$ we want to implement
- We don't have an abstract machine $\mathscr{M}_{\mathscr{L}}$
- If we have a ***host machine***, $\mathscr{M}_{\mathscr{L}_O}$, we can run programs written in $\mathscr{L}_O$
- There are two general approaches we could take to execute programs written in $\mathscr{L}$
  - ***Purely interpreted implementation***
  - ***Purely compiled implementation***



Purely Interpreted

The Ideal Form

- Interpreter for $\mathscr{M}_{\mathscr{L}}$ is a program implemented in the language $\mathscr{L}o$
- The interpreter that can execute a program $\mathscr{P}^{\mathscr{L}}$ (written in $\mathscr{L}$)
- In a purely interpreted impl., programs in $\mathscr{L}$ are not explicitly translated
- Execution happens by using instructions in $\mathscr{L}o$ that correspond to an instruction in $\mathscr{L}$

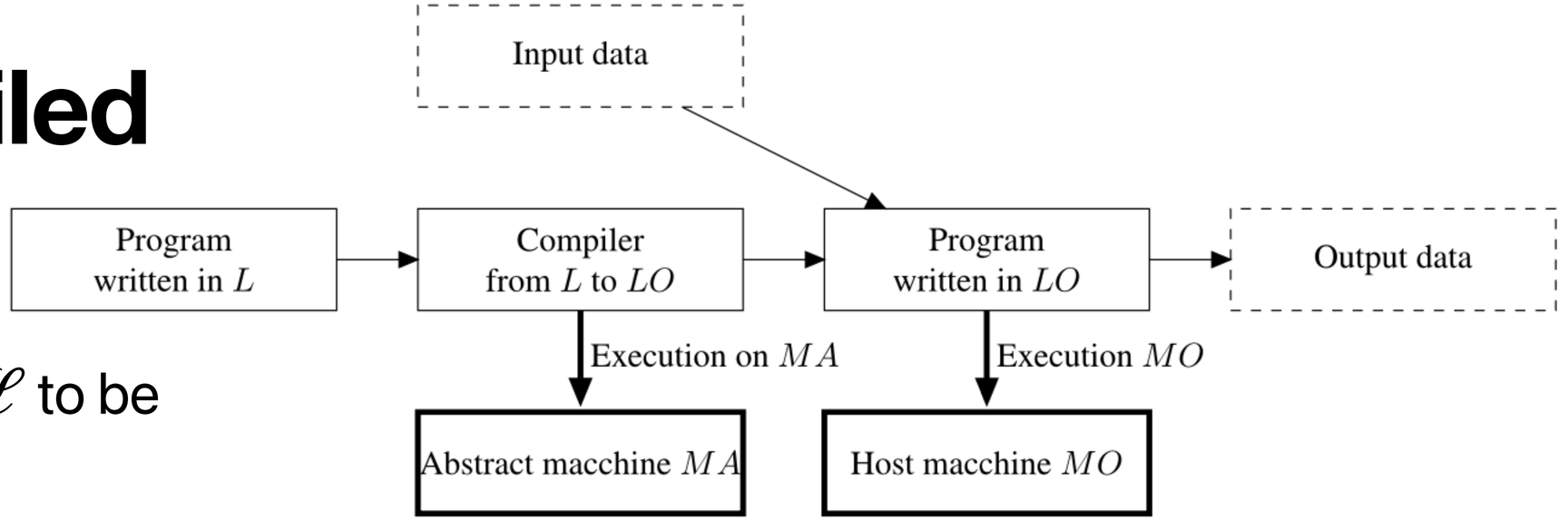


$$\mathscr{I}^{\mathscr{L}o}_{\mathscr{L}} : (\mathscr{P}^{\mathscr{L}} \times \mathscr{D}) \rightarrow \mathscr{D}$$

such that, $\mathscr{I}^{\mathscr{L}o}_{\mathscr{L}}(\mathscr{P}^{\mathscr{L}}, \texttt{Input}) = \mathscr{P}^{\mathscr{L}}(\texttt{Input})$

