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

- $(\lambda x.\ t)\ t' \rightarrow [x \rightarrow t]\ t'$

  Redex: **Red**ucible **ex**pression

  β-reduction

- The answer is obtained by
  - Replacing all bound occurrences of x in λx. t with t’ . Here we are interested in occurrences bound by the outer λx
  - Getting rid of the λx
  - 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)

Reduction Examples

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

Reduction Examples

- $(\lambda x.\ \boxed{x}\ )\ \boxed{y}$

formal parameter

actual parameter

body

Reduction Examples

- $(\lambda x.\ \boxed{x}\ )\ \boxed{y}$

formal parameter

actual parameter

body

we should replace the **x** in the body with y

Reduction Examples

- $(\lambda \mathbf{x}.\ \mathbf{x}\ )\ y \rightarrow y$

formal parameter

actual parameter

result

body

we should replace the **x** in the body with y

Reduction Examples

- $(\lambda \mathbf{x}.\ \mathbf{x}\ (\lambda \mathbf{x}.\ \mathbf{x}))\ (u\ r)$

Reduction Examples

- $(\lambda \mathbf{x}.\ \mathbf{x}\ (\lambda \mathbf{x}.\ \mathbf{x}))\ (u\ r)$

formal parameter

body

actual parameter

Reduction Examples

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

formal parameter

body

actual parameter

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

Reduction Examples

- $(\lambda \mathbf{x}.\ \mathbf{x}\ (\lambda \mathbf{x}.\ \mathbf{x}))\ (u\ r) \quad \rightarrow \quad (u\ r)\ (\lambda \mathbf{x}.\ \mathbf{x})$

formal parameter

actual parameter

result

body

Reduction Examples

$(\lambda \mathbf{x}.\ (\lambda \mathbf{x}.\ \mathbf{x})\ \mathbf{x})\ (u\ r)$

Reduction Examples

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

formal parameter

body

actual parameter

Reduction Examples

$$(\lambda \mathbf{x}.\ (\lambda \mathbf{x}.\ \mathbf{x})\ \mathbf{x})\ (u\ r)$$

formal parameter

body

actual parameter

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

Reduction Examples

$(\lambda \mathbf{x}.\ (\lambda \mathbf{x}.\ \mathbf{x})\ \mathbf{x})\ (\ u\ r) \quad \rightarrow \quad (\lambda \mathbf{x}.\ \mathbf{x})\ (u\ r)$

formal parameter

actual parameter

result

body

Reduction Examples

$(\lambda x.\ (\lambda x.\ x)\ x)\ (u\ r) \quad \to \quad (\lambda x.\ x)\ (u\ r)$

Reduction Examples

$(\lambda x.\ (\lambda x.\ x)\ x)\ (u\ r) \quad \rightarrow \quad (\lambda x.\ \boxed{x})\ \boxed{(u\ r)}$

formal parameter

body

actual parameter

Reduction Examples

$(\lambda x.\ (\lambda x.\ x)\ x)\ (u\ r) \quad \rightarrow \quad (\lambda x.\ x)\ (u\ r)$

formal parameter

actual parameter

body

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

Reduction Examples

$(\lambda x.\ (\lambda x.\ x)\ x)\ (u\ r) \quad \rightarrow \quad (\lambda x.\ x)\ (u\ r) \quad \rightarrow \quad (u\ r)$

result

formal parameter

actual parameter

body

Macro evaluation problem!

```
#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);
}
```

Macro evaluation problem!

```
#include <stdio.h>

#define INC(x) {int a = 0; x++;}

int  main()
{
        int x = 3;
        int a = 3;

        INC(x);
        INC(a);          // {int a = 0; a++;}

        printf("x = %d  a = %d\n", x, a);
}
```

Macro evaluation problem!

```
#include <stdio.h>

#define INC(x) {int a = 0; x++;}

int  main()
{
        int x = 3;
        int a = 3;

        INC(x);
        INC(a);          // {int a = 0; a++;}

        printf("x = %d  a = %d\n", x, a);
}

Output:     x = 4 a = 3
```

Macro evaluation problem!

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

Macro evaluation problem!

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


f(3) ≡  result
```

Macro evaluation problem!

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

```
f(3) ≣  result
```

where result is calculated as follows

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

Macro evaluation problem!

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

```
f(x) ≡  result
```

where result is calculated as follows

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

Macro evaluation problem!

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

```
f(j) ≣  result
```

where result is calculated as follows

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

Macro evaluation problem!

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

Result should not depend on name of local variable

```
f(j) ≡  result
```

where result is calculated as follows

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

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) ≣  result
```

where result is calculated as follows

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

Reduction Examples

- $(\lambda x. \boxed{(\lambda y.\ y\ x)\ z})\ y$

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**).

