Functional Programming

• *Functions* are the basic unit of computation
• Functions are values! (“first-class functions”)
  — Functions can take functions as arguments
• No mutable variables*
• Usually contrasted with imperative languages
• Examples: F#, OCaml, Lisp, Haskell, lambda-expressions
The First Functional Language

• Functional languages are older than computers!
• The lambda calculus was invented as a mathematical model of “what can be computed”

Normal math notation                   Lambda calculus
\[ f(x) = x + 1 \]                        \[ \lambda x. x + 1 \]
\[ g(x, y) = y \]                        \[ \lambda x. \lambda y. y \]

Currying: to make a two-argument function, return a function!
The First Functional Language

• Functional languages are older than computers!
• The lambda calculus was invented as a mathematical model of “what can be computed”
• Turing machines were invented for the same purpose!
• Lambda calculus and Turing machines can express all the same computations: lambda calculus is Turing-complete
• More generally, every popular programming language is Turing-complete: they can all compute the same things!
Lambda Calculus Basics

• Functions are values, and functions are the only values!
• A function has two parts:

\[ \lambda x. B \]

  argument name    body (any term, can contain \( x \))
  "bound variable"

• Functions can be \textit{applied} to other terms
• Application is evaluated by replacing the bound variable with the argument in the body

\[ (\lambda x. \lambda y. x) (\lambda z. z) \]
Lambda Calculus Basics

• Functions are values, and functions are the only values!
• A function has two parts:

\[ \lambda x. B \]

- argument name
- body (any term, can contain \( x \))
- “bound variable”
• Functions can be \textit{applied} to other terms
• Application is evaluated by replacing the bound variable with the argument in the body

\[
(\lambda x. \lambda y. x) (\lambda z. z) \quad [x \mapsto \lambda z. z](\lambda y. x)
\]
Lambda Calculus Basics

• Functions are values, and functions are the only values!
• A function has two parts:
  \[ \lambda x. B \]
  argument name \( x \) \quad \text{body (any term, can contain } x \text{)}
  “bound variable”
• Functions can be applied to other terms
• Application is evaluated by replacing the bound variable with the argument in the body
  \[ (\lambda x. \lambda y. x) (\lambda z. z) \quad \lambda y. (\lambda z. z) \]
Lambda Calculus Basics

- Functions are values, and functions are the only values!
- A function has two parts:
  \[ \lambda x. B \]
  
  - argument name
  - body (any term, can contain \( x \))
  - “bound variable”
- Functions can be \textit{applied} to other terms
- Application is evaluated by replacing the bound variable with the argument in the body
  \[ (\lambda x. \lambda y. x) (\lambda z. z) \] evaluates to \( \lambda y. \lambda z. z \)
• Make this more concrete!
• Exercises
Lambda Calculus: Binding and Scope

• \( \lambda x. B \) binds \( x \) in \( B \)

• In other words, wherever \( x \) appears in \( B \), it means “the argument passed to this function”

• Each variable refers to the innermost \( \lambda \)-binding around it

\[
\lambda x. (\lambda x. x (\lambda x. x x)) x
\]

• A variable that is not bound is free, like \( y \) in \( \lambda x. y x \)
Variable Binding

```c
int f(int x) { return x + 1; }
```

```c
int x = 5;
f(x + 2);
```
Lambda Calculus: Substitution

• In general, \((\lambda x. l) \ l_2\) evaluates to \([x \mapsto l_2]l\) ("\(l\) with \(l_2\) substituted for \(x\")

• \((\lambda x. x) \ z\) evaluates to
Lambda Calculus: Substitution

• In general, \((\lambda x. l) \ l_2\) evaluates to \([x \mapsto l_2]l\) (“\(l\) with \(l_2\) substituted for \(x\)”)  

• \((\lambda x. x) \ z\) evaluates to \(z\)  
• \((\lambda x. \lambda y. x) \ z\) evaluates to
Lambda Calculus: Substitution

• In general, \((\lambda x. \, l) \, l_2\) evaluates to \([x \mapsto l_2]l\)
(“\(l\) with \(l_2\) substituted for \(x\)"")

• \((\lambda x. \, x) \, z\) evaluates to \(z\)
• \((\lambda x. \, \lambda y. \, x) \, z\) evaluates to \(\lambda y. \, z\)
• \((\lambda x. \, \lambda y. \, y) \, z\) evaluates to
Lambda Calculus: Substitution

• In general, \((\lambda x. l) \ l_2\) evaluates to \([x \mapsto l_2]l\) ("\(l\) with \(l_2\) substituted for \(x\")

