

IB015 Úvod do funkcionálního programování

Programování a funkce

Jiří Barnat
Libor Škarvada

Organizace kurzu

Cíle kurzu

- Kurz seznamuje posluchače s funkcionálním programovacím paradigmatem. Toto paradigma má přivést studenty k funkcionálním návykům, které využijí při pozdější tvorbě větších programových celků i v imperativních jazycích.

Schopnosti absolventa

- Rozumí funkcionálnímu výpočetnímu paradigmatu.
- Je schopen dekomponovat výpočetní problém na jednotlivé funkce a tuto schopnost používá při vytváření vlastních kódů, zejména nepoužívá COPY-PASTE techniku programování.
- Má základní znalost programovacího jazyka Haskell.

Předpoklady

- Možné úspěšně absolvovat bez znalosti programování
- Znalost středoškolské matematiky

Navazující předměty

- IB016 Seminář z funkcionálního programování
- IA014 Funkcionální programování

Forma ukončení

- Závěrečný písemný test
- Vnitrosemestrální písemný test (! nelze opakovat)
- Povinné domácí úlohy
- Povinná účast na cvičení

Požadavky na úspěšné ukončení

- Všechny aktivity jsou bodovány
- Možnost získat extra body ve cvičeních
- Nutno získat 48 bodů ze 100+.

`http://haskell.cz/`

Zápisky k IB015

[`http://www.fi.muni.cz/~libor/vyuka/IB015/zapisky.pdf`]

Structure and Interpretation of Computer Programs

[`http://mitpress.mit.edu/sicp/full-text/book/book.html`]

Thompson, Simon. Haskell: the craft of functional programming.

Co znamená programovat?

Programování

- Vytváření a zápis postupu řešení problému s takovou úrovní detailů a přesnosti, aby tento popis mohl být mechanicky vykonáván strojem, zejména počítačem.
- Zápis postupu = zdrojový kód programu.

Programovací jazyk

- Uměle vytvořený jazyk pro přesný a jednoznačný zápis programů člověkem.

Schopnost programovat

- **Mentální schopnost nacházet mechanicky proveditelné postupy za účelem řešení daného problému.**
- Schopnost přesně formulovat postupy v daném programovacím jazyce.

Volba a znalost programovacího jazyka

- Programovacích jazyků je mnoho.
- Volba programovacího jazyka klade omezení na způsob formulace zamýšlených postupů.

Riziko

- Dokumentace k programovacím jazykům jsou snadno dostupné i ve formě tutoriálů, avšak samotné dokonalé poznání programovacího jazyka nedělá dokonalého programátora.

Riziko

- Dokumentace k programovacím jazykům jsou snadno dostupné i ve formě tutoriálů, avšak samotné dokonalé poznání programovacího jazyka nedělá dokonalého programátora.

Nedokonalé vs. dokonalé



Riziko

- Dokumentace k programovacím jazykům jsou snadno dostupné i ve formě tutoriálů, avšak samotné dokonalé poznání programovacího jazyka nedělá dokonalého programátora.

Nedokonalé vs. dokonalé



Klasifikace

- Imperativní — C/C++, Java, Perl, php, ...
- Funkcionální – **Haskell**, OCaml, Erlang, Lisp, ...
- Logické – Prolog, ...
- Kombinované – C#, Scala, ...
- ...

Jakým jazykem mluví počítač?

- Strojový kód. Program ve strojovém kódu je posloupnost čísel.
- Pro spuštění programu je potřeba provést překlad zdrojového kódu programu do strojového kódu procesoru.
- Překlad se realizuje pomocí **překladače** nebo **interpretru**.
- Pro každý programovací jazyk je potřeba jiný překladač/interpreter.

Překladač

- Pro soubor se zdrojovým kódem programu vytvoří soubor obsahující popis programu ve strojovém kódu.
- Výsledný soubor je spustitelný.
- Pracuje se soubory.

Interpret

- Pro daný výraz / příkaz vytvoří odpovídající překlad do strojového kódu a ihned jej provede.
- Nevytváří výsledný spustitelný soubor.
- Často má možnost pracovat interaktivně.
- Pracuje s jednotlivými příkazy/výrazy.

Programovací jazyk Haskell

- Překladač – ghc.
- Interaktivní interpret – hugs, ghci.
- Neinteraktivní interpretace – runhugs.

Překladače programovacího jazyka C/C++

- GNU C++ Compiler (g++, gcc)
- Intel C++ Compiler
- Microsoft Visual C++ Compiler

Funkce

Funkce v programování

- Funkce je předpis jak z nějakého vstupu vytvořit výstup.
- Transformace vstupů na výstupy musí být jednoznačná.

Příklady funkcí

- $f(x) = x(x+2)$
- objem kvadru $a b c = a*b*c$
- ...

Typ funkce

- Vymezení objektů, se kterými daná funkce pracuje a které vrací na výstup, je součástí definice funkce. Mluvíme o tzv. **typu funkce**.

Příklady

- Funkce, která otočí obrázek o 90 stupňů směru vpravo.
`rotate90r :: Obrázek -> Obrázek`
- Objem kváдру.
`objemkvadru :: Číslo × Číslo × Číslo -> Číslo`
- Počet hran polygonu.
`hranypolygonu :: Polygon -> Celé_číslo`

Předpoklady

`rotate90r :: Obrázek -> Obrázek`

`hranypolygonu :: Obrázek -> Celé_číslo`

`△ :: Obrázek`

Aplikace funkcí

`rotate90r △ = ▷`

`hranypolygonu △ = 3`

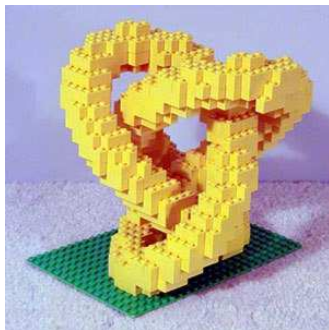
Pozorování

- Složitější úkony lze realizovat pomocí jednodušších operací.
- Složitější funkce lze definovat **složením** jednodušších.

Pozorování

- Složitější úkony lze realizovat pomocí jednodušších operací.
- Složitější funkce lze definovat **složením** jednodušších.

Skládání – cesta ke složitějším objektům a funkcím



Operátor .

- $f1 (f2 x) = (f1 . f2) x$
- Čteme jako „f1 po f2“.

Příklad

- Mějme funkci `double`, která vezme obrázek a vytvoří nový obrázek zkopírováním původního obrázku dvakrát vedle sebe.



`double :: Obrázek -> Obrázek`

`double`  = 

- Novou funkci `rotate_and_double` můžeme definovat takto:

`rotate_and_double :: Obrázek -> Obrázek`

`rotate_and_double x = (double . rotate90r) x`

`rotate_and_double`  = 

Skládání funkcí

- Je možné zapsat bez explicitního uvedení parametru.
- Tj. definici

```
rotate_and_double x = (double.rotate90r) x
```

lze zapsat také jako

```
rotate_and_double = double.rotate90r
```

POZOR na prioritu vyhodnocování v Haskellu

- Aplikace funkce na parametry má nejvyšší prioritu.



```
double.rotate90r △ = double.(rotate90r △) ~ ERROR
```


- Závorky kolem výrazu `double.rotate90r` jsou při aplikaci na hodnotu `△` nutné.


`(rotate_and_double . rotate_and_double)` \triangle =



`((double . double) . double)` \triangle =



`(double . hranypolygonu)` \triangle =


`(rotate_and_double . rotate_and_double)`  = 



`((double . double) . double)`  =



`(double . hranypolygonu)`  =


`(rotate_and_double . rotate_and_double)`  = 

`((double . double) . double)`  = 

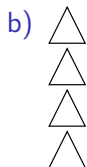
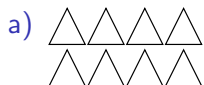
`(double . hranypolygonu)`  =

`(rotate_and_double . rotate_and_double)`  = 

`((double . double) . double)`  = 

`(double . hranypolygonu)`  = **ERROR**

Jak lze pomocí `double`, `rotate90r` a \triangle vyrobit následující?



Typová signatura:

`rotate_and_double` :: Obrázek ->Obrázek

Jméno funkce

`rotate_and_double` x = (double.rotate) x

Tělo funkce

`rotate_and_double` x = (double.rotate) x

Definice funkce

`rotate_and_double` x = (double.rotate) x

Formální parametr

rotate_and_double x = (double.rotate) x

Aktuální parametr

rotate_and_double △

Výraz

rotate_and_double △

Podvýraz

rotate_and_double (rotate_and_double △)

Funkcionální programování v Haskellu

Funkcionální výpočetní paradigma

- program = výraz + definice funkcí
- výpočet = úprava (zjednodušování) výrazu
- výsledek = hodnota (nezjednodušitelný tvar výrazu)

Příklad programu

- definice funkcí

```
square x = x * x
```

```
pyth a b = square a + square b
```

- výraz

```
pyth 3 4
```

Program

- definice funkcí

```
square x = x * x
```

```
pyth a b = square a + square b
```

- výraz

```
pyth 3 4
```

Výpočet

pyth 3 4 \rightsquigarrow square 3 + square 4 \rightsquigarrow 3 * 3 + square 4 \rightsquigarrow
 \rightsquigarrow 3 * 3 + 4 * 4 \rightsquigarrow 9 + 4 * 4 \rightsquigarrow 9 + 16 \rightsquigarrow
 \rightsquigarrow 25

Lokální definice

- Definují symboly (funkce, konstanty) pro použití v jednom výrazu, vně tohoto výrazu jsou tyto symboly nedefinované.
- Lokální definice mají vyšší prioritu než globální definice.

V Haskellu pomocí let ... in a where

- let definice in výraz

```
let fcube x = x * x * x in fcube 12
```

```
let fcube x = x * x * x in let c = 12 in fcube c
```

```
let fcube x = x * x * x; c = 12 in fcube c
```

- výraz where definice

```
fcube 12 where fcube x = x * x * x
```

```
fcube c where fcube x = x * x * x; c = 12
```

Čísla

- `Integer` – libovolně velká celá čísla
- `Int` – celá čísla do velikosti slova procesoru
- `Float` – reálná čísla
- `Fractional` – racionální čísla

Znaky a řetězce

- `Char` – znak, příklady hodnot: `'a'`, `'2'`, `'>'`
- `String` – řetězec, například: `"Toto je řetězec."`
- `String` je totéž co `[Char]`

Pravdivostní hodnoty

- `Bool`
- Typ `Bool` má pouze 2 hodnoty: `True` a `False`

Příklad

- Definujte funkci `jedna_nebo_dva`, která vrací `True` pokud dostane na vstupu číslo 1 nebo 2, jinak vrací `False`.

```
jedna_nebo_dva :: Integer -> Bool
jedna_nebo_dva 1 = True
jedna_nebo_dva 2 = True
jedna_nebo_dva _ = False
```

Víceřádkové definice funkcí

- Na místě formálních parametrů se použijí tzv. vzory.
- Použije se první vzor, který vyhovuje.
- Symbol `_` vyhovuje libovolnému parametru.
- Lze použít pro větvení výpočtu.

Podmíněný výraz

- *if* *podmínka* **then** *výraz1* **else** *výraz2*
- *podmínka* – výraz, který se vyhodnotí na hodnotu typu `Bool`
- *výraz1* se vyhodnotí pokud se podmínka vyhodnotí na hodnotu `True`, *výraz2* se vyhodnotí, pokud se podmínka vyhodnotí na hodnotu `False`.
- Výrazy *výraz1* a *výraz2* musejí být stejného typu.

Test na rovnost

- Pro dotaz na rovnost používáme symbol `==`.
- `3 == 4` \rightsquigarrow `False`
- `3 = 4` \rightsquigarrow **Error**

Možnosti zápisu binárních funkcí

- Infixový zápis binárních funkcí: $3+4$, $4*5$
- Prefixový zápis binárních funkcí: $(+) 3 4$, $(*) 4 5$

Volání funkce a parametry

- Jméno funkce a použité parametry jsou odděleny mezerou, pokud je některý z parametrů výraz, který je sám o sobě aplikace funkce na argumenty, je třeba celý tento výraz ozávkovat.
- $(*) 3 4 + 5 \rightsquigarrow 17$
- $(*) 3 + 4 5 \rightsquigarrow$ **Error**
- $(*) 3 (+) 4 5 \rightsquigarrow$ **Error**
- $(*) 3 ((+) 4 5) \rightsquigarrow 27$

Vstup výstupní operace

- Operace, které čtou a zapisují data z/do souborů, či terminálu.
- Program v Haskellu je reprezentován výrazem `main`.
- Program je sekvence vstup-výstupních akcí.
- Funkcionální princip vstup/výstupních akcí je složitý, bude vysvětlen v závěru kurzu.

do notace programu v Haskellu

- ```
main = do putStr "Zadej celé číslo"
 x <- getLine
 print (1+ read x::Int)
```
- Textové zarovnání je důležité!