Reduction Examples

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

We should rename before we do the beta reduction:

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

Reduction Examples

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

We should rename before we do the beta reduction:

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

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

Reduction Examples

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

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

  - $\underline{((\lambda x.\ \underline{(\lambda x.\ x)\ x})\ (u\ r))}\ \underline{((\lambda x.\ \underline{(\lambda x.\ x)\ x})\ (u\ r))}$

Normal order examples

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

  – $\underline{((\lambda x.\ \underline{(\lambda x.\ x)\ x})\ (u\ r))}\ \underline{((\lambda x.\ \underline{(\lambda x.\ x)\ x})\ (u\ r))}$

  – $\underline{((\lambda x.\ \underline{(\lambda x.\ x)\ x})\ (u\ r))}$

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

Call by value evaluation compared to normal order evaluation

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 $\rightarrow$

Call by value evaluation compared to normal order evaluation

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

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

Call by value evaluation compared to normal order evaluation

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

Call by value evaluation compared to normal order evaluation

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

Call by value evaluation compared to normal order evaluation

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

2. Normal order evaluation

Call by value evaluation compared to normal order evaluation

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

2. Normal order evaluation

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

Multiple arguments and <u>currying</u>

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

Multiple arguments and <u>currying</u>

- 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

Multiple arguments and currying

- 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

Multiple arguments and <u>currying</u>

- 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

Multiple arguments and currying

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

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

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

Multiple arguments and currying

- Same effect can be achieved by higher order functions

Haskell Curry

Multiple arguments and <u>currying</u>

- Same effect can be achieved by higher order functions
  - λx. λy. s is a function of x.

Multiple arguments and currying

- 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.

Multiple arguments and currying

- 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 $\lambda y.\ [x \to v]\ s$ which is a function of y.
  - If we apply the result to w , we get $[y \to w][x \to v]\ s$

Multiple arguments and currying

- 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 $\lambda y.\ [x \to v]\ s$ which is a function of y.
  - If we apply the result to w, we get $[y \to w][x \to v]\ s$

- **Exactly what we needed**

Multiple arguments and <u>currying</u>

- 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**

Church Booleans

We want to define boolean values that can be tested:

- test b v w =

Church Booleans

We want to define boolean values that can be tested:

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

Church Booleans

We want to define boolean values that can be tested:

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

Church Booleans

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

- **tru** = $\lambda t.\ \lambda f.\ t$

tru

- **tru** = λt. λf. t

tru a = (λt. λf. t) a

tru

- **tru** = $\lambda t.\ \lambda f.\ t$

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

tru

- **tru** = λt. λf. t

tru a = (λt. λf. t) a $\rightarrow$ λf. a

tru a b = ((λt. λf. t) a) b

tru

- **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

- **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

- **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

- **fls** = $\lambda t.\ \lambda f.\ f$

fls

- **fls** = λt. λf. f

  fls a = (λt. λf. f) a

fls

- **fls** = λt. λf. f

fls a = (λt. λf. f) a $\rightarrow$ λf. f

fls

- **fls** = λt. λf. f

fls a = (λt. λf. f) a $\rightarrow$ λf. f

fls a b = ((λt. λf. f) a) b

fls

- **fls** = λt. λf. f

fls a = (λt. λf. f) a $\rightarrow$ λf. f

fls a b = ((λt. λf. f) a) b $\rightarrow$ (λf. f) b

fls

- **fls** = λt. λf. f

fls a = (λt. λf. f) a → λf. f

fls a b = ((λt. λf. f) a) b → (λf. f) b → 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 = λl. λm. λn. l m n

- test tru v w =

test = λl. λm. λn. l m n

- test tru v w =
  <u>(λl. λm. λn. l m n) tru</u> v w

test = λl. λm. λn. l m n

- test tru v w =

  $\underline{(\lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n)\ tru}\ v\ w$

  $\rightarrow\ \underline{(\lambda m.\ \lambda n.\ tru\ m\ n)\ v}\ w$

test = λl. λm. λn. l m n

- test tru v w =

  $\underline{(\lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n)\ \text{tru}}\ v\ w$

