CMPUT325: Applications of λ-calculus

Dr B. Price and Dr. R Greiner

18th October 2004

Introduction

- λ-calculus models calculation using only 2 rules

Introduction

- λ-calculus models calculation using only 2 rules
- Can only represent functions and application

Introduction

- λ-calculus models calculation using only 2 rules
- Can only represent functions and application
- Explicit datastructures and control structures:

Introduction

- λ-calculus models calculation using only 2 rules
- Can only represent functions and application
- Explicit datastructures and control structures: `absent!`

Introduction

- λ-calculus models calculation using only 2 rules
- Can only represent functions and application
- Explicit datastructures and control structures: `absent!`
- How can λ-calculus implement any calcluation?

Introduction

- λ-calculus models calculation using only 2 rules
- Can only represent functions and application
- Explicit datastructures and control structures: `absent!`
- How can λ-calculus implement any calcluation?
- Possible to formalize data and control as function application

Introduction

- λ-calculus models calculation using only 2 rules
- Can only represent functions and application
- Explicit datastructures and control structures: `absent!`
- How can λ-calculus implement any calcluation?
- Possible to formalize data and control as function application
- Standard idioms map high-level data structures and control into λ-C expressions

Abstract Numbers I

- The concept of number can be built up from “0”, "successor"

  i.e., `1=successor(0)`, `2=successor(1)`, `3=successor(2) ...`

Abstract Numbers I

- The concept of number can be built up from "0", "successor"

  i.e., `1=successor(0)`, `2=successor(1)`, `3=successor(2)` ...

- Let $\sigma(n) \equiv$ `successor(n)` ... so $1 = \sigma(0)$, $3 = \sigma(\sigma(\sigma(0)))$, ...

Abstract Numbers I

- The concept of number can be built up from "0", "successor"

  i.e., `1=successor(0)`, `2=successor(1)`, `3=successor(2)` ...

- Let $\sigma(n) \equiv$ `successor(n)` ... so $1 = \sigma(0)$, $3 = \sigma(\sigma(\sigma(0)))$, ...
- Numbers $\approx$ a sequence of function applications

Abstract Numbers I

- The concept of number can be built up from "0", "successor"

  i.e., `1=successor(0)`, `2=successor(1)`, `3=successor(2)` ...

- Let $\sigma(n) \equiv$`successor(n)` ... so $1 = \sigma(0)$, $3 = \sigma(\sigma(\sigma(0)))$, ...
- Numbers $\approx$ a sequence of function applications
- Addition is a short-hand for composing successor

  2+1 $\equiv$ $\sigma(\sigma(\ \sigma(0)\ ))$ $\equiv$ 3

Abstract Numbers I

- The concept of number can be built up from “0”, "successor"

  i.e., `1=successor(0)`, `2=successor(1)`, `3=successor(2)` ...

- Let $\sigma(n) \equiv$`successor(n)` ... so $1 = \sigma(0)$, $3 = \sigma(\sigma(\sigma(0)))$, ...
- Numbers $\approx$ a sequence of function applications
- Addition is a short-hand for composing successor

  $2+1 \equiv \sigma(\sigma(\ \sigma(0)\ )) \equiv 3$

- "Zero" is called the additive identity.
  - $\Rightarrow$ $n + 0 = n$ for any n

Abstract Numbers II

- Successor and addition have natural inverses:

Abstract Numbers II

- Successor and addition have natural inverses:
- Predecessor is the number before the current one. Let $\pi(n)$ be the predecessor of n.
  - $b = \sigma(a) \Rightarrow \pi(b) = a$

Abstract Numbers II

- Successor and addition have natural inverses:
- Predecessor is the number before the current one. Let $\pi(n)$ be the predecessor of n.
  - $b = \sigma(a) \Rightarrow \pi(b) = a$
- Substraction is the inverse of addition: a+b=c $\Rightarrow$ c-b=a

Abstract Numbers II

- Successor and addition have natural inverses:
- Predecessor is the number before the current one. Let $\pi(n)$ be the predecessor of n.
  - $b = \sigma(a) \Rightarrow \pi(b) = a$
- Substraction is the inverse of addition: a+b=c $\Rightarrow$ c-b=a
- Multiplication can be defined in terms of addition; and division as inverse of multiplication

Abstract Numbers II

- Successor and addition have natural inverses:
- Predecessor is the number before the current one.
  Let $\pi(n)$ be the predecessor of n.
  - $b = \sigma(a) \Rightarrow \pi(b) = a$
- Substraction is the inverse of addition: a+b=c $\Rightarrow$ c-b=a
- Multiplication can be defined in terms of addition;
  and division as inverse of multiplication
- Negative numbers and real-numbers can be dervied from
  addition and division

Abstract Numbers II

- Successor and addition have natural inverses:
- Predecessor is the number before the current one.
  Let $\pi(n)$ be the predecessor of n.
  - $b = \sigma(a) \Rightarrow \pi(b) = a$
- Substraction is the inverse of addition: a+b=c $\Rightarrow$ c-b=a
- Multiplication can be defined in terms of addition;
  and division as inverse of multiplication
- Negative numbers and real-numbers can be dervied from addition and division
- First, we need to define the successor function and zero

λ-Calculus Numbers

- Church found idioms with the desired properties:

λ-Calculus Numbers

- Church found idioms with the desired properties:
  - Each number $\approx$ 2-argument functions

λ-Calculus Numbers

- Church found idioms with the desired properties:
  - Each number $\approx$ 2-argument functions
  - 0 $\equiv$ (λs | (λz | z)) $\equiv$ (λs z | z)

λ-Calculus Numbers

- Church found idioms with the desired properties:
  - Each number $\approx$ 2-argument functions
  - 0 $\equiv$ (λs | (λz | z)) $\equiv$ (λs z | z)
  - Successor $\sigma(n) \equiv$ (λx | (λ s z | s (x s z))) $\langle$n$\rangle$

λ-Calculus Numbers

- Church found idioms with the desired properties:
  - Each number $\approx$ 2-argument functions
  - 0 $\equiv$ (λs | (λz | z)) $\equiv$ (λs z | z)
  - Successor $\sigma(n) \equiv$ (λx | (λ s z | s (x s z))) $\langle$n$\rangle$
    - Always returns 2-arg function (λ s z | s ($\langle$n$\rangle$ s z))

λ-Calculus Numbers

- Church found idioms with the desired properties:
  - Each number $\approx$ 2-argument functions
  - 0 $\equiv$ (λs | (λz | z)) $\equiv$ (λs z | z)
  - Successor $\sigma(n) \equiv$ (λx | (λ s z | s (x s z))) ⟨n⟩
    - Always returns 2-arg function (λ s z | s (⟨n⟩ s z))
    - Note: (λ s z | s (⟨n⟩ s z)) applies ⟨n⟩ to 2 function-constants

λ-Calculus Numbers

- Church found idioms with the desired properties:
  - Each number $\approx$ 2-argument functions
  - 0 $\equiv$ (λs | (λz | z)) $\equiv$ (λs z | z)
  - Successor $\sigma(n) \equiv$ (λx | (λ s z | s (x s z))) ⟨n⟩
    - Always returns 2-arg function (λ s z | s (⟨n⟩ s z))
    - Note: (λ s z | s (⟨n⟩ s z)) applies ⟨n⟩ to 2 function-constants
    - Application "copies" body of number into new function

λ-Calculus Numbers

- Church found idioms with the desired properties:
  - Each number $\approx$ 2-argument functions
  - 0 $\equiv$ (λs | (λz | z)) $\equiv$ (λs z | z)
  - Successor $\sigma(n) \equiv$ (λx | (λ s z | s (x s z))) ⟨n⟩
    - Always returns 2-arg function (λ s z | s (⟨n⟩ s z))
    - Note: (λ s z | s (⟨n⟩ s z)) applies ⟨n⟩ to 2 function-constants
    - Application "copies" body of number into new function
- The successor of zero:

$\sigma(0) \equiv$ (λx s z | s (x s z)) (λs z| z)

λ-Calculus Numbers

- Church found idioms with the desired properties:
  - Each number $\approx$ 2-argument functions
  - 0 $\equiv$ (λs | (λz | z)) $\equiv$ (λs z | z)
  - Successor $\sigma(n) \equiv$ (λx | (λ s z | s (x s z))) ⟨n⟩
    - Always returns 2-arg function (λ s z | s (⟨n⟩ s z))
    - Note: (λ s z | s (⟨n⟩ s z)) applies ⟨n⟩ to 2 function-constants
    - Application "copies" body of number into new function

- The successor of zero:

  $\sigma(0)$ $\equiv$ (λx s z | s (x s z)) (λs z| z)

  Free vars get bound in (λs z| z)? No - no free vars!

λ-Calculus Numbers

- Church found idioms with the desired properties:
  - Each number $\approx$ 2-argument functions
  - 0 $\equiv$ (λs | (λz | z)) $\equiv$ (λs z | z)
  - Successor $\sigma(n) \equiv$ (λx | (λ s z | s (x s z))) $\langle$n$\rangle$
    - Always returns 2-arg function (λ s z | s ($\langle$n$\rangle$ s z))
    - Note: (λ s z | s ($\langle$n$\rangle$ s z)) applies $\langle$n$\rangle$ to 2 function-constants
    - Application "copies" body of number into new function
- The successor of zero:

$\sigma(0) \equiv$ (λx s z | s (x s z)) (λs z| z)

Free vars get bound in (λs z| z)? No - no free vars!

$\rightarrow$ (λ s z | s ((λs z | z) s z))

λ-Calculus Numbers

- Church found idioms with the desired properties:
  - Each number $\approx$ 2-argument functions
  - 0 $\equiv$ (λs | (λz | z)) $\equiv$ (λs z | z)
  - Successor $\sigma(n) \equiv$ (λx | (λ s z | s (x s z))) ⟨n⟩
    - Always returns 2-arg function (λ s z | s (⟨n⟩ s z))
    - Note: (λ s z | s (⟨n⟩ s z)) applies ⟨n⟩ to 2 function-constants
    - Application "copies" body of number into new function

- The successor of zero:

$\sigma(0) \equiv$ (λx s z | s (x s z)) (λs z| z)

Free vars get bound in (λs z| z)? No - no free vars!

$\rightarrow$ (λ s z | s ((λs z | z) s z))

$\rightarrow$ (λ s z | s ((λs z | z)) s z))

λ-Calculus Numbers

- Church found idioms with the desired properties:
  - Each number $\approx$ 2-argument functions
  - 0 $\equiv$ (λs | (λz | z)) $\equiv$ (λs z | z)
  - Successor $\sigma(n) \equiv$ (λx | (λ s z | s (x s z))) ⟨n⟩
    - Always returns 2-arg function (λ s z | s (⟨n⟩ s z))
    - Note: (λ s z | s (⟨n⟩ s z)) applies ⟨n⟩ to 2 function-constants
    - Application "copies" body of number into new function
- The successor of zero:

$\sigma(0) \equiv$ (λx s z | s (x s z)) (λs z| z)

Free vars get bound in (λs z| z)? No - no free vars!

→ (λ s z | s ((λs z | z) s z))

→ (λ s z | s ((λs z | z)) s z))

→ (λ s z | s z) ≡ (λ s z | (s z))

Successor

- The successor of one:

$$\sigma(1) \equiv (\lambda \texttt{x s z | s (x s z)}) \; (\lambda \texttt{ s z | (s z)})$$

Successor

- The successor of one:

$$\sigma(1) \equiv (\lambda x \ \mathtt{s}\ \mathtt{z} \mid \mathtt{s}\ (x\ \mathtt{s}\ \mathtt{z}))\ (\lambda\ \mathtt{s}\ \mathtt{z} \mid (\mathtt{s}\ \mathtt{z}))$$
$$\equiv (\lambda\ \mathtt{s}\ \mathtt{z} \mid \mathtt{s}\ ((\lambda\ \mathtt{s}\ \mathtt{z} \mid (\mathtt{s}\ \mathtt{z}))\ \mathtt{s}\ \mathtt{z}))$$

Successor

- The successor of one:

$$\sigma(1) \equiv (\lambda \mathtt{x}\ \mathtt{s}\ \mathtt{z}\ |\ \mathtt{s}\ (\mathtt{x}\ \mathtt{s}\ \mathtt{z}))\ (\lambda\ \mathtt{s}\ \mathtt{z}\ |\ (\mathtt{s}\ \mathtt{z}))$$
$$\equiv (\lambda\ \mathtt{s}\ \mathtt{z}\ |\ \mathtt{s}\ ((\lambda\ \mathtt{s}\ \mathtt{z}\ |\ (\mathtt{s}\ \mathtt{z}))\ \mathtt{s}\ \mathtt{z}))$$
$$\equiv (\lambda\ \mathtt{s}\ \mathtt{z}\ |\ \mathtt{s}\ ((\lambda\ \mathtt{s}\ \mathtt{z}\ |\ (\mathtt{s}\ \mathtt{z}))\ \mathtt{s}\ \mathtt{z}))$$

