Úvod do funkcionálního programování

IB015

Literatura

- S. Thompson: *Haskell – The Craft of Functional Programming*, Addison-Wesley, 1997
- R. Bird: *Introduction to Functional Programming using Haskell*, Prentice Hall, 1998

Všeobecné informace o kursu

(kredity, podmínky zápisu, rozvrh, přihlašování ke zkouškám . . .)

`https://is.muni.cz/auth/`

Aktuální informace

(konsultace, bodování zkoušek . . .)

`http://www.fi.muni.cz/~libor/vyuka/IB015/`

Funkcionální programování

Styl (přístup, paradigma) tvorby programů, jehož hlavní metodou je práce s *funkcemi* a jejich *aplikace na argumenty*.

Užitím funkcionálního paradigmatu dosahujeme *jednoduchosti*, *čitelnosti* a *elegance* programů.

Paradigmata

($\pi\alpha\rho\acute{\alpha}\delta\varepsilon\iota\gamma\mu\alpha$ — vzor, příklad, přístup, pojetí, metoda, způsob uvažování)

Imperativní

program	· · · · · ·	příkaz
výpočet	· · · · · ·	posloupnost akcí (stav $\mapsto$ stav)
výsledek	· · · · · ·	účinek těchto akcí
jazyky	· · · · · ·	C, C++, Java, Pascal, Perl, php, . . .

Funkcionální

program	· · · · · ·	výraz
výpočet	· · · · · ·	úprava (zjednodušování) výrazu
výsledek	· · · · · ·	hodnota (nezjednodušitelný tvar výrazu)
jazyky	· · · · · ·	Haskell, ML, CaML, Erlang, Lisp, Scheme, . . .

Logické

program	· · · · · ·	teorie a formule
výpočet	· · · · · ·	hledání důkazu
výsledek	· · · · · ·	množina splňujících substitucí
jazyky	· · · · · ·	Prolog, Gödel, (Mercury), . . .

a další . . .

Funkcionální paradigma

Neformální vymezení: Funkcionální programování je programování

- bez příkazů
- bez přepisovatelných proměnných
- beze stavů

Funkcionální programovací paradigma patří mezi tzv. *deklarativní* paradigmata:

důraz je na tom, *co* program počítá, méně na tom, *jak* to počítá.

Funkcionální programy

Stavebními prvky funkcionálních programů jsou *výrazy*.

Výrazy obsahují konstanty, proměnné, funkce a lokální definice.

Funkcionální program je tvořen jediným výrazem. Funkce a konstanty v něm použité lze ve výrazu definovat; definice na nejvyšší úrovni (nevnořené do jiných definic) nazýváme *globální*.

Příklady funkcionálních programů

Jednoduché výrazy

```
1 + 2
```

```
(1 + 2)  *  (1 + 2)
```

Výraz s lokální definicí

```
let  square x = x * x
  in  square (1 + 2)
```

Výraz s lokální definicí vnořený ve výrazu s lokální definicí

```
let  square x = x * x
  in  let  tri = 1 + 2  in  square tri
```

Příklady funkcionálních programů

```
1 * 1 * 2 * 3
```

```
let  fact n = product [1..n]
  in  fact 3
```

```
let  fact 0 = 1
     fact n = n * fact (n-1)
  in  let tri = 3 in fact tri
```

```
fact 0 = 1
fact n = n * fact (n-1)
........................
let tri = 3 in fact tri
```

Rysy současných funkcionálních jazyků

- deklarativnost – jazyky vyšší úrovně, zápis programu je bližší specifikaci
- ortogonalita – se všemi hodnotami se pracuje stejným způsobem; z ortogonality vyplývá jednoduchost syntaxe i sémantiky, a také vysoká expresivita
- bohatý typový systém a silná typová kontrola – odtud vyplývá rychlá detekce chyb a snadné ladění
- referenční transparentnost a z ní vyplývající možnost formální manipulace s programy (optimalizační transformace, dokazování správnosti, částečné vyhodnocování, paralelizace výpočtů)

Nevýhody

- komplikovanější vstup/výstup
- interakce s uživatelem, okolím, operačním systémem

Rekursivní definice

```
fact    ::  Integer -> Integer
fact 0   =  1
fact n   =  n * fact (n-1)
```

Věta: Pro každé přirozené číslo n platí `fact` $n \leadsto^* n!$.

Důkaz Indukcí podle n.

(i) Je-li $n = 0$, pak `fact` $n =$ `fact` $0 \leadsto 1 = 0!$.

(ii) Nechť $n > 0$ a nechť *věta platí pro všechna* k*,* $0 \leq k < n$ (indukční předpoklad).

Pak `fact` $n \leadsto n \cdot$ `fact` $(n-1) \leadsto^* n \cdot (n-1)! = n!$. Tedy věta platí i pro číslo n.

Podle principu indukce věta platí pro všechna přirozená čísla.

```
fact 0  =  1
fact n  =  n * fact (n-1)
```

$$
\begin{array}{lll}
\texttt{fact 4} & \leadsto & \texttt{4 * fact (4-1)} \\
 & \leadsto & \texttt{4 * fact 3} \\
 & \leadsto & \texttt{4 * (3 * fact (3-1))} \\
 & \leadsto & \texttt{4 * (3 * fact 2)} \\
 & \leadsto & \texttt{4 * (3 * (2 * fact (2-1)))} \\
 & \leadsto & \texttt{4 * (3 * (2 * fact 1))} \\
 & \leadsto & \texttt{4 * (3 * (2 * (1 * fact (1-1))))} \\
 & \leadsto & \texttt{4 * (3 * (2 * (1 * fact 0)))} \\
 & \leadsto & \texttt{4 * (3 * (2 * (1 * 1)))} \\
 & \leadsto & \texttt{4 * (3 * (2 * 1))} \\
 & \leadsto & \texttt{4 * (3 * 2)} \\
 & \leadsto & \texttt{4 * 6} \\
 & \leadsto & \texttt{24}
\end{array}
$$

```
(^)    ::  Integer -> Integer -> Integer
x ^ 0   =  1
x ^ n   =  x * (x^(n-1))
```

Věta: Pro každá dvě přirozená čísla x, y, $x > 0$, platí $x \verb|^| y \rightsquigarrow^* x^y$.

Typy

Každá hodnota, a tedy i každý výraz, má svůj *typ*.

Základní (datové) typy: `Bool`, `Char`, `Int`, `Integer`, `Float`, ...

Složené (datové) typy: `(Char,Int)`, `(Float,Float,Bool)`, `[Integer]`, `[([Char],Int)]`, ...

Funkcionální typy: `Char`→`Char`, `Int`→`Bool`, `Float`→`Float`→`Bool`, ...
Obecně, jsou-li σ, τ dva typy, pak $\sigma \rightarrow \tau$ je typ všech funkcí s parametrem typu σ a funkční hodnotou typu τ.
Jsou-li $\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_n, \tau$ typy, pak $\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$ je typ n-árních funkcí s prvním parametrem typu σ_1, druhým parametrem typu σ_2, třetím parametrem typu σ_3, ..., n-tým parametrem typu σ_n a funkční hodnotou (výsledkem) typu τ.

Typová anotace

výraz :: *typ*

Například

```
'#' :: Char
pi  :: Float
not :: Bool -> Bool
odd :: Int -> Bool
```

Typové anotace se používají především na začátku definic:

```
objemkvadru :: Float -> Float -> Float -> Float
objemkvadru a b c  =  a * b * c
```

Ale může být i ve výrazech:

```
"Hodnota n je "  ++  show (n::Int)  ++  ".\n"
```

Další příklady:

Příklady:

```
False            ::  Bool
not              ::  Bool -> Bool
(&&)             ::  Bool -> Bool -> Bool
'a'              ::  Char
['a','b','c']    ::  [Char]
[True,True]      ::  [Bool]
length [1,2,1]   ::  Int
length           ::  [a] -> Int
[]               ::  [a]
```

Polymorfní typy

Obsahuje-li zápis typu *typové proměnné*, říkáme, že typ je *polymorfní*. Polymorfní typ se může v různých kontextech upřesnit (*specializovat*) na různé typy, které vzniknou dosazením konkrétních typů za typové proměnné.

```
id   ::  a -> a
id x  =  x
```

```
id 3      ≡   (id::Int->Int) 3
id 'A'    ≡   (id::Char->Char) 'A'
id not    ≡   (id::(Bool->Bool)->(Bool->Bool)) not
```

Poznámka: Tomuto polymorfismu se přesněji říká *parametrický polymorfismus*, protože typové proměnné, za něž se dosazuje, hrají roli parametrů.

```
id   ::  a -> a
id x  =  x

const     ::  a -> b -> a
const x y  =  x

flip       ::  (a -> b -> c) -> b -> a -> c
flip f x y  =  f y x

(.)       ::  (b -> c) -> (a -> b) -> a -> c
(f . g) x  =  f (g x)

fst       ::  (a,b) -> a
fst (x,y)  =  x

snd       ::  (a,b) -> b
snd (x,y)  =  y
```

Operátor (.) pro skládání funkcí

Polymorfní operátor `(.) :: (b -> c) -> (a -> b) -> (a -> c)`

patří k nejdůležitějším funkcím ve funkcionálním programování.

```
(f . g) x   =   f (g x)
```

`(abs . sin) (3*pi/2)` $\leadsto^*$ `1.0`

`(fact . fact . abs) (-3)` $\leadsto^*$ `720`

`(toUpper . toLower) 'A'` $\leadsto^*$ `'A'`

Věta: Operace (.) je asociativní.

Základní datové struktury

Uspořádané n-tice

Mají pevný počet složek, daný typem. Složky mohou být různých typů.

Příklady:

```
(False,2)                      ::  (Bool,Int)
(odd, '*', (2.71828,not))      ::  (Int->Bool, Char, (Float,Bool->Bool))
()                             ::  ()
```

Seznamy

Jsou posloupnosti hodnot. Všechny prvky musí být stejného typu. Typ seznamu neurčuje délku, tj. seznamy stejného typu mohou mít různou délku.

Příklady:

```
[1, 2, 3, 4]                    ::  [ Int ]
[False, True, False||True ]     ::  [ Bool ]
[ not . odd,  not . even ]      ::  [ Int->Bool ]
[ [not], [] ]                   ::  [[ Bool->Bool ]]
```

Uspořádané n-tice

Uspořádané dvojice: je-li x hodnota typu A, y je hodnota typu B, pak `(`x`,`y`)` je hodnota typu „kartézský součin typů A a B“, zapisujeme `(`x`,`y`) :: (`A`,`B`)`.

Příklady:

```
(2,7)                 ::   (Int,Int)
(not True, 3.14159)   ::   (Bool, Float)
(not, ('@',42))       ::   (Bool->Bool, (Char,Int))
```

Uspořádané trojice: je-li x`::`A, y`::`B, z`::`C, pak `(`x`,`y`,`z`)` je uspořádaná trojice, tj. hodnota typu „kartézský součin typů A, B, C“. Typ se opět zapisuje podobným způsobem jako samy trojice: `(`x`,`y`,`z`) :: (`A`,`B`,`C`)`

Uspořádaná nultice `()` je (jedinou) hodnotou „kartézského součinu nulového počtu typů“ (nulté kartézské mocniny), zapisovaného rovněž `()`.

```
() :: ()
```

„Uspořádané jednice“ se nepoužívají.

Projekční funkce na uspořádaných dvojicích

```
fst         ::  (a,b) -> a
fst (x,_)    =  x

snd         ::  (a,b) -> b
snd (_,y)    =  y
```

Konstruktor (,) uspořádaných dvojic

```
(,)      ::  a -> b -> (a,b)
(,) x y   =  (x,y)
```

Seznamy

Seznam je posloupnost hodnot *stejného typu*.

```
[1],  [2,3,5],  [-1,0,1]      ::  [Int]
['a','b']                     ::  [Char]
[(odd,not), (even,not.not)]   ::  [(Int->Bool,Bool->Bool)]
```

Konstruktory seznamů:

```
[]    ::  [a]                  --  prázdný seznam (nulární datový konstruktor)
(:)   ::  a -> [a] -> [a]      --  přidání prvku na začátek seznamu (binární datový konstruktor)
```

Každý seznam vznikne (opakovanými) aplikacemi konstruktoru `(:)` na prvky a seznamy, poslední seznam v aplikaci je `[]`.

```
[]
[1,2,3,1] ≡ 1 : 2 : 3 : 1 : []
[True]    ≡ True : []
['a','b','c'] ≡ 'a' : 'b' : 'c' : [] ≡ "abc"
```

Seznamům typu `[Char]` se říká *řetězce*. Řetězce mají navíc zvláštní syntax.

