Cvičení 1

1.1 Priority operátorů, prefixový a infixový zápis

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)`

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

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

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`

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)`

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`

Příklad 1.1.6 Doplňte všechny implicitní závorky do následujících výrazů:

a) `sin pi/2`
b) `f.g x`
c) `2 ^ mod 9 5`
d) `f . (.) g h . id`
e) `2 + div m 18 * m `mod` 7 == m ^ 2 ^ n - m + 11 && m * n < 20`
f) `f 1 2 g + (+) 3 `const` g f 10`

g) `replicate 8 x ++ filter even (enumFromTo 1 (3 + 9 `mod` x))`

Příklad 1.1.7 Zjistěte (bez použití interpretru), na co se vyhodnotí následující výraz. Poté jej přepište do prefixového tvaru a pomocí interpretru ověřte, že se jeho hodnota nezměnila.
`5 + 7 * 5 `mod` 3 `div` 2 == 3 * 2 - 1`

Příklad 1.1.8 Do následujícího výrazu doplňte implicitní závorky a pak převeďte všechny operátory v něm do prefixového tvaru.
`2 + 2 * 3 == 2 * 4 && 8 `div` 2 * 2 == 2 || 0 > 7`

Příklad 1.1.9 Které z následujících výrazů jsou korektní?

a) `((+) 3)`
b) `(3 (+))`
c) `(3+)`
d) `(+3)`
e) `(.(+))`
f) `(+(.))`
g) `(.+)`
h) `(+.)`
i) `.even`
j) `(.((.).))`

1.2 Definice funkcí podle vzoru

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.

Příklad 1.2.2 Definujte rekurzivní funkci pro výpočet faktoriálu.

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

$$0!! = 1,$$
$$(2n)!! = 2 \cdot 4 \cdots (2n),$$
$$(2n+1)!! = 1 \cdot 3 \cdots (2n+1)$$

Příklad 1.2.5 Definujte funkci `linear` tak, že `linear a b` se vyhodnotí na řešení lineární rovnice $ax + b = 0, x \in \mathbb{R}$. Jestliže rovnice nemá právě 1 řešení, tuto skutečnost vypište na výstup pomocí funkce `error`. Před samotnou definicí funkce určete, jaký bude mít typ.

Příklad 1.2.6 Napište funkci `combinatorial` tak, že `combinatorial n k` se vyhodnotí na kombinační číslo $\binom{n}{k}$.

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

Příklad 1.2.8 Definujte funkci `digits`, která po aplikaci na kladné celé číslo vrátí jeho ciferný součet.

Příklad 1.2.9 Napište funkci `mygcd`, která po aplikaci na dvě kladná celá čísla vrátí jejich největšího společného dělitele. Pokuste se o co nejefektivnější implementaci.

Příklad 1.2.10 Definujte funkce `plus` a `times`, které budou ekvivalentní operátorům `(+)` a `(*)` na přirozených číslech. Je zakázáno v implementaci používat vestavěné funkce `(+)` a `(*)`. Můžete však používat libovolné jiné funkce, doporučujeme podívat se zejména na funkce `pred` a `succ` (jejich typ je ve skutečnosti o něco obecnější, ale můžete uvažovat, že to je `Integer -> Integer`).

Bonus: implementujte funkce `plus'` a `times'`, které budou fungovat na všech celých číslech.

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

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
```

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

1.3 Lokální definice a globální definice

Příklad 1.3.1 Výraz `((3+4)/2) * ((3+4)/2 - 3) * ((3+4)/2 - 4)`

a) upravte s využitím syntaktické konstrukce pro lokální definici (`let ... in`) tak, aby se v něm neopakovaly stejné složené podvýrazy;
b) upravte stejně, ovšem s využitím globálních definic (uložených v externím souboru).

Příklad 1.3.2 Napište funkci `flipNum :: Integer -> Integer`, která vrátí číslo s ciframi v opačném pořadí. Uvažujte pouze dekadickou soustavu. Případné pomocné funkce definujte pouze lokálně.

Řešení

Řešení 1.1.1

a) V prvním výrazu je implicitní závorkování v důsledku priorit operátorů kolem násobení, tj. `5 + (9 * 3)`.

b) Operace umocňování má asociativitu zprava, tedy v případě více výskytů `^` za sebou se implicitně závorkuje zprava. Obecně tedy