Successor

- The successor of one:

$$
\begin{aligned}
\sigma(1) &\equiv (\lambda x\ \mathtt{s}\ \mathtt{z}\ |\ \mathtt{s}\ (x\ \mathtt{s}\ \mathtt{z}))\ (\lambda\ \mathtt{s}\ \mathtt{z}\ |\ (\mathtt{s}\ \mathtt{z}))\\
&\equiv (\lambda\ \mathtt{s}\ \mathtt{z}\ |\ \mathtt{s}\ ((\lambda\ \mathtt{s}\ \mathtt{z}\ |\ (\mathtt{s}\ \mathtt{z}))\ \mathtt{s}\ \mathtt{z}))\\
&\equiv (\lambda\ \mathtt{s}\ \mathtt{z}\ |\ \mathtt{s}\ ((\lambda\ \mathtt{s}\ \mathtt{z}\ |\ (\mathtt{s}\ \mathtt{z}))\ \mathtt{s}\ \mathtt{z}))\\
&\equiv (\lambda\ \mathtt{s}\ \mathtt{z}\ |\ \mathtt{s}\ (\mathtt{s}\ \mathtt{z}))\ \equiv (\mathtt{s}\ \mathtt{z}\ |\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{z})))
\end{aligned}
$$

Successor

- The successor of one:

$$\sigma(1) \equiv (\lambda \mathtt{x\ s\ z} \mid \mathtt{s\ (x\ s\ z)})\ (\lambda\ \mathtt{s\ z} \mid \mathtt{(s\ z)})$$
$$\equiv (\lambda\ \mathtt{s\ z} \mid \mathtt{s}\ ((\lambda\ \mathtt{s\ z} \mid \mathtt{(s\ z)})\ \mathtt{s\ z}))$$
$$\equiv (\lambda\ \mathtt{s\ z} \mid \mathtt{s}\ ((\lambda\ \mathtt{s\ z} \mid \mathtt{(s\ z)})\ \mathtt{s\ z}))$$
$$\equiv (\lambda\ \mathtt{s\ z} \mid \mathtt{s\ (s\ z)}) \equiv (\mathtt{s\ z} \mid \mathtt{(s\ (s\ z))})$$

- The successor of two:

$$\sigma(2) \equiv (\lambda \mathtt{x\ s\ z} \mid \mathtt{s\ (x\ s\ z)})\ (\lambda\ \mathtt{s\ z} \mid \mathtt{(s\ (s\ z))})$$

Successor

- The successor of one:

$$\sigma(1) \equiv (\lambda \texttt{x s z | s (x s z)) } (\lambda \texttt{ s z | (s z))}$$
$$\equiv (\lambda \texttt{ s z | s ((}\lambda \texttt{ s z | (s z)) s z))}$$
$$\equiv (\lambda \texttt{ s z | s ((}\lambda \texttt{ s z | (s z)) s z))}$$
$$\equiv (\lambda \texttt{ s z | s (s z))} \equiv \texttt{(s z | (s (s z)))}$$

- The successor of two:

$$\sigma(2) \equiv (\lambda \texttt{x s z | s (x s z)) } (\lambda \texttt{ s z | (s (s z)))}$$
$$\equiv (\lambda \texttt{x s z | s (} (\lambda \texttt{ s z | (s (s z))) s z))}$$

Successor

- The successor of one:

$$\sigma(1) \equiv (\lambda\mathtt{x}\ \mathtt{s}\ \mathtt{z}\ |\ \mathtt{s}\ (\mathtt{x}\ \mathtt{s}\ \mathtt{z}))\ (\lambda\ \mathtt{s}\ \mathtt{z}\ |\ (\mathtt{s}\ \mathtt{z}))$$
$$\equiv (\lambda\ \mathtt{s}\ \mathtt{z}\ |\ \mathtt{s}\ ((\lambda\ \mathtt{s}\ \mathtt{z}\ |\ (\mathtt{s}\ \mathtt{z}))\ \mathtt{s}\ \mathtt{z}))$$
$$\equiv (\lambda\ \mathtt{s}\ \mathtt{z}\ |\ \mathtt{s}\ ((\lambda\ \mathtt{s}\ \mathtt{z}\ |\ (\mathtt{s}\ \mathtt{z}))\ \mathtt{s}\ \mathtt{z}))$$
$$\equiv (\lambda\ \mathtt{s}\ \mathtt{z}\ |\ \mathtt{s}\ (\mathtt{s}\ \mathtt{z}))\ \equiv\ (\mathtt{s}\ \mathtt{z}\ |\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{z})))$$

- The successor of two:

$$\sigma(2) \equiv (\lambda\mathtt{x}\ \mathtt{s}\ \mathtt{z}\ |\ \mathtt{s}\ (\mathtt{x}\ \mathtt{s}\ \mathtt{z}))\ (\lambda\ \mathtt{s}\ \mathtt{z}\ |\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{z})))$$
$$\equiv (\lambda\mathtt{x}\ \mathtt{s}\ \mathtt{z}\ |\ \mathtt{s}\ (\ (\lambda\ \mathtt{s}\ \mathtt{z}\ |\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{z})))\ \mathtt{s}\ \mathtt{z}))$$
$$\equiv (\lambda\mathtt{x}\ \mathtt{s}\ \mathtt{z}\ |\ \mathtt{s}\ (\ (\lambda\ \mathtt{s}\ \mathtt{z}\ |\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{z})))\ \mathtt{s}\ \mathtt{z}))$$

Successor

- The successor of one:

$$\sigma(1) \equiv (\lambda \texttt{x s z | s (x s z)) (}\lambda \texttt{ s z | (s z))}$$
$$\equiv (\lambda \texttt{ s z | s ((}\lambda \texttt{ s z | (s z)) s z))}$$
$$\equiv (\lambda \texttt{ s z | s ((}\lambda \texttt{ s z | (s z)) s z))}$$
$$\equiv (\lambda \texttt{ s z | s (s z))} \equiv \texttt{(s z | (s (s z)))}$$

- The successor of two:

$$\sigma(2) \equiv (\lambda \texttt{x s z | s (x s z)) (}\lambda \texttt{ s z | (s (s z)))}$$
$$\equiv (\lambda \texttt{x s z | s ((}\lambda \texttt{ s z | (s (s z))) s z))}$$
$$\equiv (\lambda \texttt{x s z | s ((}\lambda \texttt{ s z | (s (s z))) s z))}$$
$$\equiv (\lambda \texttt{x s z | s (s (s z)))}$$
$$\equiv (\lambda \texttt{x s z | (s (s (s z))))}$$

Addition

- (+ m n) $\equiv$ (λ x y | (λs z | x s (y s z))) $\langle$m$\rangle$ $\langle$n$\rangle$

Addition

- $(+\ m\ n) \equiv (\lambda\ x\ y \mid (\lambda s\ z \mid x\ s\ (y\ s\ z)\ )\ )\ \langle m \rangle\ \langle n \rangle$
  - Returns a 2-arg function: $(\lambda s\ z \mid \langle m \rangle\ s\ (\langle n \rangle\ s\ z)\ )$

Addition

- $(+ \; m \; n) \equiv (\lambda \; x \; y \mid (\lambda s \; z \mid x \; s \; (y \; s \; z) \;) \;) \; \langle m \rangle \; \langle n \rangle$
  - Returns a 2-arg function: $(\lambda s \; z \mid \langle m \rangle \; s \; (\langle n \rangle \; s \; z) \;)$
  - $\langle n \rangle$ applied to $s \; z$ (copies body into new function)

Addition

- (+ m n) $\equiv$ (λ x y | (λs z | x s (y s z))) $\langle$m$\rangle$ $\langle$n$\rangle$
  - Returns a 2-arg function: (λs z | $\langle$m$\rangle$ s ($\langle$n$\rangle$ s z))
  - $\langle$n$\rangle$ applied to s z (copies body into new function)
  - $\langle$m$\rangle$ is applied to s and copy of $\langle$n$\rangle$

Addition

- (+ m n) $\equiv$ (λ x y | (λs z | x s (y s z))) $\langle$m$\rangle$ $\langle$n$\rangle$
  - Returns a 2-arg function: (λs z | $\langle$m$\rangle$ s ($\langle$n$\rangle$ s z))
  - $\langle$n$\rangle$ applied to s z (copies body into new function)
  - $\langle$m$\rangle$ is applied to s and copy of $\langle$n$\rangle$
    - A total of $\langle$m$\rangle$ successors are composed onto $\langle$n$\rangle$

Addition

- (+ m n) ≡ (λ x y | (λs z | x s (y s z))) ⟨m⟩ ⟨n⟩
  - Returns a 2-arg function: (λs z | ⟨m⟩ s (⟨n⟩ s z))
  - ⟨n⟩ applied to s z (copies body into new function)
  - ⟨m⟩ is applied to s and copy of ⟨n⟩
    - A total of ⟨m⟩ successors are composed onto ⟨n⟩
- Addition of 1+1

```
(λx y | (λs z | x s (y s z) ) )
               (λ s z | s z)    (λ s z | s z)
```

Addition

- $(+\ m\ n) \equiv (\lambda\ x\ y \mid (\lambda s\ z \mid x\ s\ (y\ s\ z)\ )\ )\ \langle m \rangle\ \langle n \rangle$
  - Returns a 2-arg function: $(\lambda s\ z \mid \langle m \rangle\ s\ (\langle n \rangle\ s\ z)\ )$
  - $\langle n \rangle$ applied to s z (copies body into new function)
  - $\langle m \rangle$ is applied to s and copy of $\langle n \rangle$
    - A total of $\langle m \rangle$ successors are composed onto $\langle n \rangle$
- Addition of 1+1

$$(\lambda x\ y \mid (\lambda s\ z \mid x\ s\ (y\ s\ z)\ )\ )$$

$$(\lambda\ s\ z \mid s\ z) \quad (\lambda\ s\ z \mid s\ z)$$

$$\equiv (\lambda s\ z \mid (\lambda\ s\ z \mid s\ z)\ s\ ((\lambda\ s\ z \mid s\ z)\ s\ z)\ )$$

Addition

- (+ m n) ≡ (λ x y | (λs z | x s (y s z))) ⟨m⟩ ⟨n⟩
  - Returns a 2-arg function: (λs z | ⟨m⟩ s (⟨n⟩ s z))
  - ⟨n⟩ applied to s z (copies body into new function)
  - ⟨m⟩ is applied to s and copy of ⟨n⟩
    - A total of ⟨m⟩ successors are composed onto ⟨n⟩
- Addition of 1+1

```
(λx y | (λs z | x s (y s z) ) )
                (λ s z | s z)    (λ s z | s z)
≡ (λs z | (λ s z | s z) s ((λ s z | s z) s z) )
≡ (λs z | (λ s z | s z) s s z)
```

Addition

- $(+\ m\ n) \equiv (\lambda\ x\ y \mid (\lambda s\ z \mid x\ s\ (y\ s\ z)\ )\ )\ \langle m \rangle\ \langle n \rangle$
  - Returns a 2-arg function: $(\lambda s\ z \mid \langle m \rangle\ s\ (\langle n \rangle\ s\ z)\ )$
  - $\langle n \rangle$ applied to s z (copies body into new function)
  - $\langle m \rangle$ is applied to s and copy of $\langle n \rangle$
    - A total of $\langle m \rangle$ successors are composed onto $\langle n \rangle$
- Addition of 1+1

$$
\begin{aligned}
& (\lambda x\ y \mid (\lambda s\ z \mid x\ s\ (y\ s\ z)\ )\ ) \\
& \qquad\qquad (\lambda\ s\ z \mid s\ z) \quad (\lambda\ s\ z \mid s\ z) \\
\equiv\ & (\lambda s\ z \mid (\lambda\ s\ z \mid s\ z)\ s\ ((\lambda\ s\ z \mid s\ z)\ s\ z)\ ) \\
\equiv\ & (\lambda s\ z \mid (\lambda\ s\ z \mid s\ z)\ s\ s\ z) \\
\equiv\ & (\lambda s\ z \mid s\ s\ z)
\end{aligned}
$$

Successor as Addition

- Check: $\sigma(n)$=(+ 1 n)

  (λx y | (λs z | x s (y s z))) (λ s z | s z)

Successor as Addition

- Check: $\sigma(n)$=(+ 1 n)

(λx y | (λs z | x s (y s z))) (λ s z | s z)
≡ (λx y | (λs z | (λ s z | s z) s (y s z)))

Successor as Addition

- Check: $\sigma(n)$=(+ 1 n)

$$
\begin{aligned}
&(\lambda x\ y \mid (\lambda s\ z \mid x\ s\ (y\ s\ z)\ )\ )\ (\lambda\ s\ z \mid s\ z) \\
\equiv\ &(\lambda x\ y \mid (\lambda s\ z \mid (\lambda\ s\ z \mid s\ z)\ s\ (y\ s\ z)\ )\ ) \\
\equiv\ &(\lambda x\ y \mid (\lambda s\ z \mid (\lambda\ s\ z \mid s\ z)\ s\ (y\ s\ z)\ )\ )
\end{aligned}
$$

Successor as Addition

- Check: $\sigma(n)$=(+ 1 n)

$$(\lambda x\ y \mid (\lambda s\ z \mid x\ s\ (y\ s\ z)\ )\ )\ (\lambda\ s\ z \mid s\ z)$$
$$\equiv (\lambda x\ y \mid (\lambda s\ z \mid (\lambda\ s\ z \mid s\ z)\ s\ (y\ s\ z)\ )\ )$$
$$\equiv (\lambda x\ y \mid (\lambda s\ z \mid (\lambda\ s\ z \mid s\ z)\ s\ (y\ s\ z)\ )\ )$$
$$\equiv (\lambda y \mid (\lambda s\ z \mid s\ (y\ s\ z)\ )\ )$$

Successor as Addition

- Check: $\sigma(n)$=(+ 1 n)

  ```
  (λx y | (λs z | x s (y s z) ) ) (λ s z | s z)
  ≡ (λx y | (λs z | (λ s z | s z) s (y s z) ) )
  ≡ (λx y | (λs z | (λ s z | s z) s (y s z) ) )
  ≡ (λy | (λs z | s (y s z) ) )
  ```

- ... equivalent to our definition of successor !
  (λx s z | s (x s z))

Multiplication

- (* m n) $\equiv$ (λ x y (λs | x (y s))) $\langle$m$\rangle$ $\langle$n$\rangle$

Multiplication

- (* m n) $\equiv$ (λ x y (λs | x (y s))) $\langle$m$\rangle$ $\langle$n$\rangle$
  - $\langle$n$\rangle$ passed as 1st argument to number $\langle$m$\rangle$=(λsz|...)

