

Functional Programming

WS 2014/15

Cezary Kaliszyk (VO+PS)

Yann Savoye (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

week 06



Overview

- Week 6 - λ Calculus, Evaluation Strategies
 - Summary of Week 5
 - λ -Calculus - Data Types
 - Evaluation Strategies



Overview

- Week 6 - λ Calculus, Evaluation Strategies
 - Summary of Week 5
 - λ -Calculus - Data Types
 - Evaluation Strategies



λ -Calculus

λ -Terms

$$t ::= x \mid (\lambda x.t) \mid (t t)$$

λ -Calculus

λ -Terms

Variable
 $t ::= \overbrace{x} \mid (\lambda x.t) \mid (t t)$

λ -Calculus

λ -Terms

$$t ::= x \mid \underbrace{(\lambda x.t)}_{\text{Abstraction}} \mid (t t)$$

λ -Calculus

λ -Terms

Application

$$t ::= x \mid (\lambda x.t) \mid \overbrace{(t t)}$$

λ -Calculus

λ -Terms

$$t ::= x \mid (\lambda x.t) \mid (t t)$$

Example

$x y$

$\lambda x.x$

$\lambda xy.x$

$\lambda x.x x$

$(\lambda x.x) x$

λ -Calculus

λ -Terms

$$t ::= x \mid (\lambda x.t) \mid (t t)$$

Example

$x y$ $(x y)$
 $\lambda x.x$
 $\lambda xy.x$
 $\lambda x.x x$
 $(\lambda x.x) x$

λ -Calculus

λ -Terms

$$t ::= x \mid (\lambda x.t) \mid (t t)$$

Example

$x y$	$(x y)$	<i>"x applied to y"</i>
$\lambda x.x$		
$\lambda xy.x$		
$\lambda x.x x$		
$(\lambda x.x) x$		

λ -Calculus

λ -Terms

$$t ::= x \mid (\lambda x.t) \mid (t t)$$

Example

$x y$	$(x y)$	<i>"x applied to y"</i>
$\lambda x.x$	$(\lambda x.x)$	
$\lambda xy.x$		
$\lambda x.x x$		
$(\lambda x.x) x$		

λ -Calculus

λ -Terms

$$t ::= x \mid (\lambda x.t) \mid (t t)$$

Example

$x y$	$(x y)$	<i>"x applied to y"</i>
$\lambda x.x$	$(\lambda x.x)$	<i>"lambda x to x"</i>
$\lambda xy.x$		
$\lambda x.x x$		
$(\lambda x.x) x$		

λ -Calculus

λ -Terms

$$t ::= x \mid (\lambda x.t) \mid (t t)$$

Example

$x y$	$(x y)$	<i>"x applied to y"</i>
$\lambda x.x$	$(\lambda x.x)$	<i>"lambda x to x"</i>
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	
$\lambda x.x x$		
$(\lambda x.x) x$		

λ -Calculus

λ -Terms

$$t ::= x \mid (\lambda x.t) \mid (t t)$$

Example

$x y$	$(x y)$	<i>"x applied to y"</i>
$\lambda x.x$	$(\lambda x.x)$	<i>"lambda x to x"</i>
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	<i>"lambda x y to x"</i>
$\lambda x.x x$		
$(\lambda x.x) x$		

λ -Calculus

λ -Terms

$$t ::= x \mid (\lambda x.t) \mid (t t)$$

Example

$x y$	$(x y)$	<i>"x applied to y"</i>
$\lambda x.x$	$(\lambda x.x)$	<i>"lambda x to x"</i>
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	<i>"lambda x y to x"</i>
$\lambda x.x x$	$(\lambda x.(x x))$	
$(\lambda x.x) x$		

λ -Calculus

λ -Terms

$$t ::= x \mid (\lambda x.t) \mid (t t)$$

Example

$x y$	$(x y)$	<i>"x applied to y"</i>
$\lambda x.x$	$(\lambda x.x)$	<i>"lambda x to x"</i>
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	<i>"lambda x y to x"</i>
$\lambda x.x x$	$(\lambda x.(x x))$	<i>"lambda x to (x applied to x)"</i>
$(\lambda x.x) x$		

λ -Calculus

λ -Terms

$$t ::= x \mid (\lambda x.t) \mid (t t)$$

Example

$x y$	$(x y)$	<i>"x applied to y"</i>
$\lambda x.x$	$(\lambda x.x)$	<i>"lambda x to x"</i>
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	<i>"lambda x y to x"</i>
$\lambda x.x x$	$(\lambda x.(x x))$	<i>"lambda x to (x applied to x)"</i>
$(\lambda x.x) x$	$((\lambda x.x) x)$	

λ -Calculus

λ -Terms

$$t ::= x \mid (\lambda x.t) \mid (t t)$$

Example

$x y$	$(x y)$	<i>"x applied to y"</i>
$\lambda x.x$	$(\lambda x.x)$	<i>"lambda x to x"</i>
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	<i>"lambda x y to x"</i>
$\lambda x.x x$	$(\lambda x.(x x))$	<i>"lambda x to (x applied to x)"</i>
$(\lambda x.x) x$	$((\lambda x.x) x)$	<i>"(lambda x to x) applied to x"</i>

λ -Calculus (cont'd)

β -Reduction

the term s (β -)reduces to the term t in one step, i.e.,

$$s \rightarrow_{\beta} t$$

iff there exist context C and terms u, v s.t.

$$s = C[(\lambda x.u) v] \quad \text{and} \quad t = C[u\{x/v\}]$$

λ -Calculus (cont'd)

β -Reduction

the term s (β -)reduces to the term t in one step, i.e.,

$$\underbrace{s \rightarrow t}_{(\beta\text{-})\text{step}}$$

iff there exist context C and terms u, v s.t.

