Computational Logic

Functional Programming

WS 2012/13

Harald Zankl (VO)
Cezary Kaliszyk (PS) Thomas Sternagel (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

week 6

Overview

- Week 6 - λ Calculus, Evaluation Strategies
  - Summary of Week 5
  - λ-Calculus - Data Types
  - Evaluation Strategies
  - The λ Tree Tool `LTT`

Overview

λ-Calculus

λ-Terms

$$t ::= \quad x \quad | \, (\lambda x.t) \, | \, (t \;\; t)$$

λ-Calculus

λ-Terms

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

λ-Calculus

λ-Terms

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

λ-Calculus

λ-Terms

$$t ::= \quad x \quad | \, (\lambda x.t) \, | \, \overbrace{(t \;\; t)}^{\text{Application}}$$

λ-Calculus

λ-Terms

$$t ::= \quad x \quad | \, (\lambda x.t) \, | \, (t \;\; t)$$

Example

$x \;\; y$

$\lambda x.x$

$\lambda xy.x$

$\lambda x.x \;\; x$

$(\lambda x.x) \;\; x$

λ-Calculus

λ-Terms

$$t ::= \quad x \quad | \, (\lambda x.t) \, | \, (t \;\; t)$$

Example

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

λ-Calculus

λ-Terms

$$t ::= \quad x \quad | \, (\lambda x.t) \, | \, (t \;\; t)$$

Example

$x \; y$	$(x \; y)$	*"x applied to y"*
$\lambda x.x$		
$\lambda xy.x$		
$\lambda x.x \; x$		
$(\lambda x.x) \; x$		

λ-Calculus

λ-Terms

$$t ::= \quad x \quad | \, (\lambda x.t) \, | \, (t \;\; t)$$

Example

$x \; y$	$(x \; y)$	*"x applied to y"*
$\lambda x.x$	$(\lambda x.x)$	
$\lambda xy.x$		
$\lambda x.x \; x$		
$(\lambda x.x) \; x$		

λ-Calculus

λ-Terms

$$t ::= \quad x \quad | \, (\lambda x.t) \, | \, (t \;\; t)$$

Example

$x \; y$	$(x \; y)$	*"x applied to y"*
$\lambda x.x$	$(\lambda x.x)$	*"lambda x to x"*
$\lambda xy.x$		
$\lambda x.x \; x$		
$(\lambda x.x) \; x$		

λ-Calculus

λ-Terms

$$t ::= \quad x \quad | \, (\lambda x.t) \, | \, (t \;\; t)$$

Example

$x \; y$	$(x \; y)$	*"x applied to y"*
$\lambda x.x$	$(\lambda x.x)$	*"lambda x to x"*
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	
$\lambda x.x \; x$		
$(\lambda x.x) \; x$		

λ-Calculus

λ-Terms

$$t ::= \quad x \quad | \, (\lambda x.t) \, | \, (t \;\; t)$$

Example

$x \; y$	$(x \; y)$	*"x applied to y"*
$\lambda x.x$	$(\lambda x.x)$	*"lambda x to x"*
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	*"lambda x y to x"*
$\lambda x.x \; x$		
$(\lambda x.x) \; x$		

λ-Calculus

λ-Terms

$$t ::= \quad x \quad | \, (\lambda x.t) \, | \, (t \;\; t)$$

Example

$x \; y$	$(x \; y)$	*"x applied to y"*
$\lambda x.x$	$(\lambda x.x)$	*"lambda x to x"*
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	*"lambda x y to x"*
$\lambda x.x \; x$	$(\lambda x.(x \; x))$	
$(\lambda x.x) \; x$		

λ-Calculus

λ-Terms

$$t ::= \quad x \quad | \, (\lambda x.t) \, | \, (t \;\; t)$$

Example

$x \; y$	$(x \; y)$	*"x applied to y"*
$\lambda x.x$	$(\lambda x.x)$	*"lambda x to x"*
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	*"lambda x y to x"*
$\lambda x.x \; x$	$(\lambda x.(x \; x))$	*"lambda x to (x applied to x)"*
$(\lambda x.x) \; x$		

λ-Calculus

λ-Terms

$$t ::= \quad x \quad | \, (\lambda x.t) \, | \, (t \ t)$$

Example

$x \ y$	$(x \ y)$	*"x applied to y"*
$\lambda x.x$	$(\lambda x.x)$	*"lambda x to x"*
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	*"lambda x y to x"*
$\lambda x.x \ x$	$(\lambda x.(x \ x))$	*"lambda x to (x applied to x)"*
$(\lambda x.x) \ x$	$((\lambda x.x) \ x)$	

λ-Calculus

λ-Terms

$$t ::= \quad x \quad | \, (\lambda x.t) \, | \, (t \ t)$$

Example

$x \ y$	$(x \ y)$	*"x applied to y"*
$\lambda x.x$	$(\lambda x.x)$	*"lambda x to x"*
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	*"lambda x y to x"*
$\lambda x.x \ x$	$(\lambda x.(x \ x))$	*"lambda x to (x applied to x)"*
$(\lambda x.x) \ x$	$((\lambda x.x) \ x)$	*"(lambda x to x) applied to x"*

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.

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

λ-Calculus (cont'd)

β-Reduction

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

$$\overbrace{s \to_\beta t}^{(\beta\text{-})\text{step}}$$

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

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

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.

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

Example

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

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

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

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

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
  - The λ Tree Tool `LTT`

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

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

Booleans and Conditionals

OCaml

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

λ-Calculus

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

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

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

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

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

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

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

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

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

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

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

$$s^0\ t \stackrel{\text{def}}{=} t \qquad\qquad s^{n+1}\ t \stackrel{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

```
0
1
n
( + )
( * )
( ** )
```

Natural Numbers

Definition

$$s^0\ t \stackrel{\text{def}}{=} t \qquad\qquad s^{n+1}\ t \stackrel{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \stackrel{\text{def}}{=} \lambda fx.x$
`1`	
`n`	
`( + )`	
`( * )`	
`( ** )`	

Natural Numbers

Definition

$$s^0\ t \stackrel{\text{def}}{=} t \qquad\qquad s^{n+1}\ t \stackrel{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \stackrel{\text{def}}{=} \lambda fx.x$
`1`	$\overline{1} \stackrel{\text{def}}{=} \lambda fx.f\ x$
`n`	
`( + )`	
`( * )`	
`( ** )`	

Natural Numbers

Definition

$$s^0\ t \stackrel{\text{def}}{=} t \qquad\qquad s^{n+1}\ t \stackrel{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \stackrel{\text{def}}{=} \lambda fx.x$
`1`	$\overline{1} \stackrel{\text{def}}{=} \lambda fx.f\ x$
`n`	$\overline{\mathrm{n}} \stackrel{\text{def}}{=} \lambda fx.f^n\ x$
`( + )`	
`( * )`	
`( ** )`	

Natural Numbers

Definition

$$s^0\ t \stackrel{\text{def}}{=} t \qquad\qquad s^{n+1}\ t \stackrel{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \stackrel{\text{def}}{=} \lambda fx.x$
`1`	$\overline{1} \stackrel{\text{def}}{=} \lambda fx.f\ x$
`n`	$\overline{\text{n}} \stackrel{\text{def}}{=} \lambda fx.f^n\ x$
`( + )`	$\text{add} \stackrel{\text{def}}{=} \lambda mnfx.m\ f\ (n\ f\ x)$
`( * )`	
`( ** )`	

Natural Numbers

Definition

$$s^0\ t \stackrel{\text{def}}{=} t \qquad\qquad s^{n+1}\ t \stackrel{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \stackrel{\text{def}}{=} \lambda fx.x$
`1`	$\overline{1} \stackrel{\text{def}}{=} \lambda fx.f\ x$
`n`	$\overline{\text{n}} \stackrel{\text{def}}{=} \lambda fx.f^n\ x$
`( + )`	$\text{add} \stackrel{\text{def}}{=} \lambda mnfx.m\ f\ (n\ f\ x)$
`( * )`	$\text{mul} \stackrel{\text{def}}{=} \lambda mnf.m\ (n\ f)$
`( ** )`	

Natural Numbers

Definition

$$s^0\ t \stackrel{\text{def}}{=} t \qquad\qquad s^{n+1}\ t \stackrel{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \stackrel{\text{def}}{=} \lambda fx.x$
`1`	$\overline{1} \stackrel{\text{def}}{=} \lambda fx.f\ x$
`n`	$\overline{n} \stackrel{\text{def}}{=} \lambda fx.f^n\ x$
`( + )`	$\text{add} \stackrel{\text{def}}{=} \lambda mnfx.m\ f\ (n\ f\ x)$
`( * )`	$\text{mul} \stackrel{\text{def}}{=} \lambda mnf.m\ (n\ f)$
`( ** )`	$\text{exp} \stackrel{\text{def}}{=} \lambda mn.n\ m$

