CS421 Lecture 19: λ-Calculus[1]

Mark Hills
mhills@cs.uiuc.edu

University of Illinois at Urbana-Champaign

July 21, 2008

[1] Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

Objectives

Subtitle: "Everything is Really a Function"

This lecture introduces λ-calculus, and uses it to show how functions can represent arbitrary computations and data structures. After this lecture you should ...

- Know the three constructs of λ-calculus.
- Know the terminology: free vs. bound variables, scopes, α-reduction, β-reduction
- Know how to use λ terms to represent arbitrary types.
  - numbers, booleans, pairs, etc

The λ-Calculus: Motivation

All *sequential* programs may be viewed as functions from input (initial state and input values) to output (resulting state and output values).

- The λ-calculus is a mathematical formalism of functions and functional computations
- Provides a simple, flexible model to represent programming languages
- Heavily used in languages research, and highly influential in functional programming

The λ-Calculus

λ-calculus : A language with only three kinds of expressions (called *terms*):

Variables:	e	$\rightarrow$	x	
Abstraction:	e	$\rightarrow$	$\lambda x.e_1$	*i.e., Function creation*
Application:	e	$\rightarrow$	$(e_1\ e_2)$	

Terminology

Occurrence : A location of a subterm in a term

Bound variable : Variable x is said to be bound in $\lambda x.e$.

Scope : The scope of this binding is all of e except any terms within e that are of the form $\lambda x.e_1$.

Bound occurrence : An occurrence of x in e is a *bound occurrence* if it is within the scope of some binding for x.

Free occurrence : Otherwise, an occurrence is said to be a *free occurrence* (aka *free variable*)

How powerful is the λ-Calculus?

Computing Power: .

- The language is Turing Complete – we can express any sequential computation
- We don't have integers, strings, etc. – how do we represent basic data?
- How do we represent recursion?
- Constants, if-expressions, etc. are conveniences to make programming easier. *a.k.a., "syntactic sugar"*

How powerful is the λ-Calculus?

Typed vs. Untyped Lambda Calculus .

- Lambda Calculus has no notion of type. E.g., (ff) is a legal expression for any f!
- Types restrict what functions can be applied to what arguments
- Types are *not* just syntactic sugar: some terms are illegal in the typed calculus
- The simply typed λ-calculus is less powerful than the untyped λ-calculus – it is NOT Turing complete, since it lacks recursion

λ-Calculus in the Real World

- The typed and untyped λ-Calculi enable the theoretical study of sequential programming languages.
- A functional language is essentially the λ-Calculus extended with predefined constants, syntactic constructs, and types. E.g., Lisp, Scheme, functional subsets of ML and OCAML
- We can (mostly) express the λ-Calculus in OCaml: $\lambda x.x =$ `fun x -> x`

Computing with the λ-Calculus

How do we "execute" programs in the λ-calculus? We need some method to systematically process λ terms to get some kind of "result". This requires *reduction*, or *evaluation*, methods. We also need some way to tell when terms are *equivalent*. Finally, we need some notion of terms that are "answers", so we know when to stop computing.

Congruence

Let $\sim$ be a relation on lambda terms. $\sim$ is a **congruence** if

- it is an equivalence relation
- $e_1 \sim e_2$ implies that $(e\ e_1) \sim (e\ e_2)$ and $(e_1\ e) \sim (e_2\ e)$ and $\lambda x.e_1 \sim \lambda x.e_2$

α Conversion

α-conversion is defined as:

$$\lambda x.e \rightarrow^{\alpha} \lambda y.(e[y/x])$$

provided

- y in not free in e
- no free occurrence of x becomes bound when replaced by y

α Conversion Non-Examples

- y is not free in e

$$\lambda x.x\ y \to^{\alpha} \lambda y.y\ y$$

- No free occurrence of x becomes bound when replaced by y

$$\lambda x.\lambda y.x\ y \to^{\alpha} \lambda y.\lambda y.y\ y$$

But, $\lambda x.(\lambda y.y)\ x \to^{\alpha} \lambda y.(\lambda y.y)\ y$; and, $\lambda y.(\lambda y.y)\ y \to^{\alpha} \lambda x.(\lambda y.y)\ x$

α Equivalence

α equivalence is the smallest congruence containing α conversion.

