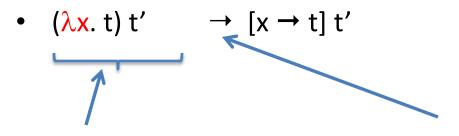
#### Lambda Calculus

CSE 340 FALL 2021 Rida Bazzi

Part of the presentation is adopted from types and Programming languages by Benjamin C. Pierce, MIT Press

#### **Operational Semantics**



Redex: Reducible expression

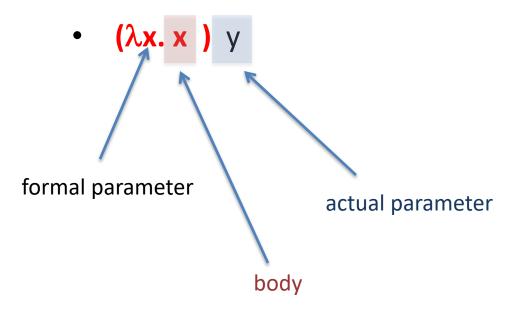
**β-reduction** 

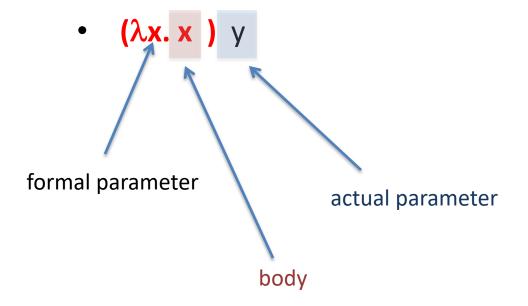
- The answer is obtained by
  - Replacing all bound occurrences of x in  $\lambda x$ . t with t'. Here we are interested in occurrences bound by the outer  $\lambda x$
  - Getting rid of the  $\lambda x$
  - Getting rid of the original t'

t' is treated as a unit and you might need to add parentheses when you do the substitution if not adding parenthesis does not keep it as a unit (always safe to add parentheses)

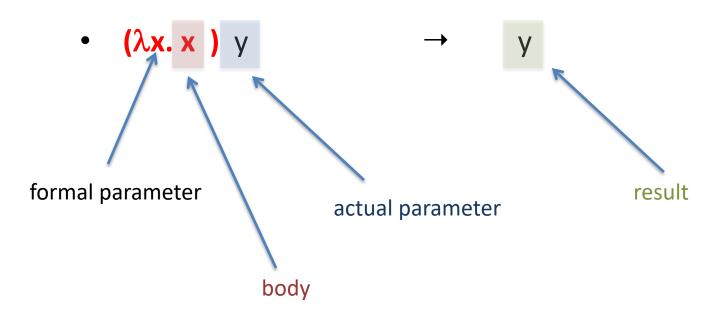
In the following examples the bound occurrences of the variable are colored with the same color as the abstraction that binds them

```
• (λx. x ) y
```



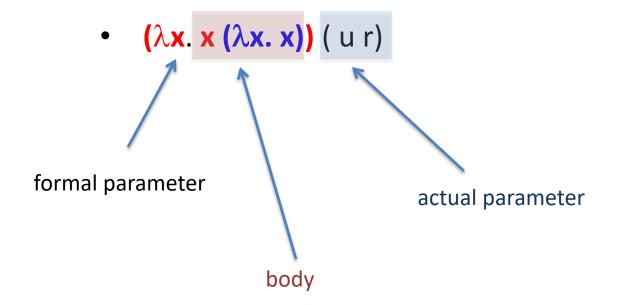


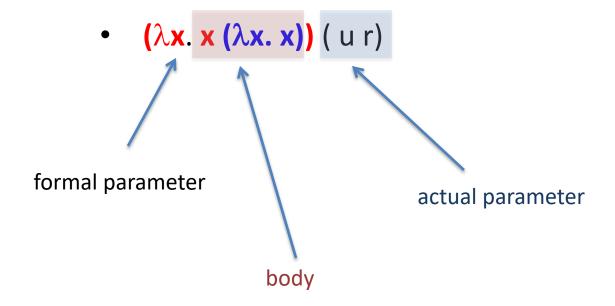
we should replace the x in the body with y



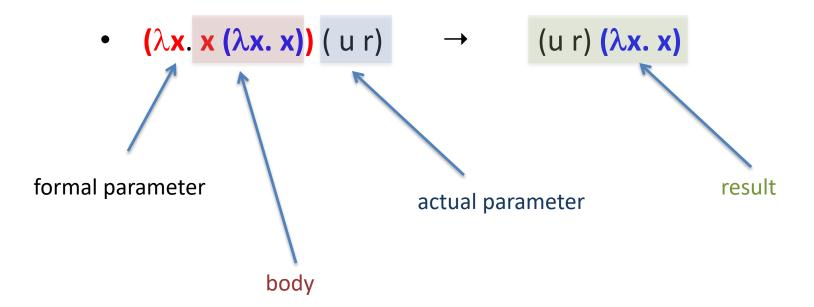
we should replace the x in the body with y

• (λx. x (λx. x)) ( u r)

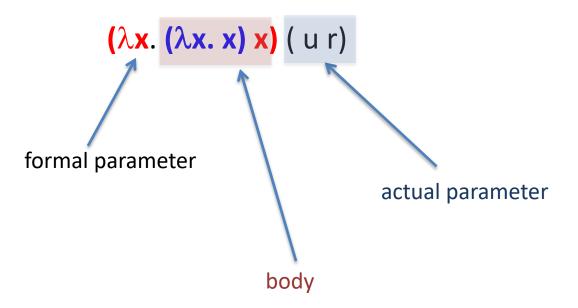


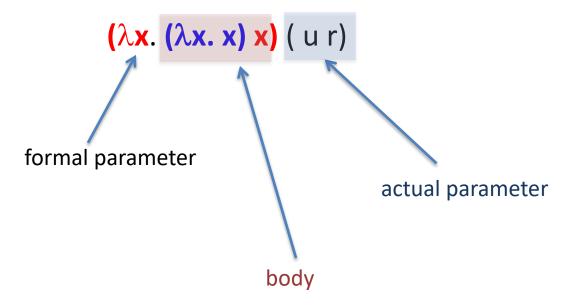


we should replace the x in the body with (u r)

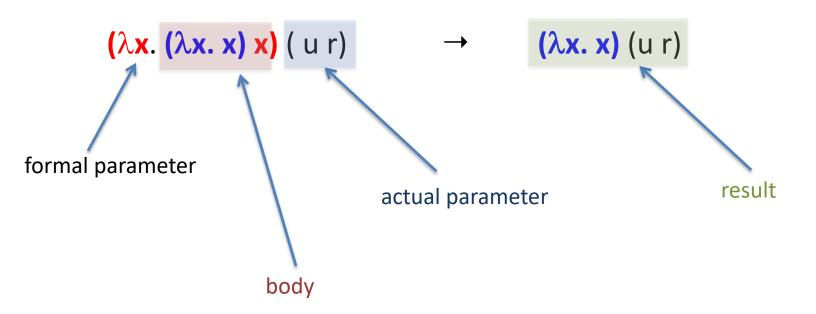


```
(\lambda x. (\lambda x. x) x) (ur)
```