```
n ^ n ^ n == n ^ (n ^ n) /= (n ^ n) ^ n == n ^ (n ^ 2)
```

Ale vidíme, že tato rovnost platí pro `n == 2`, tedy to je jediný speciální případ, kdy `n ^ n ^ n == (n ^ n) ^ n`.

c) Tady se setkáváme s případem, kdy je operátor neasociativní, tedy není definováno, jak se výraz parsuje v případě výskytu více relačních operátorů vedle sebe, a je tedy nekorektní. Důvod neasociativity je jednoduchý. Asociativitu má smysl uvažovat jen u operátorů, které mají stejný typ obou operandů a také výsledku. To zřejmě neplatí u operátoru `(==)`, který vyžaduje operandy stejného, ale jinak poměrně libovolného typu, například čísla, Boolovské hodnoty, řetězce, ..., ale výsledek je Boolovská hodnota. Pokud bychom tedy nějak asociativitu definovali, dospěli bychom v některých případech do situace, kdybychom porovnávali Boolovskou hodnotu s hodnotami jiného typu, což v Haskellu nelze.

d) v případě, že explicitně uvedeme závorkování pro relační operátory, dostaneme se do obdobné situace jako v předchozím podpříkladu, tedy porovnání Boolovské hodnoty a čísla, což v Haskellu nelze. Naproti tomu výsledkem porovnání `4 == 4` dostaneme v obou případech Boolovskou hodnotu, a ty mezi sebou porovnávat můžeme, protože jsou stejného typu.

Řešení 1.1.2 V uvedeném pořadí od nejvyšší priority (9) až k nejnižší (1).

Poznámka: Ve skutečnosti existují i operátory s prioritou 0, například `$`, ke kterému se časem dostaneme.

Řešení 1.1.3

a) Podmínkou musí být výraz Boolovského typu (`Bool`), což výraz `5 - 4` není – jde o výraz celočíselného typu.

b) Výrazy v `then` a `else` větvi musí být stejného (nebo kompatibilního) typu, protože celý podmínkový výraz musí mít vždy stejný typ bez ohledu na hodnotu podmínky.

c) Na první pohled podivně vypadající konstrukce, kde výsledkem podmínkového výrazu je prefixově zapsaný operátor `(&&)`, je správná. V Haskellu jsou funkce/operátory výrazy rovnocennými s číselnými či jinými konstantami. Problémem je chybějící větev `else`. Podmíněný výraz má syntaktické omezení, že vždy musí obsahovat jak `then`, tak `else` větev, i když by podmínka zaručovala použití jenom jedné z nich. Kdyby podmínka mohla být vyhodnocena na nepravdu a chybělo by `else`, pak by výraz neměl žádnou hodnotu, kterou by vrátil. Ale výraz v Haskellu vždy musí mít nějakou hodnotu.

Řešení 1.1.4 Je důležité zachovávat pořadí operandů! I když jde o komutativní operátor, nelze při změně mezi prefixovým a infixovým zápisem měnit jejich pořadí, protože vzniklý výraz nebude ekvivalentní.

a) `(^) 4 (mod 7 5)`
b) `3 `max` (2 + 3)`

Řešení 1.1.5 Postup implicitního závorkování je u všech výrazů stejný. Je potřeba řídit se prioritou/asociativitou infixově zapsaných operátorů a závorkováním aplikace funkcí na argumenty. Postupujeme následovně:

1. Obsahuje-li výraz infixově zapsané operátory, najdeme ty s nejnižší prioritou (samozřejmě ignorujíce obsah explicitních závorek) a jejich operandy uzávorkujeme. Pokud je těchto operátorů více než jeden, jednotlivé operandy závorkujeme dle jejich asociativity.
2. Nejsou-li ve výrazu infixově zapsané operátory, ale jenom prefixové aplikace funkcí na argumenty, závorkujeme funkci s její argumenty zleva (ve smyslu „částečné aplikace“, bude bráno později), konkrétně například `f 1 2 'd' m` závorkujeme `(((f 1) 2) 'd') m`.

Pokud nejde realizovat ani jeden z těchto kroků, tj. výraz je jednoduchý (konstanta, proměnná nebo název funkce), aplikace funkce na jeden jednoduchý argument nebo aplikace infixově zapsaného binárního operátoru na dva jednoduché argumenty, skončili jsme.

Pokud ne a vzniklé uzávorkované podvýrazy jsou složitější, opět aplikujeme tento postup na všechny rekurzivně.

Řešení jednotlivých příkladů budou následovná:

a) `(recip 2) * 5`
b) `(sin pi) / 2`
c) `(3 `mod` 8) * 2`
d) `(f g 3) + (g 5)` (funkce `f` je aplikována na 2 argumenty)
e) `(2 + (div m 18) == (m ^ (2 ^ n))) && ((m * n) < 20)`
f) `((flip (.)) snd) . (id const)`

Řešení 1.1.6

a) `(sin pi) / 2`
b) `f . (g x)`
c) `2 ^ (mod 9 5)`
d) `f . (((.) g h) . id)`
e) `((2 + ((((div m) 18) * m) `mod` 7)) == (((m ^ (2 ^ n)) - m) + 11))`
   `&& ((m * n) < 20)`
f) `(((f 1) 2) g) + (((+) 3) `const` ((g f) 10))`
g) `((replicate 8) x) ++ ((filter even) (enumFromTo 1 (3 + (9 `mod` x))))`

Řešení 1.1.7 Nejdříve za pomoci tabulky priority operátorů do výrazu zapíšeme implicitní závorky (kvůli různým prioritám operátorů):

`(5 + (((7 * 5) `mod` 3) `div` 2)) == ((3 * 2) - 1)`

Pak už lehce zjistíme, že výraz se vyhodnotí na `False`.

Při vyhodnocování výrazu v zadání se jako poslední vyhodnotí funkce s nejnižší prioritou, v našem případě `(==)`. Přepíšeme tedy do prefixu nejdříve tuto funkci:

```
(==) (5 + 7 * 5 `mod` 3 `div` 2) (3 * 2 - 1)
```

Následně v každém z argumentů opět najdeme funkci s nejnižší prioritou – v prvním je to funkce `(+)`, ve druhém pak `(-)`. Přepisem těchto funkcí do prefixu dostaneme:

```
(==) ((+) 5 (7 * 5 `mod` 3 `div` 2)) ((-) (3*2) 1)
```

Stejným způsobem pokračujeme i nadále. Jestliže narazíme na skupinu operátorů se stejnou prioritou (například `(*), mod, div`), ověříme si jejich směr sdružování (závorkování). V našem případě se sdružuje (závorkuje) zleva. To v praxi znamená, že jako poslední se vyhodnotí funkce `div`. Výraz tedy přepíšeme následovně:

```
div (7 * 5 `mod` 3) 2
```

Stejným způsobem pokračujeme, dokud nám nezůstanou žádné infixově zapsané operátory:

```
(==) ((+) 5 (div (mod ((*) 7 5) 3) 2)) ((-) ((*) 3 2) 1)
```

Řešení 1.1.8

```
(||) ((&&) ((==) ((+) 2 ((*) 2 3)) ((*) 2 4)) ((==) ((*) (div 8 2) 2) 2))
     ((>) 0 7)
```

Řešení 1.1.9

a) Korektní, „přičítačka“ tří.
b) Nekorektní, `3` je interpretována jako funkce beroucí `(+)` jako svůj argument.
c) Korektní, „přičítačka“ tří, ekvivalentní funkci `f x = 3 + x`.
d) Korektní, „přičítačka“ tří, ekvivalentní funkci `f x = x + 3`.
e) Korektní, ekvivalentní funkci `f x y = x ((+) y)`, například `f id 3 2` $\rightsquigarrow^*$ `5`.
f) Nekorektní, ekvivalentní s funkcí `f x = x + (.)`, avšak funkce není možné sečítat, typová chyba.
g) Nekorektní, není možné rozlišit, kterého operátoru to je operátorová sekce. Interpretuje se jako prefixová verze operátoru `(.+)`.
h) Nekorektní, viz předešlý příklad.
i) Nekorektní, zřejmě zamýšlená operátorová sekce `(.)`, avšak závorky okolo operátorové sekce není možné vynechat.
j) Korektní, dvojnásobné použití operátorové sekce – vnější je pravá, vnitřní je levá. Ekvivalentní se všemi následujícími funkcemi.

