Lambda Calculus

Outline

Introduction and history

Definition of lambda calculus
- Syntax and operational semantics
- Minutia of β-reduction
- Reduction strategies

Programming with lambda calculus
- Church encodings
- Recursion

De Bruijn indices

What is the lambda calculus?

A very **simple**, but **Turing complete**, programming language

- created before concept of *programming language* existed!
- helped to define what *Turing complete* means!

Lambda calculus syntax

$$v \in Var \quad ::= \quad \mathbf{x} \mid \mathbf{y} \mid \mathbf{z} \mid \ldots$$

$$\begin{array}{lcll} e \in Exp & ::= & v & \textit{variable reference} \\ & \mid & e\ e & \textit{application} \\ & \mid & \lambda v.\, e & \textit{(lambda) abstraction} \end{array}$$

Examples

$\mathtt{x} \qquad \lambda\mathtt{x.\,y} \qquad \mathtt{x\ y} \qquad (\lambda\mathtt{x.\,y})\ \mathtt{x}$

$\lambda\mathtt{f.\,}(\lambda\mathtt{x.\,f\ (x\ x)})\ (\lambda\mathtt{x.\,f\ (x\ x)})$

Correspondence to Haskell

Lambda calculus is the **theoretical foundation** for **functional programming**

Lambda calculus	Haskell
x	`x`
$f\ x$	`f x`
$\lambda x.x$	`\x -> x`
$(\lambda f.f\ x)\ (\lambda y.y)$	`(\f -> f x) (\y -> y)`

Similar to Haskell with only: variables, application, anonymous functions

- amazingly, we don't lose anything by omitting all of the other features! (for a particular definition of "anything")

Early history of the lambda calculus

Alonzo Church

Origin of the lambda calculus:

- **Alonzo Church** in 1936, to formalize "computable function"
- proves Hilbert's *Entscheidungsproblem* undecidable
  - provide an algorithm to decide truth of arbitrary propositions

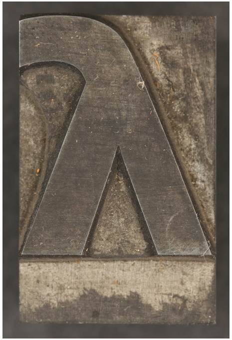
Meanwhile, in England …

- young **Alan Turing** invents the Turing machine
- devises *halting problem* and proves undecidable

Turing heads to Princeton, studies under Church

- prove lambda calculus, Turing machine, general recursion are equivalent
- **Church–Turing thesis**: these capture all that can be computed

Why lambda?



Evolution of notation for a **bound variable**:

- Whitehead and Russell, *Principia Mathematica*, 1910
  - $2\hat{x} + 3$ – corresponds to $f(x) = 2x + 3$
- Church's early handwritten papers
  - $\hat{x}.\, 2x + 3$ – makes scope of variable explicit
- Typesetter #1
  - $\wedge x.\, 2x + 3$ – couldn't typeset the circumflex!
- Typesetter #2
  - $\lambda x.\, 2x + 3$ – picked a prettier symbol

Barendregt, *The Impact of the Lambda Calculus in Logic and Computer Science*, 1997

Impact of the lambda calculus

Turing machine: theoretical foundation for **imperative languages**

- Fortran, Pascal, C, C++, C#, Java, Python, Ruby, JavaScript, ...

Lambda calculus: theoretical foundation for **functional languages**

- Lisp, ML, Haskell, OCaml, Scheme/Racket, Clojure, F#, Coq, ...