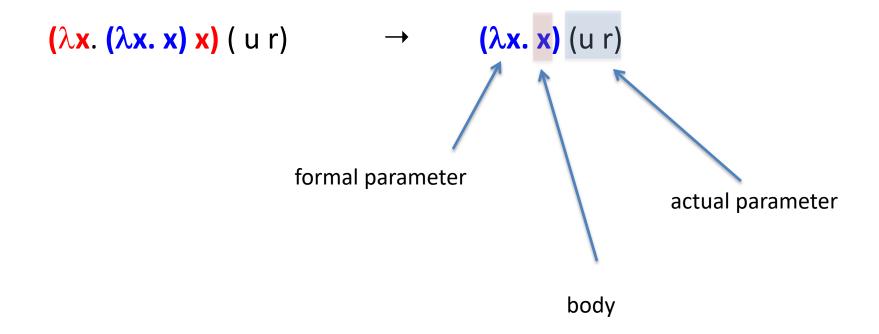


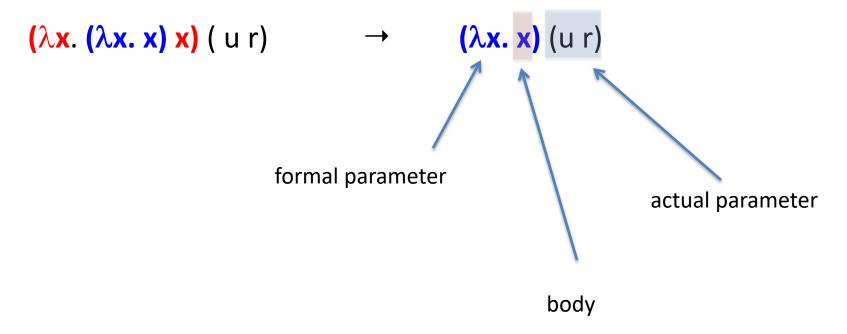


we should replace the x in the body with (u r)

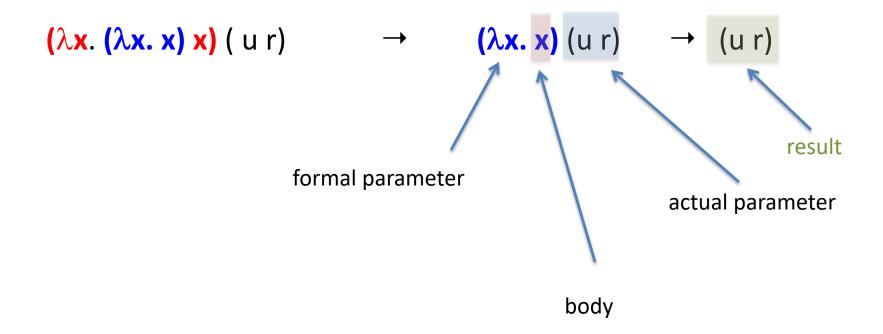


```
(\lambda x. (\lambda x. x) x) (ur) \rightarrow (\lambda x. x) (ur)
```





we should replace the x in the body with (u r)



```
#include <stdio.h>
\#define INC(x) {int a = 0; x++;}
int main()
        int x = 3;
        int a = 3;
        INC(x);
        INC(a);
        printf("x = %d a = %d\n", x, a);
```

```
#include <stdio.h>
#define INC(x) {int a = 0; x++;}
int main()
        int x = 3;
        int a = 3;
        INC(x);
                  // \{ int a = 0; a++; \}
        INC(a);
        printf("x = %d a = %d\n", x, a);
```

```
#include <stdio.h>
#define INC(x) {int a = 0; x++;}
int main()
        int x = 3;
        int a = 3;
        INC(x);
                  // \{ int a = 0; a++; \}
        INC(a);
       printf("x = %d a = %d\n", x, a);
Output: x = 4 a = 3
```

```
int f(int x)
{    int j;
    j = 0;
    return j + x;
}
```

```
int f(int x)
{    int j;
    j = 0;
    return j + x;
}
```

```
int f(int x)
{    int j;
    j = 0;
    return j + x;
}
```

```
{ int j ; j = 0; result = j+3; }
```

```
int f(int x)
{    int j;
    j = 0;
    return j + x;
}
```

```
{ int j; j = 0; result = j+x;)
```

```
int f(int x)
{    int j;
    j = 0;
    return j + x;
}
```

```
{ int \mathbf{j}'; \mathbf{j} = 0; result = \mathbf{j}+\mathbf{j};)
```

```
int f(int x)
{    int j;
    j = 0;
    return j + x;
}
```

Result should not depend on name of local variable

```
f(j) \equiv result
```

```
{ int j; j = 0; result = j+i;
```

## Hygienic macro evaluation

```
int f(int x)
{    int j;
    j = 0;
    return j + x;
}
```

Result should not depend on name of local variable

```
f(j) \equiv result
```

```
{ int \mathbf{i} ; \mathbf{i} = 0; result = \mathbf{i}+\mathbf{j};)
```

## Renaming

 If the call is done by textual substitution, we should make sure that local names are changed so that they do not clash with the actual parameters

This one is problematic because if we do the replacement, we will bring the free variable y into a **scope** in which there is a local y (the **green** abstraction on y).

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

We should rename before we do the beta reduction:

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

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

We should rename before we do the beta reduction:

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

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

We should rename before we do the beta reduction:

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

Without renaming, we get the incorrect answer: z z

#### **Evaluation Strategies: Normal Order**

Leftmost, outermost redex is always reduced first

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

 Note. outermost is outermost in scope nesting level

## Normal order examples

 The following are terms with the leftmost outermost redex underlined in red

```
-((\lambda x. (\lambda x. x) x) (ur)) ((\lambda x. (\lambda x. x) x) (ur))
```

# Normal order examples

 The following are expressions with the leftmost outermost redex underlined in red

$$-((\lambda x. (\lambda x. x) x) (ur)) ((\lambda x. (\lambda x. x) x) (ur))$$

$$-((\lambda x. (\lambda x. x) x) (u r))$$

#### Evaluation Strategies: Call by value

- Only outermost redexes are reduced (no reduction inside abstractions)
- A redex is reduced only when its right-hand side has reduced to a value
- A value is an expressions that does not have a redex or an expression in which all redexes are under abstraction
- Another way to say it is that an expression is not a value if it has at least one redex that is not under an abstraction

**Note**: under abstraction means inside the body of an abstraction

## Omega

• Omega =  $(\lambda x. x x) (\lambda x. x x)$ 

### Omega

• Omega =  $(\lambda x. x x) (\lambda x. x x)$ 

$$(\lambda x. x x) (\lambda x. x x) \rightarrow (\lambda x. x x) (\lambda x. x x)$$

### Omega

• Omega =  $(\lambda x. x x) (\lambda x. x x)$ 

$$(\lambda x. x x) (\lambda x. x x) \rightarrow (\lambda x. x x) (\lambda x. x x)$$

Omega never reduces to a value

We want to compare the evaluation of the term  $(\lambda x.$ a) Omega using call by value and normal order evaluation

1. Call by value

 $(\lambda x. a)$  Omega  $\rightarrow$ 

We want to compare the evaluation of the term

(λx. a) Omega

using call by value and normal order evaluation

1. Call by value

 $(\lambda x. a)$  Omega  $\rightarrow (\lambda x. a)$  Omega

We want to compare the evaluation of the term

(λx. a) Omega

using call by value and normal order evaluation

1. Call by value

 $(\lambda x. a)$  Omega  $\rightarrow (\lambda x. a)$  Omega  $\rightarrow (\lambda x. a)$  Omega

We want to compare the evaluation of the term

using call by value and normal order evaluation

1. Call by value

$$(\lambda x. a)$$
 Omega  $\rightarrow (\lambda x. a)$  Omega  $\rightarrow (\lambda x. a)$  Omega  $\rightarrow ...$ 

