CENG 3549 – Functional Programming

Evaluation Strategies & Abstract Data Types & Sets as Binary Search Trees

Burak Ekici

December 1, 2022

Outline

Recall λ-Terms

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

Examples

conventions	verbose	in words
$x\ y$	$(x\ y)$	“x applied to y”
$\lambda x.\, x$	$(\lambda x.\, x)$	“lambda x to x” (identity function)
$\lambda xy.\, x$	$(\lambda x.\, (\lambda y.\, x))$	“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”

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \rightarrow_\beta t$
- if there is redex $(\lambda x.\, u)\ v$ in s such that
- replacing $(\lambda x.\, u)\ v$ in s by contractum $u[x := v]$ results in t

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \rightarrow_\beta t$
- if there is redex $(\lambda x.\, u)\ v$ in s such that
- replacing $(\lambda x.\, u)\ v$ in s by contractum $u[x := v]$ results in t

Example

- recall church numerals $\mathrm{n} = \lambda f x.\, f^n\ x$

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \rightarrow_\beta t$
- if there is redex $(\lambda x.\, u)\ v$ in s such that
- replacing $(\lambda x.\, u)\ v$ in s by contractum $u[x := v]$ results in t

Example

- recall church numerals $\mathrm{n} = \lambda fx.\, f^n\ x$
- with exponentiation $\exp = \lambda mn.\, n\ m$

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \rightarrow_\beta t$
- if there is redex $(\lambda x.\, u)\ v$ in s such that
- replacing $(\lambda x.\, u)\ v$ in s by contractum $u[x := v]$ results in t

Example

- recall church numerals $\mathrm{n} = \lambda f x.\, f^n\ x$
- with exponentiation $\mathrm{exp} = \lambda mn.\, n\ m$
- compute "one to the power of one"

  $\mathrm{exp}\ \overline{1}\ \overline{1}$

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \to_\beta t$
- if there is redex $(\lambda x.\, u)\ v$ in s such that
- replacing $(\lambda x.\, u)\ v$ in s by contractum $u[x := v]$ results in t

Example

- recall church numerals $\mathsf{n} = \lambda f x.\, f^n\ x$
- with exponentiation $\mathsf{exp} = \lambda mn.\, n\ m$
- compute "one to the power of one"

$$\mathsf{exp}\ \overline{1}\ \overline{1} = \underline{(\lambda mn.\, n\ m)\ \overline{1}}\ \overline{1}$$

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \rightarrow_\beta t$
- if there is redex $(\lambda x.\, u)\ v$ in s such that
- replacing $(\lambda x.\, u)\ v$ in s by contractum $u[x := v]$ results in t

Example

- recall church numerals $\mathrm{n} = \lambda fx.\, f^n\ x$
- with exponentiation $\mathrm{exp} = \lambda mn.\, n\ m$
- compute “one to the power of one”

$$
\begin{aligned}
\mathrm{exp}\ \overline{1}\ \overline{1} &= \underline{(\lambda mn.\, n\ m)\ \overline{1}}\ \overline{1} \\
&\rightarrow_\beta \underline{(\lambda n.\, n\ \overline{1})\ \overline{1}}
\end{aligned}
$$

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \rightarrow_\beta t$
- if there is redex $(\lambda x.\, u)\; v$ in s such that
- replacing $(\lambda x.\, u)\; v$ in s by contractum $u[x := v]$ results in t

Example

- recall church numerals $\mathrm{n} = \lambda f x.\, f^n\; x$
- with exponentiation $\exp = \lambda m n.\, n\; m$
- compute "one to the power of one"

$$\begin{aligned}\exp\; \overline{1}\; \overline{1} &= \underline{(\lambda m n.\, n\; m)\; \overline{1}}\; \overline{1} \\ &\rightarrow_\beta \underline{(\lambda n.\, n\; \overline{1})\; \overline{1}} \\ &\rightarrow_\beta \overline{1}\; \overline{1}\end{aligned}$$

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \rightarrow_\beta t$
- if there is redex $(\lambda x.\, u)\ v$ in s such that
- replacing $(\lambda x.\, u)\ v$ in s by contractum $u[x := v]$ results in t

Example

- recall church numerals $\mathsf{n} = \lambda fx.\, f^n\ x$
- with exponentiation $\mathsf{exp} = \lambda mn.\, n\ m$
- compute “one to the power of one”

$$\begin{aligned}
\mathsf{exp}\ \overline{1}\ \overline{1} &= \underline{(\lambda mn.\, n\ m)\ \overline{1}}\ \overline{1} \\
&\rightarrow_\beta \underline{(\lambda n.\, n\ \overline{1})\ \overline{1}} \\
&\rightarrow_\beta \overline{1}\ \overline{1} = \underline{(\lambda fx.\, f\ x)\ (\lambda fx.\, f\ x)}
\end{aligned}$$

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \to_\beta t$
- if there is redex $(\lambda x.\, u)\ v$ in s such that
- replacing $(\lambda x.\, u)\ v$ in s by contractum $u[x := v]$ results in t

Example

- recall church numerals $\mathrm{n} = \lambda fx.\, f^n\ x$
- with exponentiation $\mathrm{exp} = \lambda mn.\, n\ m$
- compute "one to the power of one"

$$\begin{aligned}
\mathrm{exp}\ \overline{1}\ \overline{1} &= \underline{(\lambda mn.\, n\ m)\ \overline{1}}\ \overline{1} \\
&\to_\beta \underline{(\lambda n.\, n\ \overline{1})\ \overline{1}} \\
&\to_\beta \overline{1}\ \overline{1} = \underline{(\lambda fx.\, f\ x)\ (\lambda fx.\, f\ x)} \\
&\to_\beta \lambda x.\, \underline{(\lambda fx.\, f\ x)\ x}
\end{aligned}$$