Pro typ `[Char]` je v Haskellu synonymum `String`

```
type String = [Char]
```

Prázdný řetězec: `([]::String)` $\equiv$ `""`

Funkce `head` a `tail`

Vracejí *začátek* a *zbytek* neprázdného seznamu.

```
head          ::   [a] -> a
head (x:_)     =   x

tail          ::   [a] -> [a]
tail (_:s)     =   s
```

`[1,2,3]` $\equiv$ `1:2:3:[]`

head tail

1 :

2 :

3 []

Věta: Pro každý neprázdný seznam s platí

$$\texttt{head}\ s \ \ \texttt{:}\ \ \texttt{tail}\ s \ = \ s$$

Operace nad seznamy

```
head         ::   [a] -> a                -- první prvek seznamu
head (x:_)    =   x

tail         ::   [a] -> [a]              -- zbytek seznamu (bez prvního prvku)
tail (_:s)    =   s

null         ::   [a] -> Bool             -- predikát prázdnosti
null []       =   True
null (_:_)    =   False

length         ::   [a] -> Int            -- délka seznamu
length []       =   0
length (_:s)    =   1 + length s
```

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

(!!)        ::  [a] -> Int -> a       -- [x0,x1,...,xn-1]!!k = xk
(x:_) !! 0   =  x
(_:s) !! k   = s !! (k-1)

take            ::  Int -> [a] -> [a]  -- prvních n prvků seznamu
take 0 _         =  []
take _ []        =  []
take n (x:s)     =  if  n>0  then  x : take (n-1) s
                             else  error "take: záporný argument"

drop            ::  Int -> [a] -> [a]  -- seznam bez prvních n prvků
drop 0 s         =  s
drop _ []        =  []
drop n (_:s)     =  if  n>0  then  drop (n-1) s
                             else  error "drop: záporný argument"
```

Comment of (!!) in math form: $[x_0, x_1, \ldots, x_{n-1}]!!k = x_k$

Funkce `concat` spojí seznam seznamů do jediného seznamu.

```
concat         ::  [[a]] -> [a]        -- spojení seznamů tvořících seznam
concat []       =  []
concat (s:t)    =  s ++ concat t
```

Funkce `map` je důležitá funkce, pomocí níž aplikujeme unární funkci na každý prvek seznamu.

```
map         ::  (a->b) -> [a] -> [b]  -- aplikace funkce na každý prvek
map _ []       =  []
map f (x:s)    =  f x : map f s
```

Příklady:

```
map not [True,False]   ⇝*   [False,True]
map (<4) [3,4,5]   ⇝*   [True,False,False]
map (^3) [1,2,3,4]   ⇝*   [1,8,27,64]
map toUpper "toUpper"   ⇝*   "TOUPPER"
map (`mod`3) [0..5]   ⇝*   [0,1,2,0,1,2]
map (:[]) "abc"   ⇝*   ["a","b","c"]
```

Poznámka: Funkce `map` aplikuje funkci na prvky seznamu. Je speciálním případem funkce (v Haskellu nazvané `fmap`), která aplikuje funkci na prvky libovolné datové struktury. Umožňuje-li to datová struktura, lze `fmap` implementovat paralelně.

```
filter  :: (a->Bool) -> [a] -> [a]   -- výběr prvků splňujících podmínku
filter _ []      =  []
filter p (x:s)   =  if  p x  then  x : t  else  t
                    where t = filter p s
```

Příklady:

```
filter (>3) [1,5,8,2,6,3]    ⇝*    [5,8,6]
filter (=='e') "retezec"    ⇝*    "eee"
filter not [True,False]    ⇝*    [False]
filter isDigit "cd2bcg16"    ⇝*    "216"
```

Poznámka: Funkci `filter` lze ekvivalentně definovat

```
filter  :: (a->Bool) -> [a] -> [a]   -- výběr prvků splňujících podmínku
filter _ []      =  []
filter p (x:s)   =  ( if p x then (x:) else id ) (filter p s)
```

Vlastnosti operací nad seznamy

Věta: Jsou-li s, t dva konečné seznamy stejného typu, pak

$$\texttt{length}\,(s +\!\!+\, t) = \texttt{length}\, s\ +\ \texttt{length}\, t$$

Důkaz Indukcí podle délky seznamu s.

Věta: Operace `(++)` spojování seznamů je asociativní.

Důkaz je ekvivalentní s tvrzením, že pro každé tři seznamy s, t, u platí rovnost $(s +\!\!+\, t) +\!\!+\, u = s +\!\!+ (t +\!\!+\, u)$.

Dokazujeme indukcí podle délky seznamu s.

Věta: Pro každý seznam s a celé číslo $m \geq 0$ platí

$$\texttt{take}\ m\ \ s\ +\!\!+\ \texttt{drop}\ m\ \ s = s$$

Důkaz Indukcí podle m.

Operace `takeWhile`, `dropWhile`, `replicate`

```
takeWhile          ::  (a -> Bool) -> [a] -> [a]
takeWhile _ []      =  []
takeWhile p (x:s)   =  if  p x  then  x : takeWhile p s  else  []
```

Příklad: `takeWhile odd [1,3,1,2,3,5,7]` $\leadsto^*$ `[1,3,1]`

`takeWhile isAlpha "ab4cd16"` $\leadsto^*$ `"ab"`

```
dropWhile             ::  (a -> Bool) -> [a] -> [a]
dropWhile _ []         =  []
dropWhile p s@(x:s')   =  if  p x  then  dropWhile p s'  else  s
```

Příklad: `dropWhile odd [1,3,1,2,3,5,7]` $\leadsto^*$ `[2,3,5,7]`

`dropWhile isDigit "ab45"` $\leadsto^*$ `"ab45"`

```
replicate     ::  Int -> a -> [a]
replicate 0 _  =  []
replicate n x  =  x : replicate (n-1) x
```

Příklad: `replicate 4 '='` $\leadsto^*$ `"===="`

Operace zip, unzip

```
zip                 ::  [a] -> [b] -> [(a,b)]
zip []      _        =  []
zip _       []       =  []
zip (x:s) (y:t)      =  (x,y) : zip s t
```

Příklad: `zip [3,5] "abc"` $\leadsto^*$ `[(3,'a'),(5,'b')]`

```
unzip               ::  [(a,b)] -> ([a],[b])
unzip []             =  ([],[])
unzip ((x,y):s)      =  (x : fst u, y : snd u)
                        where u = unzip s
```

Příklad: `unzip [('a',3),('b',5)]` $\leadsto^*$ `("ab",[3,5])`

Operace `zipWith`

```
zipWith                :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _  _     []     = []
zipWith _  []    _      = []
zipWith op (x:s) (y:t)  = x `op` y  :  zipWith op s t
```

Příklady:

zipWith (*) [2,6,14] [21,7,3] $\leadsto^*$ [42,42,42]

zipWith (+) [3,4,8,2] [1,4,1,7] $\leadsto^*$ [4,8,9,9]

zipWith (^) [4,3,2,1] [1,2,3,4] $\leadsto^*$ [4,9,8,1]

zipWith (++) ["a","bc","de"] ["gh","","k","xyz"] $\leadsto^*$ ["agh","bc","dek"]

zipWith (:) [True,False] [[],[False]] $\leadsto^*$ [[True],[False,False]]

zipWith map [toLower,toUpper] ["abCD","UVwx"] $\leadsto^*$ ["abcd","UVWX"]

Funkci `zip` lze definovat stručně a elegantně pomocí funkce `zipWith`:

```
zip  =  zipWith (,)
```

Poznámka: Zatímco funkce `map` aplikuje unární funkci na hodnoty v jednom seznamu, funkce `zipWith` aplikuje binární funkci na hodnoty ve dvou seznamech.

Modul `Monad` tyto funkce zobecňuje a definuje `liftM1` ($\approx$ `map`), `liftM2` ($\approx$ `zipWith`), až `liftM5`.

Hodnoty a typy

Hodnoty se zapisují pomocí hodnotových výrazů.

Hodnotové výrazy obsahují proměnné, datové konstruktory a ostatní hodnoty.

Z nich se vytvářejí výrazy pomocí aplikací, abstrakcí, lokálních definic atd.

Výrazy skládající se jen z datových konstruktorů nelze zjednodušovat. Ostatní výrazy lze zjednodušovat (redukovat), lze s nimi provádět výpočty.

Datové konstruktory:

nulární `True`, `()`, `Nothing`, `[]`, `0`, `'a'`

unární `Just`

binární `(:)`, `(,)`

ternární `(,,)`

Typy se zapisují pomocí typových výrazů.

Typové výrazy obsahují typové proměnné a typové konstruktory.

Z těch se vytvářejí typové výrazy pomocí aplikace typových konstruktorů na typové podvýrazy.

Typové výrazy nelze redukovat – jsou statické.

Typové konstruktory:

nulární `Bool`, `Int`, `Char`, `()`

unární `[]`, `Maybe`, `IO`

binární `(->)`, `(,)`

ternární `(,,)`

```
data Bool = False | True

data Den = Po | Ut | St | Ct | Pa | So | Ne

data Barva = Červená | Zelená | Modrá | Šedá Int

data Barva' = RGB Int Int Int

data Maybe a = Nothing | Just a

data Either a b = Left a | Right b

data Nat = Zero | Succ Nat

data List a = Nil | Cons a (List a)

data Tree a = Empty | Node a (Tree a) (Tree a)
```

Definice nových typových a datových konstruktorů

$$\texttt{data}\quad \textit{Typcons}\ a_1\ \ldots\ a_n \quad = \quad \textit{Dcons}_1\ \textit{typ}_{1,1}\ \ldots\ \textit{typ}_{1,r_1}$$

$$|\quad \ldots$$

$$|\quad \textit{Dcons}_m\ \textit{typ}_{m,1}\ \ldots\ \textit{typ}_{m,r_m}$$

Takto je definován nový n-ární typový konstruktor *Typcons* a m datových konstruktorů *Dcons*$_1, \ldots,$ *Dcons*$_m$ s aritami $r_1, \ldots, r_m$. Jejich typy jsou

$$\textit{Dcons}_i \quad :: \quad \textit{typ}_{i,1} \texttt{ -> } \textit{typ}_{i,2} \texttt{ -> } \cdots \texttt{ -> } \textit{typ}_{i,r_i} \texttt{ -> } \textit{Typcons}\ a_1\ \ldots\ a_n$$

V typových výrazech *typ*$_{i,j}$ se mohou vyskytovat typové proměnné $a_1, \ldots, a_n$.

Pojmenování existujícího typu

$$\texttt{type}\quad \textit{Typcons}\ a_1\ \ldots\ a_n \quad = \quad \textit{typ}$$

```
type  String  =  [Char]
```

```
type  Matrix a  =  [[a]]
```

Typový konstruktor Maybe

```
data  Maybe a  =  Nothing  |  Just a
```

Rozšiřuje typ o hodnotu `Nothing`, která zpravidla označuje nedefinovaný výsledek.

Příklad: Zjištění indexu prvku v seznamu:

```
ind          ::  Eq a  =>  a -> [a] -> Maybe Int
ind _ []      =  Nothing
ind x (y:s)   =  if  x == y  then  Just 0
                             else  if  ixs == Nothing  then  Nothing
                                                       else  Just (n+1)
                 where  ixs    = ind x s
                        Just n = ixs
```

`ind 'e' "Jonatán"` $\leadsto^*$ `Nothing`

`ind 'c' "Micinka"` $\leadsto^*$ `Just 2`

Příklad: Vyhledávání v tzv. asociačním seznamu:

```
lookup              ::  Eq a  =>  a -> [(a,b)] -> Maybe b
lookup _ []          =  Nothing
lookup k ((x,y):s)   =  if  k == x  then  Just y  else  lookup k s
```

```
znamky = [("mach",1), ("sebestova",1), ("horacek",5), ("pazout",5)]
```

`lookup "sebestova" znamky` $\leadsto^*$ `Just 1`

`lookup "jonatan" znamky` $\leadsto^*$ `Nothing`

Funkce vyšších řádů

V disciplínách, v nichž se pracuje jen s prvořádovými funkcemi, bývají funkce více proměnných *zobrazeními kartézského součinu*.

```
mul'    ::  (Int,Int) -> Int
mul' (x,y)  =  x * y
```

`mul' (3,7)` $\leadsto^* 21$

Tam, kde se pracuje s funkcemi vyšších řádů, mohou být funkce více proměnných *zobrazeními do množiny funkcí* (o jedničku nižší arity).