Generally, α-equivalent terms are considered equal – an individual term is a representative of an equivalence class of terms.

Example

Show: $\lambda x.(\lambda y.y\ x)\ x \sim_{\alpha} \lambda y.(\lambda x.x\ y)\ y$.

Example

Show: $\lambda x.(\lambda y.y\ x)\ x \sim_{\alpha} \lambda y.(\lambda x.x\ y)\ y$.

- $\lambda x.(\lambda y.y\ x)\ x \to^{\alpha} \lambda z.(\lambda y.y\ z)\ z$ **so** $\lambda x.(\lambda y.y\ x)\ x \sim_{\alpha} \lambda z.(\lambda y.y\ z)\ z$

Example

Show: $\lambda x.(\lambda y.y\ x)\ x \sim_{\alpha} \lambda y.(\lambda x.x\ y)\ y$.

- $\lambda x.(\lambda y.y\ x)\ x \to^{\alpha} \lambda z.(\lambda y.y\ z)\ z$ **so** $\lambda x.(\lambda y.y\ x)\ x \sim_{\alpha} \lambda z.(\lambda y.y\ z)\ z$
- $\lambda y.y\ z \to^{\alpha} \lambda x.x\ z$ **so** $\lambda z.(\lambda y.y\ z)\ z \sim_{\alpha} \lambda z.(\lambda x.x\ z)\ z$

Example

Show: $\lambda x.(\lambda y.y\ x)\ x \sim_{\alpha} \lambda y.(\lambda x.x\ y)\ y$.

- $\lambda x.(\lambda y.y\ x)\ x \to^{\alpha} \lambda z.(\lambda y.y\ z)\ z$ **so** $\lambda x.(\lambda y.y\ x)\ x \sim_{\alpha} \lambda z.(\lambda y.y\ z)\ z$
- $\lambda y.y\ z \to^{\alpha} \lambda x.x\ z$ **so** $\lambda z.(\lambda y.y\ z)\ z \sim_{\alpha} \lambda z.(\lambda x.x\ z)\ z$
- $\lambda z.(\lambda x.x\ z)\ z \to^{\alpha} \lambda y.(\lambda x.x\ y)\ y$ **so** $\lambda z.(\lambda x.x\ z)\ z \sim_{\alpha} \lambda y.(\lambda x.x\ y)\ y$

Example

Show: $\lambda x.(\lambda y.y\ x)\ x \sim_\alpha \lambda y.(\lambda x.x\ y)\ y$.

- $\lambda x.(\lambda y.y\ x)\ x \rightarrow^\alpha \lambda z.(\lambda y.y\ z)\ z$ **so** $\lambda x.(\lambda y.y\ x)\ x \sim_\alpha \lambda z.(\lambda y.y\ z)\ z$
- $\lambda y.y\ z \rightarrow^\alpha \lambda x.x\ z$ **so** $\lambda z.(\lambda y.y\ z)\ z \sim_\alpha \lambda z.(\lambda x.x\ z)\ z$
- $\lambda z.(\lambda x.x\ z)\ z \rightarrow^\alpha \lambda y.(\lambda x.x\ y)\ y$ **so** $\lambda z.(\lambda x.x\ z)\ z \sim_\alpha \lambda y.(\lambda x.x\ y)\ y$
- Thus $\lambda x.(\lambda y.y\ x)\ x \sim_\alpha \lambda y.(\lambda x.x\ y)\ y$.

Outline
Objectives
λ-Calculus
Computational Model
Data Representation
Terminology
Congruence
α Conversion
α Equivalence
η Reduction
Substitution
β Reduction
α β Equivalence
Evaluation Order

η Reduction

The η rule states that $\lambda x.f\ x \rightarrow^\eta f$ if x is not free in f

- can be useful in each direction
- not equivalent to $(\lambda x.f)\ x$ (this is function application)

Example: $\lambda x.(\lambda y.y)\ x \rightarrow^\eta \lambda y.y$.

Outline
Objectives
λ-Calculus
Computational Model
Data Representation
Terminology
Congruence
α Conversion
α Equivalence
η Reduction
Substitution
β Reduction
α β Equivalence
Evaluation Order

Substitution

Substitution is defined over α-equivalence classes of terms.

$$[N/x]\ P$$

means replace every free occurrence of x in P by N.