In **programming languages research**:

- common language of discourse, formal foundation
- starting point for new features
  - extend syntax, type system, semantics
  - reveals precise impact and utility of feature

Outline

Introduction and history

Definition of lambda calculus

- Syntax and operational semantics
- Minutia of β-reduction
- Reduction strategies

Programming with lambda calculus

- Church encodings
- Recursion

De Bruijn indices

Syntax

Lambda calculus syntax

$$v \in Var \quad ::= \quad \mathbf{x} \mid \mathbf{y} \mid \mathbf{z} \mid \ldots$$

$$\begin{array}{rcll} e \in Exp & ::= & v & \textit{variable reference} \\ & \mid & e\ e & \textit{application} \\ & \mid & \lambda v.\, e & \textit{(lambda) abstraction} \end{array}$$

Abstractions extend as far right as possible

so … $\lambda\mathbf{x}.\,\mathbf{x}\ \ \mathbf{y} \ \equiv\ \lambda\mathbf{x}.\,(\mathbf{x}\ \ \mathbf{y})$

NOT $\cancel{(\lambda\mathbf{x}.\,\mathbf{x})\ \ \mathbf{y}}$

Syntactic sugar

Multi-parameter functions:

$$\lambda\mathbf{x}.\,(\lambda\mathbf{y}.\,e) \equiv \lambda\mathbf{x}\,\mathbf{y}.\,e$$

$$\lambda\mathbf{x}.\,(\lambda\mathbf{y}.\,(\lambda\mathbf{z}.\,e)) \equiv \lambda\mathbf{x}\,\mathbf{y}\,\mathbf{z}.\,e$$

Application is left-associative:

$$(e_1\ e_2)\ e_3 \equiv e_1\ e_2\ e_3$$

$$((e_1\ e_2)\ e_3)\ e_4 \equiv e_1\ e_2\ e_3\ e_4$$

$$e_1\ (e_2\ e_3) \equiv e_1\ (e_2\ e_3)$$

β-reduction: basic idea

$$e \in Exp \quad ::= \quad v \mid e\ e \mid \lambda v.\, e$$

A **redex** is an expression of the form: $(\lambda v.\, e_1)\ e_2$

(an application with an abstraction on left)

Reduce by **substituting** e_2 for every reference to v in e_1

write this as: $[e_2/v]e_1$

lots of different notations for this!

$[v/e_2]e_1$

$e_1[v/e_2]$

$e_1[v := e_2]$

$[v \mapsto e_2]e_1$

Simple example

$(\lambda x.\, x\ \ y\ \ x)\ \ z \quad \mapsto \quad z\ \ y\ \ z$

Operational semantics

$$e \in Exp \quad ::= \quad v \mid e \; e \mid \lambda v.\, e$$

Reduction semantics

$$(\lambda v.\, e_1)\; e_2 \mapsto [e_2/v]e_1 \qquad \frac{e \mapsto e'}{\lambda v.\, e \mapsto \lambda v.\, e'}$$

$$\frac{e_1 \mapsto e_1'}{e_1\; e_2 \mapsto e_1'\; e_2} \qquad \frac{e_2 \mapsto e_2'}{e_1\; e_2 \mapsto e_1\; e_2'}$$

Note: Reduction order is ambiguous!

Exercise

Apply β-reduction in the following expressions

Round 1:

- `(λx. x) z`
- `(λx y. x) z`
- `(λx y. x) z u`

Round 2:

- `(λx. x x) (λy. y)`
- `(λx. (λy. y) z)`
- `(λx. (x (λy. x))) z`

$$e \in Exp \quad ::= \quad v \mid e\ e \mid \lambda v.\, e$$

$$(\lambda v.\, e_1)\ e_2 \mapsto [e_2/v]e_1 \qquad \frac{e \mapsto e'}{\lambda v.\, e \mapsto \lambda v.\, e'}$$

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2} \qquad \frac{e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'}$$

Outline

Variable scoping

$$e \in Exp \quad ::= \quad v \mid e\ e \mid \lambda v.\, e$$

An abstraction consists of:

1. a **variable declaration**
2. a **function body** – the variable can be **referenced** in here

The **scope** of a declaration: the parts of a program where it can be referenced

A reference is bound by its **innermost** declaration

Mini-exercise: $(\lambda \mathbf{x}.\, e_1\ (\lambda \mathbf{y}.\, e_2\ (\lambda \mathbf{x}.\, e_3)))\ (\lambda \mathbf{z}.\, e_4)$

- What is the scope of each variable declaration?

