# IB015 Neimperativní programování

## Akumulační funkce, Typové třídy, Časová složitost

Jiří Barnat
Libor Škarvada

# Akumulační funkce

# Akumulační funkce `foldl, foldr`

**Pozorování**

- Seznam je posloupnost oddělených prvků.
- Motivací akumulačních funkcí je "spojit" jednotlivé prvky seznamu dohromady, tj. akumulovat informaci uloženou v těchto jednotlivých prvcích do jedné hodnoty.
- Počet prvků seznamu je variabilní, proto se tato akumulace realizuje pomocí binárního operátoru postupně.

**Spojení hodnot v seznamu pomocí binární operace**

$$\texttt{foldl1}\ \oplus\ [x_1,x_2,\ldots,x_n] \rightsquigarrow^* ((\ldots(x_1\oplus x_2)\ldots)\oplus x_{n-1})\oplus x_n$$

$$\texttt{foldr1}\ \oplus\ [x_1,x_2,\ldots,x_n] \rightsquigarrow^* x_1\oplus(x_2\oplus(\ldots(x_{n-1}\oplus\ x_n)\ldots))$$

# Akumulační funkce na neprázdných seznamech

## Příklady použití

```
foldl1 (*) [1,2,3,4,5] ⇝ ... ⇝ 120
foldl1 (&&) [True, True, True, False, True] ⇝ ... ⇝ False
foldl1 (-) [2,3,2] ⇝ ... ⇝ -3
foldr1 (-) [2,3,2] ⇝ ... ⇝ 1
foldr1 (min) [18,12,23] ⇝ ... ⇝ 12
```

## Funkce `foldl1, foldr1` nejsou definovány pro `[]`

```
foldl1 (*) [] ⇝ ERROR
foldr1 (&&) [] ⇝ ERROR
```

## Na jednoprvkových seznamech je to identita s kontrolou typu

```
foldr1 (*) [0] ⇝ 0
foldr1 (*) [1] ⇝ 1
foldr1 (*) [True] ⇝ ERROR
```

# Akumulační funkce `foldl, foldr`

## Princip

- Akumulační funkce, které mají fungovat i na prázdných seznamech, vyžadují navíc **iniciální hodnotu** pro proces akumulace.
- Směr závorkování určuje i místo použití iniciální hodnoty.

## Akumulace hodnot s využitím iniciální hodnoty

$$\texttt{foldl}\ \oplus\ \mathbf{v}\ [x_1,x_2,\ldots,x_n] \rightsquigarrow^* ((\ldots((\mathbf{v}\oplus x_1)\oplus x_2)\ldots)\oplus x_{n-1})\oplus x_n$$

$$\texttt{foldr}\ \oplus\ \mathbf{v}\ [x_1,x_2,\ldots,x_n] \rightsquigarrow^* x_1\oplus(x_2\oplus(\ldots(x_{n-1}\oplus(x_n\oplus\mathbf{v}))\ldots))$$

# Akumulační funkce na libovolných seznamech

## Příklady

```
foldl (*) 0 [1,2,3,4,5] ⇝ ... ⇝ 0
foldl (&&) False [True, True, True, True] ⇝ ... ⇝ False
foldl (-) 0 [2,3,2] ⇝ ... ⇝ -7
foldr (-) 0 [2,3,2] ⇝ ... ⇝ 1
```

## Aplikace na prázdné seznamy

```
foldl max 100 [] ⇝ ... ⇝ 100
foldr (++) "Nic"[] ⇝ ... ⇝ "Nic"
```

## Výsledek může být opět seznam!

```
foldr (:) [] "Coze?"⇝ ... ⇝ "Coze?"
foldr (\x y->(x+1):y) [100] [1,2,3] ⇝ ... ⇝ [2,3,4,100]
```