Multiplication

- (* m n) $\equiv$ (λ x y (λs | x (y s))) $\langle$m$\rangle$ $\langle$n$\rangle$
  - $\langle$n$\rangle$ passed as 1^{st} argument to number $\langle$m$\rangle$=(λsz|...)
  - Body of $\langle$n$\rangle$ is copied once for each successor op in $\langle$m$\rangle$

Multiplication

- (* m n) $\equiv$ (λ x y (λs | x (y s))) ⟨m⟩ ⟨n⟩
  - ⟨n⟩ passed as 1^{st} argument to number ⟨m⟩=(λsz|...)
  - Body of ⟨n⟩ is copied once for each successor op in ⟨m⟩
- (* 3 2)

  (λ x y (λs | x (y s)))
  (λs z | s (s (s z))) (λs z | s (s z))

Multiplication

- (* m n) $\equiv$ (λ x y (λs | x (y s))) ⟨m⟩ ⟨n⟩
  - ⟨n⟩ passed as 1^{st} argument to number ⟨m⟩=(λsz|...)
  - Body of ⟨n⟩ is copied once for each successor op in ⟨m⟩
- (* 3 2)

```
($\lambda$ x y ($\lambda$s | x (y s)))
      ($\lambda$s z | s (s (s z))) ($\lambda$s z | s (s z))
$\equiv$($\lambda$s | ($\lambda$s z | s (s (s z))) (($\lambda$s z | s (s z)) s))
```

Multiplication

- (* m n) $\equiv$ (λ x y (λs | x (y s))) $\langle$m$\rangle$ $\langle$n$\rangle$
  - $\langle$n$\rangle$ passed as 1^{st} argument to number $\langle$m$\rangle$=(λsz|...)
  - Body of $\langle$n$\rangle$ is copied once for each successor op in $\langle$m$\rangle$
- (* 3 2)

```
(λ x y (λs | x (y s)))
      (λs z | s (s (s z))) (λs z | s (s z))
≡(λs | (λs z | s (s (s z))) ((λs z | s (s z)) s))
≡(λs | (λs z | s (s (s z))) ((λs z | s (s z)) s))
```

Multiplication

- (* m n) ≡ (λ x y (λs | x (y s))) ⟨m⟩ ⟨n⟩
  - ⟨n⟩ passed as 1st argument to number ⟨m⟩=(λsz|...)
  - Body of ⟨n⟩ is copied once for each successor op in ⟨m⟩
- (* 3 2)

```
(λ x y (λs | x (y s)))
      (λs z | s (s (s z))) (λs z | s (s z))
≡(λs | (λs z | s (s (s z))) ((λs z | s (s z)) s))
≡(λs | (λs z | s (s (s z))) ((λs z | s (s z)) s))
≡(λs | (λs z | s (s (s z))) (λz | s (s z)))
```

Multiplication

- (* m n) $\equiv$ (λ x y (λs | x (y s))) $\langle$m$\rangle$ $\langle$n$\rangle$
  - $\langle$n$\rangle$ passed as 1^{st} argument to number $\langle$m$\rangle$=(λsz|...)
  - Body of $\langle$n$\rangle$ is copied once for each successor op in $\langle$m$\rangle$
- (* 3 2)

```
(λ x y (λs | x (y s)))
     (λs z | s (s (s z))) (λs z | s (s z))
≡(λs | (λs z | s (s (s z))) ((λs z | s (s z)) s))
≡(λs | (λs z | s (s (s z))) ((λs z | s (s z)) s))
≡(λs | (λs z | s (s (s z))) (λz | s (s z)))
≡(λs | (λs z | s (s (s z))) (λz | s (s z)))
```

Multiplication

- (* m n) ≡ (λ x y (λs | x (y s))) ⟨m⟩ ⟨n⟩
  - ⟨n⟩ passed as 1*st* argument to number ⟨m⟩=(λsz|...)
  - Body of ⟨n⟩ is copied once for each successor op in ⟨m⟩
- (* 3 2)

```
(λ x y (λs | x (y s)))
     (λs z | s (s (s z))) (λs z | s (s z))
≡(λs | (λs z | s (s (s z))) ((λs z | s (s z)) s))
≡(λs | (λs z | s (s (s z))) ((λs z | s (s z)) s))
≡(λs | (λs z | s (s (s z))) (λz | s (s z)))
≡(λs | (λs z | s (s (s z))) (λz | s (s z)))
≡(λs | (λz |
    (λz | s (s z))((λz | s (s z))((λz | s (s z)) z)))
```

Multiplication

- `(* m n)` $\equiv$ `(`λ` x y (`λ`s | x (y s)))` $\langle$`m`$\rangle$ $\langle$`n`$\rangle$
  - $\langle$`n`$\rangle$ passed as 1st argument to number $\langle$`m`$\rangle$`=(`λ`sz|...)`
  - Body of $\langle$`n`$\rangle$ is copied once for each successor op in $\langle$`m`$\rangle$
- (* 3 2)

```
(λ x y (λs | x (y s)))
     (λs z | s (s (s z))) (λs z | s (s z))
≡(λs | (λs z | s (s (s z))) ((λs z | s (s z)) s))
≡(λs | (λs z | s (s (s z))) ((λs z | s (s z)) s))
≡(λs | (λs z | s (s (s z))) (λz | s (s z)))
≡(λs | (λs z | s (s (s z))) (λz | s (s z)))
≡(λs | (λz |
    (λz | s (s z))((λz | s (s z))((λz | s (s z)) z)))
≡(λs z | s (s (s (s (s (s z))))))
```

Multiplication by Zero

- (* 0 m)

  $(\lambda$ x y | $(\lambda$s | x (y s))) $(\lambda$s z| z) $\langle$m$\rangle$

Multiplication by Zero

- (* 0 m)

  (λ x y | (λs | x (y s))) (λs z| z) ⟨m⟩
  ≡(λ y | (λs | (λs z| z) (y s))) ⟨m⟩

Multiplication by Zero

- (* 0 m)

  (λ x y | (λs | x (y s))) (λs z| z) ⟨m⟩
  ≡(λ y | (λs | (λs z| z) (y s))) ⟨m⟩
  ≡(λ y | (λs | (λs z| z) (y s))) ⟨m⟩

Multiplication by Zero

- (* 0 m)

  (λ x y | (λs | x (y s))) (λs z| z) ⟨m⟩
  ≡(λ y | (λs | (λs z| z) (y s))) ⟨m⟩
  ≡(λ y | (λs | (λs z| z) (y s))) ⟨m⟩
  ≡(λ y | (λs | (λz | z))) ⟨m⟩

Multiplication by Zero

- (* 0 m)

```
(λ x y | (λs | x (y s))) (λs z| z) ⟨m⟩
≡(λ y | (λs | (λs z| z) (y s))) ⟨m⟩
≡(λ y | (λs | (λs z| z) (y s))) ⟨m⟩
≡(λ y | (λs | (λz | z))) ⟨m⟩
≡(λ y | (λs z | z)) ⟨m⟩
```

Multiplication by Zero

- (* 0 m)

  (λ x y | (λs | x (y s))) (λs z| z) ⟨m⟩
  ≡(λ y | (λs | (λs z| z) (y s))) ⟨m⟩
  ≡(λ y | (λs | (λs z| z) (y s))) ⟨m⟩
  ≡(λ y | (λs | (λz | z))) ⟨m⟩
  ≡(λ y | (λs z | z)) ⟨m⟩

- Take any number m = (λs z | s s . . . s z)

  (λ y (λs z| z)) m

Multiplication by Zero

- (* 0 m)

  (λ x y | (λs | x (y s))) (λs z| z) ⟨m⟩
  ≡(λ y | (λs | (λs z| z) (y s))) ⟨m⟩
  ≡(λ y | (λs | (λs z| z) (y s))) ⟨m⟩
  ≡(λ y | (λs | (λz | z))) ⟨m⟩
  ≡(λ y | (λs z | z)) ⟨m⟩

- Take any number m = (λs z | s s . . . s z)

  (λ y (λs z| z)) m
  ≡ (λs z| z)

Predecessor and Subtraction

- For n>0, predecessor returns the integer before n, otherwise it returns zero

Predecessor and Subtraction

- For n>0, predecessor returns the integer before n, otherwise it returns zero
- There is no way to take apart a λ-calculus expression easily (Predecessor is not simply removing an 's' from the body of a Church number)

```
PREDECESSOR≡
  n (λag|(a(λbc|c))(⟨successor⟩(a(λbc|c))))
                                          (λg|00) (λab|a)
```

Predecessor and Subtraction

- For n>0, predecessor returns the integer before n, otherwise it returns zero
- There is no way to take apart a λ-calculus expression easily (Predecessor is not simply removing an 's' from the body of a Church number)

  PREDECESSOR≡
  n (λag|(a(λbc|c))(⟨successor⟩(a(λbc|c))))
  (λg|00) (λab|a)

- Applies a function that maps (x,y) to (y,y+1) to the pair (0,0) n times

Predecessor and Subtraction

- For n>0, predecessor returns the integer before n, otherwise it returns zero
- There is no way to take apart a λ-calculus expression easily (Predecessor is not simply removing an 's' from the body of a Church number)

  PREDECESSOR≡
  n (λag|(a(λbc|c))(⟨successor⟩(a(λbc|c))))
  (λg|00) (λab|a)

- Applies a `function` that maps
  (x,y) to (y,y+1) to the pair (0,0) n times
- Results in the pair (n-1,n)

Predecessor and Subtraction

- For n>0, predecessor returns the integer before n, otherwise it returns zero
- There is no way to take apart a λ-calculus expression easily (Predecessor is not simply removing an 's' from the body of a Church number)

  PREDECESSOR≡
  ```
  n (λag|(a(λbc|c))(⟨successor⟩(a(λbc|c))))
                                              (λg|00) (λab|a)
  ```
- Applies a `function` that maps (x,y) to (y,y+1) to the pair `(0,0)` n times
- Results in the pair (n-1,n)
- The `left number`, n-1 is the predecessor

Predecessor and Subtraction

- For n>0, predecessor returns the integer before n, otherwise it returns zero
- There is no way to take apart a λ-calculus expression easily (Predecessor is not simply removing an 's' from the body of a Church number)

  PREDECESSOR≡
  n (λag|(a(λbc|c))(⟨successor⟩(a(λbc|c))))
  (λg|00) (λab|a)

- Applies a `function` that maps (x,y) to (y,y+1) to the pair `(0,0)` n times
- Results in the pair (n-1,n)
- The `left number`, n-1 is the predecessor
- (- m n) ≡(λmn | n⟨predecessor⟩m)

Boolean Expressions

- Define the boolean values

Boolean Expressions

- Define the boolean values
  - True: `T = (`λ` c d | c)`
    - Returns its first argument

Boolean Expressions

- Define the boolean values
  - True: T = (λ c d | c)
    - Returns its first argument
  - False: F $\equiv$(λ c d | d)
    - Returns its second argument

Boolean Expressions

- Define the boolean values
  - True: `T = (λ c d | c)`
    - Returns its first argument
  - False: `F ≡(λ c d | d)`
    - Returns its second argument
- Boolean functions

```
(not m) ≡ (λx | x F T)
```

Boolean Expressions

- Define the boolean values
  - True: `T = (λ c d | c)`
    - Returns its first argument
  - False: `F` $\equiv$`(λ c d | d)`
    - Returns its second argument
- Boolean functions

```
(not m) ≡ (λx | x F T)
        ≡ (λx | x (λc d| d) (λc d|c) )
```

Boolean Expressions

- Define the boolean values
  - True: `T = (λ c d | c)`
    - Returns its first argument
  - False: `F` $\equiv$`(λ c d | d)`
    - Returns its second argument
- Boolean functions

$$\texttt{(not m)} \equiv \texttt{(}\lambda\texttt{x | x F T)}$$
$$\equiv \texttt{(}\lambda\texttt{x | x (}\lambda\texttt{c d| d) (}\lambda\texttt{c d|c))}$$

- Example `(not t)`

$$\equiv \texttt{(}\lambda\texttt{x | x (}\lambda\texttt{c d| d) (}\lambda\texttt{c d|c)) (}\lambda\texttt{c d|c)}$$

Boolean Expressions

- Define the boolean values
  - True: `T = (λ c d | c)`
    - Returns its first argument
  - False: `F` $\equiv$`(λ c d | d)`
    - Returns its second argument
- Boolean functions

```
(not m) ≡ (λx | x F T)
        ≡ (λx | x (λc d| d) (λc d|c) )
```

- Example `(not t)`

```
≡ (λx | x (λc d| d) (λc d|c) ) (λc d|c)
≡ (λc d|c) (λc d| d) (λc d|c)
```

Boolean Expressions

- Define the boolean values
  - True: `T = (λ c d | c)`
    - Returns its first argument
  - False: `F ≡(λ c d | d)`
    - Returns its second argument
- Boolean functions

```
(not m) ≡ (λx | x F T)
        ≡ (λx | x (λc d| d) (λc d|c) )
```