Free and bound variables

$$e \in Exp \quad ::= \quad v \mid e\ e \mid \lambda v.\, e$$

A variable v is **free** in e if:

- v is referenced in e
- the reference is *not* enclosed in an abstraction declaring v (within e)

If v is referenced and enclosed in such an abstraction, it is **bound**

Closed expression: an expression with no free variables

- equivalently, an expression where all variables are bound

Exercise

$$e \in Exp \quad ::= \quad v \mid e\ e \mid \lambda v.\, e$$

1. Define the abstract syntax of lambda calculus as a Haskell data type
2. Define a function: `free :: Exp -> Set Var`
   the set of free variables in an expression
3. Define a function: `closed :: Exp -> Bool`
   no free variables in an expression

Potential problem: variable capture

Principles of variable bindings:

1. variables should be bound according to their **static scope**
   - $\lambda x.\,(\lambda y.\,(\lambda x.\,y\ \ x))\ \ x \ \mapsto\ \lambda x.\,\lambda x.\,x\ \ x$
2. how we name bound variables doesn't really matter
   - $\lambda x.\,x \equiv \lambda y.\,y \equiv \lambda z.\,z$ (α-equivalence)

If violated, we can't reason about functions separately from their use!

Example with naive substitution

A binary function that always returns its first argument: $\lambda x\,y.\,x$ … or does it?

$$(\lambda x\,y.\,x)\ \ y\ \ u \ \mapsto\ (\lambda y.\,y)\ \ u \ \mapsto\ u$$

Solution: capture-avoiding substitution

Capture-avoiding (safe) substitution: $[e/v]e'$

$$
\begin{array}{lcll}
[e/v]v & = & e & \\
[e/v]w & = & w & v \neq w \\
[e/v](e_1\ e_2) & = & [e/v]e_1\ \ [e/v]e_2 & \\
[e/v](\lambda u.\, e') & = & \lambda w.\, [e/v]([w/u]e') & w \notin \{v\} \cup FV(\lambda u.\, e') \cup FV(e)
\end{array}
$$

$FV(e)$ is the set of all free variables in e

Example with safe substitution

$(\lambda \mathtt{x}\,\mathtt{y}.\,\mathtt{x})\ \mathtt{y}\ \mathtt{u}$

$\mapsto [\mathtt{y}/\mathtt{x}](\lambda \mathtt{y}.\,\mathtt{x})\ \mathtt{u} = (\lambda \mathtt{z}.\,[\mathtt{y}/\mathtt{x}]([\mathtt{z}/\mathtt{y}]\mathtt{x}))\ \mathtt{u} = (\lambda \mathtt{z}.\,[\mathtt{y}/\mathtt{x}]\mathtt{x})\ \mathtt{u} = (\lambda \mathtt{z}.\,\mathtt{y})\ \mathtt{u}$

$\mapsto [\mathtt{u}/\mathtt{z}]\mathtt{y} = \mathtt{y}$

Example

Recall example: $\lambda \mathbf{x}.\,(\lambda \mathbf{y}.\,(\lambda \mathbf{x}.\,\mathbf{y}\ \ \mathbf{x}))\ \ \mathbf{x} \ \mapsto\ \lambda \mathbf{x}.\,\lambda \mathbf{x}.\,\mathbf{x}\ \ \mathbf{x}$