We want to compare the evaluation of the term

using call by value and normal order evaluation

1. Call by value

$$(\lambda x. a)$$
 Omega  $\rightarrow (\lambda x. a)$  Omega  $\rightarrow (\lambda x. a)$  Omega  $\rightarrow ...$ 

2. Normal order evaluation

We want to compare the evaluation of the term

using call by value and normal order evaluation

1. Call by value

$$(\lambda x. a)$$
 Omega  $\rightarrow (\lambda x. a)$  Omega  $\rightarrow (\lambda x. a)$  Omega  $\rightarrow ...$ 

2. Normal order evaluation

$$(\lambda x. a)$$
 Omega  $\rightarrow a$ 

- Function of multiple parameters
  - $-(\lambda(x,y). s) (v,w)$

Function of multiple parameters

$$-(\lambda(x,y). s) (v,w)$$

We would like to get  $[y \rightarrow w][x \rightarrow v] s$  as a result of the application

Function of multiple parameters

$$-(\lambda(x,y). s) (v,w)$$

We would like to get  $[y \rightarrow w][x \rightarrow v] s$  as a result of the application

Function of multiple parameters

$$-(\lambda(x,y). s) (v,w)$$

We would like to get  $[y \rightarrow w][x \rightarrow v] s$  as a result of the application

Function of multiple parameters

$$-(\lambda(x,y). s) (v,w)$$

We would like to get  $[y \rightarrow w][x \rightarrow v] s$  as a result of the application

Note that this reads as follows: replace x with v in s, then replace y with w in  $[x \rightarrow v]$  s

Same effect can be achieved by higher order functions



Haskell Curry

- Same effect can be achieved by higher order functions
  - $-\lambda x$ .  $\lambda y$ . s is a function of x.

- Same effect can be achieved by higher order functions
  - $-\lambda x$ .  $\lambda y$ . s is a function of x.
  - If we apply it to v we get  $\lambda y$ . [x  $\rightarrow$  v] s which is a function of y.

- Same effect can be achieved by higher order functions
  - $-\lambda x$ .  $\lambda y$ . s is a function of x.
  - If we apply it to v we get  $\lambda y$ . [x  $\rightarrow$  v] s which is a function of y.
  - If we apply the result to w, we get  $[y \rightarrow w][x \rightarrow v]$  s

- Same effect can be achieved by higher order functions
  - $-\lambda x$ .  $\lambda y$ . s is a function of x.
  - If we apply it to v we get  $\lambda y$ . [x  $\rightarrow$  v] s which is a function of y.
  - If we apply the result to w, we get  $[y \rightarrow w][x \rightarrow v]$  s

Exactly what we needed

- Same effect can be achieved by higher order functions
  - $-\lambda x$ .  $\lambda y$ . s is a function of x.
  - If we apply it to v we get  $\lambda y$ . [x  $\rightarrow$  v] s which is a function of y.
  - If we apply the result to, we get  $[y \rightarrow w][x \rightarrow v] s$

 <u>Currying</u>: transformation of multi-argument function to higher-order function

We want to define boolean values that can be tested:

• test b v w =

We want to define boolean values that can be tested:

- test b v w =
  - v when b is tru

We want to define boolean values that can be tested:

- test b v w =
  - v when b is tru
  - w when b is fls

We want to define boolean values that can be tested:

- test b v w =
  - v when b is tru
  - w when b is fls

 test takes three parameters: a boolean condition and two choices. Depending on the condition one of the choices is returned.

• tru =  $\lambda t$ .  $\lambda f$ . t

• tru =  $\lambda t$ .  $\lambda f$ . t

tru  $a = (\lambda t. \lambda f. t) a$ 

• tru =  $\lambda t$ .  $\lambda f$ . t

tru a =  $(\lambda t. \lambda f. t)$  a  $\rightarrow \lambda f.$  a

• tru =  $\lambda t$ .  $\lambda f$ . t

tru a =  $(\lambda t. \lambda f. t)$  a  $\rightarrow \lambda f.$  a tru a b =  $((\lambda t. \lambda f. t)$  a) b

• tru =  $\lambda t$ .  $\lambda f$ . t

tru a =  $(\lambda t. \lambda f. t)$  a  $\rightarrow \lambda f.$  a tru a b =  $((\lambda t. \lambda f. t)$  a) b  $\rightarrow (\lambda f. a)$  b

• tru =  $\lambda t$ .  $\lambda f$ . t

tru a =  $(\lambda t. \lambda f. t)$  a  $\rightarrow \lambda f.$  a tru a b =  $((\lambda t. \lambda f. t)$  a) b  $\rightarrow (\lambda f. a)$  b  $\rightarrow$  a

• tru =  $\lambda t$ .  $\lambda f$ . t

tru a = 
$$(\lambda t. \lambda f. t)$$
 a  $\rightarrow \lambda f.$  a  
tru a b =  $((\lambda t. \lambda f. t)$  a) b  $\rightarrow (\lambda f. a)$  b  $\rightarrow$  a

tru applied to two parameters returns the first one

• fls =  $\lambda t$ .  $\lambda f$ . f

• fls =  $\lambda t$ .  $\lambda f$ . f

fls a =  $(\lambda t. \lambda f. f)$  a

• fls =  $\lambda t$ .  $\lambda f$ . f

fls a =  $(\lambda t. \lambda f. f)$  a  $\rightarrow \lambda f. f$ 

• fls =  $\lambda t$ .  $\lambda f$ . f

fls a =  $(\lambda t. \lambda f. f)$  a  $\rightarrow \lambda f. f$ fls a b =  $((\lambda t. \lambda f. f)$  a) b

### fls

• fls =  $\lambda t$ .  $\lambda f$ . f

```
fls a = (\lambda t. \lambda f. f) a \rightarrow \lambda f. f
fls a b = ((\lambda t. \lambda f. f) a) b \rightarrow (\lambda f. f) b
```

#### fls

• fls =  $\lambda t$ .  $\lambda f$ . f

```
fls a = (\lambda t. \lambda f. f) a \rightarrow \lambda f. f
fls a b = ((\lambda t. \lambda f. f) a) b \rightarrow (\lambda f. f) b \rightarrow b
```

#### fls

• fls =  $\lambda t$ .  $\lambda f$ . f

fls a = 
$$(\lambda t. \lambda f. f)$$
 a  $\rightarrow \lambda f. f$   
fls a b =  $((\lambda t. \lambda f. f)$  a) b  $\rightarrow (\lambda f. f)$  b  $\rightarrow$  b

fls applied to two parameters returns the second one

test tru v w =

• test tru v w =  $(\lambda l. \lambda m. \lambda n. l m n)$  tru v w

• test tru v w =  $\frac{(\lambda l. \lambda m. \lambda n. l m n) tru}{(\lambda m. \lambda n. tru m n) v} w$ 

- test tru v w =  $(\lambda l. \lambda m. \lambda n. l m n)$  tru v w
  - $\rightarrow$  ( $\lambda$ m.  $\lambda$ n. tru m n) v w
  - $\rightarrow$  ( $\lambda$ n. tru v n) w

- test tru v w =  $(\lambda l. \lambda m. \lambda n. l m n)$  tru v w
  - $\rightarrow$  ( $\lambda$ m.  $\lambda$ n. tru m n) v w
  - $\rightarrow$  ( $\lambda$ n. tru v n) w
  - → tru v w

