

# Introduction

---

## Design of Programming Languages

---

# Discussion: Programming Languages

Your experience:

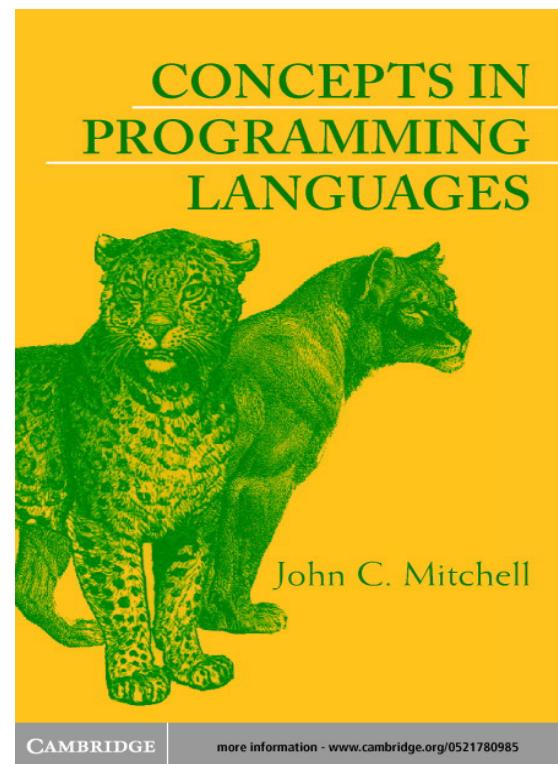
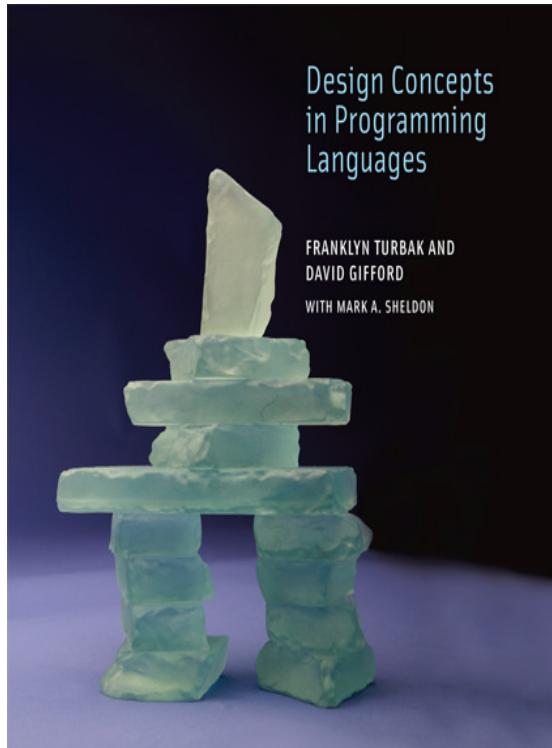
- What PLs have you used?
- Which PLs/PL features do you like/dislike. Why?

More generally:

- What is a PL?
- Why are new PLs created?
  - What are they used for?
  - Why are there so many?
- Why are certain PLs popular?
- What goes into the design of a PL?

# Textbooks

- *Design Concepts in Programming Languages, F. Turbak & D. Gifford*
- *Concepts in Programming Languages, J. Mitchell*



# General Purpose PLs

Java

Python

Perl

Fortran

ML

JavaScript

Racket

C/C++

Ruby

Haskell

*CommonLisp*

# Domain Specific PLs

*Excel*

HTML

CSS

**OpenGL**

R

**LaTeX**

Matlab

IDL

**Swift**

PostScript

# Programming Languages: Mechanical View

A computer is a machine. Our aim is to make the machine perform some specified actions. With some machines we might express our intentions by depressing keys, pushing buttons, rotating knobs, etc. For a computer, we construct a sequence of instructions (this is a "program") and present this sequence to the machine.

– *Laurence Atkinson, Pascal Programming*

# Programming Languages: Linguistic View

A computer language ... is a novel formal medium for expressing ideas about methodology, not just a way to get a computer to perform operations. Programs are written for people to read, and only incidentally for machines to execute.

– *Harold Abelson and Gerald J. Sussman*

# “Religious” Views

The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense. – *Edsger Dijkstra*

It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration. – *Edsger Dijkstra*

You're introducing your students to programming in C? You might as well give them a frontal lobotomy! – *A colleague of mine*

A LISP programmer knows the value of everything, but the cost of nothing.  
- *Alan Perlis*

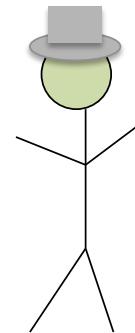
I have never met a student who cut their teeth in any of these languages and did not come away profoundly damaged and unable to cope. I mean this reads to me very similarly to teaching someone to be a carpenter by starting them off with plastic toy tools and telling them to go sculpt sand on the beach. - *Alfred Thompson, on blocks languages*

A language that doesn't affect the way you think about programming, is not worth knowing. - *Alan Perlis*