- Example `(not t)`

```
≡ (λx | x (λc d| d) (λc d|c) ) (λc d|c)
≡ (λc d|c) (λc d| d) (λc d|c)
≡ (λc d|c) (λc d| d) (λc d|c)
```

Boolean Expressions

- Define the boolean values
  - True: `T = (λ c d | c)`
    - Returns its first argument
  - False: `F ≡(λ c d | d)`
    - Returns its second argument
- Boolean functions

$$\texttt{(not m)} \equiv (\lambda \texttt{x} \mid \texttt{x F T})$$
$$\equiv (\lambda \texttt{x} \mid \texttt{x } (\lambda \texttt{c d} \mid \texttt{d}) \ (\lambda \texttt{c d} \mid \texttt{c}) \ )$$

- Example `(not t)`

$$\equiv (\lambda \texttt{x} \mid \texttt{x } (\lambda \texttt{c d} \mid \texttt{d}) \ (\lambda \texttt{c d} \mid \texttt{c}) \ ) \ (\lambda \texttt{c d} \mid \texttt{c})$$
$$\equiv (\lambda \texttt{c d} \mid \texttt{c}) \ (\lambda \texttt{c d} \mid \texttt{d}) \ (\lambda \texttt{c d} \mid \texttt{c})$$
$$\equiv (\lambda \texttt{c d} \mid \texttt{c}) \ (\lambda \texttt{c d} \mid \texttt{d}) \ (\lambda \texttt{c d} \mid \texttt{c})$$
$$\equiv (\lambda \texttt{c d} \mid \texttt{d}) \quad \equiv \texttt{F}$$

Boolean Expressions

- (and m n) $\equiv$(λx y | x y F)
  - So if x is F, will return 2^{nd} arg F
  - Otherwise if x is T will return 1^{st} arg y

Boolean Expressions

- (and m n) $\equiv$(λx y | x y F)
  - So if x is F, will return 2^{nd} arg F
  - Otherwise if x is T will return 1^{st} arg y

```
(and T F)
≡ (λx y| x y F) (λcd|c) (λcd|d)
```

Boolean Expressions

- (and m n) $\equiv(\lambda$x y | x y F)
  - So if x is F, will return 2^{nd} arg F
  - Otherwise if x is T will return 1^{st} arg y

```
(and T F)
≡ (λx y| x y F) (λcd|c) (λcd|d)
≡ ((λcd|c) (λcd|d) F)
```

Boolean Expressions

- (and m n) $\equiv(\lambda$x y | x y F)
  - So if x is F, will return 2^{nd} arg F
  - Otherwise if x is T will return 1^{st} arg y

```
(and T F)
≡ (λx y| x y F) (λcd|c) (λcd|d)
≡ ((λcd|c) (λcd|d) F)
≡ ((λcd|c) (λcd|d) F)
```

Boolean Expressions

- (and m n) $\equiv(\lambda x\ y \mid x\ y\ F)$
  - So if x is F, will return 2^{nd} arg F
  - Otherwise if x is T will return 1^{st} arg y

```
(and T F)
≡ (λx y| x y F) (λcd|c) (λcd|d)
≡ ((λcd|c) (λcd|d) F)
≡ ((λcd|c) (λcd|d) F)
≡ (λcd|d)
```

Boolean Expressions

- (and m n) $\equiv(\lambda$x y | x y F)
  - So if x is F, will return 2^{nd} arg F
  - Otherwise if x is T will return 1^{st} arg y

```
(and T F)
≡ (λx y| x y F) (λcd|c) (λcd|d)
≡ ((λcd|c) (λcd|d) F)
≡ ((λcd|c) (λcd|d) F)
≡ (λcd|d)
(and F T)
≡ (λx y| x y F) (λcd|d) (λcd|c)
```

Boolean Expressions

- (and m n) $\equiv(\lambda$x y | x y F)
  - So if x is F, will return 2^{nd} arg F
  - Otherwise if x is T will return 1^{st} arg y

```
(and T F)
≡ (λx y| x y F) (λcd|c) (λcd|d)
≡ ((λcd|c) (λcd|d) F)
≡ ((λcd|c) (λcd|d) F)
≡ (λcd|d)
(and F T)
≡ (λx y| x y F) (λcd|d) (λcd|c)
≡ (λcd|d) (λcd|c) F
```

Boolean Expressions

- (and m n) $\equiv$(λx y | x y F)
  - So if x is F, will return 2^{nd} arg F
  - Otherwise if x is T will return 1^{st} arg y

```
(and T F)
≡ (λx y| x y F) (λcd|c) (λcd|d)
≡ ((λcd|c) (λcd|d) F)
≡ ((λcd|c) (λcd|d) F)
≡ (λcd|d)
(and F T)
≡ (λx y| x y F) (λcd|d) (λcd|c)
≡ (λcd|d) (λcd|c) F
≡ (λcd|d) (λcd|c) F
```

Boolean Expressions

- (and m n) $\equiv(\lambda$x y | x y F)
  - So if x is F, will return 2^{nd} arg F
  - Otherwise if x is T will return 1^{st} arg y

```
(and T F)
≡ (λx y| x y F) (λcd|c) (λcd|d)
≡ ((λcd|c) (λcd|d) F)
≡ ((λcd|c) (λcd|d) F)
≡ (λcd|d)
(and F T)
≡ (λx y| x y F) (λcd|d) (λcd|c)
≡ (λcd|d) (λcd|c) F
≡ (λcd|d) (λcd|c) F
≡ F ≡ (λcd|d)
```

OR, ZEROP and Math Predicates

- OR(⟨F⟩, ⟨G⟩) is true if ⟨F⟩ is true or ⟨G⟩ is true

  OR(x)≡(λwz|wTz)

OR, ZEROP and Math Predicates

- OR(⟨F⟩, ⟨G⟩) is true if ⟨F⟩ is true or ⟨G⟩ is true

  OR(x)≡(λwz|wTz)

- ZEROP(n) returns true if n=0

  zerop(n)≡(λx|x F not F)

OR, ZEROP and Math Predicates

- OR(⟨F⟩, ⟨G⟩) is true if ⟨F⟩ is true or ⟨G⟩ is true

  OR(x)≡(λwz|wTz)

- ZEROP(n) returns true if n=0

  zerop(n)≡(λx|x F not F)

- Relations on integers

  x≥y≡

OR, ZEROP and Math Predicates

- OR(⟨F⟩, ⟨G⟩) is true if ⟨F⟩ is true or ⟨G⟩ is true

  OR(x)≡(λwz|wTz)

- ZEROP(n) returns true if n=0

  zerop(n)≡(λx|x F not F)

- Relations on integers

  x≥y≡zerop(x-y)

OR, ZEROP and Math Predicates

- OR(⟨F⟩, ⟨G⟩) is true if ⟨F⟩ is true or ⟨G⟩ is true

  OR(x)≡(λwz|wTz)

- ZEROP(n) returns true if n=0

  zerop(n)≡(λx|x F not F)

- Relations on integers

  x≥y≡zerop(x-y)
  x<y ≡

OR, ZEROP and Math Predicates

- OR(⟨F⟩, ⟨G⟩) is true if ⟨F⟩ is true or ⟨G⟩ is true

  OR(x)≡(λwz|wTz)

- ZEROP(n) returns true if n=0

  zerop(n)≡(λx|x F not F)

- Relations on integers

  x≥y≡zerop(x-y)
  x<y ≡ not x≥y

OR, ZEROP and Math Predicates

- OR(⟨F⟩, ⟨G⟩) is true if ⟨F⟩ is true or ⟨G⟩ is true

  OR(x)≡(λwz|wTz)

- ZEROP(n) returns true if n=0

  zerop(n)≡(λx|x F not F)

- Relations on integers

  x≥y≡zerop(x-y)
  x<y ≡ not x≥y
  x=y ≡

OR, ZEROP and Math Predicates

- OR(⟨F⟩, ⟨G⟩) is true if ⟨F⟩ is true or ⟨G⟩ is true

  OR(x)≡(λwz|wTz)

- ZEROP(n) returns true if n=0

  zerop(n)≡(λx|x F not F)

- Relations on integers

  x≥y≡zerop(x-y)
  x<y ≡ not x≥y
  x=y ≡x≥y AND y≥x

Conditional

- If P then M else N $\equiv$ (λuvw | uvw) PMN

Conditional

- If P then M else N $\equiv (\lambda uvw \mid uvw)$ PMN
- P is a function returning true T or false F

Conditional

- If P then M else N $\equiv$ (λuvw | uvw) PMN
- P is a function returning true T or false F
- Recall, T returns first argument, F returns second

Conditional

- If P then M else N $\equiv$ (λuvw | uvw) PMN
- P is a function returning true T or false F
- Recall, T returns first argument, F returns second
- Example: (IF T M N)

  $\equiv$ `(λuvw | uvw) TMN`

Conditional

- If P then M else N $\equiv (\lambda uvw \mid uvw)$ PMN
- P is a function returning true T or false F
- Recall, T returns first argument, F returns second
- Example: (IF T M N)

$\equiv$ (λuvw | uvw) TMN
$\equiv$ (λuvw | uvw) TMN

Conditional

- If P then M else N $\equiv (\lambda uvw \mid uvw)$ PMN
- P is a function returning true T or false F
- Recall, T returns first argument, F returns second
- Example: (IF T M N)

$$\equiv (\lambda \mathtt{uvw} \mid \mathtt{uvw})\ \mathtt{TMN}$$
$$\equiv (\lambda \mathtt{uvw} \mid \mathtt{uvw})\ \mathtt{TMN}$$
$$\equiv (\lambda \mathtt{vw} \mid \mathtt{Tvw})\ \mathtt{MN}$$

Conditional

- If P then M else N $\equiv (\lambda uvw \mid uvw)$ PMN
- P is a function returning true T or false F
- Recall, T returns first argument, F returns second
- Example: (IF T M N)

$\equiv$ (λuvw | uvw) TMN
$\equiv$ (λuvw | uvw) TMN
$\equiv$ (λvw | Tvw) MN
$\equiv$ (λvw | (λcd|c) vw) MN

Conditional

- If P then M else N $\equiv$ (λuvw | uvw) PMN
- P is a function returning true T or false F
- Recall, T returns first argument, F returns second
- Example: (IF T M N)

$\equiv$ (λuvw | uvw) TMN
$\equiv$ (λuvw | uvw) TMN
$\equiv$ (λvw | Tvw) MN
$\equiv$ (λvw | (λcd|c) vw) MN
$\equiv$ (λvw | v) MN

Conditional

- If P then M else N $\equiv$ (λuvw | uvw) PMN
- P is a function returning true T or false F
- Recall, T returns first argument, F returns second
- Example: (IF T M N)

$\equiv$ (λuvw | uvw) TMN
$\equiv$ (λuvw | uvw) TMN
$\equiv$ (λvw | Tvw) MN
$\equiv$ (λvw | (λcd|c) vw) MN
$\equiv$ (λvw | v) MN
$\equiv$ M

Lists

- Cons cell `(M . N)` represented as 1 arg function `(λz| z M N)`

Lists

- Cons cell `(M . N)` represented as 1 arg function $(\lambda z|\ z\ M\ N)$
- Cons operator: (cons M N) $\equiv(\lambda x\ y\ (\lambda\ z\ |\ z\ x\ y))$ M N

Lists

- Cons cell `(M . N)` represented as 1 arg function `(`λ`z| z M N)`
- Cons operator: (cons M N) $\equiv$(λx y `(`λ `z | z x y)`) M N
- First: (car m) $\equiv$ (λx | x T) m   where T$\equiv$(λcd|c)

Lists

- Cons cell `(M . N)` represented as 1 arg function `(`λ`z| z M N)`
- Cons operator: (cons M N) $\equiv$(λx y `(`λ `z | z x y)`) M N
- First: (car m) $\equiv$ (λx | x T) m   where T$\equiv$(λcd|c)
- Rest: (cdr m) $\equiv$(λx | x F) m   where F$\equiv$(λcd|d)

Lists

- Cons cell (M . N) represented as 1 arg function $(\lambda z|\ z\ M\ N)$
- Cons operator: (cons M N) $\equiv(\lambda x\ y\ (\lambda\ z\ |\ z\ x\ y))$ M N
- First: (car m) $\equiv$ $(\lambda x \mid x\ T)$ m where T$\equiv(\lambda cd|c)$
- Rest: (cdr m) $\equiv(\lambda x \mid x\ F)$ m where F$\equiv(\lambda cd|d)$
- Example: (car $(\lambda\ z \mid z\ M\ N)$)

  $\equiv\ (\lambda x\ |\ x\ T)\ (\lambda\ z\ |\ z\ M\ N)$

Lists

- Cons cell `(M . N)` represented as 1 arg function `(λz| z M N)`
- Cons operator: (cons M N) $\equiv$(λx y `(λ z | z x y)`) M N
- First: (car m) $\equiv$ (λx | x T) m   where T$\equiv$(λcd|c)
- Rest: (cdr m) $\equiv$(λx | x F) m   where F$\equiv$(λcd|d)
- Example: (car (λ z | z M N))

  $\equiv$ `(λx | x T) (λ z | z M N)`
  $\equiv$ `(λ z | z M N) T`

Lists

- Cons cell `(M . N)` represented as 1 arg function $(\lambda z|\ z\ M\ N)$
- Cons operator: (cons M N) $\equiv(\lambda x\ y\ (\lambda\ z\ |\ z\ x\ y))$ M N
- First: (car m) $\equiv (\lambda x \mid x\ T)$ m   where T$\equiv(\lambda cd|c)$
- Rest: (cdr m) $\equiv(\lambda x \mid x\ F)$ m   where F$\equiv(\lambda cd|d)$
- Example: (car $(\lambda\ z \mid z$ M N))