# Definice akumulačních funkcí

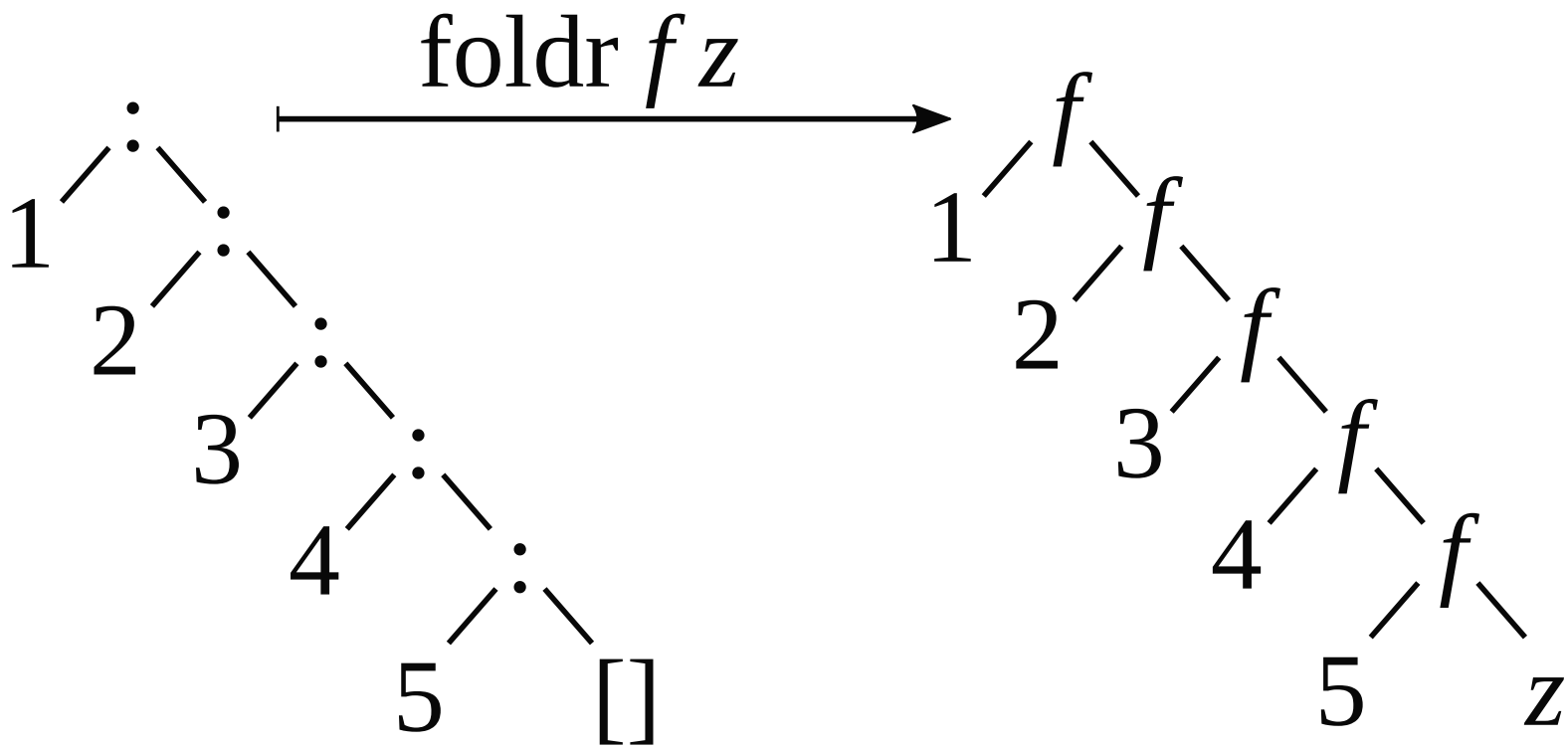
```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ v [] = v
foldl op v (x:s) = foldl op (v `op` x) s


foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ v [] = v
foldr op v (x:s) = x `op` foldr op v s

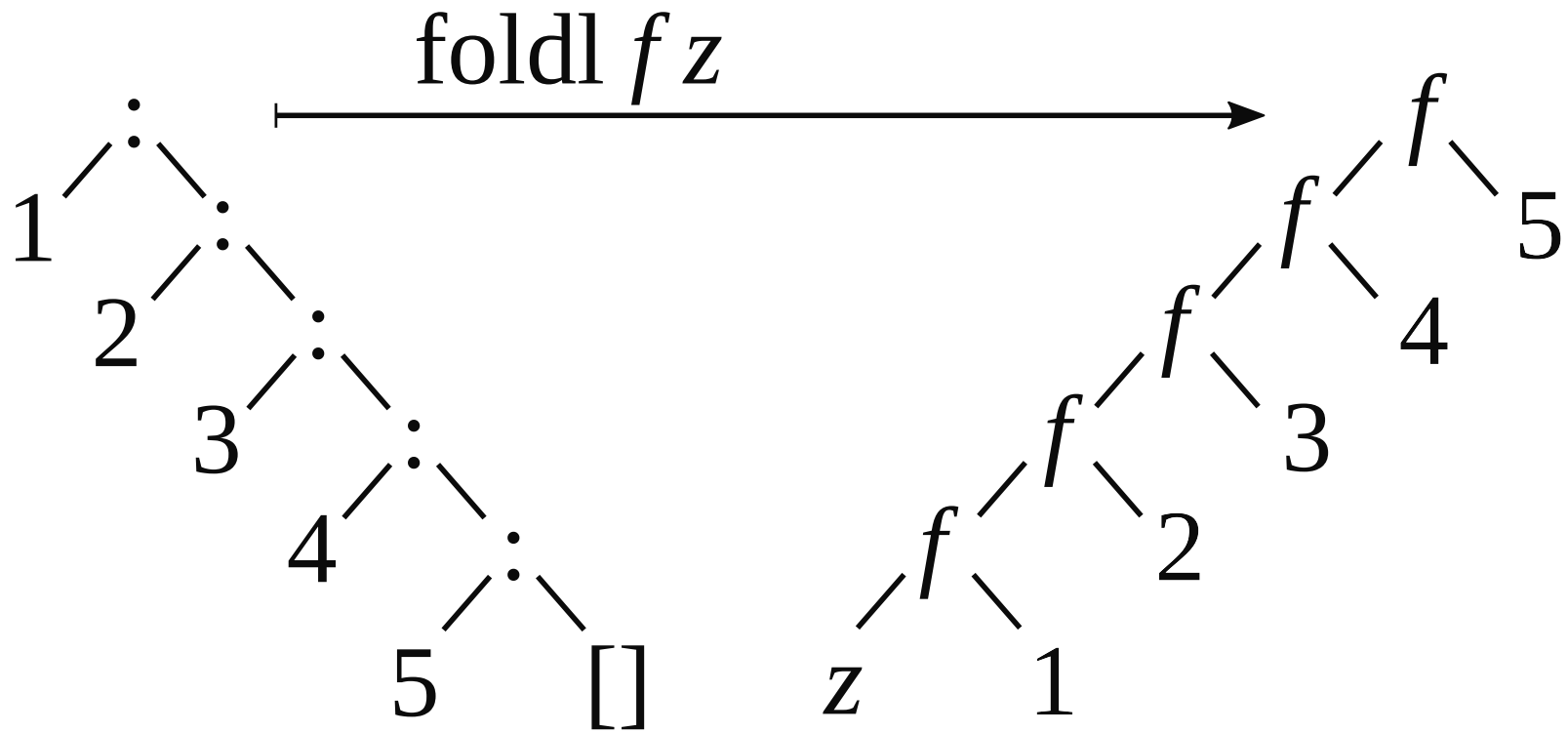

foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 op (x:s) = foldl op x s


foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 _ [x] = x
foldr1 op (x:s) = x `op` foldr1 op s
```

# Grafické znázornění `foldr` (katamorfismus)



# Grafické znázornění `foldl`



# Katamorfismus na seznamech

**Katamorfismus**

- Výraz vzniklý nahrazením hodnotových konstruktorů v nějaké hodnotě algebraického typu jinými funkcemi vhodné arity.
- Nově vzniklý výraz je následně možné vyhodnotit (pokud lze).