$$s = C[(\lambda x.u) v] \quad \text{and} \quad t = C[u\{x/v\}]$$

λ -Calculus (cont'd)

β -Reduction

the term s (β -)reduces to the term t in one step, i.e.,

$$s \rightarrow_{\beta} t$$

iff there exist context C and terms u, v s.t.

$$s = C[(\lambda x.u) v] \quad \text{and} \quad t = C[u\{x/v\}]$$

Example

$$K \stackrel{\text{def}}{=} \lambda xy.x$$

$$I \stackrel{\text{def}}{=} \lambda x.x$$

$$\Omega \stackrel{\text{def}}{=} (\lambda x.x x) (\lambda x.x x)$$

Overview

- Week 6 - λ Calculus, Evaluation Strategies
 - Summary of Week 5
 - λ -Calculus - Data Types
 - Evaluation Strategies



This Week

Practice I

OCaml introduction, lists, strings, trees

Theory I

lambda-calculus, evaluation strategies, induction,
reasoning about functional programs

Practice II

efficiency, tail-recursion, combinator-parsing, dynamic programming

Theory II

type checking, type inference

Advanced Topics

lazy evaluation, infinite data structures, monads, ...

Overview

- Week 6 - λ Calculus, Evaluation Strategies
 - Summary of Week 5
 - λ -Calculus - Data Types
 - Evaluation Strategies



Booleans and Conditionals

OCaml

- `true`
- `false`
- `if b then t else e`

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if b then t else e`

λ -Calculus

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if b then t else e`

λ -Calculus

- `true` $\stackrel{\text{def}}{=} \lambda xy.x$

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if b then t else e`

λ -Calculus

- `true` $\stackrel{\text{def}}{=} \lambda xy.x$
- `false` $\stackrel{\text{def}}{=} \lambda xy.y$

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if b then t else e`

λ -Calculus

- `true` $\stackrel{\text{def}}{=} \lambda xy.x$
- `false` $\stackrel{\text{def}}{=} \lambda xy.y$
- `if` $\stackrel{\text{def}}{=} \lambda xyz.x y z$

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if b then t else e`

λ -Calculus

- $\text{true} \stackrel{\text{def}}{=} \lambda xy.x$
- $\text{false} \stackrel{\text{def}}{=} \lambda xy.y$
- $\text{if} \stackrel{\text{def}}{=} \lambda xyz.x y z$

Example

$$\text{if true } t e \rightarrow_{\beta}^+ t$$

$$\text{if false } t e \rightarrow_{\beta}^+ e$$

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if b then t else e`

λ -Calculus

- `true` $\stackrel{\text{def}}{=} \lambda xy.x$
- `false` $\stackrel{\text{def}}{=} \lambda xy.y$
- `if` $\stackrel{\text{def}}{=} \lambda xyz.x y z$

Example

`if true t e` \rightarrow_{β}^{+} `true t e`

`if false t e` \rightarrow_{β}^{+}

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if b then t else e`

λ -Calculus

- $\text{true} \stackrel{\text{def}}{=} \lambda xy.x$
- $\text{false} \stackrel{\text{def}}{=} \lambda xy.y$
- $\text{if} \stackrel{\text{def}}{=} \lambda xyz.x y z$

Example

$\text{if true } t e \rightarrow_{\beta}^{+} \text{true } t e \rightarrow_{\beta}^{+} t$

$\text{if false } t e \rightarrow_{\beta}^{+}$

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if b then t else e`

λ -Calculus

- $\text{true} \stackrel{\text{def}}{=} \lambda xy.x$
- $\text{false} \stackrel{\text{def}}{=} \lambda xy.y$
- $\text{if} \stackrel{\text{def}}{=} \lambda xyz.x y z$

Example

$$\text{if true } t e \rightarrow_{\beta}^{+} \text{true } t e \rightarrow_{\beta}^{+} t$$

$$\text{if false } t e \rightarrow_{\beta}^{+} \text{false } t e$$

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if b then t else e`

λ -Calculus

- $\text{true} \stackrel{\text{def}}{=} \lambda xy.x$
- $\text{false} \stackrel{\text{def}}{=} \lambda xy.y$
- $\text{if} \stackrel{\text{def}}{=} \lambda xyz.x y z$

Example

$$\text{if true } t e \rightarrow_{\beta}^{+} \text{true } t e \rightarrow_{\beta}^{+} t$$

$$\text{if false } t e \rightarrow_{\beta}^{+} \text{false } t e \rightarrow_{\beta}^{+} e$$

Natural Numbers

Definition

$$s^0 t \stackrel{\text{def}}{=} t$$

$$s^{n+1} t \stackrel{\text{def}}{=} s (s^n t)$$

OCaml vs. λ -Calculus

0

1

n

(+)

(*)

(**)

Natural Numbers

Definition

$$s^0 t \stackrel{\text{def}}{=} t$$

$$s^{n+1} t \stackrel{\text{def}}{=} s (s^n t)$$

OCaml vs. λ -Calculus

$$0 \quad \bar{0} \stackrel{\text{def}}{=} \lambda f x. x$$

1

n

(+)

(*)

(**)

Natural Numbers

Definition

$$s^0 t \stackrel{\text{def}}{=} t$$

$$s^{n+1} t \stackrel{\text{def}}{=} s (s^n t)$$

OCaml vs. λ -Calculus

$$0 \quad \bar{0} \stackrel{\text{def}}{=} \lambda f x. x$$

$$1 \quad \bar{1} \stackrel{\text{def}}{=} \lambda f x. f x$$

n

(+)

(*)

(**)