Reduction with safe substitution

$$\begin{aligned}
&\lambda \mathbf{x}.\,(\lambda \mathbf{y}.\,(\lambda \mathbf{x}.\,\mathbf{y}\ \ \mathbf{x}))\ \ \mathbf{x}\\
&\quad\mapsto \lambda \mathbf{x}.\,[\mathbf{x}/\mathbf{y}](\lambda \mathbf{x}.\,\mathbf{y}\ \ \mathbf{x}) \;=\; \lambda \mathbf{x}.\,\lambda \mathbf{z}.\,[\mathbf{x}/\mathbf{y}]([\mathbf{z}/\mathbf{x}](\mathbf{y}\ \ \mathbf{x})) \;=\; \lambda \mathbf{x}.\,\lambda \mathbf{z}.\,[\mathbf{x}/\mathbf{y}](\mathbf{y}\ \ \mathbf{z})\\
&\qquad= \lambda \mathbf{x}.\,\lambda \mathbf{z}.\,\mathbf{x}\ \ \mathbf{z}
\end{aligned}$$

Outline

Normal form

Question: what is a **value** in the lambda calculus?

- how do we know when we're done reducing?

One answer: a value is an expression that **contains no redexes**

- called β**-normal form**

Not all expressions can be reduced to a value!

$$(\lambda x.\, x\, x)\ (\lambda x.\, x\, x) \mapsto (\lambda x.\, x\, x)\ (\lambda x.\, x\, x) \mapsto (\lambda x.\, x\, x)\ (\lambda x.\, x\, x) \mapsto \ldots$$

Does reduction order matter?

Recall: operational semantics is ambiguous

- in what order should we β-reduce redexes?
- does it matter?

$e \mapsto e' \quad \subseteq \quad \mathit{Exp} \times \mathit{Exp}$

$$(\lambda v.\, e_1)\ e_2 \mapsto [e_2/v]e_1 \qquad \frac{e \mapsto e'}{\lambda v.\, e \mapsto \lambda v.\, e'}$$

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2} \qquad \frac{e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'}$$

$e \mapsto^* e' \quad \subseteq \quad \mathit{Exp} \times \mathit{Exp}$

$$s \mapsto^* s$$

$$\frac{s \mapsto s' \qquad s' \mapsto^* s''}{s \mapsto^* s''}$$

Church–Rosser Theorem

Reduction is confluent

If $e \mapsto^* e_1$ and $e \mapsto^* e_2$, then
$\exists e'$ such that $e_1 \mapsto^* e'$ and $e_2 \mapsto^* e'$

$$\begin{tikzcd}
 & e_1 \arrow[dr, "\mapsto^*"] & \\
e \arrow[ur, "\mapsto^*"] \arrow[dr, "\mapsto^*"'] & & e' \\
 & e_2 \arrow[ur, "\mapsto^*"'] &
\end{tikzcd}$$

Corollary: any expression has **at most one normal form**

- if it exists, we can still reach it after any sequence of reductions
- … but if we pick badly, we might never get there!

Example: $\mathbf{(\lambda x.\, y)\ ((\lambda x.\, x\, x)\ (\lambda x.\, x\, x))}$

Reduction strategies

Redex positions

leftmost redex: the redex with the leftmost λ

outermost redex: any redex that is not part of another redex

innermost redex: any redex that does not contain another redex

Label redexes

```
(λx.
  (λy.x) z
  ((λy.y) z))
(λy.z)
```

Reduction strategies

normal order reduction: reduce the leftmost redex

applicative order reduction: reduce the leftmost of the innermost redexes

Compare reductions: `(λx.y) ((λx.x x) (λx.x x))`

Exercises

Write **two reduction sequences** for each of the following expressions

- one corresponding to a normal order reduction
- one corresponding to an applicative order reduction

1. $(\lambda x.\, x\, x)\ ((\lambda x\, y.\, y\, x)\ z\ (\lambda x.\, x))$
2. $(\lambda x\, y\, z.\, x\, z)\ (\lambda z.\, z)\ ((\lambda y.\, y)(\lambda z.\, z))\ x$

Comparison of reduction strategies

Theorem

If a normal form exists, normal order reduction will find it!

