Today

- Programming Paradigms (wrap up)
- Haskell Intro
- Wrap up Interpreter next week (HW 7)

Assignments

- Proj proposal due
- HW 6 out (due next Tuesday)
- HW 7 out (due Thursday after Spring break)
From λ-calculus to functional programming

Basic idea of λ-calculus

1. Unnamed, single-variable functions (“λ 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 \( a \) ... \( (\lambda x.(\lambda y.x))ab \Rightarrow (\lambda y.a)b \Rightarrow a \)

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

\[
\land \equiv \lambda xy.xy(\lambda uv.v) \equiv \lambda xy.xyF
\]
\[
\lor \equiv \lambda xy.x(\lambda uv.u)y \equiv \lambda xy.xTy
\]
\[
\neg \equiv \lambda x.x(\lambda uv.v)(\lambda ab.a) \equiv \lambda x.xFT
\]

For Example:

\[
T \land T = (\lambda xy.xyF)TT = (\lambda y.TyF)T = TTF = (\lambda xy.x)TF = T
\]
\[
T \land F = (\lambda xy.xyF)TF = (\lambda y.TyF)F = TFF = (\lambda xy.x)FF = F
\]

**Exercise 1:** Evaluate the expressions:

\[
F \land T = (\lambda xy.xyF)FT = (\lambda y.FyF)T = FTF = (\lambda xy.y)TF = F
\]
\[
F \land F = (\lambda xy.xyF)FF = (\lambda y.FyF)F = FFF = (\lambda xy.y)FF = F
\]
\[
F \lor T = (\lambda xy.xTyF)FT = FTT = (\lambda xy.y)TT = T
\]
\[
\neg T = (\lambda x.xFT)T = TFT = (\lambda xy.x)FT = F
\]
\[
\neg F = (\lambda x.xFT)F = FTF = (\lambda xy.y)FT = T
\]
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:

$$R_F = (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx)))F$$

$$= (\lambda x.F(xx))(\lambda x.F(xx))$$

$$= F((\lambda x.F(xx))(\lambda x.F(xx)))$$

$$= F(R_F)$$

$$= F(F(R_F))$$

$$= \text{and so on}$$

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

**Different paradigms, same power ...:**

$\lambda$-calculus and Turing Machines have the same expressive power!
On to Haskell ...

Some of the major features of Haskell

1. A purely functional language
   - Only “pure” functions
     - In general, functions do not have side effects (do not modify state)
     - some nice features: memoization, recursion
   - Values (variables) are immutable
   - Functions (and operations) always produce entirely new values
   - This is very different than most other PLs

2. Static typing
   - All type checking done at compile time (statically)
   - Employs type inference (unobtrusive—w/out type annotations)

3. “Strong” typing
   - Guarantees a program cannot contain certain type errors
   - Haskell places limits on type conversion (implicit/explicit)
4. Functions are “first-class” objects
   - i.e., used just like any other kind of value in the language
   - e.g., functions can be defined that take functions as parameters (and call them in the function body)
   - Can create new functions during program execution
   - Can store functions in data structures
   - Can pass functions as arguments to other functions
   - Can return functions as values of other functions

5. Lazy evaluation
   - Defer computation until the result is needed
   - One benefit: possible performance gain (no needless computations)
     - e.g., using quicksort, can ask for first (first two, etc.) values, without sorting entire list
   - Another benefit: “infinite” data structures
     - and in particular, the ability to compute with them
     - somewhat similar to iterators (or streams)
   - Another benefit: programmer-defined control structures
     - e.g., short circuit evaluation of \texttt{if-then-else}
     - this means you don’t need special constructs for control flow

6. Expression-oriented
   - All statements return values (e.g., even \texttt{if} statements!)