Natural Numbers

Definition

$$s^0 t \stackrel{\text{def}}{=} t$$

$$s^{n+1} t \stackrel{\text{def}}{=} s (s^n t)$$

OCaml vs. λ -Calculus

$$\begin{array}{ll}
 0 & \bar{0} \stackrel{\text{def}}{=} \lambda f x. x \\
 1 & \bar{1} \stackrel{\text{def}}{=} \lambda f x. f x \\
 n & \bar{n} \stackrel{\text{def}}{=} \lambda f x. f^n x \\
 (+) & \\
 (*) & \\
 (**) &
 \end{array}$$

Natural Numbers

Definition

$$s^0 t \stackrel{\text{def}}{=} t$$

$$s^{n+1} t \stackrel{\text{def}}{=} s (s^n t)$$

OCaml vs. λ -Calculus

$$0 \quad \bar{0} \stackrel{\text{def}}{=} \lambda f x. x$$

$$1 \quad \bar{1} \stackrel{\text{def}}{=} \lambda f x. f x$$

$$n \quad \bar{n} \stackrel{\text{def}}{=} \lambda f x. f^n x$$

$$(+) \quad \text{add} \stackrel{\text{def}}{=} \lambda m n f x. m f (n f x)$$

$$(*)$$

$$(**)$$

Natural Numbers

Definition

$$s^0 t \stackrel{\text{def}}{=} t$$

$$s^{n+1} t \stackrel{\text{def}}{=} s (s^n t)$$

OCaml vs. λ -Calculus

$$0 \quad \bar{0} \stackrel{\text{def}}{=} \lambda f x. x$$

$$1 \quad \bar{1} \stackrel{\text{def}}{=} \lambda f x. f x$$

$$n \quad \bar{n} \stackrel{\text{def}}{=} \lambda f x. f^n x$$

$$(+) \quad \text{add} \stackrel{\text{def}}{=} \lambda m n f x. m f (n f x)$$

$$(*) \quad \text{mul} \stackrel{\text{def}}{=} \lambda m n f. m (n f)$$

$$(**)$$

Natural Numbers

Definition

$$s^0 t \stackrel{\text{def}}{=} t$$

$$s^{n+1} t \stackrel{\text{def}}{=} s (s^n t)$$

OCaml vs. λ -Calculus

$$0 \quad \bar{0} \stackrel{\text{def}}{=} \lambda f x. x$$

$$1 \quad \bar{1} \stackrel{\text{def}}{=} \lambda f x. f x$$

$$n \quad \bar{n} \stackrel{\text{def}}{=} \lambda f x. f^n x$$

$$(+) \quad \text{add} \stackrel{\text{def}}{=} \lambda m n f x. m f (n f x)$$

$$(*) \quad \text{mul} \stackrel{\text{def}}{=} \lambda m n f. m (n f)$$

$$(**) \quad \text{exp} \stackrel{\text{def}}{=} \lambda m n. n m$$

Natural Numbers

Definition

$$s^0 t \stackrel{\text{def}}{=} t$$

$$s^{n+1} t \stackrel{\text{def}}{=} s (s^n t)$$

OCaml vs. λ -Calculus

$$0 \quad \bar{0} \stackrel{\text{def}}{=} \lambda f x. x$$

$$1 \quad \bar{1} \stackrel{\text{def}}{=} \lambda f x. f x$$

$$n \quad \bar{n} \stackrel{\text{def}}{=} \lambda f x. f^n x$$

$$(+) \quad \text{add} \stackrel{\text{def}}{=} \lambda m n f x. m f (n f x)$$

$$(*) \quad \text{mul} \stackrel{\text{def}}{=} \lambda m n f. m (n f)$$

$$(**) \quad \text{exp} \stackrel{\text{def}}{=} \lambda m n. n m$$

Example

$$\text{add } \bar{1} \bar{1} \rightarrow_{\beta}^*$$

Natural Numbers

Definition

$$s^0 t \stackrel{\text{def}}{=} t$$

$$s^{n+1} t \stackrel{\text{def}}{=} s (s^n t)$$

OCaml vs. λ -Calculus

$$0 \quad \bar{0} \stackrel{\text{def}}{=} \lambda f x. x$$

$$1 \quad \bar{1} \stackrel{\text{def}}{=} \lambda f x. f x$$

$$n \quad \bar{n} \stackrel{\text{def}}{=} \lambda f x. f^n x$$

$$(+) \quad \text{add} \stackrel{\text{def}}{=} \lambda m n f x. m f (n f x)$$

$$(*) \quad \text{mul} \stackrel{\text{def}}{=} \lambda m n f. m (n f)$$

$$(**) \quad \text{exp} \stackrel{\text{def}}{=} \lambda m n. n m$$

Example

$$\text{add } \bar{1} \bar{1} \rightarrow_{\beta}^* \bar{2}$$

Pairs

OCaml vs. λ -Calculus

```
fun x y -> (x,y)
```

```
fst
```

```
snd
```

Pairs

OCaml vs. λ -Calculus

```
fun x y -> (x,y)  pair  $\stackrel{\text{def}}{=} \lambda xyf.f x y$   
fst  
snd
```

Pairs

OCaml vs. λ -Calculus

```
fun x y -> (x,y)   pair  $\stackrel{\text{def}}{=} \lambda xyf.f x y$   
fst               fst  $\stackrel{\text{def}}{=} \lambda p.p \text{ true}$   
snd
```

Pairs

OCaml vs. λ -Calculus

<code>fun x y -> (x,y)</code>	<code>pair</code> $\stackrel{\text{def}}{=} \lambda xyf.f x y$
<code>fst</code>	<code>fst</code> $\stackrel{\text{def}}{=} \lambda p.p \text{ true}$
<code>snd</code>	<code>snd</code> $\stackrel{\text{def}}{=} \lambda p.p \text{ false}$