# Which Programming/PL Hat do You Wear?

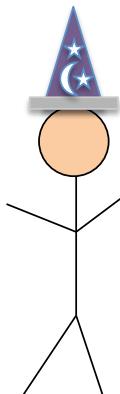
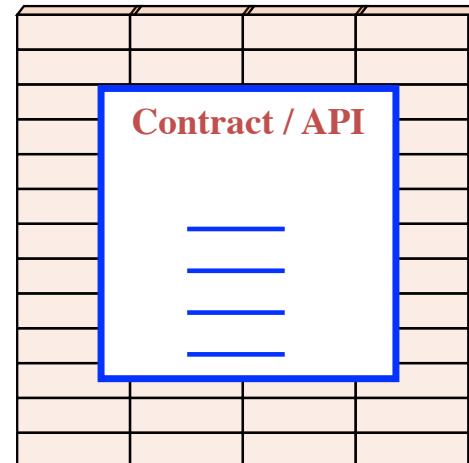


Function &  
Data Abstraction  
User / Client

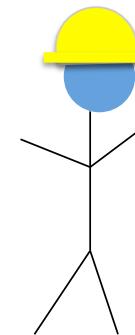


Programming Language  
Designer

Big idea #1: Abstraction



Function &  
Data Abstraction  
Implementer



# Programming Paradigms

- ***Imperative*** (e.g. *C, Python*): Computation is step-by-step execution on a stateful abstract machine involving memory slots and mutable data structures.
- ***Functional, function-oriented*** (e.g *Racket, ML, Haskell*): Computation is expressed by composing functions that manipulate immutable data.
- ***Object-oriented*** (e.g. *Simula, Smalltalk, Java*): Computation is expressed in terms of stateful objects that communicate by passing messages to one another.
- ***Logic-oriented*** (e.g. *Prolog*): Computation is expressed in terms of declarative relationships.

**Note:** In practice, most PLs involve multiple paradigms. E.g.

- Python supports functional features (map, filter, list comprehensions) and objects
- Racket and ML have imperative features.

# Paradigm Example: Quicksort

```
void qsort(int a[], int lo, int hi) {  
    int h, l, p, t;  
  
    if (lo < hi) {  
        l = lo;  
        h = hi;  
        p = a[hi];  
  
        do {  
            while ((l < h) && (a[l] <= p))  
                l = l+1;  
            while ((h > l) && (a[h] >= p))  
                h = h-1;  
            if (l < h) {  
                t = a[l];  
                a[l] = a[h];  
                a[h] = t;  
            }  
        } while (l < h);  
  
        a[hi] = a[l];  
        a[l] = p;  
  
        qsort( a, lo, l-1 );  
        qsort( a, l+1, hi );  
    }  
}
```

```
quicksort :: Ord a => [a] -> [a]  
quicksort []      = []  
quicksort (p:xs) =  
    (quicksort lesser)  
    ++ [p]  
    ++ (quicksort greater)  
where  
    lesser  = filter (< p) xs  
    greater = filter (>= p) xs
```



Functional Style (in Haskell)



Imperative Style  
(in C; Java would be similar)

# PL Dimensions

PLs differ based on decisions language designers make in many dimensions. E.g.:

- ***First-class values***: what values can be named, passed as arguments to functions, returned as values from functions, stored in data structures.  
Which of these are first-class in your favorite PL: arrays, functions, variables?
- ***Naming***: Do variables/parameters name expressions, the values resulting from evaluating expressions, or mutable slots holding the values from evaluating expressions? How are names declared and referenced? What determines their scope?
- ***State***: What is mutable and immutable; i.e., what entities in the language (variables, data structures, objects) can change over time.
- ***Control***: What constructs are there for control flow in the language, e.g. conditionals, loops, non-local exits, exception handling, continuations?
- ***Data***: What kinds of data structures are supported in the language, including products (arrays, tuples, records, dictionaries), sums (options, oneofs, variants), sum-of-products, and objects.
- ***Types***: Are programs statically or dynamically typed? What types are expressible?

# Why study PL?

- Crossroads of CS
- Approach problems as a *language designer*.
  - *"A good programming language is a conceptual universe for thinking about programming"*  
-- Alan Perlis
  - Evaluate, compare, and choose languages
  - Become better at learning new languages
  - become a better problem-solver
  - view API design as language design
- Ask:
  - Why are PLs the way they are?
  - How could they (or couldn't they) be better?
  - What is the cost-convenience trade-off for feature X?

# PL Parts

## Syntax: *form* of a PL

- What a P in a given L look like as symbols?
- Concrete syntax vs abstract syntax trees (ASTs)

## Semantics: *meaning* of a PL

- *Static Semantics*: What can we tell about P before running it?
  - Scope rules: to which declaration does a variable reference refer?
  - Type rules: which programs are well-typed (and therefore legal)?
- *Dynamic Semantics*: What is the behavior of P? What actions does it perform? What values does it produce?
  - Evaluation rules: what is the result or effect of evaluating each language fragment and how are these composed?

## Pragmatics: *implementation* of a PL (and PL environment)

- How can we evaluate programs in the language on a computer?
- How can we optimize the performance of program execution?

