A Cunning Plan

The Lambda Calculus, Lisp, Scheme, and R

Thomas P. Harte

R/Finance 2024-05-18
University of Illinois at Chicago

Outline

Disclaimer

Motivation

Lambda Calculus: Theory

Lambda Calculus: Implementation

References

Disclaimer

Disclaimer

Thomas P. Harte ("the Author") is providing this presentation and its contents ("the Content") for educational purposes only at the *R in Finance Conference*, 2024-05-18, Chicago, IL. The Author is not a registered investment advisor, nor does the Author purport to offer investment advice, nor business advice. The opinions expressed in the Content belong solely to the Author, and do not necessarily represent the opinions of the Author's employers, nor any organization, committee or other group with which the Author is affiliated.

THE AUTHOR SPECIFICALLY DISCLAIMS ANY PERSONAL LIABILITY, LOSS OR RISK INCURRED AS A CONSEQUENCE OF THE USE AND APPLICATION, EITHER DIRECTLY OR INDIRECTLY, OF THE CONTENT. THE AUTHOR SPECIFICALLY DISCLAIMS ANY REPRESENTATION, WHETHER EXPLICIT OR IMPLIED, THAT APPLYING THE CONTENT WILL LEAD TO SIMILAR RESULTS IN A BUSINESS SETTING. THE RESULTS PRESENTED IN THE CONTENT ARE NOT NECESSARILY TYPICAL AND SHOULD NOT DETERMINE EXPECTATIONS OF FINANCIAL OR BUSINESS RESULTS.

Motivation

What is this?

```
Y(\(r) \(n)
    `zerop?`(n)(C0)
    (`IF`(eq(n)(C1))
        (C1)
        (\(x) add (r( sub(n)(C1) )) (r( sub(n)(C2) )) (x))))
```

It’s R!

```r
Y(\(r) \(n)
    `zerop?`(n)(C0)
    (`IF`(eq(n)(C1))
        (C1)
        (\(x) add (r( sub(n)(C1) )) (r( sub(n)(C2) )) (x)))) (C10) |> .to.integer ()
```

```
[1] 55
```

Fibonacci: R

```
Y(\(r) \(n)
    `zerop?`(n)(C0)
    (`IF`(eq(n)(C1))
        (C1)
        (\(x) add (r( sub(n)(C1) )) (r( sub(n)(C2) )) (x)))) -> fibonacci

lapply(list(C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10),
       \(n) cat(sprintf("%+2s: %+3s\n", n |> .to.integer(), fibonacci(n) |> .to.integer())))
           -> dev.null
```

```
 0:   0
 1:   1
 2:   1
 3:   2
 4:   3
 5:   5
 6:   8
 7:  13
 8:  21
 9:  34
10:  55
```

Fibonacci: Lisp

```lisp
(defun fibonacci (n)
    (if (or (= n 0) (= n 1)) n
        (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))

(let ((x '(0 1 2 3 4 5 6 7 8 9 10)))
(mapcar (lambda (x) (princ (format "%2s:%+3s\n" x (fibonacci x)))) x))
```

```
 0:  0
 1:  1
 2:  1
 3:  2
 4:  3
 5:  5
 6:  8
 7: 13
 8: 21
 9: 34
10: 55
```

What is functional programming?

What is functional programming?

- "use functions, instead of objects..."

What is functional programming?

- "use functions, instead of objects..."
- "no side effects..."

What is functional programming?

- “use functions, instead of objects...”
- “no side effects...”
- “mumble, mumble...”

Why does functional programming matter?

Why does functional programming matter?

- a sliding scale

Why does functional programming matter?

- a sliding scale
    - Turing: imperative

Why does functional programming matter?

- a sliding scale
  - Turing: imperative
  - Church: functional

Why does functional programming matter?

- a sliding scale
  - Turing: imperative
  - Church: functional
- hardware has changed

Why does functional programming matter?

- a sliding scale
  - Turing: imperative
  - Church: functional
- hardware has changed
  - multi-core processors

Why does functional programming matter?

- a sliding scale
  - Turing: imperative
  - Church: functional
- hardware has changed
  - multi-core processors
  - clusters

Why does functional programming matter?

- a sliding scale
  - Turing: imperative
  - Church: functional
- hardware has changed
  - multi-core processors
  - clusters
- computer languages have changed

Why does functional programming matter?

- a sliding scale
  - Turing: imperative
  - Church: functional
- hardware has changed
  - multi-core processors
  - clusters
- computer languages have changed
- functional programming languages better exploit modern computer hardware than imperative languages

Why does functional programming matter?

- a sliding scale
  - Turing: imperative
  - Church: functional
- hardware has changed
  - multi-core processors
  - clusters
- computer languages have changed
- functional programming languages better exploit modern computer hardware than imperative languages
  - immutability of state (no shared mutable state)

Why does functional programming matter?

- a sliding scale
  - Turing: imperative
  - Church: functional
- hardware has changed
  - multi-core processors
  - clusters
- computer languages have changed
- functional programming languages better exploit modern computer hardware than imperative languages
  - immutability of state (no shared mutable state)
  - pure functions / “referentially transparent” (no side effects)

Why does functional programming matter?

- a sliding scale
  - Turing: imperative
  - Church: functional
- hardware has changed
  - multi-core processors
  - clusters
- computer languages have changed
- functional programming languages better exploit modern computer hardware than imperative languages
  - immutability of state (no shared mutable state)
  - pure functions / “referentially transparent” (no side effects)
  - higher-order functions (“first-class objects”)

Why does functional programming matter?

- a sliding scale
  - Turing: imperative
  - Church: functional
- hardware has changed
  - multi-core processors
  - clusters
- computer languages have changed
- functional programming languages better exploit modern computer hardware than imperative languages
  - immutability of state (no shared mutable state)
  - pure functions / "referentially transparent" (no side effects)
  - higher-order functions ("first-class objects")
  - recursion (no "loops")

Why does functional programming matter?

- a sliding scale
    - Turing: imperative
    - Church: functional
- hardware has changed
    - multi-core processors
    - clusters
- computer languages have changed
- functional programming languages better exploit modern computer hardware than imperative languages
    - immutability of state (no shared mutable state)
    - pure functions / “referentially transparent” (no side effects)
    - higher-order functions (“first-class objects”)
    - recursion (no “loops”)
- scalability

R

R: A Language for Data Analysis and Graphics

Ross IHAKA and Robert GENTLEMAN

In this article we discuss our experience designing and implementing a statistical computing language. In developing this new language, we sought to combine what we felt were useful features from two existing computer languages. We feel that the new language provides advantages in the areas of portability, computational efficiency, memory management, and scoping.

Key Words: Computer language; Statistical computing.

1. INTRODUCTION

This article discusses some issues involved in the design and implementation of a computer language for statistical data analysis. Our experience with these issues occurred while developing such a language. The work has been heavily influenced by two existing languages—Becker, Chambers, and Wilks' S (1985) and Steel and Sussman's Scheme (1975). We felt that there were strong points in each of these languages and that it would be interesting to see if the strengths could be combined. The resulting language is very similar in appearance to S, but the underlying implementation and semantics are derived from Scheme. In fact, we implemented the language by first writing an interpreter for a Scheme subset and then progressively mutating it to resemble S.

Source: [Ihaka and Gentleman, 1996]

Scheme

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 349 December 1975

SCHEME

AN INTERPRETER FOR EXTENDED LAMBDA CALCULUS

by

Gerald Jay Sussman and Guy Lewis Steele Jr.

Abstract:
Inspired by ACTORS [Greif and Hewitt] [Smith and Hewitt], we have implemented an interpreter for a LISP-like language, SCHEME, based on the lambda calculus [Church], but extended for side effects, multiprocessing, and process synchronization. The purpose of this implementation is tutorial. We wish to:

Source: [Sussman and Steele, 1975]

Functional languages

Functional languages

Haskell

An advanced, purely functional programming language

Functional languages

Haskell

An advanced, purely functional programming language

Scala

A programming language that scales with you: from small scripts to large multiplatform applications.

Lambda Calculus: Theory

Functions

- Mathematics

$$x \mapsto x^2$$

Functions

- Mathematics

$$x \mapsto x^2$$

- Haskell

```haskell
(\x -> x^2) 3
```

9

Functions

- Mathematics

$$x \mapsto x^2$$

- Haskell

```haskell
(\x -> x^2) 3
```

9

- R

```r
(\(x) x^2) (3)
```

```
[1] 9
```

Functions

- Mathematics

$$x \mapsto x^2$$

- Haskell

```haskell
(\x -> x^2) 3
```

9

- R

```r
(function(x) x^2) (3)
```

[1] 9

- Mathematics: constrain x

$$f : x \mapsto x^2, \quad \forall x \in \mathbb{Z}$$

Haskell: `Int`

- Mathematics: constrain x

$$f : x \mapsto x^2, \quad \forall x \in \mathbb{Z}$$

- type is metadata

```
:{
  f :: Int -> Int
  f = \x -> x^2
:}
f 3
```

9

Haskell: `Double`

- Mathematics: constrain x

$$f : x \mapsto x^2, \quad \forall x \in \mathbb{R}$$

Haskell: `Double`

- Mathematics: constrain x

$$f : x \mapsto x^2, \quad \forall x \in \mathbb{R}$$

- type is metadata

```
:{
  f :: Double -> Double
  f = \x -> x^2
:}
f 3.0
```

```
9.0
```

Haskell: `Num`

- Mathematics: what if there is no constraint on x?

$$x \mapsto x^2, \quad \forall x \in ???$$

Haskell: `Num`

- Mathematics: what if there is no constraint on x?

$$x \mapsto x^2, \quad \forall x \in ???$$

- Haskell infers type

```
:type \x -> x^2
```

```
\x -> x^2 :: Num a => a -> a
```

Haskell: `Num`

- Mathematics: what if there is no constraint on x?

$$x \mapsto x^2, \quad \forall x \in ???$$

- Haskell infers type

```
:type \x -> x^2
```

```
\x -> x^2 :: Num a => a -> a
```

- type is metadata

```
:{
  f :: Num a => a -> a
  f x = x^2
:}
f 3
```

```
9
```

Haskell: What is `Num`?

