Discuss: Programming Language

- What is a PL?
- What goes into the design of a PL?
- Why are new PLs created? Why are there so many?
  - TIOBE
- Why are certain PLs popular?
  - socio PLT

What? Structure and Semantics

A programming language is defined by:

- Structure: what are the primitive structures of a language?
  - Abstract syntax: the abstract structure of programs, independent of any concrete representation
- Semantics: what do the structures of a language mean?
  - Type systems: what programs have meaning?
  - Evaluation rules: what is the result or effect of evaluating each language structure or a whole program?

How? Representation, Analysis, and Implementation

The real world demands:

- Representation: how do we represent programs for humans and machines?
  - Concrete syntax: the symbols used to represent programs physically as input and output for humans or machines
- Implementation: how can we evaluate programs in the language?
  - How can we evaluate programs in the language on a physical computer system?
  - How can we optimize the performance of program execution?
- Analysis: How can we decide whether a given input constitutes a valid program? avoids simple data mismatch errors?


- 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?
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (p:xs) = quicksort lesser ++ [p] ++ quicksort greater

where
    lesser = filter (< p) xs
    greater = filter (>= p) xs

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);
    }
}

Computability

• Computable: \( f(x) = x + 1 \), for natural numbers
  – addition algorithm

• Uncomputable functions exist!

Turing-completeness

• partial recursive functions
• Turing machines
• lambda calculus
• ... general-purpose programming languages
• ... all general-purpose programming languages are "the same".

Expressiveness and Power

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

Halting Problem

• Halt(P,x): Does program P halt (i.e., finish after a finite number of steps and return a result) when run on input, x?

• Why do we care?
  – Canonical undecidable problem.
  – BIG implications for what we can and cannot decide about programs.
Hand-wavy intuition

• Run it for 100 steps. Did it halt?
• Run it for 1000 steps. Did it halt?
• ...
• Program could always run at least one step longer than we check.

Proof: Halting Problem Undecidable

• Proof by contradiction, diagonalization.
• Suppose \( Halt(P,x) \) solves the halting problem
  – halts on all inputs and returns true if running \( P(x) \) will halt and false if it will not.
• Define \( Spy(P) \) as the following program:
  – Run \( Halt(P,P) \). This will always halt and return a result.
  – If the result is true, loop forever, otherwise halt.
• So...
  – \( Spy(P) \) will run forever if \( P(P) \) would halt and
  – \( Spy(P) \) will halt if \( P(P) \) would run forever.
  – (Not running \( P(P) \), just asking what it would do if run.)
• Run \( Spy(Spy) \).
  – It first runs \( Halt(Spy,Spy) \), which halts and returns a result.
  – If the result is true, it now loops forever, otherwise it halts.
• So...
  – If \( Spy(Spy) \) halts, \( Halt(Spy,Spy) \) told us that \( Spy(Spy) \) would run forever.
  – If \( Spy(Spy) \) runs forever, \( Halt(Spy,Spy) \) told us that \( Spy(Spy) \) would halt.
• Contradiction!

Interesting things are undecidable.

• Will this Java program ever throw a \( \text{NullPointerException} \)?
• Will this program ever access a given object again?
• Will this program ever send sensitive information over the network?
• Will this program divide by 0?
• Will this program ever run out of memory, starting with a given amount available?
• Will this program ever try to treat an integer as an array?

Proving Undecidability

• To prove a problem \( P \) is undecidable, reduce a known undecidable problem \( Q \) to it:
  – Assume \( \text{DecideP} \) decides the problem \( P \).
  – Show how to translate an instance of \( Q \) to an instance of \( P \), so \( \text{DecideP} \) decides \( Q \).
  – Contradiction.
• \( Q \) is typically the halting problem.

Example: \( \text{HaltAny(Q)} \) is Undecidable

• \( \text{HaltAny(Q)} \): does program \( Q \) halts for \( \geq 1 \) input?
• Suppose that \( \text{HaltAny(Q)} \) always halts.
• Solve \( Halt(P,x) \) with \( \text{HaltAny} \):
  – Build a new program \( R \) that ignores its input and runs \( P(x) \).
  – Then \( \text{HaltAny(R)} \) returns true if and only if \( P \) halts on \( x \).
    • \( R(...) \) always does the same thing, so if one halts, all do.
• Contradiction!

in practice: be conservative

• "yes", "no", or "I give up. Not sure."
• Type systems
• Garbage Collection
• Program Analysis
Next time (this time)

- Case study: Lisp/Racket and functional programming
- Clean slate approaching language.
- Mercurial tutorial
- Linux environment -- schedule a session?
- shifted things around because of snow day
- 1st assignment, more slides/notes/code posted later today

Why PL?

- Crossroads of CS
- Understand mindset, apply elsewhere
  - "A good programming language is a conceptual universe for thinking about programming" -- Alan Perlis
  - become a better problem-solver
  - compare languages
  - prepare for future PLs, problems
- Ask why PLs are the way they are
- Implementation: understand cost-convenience trade-offs

How?

- Small scale: focus on essential language dimensions
  - Racket/Lisp, ML, functional programming, historical context
  - core language features
  - interpreters
  - foundations
- Large scale: focus on modularity
  - Different approaches to modularity, trade-offs
  - OOP vs. FP
- Parallelism and Concurrency
Racket