# Syntax (Form) vs. Semantics (Meaning) in Natural Language

Furiously sleep ideas green colorless.

Colorless green ideas sleep furiously.

Little white rabbits sleep soundly.

# Concrete Syntax: Absolute Value Function

**Logo:** to abs :n ifelse :n < 0 [output (0 - :n)] [output :n] end

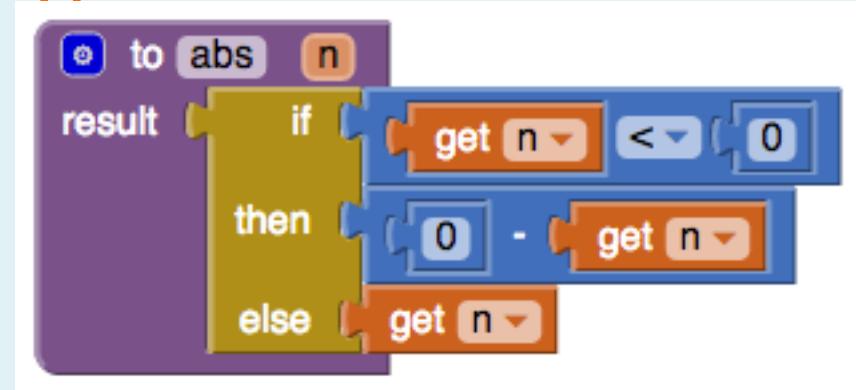
**Javascript:** function abs (n) {if (n < 0) return -n; else return n;}