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \rightarrow_\beta t$
- if there is redex $(\lambda x.\, u)\ v$ in s such that
- replacing $(\lambda x.\, u)\ v$ in s by contractum $u[x := v]$ results in t

Example

- recall church numerals $\mathrm{n} = \lambda fx.\, f^n\ x$
- with exponentiation $\mathrm{exp} = \lambda mn.\, n\ m$
- compute “one to the power of one”

$$\begin{aligned}
\mathrm{exp}\ \overline{1}\ \overline{1} &= \underline{(\lambda mn.\, n\ m)\ \overline{1}}\ \overline{1} \\
&\rightarrow_\beta \underline{(\lambda n.\, n\ \overline{1})\ \overline{1}} \\
&\rightarrow_\beta \overline{1}\ \overline{1} = \underline{(\lambda fx.\, f\ x)\ (\lambda fx.\, f\ x)} \\
&\rightarrow_\beta \lambda x.\, \underline{(\lambda fx.\, f\ x)\ x} \\
&\rightarrow_\beta \lambda x.\, \lambda x.\, x\ x
\end{aligned}$$

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \to_\beta t$
- if there is redex $(\lambda x.\, u)\ v$ in s such that
- replacing $(\lambda x.\, u)\ v$ in s by contractum $u[x := v]$ results in t

Example

- recall church numerals $\overline{n} = \lambda fx.\, f^n\ x$
- with exponentiation $\mathsf{exp} = \lambda mn.\, n\ m$
- compute "one to the power of one"

$$
\begin{aligned}
\mathsf{exp}\ \overline{1}\ \overline{1} &= \underline{(\lambda mn.\, n\ m)\ \overline{1}}\ \overline{1} \\
&\to_\beta \underline{(\lambda n.\, n\ \overline{1})\ \overline{1}} \\
&\to_\beta \overline{1}\ \overline{1} = \underline{(\lambda fx.\, f\ x)\ (\lambda fx.\, f\ x)} \\
&\to_\beta \lambda x.\, \underline{(\lambda fx.\, f\ x)\ x} \\
&\to_\beta \lambda x.\, \lambda x.\, x\ x \quad \text{✘ variable capture!}
\end{aligned}
$$

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \rightarrow_\beta t$
- if there is redex $(\lambda x.\, u)\ v$ in s such that
- replacing $(\lambda x.\, u)\ v$ in s by contractum $u[x := v]$ results in t

Example

- recall church numerals $\overline{n} = \lambda fx.\, f^n\ x$
- with exponentiation $\mathrm{exp} = \lambda mn.\, n\ m$
- compute "one to the power of one"

$$
\begin{aligned}
\mathrm{exp}\ \overline{1}\ \overline{1} &= \underline{(\lambda mn.\, n\ m)\ \overline{1}}\ \overline{1} \\
&\rightarrow_\beta \underline{(\lambda n.\, n\ \overline{1})\ \overline{1}} \\
&\rightarrow_\beta \overline{1}\ \overline{1} = \underline{(\lambda fx.\, f\ x)\ (\lambda fx.\, f\ x)} \\
&\rightarrow_\beta \lambda x.\, \underline{(\lambda fx.\, f\ x)\ x} = \lambda x.\, \underline{(\lambda fy.\, f\ y)\ x}
\end{aligned}
$$

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \rightarrow_\beta t$
- if there is redex $(\lambda x.\, u)\ v$ in s such that
- replacing $(\lambda x.\, u)\ v$ in s by contractum $u[x := v]$ results in t

Example

- recall church numerals $\mathrm{n} = \lambda f x.\, f^n\ x$
- with exponentiation $\exp = \lambda mn.\, n\ m$
- compute "one to the power of one"

$$\begin{aligned}
\exp\ \overline{1}\ \overline{1} &= \underline{(\lambda mn.\, n\ m)\ \overline{1}}\ \overline{1} \\
&\rightarrow_\beta \underline{(\lambda n.\, n\ \overline{1})\ \overline{1}} \\
&\rightarrow_\beta \overline{1}\ \overline{1} = \underline{(\lambda f x.\, f\ x)\ (\lambda f x.\, f\ x)} \\
&\rightarrow_\beta \lambda x.\, \underline{(\lambda f x.\, f\ x)\ x} = \lambda x.\, \underline{(\lambda f y.\, f\ y)\ x} \\
&\rightarrow_\beta
\end{aligned}$$

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \to_\beta t$
- if there is redex $(\lambda x.\, u)\ v$ in s such that
- replacing $(\lambda x.\, u)\ v$ in s by contractum $u[x := v]$ results in t

Example

- recall church numerals $\mathrm{n} = \lambda fx.\, f^n\ x$
- with exponentiation $\exp = \lambda mn.\, n\ m$
- compute "one to the power of one"

$$
\begin{aligned}
\exp\ \overline{1}\ \overline{1} &= \underline{(\lambda mn.\, n\ m)\ \overline{1}}\ \overline{1} \\
&\to_\beta \underline{(\lambda n.\, n\ \overline{1})\ \overline{1}} \\
&\to_\beta \overline{1}\ \overline{1} = \underline{(\lambda fx.\, f\ x)\ (\lambda fx.\, f\ x)} \\
&\to_\beta \lambda x.\, \underline{(\lambda fx.\, f\ x)\ x} = \lambda x.\, \underline{(\lambda fy.\, f\ y)\ x} \\
&\to_\beta \lambda x.\, \lambda y.\, x\ y = \lambda fx.\, f\ x
\end{aligned}
$$

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \to_\beta t$
- if there is redex $(\lambda x.\, u)\ v$ in s such that
- replacing $(\lambda x.\, u)\ v$ in s by contractum $u[x := v]$ results in t