- test tru v w =  $(\lambda l. \lambda m. \lambda n. l m n)$  tru v w
  - $\rightarrow$  ( $\lambda$ m.  $\lambda$ n. tru m n) v w
  - $\rightarrow$  ( $\lambda$ n. tru v n) w
  - → tru v w
  - $\rightarrow$  ( $\lambda t$ .  $\lambda f$ . t) v w

- test tru v w =  $(\lambda I. \lambda m. \lambda n. I m n)$  tru v w
  - $\rightarrow$  ( $\lambda$ m.  $\lambda$ n. tru m n) v w
  - $\rightarrow$  ( $\lambda$ n. tru v n) w
  - → tru v w
  - $\rightarrow$  ( $\lambda t$ .  $\lambda f$ . t) v w
  - $\rightarrow$  ( $\lambda f. v) w$

- test tru v w =  $(\lambda l. \lambda m. \lambda n. l m n)$  tru v w
  - $\rightarrow$  ( $\lambda$ m.  $\lambda$ n. tru m n) v w
  - $\rightarrow$  ( $\lambda$ n. tru v n) w
  - → tru v w
  - $\rightarrow$  ( $\lambda t$ .  $\lambda f$ . t) v w
  - $\rightarrow$  ( $\lambda f. v) w$
  - → V

• test tru v w =  $\frac{(\lambda l. \lambda m. \lambda n. l m n) tru}{(\lambda l. \lambda m. \lambda n. l m n) tru} v w$   $\rightarrow \frac{(\lambda m. \lambda n. tru m n) v}{(\lambda n. tru v n) w}$   $\rightarrow \frac{(\lambda n. tru v n) w}{(\lambda t. v w)}$   $\rightarrow \frac{(\lambda t. \lambda f. t) v}{(\lambda f. v) w}$   $\rightarrow v$ 

Test places the boolean value in front of the two other parameters and the boolean value does the work!

• and =  $\lambda$ b.  $\lambda$ c. b c fls

- and =  $\lambda$ b.  $\lambda$ c. b c fls
- and tru tru

• and =  $\lambda$ b.  $\lambda$ c. b c fls

and tru tru
 (λb. λc. b c fls) tru tru

- and =  $\lambda$ b.  $\lambda$ c. b c fls
- and tru tru
   (λb. λc. b c fls) tru tru
  - $\rightarrow$  ( $\lambda$ c. tru c fls) tru

- and =  $\lambda$ b.  $\lambda$ c. b c fls
- and tru tru  $(\lambda b. \lambda c. b c fls)$  tru tru
  - $\rightarrow$  ( $\lambda$ c. tru c fls) tru
  - → tru tru fls

- and =  $\lambda$ b.  $\lambda$ c. b c fls
- and tru tru
   (λb. λc. b c fls) tru tru
  - $\rightarrow$  ( $\lambda$ c. tru c fls) tru
  - → tru tru fls
  - → (tru tru) fls

- and =  $\lambda$ b.  $\lambda$ c. b c fls
- and tru tru
   (λb. λc. b c fls) tru tru
  - $\rightarrow$  ( $\lambda$ c. tru c fls) tru
  - → tru tru fls
  - → (tru tru) fls
  - $\rightarrow$  (( $\lambda t$ .  $\lambda f$ . t) tru) fls

- and =  $\lambda$ b.  $\lambda$ c. b c fls
- and tru tru
   (λb. λc. b c fls) tru tru
  - $\rightarrow$  ( $\lambda$ c. tru c fls) tru
  - → tru tru fls
  - → (tru tru) fls
  - $\rightarrow$  (( $\lambda t$ .  $\lambda f$ . t) tru) fls
  - $\rightarrow$  ( $\lambda f. tru$ ) fls

- and =  $\lambda$ b.  $\lambda$ c. b c fls
- and tru tru
   (λb. λc. b c fls) tru tru
  - $\rightarrow$  ( $\lambda$ c. tru c fls) tru
  - → tru tru fls
  - → (tru tru) fls
  - $\rightarrow$  (( $\lambda t$ .  $\lambda f$ . t) tru) fls
  - $\rightarrow$  ( $\lambda f. tru$ ) fls
  - → tru

• or =  $\lambda$ b.  $\lambda$ c. b tru c

- or =  $\lambda$ b.  $\lambda$ c. b tru c
- or fls tru

• or =  $\lambda b$ .  $\lambda c$ . b tru c

• or fls tru  $(\lambda b. \lambda c. b tru c)$  fls tru

- or =  $\lambda$ b.  $\lambda$ c. b tru c
- or fls tru  $(\lambda b. \lambda c. b tru c)$  fls tru
  - $\rightarrow$  ( $\lambda$ c. fls tru c) tru

- or =  $\lambda$ b.  $\lambda$ c. b tru c
- or fls tru  $(\lambda b. \lambda c. b tru c)$  fls tru
  - $\rightarrow$  ( $\lambda$ c. fls tru c) tru
  - → fls tru tru

- or =  $\lambda$ b.  $\lambda$ c. b tru c
- or fls tru
  - $(\lambda b. \lambda c. b tru c)$  fls tru
  - $\rightarrow$  ( $\lambda$ c. fls tru c) tru
  - → fls tru tru
  - → (fls tru) tru

- or =  $\lambda$ b.  $\lambda$ c. b tru c
- or fls tru
  - $(\lambda b. \lambda c. b tru c)$  fls tru
  - $\rightarrow$  ( $\lambda$ c. fls tru c) tru
  - → fls tru tru
  - → (fls tru) tru
  - $\rightarrow$  (( $\lambda t. \lambda f. f$ ) tru) tru

- or =  $\lambda$ b.  $\lambda$ c. b tru c
- or fls tru
  - $(\lambda b. \lambda c. b tru c)$  fls tru
  - $\rightarrow$  ( $\lambda$ c. fls tru c) tru
  - → fls tru tru
  - → (fls tru) tru
  - $\rightarrow$  (( $\lambda t. \lambda f. f$ ) tru) tru
  - $\rightarrow$  ( $\lambda f$ .  $\lambda f$ ) tru

- or =  $\lambda$ b.  $\lambda$ c. b tru c
- or fls tru
   (λb. λc. b tru c) fls tru
  - $\rightarrow$  ( $\lambda$ c. fls tru c) tru
  - → fls tru tru
  - → (fls tru) tru
  - $\rightarrow$  (( $\lambda t. \lambda f. f$ ) tru) tru
  - $\rightarrow$  ( $\lambda f$ .  $\lambda f$ ) tru
  - → tru

- pair =  $\lambda f. \lambda s. \lambda b. b f s$
- fst =  $\lambda p$ . p tru
- snd =  $\lambda p. p fls$

- pair =  $\lambda f. \lambda s. \lambda b. b f s$
- fst =  $\lambda p. p tru$
- snd=  $\lambda p$ . p fls

pair v w =  $(\lambda f. \lambda s. \lambda b. b f s)$  v w

- pair =  $\lambda f. \lambda s. \lambda b. b f s$
- fst =  $\lambda p. p tru$
- snd=  $\lambda p$ . p fls

```
pair v w = (\lambda f. \lambda s. \lambda b. b f s) v w \rightarrow (\lambda s. \lambda b. b v s) w
```

- pair =  $\lambda f. \lambda s. \lambda b. b f s$
- fst =  $\lambda p$ . p tru
- snd=  $\lambda p$ . p fls
- pair v w =  $(\lambda f. \lambda s. \lambda b. b f s)$  v w
  - $\rightarrow$  ( $\lambda$ s.  $\lambda$ b. b v s) w
  - $\rightarrow \lambda b. b v w$