Haskell: What is Num?

- Num is a *type class*

```
:info Num
```

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
-- Defined in ‘GHC.Num’
instance Num Word -- Defined in ‘GHC.Num’
instance Num Integer -- Defined in ‘GHC.Num’
instance Num Int -- Defined in ‘GHC.Num’
instance Num Float -- Defined in ‘GHC.Float’
instance Num Double -- Defined in ‘GHC.Float’
```

Haskell: What is `Int`?

- `Int` is a *data constructor*: it is an *instance* of type class `Num`

```
:info Int
```

```
data Int = GHC.Types.I# GHC.Prim.Int#  -- Defined in ‘GHC.Types’
instance Eq Int -- Defined in ‘GHC.Classes’
instance Ord Int -- Defined in ‘GHC.Classes’
instance Enum Int -- Defined in ‘GHC.Enum’
instance Num Int -- Defined in ‘GHC.Num’
instance Real Int -- Defined in ‘GHC.Real’
instance Show Int -- Defined in ‘GHC.Show’
instance Integral Int -- Defined in ‘GHC.Real’
instance Bounded Int -- Defined in ‘GHC.Enum’
instance Read Int -- Defined in ‘GHC.Read’
```

Haskell: What is `Double`?

- `Double` is a *data constructor*: it is an *instance* of type class `Num`

```
:info Double
```

```
data Double = GHC.Types.D# GHC.Prim.Double#
-- Defined in ‘GHC.Types’
instance Eq Double -- Defined in ‘GHC.Classes’
instance Ord Double -- Defined in ‘GHC.Classes’
instance Enum Double -- Defined in ‘GHC.Float’
instance Floating Double -- Defined in ‘GHC.Float’
instance Fractional Double -- Defined in ‘GHC.Float’
instance Num Double -- Defined in ‘GHC.Float’
instance Real Double -- Defined in ‘GHC.Float’
instance RealFloat Double -- Defined in ‘GHC.Float’
instance RealFrac Double -- Defined in ‘GHC.Float’
instance Show Double -- Defined in ‘GHC.Float’
instance Read Double -- Defined in ‘GHC.Read’
```

Back to functions...

- Mathematics

$$f : x \mapsto x^2$$

- R

```
f<- \(x) x^2
f(3)
```

```
[1] 9
```

What if we *only* had anonymous functions? (no *function names*)

- Mathematics

$$x \mapsto x^2$$

- R

```
(\(x) x^2) (3)
```

```
[1] 9
```

What if we *only* had anonymous functions? (no *numerals*...)

- Mathematics

$$x \mapsto x$$

- R