## Zadání

- Napište a spusťte program v Haskellu, který bude řešit dělitelnost dvou čísel, tj. zeptá se uživatele na dělence, načte ho, pak se zeptá na dělitele, kterého také načte, a sdělí uživateli, zda je zadaný dělenec dělitelný beze zbytku zadaným dělitelem.

# IB015 Úvod do funkcionálního programování

## Seznamy, Typy a Rekurze

Jiří Barnat  
Libor Škarvada

## Uspořádané n-tice a seznamy

## Pozorování

- Programy pro své fungování potřebují různé informace – **data**.
- Data jsou vstupní hodnoty, výstupní hodnoty, mezivýsledky výpočtů, parametry funkcí, atd.

## Programování a data

- Data je třeba uchovávat tak, aby je bylo možné zpracovat mechanicky/strojově.
- Tvorba jednoznačného popisu struktury a způsobu uložení dat je nedílná součást procesu programování.
- Pro uchování informací používáme různé **datové struktury**.

## Dekompozice dat

- Veškerá data použitá v programu je třeba vystavět ze základních datových elementů podle definovaných pravidel.
- Existují striktní pravidla pro dekompozici dat, my si však v rámci IB015 vystačíme s intuicí.

## Základní datové elementy

- Čísla, Znaky, Pravdivostní hodnoty

## Základní způsoby kompozice dat

- Uspořádané n-tice
- **Seznamy**

## Co to je

- Pevně daný počet nějakých hodnot v pevně daném pořadí.
- Prvek kartézského součinu nosných množin.

## Příklady

- Datum: (11, "březen", 1977) .
- Přihlašovací údaje: ("xbarnat", "majen10cm")
- Pozice pixelu v rastrovém obrázku: (x, y) ,  
všimněme si, že (12,43)  $\neq$  (43,12) .

## Kdy se má použít

- **Počet prvků v n-tici je znám předem**,  
tj. v okamžiku psaní zdrojového kódu.
- Počet prvků v n-tici je malý (hodnota n je malá).

## Co to je

- Posloupnost hodnot stejného charakteru (stejného typu).
- Posloupnost může být prázdná, konečná i nekonečná.
- Každý prvek v seznamu je na nějaké (unikátní) pozici.

## Příklady

- Seznam čísel: [12,43,-3,15,29]
- Nekonečný seznam: [1,2,3,4,5,6,...]
- Seznam uspořádaných dvojic:  
[("Fero",12), ("Nero",7), ("Pero",5)]
- Prázdný seznam: []

## Kdy se má použít

- **Data vznikají nebo se zpracovávají postupně.**
- Počet prvků použitých programem není předem znám.

## Aplikace – Diář squashových partnerů.

- Program pro správu kontaktů na různé squashové hráče.
- Hlavní datová struktura je seznam kontaktů.

## Datová dekompozice

- Seznam kontaktů  
[kontakt1, kontakt2, kontakt3, ..., kontakt315]
- Kontakt je uspořádaná trojice  
(Prezdivka, Telefon, Adresa)
- Adresa je uspořádaná pětice  
(Jmeno, Prijmeni, Ulice, Cislo Popisne, Mesto)
- Prezdivka, Jmeno, Prijmeno, Ulice, Mesto jsou seznamy znaků
- Telefon, Cislo Popisne jsou čísla

## Hodnoty a Typy

**Typ není váš nepřítel**

## Typ není váš nepřítel



## Co je to typ

- Označení množiny všech hodnot dané kvality.
- Komunikační prostředek napomáhající správnému skládání programů z jednotlivých funkcí.

## K čemu se používají typy

- Každá hodnota, nebo výraz má svůj typ.
- Definice typové signatury funkcí.
- Kontrola logické konzistence programu v době překladu.
- Popis způsobu kompozice složených datových struktur.  
(Typy se komponují stejně jako data.)

## Základní datové typy

- `Int`, `Integer`, `Fractional`, `Float`, `Char`, `Bool`

## Složené typy

- Uspořádané n-tice:

`(Bool, Int)`

- Seznamy:

`[Int]`, `[Char]`, `[[Char]]`

`[Char] ≡ String`

## Funkcionální typy

- `Integer -> Bool`, `Float -> Float -> Float`

## Konstrukce

- Jsou-li  $\sigma$  a  $\tau$  nějaké typy, tak  $\sigma \rightarrow \tau$  je typ všech funkcí s parametrem typu  $\sigma$  a funkční hodnotou typu  $\tau$ .

## Typ n-árních funkcí

- Jsou-li  $\sigma_1, \sigma_2, \sigma_3 \dots \sigma_n$  a  $\tau$  nějaké typy, tak

$$\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau$$

je typ všech funkcí s prvním parametrem typu  $\sigma_1$ , druhým parametrem typu  $\sigma_2, \dots$  a funkční hodnotou typu  $\tau$ .

## Terminologie

- **Arita funkce** označuje počet parametrů funkce.
- Konstanty, unární, binární, ternární funkce.
- Nulární funkce ( $n=0$ ) jsou konstanty daného typu.

## Pozorování

- Typ výrazu, který je úplná aplikace funkce na parametry, lze odvodit z typu použité funkce **bez nutnosti výpočtu výsledné hodnoty**.

## Příklad

```
odd :: Integer -> Bool
27 :: Integer
odd 27 :: Bool
```

## Pozorování

- Některé funkce nepotřebují znát konkrétní typy formálních parametrů, pouze jejich strukturu.
- Místo konkrétního typu se použije **typová proměnná**.
- Při aplikaci funkce na konkrétní parametry, se za typovou proměnnou dosadí typ, který odpovídá použitému parametru. (Typová proměnná se specializuje.)
- **POZOR! Typová proměnná zastupuje i složené typy.**

## Příklad

```
fst :: (a,b) -> a
(not, "Coze?") :: (Bool -> Bool, [Char])
fst (not , "Coze?") :: Bool -> Bool
```

## Pozorování

- Některé funkce nevyžadují konkrétní typ, ale zároveň nedovolují použití libovolného typu, proto je třeba specializaci typové proměnné omezit na vybranou podtřídu typů.

## Základní typové třídy

- `Integral` – celočíselné
- `Num` – numerické
- `Ord` – uspořadatelné
- `Eq` – porovnatelné na rovnost

## Příklady typů s omezením specializace typové proměnné

```
odd :: Integral a => a -> Bool
```

```
(+) :: Num a => a -> a -> a
```

## Uspořádané n-tice a seznamy v Haskellu

## Zápis uspořádaných n-tic

- Přirozený, pomocí závorek a čárek.
- Příklady zápisu uspořádaných n-tic v Haskellu:

```
(12,15)
```

```
(2,3,'a',5,6)
```

```
("Fiii","jo", 350, "tisíc", '!')
```

```
((1,1),(2,2),(3,3))
```

## Krajní případy

- Jednotice se nepoužívají.
- Nultice: `()`

## Datové konstruktory

- $(,,\dots,)$  – datový konstruktor uspořádané n-tice
- $(,)$  – datový konstruktor uspořádané dvojice

$$(,) :: a \rightarrow b \rightarrow (a,b)$$
$$(,) x y = (x,y)$$

## Projekce

- `fst` , `snd` – projekce na první a druhou složku

$$\text{fst} :: (a,b) \rightarrow a$$
$$\text{fst } (x,y) = x$$
$$\text{snd} :: (a,b) \rightarrow b$$
$$\text{snd } (x,y) = y$$

## Zápis seznamů

- V hranatých závorkách uzavřená posloupnost prvků oddělených čárkou.
- Seznam znaků též jako řetězec (text v uvozovkách).

## Příklady

```
[3,3,3,3]
```

```
[[1], [1,2], [1,2,3]]
```

```
[]
```

```
"ahoj" = ['a','h','o','j']
```

```
"toto je také seznam"
```

```
[or, or, or, and]
```

## Datové konstruktory

- Prázdný seznam: `[]`  
`[] :: [a]`
- Operátor připojení prvku na začátek seznamu: `(:)`  
`(:) :: a -> [a] -> [a]`

## Příklady

- Správné použití  
`(:) 3 [3,3,3] ~> [3,3,3,3]`  
`1:2:3:[] ~> [1,2,3]`  
`4:[4,4,4,4] ~> [4,4,4,4,4]`  
`'A':"hoj" ~> "Ahoj"`
- Nesprávné použití  
`[2] : [3,4,5] ~> ERROR`  
`[2,3,4] : 5 ~> ERROR`  
`'A' : [1,2,3] ~> ERROR`

## Funkce pro spojení seznamů

- Seznamy **stejného typu** lze spojit pomocí funkce `(++)`  
`(++) :: [a] -> [a] -> [a]`

## Příklady

- Správné použití

`(++) "Ahoj " "světe!" ~> "Ahoj světe!"`

`"Ahoj" ++ " " ++ "světe!" ~> "Ahoj světe!"`

`[1,2,3] ++ [4,5,6] ~> [1,2,3,4,5,6]`

- Nesprávné použití

`2 ++ [3,4,5] ~> ERROR`

`[2,3,4] ++ 5 ~> ERROR`

`[2,3] ++ "text" ~> ERROR`

## Datové konstruktory ve víceřádkových definicích

- Fungují jako vzory na levých stranách definice.
- Mapují se vždy na nejvnějšnější výskyt.

## Příklady

- Funkce `null` aplikovaná na nějaký seznam, vrací `True` pokud je seznam prázdný a `False` pokud je neprázdný.

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

- Funkce `snd` aplikovaná na uspořádanou dvojici, vrací druhý prvek dvojice.

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

# Rekurze

## Co je to rekurze

- Definice funkce, nebo datové struktury, s využitím sebe sama.

## Význam v programování

- Umožňuje konečně dlouhý zápis definice funkce, která je definována pro nekonečně mnoho parametrů.
- V čistě funkcionálním jazyce nahrazuje cykly známé z imperativních programovacích jazyků.

## Příklad

- Funkci `length`, která při aplikaci na seznam vrátí jeho délku, je nutné definovat rekurzivně.

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

```
length [] = 0
```

```
length (.:s) = 1 + length s
```

## Příklad výpočtu rekurzivní definice

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

```
length [6,7,8,9] ~> 1 + length [7,8,9]
 ~> 1 + (1 + length [8,9])
 ~> 1 + (1 + (1 + length [9]))
 ~> 1 + (1 + (1 + (1 + length [])))

 ~> 1 + (1 + (1 + (1 + 0)))
 ~> 1 + (1 + (1 + 1))
 ~> 1 + (1 + 2)
 ~> 1 + 3
 ~> 4
```

## Číselné funkce

```
factorial :: Integer -> Integer

factorial 0 = 1
factorial x = x * factorial (x-1)
```

## Nekonečné opakování (teoreticky)

```
main :: IO()

main = do putStr "Vlož text:"
 x <- getLine
 putStrLn ("Zadal jsi:" ++ x)
 main
```

## Pozorování

- Použití rekurze je možné pouze tehdy, je-li podproblém, na nějž se rekurzivně aplikuje dané řešení, **stejného typu**.

## Slovní úloha

- Na Jiříkovu narozeninovou párty přišlo 7 kamarádů a přineslo mimo jiné jeden narozeninový dort. Jiřík nebyl lakomý, rozdělil dort rovnoměrně mezi všechny přítomné. Jakou k tomu použil rekurzivní funkci? Jaký je typ této rekurzivní funkce?



## Pozorování

- Použití rekurze je možné pouze tehdy, je-li podproblém, na nějž se rekurzivně aplikuje dané řešení, **stejného typu**.

## Slovní úloha

- Na Jiříkovu narozeninovou párty přišlo 7 kamarádů a přineslo mimo jiné jeden narozeninový dort. Jiřík nebyl lakomý, rozdělil dort rovnoměrně mezi všechny přítomné. Jakou k tomu použil rekurzivní funkci? Jaký je typ této rekurzivní funkce?