  $\rightarrow\ \underline{(\lambda m.\ \lambda n.\ \text{tru}\ m\ n)\ v}\ w$

  $\rightarrow\ \underline{(\lambda n.\ \text{tru}\ v\ n)\ w}$

test = λl. λm. λn. l m n

- test tru v w =

  $\underline{(\lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n)\ \text{tru}}\ v\ w$

  $\rightarrow\ \underline{(\lambda m.\ \lambda n.\ \text{tru}\ m\ n)\ v}\ w$

  $\rightarrow\ \underline{(\lambda n.\ \text{tru}\ v\ n)\ w}$

  $\rightarrow\ \text{tru}\ v\ w$

test = $\lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n$

- test tru v w =

  $\underline{(\lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n)\ \text{tru}}\ v\ w$

  $\rightarrow\ \underline{(\lambda m.\ \lambda n.\ \text{tru}\ m\ n)\ v}\ w$

  $\rightarrow\ \underline{(\lambda n.\ \text{tru}\ v\ n)\ w}$

  $\rightarrow\ \text{tru}\ v\ w$

  $\rightarrow\ \underline{(\lambda t.\ \lambda f.\ t)\ v}\ w$

test = λl. λm. λn. l m n

- test tru v w =

  $\underline{(\lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n)\ tru}\ v\ w$

  $\to\ \underline{(\lambda m.\ \lambda n.\ tru\ m\ n)\ v}\ w$

  $\to\ \underline{(\lambda n.\ tru\ v\ n)\ w}$

  $\to\ tru\ v\ w$

  $\to\ \underline{(\lambda t.\ \lambda f.\ t)\ v}\ w$

  $\to\ \underline{(\lambda f.\ v)\ w}$

test = λl. λm. λn. l m n

- test tru v w =

  $\underline{(\lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n)\ tru}\ v\ w$

  $\rightarrow\ \underline{(\lambda m.\ \lambda n.\ tru\ m\ n)\ v}\ w$

  $\rightarrow\ \underline{(\lambda n.\ tru\ v\ n)\ w}$

  $\rightarrow\ tru\ v\ w$

  $\rightarrow\ \underline{(\lambda t.\ \lambda f.\ t)\ v}\ w$

  $\rightarrow\ \underline{(\lambda f.\ v)\ w}$

  $\rightarrow\ v$

test = λl. λm. λn. l m n

- test tru v w =

  $\underline{(\lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n)\ tru}\ v\ w$

