λ-calculus

CS 4301/6371: Advanced Programming Languages

Kevin W. Hamlen

March 19, 2024

Historical Roots

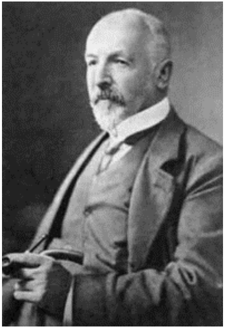
First, some mathematical history...

Deductive Logic

- Euclid's *The Elements*
  - written c. 300 B.C.
  - deductive reasoning: 23 definitions, 10 axioms
  - geometry, algebra, number theory
  - foundation of western mathematics for about 2000 years
- Problem: Some theorems unprovable from axioms
  - Example: Two circles with centers closer than the sum of their radii have an intersection point.

Set Theory

- First proposed by Georg Cantor in 1874
  - new foundation for mathematics
  - early versions contained paradoxes
    - Russel's Paradox: the set of all sets that do not contain themselves
- Deductive Set Theory
  - axiomized by Zermelo and Fraenkel between 1908 and 1930
  - Zermelo-Fraenkel set theory with axiom of choice (ZFC)
- Problem: some theorems still unprovable!
  - Example (Continuum Hypothesis): There is no set larger than $\mathbb{N}$ but smaller than $\mathbb{R}$.

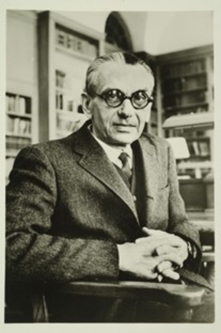


Hilbert's Program

- Proposed by David Hilbert in 1921
- Goals:
  - Provide an unassailable foundation for all mathematics
  - Find a set of axioms and rules of logical inference sufficient to deductively prove all mathematical theorems.
- Required properties:
  - **Soundness:** no untrue statement provable
  - **Completeness:** all true statements provable
  - **Decidability:** procedure for determining whether any mathematical statement is true or false

Gödel's Incompleteness Theorem

- Proved by Kurt Gödel in 1931
- Theorem: No finite collection of axioms is both sound and complete(!)
- Ramifications:
    - Given any sound axiomization of mathematics, there are true statements that are unprovable.
    - There exists no decision algorithm for mathematical truth.
- Essentially destroyed Hilbert's program
- Raised another question: What is decidable?



Theory of Computation

Alan Turing

Alonzo Church

- "Decide" = "Compute"
- 1936: Two models of "computation" proposed:
  - Turing Machines (Alan Turing)
  - λ-calculus (Alonzo Church)
- Both models equivalent in power
- Church-Turing Thesis: All (reasonable) models of computation are equally powerful.
- Birth of Computer Science
  - Turing Machines = imperative programming
  - λ-calculus = functional programming

Fun Fact: My Mathematical Ancestry

Alonzo Church
(PhD Princeton 1927)
|
Stephen Kleene
(PhD Princeton 1934)
|
Bob Constable
(PhD Wisconsin-Madison 1968)
|
Bob Harper
(PhD Cornell 1985)
|
Greg Morrisett
(PhD Carnegie Mellon 1995)
|
Kevin Hamlen
(PhD Cornell 2006)

Today

Today: λ-calculus

Syntax

$$e ::= v \mid \lambda v.e \mid e_1 e_2$$

Only three syntaxes:

- variables v
- abstractions $\lambda v.e$ (functions)
- applications $e_1 e_2$

Some simple examples:

- $\lambda x.x$ (the identity function)
- $(\lambda x.x)(\lambda y.yy) \rightarrow_1 \lambda y.yy$
- $\lambda x.((\lambda y.y)x)$ does not reduce (already a value)

Free Variables

Legal λ-expressions must be closed (no free variables), where we define the set of free variables $FV(e)$ by

$$\begin{aligned} FV(v) &= \{v\} \\ FV(\lambda v.e) &= FV(e) \backslash \{v\} \\ FV(e_1 e_2) &= FV(e_1) \cup FV(e_2) \end{aligned}$$

We require $FV(e) = \emptyset$.