**Katamorfismus na seznamech**

- Realizován funkcí `foldr` .
- Porovnejte typy hodnotových konstruktorů seznamu

```
(:) :: a -> [a] -> [a]
[] :: [a]
```

s typem funkce `foldr`

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

# Katamorfismus na Peanových číslech

## Peanova čísla

- `data Nat = Succ Nat | Zero`

## Katamorfismus na typu `Nat`

- ```
  natFold :: (a -> a) -> a -> Nat -> a
  natFold s z (Succ n) = s (natFold s z n)
  natFold s z Zero = z
  ```

## Příklad

- Katamorfismus: `Zero` $\longrightarrow$ `0`
  `Succ` $\longrightarrow$ `(1+)`
- `natFold (1+) 0 (Succ (Succ Zero)` $\rightsquigarrow^*$ `(1+) ( (1+) 0 )` $\rightsquigarrow^*$ `2`
  ```

# Katamorfismus na binárních stromech

## Připomenutí

- `data BinTree a = Node a (BinTree a) (BinTree a) | Empty`

## Hodnotové konstruktory

- ```
  Node :: a -> BinTree a -> BinTree a -> BinTree a
  Empty :: BinTree a
  ```
- Hodnotový konstruktor `Node` je ternární, i když se používá pro konstrukci binárních stromů.

## Katamorfismus na binárních stromech

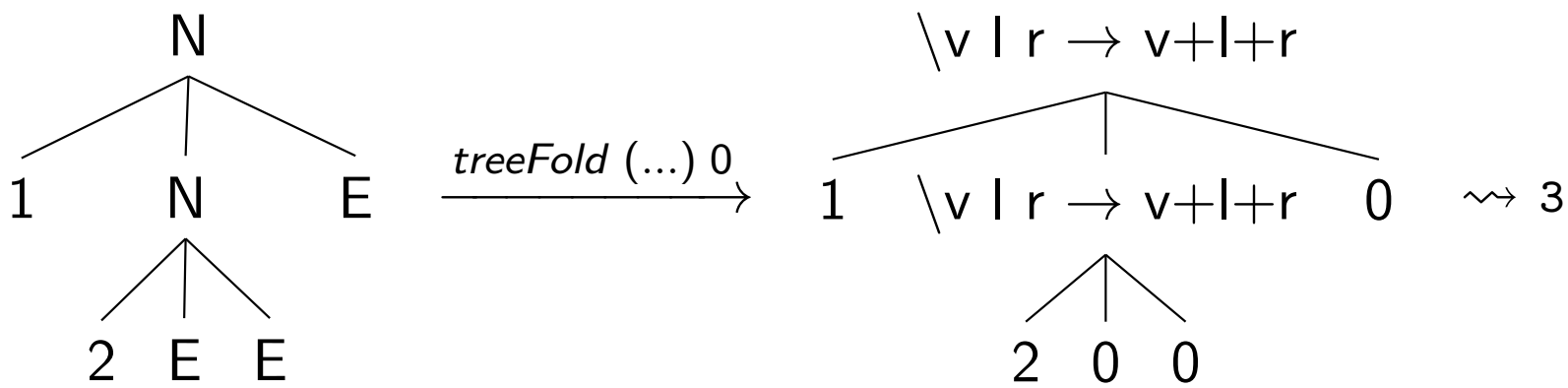
- ```
  treeFold :: (a -> b -> b -> b) -> b -> BinTree a -> b
  treeFold n e (Node v l r) = n v (treeFold n e l)
                                  (treeFold n e r)
  treeFold n e Empty = e
  ```

# Katamorfismus na binárních stromech – příklad

## Fold na binárních stromech

- `data BinTree a = Node a (BinTree a) (BinTree a) | Empty`
- ```
  let tree = Node 1 (Node 2 Empty Empty) Empty in
      treeFold (\v l r -> v+l+r) 0 tree
  ```
  $\rightsquigarrow^*$ `1+(2+0+0)+0` $\rightsquigarrow^*$ `3`

## Graficky:


  ```

# Typové třídy

# Připomenutí klasifikace typů

## Monomorfní typy

- `not :: Bool -> Bool`

  `(&&) :: Bool -> Bool -> Bool`

## Polymorfní typy

- `length :: [a] -> Int`

  `flip :: (a -> b -> c) -> b -> a -> c`

## Kvalifikované typy

- `(==), (/=) :: Eq a => a -> a -> Bool`

  `sum, product :: Num a => [a] -> a`

  `minimum, maximum :: Ord a => [a] -> a`

  `print :: Show a => a -> IO ()`

# Typové třídy

## Typové třídy

- Sdružují a identifikují typy se společnými vlastnostmi.
- Typová kvalifikace – `Ord,Nat,Eq,Show,Foldable,...`

## Programátorský význam

- Existence typových tříd umožňuje i ve striktně typovaných jazycích sdílet kód funkcí, které dělají totéž, avšak pracují s hodnotami různých typů.

- Sdílení kódu funkcí, které dělají totéž, by měl být **svatý grál** všech programátorů.