Natural Numbers

Definition

$$s^0\ t \stackrel{\text{def}}{=} t \qquad s^{n+1}\ t \stackrel{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \stackrel{\text{def}}{=} \lambda fx.x$
`1`	$\overline{1} \stackrel{\text{def}}{=} \lambda fx.f\ x$
`n`	$\overline{n} \stackrel{\text{def}}{=} \lambda fx.f^n\ x$
`( + )`	$\text{add} \stackrel{\text{def}}{=} \lambda mnfx.m\ f\ (n\ f\ x)$
`( * )`	$\text{mul} \stackrel{\text{def}}{=} \lambda mnf.m\ (n\ f)$
`( ** )`	$\text{exp} \stackrel{\text{def}}{=} \lambda mn.n\ m$

Example

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

Natural Numbers

Definition

$$s^0\ t \stackrel{\text{def}}{=} t \qquad s^{n+1}\ t \stackrel{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \stackrel{\text{def}}{=} \lambda fx.x$
`1`	$\overline{1} \stackrel{\text{def}}{=} \lambda fx.f\ x$
`n`	$\overline{n} \stackrel{\text{def}}{=} \lambda fx.f^n\ x$
`( + )`	$\text{add} \stackrel{\text{def}}{=} \lambda mnfx.m\ f\ (n\ f\ x)$
`( * )`	$\text{mul} \stackrel{\text{def}}{=} \lambda mnf.m\ (n\ f)$
`( ** )`	$\text{exp} \stackrel{\text{def}}{=} \lambda mn.n\ m$

Example

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

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)
fst
snd
```

$\text{pair} \stackrel{\text{def}}{=} \lambda xyf.f\ x\ y$

$\text{fst} \stackrel{\text{def}}{=} \lambda p.p\ \text{true}$

Pairs

OCaml vs. λ-Calculus

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

Pairs

OCaml vs. λ-Calculus

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

Example

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

Pairs

OCaml vs. λ-Calculus

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

Example

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

Lists

OCaml vs. λ-Calculus

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

Lists

OCaml vs. λ-Calculus

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

Lists

OCaml vs. λ-Calculus

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

Lists

OCaml vs. λ-Calculus

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

```
fun x -> x = []
```

Lists

OCaml vs. λ-Calculus

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

```
fun x -> x = []
```

Lists

OCaml vs. λ-Calculus

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

```
fun x -> x = []
```

Lists

OCaml vs. λ-Calculus

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

Lists

OCaml vs. λ-Calculus

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

Example

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

Lists

OCaml vs. λ-Calculus

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

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

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

Recursion

OCaml

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

λ-Calculus

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

Recursion

OCaml

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

λ-Calculus

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

Recursion

OCaml

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

λ-Calculus

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

Recursion

OCaml

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

λ-Calculus

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

Definition (Y-combinator)

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

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

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

Recursion

OCaml

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

λ-Calculus

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

Definition (Y-combinator)

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

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

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

Overview

- Week 6 - λ Calculus, Evaluation Strategies
  - Summary of Week 5
  - λ-Calculus - Data Types
  - Evaluation Strategies
  - The λ Tree Tool `LTT`

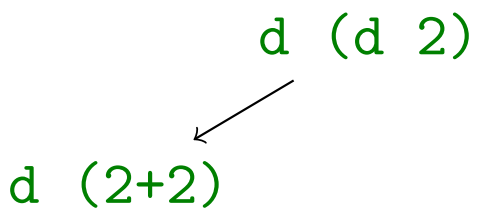
Example

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

