Big Ideas for CS 251
Theory of Programming Languages
Principles of Programming Languages

PL is my passion!

• First PL project in 1982 as intern at Xerox PARC
• Created visual PL for 1986 MIT masters thesis
• 1994 MIT PhD on PL feature (synchronized lazy aggregates)
• 1996 – 2006: worked on types as member of Church project
• 1988 – 2008: Design Concepts in Programming Languages
• 2011 – current: lead TinkerBlocks research team at Wellesley
• 2012 – current: member of App Inventor development team

General Purpose PLs

Java  Python  Perl
Fortran  ML  JavaScript
Racket  Ruby  Haskell
C/C++  CommonLisp

Programming Languages

• 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?
  – What features must/should it contain?
  – What are the design dimensions?
  – What are design decisions that must be made?
• Why should you take this course? What will you learn?
Domain Specific PLs

- Excel
- HTML
- CSS
- OpenGL
- R
- Matlab
- LaTeX
- 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?**

CS111 Big idea #1: Abstraction

**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
- *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?
- *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)?

**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?

**Programming Language Essentials**

- **Primitives**
- **Means of Combination**
- **Means of Abstraction**

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

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

**Scheme/Racket:**
```
(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?

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

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(igno, x) { return x + x; }
print(dbl(inc(), inc()))
```
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).

\[
\begin{array}{|c|c|c|c|}
\hline
\hline
\text{Java} & & & & \\
\text{C} & & & & \\
\text{Python} & & & & \\
\text{JavaScript} & & & & \\
\text{Pascal} & & & & \\
\text{App Inventor} & & & & \\
\hline
\end{array}
\]

How do you determine the answers???

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?

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

Static Semantics Example 2: Detecting Loops

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

```python
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
```

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 (see CS235).

For example, will this program ever:

- halt on certain inputs
- encounter an array index out of bounds error?
- throw a NullPointerError?
- 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 (CS235) have become the gold standard for effectively computable.
- We’ll see in CS251 that Church’s lambda-calculus formalism is the foundation of modern programming languages.
- A consequence: programming languages all have the “same” computational “power” in terms 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

Pragmatics: Metaprogramming

PLs are implemented in terms of **metapograms** = 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 CS230 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: Embedding**

Program in language A embedded in language B → Interpreter for language B on machine M → Machine M

**Metaprogramming: Bootstrapping Puzzles**

In what language is the gcc C compiler implemented?

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

How can we write a Racket interpreter in Racket?

We’ll learn how to understand such puzzles!
Metaprogramming: Programming Language Layers

- **Kernel**: The core of the programming language, providing fundamental operations and system services.
- **Primitive Values/Datatypes**: The basic building blocks of data.
- **System Libraries**: Collections of functions and utilities that support specific tasks.
- **User Libraries**: Libraries written by users to extend the functionality of the language.
- **Syntax Sugar**: Additional syntactic constructs that make the language more readable.
- **Big ideas 29**

**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?

**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**

**Imperative Style (in C)**

```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);
    }
}
```

**Functional Style (in Haskell)**

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

Big ideas 29
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?

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 programmer by leveraging powerful features (first-class functions, tree recursion, sum-of-product datatypes, pattern matching)
  – You probably won’t design a general-purpose PL, but might design a DSL
  – 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?

Administrivia

• Schedule, psets, quizzes, lateness policy, etc.: see http://cs.wellesley.edu/~cs251/.
• Do (most of) PS0 tonight
  – Fill out “get to know you” Introze introduction.
  – Review course syllabus and policies (we’ll go over these tomorrow)
  – Read Wed slides on “big-step semantics” of Racket
  – Install Dr. Racket
• PS1 is available; due next Friday. Start it this week!
• Credit/non is a bad idea for 251. Talk to me first!
• Visit me in office hours before next Friday!