# Příklad typové třídy a jejího využití

## Typová třída Eq

- ```
  class Eq a where
      (==), (/=) :: a -> a -> Bool
      x /= y = not (x == y)
  ```

## Přidružení typů k typové třídě (deklarace instance)

- ```
  instance Eq Bool where
      False == False = True
      True == True = True
      _ == _ = False
  ```
- ```
  instance Eq Int where
      (==) = primEqInt
  ```
- ```
  instance (Eq a, Eq b) => Eq (a,b) where
      (x,y) == (u,v) = x == u && y == v
  ```

# Využití typové třídy jinou typovou třídou

## Typová třída Ord využívající typovou třídu Eq

- ```
  class (Eq a) => Ord a where
      (<=), (>=), (<), (>) :: a -> a -> Bool
      max, min :: a -> a -> a
      x >= y = y <= x
      x < y = x <= y && x /= y
      x > y = y < x
      max x y = if x>= y then x else y
      min x y = if x<= y then x else y
  ```

## Deklarace instance

- ```
  instance Ord Bool where
      False <= _ = True
      _ <= True = True
      _ <= _ = False
  ```
- ```
  instance (Ord a, Ord b) => Ord (a,b) where
      (x,y) <= (u,v) = x < u || (x == u && y <= v)
  ```
  ```

# Přenos vlastností typu na složený typ

**Pozorování**

- Instanciací lze přenést vlastnosti typu na složené typy.

**Příklad**

- Rozšíření uspořadatelnosti hodnot typu na uspořadatelnost seznamů hodnot daného typu.
- ```
  instance (Ord a) => Ord [a] where
      [] <= _ = True
      (_:_) <= [] = False
      (x:s) <= (y:t) = x < y || (x == y && s <= t)
  ```
  ```

# Obecná definice typové třídy a deklarace instance

## Definice typové třídy

- `class` $[\ (C_1\ a, \ldots, C_k\ a)$ `=>` $]\ C\ a$
  $\left[\ \texttt{where}\ \begin{array}{l} op_1 :: typ_1 \\ \vdots \\ op_n :: typ_n \\ \left[\ \begin{array}{l} default_1 \\ \vdots \\ default_m \end{array}\ \right] \end{array}\ \right]$

## Deklarace instance

- `instance` $[\ (C_1\ a_1, \ldots, C_k\ a_k)$ `=>` $]\ C\ typ$
  $\left[\ \texttt{where}\ \begin{array}{l} valdef_1 \\ \vdots \\ valdef_n \end{array}\ \right]$

# Přetížení

**Přetížení**

- Má-li třída více než jednu instanci, jsou její funkce **přetíženy**.

**Přetížení operací**

- Jedna operace je pro několik různých typů operandů definována obecně různým způsobem.
- To, která definice operace se použije při výpočtu, závisí na typu operandů, se kterými operace pracuje.
- Srovnej s parametricky polymorfními operacemi, které jsou definovány jednotně pro všechny typy operandů.

# Příklad přetížení operace

## Typová třída Num

- `class (Eq a, Show a) => Num a where`
  `(+), (-), (*) :: a -> a -> a`
  `negate, abs, signum :: a -> a`

## Přetížení operací při deklaraci instancí

- `instance Num Int where`
  `(+) = primPlusInt`
  ⋮
- `instance Num Integer where`
  `(+) = primPlusInteger`
  ⋮
- `instance Num Float where`
  `(+) = primPlusFloat`
  ⋮

