The Plan

PL = Programming Language

1. What is a PL?
2. What goes into PL design?
3. How is a PL defined?
4. Why study PLs? What will you learn?

What is a Programming Language?

PL = Procedural Lever

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*

PL = Presentation of Logic

… a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute.

– Harold Abelson and Gerald J. Sussman,
Structure and Interpretation of Computer Programs

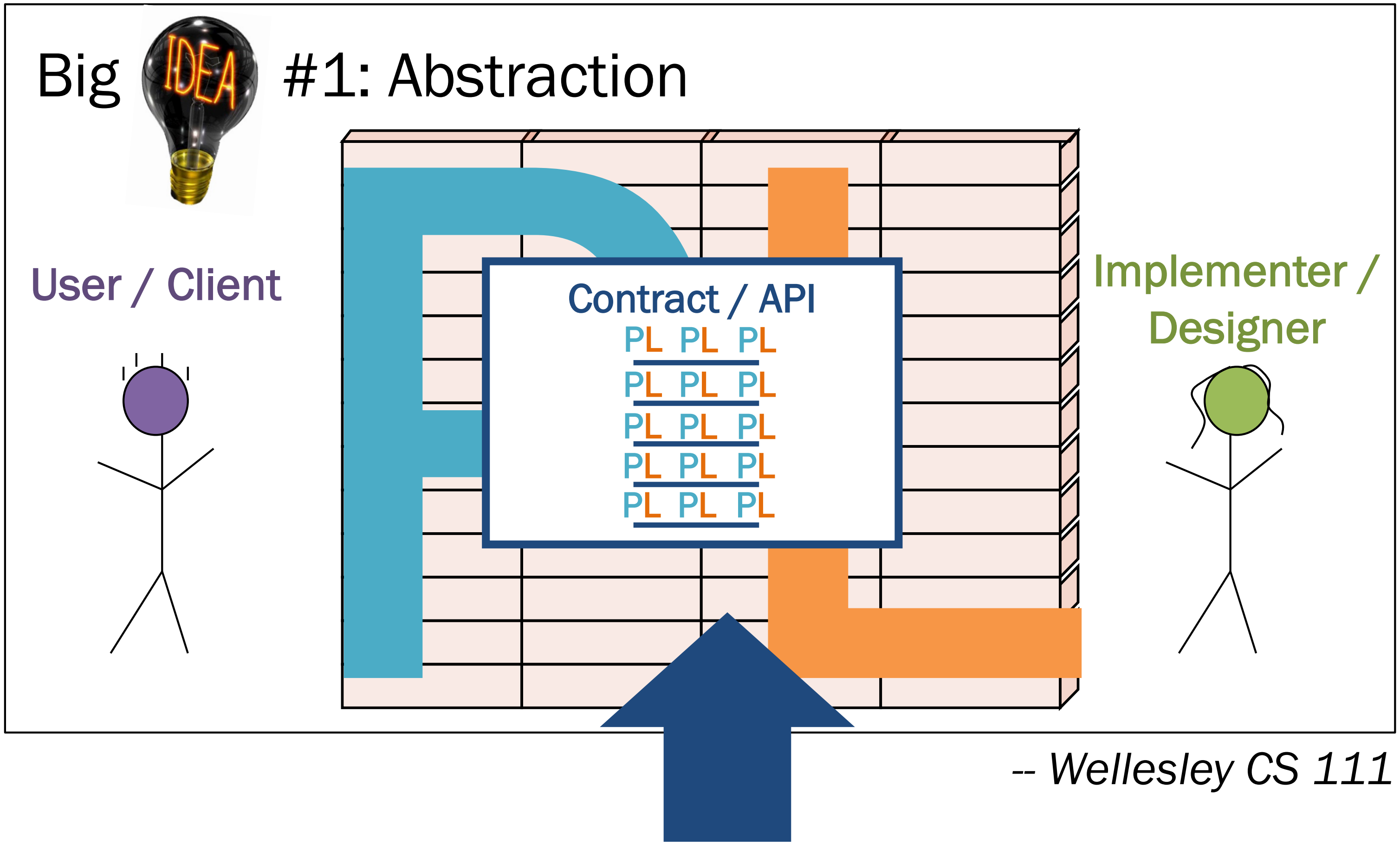
PL = Problem-solving Lens

A good programming language is a conceptual universe for thinking about programming.