```r
\(x) x
```

```
\(x) x
```

Lambda terms

Lambda terms

- Lambda Calculus has three basic components, or *lambda terms*

Lambda terms

- Lambda Calculus has three basic components, or *lambda terms*
    1. λ expressions

Lambda terms

- Lambda Calculus has three basic components, or *lambda terms*
  1. λ expressions
  2. variables

Lambda terms

- Lambda Calculus has three basic components, or *lambda terms*
  1. λ expressions
  2. variables
  3. abstractions

Lambda terms

- Lambda Calculus has three basic components, or *lambda terms*
  1. λ expressions
  2. variables
  3. abstractions
- λ expressions form a superset Λ

Lambda terms

- Lambda Calculus has three basic components, or *lambda terms*
  1. λ expressions
  2. variables
  3. abstractions
- λ expressions form a superset Λ

$$\lambda \text{ expressions} \supset \{\text{variables}, \text{abstractions}\}$$

Lambda terms

- Lambda Calculus has three basic components, or *lambda terms*
  1. λ expressions
  2. variables
  3. abstractions
- λ expressions form a superset Λ

$$\lambda \text{ expressions} \supset \{\text{variables}, \text{abstractions}\}$$

- variables

Lambda terms

- Lambda Calculus has three basic components, or *lambda terms*
    1. λ expressions
    2. variables
    3. abstractions
- λ expressions form a superset Λ

$$\lambda \text{ expressions} \supset \{\text{variables}, \text{abstractions}\}$$

- variables
    - no meaning or value

Lambda terms

- Lambda Calculus has three basic components, or *lambda terms*
    1. λ expressions
    2. variables
    3. abstractions
- λ expressions form a superset Λ

$$\lambda \text{ expressions} \supset \{\text{variables}, \text{abstractions}\}$$

- variables
    - no meaning or value
    - just a *name* for an input to a function

Lambda terms

- Lambda Calculus has three basic components, or *lambda terms*
  1. λ expressions
  2. variables
  3. abstractions
- λ expressions form a superset Λ

$$\lambda \text{ expressions} \supset \{\text{variables}, \text{abstractions}\}$$

- variables
  - no meaning or value
  - just a *name* for an input to a function
  - *e.g.* x, y, z

Lambda terms

- Lambda Calculus has three basic components, or *lambda terms*
    1. λ expressions
    2. variables
    3. abstractions
- λ expressions form a superset Λ

$$\lambda \text{ expressions} \supset \{\text{variables}, \text{abstractions}\}$$

- variables
    - no meaning or value
    - just a *name* for an input to a function
    - *e.g.* x, y, z
- abstractions

Lambda terms

- Lambda Calculus has three basic components, or *lambda terms*
    1. λ expressions
    2. variables
    3. abstractions
- λ expressions form a superset Λ

$$\lambda \text{ expressions} \supset \{\text{variables}, \text{abstractions}\}$$

- variables
    - no meaning or value
    - just a *name* for an input to a function
    - *e.g.* x, y, z
- abstractions
    - an abstraction is a *function* (usually called a "lambda")

Lambda terms

- Lambda Calculus has three basic components, or *lambda terms*
    1. λ expressions
    2. variables
    3. abstractions
- λ expressions form a superset Λ

$$\lambda \text{ expressions} \supset \{\text{variables}, \text{abstractions}\}$$

- variables
    - no meaning or value
    - just a *name* for an input to a function
    - *e.g.* x, y, z
- abstractions
    - an abstraction is a *function* (usually called a "lambda")
    - two parts (separated by a dot):

$$\underbrace{\lambda x}_{\text{head}} . \underbrace{x}_{\text{body}}$$

Lambda terms

- Lambda Calculus has three basic components, or *lambda terms*
  1. λ expressions
  2. variables
  3. abstractions
- λ expressions form a superset Λ

$$\lambda \text{ expressions} \supset \{\text{variables}, \text{abstractions}\}$$

- variables
  - no meaning or value
  - just a *name* for an input to a function
  - *e.g.* x, y, z
- abstractions
  - an abstraction is a *function* (usually called a “lambda”)
  - two parts (separated by a dot):

$$\underbrace{\lambda x}_{\text{head}} . \underbrace{x}_{\text{body}}$$

  - the body is a λ expression

Lambda terms

- Lambda Calculus has three basic components, or *lambda terms*
  1. λ expressions
  2. variables
  3. abstractions
- λ expressions form a superset Λ

$$\lambda \text{ expressions} \supset \{\text{variables}, \text{abstractions}\}$$

- variables
  - no meaning or value
  - just a *name* for an input to a function
  - *e.g.* x, y, z
- abstractions
  - an abstraction is a *function* (usually called a “lambda”)
  - two parts (separated by a dot):

$$\underbrace{\lambda x}_{\text{head}} . \underbrace{x}_{\text{body}}$$

  - the body is a λ expression

$$\underbrace{\lambda x}_{\text{head}} . \underbrace{\lambda \text{ expression}}_{\text{body}}$$

Application

Application

- the action part of Lambda Calculus: evaluate (*i.e.* reduce) lambda terms

Application

- the action part of Lambda Calculus: evaluate (*i.e.* reduce) lambda terms
  - bind the parameter to the concrete argument

Application

- the action part of Lambda Calculus: evaluate (*i.e.* reduce) lambda terms
  - bind the parameter to the concrete argument

$$(\lambda \underbrace{x}_{\text{parameter}} . \underbrace{x}_{\text{body}}) \underbrace{1}_{\text{argument}}$$

Application

- the action part of Lambda Calculus: evaluate (*i.e.* reduce) lambda terms
  - bind the parameter to the concrete argument

$$(\lambda \underbrace{x}_{\text{parameter}} . \underbrace{x}_{\text{body}}) \underbrace{1}_{\text{argument}}$$

  - the argument is the specific lambda term that the abstraction (*i.e.* the lambda) is applied to

Application

- the action part of Lambda Calculus: evaluate (*i.e.* reduce) lambda terms
  - bind the parameter to the concrete argument

$$(\lambda \underbrace{x}_{\text{parameter}} . \underbrace{x}_{\text{body}}) \underbrace{1}_{\text{argument}}$$

  - the argument is the specific lambda term that the abstraction (*i.e.* the lambda) is applied to
  - a *computation* is the repeated application of lambdas to arguments until there are no applications left to perform

Application

- the action part of Lambda Calculus: evaluate (*i.e.* reduce) lambda terms
  - bind the parameter to the concrete argument

$$(\lambda \underbrace{x}_{\text{parameter}} . \underbrace{x}_{\text{body}}) \underbrace{1}_{\text{argument}}$$

  - the argument is the specific lambda term that the abstraction (*i.e.* the lambda) is applied to
  - a *computation* is the repeated application of lambdas to arguments until there are no applications left to perform
- simplest example of computation: the identity

Application

- the action part of Lambda Calculus: evaluate (*i.e.* reduce) lambda terms
  - bind the parameter to the concrete argument

$$(\lambda \underbrace{x}_{\text{parameter}} . \underbrace{x}_{\text{body}}) \underbrace{1}_{\text{argument}}$$

  - the argument is the specific lambda term that the abstraction (*i.e.* the lambda) is applied to
  - a *computation* is the repeated application of lambdas to arguments until there are no applications left to perform
- simplest example of computation: the identity

$$(\lambda x.x)\,1$$

Application

- the action part of Lambda Calculus: evaluate (*i.e.* reduce) lambda terms
    - bind the parameter to the concrete argument

$$(\lambda \underbrace{x}_{\text{parameter}} . \underbrace{x}_{\text{body}}) \underbrace{1}_{\text{argument}}$$

    - the argument is the specific lambda term that the abstraction (*i.e.* the lambda) is applied to
    - a *computation* is the repeated application of lambdas to arguments until there are no applications left to perform
- simplest example of computation: the identity

$$(\lambda x.x)\, 1$$

$$(\lambda \underbrace{x}_{\text{parameter}} . \underbrace{x}_{\text{body}}) \underbrace{1}_{\text{argument}}$$

Application

- the action part of Lambda Calculus: evaluate (*i.e.* reduce) lambda terms
  - bind the parameter to the concrete argument

$$(\lambda \underbrace{x}_{\text{parameter}} . \underbrace{x}_{\text{body}}) \underbrace{1}_{\text{argument}}$$

  - the argument is the specific lambda term that the abstraction (*i.e.* the lambda) is applied to
  - a *computation* is the repeated application of lambdas to arguments until there are no applications left to perform
- simplest example of computation: the identity

$$(\lambda x.x)\,1$$

$$(\lambda \underbrace{x}_{\text{parameter}} . \underbrace{x}_{\text{body}}) \underbrace{1}_{\text{argument}}$$

$$\lambda \underbrace{[x := 1]}_{\text{parameter}} . \underbrace{x}_{\text{body}}$$

Application

- the action part of Lambda Calculus: evaluate (*i.e.* reduce) lambda terms
  - bind the parameter to the concrete argument

$$(\lambda \underbrace{x}_{\text{parameter}} . \underbrace{x}_{\text{body}}) \underbrace{1}_{\text{argument}}$$

  - the argument is the specific lambda term that the abstraction (*i.e.* the lambda) is applied to
  - a *computation* is the repeated application of lambdas to arguments until there are no applications left to perform
- simplest example of computation: the identity

$$(\lambda x.x)\,1$$

$$(\lambda \underbrace{x}_{\text{parameter}} . \underbrace{x}_{\text{body}}) \underbrace{1}_{\text{argument}}$$

$$\lambda \underbrace{[x := 1]}_{\text{parameter}} . \underbrace{x}_{\text{body}}$$

$$\underbrace{1}_{\text{reduced}}$$

α equivalence, β reduction

α equivalence, β reduction

- alpha equivalence

α equivalence, β reduction

- alpha equivalence
  $\lambda x.x \overset{\alpha}{=} \underbrace{\lambda y.y}_{\text{same thing}}$

α equivalence, β reduction

- alpha equivalence

  $\lambda x.x \stackrel{\alpha}{=} \underbrace{\lambda y.y}_{\text{same thing}}$

  $\lambda x.xy \stackrel{\alpha}{=} \underbrace{\lambda x.xz}_{\text{same thing}}$

α equivalence, β reduction

- alpha equivalence
  $\lambda x.x \stackrel{\alpha}{=} \underbrace{\lambda y.y}_{\text{same thing}}$
  $\lambda x.xy \stackrel{\alpha}{=} \underbrace{\lambda x.xz}_{\text{same thing}}$
- beta reduction

α equivalence, β reduction

- alpha equivalence
  $\lambda x.x \stackrel{\alpha}{=} \underbrace{\lambda y.y}_{\text{same thing}}$
  $\lambda x.xy \stackrel{\alpha}{=} \underbrace{\lambda x.xz}_{\text{same thing}}$
- beta reduction
  1. application: substitute the input λ expression for all instances of bound variables within the body of the abstraction
  2. eliminate the head of the abstraction (its only purpose is to bind a variable)

α equivalence, β reduction

- alpha equivalence

  $\lambda x.x \stackrel{\alpha}{=} \underbrace{\lambda y.y}_{\text{same thing}}$

  $\lambda x.xy \stackrel{\alpha}{=} \underbrace{\lambda x.xz}_{\text{same thing}}$

- beta reduction
  1. application: substitute the input λ expression for all instances of bound variables within the body of the abstraction
  2. eliminate the head of the abstraction (its only purpose is to bind a variable)

$$(\lambda z.z)(\lambda y.y) \quad \stackrel{\alpha}{=} \quad \underbrace{(\lambda x.x)}_{\alpha \text{ equivalence}} \ (\lambda y.y)$$

α equivalence, β reduction

- alpha equivalence
  $\lambda x.x \overset{\alpha}{=} \underbrace{\lambda y.y}_{\text{same thing}}$
  $\lambda x.xy \overset{\alpha}{=} \underbrace{\lambda x.xz}_{\text{same thing}}$
- beta reduction
  1. application: substitute the input λ expression for all instances of bound variables within the body of the abstraction
  2. eliminate the head of the abstraction (its only purpose is to bind a variable)