Pairs

OCaml vs. λ -Calculus

<code>fun x y -> (x,y)</code>	$\text{pair} \stackrel{\text{def}}{=} \lambda xyf.f x y$
<code>fst</code>	$\text{fst} \stackrel{\text{def}}{=} \lambda p.p \text{ true}$
<code>snd</code>	$\text{snd} \stackrel{\text{def}}{=} \lambda p.p \text{ false}$

Example

$$\text{fst} (\text{pair } \bar{m} \bar{n}) \rightarrow_{\beta}^*$$

Pairs

OCaml vs. λ -Calculus

<code>fun x y -> (x,y)</code>	$\text{pair} \stackrel{\text{def}}{=} \lambda xyf.f x y$
<code>fst</code>	$\text{fst} \stackrel{\text{def}}{=} \lambda p.p \text{ true}$
<code>snd</code>	$\text{snd} \stackrel{\text{def}}{=} \lambda p.p \text{ false}$

Example

$$\text{fst} (\text{pair } \bar{m} \bar{n}) \rightarrow_{\beta}^* \bar{m}$$

Lists

OCaml vs. λ -Calculus

```
::  
hd  
tl  
[]  
fun x -> x = []
```

Lists

OCaml vs. λ -Calculus

<code>::</code>	<code>cons</code> $\stackrel{\text{def}}{=} \lambda xy.$	<code>pair x y</code>
<code>hd</code>		
<code>tl</code>		
<code>[]</code>		
<code>fun x -> x = []</code>		

Lists

OCaml vs. λ -Calculus

```
::                                cons  $\stackrel{\text{def}}{=} \lambda xy. \text{pair false (pair x y)}$   
hd  
tl  
[]  
fun x -> x = []
```

Lists

OCaml vs. λ -Calculus

```
::                cons  $\stackrel{\text{def}}{=} \lambda xy. \text{pair } \text{false } (\text{pair } x y)$   
hd                hd  $\stackrel{\text{def}}{=} \lambda z. \text{fst } (\text{snd } z)$   
tl  
[]  
fun x -> x = []
```

Lists

OCaml vs. λ -Calculus

```
::          cons  $\stackrel{\text{def}}{=} \lambda xy. \text{pair } \text{false} (\text{pair } x y)$   
hd         hd  $\stackrel{\text{def}}{=} \lambda z. \text{fst } (\text{snd } z)$   
tl         tl  $\stackrel{\text{def}}{=} \lambda z. \text{snd } (\text{snd } z)$   
[]  
fun x -> x = []
```

Lists

OCaml vs. λ -Calculus

<code>::</code>	$\text{cons} \stackrel{\text{def}}{=} \lambda xy. \text{pair } \text{false} (\text{pair } x \ y)$
<code>hd</code>	$\text{hd} \stackrel{\text{def}}{=} \lambda z. \text{fst } (\text{snd } z)$
<code>tl</code>	$\text{tl} \stackrel{\text{def}}{=} \lambda z. \text{snd } (\text{snd } z)$
<code>[]</code>	$\text{nil} \stackrel{\text{def}}{=} \lambda x. x$
<code>fun x -> x = []</code>	

Lists

OCaml vs. λ -Calculus

<code>::</code>	$\text{cons} \stackrel{\text{def}}{=} \lambda xy. \text{pair false (pair } x \ y)$
<code>hd</code>	$\text{hd} \stackrel{\text{def}}{=} \lambda z. \text{fst (snd } z)$
<code>tl</code>	$\text{tl} \stackrel{\text{def}}{=} \lambda z. \text{snd (snd } z)$
<code>[]</code>	$\text{nil} \stackrel{\text{def}}{=} \lambda x. x$
<code>fun x -> x = []</code>	$\text{null} \stackrel{\text{def}}{=} \text{fst}$

Lists

OCaml vs. λ -Calculus

<code>::</code>	$\text{cons} \stackrel{\text{def}}{=} \lambda xy. \text{pair false (pair } x \ y)$
<code>hd</code>	$\text{hd} \stackrel{\text{def}}{=} \lambda z. \text{fst (snd } z)$
<code>tl</code>	$\text{tl} \stackrel{\text{def}}{=} \lambda z. \text{snd (snd } z)$
<code>[]</code>	$\text{nil} \stackrel{\text{def}}{=} \lambda x. x$
<code>fun x -> x = []</code>	$\text{null} \stackrel{\text{def}}{=} \text{fst}$

Example

$$\text{null nil} \rightarrow_{\beta}^*$$

Lists

OCaml vs. λ -Calculus

<code>::</code>	$\text{cons} \stackrel{\text{def}}{=} \lambda xy. \text{pair } \text{false} (\text{pair } x \ y)$
<code>hd</code>	$\text{hd} \stackrel{\text{def}}{=} \lambda z. \text{fst} (\text{snd } z)$
<code>tl</code>	$\text{tl} \stackrel{\text{def}}{=} \lambda z. \text{snd} (\text{snd } z)$
<code>[]</code>	$\text{nil} \stackrel{\text{def}}{=} \lambda x. x$
<code>fun x -> x = []</code>	$\text{null} \stackrel{\text{def}}{=} \text{fst}$

Example

$$\text{null nil} \rightarrow_{\beta}^* \text{true}$$

Recursion

OCaml

```
let rec length x = if x = [] then 0
                   else 1 + length(tl x)
```

λ -Calculus

$$\text{length} \stackrel{\text{def}}{=} \lambda x. \text{if } (\text{null } x) \bar{0} (\text{add } \bar{1} (\text{length } (\text{tl } x)))$$

Recursion

OCaml

```
let rec length x = if x = [] then 0
                   else 1 + length(tl x)
```