  $\equiv\ (\lambda x\ |\ x\ T)\ (\lambda\ z\ |\ z\ M\ N)$
  $\equiv\ (\lambda\ z\ |\ z\ M\ N)\ T$
  $\equiv\ (T\ M\ N)\ \equiv\ ((\lambda cd|c)\ M\ N)\ \equiv\ M$

Lists

- Cons cell $(M\ .\ N)$ represented as 1 arg function $(\lambda z |\ z\ M\ N)$
- Cons operator: (cons M N) $\equiv(\lambda x\ y\ (\lambda\ z\ |\ z\ x\ y))\ M\ N$
- First: (car m) $\equiv (\lambda x \mid x\ T)\ m$ where $T \equiv (\lambda cd|c)$
- Rest: (cdr m) $\equiv(\lambda x \mid x\ F)\ m$ where $F\equiv(\lambda cd|d)$
- Example: (car $(\lambda\ z \mid z\ M\ N)$)

  $\equiv\ (\lambda x\ |\ x\ T)\ (\lambda\ z\ |\ z\ M\ N)$
  $\equiv\ (\lambda\ z\ |\ z\ M\ N)\ T$
  $\equiv\ (T\ M\ N)\ \equiv\ ((\lambda cd|c)\ M\ N)\ \equiv\ M$

- Example: (cdr $(\lambda z|\ z\ M\ N)$)

  $\equiv\ (\lambda x\ |\ x\ F)\ (\lambda\ z\ |\ z\ M\ N)$

Lists

- Cons cell `(M . N)` represented as 1 arg function $(\lambda z|$ `z M N)`
- Cons operator: (cons M N) $\equiv(\lambda x\ y\ (\lambda\ z\ |\ z\ x\ y))$ M N
- First: (car m) $\equiv$ $(\lambda x \mid x\ T)$ m   where T$\equiv(\lambda cd|c)$
- Rest: (cdr m) $\equiv(\lambda x \mid x\ F)$ m   where F$\equiv(\lambda cd|d)$
- Example: (car $(\lambda\ z \mid z$ M N))

  $\equiv$ $(\lambda x$ | x T) $(\lambda$ z | z M N)
  $\equiv$ $(\lambda$ z | z M N) T
  $\equiv$ (T M N) $\equiv$ $((\lambda cd|c)$ M N) $\equiv$ M

- Example: (cdr $(\lambda z|$ z M N))

  $\equiv$ $(\lambda x$ | x F) $(\lambda$ z | z M N)
  $\equiv$ $(\lambda$ z | z M N) F

Lists

- Cons cell `(M . N)` represented as 1 arg function `(λz| z M N)`
- Cons operator: (cons M N) $\equiv$(λx y `(λ z | z x y)`) M N
- First: (car m) $\equiv$ (λx | x T) m   where T$\equiv$(λcd|c)
- Rest: (cdr m) $\equiv$(λx | x F) m   where F$\equiv$(λcd|d)
- Example: (car (λ z | z M N))

  ```
  ≡ (λx | x T) (λ z | z M N)
  ≡ (λ z | z M N) T
  ≡ (T M N) ≡ ((λcd|c) M N) ≡ M
  ```

- Example: (cdr (λz| z M N))

  ```
  ≡ (λx | x F) (λ z | z M N)
  ≡ (λ z | z M N) F
  ≡ (F M N) ≡ ((λcd|d) M N) ≡ N
  ```

Alternative Definition of Numbers I

- Define numerals as recursive lists

Alternative Definition of Numbers I

- Define numerals as recursive lists
  - $0 \equiv (\lambda x | \ x)$

Alternative Definition of Numbers I

- Define numerals as recursive lists
  - $0 \equiv (\lambda \text{x} |\ \text{x})$
  - $\sigma(n) \equiv (1+\ \text{n}) \equiv (\text{cons F n})$ for all $n \geq 0$ where $\text{F} \equiv (\lambda \text{cd} | \text{d})$

Alternative Definition of Numbers I

- Define numerals as recursive lists
  - $0 \equiv (\lambda x|\ x)$
  - $\sigma(n) \equiv$(1+ n) $\equiv$(cons F n) for all $n \geq 0$ where F$\equiv(\lambda$cd$|$d)
  - $\pi(n) \equiv$(1- n) $\equiv$(cdr n) $\equiv (\lambda x \mid x$ F)

Alternative Definition of Numbers I

- Define numerals as recursive lists
  - $0 \equiv (\lambda x|\ x)$
  - $\sigma(n) \equiv$(1+ n) $\equiv$(cons F n) for all $n \geq 0$ where F$\equiv(\lambda$cd$|$d)
  - $\pi(n) \equiv$(1- n) $\equiv$(cdr n) $\equiv (\lambda x \mid x\ F)$
  - (zerop n) $\equiv$ (first n) $\equiv (\lambda x \mid x\ T)$

Alternative Definition of Numbers I

- Define numerals as recursive lists
  - $0 \equiv (\lambda x|\ x)$
  - $\sigma(n) \equiv$(1+ n) $\equiv$(cons F n) for all $n \geq 0$ where F$\equiv(\lambda$cd$|$d)
  - $\pi(n) \equiv$(1- n) $\equiv$(cdr n) $\equiv (\lambda$x | x F)
  - (zerop n) $\equiv$ (first n) $\equiv (\lambda$x | x T)
- Examples

$\sigma(n) \equiv$ `(cons F ·)`

Alternative Definition of Numbers I

- Define numerals as recursive lists
  - $0 \equiv (\lambda x | \; x)$
  - $\sigma(n) \equiv (1+ \; n) \equiv (\text{cons F n})$ for all $n \geq 0$ where $F \equiv (\lambda cd | d)$
  - $\pi(n) \equiv (1- \; n) \equiv (\text{cdr n}) \equiv (\lambda x \mid x \; F)$
  - $(\text{zerop n}) \equiv (\text{first n}) \equiv (\lambda x \mid x \; T)$
- Examples

$$\sigma(n) \equiv \texttt{(cons F }\cdot\texttt{)}$$
$$\equiv \texttt{(}\lambda\texttt{x y (}\lambda\texttt{ z | z x y)) F}$$

Alternative Definition of Numbers I

- Define numerals as recursive lists
  - $0 \equiv (\lambda x |\ x)$
  - $\sigma(n) \equiv$(1+ n) $\equiv$(cons F n) for all $n \geq 0$ where F$\equiv(\lambda$cd$|$d)
  - $\pi(n) \equiv$(1- n) $\equiv$(cdr n) $\equiv (\lambda x \mid x\ F)$
  - (zerop n) $\equiv$ (first n) $\equiv (\lambda x \mid x\ T)$
- Examples

$$\sigma(n) \equiv \texttt{(cons F} \cdot\texttt{)}$$
$$\equiv \texttt{(}\lambda\texttt{x y (}\lambda\texttt{ z | z x y)) F}$$
$$\equiv \texttt{(}\lambda\texttt{y (}\lambda\texttt{z | z F y))}$$

Alternative Definition of Numbers II

$$1 \equiv \sigma(0) \equiv \texttt{(cons F 0)}$$

Alternative Definition of Numbers II

$$1 \equiv \sigma(0) \equiv \texttt{(cons F 0)}$$
$$\equiv \texttt{(}\lambda\texttt{y | (}\lambda\texttt{z| z F y)) (}\lambda\texttt{x| x)}$$

Alternative Definition of Numbers II

$$1 \equiv \sigma(0) \equiv \texttt{(cons F 0)}$$
$$\equiv \texttt{(}\lambda\texttt{y | (}\lambda\texttt{z| z F y)) (}\lambda\texttt{x| x)}$$
$$\equiv \texttt{(}\lambda\texttt{z| z F (}\lambda\texttt{x| x)))}$$

Alternative Definition of Numbers II

$$
\begin{aligned}
\texttt{1} &\equiv \sigma(\texttt{0}) \equiv \texttt{(cons F 0)} \\
&\equiv (\lambda\texttt{y | (}\lambda\texttt{z| z F y)) (}\lambda\texttt{x| x)} \\
&\equiv (\lambda\texttt{z| z F (}\lambda\texttt{x| x)))} \\
&\equiv \texttt{[F 0]}
\end{aligned}
$$

Alternative Definition of Numbers II

$$1 \equiv \sigma(0) \equiv \texttt{(cons F 0)}$$
$$\equiv \texttt{(}\lambda\texttt{y | (}\lambda\texttt{z| z F y)) (}\lambda\texttt{x| x)}$$
$$\equiv \texttt{(}\lambda\texttt{z| z F (}\lambda\texttt{x| x)))}$$
$$\equiv \texttt{[F 0]}$$

$$2 \equiv \sigma(1) \equiv \texttt{(cons F 1)} \equiv \texttt{(cons F (cons F 0))}$$

Alternative Definition of Numbers II

$1 \equiv \sigma(0) \equiv$ (cons F 0)
$\equiv$ (λy | (λz| z F y)) (λx| x)
$\equiv$ (λz| z F (λx| x)))
$\equiv$ [F 0]

$2 \equiv \sigma(1) \equiv$ (cons F 1) $\equiv$ (cons F (cons F 0))
$\equiv$ (λy | (λz| z F y)) (λz| z F (λx| x)))

Alternative Definition of Numbers II

$1 \equiv \sigma(0) \equiv$ (cons F 0)
$\equiv$ (λy | (λz| z F y)) (λx| x)
$\equiv$ (λz| z F (λx| x)))
$\equiv$ [F 0]

$2 \equiv \sigma(1) \equiv$ (cons F 1) $\equiv$ (cons F (cons F 0))
$\equiv$ (λy | (λz| z F y)) (λz| z F (λx| x)))
$\equiv$ (λz| z F (λz| z F (λx| x))))

Alternative Definition of Numbers II

```
1 ≡ σ(0) ≡ (cons F 0)
≡ (λy | (λz| z F y)) (λx| x)
≡ (λz| z F (λx| x)))
≡ [F 0]

2 ≡σ(1) ≡ (cons F 1) ≡ (cons F (cons F 0))
≡ (λy | (λz| z F y)) (λz| z F (λx| x)))
≡ (λz| z F (λz| z F (λx| x))) )
≡ [F F 0]
```

Alternative Definition of Numbers II

$1 \equiv \sigma(0) \equiv$ (cons F 0)
$\equiv$ (λy | (λz| z F y)) (λx| x)
$\equiv$ (λz| z F (λx| x)))
$\equiv$ [F 0]

$2 \equiv \sigma(1) \equiv$ (cons F 1) $\equiv$ (cons F (cons F 0))
$\equiv$ (λy | (λz| z F y)) (λz| z F (λx| x)))
$\equiv$ (λz| z F (λz| z F (λx| x))))
$\equiv$ [F F 0]

$3 \equiv \sigma(1) \equiv$ (cons F 1) $\equiv$ (cons F (cons F 0))

Alternative Definition of Numbers II

```
1 ≡ σ(0) ≡ (cons F 0)
≡ (λy | (λz| z F y)) (λx| x)
≡ (λz| z F (λx| x)))
≡ [F 0]

2 ≡σ(1) ≡ (cons F 1) ≡ (cons F (cons F 0))
≡ (λy | (λz| z F y)) (λz| z F (λx| x)))
≡ (λz| z F (λz| z F (λx| x))) )
≡ [F F 0]

3 ≡σ(1) ≡ (cons F 1) ≡ (cons F (cons F 0))
≡ (λy | (λz| z F y))
        (λz| z F (λz| z F (λx| x))) )
```

Alternative Definition of Numbers II

```
1 ≡ σ(0) ≡ (cons F 0)
≡ (λy | (λz| z F y)) (λx| x)
≡ (λz| z F (λx| x)))
≡ [F 0]

2 ≡σ(1) ≡ (cons F 1) ≡ (cons F (cons F 0))
≡ (λy | (λz| z F y)) (λz| z F (λx| x)))
≡ (λz| z F (λz| z F (λx| x))) )
≡ [F F 0]

3 ≡σ(1) ≡ (cons F 1) ≡ (cons F (cons F 0))
≡ (λy | (λz| z F y))
        (λz| z F (λz| z F (λx| x))) )
≡ (λz| z F
        (λz| z F (λz| z F (λx| x))) ) )
```

Alternative Definition of Numbers II

```
1 ≡ σ(0) ≡ (cons F 0)
≡ (λy | (λz| z F y)) (λx| x)
≡ (λz| z F (λx| x)))
≡ [F 0]

2 ≡σ(1) ≡ (cons F 1) ≡ (cons F (cons F 0))
≡ (λy | (λz| z F y)) (λz| z F (λx| x)))
≡ (λz| z F (λz| z F (λx| x))) )
≡ [F F 0]

3 ≡σ(1) ≡ (cons F 1) ≡ (cons F (cons F 0))
≡ (λy | (λz| z F y))
        (λz| z F (λz| z F (λx| x))) )
≡ (λz| z F
       (λz| z F (λz| z F (λx| x))) ) )
≡ [F F F 0]
```

Alternate Definition of Plus

- Analysis of examples of (+ m n)

  `(+ [0] [0])` $\longrightarrow$ `[0] Easy`

Alternate Definition of Plus

- Analysis of examples of (+ m n)

```
(+ [0] [0])   → [0] Easy
(+ [0] [F 0]) → [F 0] Easy
```