Applicative order: reduces arguments first

- evaluates every argument exactly once, even if it's not needed
- corresponds to "call by value" parameter passing scheme

Normal order: copies arguments first

- doesn't evaluate unused arguments, but may re-evaluate each one many times
- guaranteed to reduce to normal form, if possible
- corresponds to "call by name" parameter passing scheme

Brief notes on lazy evaluation

Lazy evaluation: reduces arguments only if used, but **at most once**

- essentially, an efficient implementation of normal order reduction
- only evaluates to "weak head normal form"
- corresponds to "call by need" parameter passing scheme

Expression e is in **weak head normal form** if:

- e is a variable or lambda abstraction
- e is an application with a variable in the left position

… in other words, e does not start with a redex

Outline

Church Booleans

Data and operations are encoded as **functions** in the lambda calculus

For Booleans, need lambda calculus terms for *true*, *false*, and *if*, where:

- $\mathit{if}\ \mathit{true}\ e_1\ e_2 \quad \mapsto^* \quad e_1$
- $\mathit{if}\ \mathit{false}\ e_1\ e_2 \quad \mapsto^* \quad e_2$

Church Booleans

$$\begin{aligned} \mathit{true} &= \lambda \mathtt{x}\,\mathtt{y}.\,\mathtt{x} \\ \mathit{false} &= \lambda \mathtt{x}\,\mathtt{y}.\,\mathtt{y} \\ \mathit{if} &= \lambda \mathtt{b}\,\mathtt{t}\,\mathtt{e}.\,\mathtt{b}\,\mathtt{t}\,\mathtt{e} \end{aligned}$$

More Boolean operations

$$\begin{aligned} \mathit{and} &= \lambda \mathtt{p}\,\mathtt{q}.\,\mathit{if}\ \mathtt{p}\,\mathtt{q}\,\mathtt{p} \\ \mathit{or} &= \lambda \mathtt{p}\,\mathtt{q}.\,\mathit{if}\ \mathtt{p}\,\mathtt{p}\,\mathtt{q} \\ \mathit{not} &= \lambda \mathtt{p}.\,\mathit{if}\ \mathtt{p}\ \mathit{false}\ \mathit{true} \end{aligned}$$

Church numerals

A natural number n is encoded as a function that applies **f** to **x** n times

Church numerals

$$
\begin{aligned}
\mathit{zero} &= \lambda \mathtt{f\,x.\,x} \\
\mathit{one} &= \lambda \mathtt{f\,x.\,f\,x} \\
\mathit{two} &= \lambda \mathtt{f\,x.\,f\,(f\,x)} \\
\mathit{three} &= \lambda \mathtt{f\,x.\,f\,(f\,(f\,x))} \\
\ldots & \\
n &= \lambda \mathtt{f\,x.\,f}^{n}\,\mathtt{x}
\end{aligned}
$$

Operations on Church numerals

$$
\begin{aligned}
\mathit{succ} &= \lambda \mathtt{n\,f\,x.\,f\,(n\,f\,x)} \\
\mathit{add} &= \lambda \mathtt{n\,m\,f\,x.\,n\,f\,(m\,f\,x)} \\
\mathit{mult} &= \lambda \mathtt{n\,m\,f.\,n\,(m\,f)} \\
\mathit{isZero} &= \lambda \mathtt{n.\,n}\,(\lambda \mathtt{x}.\,\mathit{false})\,\mathit{true}
\end{aligned}
$$

Encoding values of more complicated data types

At a minimum, need **functions** that encode how to:

- **construct** new values of the data type — data constructors
- **destruct and use** values of the data type in a general way — pattern matching

Can encode values of many data types as **sums** of **products**

- corresponds to **Either** and **tuples** in Haskell

```
data Val = A Nat | B Bool | C Nat Bool
                    ≡
type Val' = Either Nat (Either Bool (Nat,Bool))
```

Exercise

```
data Val = A Nat | B Bool | C Nat Bool
                    ≡
type Val' = Either Nat (Either Bool (Nat,Bool))
```

