

# IB015 Neimperativní programování

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

Jiří Barnat  
Libor Škarvada

## Datové struktury

- Uspořádané n-tice, seznamy.

## Hodnoty a typy

- Elementární, složené typy, typy funkcí.
- Polymorfní a kvalifikované typy.
- Hodnotové konstruktory.

## Rekurze

- Rekurzivní funkce na seznamech.

## Funkce vyšších řádů

## Definice

- Funkce, je považována za funkci vyššího řádu, pokud alespoň jeden z jejich argumentů, které funkce má, nebo výsledek, který funkce vrací, 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
```

- Uvedte 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
```

- Uvedte 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  $(\oplus q)$   
 $(\oplus q) = \text{flip } (\oplus) q$

## Příklad 1

- Jednoduché použití operátorových sekcí.

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

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

## Příklad 2

- Odvodte typ následujících funkcí, popište jejich význam:

`(1.0/)`

`('mod' 3)`

`(!!0)`

`(>0)`

## Příklad 1

- Jednoduché použití operátorových sekcí.

`(*2) 34 ~> 68`

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

## Příklad 2

- 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

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

## Pozorování

- Operátor pro skládání funkcí lze chápat také jako binární.

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

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

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

## Skládání funkcí

- Základní operátor funkcionálního programování

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

$(.) f g x = f ( g x )$

## Pozorování

- Operátor pro skládání funkcí lze chápat také jako binární.

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

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



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

## 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]
```



## 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 nepojmenovaným 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++, 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 -> x*x + x*x) 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  
 $\backslash x \rightarrow (\backslash y \rightarrow x - y)$   
 $(\backslash x \rightarrow (\backslash y \rightarrow x - y)) 3 4 \rightsquigarrow \dots \rightsquigarrow -1$
- Zkrácený z důvodu čitelnosti kódu a pohodlí programátorů  
 $\backslash x y \rightarrow x - y$   
 $(\backslash x y \rightarrow x - y) 3 4 \rightsquigarrow \dots \rightsquigarrow -1$

## 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
```

## Pozorování

- Vnořené aplikace funkcí mohou být díky závorkování nečitelné.
- Něterým uzávorkováním se lze vyhnout použitím aplikačního operátoru \$ , který má při vyhodnocování nejnižší prioritu.

## Aplikační operátor \$

- $(\$)$  ::  $(a \rightarrow b) \rightarrow a \rightarrow b$   
 $(\$) f x = f x$
- $f(g x) \equiv (\$) f (g x) \equiv f \$ (g x) \equiv f \$ g x$
- $f(g(h x)) \equiv f \$ g \$ h x$

## Ekvivalentní zápisy pomocí operátoru \$

- `not ( not ( not ( not ( not True ))) ) )`  
`not $ not $ not $ not $ not True`
- `(+1) ((+2) ((+3) ((+4) 0)))`  
`(+1) $ (+2) $ (+3) $ (+4) 0`
- `(even ((+1) 0)) || (odd ((+2) 0))`  
`(even $ (+1) 0) || (odd $ (+2) 0)`

## Příklady

- `(++) [1] $ map (+1) [1] ~>* [1,2]`
- `[1] ++ $ map (+1) [1] ~>* ERROR`
- Rozsah platnost \$ je podřízen syntaktickému pravidlu o zarovnání, tj. v do notaci "končí s koncem řádku".



## Mentální cvičení

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