Semantics

Small-step semantics of λ-calculus:

$$\frac{e_1 \to_1 e_1'}{e_1 e_2 \to_1 e_1' e_2} \qquad \frac{}{(\lambda v.e_1)e_2 \to_1 e_1[e_2/v]}(\beta\text{-reduction})$$

where notation $e_1[e/v]$ denotes **capture-avoiding substitution**:

$$\begin{aligned}
v[e/v] &= e \\
v_1[e/v_2] &= v_1 \text{ when } v_1 \neq v_2 \text{ (i.e., different variables)} \\
(\lambda v.e_1)[e/v] &= \lambda v.e_1 \\
(\lambda v_1.e_1)[e/v_2] &= \lambda v_1.(e_1[e/v_2]) \text{ when } v_1 \neq v_2 \text{ (i.e. different variables)} \\
(e_1 e_2)[e/v] &= (e_1[e/v])(e_2[e/v])
\end{aligned}$$

Intuition: $e_1[e_2/x]$ means replace only the **free** x's in e_1 with e_2.

Optional exercise: Devise equivalent large-step and denotational semantics.

Reduction example

$$\big((\lambda x.(\lambda y.(xy)))(\lambda y.y)\big)(\lambda z.z) \rightarrow_1 ?$$

Reduction example

$$\big((\lambda x.(\lambda y.(xy)))(\lambda y.y)\big)(\lambda z.z) \rightarrow_1 ?$$

Reduction example

$$\big((\lambda x.(\lambda y.(xy)))(\lambda y.y)\big)(\lambda z.z) \to_1$$
$$\big(\lambda y.((\lambda y.y)y)\big)(\lambda z.z) \to_1 ?$$

Reduction example

$$
\begin{aligned}
\big((\lambda x.(\lambda y.(xy)))(\lambda y.y)\big)(\lambda z.z) &\to_1 \\
\big(\cancel{\lambda y.}((\lambda y.y)y)\big)(\lambda z.z) &\to_1 \\
(\lambda y.y)(\lambda z.z) &\to_1 ?
\end{aligned}
$$

Reduction example

$$
\begin{aligned}
\big((\lambda x.(\lambda y.(xy)))(\lambda y.y)\big)(\lambda z.z) &\rightarrow_1 \\
\big(\lambda y.((\lambda y.y)y)\big)(\lambda z.z) &\rightarrow_1 \\
(\cancel{\lambda y.}\,y)(\lambda z.z) &\rightarrow_1 \\
(\lambda z.z)
\end{aligned}
$$

Reduction example

$$
\begin{aligned}
\big((\lambda x.(\lambda y.(xy)))(\lambda y.y)\big)(\lambda z.z) &\rightarrow_1 \\
\big(\lambda y.((\lambda y.y)y)\big)(\lambda z.z) &\rightarrow_1 \\
(\lambda y.y)(\lambda z.z) &\rightarrow_1 \\
(\lambda z.z)
\end{aligned}
$$

Important observations:

- Don't change any variable names as you evaluate!
- There are no stores involved here!
- Semantics of λ-calculus are based on capture-avoiding substitution, not stores or variable renaming.
- Function bodies never evaluate (even if they could) until their λ-binder gets stripped off (at which point they're not functions anymore).

Strategy: Pretend that "$\lambda v.e$" is OCaml "`fun` $v \rightarrow e$".

Precedence and Associativity

Precedence and associativity conventions:

$$\lambda v.e_1 e_2 = \lambda v.(e_1 e_2) \qquad \text{(application binds tighter than abstraction)}$$

$$e_1 e_2 e_2 = (e_1 e_2) e_3 \qquad \text{(application associates left)}$$

Parenthesize anything else that might be ambiguous.

Encodings and Reductions

Amazing fact: This extremely simple language is Turing-complete (can perform any computation implementable by modern computers)!

Proof by reduction (recall from computability theory): Let's reduce a (simple) Turing-complete programming language to λ-calculus.

