Functional Programming
CS 340: Programming Paradigms / Patterns
Questions:
- What is functional programming? (What is it not?)
- Some f.p. history/ background, please?
- How is f.p. different from imperative programming?
  - What are some advantages of f.p.?
    (Per Hughes, what "give" feature does it provide?)
  - What are some disadvantages of f.p.?
Functional programming has roots in mathematical logic and computational theory.

In 1936, three formal approaches to computability:

1. Turing machines
2. Kleene's recursive function theory
3. Church's lambda ($\lambda$) calculus

Each specifies:

- set of primitive operations
- rules for structuring operations
- proof theory — "can I compute this?"

—all are equivalent" (what is computable in one model is computable in others)
\( \lambda \) calculus:
- very simple model for functional programming
- basic syntax \( \rightarrow \) universal machine code for f.p. languages

Grammar:
\[
expr := \lambda \text{var. expr} \mid (\text{expr expr})
\]

- identifier (bound variable)
- \( \text{fn.body} \)
- \( \text{fn.call} \)
- \( \text{fn.arg} \)
Evaluation:

- When evaluating a function application, the argument is substituted for all instances of the bound variable in the called function's body.

  e.g. \((\lambda x \cdot x \; 10) \Rightarrow \lambda x . x \Rightarrow 10\)

\((\lambda x \cdot x^2 + 2x + 5 \; 10) \Rightarrow 10^2 + 2 \cdot 10 + 5\)

  allowing arith. eops.
e.g., \( \text{id} = \lambda x.x \) 
\[ \{ \text{id} y \} \Rightarrow y \]

\( \text{fst} = \lambda x. \lambda y. x \) 
\[ \{ \text{fst} 5 \} \Rightarrow \lambda y. 5 \]
\[ \{(\text{fst} 5) \ 6) \Rightarrow 5 \] 

\( \text{app} = \lambda f. \lambda x. (f \ x) \) 
\[ \{(\text{app} \ \text{id})\ 5) \Rightarrow 5 \] 
\[ \{(\text{app} (\text{app} \ \text{fst}) \ 5))\ 10)\Rightarrow 5 \]

\( \text{rec} = \lambda f. (f \ f) \) 
\[ \{(\text{rec} \ \text{id}) \Rightarrow \text{id} \] 
\[ \{(\text{rec} \ \text{rec}) \Rightarrow ? \]
Imperative vs. Functional
Imperative programs are made up of sequences of statements:

\[ a = 10 \]
\[ b = 20 \]
\[ c = a + b \]

Functional programs are based on nesting/composing function calls:

\[ f(w, g(x, y), h(z)) \]
Imperative programs allow values associated with variables to change over time ("state mutations")

\[
\begin{align*}
x &= 5 \\
y &= 6 \\
\text{tmp} &= x \\
x &= y \\
y &= \text{tmp}
\end{align*}
\]

Functional programs do not allow variables to be re-bound. (no assignment statements)

\[
\begin{align*}
\text{a} &= 10 \\
\text{a} &= 20 \\
\text{swap } x \ y &= \text{swap } y \ x
\end{align*}
\]
Imperative languages have built-in "control flow" constructs that alter the default ordering of statements.

\[
a = 10 \\
b = 20 \\
\text{if } a < b \\
\quad \text{max} = b \\
\text{else} \\
\quad \text{max} = a
\]

Functional languages only have expressions and function calls. "Control flow" is meaningless!

\[
\text{max } a \ b = \text{if } a < b \text{ then } b \text{ else } a
\]

\[
\text{sum } 0 = 0 \\
\text{while } x > 0 \\
\quad \text{sum } += x \\
\quad x -= 1
\]

\[
\text{sum } x = x + \text{sum } (x - 1)
\]
Imperative programs may define procedures with "side-effects."

```
def sum(x, y):
    global z
    z += x+y
    return x+y
```

result in different states!

```
z = 0
sum(10, 20)  # result depends solely on input.
```

Functional programs contain no side-effects. i.e., funs only compute their results—nothing else!

```
sum x y = x + y
```

"pure" function—result depends solely on input.
Imperative programs require "strict" evaluation (aka "eager"/"greedy" evaluation)

foo (sum (1, 2), sum (2, 3))

must be executed in order!

foo (print("hello"),
    print("bye"))

Functional programs permit "lazy" evaluation, due to referential transparency.

foo (bar (1, 2, 3, 4),
    baz (2, 3, 4, 5))

foo x y = 10

contain "thanks"
**First-class fn = fn as a value**

Imperative languages have varying support for first-class and higher-order functions. HOF = fn that takes/returns fns. Functional languages universally support first-class and higher-order functions.

```python
def app(f, x):
    f(x)

app(lambda y: y+1, 10)
```

```
int app(int (*f)(int), int x) {
    return f(x);
}
```

```
int int mic(int x) {
    return x+1;
}
```

```
app(mic, 10) => 11
```
Imperative programs support direct input/output

def foo(x):
    y = input
    return x + y

foo is not a pure function!

Functional languages prohibit direct I/O functions, as they introduce side effects!

(How do we do input?)