λ -Calculus

$$\text{length} \stackrel{\text{def}}{=} \lambda x. \text{if } (\text{null } x) \bar{0} (\text{add } \bar{1} (\text{length } (\text{tl } x)))$$

Recursion

OCaml

```
let rec length x = if x = [] then 0
                   else 1 + length(tl x)
```

λ -Calculus

$$\text{length} \stackrel{\text{def}}{=} \lambda f x. \text{if } (\text{null } x) \bar{0} (\text{add } \bar{1} (f (\text{tl } x)))$$

Recursion

OCaml

```
let rec length x = if x = [] then 0
                   else 1 + length(tl x)
```

λ -Calculus

$$\text{length} \stackrel{\text{def}}{=} \mathbf{Y} (\lambda f x. \text{if } (\text{null } x) \bar{0} (\text{add } \bar{1} (f (\text{tl } x))))$$

Recursion

OCaml

```
let rec length x = if x = [] then 0
                  else 1 + length(tl x)
```

λ -Calculus

$$\text{length} \stackrel{\text{def}}{=} \mathbf{Y} (\lambda f x. \text{if } (\text{null } x) \bar{0} (\text{add } \bar{1} (f (\text{tl } x))))$$

Definition (Y-combinator)

$$\mathbf{Y} \stackrel{\text{def}}{=} \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

\mathbf{Y} has fixed point property, i.e., for all $t \in \mathcal{T}(\mathcal{V})$

$$\mathbf{Y} t \leftrightarrow^* t (\mathbf{Y} t)$$

Recursion

OCaml

```
let rec length x = if x = [] then 0
                  else 1 + length(tl x)
```

λ -Calculus

$$\text{length} \stackrel{\text{def}}{=} \mathbf{Y} (\lambda f x. \text{if } (\text{null } x) \bar{0} (\text{add } \bar{1} (f (\text{tl } x))))$$

Definition (Y-combinator)

$$\mathbf{Y} \stackrel{\text{def}}{=} \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

\mathbf{Y} has **fixed point property**, i.e., for all $t \in \mathcal{T}(\mathcal{V})$

$$\mathbf{Y} t \leftrightarrow^* t (\mathbf{Y} t)$$

Overview

- Week 6 - λ Calculus, Evaluation Strategies
 - Summary of Week 5
 - λ -Calculus - Data Types
 - Evaluation Strategies



Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows

`d (d 2)`

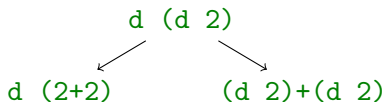
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows

`d (d 2)`
↙
`d (2+2)`

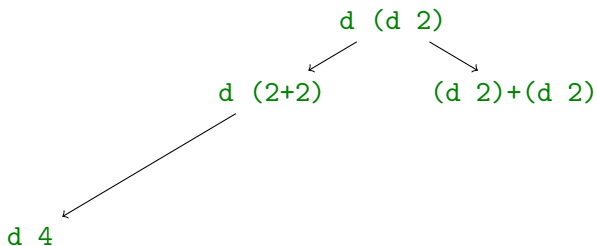
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



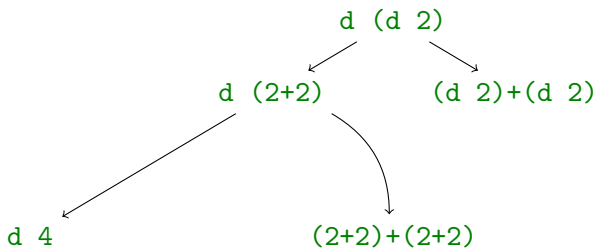
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



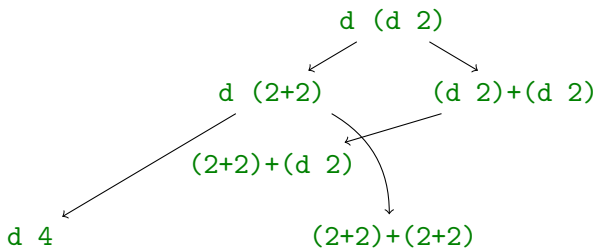
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



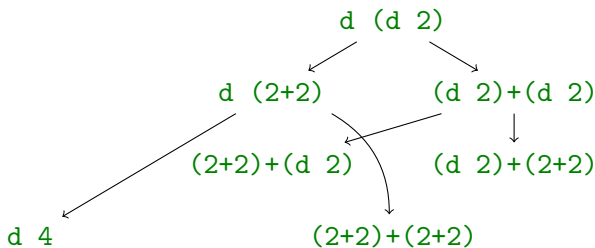
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



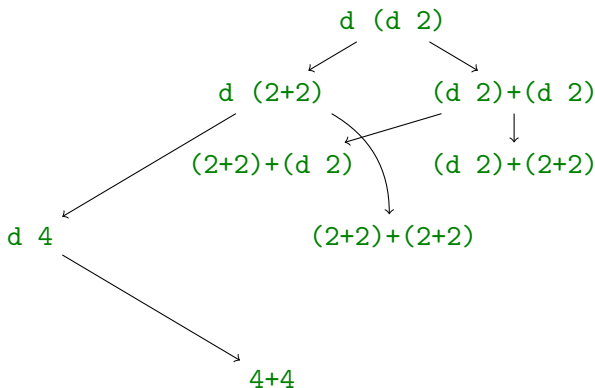
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