Alternate Definition of Plus

- Analysis of examples of (+ m n)

```
(+ [0] [0])     → [0] Easy
(+ [0] [F 0])   → [F 0] Easy
(+ [F 0] [F 0]) → [F F 0]
```

Alternate Definition of Plus

- Analysis of examples of (+ m n)

```
(+ [0] [0])   → [0] Easy
(+ [0] [F 0]) → [F 0] Easy
(+ [F 0] [F 0]) → [F F 0]
  ≡(+ [0] [F F 0]) Made easy
```

Alternate Definition of Plus

- Analysis of examples of (+ m n)

```
(+ [0] [0])    → [0] Easy
(+ [0] [F 0]) → [F 0] Easy
(+ [F 0] [F 0]) → [F F 0]
  ≡(+ [0] [F F 0]) Made easy
(+ [F F 0] [F 0])
```

Alternate Definition of Plus

- Analysis of examples of (+ m n)

```
(+ [0] [0])    → [0] Easy
(+ [0] [F 0]) → [F 0] Easy
(+ [F 0] [F 0]) → [F F 0]
  ≡(+ [0] [F F 0]) Made easy
(+ [F F 0] [F 0])
  ≡ (+ [0] [F F F 0]) →[F F F 0] Made easy
```

Alternate Definition of Plus

- Analysis of examples of (+ m n)

```
(+ [0] [0])   → [0] Easy
(+ [0] [F 0]) → [F 0] Easy
(+ [F 0] [F 0]) → [F F 0]
  ≡(+ [0] [F F 0]) Made easy
(+ [F F 0] [F 0])
  ≡ (+ [0] [F F F 0]) →[F F F 0] Made easy
```

- Leads to a recursive definition of plus (+ m n)

```
plus≡(λx y |
        (zerop x)   ;; T returns first arg!
          y
          (plus (pred x) (succ y)) )  m n
```

Alternate Definition of Plus

- Analysis of examples of (+ m n)

```
(+ [0] [0])    → [0] Easy
(+ [0] [F 0]) → [F 0] Easy
(+ [F 0] [F 0]) → [F F 0]
  ≡(+ [0] [F F 0]) Made easy
(+ [F F 0] [F 0])
  ≡ (+ [0] [F F F 0]) →[F F F 0] Made easy
```

- Leads to a recursive definition of plus (+ m n)

```
plus≡(λx y |
         (zerop x)   ;; T returns first arg!
          y
          (plus (pred x) (succ y)) )  m n
```

- Recursive plus is self-referential

Alternate Definition of Plus

- Analysis of examples of (+ m n)

```
(+ [0] [0])    → [0] Easy
(+ [0] [F 0]) → [F 0] Easy
(+ [F 0] [F 0]) → [F F 0]
  ≡(+ [0] [F F 0]) Made easy
(+ [F F 0] [F 0])
  ≡ (+ [0] [F F F 0]) →[F F F 0] Made easy
```

- Leads to a recursive definition of plus (+ m n)

```
plus≡(λx y |
        (zerop x)    ;; T returns first arg!
          y
          (plus (pred x) (succ y)) )  m n
```

- Recursive plus is self-referential
- Need a technique for recursion

Recursion in λ-Calculus

- Recursion reuses same code repeatedly. Earlier we saw

```
(λ x | x x) (λ x | x x)
```

Recursion in λ-Calculus

- Recursion reuses same code repeatedly. Earlier we saw

```
(λ x | x x) (λ x | x x)
→ (λ x | x x)  (λ x | x x)
```

Recursion in λ-Calculus

- Recursion reuses same code repeatedly. Earlier we saw

  (λ x | x x) (λ x | x x)
  → (λ x | x x)   (λ x | x x)

- The expression (λ x | x x) is preserved indefinitely!

Recursion in λ-Calculus

- Recursion reuses same code repeatedly. Earlier we saw

```
(λ x | x x) (λ x | x x)
→ (λ x | x x)  (λ x | x x)
```

- The expression (λ x | x x) is preserved indefinitely!
- Let R be a function. What does this variation do?

```
(λ x | R (x x)) (λ x | R (x x))
```

Recursion in λ-Calculus

- Recursion reuses same code repeatedly. Earlier we saw

  (λ x | x x) (λ x | x x)
  $\rightarrow$ (λ x | x x) (λ x | x x)

- The expression (λ x | x x) is preserved indefinitely!

- Let R be a function. What does this variation do?

  (λ x | R (x x)) (λ x | R (x x))
  $\rightarrow$ R ((λ x | R (x x)) (λ x | R (x x)))

Recursion in λ-Calculus

- Recursion reuses same code repeatedly. Earlier we saw

```
(λ x | x x) (λ x | x x)
→ (λ x | x x)  (λ x | x x)
```

- The expression (λ x | x x) is preserved indefinitely!
- Let R be a function. What does this variation do?

```
(λ x | R (x x)) (λ x | R (x x))
→ R ( (λ x | R (x x)) (λ x | R (x x)) )
```

- Note:

Recursion in λ-Calculus

- Recursion reuses same code repeatedly. Earlier we saw

```
(λ x | x x) (λ x | x x)
→ (λ x | x x)  (λ x | x x)
```

- The expression $(\lambda$ x | x x$)$ is preserved indefinitely!
- Let R be a function. What does this variation do?

```
(λ x | R (x x)) (λ x | R (x x))
→ R ( (λ x | R (x x)) (λ x | R (x x)) )
```

- Note:
  - 1 copy of R "peeled" off

Recursion in λ-Calculus

- Recursion reuses same code repeatedly. Earlier we saw

  ```
  (λ x | x x) (λ x | x x)
  → (λ x | x x)  (λ x | x x)
  ```

- The expression (λ x | x x) is preserved indefinitely!
- Let R be a function. What does this variation do?

  ```
  (λ x | R (x x)) (λ x | R (x x))
  → R ( (λ x | R (x x)) (λ x | R (x x)) )
  ```

- Note:
  - 1 copy of R "peeled" off
  - Self-replicating expression preserved:
    (λ x | R (x x)) (λ x | R (x x))

Fixed-Point Combinator

- A fixed point for a function is an argument whose image is itself

Fixed-Point Combinator

- A fixed point for a function is an argument whose image is itself

```
x is a fixed-point of f, if f(x)=x
```

Fixed-Point Combinator

- A fixed point for a function is an argument whose image is itself

  `x is a fixed-point of f, if f(x)=x`

- The square function has 2 fixed points $0^2 = 0$ and $1^2 = 1$

Fixed-Point Combinator

- A fixed point for a function is an argument whose image is itself

```
x is a fixed-point of f, if f(x)=x
```

- The square function has 2 fixed points $0^2 = 0$ and $1^2 = 1$
- A fixed-point combinator finds the fixed point of a function.

Fixed-Point Combinator

- A fixed point for a function is an argument whose image is itself

  `x is a fixed-point of f, if f(x)=x`

- The square function has 2 fixed points $0^2 = 0$ and $1^2 = 1$
- A fixed-point combinator finds the fixed point of a function.
- Let `Y` be a fixed-point combinator. `Y(square)=1`

Fixed-Point Combinator

- A fixed point for a function is an argument whose image is itself

  `x is a fixed-point of f, if f(x)=x`

- The square function has 2 fixed points $0^2 = 0$ and $1^2 = 1$
- A fixed-point combinator finds the fixed point of a function.
- Let `Y` be a fixed-point combinator. `Y(square)=1`
  - Because `square(1)=1`

Fixed-Point Combinator

- A fixed point for a function is an argument whose image is itself

  `x is a fixed-point of f, if f(x)=x`

- The square function has 2 fixed points $0^2 = 0$ and $1^2 = 1$
- A fixed-point combinator finds the fixed point of a function.
- Let `Y` be a fixed-point combinator. `Y(square)=1`
  - Because `square(1)=1`
  - By definition: `square(Y(square))=Y(square)`

Fixed-Point Combinator

- A fixed point for a function is an argument whose image is itself

  `x is a fixed-point of f, if f(x)=x`

- The square function has 2 fixed points $0^2 = 0$ and $1^2 = 1$
- A fixed-point combinator finds the fixed point of a function.
- Let `Y` be a fixed-point combinator. `Y(square)=1`
  - Because `square(1)=1`
  - By definition: `square(Y(square))=Y(square)`
- In general: a fixed-point combinator is a function `Y` with the property `F(Y(F)) = Y(F)` for all functions `F`

λ-Calculus Fixed-Point Combinator I

- Define a function: `R`$\equiv$`(`λ `f | `$\langle$`body`$\rangle$`)`

λ-Calculus Fixed-Point Combinator I

- Define a function: R≡(λ f | ⟨body⟩)
- Define a fixed-point combinator

Y ≡ (λy | (λx | y (x x)) (λx | y (x x)))

λ-Calculus Fixed-Point Combinator I

- Define a function: `R≡(λ f | ⟨body⟩)`
- Define a fixed-point combinator

  `Y ≡ (λy | (λx | y (x x)) (λx | y (x x)) )`

- Apply fixed-point combinator to R to find a fixed-point

  `(Y R)≡(λy | (λx | y (x x)) (λx | y (x x)) )  R`

λ-Calculus Fixed-Point Combinator I

- Define a function: $\texttt{R} \equiv (\lambda\ \texttt{f}\ \mid\ \langle\texttt{body}\rangle)$
- Define a fixed-point combinator

  $\texttt{Y} \equiv (\lambda\texttt{y}\ \mid\ (\lambda\texttt{x}\ \mid\ \texttt{y}\ (\texttt{x}\ \texttt{x}))\ (\lambda\texttt{x}\ \mid\ \texttt{y}\ (\texttt{x}\ \texttt{x}))\ )$

- Apply fixed-point combinator to R to find a fixed-point

$$(\texttt{Y}\ \texttt{R}) \equiv (\lambda\texttt{y}\ \mid\ (\lambda\texttt{x}\ \mid\ \texttt{y}\ (\texttt{x}\ \texttt{x}))\ (\lambda\texttt{x}\ \mid\ \texttt{y}\ (\texttt{x}\ \texttt{x}))\ )\quad \texttt{R}$$
$$\xrightarrow{\beta} (\lambda\texttt{x}\ \mid\ \texttt{R}\ (\texttt{x}\ \texttt{x}))\ (\lambda\texttt{x}\ \mid\ \texttt{R}\ (\texttt{x}\ \texttt{x}))$$

λ-Calculus Fixed-Point Combinator I

- Define a function: $\texttt{R}\equiv(\lambda\ \texttt{f}\ \mid\ \langle\text{body}\rangle)$
- Define a fixed-point combinator

  $\texttt{Y}\ \equiv\ (\lambda\texttt{y}\ \mid\ (\lambda\texttt{x}\ \mid\ \texttt{y}\ (\texttt{x}\ \texttt{x}))\ (\lambda\texttt{x}\ \mid\ \texttt{y}\ (\texttt{x}\ \texttt{x}))\ )$

- Apply fixed-point combinator to R to find a fixed-point

$$(\texttt{Y}\ \texttt{R})\equiv(\lambda\texttt{y}\ \mid\ (\lambda\texttt{x}\ \mid\ \texttt{y}\ (\texttt{x}\ \texttt{x}))\ (\lambda\texttt{x}\ \mid\ \texttt{y}\ (\texttt{x}\ \texttt{x}))\ )\quad \texttt{R}$$
$$\xrightarrow{\beta}(\lambda\texttt{x}\ \mid\ \texttt{R}\ (\texttt{x}\ \texttt{x}))\ (\lambda\texttt{x}\ \mid\ \texttt{R}\ (\texttt{x}\ \texttt{x}))$$

- Denote fixed-point: $\langle\texttt{YR}\rangle\equiv(\lambda\texttt{x}\ \mid\ \texttt{R}\ (\texttt{x}\ \texttt{x}))\ (\lambda\texttt{x}\ \mid\ \texttt{R}\ (\texttt{x}\ \texttt{x}))$

λ-Calculus Fixed-Point Combinator II

- Evaluating ⟨YR⟩

⟨YR⟩≡(λ x | R (x x)) (λ x | R (x x))

λ-Calculus Fixed-Point Combinator II

- Evaluating $\langle \mathtt{YR} \rangle$

$$\langle \mathtt{YR} \rangle \equiv (\lambda\ \mathtt{x}\ |\ \mathtt{R}\ (\mathtt{x}\ \mathtt{x}))\ (\lambda\ \mathtt{x}\ |\ \mathtt{R}\ (\mathtt{x}\ \mathtt{x}))$$
$$\xrightarrow{\beta}\ \mathtt{R}\ (\ (\lambda\ \mathtt{x}\ |\ \mathtt{R}\ (\mathtt{x}\ \mathtt{x}))\ (\lambda\ \mathtt{x}\ |\ \mathtt{R}\ (\mathtt{x}\ \mathtt{x}))\ )$$

λ-Calculus Fixed-Point Combinator II

- Evaluating ⟨YR⟩

$$\langle \texttt{YR} \rangle \equiv (\lambda\ \texttt{x}\ |\ \texttt{R}\ (\texttt{x}\ \texttt{x}))\ (\lambda\ \texttt{x}\ |\ \texttt{R}\ (\texttt{x}\ \texttt{x}))$$

$$\xrightarrow{\beta}\ \texttt{R}\ (\ (\lambda\ \texttt{x}\ |\ \texttt{R}\ (\texttt{x}\ \texttt{x}))\ (\lambda\ \texttt{x}\ |\ \texttt{R}\ (\texttt{x}\ \texttt{x}))\ )$$

$$\underbrace{\texttt{R}}_{\texttt{function}}\quad \underbrace{((\lambda \texttt{x}|\texttt{R}\ (\texttt{x}\ \texttt{x}))\ (\lambda \texttt{x}|\texttt{R}\ (\texttt{x}\ \texttt{x})))}_{\texttt{self-replicating form}}$$

