EECS 490 – Lecture 1

Introduction and Basic Elements

10/16/17

Essentials

- Google Drive – eecs490.org
  - Syllabus
  - Schedule of Topics
  - All other course materials (slides, assignments, etc.)
- Canvas
- Piazza: piazza.com/umich/fall2017/eecs490
- Calendar: calendar.eecs490.org
- To contact course staff: eecs490staff@umich.edu

Announcements

- Entry survey due Thursday at 8pm
  - survey1.eecs490.org

- Enrollment will be finalized by early next week

- Homework 1 will be released shortly, due 9/15

Agenda

- EECS 490 Overview
- Logistics
- Introduction to Programming Languages
- Basic Elements

Course Purpose

- Purpose is **<u>not</u>** to teach you:
  - A bunch of different languages
  - The esoteric details of a particular language
- Instead, it covers general concepts in programming languages that are applicable to many languages
  - Analogous to *linguistics* rather than specific languages
- End goals
  - Be able to quickly learn a new language
  - Make better use of the programming constructs and paradigms provided by a language

Course Description

- Official course description:

  "Fundamental concepts in programming languages. Course covers different programming languages including functional, imperative, object-oriented, and logic programming languages; different programming language features for naming, control flow, memory management, concurrency, and modularity; as well as methodologies, techniques, and tools for writing correct and maintainable programs."

- Other topics:
  - Basic language theory (e.g. grammars, type systems)
  - Advanced programming techniques (e.g. metaprogramming)

EECS 490 is in Beta

- This is only the second offering in the last 10 years
- We're still working on improving things
  - Two new projects
  - Updates to existing projects
  - Autograder
  - 2x the enrollment
- Things will be better than last year, but not perfect
- There will (hopefully) be compensations
  - You get to learn a lot of cool things about languages
  - Your experience and feedback will shape the course for the future
  - The grading curve will be somewhat higher than other courses

Course Staff

- Instructor: Amir Kamil
- TAs: Holly Borla and Madeline Endres
- See the Staff Profiles doc

Course Notes and Textbook

- Course notes on the Google Drive covering all the material
  - **<u>Required reading</u>** (unless a section is explicitly marked optional)
  - Will be updated throughout the term; check timestamp
- Recommended text: *Programming Languages: Principles and Paradigms*, by Gabbrielli and Martini
  - Available in both print and electronic form
  - Reading assignments on schedule of topics

Exams and Grades

- Grades will be curved.
- Midterm Exam
  Tue. 10/31, in class
- Final Exam
  Thu. 12/21, 10:30am-12:30pm
- Check for conflicts NOW.
- **More?** See Syllabus.

Homework	15%
Projects	40%
Midterm	20%
Final	24%
Participation	1%

Assignments

- Five homework assignments
  - Smaller programming exercises and written-response questions
- Five programming projects
  - Larger programming exercises to gain deeper understanding of important PL concepts
- Assignments will be submitted to the autograder and Gradescope
- See schedule of topics for due dates
- **All deadlines are at <u>8pm</u>**

Projects

- P1: shorter project for practicing Python, reviewing abstract data types (ADTs) and object-oriented programming
- P2: Scheme parser, written in Scheme
- P3: Scheme interpreter, written in Python
- P4: uC static analyzer, written in Python
- P5: uC code generator, written in Python and generating C++

Project	Weight	Due
P1	4%	9/20
P2	9%	10/6
P3	9%	10/27
P4	9%	11/21
P5	9%	12/12

Collaboration

- Homework
  - May discuss approaches to problems with up to 3 other students
  - Must write actual solutions on your own
  - Acknowledge who you discussed the assignment with when you turn it in
- Projects
  - Must be done individually

Office Hours and Piazza

- Check calendar for office hours
- To ensure fair access, we will not help anyone individually outside of class sessions and office hours
- Outside of office hours, post questions on Piazza