## Řešení

```
dort :: [KouskyDortu] -> [KouskyDortu]
dort s = if (length s >= 8)
 then s
 else dort (map (/2) s ++ map (/2) s)
```



## Práce se seznamy v Haskellu

# head, tail, init, last

První prvek seznamu

```
head :: [a] -> a
```

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

Seznam bez prvního prvku

```
tail :: [a] -> [a]
```

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

Poslední prvek seznamu

```
last :: [a] -> a
```

```
last (x:[]) = x
```

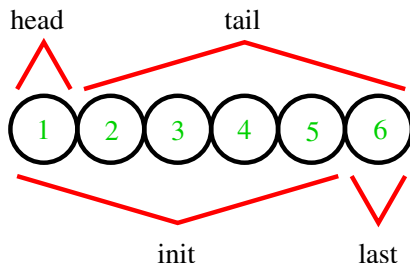
```
last (_:s) = last s
```

Seznam bez posledního prvku

```
init :: [a] -> [a]
```

```
init (x_:[]) = []
```

```
init (x:s) = [x] ++ init s
```



## Test na prázdný seznam

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

## Délka seznamu

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

## N-tý prvek seznamu

```
(!!) :: [a] -> Int -> a
(x:_:) !! 0 = x
(:_:s) !! k = s !! (k-1)
```

Prvních n prvků seznamu

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

Seznam bez prvních n prvků;

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

## Poznámka

- Při infixovém použití binární funkce klesá její priorita!

**$x : \text{take } (n-1) s = x : (\text{take } (n-1) s)$**

## Spojení seznamů v seznamu

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

## Vynechání prvků nesplňujících danou podmínku

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

## Vytvoření seznamu n-násobným kopírováním daného prvku

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

Vynechání prvků seznamu od prvního, který nesplňuje podmínku

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

```
takeWhile _ [] = []
```

```
takeWhile p (x:s) = if (p x) then x : takeWhile p s
 else []
```

Vynechání prvků seznamu po první, který nesplňuje podmínku

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

```
dropWhile _ [] = []
```

```
dropWhile p (x:s) = if (p x) then dropWhile p s
 else x:s
```

Aplikace funkce na všechny prvky seznamu

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

```
map _ [] = []
```

```
map f (x:s) = f x : map f s
```

Spojení dvou seznamů do seznamu uspořádaných dvojic

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

Rozdělení seznamu dvojic na dvojici seznamů

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

Výpočet aplikace binární funkce na seznamy argumentů

```
zipWith :: (a->b->c)->[a]->[b]->[c]
```

```
zipWith _ _ [] = []
```

```
zipWith _ [] _ = []
```

```
zipWith f (x:s) (y:t) = f x y : zipWith f s t
```

Pozorování

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

## Technická cvičení

- Vyzkoušejte si všechny odpřednášené funkce na modelových seznamech v prostředí interpretru jazyka Haskell.

## Mentální cvičení

- Napište program, který pomocí principu rekurze a s využitím odpřednášených operací na seznamech vypočítá seznam obsahující čísla od 1 do 1024. Snažte se o to, aby hloubka rekurze byla co nejmenší.
- Jsou-li vám známy cykly s pevným počtem opakování z nějakého imperativního programovacího jazyka, popřemýšlejte o obecném postupu, jak nahradit tyto cykly voláním rekurzivní funkce.

# IB015 Úvod do funkcionálního programování

## Funkce vyšších řádů a $\lambda$ -funkce

Jiří Barnat  
Libor Škarvada

## Funkce vyšších řádů

## Definice

- Funkce, je považována za funkci vyššího řádu, pokud alespoň jeden z jejich argumentů, které si funkce má, nebo výsledek, který funkce vrátí, je opět funkce.
- Funkce vyššího řádu se též označují jako funkcionály.

## Příklady

$(.) :: (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow c \rightarrow b$

$\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$

## Pozorování

- Každou funkci, která má alespoň 2 argumenty, lze chápat jako funkci vyššího řádu.

## Příklad

- Funkci

$(*) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$

lze číst jako

$(*) :: \text{Num } a \Rightarrow a \rightarrow (a \rightarrow a)$

- Funkci, která bere dva číselné argumenty typu  $a$  a vrací hodnotu typu  $a$ , lze také chápat jako funkci, která bere hodnotu číselného typu  $a$  a vrací hodnotu typu  $a \rightarrow a$ .

## Pozorování

- Chápeme-li  $n$ -ární ( $n \geq 2$ ) funkce jako funkce vyššího řádu, lze tyto funkce tzv. **částečně aplikovat**, tj. vyhodnotit je i pro neúplný výčet argumentů.

## Příklad

- Uvažme funkci násobení
  - $(*) :: \text{Num } a \Rightarrow a \rightarrow (a \rightarrow a)$
  - $(*) \ x \ y = x*y$
- Výsledkem aplikace  $(*)$  na hodnotu  $3$  je funkce.
  - $(*) \ 3 :: \text{Num } a \Rightarrow a \rightarrow a$
- Tuto funkci je možné označit, a posléze použít.
  - $f = (*) \ 3$
  - $f :: \text{Num } a \Rightarrow a \rightarrow a$
  - $f \ 4 \rightsquigarrow ((*) \ 3) \ 4 \rightsquigarrow 12$

## Připomenutí

- Typový konstruktor  $\rightarrow$  je binární.
- Typový konstruktor  $\rightarrow$  se používá pouze infixově.

## Implicitní závorkování

- Typový konstruktor  $\rightarrow$  implicitně sdružuje zprava

$f :: a_1 \rightarrow (a_2 \rightarrow (a_3 \rightarrow \dots \rightarrow (a_{n-1} \rightarrow (a_n \rightarrow a)) \dots ))$

- Aplikace funkce na argumenty implicitně sdružuje zleva

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

## Pozorování

- Libovolnou  $n$ -ární funkci lze také chápat jako  $k$ -ární, kde  $k$  nabývá hodnot od 1 do  $n$ .

**S každou aplikací ubude jeden výskyt  $\rightarrow$  v typu výrazu**

(+)            :: Num a => a -> a -> a

(+) 2         :: Num a => a -> a

((+) 2) 3    :: Num a => a

**S každou aplikací ubude jeden výskyt  $\rightarrow$  v typu výrazu**

$(+)$   $:: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$

$(+) 2$   $:: \text{Num } a \Rightarrow a \rightarrow a$

$((+) 2) 3$   $:: \text{Num } a \Rightarrow a$

**Specializací typové proměnné však mohou  $\rightarrow$  přibýt.**

- Vezměme například funkci identity

$\text{id} :: a \rightarrow a$

$\text{id } x = x$

- Při aplikaci  $\text{id}$  na  $(+)$  ubude  $\rightarrow$  z typu  $\text{id}$ , tj.

$\text{id } (+) :: a$

- Avšak po specializaci typové proměnné  $a$  na typ funkce  $(+)$

$\text{id } (+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$

## Částečná aplikace

- Má-li funkce více formálních parametrů částečná aplikace probíhá vždy od parametru nejvíce vlevo.

## Problém

- Funkci nelze přímo částečně aplikovat na jiný než první parametr.

## Funkce `flip`

- Při aplikaci na binární funkci tuto funkci modifikuje tak, aby své dva argumenty přijímala v obráceném pořadí.

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

- Funkci `flip` je třeba chápat jako **modifikátor funkce**, ne jako prohazovačku parametrů!

## Příklady

```
(-) 3 4 ~> -1
```

```
flip (-) 3 4 ~> 1
```

```
(-) flip 3 4 ~> ERROR
```

## Příklad

- Pomocí částečné aplikace funkce `(-)` definujte novou funkci `minus4`, která při aplikaci na číselnou hodnotu vrátí hodnotu o 4 menší.

## Řešení

- Definice s využitím částečné aplikace, bez formálních parametrů.

```
minus4 :: Num a => a -> a
minus4 = flip (-) 4
```

- Standardní definice téhož s využitím formálních parametrů.

```
minus4 :: Num a => a -> a
minus4 x = (-) x 4
```

## Pozorování

- Funkce vyššího řádu a částečná aplikace souvisí s násobným použitím funkcionálního typového konstrukturu  $\rightarrow$  .
- Chceme-li zabránit částečné aplikaci, musíme definovat funkci tak, aby v jejím typu byl pouze jeden výskyt  $\rightarrow$  .
- Pokud chceme funkci předat více argumentů najednou, předáme je jako uspořádanou n-tici.

## Příklad

- Všimněte si rozdílu v typech a definicích následujících funkcí.

```
krat :: Num a => a -> a -> a
```

```
krat x y = x * y
```

```
krat1 :: Num a => (a,a) -> a
```

```
krat1 (x,y) = x * y
```

## **Funkce** `curry` a `uncurry`

- Předdefinované funkce pro změnu řádu binárních funkcí.

### **curry**

- Modifikuje funkci tak, že tato funkce místo uspořádané dvojice hodnot přijímá dva samostatné parametry.

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

### **uncurry**

- Modifikuje funkci tak, že tato funkce místo dvou samostatných parametrů přijímá uspořádanou dvojici hodnot.

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

## Příklad

- Mějme definovány následující funkce

```
krat :: Num a => a -> a -> a
```

```
krat x y = x * y
```

```
krat1 :: Num a => (a,a) -> a
```

```
krat1 (x,y) = x * y
```

- Uveďte alternativní definici funkce `krat` pomocí funkce `krat1` a obráceně.

## Příklad

- Mějme definovány následující funkce

```
krat :: Num a => a -> a -> a
```

```
krat x y = x * y
```

```
krat1 :: Num a => (a,a) -> a
```

```
krat1 (x,y) = x * y
```

- Uved'te alternativní definici funkce `krat` pomocí funkce `krat1` a obráceně.

## Řešení

```
krat = curry krat1
```

```
krat1 = uncurry krat
```

## Co je to

- Pro každý **binární operátor** je možné definovat funkci, jež odpovídá částečné aplikaci funkce na první formální parametr a funkci, jež odpovídá částečné aplikaci funkce na druhý formální parametr. Těmto funkcím se říká **operátorové sekce**.

## Operátorové sekce

- Předpokládejme binární operátor  $\oplus$  a hodnoty  $p$  a  $q$   
 $\oplus :: a \rightarrow b \rightarrow c$   
 $p :: a$   
 $q :: b$
- Částečnou aplikaci na první argument zapíšeme jako  $(p\oplus)$   
 $(p\oplus) = (\oplus) p$
- Částečnou aplikaci na druhý argument zapíšeme jako  $(\oplus q)$   
 $(\oplus q) = \text{flip } (\oplus) q$

## Příklad1

- Jednoduché použití operátorových sekcí.

`(*2) 34`  $\rightsquigarrow$  68

`(++".") ['V', 'ě', 't', 'a']`  $\rightsquigarrow$  "Věta."

## Příklad2

- Odvoďte typ následujících funkcí, popište jejich význam:

`(1.0/)`

`('mod' 3)`

`(!!0)`

`(>0)`

## Příklad1

- Jednoduché použití operátorových sekcí.

`(*2) 34 ~> 68`

`(++".") ['V', 'ě', 't', 'a'] ~> "Věta."`

## Příklad2

- Odvoďte typ následujících funkcí, popište jejich význam:

`(1.0/) :: Fractional a => a -> a`

`('mod' 3) :: Integral a => a -> a`

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

`(>0) :: (Num a, Ord a) => a -> Bool`

## Skládání funkcí

- Základní operátor funkcionálního programování

$$(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$
$$(\cdot) f g x = f ( g x )$$

## Pozorování

- Operátor pro skládání funkcí lze chápat také jako binární.

$$(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$

- Pro operátor  $(\cdot)$  je možné definovat operátorovou sekci.
- Použití operátorové sekce pro operátor  $(\cdot)$  na jiné než jednoduché funkce je však velmi matoucí a v praxi se nepoužívá.

$$(\cdot(\cdot)) :: (((a \rightarrow b) \rightarrow a \rightarrow c) \rightarrow d) \rightarrow (b \rightarrow c) \rightarrow d$$

## Skládání funkcí

- Základní operátor funkcionálního programování

$$(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$
$$(\cdot) f g x = f ( g x )$$

## Pozorování

- Operátor pro skládání funkcí lze chápat také jako binární.

$$(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$

- Pro operátor  $(\cdot)$  je možné definovat operátorovou sekci.
- Použití operátorové sekce pro operátor  $(\cdot)$  na jiné než jednoduché funkce je však velmi matoucí a v praxi se nepoužívá.


$$(\cdot(\cdot)) :: (((a \rightarrow b) \rightarrow a \rightarrow c) \rightarrow d) \rightarrow (b \rightarrow c) \rightarrow d$$

## Pozorování

- Funkce `flip`, `curry`, `uncurry`, `(.)` a operátorové sekce používáme k vytvoření tzv. **bezparametrové** definice funkce.