```
mul     ::  Int -> (Int -> Int)
mul x y  =  x * y
```

`(mul 3) 7` $\leadsto^* 21$

Aplikace funkce `mul` na jeden argument se nazývá *částečná*. Například částečná aplikace `mul 3` má hodnotu funkce, která ztrojnásobuje svůj argument.

`(+)`	`:: (Int -> (Int -> Int))`	Funkce, jejíž argument je číslo a výsledek je *funkce, jejíž argument je číslo a výsledek číslo*
`((+) 2)`	`:: (Int -> Int)`	funkce, jejíž argument je číslo a výsledek je číslo
`(((+) 2) 3)`	`:: Int`	číslo

Typový konstruktor `(->)` se zapisuje infixově mezi typy a implicitně sdružuje zprava.

$$f \quad :: \quad a_1 \texttt{ -> } (a_2 \texttt{ -> } (a_3 \texttt{ -> } \cdots \texttt{ -> } (a_{n-1} \texttt{ -> } (a_n \texttt{ -> } a))\ldots))$$

Aplikace funkce na argument implicitně sdružuje zleva.

$$(\ldots(((f \quad x_1) \quad x_2) \quad x_3) \quad \ldots \quad x_{n-1}) \quad x_n$$

Příklad: Logaritmické funkce

$$\ln : \mathbb{R}^+ \to \mathbb{R}, \quad \log_2 : \mathbb{R}^+ \to \mathbb{R}, \quad \log_{10} : \mathbb{R}^+ \to \mathbb{R}, \quad \ldots$$

Zavedeme funkci $\mathrm{logBase} : (\mathbb{R}^+ - \{1\}) \;\to\; (\mathbb{R}^+ \to \mathbb{R})$

$$\begin{aligned}
\ln &= \mathrm{logBase}\,(\mathrm{e}) \\
\log_2 &= \mathrm{logBase}\,(2) \\
\log_{10} &= \mathrm{logBase}\,(10) \\
&\vdots
\end{aligned}$$

Pak lze psát

$$\begin{aligned}
\mathrm{logBase}\,(\mathrm{e})\,(\mathrm{e}) &= \ln \mathrm{e} &= 1 \\
\mathrm{logBase}\,(2)\,(32) &= \log_2 32 &= 5 \\
\mathrm{logBase}\,(10)\,(1000) &= \log_{10} 1000 &= 3 \\
&\vdots
\end{aligned}$$

Zápis v Haskellu:

```
logBase :: Floating a => a -> a -> a

log :: Floating a => a -> a
log = logBase 2.71828182845904523536

log2 :: Floating a => a -> a
log2 = logBase 2

log10 :: Floating a => a -> a
log10 = logBase 10
```

Funkce `curry` a `uncurry`

```
mul'      ::  (Int,Int) -> Int
mul' (x,y)  =  x * y

mul       ::  Int -> (Int -> Int)
mul x y   =  x * y
```

Převod mezi binární funkcí `mul` a funkcí na dvojicích `mul'` umožňují haskellovské funkce `curry` a `uncurry`:

```
mul   =  curry mul'
mul'  =  uncurry mul
```

```
curry          ::  ((a,b) -> c) -> (a -> b -> c)
curry g         =  f    where f x y = g (x,y)
```

nebo ekvivalentně:

```
curry          ::  ((a,b) -> c) -> a -> b -> c
curry g x y     =  g (x,y)
```

```
uncurry          ::  (a -> b -> c) -> ((a,b) -> c)
uncurry f         =  g    where g (x,y) = f x y
```

nebo ekvivalentně:

```
uncurry          ::  (a -> b -> c) -> (a,b) -> c
uncurry f (x,y)   =  f x y
```

Lambda abstrakce

Lambda abstrakce je způsob, jak z výrazu obsahujícího volnou proměnnou udělat funkci.

Nechť N je výraz obsahující proměnnou x (tzv. volnou proměnnou, tj. nepoužitou v jiné lambda abstrakci). Pak výraz $\lambda\, x.N$ označuje funkci, která po aplikaci na libovolný výraz M vrací hodnotu výrazu N', kde N' vznikne z výrazu N, když v něm všechny (volné) výskyty proměnné x nahradíme výrazem M.

Haskellovská syntax: `\` x `->` N

Příklady: Funkce, která ke svému argumentu přičte 5 a součet umocní na druhou, je $\lambda\, n.(n+5)^2$. Zapsáno v Haskellu: `\n -> (n+5)^2`.

Takže třeba `(\n -> (n+5)^2) (3+4)` $\leadsto^*$ `144`.

Funkce, která obalí řetězec hranatými závorkami, je

`\r -> "[" ++ r ++ "]" :: String -> String`.

```
map (\ n -> 3 * n + 1) [0,1,2,3,4,5]  ⇝*  [1,4,7,10,13,16]

map (\ x -> [x,x,x]) "abc"  ⇝*  ["aaa","bbb","ccc"]

(\ f -> f . f . f) (\ n -> 2 * n + 1) 1  ⇝*  15
```

Definice

```
f x  =  M
```

je ekvivalentní definici

```
f  =  \ x -> M
```

Podobně i definici funkce více proměnných

```
((f x) y) z  =  M
```

lze zapsat

```
f  =  \ x -> (\ y -> (\ z -> M))
```

nebo zkráceně

```
f  =  \ x y z -> M
```

Obecně

$$\lambda x_1 \lambda x_2 \ldots \lambda x_n . M \quad \equiv \quad \lambda x_1\, x_2\, \ldots\, x_n . M$$

Lambda abstrakce lze nahradit lokálními definicemi:

$$\lambda\, x.E \quad \approx \quad \texttt{let f } x \texttt{ = } E \texttt{ in f}$$

$$\lambda\, x_1\, x_2\, \ldots\, x_n.E \quad \approx \quad \texttt{let f } x_1 \;\; x_2 \; \ldots \; x_n \texttt{ = } E \texttt{ in f}$$

ale lokálně definovaná funkce ve výrazu let musí být pojmenovaná.

Lambda abstrakce je *anonymní* funkce.

Operátorové sekce

Pro každý binární operátor $(\oplus) :: a \to b \to c$ a pro hodnoty $p :: a$, $q :: b$ se zavádí *pravá* a *levá operátorová sekce* takto:

$$(p\oplus) \quad = \quad \backslash\texttt{y} \texttt{ -> } p \oplus \texttt{y}$$

$$(\oplus q) \quad = \quad \backslash\texttt{x} \texttt{ -> } \texttt{x} \oplus q$$

Tedy $(p\oplus) = (\oplus)\ p$

$(\oplus q) = \texttt{flip}\ (\oplus)\ q$

Příklady:	`(1+)`	funkce přičítající jedničku, následník
	`(*2)`	zdvojnásobující funkce
	`(^2)`	kvadratická funkce
	`(2^)`	exponenciální funkce se základem 2
	`(1.0/)`	funkce převrácené hodnoty
	`(:[])`	funkce vytvářející jednoprvkový seznam, singleton
	``(`mod`2)``	funkce parity
	`(>0)`	predikát kladnosti
	`(==False)`	ekvivalentní s funkcí `not`
	`(!!0)`	ekvivalentní s funkcí `head`
	`(++".")`	funkce připojující tečku na konec řetězce

Příklady:

`(0==).(`mod`2)`	ekvivalentní s funkcí `even`
`(1==).(`mod`2)`	ekvivalentní s funkcí `odd`
`(^2).(^2)`	ekvivalentní s `(^4)`
`(^2).(2^)`	ekvivalentní s `(4^)`
`(2^).(^2)`	ekvivalentní s $\lambda\, n.\, (2^n)^n$
`(2^).(2^)`	dvojitě exponenciální funkce, $\lambda\, n.\, 2^{2^n}$
`('[':).(++"]")`	funkce, která obalí řetězec závorkami, $\lambda\, r.$ `'['` : r ++ `"]"`

Akumulační funkce na seznamech `foldr`, `foldl`

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

```
and          ::   [Bool] -> Bool
and []        =   True
and (x:s)     =   x && and s
```

```
(||)  ::  Bool -> Bool -> Bool
False ::  Bool
```

```
or           ::   [Bool] -> Bool
or []         =   False
or (x:s)      =   x || or s
```

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

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

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

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

```
and []      =  True                          and = foldr (&&) True
and (x:s)   =  x && and s
```

```
(||)  ::  Bool -> Bool -> Bool
False ::  Bool
```

```
or []      =  False                          or = foldr (||) False
or (x:s)   =  x || or s
```

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

```
concat []      =  []                         concat = foldr (++) []
concat (t:s)   =  t ++ concat s
```

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

$$\texttt{foldr}\ (\otimes)\ v\ [x_1,\ldots,x_n] \quad \leadsto^* \quad x_1 \otimes (x_2 \otimes (\ldots (x_{n-1} \otimes (x_n \otimes v)) \ldots))$$

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

$$\texttt{foldl}\ (\otimes)\ v\ [x_1,\ldots,x_n] \quad \leadsto^* \quad ((\ldots((v \otimes x_1) \otimes x_2) \ldots) \otimes x_{n-1}) \otimes x_n$$

Poznámka: Funkci `foldl` lze definovat pomocí `foldr`:

```
foldl f  =  flip ( foldr (\ x g z -> g (f z x)) id )
```

```
sum  =  foldl (+) 0
```

sum $[x_1, \ldots, x_n] \leadsto^* ((\ldots((0 + x_1) + x_2)\ldots) + x_{n-1}) + x_n \;=\; x_1 + \cdots + x_n$

```
product  =  foldl (*) 1
```

product $[x_1, \ldots, x_n] \leadsto^* ((\ldots((1 * x_1) * x_2)\ldots) * x_{n-1}) * x_n \;=\; x_1 * \cdots * x_n$

Poznámka: Alternativní definice funkcí `map` a `filter`:

```
map g  =  foldr ((:)·g) []
filter p  =  foldr (\ x -> if p x then (x:) else id) []
```

Funkce `foldr` je tzv. *katamorfismus* na seznamech.

Katamorfismy jsou funkce, které k dané datové struktuře sestrojí výraz stejného tvaru, ale datové konstruktory nahradí předepsanými funkcemi.

$$\texttt{foldr}\ (\otimes)\ v\quad x : (y : (z : []))\quad \rightsquigarrow^{*}\quad x \otimes (y \otimes (z \otimes v))$$

Akumulační funkce pro neprázdné seznamy

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

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

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

$$\texttt{foldl1 } (\otimes)\ [x_1, \ldots, x_n] \quad \rightsquigarrow^* \quad ((\ldots ((x_1 \otimes x_2) \otimes x_3) \ldots) \otimes x_{n-1}) \otimes x_n$$

```
maximum ::  Ord a =>  [a] -> a
maximum  =  foldl1 max
```

```
minimum ::  Ord a =>  [a] -> a
minimum  =  foldl1 min
```

```
unwords     :: [String] -> String
unwords []   =  ""
unwords s    =  foldr1 (\ w t -> w ++ (' ':t)) s
```

Redukce, redukční strategie

Redukční krok je úprava výrazu, v němž se některý jeho podvýraz nahradí zjednodušeným podvýrazem. Přitom upravovaný podvýraz (tzv. *redex*) má tvar aplikace funkce na argumenty a upravený podvýraz má tvar pravé strany definice funkce nebo těla abstrakce, do nichž za formální parametry dosadíme skutečné argumenty.

Redukční strategie je pravidlo, které pro každý výraz jednoznačně určuje první redukční krok.

Striktní redukční strategie

Při úpravě aplikace $F\ X$ nejdříve úplně upravíme argument X. Teprve nelze-li už upravovat argument X, upravujeme výraz F. Až nakonec upravíme (podle definice funkce) celý výraz $F\ X$. Při úpravě výrazů tedy postupujeme *zevnitř*.

Normální redukční strategie

Upravovaným podvýrazem je celý výraz; nelze-li takto upravit aplikaci $F\ X$, upravíme nejdříve výraz F, abychom mohli upravit $F\ X$. Při úpravě výrazů tedy postupujeme *zvnějšku*.

Normální redukční strategii, při níž si pamatujeme hodnoty upravených podvýrazů a žádný s opakovaným výskytem nevyhodnocujeme více než jednou, se říká *líná redukční strategie*.

Haskell používá línou redukční strategii. Funkcionální jazyky ML, Lisp, Scheme používají striktní redukční strategii. Říkáme také, že jazyky mají líné resp. striktní *vyhodnocování*.

Příklady:

```
cube x  =  x * x * x
```

Striktně:

cube (3+5) ⇝ cube 8 ⇝ 8 * 8 * 8 ⇝ 64 * 8 ⇝ 512

Normálně:

cube (3+5) ⇝ (3+5) * (3+5) * (3+5) ⇝ 8 * (3+5) * (3+5) ⇝
8 * 8 * (3+5) ⇝ 64 * (3+5) ⇝ 64 * 8 ⇝ 512

Líně:

cube (3+5) ⇝ (3+5) * (3+5) * (3+5) ⇝ 8 * 8 * 8 ⇝ 64 * 8 ⇝ 512