- **Requirement**: No free variable in P should become bound in [N/x] P – hence definition over α-equivalence classes
- Renaming/alpha conversion used to avoid name capture
- Often stated as a "hygiene" requirement for substitution, so we can assume it during discussions, but still need to be aware of it

Outline
Objectives
λ-Calculus
Computational Model
Data Representation
Terminology
Congruence
α Conversion
α Equivalence
η Reduction
Substitution
β Reduction
α β Equivalence
Evaluation Order

Substitution Rules

Substitution defined inductively over the structure of λ terms

- $[N/x]x = N$
- $[N/x]y = y$ if $y \neq x$
- $[N/x](e_1\ e_2) = (([N/x]e_1)\ ([N/x]e_2))$
- $[N/x]\lambda x.e = \lambda x.e$
- $[N/x]\lambda y.e = \lambda y.([N/x]e)$ if $y \neq x$ and y not free in N (rename y if needed to ensure)

Outline
Objectives
λ-Calculus
Computational Model
Data Representation
Terminology
Congruence
α Conversion
α Equivalence
η Reduction
Substitution
β Reduction
α β Equivalence
Evaluation Order

Substitution Example

$$[(\lambda x.x\ y)/z](\lambda y.y\ z) = ?$$

Any problems?

Outline
Objectives
λ-Calculus
Computational Model
Data Representation
Terminology
Congruence
α Conversion
α Equivalence
η Reduction
Substitution
β Reduction
α β Equivalence
Evaluation Order

Substitution Example

$$[(\lambda x.x\ y)/z](\lambda y.y\ z) = ?$$

Any problems?

- z in redex (term being substituted into) in scope of y binding
- y is free in the residue (term being substituted)

Substitution Example

$$[(\lambda x.x\ y)/z](\lambda y.y\ z) = ?$$

Any problems?

- z in redex (term being substituted into) in scope of y binding
- y is free in the residue (term being substituted)

$$[(\lambda x.x\ y)/z](\lambda y.y\ z) \rightarrow^{\alpha}$$
$$[(\lambda x.x\ y)/z](\lambda w.w\ z) =$$
$$\lambda w.w(\lambda x.x\ y)$$

Substitution Example 2

$$[(\lambda x.x)/z](\lambda y.y\ z(\lambda z.z)) = ?$$

How do we do substitution here?

Substitution Example 2

$$[(\lambda x.x)/z](\lambda y.y\ z(\lambda z.z)) = ?$$

How do we do substitution here?
Is this correct?

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

Substitution Example 2

$$[(\lambda x.x)/z](\lambda y.y\ z(\lambda z.z)) = ?$$

How do we do substitution here?
Is this correct?

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

No, this is, since we only replace *free* occurrences.

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

β Reduction

β reduction is the essence of computation in the λ calculus.

$$\beta \text{ rule: } (\lambda x.P)\ N \rightarrow^{\beta} [N/x]P$$

- Usually defined on α-equivalence classes of terms

β Reduction Example

$$(\lambda z.(\lambda x.x\ y)\ z)(\lambda y.y\ z) \rightarrow^{\beta}$$

β Reduction Example

$$(\lambda z.(\lambda x.x\ y)\ z)(\lambda y.y\ z) \rightarrow^{\beta}$$
$$(\lambda x.x\ y)\ (\lambda y.y\ z) \rightarrow^{\beta}$$

β Reduction Example

$$(\lambda z.(\lambda x.x\ y)\ z)(\lambda y.y\ z) \rightarrow^{\beta}$$
$$(\lambda x.x\ y)\ (\lambda y.y\ z) \rightarrow^{\beta}$$
$$(\lambda y.y\ z)\ y \rightarrow^{\beta}$$

β Reduction Example

$$(\lambda z.(\lambda x.x\ y)\ z)(\lambda y.y\ z) \rightarrow^{\beta}$$
$$(\lambda x.x\ y)\ (\lambda y.y\ z) \rightarrow^{\beta}$$
$$(\lambda y.y\ z)\ y \rightarrow^{\beta}$$
$$y\ z$$