A language that doesn't affect the way you think about programming is not worth knowing.

– Alan Perlis

PL = Precise Laws



-- *Wellesley CS 111*

Determine what and how abstractions can be expressed and manipulated.

Enable precise manual and automated reasoning about properties of programs.

What goes into PL design?

PL design: application / purpose

General computation

Domain-specific computation

Motivating application

Computability

Turing-complete = equivalent to key models of computation

- ***Turing machine*** (CS 235)
- ***(Lambda) λ-calculus*** (CS 251)
- ***...***

Church-Turing thesis: Turing-complete = computable

⇒ All Turing-complete PLs (roughly, general-purpose PLs or just "PLs")

- have "same" computational "power"; and
- can express all possible computations; but
  - the ease, concision, elegance, clarity, modularity, abstractness, efficiency, style, of these computations may vary radically across such languages.

PL design: goals/values

PL design affects goals/values for programs:

- Correctness, Reliability, Security
- Clarity, Explainability, Learnability, Analyzability, Audibility
- Fairness, Privacy
- Maintainability, Extensibility
- Efficiency (of programs, programmers), Optimizability
- Creativity, Expressivity, Flexibility
- …

"Programming paradigms"

- *Imperative*: execute step-by-step statements to change mutable state.
  Lens: statements, execution, mutation, side effects.
- *Functional*: compose functions over immutable data.
  Lens: expressions, evaluation, results, composition.
- *Object-oriented*: pass (typically imperative) messages between objects.
  Lens: behaviors, methods, encapsulation, extension.
- *Deductive*: query over declarative relationships.
  Lens: relations, implications, constraints, satisfiability.
- **Plenty more...**

Imprecisely defined, overlapping. Most PLs blend a few.

Quicksort

```c
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 );
  }
}
```

Imperative Style
(C; Java would be similar)

Functional Style (SML)

```sml
fun qsort [] = []
  | qsort (x::xs) =
    let
      (lt, ge) = List.partition (fn n => n < x) xs
    in
      (qsort lt) @ (x :: (qsort ge))
    end
```

PL design: dimensions

- *First-class values:* What can be named, passed as an argument, returned as a result, stored in a data structure?
- *Naming*: Do variables/parameters name expressions, values, or storage cells? How are names declared, referenced, scoped?
- *State*: What is mutable or immutable?
- *Control*: Conditionals, pattern matching, loops, exception handling, continuations, parallelism, concurrency?
- *Data*: Products (arrays, tuples, records, maps), sums (options, one-ofs, variants), objects with behavior?
- *Types*: Static? Dynamic? Polymorphic? Abstract? First-class?
- …

How is a PL defined?

Defining a programming language

Syntax: *form* of a PL

- Structure of programs: symbols and grammar
- Concrete syntax vs. abstract syntax trees (ASTs)

Semantics: *meaning* of a PL

- *Dynamic Semantics*:
  Behavior, actions, results of programs **when evaluated.**
  - **Evaluation rules:** What is the result or effect of evaluating each language construct? How are these composed?
- *Static Semantics:*
  Properties of programs determined **without evaluation.**
  - **Scope rules:** to which declaration may a variable reference refer?
  - **Type rules:** is a program well-typed (and therefore legal)?

Syntax (form) vs. Semantics (meaning)

Furiously sleep ideas green colorless.

Colorless green ideas sleep furiously.

Little brown rabbits sleep soundly.

Concrete syntax: absolute value function

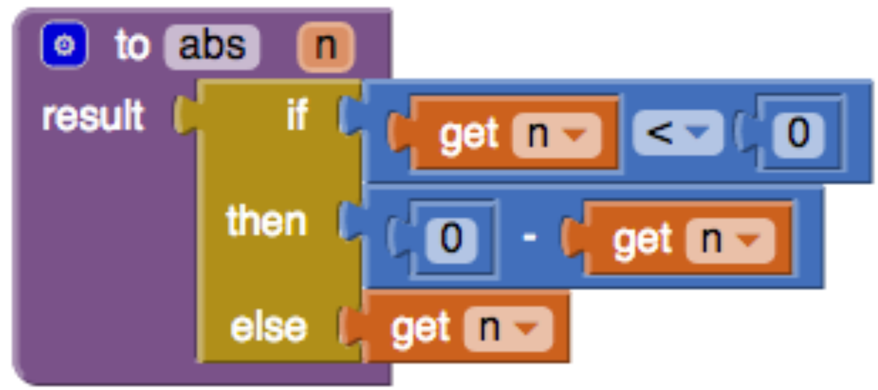
Logo:
```
to abs :n
   ifelse :n < 0 [output (0 - :n)] [output :n]
end
```