Example

- recall church numerals $\mathrm{n} = \lambda f x.\, f^n\ x$
- with exponentiation $\exp = \lambda m n.\, n\ m$
- compute "one to the power of one"

$$
\begin{aligned}
\exp\ \overline{1}\ \overline{1} &= \underline{(\lambda m n.\, n\ m)\ \overline{1}}\ \overline{1} \\
&\to_\beta \underline{(\lambda n.\, n\ \overline{1})\ \overline{1}} \\
&\to_\beta \overline{1}\ \overline{1} = \underline{(\lambda f x.\, f\ x)\ (\lambda f x.\, f\ x)} \\
&\to_\beta \lambda x.\, \underline{(\lambda f x.\, f\ x)\ x} = \lambda x.\, \underline{(\lambda f y.\, f\ y)\ x} \\
&\to_\beta \lambda x.\, \lambda y.\, x\ y = \lambda f x.\, f\ x = \overline{1}
\end{aligned}
$$

Evaluation Order

- consider `d x = x + x`
- term `d (d 2)` may be evaluated as follows

```
d (d 2)
```

Evaluation Order

- consider `d x = x + x`
- term `d (d 2)` may be evaluated as follows

$$\texttt{d (d 2)} \rightarrow \texttt{d (2 + 2)}$$

Evaluation Order

- consider `d x = x + x`
- term `d (d 2)` may be evaluated as follows

$$
\begin{tikzcd}
 & \texttt{d (d 2)} \arrow[dl] \arrow[dr] & \\
\texttt{d (2 + 2)} & & \texttt{d 2 + d 2}
\end{tikzcd}
$$

Evaluation Order

- consider d x = x + x
- term d (d 2) may be evaluated as follows

$$
\begin{tikzcd}[column sep=small]
 & & \texttt{d (d 2)} \arrow[dl] \arrow[dr] & \\
 & \texttt{d (2 + 2)} \arrow[dl] & & \texttt{d 2 + d 2} \\
\texttt{d 4} & & &
\end{tikzcd}
$$

Evaluation Order

- consider d x = x + x
- term d (d 2) may be evaluated as follows

$$\begin{tikzcd}
 & & \mathtt{d\ (d\ 2)} \arrow[dl] \arrow[dr] & \\
 & \mathtt{d\ (2 + 2)} \arrow[dl] \arrow[dr, bend left] & & \mathtt{d\ 2 + d\ 2} \\
\mathtt{d\ 4} & & \mathtt{(2 + 2) + (2 + 2)} &
\end{tikzcd}$$

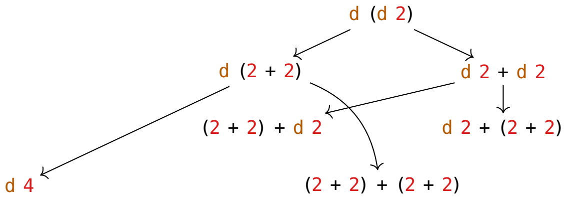
Evaluation Order

- consider d x = x + x
- term d (d 2) may be evaluated as follows

$$
\begin{tikzcd}
 & & \mathrm{d}\ (\mathrm{d}\ 2) \arrow[dl] \arrow[dr] & \\
 & \mathrm{d}\ (2 + 2) \arrow[ddl] \arrow[ddr, bend left] & & \mathrm{d}\ 2 + \mathrm{d}\ 2 \arrow[dll] \\
 & (2 + 2) + \mathrm{d}\ 2 & & \\
\mathrm{d}\ 4 & & (2 + 2) + (2 + 2) &
\end{tikzcd}
$$