λ-Calculus Fixed-Point Combinator II

- Evaluating ⟨YR⟩

$$\langle \texttt{YR} \rangle \equiv (\lambda\ \texttt{x}\ |\ \texttt{R}\ (\texttt{x}\ \texttt{x}))\ (\lambda\ \texttt{x}\ |\ \texttt{R}\ (\texttt{x}\ \texttt{x}))$$

$$\xrightarrow{\beta}\ \texttt{R}\ (\ (\lambda\ \texttt{x}\ |\ \texttt{R}\ (\texttt{x}\ \texttt{x}))\ (\lambda\ \texttt{x}\ |\ \texttt{R}\ (\texttt{x}\ \texttt{x}))\ )$$

$$\underbrace{\texttt{R}}_{\texttt{function}}\quad \underbrace{((\lambda \texttt{x}|\texttt{R}\ (\texttt{x}\ \texttt{x}))\ (\lambda \texttt{x}|\texttt{R}\ (\texttt{x}\ \texttt{x})))}_{\texttt{self-replicating form}}$$

$$\equiv\ \texttt{R}\ \langle \texttt{YR} \rangle$$

λ-Calculus Fixed-Point Combinator II

- Evaluating ⟨YR⟩

$$\langle \texttt{YR} \rangle \equiv (\lambda\ \texttt{x}\ |\ \texttt{R}\ (\texttt{x}\ \texttt{x}))\ (\lambda\ \texttt{x}\ |\ \texttt{R}\ (\texttt{x}\ \texttt{x}))$$
$$\xrightarrow{\beta}\ \texttt{R}\ (\ (\lambda\ \texttt{x}\ |\ \texttt{R}\ (\texttt{x}\ \texttt{x}))\ (\lambda\ \texttt{x}\ |\ \texttt{R}\ (\texttt{x}\ \texttt{x}))\ )$$
$$\underbrace{\texttt{R}}_{\texttt{function}}\quad \underbrace{((\lambda \texttt{x}|\texttt{R}\ (\texttt{x}\ \texttt{x}))\ (\lambda \texttt{x}|\texttt{R}\ (\texttt{x}\ \texttt{x})))}_{\texttt{self-replicating form}}$$
$$\equiv\ \texttt{R}\ \langle \texttt{YR} \rangle$$

- Evaluate ⟨YR⟩ whenever we need a copy of R

λ-Calculus Fixed-Point Combinator II

- Evaluating $\langle \mathtt{YR} \rangle$

$$\langle \mathtt{YR} \rangle \equiv (\lambda\ \mathtt{x}\ |\ \mathtt{R}\ (\mathtt{x}\ \mathtt{x}))\ (\lambda\ \mathtt{x}\ |\ \mathtt{R}\ (\mathtt{x}\ \mathtt{x}))$$
$$\xrightarrow{\beta}\ \mathtt{R}\ (\ (\lambda\ \mathtt{x}\ |\ \mathtt{R}\ (\mathtt{x}\ \mathtt{x}))\ (\lambda\ \mathtt{x}\ |\ \mathtt{R}\ (\mathtt{x}\ \mathtt{x}))\ )$$
$$\underbrace{\mathtt{R}}_{\mathtt{function}}\quad \underbrace{((\lambda\mathtt{x}|\mathtt{R}\ (\mathtt{x}\ \mathtt{x}))\ (\lambda\mathtt{x}|\mathtt{R}\ (\mathtt{x}\ \mathtt{x})))}_{\mathtt{self\text{-}replicating\ form}}$$
$$\equiv\ \mathtt{R}\ \langle \mathtt{YR} \rangle$$

- Evaluate $\langle \mathtt{YR} \rangle$ whenever we need a copy of $\mathtt{R}$
- $\langle \mathtt{YR} \rangle$ is an $\mathtt{R}$ factory

Specific Fixed-Point Combinators

- Combinator we use was discovered by Haskell B. Curry

  $Y \equiv (\lambda y \mid (\lambda x \mid y\ (x\ x))\ (\lambda x \mid y\ (x\ x)))$

Specific Fixed-Point Combinators

- Combinator we use was discovered by Haskell B. Curry

$$Y \equiv (\lambda y \mid (\lambda x \mid y\ (x\ x))\ (\lambda x \mid y\ (x\ x)))$$

- Combinator discovered by Alan Turing

$$\Theta = (\lambda x|(\lambda y|(y\ (x\ x\ y)))\ (\lambda x|(\lambda y|(y\ (x\ x\ y)))))$$

Specific Fixed-Point Combinators

- Combinator we use was discovered by Haskell B. Curry

  $\texttt{Y} \equiv (\lambda\texttt{y} \mid (\lambda\texttt{x} \mid \texttt{y (x x)}) \; (\lambda\texttt{x} \mid \texttt{y (x x)}))$

- Combinator discovered by Alan Turing

  $\Theta = (\lambda\texttt{x}|(\lambda\texttt{y}|(\texttt{y (x x y)})) \; (\lambda\texttt{x}|(\lambda\texttt{y}|(\texttt{y (x x y)}))))$

- This one works for applicative order reduction

  $$\begin{aligned}\Theta_V = (\lambda\texttt{h}| \; & (\lambda\texttt{x}|(\texttt{h} \; (\lambda\texttt{y}|(\texttt{y (x x y)}))))\\ & (\lambda\texttt{x}|(\texttt{h} \; (\lambda\texttt{y}|(\texttt{y (x x y)})))) \;))\end{aligned}$$

Recursion with Haskell Combinator

- Define: `R`$\equiv$`(`λ `f y |` $\langle$`body`$\rangle$`)`

Recursion with Haskell Combinator

- Define: `R`$\equiv$`(`λ` f y | `$\langle$`body`$\rangle$`)`
- **Normal order** eval of combinated `R` to argument `m`

$\langle$`YR`$\rangle$` m    ;; what happen's when we eval `$\langle$`YR`$\rangle$`?`

Recursion with Haskell Combinator

- Define: $\texttt{R}\equiv(\lambda\ \texttt{f y}\ |\ \langle\texttt{body}\rangle)$
- **Normal order** eval of combinated R to argument m

```
⟨YR⟩ m   ;; what happen's when we eval ⟨YR⟩?
≡ R ⟨YR⟩ m
```

Recursion with Haskell Combinator

- Define: $\texttt{R} \equiv (\lambda \ \texttt{f} \ \texttt{y} \mid \langle\texttt{body}\rangle)$
- **Normal order** eval of combinated R to argument m

  $\langle\texttt{YR}\rangle$ m ;; what happen's when we eval $\langle\texttt{YR}\rangle$?
  $\equiv$ R $\langle\texttt{YR}\rangle$ m

- What's next step in Normal order reduction?

Recursion with Haskell Combinator

- Define: R≡(λ f y | ⟨body⟩)
- **Normal order** eval of combinated R to argument m

```
⟨YR⟩ m    ;; what happen's when we eval ⟨YR⟩?
≡ R ⟨YR⟩ m
```

- What's next step in Normal order reduction?
- Do leftmost apply: Apply R to its arguments

Recursion with Haskell Combinator

- Define: `R`$\equiv$`(`λ `f y | ` $\langle$`body`$\rangle$`)`
- **Normal order** eval of combinated `R` to argument `m`

  $\langle$`YR`$\rangle$ `m    ;; what happen's when we eval ` $\langle$`YR`$\rangle$`?`
  $\equiv$ `R` $\langle$`YR`$\rangle$ `m`

- What's next step in Normal order reduction?
- Do leftmost apply: Apply R to its arguments
  - R's 1^{st} arg `f` :

Recursion with Haskell Combinator

- Define: `R`$\equiv$`(`λ `f y | `$\langle$`body`$\rangle$`)`
- **Normal order** eval of combinated `R` to argument `m`

```
⟨YR⟩ m    ;; what happen's when we eval ⟨YR⟩?
≡ R ⟨YR⟩ m
```

- What's next step in Normal order reduction?
- Do leftmost apply: Apply R to its arguments
  - R's 1^{st} arg `f` : self-replicating combinated function $\langle$`YR`$\rangle$

Recursion with Haskell Combinator

- Define: `R`$\equiv$`(`λ `f y | `$\langle$`body`$\rangle$`)`
- **Normal order** eval of combinated `R` to argument `m`

```
⟨YR⟩ m    ;; what happen's when we eval ⟨YR⟩?
≡ R ⟨YR⟩ m
```

- What's next step in Normal order reduction?
- Do leftmost apply: Apply R to its arguments
  - R's 1^{st} arg `f` : self-replicating combinated function $\langle$`YR`$\rangle$
  - R's 2^{nd} arg `y` :

Recursion with Haskell Combinator

- Define: R≡(λ f y | ⟨body⟩)
- **Normal order** eval of combinated R to argument m

```
⟨YR⟩ m    ;; what happen's when we eval ⟨YR⟩?
≡ R ⟨YR⟩ m
```

- What's next step in Normal order reduction?
- Do leftmost apply: Apply R to its arguments
  - R's 1^{st} arg f : self-replicating combinated function ⟨YR⟩
  - R's 2^{nd} arg y : m

Recursion with Haskell Combinator

- Define: `R≡(λ f y | ⟨body⟩)`
- **Normal order** eval of combinated `R` to argument `m`

```
⟨YR⟩ m    ;; what happen's when we eval ⟨YR⟩?
≡ R ⟨YR⟩ m
```

- What's next step in Normal order reduction?
- Do leftmost apply: Apply R to its arguments
  - R's 1st arg `f` : self-replicating combinated function `⟨YR⟩`
  - R's 2nd arg `y` : `m`
  - R "performs calculations on" its argument `y=m`

Recursion with Haskell Combinator

- Define: `R≡(λ f y | ⟨body⟩)`
- **Normal order** eval of combinated `R` to argument `m`

```
⟨YR⟩ m    ;; what happen's when we eval ⟨YR⟩?
≡ R ⟨YR⟩ m
```

- What's next step in Normal order reduction?
- Do leftmost apply: Apply R to its arguments
  - R's 1st arg `f` : self-replicating combinated function `⟨YR⟩`
  - R's 2nd arg `y` : `m`
  - R "performs calculations on" its argument `y=m`
  - R evaluates to either:

Recursion with Haskell Combinator

- Define: `R≡(λ f y | ⟨body⟩)`
- **Normal order** eval of combinated `R` to argument `m`

```
⟨YR⟩ m    ;; what happen's when we eval ⟨YR⟩?
≡ R ⟨YR⟩ m
```

- What's next step in Normal order reduction?
- Do leftmost apply: Apply R to its arguments
  - R's 1st arg `f` : self-replicating combinated function `⟨YR⟩`
  - R's 2nd arg `y` : `m`
  - R "performs calculations on" its argument `y=m`
  - R evaluates to either:
    - a single value, say `n`, for a base case

Recursion with Haskell Combinator

- Define: `R`$\equiv$`(`λ `f y | ` $\langle$`body`$\rangle$`)`
- **Normal order** eval of combinated `R` to argument `m`

```
⟨YR⟩ m    ;; what happen's when we eval ⟨YR⟩?
≡ R ⟨YR⟩ m
```

- What's next step in Normal order reduction?
- Do leftmost apply: Apply R to its arguments
  - R's 1^{st} arg `f` : self-replicating combinated function $\langle$`YR`$\rangle$
  - R's 2^{nd} arg `y` : `m`
  - R "performs calculations on" its argument `y=m`
  - R evaluates to either:
    - a single value, say `n`, for a base case
    - "copy" of itself stored in `f` applied to a reduced value, $\langle$`YR`$\rangle$ `m'` for recursive case

Recursion Example: Non-recursive

- Ignoring f arg, what does this "sort-of" compute?

```
F = (λf n | (zerop n)        ;; T returns 1st arg
              1
              (* n (f (1- n))) )
```

Recursion Example: Non-recursive

- Ignoring `f` arg, what does this "sort-of" compute?

```
F = (λf n | (zerop n)          ;; T returns 1st arg
                1
                (* n (f (1- n))) )
```

- Basically, factorial: f(0)=1, f(1)=1, f(2)=2, f(3)=6 ...

Recursion Example: Non-recursive

- Ignoring `f` arg, what does this "sort-of" compute?

```
F = (λf n | (zerop n)        ;; T returns 1st arg
               1
               (* n (f (1- n))) )
```

- Basically, factorial: f(0)=1, f(1)=1, f(2)=2, f(3)=6 ...
- Let d be a dummy function constant:

F d 0 $\longrightarrow$

Recursion Example: Non-recursive

- Ignoring f arg, what does this “sort-of” compute?

```
F = (λf n | (zerop n)        ;; T returns 1st arg
              1
              (* n (f (1- n))) )
```

- Basically, factorial: f(0)=1, f(1)=1, f(2)=2, f(3)=6 ...
- Let d be a dummy function constant:

```
F d 0 →  1
```

Recursion Example: Non-recursive

- Ignoring f arg, what does this "sort-of" compute?

```
F = (λf n | (zerop n)          ;; T returns 1st arg
               1
               (* n (f (1- n))) )
```

- Basically, factorial: f(0)=1, f(1)=1, f(2)=2, f(3)=6 . . .
- Let d be a dummy function constant:

```
F d 0 →  1
For m>0  F d m
→
```

Recursion Example: Non-recursive

- Ignoring f arg, what does this “sort-of” compute?

```
F = (λf n | (zerop n)          ;; T returns 1st arg
                1
                (* n (f (1- n))) )
```

- Basically, factorial: f(0)=1, f(1)=1, f(2)=2, f(3)=6 ...
- Let d be a dummy function constant:

```
F d 0 →  1
For m>0  F d m
→  (* n (d (1- n)))
```

Recursion Example: Non-recursive

- Ignoring `f` arg, what does this "sort-of" compute?

```
F = (λf n | (zerop n)          ;; T returns 1st arg
              1
              (* n (f (1- n))) )
```

- Basically, factorial: f(0)=1, f(1)=1, f(2)=2, f(3)=6 ...
- Let d be a dummy function constant:

```
F d 0 →  1
For m>0  F d m
→  (* n (d (1- n)))
```

- d undefined!