β Reduction Example 2

$$(\lambda x.x\ x)(\lambda x.x\ x) \rightarrow^{\beta}$$

β Reduction Example 2

$$(\lambda x.x\ x)(\lambda x.x\ x) \rightarrow^{\beta}$$
$$(\lambda x.x\ x)(\lambda x.x\ x) \rightarrow^{\beta}$$

β Reduction Example 2

$$(\lambda x.x\ x)(\lambda x.x\ x) \rightarrow^{\beta}$$
$$(\lambda x.x\ x)(\lambda x.x\ x) \rightarrow^{\beta}$$
$$(\lambda x.x\ x)(\lambda x.x\ x) \rightarrow^{\beta}$$
$$(\lambda x.x\ x)(\lambda x.x\ x) \rightarrow^{\beta}$$
$$(\lambda x.x\ x)(\lambda x.x\ x) \rightarrow^{\beta}$$
$$(\lambda x.x\ x)(\lambda x.x\ x) \rightarrow^{\beta}$$
$$\ldots$$

α β Equivalence

- α β equivalence is the smallest congruence containing α equivalence and β reduction
- **Definition**: a term is in *normal form* if no subterm is α equivalent to a term that can be β reduced – i.e. there is no way to reduce the term further
- hard fact (Church-Rosser): if e_1 and e_2 are $\alpha\beta$-equivalent and both are normal forms, then they are α equivalent

Order of Evaluation

Order of Evaluation refers to the order of operations we choose to perform when reducing λ terms. For instance, should we substitute a λ term before or after reducing it? With this, there are two important points:

- Not all terms reduce to normal forms
- Not all reduction strategies will produce a normal form if one exists

Lazy Evaluation

- Always reduce the left-most, top-most application
- Stop when left-most term is not an application of an abstraction to a term

Note here that we choose to **not** evaluate under a λ – in some sense we require the term to be on "top", so we don't execute the bodies of functions before calling them.

Eager Evaluation

- (Eagerly) reduce left of top-most application to an abstraction
- Then (eagerly) reduce argument
- Then β-reduce the application (the abstraction and argument)

Key point here: arguments reduced *before* β-reduction, not after

Evaluation Order Example 1

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

Evaluation Order Example 1

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

For lazy evaluation, first reduce the left-most application:

$$(\lambda z.(\lambda x.x))((\lambda y.y\ y)(\lambda y.y\ y)) \rightarrow^{\beta} (\lambda x.x)$$

Evaluation Order Example 1

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

For eager evaluation, first reduce the left-most operator of the top-most application to an abstraction.

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

Evaluation Order Example 1

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

For eager evaluation, first reduce the left-most operator of the top-most application to an abstraction.

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

It was, so no changes. Now, reduce the argument:

$$\begin{array}{l}(\lambda z.(\lambda x.x))((\lambda y.y\ y)(\lambda y.y\ y)) \to^{\beta} \\ (\lambda z.(\lambda x.x))((\lambda y.y\ y)(\lambda y.y\ y)) \to^{\beta} \\ (\lambda z.(\lambda x.x))((\lambda y.y\ y)(\lambda y.y\ y)) \to^{\beta} \\ \ldots\end{array}$$

Evaluation Order Example 2

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

Evaluation Order Example 2

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

Lazy evaluation:

$$(\lambda x.x\ x)((\lambda y.y\ y)(\lambda z.z)) \to^{\beta}$$

Evaluation Order Example 2

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

Lazy evaluation:

$$(\lambda x.\boxed{x}\ \boxed{x})\underline{((\lambda y.y\ y)(\lambda z.z))} \to^{\beta}$$

Evaluation Order Example 2

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

Lazy evaluation:

$$\begin{array}{c}(\lambda x.\boxed{x}\ \boxed{x})\underline{((\lambda y.y\ y)(\lambda z.z))} \to^{\beta} \\ \boxed{((\lambda y.y\ y)(\lambda z.z))}\ \boxed{((\lambda y.y\ y)(\lambda z.z))}\end{array}$$

Evaluation Order Example 2

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

Lazy evaluation:

$$(\lambda x.x\ x)((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}$$
$$((\lambda y.y\ y)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z))$$

Evaluation Order Example 2

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

Lazy evaluation:

$$(\lambda x.x\ x)((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}$$
$$((\lambda y.y\ y)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z))$$