```
d (d 2)
```

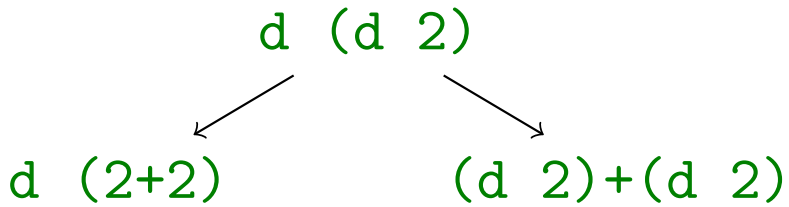
Example

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



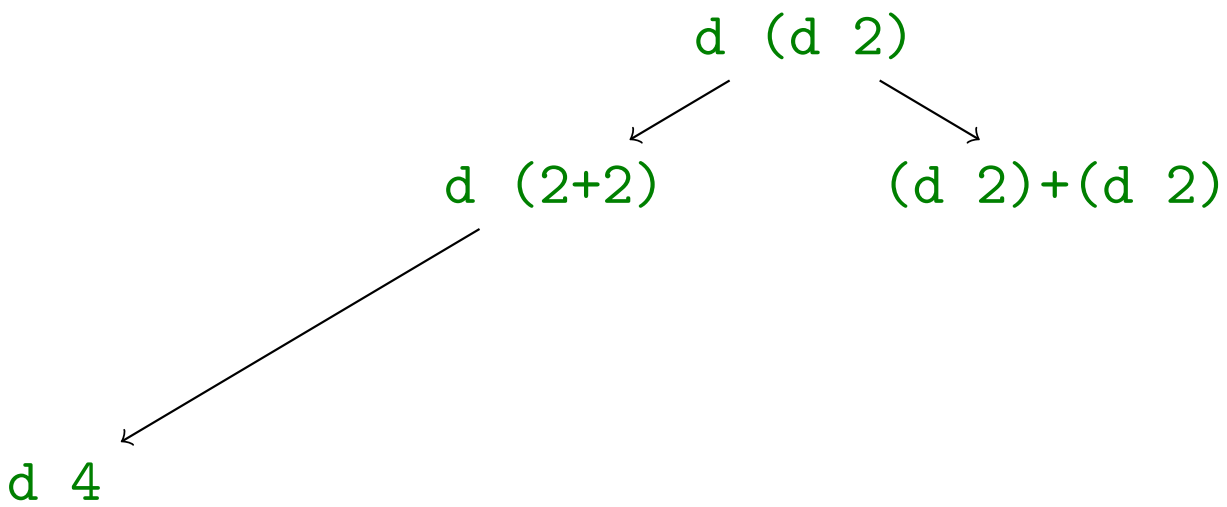
Example

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



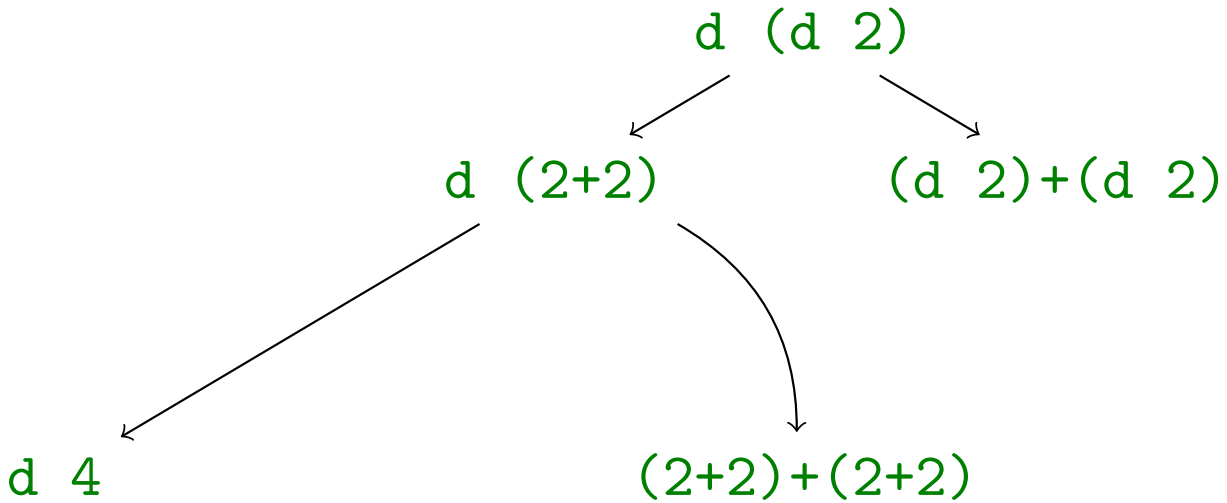
Example

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



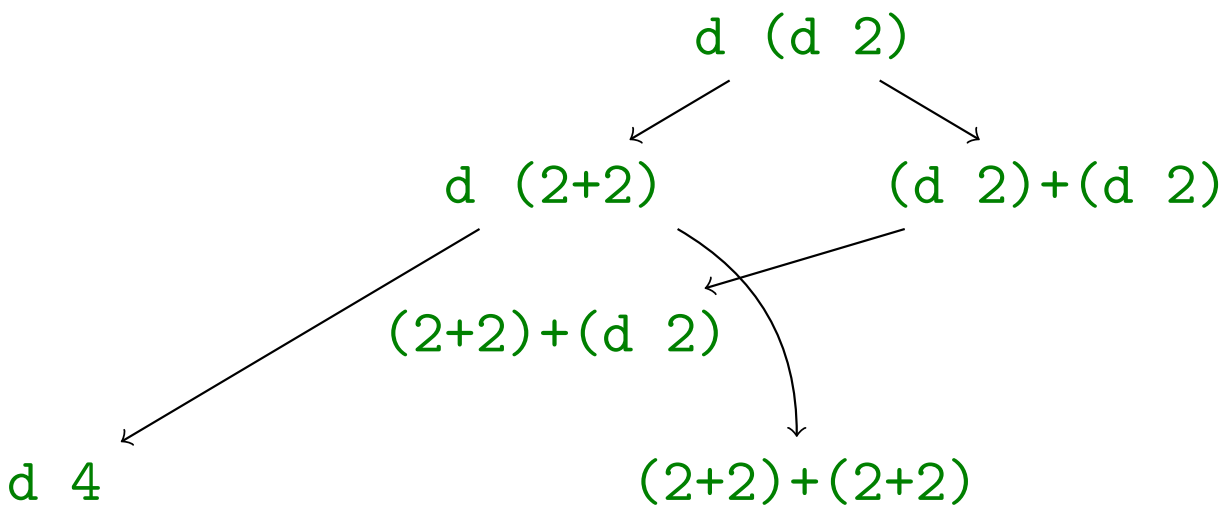
Example

