EECS 390 – Lecture 2

Basic Elements

1/16/24

Digression: Value and Reference Semantics

- ***Value semantics***: a variable is nothing more than a name associated with an object
  - The storage for the variable is the same as that of the object itself
  - The association between a variable and an object cannot be broken as long as the variable is in scope
- ***Reference semantics***: a variable is an indirect reference to an object
  - A variable has its own storage that is distinct from that of the object it refers to
  - A variable can be modified to be associated with a different object

Digression: Reference Semantics

- In a language with reference semantics, variables behave like C/C++ pointers
  - But can't do arithmetic on them

- Example:

```
>>> x = []
>>> y = x
>>> print(id(x))
4546751752
>>> print(id(y))
4546751752
```

Get unique ID of object

C++ equivalent:

```
list *x = new list();
list *y = x;
cout << x << endl;

cout << y << endl;
```

- Python: everything has reference semantics

- Java: primitives have value semantics, everything else has reference semantics

Programming Paradigms

- Languages can be classified in many ways
- A fundamental classification is by what programming paradigms they support
  - Imperative programming
  - Declarative programming
    - Functional programming
    - Logic programming
  - Object-oriented programming

Imperative Programming

- Program decomposed into explicit computational steps in the form of ***statements***
  - A statement executes some operation, generally changing the state of the program
  - Statements (appear to) execute in a well-defined sequence
- Primary paradigm in most commonly used languages (C, C++, Java, Python, etc.)

Declarative Programming

- Expresses computation in terms of *what* it should compute rather than *how*

- ***Functional programming***: models computation after mathematical functions
  - Generally avoids mutation
  - Primary paradigm in the Lisp family (including Scheme), Haskell, ML
  - Some support in C++, Java, Python

- ***Logic programming***: expresses a program in the form of facts and rules
  - Primary paradigm in Prolog, SQL, Make

Type Systems

- All data are represented as bits
- Types determine:
  - What data means
  - What operations are valid on that data
  - How to perform those operations
- ***Static typing*** infers types directly from the source code and checks their use at compile time
- ***Dynamic typing*** tracks and checks types at runtime

