Priority operátorů, prefix/infix, if, definice podle vzoru, lokální definice

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2017

A Cup of Tea

Nan-in, a Japanese master during the Meiji era (1868–1912), received a university professor who came to inquire about Zen. Nan-in served tea. He poured his visitor's cup full, and then kept on pouring.
The professor watched the overflow until he no longer could restrain himself. "It is overfull. No more will go in!"
"Like this cup," Nan-in said, "you are full of your own opinions and speculations. How can I show you Zen unless you first empty your cup?"

The hardest thing about learning functional programming is forgetting what you think you "know". It might be true, but not in this context. Everything is different, so you need to start from the very beginning.
If you know C++, and you want to learn Java, you compare both languages all the time, and this works fine. It's the same with natural languages: If you know English and want to learn Spanish, it's okay to compare both languages all the time, as they follow the same indo-germanic paradigm. But if you want to learn something fundamental different, like Japanese, or Haskell, you have to "unlearn" first. You won't make real progress until you stop comparing.

Zenový kōan a komentář o výuce funkcionálního programování
přebrán z diskusního fóra *Programmers Stack Exchange* od uživatele *Landei*.

Haskell v praxi

Haskell:

- kompilátor vs. interpret
- GHC, Haskell Platform
  (instalace viz osnova v ISu)
- GHC a Haskell Platform: `ghc`, `ghci`, `runghc`

Referenční kompilátor pro podzim 2017:

- GHC-7.10.3
- existují i novější verze (nejnovější 8.2.1), klidně instalujte ty

Haskell, počítače na FI

Spouštení:

- všechny nástroje jsou spustitelné v shellu
- spuštení shellu např.: `Alt-F2`, `gnome-terminal`
- prompt shellu: `xlogin@nymfe12:/home/xlogin$`
- prompt interpretu: `Prelude>` nebo `Main>`

Stroje na FI (nymfeXX):

- Ubuntu 16.04 má lokálně instalovaný GHC-7.10.3

Práce s interpretem

Do interpretu (GHCi) lze zadávat příkazy a výrazy.
Příkazy:

- `:h[elp]` – nápověda
- `:t[ype] expr` – typ výrazu
- `:i[nfo] ident` – informace o objektech jazyka
- `:l[oad] file.hs` – načtení souboru s Haskellovým kódem
- `:r[eload]` – znovunačtení posledního souboru
- `:q[uit]` – ukončení práce s interpretem
- `:m[odule] Modul` – načtení modulu

Výrazy:

- vše ostatní...

Nekončící výpočty lze přerušit pomocí `Ctrl-C`.

Úvod k operátorům

Příklad 1.1.1: S použitím interpretru jazyka Haskell porovnejte vyhodnocení následujících dvojic výrazů a rozdíl vysvětlete.

a) `5 + 9 * 3` versus `(5 + 9) * 3`

b) `2 ^ 2 ^ 2 == (2 ^ 2) ^ 2` versus
`3 ^ 3 ^ 3 == (3 ^ 3) ^ 3`

c) `3 + 3 + 3` versus `3 == 3 == 3`

d) `(3 == 3) == 3` versus `(4 == 4) == (4 == 4)`

Vlastnosti operátorů

Příklad 1.1.2: S využitím interního příkazu `:info` interpretru `ghci` zjistěte prioritu a směr vyhodnocování následujících operací:

```
^, *, /, `div`, `mod`, +, -, ==, /=, >, <, >=, <=,
                                &&, ||
```

Vlastnosti operátorů

operátor	priorita	asociativita
`.`	9	$\leftarrow$
`!!`	9	$\rightarrow$
`^`	8	$\leftarrow$
`*, /, `div`, `mod``	7	$\rightarrow$
`+, -`	6	$\rightarrow$
`:, ++`	5	$\leftarrow$
`==, /=, <, <=, >, >=`	4	–
`&&`	3	$\leftarrow$
`||`	2	$\leftarrow$
`>>=, >>`	1	$\rightarrow$
`$`	0	$\leftarrow$

Pokud to nemá operátor/funkce explicitně definované, jeho priorita je 9 a je asociativní zleva ($\rightarrow$).

Podmíněný výraz

Syntax: `if bool_expr then expr1 else expr2`
Výrazy *expr1* a *expr2* musí být stejného typu.