```
const x y  =  x
```

Striktně:

$\texttt{const 2 }\underline{\texttt{(1/0)}} \leadsto \bot$

Líně:

$\underline{\texttt{const 2 (1/0)}} \leadsto 2$

```
head (x:_)  =  x
```

```
tail (_:s)  =  s
```

```
undf  ::  Int -> Int
undf x  =  undf x
```

Striktně:

head (tail [undf 1, 4]) $=$ head (tail (undf 1 : 4 : [])) $\leadsto \cdots$

Líně:

head (tail [undf 1, 4]) $=$ head (tail (undf 1 : 4 : [])) $\leadsto$

head (4 : []) $\leadsto$ 4

```
or  =  foldr (||) False

foldr _  v []       =  v
foldr op v (x:s)    =  x `op` foldr op v s

True  || _  =  True
False || y  =  y
```

Striktně :

```
or [True,False] ⇝
foldr (||) False [True,False] ⇝
True || foldr (||) False [False] ⇝
True || (False || foldr (||) False []) ⇝
True || (False || False) ⇝
True || False ⇝
True
```

Líně:

```
or [True,False] ⇝
foldr (||) False [True,False] ⇝
True || foldr (||) False [False] ⇝
True
```

Věta: [Churchova-Rosserova] Výsledek ukončeného výpočtu výrazu nezáleží na redukční strategii: pokud výpočet skončí, je jeho výsledek vždy stejný.

Churchova-Rosserova věta nevylučuje různé *chování* výpočtu při různých strategiích. Při některých strategiích může výpočet skončit, při jiných cyklit. Nebo je výpočet podle jedné strategie delší než podle jiné. Nikdy však nemůže *skončit* dvěma různými výsledky.

Věta: [o perpetualitě] Jestliže pro nějaký výraz M existuje redukční strategie, s jejímž použitím se úprava výrazu M zacyklí, pak se tento výpočet zacyklí i s použitím striktní redukční strategie.

Věta o perpetualitě říká, že z hlediska možnosti zacyklení výpočtu je striktní redukční strategie nejméně bezpečná. Když se při jejím použití výpočet nezacyklí, pak se nezacyklí ani při žádné jiné strategii.

Věta: [o normalizaci] Jestliže pro nějaký výraz M existuje redukční strategie, s jejímž použitím se úprava výrazu M nezacyklí, pak se tento výpočet nezacyklí ani s použitím normální redukční strategie.

Věta o normalizaci tedy říká, že z hlediska možnosti zacyklení výpočtu je normální redukční strategie nejbezpečnější. To neznamená, že by se s jejím použitím výpočet zacyklit nemohl; z věty však plyne, že když se to stane a výpočet se i při normální redukční strategii zacyklí, pak se zacyklí i při každé jiné strategii.

Nekonečné seznamy

V jazycích s líným vyhodnocováním lze pracovat s nekonečnými datovými strukturami, například se seznamy (potenciálně) nekonečné délky.

Příklad:

```
jednicky  =  1 : jednicky
```

Vyhodnocení výrazu `jednicky` se zacyklí při každé strategii:

`jednicky` $\leadsto$ `1:jednicky` $\leadsto$ `1:1:jednicky` $\leadsto$ `1:1:1:jednicky` $\leadsto \cdots$

Ale je-li výraz `jednicky` podvýrazem většího výrazu, tak se jeho vyhocení při líné strategii nemusí zacyklit.

Striktně: $\texttt{head}\ \underline{\texttt{jednicky}} \leadsto \texttt{head}\ (\texttt{1:}\underline{\texttt{jednicky}}) \leadsto \texttt{head}\ (\texttt{1:1:}\underline{\texttt{jednicky}}) \leadsto \cdots$

Líně: $\underline{\texttt{head jednicky}} \leadsto \underline{\texttt{head (1:jednicky)}} \leadsto \texttt{1}$

Příklady:

Nekonečný rostoucí seznam všech přirozených čísel

```
nats  =  0 : zipWith (+) jednicky nats
```

Fibonacciho posloupnost

```
fibs  =  0 : 1 : zipWith (+) fibs (tail fibs)
```

Funkce generující nekonečné seznamy

```
repeat    ::  a -> [a]
repeat x  =   x : repeat x
```

Platí `jednicky` = `repeat 1`

```
iterate       ::  (a -> a) -> a -> [a]
iterate f z   =   z : iterate f (f z)
```

Platí `nats` = `iterate (1+) 0`

Poznámka: Funkce `iterate` je *anamorfismus* kodatového typu nekonečných seznamů.

Zápis seznamů hromadným výčtem

Konečné seznamy s malým počtem prvků lze zapisovat *prostým výčtem*:

`[1,2,3,4,5]` — ekvivalentní zápisu `1:2:3:4:5:[]`.

Seznamy diskrétně seřaditelných hodnot (přesněji: hodnot typů třídy `Enum`) lze zapisovat také *hromadným výčtem*: `[1 .. 5]` je ekvivalentní výrazu `enumFromTo 1 5`.

Hromadným výčtem lze zapsat i nekonečné seznamy. Například rostoucí seznam všech přirozených čísel: `[0 ..]` je ekvivalentní výrazu `enumFrom 0`.

Příklady:

`[0 .. 5]`	$\rightsquigarrow^*$	`[0,1,2,3,4,5]`
`[0 .. 0]`	$\rightsquigarrow^*$	`[0]`
`[1 .. 0]`	$\rightsquigarrow^*$	`[]`
`['a' .. 'e']`	$\rightsquigarrow^*$	`"abcde"`
`['z' .. 'r']`	$\rightsquigarrow^*$	`""`
`[3 ..]`	$\rightsquigarrow^\omega$	`[3,4,5,6,7, .....]`
`[-2 ..]`	$\rightsquigarrow^\omega$	`[-2,-1,0,1,2,3, .....]`

Syntax zápisu seznamů hromadným výčtem má i variantu s udaným druhým prvkem.

Například `[3,6 .. 99]` je seznam kladných násobků trojky menších než sto, `[0,2 ..]` je rostoucí seznam všech sudých přirozených čísel.

Odpovídající funkce jsou `enumFromThenTo` a `enumFromThen`.

`[`m `..]`	$\equiv$	`enumFrom` m
`[`m`,`m' `..]`	$\equiv$	`enumFromThen` m m'
`[`m `..` n`]`	$\equiv$	`enumFromTo` m n
`[`m`,`m' `..` n`]`	$\equiv$	`enumFromThenTo` m m' n

Příklady:

`[1,3 .. 8]`	$\rightsquigarrow^*$	`[1,3,5,7]`
`[1,-2 .. -8]`	$\rightsquigarrow^*$	`[1,-2,-5,-8]`
`[1,3 .. 2]`	$\rightsquigarrow^*$	`[1]`
`[1,5 .. 0]`	$\rightsquigarrow^*$	`[]`
`['a','d' .. 'z']`	$\rightsquigarrow^*$	`"adgjmpsvy"`
`[2,6 ..]`	$\rightsquigarrow^\omega$	`[2,6,10,14,18,22, ......]`
`[1,-1 ..]`	$\rightsquigarrow^\omega$	`[1,-1,-3,-5,-7,-9, ......]`
`[1,1 ..]`	$\rightsquigarrow^\omega$	`[1,1,1,1,1,1,1,1,1, ......]`
`['*','*' ..]`	$\rightsquigarrow^\omega$	`"*************** ......"`

Intensionální zápis seznamů

Množinu v matematickém zápisu můžeme zadat tak, že explicitně udáme všechny její prvky výčtem: $\{0, 2, 4, 6, 8, 10, 12, 14, 16, 18\}$.

Výčet může být dán i pomocí implicitního pravidla „tři tečky“: $\{0, 2, 4, \ldots, 16, 18\}$.

Množinu lze zadat také explicitním pravidlem: $\{2n \mid n \in \mathbb{N}, 0 \leq n \leq 9\}$.

Podobně lze v Haskellu zapisovat seznamy:

Explicitně prostým výčtem:

`[0,2,4,6,8,10,12,14,16,18]` — *extensionální zápis*

Hromadným výčtem:

`[0,2 .. 18]` — *intensionální zápis*

Pravidlem:

`[ 2*n | n<- [0 .. 9] ]` — *intensionální zápis*

Příklady:

```
[ n^2 | n <- [0 .. 3] ]
    ⇝*   [0,1,4,9]

[ (c,k) | c <- "abc", k <- [1,2] ]
    ⇝*   [('a',1),('a',2),('b',1),('b',2),('c',1),('c',2)]

[ 3*n | n <- [0 .. 6], odd n ]
    ⇝*   [3,9,15]

[ (m,n) | m <- [1 .. 3], n <- [1 .. 3], n <= m ]
    ⇝*   [(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]

[ (m,n) | m <- [1 .. 3], n <- [1 .. m] ]
    ⇝*   [(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]

[ (x,y) | z <- [0 .. 2], x <- [0 .. z], let y = z-x ]
    ⇝*   [(0,0),(0,1),(1,0),(0,2),(1,1),(2,0)]
```

Příklady:

```
[ replicate n c | c <- "xyz", n <- [2,3] ]
    ⇝*    ["xx","xxx","yy","yyy","zz","zzz"]

[ replicate n c | n <- [2,3], c <- "xyz" ]
    ⇝*    ["xx","yy","zz","xxx","yyy","zzz"]

[ x^2 | [x] <- [[],[2,3],[4],[1,1..],[],[7],[0..]] ]
    ⇝*    [16,49]

[ 0 | [] <- [[],[2,3],[4],[0..],[],[5]] ]
    ⇝*    [0,0]

[ x^3 | x <- [0..10], odd x ]
    ⇝*    [1,27,125,343,729]

[ x^3 | x <- [0..10], odd x, x < 1 ]
    ⇝*    []
```

Příklady:

```
length    ::   [a] -> Int
length s  =  sum [ 1 | _ <- s ]
```

nebo `length = foldl’ (const•(1+)) []`

```
map      ::   (a->b) -> [a] -> [b]
map f s  =   [ f x | x <- s ]
```

nebo `map f = foldr ((:)•f) []`

```
filter      ::   (a->Bool) -> [a] -> [a]
filter p s  =   [ x | x <- s, p x ]
```

nebo `filter p = foldr (λ x -> if p x then (x:) else id) []`

```
concat   ::  [[a]] -> [a]
concat s  =  [ x | t <- s, x <- t ]
```

nebo `concat = foldr (++) []`

```
serazene   ::  Ord a  =>  [a] -> Bool
serazene s  =  and [ x<=y | (x,y) <- zip s (tail s) ]
```

nebo `serazene s = and (zipWith (<=) s (tail s))`

```
samohlasky   ::  String -> String
samohlasky s  =  [ v | v <- s, v `elem` "aeiouy" ]
```

nebo `samohlasky = filter (`elem`"aeiouy")`

Příklad: Funkce `qSort` seřadí seznam hodnot vzestupně. Seznam musí být konečný a hodnoty v něm porovnatelné podle velikosti.

```
qSort         ::  Ord a  =>  [a] -> [a]
qSort []       =  []
qSort (p:s)    =  qSort [ x | x<-s, x<p ] ++ [p] ++ qSort [ x | x<-s, x>=p ]
```

Příklad: Seznam všech uspořádaných dvojic přirozených čísel.

$$\begin{array}{cccccc}
(0,0) & (0,1) & (0,2) & (0,3) & (0,4) & \cdots \\
(1,0) & (1,1) & (1,2) & (1,3) & (1,4) & \cdots \\
(2,0) & (2,1) & (2,2) & (2,3) & (2,4) & \cdots \\
(3,0) & (3,1) & (3,2) & (3,3) & (3,4) & \cdots \\
(4,0) & (4,1) & (4,2) & (4,3) & (4,4) & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

Správně: `[ (x,y) | z <- [0 ..], x <- [0 .. z], let y = z-x ]`

Příklad: Seznam všech přirozených čísel, která lze vyjádřit aspoň dvěma různými způsoby jako součet třetích mocnin dvou přirozených čísel. (Například $1729 = 1^3 + 12^3 = 9^3 + 10^3$.)

```
[ z | a <- [4 ..], b <- [3 .. a-1], c <- [2 .. b-1], d <- [1 .. c-1],
        let z = a^3 + d^3, b^3 + c^3 == z ]
```

Příklad: Eratosthenovo síto. $({}^{\prime}E\rho\alpha\tau o\sigma\vartheta\acute{\varepsilon}\nu\eta\varsigma)$.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	3		5		7		9		11		13		15		17		19		21		23		25		27		29
			5		7				11		13				17		19				23		25				29
					7				11		13				17		19				23						29
									11		13				17		19				23						29
											13				17		19				23						29
															17		19				23						29
																	19				23						29
																					23						29
																											29

Pro každé p, $2 \leq p \in \mathbb{N}$ platí: p je prvočíslo, právě když p není násobkem žádného prvočísla menšího než p.