Higher-arity Functions

λ-calculus only gives us 1-argument functions $\lambda v.e$.

Q: How could I create a multi-argument function?

Higher-arity Functions

λ-calculus only gives us 1-argument functions $\lambda v.e$.

Q: How could I create a multi-argument function?
A: Nest the λ's: $\lambda x.\lambda y.\lambda z.(\ldots)$

Definition (currying): In functional programming, changing a function on tuple-arguments to use distinct (non-tuple) arguments is called *currying* the function.

Example:
Uncurried: `let add (x,y) = x+y;;`
Curried: `let add x y = x+y;;`

Benefits: More opportunities for code-reuse through partial evaluation, and more opportunities for compiler optimization through specialization

Booleans

How might we encode boolean expressions as λ-terms? Let's start with constants and the ternary operator:

$$
\begin{aligned}
\texttt{true} &= ? \\
\texttt{false} &= ? \\
e_1 \mathbin{?} e_2 : e_3 &= ?
\end{aligned}
$$

Booleans

How might we encode boolean expressions as λ-terms? Let's start with constants and the ternary operator:

$$
\begin{aligned}
\texttt{true} &= (\lambda x.\lambda y.x) \\
\texttt{false} &= (\lambda x.\lambda y.y) \\
e_1 \mathbin{?} e_2 : e_3 &= ((e_1)(e_2)(e_3))
\end{aligned}
$$

Booleans

How might we encode boolean expressions as λ-terms? Let's start with constants and the ternary operator:

$$\begin{aligned}\texttt{true} &= (\lambda x.\lambda y.x)\\ \texttt{false} &= (\lambda x.\lambda y.y)\\ e_1 \mathbin{?} e_2 : e_3 &= ((e_1)(e_2)(e_3))\end{aligned}$$

Using the above, how might we encode `not`, `and`, and `or` as functions over booleans?

$$\begin{aligned}\texttt{not} &= ?\\ \texttt{and} &= ?\\ \texttt{or} &= ?\end{aligned}$$

Booleans

How might we encode boolean expressions as λ-terms? Let's start with constants and the ternary operator:

$$\begin{aligned} \texttt{true} &= (\lambda x.\lambda y.x) \\ \texttt{false} &= (\lambda x.\lambda y.y) \\ e_1 \mathrel{?} e_2 : e_3 &= ((e_1)(e_2)(e_3)) \end{aligned}$$

Using the above, how might we encode `not`, `and`, and `or` as functions over booleans?

$$\begin{aligned} \texttt{not} &= (\lambda b.(b \mathrel{?} \texttt{false} : \texttt{true})) \\ \texttt{and} &= ? \\ \texttt{or} &= ? \end{aligned}$$

Booleans

How might we encode boolean expressions as λ-terms? Let's start with constants and the ternary operator:

$$\begin{aligned} \texttt{true} &= (\lambda x.\lambda y.x) \\ \texttt{false} &= (\lambda x.\lambda y.y) \\ e_1 \,?\, e_2 : e_3 &= ((e_1)(e_2)(e_3)) \end{aligned}$$

Using the above, how might we encode `not`, `and`, and `or` as functions over booleans?

$$\begin{aligned} \texttt{not} &= (\lambda b.(b \,?\, \texttt{false} : \texttt{true})) \\ \texttt{and} &= (\lambda b_1.\lambda b_2.(b_1 \,?\, b_2 : \texttt{false})) \\ \texttt{or} &= \,? \end{aligned}$$

Booleans

How might we encode boolean expressions as λ-terms? Let's start with constants and the ternary operator:

$$\begin{aligned} \texttt{true} &= (\lambda x.\lambda y.x) \\ \texttt{false} &= (\lambda x.\lambda y.y) \\ e_1 \mathbin{?} e_2 : e_3 &= ((e_1)(e_2)(e_3)) \end{aligned}$$

Using the above, how might we encode `not`, `and`, and `or` as functions over booleans?

$$\begin{aligned} \texttt{not} &= (\lambda b.(b \mathbin{?} \texttt{false} : \texttt{true})) \\ \texttt{and} &= (\lambda b_1.\lambda b_2.(b_1 \mathbin{?} b_2 : \texttt{false})) \\ \texttt{or} &= (\lambda b_1.\lambda b_2.(b_1 \mathbin{?} \texttt{true} : b_2)) \end{aligned}$$

Tuples

How might we encode pairs?