$$\begin{aligned}(\lambda z.z)(\lambda y.y) &\overset{\alpha}{=} \underbrace{(\lambda x.x)}_{\alpha \text{ equivalence}} (\lambda y.y) \\ &= \lambda[x := \lambda y.y].x\end{aligned}$$

α equivalence, β reduction

- alpha equivalence

  $\lambda x.x \stackrel{\alpha}{=} \underbrace{\lambda y.y}_{\text{same thing}}$

  $\lambda x.xy \stackrel{\alpha}{=} \underbrace{\lambda x.xz}_{\text{same thing}}$

- beta reduction
  1. application: substitute the input λ expression for all instances of bound variables within the body of the abstraction
  2. eliminate the head of the abstraction (its only purpose is to bind a variable)

$$
\begin{aligned}
(\lambda z.z)(\lambda y.y) \quad &\stackrel{\alpha}{=} \quad \underbrace{(\lambda x.x)}_{\alpha \text{ equivalence}} \; (\lambda y.y) \\
&= \quad \lambda[x := \lambda y.y].x \\
&\stackrel{\beta}{\rightarrow} \quad \underbrace{\lambda y.y}_{\text{reduced}}
\end{aligned}
$$

Free variables

Free variables

- here, z is a *free variable*

Free variables

- here, z is a *free variable*
  - $\big((\lambda x.x)(\lambda y.y)\big)z$

Free variables

- here, z is a *free variable*
  - $\big((\lambda x.x)(\lambda y.y)\big)z$
  - β reduce the λ expression:

Free variables

- here, z is a *free variable*
  - $\big((\lambda x.x)(\lambda y.y)\big)\,z$
  - β reduce the λ expression:

$$\big((\lambda x.x)(\lambda y.y)\big)\,z \quad = \quad \big(\lambda[x := \lambda y.y].x\big)\,z$$

Free variables

- here, z is a *free variable*
    - $\big((\lambda x.x)(\lambda y.y)\big)\, z$
    - β reduce the λ expression:

$$\begin{aligned}\big((\lambda x.x)(\lambda y.y)\big)\, z &= \big(\lambda[x := \lambda y.y].x\big)\, z \\ &\xrightarrow{\beta} (\lambda y.y)\, z\end{aligned}$$

Free variables

- here, z is a *free variable*
  - $\big((\lambda x.x)(\lambda y.y)\big)\,z$
  - β reduce the λ expression:

$$\begin{aligned}\big((\lambda x.x)(\lambda y.y)\big)\,z &= \big(\lambda[x := \lambda y.y].x\big)\,z \\ &\overset{\beta}{\to} (\lambda y.y)\,z \\ &= \lambda[y := z].y\end{aligned}$$

Free variables

- here, z is a *free variable*
  - $\big((\lambda x.x)(\lambda y.y)\big)\, z$
  - β reduce the λ expression:

$$
\begin{aligned}
\big((\lambda x.x)(\lambda y.y)\big)\, z &= \big(\lambda[x := \lambda y.y].x\big)\, z \\
&\overset{\beta}{\to} (\lambda y.y)\, z \\
&= \lambda[y := z].y \\
&\overset{\beta}{\to} z
\end{aligned}
$$

Free variables

- here, z is a *free variable*
  - $\big((\lambda x.x)(\lambda y.y)\big)\,z$
  - β reduce the λ expression:

$$
\begin{aligned}
\big((\lambda x.x)(\lambda y.y)\big)\,z &= \big(\lambda[x := \lambda y.y].x\big)\,z \\
&\overset{\beta}{\to} (\lambda y.y)\,z \\
&= \lambda[y := z].y \\
&\overset{\beta}{\to} z
\end{aligned}
$$

- here, y is a *free variable*

Free variables

- here, z is a *free variable*
  - $\big((\lambda x.x)(\lambda y.y)\big)\, z$
  - β reduce the λ expression:

$$
\begin{aligned}
\big((\lambda x.x)(\lambda y.y)\big)\, z &= \big(\lambda[x := \lambda y.y].x\big)\, z \\
&\overset{\beta}{\rightarrow} (\lambda y.y)\, z \\
&= \lambda[y := z].y \\
&\overset{\beta}{\rightarrow} z
\end{aligned}
$$

- here, y is a *free variable*
  - $\lambda x.xy$

Combinators

Combinators

- combinators serve to combine the arguments they are given, without introducing new values (*i.e.* free variables)

Combinators

- combinators serve to combine the arguments they are given, without introducing new values (*i.e.* free variables)
- combinators are λ expressions in which there are *no* free variables

Combinators

- combinators serve to combine the arguments they are given, without introducing new values (*i.e.* free variables)
- combinators are λ expressions in which there are *no* free variables
- examples

Combinators

- combinators serve to combine the arguments they are given, without introducing new values (*i.e.* free variables)
- combinators are λ expressions in which there are *no* free variables
- examples
  - combinator (every λ expression in the body occurs in the head):

$$\lambda x.x$$

Combinators

- combinators serve to combine the arguments they are given, without introducing new values (*i.e.* free variables)
- combinators are λ expressions in which there are *no* free variables
- examples
  - combinator (every λ expression in the body occurs in the head):

$$\lambda x.x$$

  - combinator (every λ expression in the body occurs in the head):

$$\underbrace{\lambda xy}_{\text{note}}.x = \lambda x.(\lambda y.x) = \underbrace{\lambda x.\lambda y.}_{\text{“currying”}}x$$

Combinators

- combinators serve to combine the arguments they are given, without introducing new values (*i.e.* free variables)
- combinators are λ expressions in which there are *no* free variables
- examples
  - combinator (every λ expression in the body occurs in the head):

$$\lambda x.x$$

  - combinator (every λ expression in the body occurs in the head):

$$\underbrace{\lambda xy}_{\text{note}}.x = \lambda x.(\lambda y.x) = \underbrace{\lambda x.\lambda y.}_{\text{“currying”}}x$$

  - *not* a combinator (y is a free variable: it does not occur in the head):

$$\lambda x.y$$

Combinators in use: select *left*

Combinators in use: select *left*

$$(\lambda xy.x)(ab) \quad = \quad (\lambda x.\lambda y.x)(ab)$$

Combinators in use: select *left*

$$
\begin{aligned}
(\lambda xy.x)(ab) &= (\lambda x.\lambda y.x)(ab) \\
&= \big((\lambda x.\lambda y.x)(a)\big)(b)
\end{aligned}
$$

Combinators in use: select *left*

$$\begin{aligned}
(\lambda xy.x)(ab) &= (\lambda x.\lambda y.x)(ab) \\
&= \big((\lambda x.\lambda y.x)(a)\big)(b) \\
&= (\lambda[x := a].\lambda y.x)(b)
\end{aligned}$$

Combinators in use: select *left*

$$
\begin{aligned}
(\lambda xy.x)(ab) \quad &= \quad (\lambda x.\lambda y.x)(ab) \\
&= \quad \big((\lambda x.\lambda y.x)(a)\big)(b) \\
&= \quad (\lambda[x := a].\lambda y.x)(b) \\
&\overset{\beta}{\rightarrow} \quad (\lambda y.a)(b)
\end{aligned}
$$

Combinators in use: select *left*

$$
\begin{aligned}
(\lambda xy.x)(ab) &= (\lambda x.\lambda y.x)(ab) \\
&= \big((\lambda x.\lambda y.x)(a)\big)(b) \\
&= (\lambda[x := a].\lambda y.x)(b) \\
&\overset{\beta}{\rightarrow} (\lambda y.a)(b) \\
&= \lambda[y := b].a
\end{aligned}
$$

Combinators in use: select *left*

$$
\begin{aligned}
(\lambda xy.x)(ab) &= (\lambda x.\lambda y.x)(ab) \\
&= \big((\lambda x.\lambda y.x)(a)\big)(b) \\
&= (\lambda[x := a].\lambda y.x)(b) \\
&\overset{\beta}{\rightarrow} (\lambda y.a)(b) \\
&= \lambda[y := b].a \\
&\overset{\beta}{\rightarrow} a
\end{aligned}
$$

Combinators in use: select *right*

Combinators in use: select *right*

$$(\lambda xy.y)(ab) \quad = \quad (\lambda x.\lambda y.y)(ab)$$

Combinators in use: select *right*

$$
\begin{aligned}
(\lambda xy.y)(ab) &= (\lambda x.\lambda y.y)(ab) \\
&= \big((\lambda x.\lambda y.y)(a)\big)(b)
\end{aligned}
$$

Combinators in use: select *right*

$$
\begin{aligned}
(\lambda xy.y)(ab) &= (\lambda x.\lambda y.y)(ab) \\
&= \big((\lambda x.\lambda y.y)(a)\big)(b) \\
&= (\lambda[x := a].\lambda y.y)(b)
\end{aligned}
$$

Combinators in use: select *right*

$$
\begin{aligned}
(\lambda xy.y)(ab) &= (\lambda x.\lambda y.y)(ab) \\
&= \big((\lambda x.\lambda y.y)(a)\big)(b) \\
&= (\lambda[x := a].\lambda y.y)(b) \\
&\overset{\beta}{\rightarrow} (\lambda y.y)(b)
\end{aligned}
$$

Combinators in use: select *right*

$$
\begin{aligned}
(\lambda xy.y)(ab) &= (\lambda x.\lambda y.y)(ab) \\
&= \big((\lambda x.\lambda y.y)(a)\big)(b) \\
&= (\lambda[x := a].\lambda y.y)(b) \\
&\overset{\beta}{\to} (\lambda y.y)(b) \\
&= \lambda[y := b].b
\end{aligned}
$$

Combinators in use: select *right*

$$
\begin{aligned}
(\lambda xy.y)(ab) &= (\lambda x.\lambda y.y)(ab) \\
&= \big((\lambda x.\lambda y.y)(a)\big)(b) \\
&= (\lambda[x := a].\lambda y.y)(b) \\
&\stackrel{\beta}{\rightarrow} (\lambda y.y)(b) \\
&= \lambda[y := b].b \\
&\stackrel{\beta}{\rightarrow} b
\end{aligned}
$$

Divergence: non-terminating λ expressions

Divergence: non-terminating λ expressions

- consider

$$(\lambda x.xx)(\lambda x.xx)$$

Divergence: non-terminating λ expressions

- consider

$$(\lambda x.xx)(\lambda x.xx)$$

$$\lambda[x := (\lambda x.xx)].xx$$

Divergence: non-terminating λ expressions

- consider

$$(\lambda x.xx)(\lambda x.xx)$$

$$\lambda[x := (\lambda x.xx)].xx$$

$$(\lambda x.xx)(\lambda x.xx)$$

Divergence: non-terminating λ expressions

- consider

$$(\lambda x.xx)(\lambda x.xx)$$

$$\lambda[x := (\lambda x.xx)].xx$$

$$(\lambda x.xx)(\lambda x.xx)$$

- some λ expressions clearly disallow reduction to normal form: such λ expressions *diverge*

Divergence: non-terminating λ expressions

- consider

$$(\lambda x.xx)(\lambda x.xx)$$

$$\lambda[x := (\lambda x.xx)].xx$$

$$(\lambda x.xx)(\lambda x.xx)$$

- some λ expressions clearly disallow reduction to normal form: such λ expressions *diverge*
- the λ expression $(\lambda x.xx)(\lambda x.xx)$ is called “omega”

Backus-Naur Form

The BNF form summarizes the grammar for the Lambda Calculus:

$$\begin{array}{lcl}
\langle \lambda \textit{ expression} \rangle & ::= & <\text{variable}> \\
& \mid & <\text{application}> \\
& \mid & <\text{abstraction}> \\
& & \\
\langle \textit{application} \rangle & ::= & (<\lambda \text{ expression}>) <\lambda \text{ expression}> \\
& & \\
\langle \textit{abstraction} \rangle & ::= & \lambda <\text{variable}>.\ <\lambda \text{ expression}>
\end{array}$$

Source: [Révész, 1988]

Lambda Calculus: Implementation

R package: `al`

https://github.com/tharte/al

R package: `al`

Source: https://en.wikipedia.org/wiki/Alonzo_Church

Booleans & branching

- Truth and Falsity

Booleans & branching

- Truth and Falsity

$$\texttt{true} \stackrel{\text{def}}{=} T \stackrel{\alpha}{=} \lambda x \lambda y . x$$

Booleans & branching

- Truth and Falsity

$$\texttt{true} \stackrel{\text{def}}{=} T \stackrel{\alpha}{=} \lambda x \lambda y.x$$

$$\texttt{false} \stackrel{\text{def}}{=} F \stackrel{\alpha}{=} \lambda x \lambda y.y$$

Booleans & branching

- Truth and Falsity

$$\texttt{true} \stackrel{\text{def}}{=} T \stackrel{\alpha}{=} \lambda x \lambda y.x$$

$$\texttt{false} \stackrel{\text{def}}{=} F \stackrel{\alpha}{=} \lambda x \lambda y.y$$

- if-then-else (the Branching Combinator)

Booleans & branching

- Truth and Falsity

$$\texttt{true} \stackrel{\text{def}}{=} T \stackrel{\alpha}{=} \lambda x \lambda y.x$$

$$\texttt{false} \stackrel{\text{def}}{=} F \stackrel{\alpha}{=} \lambda x \lambda y.y$$

- if-then-else (the Branching Combinator)

$$\texttt{IF} \stackrel{\text{def}}{=} p(ab)$$

Booleans & branching

- Truth and Falsity

$$\texttt{true} \stackrel{\text{def}}{=} T \stackrel{\alpha}{=} \lambda x \lambda y.x$$

$$\texttt{false} \stackrel{\text{def}}{=} F \stackrel{\alpha}{=} \lambda x \lambda y.y$$

- if-then-else (the Branching Combinator)

$$\texttt{IF} \stackrel{\text{def}}{=} p(ab)$$

$$p(ab) \xrightarrow{\beta} T(ab) \stackrel{\alpha}{=} (\lambda x \lambda y.x)(ab) \xrightarrow{\beta} a$$

Booleans & branching

- Truth and Falsity

$$\texttt{true} \stackrel{\text{def}}{=} T \stackrel{\alpha}{=} \lambda x \lambda y.x$$

$$\texttt{false} \stackrel{\text{def}}{=} F \stackrel{\alpha}{=} \lambda x \lambda y.y$$

- if-then-else (the Branching Combinator)

$$\texttt{IF} \stackrel{\text{def}}{=} p(ab)$$

$$p(ab) \stackrel{\beta}{\to} T(ab) \stackrel{\alpha}{=} (\lambda x \lambda y.x)(ab) \stackrel{\beta}{\to} a$$

$$p(ab) \stackrel{\beta}{\to} F(ab) \stackrel{\alpha}{=} (\lambda x \lambda y.y)(ab) \stackrel{\beta}{\to} b$$

Verify that branching works

Verify that branching works