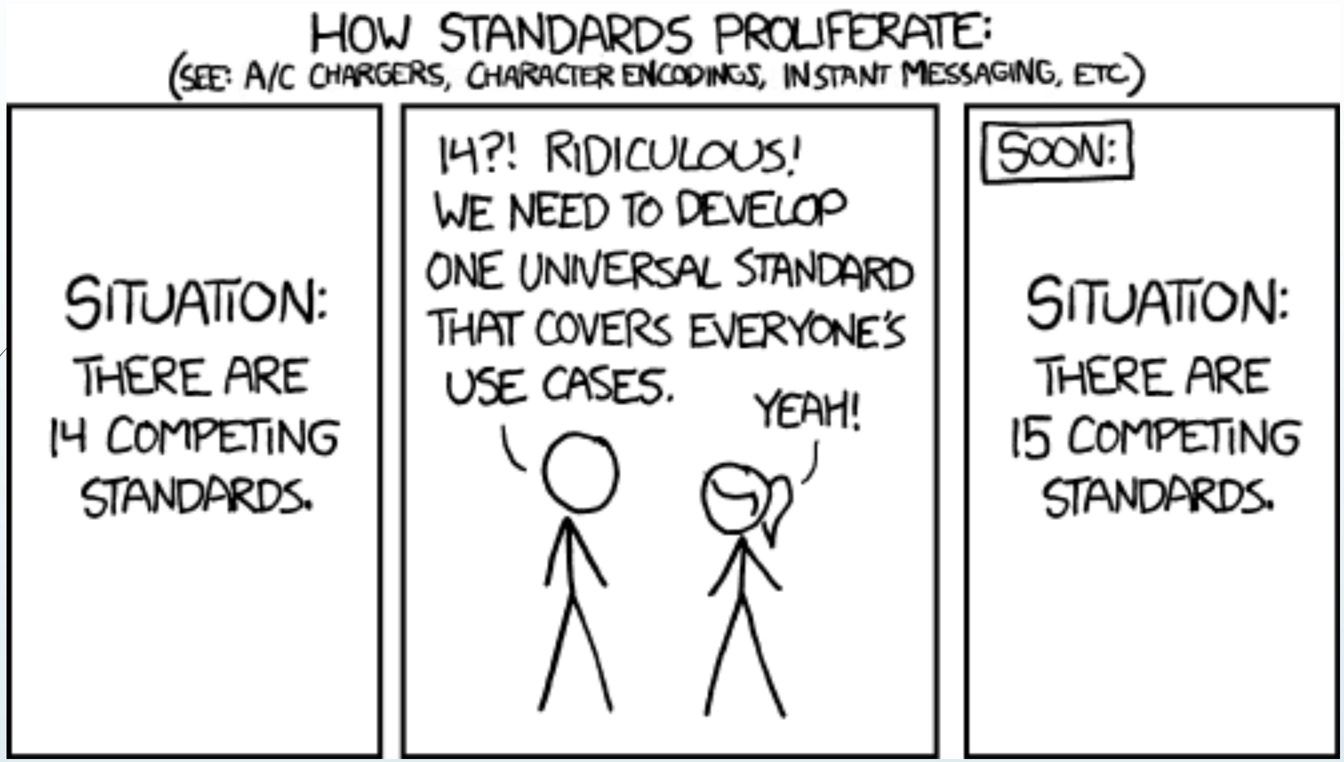
Programming Languages

- Designed for expressing computation at a higher level than machine code
  - Provide a view of computation that is *abstracted* from that of the machine
- Facilitate writing, reading, and maintaining code
- Provide abstractions for common programming patterns
- A common base for modules written by different programmers

Turing Completeness

- A language is *Turing complete* if it can compute the same functions as a Turing machine
  - Church-Turing thesis: all functions that can be computed by humans can be computed by a Turing machine
- All general-purpose languages are Turing complete
- However, languages differ in the abstractions they provide, their performance, etc.

One Language to Rule Them All?



https://xkcd.com/927/

Language Design Goals

- Some language design goals
  - Ease of writing
  - Ease of reading
  - Maintainability
  - Reliability and safety
  - Performance
  - Modularity
  - Portability
- These goals are often in conflict with each other
  - "There are no solutions; there are only trade-offs."
    – Thomas Sowell

Problem Domains

- Languages are often well-suited to a particular problem domain
  - Shell scripting: Bash
  - High-performance numerical codes: Fortran
  - Writing documents: Latex
  - Build automation and dependency tracking: Make
  - Web programming: Javascript
  - Systems programming: C
  - Etc.