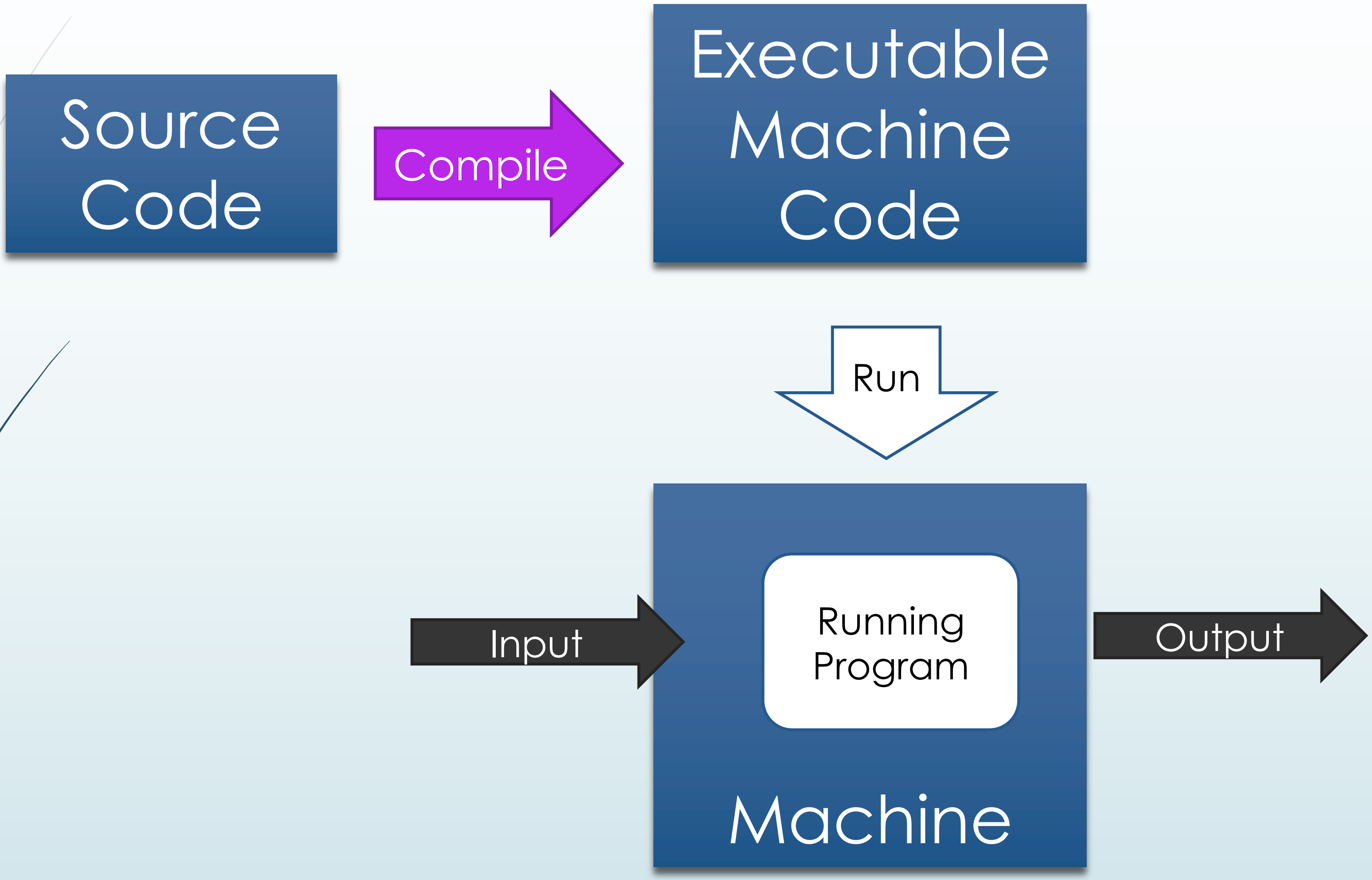
Compilation and Interpretation

- Programs can be compiled, interpreted, or some combination of the two

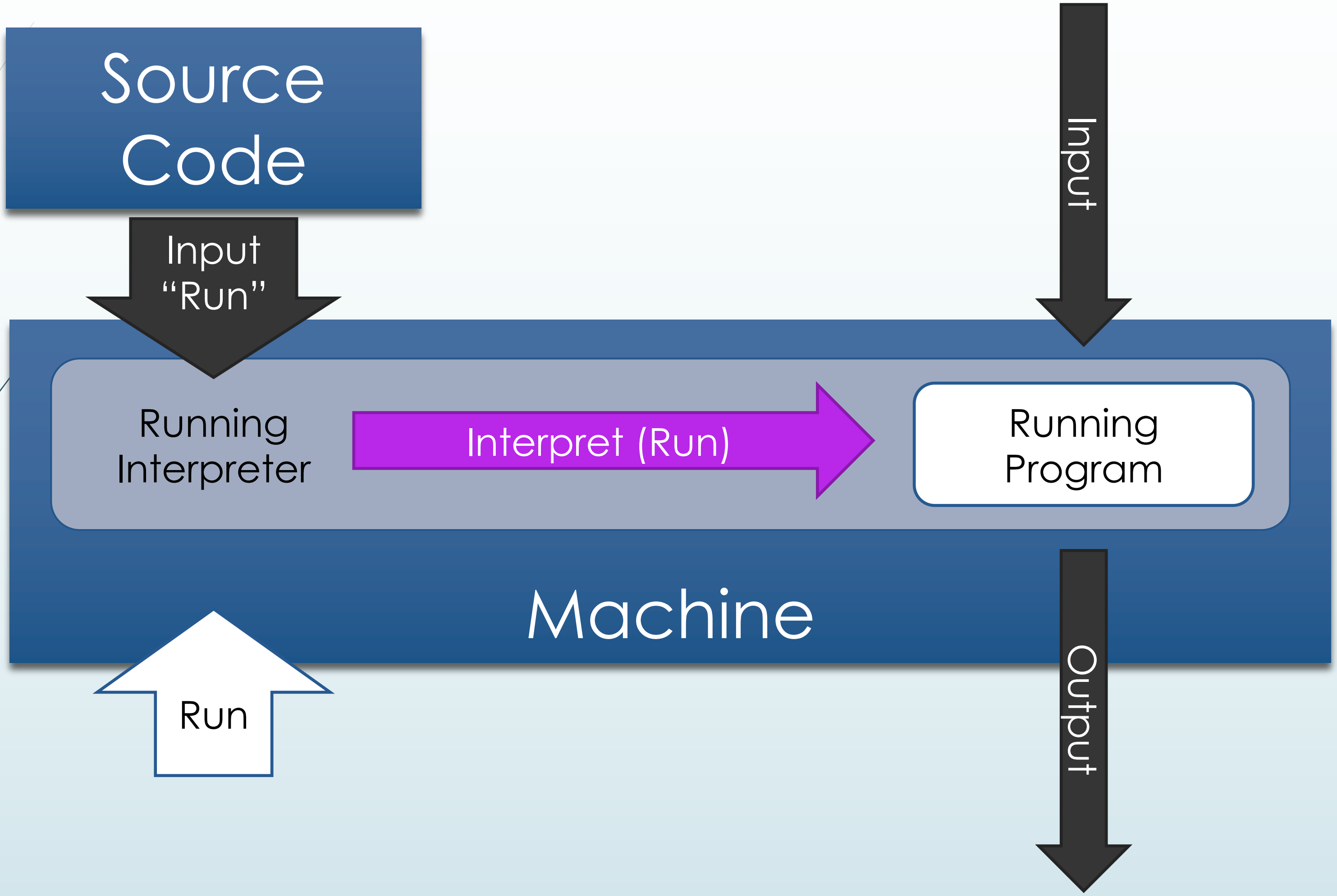
- Compilation: program translated to a form more suitable for execution on a machine
  - Target is often, but not necessarily, machine code