# Implicitní deklarace instance

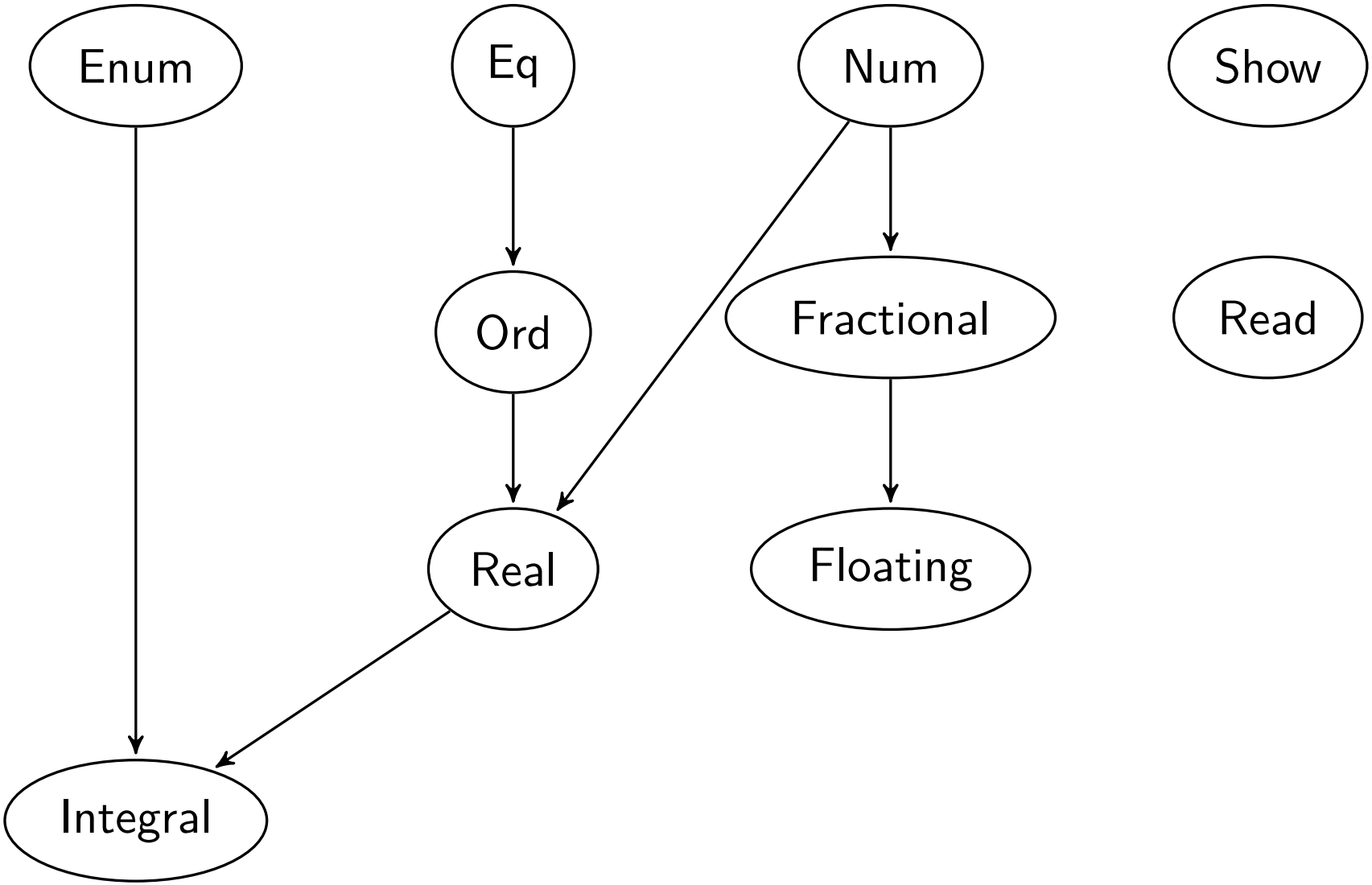
## Implicitní deklarace instance

- V Haskellu lze deklarovat datový typ jako instanci typové třídy (nebo více typových tříd) též implicitně, pomocí klausule `deriving` v definici datového typu.
- Při implicitní deklaraci instance se požadované funkce definují automaticky podle způsobu zápisu hodnot definovaného typu
- Funkce `(==)` se při implicitní deklaraci instance realizuje jako syntaktická rovnost.

## Příklad

- ```
  data Nat = Zero | Succ Nat
      deriving (Eq, Show)
  ```
  ```

# Hierarchie základních typových tříd



# Definice typové třídy – příklad

## Definice typové třídy

- `class Boolable a where getBool :: a -> Bool`

## Instanciace

- ```
  instance Boolable Bool where
      getBool x = x
  ```
- ```
  instance Boolable Int where
      getBool 0 = False
      getBool _ = True
  ```

## Použití

- ```
  myIf :: Boolable a => a -> b -> b -> b
  myIf x t f = if getBool x then t else f
  ```
- `myIf (3-3) "Pravda" "Nepravda"` $\rightsquigarrow^*$ `"Nepravda"`
- `myIf (3-4) "Pravda" "Nepravda"` $\rightsquigarrow^*$ `"Pravda"`
  ```

# Časová složitost

# Časová složitost algoritmu

**Podstata**

- Časová složitost funkce popisuje **délku výpočtu** v nejhorším případě vzhledem k velikosti vstupních parametrů.

**Délka výpočtu v nejhorším případě**

- Maximální počet redukčních kroků přes všechny možné výpočty aplikace programu na vstupní parametry stejné velikosti.

# Časová složitost algoritmu

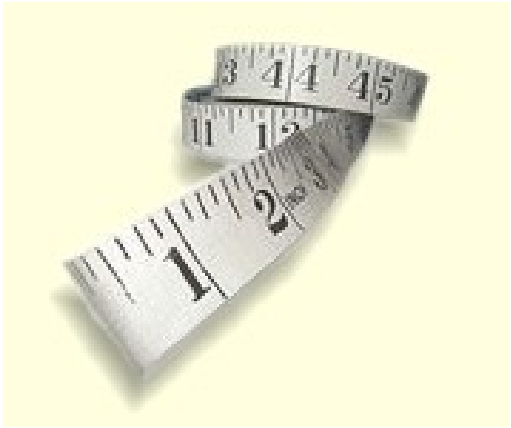
**Podstata**

- Časová složitost funkce popisuje **délku výpočtu** v nejhorším případě vzhledem k velikosti vstupních parametrů.

**Délka výpočtu v nejhorším případě**

- Maximální počet redukčních kroků přes všechny možné výpočty aplikace programu na vstupní parametry stejné velikosti.

**Na délce záleží!**



# Funkce pro reverzi seznamu

## Reverze seznamu funkce `reverse'`

- ```
  reverse' :: [a] -> [a]
  reverse' [] = []
  reverse' (x:s) = reverse' s ++ [x]
  ```
- ```
  (++) :: [a] -> [a] -> [a]
  [] ++ t = t
  (x:s) ++ t = x : (s++t)
  ```

## Reverze seznamu funkce `reverse`

- ```
  reverse :: [a] -> [a]
  reverse = rev []
            where rev s [] = s
                  rev s (x:t) = rev (x:s) t
  ```
  ```

# Reverze seznamu funkcí `reverse'`

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:s) = reverse' s ++ [x]

(++) :: [a] -> [a] -> [a]
[] ++ t = t
(x:s) ++ t = x : (s++t)
```

```
    reverse' [1,2,3]
⇝   reverse' [2,3] ++ [1]
⇝   (reverse' [3] ++ [2]) ++ [1]
⇝   ((reverse' [] ++ [3]) ++ [2]) ++ [1]
⇝   (([] ++ [3]) ++ [2]) ++ [1]

⇝   ([3] ++ [2]) ++ [1]

⇝   (3 : ([] ++ [2])) ++ [1]
⇝   (3 : [2]) ++ [1]

⇝   3 : ([2] ++ [1])
⇝   3 : (2 : ([] ++ [1]))
⇝   3 : (2 : [1])   ≡   [3,2,1]
```

# Reverze seznamu funkcí `reverse'`

- Délka výpočtu funkce při aplikaci na seznam délky $n$.
- **n+1**

```
      reverse' [1,2,3]
~>    reverse' [2,3] ++ [1]
~>    (reverse' [3] ++ [2]) ++ [1]
~>    ((reverse' [] ++ [3]) ++ [2]) ++ [1]
~>    (([] ++ [3]) ++ [2]) ++ [1]

~>    ([3] ++ [2]) ++ [1]

~>    (3 : ([] ++ [2])) ++ [1]
~>    (3 : [2]) ++ [1]

~>    3 : ([2] ++ [1])
~>    3 : (2 : ([] ++ [1]))
~>    3 : (2 : [1])
```