- A programmer should use the right tool for the job

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

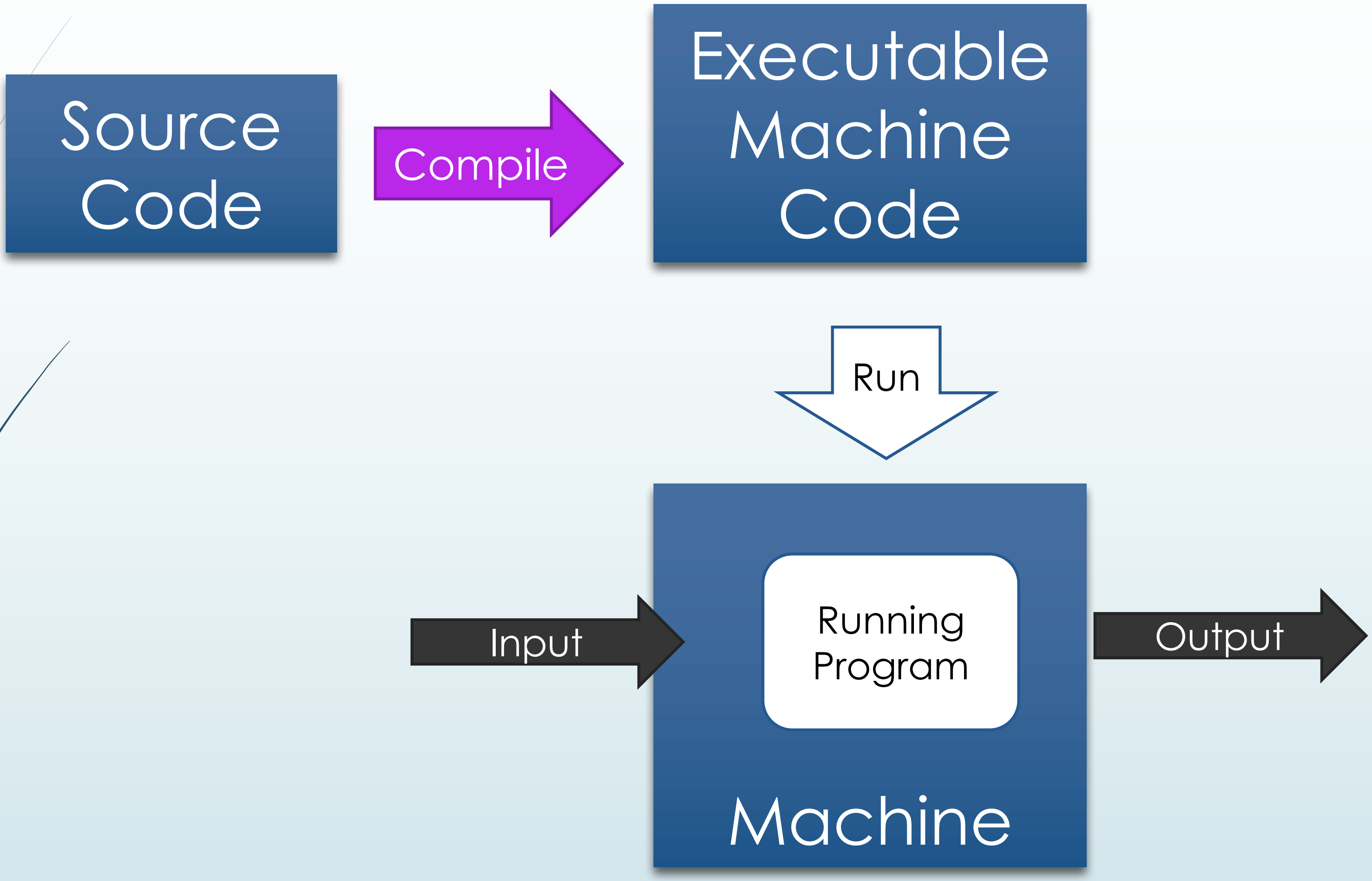
x = 3
type(x)