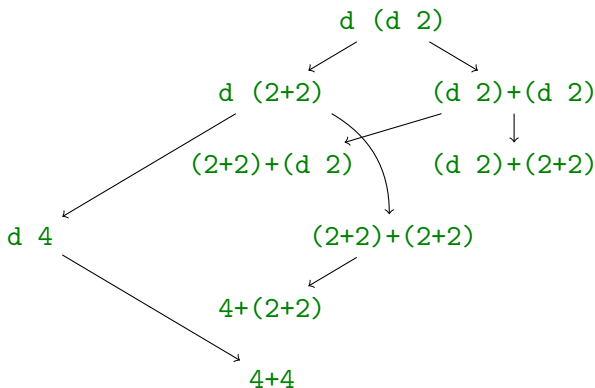
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



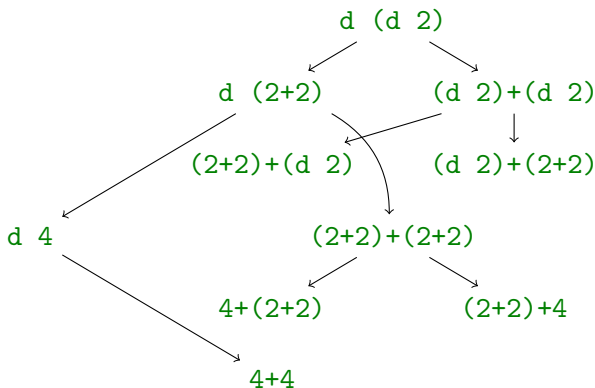
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



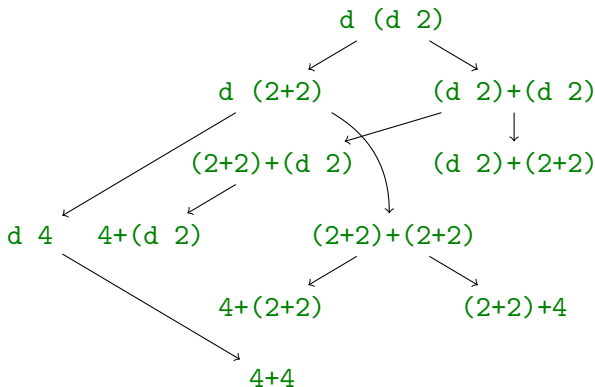
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



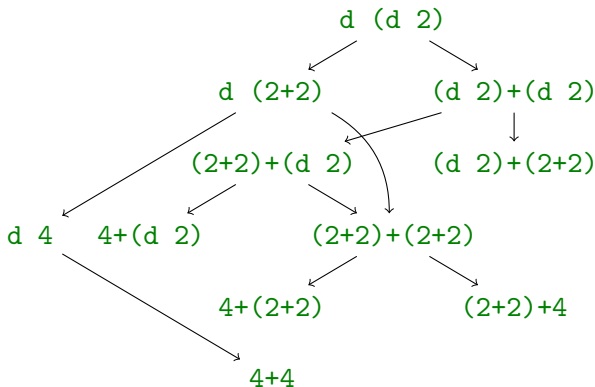
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



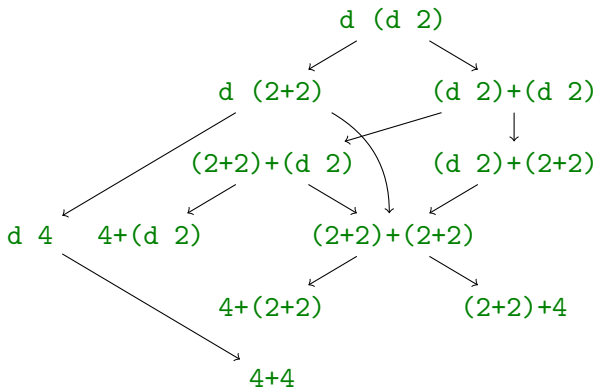
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



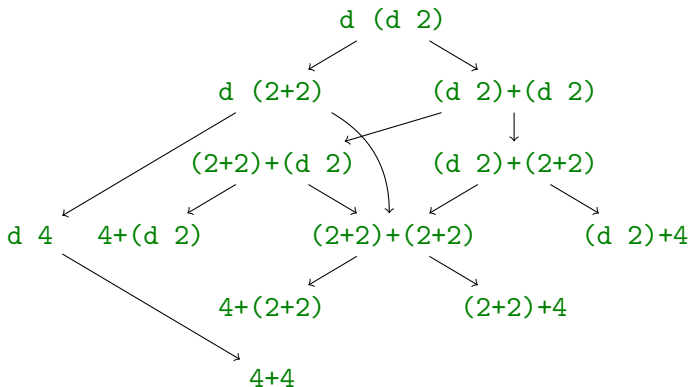
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



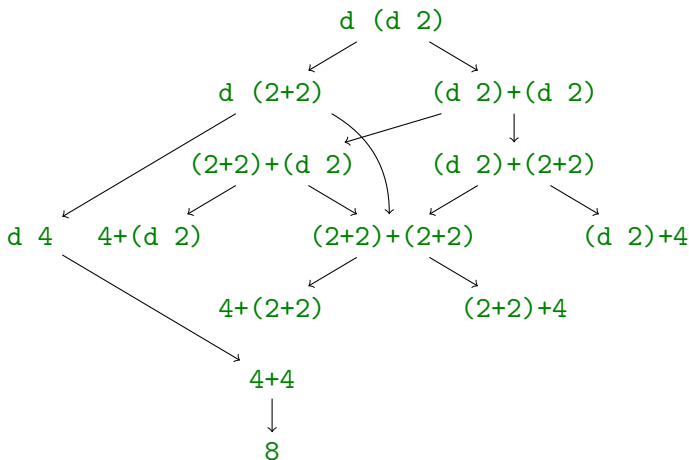
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