- The `pair` function should take two arguments (could be anything) and package them together into some kind of object.
- The π_1 function (`fst` in OCaml) should accept a pair as input and recover (project out) the first element.
- The π_2 function (`snd` in OCaml) should analogously project out the second element.

$$\begin{aligned} \texttt{pair} &= (\lambda x.\lambda y.\,?) \\ \pi_1 &= (\lambda p.\,?) \\ \pi_2 &= (\lambda p.\,?) \end{aligned}$$

Tuples

How might we encode pairs?

- The `pair` function should take two arguments (could be anything) and package them together into some kind of object.
- The π_1 function (`fst` in OCaml) should accept a pair as input and recover (project out) the first element.
- The π_2 function (`snd` in OCaml) should analogously project out the second element.

$$\begin{aligned} \texttt{pair} &= (\lambda x.\lambda y.\lambda b.(b\,?\,x : y)) \\ \pi_1 &= (\lambda p\,.\,p\,\texttt{true}) \\ \pi_2 &= (\lambda p\,.\,p\,\texttt{false}) \end{aligned}$$

Natural Numbers

How might we encode natural numbers?

- Each number $0_{\mathbb{N}}, 1_{\mathbb{N}}, 2_{\mathbb{N}}, \ldots$ should be encoded as a λ-calculus **value** (must not reduce to something else).
- Approach: Encode $0_{\mathbb{N}}$, then code up a successor function $\texttt{succ}_{\mathbb{N}}$.
- Should also have predecessor $\texttt{pred}_{\mathbb{N}}$ (don't care what it returns for $0_{\mathbb{N}}$)
- Also need a test $\texttt{iszero}_{\mathbb{N}}$ (returns a boolean).

$$\begin{aligned}
0_{\mathbb{N}} &= ? \\
\texttt{succ}_{\mathbb{N}} &= (\lambda n \,.\, ?) \\
\texttt{pred}_{\mathbb{N}} &= (\lambda n \,.\, ?) \\
\texttt{iszero}_{\mathbb{N}} &= (\lambda n \,.\, ?)
\end{aligned}$$

Natural Numbers

How might we encode natural numbers?

- Each number $0_{\mathbb{N}}, 1_{\mathbb{N}}, 2_{\mathbb{N}}, \ldots$ should be encoded as a λ-calculus **value** (must not reduce to something else).
- Approach: Encode $0_{\mathbb{N}}$, then code up a successor function $\texttt{succ}_{\mathbb{N}}$.
- Should also have predecessor $\texttt{pred}_{\mathbb{N}}$ (don't care what it returns for $0_{\mathbb{N}}$)
- Also need a test $\texttt{iszero}_{\mathbb{N}}$ (returns a boolean).

$$
\begin{aligned}
0_{\mathbb{N}} &= (\lambda x.x) \\
\texttt{succ}_{\mathbb{N}} &= (\lambda n\,.\,\texttt{pair}\ (?)\ n) \\
\texttt{pred}_{\mathbb{N}} &= (\lambda n\,.\,?) \\
\texttt{iszero}_{\mathbb{N}} &= (\lambda n\,.\,?)
\end{aligned}
$$

Natural Numbers

How might we encode natural numbers?