  $\rightarrow \underline{(\lambda m.\ \lambda n.\ tru\ m\ n)\ v}\ w$

  $\rightarrow \underline{(\lambda n.\ tru\ v\ n)\ w}$

  $\rightarrow$ tru v w

  $\rightarrow \underline{(\lambda t.\ \lambda f.\ t)\ v}\ w$

  $\rightarrow \underline{(\lambda f.\ v)\ w}$

  $\rightarrow$ v

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

Logical Operators

- and = λb. λc. b c fls

Logical Operators

- and = λb. λc. b c fls
- and tru tru

Logical Operators

- and = λb. λc. b c fls

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

Logical Operators

- and = λb. λc. b c fls

- and tru tru

  (λb. λc. b c fls) tru tru

  → (λc. tru c fls) tru

Logical Operators

- and = λb. λc. b c fls

- and tru tru

  (λb. λc. b c fls) tru tru

  $\rightarrow$ (λc. tru c fls) tru

  $\rightarrow$ tru tru fls

should be grouped as follows

Logical Operators

- and = λb. λc. b c fls

- and tru tru

  (λb. λc. b c fls) tru tru

  $\rightarrow$ (λc. tru c fls) tru

  $\rightarrow$ tru tru fls

  $\rightarrow$ (tru tru) fls

should be grouped as follows

Logical Operators

- and = λb. λc. b c fls

- and tru tru

  (λb. λc. b c fls) tru tru

  $\rightarrow$ (λc. tru c fls) tru

  $\rightarrow$ tru tru fls should be grouped as follows

  $\rightarrow$ (tru tru) fls

  $\rightarrow$ $\underline{((\lambda t.\ \lambda f.\ t)\ tru)}$ fls

Logical Operators

- and = λb. λc. b c fls

- and tru tru

  (λb. λc. b c fls) tru tru

  → (λc. tru c fls) tru

  → tru tru fls   should be grouped as follows

  → (tru tru) fls

  → <u>((λt. λf. t) tru)</u> fls

  → <u>(λf. tru) fls</u>

Logical Operators

- and = λb. λc. b c fls

- and tru tru

  (λb. λc. b c fls) tru tru

  → (λc. tru c fls) tru

  → tru tru fls should be grouped as follows

  → (tru tru) fls

  → <u>((λt. λf. t) tru)</u> fls

  → <u>(λf. tru) fls</u>

  → tru

Logical Operators

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

Logical Operators

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

Logical Operators

- or = λb. λc. b tru c

- or fls tru

  (λb. λc. b tru c) fls tru

Logical Operators

- or = λb. λc. b tru c

- or fls tru

  (λb. λc. b tru c) fls tru

  $\rightarrow$ (λc. fls tru c) tru

Logical Operators

- or = λb. λc. b tru c

- or fls tru

$$\begin{array}{ll} & (\lambda b.\ \lambda c.\ \ b\ \mathrm{tru}\ c)\ \mathrm{fls}\ \mathrm{tru} \\ \to & (\lambda c.\ \ \mathrm{fls}\ \mathrm{tru}\ c)\ \mathrm{tru} \\ \to & \mathrm{fls}\ \mathrm{tru}\ \mathrm{tru} \end{array}$$

Logical Operators

- or = λb. λc. b tru c

- or fls tru

  (λb. λc. b tru c) fls tru

  → (λc. fls tru c) tru

  → fls tru tru should be grouped as:

  → (fls tru) tru

Logical Operators

- or = λb. λc. b tru c

- or fls tru

  (λb. λc. b tru c) fls tru

  → (λc. fls tru c) tru

  → fls tru tru   should be grouped as:

  → (fls tru) tru

  → $\underline{\text{((}\lambda\text{t. }\lambda\text{f. f) tru)}}$ tru

Logical Operators

- or = λb. λc. b tru c

- or fls tru

  (λb. λc. b tru c) fls tru

  → (λc. fls tru c) tru

  → fls tru tru

  should be grouped as:

  → (fls tru) tru

  → <u>((λt. λf. f) tru)</u> tru

  → <u>(λf. λf) tru</u>

Logical Operators

- or = λb. λc. b tru c

- or fls tru

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

$\to\ (\lambda c.\ fls\ tru\ c)\ tru$

$\to\ fls\ tru\ tru$ should be grouped as:

$\to\ (fls\ tru)\ tru$

$\to\ \underline{((\lambda t.\ \lambda f.\ f)\ tru)}\ tru$

$\to\ \underline{(\lambda f.\ \lambda f)\ tru}$

$\to\ tru$

Pairs

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

Pairs

- pair = λf. λs. λb. b f s
- fst = λp. p tru
- snd = λp. p fls

pair v w = (λf. λs. λb. b f s) v w

Pairs

- pair = λf. λs. λb. b f s
- fst = λp. p tru
- snd = λp. p fls

pair v w = (λf. λs. λb. b f s) v w
$\to$ (λs. λb. b v s) w

Pairs

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

$$\begin{aligned} \text{pair } v\ w &= (\lambda f.\ \lambda s.\ \lambda b.\ \ b\ f\ s)\ v\ w \\ &\to (\lambda s.\ \lambda b.\ \ b\ v\ s)\ w \\ &\to \lambda b.\ \ b\ v\ w \end{aligned}$$

Pairs

- pair = λf. λs. λb. b f s
- fst = λp. p tru
- snd= λp. p fls

pair v w = (λf. λs. λb. b f s) v w
→ (λs. λb. b v s) w
→ λb. 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 $(\underline{(\lambda f.\ \lambda s.\ \lambda b.\ \ b\ f\ s)\ v}\ w)$

$\rightarrow$ fst $(\underline{(\lambda s.\ \lambda b.\ \ b\ v\ s)\ w}\ )$

$\rightarrow$ fst $(\underline{\lambda b.\ \ b\ v\ w})$

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

$\rightarrow$ fst $(\lambda b.\ b\ v\ w)$

$\rightarrow$ $(\lambda p.\ \mathbf{p}\ tru\ )\ (\lambda b.\ b\ v\ w)$

formal parameter

body

actual parameter

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

$\rightarrow$ fst $(\lambda b.\ b\ v\ w)$

$\rightarrow$ $(\lambda p.\ \mathbf{p}\ tru\ )\ (\lambda b.\ b\ v\ w)$

formal parameter　　body　　actual parameter

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

Extracting the first element from a pair

- fst (pair v w)

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

→ fst $((\underline{\lambda s.\ \lambda b.\ \ b\ v\ s)\ w}\ )$

→ fst $(\underline{\lambda b.\ \ b\ v\ w)}$

→ $\underline{(\lambda p.\ p\ \text{tru}\ )\ (\lambda b.\ \ b\ v\ w)}$

→ $\underline{(\lambda b.\ \ b\ v\ w)\ \text{tru}}$

Extracting the first element from a pair

- fst (pair v w)

