Lecture 30:

- Quiz 7
- More on PL paradigms

Announcements:

- HW-6 out

Exercise: Write a turing machine to flip a's and b's

Current State	Current Symbol	New Symbol	New State	Direction
s_1	a	b	s_1	Right
s_1	b	a	s_1	Right
s_1	Blank	Blank	s_2	Left

- s_1 is the start state, s_2 is halt state

Exercise: Write a turing machine to subtract 1 from a binary number ≥ 1

Basic Approach: Find first 1, flip to 0, then write 1's until end

- the "alphabet" is $\{0, 1\}$ (binary digits) as opposed to $\{a, b\}$
- s_1 is the start state (go to end), s_2 (find first 1), s_3 (write 1's), s_4 (halt)

Current State	Current Symbol	New Symbol	New State	Direction
s_1	0	0	s_1	Right
s_1	1	1	s_1	Right
s_1	Blank	Blank	s_2	Left
s_2	0	0	s_2	Left
s_2	1	0	s_3	Right
s_2	Blank	Blank	s_4	Left
s_3	0	1	s_3	Right
s_3	Blank	Blank	s_4	Left

Programming Languages and "Turing Completeness"

A PL is "**Turing Complete**" if it can simulate any Turing Machine

- Every computable function can be computed by a TM (Church-Turing thesis)
- If a PL is turing complete, it can express *all possible computations*

Note: Can write a TM that can simulate (run) all other TMs (encoded on tape)

- such a TM is called "**universal**" (i.e., a machine that can run machines)

Examples of languages that are not Turing Complete:

- Markup languages: HTML, XML, JSON, YAML, ...
- Many "domain-specific" languages: (basic) SQL, regular expressions

Turing Completeness not necessarily tied to specific constructs

- imperative languages with conditional branching (if-goto, while loops) and arbitrary mem access (# of variables)
- whereas functional and logic-based languages have other constructs such as pattern matching and recursion (no goto, no loops)

"Languages" that are (accidentally) Turing Complete

- Musical Notation (requires human to be the memory/tape)
- Excel spreadsheets w/ formulas
- Pokemon Yellow (`https://www.youtube.com/watch?v=p5T81yHkHtI`)
- Magic The Gathering card game (human selects moves)
- PowerPoint animations (requires human to follow links)

The Lambda (λ) Calculus

From λ-calculus to functional programming

- TMs are (roughly) the computation model behind imperative languages
- λ-calculus is (roughly) the computation model behind functional languages

Basic idea of λ-calculus

1. Unnamed, single-variable functions (λ "functions" aka "abstractions")
   - $\lambda x.x$ takes an x and returns an x
   - $\lambda x.(\lambda y.x)$ takes x and returns a function that takes y and returns x
   - shorthand for multi-argument functions: $\lambda xy.x$
2. Function application
   - $(\lambda x.x)0$ applies the identity function to 0 (resulting in 0)
   - $(\lambda x.(\lambda y.x))ab$ reduces to a ... $(\lambda x.(\lambda y.x))ab \Rightarrow (\lambda y.a)b \Rightarrow a$
3. Expressions
   - Either a function, an application, a variable, or a constant
   - A function has the form: $\lambda x.e$ where x is a name and e an expression
   - An application has the form: $e_1 e_2$ where both e's are expressions

Computation in λ-calculus is via function application

- Given a function application such as:

$$(\lambda x.x)y$$

- An application is evaluated by substituting x's in the function body with y:

$$(\lambda x.x)y = [y/x]x = y$$