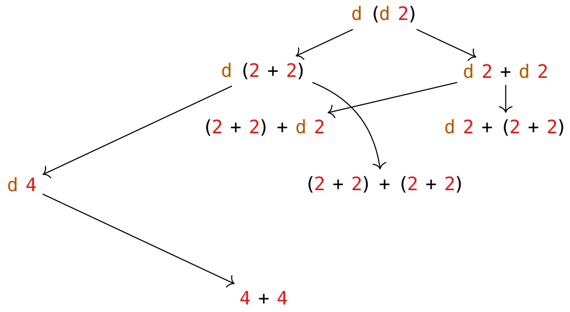
Evaluation Order

- consider d x = x + x
- term d (d 2) may be evaluated as follows



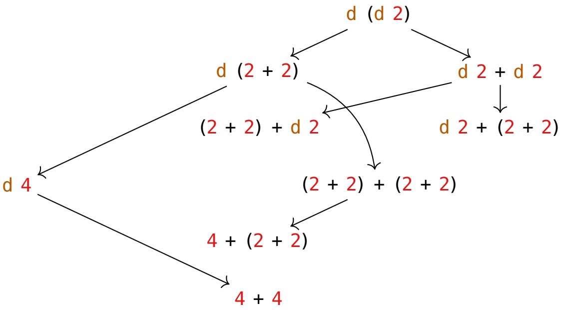
Evaluation Order

- consider `d x = x + x`
- term `d (d 2)` may be evaluated as follows



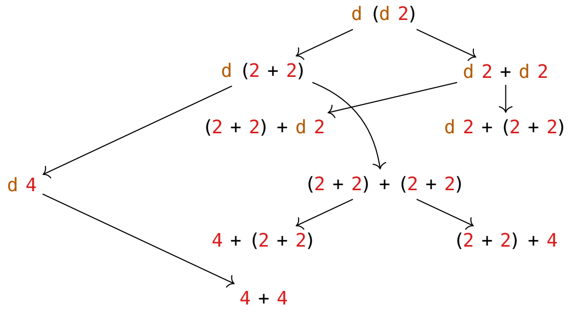
Evaluation Order

- consider d x = x + x
- term d (d 2) may be evaluated as follows



Evaluation Order

- consider d x = x + x
- term d (d 2) may be evaluated as follows



Evaluation Order

- consider `d x = x + x`
- term `d (d 2)` may be evaluated as follows

$$
\begin{tikzcd}[column sep=small]
 & & & \mathtt{d\ (d\ 2)} \arrow[dl] \arrow[dr] & \\
 & & \mathtt{d\ (2 + 2)} \arrow[ddll] \arrow[ddr, bend left] & & \mathtt{d\ 2 + d\ 2} \arrow[dll] \arrow[d] \\
 & & \mathtt{(2 + 2)\ +\ d\ 2} \arrow[dl] & & \mathtt{d\ 2 + (2 + 2)} \\
\mathtt{d\ 4} \arrow[ddrr] & \mathtt{4 + d\ 2} & & \mathtt{(2 + 2) + (2 + 2)} \arrow[dl] \arrow[dr] & \\
 & & \mathtt{4 + (2 + 2)} & & \mathtt{(2 + 2)\ +\ 4} \\
 & & \mathtt{4 + 4} & &
\end{tikzcd}
$$

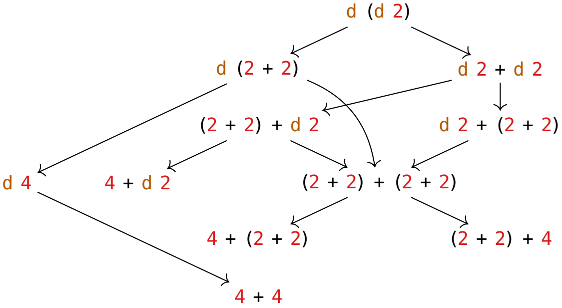
Evaluation Order

- consider d x = x + x
- term d (d 2) may be evaluated as follows

$$
\begin{tikzcd}[column sep=small]
 & & & \text{d (d 2)} \arrow[dl] \arrow[dr] & \\
 & & \text{d (2 + 2)} \arrow[ddll] \arrow[ddr, bend left] & & \text{d 2 + d 2} \arrow[dll] \arrow[d] \\
 & & \text{(2 + 2) + d 2} \arrow[dl] \arrow[dr] & & \text{d 2 + (2 + 2)} \\
\text{d 4} \arrow[ddrr] & \text{4 + d 2} & & \text{(2 + 2) + (2 + 2)} \arrow[dl] \arrow[dr] & \\
 & & \text{4 + (2 + 2)} & & \text{(2 + 2) + 4} \\
 & & \text{4 + 4} & &
\end{tikzcd}
$$