# Reverze seznamu funkcí `reverse'`

- Délka výpočtu funkce při aplikaci na seznam délky $n$.
- n+1 + **1**

```
    reverse' [1,2,3]
⇝   reverse' [2,3] ++ [1]
⇝   (reverse' [3] ++ [2]) ++ [1]
⇝   ((reverse' [] ++ [3]) ++ [2]) ++ [1]
⇝   (([] ++ [3]) ++ [2]) ++ [1]

⇝   ([3] ++ [2]) ++ [1]

⇝   (3 : ([] ++ [2])) ++ [1]
⇝   (3 : [2]) ++ [1]

⇝   3 : ([2] ++ [1])
⇝   3 : (2 : ([] ++ [1]))
⇝   3 : (2 : [1])
```

# Reverze seznamu funkcí `reverse'`

- Délka výpočtu funkce při aplikaci na seznam délky $n$.
- n+1 + 1 + **2**

```
    reverse' [1,2,3]
⇝   reverse' [2,3] ++ [1]
⇝   (reverse' [3] ++ [2]) ++ [1]
⇝   ((reverse' [] ++ [3]) ++ [2]) ++ [1]
⇝   (([] ++ [3]) ++ [2]) ++ [1]

⇝   ([3] ++ [2]) ++ [1]

⇝   (3 : ([] ++ [2])) ++ [1]
⇝   (3 : [2]) ++ [1]

⇝   3 : ([2] ++ [1])
⇝   3 : (2 : ([] ++ [1]))
⇝   3 : (2 : [1])
```

# Reverze seznamu funkcí `reverse'`

- Délka výpočtu funkce při aplikaci na seznam délky $n$.
- n+1 + 1 + 2 **+ 3 + ... + n**

```
    reverse' [1,2,3]
⇝   reverse' [2,3] ++ [1]
⇝   (reverse' [3] ++ [2]) ++ [1]
⇝   ((reverse' [] ++ [3]) ++ [2]) ++ [1]
⇝   (([] ++ [3]) ++ [2]) ++ [1]

⇝   ([3] ++ [2]) ++ [1]

⇝   (3 : ([] ++ [2])) ++ [1]
⇝   (3 : [2]) ++ [1]

⇝   3 : ([2] ++ [1])
⇝   3 : (2 : ([] ++ [1]))
⇝   3 : (2 : [1])
```

# Reverze seznamu funkcí `reverse'`

- Délka výpočtu funkce při aplikaci na seznam délky $n$.
- n+1 + 1 + 2 + 3 + ... + n

```
    reverse' [1,2,3]
⇝   reverse' [2,3] ++ [1]
⇝   (reverse' [3] ++ [2]) ++ [1]
⇝   ((reverse' [] ++ [3]) ++ [2]) ++ [1]
⇝   (([] ++ [3]) ++ [2]) ++ [1]

⇝   ([3] ++ [2]) ++ [1]

⇝   (3 : ([] ++ [2])) ++ [1]
⇝   (3 : [2]) ++ [1]

⇝   3 : ([2] ++ [1])
⇝   3 : (2 : ([] ++ [1]))
⇝   3 : (2 : [1])
```

# Reverze seznamu funkcí `reverse`

```
reverse :: [a] -> [a]
reverse = rev []
          where rev s [] = s
                rev s (x:t) = rev (x:s) t
```

```
    reverse [1,2,3]
⇝   rev [] [1,2,3]
⇝   rev [1] [2,3]
⇝   rev [2,1] [3]
⇝   rev [3,2,1] []
⇝   [3,2,1]
```

# Reverze seznamu funkcí `reverse`

- Délka výpočtu funkce při aplikaci na seznam délky $n$.
- **1**

```
      reverse [1,2,3]
⇝     rev [] [1,2,3]
⇝     rev [1] [2,3]
⇝     rev [2,1] [3]
⇝     rev [3,2,1] []
⇝     [3,2,1]
```

# Reverze seznamu funkcí `reverse`

- Délka výpočtu funkce při aplikaci na seznam délky $n$.
- $1 + \mathbf{n}$

```
    reverse [1,2,3]
⇝   rev [] [1,2,3]
⇝   rev [1] [2,3]
⇝   rev [2,1] [3]
⇝   rev [3,2,1] []
⇝   [3,2,1]
```

# Reverze seznamu funkcí `reverse`

- Délka výpočtu funkce při aplikaci na seznam délky $n$.
- $1 + n + 1$

```
    reverse [1,2,3]
⇝   rev [] [1,2,3]
⇝   rev [1] [2,3]
⇝   rev [2,1] [3]
⇝   rev [3,2,1] []
⇝   [3,2,1]
```

# Reverze seznamu funkcí `reverse`

- Délka výpočtu funkce při aplikaci na seznam délky $n$.
- $1 + n + 1$

```
    reverse [1,2,3]
⇝   rev [] [1,2,3]
⇝   rev [1] [2,3]
⇝   rev [2,1] [3]
⇝   rev [3,2,1] []
⇝   [3,2,1]
```

# Asymptotický růst funkcí

**Pozorování**

- Při určování časové složitosti algoritmů je nepraktické a často i obtížné určovat tuto složitost přesně.
- Funkce vyjadřující délku výpočtu vzhledem k velikosti parametru klasifikujeme podle **asymptotického chování**.

**Asymptotický růst funkcí**

- Při zápisu funkční hodnoty v proměnné $n$ **rozhoduje nejrychleji rostoucí člen**. U něj navíc zanedbáváme kladnou multiplikativní konstantu.
- Podle toho hovoříme o funkcích lineárních, kvadratických, exponenciálních apod.