**Java:** public static int abs (int n) {if (n < 0) return -n; else return n;}

**Python:**

```
def abs(n):
    if n < 0:
        return -n
    else:
        return n
```

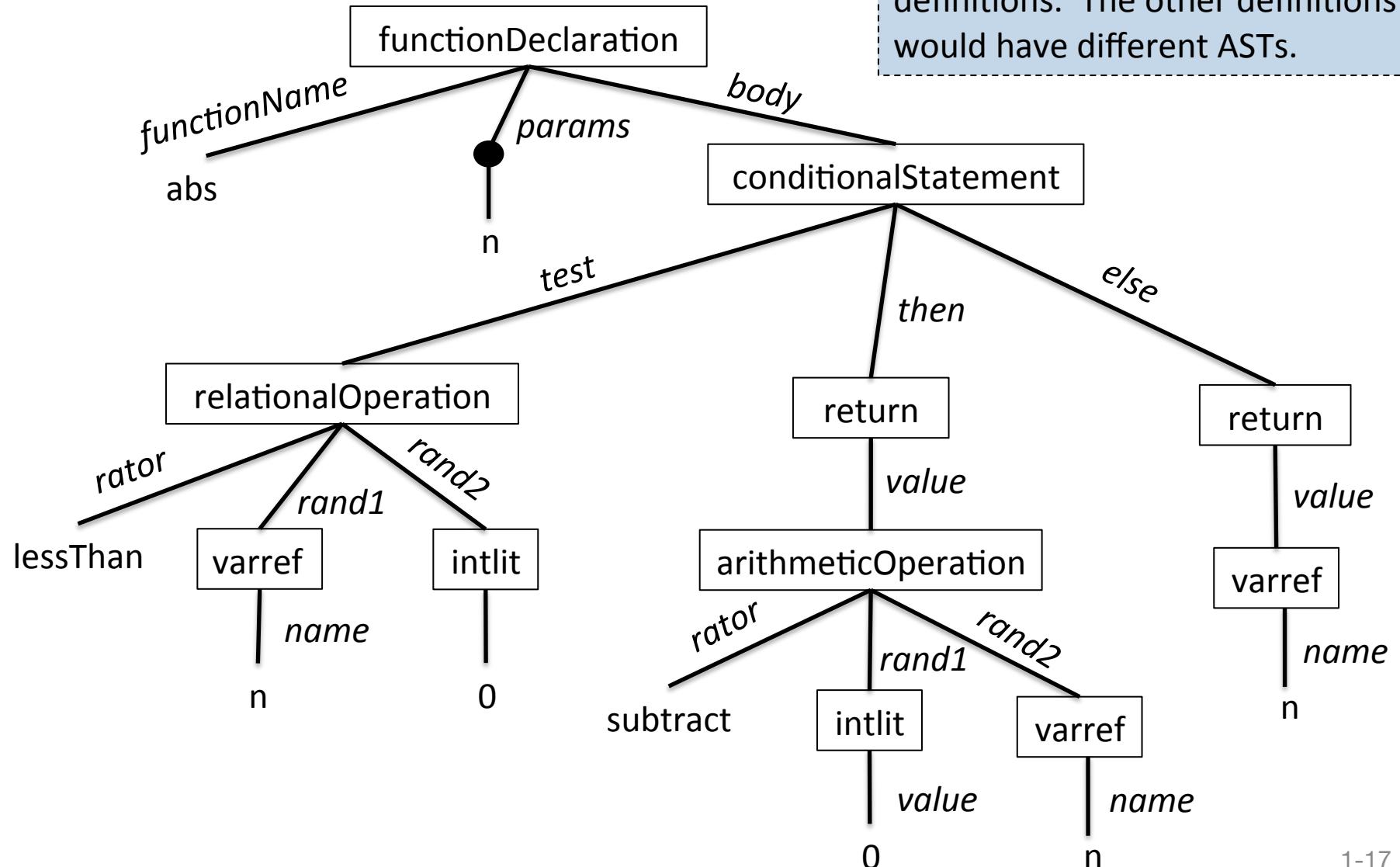
**App Inventor:**



**Scheme:** (define abs (lambda (n) (if (< n 0) (- n) n)))

**PostScript:** /abs {dup 0 lt {0 swap sub} if} def

# Abstract Syntax Tree (AST): Absolute Value Function



This AST abstracts over the concrete syntax for the Logo, JavaScript, and Python definitions. The other definitions would have different ASTs.

# Dynamic Semantics Example 1

What is the meaning of the following expression?

$$(1 + 11) * 10$$

# Dynamic Semantics Example 1

What is the meaning of the following expression?

$$(1 + 11) * 10$$

Some possible answers:

- 120 (regular interpretation of numbers, operators)
- 1000 (binary numbers, regular operators)
- 0 (“+” means “minus”, “\*” means “plus”)
- 13 (number of characters in string)
- 5 (number of nodes in AST)
- 3 (number of leaves in AST)

# Dynamic Semantics Example 2

What is printed by the following program?

```
a = 1;  
b = a + 20;  
print(b);  
  
a = 300  
print(b);  
  
count = 0;  
  
fun inc() { count = count + 1; return count; }  
  
fun dbl(ignore, x) { return x + x; }  
  
print(dbl(inc(), inc()))
```

# Dynamic Semantics Example 2

What is printed by the following program?

```
a = 1;  
b = a + 20;  
print(b);  
a = 300  
print(b);  
  
count = 0;  
  
fun inc() { count = count + 1; return count; }  
  
fun dbl(ignore, x) { return x + x; }  
  
print(dbl(inc(), inc()))
```

Here are some possible answers. What execution models give rise to these answers?

21  
21  
4

21  
21  
2

21  
320  
3

21  
320  
2

# Dynamic Semantics Example 3

Suppose `a` is an array (or list) containing the three integer values 10, 20, and 30 in the following languages. What is the meaning of the following expressions/statements in various languages (the syntax might differ from what's shown).

	<code>a[1]</code>	<code>a[3]</code>	<code>a[2] = "foo"</code>	<code>a[3] = 17</code>
Java				
C				
Python				
JavaScript				
Pascal				
App Inventor				

How do you determine the answers?

# Dynamic Semantics Example 3

Suppose `a` is an array (or list) containing the three integer values 10, 20, and 30 in the following languages. What is the meaning of the following expressions/statements in various languages (the syntax might differ from what's shown).

	<code>a[1]</code>	<code>a[3]</code>	<code>a[2] = "foo"</code>	<code>a[3] = 17</code>
Java	20	dynamic index out of bounds error	static type error	dynamic index out of bounds error
C	20	returns value in memory slot after <code>a[2]</code>	static type error	Stores 17 in memory slot after <code>a[2]</code>
Python	20	dynamic list index out of range error	stores “foo” in third slot of <code>a</code>	dynamic list index out of range error
JavaScript	20	“undefined” value	stores “foo” in third slot of <code>a</code>	Stores 17 in <code>a[3]</code>
Pascal	20	static index out of bounds error	static type error	static index out of bounds error
App Inventor	10	30	stores “foo” in second slot of <code>a</code>	Stores 17 in third slot of <code>a</code>

How do you determine the answers?

**Try in implementation; consult documentation**

# Static Semantics Example 1: Type Checking

Which of the following Java examples can be well-typed (i.e., pass the type checker)? How do you know? What assumptions are you making?

A `2 * (3 + 4)`

B `2 < (3 + 4)`

C `2 < True`

D `if (a < b) {  
 c = a + b;  
} else {  
 c = a * b;  
}`

E `if (a < b) {  
 c = a + b;  
} else {  
 c = a > b;  
}`

F `if (a) {  
 c = a + b;  
} else {  
 c = a * b;  
}`

G `public boolean f(int i, boolean b) {  
 return b && (i > 0);  
}`

H `public int g(int i, boolean b) {  
 return i * (b ? 1 : -1);  
}`

I `public int p(int w) {  
 if (w > 0) { return 2*w; }  
}`

J `public int q(int x) { return x > 0; }`

K `public int r(int y) { return g(y, y>0); }`

L `public boolean s(int z) { return f(z); }`

# Static Semantics Example 1: Type Checking

Which of the following Java examples can be well-typed (i.e., pass the type checker)? How do you know? What assumptions are you making?

Green indicates statically well-typed; Red indicates static type error.

