The Plan

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?

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**

Big Idea #1: Abstraction

Determine what and how abstractions can be expressed and manipulated.

Enable precise manual and automated reasoning about properties of programs.

– Wellesley CS 111

**What goes into PL design?**
PL design: application / purpose

General computation

Domain-specific computation

Motivating application

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
- ...

Computability

Turing-complete = equivalent to key models of computation

- Turing machine (CS 235)
- (Lambda) \( \lambda \)-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.

"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

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

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

**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):

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?

\[(1 + 11) \times 10\]

What is printed by the following program?

```plaintext
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?

\[ 2 \times (3 + 4) \]

\[ \begin{array}{l}
\text{A} & 2 \times (3 + 4) \\
\text{B} & 2 \leq (3 + 4) \\
\text{C} & 2 \leq \text{True} \\
\text{D} & \text{if} (a < b) \{ \\
& \quad \text{c} = a + b; \\
& \quad \text{else} \{ \\
& \quad \quad \text{c} = a \times b; \\
& \} \\
\text{E} & \text{if} (a < b) \{ \\
& \quad \quad \text{c} = a + b; \\
& \quad \} \\
\text{F} & \text{if} (a) \{ \\
& \quad \quad \text{c} = a + b; \\
& \quad \} \\
\text{G} & \text{public boolean f(int i, boolean b) \{ \\
& \quad \quad \text{return b && (i > 0); } \\
& \} } \\
\text{H} & \text{public int g(int i, boolean b) \{ \\
& \quad \quad \text{return i \ast (b ? 1 : -1); } \\
& \} } \\
\text{I} & \text{public int p(int w) \{ \\
& \quad \quad \text{if } (w > 0) \{ \text{return } 2 \ \ast w; \} \\
& \} } \\
\text{J} & \text{public int q(int x) \{ \text{return } x > 0; \} } \\
\text{K} & \text{public int r(int y) \{ \text{return } g(y, y>0); \} } \\
\text{L} & \text{public boolean s(int z) \{ \text{return } f(z); \} } \\
\end{array} \]

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)

Static semantics example: termination checking

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

\[
\begin{align*}
&\text{def } f(x): \\
&\quad \text{return } x + 1 \\
&\text{def } g(x): \\
&\quad \text{while } \text{True}: \\
&\quad \quad \text{pass} \\
&\quad \text{return } x \\
&\text{def } h(x): \\
&\quad \text{while } x > 0: \\
&\quad \quad x = x + 1 \\
&\quad \text{return } x \\
&\text{def } h2(x): \\
&\quad \text{if } x \leq 0: \\
&\quad \quad \text{return } x \\
&\quad \text{else}: \\
&\quad \quad \text{return } h2(x + 1) \\
&\text{def } g2(x): \\
&\quad \text{return } g2(x) \\
&\text{def } collatz(x): \\
&\quad \text{while } x \neq 1: \\
&\quad \quad \text{if } (x \% 2) == 0: \\
&\quad \quad \quad x = x / 2 \\
&\quad \quad \text{else}: \\
&\quad \quad \quad x = 3 \times x + 1 \\
&\quad \text{return } 1
\end{align*}
\]

PL implementation

PLs are implemented by metaprogams, 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?

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.

Why study PLs?

What will you learn?

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, ...

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, due in a week.
– Expect assignments to require:
  • deep thought, sometimes to discover a surprisingly concise solution;
  • practical application of basic definitions given in class;
  • independently extending / learning ideas beyond lecture coverage.

Learning is an adventure in an unknown land. Explore and experiment!