JS:
```
function abs(n) {if (n<0) return -n; else return n;}
```

Java:
```
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:



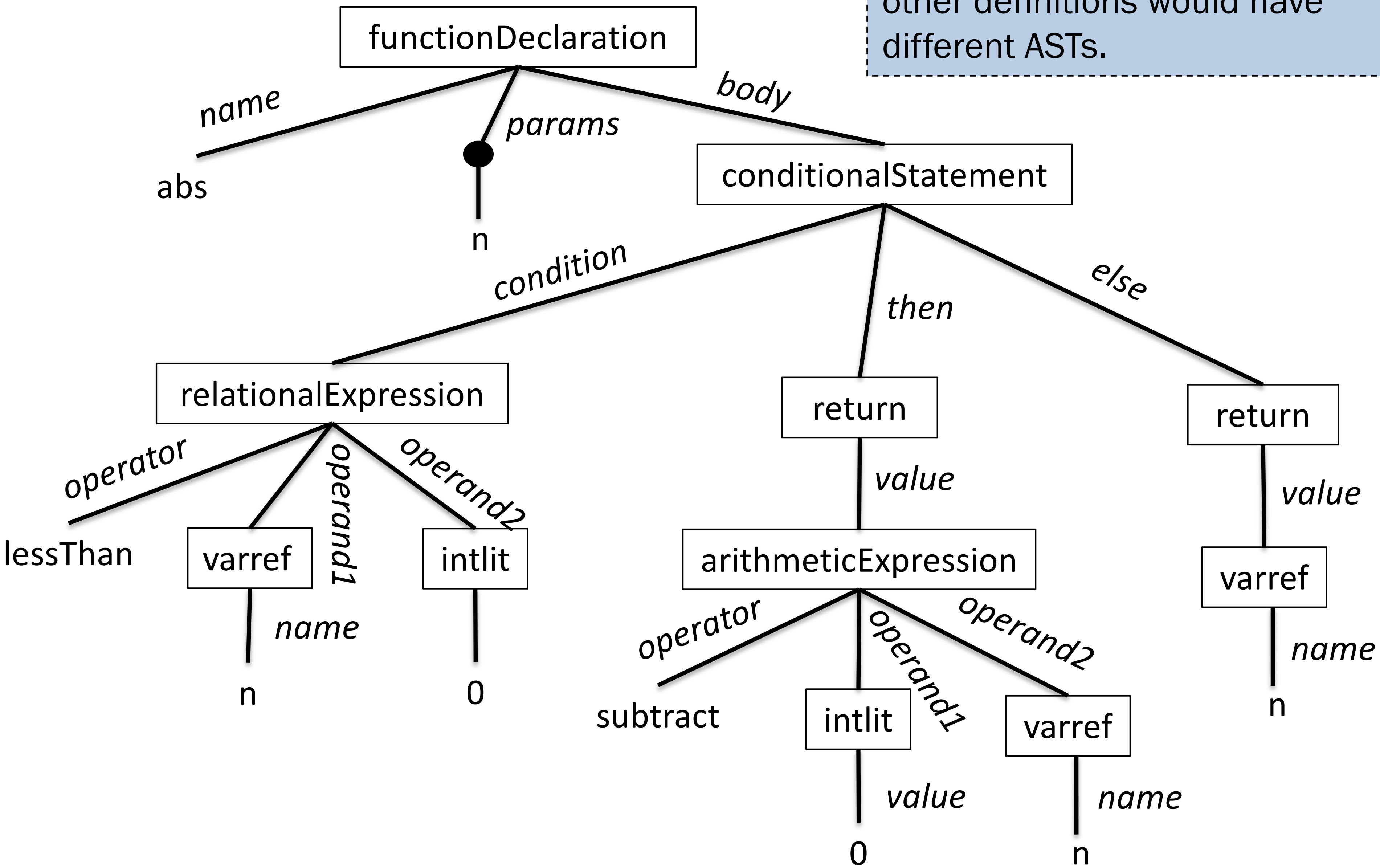
Racket:
```
(define abs (lambda (n) (if (< n 0) (- n) n)))
```

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

Forth:
```
: abs dup 0 < if 0 swap - then ;
```

Abstract Syntax Tree (AST): absolute value function

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



Dynamic semantics examples

What is the meaning of the following expression?