A `2 * (3 + 4)`

B `2 < (3 + 4)`

C `2 < True`

D Assume a,b,c are ints  
if (a < b) {  
 c = a + b;  
} else {  
 c = a \* b;  
}

E Assume a,b are ints;  
can't find type for c  
if (a < b) {  
 c = a + b;  
} else {  
 c = a > b;  
}

F A can't both be  
bool for test and  
int for +/\*  
`if (a) {  
 c = a + b;  
} else {  
 c = a * b;  
}`

G `public boolean f(int i, boolean b) {  
 return b && (i > 0);  
}`

H `public int g(int i, boolean b) {  
 return i * (b ? 1 : -1);  
}`

I `public int p(int w) {  
 if (w > 0) { return 2*w; }  
} No else clause, so no return of in when w <= 0`

J Return type needs to be boolean, not int  
`public int q(int x) { return x > 0; }`

K Assume g has type from above: `g : (int, boolean) -> int`  
`public int r(int y) { return g(y, y>0); }`

L If f has type `(int, boolean) -> boolean`, not well-typed;  
But \*would\* be well typed if `f : (int) -> boolean`  
`public boolean s(int z) { return f(z); }`

## Static Semantics Example 2: Detecting Loops

Which of these Python programs has inputs for which it loops forever?

```
def f(x):
    return x+1
```

```
def g(x):
    while True:
        pass
    return x
```

```
def h(x):
    while x > 0:
        x = x+1
    return x
```

```
def g2(x):
    return g2(x)
```

```
def h2(x):
    if x <= 0:
        return x
    else:
        return h2(x+1)
```



```
def collatz(x):
    while x != 1:
        if (x % 2) == 0:
            x = x/2
        else:
            x = 3*x + 1
    return 1
```

# Static Semantics Example 2: Detecting Loops

Which of these Python programs has inputs for which it loops forever?

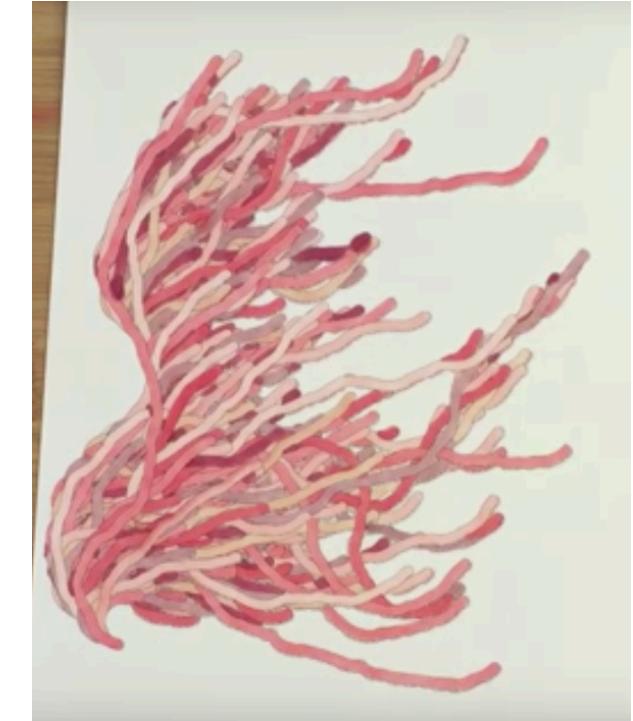
```
def f(x):
    return x+1
No ∞ loop on any input
```

```
def g(x):
    while True:
        pass
    return x
∞ loop on all inputs
```

```
def h(x):
    while x > 0:
        x = x+1
    return x
No ∞ loop for x <= 0.
∞ loop for x > 0, assuming arbitrarily large numbers.
In practice, will either run out of memory when x gets too big, or will wrap to negative and halt.
```

```
def g2(x):
    return g2(x)
∞ recursion on all inputs.
In practice, runs out of stack space in Python. Similar Racket program is true ∞ loop due to proper tail recursion.
```

```
def h2(x):
    if x <= 0:
        return x
    else:
        return h2(x+1)
No ∞ loop for x <= 0.
∞ recursion for x > 0.
In practice, runs out of stack space in Python. Similar Racket program will loop, but run out of memory when x gets too big.
```



```
def collatz(x):
    while x != 1:
        if (x % 2) == 0:
            x = x/2
        else:
            x = 3*x + 1
    return 1
```

Although this terminates for all  $x > 0$  that have been tested, no one knows the answer for all  $x$ . This is a famous open problem in mathematics.  
Solve it and win a Fields medal!

# Static Semantics and Uncomputability

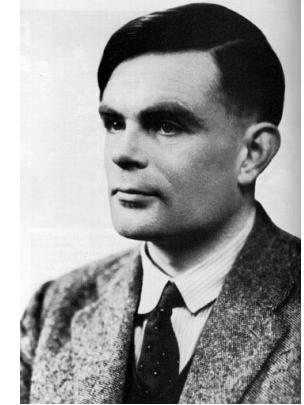
It is generally **impossible** to answer any interesting question about static program analysis!

This is a consequence of **Rice's Theorem**.