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



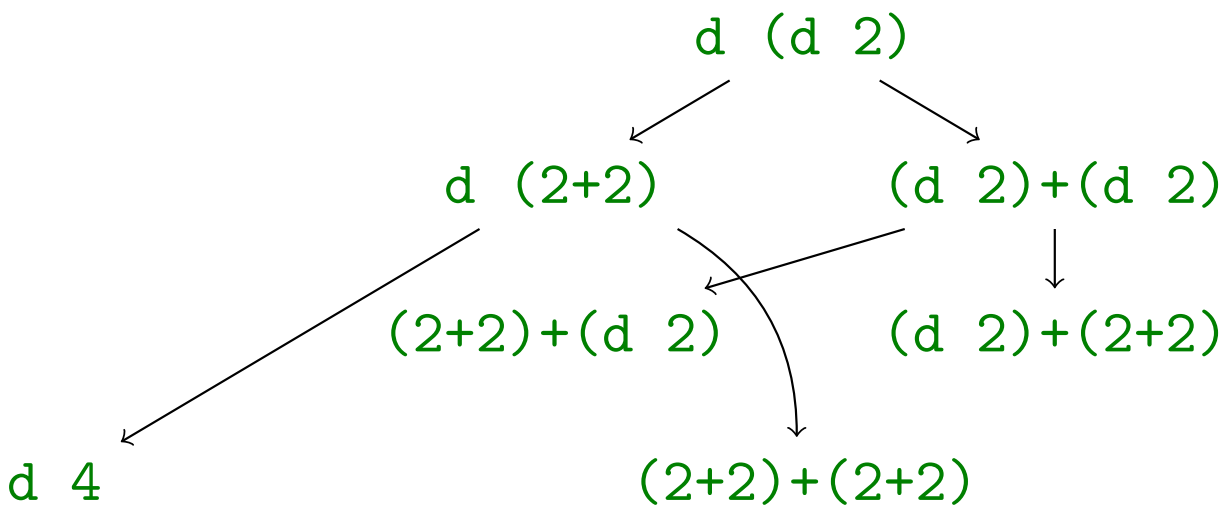
Example

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



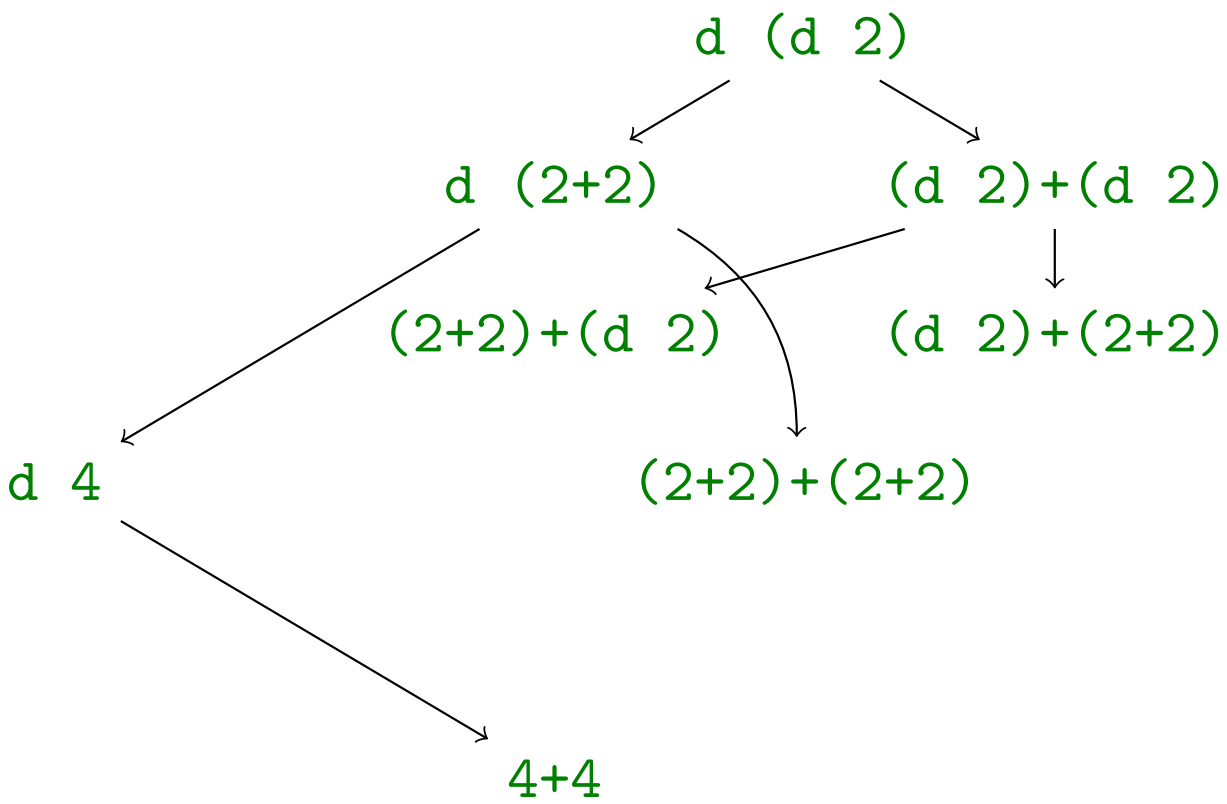
Example

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



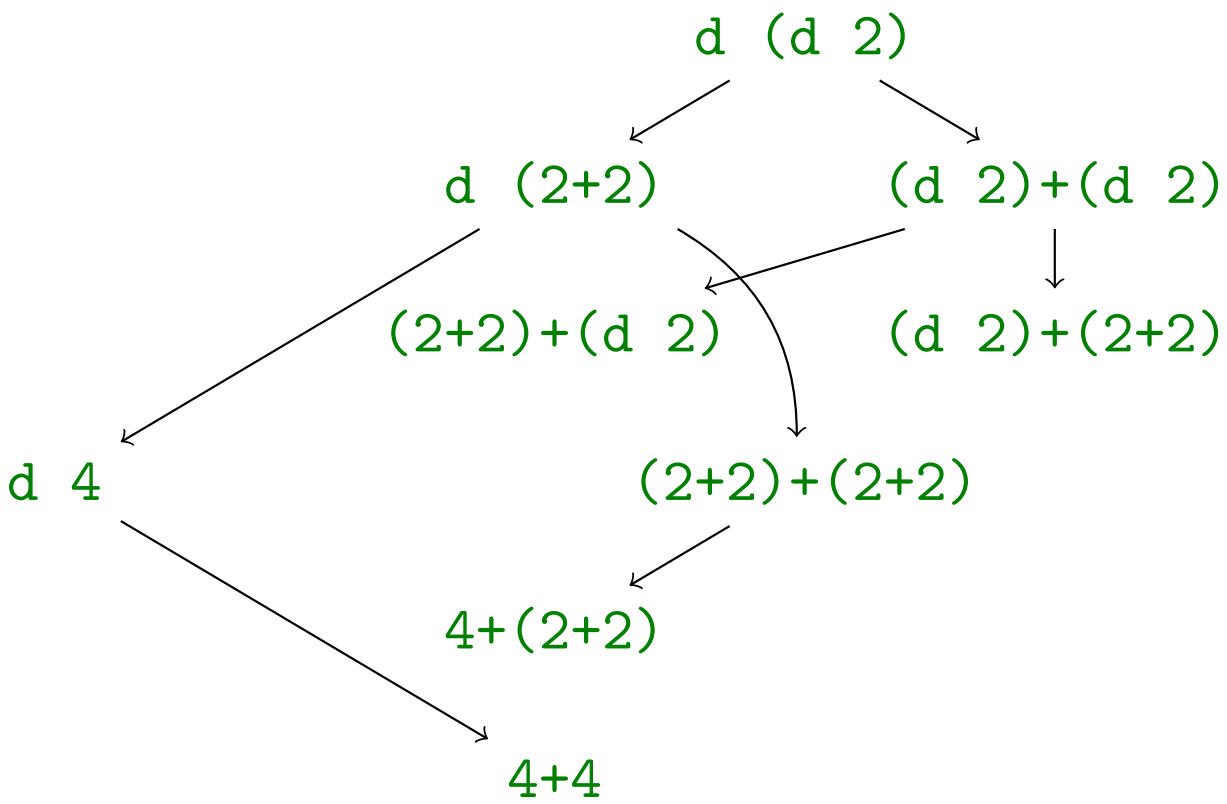
Example

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