```
primes = es [2 ..]  where es (p:t) = p : es [ n | n<-t, n`mod`p/=0 ]
```

Syntax seznamů zapsaných pomocí pravidel

intensionální_seznam ::= `[` *výraz* `|` *kvalifikátory* `]`

kvalifikátory ::= *kvalifikátor* `, ... ,` *kvalifikátor*

kvalifikátor ::= *generátor*

::= *podmínka*

::= *lokální_definice*

generátor ::= *vzor* `<-` *seznam*

podmínka ::= *výraz_typu_Bool*

lokální_definice ::= `let` *vzor* = *výraz*

Sémantika seznamů zapsaných pomocí pravidel

[e | q, Q] $\equiv$ concat [[e | Q] | q]

[e | p <- t] $\equiv$ loop t where loop [] = []
loop (p:s) = e : loop s
loop (_:s) = loop s

[e | b] $\equiv$ if b then [e] else []

[e | let p = e'] $\equiv$ let p = e' in [e]

Časová složitost

Časová složitost funkce popisuje délku výpočtu, tj. počet redukčních kroků, *v nejhorším případě* pro danou velikost parametru.

Nejhorší případ znamená *maximum* z délek všech výpočtů při aplikacích dané funkce na parametry o stejné velikosti.

Příklad: Funkce `reverse'` a `reverse` obracejí konečné seznamy.

```
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]
```

```
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]
```

Počet redukčních kroků výrazu `reverse'` $[x_1, \ldots, x_n]$ na každém seznamu délky n je $n + 1 + 1 + 2 + 3 + \cdots + n = \dfrac{n^2 + 3n + 2}{2}$.

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

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.

Příklad:

```
mocnina'     ::  Int -> Int -> Int
mocnina' m 0  =  1
mocnina' m n  =  m * mocnina' m (n-1)
```

```
mocnina     ::  Int -> Int -> Int
mocnina m 0  =  1
mocnina m n  =  if even n then r else m * r
                where  r = mocnina (m * m) (n `div` 2)
```

Složitost funkce `mocnina'` je *lineární* vzhledem k exponentu.

Složitost funkce `mocnina` je *logaritmická* vzhledem k exponentu.

Příklad:

```
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 funkce `fib'` je *exponenciální* vzhledem k argumentu.

Složitost funkce `fib` je *lineární* vzhledem k argumentu.

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

Příklad: Vyšetřujeme časovou složitost funkce `ins` v jejím druhém parametru. Funkce vkládá prvek na „správné místo“ v seřazeném seznamu. V aplikaci `ins` $x\ t$ se předpokládá, že seznam t je neklesající, a číslo x se zařadí na takové místo do t, aby zůstal neklesající.

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

Počet kroků při volání `ins` x $[x_1, \ldots, x_n]$ je od 3 (např. `ins 1 [2,4,6,8]`) do $3n + 1$ (např. `ins 9 [2,4,6,8]`).

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

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

```
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
```

$$\texttt{inssort}\ [x_1, x_2, \ldots, x_{n-1}, x_n]$$

$$\leadsto \quad \texttt{foldr ins []}\ [x_1, x_2, \ldots, x_{n-1}, x_n]$$

$$\leadsto^{n+1} \quad \texttt{ins}\ x_1\ (\texttt{ins}\ x_2\ (\ldots(\texttt{ins}\ x_{n-1}\ (\texttt{ins}\ x_n\ \texttt{[]}))\ldots))$$

```
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
```

Striktně (nejhorší případ – seznam je klesající):

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

```
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
```

Líně:

	minim [x_1,...,x_n]
$\leadsto$	(head.inssort) [x_1,...,x_n]
$\leadsto$	head (inssort [x_1,...,x_n])
$\leadsto$	head (foldr ins [] [x_1,...,x_n])
$\leadsto^{n+1}$	head (ins x_1 (...(ins x_{n-2} (ins x_{n-1} (ins x_n []))...))
$\leadsto$	head (ins x_1 (...(ins x_{n-2} (ins x_{n-1} (x_n : []))...))
$\leadsto^3$	head (ins x_1 (...(ins x_{n-2} (x_n : (ins x_{n-1} []))...))
$\leadsto^3$	head (ins x_1 (...(x_n : (ins (x_{n-2} (ins x_{n-1} []))...))
$\vdots$	
$\leadsto^3$	head (ins x_1 (x_n : (ins x_2 (ins x_3 (... (ins x_{n-1} [])...)))))
$\leadsto^3$	head (x_n : (ins x_1 (ins x_2 (... (ins x_{n-1} [])...))))
$\leadsto$	x_n

Funkce `minim` má při *striktním* vyhodnocování *kvadratickou* časovou složitost.

Nalezení minima n-prvkového seznamu trvá v nejhorším případě

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

Funkce `minim` má při *líném* vyhodnocování *lineární* časovou složitost.

Nalezení minima n-prvkového seznamu trvá v nejhorším (a každém) případě

$3 + n + 1 + 1 + 3 \cdot (n - 1) + 1 = 4n + 3$ kroky.

Asymptotický růst funkcí

Při určování časové složitosti algoritmů je nepraktické a často i obtížné určovat tuto složitost přesně. Místo toho ji zařazujeme do množin funkcí, které charakterizuje stejné *asymptotické chování* – růst funkcí pro libovolně velký měnící se parametr.

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

Příklady:

$t(n)$	růst funkce t	
$1, \quad 20, \quad 729, \quad 2^{64}$		konstantní
$\log n, \quad 1000 \ln n, \quad 3 \log_2 n + \log_2(\log_2 n)$		logaritmický
$n, \quad 2n+1, \quad n+\sqrt{n}$	*lineární*	polynomiální
$n^2, \quad 3n^2+4n-1, \quad n^2+10\log n$	*kvadratický*	
$n^3, \quad n^3+3n^2$	*kubický*	
2^n		exponenciální
$\left(\frac{1+\sqrt{5}}{2}\right)^n$		
3^n		

Prohledávání s návraty (backtracking)

Problém n dam: Rozestavit n šachových dam na pole čtvercové šachovnice $n \times n$ tak, aby se žádné dvě dámy navzájem neohrožovaly.

$n = 0$: jediné, triviální řešení `damy 0` $\leadsto^*$ `[[]]`

$n = 1$: `damy 1` $\leadsto^*$ `[[1]]`

$n = 2$: žádné řešení `damy 2` $\leadsto^*$ `[]`

$n = 3$: žádné řešení `damy 3` $\leadsto^*$ `[]`

$n = 4$: `damy 4` $\leadsto^*$ `[[3,1,4,2],[2,4,1,3]]`

$n = 5$:

`damy 5` $\leadsto^*$ `[[4,2,5,3,1], [3,5,2,4,1], [5,3,1,4,2], [4,1,3,5,2], [5,2,4,1,3], [1,4,2,5,3], [2,5,3,1,4], [1,3,5,2,4], [3,1,4,2,5], [2,4,1,3,5]]`

$n = 8$:

`damy 8` $\rightsquigarrow^*$ `[[4,2,7,3,6,8,5,1],[5,2,4,7,3,8,6,1],...,[5,7,2,6,3,1,4,8]]`

Seznam všech řešení pro čtvercovou šachovnici $n \times n$: `damy` n

Seznam všech řešení pro obdélníkovou šachovnici s m řádky a n sloupci: `da` m n

```
damy   ::  Int -> [[Int]]
damy n  =  da n n
```

`da` n m je seznam všech možností, jak postavit m dam do obdélníka o výšce m a šířce n (do každého řádku jednu) tak, aby se navzájem neohrožovaly.

Pomocná funkce `h` má první parametr pozici nové dámy v přidávaném (nultém) řádku a druhý parametr jedno řešení pro obdélník $(m-1) \times n$. Výsledkem je pravdivostní hodnota toho, zda sloupec a hlavní i vedlejší diagonála jsou volné.

```
h :: Int -> [Int] -> Bool
h k p  =  k `notElem` (p ++ zipWith (+) p [1 ..] ++ zipWith (-) p [1 ..])
```

$1 \quad 2 \quad 3 \quad 4 \quad \ldots \quad n$

$m-1$

$\vdots$

2

1

0

Částečné řešení `[6,1,3,5]` je seznam obsazených sloupců (číslo 4 se zde nevyskytuje – sloupec je volný)

`zipWith (+) [6,1,3,5] [1,2,3,4]` $\leadsto^*$ `[7,3,6,9]`

(číslo $4+0$ se zde nevyskytuje – zelená diagonála je volná)

`zipWith (-) [6,1,3,5] [1,2,3,4]` $\leadsto^*$ `[5,-1,0,1]`

(číslo $4-0$ se zde nevyskytuje – červená diagonála je volná)

```
damy     ::  Int -> [[Int]]
damy n   =   da n n

da       ::  Int -> Int -> [[Int]]
da _ 0   =   [[]]
da n m   =   [ k:p | p <- da n (m-1), k <- [1..n], h k p ]
             where h k p  =  k `notElem` (   p
                                          ++ zipWith (+) p [1..]
                                          ++ zipWith (-) p [1..] )
```

Ohraničení množiny možných řešení pomocí funkce h se nazývá *prořezávání* (*pruning*) stromu všech řešení.

Rekursivní generování řešení řízené prořezáváním se nazývá *prohledávání s návraty* — *backtracking*.

Binární stromy

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

```
tc ::  BinTree Char
tc  =  Node 'e' (Node 'í' Empty (Node 'c' Empty Empty))
                (Node 'j' (Node 'd' Empty Empty) (Node 'r' Empty Empty))
```

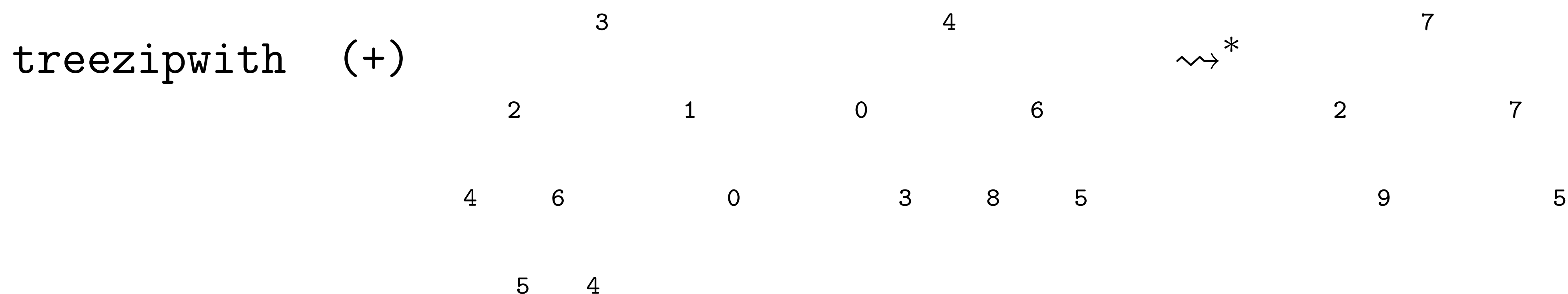
```
           'e'
      'í'        'j'
         'c'   'd'   'r'
```

```
tn ::  BinTree Int
tn  =  Node 4 (Node 2 (Node 0 Empty Empty) (Node 3 Empty Empty))
              (Node 7 Empty (Node 9 Empty Empty))
```

```
           4
      2          7
   0     3           9
```

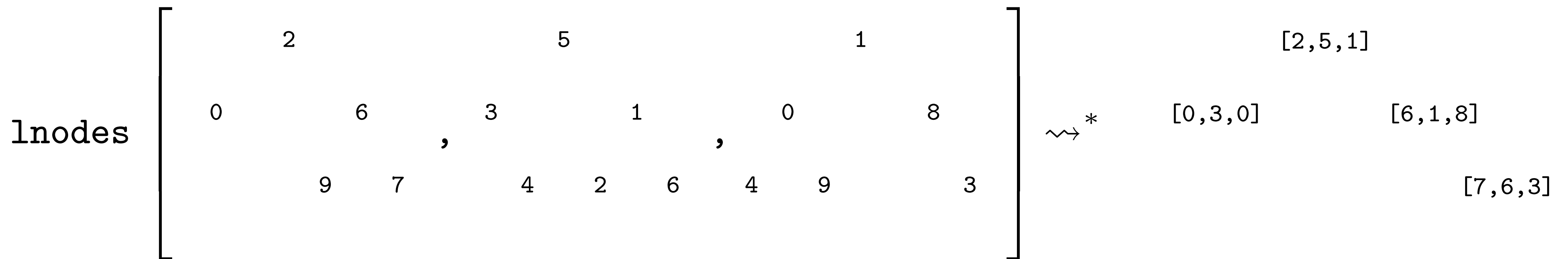
Příklad: Funkce `treezipwith` pomocí binární operace `op` vytvoří ze dvou stromů nový strom, jehož struktura bude „průnikem“ obou stromů a v jehož uzlech budou výsledky aplikace operace `op` na hodnoty uzlů ze stejné pozice v obou stromech.

```
treezipwith :: (a->b->c) -> BinTree a -> BinTree b -> BinTree c
treezipwith op (Node v1 l1 r1) (Node v2 l2 r2)
     = Node (v1 `op` v2) (treezipwith op l1 l2) (treezipwith op r1 r2)
treezipwith  _  _  _  =  Empty
```