Evaluation Order Example 2

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

Lazy evaluation:

$$(\lambda x.x\ x)((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}$$
$$((\lambda y.y\ y)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}$$
$$((\lambda z.z)\ (\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z))$$

Evaluation Order Example 2

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

Lazy evaluation:

$$(\lambda x.x\ x)((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}$$
$$((\lambda y.y\ y)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}$$
$$((\lambda z.z)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z))$$

Evaluation Order Example 2

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

Lazy evaluation:

$$(\lambda x.x\ x)((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}$$
$$((\lambda y.y\ y)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}$$
$$((\lambda z.z)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z))$$

Evaluation Order Example 2

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

Lazy evaluation:

$$(\lambda x.x\ x)((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}$$
$$((\lambda y.y\ y)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}$$
$$((\lambda z.z)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}$$
$$(\lambda z.z)\ ((\lambda y.y\ y)(\lambda z.z))$$

Evaluation Order Example 2

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

Lazy evaluation:

$$\begin{array}{c}(\lambda x.x\ x)((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}\\ ((\lambda y.y\ y)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}\\ ((\lambda z.z)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}\\ (\lambda z.\boxed{z})\ \underline{((\lambda y.y\ y)(\lambda z.z))}\end{array}$$

Evaluation Order Example 2

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

Lazy evaluation:

$$\begin{array}{c}(\lambda x.x\ x)((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}\\ ((\lambda y.y\ y)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}\\ ((\lambda z.z)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}\\ (\lambda z.\boxed{z})\ \underline{((\lambda y.y\ y)(\lambda z.z))} \rightarrow^{\beta}\\ (\lambda y.y\ y)(\lambda z.z)\end{array}$$

Evaluation Order Example 2

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

Lazy evaluation:

$$\begin{array}{c}(\lambda x.x\ x)((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}\\ ((\lambda y.y\ y)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}\\ ((\lambda z.z)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}\\ (\lambda z.\boxed{z})\ \underline{((\lambda y.y\ y)(\lambda z.z))} \rightarrow^{\beta}\\ (\lambda y.y\ y)(\lambda z.z) \rightarrow^{\beta}\\ (\lambda z.z)(\lambda z.z) \rightarrow^{\beta}\\ \lambda z.z\end{array}$$

Evaluation Order Example 2

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

Lazy evaluation:

$$\begin{array}{c}(\lambda x.x\ x)((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}\\ ((\lambda y.y\ y)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}\\ ((\lambda z.z)(\lambda z.z))\ ((\lambda y.y\ y)(\lambda z.z)) \rightarrow^{\beta}\\ (\lambda z.\boxed{z})\ \underline{((\lambda y.y\ y)(\lambda z.z))} \rightarrow^{\beta}\\ (\lambda y.y\ y)(\lambda z.z) \rightarrow^{\beta}\\ (\lambda z.z)(\lambda z.z) \rightarrow^{\beta}\\ \lambda z.z \sim_{\beta} ((\lambda y.y\ y)(\lambda z.z))\end{array}$$

Evaluation Order Example 2

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

Eager evaluation:

$$\begin{array}{c}(\lambda x.x\ x)\ \boxed{((\lambda y.y\ y)(\lambda z.z))} \rightarrow^{\beta}\\ (\lambda x.x\ x)\ \boxed{((\lambda z.z)(\lambda z.z))} \rightarrow^{\beta}\\ (\lambda x.x\ x)\ \boxed{(\lambda z.z)} \rightarrow^{\beta}\\ (\lambda z.z)(\lambda z.z)\end{array}$$

But...

Great, we can compute! But with what? We have functions, but we don't have data yet!

But...

Great, we can compute! But with what? We have functions, but we don't have data yet!

- **Key Concept**: We can encode data as functions!