Purely Compiled

The Ideal Form



- Translate programs in $\mathscr{L}$ to be programs in $\mathscr{L}o$
- A program called a *compiler* $\mathscr{C}_{\mathscr{L},\mathscr{L}o}$ accepts *source language* programs (in $\mathscr{L}$) and translates them to *object language* programs (in $\mathscr{L}o$)
- The compiled program $\mathscr{P}c^{\mathscr{L}o}$ can be run directly on the host machine $\mathscr{M}o_{\mathscr{L}o}$

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

such that, given a program $\mathscr{P}^{\mathscr{L}}$, if

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

then, for every $\texttt{Input} \in \mathscr{D}$:

$$\mathscr{P}^{\mathscr{L}}(\texttt{Input}) = \mathscr{P}c^{\mathscr{L}o}(\texttt{Input})$$

Comparison
Interpreted vs Compiled

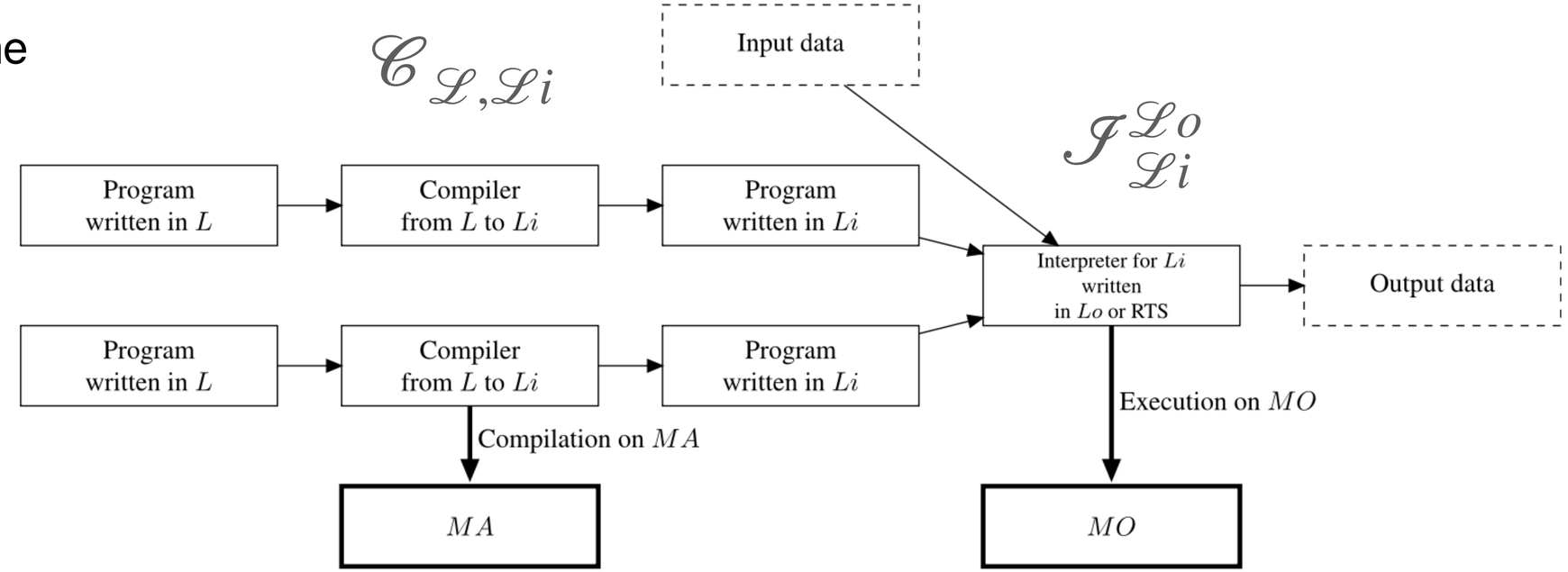
- Purely interpreted
  - Inefficient
    - E.g., parsing must happen during interpretation
  - Flexibility
    - Can easily change the semantics of the language or modify implementation to better support tools/debugging
  - Simpler
    - Often, but not always, an interpreter is simpler to build