## Připomenutí bezparametrové definice

- Funkci `f` definovanou s použitím formálního parametru

```
f x = (not.odd) x
```

- Je možné definovat i bez použití formálního parametru

```
f = (not.odd)
```

## Příklad

```
f x = (3*x)^7
```

```
f x = flip (^) 7 (3*x)
```

```
f x = flip (^) 7 ((* 3) x)
```

```
f x = (.) (flip (^) 7) ((* 3) x)
```

```
f x = (.) (flip (^) 7) (3*) x
```

```
f = (.) (flip (^) 7) (3*)
```

## Nepojmenované funkce

## Motivace

- Při standardní definici funkce musíme tuto funkci pojmenovat.
- Mnohé funkce, často jednoduché, použijeme jednorázově.
- Funkce s jednorázovým použitím je zbytečné pojmenovávat.

## Příklad

- Globální definice a použití jednoduché funkce

```
f x = x*x + 1
```

```
map f [1,2,3,4,5] ~> [2,5,10,17,26]
```

- Lokální definice a použití jednoduché funkce

```
let f x = x*x + 1 in map f [1,2,3,4,5] ~> [2,5,10,17,26]
```

```
map f [1,2,3,4,5] where f x = x*x + 1 ~> [2,5,10,17,26]
```

## Co to je

- Definice funkce v místě jejího použití bez uvedení jejího jména.
- Příklad:

`map (\x -> x*x+1) [1,2,3,4,5]  $\rightsquigarrow$  [2,5,10,17,26]`

## Původ a všeobecné označení

- Myšlenka a teoretický základ pochází z Lambda kalkulu, proto se též nepojmenováním funkcím říká **lambda funkce**.
- Principu vytváření lambda funkcí, se říká lambda abstrakce.

## Pozorování

- Koncept lambda funkcí se vyskytuje v mnoha imperativních programovacích jazycích, např. C#, SCALA, PHP, ...

## Lambda abstrakce

- Uvažme výraz  $M$ , který představuje tělo funkce.

$$M \equiv x * x + 1$$

- Z těla funkce vytvoříme funkci použitím lambda abstrakce.

$$\lambda x.M \equiv \lambda x.(x * x + 1)$$

- Při aplikaci lambda funkce  $\lambda x.M$  na výraz  $N$ , se všechny volné výskyty formálního parametru  $x$  v  $M$  nahradí výrazem  $N$ . Výskyt proměnné  $x$  je označován jako volný, pokud není v rozsahu žádné lambda abstrakce.

$$\lambda x.M N \equiv \lambda x.(x * x + 1) N = N * N + 1$$

## Příklady

- $(\lambda x.x * x + 1) 3 = 3 * 3 + 1 = 10$ .
- $(\lambda x.x + (\lambda x.x + x) x) 5 = 5 + (\lambda x.x + x) 5 = 5 + (5 + 5) = 15$ .
- $(\lambda x.34) 3 = 34$

## Definice a použití nepojmenované funkce

- $\lambda$  se špatně píše na klávesnici.
- V programovacím jazyce *Haskell* je lambda abstrakce zapsána pomocí zpětného lomítka a šipky:

```
\ formální parametry -> tělo funkce
```

## Příklady

```
(\ x -> x*x + x*x) 3 ~> ... ~> 18
```

```
(\ x -> x False) not ~> not False ~> True
```

## Nepojmenovanou funkci je možné pojmenovat

```
f = (\ x -> x*x + x*x)
```

```
f 3 ~> ... ~> 18
```

```
f 3 ~> (\ x y -> x * x + y) 3 ~> ... ~> 18
```

## Pozorování

- Vnořená lambda abstrakce vytváří funkce vyšších řádů.
- $(\lambda x.(\lambda y.(x - y))) 3 4 = (\lambda y.(3 - y)) 4 = 3 - 4 = -1$

## Zápis v Haskellu

- Odvozený přímo z vícenásobné aplikace lambda abstrakce  
 $\lambda x \rightarrow (\lambda y \rightarrow x - y)$   
 $(\lambda x \rightarrow (\lambda y \rightarrow x - y)) 3 4 \rightsquigarrow \dots \rightsquigarrow -1$
- Zkrácený z důvodu čitelnosti kódu a pohodlí programátorů  
 $\lambda x y \rightarrow x - y$   
 $(\lambda x y \rightarrow x - y) 3 4 \rightsquigarrow \dots \rightsquigarrow -1$

## Nepojmenované funkce a jejich typy

- Obecně platí stejná pravidla jako pro pojmenované funkce.
- Název funkce (ani jeho neexistence) nemá vliv na typ funkce.

## Efekt lambda abstrakce na typ

- Každý formální parametr v lambda abstrakci přidá do typu výrazu jeden výskyt typového konstrukturu  $\rightarrow$  a novou typovou proměnnou, která se navíc specializuje podle kontextu použití konkrétního formálního parametru.

## Pozorování

- Typ funkce si Hugs/GHCi umí odvodit z její definice.

## Příklad 1

```
'a' :: Char
(\ x -> 'a') :: a -> Char
(\ x y -> 'a') :: a -> b -> Char
(\ x -> (\ y -> 'a')) :: a -> b -> Char
```

## Příklad 2

```
\ x y -> x !! y :: [a] -> Int -> a
\ x y -> x || y :: Bool -> Bool -> Bool
```

## Příklad 3

```
(\ x y -> x . y) :: (a -> b) -> (c -> a) -> c -> b
(\ x y -> x y) :: (a -> b) -> a -> b
```

## Mentální cvičení

- Definujte operátorové sekce pomocí lambda abstrakce.
- Identifikujte funkce vyššího řádu ve svém každodenním životě.

# IB015 Úvod do funkcionálního programování

Akumulační funkce, Definice typů, Vstup/výstup

Jiří Barnat  
Libor Škarvada

## Akumulační funkce

## Pozorování

- Seznam je posloupnost oddělených prvků.
- Motivací akumulčních 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

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

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

## 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
```

## 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

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

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

## 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 akumulčních 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
```

## Uživatелеm definované typy

## Pozorování

- Počítač veškerá data reprezentuje čísly.
- Programátoři jej dobrovolně, či nedobrovolně napodobují.

## Riziko

- Mnohdy číselná reprezentace různých hodnot není přímočará a tedy umožňuje nechtěné zadání neplatných hodnot.
- Neplatné hodnoty mohou vzniknout i neopatrnou aplikací číselných operací.
- **Použití neplatných hodnot může být nebezpečné.**

## Příklad

- Chceme reprezentovat den v týdnu a definovat funkce pracující s touto reprezentací.
- Možné číselné kódování, je následující:  
pondělí = 1, úterý = 2, ..., neděle = 7
- Funkce `zitra` (s chybou) a funkce `je_pondeli` :

```
zitra :: Int -> Int
```

```
zitra x = x+1 -- chyba, má být (x+1) 'mod' 7
```

```
je_pondeli :: Int -> Bool
```

```
je_pondeli x = if (x==1) then True else False
```

## Chyba ve výpočtu

```
je_pondeli 8 ~> False
```

```
je_pondeli (zitra 7) ~> ... ~> False
```

## Definice typů

- V Haskellu pomocí klíčového slova `data` .
- Obecná šablona:  
`data Název_typu = Datové_konstruktory`
- Jednotlivé datové konstruktory se oddělují znakem `|`
- Syntaktické omezení Haskellu: nově definovaný typ i datové konstruktory musí začínat velkým písmenem.

## Příklad

- Dny v týdnu lze definovat jako nový typ, který má 7 hodnot.  
`data Dny = Po | Ut | St | Ct | Pa | So | Ne`
- Hodnoty jsou definovány výčtem.
- Jsou použity nulární datové konstruktory – konstanty.

## Uživatелеm definované

- Obecná šablona pro n-ární datový konstruktor:

**Jméno** **Typ<sub>1</sub>** ... **Typ<sub>n</sub>**

- Příklad typu s ternárním datovým konstruktorem:

data Barva = **RGB Int Int Int**

- Hodnoty typu Barva:

RGB 42 42 42

RGB 12 (-23) 45

## Částečná aplikace datového konstruktoru

RGB :: Int -> Int -> Int -> Barva

RGB 23 :: Int -> Int -> Barva

RGB 23 23 :: Int -> Barva

RGB 23 23 23 :: Barva

## Typové konstanty

- Definicí dle šablony:

```
data Název_typu = Datové_konstruktory
zavádíme nový typ s označením Název_typu.
```

- `Název_typu` je nulární typový konstruktork, typová konstanta.

## N-ární typové konstruktory

- Typové konstruktory jako například `->` nebo `[]` nedefinují typ, pouze předpis jak nový typ vyrobit.

## Tvorba typu

- Každá typová konstanta definuje typ.
- Typ získám také úplnou aplikací n-árních typových konstruktorů na již definované typy.

```
(->) Dny Bool = Dny -> Bool
```

```
[] Dny = [Dny]
```

```
(->) (Dny -> Bool) [Dny] = (Dny -> Bool) -> [Dny]
```

## Tvorba nových hodnot

- Aplikace datových konstruktorů vytváří nové hodnoty.

## Tvorba nových typů

- Aplikace typových konstruktorů vytváří nové typy.

## Uspořádané n-tice a seznamy

- Používá se stejné označení pro typové i datové konstruktory!

'a' :: Char

**[('a','a'), ('a','a')] :: [(Char,Char)]** -- **datové**

**[('a','a'), ('a','a')] :: [(Char,Char)]** -- **typové**

# Příklad použití uživatelem definovaných typů

```
data Policie = Hlidka (String,String) | Oddeleni [Policie]
 deriving Show
```

```
h1 = Hlidka ("Pepa", "Emil")
h2 = Hlidka ("Jason", "Drson")
o1 = Oddeleni [h1, h2]
```

```
jmena :: Policie -> [String]
jmena (Hlidka (a,b)) = a:b:[]
jmena (Oddeleni []) = []
jmena (Oddeleni (x:s)) = jmena x ++ jmena (Oddeleni s)
```

## Polymorfní typové konstruktory

- Seznam prvků typu  $a$ , strom hodnot typu  $a$ , ...

## Definice polymorfních typových konstruktorů

- Definice s využitím typových proměnných:  
data **Název\_typu**  $a_1 \dots a_n = \dots$
- Typové proměnné lze použít pro definici datových konstruktorů.

## Kompletní obecná šablona

```
data Tcons $a_1 \dots a_n =$ Dcons1 typ(1,1) typ(1,2) ... typ(1,arita1)
 ⋮
 Dcons m typ(m ,1) typ(m ,2) ... typ(m ,arita m)
```

## Maybe

- Předdefinovaný unární polymorfní typový konstruktor.

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

- Zamýšlené použití pro funkce, jejichž hodnota může být nedefinována.

## Příklad

- Chceme ošetřit dělení nulou, definujeme novou funkci `deleni`

```
deleni :: Fractional a => a -> a -> Maybe a
```

```
deleni x y = if (y==0) then Nothing else Just (x/y)
```

- Jaký je výsledek aplikace `deleni` na argumenty `32` a `8` ?

## Maybe

- Předdefinovaný unární polymorfni typový konstruktor.  
`data Maybe a = Nothing | Just a`
- Zamýšlené použití pro funkce, jejichž hodnota může být nedefinována.

## Příklad

- Chceme ošetřit dělení nulou, definujeme novou funkci `deleni`  
`deleni :: Fractional a => a -> a -> Maybe a`  
`deleni x y = if (y==0) then Nothing else Just (x/y)`
- Jaký je výsledek aplikace `deleni` na argumenty 32 a 8?  
**Just 4.0**
- Proč je následující definice špatně?  
`deleni x y = if (y==0) then Nothing else (x/y)`

## Vstup/výstup

## Referenční transparentnost

- Daný výraz se vždy vyhodnotí na stejnou hodnotu, bez ohledu na okolí (kontext), ve kterém je použit.
- Programovací jazyk Haskell je referenčně transparentní.

## Dopady na vstup-výstupní chování

- **Nelze napsat program, který by zpracoval vstup uživatele a vyhodnotil se podle zadaného vstupu na různé hodnoty.**
- Lze napsat program, který zpracuje vstup a podle vstupu vypíše na výstup různé výsledky.
- Hodnoty předávané skrze vstup-výstupní akce nesouvisí s hodnotou výrazu, který tuto vstup-výstupní akci realizuje.

## Vstup-výstupní akce

- Interakce programu s uživatelem nebo operačním systémem.
- Například výpis textu na terminálu, vytvoření nového souboru, načtení hodnoty proměnné prostředí, ...