Příklad 1.1.3: Vysvětlete, co je chybné na následujících podmíněných výrazech, a výrazy vhodným způsobem upravte.

a) `if 5 - 4 then False else True`

b) `if 0 < 3 && odd 6 then 1 else "chyba"`

c) `(if even 8 then (&&)) (0 > 7) True`

Prefix, infix, postfix

Binární funkce se dají zapisovat třemi různými způsoby:

1. **prefixový zápis:** `(+) 3 4`, `mod 7 5`
   - operátor/funkce před argumenty
   - je nativně podporovaný v Haskellu (má vyšší prioritu než infix)
   - nealfanumerické funkce musí být v závorkách
2. **infixový zápis:** `3 + 4`, `7 `mod` 5`
   - operátor/funkce mezi argumenty
   - je nativně podporovaný v Haskellu (má nižší prioritu než prefix)
   - alfanumerické funkce musí být ve zpětných apostrofech
3. **postfixový zápis:** `3 4 (+)`, `7 5 mod`
   - operátor/funkce za argumenty
   - není podporovaný v Haskellu

Prefixový a infixový zápis

Příklad 1.1.4: Přepište infixové zápisy výrazů do syntakticky správných prefixově zapsaných výrazů a naopak:

a) `4 ^ (7 `mod` 5)`

b) `max 3 ((+) 2 3)`

Syntaktická struktura aneb závorkování výrazů

Příklad 1.1.5: Doplňte všechny implicitní závorky do následujících výrazů:

a) `recip 2 * 5`

b) `sin pi/2`

c) `3 `mod` 8 * 2`

d) `f g 3 + g 5`

e) `2 + div m 18 == m ^ 2 ^ n && m * n < 20`

f) `flip (.) snd . id const`

Programy v souborech

Zapisování programů v souborech:

- soubory mají příponu `.hs`
- po spuštění `ghci` se načtou pomocí povelu `:l filename`
- soubor obsahuje definice uživatelských funkcí
- pro znovunačtení stejných souborů je možné použít povel `:r`
- komentáře v kódu:

```
-- line comment
function :: Int -> Int
function x = x + 1
{-
block comment
multiple lines
-}
```

Definice funkce podle vzoru

Definice funkce podle vzoru obsahuje:

- název funkce (`tell`),
- hodnoty argumentů (`1`, `2`), formální parametry (`x`), anonymní parametry (`_`),
- rovnítko oddělující pravou a levou stranu definice (=),
- pravou stranu/tělo funkce.

```
tell 1 = "one"
tell 2 = "two"
tell x = if even x then "(even)" else "(odd)"
tell _ = "never happens"
```

- Všechny řádky definující funkci musí být spolu.
- Není možné použít stejný parametr na levé straně víckrát.

Funkce logické konjunkce

Příklad 1.2.1: Definujte funkci `logicalAnd`, která se chová stejně jako funkce logické konjunkce, tak, abyste v definici

a) využili podmíněný výraz.

b) nepoužili podmíněný výraz.

Cvičení

Příklad 1.2.7: Napište funkci `roots`, která se po aplikaci na koeficienty a, b, c vyhodnotí na počet reálných kořenů kvadratické rovnice $ax^2 + bx + c = 0$.

Lokální definice

Lokální definice jsou pojmenované výrazy s lokální platností, jejich primárním účelem je zpřehlednit kód (a nezavádět nová globální jména). Rozeznáváme 2 druhy:

1. s definicí před výrazem (**let-in**)
   Syntax: `let var = subexpr in expr`
2. s definicí za výrazem (**where**)
   Syntax: `expr where var = subexpr`

- Je možné definovat i více výrazů v jedné definici:

```
let a = 25 + b
    b = 5
    in (a + b) * b - 1
```

- Na zarovnání záleží!
- V lokálních definicích je možné používat vzory.

Rekurze

Příklad 1.2.3: Upravte následující kód tak, aby funkce pro záporná čísla necyklila, ale skončila s chybovou hláškou. Použijte k tomu funkci `error :: String -> a`.

```
power :: Double -> Int -> Double
_ `power` 0 = 1
z `power` n = z * (z `power` (n-1))
```

Rekurze

Příklad 1.2.3: Upravte následující kód tak, aby funkce pro záporná čísla necyklila, ale skončila s chybovou hláškou. Použijte k tomu funkci `error :: String -> a`.

```
power :: Double -> Int -> Double
_ `power` 0 = 1
z `power` n = z * (z `power` (n-1))
```

Příklad 1.2.13: Naprogramujte rekurzivně funkci, která pro dané kladné celé číslo vypočítá jeho zbytek po dělení 3. Nesmíte použít funkci `mod`.

Příklad 1.2.11: Napište funkci, která o zadaném přirozeném čísle rozhodne, jestli je mocninou dvojky.

Rekurze

Příklad 1.2.4: Definujte v Haskellu funkci `dfct` (v kombinatorice někdy značenou !!), kde

$$\begin{aligned} 0!! &= 1, \\ (2n)!! &= 2 \cdot 4 \cdots (2n), \\ (2n+1)!! &= 1 \cdot 3 \cdots (2n+1) \end{aligned}$$

Zkuste funkci naprogramovat vícero způsoby.

Příklad 1.2.12: Co počítá následující funkce? Jak se chová na argumentech, kterými jsou nezáporná čísla? Jak se chová na záporných argumentech?

```
fun :: Integer -> Integer
fun 0 = 0
fun n = fun (n - 1) + 2 * n - 1
```