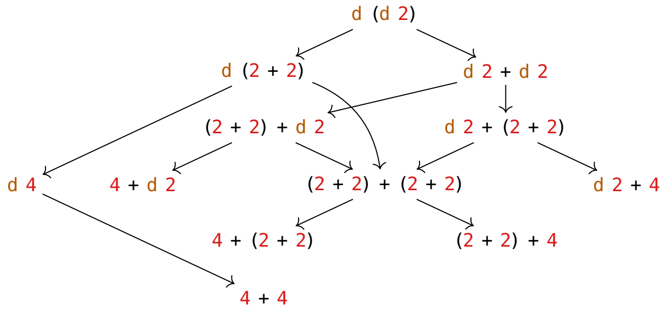
Evaluation Order

- consider d x = x + x
- term d (d 2) may be evaluated as follows



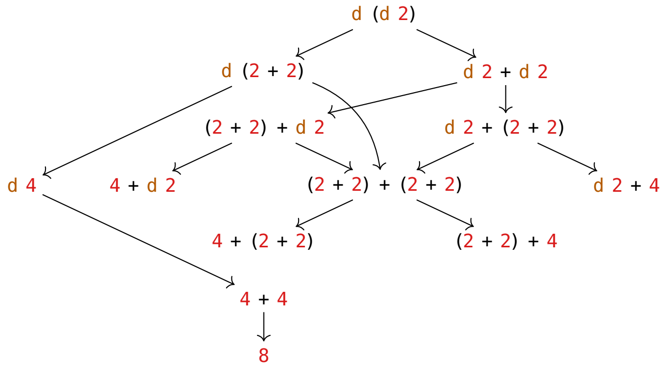
Evaluation Order

- consider `d x = x + x`
- term `d (d 2)` may be evaluated as follows



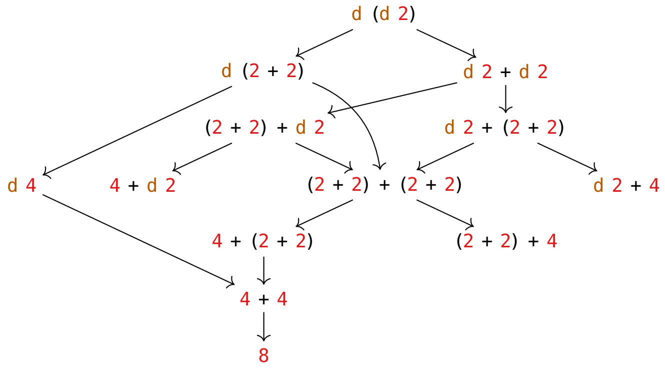
Evaluation Order

- consider d x = x + x
- term d (d 2) may be evaluated as follows



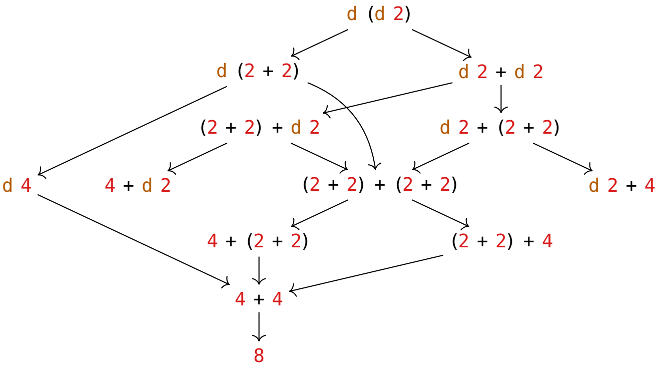
Evaluation Order

- consider `d x = x + x`
- term `d (d 2)` may be evaluated as follows



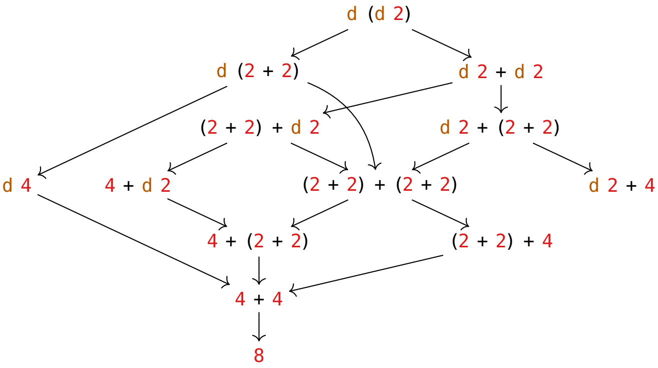
Evaluation Order

- consider d x = x + x
- term d (d 2) may be evaluated as follows



Evaluation Order

- consider d x = x + x
- term d (d 2) may be evaluated as follows



Evaluation Order

- consider `d x = x + x`
- term `d (d 2)` may be evaluated as follows

$$
\begin{tikzcd}
 & & & \mathtt{d\ (d\ 2)} \arrow[dl] \arrow[dr] & & \\
 & & \mathtt{d\ (2 + 2)} \arrow[ddll] \arrow[ddr, bend left] & & \mathtt{d\ 2 + d\ 2} \arrow[dll] \arrow[d] & \\
 & & \mathtt{(2 + 2) + d\ 2} \arrow[dl] \arrow[dr] & & \mathtt{d\ 2 + (2 + 2)} \arrow[dl] \arrow[dr] & \\
\mathtt{d\ 4} \arrow[ddrr] & \mathtt{4 + d\ 2} \arrow[dr] & & \mathtt{(2 + 2) + (2 + 2)} \arrow[dl] \arrow[dr] & & \mathtt{d\ 2 + 4} \arrow[dl] \\
 & & \mathtt{4 + (2 + 2)} \arrow[d] & & \mathtt{(2 + 2) + 4} \arrow[dll] & \\
 & & \mathtt{4 + 4} \arrow[d] & & & \\
 & & \mathtt{8} & & &
\end{tikzcd}
$$

(Reduction) Strategies

what is called evaluation strategy in programming, is typically called reduction strategy in λ-calculus

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: pass expressions rather than their results)

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: pass expressions rather than their results)

Example

- call by value

```
d (d 2)
```

- call by name

```
d (d 2)
```

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: pass expressions rather than their results)

Example

- call by value

$$\mathsf{d}\ (\mathsf{d}\ 2) = \mathsf{d}\ (2 + 2)$$

- call by name

$$\mathsf{d}\ (\mathsf{d}\ 2)$$

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: pass expressions rather than their results)

Example

- call by value

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

- call by name

$$d\ (d\ 2)$$

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: pass expressions rather than their results)

Example

- call by value

$$\begin{aligned} \mathrm{d}\ (\mathrm{d}\ 2) &= \mathrm{d}\ (2 + 2) \\ &= \mathrm{d}\ 4 \\ &= 4 + 4 \end{aligned}$$

- call by name

$$\mathrm{d}\ (\mathrm{d}\ 2)$$

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: pass expressions rather than their results)

Example

- call by value

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

- call by name

$$d\ (d\ 2)$$

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: pass expressions rather than their results)

Example

- call by value

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

- call by name

$$\mathsf{d}\ (\mathsf{d}\ 2) = \mathsf{d}\ 2 + \mathsf{d}\ 2$$

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: pass expressions rather than their results)

Example

- call by value

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

- call by name

$$
\begin{aligned}
d\ (d\ 2) &= d\ 2 + d\ 2\\
&= (2 + 2) + d\ 2
\end{aligned}
$$

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: pass expressions rather than their results)

Example

- call by value

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

- call by name

$$
\begin{aligned}
\mathrm{d}\ (\mathrm{d}\ 2) &= \mathrm{d}\ 2 + \mathrm{d}\ 2\\
&= (2 + 2) + \mathrm{d}\ 2\\
&= 4 + \mathrm{d}\ 2
\end{aligned}
$$

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: pass expressions rather than their results)

Example

- call by value

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

- call by name

$$\begin{aligned} \mathrm{d}\ (\mathrm{d}\ 2) &= \mathrm{d}\ 2 + \mathrm{d}\ 2 \\ &= (2 + 2) + \mathrm{d}\ 2 \\ &= 4 + \mathrm{d}\ 2 \\ &= 4 + (2 + 2) \end{aligned}$$

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: pass expressions rather than their results)

Example

- call by value

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

- call by name

$$\begin{aligned} \mathrm{d}\ (\mathrm{d}\ 2) &= \mathrm{d}\ 2 + \mathrm{d}\ 2 \\ &= (2 + 2) + \mathrm{d}\ 2 \\ &= 4 + \mathrm{d}\ 2 \\ &= 4 + (2 + 2) \\ &= 4 + 4 \end{aligned}$$