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

$\rightarrow$ $\quad$ fst $((\lambda s.\ \lambda b.\ b\ v\ s)\ w)$

$\rightarrow$ $\quad$ fst $(\lambda b.\ b\ v\ w)$

$\rightarrow$ $\quad$ $(\lambda p.\ p\ \text{tru})\ (\lambda b.\ b\ v\ w)$

$\rightarrow$ $\quad$ $(\lambda b.\ b\ v\ w)\ \text{tru}$

$\rightarrow$ $\quad$ tru v w

$\rightarrow$ $\quad$ v

Church Numerals

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

Applying Church Numerals

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

Applying Church Numerals

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

Applying Church Numerals

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

Applying Church Numerals

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

Applying Church Numerals

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

Applying Church Numerals

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

Applying Church Numerals

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

Applying Church Numerals

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

  $\rightarrow (\lambda z.\ a\ (a\ (a\ z)))\ b$

Applying Church Numerals

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

  $\rightarrow (\lambda z.\ a\ (a\ (a\ z)))\ b$

  $\rightarrow\ a\ (a\ (a\ b)))$

Arithmetic: Successor Function

- scc = λn. λs. λz. s (n s z)

  Apply n to function s (successor) and z (zero) and apply s one more time

Arithmetic: Successor Function Example

- scc 2 = (λn. λs. λz. s (n s z)) 2

Arithmetic: Successor Function Example

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

  $\rightarrow (\lambda s.\ \lambda z.\ s\ (2\ s\ z))$

Arithmetic: Successor Function Example

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

  $\rightarrow (\lambda s.\ \lambda z.\ s\ (2\ s\ z)\ )$

  $\rightarrow (\lambda s.\ \lambda z.\ s\ (s\ (s\ z))\ ) = 3$

Arithmetic: Successor Function Example

- scc 4 = (λn. λs. λz. s (n s z)) 4

Arithmetic: Successor Function Example

- scc 4 = (λn. λs. λz. s (n s z)) 4

  $\rightarrow$ (λs. λz. s (4 s z))

Arithmetic: Successor Function Example

- scc 4 = (λn. λs. λz. s (n s z)) 4

  $\rightarrow$ (λs. λz. s (4 s z))

  $\rightarrow$ (λs. λz. s (s (s (s (s z)))))

Arithmetic: Successor Function Example

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

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

  $\rightarrow (\lambda s.\ \lambda z.\ s\ (s\ (s\ (s\ (s\ z)))))= 5$

Arithmetic: Addition

- plus = λm. λn. λs. λz. m s (n s z)