For example, will this program ever:

- halt on certain inputs
- encounter an array index out of bounds error?
- throw a NullPointerException?
- access a given object again?
- send sensitive information over the network?
- divide by 0?
- run out of memory, starting with a given amount available?
- try to treat an integer as an array?

# The Church-Turing Thesis and Turing-Completeness



- **Church-Turing Thesis:** Computability is the common spirit embodied by this collection of formalisms.
- This thesis is a claim that is widely believed about the intuitive notions of **algorithm** and **effective computation**. It is not a theorem that can be proved.
- Because of their similarity to later computer hardware, Turing machines have become the gold standard for effectively computable.
- We'll see here that Church's lambda-calculus formalism is the foundation of modern programming languages.
- A consequence: programming languages all have the “same” computational “power” in term of what they can express. All such languages are said to be **Turing-complete**.

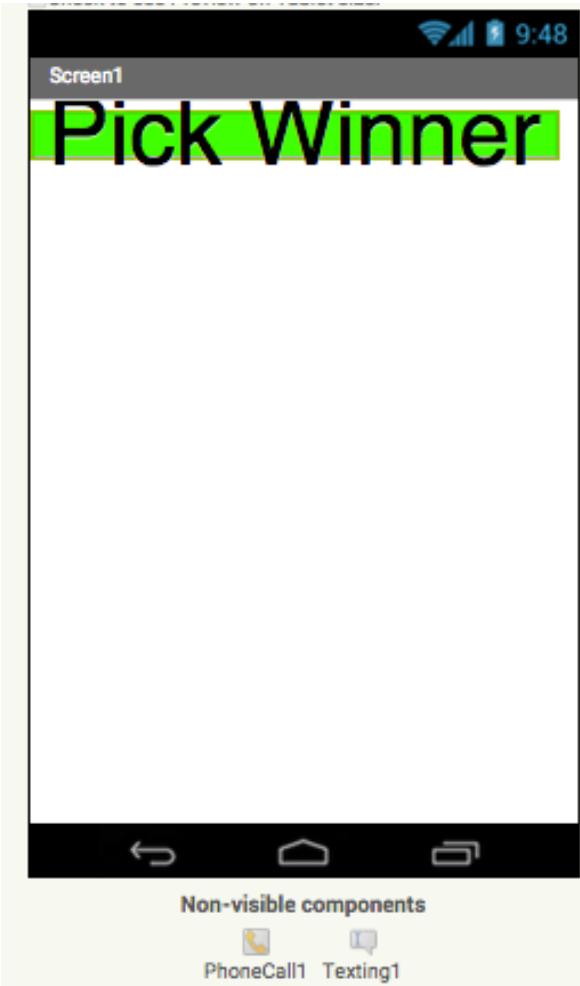
# Expressiveness and Power

- About:
  - ease
  - elegance
  - clarity
  - modularity
  - abstraction
  - ...
- Not about: computability
- Different problems, different languages
  - Facebook or web browser in assembly language?

# Pragmatics: Raffle App In App Inventor

<http://ai2.appinventor.mit.edu>

## Designer Window



## Blocks Editor

```
initialize global numbers to create empty list
when Texting1 .MessageReceived
  number messageText
  do add items to list list get global numbers
    item get number
when Button1 .Click
  do set PhoneCall1 .PhoneNumber to pick a random item list get global numbers
    call PhoneCall1 .MakePhoneCall
```

To enter the raffle, text me now with  
an empty message: **339-225-0287**

How hard is this to do in more traditional  
development environments for Android/  
iOS?

# Pragmatics: Metaprogramming

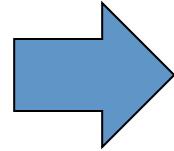
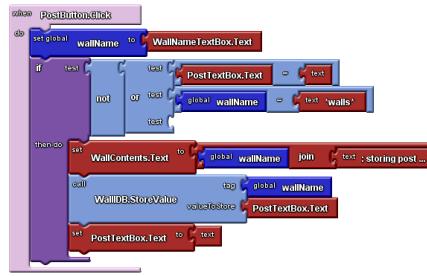
PLs are implemented in terms of **metaprograms** = programs that manipulate other programs.

This may sound weird, but programs are just trees (ASTs), so a metaprogram is just a program that manipulates trees (think a more complex version of Data Structure's binary tree programs).

Implementation strategies:

- ***Interpretation***: interpret a program P in a source language S in terms of an implementation language I.
- ***Translation (compilation)***: translate a program P in a source language S to a program P' in a target language T using a translator written in implementation language I.
- ***Embedding***: express program P in source language S in terms of data structures and functions in implementation language I.