## Myšlenka

- Pro vstup-výstupní akce je zaveden speciální typ – **IO a** .
- Tento typ má formálně jednu jedinou textově nereprezentovatelnou hodnotu, a to **vstup-výstupní akci**.
- Výstupní akce mají typ **IO ()** .  
`putStrLn "Ahoj!":: IO ()`
- Vstupní akce mají typ **IO a** , kde typová proměnná `a` nabývá hodnoty (typu) podle typu objektu, který vstupuje.  
`getLine :: IO String`

## Otázka

- Jestliže `getline` načte řetězec ze vstupu a přitom má hodnotu vstup-výstupní akce, což je hodnota typu `IO a` , konkrétně zde `IO String` , tak potom by nás zajímalo, kde je onen načtený řetězec?

## Otázka

- Jestliže `getline` načte řetězec ze vstupu a přitom má hodnotu vstup-výstupní akce, což je hodnota typu `IO a` , konkrétně zde `IO String` , tak potom by nás zajímalo, kde je onen načtený řetězec?

## Odpověď

- Načtený řetězec se uchová jako tzv. **vnitřní výsledek** provedení této vstupní akce.
- Skutečné načtení řetězce a zapamatování si vnitřního výsledku je realizováno jako **vedlejší efekt** vyhodnocení výrazu `getline` .

## Přístup k hodnotě vnitřního operátoru

- Pomocí binárního operátoru `>>=` .
- Ve výrazu `f >>= g` funguje operátor `>>=` tak, že vezme vnitřní výsledek vstupní akce `f` a na tento aplikuje unární funkci `g` .
- Výraz `f >>= g` tedy znamená, že:  
 $f :: IO\ a$   
 $g :: a \rightarrow IO\ b$   
 $f \gg= g :: IO\ b$

## Operátor >>=

- $(\gg=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$
- Následující zápis je ekvivalentní:  
`getLine >>= putStr`  
`getLine >>= (\x -> putStr x)`

## Otázka

- Operátor (>>=) nelze použít ke spojení vstup-výstupní akce a funkce, která není vstup-výstupní akce, proč?
- Příklad, **takto nelze**:

```
getLine >>= length
```

```
getLine >>= (\ x -> length x)
```

## Otázka

- Operátor (>>=) nelze použít ke spojení vstup-výstupní akce a funkce, která není vstup-výstupní akce, proč?
- Příklad, **takto nelze**:

```
getLine >>= length
```

```
getLine >>= (\ x -> length x)
```

## Odpověď

- Hodnota výrazu je závislá na zadaném vstupu!
- Porušuje referenční transparentnost.
- Typově nesprávně.
- Správné použití:

```
getLine >>= print . length
```

```
getLine >>= (\ x -> print (length x))
```

## Funkce return

- Prázdná akce, jejíž provedení má za cíl pouze naplnit hodnotu vnitřního výsledku.

```
return :: a -> IO a
```

```
return ['A', 'h', 'o', 'j'] >>= putStr
```

## Řazení akcí, operátor >>

- Binární operátor, který řadí vstup-výstupní akce.
- Zapomíná/ničí hodnotu vnitřního výsledku.
- Výraz má hodnotu poslední (druhé) vstup-výstupní akce.

- $(\gg) :: IO\ a \rightarrow IO\ b \rightarrow IO\ b$

- Příklady použití:

```
putStr "Jeje" >> putChar '!'
```

```
getLine >> putStr "nic"
```

# Základní funkce pro výstup

`putChar :: Char -> IO ()`

- Zapiše znak na standardní výstup.

`putStr :: String -> IO ()`

- Zapiše řetězec na standardní výstup.

`putStr :: String -> IO ()`

- Zapiše řetězec na standardní výstup a přidá znak konec řádku.

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

- Vypíše hodnotu jakéhokoliv tisknutelného typu na standardní výstup, a přidá znak konec řádku.
- Tisknutelné typy jsou instancí třídy `Show`.
- Uživatelem definované typy nutno označit přídomkem `deriving Show`.

# Základní funkce pro vstup

`getChar :: IO Char`

- Načte znak ze standardního vstupu.

`getLine :: IO String`

- Načte řádek ze standardního vstupu.

`getContents :: IO String`

- Čte veškerý obsah ze standardního vstupu jako jeden řetězec. Obsah je čten líně, tj. až když je potřeba.

`interact :: (String -> String) -> IO ()`

- Argumentem funkce `interact` je funkce, která zpracovává řetězec a vrací řetězec.
- Veškerý obsah ze standardního vstupu je předán této funkci a výsledek vytištěn na standardní výstup.

```
type FilePath = String
```

- Definuje typový alias.

```
readFile :: FilePath -> IO String
```

- Načte obsah souboru jako řetězec. Soubor je čten líně.

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

- Zapiše řetězec do daného souboru (existující obsah smaže).
- Hodnoty jiného typu než `String` lze konvertovat funkcí `show`.

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

- Připíše řetězec do daného souboru.
- Hodnoty jiného typu než `String` lze konvertovat funkcí `show`.

## Moduly `System` a `Directory`

- Existují další vstup-výstupní funkce pro práci s adresáři či systémovými proměnnými.
- Tyto funkce jsou předdefinovány v modulech `System` a `Directory`.

## Použití modulu

- Moduly, jejichž funkce chceme použít, je třeba označit.
- Lze to učinit v souboru s globálními definicemi použitím klíčového slova `import`.
- Příklad:

```
import Char
import Directory
main = getDirectoryContents ".." >>=
 print . map (\x -> (toUpper.head) x : tail x)
```

## Pozorování

- Syntaktická konstrukce `do` slouží k alternativnímu zápisu výrazu s operátory `>>=` a `>>`.

## Následující zápis je ekvivalentní

- ```
putStr "vstup?" >>
getLine          >>= \ x ->
putStr "výstup?" >>
getLine          >>= \ y ->
readFile x       >>= \ z ->
writeFile y (map toUpper z)
```

```
do putStr "vstup?"
  x <- getLine
  putStr "výstup?"
  y <- getLine
  z <- readFile x
  writeFile y (map toUpper z)
```

Funkce sequence

- Máme-li seznam vstup-výstupních akcí, můžeme je pomocí funkce `sequence` provést dávkově naráz.

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

Příklady použití

- V případě výstupních akcí je výsledkem vyhodnocení výrazu posloupnost výstupů, viz:

```
sequence [ putStr "Ahoj", putStr " ", putStr "světe!" ]
```

- V případě vstupních akcí, je výsledkem vyhodnocení výrazu seznam vstupů, který je uložený jako vnitřní výsledek vstup-výstupní akce, viz:

```
sequence [ getLine, getLine, getLine ] >>= print
```

Funkce mapM

- Aplikuje unární funkci, jejíž výsledkem je vstup-výstupní akce, na seznam hodnot a vzniklý seznam vstup-výstupních akcí provede.

```
mapM :: (a -> IO b) -> [a] -> IO [b]
```

```
mapM f = sequence . map f
```

Příklady použití

- `mapM putStr ["Den", "Noc"]`
vypíše `DenNoc`
- `mapM (\ t -> putStr "Aa") [1,2,3,4,5]`
vypíše `AaAaAaAaAa`
- `mapM (\ x->getLine) [1,2] >>= print`
po zadání dvou řádků s obsahem `radek1` a `radek2`
vypíše `["radek1", "radek2"]`

Mentální cvičení

- Zdůvodněte (uvědomte si) proč je typ funkcí `foldr` a `foldl` takový, jaký je.
- Zdefinujte funkci `sequence` bez použití notace `do`.

Programování v Haskellu

- Definujte funkci, která pro 12 měsíčních platů zadaných seznamem vypočítá 15% daň z příjmu s přihlédnutím k tomu, že z celkové výše ročního příjmu se daní pouze část, která převyšuje nezdanitelné minimum ve výši 24 600 Kč.
- Vyhledejte popis funkcí obsažených v modulech `Char`, `Directory` a `IO` a vyzkoušejte je ve svých programech.
- Napište program, který vyzve uživatele, aby zadal 16 čísel oddělených mezerou, a poté tyto čísla vypíše v matici velikosti 4x4.

IB015 Úvod do funkcionálního programování

Redukční strategie, Seznamy
program QuickCheck

Jiří Barnat
Libor Škarvada

Redukční strategie

Redukční krok

- Úprava výrazu, v němž se některý jeho podvýraz nahradí zjednodušeným podvýrazem.
- Upravovaný podvýraz (**redex**) má tvar aplikace funkce na argumenty, upravený podvýraz má tvar pravé strany definice této funkce do níž jsou za formální parametry dosažené skutečné argumenty.

Redukční strategie

- Předpis, který určuje jaký podvýraz se bude upravovat v následujícím redukčním kroku.

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 , pokud to nestačí k tomu, abychom mohli upravit $F X$, upravujeme částečně výraz X , ale pouze do té míry, abychom mohli upravit výraz $F X$.
- Při úpravě výrazů tedy postupujeme **zvnějšku**.

Líná redukční strategie

- Normální redukční strategii, při níž si pamatujeme hodnoty upravených podvýrazů a žádný s opakovaným výskytem nevyhodnocujeme více než jednou.
- Využívá referenční transparentnost.
- Nelze aplikovat na výrazy s vedlejším efektem.

Haskell

- Používá línou redukční strategii.
- Též označováno jako **líné vyhodnocování**.

Definice funkce

- $\text{cube } x = x * x * x$

Striktní redukční strategie

- $\text{cube } \underline{(3+5)} \rightsquigarrow \underline{\text{cube } 8} \rightsquigarrow \underline{8 * 8 * 8} \rightsquigarrow \underline{64 * 8} \rightsquigarrow 512$

Normální redukční strategie

- $\underline{\text{cube } (3+5)} \rightsquigarrow \underline{(3+5) * (3+5) * (3+5)} \rightsquigarrow 8 * \underline{(3+5) * (3+5)}$
 $\rightsquigarrow \underline{8 * 8} * (3+5) \rightsquigarrow 64 * \underline{(3+5)} \rightsquigarrow \underline{64 * 8} \rightsquigarrow 512$

Líná redukční strategie

- $\underline{\text{cube } (3+5)} \rightsquigarrow \underline{(3+5) * (3+5) * (3+5)} \rightsquigarrow \underline{8 * 8 * 8} \rightsquigarrow$
 $\underline{64 * 8} \rightsquigarrow 512$

Pozorování

- Použitá strategie může ovlivnit chování programu.

Příklad 1

- Uvažme funkci `const`

```
const :: a -> b -> a
```

```
const x y = x
```

- Při striktním vyhodnocování dojde k dělení nulou

```
const 2 (1/0)  $\rightsquigarrow$  ERROR
```

- Při líném vyhodnocování k němu nedojde

```
const 2 (1/0)  $\rightsquigarrow$  2
```

Příklad 2

- Uvažme funkci `undf`

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

```
undf x = undf x
```

- Striktní vyhodnocování následujícího výrazu vede k zacyklení

```
head (tail [undf 1, 4]) =
```

```
head (tail (undf 1 : 4 : [])) ~>
```

```
head (tail (undf 1 : 4 : [])) ~>
```

```
...
```

- Při líném vyhodnocování k zacyklení nedojde:

```
head (tail [undf 1, 4]) =
```

```
head (tail (undf 1 : 4 : [])) =
```

```
head (tail (undf 1 : 4 : [])) ~>
```

```
head (4 : []) ~> 4
```

Churchova-Rosserova věta

- Výsledná hodnota ukončeného výpočtu výrazu nezáleží na redukční strategii: pokud výpočet skončí, je jeho výsledek vždy stejný.

Interpretace věty

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

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.

Interpretace věty

- 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.

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.

Interpretace věty

- Věta o normalizaci 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.

Jiný pohled

- Při použití líné/normální redukční strategie je výraz vyhodnocen až v okamžiku, kdy je potřeba pro další výpočet.
- Přístup, který jde nad rámec redukční strategie.

Příklady

- Líné čtení řetězce ze vstupu:
`getContents :: IO String`
- Líné vyhodnocování Boolovských operátorů v imperativních programovacích jazycích.
`(True OR (1/0)) = True`
`(open(...) OR die) - "umře" pokud open selže.`

Nekonečné datové struktury

- Vyhodnocení výrazu až v okamžiku, kdy je potřeba pro další výpočet, umožňuje manipulaci s nekonečnými datovými strukturami.
- Příkladem nekonečné datové struktury je **nekonečný seznam**.

Příklad

- Seznam nekonečně mnoha jedniček:

`jednicka = 1 : jednicka`

- Vyhodnocení výrazu `jednicka` se zacyklí při každé strategii:

`jednicka` \rightsquigarrow `1:jednicka` \rightsquigarrow `1:1:jednicka` \rightsquigarrow ...