  Apply m to the function s and the result of applying n to s and z

Arithmetic: Addition Example

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

Arithmetic: Addition Example

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

  $\rightarrow (\lambda n.\ \lambda s.\ \lambda z.\ 3\ s\ (n\ s\ z))\ 4$

Arithmetic: Addition Example

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

  $\rightarrow (\lambda n.\ \lambda s.\ \lambda z.\ 3\ s\ (n\ s\ z))\ 4$

  $\rightarrow \lambda s.\ \lambda z.\ 3\ s\ (4\ s\ z)$

Arithmetic: Addition Example

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

  $\rightarrow (\lambda n.\ \lambda s.\ \lambda z.\ 3\ s\ (n\ s\ z))\ 4$

  $\rightarrow \lambda s.\ \lambda z.\ 3\ s\ (4\ s\ z)$

  $\rightarrow \lambda s.\ \lambda z.\ 3\ s\ (s\ (s\ (\ s\ (s\ z))$

Arithmetic: Addition Example

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

  $\rightarrow (\lambda n.\ \lambda s.\ \lambda z.\ 3\ s\ (n\ s\ z))\ 4$

  $\rightarrow \lambda s.\ \lambda z.\ 3\ s\ (4\ s\ z)$

  $\rightarrow \lambda s.\ \lambda z.\ 3\ s\ (s\ (s\ (\ s\ (s\ z))))$

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

Arithmetic: Addition Example

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

  $\rightarrow (\lambda n.\ \lambda s.\ \lambda z.\ 3\ s\ (n\ s\ z))\ 4$

  $\rightarrow \lambda s.\ \lambda z.\ 3\ s\ (4\ s\ z)$

  $\rightarrow \lambda s.\ \lambda z.\ 3\ s\ (s\ (s\ (\ s\ (s\ z)))$

  $\rightarrow \lambda s.\ \lambda z.\ s\ (\ s\ (\ s\ (s\ (s\ (\ s\ (s\ z)))\ = 7$

Arithmetic: Multiplication

- times = λm. λn. m (plus n) 0

  Apply "m times" the function that add n to its argument to 0

Arithmetic: Multiplication Example

- times 3 4 = (λm. λn. m (plus n) 0) 3 4

Arithmetic: Multiplication Example

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

  $\rightarrow (\lambda n.\ 3\ (\text{plus}\ n)\ 0)\ 4$

Arithmetic: Multiplication Example

- times 3 4 = (λm. λn. m (plus n) 0) 3 4

  → (λn. 3 (plus n) 0) 4

  → 3 (plus 4) 0

Arithmetic: Multiplication Example

- 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))

Arithmetic: Multiplication Example

- 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)

Arithmetic: Multiplication Example

- 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 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

  = 12

Latent Computation

- Result of a computation depends on order of evaluation

Latent Computation

- 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

Latent Computation

- 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

Latent Computation

- 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.

Latent Computation Example

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

Latent Computation Example

- Consider the two terms
  - λm. λn. m ((λx. x) n)
  - λm. λn. m n

The two expressions are not identical

Latent Computation Example

- Consider the two terms
  - λm. λn. m ((λx. x) n)
  - λm. λn. m n

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

- (λm. λn. m ((λx. x) n)) a b

Latent Computation Example

- Consider the two terms
  - λm. λn. m ((λx. x) n)
  - λm. λn. m n

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

- (λm. λn. m ((λx. x) n)) a b
  $\rightarrow$ (λn. a ((λx. x) n)) b

Latent Computation Example

- Consider the two terms
  - λm. λn. m ((λx. x) n)
  - λm. λn. m n

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

- (λm. λn. m ((λx. x) n)) a b
  $\rightarrow$ (λn. a ((λx. x) n)) b
  $\rightarrow$ a ((λx. x) b)

Latent Computation Example

- Consider the two terms
  - λm. λn. m ((λx. x) n)
  - λm. λn. m n

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

- (λm. λn. m ((λx. x) n)) a b
  → (λn. a ((λx. x) n)) b
  → a ((λx. x) b)
  → a b

Latent Computation Example

- Consider the two terms
  - λm. λn. m ((λx. x) n)
  - λm. λn. m n

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

- (λm. λn. m n) a b
  → (λn. a n) b
  → a b

Latent Computation Example

- Consider the two terms
  - λm. λn. m ((λx. x) n)
  - λm. λn. 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?

Predecessor: Wisdom Tooth Trick

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

Predecessor: Wisdom Tooth Trick

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

0 0
0 1
1 2
2 3
3 4
..

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

If n is not 0:

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

Predecessor: Wisdom Tooth Trick

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

How do we achieve this with lambda calculus?

Predecessor: Wisdom Tooth Trick

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

Let us think about this a little differently

Predecessor: Wisdom Tooth Trick

$x = 0 \quad y = 0$

for $i = 1$ to n

$x_{new} = y_{old}$

$y_{new} = y_{old} + 1$

Predecessor: Wisdom Tooth Trick

$x = 0 \;\; y = 0$

for $i = 1$ to n

$$(x_{new}, \; y_{new}) = (y_{old}, y_{old} + 1)$$

Predecessor: Wisdom Tooth Trick

$(x, y) = (0 , 0)$

for $i = 1$ to n

$(x_{new} , y_{new}) = (y_{old} , y_{old} + 1)$

Predecessor: Wisdom Tooth Trick

(x, y) = pair 0 0

for i = 1 to n

$(x_{new}, \ y_{new}) = \text{pair snd}\ (x,y)\ \ (\text{snd}(x,y) + 1)$

Warning! This is not lambda calculus notation

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

Predecessor: Wisdom Tooth Trick

// Initial pair

```
zz = pair 0 0
```

Predecessor: Wisdom Tooth Trick

// Initial pair

```
zz = pair 0 0
```

// body of the loop

```
ss = λp. pair (snd p) (succ (snd p))
```

Predecessor: Wisdom Tooth Trick

// Initial pair

zz = pair 0 0

// body of the loop

ss = λp. pair (snd p) (succ (snd p))

// apply n times and extract x

prd = λn. fst (n ss zz)

Predecessor example

// Initial pair

zz = pair 0 0

// body of the loop

ss = λp. pair (snd p) (succ (snd p))