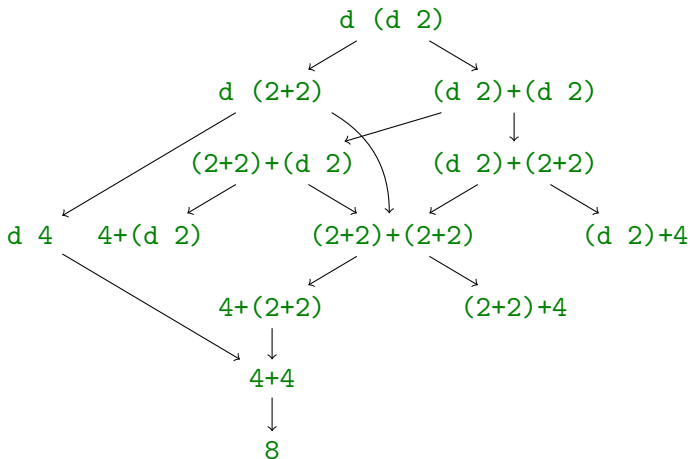
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



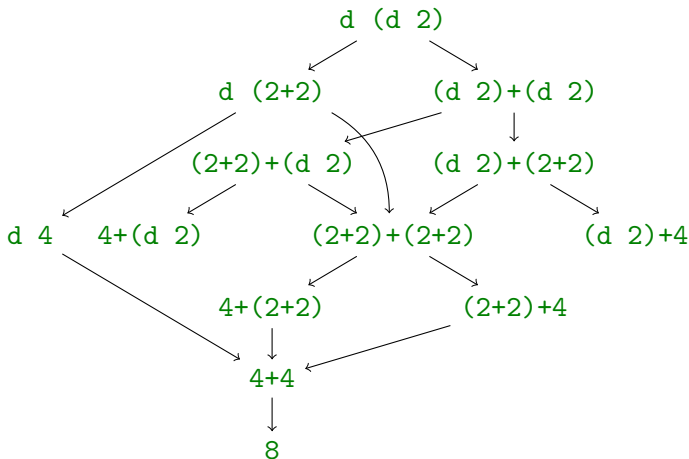
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



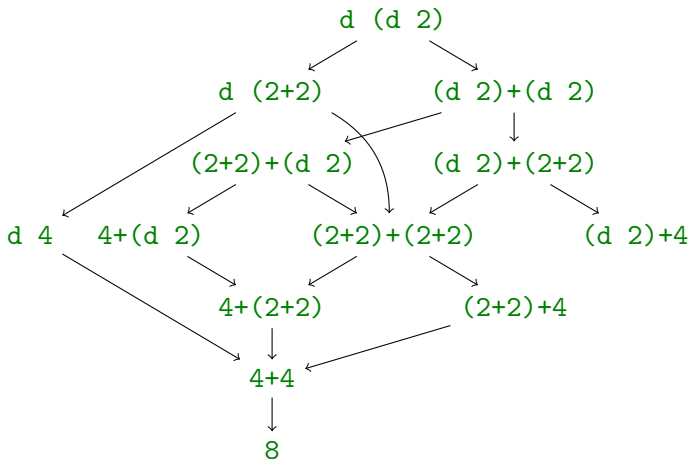
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



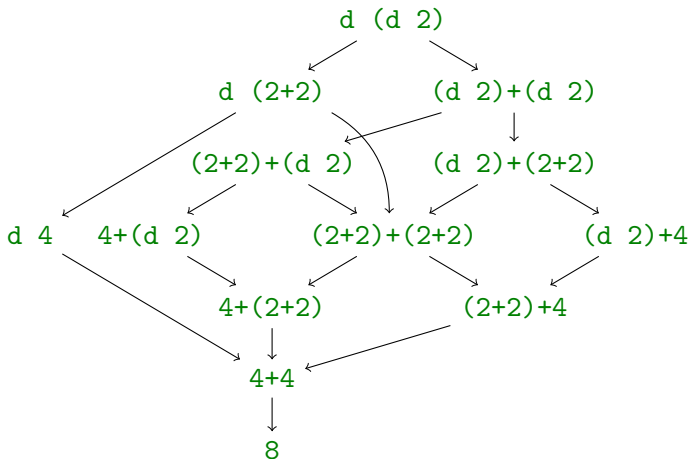
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



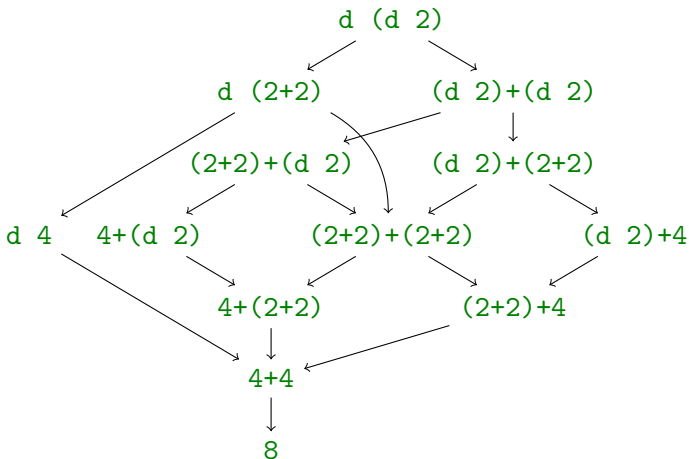
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows (9 possibilities)



Strategies

Strategy

- fixes evaluation order
- examples: **call-by-value** and **call-by-name**

Example

`let d x = x + x`

- call-by-value:

$$\begin{aligned} d (d 2) &\rightarrow d (2+2) \\ &\rightarrow d 4 \\ &\rightarrow 4 + 4 \\ &\rightarrow 8 \end{aligned}$$