```
IF(true)(\(a) a)(\(b) b)
```

```
\(a) a
```

Verify that branching works

```
IF(true)(\(a) a)(\(b) b)
```

```
\(a) a
```

```
IF(false)(\(a) a)(\(b) b)
```

```
\(b) b
```

Switching between worlds: `.to.logical`

Switching between worlds: `.to.logical`

```
.to.logical
```

```
function (b)
(IF(b)(TRUE))(FALSE)
<bytecode: 0x55fc452616e0>
<environment: namespace:al>
```

Switching between worlds: `.to.logical`

```
.to.logical
```

```
function (b)
(IF(b)(TRUE))(FALSE)
<bytecode: 0x55fc452616e0>
<environment: namespace:al>
```

```
true |> .to.logical()
```

```
[1] TRUE
```

Switching between worlds: `.to.logical`

```
.to.logical
```

```
function (b)
(IF(b)(TRUE))(FALSE)
<bytecode: 0x55fc452616e0>
<environment: namespace:al>
```

```
true |> .to.logical()
```

```
[1] TRUE
```

```
false |> .to.logical()
```

```
[1] FALSE
```

Logic: And

Logic: And

$$\texttt{and} \stackrel{\text{def}}{=} \lambda x \lambda y.xyF$$

Logic: And

$$\texttt{and} \stackrel{\text{def}}{=} \lambda x \lambda y . xyF$$

$$xyF \xrightarrow{\beta} (FT)F \xrightarrow{\beta} F$$

Logic: And

$$\texttt{and} \stackrel{\text{def}}{=} \lambda x \lambda y.xyF$$

$$xyF \xrightarrow{\beta} (FT)F \xrightarrow{\beta} F$$
$$xyF \xrightarrow{\beta} (FF)F \xrightarrow{\beta} F$$

Logic: And

$$\texttt{and} \stackrel{\text{def}}{=} \lambda x \lambda y. xyF$$

$$xyF \xrightarrow{\beta} (FT)F \xrightarrow{\beta} F$$

$$xyF \xrightarrow{\beta} (FF)F \xrightarrow{\beta} F$$

$$xyF \xrightarrow{\beta} (TF)F \xrightarrow{\beta} F$$

Logic: And

$$\texttt{and} \stackrel{\text{def}}{=} \lambda x \lambda y.xyF$$

$$xyF \xrightarrow{\beta} (FT)F \xrightarrow{\beta} F$$

$$xyF \xrightarrow{\beta} (FF)F \xrightarrow{\beta} F$$

$$xyF \xrightarrow{\beta} (TF)F \xrightarrow{\beta} F$$

$$xyF \xrightarrow{\beta} (TT)F \xrightarrow{\beta} T$$

Logic: And

$$\text{and} \overset{\text{def}}{=} \lambda x \lambda y.xyF$$

$$xyF \xrightarrow{\beta} (FT)F \xrightarrow{\beta} F$$
$$xyF \xrightarrow{\beta} (FF)F \xrightarrow{\beta} F$$
$$xyF \xrightarrow{\beta} (TF)F \xrightarrow{\beta} F$$
$$xyF \xrightarrow{\beta} (TT)F \xrightarrow{\beta} T$$

```
and(false)(true) |> .to.logical()
and(false)(false) |> .to.logical()
and(true)(false) |> .to.logical()
and(true)(true) |> .to.logical()
```

```
[1] FALSE
[1] FALSE
[1] FALSE
[1] TRUE
```

Logic: Or

Logic: Or

$$\texttt{or} \stackrel{\text{def}}{=} \lambda x \lambda y . x T y$$

Logic: Or

$$\texttt{or} \stackrel{\text{def}}{=} \lambda x \lambda y . x T y$$

$$x T y \xrightarrow{\beta} (F) T (T) \xrightarrow{\beta} T$$

Logic: Or

$$\texttt{or} \stackrel{\text{def}}{=} \lambda x \lambda y. xTy$$

$$xTy \xrightarrow{\beta} (F)T(T) \xrightarrow{\beta} T$$
$$xTy \xrightarrow{\beta} (F)T(F) \xrightarrow{\beta} F$$

Logic: Or

$$\texttt{or} \stackrel{\text{def}}{=} \lambda x \lambda y. xTy$$

$$xTy \xrightarrow{\beta} (F)T(T) \xrightarrow{\beta} T$$
$$xTy \xrightarrow{\beta} (F)T(F) \xrightarrow{\beta} F$$
$$xTy \xrightarrow{\beta} (T)T(F) \xrightarrow{\beta} T$$

Logic: Or

$$\texttt{or} \stackrel{\text{def}}{=} \lambda x \lambda y.xTy$$

$$xTy \xrightarrow{\beta} (F)T(T) \xrightarrow{\beta} T$$
$$xTy \xrightarrow{\beta} (F)T(F) \xrightarrow{\beta} F$$
$$xTy \xrightarrow{\beta} (T)T(F) \xrightarrow{\beta} T$$
$$xTy \xrightarrow{\beta} (T)T(T) \xrightarrow{\beta} T$$

Logic: Or

$$\texttt{or} \stackrel{\text{def}}{=} \lambda x \lambda y . x T y$$

$$xTy \xrightarrow{\beta} (F)T(T) \xrightarrow{\beta} T$$
$$xTy \xrightarrow{\beta} (F)T(F) \xrightarrow{\beta} F$$
$$xTy \xrightarrow{\beta} (T)T(F) \xrightarrow{\beta} T$$
$$xTy \xrightarrow{\beta} (T)T(T) \xrightarrow{\beta} T$$

```
or(false)(true) |> .to.logical()
or(false)(false) |> .to.logical()
or(true)(false) |> .to.logical()
or(true)(true) |> .to.logical()
```

```
[1] TRUE
[1] FALSE
[1] TRUE
[1] TRUE
```

Logic: Not

Logic: Not

$$\texttt{not} \stackrel{\text{def}}{=} pFT$$

Logic: Not

$$\texttt{not} \stackrel{\text{def}}{=} pFT$$

$$pFT \xrightarrow{\beta} (F)FT \xrightarrow{\beta} T$$

Logic: Not

$$\texttt{not} \stackrel{\text{def}}{=} pFT$$

$$pFT \xrightarrow{\beta} (F)FT \xrightarrow{\beta} T$$

$$pFT \xrightarrow{\beta} (T)FT \xrightarrow{\beta} F$$

Logic: Not

$$\texttt{not} \overset{\text{def}}{=} pFT$$

$$pFT \xrightarrow{\beta} (F)FT \xrightarrow{\beta} T$$

$$pFT \xrightarrow{\beta} (T)FT \xrightarrow{\beta} F$$