# Přehled asymptotických funkcí

| **t(n)** | **růst funkce t** |
|---|---|
| $1, \quad 20, \quad 729, \quad 2^{64}$ | konstantní |
| $2 \log n + 5, \quad 3 \log_2 n + \log_2(\log_2 n)$ | logaritmický |
| $n, \quad 2n + 1, \quad n + \sqrt{n}$ | *lineární* |
| $n \log n, \quad 3n \log n + 6n + 9$ | *n log n* |
| $n^2, \quad 3n^2 + 4n - 1, \quad n^2 + 10 \log n$ | *kvadratický* |
| $n^3, \quad n^3 + 3n^2$ | *kubický* |
| $2^n$ | |
| $\left(\frac{1+\sqrt{5}}{2}\right)^n$ | exponenciální |
| $3^n$ | |

# Asymptotická složitost `reverse` a `reverse'`

**reverse'**

- Počet redukčních kroků výrazu `reverse'` $[x_1,\ldots,x_n]$ na každém seznamu délky $n$ je

$$n + 1 + 1 + 2 + 3 + \cdots + n = \frac{n^2 + 3n + 2}{2}$$

Složitost funkce `reverse'` je **kvadratická** vzhledem k délce obraceného seznamu.

**reverse**

- Počet redukčních kroků výrazu `reverse` $[x_1,\ldots,x_n]$ na každém seznamu délky $n$ je

$$1 + n + 1 = n + 2$$

Složitost funkce `reverse` je **lineární** vzhledem k délce obraceného seznamu.

# Časová složitost algoritmu a problému

**Časová složitost algoritmu**

- Posuzuje konkrétní algoritmus.
- Nevypovídá a jiných algoritmech pro řešení téhož problému.

**Časová složitost problému**

- Daný problém je možné řešit různými algoritmy.
- Složitost problému vypovídá a časové složitosti nejlepšího možného algoritmu pro řešení problému.
- Určovat složitost problému je výrazně obtížnější, než určování složitosti algoritmu.
- Bez znalosti složitosti problému nelze určit, zda daný algoritmus pro řešení problému je optimální.

**Pozor**

- Časová složitost popisuje délku výpočtu **v nejhorším případě** pro danou velikost argumentu.

**Příklad**

- Vyšetřujeme časovou složitost funkce `ins` vzhledem k jejímu druhému parametru.
- Funkce `ins` zařazuje prvek do seřazeného seznamu.

```
ins :: Int -> [Int] -> [Int]
ins x [] = [x]
ins x (y:t) = if x <= y then x : y : t else y : ins x t
```

## Různé délky výpočtu

- Počet kroků při volání `ins x [x1, ..., xn]` je různý.
- Nejkratší výpočet má délku 3:

  `ins 1 [2,4,6,8]` $\rightsquigarrow^{3}$ `[1,2,4,6,8]`
- Nejdelší výpočet má délku $3n + 1$:

  `ins 9 [2,4,6,8]` $\rightsquigarrow^{3*4+1}$ `[2,4,6,8,9]`

## Časová složitost

- Časová složitost funkce `ins` je **lineární** vzhledem k velikosti jejího druhého argumentu (tj. vzhledem k délce seznamu).

# Časová složitost a redukční strategie

## Pozorování

- Časová složitost závisí nejen na algoritmu (způsobu definování funkce), ale také na redukční strategii.

## Příklad

- Uvažme funkcí pro uspořádání prvků v seznamu pomocí postupného zařazování.

```
inssort :: Ord a => [a] -> [a]
inssort = foldr ins []
          where ins x [] = [x]
                ins x (y:t) = if x <= y then x : y : t
                                        else y : ins x t
```

- Princip řazení funkcí `inssort`

$$
\begin{aligned}
& \texttt{inssort [}x_1\texttt{, }x_2\texttt{, ..., }x_{n-1}\texttt{, }x_n\texttt{]} \\
\rightsquigarrow \quad & \texttt{foldr ins [] [}x_1\texttt{, }x_2\texttt{, ..., }x_{n-1}\texttt{, }x_n\texttt{]} \\
\rightsquigarrow^{n+1} \quad & \texttt{ins }x_1\texttt{ (ins }x_2\texttt{ (...(ins }x_{n-1}\texttt{ (ins }x_n\texttt{ [])...))}
\end{aligned}
$$

# Příklad závislosti časové složitosti na redukční strategii

## Definice funkce

- ```
  inssort :: Ord a => [a] -> [a]
  inssort = foldr ins []
            where ins x [] = [x]
                  ins x (y:t) = if x <= y then x : y : t
                                          else y : ins x t
  ```
- `minim = head . inssort`

## Striktní vyhodnocování (nejhorší případ – seznam je klesající)

$$
\begin{array}{ll}
 & \texttt{minim [}x_1\texttt{, ..., }x_n\texttt{]} \\
\rightsquigarrow & \texttt{(head.inssort) [}x_1\texttt{,...,}x_n\texttt{]} \\
\rightsquigarrow & \texttt{head (inssort [}x_1\texttt{,...,}x_n\texttt{])} \\
\rightsquigarrow & \texttt{head (foldr ins [] [}x_1\texttt{,...,}x_n\texttt{])} \\
\rightsquigarrow^{n+1} & \texttt{head (ins }x_1\texttt{ (...(ins }x_{n-2}\texttt{ (ins }x_{n-1}\texttt{ (ins }x_n\texttt{ [])))...))} \\
\rightsquigarrow^{3\cdot 0+1} & \texttt{head (ins }x_1\texttt{ (...(ins }x_{n-2}\texttt{ (ins }x_{n-1}\texttt{ [}x_n\texttt{]))...))} \\
\rightsquigarrow^{3\cdot 1+1} & \texttt{head (ins }x_1\texttt{ (...(ins }x_{n-2}\texttt{ [}x_n\texttt{,}x_{n-1}\texttt{])...) )} \\
\rightsquigarrow^{3\cdot 2+1} & \texttt{head (ins }x_1\texttt{ (...[}x_n\texttt{,}x_{n-1}\texttt{,}x_{n-2}\texttt{]...) )} \\
\vdots & \\
\rightsquigarrow^{3\cdot (n-2)+1} & \texttt{head (ins }x_1\texttt{ [}x_n\texttt{,...,}x_2\texttt{] )} \\
\rightsquigarrow^{3\cdot (n-1)+1} & \texttt{head [}x_n\texttt{,...,}x_1\texttt{]} \\
\rightsquigarrow & x_n
\end{array}
$$
  ```

# Příklad závislosti časové složitosti na redukční strategii

## Definice funkce