Encode the following values of type **`Val`** as values of type **`Val'`**

- **`A 2`**
- **`B True`**
- **`C 3 False`**

Products (a.k.a. tuples)

A tuple is defined by:

- a tupling function (constructor)
- a set of selecting functions (destructors)

Church pairs

$$\begin{aligned} pair &= \lambda \mathtt{x\,y\,s.\,s\,x\,y} \\ fst &= \lambda \mathtt{t.\,t\,(}\lambda \mathtt{x\,y.\,x)} \\ snd &= \lambda \mathtt{t.\,t\,(}\lambda \mathtt{x\,y.\,y)} \end{aligned}$$

Church triples

$$\begin{aligned} tuple_3 &= \lambda \mathtt{x\,y\,z\,s.\,s\,x\,y\,z} \\ sel_{1/3} &= \lambda \mathtt{t.\,t\,(}\lambda \mathtt{x\,y\,z.\,x)} \\ sel_{2/3} &= \lambda \mathtt{t.\,t\,(}\lambda \mathtt{x\,y\,z.\,y)} \\ sel_{3/3} &= \lambda \mathtt{t.\,t\,(}\lambda \mathtt{x\,y\,z.\,z)} \end{aligned}$$

Sums (a.k.a. tagged unions)

```
either :: (a -> c) -> (b -> c)
          -> Either a b -> c
either f _ (Left x)  = f x
either _ g (Right y) = g y
```

A tagged union is defined by:

- a case function: a tuple of functions (destructor)
- a set of tags that select the correct function and apply it (constructors)

Church either

$$\begin{aligned} \mathit{either} &= \lambda \mathtt{f\,g\,u.\,u\,f\,g} \\ \mathit{in}_L &= \lambda \mathtt{x\,f\,g.\,f\,x} \\ \mathit{in}_R &= \lambda \mathtt{y\,f\,g.\,g\,y} \end{aligned}$$

Church union

$$\begin{aligned} \mathit{case}_3 &= \lambda \mathtt{f\,g\,h\,u.\,u\,f\,g\,h} \\ \mathit{in}_{1/3} &= \lambda \mathtt{x\,f\,g\,h.\,f\,x} \\ \mathit{in}_{2/3} &= \lambda \mathtt{y\,f\,g\,h.\,g\,y} \\ \mathit{in}_{3/3} &= \lambda \mathtt{z\,f\,g\,h.\,h\,z} \end{aligned}$$

Exercise

```
data Val = A Nat | B Bool | C Nat Bool

foo :: Val -> Nat
foo (A n)   = n
foo (B b)   = if b then 0 else 1
foo (C n b) = if b then 0 else n
```

1. Encode the following values of type **`Val`** as lambda calculus terms
   - **`A 2`**
   - **`B True`**
   - **`C 3 False`**

2. Encode the function **`foo`** in lambda calculus

Outline

Naming in lambda calculus

Observation: can use abstractions to define names

```
let succ = \n -> n+1
in ... succ 3 ... succ 7 ...
```

$\Longrightarrow$

```
(λsucc.
 ... succ 3 ... succ 7 ...
) (λn f x. f (n f x))
```

But this pattern doesn't work for **recursive** functions!

```
let fac = \n ->
  ... n * fac (n-1)
in ... fac 5 ... fac 8 ...
```

$\Longrightarrow$

```
(λfac.
... fac 5 ... fac 8 ...
) (λn f x. ... mult n (??? (pred n)))
```