```
not(false) |> .to.logical()
not(true) |> .to.logical()
```

```
[1] TRUE
[1] FALSE
```

Church Numerals

$$\texttt{C0} \quad \stackrel{\text{def}}{=} \quad C_0 \stackrel{\alpha}{=} \lambda f.\lambda x.x \stackrel{\alpha}{=} \lambda x.\lambda y.y = \texttt{false}$$

Church Numerals

$$\texttt{C0} \stackrel{\text{def}}{=} C_0 \stackrel{\alpha}{=} \lambda f.\lambda x.x \stackrel{\alpha}{=} \lambda x.\lambda y.y = \texttt{false}$$

$$\texttt{C1} \stackrel{\text{def}}{=} C_1 \stackrel{\alpha}{=} \lambda f.\lambda x.f(x)$$

Church Numerals

$$\texttt{C0} \stackrel{\text{def}}{=} C_0 \stackrel{\alpha}{=} \lambda f.\lambda x.x \stackrel{\alpha}{=} \lambda x.\lambda y.y = \texttt{false}$$

$$\texttt{C1} \stackrel{\text{def}}{=} C_1 \stackrel{\alpha}{=} \lambda f.\lambda x.f(x)$$

$$\texttt{C2} \stackrel{\text{def}}{=} C_2 \stackrel{\alpha}{=} \lambda f.\lambda x.f(f(x))$$

Church Numerals

$$\texttt{C0} \stackrel{\text{def}}{=} C_0 \stackrel{\alpha}{=} \lambda f.\lambda x.x \stackrel{\alpha}{=} \lambda x.\lambda y.y = \texttt{false}$$

$$\texttt{C1} \stackrel{\text{def}}{=} C_1 \stackrel{\alpha}{=} \lambda f.\lambda x.f(x)$$

$$\texttt{C2} \stackrel{\text{def}}{=} C_2 \stackrel{\alpha}{=} \lambda f.\lambda x.f(f(x))$$

$$\texttt{C3} \stackrel{\text{def}}{=} C_3 \stackrel{\alpha}{=} \lambda f.\lambda x.f(f(f(x)))$$

Church Numerals

$$
\begin{aligned}
\texttt{C0} &\stackrel{\text{def}}{=} C_0 \stackrel{\alpha}{=} \lambda f.\lambda x.x \stackrel{\alpha}{=} \lambda x.\lambda y.y = \texttt{false} \\
\texttt{C1} &\stackrel{\text{def}}{=} C_1 \stackrel{\alpha}{=} \lambda f.\lambda x.f(x) \\
\texttt{C2} &\stackrel{\text{def}}{=} C_2 \stackrel{\alpha}{=} \lambda f.\lambda x.f(f(x)) \\
\texttt{C3} &\stackrel{\text{def}}{=} C_3 \stackrel{\alpha}{=} \lambda f.\lambda x.f(f(f(x))) \\
&\vdots
\end{aligned}
$$

Church Numerals

$$\begin{aligned}
\texttt{C0} &\overset{\text{def}}{=} C_0 \overset{\alpha}{=} \lambda f.\lambda x.x \overset{\alpha}{=} \lambda x.\lambda y.y = \texttt{false} \\
\texttt{C1} &\overset{\text{def}}{=} C_1 \overset{\alpha}{=} \lambda f.\lambda x.f(x) \\
\texttt{C2} &\overset{\text{def}}{=} C_2 \overset{\alpha}{=} \lambda f.\lambda x.f(f(x)) \\
\texttt{C3} &\overset{\text{def}}{=} C_3 \overset{\alpha}{=} \lambda f.\lambda x.f(f(f(x))) \\
&\vdots \\
&\overset{\text{def}}{=} C_N \overset{\alpha}{=} \lambda f.\lambda x.f^N(x)
\end{aligned}$$

Switching between worlds: `.to.integer`

Switching between worlds: `.to.integer`

```
.to.integer
```

```
function (n)
n(function(x) x + 1)(0)
<bytecode: 0x55fc442ceef0>
<environment: namespace:al>
```

Switching between worlds: `.to.integer`

```
.to.integer
```

```
function (n)
n(function(x) x + 1)(0)
<bytecode: 0x55fc442ceef0>
<environment: namespace:al>
```

```
C0 |> .to.integer()
C1 |> .to.integer()
C2 |> .to.integer()
C3 |> .to.integer()
```

```
[1] 0
[1] 1
[1] 2
[1] 3
```

Switching between worlds: `.to.integer`

```
`.to.integer`<- \(n) n(\(x) x+1)(0)
```

Switching between worlds: `.to.integer`

```
`.to.integer`<- \(n) n(\(x) x+1)(0)
```

$$\underbrace{(\lambda f.\lambda x.x)}_{n=C_0}\underbrace{(\lambda x.x+1)}_{f}\underbrace{(0)}_{x} \xrightarrow{\beta} 0$$

Switching between worlds: `.to.integer`

```
`.to.integer`<- \(n) n(\(x) x+1)(0)
```

$$\underbrace{(\lambda f.\lambda x.x)}_{n=C_0}\underbrace{(\lambda x.x+1)}_{f}\underbrace{(0)}_{x} \quad \xrightarrow{\beta} \quad 0$$

$$\underbrace{(\lambda f.\lambda x.f(x))}_{n=C_1}(\lambda x.x+1)(0) \quad \xrightarrow{\beta} \quad \underbrace{(\lambda x.x+1)}_{f}\underbrace{(0)}_{x}$$

Switching between worlds: `.to.integer`

```
`.to.integer`<- \(n) n(\(x) x+1)(0)
```

$$\underbrace{(\lambda f.\lambda x.x)}_{n=C_0}\underbrace{(\lambda x.x+1)}_{f}\underbrace{(0)}_{x} \xrightarrow{\beta} 0$$

$$\begin{aligned}\underbrace{(\lambda f.\lambda x.f(x))}_{n=C_1}(\lambda x.x+1)(0) &\xrightarrow{\beta} \underbrace{(\lambda x.x+1)}_{f}\underbrace{(0)}_{x} \\ &\xrightarrow{\beta} 1\end{aligned}$$

Switching between worlds: `.to.integer`

```r
`.to.integer`<- \(n) n(\(x) x+1)(0)
```

$$\underbrace{(\lambda f.\lambda x.x)}_{n=C_0}\underbrace{(\lambda x.x+1)}_{f}\underbrace{(0)}_{x} \xrightarrow{\beta} 0$$

$$\underbrace{(\lambda f.\lambda x.f(x))}_{n=C_1}(\lambda x.x+1)(0) \xrightarrow{\beta} \underbrace{(\lambda x.x+1)}_{f}\underbrace{(0)}_{x}$$

$$\xrightarrow{\beta} 1$$

$$\Big(\underbrace{(\lambda f.\lambda x.f(f(x)))}_{n=C_2}(\lambda x.x+1)\Big)(0) \xrightarrow{\beta} \underbrace{(\lambda x.x+1)}_{f}\underbrace{(\lambda x.x+1)}_{f}\underbrace{(0)}_{x}$$

Switching between worlds: `.to.integer`

```
`.to.integer`<- \(n) n(\(x) x+1)(0)
```

$$\underbrace{(\lambda f.\lambda x.x)}_{n=C_0}\underbrace{(\lambda x.x+1)}_{f}\underbrace{(0)}_{x} \xrightarrow{\beta} 0$$

$$\begin{aligned}
\underbrace{(\lambda f.\lambda x.f(x))}_{n=C_1}(\lambda x.x+1)(0) &\xrightarrow{\beta} \underbrace{(\lambda x.x+1)}_{f}\underbrace{(0)}_{x} \\
&\xrightarrow{\beta} 1
\end{aligned}$$

$$\begin{aligned}
\Big(\underbrace{(\lambda f.\lambda x.f(f(x)))}_{n=C_2}(\lambda x.x+1)\Big)(0) &\xrightarrow{\beta} \underbrace{(\lambda x.x+1)}_{f}\underbrace{(\lambda x.x+1)}_{f}\underbrace{(0)}_{x} \\
&\xrightarrow{\beta} \underbrace{(\lambda x.(\lambda x.x+1)+1)}_{f(f(x))}\underbrace{(0)}_{x}
\end{aligned}$$

Switching between worlds: `.to.integer`

```
`.to.integer`<- \(n) n(\(x) x+1)(0)
```

$$\underbrace{(\lambda f.\lambda x.x)}_{n=C_0}\underbrace{(\lambda x.x+1)}_{f}\underbrace{(0)}_{x} \xrightarrow{\beta} 0$$

$$\begin{aligned}
\underbrace{(\lambda f.\lambda x.f(x))}_{n=C_1}(\lambda x.x+1)(0) &\xrightarrow{\beta} \underbrace{(\lambda x.x+1)}_{f}\underbrace{(0)}_{x} \\
&\xrightarrow{\beta} 1
\end{aligned}$$

$$\begin{aligned}
\Big(\underbrace{(\lambda f.\lambda x.f(f(x)))}_{n=C_2}(\lambda x.x+1)\Big)(0) &\xrightarrow{\beta} \underbrace{(\lambda x.x+1)}_{f}\underbrace{(\lambda x.x+1)}_{f}\underbrace{(0)}_{x} \\
&\xrightarrow{\beta} \underbrace{(\lambda x.(\lambda x.x+1)+1)}_{f(f(x))}\underbrace{(0)}_{x} \\
&\xrightarrow{\beta} 2
\end{aligned}$$

Arithmetic

Arithmetic

- successor

$$\texttt{succ} \stackrel{\text{def}}{=} \lambda n.\lambda f.\lambda x.f((nf)x)$$

Arithmetic

- successor

$$\text{succ} \stackrel{\text{def}}{=} \lambda n.\lambda f.\lambda x.f((nf)x)$$

```
succ(C0) |> .to.integer()
succ(C10) |> .to.integer()
```

```
[1] 1
[1] 11
```

Arithmetic

- successor

$$\mathtt{succ} \stackrel{\text{def}}{=} \lambda n.\lambda f.\lambda x.f((nf)x)$$

```
succ(C0) |> .to.integer()
succ(C10) |> .to.integer()
```

```
[1] 1
[1] 11
```

- predecessor

$$\mathtt{pred} \stackrel{\text{def}}{=} \lambda n.\lambda n(\lambda p.\lambda z.z(\mathtt{succ}(p(T)))(p(T)))(\lambda z.z(C_0)(C_0))(F)$$

Arithmetic

- successor

$$\texttt{succ} \overset{\text{def}}{=} \lambda n.\lambda f.\lambda x.f((nf)x)$$

```
succ(C0) |> .to.integer()
succ(C10) |> .to.integer()
```

```
[1] 1
[1] 11
```

- predecessor

$$\texttt{pred} \overset{\text{def}}{=} \lambda n.\lambda n(\lambda p.\lambda z.z(\texttt{succ}(p(T)))(p(T)))(\lambda z.z(C_0)(C_0))(F)$$

```
pred(C1) |> .to.integer()
pred(C10) |> .to.integer()
```

```
[1] 0
[1] 9
```

Arithmetic

Arithmetic

- addition

$$\texttt{add} \overset{\text{def}}{=} \lambda nm.(m\,\texttt{succ})n$$

Arithmetic

- addition

$$\mathtt{add} \stackrel{\text{def}}{=} \lambda nm.(m\,\mathtt{succ})n$$

```
add(C0)(C1) |> .to.integer()
add(C1)(C2) |> .to.integer()
```

```
[1] 1
[1] 3
```

Arithmetic

- addition

$$\mathtt{add} \stackrel{\text{def}}{=} \lambda nm.(m\,\mathtt{succ})n$$