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: pass expressions rather than their results)

Example

- call by value

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

- call by name

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

Applicative Order Reduction

- reduce rightmost innermost redex
- redex is innermost if it does not contain redexes itself

Applicative Order Reduction

- reduce rightmost innermost redex
- redex is innermost if it does not contain redexes itself

Example

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

Normal Order Reduction

- reduce leftmost outermost redex
- redex is outermost if it is not contained in another redex

Normal Order Reduction

- reduce leftmost outermost redex
- redex is outermost if it is not contained in another redex

Example

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

Examples

- consider the λ-terms
- $S = \lambda xyz.\, x\ z\ (y\ z)$
- $K = \lambda xy.\, x$
- $I = \lambda x.\, x$
- reduce $S\ K\ I$ to NF using applicative order reduction
- reduce $S\ K\ I$ to NF using normal order reduction

Further Classification of λ-Terms

- term is value iff not application

Further Classification of λ-Terms

- term is value iff not application
- term t is (in) weak head normal form (WHNF) iff $\mathrm{whnf}(t) = \mathrm{true}$:

$$
\begin{aligned}
\mathrm{whnf}(x) &= \mathrm{true} \\
\mathrm{whnf}(\lambda x.\, t) &= \mathrm{true} \\
\mathrm{whnf}((\lambda x.\, t)\; u) &= \mathrm{false} \\
\mathrm{whnf}(t\; u) &= \mathrm{whnf}(t) && \text{if } t \text{ not abstraction}
\end{aligned}
$$

Further Classification of λ-Terms

- term is value iff not application
- term t is (in) weak head normal form (WHNF) iff $\mathrm{whnf}(t) = \mathrm{true}$:

$$
\begin{aligned}
\mathrm{whnf}(x) &= \mathrm{true} \\
\mathrm{whnf}(\lambda x.\, t) &= \mathrm{true} \\
\mathrm{whnf}((\lambda x.\, t)\; u) &= \mathrm{false} \\
\mathrm{whnf}(t\; u) &= \mathrm{whnf}(t) && \text{if } t \text{ not abstraction}
\end{aligned}
$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\; x$		
$x\; y$		
x		
$\lambda x.\, (\lambda y.\, y)\; x$		

Further Classification of λ-Terms

- term is value iff not application
- term t is (in) weak head normal form (WHNF) iff $\mathsf{whnf}(t) = \mathsf{true}$:

$$
\begin{aligned}
\mathsf{whnf}(x) &= \mathsf{true} \\
\mathsf{whnf}(\lambda x.\, t) &= \mathsf{true} \\
\mathsf{whnf}((\lambda x.\, t)\; u) &= \mathsf{false} \\
\mathsf{whnf}(t\; u) &= \mathsf{whnf}(t) && \text{if } t \text{ not abstraction}
\end{aligned}
$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\; x$	✘	
$x\; y$		
x		
$\lambda x.\, (\lambda y.\, y)\; x$		

Further Classification of λ-Terms

- term is value iff not application
- term t is (in) weak head normal form (WHNF) iff $\mathrm{whnf}(t) = \mathrm{true}$:

$$
\begin{aligned}
\mathrm{whnf}(x) &= \mathrm{true} \\
\mathrm{whnf}(\lambda x.\, t) &= \mathrm{true} \\
\mathrm{whnf}((\lambda x.\, t)\; u) &= \mathrm{false} \\
\mathrm{whnf}(t\; u) &= \mathrm{whnf}(t) && \text{if } t \text{ not abstraction}
\end{aligned}
$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\; x$	✘	✘
$x\; y$		
x		
$\lambda x.\, (\lambda y.\, y)\; x$		

Further Classification of λ-Terms

- term is value iff not application
- term t is (in) weak head normal form (WHNF) iff $\mathrm{whnf}(t) = \mathrm{true}$:

$$
\begin{aligned}
\mathrm{whnf}(x) &= \mathrm{true} \\
\mathrm{whnf}(\lambda x.\, t) &= \mathrm{true} \\
\mathrm{whnf}((\lambda x.\, t)\; u) &= \mathrm{false} \\
\mathrm{whnf}(t\; u) &= \mathrm{whnf}(t) && \text{if } t \text{ not abstraction}
\end{aligned}
$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\; x$	✘	✘
$x\; y$	✘	
x		
$\lambda x.\, (\lambda y.\, y)\; x$		

Further Classification of λ-Terms

- term is value iff not application
- term t is (in) weak head normal form (WHNF) iff $\mathrm{whnf}(t) = \mathrm{true}$:

$$\begin{aligned}
\mathrm{whnf}(x) &= \mathrm{true} \\
\mathrm{whnf}(\lambda x.\, t) &= \mathrm{true} \\
\mathrm{whnf}((\lambda x.\, t)\; u) &= \mathrm{false} \\
\mathrm{whnf}(t\; u) &= \mathrm{whnf}(t) && \text{if } t \text{ not abstraction}
\end{aligned}$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\; x$	✘	✘
$x\; y$	✘	✔
x		
$\lambda x.\, (\lambda y.\, y)\; x$		

Further Classification of λ-Terms

- term is value iff not application
- term t is (in) weak head normal form (WHNF) iff $\text{whnf}(t) = \text{true}$:

$$
\begin{aligned}
\text{whnf}(x) &= \text{true} \\
\text{whnf}(\lambda x.\, t) &= \text{true} \\
\text{whnf}((\lambda x.\, t)\; u) &= \text{false} \\
\text{whnf}(t\; u) &= \text{whnf}(t) && \text{if } t \text{ not abstraction}
\end{aligned}
$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\; x$	✘	✘
$x\; y$	✘	✔
x	✔	
$\lambda x.\, (\lambda y.\, y)\; x$		