• \((\lambda x. x) \ z\) evaluates to \(z\)
• \((\lambda x. \lambda y. x) \ z\) evaluates to \(\lambda y. z\)
• \((\lambda x. \lambda y. y) \ z\) evaluates to \(\lambda y. y\)
• \((\lambda x. x (\lambda y. y)) \ z\) evaluates to
Lambda Calculus: Substitution

• In general, \((\lambda x. l) \, l_2\) evaluates to \([x \mapsto l_2] l\)
  ("\(l\) with \(l_2\) substituted for \(x\)"")

• \((\lambda x. x) \, z\) evaluates to \(z\)
• \((\lambda x. \lambda y. x) \, z\) evaluates to \(\lambda y. z\)
• \((\lambda x. \lambda y. y) \, z\) evaluates to \(\lambda y. y\)
• \((\lambda x. x \, (\lambda y. y)) \, z\) evaluates to \(z \, (\lambda y. y)\)
• \((\lambda x. x) \, (\lambda y. y)\)
• \((\lambda x. x \, (\lambda x. x)) \, z\) evaluates to
Lambda Calculus: Substitution

• In general, \((\lambda x. l) \ l_2\) evaluates to \([x \mapsto l_2]l\) ("\(l\) with \(l_2\) substituted for \(x\)"")

• \((\lambda x. x) \ z\) evaluates to \(z\)
• \((\lambda x. \lambda y. x) \ z\) evaluates to \(\lambda y. z\)
• \((\lambda x. \lambda y. y) \ z\) evaluates to \(\lambda y. y\)
• \((\lambda x. x (\lambda y. y)) \ z\) evaluates to \(z (\lambda y. y)\)
• \((\lambda x. x (\lambda x. x)) \ z\) evaluates to \(z (\lambda x. x)\)
Lambda Calculus: Substitution

How do we define substitution?

\[ [x \mapsto s]x = s \]
\[ [x \mapsto s]y = y \]
\[ [x \mapsto s](\lambda x. l) = \lambda x. l \]
\[ [x \mapsto s](\lambda y. l) = \lambda y. [x \mapsto s]l \]
\[ [x \mapsto s](l_1 l_2) = ([x \mapsto s]l_1) ([x \mapsto s]l_2) \]
Lambda Calculus: Substitution

How do we define substitution?

\[ [x \mapsto s]x = s \]
\[ [x \mapsto s]y = y \]
\[ [x \mapsto s](\lambda x. l) = \lambda x. l \]
\[ [x \mapsto s](\lambda y. l) = \lambda y. [x \mapsto s]l \quad \text{except...} \]
\[ [x \mapsto s](l_1 \ l_2) = ([x \mapsto s]l_1) \ ([x \mapsto s]l_2) \]
Lambda Calculus: Substitution

How do we define substitution?

\[ [x \mapsto s](\lambda y. l) = \lambda y. [x \mapsto s]l \quad \text{except...} \]

\((\lambda x. \lambda y. x y) y\) evaluates to?
Lambda Calculus: Substitution

How do we define substitution?

\[ [x \mapsto s](\lambda y. l) = \lambda y. [x \mapsto s]l \]

except...

\((\lambda x. \lambda y. x \ y) \ y\) evaluates to \(\lambda y. [x \mapsto y](x \ y)\)

which is \(\lambda y. y \ y\) ... but that’s not right!

It should be “a function that takes an argument, and applies the free variable \(y\) to that argument”
Lambda Calculus: Substitution

How do we define substitution?

\[ [x \mapsto s](\lambda y. l) = \lambda y. [x \mapsto s]l \quad \text{except...} \]

\((\lambda x. \lambda y. x y) \; y\) evaluates to \(\lambda y. [x \mapsto y](x \; y)\)

which is \(\lambda y. y \; y\) ... but that’s not right!

It should be “a function that takes an argument, and applies the free variable \(y\) to that argument”: \(\lambda z. \; y \; z\)
The name of the argument to a function doesn’t really matter

\( \lambda x. x \) is the same as \( \lambda y. y \)

We can always *rename* a bound variable

\[
\lambda x. (\lambda x. x (\lambda x. x x)) \ x
\]
Lambda Calculus: Renaming

• The name of the argument to a function doesn’t really matter
• \( \lambda x. x \) is the same as \( \lambda y. y \)
• We can always rename a bound variable

\[ \lambda x. (\lambda x. x (\lambda y. y y)) x \]
Lambda Calculus: Renaming

• The name of the argument to a function doesn’t really matter
• $\lambda x. x$ is the same as $\lambda y. y$
• We can always *rename* a bound variable

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

• Renaming (sometimes called “alpha-conversion”) shouldn’t change the behavior of a function – in particular, it doesn’t change the effect of substitution
Lambda Calculus: Substitution

