Today

- Intro to Programming Paradigms
- Getting started with Haskell

Assignments

- HW-1, R-1, PROJ-1 (out)
Programming Paradigms

Imperative vs Declarative Languages

**Imperative Languages**: Programmers specify how to solve the problem and the system carries out the steps

**Declarative Languages**: Programmers specify what the solution should look like and the system determines how best to compute the solution

Logic and Functional languages are generally considered (more) declarative

- compared to object-oriented & procedural languages (C/C++/Python/Java/etc.)
- largely has to do with the underlying models of computation used
- we’ll look later at a truly declarative language
From Turing Machines to Imperative Programming

Turing Machines:

1. infinite **tape** divided into memory cells (one symbol per cell)
2. read/write **head** that can move left/right one cell at a time
3. **state register** that stores the current state of the machine
4. state **transition table**: state + curr head symbol $\rightarrow$ write symbol + move head + new state

Example: replace a’s with b’s

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
  a & b & b & a & \cdots & \\
\end{array}
\]

the infinite tape

\[
\uparrow
\]

the tape head (with the machine in state $s_i$)

<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>Blank</td>
<td>Blank</td>
<td>$s_2$</td>
<td>Right</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$a$</td>
<td>$b$</td>
<td>$s_2$</td>
<td>Right</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$b$</td>
<td>$b$</td>
<td>$s_2$</td>
<td>Right</td>
</tr>
</tbody>
</table>

Turing Machines are imperative ...

- they specify how the computation should be carried out (very low level)
- close to how computers function and to many imperative languages
- tape stores data, state determines what operations occur, etc.
From $\lambda$-calculus to functional programming

Basic idea of $\lambda$-calculus

1. Unnamed, single-variable functions (“$\lambda$ functions”)
   - $\lambda x.x$ 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 $a$

3. Expressions
   - Either a function, an application, or a name (like $x$, $a$, $0$, etc.)
   - 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 $y$:
  $$(\lambda x.x)y = [y/x]x = y$$
Substitutions give a way to simplify $\lambda$-expressions:

$$T \equiv \lambda xy.x \quad \text{(True)}$$
$$F \equiv \lambda xy.y \quad \text{(False)}$$

We can use these to define basic logical operators (AND, OR, NOT):

$$\land \equiv \lambda xy.xy(\lambda uv.v) \equiv \lambda xy.xyF$$
$$\lor \equiv \lambda xy.x(\lambda uv.u)y \equiv \lambda xy.xTy$$
$$\lnot \equiv \lambda x.x(\lambda uv.v)(\lambda ab.a) \equiv \lambda x.xFT$$

For Example:

$$T \land T = (\lambda xy.x)T = (\lambda y.TyF)T = TTF = (\lambda x.x)TF = T$$
$$T \land F = (\lambda xy.x)F = (\lambda y.FyF)F = TTT = (\lambda x.x)FF = F$$

**Exercise 1:** Evaluate the expressions:

$$F \land T = (\lambda xy.x)F = (\lambda y.FyF)T = FTF = (\lambda x.x)TF = F$$
$$F \land F = (\lambda xy.x)F = (\lambda y.FyF)F = FFF = (\lambda x.x)FF = F$$
$$F \lor T = (\lambda xy.xTy)F = FTT = (\lambda x.x)TT = T$$
$$\lnot T = (\lambda x.xFT)T = TFT = (\lambda x.x)FT = F$$
$$\lnot F = (\lambda x.xFT)F = FFT = (\lambda x.x)FT = T$$
You can even express recursion using $\lambda$-calculus ...

$$R \equiv (\lambda y. (\lambda x. y(xx))(\lambda x. y(xx)))$$

- The basic idea is that $R$ calls a function $y$ then “regenerates” itself
- For example, applying $R$ to a function $F$ yields:

$$RF = (\lambda y. (\lambda x. y(xx))(\lambda x. y(xx)))F \quad (1)$$
$$= (\lambda x. F(xx))(\lambda x. F(xx)) \quad (2)$$
$$= F((\lambda x. F(xx))(\lambda x. F(xx))) \quad (3)$$
$$= F(RF) \quad (4)$$
$$= F(F(RF)) \quad (5)$$
$$= \text{and so on} \quad (6)$$

- Note in (4) that $F(RF)$ since $RF = (\lambda x. F(xx))(\lambda x. F(xx))$ from (2)
- We can stop recursion using conditional functions (similar to Boolean ops)

$\lambda$-calculus and Turing Machines have the same expressive power