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

`head jednicka` = `head jednicka` \rightsquigarrow `head (1:jednicka)` \rightsquigarrow 1

Nekonečný rostoucí seznam všech přirozených čísel

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

	0	1	2	3	4	5	...	nats
+	1	1	1	1	1	1	...	jednicky
<hr/>								
0	1	2	3	4	5	6	...	

Fibonacciho posloupnost

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

		0	1	1	2	3	5	...	fibs
+		1	1	2	3	5	8	...	tail fibs
<hr/>									
0	1	1	2	3	5	8	13	...	

Nekonečné opakování jednoho prvku

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

Nekonečné opakování seznamu

- `cycle :: [a] -> [a]`
`cycle x = x ++ cycle x`

Opakovaná aplikace funkce

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

Alternativní definice

- `jednicky = repeat 1`
- `jednicky = iterate (+0) 1`
- `jednicky = iterate (id) 1`
- `jednicky = cycle [1]`
- `nats = iterate (+1) 0`

Další příklady

- `take 10 (iterate (*2) 1) ~>*`
- `take 5 (iterate ('a':) []) ~>*`
- `take 10 (iterate (*(-1)) 1) ~>*`
- `take 8 (cycle "Ha ") ~>*`

Alternativní definice

- `jednicka = repeat 1`
- `jednicka = iterate (+0) 1`
- `jednicka = iterate (id) 1`
- `jednicka = cycle [1]`
- `nats = iterate (+1) 0`

Další příklady

- `take 10 (iterate (*2) 1) ~>*` `[1,2,4,8,16,32,64,128,256,512]`
- `take 5 (iterate ('a':) []) ~>*` `["","a","aa","aaa","aaaa"]`
- `take 10 (iterate *(-1)) 1) ~>*` `[1,-1,1,-1,1,-1,1,-1,1,-1]`
- `take 8 (cycle "Ha ") ~>*` `"Ha Ha Ha"`

Zápis seznamů

Prostý výčet

- Zápis pomocí základních datových konstruktorů (`:`) a `[]`.
`1:2:3:4:5:[]`
- Ekvivalentní zkrácený zápis (syntaktická zkratka pro totéž).
`[1,2,3,4,5]`

Hromadný výčet

- Seznamy hodnot, které lze systematicky vyjmenovat (enumerovat) lze zadat tzv. **hromadným výčtem**.
- Seznamy zadané enumerační funkcí `enumFromTo`
`enumFromTo 1 12 ~>* [1,2,3,4,5,6,7,8,9,10,11,12]`
`enumFromTo 'A' 'Z' ~>* "ABCDEFGHIJKLMNOPQRSTUVWXYZ"`
- Všechny uspořádatelné typy jsou enumerovatelné.

Nekonečná enumerace

- Enumerovat lze i hodnoty typů s nekonečnou doménou.
- Hromadným výčtem lze definovat nekonečné seznamy.

```
nats = enumFrom 0
```

Enumerace s udaným vzorem

- Udáním druhého prvku lze definovat vzor enumerace:

```
take 10 (enumFromThen 0 2) ~>* [0,2,4,6,8,10,12,14,16,18]
```

```
enumFromThenTo 0 3 15 ~>* [0,3,6,9,12,15]
```

Přehled enumeračních funkcí a syntaktických zkratk

Enumerační funkce	Typ	Zkratka
<code>enumFrom m</code>	<code>Enum a => a->[a]</code>	<code>[m..]</code>
<code>enumFromTo m n</code>	<code>Enum a => a->a->[a]</code>	<code>[m..n]</code>
<code>enumFromThen m m'</code>	<code>Enum a => a->a->[a]</code>	<code>[m,m'..]</code>
<code>enumFromThenTo m m' n</code>	<code>Enum a => a->a->a->[a]</code>	<code>[m,m'..n]</code>

Intenzionální definice seznamu

- Prvky seznamu jsou generovány společným pravidlem, které předepisuje jak prvky z nějaké nosné množiny přepsat na prvky generovaného seznamu.
- Příklad: prvních deset násobků čísla 2
[2*n | n <- [0..9]]

Obecná šablona

- [definiční_výraz | generátor a kvalifikátory]
- Za každý vygenerovaný prvek vyhovující všem kvalifikátorům se do definovaného seznamu přidá jedna hodnota definičního výrazu.
- Definiční výraz může a nemusí použít generované prvky.
- Kvalifikátory a generátory se vyhodnocují **zleva doprava**.

Generátor

- `nová_proměnná <- seznam` nebo `vzor <- seznam`
- Definuje novou proměnou použitelnou buď v definičním výrazu nebo v libovolném kvalifikátoru vyskytujícím se vpravo.
- Nová proměnná postupně nabývá hodnot prvků v seznamu.
- V případě použití vzoru, se vygeneruje tolik instancí, kolik prvků v seznamu odpovídá použitému vzoru.

Kombinace více generátorů

- Při použití více generátorů se generují všechny kombinace.
- Pořadí kombinací je dáno uspořádáním generátorů v definici.
- Nejvyšší váhu má generátor vlevo, směrem doprava váha klesá.

Predikát

- Výraz typu `Bool` .
- Může využít proměnné definované od predikátu vlevo.
- Vygenerované instance, které nevyhovují predikátu, nebudou brány v potaz pro definici výsledného seznamu.

Lokální definice

- `let nová_proměnná = výraz`
- Definuje novou proměnou použitelnou buď v definičním výrazu nebo v libovolném kvalifikátoru vyskytující se vpravo.
- Výraz může využít proměnné definované vlevo.

- [$n^2 \mid n \leftarrow [0..3]$]
 \rightsquigarrow^* [0,1,4,9]
- [(c,k) | c \leftarrow "abc", k \leftarrow [1,2]]
 \rightsquigarrow^* [('a',1), ('a',2), ('b',1), ('b',2), ('c',1), ('c',2)]
- [$3*n \mid n \leftarrow [0..6]$, odd n]
 \rightsquigarrow^* [3,9,15]
- [(m,n) | m \leftarrow [1..3], n \leftarrow [1..3], n \leq m]
 \rightsquigarrow^* [(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]
- [(m,n) | m \leftarrow [1..3], n \leftarrow [1..m]]
 \rightsquigarrow^* [(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]
- [(x,y) | z \leftarrow [0..2], x \leftarrow [0..z], let y=z-x]
 \rightsquigarrow^* [(0,0), (0,1), (1,0), (0,2), (1,1), (2,0)]

- `[replicate n c | c<-"xyz", n<-[2,3]]`
 \rightsquigarrow^* `["xx","xxx","yy","yyy","zz","zzz"]`
- `[replicate n c | n<-[2,3], c<-"xyz"]`
 \rightsquigarrow^* `["xx","yy","zz","xxx","yyy","zzz"]`
- `[x^2 | [x]<-[[], [2,3], [4], [1,1..], [], [7], [0..]]]`
 \rightsquigarrow^* `[16,49]`
- `[0 | []<-[[], [2,3], [4], [0..], [], [5]]]`
 \rightsquigarrow^* `[0,0]`
- `[x^3 | x<-[0..10], odd x]`
 \rightsquigarrow^* `[1,27,125,343,729]`
- `[x^3 | x<-[0..10], odd x, x < 1]`
 \rightsquigarrow^* `[]`

Redefinice známých funkcí

- `length :: [a] -> Int`
`length s = sum [1 | _ <- s]`
- `map :: (a->b) -> [a] -> [b]`
`map f s = [f x | x <- s]`
- `filter :: (a->Bool) -> [a] -> [a]`
`filter p s = [x | x <- s, p x]`
- `concat :: [[a]] -> [a]`
`concat s = [x | t <- s, x <- t]`

Nové funkce

- `isOrdered :: Ord a => [a] -> Bool`
`isOrdered s = and [x<=y | (x,y) <- zip s (tail s)]`
- `samohlasky :: String -> String`
`samohlasky s = [v | v <- s, v `elem` "aeiouy"]`

Úkol

- Napište funkci, která při aplikaci na konečný seznam uspořadatelných hodnot vrátí seznam těchto hodnot uspořádaných operátorem `<`.

Řešení

- Funkce `qSort` seřadí seznam hodnot vzestupně.
- `qSort :: Ord a => [a] -> [a]`
`qSort [] = []`
`qSort (p:s) = qSort [x | x<-s, x<p]`
`++ [p] ++`
`qSort [x | x<-s, x>=p]`

Prvočísla – Eratosthenovo síto

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
																						23						29
																							23					29
																								23				29
																									23			29
																										23		29
																											23	29
																												29

Prvočísla

- Pro každé p , $2 \leq p \in \mathbb{N}$ platí: p je prvočíslo, právě když p není násobkem žádného prvočísla menšího než p .
- `es :: Integral a => [a] -> [a]`
`es (p:t) = p : es [n | n<-t, n`mod`p/=0]`

```
primes = es [2..]
```

QuickCheck

Myšlenka programu QuickCheck

- Generování náhodných hodnot.
- Testování chování programu na daném počtu těchto hodnot.

Testovaná vlastnost

- **Unární funkce**, která vrací hodnotu typu `Bool`.
- Hodnota `True` indikuje správný výsledek, `False` nesprávný.
- Může volat jiné funkce.

Standardní použití

```
import Test.QuickCheck
quickCheck (\z -> muj_program z == ocekavany_vysledek)
```

Typová specializace a QuickCheck

- Generování náhodných hodnot je možné pouze pokud je přesně znám jejich typ.
- Vlastnosti testované programem QuickCheck nesmí být polymorfního typu, tj. je třeba je typově specializovat.

Typová specializace

- Uvedení typu v globální definici nebo za typovaný výraz.
- Typově specializovat lze i funkce.

Příklady typové specializace

Výraz	Typ
4	Num a => a
4 :: Int	Int
(\ z -> z == z)	Eq a => a -> Bool
(\ z -> z == z) :: Int -> Bool	Int -> Bool

Testované vlastnosti

- `prop1 :: Int -> Bool`
`prop1 = (x+1)*(x+1) == x*x + 2*x + 1`
- `prop2 :: Float -> Bool`
`prop2 = (x+1)*(x+1) == x*x + 2*x + 1`

Použití programu quickCheck v interpretu

- `quickCheck prop1`
OK, passed 100 tests.
- `quickCheck prop2`
Falsifiable, after 9 tests:
-2.166667

Generování náhodných hodnot

- QuickCheck umí generovat náhodné hodnoty číselných typů a z nich umí konstruovat náhodné seznamy a uspořádané n-tice.

Náhodné hodnoty typu Char

- Pro náhodné hodnoty typu Char je třeba prohlásit typovou třídu Char za instanci třídy Arbitrary .

```
import Data.Char
instance Arbitrary Char where
  arbitrary = choose ('\32', '\128')
  coarbitrary c = variant (ord c `rem` 4)
```

- Použití na hodnoty typu Char

```
quickCheck (\z->z=='a')
Falsifiable, after 0 tests:
't'
```

Počet testů

- Přednastavený počet testů může být nedostatečný.
- Definice procedury s větším počtem testů:

```
myCheck = check (defaultConfig { configMaxTest = 10000})
```

- Použití nové testovací procedury:

```
myCheck (\z->z/='a')
```

Falsifiable, after 212 tests:

```
'a'
```

"Ukecané" testování

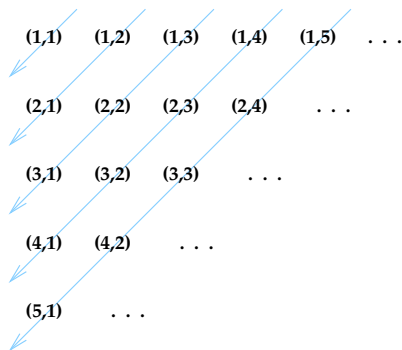
- Při testování vypisuje použité hodnoty

```
verboseCheck (\z->z/="aa")
```

```
...
```

Definice seznamů

- Definujte seznam všech uspořádaných dvojic přirozených čísel tak, aby dvojice byly v definovaném seznamu uspořádány dle následujícího schématu:



- Nápověda: součet čísel v dvojici je po diagonále shodný a postupně se zvyšuje o jedna.

QuickCheck

- Programem QuickCheck ověřte, že dvojice (3, 4) a (4, 3) jsou v seznamu uvedeny ve správném pořadí.
- Programem QuickCheck ověřte, že náhodně generované dvojice dvojic přirozených čísel jsou v seznamu uvedeny ve správném pořadí.

Nápověda: dvojice s čísly < 1 automaticky vyhovují testu.

Verze s lexikografickým uspořádáním

- Opakujte celé zadání s tím, že dvojice jsou v seznamu uspořádány lexikograficky:

$$(a, b) < (c, d) \iff (a < c) \vee (a = c \wedge b < d)$$

- Vysvětlete nastalý problém s testováním.