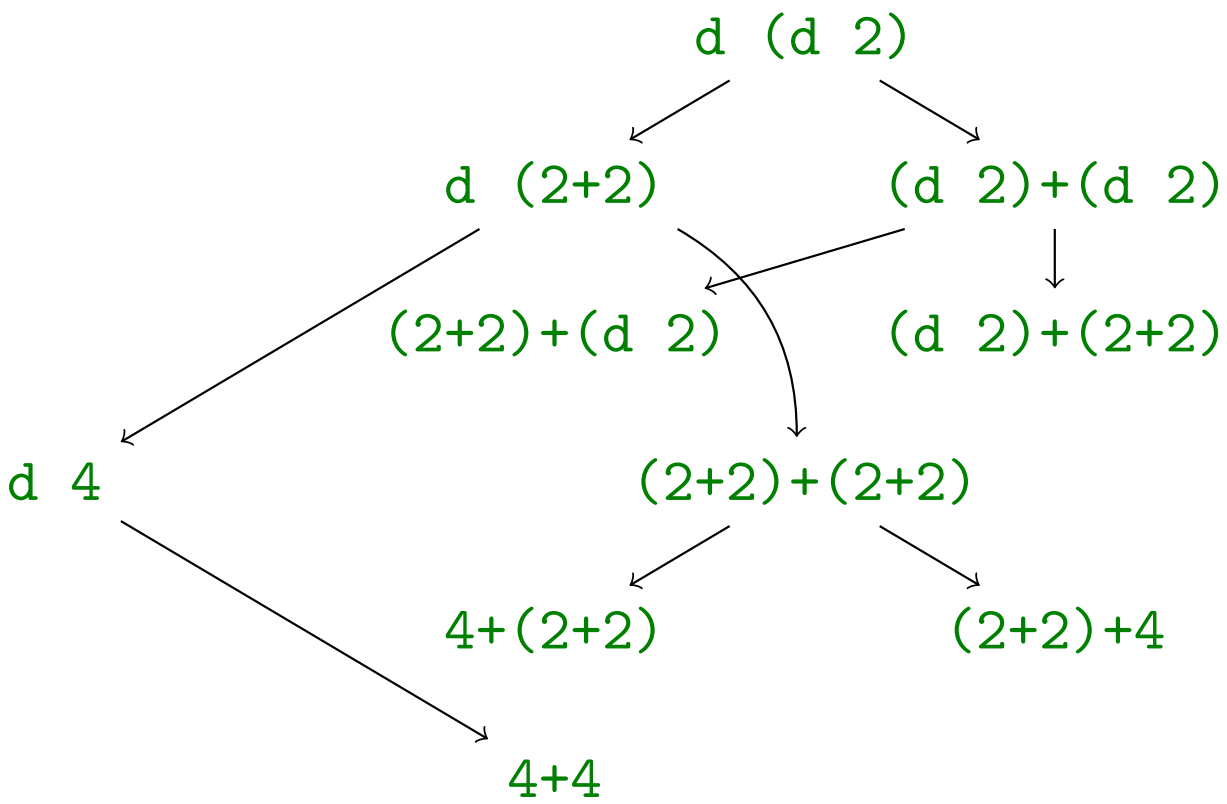
Example

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



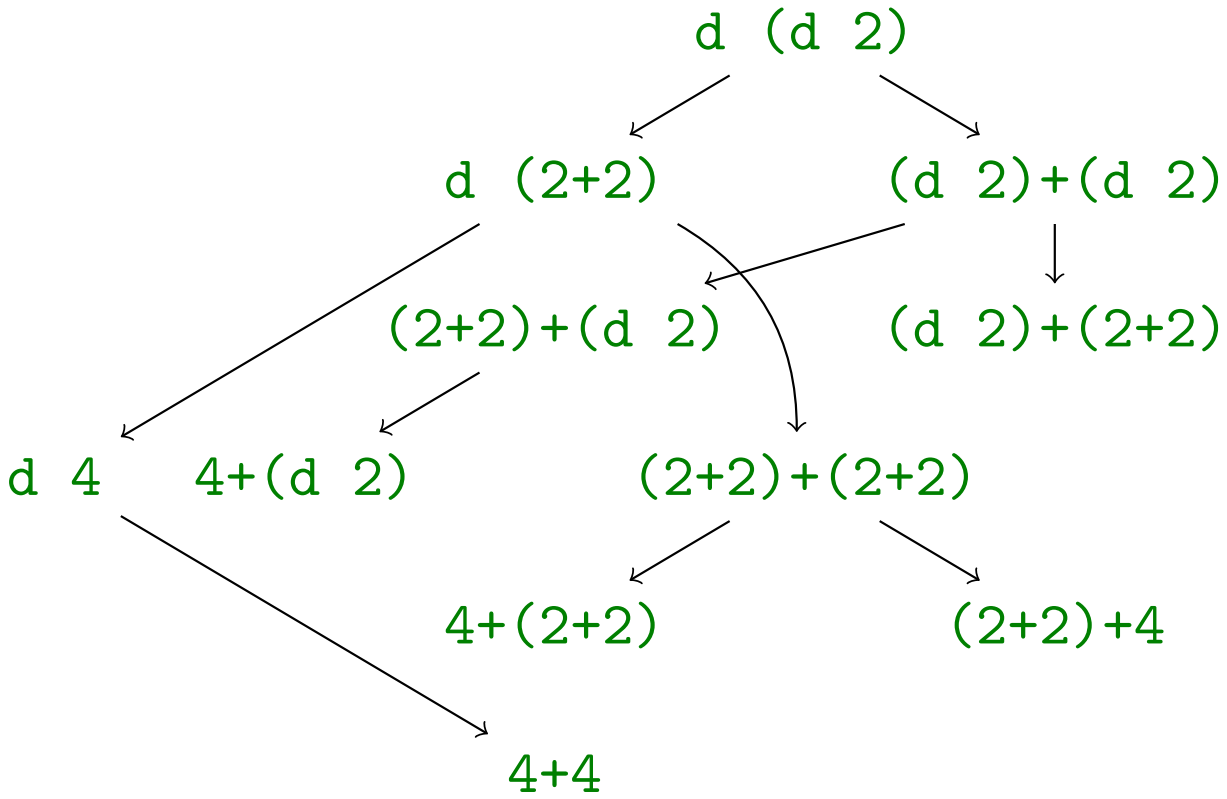
Example

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



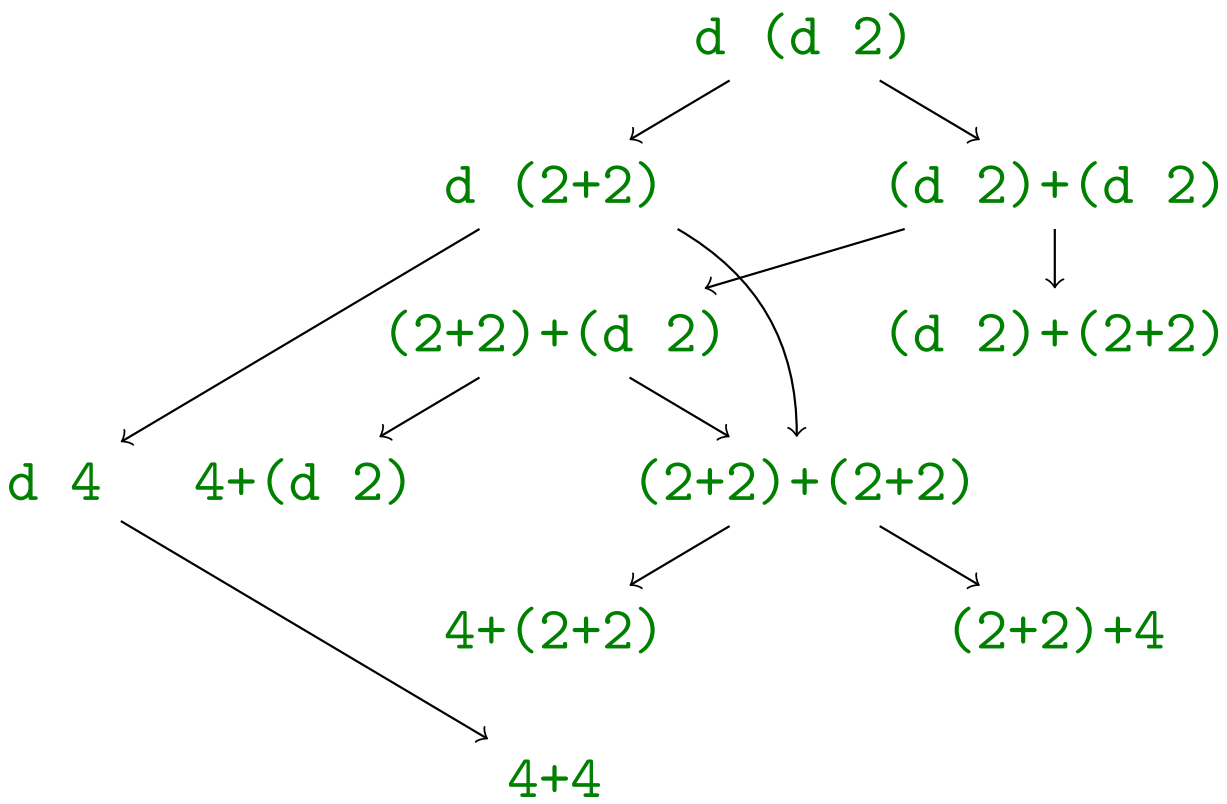
Example

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



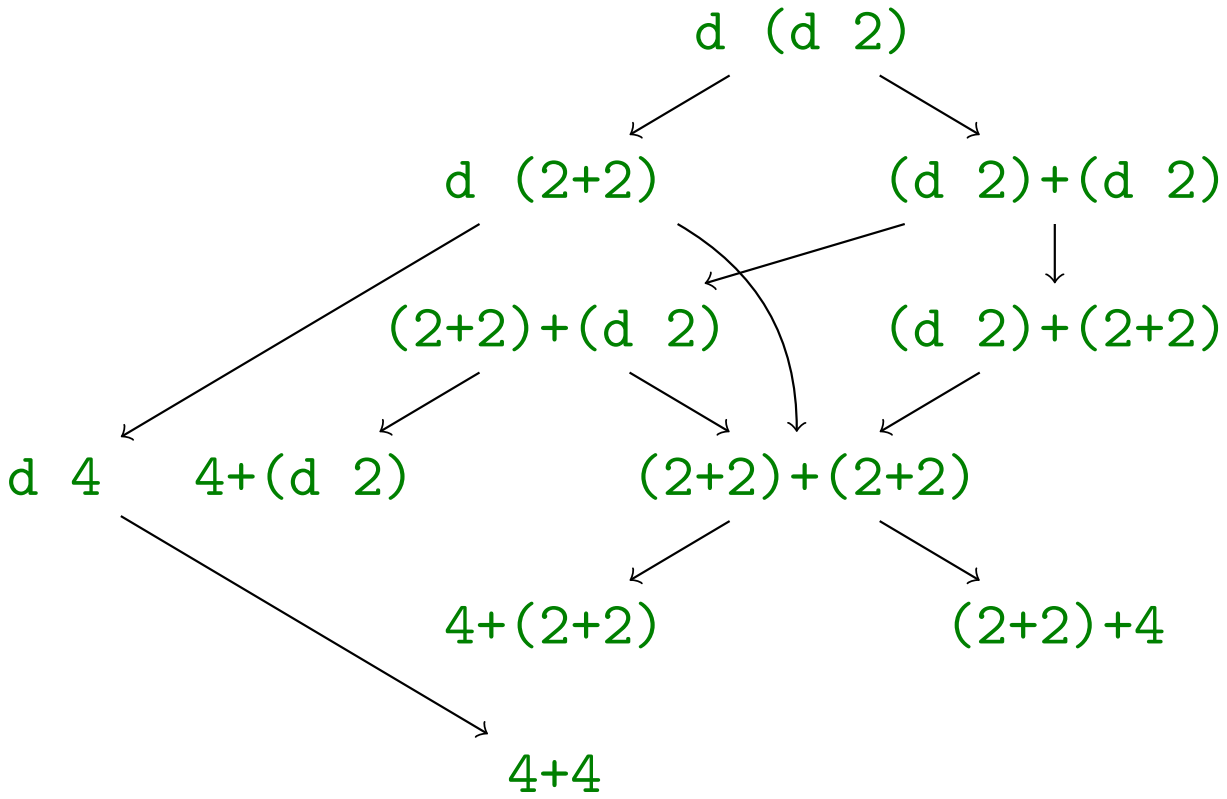
Example

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



