Lecture 30:
• Quiz 7
• More on PL paradigms

Announcements:
• HW-6 out
**Exercise:** Write a turing machine to flip a’s and b’s

<table>
<thead>
<tr>
<th>Current State</th>
<th>Current Symbol</th>
<th>New Symbol</th>
<th>New State</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>$a$</td>
<td>$b$</td>
<td>$s_1$</td>
<td>Right</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$b$</td>
<td>$a$</td>
<td>$s_1$</td>
<td>Right</td>
</tr>
<tr>
<td>$s_1$</td>
<td>Blank</td>
<td>Blank</td>
<td>$s_2$</td>
<td>Left</td>
</tr>
</tbody>
</table>

- $s_1$ is the start state, $s_2$ is halt state

**Exercise:** Write a turing machine to subtract 1 from a binary number $\geq 1$

**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)

<table>
<thead>
<tr>
<th>Current State</th>
<th>Current Symbol</th>
<th>New Symbol</th>
<th>New State</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>0</td>
<td>0</td>
<td>$s_1$</td>
<td>Right</td>
</tr>
<tr>
<td>$s_1$</td>
<td>1</td>
<td>1</td>
<td>$s_1$</td>
<td>Right</td>
</tr>
<tr>
<td>$s_1$</td>
<td>Blank</td>
<td>Blank</td>
<td>$s_2$</td>
<td>Left</td>
</tr>
<tr>
<td>$s_2$</td>
<td>0</td>
<td>0</td>
<td>$s_2$</td>
<td>Left</td>
</tr>
<tr>
<td>$s_2$</td>
<td>1</td>
<td>0</td>
<td>$s_3$</td>
<td>Right</td>
</tr>
<tr>
<td>$s_2$</td>
<td>Blank</td>
<td>Blank</td>
<td>$s_4$</td>
<td>Left</td>
</tr>
<tr>
<td>$s_3$</td>
<td>0</td>
<td>1</td>
<td>$s_3$</td>
<td>Right</td>
</tr>
<tr>
<td>$s_3$</td>
<td>Blank</td>
<td>Blank</td>
<td>$s_4$</td>
<td>Left</td>
</tr>
</tbody>
</table>
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=p5T8lyHkHtI)
- Magic The Gathering card game (human selects moves)
- PowerPoint animations (requires human to follow links)
The Lambda ($\lambda$) Calculus

From $\lambda$-calculus to functional programming

- TMs are (roughly) the computation model behind imperative languages
- $\lambda$-calculus is (roughly) the computation model behind functional languages

Basic idea of $\lambda$-calculus

1. Unnamed, single-variable functions ($\lambda$ “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: $e_1e_2$ where both $e$’s are expressions

Computation in $\lambda$-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 $y$:
  $$(\lambda x.x)y = [y/x]x = y$$