Příklad: Funkce `lnodes` vytvoří ze seznamu stromů jeden strom, jehož struktura bude průnikem všech stromů ze seznamu a v jehož uzlech budou seznamy hodnot z uzlů na odpovídajících pozicích.

```
lnodes ::  [ BinTree a ]  ->  BinTree [a]
lnodes  =  foldr (treezipwith (:)) niltree
           where  niltree = Node [] niltree niltree
```

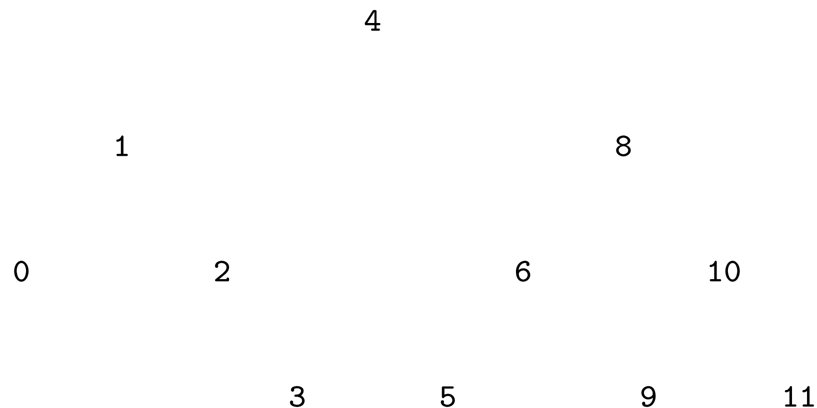


lnodes ['P' 'e' 's' / 'F' 'Á' , 'í' 'j' , 'k' 'a' / 'B' 'U' 'V' 'c' 'd' 'ř' 'm' 'n'] $\leadsto^*$ "Pes" "Fík" "Ája"

Binární vyhledávací stromy

Binární vyhledávací stromy jsou binární stromy, které mají v uzlech hodnoty, na nichž existuje úplné uspořádání (typicky čísla), a navíc levý podstrom každého uzlu v obsahuje jen hodnoty menší a pravý podstrom obsahuje jen hodnoty větší než je hodnota v uzlu v.

Příklad:



```
type  Vyhledavaci'  =  BinTree Int
-- nebo obecněji:
type  Vyhledavaci a  =  BinTree a
```

Test výskytu klíče

```
velem                 ::  Ord a => a -> Vyhledavaci (a,b) -> Bool
velem _ Empty          =  False
velem k (Node (v,_) l r)  =       k == v
                             ||   k < v   &&  velem k l
                             ||   k > v   &&  velem k r
```

Díky línému vyhodnocování operátorů `(||)` a `(&&)` se strom neprochází celý, ale jen jedna jeho větev.

Časová složitost funkce `velem` je lineární vzhledem k hloubce stromu.

Jaká je časová složitost funkce `velem` vzhledem k *velikosti* stromu?

Vyhledání položky

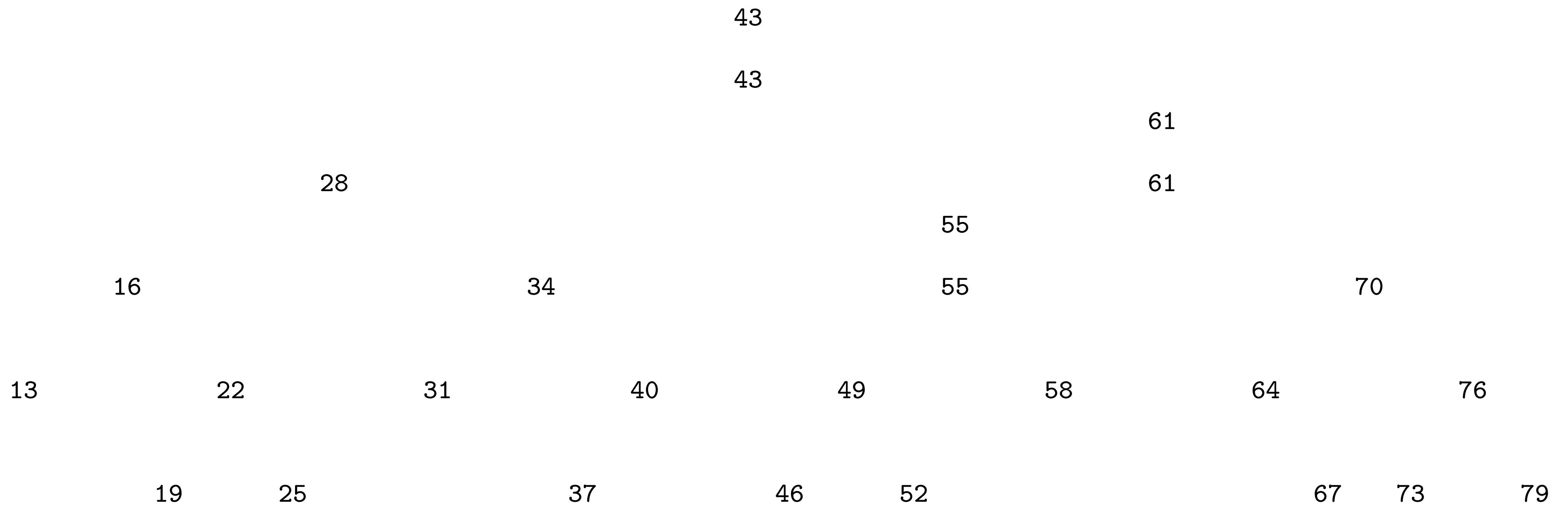
```
vfind    ::  Ord a => a -> Vyhledavaci (a,b) -> Maybe b
vfind _ Empty              =  Nothing
vfind k (Node (v,d) l r)   =  if           k < v   then  vfind k l
                                 else if k > v   then  vfind k r
                                 else {- k == v -}     Just d
```

Časová složitost funkce `vfind` je lineární vzhledem k hloubce stromu.

Vložení nové položky

```
vins                 ::  Ord a => (a,b) -> Vyhledavaci (a,b) -> Vyhledavaci (a,b)
vins p Empty          =  Node p Empty Empty
vins p@(k,_) (Node q@(v,_) l r)
                      =  if        k < v  then Node q (vins p l) r
                           else if k > v  then Node q l (vins p r)
                           else {- k == v -}   error "vins: duplicitní klíč"
```

Strom se nerekonstruuje celý, ale jen jeho jedna větev.


43
43
61
28
61
55
16
34
55
70
13
22
31
40
49
58
64
76
19
25
37
46
52
67
73
79

Věta: Je-li t vyhledávací strom, $p = (k, d)$ položka s klíčem k, který se v uzlech stromu t nevyskytuje, pak také `vins` p t je vyhledávací strom.

Důkaz Strukturní indukcí podle stromu t.

(i) Vyhledávací strom t je prázdný, $t = \texttt{Empty}$. Pak
$\texttt{vins}\, p\, t \;=\; \texttt{Node}\, p\, \texttt{Empty}\, \texttt{Empty}$, což je vyhledávací strom.

(ii) Nechť je vyhledávací strom t neprázdný, $t = \texttt{Node}\ (v, c)\ t'\ t''$.
I.P.: Jestliže $k < v$, pak je strom $\texttt{vins}\ (k, d)\ t'$ vyhledávací, a jestliže $k > v$, pak je strom $\texttt{vins}\ (k, d)\ t''$ vyhledávací.
Pak pro každý ze dvou případů $k < v$, $k > v$, z indukčního předpokladu, definice vyhledávacího stromu a definice funkce `vins` plyne, že strom $\texttt{vins}\ (k, d)\ t$ je vyhledávací.

Vyvážené vyhledávací stromy

Doba vyhledání (vložení, zrušení) uzlu je lineární vzhledem k *hloubce*, tedy pro obecné nevyvážené vyhledávací stromy je časová složitost lineární i vzhledem k *velikosti* stromu.

Snížíme-li hloubku stromu na logaritmus jeho velikosti, můžeme v něm vyhledávat s logaritmickou časovou složitostí (vzhledem k velikosti stromu). Ve vhodně definovaných stromech lze také přidání a zrušení uzlu provádět v čase lineárně závislém na hloubce, tedy stále s logaritmickou časovou složitostí.

Mluvíme pak o *vyvážených vyhledávacích stromech*.

Vstup a výstup

Zvláštními hodnotami v Haskellu jsou tzv. *vstupně-výstupní akce*.

Tyto akce mají typ `IO` a.

Při vyhodnocení akce typu `IO` a zpravidla dochází k nějakému načítání vstupu, vypisování výstupu, případně k jiné interakci s operačním systémem. Výsledek této interakce je typu a a uchová se jako tzv. *vnitřní výsledek* akce. Tento vnitřní výsledek je dostupný pouze speciálním operátorem `(>>=)` .

```
getChar   ::  IO Char
putChar   ::  Char -> IO ()
(>>=)     ::  IO a -> (a -> IO b) -> IO b
```

```
getChar  >>=  putChar . toUpper    ::    IO ()
```

Pro každý typ a má typ `IO` a formálně jedinou (navenek rozlišitelnou) hodnotu. Podle vnitřního výsledku nelze akce navenek rozlišit. Vnitřní výsledek akce je viditelný pouze ve druhém parametru operátoru `(>>=)` , což musí být funkce vracející akci.

Tím je formálně zachována referenční transparentnost i pro programy, které mění globální stav (obsah souborů, vstup, výstup).

Vytváření a skládání vstupních a výstupních akcí

Příklady:

```
putChar '*'                              ::   IO ()
getLine                                  ::   IO String
putStr "abc"                             ::   IO ()
writeFile "soubor.txt" "obsah\nsouboru"  ::   IO ()
```

Prázdná akce (bez účinku na vstup / výstup, s vnitřní hodnotou x):

```
return x
```

Řazení akcí za sebe:

```
putStr "abc"  >>  putStr "def"
```

Svázání s funkcí a řazení za sebe:

```
getLine  >>=  putStr . reverse
```

```
return  ::  a -> IO a
(>>)    ::  IO a -> IO b -> IO b
(>>=)   ::  IO a -> (a -> IO b) -> IO b
```

```
m1 >> m2  =  m1 >>= const m2
```

Knihovní akce a funkce a typy pro vstup a výstup

```
getChar      ::  IO Char
getLine      ::  IO String
getContents  ::  IO String
```

```
putChar    ::  Char -> IO ()
putStr     ::  String -> IO ()
putStrLn   ::  String -> IO ()
print      ::  (Show a)  =>  a -> IO ()
```

```
type FilePath = String

readFile     ::  FilePath -> IO String
writeFile    ::  FilePath -> String -> IO ()
appendFile   ::  FilePath -> String -> IO ()
```

```
getDirectoryContents   ::  FilePath -> IO [FilePath]
getCurrentDirectory    ::  IO FilePath
setCurrentDirectory    ::  FilePath -> IO ()
createDirectory, removeDirectory, removeFile :: FilePath -> IO ()
renameDirectory, renameFile  ::  FilePath -> FilePath -> IO ()
doesFileExist, doesDirectoryExist  ::  FilePath -> IO Bool
```

Zápis pomocí do

Syntaktická konstrukce do slouží k alternativnímu vyjádření výrazu s operátory `(>>=)` a `(>>)`.

```
putStr "vstupní soubor:"    >>
  getLine                   >>= (\vstup ->
  putStr "výstupní soubor:" >>
  getLine                   >>= (\vystup ->
  readFile vstup            >>= (\s ->
  writeFile vystup (map toUpper s))))
```

```
do putStr "vstupní soubor:"
   vstup <- getLine
   putStr "výstupní soubor:"
   vystup <- getLine
   s <- readFile vstup
   writeFile vystup (map toUpper s)
```