```
add(C0)(C1) |> .to.integer()
add(C1)(C2) |> .to.integer()
```

```
[1] 1
[1] 3
```

- subtraction

$$\mathtt{sub} \stackrel{\text{def}}{=} \lambda nm.(m\,\text{pred})n$$

Arithmetic

- addition

$$\mathtt{add} \stackrel{\text{def}}{=} \lambda nm.(m\,\mathtt{succ})n$$

```
add(C0)(C1) |> .to.integer()
add(C1)(C2) |> .to.integer()
```

```
[1] 1
[1] 3
```

- subtraction

$$\mathtt{sub} \stackrel{\text{def}}{=} \lambda nm.(m\,\mathsf{pred})n$$

```
sub(C1)(C1) |> .to.integer()
sub(C2)(C1) |> .to.integer()
```

```
[1] 0
[1] 1
```

Arithmetic

Arithmetic

- multiplication

$$\texttt{mul} \stackrel{\text{def}}{=} \lambda nm.m(\texttt{add}n)C_0$$

Arithmetic

- multiplication

$$\texttt{mul} \stackrel{\text{def}}{=} \lambda nm.m(\texttt{add}n)C_0$$

```
mul(C0)(C1) |> .to.integer()
mul(C1)(C2) |> .to.integer()
```

```
[1] 0
[1] 2
```

Arithmetic

- multiplication

$$\texttt{mul} \stackrel{\text{def}}{=} \lambda nm.m(\texttt{add}n)C_0$$

```
mul(C0)(C1) |> .to.integer()
mul(C1)(C2) |> .to.integer()
```

```
[1] 0
[1] 2
```

- exponentiation

$$\texttt{exp} \stackrel{\text{def}}{=} \lambda nm.mn$$

Arithmetic

- multiplication

$$\texttt{mul} \stackrel{\text{def}}{=} \lambda nm.m(\texttt{add}n)C_0$$

```
mul(C0)(C1) |> .to.integer()
mul(C1)(C2) |> .to.integer()
```

```
[1] 0
[1] 2
```

- exponentiation

$$\texttt{exp} \stackrel{\text{def}}{=} \lambda nm.mn$$

```
exp(C2)(C1) |> .to.integer()
exp(C2)(C2) |> .to.integer()
```

```
[1] 2
[1] 4
```

Comparisons

Comparisons

- is-zero

$$\texttt{zerop?} \overset{\text{def}}{=} \lambda n.n(\lambda m.F)\,T$$

Comparisons

- is-zero

$$\texttt{zerop?} \stackrel{\text{def}}{=} \lambda n.n(\lambda m.F)\,T$$

```
`zerop?`(C0) |> .to.logical()
`zerop?`(C1) |> .to.logical()
`zerop?`(C2) |> .to.logical()
```

```
[1] TRUE
[1] FALSE
[1] FALSE
```

Comparisons

- is-zero

$$\texttt{zerop?} \stackrel{\text{def}}{=} \lambda n.n(\lambda m.F)\,T$$

```
`zerop?`(C0) |> .to.logical()
`zerop?`(C1) |> .to.logical()
`zerop?`(C2) |> .to.logical()
```

```
[1] TRUE
[1] FALSE
[1] FALSE
```

- less-than-or-equal

$$\texttt{le} \stackrel{\text{def}}{=} \lambda nm.\texttt{zerop?}(\texttt{sub}(n)(m))$$

Comparisons

- is-zero

$$\texttt{zerop?} \stackrel{\text{def}}{=} \lambda n.n(\lambda m.F)\,T$$

```
`zerop?`(C0) |> .to.logical()
`zerop?`(C1) |> .to.logical()
`zerop?`(C2) |> .to.logical()
```

```
[1] TRUE
[1] FALSE
[1] FALSE
```

- less-than-or-equal

$$\texttt{le} \stackrel{\text{def}}{=} \lambda nm.\texttt{zerop?}(\texttt{sub}(n)(m))$$

```
le(C1)(C1) |> .to.logical()
le(C1)(C2) |> .to.logical()
le(C2)(C1) |> .to.logical()
```

```
[1] TRUE
[1] TRUE
[1] FALSE
```

Comparisons

Comparisons

- is-equal

$$\mathtt{eq} \stackrel{\text{def}}{=} \lambda nm.\mathtt{and}(\mathtt{le}(n)(m))(\mathtt{le}(n)(m))$$

Comparisons

- is-equal

$$\text{eq} \stackrel{\text{def}}{=} \lambda nm.\text{and}(\texttt{le}(n)(m))(\texttt{le}(n)(m))$$

```
eq(C1)(C1) |> .to.logical()
eq(C1)(C2) |> .to.logical()
eq(C2)(C1) |> .to.logical()
eq(C2)(C2) |> .to.logical()
```

```
[1] TRUE
[1] FALSE
[1] FALSE
[1] TRUE
```

Recursion with the Y-Combinator

Recursion with the Y-Combinator

the Y-combinator: Curry's paradoxical combinator

$$\mathtt{Y} \stackrel{\text{def}}{=} \lambda y.\Big(\lambda x.y(xx)\Big)\Big(\lambda x.y(xx)\Big)$$

Recursion with the Y-Combinator

the Y-combinator: Curry's paradoxical combinator

$$\mathtt{Y} \stackrel{\text{def}}{=} \lambda y.\Big(\lambda x.y(xx)\Big)\Big(\lambda x.y(xx)\Big)$$

apply Y to an arbitrary function F

Recursion with the Y-Combinator

the Y-combinator: Curry's paradoxical combinator

$$\mathtt{Y} \stackrel{\text{def}}{=} \lambda y.\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big)$$

apply Y to an arbitrary function F

$$YF \quad = \quad \Big(\lambda y.\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big)\Big)F$$

Recursion with the Y-Combinator

the Y-combinator: Curry's paradoxical combinator

$$\mathtt{Y} \stackrel{\text{def}}{=} \lambda y.\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big)$$

apply Y to an arbitrary function F

$$\begin{aligned} YF &= \Big(\lambda y.\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big)\Big)F \\ &= \lambda[y := F].\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big) \end{aligned}$$

Recursion with the Y-Combinator

the Y-combinator: Curry's paradoxical combinator

$$\mathtt{Y} \stackrel{\text{def}}{=} \lambda y.\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big)$$

apply Y to an arbitrary function F

$$\begin{aligned} YF &= \Big(\lambda y.\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big)\Big)F \\ &= \lambda[y := F].\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big) \\ &\stackrel{\beta}{\to} \big(\lambda x.F(xx)\big)\big(\lambda x.F(xx)\big) \end{aligned}$$

Recursion with the Y-Combinator

the Y-combinator: Curry's paradoxical combinator

$$\mathtt{Y} \stackrel{\text{def}}{=} \lambda y.\Big(\lambda x.y(xx)\Big)\Big(\lambda x.y(xx)\Big)$$

apply Y to an arbitrary function F

$$\begin{aligned} YF &= \Big(\lambda y.\Big(\lambda x.y(xx)\Big)\Big(\lambda x.y(xx)\Big)\Big)F \\ &= \lambda[y := F].\Big(\lambda x.y(xx)\Big)\Big(\lambda x.y(xx)\Big) \\ &\stackrel{\beta}{\to} \Big(\lambda x.F(xx)\Big)\Big(\lambda x.F(xx)\Big) \ = YF \end{aligned}$$

Recursion with the Y-Combinator

the Y-combinator: Curry's paradoxical combinator

$$\mathtt{Y} \stackrel{\text{def}}{=} \lambda y.\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big)$$

apply Y to an arbitrary function F

$$\begin{aligned} YF &= \Big(\lambda y.\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big)\Big)F \\ &= \lambda[y := F].\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big) \\ &\stackrel{\beta}{\to} \big(\lambda x.F(xx)\big)\big(\lambda x.F(xx)\big) \ {\color{blue} = YF} \\ &= \lambda[x := \lambda x.F(xx)].F(xx) \end{aligned}$$

Recursion with the Y-Combinator

the Y-combinator: Curry's paradoxical combinator

$$\mathtt{Y} \stackrel{\text{def}}{=} \lambda y.\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big)$$

apply Y to an arbitrary function F

$$\begin{aligned}
YF &= \Big(\lambda y.\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big)\Big)F \\
&= \lambda[y := F].\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big) \\
&\stackrel{\beta}{\to} \big(\lambda x.F(xx)\big)\big(\lambda x.F(xx)\big) \; {\color{blue} = YF} \\
&= \lambda[x := \lambda x.F(xx)].F(xx) \\
&\stackrel{\beta}{\to} F{\color{blue}\big(\lambda x.F(xx)\big)\big(\lambda x.F(xx)\big)}
\end{aligned}$$

Recursion with the Y-Combinator

the Y-combinator: Curry's paradoxical combinator

$$\mathtt{Y} \stackrel{\text{def}}{=} \lambda y.\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big)$$

apply Y to an arbitrary function F

$$\begin{aligned}
YF &= \Big(\lambda y.\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big)\Big)F \\
&= \lambda[y := F].\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big) \\
&\stackrel{\beta}{\to} \big(\lambda x.F(xx)\big)\big(\lambda x.F(xx)\big) \;= YF \\
&= \lambda[x := \lambda x.F(xx)].F(xx) \\
&\stackrel{\beta}{\to} F\big(\lambda x.F(xx)\big)\big(\lambda x.F(xx)\big) \\
&= F(YF)
\end{aligned}$$

Recursion with the Y-Combinator

the Y-combinator: Curry's paradoxical combinator

$$\mathtt{Y} \stackrel{\text{def}}{=} \lambda y.\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big)$$

apply Y to an arbitrary function F

$$\begin{aligned}
YF &= \Big(\lambda y.\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big)\Big)F \\
&= \lambda[y := F].\big(\lambda x.y(xx)\big)\big(\lambda x.y(xx)\big) \\
&\xrightarrow{\beta} \big(\lambda x.F(xx)\big)\big(\lambda x.F(xx)\big) \ = YF \\
&= \lambda[x := \lambda x.F(xx)].F(xx) \\
&\xrightarrow{\beta} F\big(\lambda x.F(xx)\big)\big(\lambda x.F(xx)\big) \\
&= F(YF)
\end{aligned}$$

x in $Fx = x$ is called a *fixed point* of F

Recursion with the Y-Combinator

the Y-combinator: Curry's paradoxical combinator

$$\mathtt{Y} \stackrel{\text{def}}{=} \lambda y.\Big(\lambda x.y(xx)\Big)\Big(\lambda x.y(xx)\Big)$$

apply Y to an arbitrary function F

$$\begin{array}{rcl} YF & = & \Big(\lambda y.\Big(\lambda x.y(xx)\Big)\Big(\lambda x.y(xx)\Big)\Big)F \\ & = & \lambda[y := F].\Big(\lambda x.y(xx)\Big)\Big(\lambda x.y(xx)\Big) \\ & \stackrel{\beta}{\rightarrow} & \Big(\lambda x.F(xx)\Big)\Big(\lambda x.F(xx)\Big) \;\; = YF \\ & = & \lambda[x := \lambda x.F(xx)].F(xx) \\ & \stackrel{\beta}{\rightarrow} & F\Big(\lambda x.F(xx)\Big)\Big(\lambda x.F(xx)\Big) \\ & = & F(YF) \end{array}$$

x in $Fx = x$ is called a *fixed point* of F

x ends up back at x when F is applied to x

Infinite loop

Infinite loop

$$YI = I(YI)$$

Infinite loop

$$\begin{aligned} YI &= I(YI) \\ &= (\lambda x.x)\big(Y(\lambda x.x)\big) \end{aligned}$$

Infinite loop

$$
\begin{aligned}
YI &= I(YI) \\
&= (\lambda x.x)\big(Y(\lambda x.x)\big) \\
&\ \lambda[x := Y(\lambda x.x)].x
\end{aligned}
$$

Infinite loop

$$
\begin{aligned}
YI &= I(YI) \\
&= (\lambda x.x)\big(Y(\lambda x.x)\big) \\
&\quad \lambda[x := Y(\lambda x.x)].x \\
&\xrightarrow{\beta} Y(\lambda x.x)
\end{aligned}
$$

Infinite loop

$$
\begin{aligned}
YI &= I(YI) \\
&= (\lambda x.x)\big(Y(\lambda x.x)\big) \\
&\quad \lambda[x := Y(\lambda x.x)].x \\
&\xrightarrow{\beta} Y(\lambda x.x) \\
&= YI
\end{aligned}
$$

Infinite loop

$$
\begin{aligned}
YI &= I(YI) \\
&= (\lambda x.x)\big(Y(\lambda x.x)\big) \\
& \quad \lambda[x := Y(\lambda x.x)].x \\
&\xrightarrow{\beta} Y(\lambda x.x) \\
&= YI
\end{aligned}
$$