- Each number $0_{\mathbb{N}}, 1_{\mathbb{N}}, 2_{\mathbb{N}}, \ldots$ should be encoded as a λ-calculus **value** (must not reduce to something else).
- Approach: Encode $0_{\mathbb{N}}$, then code up a successor function $\mathtt{succ}_{\mathbb{N}}$.
- Should also have predecessor $\mathtt{pred}_{\mathbb{N}}$ (don't care what it returns for $0_{\mathbb{N}}$)
- Also need a test $\mathtt{iszero}_{\mathbb{N}}$ (returns a boolean).

$$
\begin{aligned}
0_{\mathbb{N}} &= (\lambda x.x) \\
\mathtt{succ}_{\mathbb{N}} &= (\lambda n\,.\,\mathtt{pair}\ (?)\ n) \\
\mathtt{pred}_{\mathbb{N}} &= \pi_2 \\
\mathtt{iszero}_{\mathbb{N}} &= (\lambda n\,.\,?)
\end{aligned}
$$

Natural Numbers

How might we encode natural numbers?

- Each number $0_{\mathbb{N}}, 1_{\mathbb{N}}, 2_{\mathbb{N}}, \ldots$ should be encoded as a λ-calculus **value** (must not reduce to something else).
- Approach: Encode $0_{\mathbb{N}}$, then code up a successor function $\texttt{succ}_{\mathbb{N}}$.
- Should also have predecessor $\texttt{pred}_{\mathbb{N}}$ (don't care what it returns for $0_{\mathbb{N}}$)
- Also need a test $\texttt{iszero}_{\mathbb{N}}$ (returns a boolean).

$$
\begin{aligned}
0_{\mathbb{N}} &= (\lambda x.x) \\
\texttt{succ}_{\mathbb{N}} &= (\lambda n \,.\, \texttt{pair false } n) \\
\texttt{pred}_{\mathbb{N}} &= \pi_2 \\
\texttt{iszero}_{\mathbb{N}} &= (\lambda n \,.\, ?)
\end{aligned}
$$

Natural Numbers

How might we encode natural numbers?

- Each number $0_{\mathbb{N}}, 1_{\mathbb{N}}, 2_{\mathbb{N}}, \ldots$ should be encoded as a λ-calculus **value** (must not reduce to something else).
- Approach: Encode $0_{\mathbb{N}}$, then code up a successor function $\texttt{succ}_{\mathbb{N}}$.
- Should also have predecessor $\texttt{pred}_{\mathbb{N}}$ (don't care what it returns for $0_{\mathbb{N}}$)
- Also need a test $\texttt{iszero}_{\mathbb{N}}$ (returns a boolean).

$$\begin{aligned} 0_{\mathbb{N}} &= (\lambda x.x) \\ \texttt{succ}_{\mathbb{N}} &= (\lambda n\,.\,\texttt{pair false}\ n) \\ \texttt{pred}_{\mathbb{N}} &= \pi_2 \\ \texttt{iszero}_{\mathbb{N}} &= \pi_1 \end{aligned}$$

Natural Numbers

$$\begin{aligned} 0_{\mathbb{N}} &= (\lambda x.x) \\ \texttt{succ}_{\mathbb{N}} &= (\lambda n\,.\,\texttt{pair false}\ n) \\ \texttt{pred}_{\mathbb{N}} &= \pi_2 \\ \texttt{iszero}_{\mathbb{N}} &= \pi_1 \end{aligned}$$

Does $\texttt{iszero}_{\mathbb{N}}(0_{\mathbb{N}})$ really work (should return `true`)?

$0_{\mathbb{N}} = (\lambda x.x)$ is not even a pair!

Natural Numbers

$$
\begin{aligned}
0_{\mathbb{N}} &= (\lambda x.x) \\
\texttt{succ}_{\mathbb{N}} &= (\lambda n\,.\,\texttt{pair false}\ n) \\
\texttt{pred}_{\mathbb{N}} &= \pi_2 \\
\texttt{iszero}_{\mathbb{N}} &= \pi_1
\end{aligned}
$$

Does $\texttt{iszero}_{\mathbb{N}}(0_{\mathbb{N}})$ really work (should return `true`)?

$$
\begin{aligned}
&\texttt{iszero}_{\mathbb{N}}(0_{\mathbb{N}}) = \pi_1(\lambda x.x) = (\lambda p\,.\,p\ \texttt{true})(\lambda x.x) \\
&\rightarrow_1 (\lambda x.x)\texttt{true} \\
&\rightarrow_1 \texttt{true}
\end{aligned}
$$

It worked!*

*Warning: On the homework, I'll ask you to first fully expand all the encodings into pure λ-terms before doing any evaluation steps. I did it without expanding `true` here to illustrate a point, but technically I should have first expanded `true` into a λ-term before applying the small-step semantics of λ-calculus to a term containing it.