Factorial Example: Base Case

```
F ≡ (λf n| (zerop n)   ;; T returns first arg
              1
              (* n (f (1- n))) )
```

Factorial Example: Base Case

```
F ≡ (λf n| (zerop n)   ;; T returns first arg
                1
                (* n (f (1- n))) )
```

- Use Haskell combinator Y to make F recursive

```
(Y F)
≡ (((λ y | (λ x | y (x x)) (λ x | y (x x)))
   (λf n|(zerop n) 1 (* n (f (1- n)))) )
```

Factorial Example: Base Case

```
F ≡ (λf n| (zerop n)   ;; T returns first arg
              1
              (* n (f (1- n))) )
```

- Use Haskell combinator Y to make F recursive

```
(Y F)
≡ (((λ y | (λ x | y (x x)) (λ x | y (x x)))
    (λf n|(zerop n) 1 (* n (f (1- n)))) )
```

$\xrightarrow{\beta}$ ⟨YF⟩   ;; long expr with 2 copies of F

Factorial Example: Base Case

```
F ≡ (λf n| (zerop n)   ;; T returns first arg
                  1
                  (* n (f (1- n))) )
```

- Use Haskell combinator Y to make F recursive

```
(Y F)
≡ (((λ y | (λ x | y (x x)) (λ x | y (x x)))
    (λf n|(zerop n) 1 (* n (f (1- n)))) )
→β ⟨YF⟩    ;; long expr with 2 copies of F
```

- Factorial example: base case

```
⟨YF⟩ 0   ;; what happens when ⟨YF⟩ is eval'd?
```

Factorial Example: Base Case

```
F ≡ (λf n| (zerop n)   ;; T returns first arg
                 1
                 (* n (f (1- n))) )
```

- Use Haskell combinator Y to make F recursive

```
(Y F)
≡ (((λ y | (λ x | y (x x)) (λ x | y (x x)))
    (λf n|(zerop n) 1 (* n (f (1- n)))) )
→β ⟨YF⟩    ;; long expr with 2 copies of F
```

- Factorial example: base case

```
⟨YF⟩ 0   ;; what happens when ⟨YF⟩ is eval'd?
→β F ⟨YF⟩ 0
```

Factorial Example: Base Case

```
F ≡ (λf n| (zerop n)   ;; T returns first arg
                 1
                 (* n (f (1- n))) )
```

- Use Haskell combinator Y to make F recursive

```
(Y F)
≡ (((λ y | (λ x | y (x x)) (λ x | y (x x)))
    (λf n|(zerop n) 1 (* n (f (1- n)))) )
→β ⟨YF⟩   ;; long expr with 2 copies of F
```

- Factorial example: base case

```
⟨YF⟩ 0  ;; what happens when ⟨YF⟩ is eval'd?
→β F ⟨YF⟩ 0
→β 1
```

Factorial Example: Recursive Case

```
F ≡ (λf n| (zerop n)   ;; T returns first arg
                1
                (* n (f (1- n))) )
```

Factorial Example: Recursive Case

```
F ≡ (λf n| (zerop n)   ;; T returns first arg
                1
                (* n (f (1- n))) )
```

- Factorial example: recursive case
  (Roughly in partly applicative order...)

  ⟨YF⟩ 1

Factorial Example: Recursive Case

```
F ≡ (λf n| (zerop n)   ;; T returns first arg
                1
                (* n (f (1- n))) )
```

- Factorial example: recursive case
  (Roughly in partly applicative order...)

$\langle$YF$\rangle$ 1
$\xrightarrow{\beta}$ F $\langle$YF$\rangle$ 1

Factorial Example: Recursive Case

```
F ≡ (λf n| (zerop n)  ;; T returns first arg
                1
                (* n (f (1- n))) )
```

- Factorial example: recursive case
  (Roughly in partly applicative order...)

$\langle$YF$\rangle$ 1

$\xrightarrow{\beta}$ F $\langle$YF$\rangle$ 1

$\xrightarrow{\beta}$ (* 1 ($\langle$YF$\rangle$ (1- 1))

Factorial Example: Recursive Case

```
F ≡ (λf n| (zerop n)   ;; T returns first arg
                1
                (* n (f (1- n))) )
```

- Factorial example: recursive case
  (Roughly in partly applicative order...)

$\langle$YF$\rangle$ 1
$\xrightarrow{\beta}$ F $\langle$YF$\rangle$ 1
$\xrightarrow{\beta}$ (* 1 ($\langle$YF$\rangle$ (1- 1))
$\xrightarrow{\beta}$ (* 1 (F $\langle$YF$\rangle$ (1- 1))

Factorial Example: Recursive Case

```
F ≡ (λf n| (zerop n)   ;; T returns first arg
                 1
                 (* n (f (1- n))) )
```

- Factorial example: recursive case
  (Roughly in partly applicative order...)

$\langle$YF$\rangle$ 1
$\xrightarrow{\beta}$ F $\langle$YF$\rangle$ 1
$\xrightarrow{\beta}$ (* 1 ($\langle$YF$\rangle$ (1- 1))
$\xrightarrow{\beta}$ (* 1 (F $\langle$YF$\rangle$ (1- 1))
$\xrightarrow{\beta}$ (* 1 1)

Factorial Example: Recursive Case

```
F ≡ (λf n| (zerop n)  ;; T returns first arg
                1
                (* n (f (1- n))) )
```

- Factorial example: recursive case
  (Roughly in partly applicative order...)

$\langle$YF$\rangle$ 1
$\xrightarrow{\beta}$ F $\langle$YF$\rangle$ 1
$\xrightarrow{\beta}$ (* 1 ($\langle$YF$\rangle$ (1- 1))
$\xrightarrow{\beta}$ (* 1 (F $\langle$YF$\rangle$ (1- 1))
$\xrightarrow{\beta}$ (* 1 1)
$\xrightarrow{\beta}$ 1

Plus Example: Recursive Solution

```
(+ [F F 0] [F 0])
  ≡ (+ [0] [F F F 0]) →[F F F 0]
```

Plus Example: Recursive Solution

```
(+ [F F 0] [F 0])
  ≡ (+ [0] [F F F 0]) →[F F F 0]

P≡(λp x y |
      (zerop x)
         y
         (p (pred x) (succ y)))
```

Plus Example: Recursive Solution

```
(+ [F F 0] [F 0])
  ≡ (+ [0] [F F F 0]) →[F F F 0]

P≡(λp x y |
      (zerop x)
         y
         (p (pred x) (succ y)))
```

$(\mathtt{Y}\ \mathtt{P}) \xrightarrow{\beta} \langle \mathtt{YP} \rangle$

Plus Example: Recursive Solution

```
(+ [F F 0] [F 0])
  ≡ (+ [0] [F F F 0]) →[F F F 0]

P≡(λp x y |
     (zerop x)
        y
        (p (pred x) (succ y)))
```

(Y P) $\stackrel{\beta}{\longrightarrow}$ ⟨YP⟩
⟨YP⟩ 1 2

Plus Example: Recursive Solution

```
(+ [F F 0] [F 0])
  ≡ (+ [0] [F F F 0]) →[F F F 0]

P≡(λp x y |
      (zerop x)
        y
        (p (pred x) (succ y)))
```

$(\texttt{Y P}) \xrightarrow{\beta} \langle \texttt{YP} \rangle$

$\langle \texttt{YP} \rangle$ 1 2

$\xrightarrow{\beta}$ P $\langle \texttt{YP} \rangle$ 1 2

Plus Example: Recursive Solution

```
(+ [F F 0] [F 0])
  ≡ (+ [0] [F F F 0]) →[F F F 0]

P≡(λp x y |
     (zerop x)
        y
        (p (pred x) (succ y)))
```

$(\texttt{Y P}) \xrightarrow{\beta} \langle \texttt{YP} \rangle$

$\langle \texttt{YP} \rangle$ 1 2

$\xrightarrow{\beta}$ P $\langle \texttt{YP} \rangle$ 1 2

$\xrightarrow{\beta}$ $\langle \texttt{YP} \rangle$ 0 3

Plus Example: Recursive Solution

```
(+ [F F 0] [F 0])
  ≡ (+ [0] [F F F 0]) →[F F F 0]

P≡(λp x y |
    (zerop x)
      y
      (p (pred x) (succ y)))
```

$(\texttt{Y P}) \xrightarrow{\beta} \langle \texttt{YP} \rangle$

$\langle \texttt{YP} \rangle$ 1 2

$\xrightarrow{\beta}$ P $\langle \texttt{YP} \rangle$ 1 2

$\xrightarrow{\beta}$ $\langle \texttt{YP} \rangle$ 0 3

$\xrightarrow{\beta}$ P $\langle \texttt{YP} \rangle$ 0 3

Plus Example: Recursive Solution

```
(+ [F F 0] [F 0])
  ≡ (+ [0] [F F F 0]) →[F F F 0]

P≡(λp x y |
      (zerop x)
         y
         (p (pred x) (succ y)))
```

$(\text{Y P}) \xrightarrow{\beta} \langle \text{YP} \rangle$

$\langle \text{YP} \rangle$ 1 2

$\xrightarrow{\beta}$ P $\langle \text{YP} \rangle$ 1 2

$\xrightarrow{\beta}$ $\langle \text{YP} \rangle$ 0 3

$\xrightarrow{\beta}$ P $\langle \text{YP} \rangle$ 0 3 $\xrightarrow{\beta}$ 3

Summation Example: Recursive Solution

`Key idiom: sum(n)=n+sum(n-1)`

Summation Example: Recursive Solution

```
Key idiom: sum(n)=n+sum(n-1)

S≡(λs n | (zerop n) 0 (+ n (s (1- n))))
```

Summation Example: Recursive Solution

`Key idiom: sum(n)=n+sum(n-1)`

S$\equiv$(λs n | (zerop n) 0 (+ n (s (1- n))))

(Y S)$\xrightarrow{\beta}$$\langle$YS$\rangle$

Summation Example: Recursive Solution

Key idiom: sum(n)=n+sum(n-1)

$S \equiv (\lambda$s n | (zerop n) 0 (+ n (s (1- n))))

(Y S) $\xrightarrow{\beta}$ $\langle$YS$\rangle$

$\langle$YS$\rangle$ 1

Summation Example: Recursive Solution

`Key idiom: sum(n)=n+sum(n-1)`

$S \equiv (\lambda s\ n \mid (zerop\ n)\ 0\ (+\ n\ (s\ (1-\ n))))$

$(Y\ S) \xrightarrow{\beta} \langle YS \rangle$

$\langle YS \rangle\ 1$

$\xrightarrow{\beta}\ S\ \langle YS \rangle\ 1$

Summation Example: Recursive Solution

Key idiom: sum(n)=n+sum(n-1)

S$\equiv$(λs n | (zerop n) 0 (+ n (s (1- n))))

(Y S)$\xrightarrow{\beta}\langle$YS$\rangle$

$\langle$YS$\rangle$ 1

$\xrightarrow{\beta}$ S $\langle$YS$\rangle$ 1

$\xrightarrow{\beta}$ (+ 1 ($\langle$YS$\rangle$ (1- 1)))

Summation Example: Recursive Solution

Key idiom: sum(n)=n+sum(n-1)

$\texttt{S}\equiv(\lambda\texttt{s n | (zerop n) 0 (+ n (s (1- n))))}$

$\texttt{(Y S)}\xrightarrow{\beta}\langle\texttt{YS}\rangle$

$\langle\texttt{YS}\rangle\ \texttt{1}$

$\xrightarrow{\beta}\ \texttt{S}\ \langle\texttt{YS}\rangle\ \texttt{1}$

$\xrightarrow{\beta}\ \texttt{(+ 1 (}\langle\texttt{YS}\rangle\ \texttt{(1- 1)))}$

$\xrightarrow{\beta}\ \texttt{(+ 1 (S}\ \langle\texttt{YS}\rangle\ \texttt{0))}$

Summation Example: Recursive Solution

Key idiom: sum(n)=n+sum(n-1)

S$\equiv$(λs n | (zerop n) 0 (+ n (s (1- n))))

(Y S)$\xrightarrow{\beta}\langle$YS$\rangle$

$\langle$YS$\rangle$ 1

$\xrightarrow{\beta}$ S $\langle$YS$\rangle$ 1

$\xrightarrow{\beta}$ (+ 1 ($\langle$YS$\rangle$ (1- 1)))

$\xrightarrow{\beta}$ (+ 1 (S $\langle$YS$\rangle$ 0))

$\xrightarrow{\beta}$ (+ 1 0) $\xrightarrow{\beta}$ 1