```
(\(y) (\(x) y(x(x))) (\(x) y(x(x)))) (\(x) x)
```

Infinite loop

$$
\begin{aligned}
YI &= I(YI) \\
&= (\lambda x.x)\big(Y(\lambda x.x)\big) \\
&\quad \lambda[x := Y(\lambda x.x)].x \\
&\xrightarrow{\beta} Y(\lambda x.x) \\
&= YI
\end{aligned}
$$

```
(\(y) (\(x) y(x(x))) (\(x) y(x(x)))) (\(x) x)
```

```
Error: C stack usage  9522372 is too close to the limit
```

Arithmetic series

Arithmetic series

$$S_n = \sum_{i=0}^{i=n} i$$

Arithmetic series

$$S_n = \sum_{i=0}^{i=n} i \quad = \quad \begin{cases} S_0 & = 0 \\ S_n & = n + S_{n-1} \end{cases}$$

Arithmetic series

$$S_n = \sum_{i=0}^{i=n} i \quad = \quad \begin{cases} S_0 & = 0 \\ S_n & = n + S_{n-1} \end{cases}$$

$$F \stackrel{\text{def}}{=} \lambda r.\lambda n.\mathtt{IF}(\grave{}\,\mathtt{zerop?}\grave{}\,n)(C_0)\Big(\mathtt{add}(n)(r(\mathtt{pred}n))\Big)$$

Arithmetic series

$$S_n = \sum_{i=0}^{i=n} i \quad = \quad \begin{cases} S_0 & = 0 \\ S_n & = n + S_{n-1} \end{cases}$$

$$F \stackrel{\text{def}}{=} \lambda r.\lambda n.\mathtt{IF}(\texttt{`zerop?`}\, n)(C_0)\Big(\mathtt{add}(n)(r(\mathtt{pred} n))\Big)$$

$$YFC_3 \xrightarrow{\beta} F(YF)C_3$$

Arithmetic series

$$S_n = \sum_{i=0}^{i=n} i \quad = \quad \begin{cases} S_0 & = 0 \\ S_n & = n + S_{n-1} \end{cases}$$

$$F \stackrel{\text{def}}{=} \lambda r.\lambda n.\mathtt{IF}(\mathtt{`zerop?`}\, n)(C_0)\Big(\mathtt{add}(n)(r(\mathtt{pred}\, n))\Big)$$

$$YFC_3 \xrightarrow{\beta} F(YF)C_3$$

```
F<- (\(r) \(n) IF(`zerop?`(n)) (C0) (add(n)(r(pred(n)))))

Y(F)(C3) |> .to.integer()
```

Arithmetic series

$$S_n = \sum_{i=0}^{i=n} i \quad = \quad \begin{cases} S_0 & = 0 \\ S_n & = n + S_{n-1} \end{cases}$$

$$F \stackrel{\text{def}}{=} \lambda r.\lambda n.\mathtt{IF}(\mathtt{`zerop?`}\, n)(C_0)\Big(\mathtt{add}(n)(r(\mathtt{pred}\, n))\Big)$$

$$YFC_3 \xrightarrow{\beta} F(YF)C_3$$

```
F<- (\(r) \(n) IF(`zerop?`(n)) (C0) (add(n)(r(pred(n)))))

Y(F)(C3) |> .to.integer()
```

```
[1] 6
```

Arithmetic series

```
F<- (\(r) \(n) IF(`zerop?`(n)) (C0) (add(n)(r(pred(n)))))

Y(F)(C3) |> .to.integer()
```

```
[1] 6
```

Arithmetic series

```
Y((\(r) \(n) IF(`zerop?`(n)) (C0) (add(n)(r(pred(n))))))(C3) |> .to.integer()
```

```
[1] 6
```

Arithmetic series

```
(\(y) (\(x) y(x(x))) (\(x) y(x(x)))) (
    \(r) \(n) IF(`zerop?`(n)) (C0) (add(n)(r(pred(n))))) (C3) |> .to.integer()
```

```
[1] 6
```

Arithmetic series

```
(\(y) (\(x) y(x(x))) (\(x) y(x(x)))) (
    (\(r) \(n) (
        \(p) \(x) \(y) p(x)(y)) (
            (\(n) n(\(m) \(x) \(y) y)(\(x) \(y) x))(n)) (
                \(f) \(x) x) ((\(n) \(m) m(\(n) \(f) \(x) f(n(f)(x)))(n))(n)(
                    r((\(n) n(\(p) \(z) z((\(n) \(f) \(x) f(n(f)(x)))(p(\(x) \(y) x)))(
                        p(\(x) \(y) x)))(\(z) z(\(f) \(x) x)(\(f) \(x) x))(
                            \(x) \(y) y))(n)))))) (\(f) \(x) f(f(f(x)))) |> .to.integer()
```

```
[1] 6
```

Fibonacci: R

Fibonacci: R

$$F_n = \begin{cases} F_0 & = 0 \\ F_1 & = 1 \\ F_n & = F_{n-1} + F_{n-2} \end{cases}$$

Fibonacci: R

$$F_n = \begin{cases} F_0 & = 0 \\ F_1 & = 1 \\ F_n & = F_{n-1} + F_{n-2} \end{cases}$$

```
Y(\(r) \(n)
    `zerop?`(n)(C0)
    (`IF`(eq(n)(C1))
        (C1)
        (\(x) add (r( sub(n)(C1) )) (r( sub(n)(C2) )) (x)))) -> fibonacci

lapply(list(C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10),
       \(n) cat(sprintf("%+2s: %+3s\n", n |> .to.integer(), fibonacci(n) |> .to.integer())))
           -> dev.null
```

```
 0:   0
 1:   1
 2:   1
 3:   2
 4:   3
 5:   5
 6:   8
 7:  13
 8:  21
 9:  34
10:  55
```

References

Ihaka, R. and Gentleman, R. (1996).
R: A Language for Data Analysis and Graphics.
Journal of Computational and Graphical Statistics *5*, 299–314.

Révész, G. E. (1988).
Lambda-Calculus, Combinators, and Functional Programming.
Cambridge University Press, Cambridge, England.

Sussman, G. J. and Steele, G. L. (1975).
Scheme: An Interpreter for the Extended Lambda Calculus.
AI Memo No. 349 Massachusetts Institute of Technology.
PDF available online at MIT Libraries:
`https://dspace.mit.edu/handle/1721.1/5794`.