- pair =  $\lambda f. \lambda s. \lambda b. b f s$
- fst =  $\lambda p$ . p tru
- snd=  $\lambda p$ . p fls

pair v w =  $(\lambda f. \lambda s. \lambda b. b f s)$  v w

- $\rightarrow$  ( $\lambda$ s.  $\lambda$ b. b v s) w
- $\rightarrow \lambda b. b v w$

pair applied to v and w yields an abstraction containing v and w

### Extracting the first element from a pair

- fst (pair v w)
- = fst  $((\lambda f. \lambda s. \lambda b. b f s) v w)$
- $\rightarrow$  fst  $((\underline{\lambda s. \lambda b. b v s) w})$
- $\rightarrow$  fst ( $\lambda b$ . b v w)

```
fst (pair v w)
          fst ((\lambda f. \lambda s. \lambda b. b f s) v w)
         fst ((\lambda s. \lambda b. b v s) w)
      fst (\lambda b. bvw)
        (\lambda p. p tru) (\lambda b. b v w)
 formal parameter
                       body
                                                actual parameter
```

```
fst (pair v w)
          fst ((\lambda f. \lambda s. \lambda b. b f s) v w)
         fst ((\lambda s. \lambda b. b v s) w)
      fst (\lambdab. b v w)
        (\lambda p. p tru) (\lambda b. b v w)
 formal parameter
                       body
                                                 actual parameter
```

We should replace **p** with the actual parameter ( $\lambda b$ . b v w)

```
    fst (pair v w)
    fst ((λf. λs. λb. b f s) v w)
    fst ((λs. λb. b v s) w)
    fst (λb. b v w)
    (λp. p tru) (λb. b v w)
    (λb. b v w) tru
```

```
fst (pair v w)
         fst ((\lambda f. \lambda s. \lambda b. b f s) v w)
        fst ((\lambda s. \lambda b. b v s) w)
\rightarrow fst (\lambdab. b v w)
        (\lambda p. p tru) (\lambda b. b v w)
         (λb. b v w) tru
         tru v w
         V
```

#### **Church Numerals**

- $0 = \lambda s. \lambda z. z$
- $1 = \lambda s. \lambda z. s z$
- $2 = \lambda s. \lambda z. s (s z)$
- $3 = \lambda s. \lambda z. s (s (s z))$
- •

• 0 a b =  $(\lambda s. \lambda z. z)$  a b

• 0 a b =  $(\lambda s. \lambda z. z)$  a b  $\rightarrow (\lambda z. z)$  b

• 0 a b =  $(\lambda s. \lambda z. z)$  a b  $\rightarrow$   $(\lambda z. z)$  b  $\rightarrow$  b

- 0 a b =  $(\lambda s. \lambda z. z)$  a b  $\rightarrow$   $(\lambda z. z)$  b  $\rightarrow$  b
- 1 a b =  $(\lambda s. \lambda z. sz)$  a b

- 0 a b =  $(\lambda s. \lambda z. z)$  a b  $\rightarrow$   $(\lambda z. z)$  b  $\rightarrow$  b
- 1 a b =  $(\lambda s. \lambda z. sz)$  a b  $\rightarrow$   $(\lambda z. az)$  b

- 0 a b =  $(\lambda s. \lambda z. z)$  a b  $\rightarrow$   $(\lambda z. z)$  b  $\rightarrow$  b
- 1 a b =  $(\lambda s. \lambda z. sz)$  a b  $\rightarrow$   $(\lambda z. az)$  b  $\rightarrow$  a b

- 0 a b =  $(\lambda s. \lambda z. z)$  a b  $\rightarrow$   $(\lambda z. z)$  b  $\rightarrow$  b
- 1 a b =  $(\lambda s. \lambda z. sz)$  a b  $\rightarrow$   $(\lambda z. az)$  b  $\rightarrow$  a b
- •
- 3 a b =  $(\lambda s. \lambda z. s (s (s z)))$  a b

- 0 a b =  $(\lambda s. \lambda z. z)$  a b  $\rightarrow$   $(\lambda z. z)$  b  $\rightarrow$  b
- 1 a b =  $(\lambda s. \lambda z. sz)$  a b  $\rightarrow$   $(\lambda z. az)$  b  $\rightarrow$  a b
- •
- 3 a b =  $(\lambda s. \lambda z. s (s (s z)))$  a b  $\rightarrow (\lambda z. a (a (a z)))$  b

- 0 a b =  $(\lambda s. \lambda z. z)$  a b  $\rightarrow$   $(\lambda z. z)$  b  $\rightarrow$  b
- 1 a b =  $(\lambda s. \lambda z. sz)$  a b  $\rightarrow$   $(\lambda z. az)$  b  $\rightarrow$  a b
- •
- 3 a b =  $(\lambda s. \lambda z. s (s (s z)))$  a b
  - $\rightarrow$  ( $\lambda z$ . a (a (a z))) b
  - $\rightarrow$  a (a (a b)))

#### **Arithmetic: Successor Function**

•  $scc = \lambda n. \lambda s. \lambda z. s$  (n s z) Apply n to function s (successor) and z (zero) and apply s one more time

•  $scc 2 = (\lambda n. \lambda s. \lambda z. s (n s z)) 2$ 

• 
$$scc 2 = (\lambda n. \lambda s. \lambda z. s (n s z)) 2$$
  
 $\rightarrow (\lambda s. \lambda z. s (2 s z))$ 

•  $scc 2 = (\lambda n. \lambda s. \lambda z. s (n s z)) 2$   $\rightarrow (\lambda s. \lambda z. s (2 s z))$  $\rightarrow (\lambda s. \lambda z. s (s (s z))) = 3$ 

•  $scc 4 = (\lambda n. \lambda s. \lambda z. s (n s z)) 4$ 

• 
$$scc 4 = (\lambda n. \lambda s. \lambda z. s (n s z)) 4$$
  
 $\rightarrow (\lambda s. \lambda z. s (4 s z))$ 

```
• scc 4 = (\lambda n. \lambda s. \lambda z. s (n s z)) 4

\rightarrow (\lambda s. \lambda z. s (4 s z))

\rightarrow (\lambda s. \lambda z. s (s (s (s z)))))
```

•  $scc 4 = (\lambda n. \lambda s. \lambda z. s (n s z)) 4$   $\rightarrow (\lambda s. \lambda z. s (4 s z))$  $\rightarrow (\lambda s. \lambda z. s (s (s (s z)))) = 5$ 

#### **Arithmetic: Addition**

• plus =  $\lambda$ m.  $\lambda$ n.  $\lambda$ s.  $\lambda$ z. m s (n s z) Apply m to the function s and the result of applying n to s and z

• plus 3 4=  $(\lambda m. \lambda n. \lambda s. \lambda z. m s (n s z))$  3 4

• plus 3 4=  $(\lambda m. \lambda n. \lambda s. \lambda z. m s (n s z))$  3 4  $\rightarrow (\lambda n. \lambda s. \lambda z. 3 s (n s z))$  4

- plus 3 4=  $(\lambda m. \lambda n. \lambda s. \lambda z. m s (n s z))$  3 4
  - $\rightarrow$  ( $\lambda$ n.  $\lambda$ s.  $\lambda$ z. 3 s (n s z)) 4
  - $\rightarrow \lambda s. \lambda z. 3 s (4 s z)$

- plus 3 4=  $(\lambda m. \lambda n. \lambda s. \lambda z. m s (n s z))$  3 4
  - $\rightarrow$  ( $\lambda$ n.  $\lambda$ s.  $\lambda$ z. 3 s (n s z)) 4
  - $\rightarrow \lambda s. \lambda z. 3 s (4 s z)$
  - $\rightarrow \lambda s. \lambda z. 3 s (s (s (s (s z)))$