```
f1 x = x . ((.).)
f2 x y = x (((.).) y)
f3 x y = x ((.) . y)
f4 x y = x (\z -> (.) (y z))
f5 x y = x (\z w q -> y z (w q))
```

Řešení 1.2.1

a)
```
logicalAnd :: Bool -> Bool -> Bool
logicalAnd x y = if x then y else False
```

b)
```
logicalAnd' :: Bool -> Bool -> Bool
logicalAnd' True True = True
logicalAnd' _    _    = False
```

Řešení 1.2.2

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

Funkce je definována po částech a předpokládáme, že dostane jako argument jenom nezáporné celé číslo. Jako první bázový případ je 0, kdy přímo vrátíme výsledek. V opačném případě použijeme druhý rekurzivní případ, kdy víme, že $n! = n \times (n-1)!$. Poznamenejme, že závorky kolem `n - 1` je nutno použít, protože jinak by se výraz implicitně uzávorkoval jako `(fact n) - 1`, protože aplikace funkcí na argumenty má vyšší prioritu než operátory.

Řešení 1.2.3 Nejjednodušším, i když ne nejefektivnějším způsobem úpravy je doplnit kontrolu nezápornosti argumentu v druhém případě definice funkce, tedy:

```
z `power` n = if n < 0 then error "negative!" else z * (z `power` (n-1))
```

Neefektivita spočívá v tom, že kontrolu nezápornosti stačí udělat pouze jednou na začátku, protože pak už nelze dosáhnout záporného čísla. V našem případě se kontrola provede zbytečně n-krát.

Řešení 1.2.4

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

Řešení 1.2.5 Musíme rozlišit následující tři případy:

a) rovnice má nekonečně mnoho řešení ($a = b = 0$)
b) rovnice nemá řešení ($a = 0 \wedge b \neq 0$)
c) rovnice má právě jedno řešení tvaru $\frac{-b}{a}$ ($a \neq 0$)

Funkci zadefinujeme podle vzoru následovně (pozor na pořadí vzorů!).

```
linear :: Double -> Double -> Double
linear 0 0 = error "Infinitely many solutions"
linear 0 _ = error "No solution"
linear a b = -b/a
```

Řešení 1.2.6 Využijeme funkci `fact` definovanou na cvičeních. Jelikož pracujeme s celými čísly, musíme použít celočíselné dělení místo reálného (jinak bychom porušili deklarovaný typ).

```
combinatorial :: Integer -> Integer -> Integer
combinatorial n k = div (fact n) ((fact k) * (fact (n-k)))
```

Další možností, jak tuto funkci vyřešit je vyjít z Pascalova trojůhelníku:

```
combinatorial :: Integer -> Integer -> Integer
combinatorial _ 0 = 1
combinatorial 0 _ = 1
combinatorial x y = if x == y
                    then 1
                    else combinatorial (x-1) (y-1) + combinatorial (x-1) y
```

Zkuste se zamyslet nad složitostí těchto řešení, následně zkuste experimentálně zjistit, pro jak velké vstupy začne být rozdíl ve složitosti pozorovatelný.

Řešení 1.2.7 Jak víme, počet kořenů kvadratické rovnice závisí na hodnotě diskriminantu – pro záporný diskriminant rovnice řešení nemá, pro nulový má právě jedno a pro kladný právě dvě. Funkci můžeme definovat pomocí podmíněného výrazu a lokální definice například následovně:

```
roots :: Double -> Double -> Double -> Int
roots a b c = if d < 0 then 0
                       else if d == 0 then 1 else 2
    where d = b ^ 2 - 4 * a * c
```

S využitím funkce `signum` můžeme naše řešení značně zkrátit – zamyslete se, proč je to možné. (Poznámka: funkce `signum` vrátí výsledek stejného typu, jako je její argument, musíme tedy použít některou ze zaokrouhlovacích funkcí pro převod `Double` na `Int`.)