How to Represent (Free) Data Structures (First Pass)

- Suppose τ is a type with n constructors with no arguments
$$C_1, \cdots, C_n$$
- We can represent each term as an abstraction
$$C_i \rightarrow \lambda x_1 \cdots \lambda x_n . x_i$$
- Think: you give me what to return in each case (like a match statement), I'll apply the correct case based on how I was constructed

How to Represent Booleans

For `bool = True | False`

- `True` $\rightarrow \lambda x.\lambda y.x$
- `False` $\rightarrow \lambda x.\lambda y.y$

Some New Notation

From here out, we will write

- $\lambda x_1\ x_2 \cdots x_n.e$ for $\lambda x_1.\ \lambda x_2. \cdots \lambda x_n.e$
- $e_1 e_2 \cdots e_n$ for $((e_1\ e_2) \cdots e_n)$

How to Write Functions over Data Structures

We have basic data, but we need to be able to use them. We still need some kind of case or match functionality.

```
match c with
  | C_1 -> x_1
  | C_2 -> x_2
  ...
  | C_n -> x_n
```

We can encode this as
$$\lambda c\ x_1\ x_2\ \cdots\ x_n.c\ x_1\ x_2\ \cdots\ x_n$$

- **Key Idea**: give me what to do in each case and give me a case, and I'll apply that case

How to Write Functions over Booleans

How do we represent conditionals?

- `if b then c else d`
- Represent as
$$\texttt{if\_then\_else} \equiv \lambda b\ c\ d.b\ c\ d$$

Booleans Example

```
not b = if b then False else True
= if_then_else b False True
```
$= (\lambda \texttt{b c d.b c d})\ \texttt{b}\ (\lambda \texttt{x y.y})(\lambda \texttt{x y.x})$

$= \texttt{b}(\lambda \texttt{x y.y})(\lambda \texttt{x y.x})$

So, lift this to:

$\texttt{not} \equiv \lambda \texttt{b.b}(\lambda \texttt{x y.y})(\lambda \texttt{x y.x})$

And and Or are similar, you should try to devise them yourself...

How to Represent (Free) Data Structures (Second Pass)

- Suppose τ is a type with n constructors
$$C_1\ t_{11} \cdots t_{1k}, \cdots, C_n\ t_{n1} \cdots t_{nm}$$
- Represent each term as an abstraction:
$$C_i\ t_{i1} \cdots t_{ij} \rightarrow \lambda \texttt{x}_1 \cdots \texttt{x}_n.\texttt{x}_i\ t_{i1} \cdots t_{ij}$$
or
$$C_i \rightarrow \lambda\ t_{i1} \cdots t_{ij}\ \texttt{x}_1 \cdots \texttt{x}_n.\texttt{x}_i\ t_{i1} \cdots t_{ij}$$
- **Key Idea**: You need to give each constructor its arguments first

How to Represent Pairs

Pair has one constructor, the comma, that takes two arguments.

- $(\texttt{a,b}) \rightarrow \lambda \texttt{x.x a b}$
- $(\_,\_) \rightarrow \lambda \texttt{a b x.x a b}$

Functions over Pairs

- $\texttt{fst} \equiv \lambda \texttt{p.p}(\lambda \texttt{x y.x})$
$$\begin{array}{c} \texttt{fst}(\texttt{u},\texttt{v}) \rightarrow \\ (\lambda \texttt{p.p}(\lambda \texttt{x y.x}))((\lambda \texttt{a b x.x a b})\ \texttt{u v}) \rightarrow \\ (\lambda \texttt{p.p}(\lambda \texttt{x y.x}))(\lambda \texttt{x.x u v}) \rightarrow \\ (\lambda \texttt{x.x u v})(\lambda \texttt{x y.x}) \rightarrow \\ (\lambda \texttt{x y.x})\ \texttt{u v} \rightarrow \\ (\lambda \texttt{y.u})\ \texttt{v} \rightarrow \\ \texttt{u} \end{array}$$
- $\texttt{snd} \equiv \lambda \texttt{p.p}(\lambda \texttt{x y.y})$

How to Represent (Free) Data Structures (Third Pass)

- Suppose τ is a type with n constructors
$$C_1\ t_{11} \cdots t_{1k}, \cdots, C_n\ t_{n1} \cdots t_{nm}$$
- Suppose $t_{ih} : \tau$ (i.e. it is recursive); then, in place of a value t_{ih} we use a function to compute the recursive value $t_{ih}\ \texttt{x}_1 \cdots \texttt{x}_n$
$$C_i \rightarrow \lambda\ t_{i1} \cdots t_{ij}\ \texttt{x}_1 \cdots \texttt{x}_n.\texttt{x}_i\ t_{i1} \cdots (t_{ih}\texttt{x}_1 \cdots \texttt{x}_n) \cdots t_{ij}$$

Representing Natural Numbers

- $\texttt{nat} \equiv \texttt{Suc nat} \mid \texttt{0}$
- $\texttt{0} \equiv \lambda \texttt{f x.x}$
- $\texttt{Suc n} \equiv \lambda \texttt{f x.f(n f x)}$
- $\texttt{Suc} \equiv \lambda \texttt{n f x.f(n f x)}$

This numeric representation is referred to as *Church Numerals* for Alonzo Church, the creator of the λ-calculus.