- Interpretation: program is input to interpreter, which interprets and performs the computation it specifies
  - Generally, code is directly interpreted rather than first translated into a different form

Compilation



Interpretation



Compiled vs. Interpreted

Compiled	Interpreted
Faster	Slower
No execution engine	Must go through engine
Less portable	More portable
Must compile for each machine	As long as each machine has an interpreter
Less flexible	More flexible
Program is fixed at compile-time	Program can change at runtime

Hybrids also exist!

Review: Abstraction

- ***Abstraction*** is the idea of using something for *what* it does without needing to know the details of *how* it does it

- Primary tool for managing complexity
  - Facilitates separation of concerns
  - Results in better modularity, maintainability

- However, there can be performance tradeoffs
  - Higher-level abstractions generally do not provide control over how they are implemented

Levels of Description

- ***Grammar***: what phrases are correct
  - ***Lexical structure***: what sequences of symbols represent correct words
  - ***Syntax***: what sequences of words represent correct phrases
- ***Semantics***: what does a correct phrase mean
- ***Pragmatics***: how do we use a meaningful phrase
- ***Implementation***: how are the actions specified by a meaningful phrase accomplished

Lexical Structure

- A ***character set*** is the alphabet of a language
  - e.g. ASCII, Unicode, or subsets thereof
- ***Tokens*** are the "words" in a programming language
  - Smallest element that is meaningful to the compiler or interpreter
  - Lexical analysis is often the first step in interpreting or compiling a program
- A token ends at a character that is invalid for the token, including whitespace
- Types of tokens
  - Literals
  - Identifiers
  - Keywords
  - Operators
  - Separators

Literals and Keywords

- ***Literals*** represent a particular value directly in source code
  - Examples: `3`, `1.4`, `"hello world"`
- Each primitive type often has its own set of literals

- ***Keywords*** are words that have special meaning in the language
  - Examples: `if`, `while`
- In many languages, keywords are reserved and cannot be used for other purposes (e.g. as identifiers)