IB015 Úvod do funkcionálního programování

Časová složitost, Typové třídy, Moduly

Jiří Barnat
Libor Škarvada

Časová složitost

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.

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!



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`

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

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

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

- 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])
```

- 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])
```

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

```
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])
```

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

```
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])) ++ [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]
```

- 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]
```

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

```
reverse [1,2,3]
~> rev [] [1,2,3]
~> rev [1] [2,3]
~> rev [2,1] [3]
~> rev [3,2,1] []
~> [3,2,1]
```

- 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]
```

- 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]
```

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 multiplikační konstantu.
- Podle toho hovoříme o funkcích lineárních, kvadratických, exponenciálních apod.

Přehled asymptotických funkcí

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

`reverse'`

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

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

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

`reverse`

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

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

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

Definice funkcí

```
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 výpočtu vzhledem k exponentu

- Složitost funkce `mocnina'` je **lineární**.
- Složitost funkce `mocnina` je **logaritmická**.

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**.

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).

Pozorování

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

Příklad

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

```
insort :: Ord a => [a] -> [a]
insort = 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 insort

```
insort [x1, x2, ..., xn-1, xn]
  ~> foldr ins [] [x1, x2, ..., xn-1, xn]
  ~>n+1 ins x1 (ins x2 (... (ins xn-1 (ins xn []))...))
```

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)

```

      minim [x1, ..., xn]
↪ (head.inssort) [x1, ..., xn]
↪ head (inssort [x1, ..., xn])
↪ head (foldr ins [] [x1, ..., xn])
↪n+1 head (ins x1 (... (ins xn-2 (ins xn-1 (ins xn []))))))
↪3·0+1 head (ins x1 (... (ins xn-2 (ins xn-1 [xn])))
↪3·1+1 head (ins x1 (... (ins xn-2 [xn, xn-1])))
↪3·2+1 head (ins x1 (... [xn, xn-1, xn-2] ...))
      ⋮
↪3·(n-2)+1 head (ins x1 [xn, ..., x2] )
↪3·(n-1)+1 head [xn, ..., x1]
↪ xn
```


Striktní vyhodnocování

- Počet redukčních kroků výrazu $\text{minim } [x_1, \dots, x_n]$ 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 $\text{minim } [x_1, \dots, x_n]$ je:

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

- Při líném vyhodnocování má funkce **lineární** 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 [a1, ..., an]`

Typové třídy

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 ()`

Význam

- Identifikují společné vlastnosti různých typů.
- Umožňují definici funkcí polymorfních typů zúžených na typy požadovaných vlastností.

Programátorský význam

- Definice a použití typových tříd umožňují 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.



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`

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)`

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)
```

## Definice typové třídy

- $\text{class } [ (C_1 a, \dots, C_k a) \Rightarrow ] C a$   
     $\left[ \begin{array}{l} \text{where } op_1 :: typ_1 \\ \quad \vdots \\ op_n :: typ_n \\ \quad \left[ \begin{array}{l} default_1 \\ \quad \vdots \\ default_m \end{array} \right] \end{array} \right]$

## Deklarace instance

- $\text{instance } [ (C_1 a_1, \dots, C_k a_k) \Rightarrow ] C typ$   
     $\left[ \begin{array}{l} \text{where } valdef_1 \\ \quad \vdots \\ valdef_n \end{array} \right]$

## 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.

## 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

- 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)
```

Moduly a modulární návrh programů

Motivace

- Oddělení nezávislých, znovupoužitelných, logicky ucelených částí kódu do separátních celků – **modulů**.

Zapouzdření

- Při definici modulu je nutné explicitně vyjmenovat funkce, které mají být viditelné a použitelné mimo rozsah modulu, tzv. **exportované** funkce.
- Ostatní funkce a datové typy definované v modulu nejsou z vnějšku modulu viditelné.
- Moduly by měly exportovat jen to, co je nutné.
- Modul může exportovat hodnoty, typy a typové konstruktory, typové a konstruktorové třídy, jména modulů.

Obecná definice

- `[module Jméno [(export1, ..., exportn)] where]`
`[import M1 [spec1]]`
`[⋮]`
`[import Mm [specm]]]`
`[globální deklarace]`

Automatické doplnění definice

- Není-li uvedena hlavička, doplní se
`module Main (main) where`
- Nevyskytuje-li se mezi importovanými moduly M_1, \dots, M_m modul `Prelude`, doplní se
`import Prelude`

Hlavní funkce

- Program musí mít definovanou hlavní funkci – funkci `main`.
- Právě jeden modul v programu musí být `Main`.

Modul `Main`

- Modul `Main` musí exportovat hodnotu

`main :: IO τ`

pro nějaký typ τ , (obvykle $\tau = ()$).

Datový typ Fifo

- Datový kontejner (struktura, která uchovává prvky) přístupovaný operacemi **vlož prvek** a **vyber prvek**.
- Prvky jsou z datové struktury odebírány v tom pořadí, ve kterém byly vkládány.
- First-In-First-Out = FIFO
- Operace by měly mít konstantní časovou složitost.

Realizace v Haskellu

- Definice modulu `Fifo`
- Použití modulu:

```
import Fifo
```

Příklad Modulu – Datový typ Fifo

```
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) [])
```

Domácí cvičení

- Příprava na vnitro-semestrální písemku!

IB015 Úvod do funkcionálního programování

A zase ta REKURZE ...

Jiří Barnat
Libor Škarvada

Jiný pohled na rekurzivní funkce

Rekurze

- Definice funkce, nebo datové struktury, s využitím sebe sama.

Příklad

- Funkce `length`, která při aplikaci na seznam vrací jeho délku, je definovaná rekurzivně:

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

Zacyklení výpočtu

- Ne každé použití definovaného objektu na pravé straně definice je smysluplné.
- Nesprávné použití může vést k nekonečnému vyhodnocování, které nemá žádný efekt – **výpočet cyklí**.

Příklad

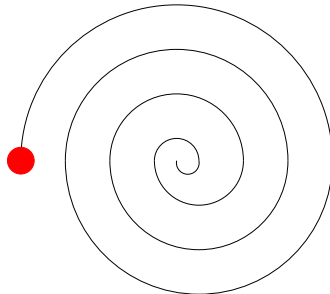
- Nesprávné použití rekurze ve funkci `length'` :
 `length' :: [a] -> Integer`
 `length' [] = 0`
 `length' x = length' x`
- Při aplikaci `length'` na neprázdný seznam výpočet cyklí.

Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu**.
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
= 1 + length [3,2,1]
= 1 + 1 + length [2,1]
= 1 + 1 + 1 + length [1]
= 1 + 1 + 1 + 1 + length []
= 1 + 1 + 1 + 1 + 0
= 4
```

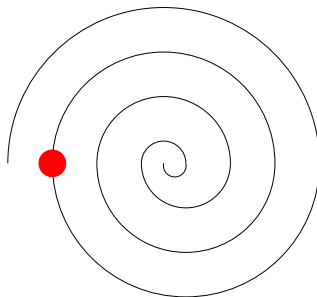


Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu**.
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
  = 1 + length [3,2,1]
  = 1 + 1 + length [2,1]
  = 1 + 1 + 1 + length [1]
  = 1 + 1 + 1 + 1 + length []
  = 1 + 1 + 1 + 1 + 0
  = 4
```

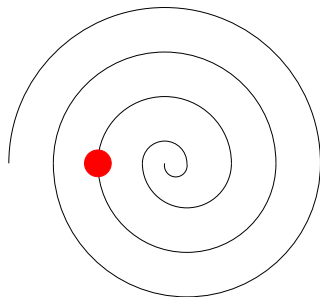


Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu**.
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
  = 1 + length [3,2,1]
  = 1 + 1 + length [2,1]
  = 1 + 1 + 1 + length [1]
  = 1 + 1 + 1 + 1 + length []
  = 1 + 1 + 1 + 1 + 0
  = 4
```

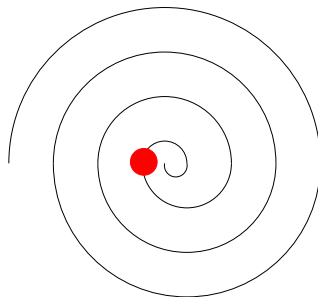


Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu**.
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
  = 1 + length [3,2,1]
  = 1 + 1 + length [2,1]
  = 1 + 1 + 1 + length [1]
  = 1 + 1 + 1 + 1 + length []
  = 1 + 1 + 1 + 1 + 0
  = 4
```

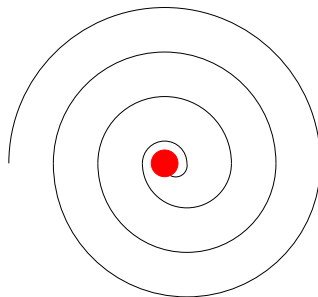


Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu**.
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
  = 1 + length [3,2,1]
  = 1 + 1 + length [2,1]
  = 1 + 1 + 1 + length [1]
  = 1 + 1 + 1 + 1 + length []
  = 1 + 1 + 1 + 1 + 0
  = 4
```



Pozorování

- Uvědomění si toho, co udává vzdálenost od středu pomyslné spirály, je klíč k správnému použití rekurze.

Rekurze ve funkci `length`

- Vzdálenost od středu odpovídá délce zbývajících částí seznamu.
- S každým dalším rekurzivním voláním funkce se seznam, který je argumentem funkce, zkracuje.
- Funkce `length` je tedy jednou nevyhnutelně volána pro prázdný seznam, což je volání, které rekurzi zastaví.

2 části definice

- Při definice rekurzivní funkce je nutné si uvědomit, co je **středem spirály**, tj. kde se má výpočet rekurzivní funkce zastavit, a **jak se k tomuto středu bude výpočet blížit**.

Příklad – 2 části definice funkce length

- Ukončení rekurzivního výpočtu (střed spirály)
`length [] = []`
- Jedno rekurzivní volání (přiblížení se o "jednu otáčku")
`length (x:s) = 1 + length s`

Příklad – 2 části definice v jednom výrazu

- Obě části v jednom řádku definice
`f1 x = if (odd x) then x else f1 (x/2)`

Rekurzivní funkce a větvení

- V případě, že se výpočet funkce větví, vzdálenost od středu pomyslné spirály musí klesat s každou větví.

Funkce s nekonečnou rekurzí

- Teoreticky je možné použít rekurzi pro realizaci nekonečného cyklu. V praxi však toto řešení nefunguje vzhledem k omezené velikosti paměti, pro uchování návratových adres.

Vzdálenost od středu

- To, že pomyslná vzdálenost od středu klesá, nemusí nutně znamenat, že datová struktura, se kterou rekurzivní funkce pracuje, se zmenšuje.

Příklad

- Je-li cílem algoritmu opakovaným dělením celku dosáhnout určitého počtu dílků, počet dílků při každém dělení roste.
- Vzdálenost od středu pomyslné spirály lze v tomto případě identifikovat jako počet dělení, které zbývá k dosažení cílového počtu.
- Všimněme si, že pokud se při každém kroku zdvojnásobí počet dílků, jejich počet roste vzhledem k počtu rekurzivních kroků exponenciálně.

Pozadí rekurze

- Struktura, podle níž se řídí rekurze, nemusí být spojena s úplným uspořádáním.
- Musí však být **dobře založená** (well-founded), což znamená, že v ní neexistuje nekonečně dlouhá klesající posloupnost prvků.

Příklad

- Množina všech podmnožin dané množiny je pouze částečně uspořádána vzhledem k inkluzi, avšak postupné odebrání prvků z libovolné podmnožiny nevyhnutelně dospěje k prázdné množině.

Rekurzivní datové struktury

Pozorování

- Pro definici rekurzivních datových struktur (hodnot rekurzivních typů) platí podobná pravidla jako pro definice rekurzivních funkcí.

Opačný směr

- Vytváření hodnot rekurzivního datového typu probíhá od středu pomyslné spirály směrem ven.
- Rekurzivní datová struktura má **základní** (bázovou) **hodnotu**.
- Základní hodnota je rozvíjena **rekurzivním pravidlem**.

Klasický pohled na seznam

- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

Rekurzivní pohled na seznam

- `[]` je **seznam**.
- `(a : seznam)` je **seznam**.