Further Classification of λ-Terms

- term is value iff not application
- term t is (in) weak head normal form (WHNF) iff $\mathrm{whnf}(t) = \mathrm{true}$:

$$
\begin{aligned}
\mathrm{whnf}(x) &= \mathrm{true} \\
\mathrm{whnf}(\lambda x.\, t) &= \mathrm{true} \\
\mathrm{whnf}((\lambda x.\, t)\; u) &= \mathrm{false} \\
\mathrm{whnf}(t\; u) &= \mathrm{whnf}(t) && \text{if } t \text{ not abstraction}
\end{aligned}
$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\; x$	✘	✘
$x\; y$	✘	✔
x	✔	✔
$\lambda x.\, (\lambda y.\, y)\; x$		

Further Classification of λ-Terms

- term is value iff not application
- term t is (in) weak head normal form (WHNF) iff $\mathrm{whnf}(t) = \mathrm{true}$:

$$
\begin{aligned}
\mathrm{whnf}(x) &= \mathrm{true} \\
\mathrm{whnf}(\lambda x.\, t) &= \mathrm{true} \\
\mathrm{whnf}((\lambda x.\, t)\; u) &= \mathrm{false} \\
\mathrm{whnf}(t\; u) &= \mathrm{whnf}(t) && \text{if } t \text{ not abstraction}
\end{aligned}
$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\; x$	✘	✘
$x\; y$	✘	✔
x	✔	✔
$\lambda x.\, (\lambda y.\, y)\; x$	✔	

Further Classification of λ-Terms

- term is value iff not application
- term t is (in) weak head normal form (WHNF) iff $\mathrm{whnf}(t) = \mathrm{true}$:

$$\begin{aligned}
\mathrm{whnf}(x) &= \mathrm{true} \\
\mathrm{whnf}(\lambda x.\, t) &= \mathrm{true} \\
\mathrm{whnf}((\lambda x.\, t)\ u) &= \mathrm{false} \\
\mathrm{whnf}(t\ u) &= \mathrm{whnf}(t) && \text{if } t \text{ not abstraction}
\end{aligned}$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\ x$	✘	✘
$x\ y$	✘	✔
x	✔	✔
$\lambda x.\, (\lambda y.\, y)\ x$	✔	✔

Call by Value

- stop at values
- otherwise choose outermost redex whose right-hand side is value
- corresponds to strict (or eager) evaluation
- adopted by most programming languages

Call by Name

- stop at WHNFs
- otherwise same as normal order (that is, leftmost outermost redex)
- corresponds to lazy evaluation (without memoization)
- adopted for example by Haskell

Outline

Idea

- hide implementation details
- just provide interface
- allows us to change implementation (e.g., make more efficient) without breaking client code

Idea

- hide implementation details
- just provide interface
- allows us to change implementation (e.g., make more efficient) without breaking client code

Haskell

- consider module

```haskell
module M (T, ...) where
data T = C1 | ... | CN
```

- only name `T` is exported, but none of constructors `C1` to `CN`
- thus we are not able to directly construct values of type `T`
- if we want to export `C1` to `CN`, we can use `T(..)` in export list

Characteristics of Sets

- order of elements not important
- no duplicates

Characteristics of Sets

- order of elements not important
- no duplicates

Examples

$$\{1,2,3,5\} = \{5,1,3,2\}$$
$$\{1,1,2,2\} = \{1,2\}$$

Characteristics of Sets

- order of elements not important
- no duplicates

Examples

$$\{1, 2, 3, 5\} = \{5, 1, 3, 2\}$$
$$\{1, 1, 2, 2\} = \{1, 2\}$$

Operations on Sets

description	notation	Haskell
empty set	$\emptyset$	`empty :: Set a`
insertion	$\{x\} \cup S$	`insert :: a -> Set a -> Set a`
membership	$e \in S$	`mem :: a -> Set a -> Bool`
union	$S \cup T$	`union :: Set a -> Set a -> Set a`
difference	$S \setminus T$	`diff :: Set a -> Set a -> Set a`

Example (Sets as Lists)

```haskell
module Set (Set,empty,insert,mem,union,diff,...) where
import qualified Data.List as List
data Set a = Set [a]

empty :: Set a
empty = Set []

insert :: Eq a => a -> Set a -> Set a
insert x (Set xs) = Set $ List.nub $ x:xs

mem :: Eq a => a -> Set a -> Bool
x `mem` Set xs = x `elem` xs

union, diff :: Eq a => Set a -> Set a -> Set a
union (Set xs) (Set ys) = Set $ List.nub $ xs ++ ys
diff (Set xs) (Set ys) = Set $ xs List.\\ ys
```

New Types

- **data** with single constructor `Set` used to hide implementation
- in this common special case use `newtype Set a = Set [a]` instead
- only difference: **newtype** has better performance than **data**

New Types

- **data** with single constructor `Set` used to hide implementation
- in this common special case use `newtype Set a = Set [a]` instead
- only difference: **newtype** has better performance than **data**

Record Syntax

- for data type / new type `T`, instead of `C t1 ... tN`, we may use
- `C {n1 :: t1, ..., nN :: tN}` as constructor
- provides selector functions `n1 :: T -> t1, ..., nN :: T -> tN`

New Types

- **data** with single constructor `Set` used to hide implementation
- in this common special case use `newtype Set a = Set [a]` instead
- only difference: **newtype** has better performance than **data**

Record Syntax

- for data type / new type `T`, instead of `C t1 ... tN`, we may use
- `C {n1 :: t1, ..., nN :: tN}` as constructor
- provides selector functions `n1 :: T -> t1, ..., nN :: T -> tN`