Identifiers

- Used to name an entity in a program

- The language specifies what characters can be used in an identifier
  - C++
    - First character: _, lowercase and uppercase letters, some escape sequences representing non-ASCII characters
    - Remaining characters: all of the above, plus digits
  - Scheme
    - Allows `! $ % & * + - . / : < = > ? @ ^ _` ~ in identifier!
    - Some implementations are even more permissive

- Something to think about during the break:
  Suppose you want to count the number of bits that are 1 in a very long bitstring (e.g. a **vector<int>**)
- Here's a C++ function to do so:

```
std::size_t count(const std::vector<int> &data) {
  constexpr int num_bits = sizeof(int) * 8;
  std::size_t count = 0;
  for (auto item : data) {
    for (int i = 0; i < num_bits; ++i) {
      count += (item >> i) & 1;
    }
  }
  return count;
}
```

- Can you improve this (by a constant factor)?

Syntax

- Concerned with the *structure* of code fragments
- Specifies what sequences of tokens constitute valid program fragments
  - Example: an expression must have balanced sets of parentheses
- Specified using a formal grammar (future topic!)

Semantics

- Concerned with the _meaning_ of code fragments
  - e.g. what a piece of code defines, what value it computes, or what action it takes
- Further restrict what is valid code
  - Many things are syntactically correct but semantically invalid
- There are formal systems for specifying semantics, but natural language is often used instead

First-Class Entities

- We use ***entity*** to denote something that can be named in a program
  - Other terms also used: ***citizen***, ***object***
  - Examples: types, functions, data objects, values
- A ***first-class entity*** is an entity that supports all operations generally available to other entities
  - e.g. can be assigned to a variable, passed to or returned from a function

	C++	Java	Python	Scheme
Functions	sort of	no	yes	yes
Types	no	no	yes	no
Control	no	no	no	yes

Expressions

- An ***expression*** is a syntactic construct that is *evaluated* to produce a value
  - Examples: `3 + 4`, `foo()`
- Literals are one of the simplest kinds of expressions
  - Evaluate to the value they represent
- An identifier can syntactically be an expression
  - But only semantically valid if it names a first-class entity
  - Evaluates to the entity it names

Data Objects

- Consider the following code. What does the identifier x evaluate to when used as an expression?

```
int x = 3;
... x ...; // x used as an expression
```

Poll: What does x evaluate to?

A) Always the value 3

B) It depends on how x is used

L-Values and R-Values

- An object can have two values associated with it
  - Its location in memory, called its ***l-value***
  - The value that it contains, called its ***r-value***
- Some objects, like temporaries, only have r-values
  - They may not actually exist in memory
- When an expression results in an object that has an l-value, it evaluates to the l-value
- The l-value is implicitly converted to an r-value if necessary

Compound Expressions

- ***Precedence*** and ***associativity*** rules determine how subexpressions are grouped when multiple operators are involved
- Precedence: divides operators into priority groups
  - Example: `{*,/,%}` > `{+,-}`
- Associativity: how operators in the <u>*same*</u> precedence group apply
  - Example: `x = y = 3 + 4 - - - 5`
- Order of evaluation is distinct from precedence and associativity
  - Can be specified (mostly left to right in Python, Java), unspecified (function arguments in Scheme), or partially specified (C++)
  - Example: `cout << ++x << x;`

Statements and Side Effects

- Imperative languages have ***statements***, which are ***executed*** to carry out some action

- Generally have ***side effects***, which change the state of the machine

- Language syntax determines what constitutes a statement and how it is terminated
  - C family: simple statements terminated by semicolon
  - Python: newline (usually) or semicolon (rare)
  - Scheme?

Declarations and Definitions

- A ***declaration*** introduces a name into a program, along with properties about what it names
  - Examples
```
extern int x;
void foo(int, int);
class SomeClass;
```

- A ***definition*** additionally specifies the actual data or code that the name refers to
  - C, C++: definitions are declarations, but a declaration need not be a definition
  - Java: no distinction between definitions and declarations
  - Python: no declarations[1], definitions are statements that are executed

[1] Type annotations are not considered declarations. Quoting from PEP 526: "Type annotations should not be confused with variable declarations in statically typed languages."