How do we define substitution?

\[ [x \mapsto s](\lambda y. l) = \lambda y. [x \mapsto s]l \quad \text{except...} \]

\((\lambda x. \lambda y. x y) y\) is the same as \((\lambda x. \lambda z. x z) y\)
which evaluates to \(\lambda z. [x \mapsto y](x z)\)
which is \(\lambda z. y z\)
Lambda Calculus: Substitution

In OCaml:

```ocaml
let f x y = x + y;;  (* f = fun x -> fun y -> x + y *)
let y = 5;;
let g = f y;;  (* g = fun y -> y + y would be wrong *)
(* g = fun z -> y + z would be right *)
```
Lambda Calculus: Substitution

How do we define substitution?

\[ [x \mapsto s](\lambda y.l) = \lambda y. [x \mapsto s]l, \text{ but first make sure } y \text{ isn’t free in } s \]

“Capture-avoiding substitution”

\((\lambda x. \lambda y. x \ y) \ y\) \ is \ the \ same \ as \ \((\lambda x. \lambda z. x \ z) \ y\)

which \ evaluates \ to \ \(\lambda z. [x \mapsto y](x \ z)\)

which \ is \ \(\lambda z. y \ z\)
Lambda Calculus: Syntax

\[ L ::= <\text{ident}> \mid \lambda <\text{ident}>(). L \mid LL \]
Lambda Calculus: Semantics

\[ L ::= \text{id} \mid \lambda \text{id}. \ L \mid L \ L \]

- Functions are values
- Application is evaluated by substitution
Lambda Calculus: Semantics

\[ L ::= \langle \text{ident} \rangle \mid \lambda \langle \text{ident} \rangle . L \mid L \ L \]

- \( \lambda x. l \) is a value
- Application is evaluated by substitution
Lambda Calculus: Semantics

\[ L ::= \text{id} \mid \lambda \text{id}. L \mid L \, L \]

• \( \lambda x. l \) is a value

\[
\frac{\lambda x \cdot L \rightarrow \lambda x \cdot L'}{L \rightarrow L'}
\]

\[
\frac{(\lambda x \cdot L) \rightarrow x \mapsto L'}{L ightarrow L'}
\]

• “Call by name”
Lambda Calculus: Semantics

$L ::= <\text{ident}> \mid \lambda <\text{ident}>. \ L \mid LL$

\[
\frac{l_1 \to l_1'}{l_1 \ l_2 \to l_1' \ l_2}
\]

\[
\frac{l_2 \to l_2'}{v \ l_2 \to v \ l_2'}
\]

\[
(\lambda x. \ l) \ v \to [x \mapsto v]l
\]

• “Call by value”
Call-By-Name vs. Call-By-Value

$$(\lambda x. \lambda y. y) \, l$$ where $l$ becomes a value in 10 steps

Call-by-name:
$$\rightarrow (\lambda y. y)$$

Call-by-value:
$$\rightarrow (\lambda x. \lambda y. y) \, l_1 \rightarrow (\lambda x. \lambda y. y) \, l_2 \rightarrow \cdots \rightarrow (\lambda x. \lambda y. y) \, v$$
$$\rightarrow (\lambda y. y)$$
Call-By-Name vs. Call-By-Value

$$(\lambda x. \lambda y. y) \ l \text{ where } l \text{ runs forever}$$

$$(\lambda x. x \ x) \ (\lambda x. x \ x) \rightarrow [x \mapsto \lambda x. x \ x](x \ x)$$

which is $$(\lambda x. x \ x) \ (\lambda x. x \ x)!$$

Call-by-name:
$$\rightarrow (\lambda y. y)$$

Call-by-value:
$$\rightarrow (\lambda x. \lambda y. y) \ l \rightarrow (\lambda x. \lambda y. y) \ l \rightarrow \cdots$$
Call-By-Name vs. Call-By-Value

$(\lambda x. \ldots x \ldots x \ldots ) \; l$ where $l$ becomes a value in 10 steps

Call-by-name:
$\rightarrow \; \ldots l \ldots l \ldots \rightarrow \; \ldots l_1 \ldots l \ldots \rightarrow \; \ldots v \ldots l \ldots \rightarrow \; \ldots v \ldots l_1 \ldots \rightarrow \; \ldots$

Call-by-value:
$\rightarrow \; (\lambda x. \ldots x \ldots x \ldots ) \; l_1 \rightarrow \; \ldots \rightarrow \; (\lambda x. \ldots x \ldots x \ldots ) \; v \rightarrow \; \ldots v \ldots v \ldots$
Lambda Calculus: Interpreter