- ```
  inssort :: Ord a => [a] -> [a]
  inssort = foldr ins []
            where ins x [] = [x]
                  ins x (y:t) = if x <= y then x : y : t
                                          else y : ins x t
  ```
- `minim = head . inssort`

## Líné vyhodnocování (nejhorší případ – nejmenší prvek na konci)

$$
\begin{array}{ll}
 & \texttt{minim [}x_1\texttt{,...,}x_n\texttt{]} \\
\rightsquigarrow & \texttt{(head.inssort) [}x_1\texttt{,...,}x_n\texttt{]} \\
\rightsquigarrow & \texttt{head (inssort [}x_1\texttt{,...,}x_n\texttt{])} \\
\rightsquigarrow & \texttt{head (foldr ins [] [}x_1\texttt{,...,}x_n\texttt{])} \\
\rightsquigarrow^{n+1} & \texttt{head (ins }x_1\texttt{ (...(ins }x_{n-2}\texttt{ (ins }x_{n-1}\texttt{ (ins }x_n\texttt{ [])))...) )} \\
\rightsquigarrow & \texttt{head (ins }x_1\texttt{ (...(ins }x_{n-2}\texttt{ (ins }x_{n-1}\texttt{ (}x_n\texttt{ : [])))...) )} \\
\rightsquigarrow^{3} & \texttt{head (ins }x_1\texttt{ (...(ins }x_{n-2}\texttt{ (}x_n\texttt{ :(ins }x_{n-1}\texttt{ [])))...) )} \\
\rightsquigarrow^{3} & \texttt{head (ins }x_1\texttt{ (...(}x_n\texttt{ : (ins }x_{n-2}\texttt{ (ins }x_{n-1}\texttt{ [])))...) )} \\
\vdots & \\
\rightsquigarrow^{3} & \texttt{head (ins }x_1\texttt{ (}x_n\texttt{ :(ins }x_2\texttt{ (ins }x_3\texttt{ (...(ins }x_{n-1}\texttt{ []))...)))))} \\
\rightsquigarrow^{3} & \texttt{head (}x_n\texttt{ :(ins }x_1\texttt{ (ins }x_2\texttt{ (...(ins }x_{n-1}\texttt{ []))...))))} \\
\rightsquigarrow & x_n
\end{array}
$$
  ```

# Příklad závislosti časové složitosti na redukční strategii

**Striktní vyhodnocování**

- Počet redukčních kroků výrazu `minim [x`$_1$`,...,x`$_n$`]` je:

$$3 + n + 1 + \sum_{k=0}^{n-1}(3k + 1) + 1 = \frac{3n^2 + n + 10}{2}$$

- Při striktním vyhodnocování má funkce **kvadratickou** časovou složitost.

**Líné vyhodnocování**

- Počet redukčních kroků výrazu `minim [x`$_1$`,...,x`$_n$`]` je:

$$3 + n + 1 + 1 + 3.(n - 1) + 1 = 4n + 3$$

- Při líném vyhodnocování má funkce **lineární** časovou složitost.

# Vliv redukční strategie na časovou složitost

**Pozorování**

- Není pravda, že časová složitost výpočtu se při líném a striktním vyhodnocování vždy liší.
- Pokud se časová složitost liší, může se lišit víc než o jeden řádek ve zmiňované tabulce asymptotických růstů funkcí.

**Příklady**

- Konstantní (líně) versus exponenciální (striktně):

```
f n = const n (fib' n)
```

- Lineární líně i striktně:

`length` $[a_1, \ldots, a_n]$

# Akumulátor


TAB
traction flooded
Tubular
Motion
12V 95T, 130Ah (C120), 115Ah (C20), 104Ah (C10), 95Ah (C5)

# Akumulátor a rekurze

**Obecný princip**

- Vícenásobné nebo nevhodné použití rekurzivního volání v těle rekurzivně definované funkce, může v obecné rovině vést na časově neoptimální algoritmus.
- Opakovanému rekurzivnímu volání pro tutéž hodnotu lze zabránit uchováváním mezivýsledků rekurzivního volání.
- Uchování výsledků se provádí přidáním parametru rekurzivní funkce, tzv. **akumulátoru**.

**Pozorování**

- Přímé použití rekurzivní funkce má tendenci být čitelnější.

# Výpočet Fibonacciho číselné řady

## Definice funkcí

```
fib' :: Integer -> Integer
fib' 0 = 0
fib' 1 = 1
fib' n = fib' (n-2) + fib' (n-1)

fib :: Integer -> Integer
fib = f 0 1
    where f a _ 0 = a
          f a b k = f b (a+b) (k-1)
```

## Složitost vzhledem k argumentu

- Složitost funkce `fib'` je **exponenciální**.
- Složitost funkce `fib` je **lineární**.

# Checkpoint

**Analyzujte časovou složitost**

- funkce `map` , vzhledem k délce seznamu.
- funkce `es` (Eratosthenovo síto) vzhledem k délce seznamu.

**Lineární vs. kvadratická**

- Vytvořte libovolnou funkci `f1::a->[a]` , která má kvadratickou složitost.
- Vytvořte libovolnou funkci `f2::a->[a]` , která má lineární složitost.
- Použijte obě funkce v kontextu funkce `map` na dlouhých seznamech a pozorujte rozdíl.