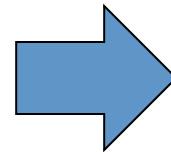
# Metaprogramming: Interpretation



# Program in language L

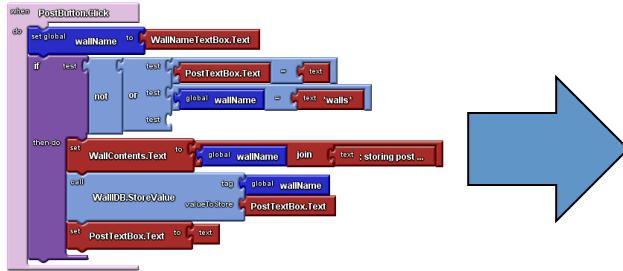


# Interpreter for language L on machine M



# Machine M

# Metaprogramming: Translation



# Program in language A

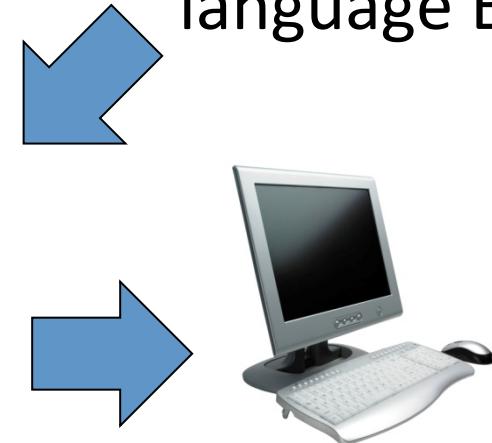


# A to B translator

# Program in language B



# Interpreter for language B on machine M



# Machine M

# Metaprogramming: Bootstrapping Puzzles

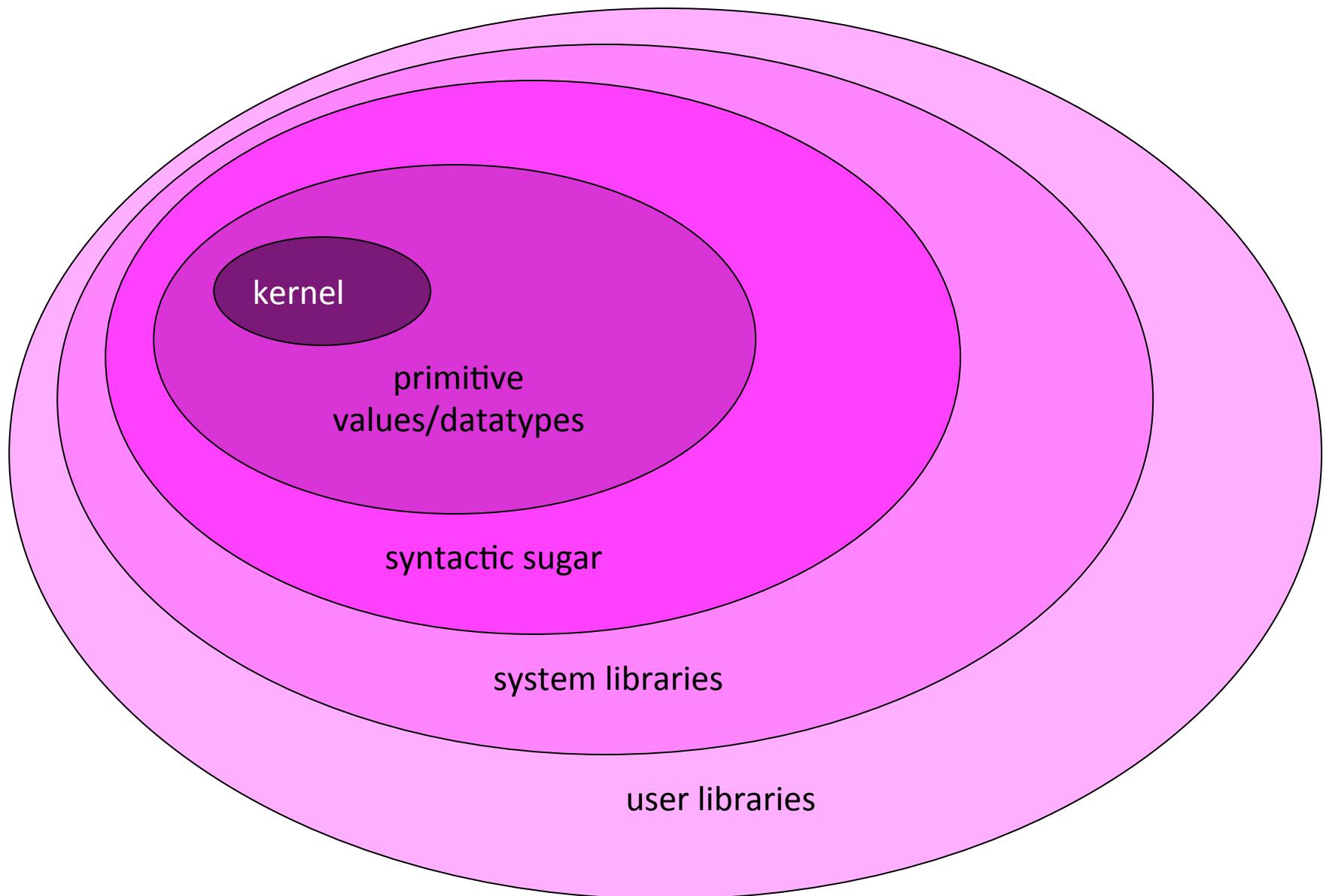
How can we write a Java-to-x86 compiler in Java?