// apply n times and extract x

prd = λn. fst (n ss zz)

prd 4 = (λn. fst (n ss zz)) 4

= fst (4 ss zz)

= fst (ss (ss (ss (ss zz))))

= fst (ss (ss (ss (ss (pair 0 0)))))

= fst (ss (ss (ss (pair 0 1))))

= fst (ss (ss (pair 1 2)))

= fst (ss (pair 2 3))

= fst (pair 3 4)

= 3

Subtraction

minus = = λm. λn. n prd m

iszero

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

iszero

- iszero = λn. n (?) tru

iszero

- iszero = λn. n (?) tru
- iszero 0 = 0 (?) tru = tru

iszero

- iszero = λn. n (?) tru
- iszero 0 = 0 (?) tru = tru
- iszero n = n (?) tru $\rightarrow$ (?) ((?) ((?) ... tru))...))

iszero

- iszero = λn. n (?) tru
- iszero 0 = 0 (?) tru = tru
- iszero n = n (?) tru $\rightarrow$ (?) ((?) ((?) ... tru))...))
- We want repeated applications of (?) to result in fls always.

iszero

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

$$\boxed{\text{iszero} = \lambda n.\ n\ (\lambda x.\ \text{fls})\ \text{tru}}$$

Example. iszero 3 = (λn. n (λx. fls) tru) 3
= 3 (λx. fls) tru
= (λx. fls) ((λx. fls) ((λx. fls) tru))
= (λx. fls) ((λx. fls) fls)
= (λx. 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 = λm. λn. and (gteq m n) (gteq n m)
- gteq = λm. λn. iszero (m prd n)

Example. gteq 4 3 = (λm. λn. 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 = (λx. x x) (λx. x x)
- If we try to reduce omega, we get omega!

Recursive definitions

Let us look at the factorial example, we can write

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

Recursive definitions

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 ....
                                         )
                               )
```

Recursive definition

- 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>

Recursive definition

- 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 = λf. <body containing f>

  In the case of factorial this will look as follows:
  - g = λfct. λn. if = n 0 then 1
    else (times n (fct (prd n)))

Recursive definition

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

– **fix** = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))

Recursive definition

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

- **fix** = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))

If we apply fix to the function

- g = λfct. λn. if = n 0 then 1
  else (times n (fct (prd n)))

Recursive definition

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

- **fix** = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))

If we apply fix to the function

- g = λfct. λn. if = n 0 then 1
  else (times n (fct (prd n)))

We get the expansion