Some Church Numerals

$$
\begin{aligned}
\texttt{Suc 0} &= (\lambda\texttt{n f x.f(n f x)})(\lambda\texttt{f x.x}) \rightarrow \\
&\lambda\texttt{f x.f}((\lambda\texttt{f x.x})\texttt{f x}) \rightarrow \\
&\lambda\texttt{f x.f}((\lambda\texttt{x.x})\texttt{x}) \rightarrow \\
&\lambda\texttt{f x.f x}
\end{aligned}
$$

So, this says to apply the function to its argument once. This defines numbers, in some sense, in terms of their *potential* – what can we do with 0 of something, or 1 of something, 2, etc.

Some Church Numerals

$$
\begin{aligned}
\texttt{Suc(Suc 0)} &= (\lambda\texttt{n f x.f(n f x)})(\texttt{Suc 0}) \rightarrow \\
&(\lambda\texttt{n f x.f(n f x)})(\lambda\texttt{f x.f x}) \rightarrow \\
&\lambda\texttt{f x.f}((\lambda\texttt{f x.f x})\texttt{f x}) \rightarrow \\
&\lambda\texttt{f x.f}((\lambda\texttt{x.f x})\texttt{x}) \rightarrow \\
&\lambda\texttt{f x.f(f x)}
\end{aligned}
$$

In general, for any number n, we will apply function f a total of n times.

Adding Church Numerals

For two numbers n and m, we will have Church numerals $\lambda\texttt{f x.f}^n\texttt{x}$ and $\lambda\texttt{f x.f}^m\texttt{x}$ respectively. Adding the two, we would expect to get something of the form $\lambda\texttt{f x.f}^{n+m}\texttt{x}$. What would this look like?

$$
\begin{aligned}
\texttt{n} + \texttt{m} &= \lambda\texttt{f x.f}^{n+m}\texttt{x} = \\
&\lambda\texttt{f x.f}^n(\texttt{f}^m\texttt{x}) = \\
&\lambda\texttt{f x.n f(m f x)} \\
&\text{or} \\
+ &\equiv \lambda\texttt{n m f x.n f(m f x)}
\end{aligned}
$$

Multiplying Church Numerals

For two numbers n and m, we will have Church numerals $\lambda\texttt{f x.f}^n\texttt{x}$ and $\lambda\texttt{f x.f}^m\texttt{x}$ respectively. Multiplying the two, we would expect to get something of the form $\lambda\texttt{f x.f}^{n\times m}\texttt{x}$. What would this look like?

$$
\begin{aligned}
\texttt{n} \times \texttt{m} &= \lambda\texttt{f x.f}^{n\times m}\texttt{x} = \\
&\lambda\texttt{f x.}(\texttt{f}^m)^n\ \texttt{x} = \\
&\lambda\texttt{f x.n(m f)x} \\
&\text{or} \\
\times &\equiv \lambda\texttt{n m f x.n(m f)x}
\end{aligned}
$$

Primitive Recursion over Nat

Recall folding:

```
fold f z n =
  match n with 0 -> z
             | Suc m -> f (fold f z m)
```

We can represent this as:

$$\texttt{fold} \equiv \lambda\texttt{f z n.n f z}$$

For instance:

$$
\begin{aligned}
\texttt{is\_zero n} &= \texttt{fold}(\lambda\texttt{r.False})\texttt{True n} = \\
&(\lambda\texttt{f x.f}^n\texttt{x})(\lambda\texttt{r.False})\texttt{True} = \\
&((\lambda\texttt{r.False})^n)\texttt{True} = \\
&\texttt{if n} = 0\ \texttt{then True else False}
\end{aligned}
$$