Recursion via fixpoints

Solution: Fixpoint function

$Y = \lambda \mathbf{f}.\,(\lambda \mathbf{x}.\,\mathbf{f}\ (\mathbf{x}\ \mathbf{x}))\ (\lambda \mathbf{x}.\,\mathbf{f}\ (\mathbf{x}\ \mathbf{x}))$

$$
\begin{aligned}
& Y\ \mathbf{g} \\
& \quad \mapsto (\lambda \mathbf{x}.\,\mathbf{g}\ (\mathbf{x}\ \mathbf{x}))\ (\lambda \mathbf{x}.\,\mathbf{g}\ (\mathbf{x}\ \mathbf{x})) \\
& \quad \mapsto \mathbf{g}\ ((\lambda \mathbf{x}.\,\mathbf{g}\ (\mathbf{x}\ \mathbf{x}))\ (\lambda \mathbf{x}.\,\mathbf{g}\ (\mathbf{x}\ \mathbf{x}))) \\
& \quad \mapsto \mathbf{g}\ (\mathbf{g}\ ((\lambda \mathbf{x}.\,\mathbf{g}\ (\mathbf{x}\ \mathbf{x}))\ (\lambda \mathbf{x}.\,\mathbf{g}\ (\mathbf{x}\ \mathbf{x})))) \\
& \quad \mapsto \mathbf{g}\ (\mathbf{g}\ (\mathbf{g}\ ((\lambda \mathbf{x}.\,\mathbf{g}\ (\mathbf{x}\ \mathbf{x}))\ (\lambda \mathbf{x}.\,\mathbf{g}\ (\mathbf{x}\ \mathbf{x}))))) \\
& \quad \mapsto \ldots
\end{aligned}
$$

Example recursive function (factorial)

$Y\ (\lambda \mathbf{fac}\ \mathbf{n}.\,\mathit{if}\ (\mathit{isZero}\ \mathbf{n})\ \mathit{one}\ (\mathit{mult}\ \mathbf{n}\ (\mathbf{fac}\ (\mathit{pred}\ \mathbf{n}))))$

Outline

The role of names in lambda calculus

Variable names are a convenience for readability (mnemonics)
…but they're annoying in implementations and proofs

Annoyances related to names

- safe substitution is complicated, requires generating fresh names
- checking and maintaining α-equivalence is complicated and expensive

Recall: α-equivalence

Expressions are the same up to variable renaming

- $\lambda x.\, x \equiv \lambda y.\, y \equiv \lambda z.\, z$
- $\lambda x\, y.\, x \equiv \lambda y\, x.\, y$

A nameless representation of lambda calculus

Basic idea: de Bruijn indices

- an abstraction implicitly declares its input (no variable name)
- a variable reference is a number n, called a **de Bruijn index**, that refers to the nth abstraction up the AST

Nameless lambda calculus

$$
\begin{array}{rcll}
n \in \mathit{Nat} & ::= & \text{(any natural number)} & \\
e \in \mathit{Exp} & ::= & e\ e & \textit{application} \\
& | & \lambda\, e & \textit{lambda abstraction} \\
& | & n & \textit{de Bruijn index}
\end{array}
$$

Named $\rightsquigarrow$ nameless

- λ`x.x` $\rightsquigarrow$ λ `0`
- λ`xy.x` $\rightsquigarrow$ $\lambda\lambda$ `1`
- λ`xy.y` $\rightsquigarrow$ $\lambda\lambda$ `0`
- λ`x.(`λ`y.y) x` $\rightsquigarrow$ λ `(`λ `0) 0`

Main advantage: α-equivalence is just syntactic equality!

Deciphering de Bruijn indices

De Bruijn index: the number of λs you have to *skip* when moving up the AST

$$\lambda\ 0\ (\lambda\ 1\ (\lambda\ 0\ 1\ 2)\ 0)$$

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

Gotchas:

- the same variable will be a different number in different contexts
- scopes work the same as before; references respect the AST
  - e.g. the blue `0` refers to the blue λ since it is not in scope of the green λ, and the green λ does not count as a *skip*

Free variables in nameless encoding

Free variable in e: a de Bruijn index that skips over all of the λs in e

- the same free variables will have the same number of λs left to skip

`λ 1 (λ 2 3 0 1) 2 0`

`λx. w (λy. w v y x) v x`