- plus 3 4=  $(\lambda m. \lambda n. \lambda s. \lambda z. m s (n s z))$  3 4
  - $\rightarrow$  ( $\lambda$ n.  $\lambda$ s.  $\lambda$ z. 3 s (n s z)) 4
  - $\rightarrow \lambda s. \lambda z. 3 s (4 s z)$
  - $\rightarrow \lambda s. \lambda z. 3 s (s (s (s (s z))$
  - $\rightarrow \lambda s. \lambda z. s$  ( s (s (s (s (s z))

- plus 3 4=  $(\lambda m. \lambda n. \lambda s. \lambda z. m s (n s z))$  3 4
  - $\rightarrow$  ( $\lambda$ n.  $\lambda$ s.  $\lambda$ z. 3 s (n s z)) 4
  - $\rightarrow \lambda s. \lambda z. 3 s (4 s z)$
  - $\rightarrow \lambda s. \lambda z. 3 s (s (s (s (s z))$
  - $\rightarrow \lambda s. \lambda z. s (s (s (s (s (s z)) = 7)$

## Arithmetic: Multiplication

• times =  $\lambda$ m.  $\lambda$ n. m (plus n) 0 Apply "m times" the function that add n to its argument to 0

• times  $\frac{3}{4} = (\lambda m. \lambda n. m (plus n) 0) \frac{3}{4}$ 

• times  $\frac{3}{4} = (\lambda m. \lambda n. m \text{ (plus n) 0) } \frac{3}{4}$  $\rightarrow (\lambda n. 3 \text{ (plus n) 0) } \frac{4}{4}$ 

- times  $\frac{3}{4} = (\lambda m. \lambda n. m (plus n) 0) \frac{3}{4}$ 
  - $\rightarrow$  ( $\lambda$ n. 3 (plus n) 0) 4
  - $\rightarrow$  3 (plus 4) 0

• times 3 4 =  $(\lambda m. \lambda n. m \text{ (plus n) 0) 3 4}$   $\rightarrow (\lambda n. 3 \text{ (plus n) 0) 4}$   $\rightarrow 3 \text{ (plus 4) 0}$  $\rightarrow \text{ (plus 4) ((plus 4) 0))}$ 

```
• times 3.4 = (\lambda m. \lambda n. m (plus n) 0) 3.4

\rightarrow (\lambda n. 3 (plus n) 0) 4

\rightarrow 3 (plus 4) 0

\rightarrow (plus 4) ((plus 4) ((plus 4) 0))

\rightarrow (plus 4) ((plus 4) 4)
```

```
    times 3 4 = (λm. λn. m (plus n) 0) 3 4
    → (λn. 3 (plus n) 0) 4
    → 3 (plus 4) 0
    → (plus 4) ((plus 4) ((plus 4) 0))
    → (plus 4) ((plus 4) 4)
    → (plus 4) 8
```

### Arithmetic: Multiplication Example

```
• times \frac{3}{4} = (\lambda m. \lambda n. m (plus n) 0) \frac{3}{4}
         \rightarrow (\lambdan. 3 (plus n) 0) 4
         \rightarrow 3 (plus 4) 0
         \rightarrow (plus 4) ((plus 4) ((plus 4) 0))
         \rightarrow (plus 4) ((plus 4) 4)
         \rightarrow (plus 4) 8
         = 12
```

Result of a computation depends on order of evaluation

- Result of a computation depends on order of evaluation
- In call-by-value scc of 1 is not 2. This means that the expression for scc 1 is not the same expression as the one we define for 2

- Result of a computation depends on order of evaluation
- In call-by-value scc of 1 is not 2. This means that the expression for scc 1 is not the same expression as the one we define for 2
- The expression obtained has latent computations under the lambda abstraction. These are redexes that are not reduced because of the evaluation strategy

- Result of a computation depends on order of evaluation
- In call-by-value scc of 1 is not 2. This means that the expression for scc 1 is not the same expression as the one we define for 2
- The expression obtained has latent computations under the lambda abstraction. These are redexes that are not reduced because of the evaluation strategy
- Nevertheless, if we use scc 1 in a computation, it will behave the same as 2.

- Consider the two terms
  - $-\lambda m. \lambda n. m ((\lambda x. x) n)$
  - $-\lambda m$ .  $\lambda n$ . m n

- Consider the two terms
  - $-\lambda m. \lambda n. m ((\lambda x. x) n)$
  - $-\lambda m. \lambda n. m n$

The two expressions are not identical

- Consider the two terms
  - $-\lambda m. \lambda n. m ((\lambda x. x) n)$
  - $-\lambda m. \lambda n. m n$

If we apply the first expression to and a and b we get:

•  $(\lambda m. \lambda n. m ((\lambda x. x) n))$  a b

- Consider the two terms
  - $-\lambda m. \lambda n. m ((\lambda x. x) n)$
  - $-\lambda m. \lambda n. m n$

If we apply the first expression to and a and b we get:

•  $(\lambda m. \lambda n. m ((\lambda x. x) n))$  a b  $\rightarrow (\lambda n. a ((\lambda x. x) n))$  b

- Consider the two terms
  - $-\lambda m. \lambda n. m ((\lambda x. x) n)$
  - $-\lambda m. \lambda n. m n$

If we apply the first expression to and a and b we get:

- $(\lambda m. \lambda n. m ((\lambda x. x) n)) a b$ 
  - $\rightarrow$  ( $\lambda$ n. a (( $\lambda$ x. x) n)) b
  - $\rightarrow$  a (( $\lambda x. x$ ) b)

- Consider the two terms
  - $-\lambda m. \lambda n. m ((\lambda x. x) n)$
  - $-\lambda m. \lambda n. m n$

If we apply the first expression to and a and b we get:

- $(\lambda m. \lambda n. m ((\lambda x. x) n)) a b$ 
  - $\rightarrow$  ( $\lambda$ n. a (( $\lambda$ x. x) n)) b
  - $\rightarrow$  a (( $\lambda x. x$ ) b)
  - $\rightarrow$  a b

- Consider the two terms
  - $-\lambda m. \lambda n. m ((\lambda x. x) n)$
  - $-\lambda m$ .  $\lambda n$ . m n

If we apply the second expression to and a and b we get:

- $(\lambda m. \lambda n. m n) a b$ 
  - $\rightarrow$  ( $\lambda$ n. a n) b
  - $\rightarrow$  a b

- Consider the two terms
  - $-\lambda m. \lambda n. m ((\lambda x. x) n)$
  - $-\lambda m. \lambda n. m n$

Even though they are not the same expression, they behave the same way

#### **Arithmetic: Predecessor**

The predecessor function is trickier

#### Arithmetic: Predecessor

- The predecessor function is trickier
- How can we subtract by adding?

$$x = 0$$
  $y = 0$   
for  $i = 1$  to  $n$   
 $x = y$   
 $y = y + 1$ 

$$x = 0$$
  $y = 0$   
for  $i = 1$  to  $n$   
 $x = y$   
 $y = y + 1$ 

If n is not 0:

At all times except at the beginning, x is behind y

x is incremented n-1 times y is increment n times

$$x = 0$$
  $y = 0$   
for  $i = 1$  to  $n$   
 $x = y$   
 $y = y + 1$ 

How do we achieve this with lambda calculus?