Predecessor

Predecessor is, surprisingly, rather painful. First we define an auxilliary function that calculates the predecessor as a pair – the predecessor of 20 would, as a pair, be listed as $(20, 19)$:

$$
\begin{aligned}
&\texttt{let pred\_aux n} = \\
&\texttt{match n with } 0 = (0,0) \mid \texttt{Suc m} = \\
&(\texttt{Suc(fst(pred\_aux m)), fst(pred\_aux m)}) = \\
&\texttt{fold}(\lambda\texttt{r.(Suc(fst r), fst r)})\ (0,0)\ \texttt{n}
\end{aligned}
$$

With this, we can now devise a predecessor function directly:

$$
\begin{aligned}
\texttt{pred} &\equiv \lambda\texttt{n.snd(pred\_aux n)n} = \\
&\lambda\texttt{n.snd(fold}(\lambda\texttt{r.(Suc(fst r), fst r)})\ (0,0)\ n)
\end{aligned}
$$

Recursion

To support recursion *directly*, versus by just using textual substitution, we want a λ-term $\mathtt{Y}$ such that for any term $\mathtt{R}$ the following holds:

$$\mathtt{Y\ R} = \mathtt{R(YR)}$$

$\mathtt{Y}$ needs to use replication to "remember" $\mathtt{R}$.

$$\mathtt{Y} = \lambda \mathtt{y}.(\lambda \mathtt{x}.\mathtt{y}(\mathtt{x\ x}))(\lambda \mathtt{x}.\mathtt{y}(\mathtt{x\ x}))$$
$$\mathtt{Y\ R} = (\lambda \mathtt{x}.\mathtt{R}(\mathtt{x\ x}))(\lambda \mathtt{x}.\mathtt{R}(\mathtt{x\ x})) =$$
$$\mathtt{R}((\lambda \mathtt{x}.\mathtt{R}(\mathtt{x\ x}))(\lambda \mathtt{x}.\mathtt{R}(\mathtt{x\ x})))$$

Note that this **requires** lazy evaluation!

Factorial

We can define factorial as
$\mathtt{F} = \lambda \mathtt{f\ n.if\ n} = 0\ \mathtt{then}\ 1\ \mathtt{else\ n} \times \mathtt{f}(\mathtt{n} - 1)$. Now,

$$\mathtt{Y\ F}\ 3 = \mathtt{F(Y\ F)}\ 3 =$$
$$\mathtt{if}\ 3 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 3 \times ((\mathtt{Y\ F})(3 - 1)) =$$
$$3 \times (\mathtt{Y\ F})\ 2 =$$
$$3 \times (\mathtt{F(Y\ F)}\ 2) =$$
$$3 \times (\mathtt{if}\ 2 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 2 \times (\mathtt{Y\ F})(2 - 1)) =$$
$$3 \times (2 \times (\mathtt{Y\ F})1) =$$
$$3 \times (2 \times (\mathtt{F(Y\ F)}1)) = \cdots$$
$$3 \times (2 \times (1 \times (\mathtt{if}\ 0 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 0 \times (\mathtt{Y\ F})(0 - 1)))) =$$
$$3 \times 2 \times 1 \times 1 = 6$$

Y in OCaml

```
# let rec y f = f (y f);;
val y : ('a -> 'a) -> 'a = <fun>
# let mk_fact =
  fun f n -> if n = 0 then 1 else n * f(n - 1);;
val mk_fact : (int -> int) -> int -> int = <fun>
# y mk_fact;;
Stack overflow during evaluation (looping recursion?).
```


Eager Eval Y in OCaml

```
# let rec y f x = f (y f) x;;
val y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
# y mk_fact;;
- : int -> int = <fun>
# y mk_fact 5;;
- : int = 120
```


Some other Combinators

- $\mathtt{I} = \lambda \mathtt{x}.\mathtt{x}$
- $\mathtt{K} = \lambda \mathtt{x}.\lambda \mathtt{y}.x$
- $\mathtt{K}_* = \lambda \mathtt{x}.\lambda \mathtt{y}.y$
- $\mathtt{S} = \lambda \mathtt{x}.\lambda \mathtt{y}.\lambda \mathtt{z}.\mathtt{x\ z}(\mathtt{y\ z})$