```
roots' :: Double -> Double -> Double -> Int
roots' a b c = floor (1 + signum (b ^ 2 - 4 * a * c))
```

Řešení 1.2.8

```
digits :: Integer -> Integer
digits 0 = 0
digits x = x `mod` 10 + digits (x `div` 10)
```

Řešení 1.2.9 K řešení použijeme známý Euklidův algoritmus, konkrétněji jeho rekurzivní verzi využívající zbytky po dělení.

```
mygcd :: Integer -> Integer -> Integer
mygcd x y = if y == 0 then x
                      else mygcd (min x y) ((max x y) `mod` (min x y))
```

Řešení 1.2.10

```
plus :: Integer -> Integer -> Integer
plus 0 y = y
plus x y = plus (pred x) (succ y)

times :: Integer -> Integer -> Integer
times 0 y = 0
```

```
times x 0 = 0
times 1 y = y
times x y = plus y (times (pred x) y)

plus' :: Integer -> Integer -> Integer
plus' x y = if x >= 0 then plus x y
                      else negate (plus (negate x) (negate y))

times' :: Integer -> Integer -> Integer
times' x y = if (x < 0 && y < 0) || (x >= 0 && y >= 0)
                then times (abs x) (abs y)
                else negate (times (abs x) (abs y))
```

Řešení 1.2.11 Tuto úlohu je možno řešit různými způsoby. Asi nejpřímočařejší je porovnávat hodnotu se všemi mocninami dvou, které ji nepřesahují. Je tedy nutné použít ještě pomocnou binární funkci, která bude mít aktuálně zpracovávanou mocninu jako svůj druhý argument.

```
power2 :: Integer -> Bool
power2 x = power2' x 1
power2' :: Integer -> Integer -> Bool
power2' x y = if y > x then False
                       else if x == y then True else power2' x (y * 2)
```

Existují však i mnohem efektivnější řešení. Jedno z nich, postavené na logaritmech, je uvedeno níže; musíme však vzít v úvahu, že funkce `log` očekává jako svůj parametr desetinné číslo, proto musíme `x` konvertovat pomocí funkce `fromIntegral`:

```
power2 :: Integer -> Bool
power2 x = 2 ^ (floor (log (fromIntegral x) / log 2)) == x
```

Řešení 1.2.12 Nejsnáze to lze zjistit výpočtem několika prvních hodnot. Pak lze snadno odhalit zákonitost a formulovat hypotézu, že `fun n == n ^ 2` pro nezáporná `n`. Následně je potřeba tuto hypotézu dokázat, což lze provést matematickou indukcí. Důkaz přenecháváme čtenáři.

Na tuto definici lze taky nahlížet tak, že všechny druhé mocniny se dají rozepsat jako součet lichých čísel nepřevyšujících dvojnásobek argumentu.

V případě záporných čísel lze ručním vyhodnocením zjistit, že vyhodnocování se „zacyklí“ na druhém řádku definice a bude postupně voláno pro nižší a nižší záporná čísla. Definice totiž neposkytuje žádný zastavovací případ, který by rekurzi ukončil.

Řešení 1.2.13

```
mod3 0 = 0
mod3 1 = 1
mod3 2 = 2
mod3 x = mod3 (x - 3)
```

Řešení 1.3.1

a) `let t = (3+4)/2 in t * (t - 3) * (t - 4)`

b) Soubor bude obsahovat řádek `t = (3 + 4) / 2` a v interpretru pak lze po načtení tohoto souboru zadat k vyhodnocení výraz `t * (t - 3) * (t - 4)`.

Řešení 1.3.2 Jsou dvě varianty – první funguje jen pro nezáporná čísla (zkuste si ji rozšířit), druhá je pravděpodobně efektivnější (přičemž to záleží i na implementaci `read`).

```
flipNum :: Integer -> Integer
flipNum x = read (reverse (show x))

flipNum' :: Integer -> Integer
flipNum' x = fromDigits 0 (reverse (toDigits x []))
  where
    toDigits 0 xs = xs
    toDigits n xs = toDigits (n `div` 10) (n `mod` 10 : xs)
    fromDigits n []     = n
    fromDigits n (x:xs) = fromDigits (n * 10 + x) xs
```

Na druhou stranu je první verze podstatně elegantnější, což se cení. Navíc ji lze ještě zlepšit na `flipNum = read . reverse . show`.