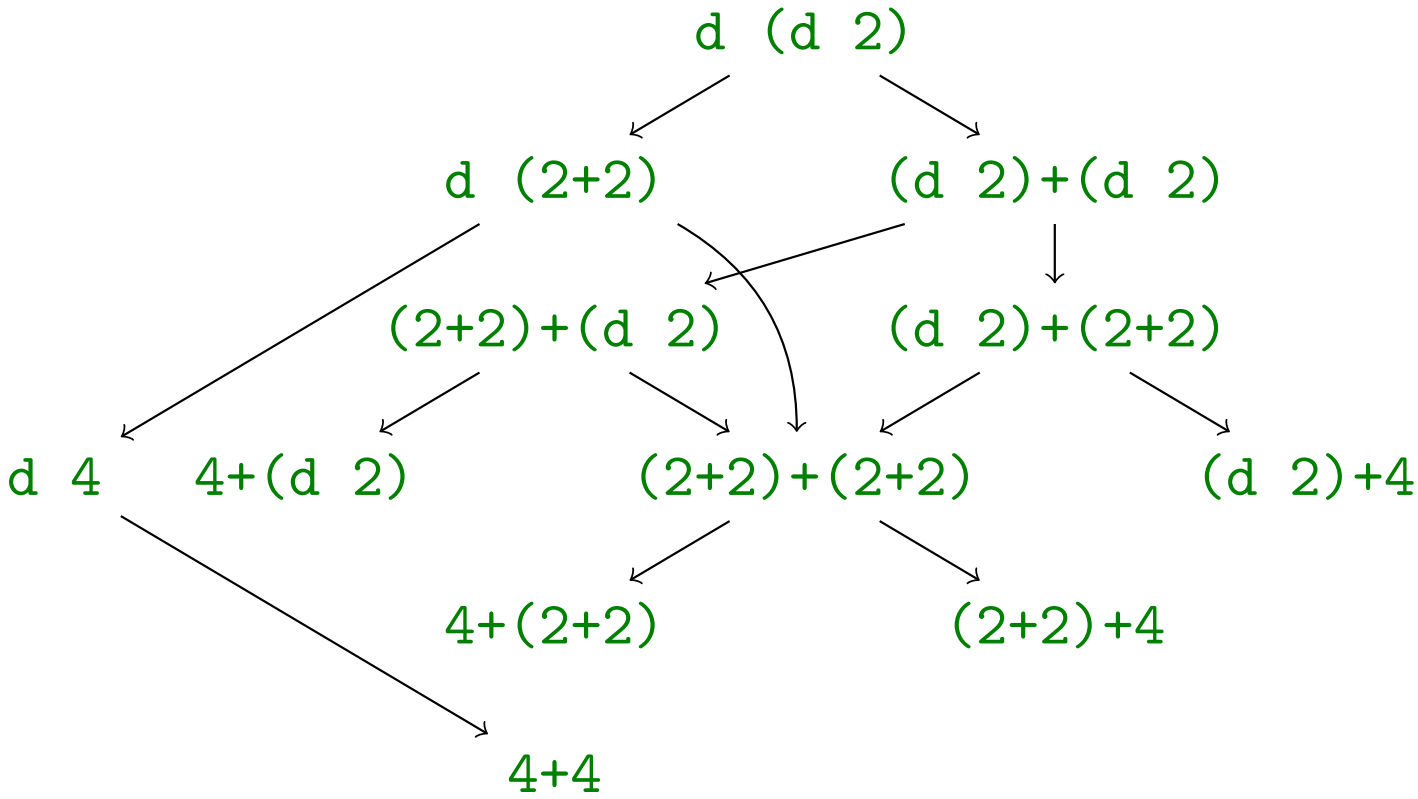
Example

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



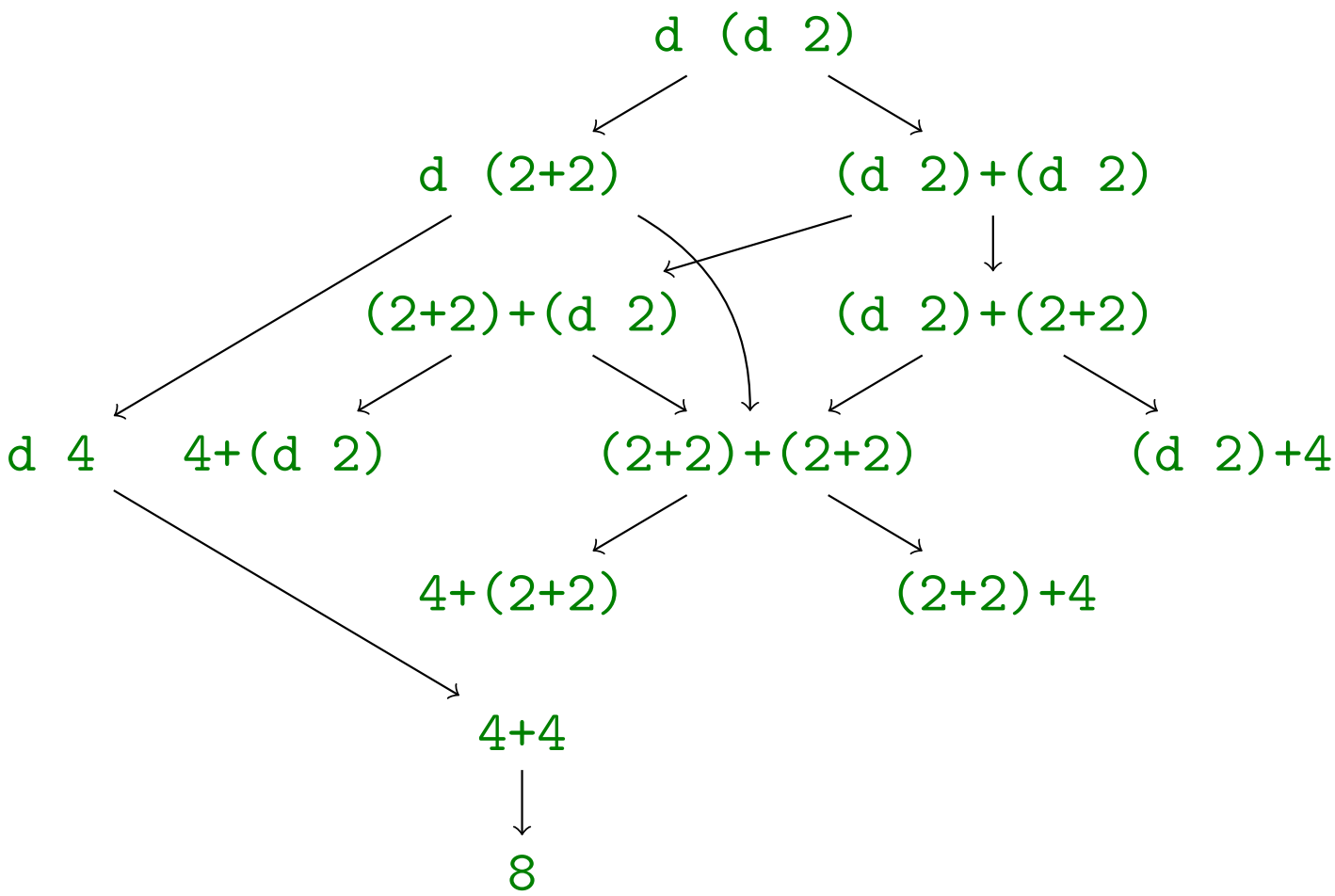
Example

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



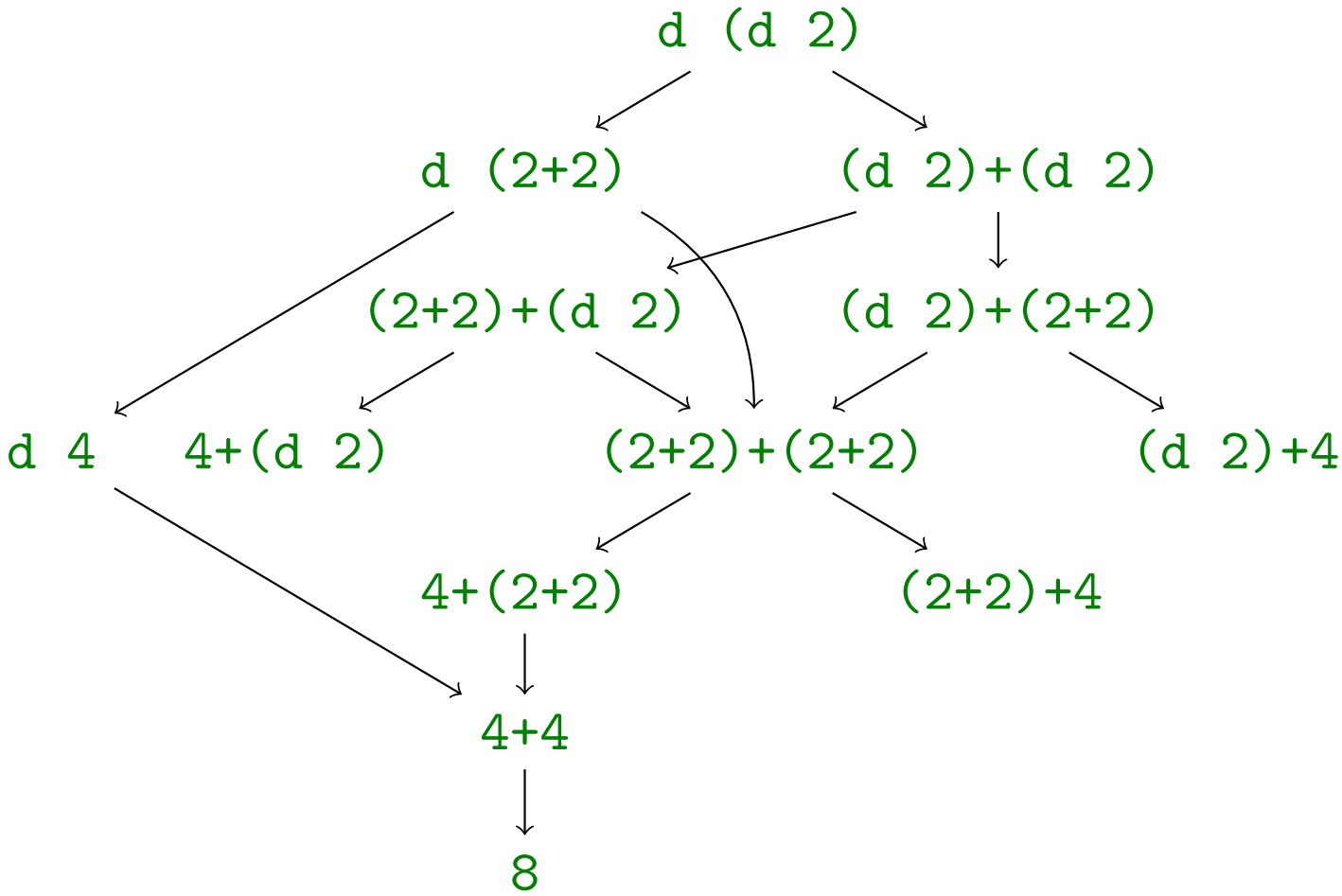
Example

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



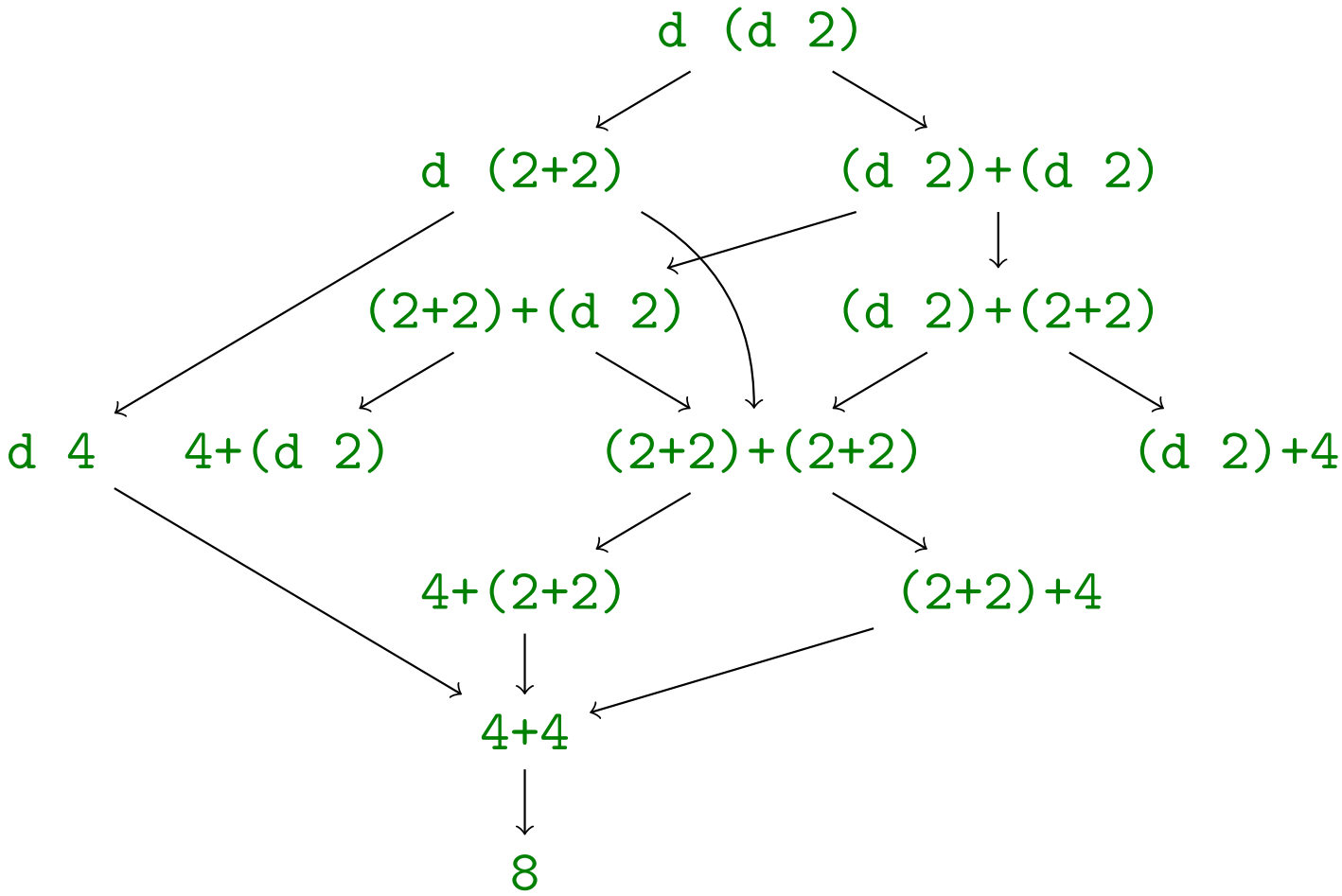