Demonstrace

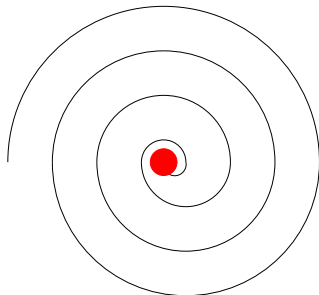
`[]`

`[1] = 1 : []`

`[2,1] = 2 : [1]`

`[3,2,1] = 3 : [2,1]`

`[4,3,2,1] = 4 : [3,2,1]`



Klasický pohled na seznam

- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

Rekurzivní pohled na seznam

- `[]` je **seznam**.
- `(a : seznam)` je **seznam**.

Demonstrace

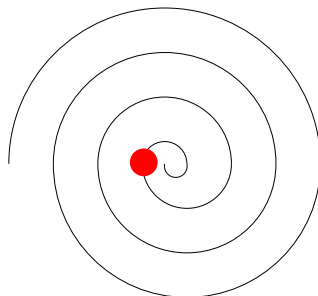
`[]`

`[1] = 1 : []`

`[2,1] = 2 : [1]`

`[3,2,1] = 3 : [2,1]`

`[4,3,2,1] = 4 : [3,2,1]`



Klasický pohled na seznam

- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

Rekurzivní pohled na seznam

- `[]` je **seznam**.
- `(a : seznam)` je **seznam**.

Demonstrace

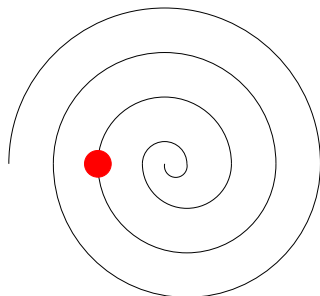
`[]`

`[1] = 1 : []`

`[2,1] = 2 : [1]`

`[3,2,1] = 3 : [2,1]`

`[4,3,2,1] = 4 : [3,2,1]`



Klasický pohled na seznam

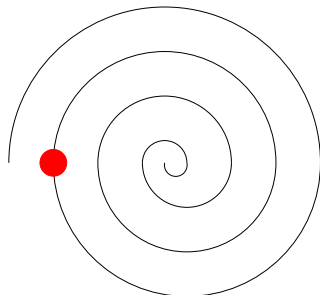
- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

Rekurzivní pohled na seznam

- `[]` je **seznam**.
- `(a : seznam)` je **seznam**.

Demonstrace

```
[]  
[1] = 1 : []  
[2,1] = 2 : [1]  
[3,2,1] = 3 : [2,1]  
[4,3,2,1] = 4 : [3,2,1]
```



Klasický pohled na seznam

- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

Rekurzivní pohled na seznam

- `[]` je **seznam**.
- `(a : seznam)` je **seznam**.

Demonstrace

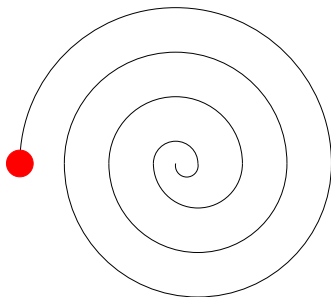
`[]`

`[1] = 1 : []`

`[2,1] = 2 : [1]`

`[3,2,1] = 3 : [2,1]`

`[4,3,2,1] = 4 : [3,2,1]`



Pozorování

- Rekurzivní nahlížení na seznam se může jevit jen jako mentální hříčka.

Stromy jako rekurzivní datové struktury

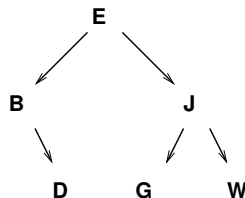
- Mnoho problémů je přirozené řešit s využitím jiné rekurzivně definované datové struktury – **binárního stromu**.
- Nelineární rekurzivní datová struktura.

Rekurzivní definice binárního stromu

- Prázdný strom je **binární strom**
- Hodnota a a k ní asociovaný levý a pravý **binární strom** je **binární strom**

Příklad

- Graficky zadaný binární strom.
- E označujeme jako **kořen** stromu
- E, B a J jsou **vnitřní vrcholy** stromu
- D, G a W označujeme jako **listy**
- Levý a pravý binární strom asociovaný s danou hodnotou označujeme jako levý a pravý **podstrom**.
- Binární stromy asociované k hodnotám D, G, W a levý podstrom asociovaný s hodnotou B jsou prázdné stromy.



Definice datového typu `BinTree a`

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

Příklady hodnot definovaného typu

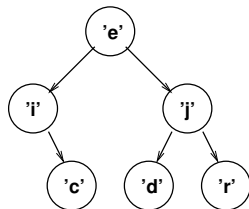
```
tc :: BinTree Char
```

```
tc = Node 'e'
```

```
      (Node 'i' Empty (Node 'c' Empty Empty))
```

```
      (Node 'j' (Node 'd' Empty Empty)
```

```
            (Node 'r' Empty Empty))
```



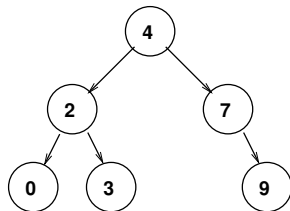
```
tn :: BinTree Int
```

```
tn = Node 4
```

```
      (Node 2 (Node 0 Empty Empty)
```

```
            (Node 3 Empty Empty))
```

```
      (Node 7 Empty (Node 9 Empty Empty))
```



Problém

- Chceme definovat funkci, která při aplikaci na hodnotu typu `BinTree Int` zvýší o jedna všechny hodnoty uložené v uzlech stromu.

Jak takovou funkci definovat?

- Výčtem hodnot nelze – možných hodnot je nekonečně mnoho.

```
treeP1' :: Num a => BinTree a -> BinTree a
treeP1' Empty = Empty
treeP1' (Node x) Empty Empty = Node (x+1) Empty Empty
⋮
```

- **Rekurzivně**, rekurzi vedeme podle struktury stromu

```
treeP1 :: Num a => BinTree a -> BinTree a
treeP1 Empty = Empty
treeP1 (Node x left right)
    = Node (x+1) (treeP1 left) (treeP1 right)
```

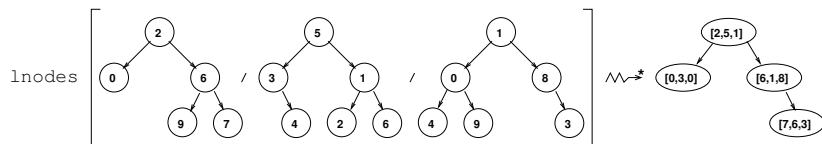
Popis funkce `treezipwith`

- 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.

Popis funkce lnodes

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

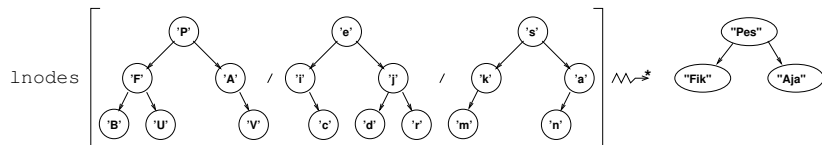
Příklad



Definice funkce `lnodes`

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

Příklad



Dokazování rekurzivních programů

Fakta

- Ověřování správnosti navržených algoritmů je součástí práce programátora.
- Testování je nedokonalé.
- Správnost algoritmu můžeme prokázat například tím, že ji formálně (= s matematickou přesností) dokážeme.

Důkaz korektnosti algoritmu

- Dokazujeme, že pokud výpočet algoritmu na platných vstupech skončí, tak algoritmus vrací korektní výsledek. O algoritmu, který má tuto vlastnost říkáme, že je **částečně správný**.
- Pokud je algoritmus částečně správný a dokážeme, že na platných vstupech svůj výpočet vždy skončí, pak říkáme, že algoritmus je **úplně správný**.

Pozorování

- Pro důkazy částečné správnosti i terminace rekurzivních funkcí se používá **matematická indukce**.

Matematická indukce

- Matematická indukce je metoda dokazování tvrzení, která se používá, pokud chceme ukázat, že dané tvrzení platí pro všechny prvky dobře založené rekurzivně definované nekonečné posloupnosti. (Jako jsou například přirozená čísla.)

Princip matematické indukce

- Ukážeme platnost tvrzení pro bázovou hodnotu.
- Ukážeme, že tvrzení se přenáší při aplikaci rekurzivního kroku.

$$\underline{T(0)} \text{ a } \underline{T(i) \Rightarrow T(i+1)} \implies T(0), T(1), T(2), \dots$$

Pozorování

- Klíčovým problémem při použití matematické indukce je identifikace toho, podle čeho má být indukce vedena.
- Napovědět může místo rekurzivního volání funkce, neboť rekurze a matematická indukce spolu úzce souvisí.

Příklad

- Dokažte, že pro každá dvě přirozená čísla x a y taková, že $x > 0$ platí, že funkce `fpow` aplikovaná na argumenty x a y vrátí hodnotu x^y .
- `fpow :: Integer -> Integer -> Integer`
`fpow x 0 = 1`
`fpow x y = x * fpow x (y-1)`

Pozorování

- Klíčovým problémem při použití matematické indukce je identifikace toho, podle čeho má být indukce vedena.
- Napovědět může místo rekurzivního volání funkce, neboť rekurse a matematická indukce spolu úzce souvisí.

Příklad

- Dokažte, že pro každá dvě přirozená čísla x a y taková, že $x > 0$ platí, že funkce `fpow` aplikovaná na argumenty x a y vrátí hodnotu x^y .
- `fpow :: Integer -> Integer -> Integer`
`fpow x 0 = 1`
`fpow x y = x * fpow x (y-1)`
- **Důkaz povedeme indukcí vzhledem k hodnotě y .**

Bázový krok, T(0)

- Necht' $y=0$, a necht' x je libovolné.
- $f_{\text{pow } x } y$ se redukuje dle $f_{\text{pow } x } 0 = 1$ na hodnotu 1 .
- $x^0 = 1$, pro libovolné x .
- Tudíž pro $y = 0$ tvrzení platí.

Indukční krok, $T(i) \Rightarrow T(i+1)$

- Dokazujeme, že pokud tvrzení platí pro hodnotu i , pak tvrzení platí i pro hodnotu $i + 1$. Platnost tvrzení pro hodnotu i se označuje jako **indukční předpoklad**.
- Platnost tvrzení pro hodnotu i říká, že $\text{fpow } x \ i \rightsquigarrow^* x^i$ pro libovolnou hodnotu x .
- $\text{fpow } x \ (i+1)$
 - $\rightsquigarrow x * \text{fpow } x \ (i+1-1)$
 - $\rightsquigarrow x * \text{fpow } x \ i \stackrel{\text{dleIP}}{=} x * x^i$
 - $\rightsquigarrow x^{i+1}$
- Ukázali jsme, že pokud tvrzení platí pro i , pak platí i pro $i + 1$.
- Z platnosti báze a vlastností matematické indukce plyne, že pro libovolnou hodnotu x tvrzení platí pro všechny hodnoty y .

Věta 1

- Jsou-li s , t dva konečné seznamy stejného typu a délek, a označíme-li $m = \text{length } s$, $n = \text{length } t$, pak

$$\text{length } (s ++ t) = m + n.$$

- Důkaz veden indukcí podle délky seznamu s .

Věta 2

- Pro každé tři seznamy s , t , u platí rovnost

$$(s ++ t) ++ u = s ++ (t ++ u).$$

- Důkaz veden indukcí podle délky seznamu s .

Věta 3

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

$$\text{take } m \ s ++ \text{drop } m \ s = s.$$

- Důkaz veden indukcí podle m .

A to je konec ...

Co jsme se dozvěděli ...

- Funkcionální výpočetní paradigma.
- Solidní základy programovacího jazyka Haskell.

Čím ještě nám byl kurz prospěšný ...

- Správné funkcionální návyky pro programování použijeme v imperativních programovacích jazycích.
- Mentální posilovna.

Co jsme se dozvěděli ...

- Funkcionální výpočetní paradigma.
- Solidní základy programovacího jazyka Haskell.

Čím ještě nám byl kurz prospěšný ...

- Správné funkcionální návyky pro programování použijeme v imperativních programovacích jazycích.
- Mentální posilovna.



IB016 Seminář z funkcionálního programování

- Programování v Haskellu

IA014 Funkcionální programování

- Teoretické základy funkcionálního programování
- Implementace funkcionálních jazyků
- Vybrané techniky funkcionálního programování

Přednášejícího studentům

- Za vzornou docházku a přípravu jak na přednášky, tak i na vnitrosemestrální a zkouškové písemky a za celkově poctivý přístup ke studiu.

Studentů přednášejícímu

- Formou zpětné vazby například vyplněním studentské ankety a upozorněním na zásadní, ale i okrajové nedostatky jak přednášejícího, tak i jím připravených studijních materiálů.