Untypedness

Take-aways:

- λ-calculus is an **untyped** language.
  - Every syntactically legal, closed term evaluates to something.
  - Can do some very weird things (as we will see...)!
- There is a different language (which we will learn) called **typed** λ-calculus.
  - Don't confuse it with this language!
  - Watch out for web resources that look similar but that concern a different λ-calculus (there are many)!

Loops

We're close to a full Turing-complete language now, but one major thing is missing: loops.

Q: Is it possible to code an infinite loop in λ-calculus?

Loops

We're close to a full Turing-complete language now, but one major thing is missing: loops.

Q: Is it possible to code an infinite loop in λ-calculus?
A: Yes. Smallest example: $(\lambda x.xx)(\lambda x.xx)$

Recursion

What about useful loops?
Case-study: Can we code an addition function for natural numbers?

$$\mathtt{add}_{\mathbb{N}} = \lambda m.\lambda n.?$$

Recursion

What about useful loops?
Case-study: Can we code an addition function for natural numbers?

$$\mathtt{add}_{\mathbb{N}} = \lambda m.\lambda n.\big(\mathtt{iszero}_{\mathbb{N}}\, m \,?\, n : \mathtt{add}_{\mathbb{N}}(\mathtt{pred}_{\mathbb{N}}\, m)(\mathtt{succ}_{\mathbb{N}}\, n)\big)$$

Recursion

What about useful loops?
Case-study: Can we code an addition function for natural numbers?

$$\texttt{add}_{\mathbb{N}} = \lambda m.\lambda n.\big(\texttt{iszero}_{\mathbb{N}}\, m \,?\, n : \texttt{add}_{\mathbb{N}}(\texttt{pred}_{\mathbb{N}}\, m)(\texttt{succ}_{\mathbb{N}}\, n)\big)$$

Circular definition! Remember, the encoding part ($=$) is supposed to be a definition; it's not part of the λ-term.

How can we remove the recursion from this formula?

Fixed points

$$\texttt{add}_{\mathbb{N}} = \lambda m.\lambda n.\big(\texttt{iszero}_{\mathbb{N}}\, m \;?\; n : \texttt{add}_{\mathbb{N}}(\texttt{pred}_{\mathbb{N}}\, m)(\texttt{succ}_{\mathbb{N}}\, n)\big)$$

Define a functional whose least fixed point is $\texttt{add}_{\mathbb{N}}$:

$$\texttt{Add}_{\mathbb{N}} = \lambda f.\lambda m.\lambda n.\big(\texttt{iszero}_{\mathbb{N}}\, m \;?\; n : f(\texttt{pred}_{\mathbb{N}}\, m)(\texttt{succ}_{\mathbb{N}}\, n)\big)$$

Then define $\texttt{add}_{\mathbb{N}}$ to be its least fixed point:

$$\texttt{add}_{\mathbb{N}} = \mathit{fix}(\texttt{Add}_{\mathbb{N}})$$

But fix is not part of λ-calculus, so we're still stuck...?

Y-combinator

A very interesting function (discovered by Haskell Curry):

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Amazing claim: $Y = \mathit{fix}$

Proof: Let's evaluate it...

$$Y\ g \rightarrow_1 ?$$

Y-combinator

A very interesting function (discovered by Haskell Curry):

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Amazing claim: $Y = \mathit{fix}$

Proof: Let's evaluate it...

$$\begin{aligned} Y\ g &\rightarrow_1 (\lambda x.g(xx))(\lambda x.g(xx)) \\ &\rightarrow_1 \text{?} \end{aligned}$$

Y-combinator

A very interesting function (discovered by Haskell Curry):

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Amazing claim: $Y = \mathit{fix}$

Proof: Let's evaluate it...