Example

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



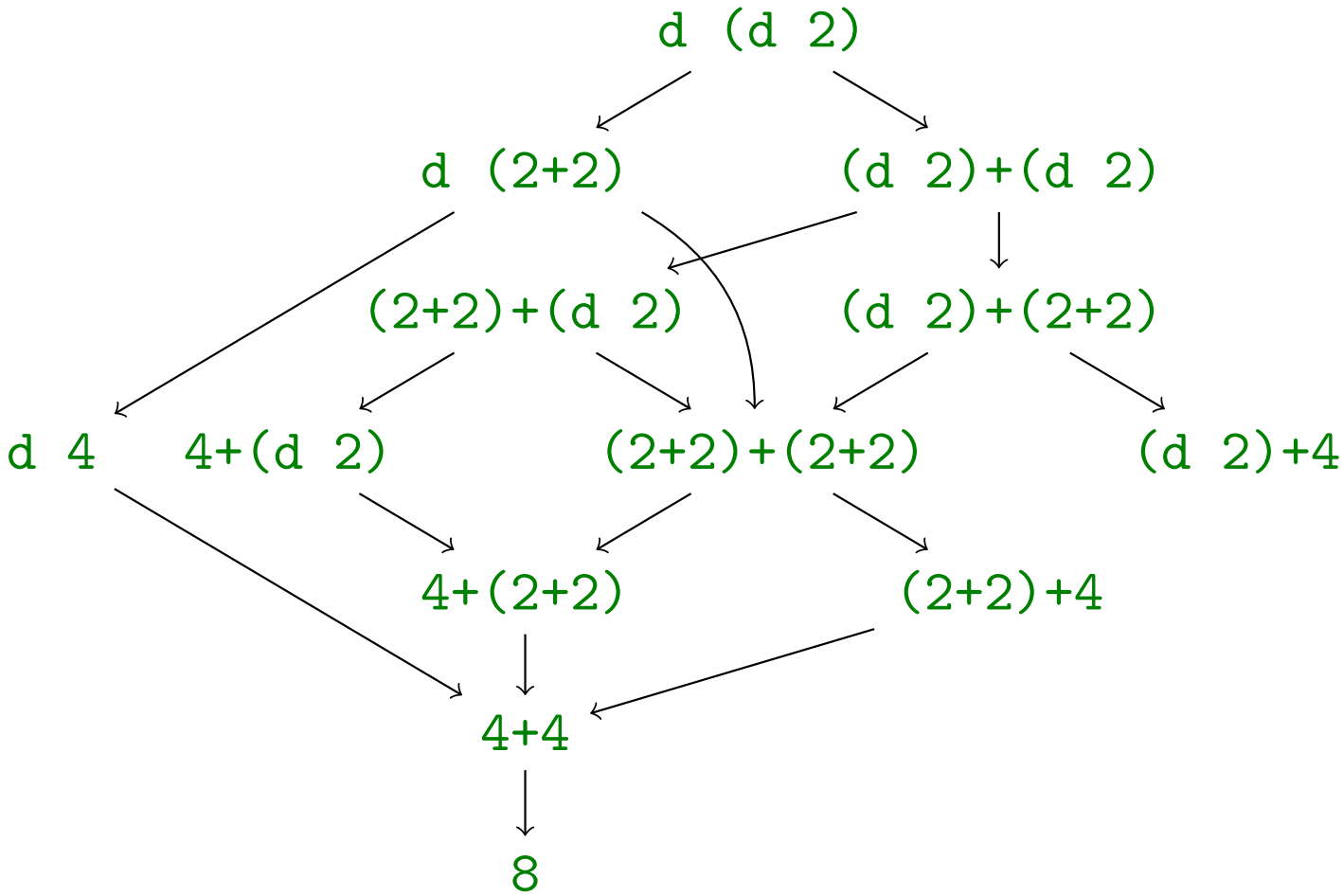
Example

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



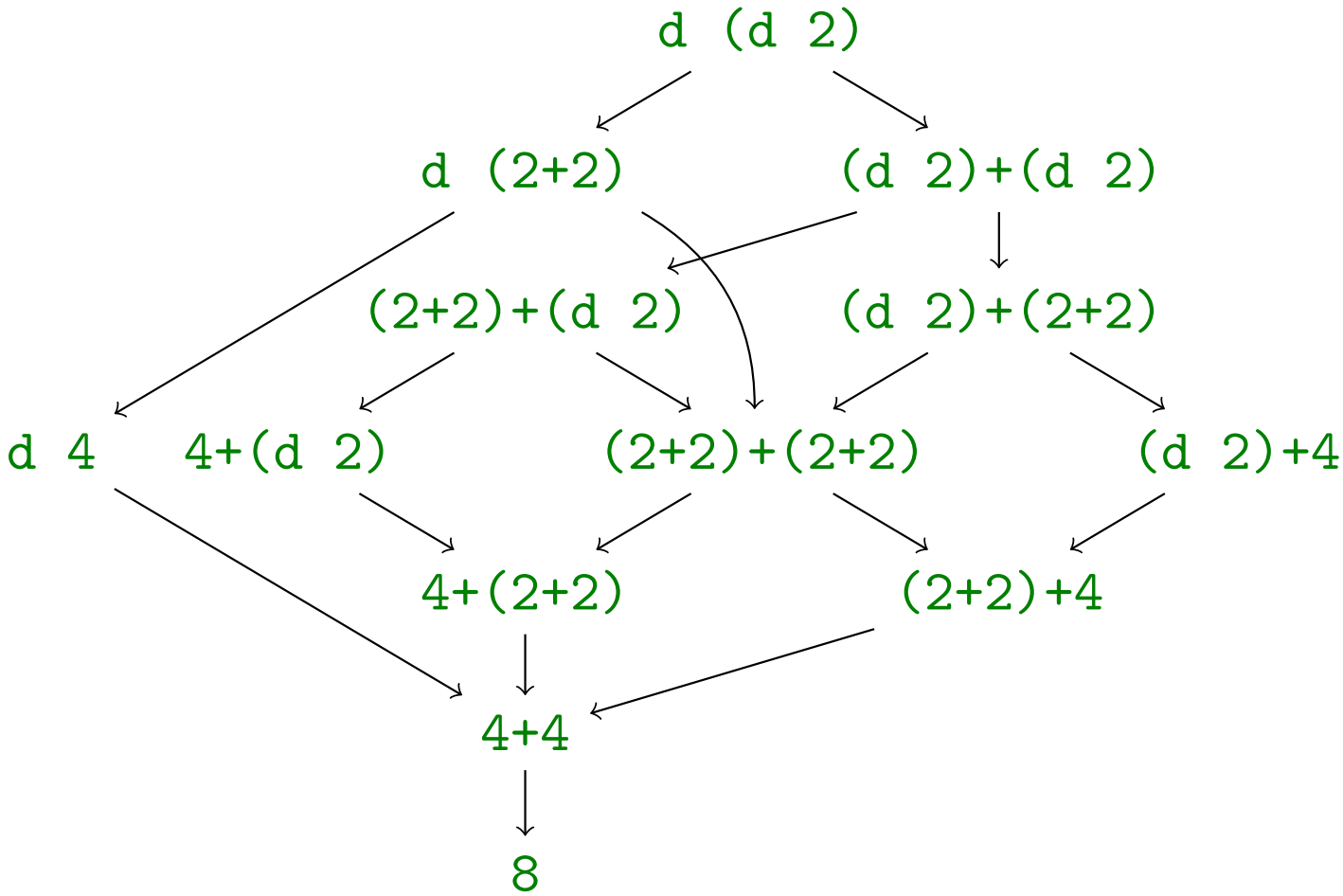
Example

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



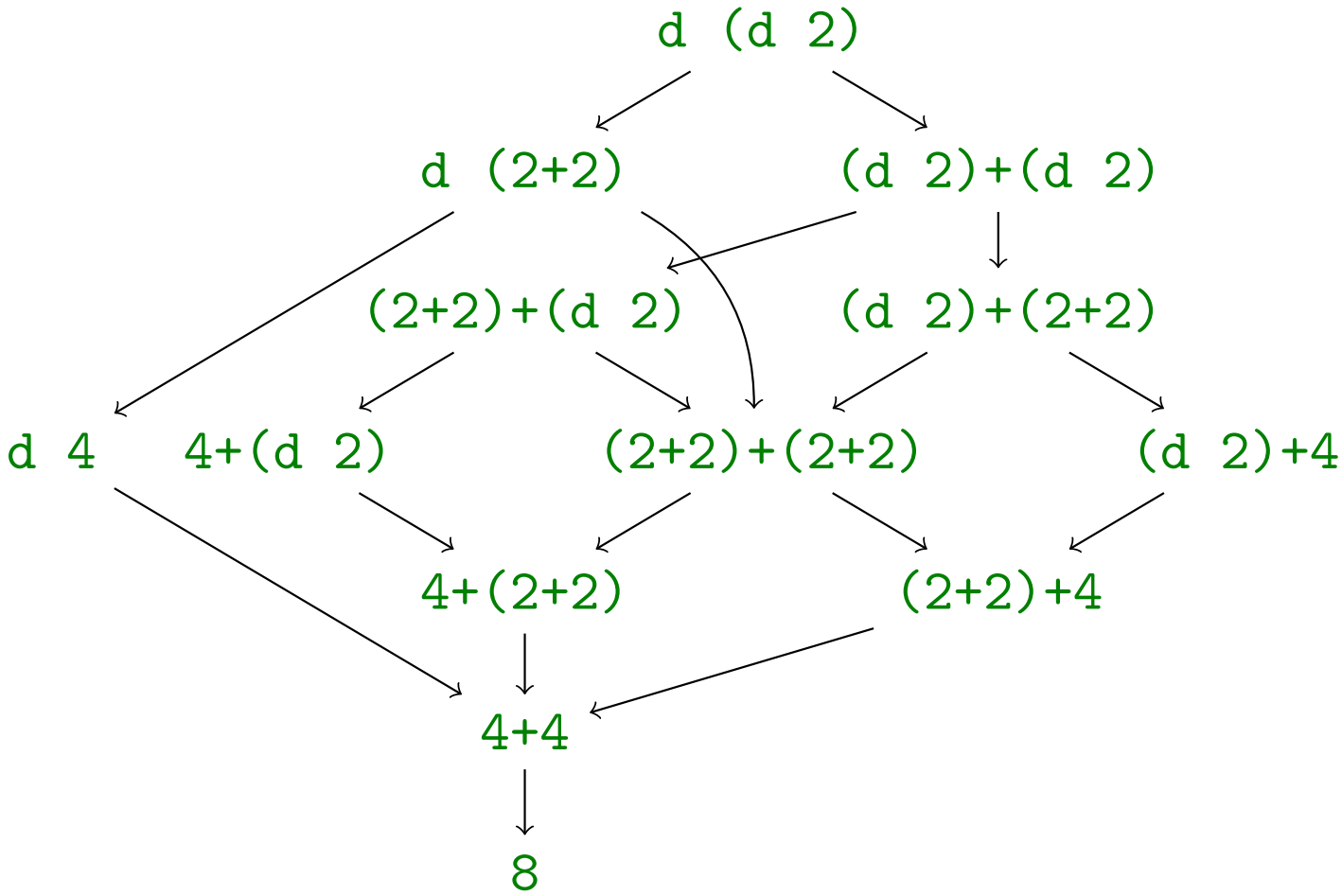
Example

- 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:

```
d (d 2) → d (2+2)
        → d 4
        → 4 + 4
        → 8
```

- call-by-name:

```
d (d 2) → (d 2)+(d 2)
        → (2+2)+(d 2)
        → 4+(d 2)
        → 4+(2+2)
        → 4+4
        → 8
```

(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.,

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

(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.,

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

Example

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

(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.,

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

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.,

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

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.,

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

(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.,

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

Example

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

(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.,

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

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.,

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

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

Overview

- Week 6 - λ Calculus, Evaluation Strategies
  - Summary of Week 5
  - λ-Calculus - Data Types
  - Evaluation Strategies
  - The λ Tree Tool LTT

λ Tree Tool

developed by Stefan Widerin in bachelor project

λ Tree Tool

developed by Stefan Widerin in bachelor project

λ-Terms

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

λ Tree Tool

developed by Stefan Widerin in bachelor project

λ-Terms

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

Conventions

- nested abstractions use spaces to separate variable names, e.g.,

$\lambda xy.x$	`\x y.x`
$\lambda x_1.y$	`\x1.y`

Result

Animator

- animation of β-steps
- innermost/outermost
- leftmost/rightmost

Result

Animator

- animation of β-steps
- innermost/outermost
- leftmost/rightmost

Output

- reduction sequence to NF
- innermost/outermost
- leftmost/rightmost