Notace pomocí `do` je přehlednější (méně operátorů, lambda-abstrakcí), ale někdy je kratší a čitelnější výraz s `(>>=)`, `(>>)`, `return`.

```
sequence_ :: [IO ()] -> IO ()
sequence_  = foldr (>>) (return ())
```

```
sequence_         :: [IO ()] -> IO ()
sequence_ []       = return ()
sequence_ (a:s)    = do  a
                         sequence_ s
```

Podobně

```
sequence :: [IO a] -> IO [a]
sequence = foldr (\a t -> a >>= (t >>=).(return.).(:)) (return [])
```

nebo ekvivalentně

```
sequence :: [IO a] -> IO [a]
sequence [] = return []
sequence (a:s) = do x <- a
                    t <- sequence s
                    return (x:t)
```

Pomocí `sequence` je zavedena funkce `mapM` (analogická funkci `map`).

```
mapM   ::  (a -> IO b) -> [a] -> IO [b]
mapM f  =  sequence . map f
```

Sémantika konstrukce `do`

`do` e	$\equiv$	e
`do` e *další_akce*	$\equiv$	e `>>` `do` *další_akce*
`do` v `<-` e *další_akce*	$\equiv$	e `>>=` $\lambda\, v$ `->` `do` *další_akce*
`do` `let` *definice* *další_akce*	$\equiv$	`let` *definice* `in` `do` *další_akce*

Program v Haskellu je akce typu `IO ()` pojmenovaná `main` a nacházející se v modulu `Main`.

```
module Main where
import Char

main ::  IO ()
main  =  do putStr "řetězec: "
            s <- getLine
            if null s
               then return ()
               else do putStr (if pal s then "Je" else "Není")
                       putStrLn " palindrom."
                       main

pal   ::  String -> Bool
pal s  =  t == reverse t
          where t = map toLower (filter isAlpha s)
```

Program lze:

1. Spustit v prostředí interaktivního interpretu.

```
$ hugs Main.hs
Main> main
    ⋮
Main> :quit
$
```

2. Spustit pomocí dávkového interpretu.

```
$ runhugs Main.hs
    ⋮
$
```

3. Dávkový interpret lze aktivovat také přidáním řádku „`#!/usr/bin/runhugs`“ na začátek skriptu.

```
$ ./Main.hs
    ⋮
$
```

4. Přeložit do binárního programu, ten pak spustit.

```
$ ghc -o pal Main.hs
$ ./pal
    ⋮
$
```

Interakce s operačním systémem

```
data ExitCode  =  ExitSuccess  |  ExitFailure Int

getArgs      ::  IO [String]
getProgName  ::  IO String
getEnv       ::  String -> IO String
system       ::  String -> IO ExitCode
```

Ošetření chyb při vstupu a výstupu

Chyby IO jsou hodnoty abstraktního typu `IOError`.

Chyby lze zpřístupnit funkcí `try`.

```
try  ::  IO a -> IO (Either IOError a)
```

```
isAlreadyExistsError      ::  IOError -> Bool
isDoesNotExistError       ::  IOError -> Bool
isAlreadyInUse            ::  IOError -> Bool
isFullError               ::  IOError -> Bool
isEOFError                ::  IOError -> Bool
isIllegalOperationError   ::  IOError -> Bool
isPermissionError         ::  IOError -> Bool

ioeGetErrorString         ::  IOError -> String
```

Příklad: Cat bez ošetření chyb

Cat je program, který spojí soubory vyjmenované na příkazovém řádku do jednoho souboru, ten pak pošle na standardní výstup.

```
module Main where
import System

main ::  IO ()
main  =  do args <- getArgs
            s <- sequence (map readFile args)
            putStr (unlines s)
```

Seznam `s` je například `["obsah prvniho", "obsah druheho", "obsah tretiho"]`.

Poznámka: Alternativně a kratčeji:

```
main  =  getArgs  >>=  mapM readFile  >>=  putStr . unlines
```

Cat s ošetřením chyb

```
module Main where
import IO
import System

main ::  IO ()
main  =  do args <- getArgs
            es <- sequence (map (try . readFile) args)
            (putStr . unlines . map ioe) es

ioe            ::  Either IOError String -> String
ioe (Left e)    =  "!!" ++ ioeGetErrorString e ++ "!!"
ioe (Right s)   =  s
```

Seznam es je například

`[Right "obsah prvniho", Left dnee, Right "obsah tretiho"]`,

kde `ioe (Left dnee)` = `"!!File does not exist!!"`.

Kvalifikované typy a typové třídy

Hodnoty monomorfních typů

```
False      ::  Bool
not        ::  Bool -> Bool
(&&)       ::  Bool -> Bool -> Bool
toUpper    ::  Char -> Char
isDigit    ::  Char -> Bool
chr        ::  Int -> Char
```

Hodnoty polymorfních typů

```
length     ::  [a] -> Int
const      ::  a -> b -> a
fst        ::  (a,b) -> a
curry      ::  ((a,b) -> c)  ->  (a -> b -> c)
flip       ::  (a -> b -> c)  ->  b -> a -> c
id         ::  a -> a
```

Hodnoty kvalifikovaných typů

```
(==), (/=)        :: Eq a   =>  a -> a -> Bool
elem, notelem     :: Eq a   =>  a -> [a] -> Bool
sum, product      :: Num a  =>  [a] -> a
(<), (<=), (>)    :: Ord a  =>  a -> a -> Bool
minimum, maximum  :: Ord a  =>  [a] -> a
print             :: Show a  =>  a -> IO ()
qSort             :: Ord a  =>  [a] -> [a]
vins              :: Ord a  =>  (a,b) -> Vyhledavaci (a,b) -> Vyhledavaci (a,b)
```

Typové třídy

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

```
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
```

```
instance  (Eq a)  =>  Eq (Tree a)   where
     Empty == Empty                    =   True
     Node u1 l1 r1 == Node u2 l2 r2    =      u1 == u2
                                          && l1 == l2
                                          && r1 == r2
     _ == _                            =   False
```

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

```
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
```

```
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)
```

```
instance  (Ord a)  =>  Ord [a]   where
     [] <= _          =   True
     (_:_) <= []      =   False
     (x:s) <= (y:t)   =   x < y  ||  (x == y && s <= t)
```

Definice typové třídy

`class` $\left[\ (C_1\ a, \ldots, C_k\ a)$ `=>` $\ \right]\ C\ a$

$\left[\ \texttt{where}\ \ op_1 :: typ_1\ \vdots\ op_n :: typ_n\ \left[\ default_1\ \vdots\ default_m\ \right]\ \right]$

Deklarace instance

`instance` $\left[\ (C_1\ a_1, \ldots, C_k\ a_k)$ `=>` $\ \right]\ C\ typ$

$\left[\ \texttt{where}\ \ valdef_1\ \vdots\ valdef_n\ \right]$

Přetížení operací

Jedna operace je pro několik různých typů operandů definována obecně různým způsobem (na rozdíl od parametricky polymorfních operací, které jsou definovány jednotně pro všechny typy operandů).

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

```
instance  Num Int  where
    (+)     =  primPlusInt
             ⋮

instance  Num Integer  where
    (+)     =  primPlusInteger
             ⋮

instance  Num Float  where
    (+)     =  primPlusFloat
             ⋮
```

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.

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

Hodnoty, typy, druhy

Podobně jako každá hodnota má svůj typ, můžeme i typy a typové konstruktory rozdělit do skupin podle takzvaných *druhů*.

Protože však typy jsou statické (nelze s nimi provádět výpočty a jediná užívaná ekvivalence na typech je syntaktická rovnost), je druhový systém (podjazyk druhových termů) mnohem chudší než typový systém Haskellu. Pomocí druhů se rozlišují typové konstruktory podle arity a druhů typových parametrů, ale všechny vlastní typy mají jen jediný druh – tzv. druh všech typů, označovaný symbolem $*$.

```
True         ::  Bool
                 Bool                ::  *
(+)          ::  (Num a) => a->a->a
                 (Num a) => a->a->a  ::  *
Just 'o'     ::  Maybe Char
                 Maybe Char          ::  *
Just         ::  a -> Maybe a
                 a -> Maybe a        ::  *
                 Maybe               ::  * -> *
getLine      ::  IO String
                 IO String           ::  *
                 IO                  ::  * -> *
(:)          ::  a -> [a] -> [a]
                 a -> [a] -> [a]     ::  *
                 [a]  ≡  [] a        ::  *
                 []                  ::  * -> *
                 (->)                ::  * -> * -> *
Left False   ::  Either Bool a
                 Either Bool a       ::  *
                 Either              ::  * -> * -> *
                 Either Bool         ::  * -> *
```

Konstruktorové třídy

Do tříd lze sdružovat nejen typy, ale obecněji i typové konstruktory.

Příklady konstruktorových tříd jsou `Functor` nebo `Monad`.

Instancemi konstruktorové třídy `Functor` jsou unární typové konstruktory, tj. typové konstruktory druhu `* -> *`, na jejichž typech je definovaná funkce `fmap`.

Instancemi konstruktorové třídy `Monad` jsou unární typové konstruktory, tj. typové konstruktory druhu `* -> *`, na jejichž typech jsou definované funkce `return`, `(>>=)`, `(>>)`.

Třída `Functor`

Srovnejme funkce `map` na seznamech, binárních stromech a hodnotách typu `Maybe a`.

```
map                   ::  (a -> b) -> [a] -> [b]
map _ []               =  []
map g (x:s)            =  g x : map g s
```

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

tmap                  ::  (a -> b) -> BinTree a -> BinTree b
tmap _ Empty           =  Empty
tmap g (Node v l r)    =  Node (g v) (tmap g l) (tmap g r)
```

```
data Maybe a   =   Nothing  |  Just a

mmap                  ::  (a -> b) -> Maybe a -> Maybe b
mmap _ Nothing         =  Nothing
mmap g (Just x)        =  Just (g x)
```

Typy všech tří funkcí se liší jen v typovém konstruktoru: `[]`, `BinTree`, `Maybe`.

```
class  Functor f  where
     fmap  ::  (a -> b) -> f a -> f b
```

```
instance  Functor []  where
     fmap  =  map

instance  Functor BinTree  where
     fmap  =  tmap

instance  Functor Maybe  where
     fmap  =  mmap
```

Třída má název `Functor`, protože ve všech instancích by mělo platit, že pro každé dvě funkce f, g jsou splněny rovnosti

$$\begin{aligned} \texttt{fmap id} &= \texttt{id} \\ \texttt{fmap}\,(f \cdot g) &= \texttt{fmap}\,f \cdot \texttt{fmap}\,g \end{aligned}$$

Třída `Monad`

```
class  Monad m  where
     return  ::   a -> m a
     (>>=)   ::   m a -> (a -> m b) -> m b

     (>>)    ::   m a -> m b -> m b
     p >> q   =   p  >>=  const q
```

Typy operací `return`, `(>>=)`, `(>>)` jsou tedy obecnější než pro monádu `IO`.

```
return  ::   Monad m  =>  a -> m a
(>>=)   ::   Monad m  =>  m a -> (a -> m b) -> m b
(>>)    ::   Monad m  =>  m a -> m b -> m b
```

```
instance  Monad Maybe  where
     return           =   Just
     Nothing >>= _    =   Nothing
     Just x  >>= k    =   k x
```

```
instance  Monad []  where
     return    =   (:[])
     (>>=)     =   flip concatMap
```

Algebraické monády mají operace `return` a `(>>=)`, které pro každé a, m, k, h splňují rovnosti

$$
\begin{aligned}
\texttt{return}\, a \ \texttt{>>=}\ k &= k\ a \\
m \ \texttt{>>=}\ \texttt{return} &= m \\
m \ \texttt{>>=}\ (\lambda x.\, k\ x \ \texttt{>>=}\ h) &= (m \ \texttt{>>=}\ k) \ \texttt{>>=}\ h
\end{aligned}
$$

Speciálně pro monádu `[]` (seznamový typový konstruktor) lze zápis seznamů pomocí pravidel (intensionální zápis) chápat jako třetí alternativní syntax pro výrazy řazené pomocí `(>>=)`.

$$m_1 \texttt{ >>= } (\backslash\, x_1 \texttt{->}\, m_2 \texttt{ >>= } (\ldots\ (\backslash\, x_n \texttt{->}\, \texttt{return}\ x_n) \ldots))$$

$$\equiv$$

$$\texttt{do \{ } x_1 \texttt{ <- } m_1 \texttt{; } \ldots \texttt{ ; } x_n \texttt{ <- } m_n \texttt{; return } x_n \texttt{ \}}$$

$$\equiv$$

$$\texttt{[} x_n \texttt{ | } x_1 \texttt{ <- } m_1 \texttt{, } \ldots \texttt{ , } x_n \texttt{ <- } m_n \texttt{]}$$

Zabudovaný typový konstruktor `IO` je instancí konstruktorové třídy `Monad`.