New Types

- **data** with single constructor `Set` used to hide implementation
- in this common special case use `newtype Set a = Set [a]` instead
- only difference: **newtype** has better performance than **data**

Record Syntax

- for data type / new type `T`, instead of `C t1 ... tN`, we may use
- `C {n1 :: t1, ..., nN :: tN}` as constructor
- provides selector functions `n1 :: T -> t1, ..., nN :: T -> tN`

Example

- `data Equation a = E { lhs :: a, rhs :: a }`

```
ghci> let e1 = E "10" "5+5"
ghci> let e2 = E { rhs = "5+5", lhs = "10" }
ghci> lhs e1
"10"
ghci> rhs e2
"5+5"
```

Outline

The Type

- use type BTree without prefix: `import BTree (BTree(..))`
- import remaining functions from BTree with prefix

```
import qualified BTree
```

- internal representation of set is binary tree (with selector rep)

```
newtype Set a = Set { rep :: BTree a }
```

The Type

- use type `BTree` without prefix: `import BTree (BTree(..))`
- import remaining functions from `BTree` with prefix

```
import qualified BTree
```

- internal representation of set is binary tree (with selector `rep`)

```
newtype Set a = Set { rep :: BTree a }
```

Remark

- `newtype Set a = Set { rep :: BTree a }` is almost same as writing `type Set a = BTree a`
- additionally type system prevents “accidental” (that is, without constructor `Set`) use of `BTree`s as `Set`s
- no runtime penalty (in contrast to `data Set a = Set { rep :: BTree }`)
- reason: **newtype** restricted to single constructor (usually of same name as newly introduced type)
- **data** may have arbitrarily many constructors (e.g., `Empty` and `Node`)

Empty Set

```
empty :: Set a
empty = Set Empty
```

Membership

```
mem :: Ord a => a -> Set a -> Bool
x `mem` s = x `memTree` rep s
  where
    memTree x Empty = False
    memTree x (Node y l r) =
      case compare x y of
        EQ -> True
        LT -> x `memTree` l
        GT -> x `memTree` r
```

Insertion

```haskell
insert :: Ord a => a -> Set a -> Set a
insert x s = Set $ insertTree x $ rep s

insertTree :: Ord a => a -> BTree a -> BTree a
insertTree x Empty        = Node x Empty Empty
insertTree x (Node y l r) =
  case compare x y of
    EQ -> Node y l r
    LT -> Node y (insertTree x l) r
    GT -> Node y l (insertTree x r)
```

Union

```
union :: Ord a => Set a -> Set a -> Set a
union s t = Set $ rep s `unionTree` rep t

unionTree :: Ord a => BTree a -> BTree a -> BTree a
unionTree Empty s        = s
unionTree (Node x l r) s =
  insertTree x $ l `unionTree` r `unionTree` s
```

Removing the Maximal Element

```
splitMaxFromTree :: BTree a -> Maybe (a, BTree a)
splitMaxFromTree Empty            = Nothing
splitMaxFromTree (Node x l Empty) = Just (x, l)
splitMaxFromTree (Node x l r)     =
  let Just (m, r') = splitMaxFromTree r
  in Just (m, Node x l r')
```

Removing the Maximal Element

```haskell
splitMaxFromTree :: BTree a -> Maybe (a, BTree a)
splitMaxFromTree Empty            = Nothing
splitMaxFromTree (Node x l Empty) = Just (x, l)
splitMaxFromTree (Node x l r)     =
  let Just (m, r') = splitMaxFromTree r
  in Just (m, Node x l r')
```

The Maybe Type

- Prelude: `data Maybe a = Just a | Nothing`
- used for type-based error handling
- if an error occurs, we return `Nothing`
- otherwise `Just` the result

Removing the Maximal Element

```
splitMaxFromTree :: BTree a -> Maybe (a, BTree a)
splitMaxFromTree Empty            = Nothing
splitMaxFromTree (Node x l Empty) = Just (x, l)
splitMaxFromTree (Node x l r)     =
  let Just (m, r') = splitMaxFromTree r
  in Just (m, Node x l r')
```

The Maybe Type

- Prelude: `data Maybe a = Just a | Nothing`
- used for type-based error handling
- if an error occurs, we return `Nothing`
- otherwise `Just` the result

Example (safe head)

```
safeHead (x:_) = Just x
safeHead _     = Nothing
```

Remove Given Element

```haskell
removeFromTree :: Ord a => a -> BTree a -> BTree a
removeFromTree x Empty        = Empty
removeFromTree x (Node y l r) = case compare x y of
  LT -> Node y (removeFromTree x l) r
  GT -> Node y l (removeFromTree x r)
  EQ -> case splitMaxFromTree l of
    Nothing      -> r
    Just (m, l') -> Node m l' r
```

Remove Given Element

```
removeFromTree :: Ord a => a -> BTree a -> BTree a
removeFromTree x Empty        = Empty
removeFromTree x (Node y l r) = case compare x y of
  LT -> Node y (removeFromTree x l) r
  GT -> Node y l (removeFromTree x r)
  EQ -> case splitMaxFromTree l of
    Nothing      -> r
    Just (m, l') -> Node m l' r
```

For Binary Search Tree (BST)

- x smaller y: x can only occur in l
- x greater y: x can only occur in r
- x equals y: remove current node and
- combine l and r into new BST
- therefore, take maximum of l as new root
- guarantees that all other elements in l are smaller and
- that all elements in r are greater

Difference

```
diff :: Ord a => Set a -> Set a -> Set a
diff s t = Set $ rep s `diffTree` rep t

diffTree :: Ord a => BTree a -> BTree a -> BTree a
diffTree t Empty        = t
diffTree t (Node x l r) =
  removeFromTree x t `diffTree` l `diffTree` r
```

Thanks! & Questions?