- 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

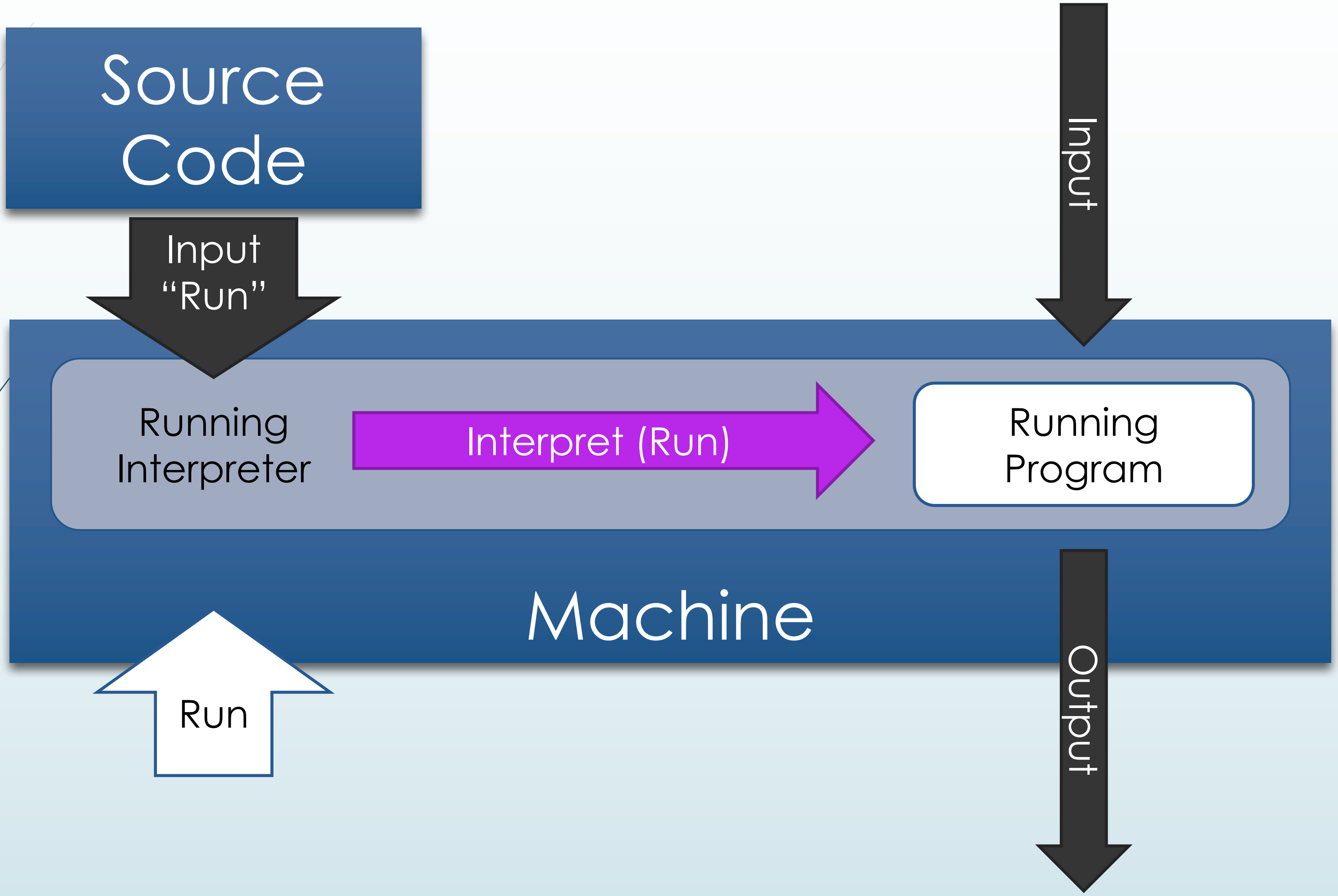
Compilation and Interpretation

- 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

- Faster
  - No execution engine
- Less portable
  - Must compile for each machine
- Less flexible
  - Program is fixed at compile-time

Interpreted

- Slower
  - Must go through engine
- More portable
  - As long as each machine has an interpreter
- More flexible
  - Program can change at runtime

Hybrids also exist!

Break Time!

- We'll start again in five minutes.

Review: Abstraction

- *Abstraction* is the idea of using something for **what** it does without need 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

- Represent a particular value directly in source code
  - Examples: `3`, `1.4`, `"hello world"`
- Each primitive type often has its own set of literals
  - Multiple sets of literals can be provided for a single type
    - e.g. binary, octal, decimal, hexadecimal integers

1.3

1.3e10

"10111"_bv

"hello\n world"

Identifiers

- Used to name an entity in a program

  int x;

- The language specifies what characters can be used in an identifier
  - Often special rules for first character
  - 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

Keywords

- Identifiers that have special meaning in the language
  - Examples: `if`, `while`
- In many languages, keywords are reserved and cannot be used as an identifier

if + 1

- Other languages interpret keywords based on context
- Some languages such as Scheme don't really have keywords

(define define 3)

Operators

- Tokens that specify a specific operation
  - Examples: +, ==, ->
- Some languages, such as Scheme, do not have operators
- Operators are often grouped with separators, particularly if a token can be either depending on context
  - Examples: parentheses and commas in C++

func(...)