$$\begin{aligned} Y\, g &\rightarrow_1 (\lambda x.g(xx))(\lambda x.g(xx)) \\ &\rightarrow_1 g\big((\lambda x.g(xx))(\lambda x.g(xx))\big) \end{aligned}$$

Y-combinator

A very interesting function (discovered by Haskell Curry):

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Amazing claim: $Y = \mathit{fix}$

Proof: Let's evaluate it...

$$\begin{aligned} Y\ g &\to_1 (\lambda x.g(xx))(\lambda x.g(xx)) \\ &\to_1 g\big((\lambda x.g(xx))(\lambda x.g(xx))\big) = g(Y\ g) \end{aligned}$$

Conclusion: $Y\ g$ is the least fixed point of g. (Whoa!)

Solving Recursion Problems with Y

Exercise: Define an addition function in λ-calculus.

The following definition is illegal (not well-founded):

$$\texttt{add}_{\mathbb{N}} = \lambda m.\lambda n.\big(\texttt{iszero}_{\mathbb{N}} m \,?\, n : \texttt{add}_{\mathbb{N}}(\texttt{pred}_{\mathbb{N}} m)(\texttt{succ}_{\mathbb{N}} n)\big)$$

So instead define a functional whose least fixed point is $\texttt{add}_{\mathbb{N}}$:

$$\lambda f.\lambda m.\lambda n.\big(\texttt{iszero}_{\mathbb{N}} m \,?\, n : f(\texttt{pred}_{\mathbb{N}} m)(\texttt{succ}_{\mathbb{N}} n)\big)$$

Then apply Y to it:

$$\texttt{add}_{\mathbb{N}} = Y\big(\lambda f.\lambda m.\lambda n.(\texttt{iszero}_{\mathbb{N}} m \,?\, n : f(\texttt{pred}_{\mathbb{N}} m)(\texttt{succ}_{\mathbb{N}} n))\big)$$

Now we have a legal definition of an addition function with no explicit recursions in it.

Exercise: Multiplication

Exercise: Define a multiplication function for natural numbers in λ-calculus.

Try to define it recursively first:

$$\mathtt{mul}_{\mathbb{N}} = \lambda m.\lambda n.$$

Exercise: Multiplication

Exercise: Define a multiplication function for natural numbers in λ-calculus.

Try to define it recursively first:

$$\texttt{mul}_{\mathbb{N}} = \lambda m.\lambda n.\big(\texttt{iszero}_{\mathbb{N}}\, m \;?\; 0_{\mathbb{N}} : \texttt{add}_{\mathbb{N}}(\texttt{mul}_{\mathbb{N}}(\texttt{pred}_{\mathbb{N}} m)n)n\big)$$

Exercise: Multiplication

Exercise: Define a multiplication function for natural numbers in λ-calculus.

Try to define it recursively first:

$$\texttt{mul}_{\mathbb{N}} = \lambda m.\lambda n.\big(\texttt{iszero}_{\mathbb{N}}\, m\,?\,0_{\mathbb{N}} : \texttt{add}_{\mathbb{N}}(\texttt{mul}_{\mathbb{N}}(\texttt{pred}_{\mathbb{N}} m)n)n\big)$$

Then change it to a non-recursive functional and apply Y to it:

$$\texttt{mul}_{\mathbb{N}} = Y\big(\lambda f.\lambda m.\lambda n.(\texttt{iszero}_{\mathbb{N}}\, m\,?\,0_{\mathbb{N}} : \texttt{add}_{\mathbb{N}}(f(\texttt{pred}_{\mathbb{N}} m)n)n)\big)$$

Readability

When solving these sorts of problems on homeworks, quizzes, and exams:

- Please DO use the abbreviations in your code.
  - Don't write $(\lambda x.\lambda y.x)$ when you mean `true`.
  - Strive for readability (otherwise becomes very hard to grade!).
- Please DO define named helper functions.
  - Less writing is good; don't repeatedly write out same subroutine.
  - But any recursions must always be eliminated with Y.
  - Use informative names (not f).
- Don't name variables the same as any helper functions (really confusing!).
- λ-calculus is a math formalism not a modern language, so extra effort is required to make it readable.