```
(1 + 11) * 10
```

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()));
```

Static semantics example: 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: termination checking

Which of these Python programs has inputs for which it does not terminate (runs forever)?

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

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

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

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

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

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

Static semantics

Properties of programs determined **without evaluation.**

- **Scope:** To which declarations do variable references refer?
- **Types:** What are the types of entities in the program?
- ...

Goal: Accept only (and all) ***safe*** programs free of various problems.

Will any evaluation of this program ever:

- reference a nonexistent variable?
- index outside an array's bounds? dereference null? divide by zero?
- apply an array operation to an integer?
- coordinate concurrency unsafely?
- access a given object again? surpass a given memory budget?
- leak sensitive information over the network?
- **...** not terminate (run forever)? reach a given point in the program?
- ...

Reality: Most useful static semantics questions for Turing-complete languages are **uncomputable!** (Rice's Theorem, CS 235)

PL implementation

PLs are implemented by **metaprograms**, programs in an *implementation language* that manipulate programs in a *source language*.

- An *interpreter* **evaluates** a program in the *source* language.
  A ***processor*** is an interpreter implemented in physical hardware.
- A *compiler* **translates** a program in the *source* language to a program in a *target* language.
- An *embedding* **defines** the features of the *source* (a.k.a. *guest*) language directly as data structures, functions, macros, or other features of a *host* language.

Program analysis

Automated reasoning about program properties

But isn't that uncomputable?

Program analysis: *effective solutions to unsolvable problems™*

- Conservative static analysis
- Dynamic analysis
- Hybrid analysis
- Extend the language to make more explicit
- Static semantics = integrate language and analysis

Why study PLs?
What will you learn?

Why study PLs?

Be a more effective programmer and computer scientist:

- Leverage powerful features, idioms, and tools.
- Think critically about PL design trade-offs and their implications for your values.
- Learn, evaluate, compare, choose languages.
- Communicate technical ideas, problems, and solutions precisely.

Approach problem-solving as a *language designer / program analyst:*

- Problem-solving = designing the language of your problem and its solutions.
- You may not design a general-purpose PL, but you will design a DSL.
- API and library design = language design = DSL.

Broad active area of research:

- Invent better general-purpose programming tools, features, analyses.
- Apply PL mindset to broader problem domains and applications, e.g.:
  - Analyze/enforce fairness/non-bias, privacy, security properties.
  - High-performance/high-assurance DSLs for machine learning, graphics, Uis, data science.
  - Model and control biochemical systems.
  - Automated verification of website accessibility compliance.
  - Support large-scale systems programming or specialized hardware.

Plan

1. **How to Program**
   – Topics: syntax, dynamic semantics, functional programming
   – Lens: Racket
2. **What's in a Type**
   – Topics: static types, data, patterns, abstractions
   – Lens: Standard ML
3. **When Things Happen**
   – Topics: evaluation order, parallelism, concurrency
   – Lens: Standard ML/Manticore?, Java, ...
4. **Why a Broader PL Mindset**
   – Topics: problem decomposition, deductive programming, program analysis, DSLs
   – Lens: Racket, Standard ML, Java, Prolog/Datalog, ...

Metaprogramming

Expect some adjustments.

Administrivia

Everything is here: https://cs.wellesley.edu/~cs251/

- Material posted ahead of class meetings.
  - PYO: Print your own if you like taking notes on slide copies.
- First assignment out soon, due in a week.
- New space: SCI L037 CS Systems Lab, ***mostly*** finished...
  - Expect a couple hiccups as we iron out a few things.
  - Potential experiments with class format dependent on these.
- Expect assignments to require:
  - deep thought, sometimes to discover a surprisingly concise solution;
  - independently extending / learning ideas beyond lecture coverage.

  Learning is an adventure in an unknown land. Explore and experiment!
- Enjoying PLs? Reading group forming soon...