```
factorial = fix g =  if (n = 0) then 1
                 else n * ( if ((prd n) = 0 then 1
                            else (prd n) * (  if (prd (prd n)) = 0 then 1
                                 else ....
                               )
                          )
```

Example

$g = \lambda fct.\ \lambda n.$ 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

Example

g = λfct. λn. if = n 0 then 1
else (times n (fct (prd n)))

factorial = fix g = (λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))) g

factorial 2 = (fix g) 2
= ((λ**f**. (λx. **f** (λy. x x y)) (λx. **f** (λy. x x y))) g) 2

Example

g = λfct. λn. if = n 0 then 1
else (times n (fct (prd n)))

factorial = fix g = (λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))) g

factorial 2 = (fix g) 2
= ((λ**f**. (λx. **f** (λy. x x y)) (λx. **f** (λy. x x y))) g) 2
= ((λx. **g** (λy. x x y)) (λx. **g** (λy. x x y))) 2

Example

$g = \lambda fct.\ \lambda n.$ 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
$= (\ (\lambda \mathbf{f}.\ \ (\lambda x.\ \mathbf{f}\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ \mathbf{f}\ (\lambda y.\ x\ x\ y))\ )\ g\ )\ 2$
$= (\ (\lambda x.\ \mathbf{g}\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ \mathbf{g}\ (\lambda y.\ x\ x\ y)))\ 2$

$\mathbf{h} = \quad (\lambda x.\ \mathbf{g}\ (\lambda y.\ x\ x\ y))$

Example

g = λfct. λn. if = n 0 then 1
else (times n (fct (prd n)))

factorial = fix g = (λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))) g

factorial 2 = (fix g) 2
= ((λ**f**. (λx. **f** (λy. x x y)) (λx. **f** (λy. x x y))) g) 2
= ((λx. **g** (λy. x x y)) (λx. **g** (λy. x x y))) 2
= **(h h) 2**

h = (λx. **g** (λy. x x y))

Example

$g = \lambda fct.\ \lambda n.$ 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 $= (\text{fix } g)\ 2$
$= (\ (\lambda \mathbf{f}.\ \ (\lambda x.\ \mathbf{f}\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ \mathbf{f}\ (\lambda y.\ x\ x\ y))\ )\ g\ )\ 2$
$= (\ (\lambda x.\ \mathbf{g}\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ \mathbf{g}\ (\lambda y.\ x\ x\ y)))\ 2$
$= \mathbf{(h\ h)\ 2}$
$= ((\lambda x.\ \mathbf{g}\ (\lambda y.\ x\ x\ y))\ \mathbf{h})\ 2$

Example

$g = \lambda fct.\ \lambda n.\ \text{if} = n\ 0\ \text{then}\ 1$
$\qquad\qquad\qquad \text{else}\ (\text{times}\ n\ (fct\ (prd\ n)))$

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

$$
\begin{aligned}
\text{factorial}\ 2 \quad &= (\text{fix}\ g)\ 2 \\
&= (\ (\lambda \mathbf{f}.\ (\lambda x.\ \mathbf{f}\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ \mathbf{f}\ (\lambda y.\ x\ x\ y))\ )\ g\ )\ 2 \\
&= (\ (\lambda x.\ \mathbf{g}\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ \mathbf{g}\ (\lambda y.\ x\ x\ y)))\ 2 \\
&= \mathbf{(h\ h)\ 2} \\
&= ((\lambda x.\ \mathbf{g}\ (\lambda y.\ x\ x\ y))\ \mathbf{h})\ 2 \\
&= (\mathbf{g}\ (\lambda y.\ \mathbf{h}\ \mathbf{h}\ y))\ 2
\end{aligned}
$$

Example

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

Example

g = λfct. λn. if = n 0 then 1
else (times n (fct (prd n)))

factorial = fix g = (λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))) g

factorial 2 = (fix g) 2
= ((λ**f**. (λx. **f** (λy. x x y)) (λx. **f** (λy. x x y))) g) 2
= ((λx. **g** (λy. x x y)) (λx. **g** (λy. x x y))) 2
= **(h h) 2**
= ((λx. **g** (λy. x x y)) **h**) 2
= (**g** (λy. **h h** y)) 2
= ((λfct. λn. if = n 0 then 1 else (times n (fct (prd n)))) (λy. **h h** y)) 2
= (λn. if = n 0 then 1 else (times n ((λy. **h h** y)) (prd n)))) 2
= if = 2 0 then 1 else (times 2 ((λy. **h h** y)) (prd 2)))

Example

$g = \lambda fct.\ \lambda n.$ 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
= $(\ (\lambda f.\ \ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ )\ g\ )\ 2$
= $(\ (\lambda x.\ g\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ g\ (\lambda y.\ x\ x\ y)))\ 2$
= $(h\ h)\ 2$
= $((\lambda x.\ g\ (\lambda y.\ x\ x\ y))\ h)\ 2$
= $(g\ (\lambda y.\ h\ h\ y))\ 2$
= $((\lambda fct.\ \lambda n.$ if $= n\ 0$ 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)))

Example

g = λfct. λn. if = n 0 then 1
else (times n (fct (prd n)))

factorial = fix g = (λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))) g

factorial 2 = (fix g) 2
= ((λ**f**. (λx. **f** (λy. x x y)) (λx. **f** (λy. x x y))) g) 2
= ((λx. **g** (λy. x x y)) (λx. **g** (λy. x x y))) 2
= **(h h) 2**
= ((λx. **g** (λy. x x y)) **h**) 2
= (**g** (λy. **h h** y)) 2
= ((λfct. λn. if = n 0 then 1 else (times n (fct (prd n)))) (λy. **h h** y)) 2
= if = 2 0 then 1 else (times 2 ((λy. h h y) (prd 2)))
= if = 2 0 then 1 else (times 2 **((h h) (prd 2))**)

Recursive definition

In general:

– g = λf. <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 $M \rightarrow^* M_1$ and $M \rightarrow^* M_2$ then there exists N such that $M_1 \rightarrow^* N$ and $M_2 \rightarrow^* N$

$$\begin{tikzcd} & M \arrow[dl, dashed] \arrow[dr, dashed] & \\ M_1 \arrow[dr, dashed] & & M_2 \arrow[dl, dashed] \\ & N & \end{tikzcd}$$

η-reduction

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

η-reduction

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

- Justification: (λx. c x) a $\rightarrow^{\beta}$ c a