Equality

λ-terms are ASTs. They are only "equal" ($=$) if they are identical after expansion of all macro abbreviations.

(Also recall that the parentheses are not symbols in the AST; they just show the structure of the AST.)

Examples:

$$(\lambda y.y)(\lambda x.x) \neq \lambda x.x \qquad \text{(though they evaluate to the same terms)}$$
$$(\lambda x.(x)) = \lambda x.x$$
$$\lambda x.x \neq \lambda y.y$$

However, there are some notions of term **equivalence** that are important to understand.

α-equivalence

Definition (α-equivalence): Term $\lambda x.e$ is α-equivalent to term $\lambda y.(e'[y/x])$ (written $\lambda x.e \equiv_\alpha \lambda y.(e'[y/x])$) whenever $e \equiv_\alpha e'$ (recursively).

Intuition: Terms that are identical except for consistent, capture-avoiding renaming of the variables are α-equivalent.

Examples:

$$\lambda x.x \equiv_\alpha \lambda y.y$$
$$\lambda x.\lambda x.x \equiv_\alpha \lambda y.\lambda x.x$$
$$\lambda x.\lambda x.x \not\equiv_\alpha \lambda y.\lambda x.y$$

Colloquially: Functional programmers refer to renaming their variables as "α-conversion".

β-equivalence

Definition (β-equivalence): Terms $(\lambda v.e_1)e_2$ and $e_1[e_2/x]$ are β-equivalent (written $(\lambda v.e_1)e_2 \equiv_\beta e_1[e_2/x]$).

Intuition: An application of a function f to an argument a is β-equivalent to a term consisting of the body of f with all its parameters replaced with the argument term a.

Examples:

$$
\begin{aligned}
(\lambda x.xx)(\lambda y.y) &\equiv_\beta (\lambda y.y)(\lambda y.y) \\
(\lambda x.xx)(\lambda y.y) &\equiv_\beta \lambda y.y && \text{(by transitivity)} \\
((\lambda x.xx)(\lambda y.y))(\lambda z.z) &\not\equiv_\beta ((\lambda y.y)(\lambda y.y))(\lambda z.z)
\end{aligned}
$$

The last example is because that reduction doesn't only use the β-rule. In that case the left subterms are β-equivalent, but not the full-sized terms that contain them.

η-equivalence

Definition (η-equivalence): Terms $\lambda v.(fv)$ and f are η-equivalent (written $\lambda v.(fv) \equiv_\eta f$) if $v \notin FV(f)$.

Intuition: A "wrapper function" that merely applies some other function f to whatever argument it receives is equivalent to just f.

Example:

$$\lambda n\,.\,\texttt{pair false}\ n \equiv_\eta \texttt{pair false}$$

Example from OCaml:

```
let sum x = List.fold_left (+) 0 x;;
```

$$\equiv_\eta$$

```
let sum = List.fold_left (+) 0;;
```

Equivalence vs. Operational and Denotational Semantics

Don't confuse equivalence with the operational semantics of λ-calculus:

- Only β-equivalence is a rule of the operational semantics.
  - α-equivalent terms don't always evaluate to the same final terms (variables might be different, which makes them different ASTs).
  - β-equivalent terms do always evaluate to the same terms.
  - η-equivalent terms "behave the same" when applied, but η-equivalence is not a reduction step of λ-calculus.
- There is no $=$ or $\equiv$ test operation in λ-calculus!
  - The following is NOT a legal λ-term:

$$\lambda x.\lambda y.(x = y)\;?\;\texttt{true} : \texttt{false}$$

  - It is impossible to code up such an operation (exercise: prove it!).
- In denotational semantics, λ-terms denote (mathematical) functions.
  - In math we have another definition of functional equivalence (identical input-output relations).
  - But functional equivalence is not decidable (Rice's Theorem).
  - And equivalence of λ-term *denotations* is NOT the same as equivalence of the terms themselves.