- Purely compiled
  - Efficient (at runtime)
    - Compiled program executes natively on $\mathscr{M}o_{\mathscr{L}o}$
  - Information loss
    - Details of the source program are lost
    - E.g., identity of variables in a function
  - More complex
    - Often, but not always, a compiler is more complex to build

Intermediate Machine

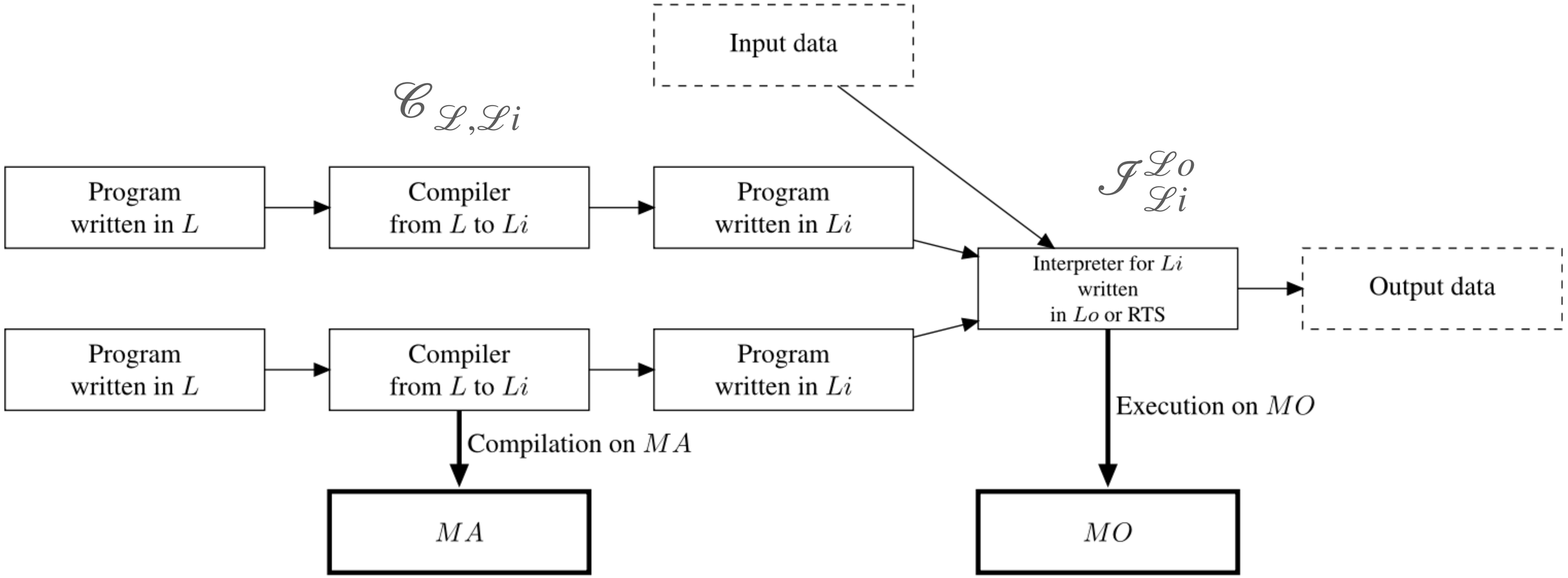
The Real Case

- Purely compiled and purely implemented are the extreme cases
- All real interpreters operate on an intermediate representation (IR)
- All real compiled programs use some simulated constructs
- $\mathscr{L}i$ and $\mathscr{M}i_{\mathscr{L}i}$ are the intermediate language and machine



Intermediate Machine

The Real Case



Hierarchies of Abstract Machines

- Each level implements an abstract machine with its own language
- We are constantly implementing abstract machines
  - A new program can be viewed as an abstract machine
- Software all the way down…until we hit hardware