```
instance  Monad IO  where
     return   =   primretIO
     (>>=)    =   primbindIO
```

Hodnoty typu `IO` a jsou *akce*. Akce typu `IO` a má vnitřní výsledek typu a. Akce stejného typu s různými vnitřními výsledky nelze navenek rozlišit, protože je nelze porovnávat, zobrazovat, nelze definovat funkci z akcí do neakčních hodnot, která by závisela na vnitřním výsledku akcí. Jediná fukce, která má přístup k vnitřnímu výsledku akce, je zabudovaný operátor `primbindIO`.

Tato vlastnost „odstínění" vnitřních výsledků vstupně-výstupních akcí je důsledkem zabudované definice operací `(>>=)`, `return` *v instanci `IO`* konstruktorové třídy `Monad` a zabudovaných definic knihovních akcí a funkcí pro vstup/výstup.

Aditivní monády

Zvláštním případem monád jsou aditivní monády. Mají navíc konstantu `mzero` a operaci `mplus`.

```
class  Monad m  =>  MonadPlus m  where
     mzero  ::  m a
     mplus  ::  m a -> m a -> m a
```

Algebraické aditivní monády pro každé m splňují rovnosti

$$
\begin{aligned}
m \gg \texttt{mzero} &= \texttt{mzero} \\
\texttt{mzero} \gg m &= \texttt{mzero} \\
m \;\text{`}\texttt{mplus}\text{`}\; \texttt{mzero} &= m \\
\texttt{mzero} \;\text{`}\texttt{mplus}\text{`}\; m &= m
\end{aligned}
$$

Instance konstruktorové třídy MonadPlus

```
instance  MonadPlus Maybe  where
     mzero              =  Nothing
     Nothing `mplus` m  =  m
     Just x  `mplus` _  =  Just x
```

```
instance  MonadPlus []  where
     mzero  =  []
     mplus  =  (++)
```

Použití: nedeterministické výpočty, monadické kombinátory pro syntaktickou analýzu.

Monadické kombinátory pro syntaktickou analýzu

Typ parserů (syntaktických analyzátorů):

```
newtype Parser a  =  P (String -> [(a, String)])
```

Aplikace parseru na řetězec:

```
papply  ::  Parser a -> String -> [(a, String)]
papply (P p) s  =  p s
```

Typový konstruktor `Parser` je instancí konstruktorových tříd `Functor` a `Monad`.

```
instance Functor Parser  where
     fmap f (P p)  =  P (\ inp -> [(f v, out) | (v,out) <- p inp])

instance Monad Parser  where
     return v     =  P (\ i -> [(v,i)])
     (P p) >>= f  =  P (\ i -> concat [papply (f v) o | (v,o) <- p i])
```

`return` x je konstantní parser, který vždy uspěje (bez čtení vstupu).

Operátor `(>>=)` vytváří sekvence z parserů.

Věta: Typový konstruktor `Parser` splňuje monadické rovnosti i rovnosti pro funktory.

Typový konstruktor `Parser` je instancí konstruktorové třídy `MonadPlus`.

```
instance MonadPlus Parser  where
     mzero              =  P (\ i -> [])
     P p `mplus`  P q   =  P (\ i -> p i ++ q i)
```

`mzero` je nulový parser, který vždy selže.

`mplus` vytváří alternativy z parserů.

Věta: Typový konstruktor `Parser` splňuje rovnosti aditivních monád.

Knihovny parserů, a funkcí nad nimi, tzv. parserových kombinátorů:

- `ParseLib` jednodušší
- `Parsec` sofistikovanější (obecnější, s ošetřením chyb...)

Definují například základní parsery pro přečtení čísla, identifikátoru, klíčového slova,

kombinátor pro přečtení a analýzu posloupnosti

```
many  ::  Parser a -> Parser [a]
```

kombinátor pro přečtení a analýzu posloupnosti s oddělovači

```
sepBy  ::  Parser a -> Parser b ->  Parser [a]
```

kombinátory pro přečtení a analýzu výrazu s binárními operátory

```
chainl1, chainr1  ::  Parser a -> Parser (a -> a -> a) -> Parser a
```

Pole

Pole jsou v programování a v datových strukturách a algoritmech ortotopní schémata skládající se z prvků stejného typu a mající konečnou velikost. Podle počtu rozměrů ortotopu pak hovoříme o polích jednorozměrných (vektorech), dvourozměrných (maticích) a vícerozměrných.

V ideálním případě by nebylo nutno rozlišovat mezi poli a (konečnými) funkcemi.

$$\texttt{Array}\, I\; A \quad \cong \quad I \rightarrow A$$

Pro praktickou tabelaci funkčních hodnot a efektivní operaci, která tyto funkční hodnoty zpřístupňuje (indexační operaci `(!)`) však pro pole zavádíme nový binární typový konstruktor

$$\texttt{Array :: } * \texttt{ -> } * \texttt{ -> } *$$

Typ $\texttt{Array}\, I\; A$ je typem všech polí, která jsou indexovaná indexy typu I a obsahují prvky typu A.

V každém typu pole `Array` I A musí být jeho indexový typ I instancí typové třídy `Ix`. To umožňuje efektivní uložení pole v paměti a efektivní implementaci indexační operace `(!)`.

```
class  (Ord a) => Ix a  where
    range       :: (a,a) -> [a]
    index       :: (a,a) a -> Int
    inRange     :: (a,a) -> a -> Bool
```

Instancemi třídy `Ix` jsou typy `Int`, `Integer`, `Char`, `Bool` a jejich kartézské součiny.

Je-li a `::` `Array` I A a i `::` I je index, pak a`!`i je *hodnota pole a v indexu i*, neboli *i-tý prvek pole a*.

```
(!)  ::  Ix i  =>  Array i e -> i -> e
```

Indexování

Uspořádaná dvojice indexů tvoří *meze* pole.

Příklad:

`( 0, 9 )`	meze jednorozměrného 10prvkového pole
`( (1,1), (20,20) )`	meze čtvercové 400prvkové matice

`range ((0,0),(1,2))` $\leadsto^*$ `[(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)]`

`range (('a',1),('b',2))` $\leadsto^*$ `[('a',1),('a',2),('b',1),('b',2)]`

`index ((0,0),(3,4)) (1,1)` $\leadsto^*$ 6

`index ((0,0,0),(2,3,4)) (1,2,3)` $\leadsto^*$ 33

`inRange ((0,0,0),(2,3,4)) (3,3,3)` $\leadsto^*$ `False`

`inRange (('a','a'),('z','z')) ('x','y')` $\leadsto^*$ `True`

Příklad:

```
data  Barva  =  Cerna | Hneda | Cervena | Oranzova | Zluta
             | Zelena | Modra | Fialova | Seda | Bila
             deriving Ix
```

range (Cervena,Zluta) $\leadsto^*$ [Cervena,Oranzova,Zluta]

index (Hneda,Modra) Cervena $\leadsto^*$ 1

inRange (Seda,Bila) Oranzova $\leadsto^*$ False

index ((Cerna,Zelena),(Zluta,Bila)) (Cervena,Seda) $\leadsto^*$ 16

Vlastnosti operací

Pro každé meze l, u a každý index i platí:

$$\texttt{inRange}\ (l\texttt{,}u)\ i \quad = \quad i\ \texttt{`elem`}\ \texttt{range}\ (l\texttt{,}u)$$

jestliže $\texttt{inRange}\ (l\texttt{,}u)\ i\,,$ pak $\texttt{range}\ (l\texttt{,}u)\ \texttt{!!}\ \texttt{index}\ (l\texttt{,}u)\ i \quad = \quad i$

$$\texttt{map}\ \texttt{index}\ \big(\texttt{range}\ (l\texttt{,}u)\big) \quad = \quad \texttt{[0..rangeSize}\ (l\texttt{,}u)\ \texttt{-}\ \texttt{1]}$$

Operace s poli

Meze pole:

```
bounds   ::  Ix i  =>  Array i e -> (i, i)
```

Seznam indexů pole:

```
indices  ::  Ix i  =>  Array i e -> [i]
```

Převody mezi poli a seznamy:

```
elems      ::  Ix i  =>  Array i e -> [e]
listArray  ::  Ix i  =>  (i,i) -> [e] -> Array i e
```

Převody mezi poli a asociačními seznamy:

```
assocs   ::  Ix i  =>  Array i e -> [(i, e)]
array    ::  Ix i  =>  (i,i) -> [(i,e)] -> Array i e
```

Operace s poli

Změny indexování:

```
ixmap  ::  (Ix i, Ix j)  =>  (i, i) -> (i -> j) -> Array j e -> Array i e
```

Příklad: Transpozice matice

```
transpose a  =  ixmap newbounds swap a
                where newbounds = (fst ba, swap (snd ba))
                      ba = bounds a
                      swap (x,y) = (y,x)
```

Modifikace pole:

```
(//)  ::  Ix i  =>  Array i e -> [(i,e)] -> Array i e
```

Příklad: Čtvercová matice s vynulovanou hlavní diagonálou:

```
m // [((i,i), 0) | i <- [1..n]]
```

Poznámka: Modul `Aray.Diff` poskytuje efektivní implementaci operace `(//)`.

Příklad: Skalární součin dvou vektorů

```
scalarproduct :: (Num e, Ix i) => Array i e -> Array i e -> e
scalarproduct u v  =  if  b == bounds v
                        then  sum [ u!j * v!j | j <- range b ]
                        else  error "scalarproduct: nekompatibilni vektory"
                      where  b = bounds u
```

Příklad: Procházení grafu do hloubky

```
import Array

type Vertex = Char
type Graph = Array Vertex [Vertex]
data Tree = Node Vertex [Tree]
type Visited = Array Vertex Bool

dfs :: Graph -> [Vertex] -> [Tree]
dfs gr vs = fst $ dfs1 gr vs (listArray (bounds gr) (repeat False))

dfs1 :: Graph -> [Vertex] -> Visited -> ([Tree],Visited)
dfs1 gr [] vi = ([], vi)
dfs1 gr (v:vs) vi = if vi!v then dfs1 gr vs vi
                            else ( Node v ts : tsu, vi'' )
                    where (tsu, vi'') = dfs1 gr vs vi'
                          (ts, vi')   = dfs1 gr (gr!v) (vi//[(v,True)])
```

Moduly

$\Big[$ `module` *Jméno* $\Big[$ `(`*export*$_1$ `,...,` *export*$_n$`)` $\Big]$ `where` $\Big]$

$\Bigg[$ `import` M_1 $\Big[$*spec*$_1\Big]$

$\vdots$

`import` M_m $\Big[$*spec*$_m\Big]$ $\Bigg]$

$\Big[$ *globální_deklarace* $\Big]$

Není-li uvedena hlavička, doplní se

```
module Main (main) where
```

Nevyskytuje-li se mezi importovanými moduly ($M_1, \ldots M_m$) modul `Prelude`, doplní se

```
import Prelude
```

Modul může exportovat hodnoty, typy a typové konstruktory, typové a konstruktorové třídy, jména modulů.

Právě jeden modul v programu musí být `Main`.

Modul `Main` musí exportovat hodnotu `main :: IO` τ pro nějaký typ τ (obvykle $\tau =$ `()`).

`http://www.haskell.org/onlinereport/modules.html`

Modulární návrh

- Každý modul má mít jednoznačně definovanou roli (dělat jen jednu věc).
- Každá část systému má být obsloužena jedním modulem (sendvičová struktura).
- Moduly mají mít minimální rozhraní – exportovat jen to, co je nutné.
- Moduly mají být malé.

Abstraktní datové typy

Abstraktní datový typ je datový typ, jehož vnitřní reprezentace je skryta za sadou přístupových operací (metod).

Abstraktní datové typy lze zavádět pomocí modulů.

Příklad: Fronta

```
module Fifo (FifoTyp, emptyq, headq, enqueue, dequeue)  where

data  FifoTyp a  =  Q [a] [a]

emptyq ::  FifoTyp a
emptyq  =  Q [] []

enqueue  ::  a -> FifoTyp a -> FifoTyp a
enqueue x (Q h t)  =  Q h (x:t)

headq  ::  FifoTyp a -> a
headq (Q (x:_) _)  =  x
headq (Q [] [])    =  error "headq: prázdná fronta"
headq (Q [] t)     =  headq (Q (reverse t) [])

dequeue  ::  FifoTyp a -> FifoTyp a
dequeue (Q (_:h) t)  =  Q h t
dequeue (Q [] [])    =  error "dequeue: prázdná fronta"
dequeue (Q [] t)     =  dequeue (Q (reverse t) [])
```