$$x = 0$$
  $y = 0$   
for  $i = 1$  to  $n$   
 $x = y$   
 $y = y + 1$ 

Let us think about this a little differently

$$x = 0$$
  $y = 0$   
for  $i = 1$  to  $n$   
 $x_{new} = y_{old}$   
 $y_{new} = y_{old} + 1$ 

$$x = 0$$
  $y = 0$   
for  $i = 1$  to  $i = 1$  to

$$(x, y) = (0, 0)$$
  
for i = 1 to n  
 $(x_{new}, y_{new}) = (y_{old}, y_{old} + 1)$ 

$$(x, y) = pair 0 0$$
  
for  $i = 1$  to n  
 $(x_{new}, y_{new}) = pair snd(x,y) (snd(x,y) +1)$ 

Warning! This is not lambda calculus notation

remember n a b = a (a(a(...(ab)...))

```
// Initial pair zz = pair 0 0
```

```
// Initial pair
    zz = pair 0 0

// body of the loop
    ss = λp. pair (snd p) (succ (snd p))
```

```
// Initial pair
      zz = pair 0 0
// body of the loop
      ss = \lambda p. pair (snd p) (succ (snd p))
// apply n times and extract x
      prd = \lambdan. fst (n ss zz)
```

### Predecessor example

```
// Initial pair
           zz = pair 0 0
// body of the loop
           ss = \lambda p. pair (snd p) (succ (snd p))
// apply n times and extract x
           prd = \lambdan. fst (n ss zz)
         = (\lambda n. fst (n ss zz)) 4
prd 4
           = fst (4 ss zz)
           = fst ( ss ( ss ( ss zz ) ) ) )
           = fst (ss (ss (ss (pair 0 0)))))
           = fst (ss (ss (pair 0 1)))
           = fst (ss (ss (pair 1 2)))
           = fst ( ss (pair 2 3) )
           = fst (pair 3 4)
           =3
```

### Subtraction

minus = =  $\lambda$ m.  $\lambda$ n. n prd m

- We want a function to check if a number is 0
- It should return either tru or fls

• iszero =  $\lambda$ n. n (?) tru

• iszero =  $\lambda$ n. n (?) tru

• iszero 0 = 0 (?) tru = tru

• iszero =  $\lambda$ n. n (?) tru

• iszero 0 = 0 (?) tru = tru

• iszero n = n (?) tru  $\rightarrow$  (?) ( (?) ( (?) ... tru))...))

• iszero =  $\lambda$ n. n (?) tru

• iszero 0 = 0 (?) tru = tru

• iszero n = n (?) tru  $\rightarrow$  (?) ( (?) ( (?) ... tru))...))

 We want repeated applications of (?) to result in fls always.

(?) =

- iszero =  $\lambda$ n. n (?) tru
- iszero 0 = 0 (?) tru = tru
- iszero n = n (?) tru → (?) ( (?) ( (?) ... tru))...))
- We want repeated applications of (?) to result in fls always.
   λx. fls !!!

iszero =  $\lambda n$ . n ( $\lambda x$ . fls ) tru

```
Example. iszero 3 = (\lambda n. n (\lambda x. fls) tru) 3
= 3 (\lambda x. fls) tru
= (\lambda x. fls) ((\lambda x. fls) ((\lambda x. fls) tru))
= (\lambda x. fls) ((\lambda x. fls) fls)
= (\lambda x. fls) (fls)
= (fls) = fls
```

# checking for equality

equal m n = and (gteq m n) (gteq n m)

# checking for equality

- equal m n = and (gteq m n) (gteq n m)
- gteq m n = iszero m prd n

# checking for equality

- equal m n = and (gteq m n) (gteq n m)
- gteq m n = iszero m prd n
- equal =  $\lambda$ m.  $\lambda$ n. and (gteq m n) (gteq n m)
- gteq =  $\lambda$ m.  $\lambda$ n. iszero (m prd n)

```
Example. gteq 4 3 = (\lambda m. \lambda n. iszero (m prd n)) 4 3

= iszero 4 prd 3

= iszero (prd (prd (prd (prd 3))))

= iszero (prd (prd (prd (2))))

= iszero (prd (prd (1)))

= iszero (prd (0))

= iszero 0

= tru
```

#### Recursion

- Omega =  $(\lambda x. x x) (\lambda x. x x)$
- If we try to reduce omega, we get omega!

Let us look at the factorial example, we can write

```
factorial(n) = if (n = 0) then

1
else

n * factorial (n-1)
```

Let us look at the factorial example, we can write

```
factorial(n) = if (n = 0) then 1
else n * factorial (n-1)
```

We can think of factorial(2) as

```
factorial(2) = if (2 = 0) then 1
else 2 * (if ((2-1) = 0 then 1
else (2-1) * (if (2-1-1) = 0 then 1
else ....
else ....
)
```

 In general, we start with a function that appears in its own body. Since we cannot have the function literally appear in its own body, we have a definition of the form:

```
-g = \lambda f. <body containing f>
```

 In general, we start with a function that appears in its own body. Since we cannot have the function literally appear in its own body, we have a definition of the form:

$$-g = \lambda f$$
. 

In the case of factorial this will look as follows:

$$-g = \lambda fct$$
.  $\lambda n$ . if = n 0 then 1 else (times n (fct (prd n)))

The approach taken is to use a Fixed point combinator that would replicate f as needed in the evaluation:

```
- fix = \lambdaf. (\lambdax. f (\lambday. x x y)) (\lambdax. f (\lambday. x x y))
```

The approach taken is to use a Fixed point combinator that would replicate f as needed in the evaluation:

```
- fix = \lambdaf. (\lambdax. f (\lambday. x x y)) (\lambdax. f (\lambday. x x y))
```

If we apply fix to the function

```
- g = \lambda fct. \lambda n. if = n 0 then 1
else (times n (fct (prd n)))
```

The approach taken is to use a Fixed point combinator that would replicate f as needed in the evaluation:

```
- fix = \lambdaf. (\lambdax. f (\lambday. x x y)) (\lambdax. f (\lambday. x x y))
```

If we apply fix to the function

```
- g = \lambda fct. \lambda n. if = n 0 then 1
else (times n (fct (prd n)))
```

We get the expansion

```
g = \lambda fct. \lambda n. if = n 0 then 1
else (times n (fct (prd n)))
```

factorial = 
$$\frac{fix}{g} = \frac{(\lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)))}{g}$$

factorial 2 = 
$$(fix g) 2$$

```
g = \lambda fct. \ \lambda n. \ if = n \ 0 \ then \ 1 else \ (times \ n \ (fct \ (prd \ n))) factorial = fix \ g = (\lambda f. \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y)) \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y)) \ ) \ g factorial \ 2 \qquad = (fix \ g) \ 2 = ((\lambda f. \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y)) \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y)) \ ) \ g \ ) \ 2
```

```
g = \lambda fct. \ \lambda n. \ if = n \ 0 \ then \ 1
else \ (times \ n \ (fct \ (prd \ n)))
factorial = fix \ g = (\lambda f. \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y)) \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y)) \ ) \ g
factorial \ 2
= (fix \ g) \ 2
= ((\lambda f. \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y)) \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y)) \ ) \ g) \ 2
= ((\lambda x. \ g \ (\lambda y. \ x \ x \ y)) \ (\lambda x. \ g \ (\lambda y. \ x \ x \ y))) \ 2
```

```
g = \lambda fct. \ \lambda n. \ if = n \ 0 \ then \ 1 else \ (times \ n \ (fct \ (prd \ n))) factorial = fix \ g = (\lambda f. \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y)) \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y)) \ ) \ g factorial \ 2 = (fix \ g) \ 2 = ((\lambda f. \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y)) \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y))) \ ) \ g) \ 2 = ((\lambda x. \ g \ (\lambda y. \ x \ x \ y)) \ (\lambda x. \ g \ (\lambda y. \ x \ x \ y))) \ 2
```