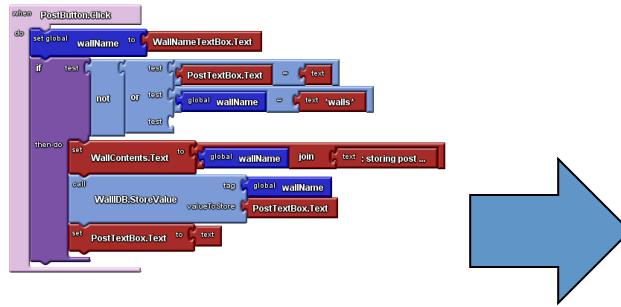
We'll learn how to  
understand such puzzles!



# Metaprogramming: Programming Language Layers



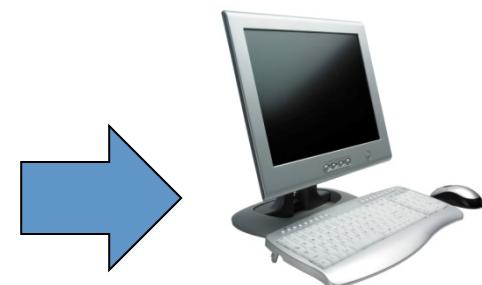
# Metaprogramming: Embedding



Program in  
language A  
embedded in  
language B

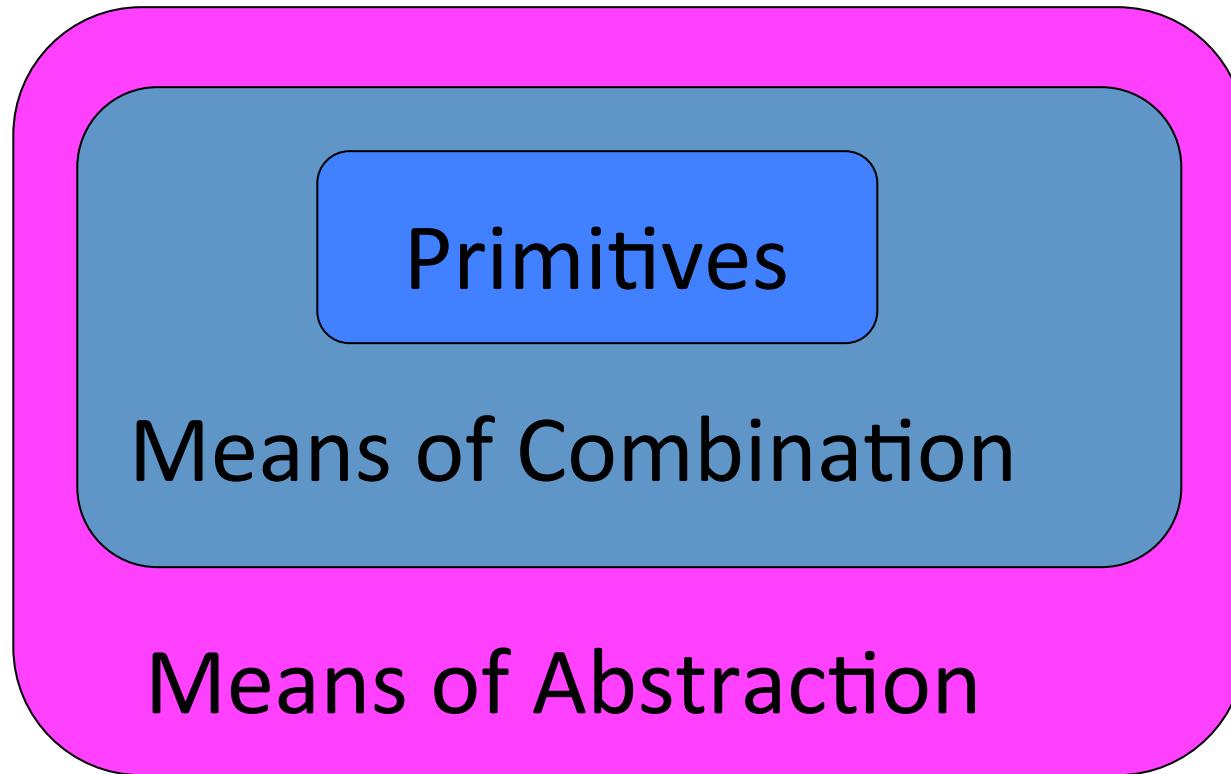


Interpreter  
for language B  
on machine M



Machine M

# Programming Language Essentials



Think of the languages you know. What means of abstraction do they have?

# Why? Who? When? Where? Design and Application

- Historical context
- Motivating applications
  - Lisp: symbolic computation, logic, AI, experimental programming
  - ML: theorem-proving, case analysis, type system
  - C: Unix operating system
  - Simula: simulation of physical phenomena, operations, objects
  - Smalltalk: communicating objects, user-programmer, pervasiveness
- Design goals, implementation constraints
  - performance, productivity, reliability, modularity, abstraction, extensibility, strong guarantees, ...
- Well-suited to what sorts of problems?