(3 + 4) * 5

Separators

- The punctuation of a language
- Also called *delimiters* or *punctuators*
- Denote the boundary between different constructs or components of a construct
  - Examples: { and } in C++

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!)

int x;

1 * x y;

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

```
int x;
x y;
x = "abc";
```

First-Class Entities

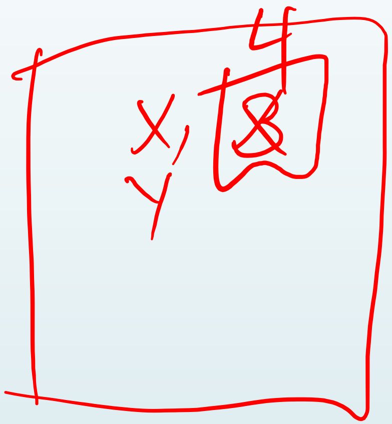
- We use *entity* to denote something that can be named in a program
  - Other terms also used: *citizen*, *object*
  - Examples: types, functions, data objects, values
- A *first-class entity* is an entity that supports all operations generally available to other entities
  - e.g. can be assigned to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

Objects and Variables

- An *object* is a location in memory that holds a value
- A *variable* is a name paired with an object
  - Two variables that name the same object *alias* the object

```
int x = 3;
int &y = x;
y = 4;
cout << x;
```



Lifetime and Scope

- An object has a *lifetime* during which it is legal to use that object
  - Can be managed by the compiler/interpreter/runtime or by the programmer
- A variable has a *scope* that specifies the region of a program that has access to that variable
- More on these topics in the next few lectures

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

~~int~~;   string;   int x=3;
x;

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
```

cout << x;

x = 4;

int z = x;

4

x $\boxed{\cancel{3}}$

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

```
int &y = x;
cout << x;
```

Compound Expressions

- Consist of multiple subexpressions combined according to the rules of the language

- Example: operators
```
3 + 4
x = y & 0x3
```

- Example: function calls
```
print("Hello", "world")
```

Function Calls

- A function call evaluates to the return value produced when running the function

```
int add1(int x) {
  return x + 1;
}

int x = add1(3) - 2;
```

- What about the following function call?

```
void foo();

foo()   // is it an expression?
```

Precedence

- Rules determine how subexpressions are grouped when multiple operators are involved
  - Example: `3 + 4 * 5`
- C++ order
  1. Scope resolution operator (`::`)
  2. Postfix increment/decrement, function calls, subscript, member access (`.`, `->`)
  3. Unary prefix operators (`-`, `!`, `&`, `new`, `delete`)
  4. Pointer-to-member operators (`.*`, `->*`)
  5. Multiplication, division, remainder
  6. Addition, Subtraction
  7. Shift operators

  …

Associativity

- Rules determine how subexpressions are grouped when multiple operators **with the same precedence** are involved
  - Example: `8 / 2 / 2`
  - Example: `a = b = c`

- C++ rules
  - Unary prefix operators are right-to-left
  - Assignment operators and ternary conditional (`a?b:c`) are right-to-left
  - Everything else is left-to-right

Order of Evaluation

- Precedence and associativity determine how subexpressions are grouped, but not in what order they are evaluated
- Python, Java, Scheme: subexpressions evaluated in order from left to right
  - Exception: assignment in Python
- C, C++: Order largely undefined

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?

Simple Statements

- Expression statements consist of just an expression
  - Examples

```
x + 1;
x = 3;
foo(1, 2, 3);
a[3] = 4;
```

- Other simple statements
  - return
  - break
  - continue
  - goto

int x = 3;

const

void foo(int &x);

foo(3) X √

foo(x+4) X √

Compound Statements

- Composed of multiple subexpressions or statements
  - Blocks
  - Conditionals
  - Loops
  - Try/catch
  - Function and class definitions in Python

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;
```

```
void foo(int x, int y){
  cout << (x+y);
}
```

- 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, definitions are statements that are executed