- call-by-name:

$$\begin{aligned} d (d 2) &\rightarrow (d 2)+(d 2) \\ &\rightarrow (2+2)+(d 2) \\ &\rightarrow 4+(d 2) \\ &\rightarrow 4+(2+2) \\ &\rightarrow 4+4 \\ &\rightarrow 8 \end{aligned}$$

(Leftmost) Innermost Reduction

- always reduce (leftmost) innermost redex

Definition

redex t of term u is **innermost** if it does not contain a redex as **proper** subterm, i.e.,

$$\nexists s \in \text{Sub}(t) \text{ s.t. } s \neq t \text{ and } s \text{ is a redex}$$

(Leftmost) Innermost Reduction

- always reduce (leftmost) innermost redex

Definition

redex t of term u is **innermost** if it does not contain a redex as **proper** subterm, i.e.,

$$\nexists s \in \text{Sub}(t) \text{ s.t. } s \neq t \text{ and } s \text{ is a redex}$$

Example

Consider $t = \underline{(\lambda x. (\lambda y. y) x)} z$

(Leftmost) Innermost Reduction

- always reduce (leftmost) innermost redex

Definition

redex t of term u is **innermost** if it does not contain a redex as **proper** subterm, i.e.,

$$\nexists s \in \text{Sub}(t) \text{ s.t. } s \neq t \text{ and } s \text{ is a redex}$$

Example

Consider $t = (\lambda x. (\lambda y. y) x) z$

- $(\lambda y. y) x$ is innermost redex

(Leftmost) Innermost Reduction

- always reduce (leftmost) innermost redex

Definition

redex t of term u is **innermost** if it does not contain a redex as **proper** subterm, i.e.,

$$\nexists s \in \text{Sub}(t) \text{ s.t. } s \neq t \text{ and } s \text{ is a redex}$$

Example

Consider $t = (\lambda x. (\lambda y. y) x) z$

- $(\lambda y. y) x$ is innermost redex
- $(\lambda x. (\lambda y. y) x) z$ is redex, but not innermost

(Leftmost) Outermost Reduction

- always reduce leftmost outermost redex

Definition

redex t of term u is **outermost** if it is not a **proper** subterm of some other redex in u , i.e.,

$$\nexists s \in \text{Sub}(u) \text{ s.t. } s \text{ is a redex and } t \in \text{Sub}(s) \text{ and } s \neq t$$

(Leftmost) Outermost Reduction

- always reduce leftmost outermost redex

Definition

redex t of term u is **outermost** if it is not a **proper** subterm of some other redex in u , i.e.,

$$\nexists s \in \text{Sub}(u) \text{ s.t. } s \text{ is a redex and } t \in \text{Sub}(s) \text{ and } s \neq t$$

Example

Consider $t = \underline{(\lambda x. (\lambda y. y) x)} z$

(Leftmost) Outermost Reduction

- always reduce leftmost outermost redex

Definition

redex t of term u is **outermost** if it is not a **proper** subterm of some other redex in u , i.e.,

$$\nexists s \in \text{Sub}(u) \text{ s.t. } s \text{ is a redex and } t \in \text{Sub}(s) \text{ and } s \neq t$$

Example

Consider $t = (\lambda x. (\lambda y. y) x) z$

- $(\lambda x. (\lambda y. y) x) z$ is outermost redex

(Leftmost) Outermost Reduction

- always reduce leftmost outermost redex

Definition

redex t of term u is **outermost** if it is not a **proper** subterm of some other redex in u , i.e.,

$$\nexists s \in \text{Sub}(u) \text{ s.t. } s \text{ is a redex and } t \in \text{Sub}(s) \text{ and } s \neq t$$

Example

Consider $t = (\lambda x. (\lambda y. y) x) z$

- $(\lambda x. (\lambda y. y) x) z$ is outermost redex
- $(\lambda y. y) x$ is redex, but not outermost

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Definition (Weak head normal form)

term t is in **weak head normal form** (*WHNF*) iff

$$t \neq u v$$

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Definition (Weak head normal form)

term t is in **weak head normal form** (*WHNF*) iff

$$t \neq u \ v$$

Example (WHNF)

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

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Definition (Weak head normal form)

term t is in **weak head normal form** (*WHNF*) iff

$$t \neq u v$$

Example (WHNF)

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

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Definition (Weak head normal form)

term t is in **weak head normal form** (WHNF) iff

$$t \neq u \ v$$

Example (WHNF)

$\lambda x.x$ ✓ $(\lambda x.x) \ y$ ✗ $(\lambda x.x) \ y \ z$ $\lambda x.(\lambda y.y) \ x$ $x \ x$

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Definition (Weak head normal form)

term t is in **weak head normal form (WHNF)** iff

$$t \neq u \ v$$

Example (WHNF)

$\lambda x.x$ ✓ $(\lambda x.x) \ y$ ✗ $(\lambda x.x) \ y \ z$ ✗ $\lambda x.(\lambda y.y) \ x$ $x \ x$

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Definition (Weak head normal form)

term t is in **weak head normal form** (WHNF) iff

$$t \neq u \ v$$

Example (WHNF)

$\lambda x.x$ ✓ $(\lambda x.x) \ y$ ✗ $(\lambda x.x) \ y \ z$ ✗ $\lambda x.(\lambda y.y) \ x$ ✓ $x \ x$

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Definition (Weak head normal form)

term t is in **weak head normal form** (WHNF) iff

$$t \neq u \ v$$

Example (WHNF)

$\lambda x.x$ ✓ $(\lambda x.x) \ y$ ✗ $(\lambda x.x) \ y \ z$ ✗ $\lambda x.(\lambda y.y) \ x$ ✓ $x \ x$ ✗

Call-by-Name

- use outermost reduction
- corresponds to lazy evaluation (without memoization), e.g., Haskell
- slight modification: only reduce terms that are not in WHNF