 $h = (\lambda x. g (\lambda y. x x y))$ 

```
g = \lambda fct. \lambda n. if = n 0 then 1
                                else (times n (fct (prd n)))
factorial = fix g = (\lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)))
factorial 2
                              = (fix g) 2
                              = ((\lambda f. (\lambda x. f(\lambda y. x x y)) (\lambda x. f(\lambda y. x x y))) g) 2
                               = ((\lambda x. g(\lambda y. x x y))(\lambda x. g(\lambda y. x x y))) 2
                               = (h h) 2
      \mathbf{h} = (\lambda \mathbf{x}. \mathbf{g} (\lambda \mathbf{y}. \mathbf{x} \mathbf{x} \mathbf{y}))
```

```
g = \lambda fct. \ \lambda n. \ if = n \ 0 \ then \ 1
else \ (times \ n \ (fct \ (prd \ n)))
factorial = fix \ g = (\lambda f. \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y)) \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y)) \ ) \ g
factorial 2
= (fix \ g) \ 2
= ((\lambda f. \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y)) \ (\lambda x. \ f \ (\lambda y. \ x \ x \ y)) \ ) \ g) \ 2
= ((\lambda x. \ g \ (\lambda y. \ x \ x \ y)) \ (\lambda x. \ g \ (\lambda y. \ x \ x \ y))) \ 2
= (h \ h) \ 2
= ((\lambda x. \ g \ (\lambda y. \ x \ x \ y)) \ h) \ 2
```

```
g = \lambda fct. \lambda n. if = n 0 then 1
                            else (times n (fct (prd n)))
factorial = fix g = (\lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)))
factorial 2
                          = (fix g) 2
                          = ((\lambda f. (\lambda x. f(\lambda y. x x y)) (\lambda x. f(\lambda y. x x y))) g) 2
                          = ((\lambda x. g(\lambda y. x x y))(\lambda x. g(\lambda y. x x y))) 2
                          = (h h) 2
                          = ((\lambda x. g (\lambda y. x x y)) h) 2
                          = (g (\lambda v. h h v)) 2
```

```
g = \lambda fct. \lambda n. if = n 0 then 1
                             else (times n (fct (prd n)))
factorial = fix g = (\lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)))
factorial 2
                           = (fix g) 2
                           = ((\lambda f. (\lambda x. f(\lambda y. x x y)) (\lambda x. f(\lambda y. x x y))) g) 2
                           = ((\lambda x. g(\lambda y. x x y))(\lambda x. g(\lambda y. x x y))) 2
                           = (h h) 2
                           = ((\lambda x. g (\lambda y. x x y)) h) 2
                           = (g (\lambda y. h h y)) 2
              = ((\lambda \text{fct. } \lambda \text{n. if} = \text{n 0 then 1 else (times n (fct (prd n))))} (\lambda y. h h y)) 2
```

```
g = \lambda fct. \lambda n. if = n 0 then 1
                             else (times n (fct (prd n)))
factorial = fix g = (\lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)))
                            = (fix g) 2
factorial 2
                            = ((\lambda f. (\lambda x. f(\lambda y. x x y)) (\lambda x. f(\lambda y. x x y))) g) 2
                            = ((\lambda x. g(\lambda y. x x y))(\lambda x. g(\lambda y. x x y))) 2
                            = (h h) 2
                            = ((\lambda x. g (\lambda y. x x y)) h) 2
                            = (g (\lambda y. h h y)) 2
              = ((\frac{\lambda fct}{\lambda n}, \lambda n), if = n 0 then 1 else (times n (fct (prd n)))) (\frac{\lambda y}{\lambda n}, \frac{h h y}{\lambda n})
              = (\lambda n. \text{ if } = n \text{ 0 then 1 else (times } n ((\lambda y. h h y))) (prd n)))) 2
              = if = 2 0 then 1 else (times 2 ((\lambda y. h h y)) (prd 2)))
```

```
g = \lambda fct. \lambda n. if = n 0 then 1
                                 else (times n (fct (prd n)))
factorial = fix g = (\lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)))
factorial 2
                               = (fix g) 2
                               = ((\lambda f. (\lambda x. f(\lambda y. x x y)) (\lambda x. f(\lambda y. x x y))) g) 2
                               = ((\lambda x. g(\lambda y. x x y))(\lambda x. g(\lambda y. x x y))) 2
                               = (h h) 2
                               = ((\lambda x. g (\lambda y. x x y)) h) 2
                               = (g (\lambda y. h h y)) 2
                = ((\frac{\lambda fct}{\lambda fct}, \lambda n), \text{ if } = n \text{ 0 then 1 else (times n (} \frac{fct}{\lambda fct}, \text{ (prd n))))} (\frac{\lambda y}{\lambda fct}, \text{ h h y))} 2
                               = if = \frac{2}{2} 0 then 1 else (times \frac{2}{((\lambda y. h h y))} (prd \frac{2}{2})))
```

```
g = \lambda fct. \lambda n. if = n 0 then 1
                           else (times n (fct (prd n)))
factorial = fix g = (\lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)))
factorial 2
                         = (fix g) 2
                          = ((\lambda f. (\lambda x. f(\lambda y. x x y)) (\lambda x. f(\lambda y. x x y))) g) 2
                          = ((\lambda x. g(\lambda y. x x y))(\lambda x. g(\lambda y. x x y))) 2
                          = (h h) 2
                          = ((\lambda x. g (\lambda y. x x y)) h) 2
                          = (g (\lambda y. h h y)) 2
             = ((\lambda \text{fct. } \lambda \text{n. if} = \text{n 0 then 1 else (times n (fct (prd n))))} (\lambda y. h h y)) 2
                          = if = 2 0 then 1 else (times 2 ((\lambda y. h h y) (prd 2)))
                          = if = 2 0 then 1 else (times 2 ((h h) (prd 2)))
```

#### In general:

- $-g = \lambda f$ . <body containing f>
- H = fix g is the recursive function

 When applied to a parameter, H will act like the recursive function we want

### Linked Lists with Lambda Calculus

```
empty list pair fls fls this fls indicates that the list is empty
list of at least one element pair tru (pair a L') this tru indicates that the list is not
empty
empty list: pair fls fls
list that contains only 1: pair tru (pair 1 (pair fls fls))
list that contains 1 and 5: pair tru (pair 1 (pair tru (pair 5 (pair fls fls))))
list that contains 2, 1 and 5: pair tru (pair 2 ( pair tru (pair 1 (pair tru (pair 5 (pair fls fls))))))
```

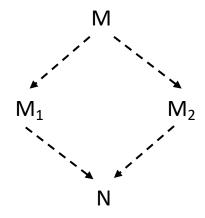
#### Church-Rosser Theorem

Informally: the order of reductions does not matter

#### Church-Rosser Theorem

Informally: the order of reductions does not matter

• More formally: If  $M \to^* M_1$  and  $M \to^* M_2$  then there exists N such that  $M_1 \to^* N$  and  $M_2 \to^* N$ 



# $\eta$ -reduction

•  $\lambda x. c x \rightarrow^{\eta} c$  if c does not contain free occurrences of x (note here we are talking about c by itself

# $\eta$ -reduction

•  $\lambda x. c x \rightarrow^{\eta} c$  if c does not contain free occurrences of x (note here we are talking about c by itself

• Justification:  $(\lambda x. c x) a \rightarrow^{\beta} c a$