

Cvičení 7: Vstup a výstup (bonus)



Pan Sazeč dává na vědomí: Na vstup a výstup historicky bylo cvičení, ale rozhodli jsme se zaměřit se více na věci, které ukazují typické funkcionální principy. Proto je tohle cvičení označené jako bonusové – nebude k němu žádné reálné cvičení ani konzultace, jen tyto materiály. Pokud máte k těmto příkladům dotazy, zeptejte se například na diskusním fóru. Stejně tak na cvičení nenavazuje žádný domácí úkol.

Cvičení bylo víceméně ponecháno v původním stavu, včetně označení které příklady byly určeny k práci na cvičení.

Před bonusovým vstupně-výstupním cvičením je zapotřebí:

znát význam pojmů *vstupně-výstupní akce* a *vnitřní výsledek*;

rozumět, co znamenají typy jako `IO Integer` a `IO ()`, a chápat jejich odlišnost od `Integer` a `()`;

vědět, k čemu slouží funkce:

```
pure      :: a -> IO a
getLine   :: IO String
putStrLn  :: String -> IO ()
readFile  :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

být seznámeni s operátory pro skládání vstupně-výstupních akcí:

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

být seznámeni se syntaxí a významem konstrukce `do`:

```
main = do input <- getLine
         let hello = "Hello there, "
         putStrLn (hello ++ input)
```

umět přeložit zdrojový kód na spustitelný program pomocí `ghc zdroják`.

Vstup a výstup v Haskellu je na první pohled zvláštní z prostého důvodu – vyhodnocení výrazu nemůže mít vedlejší efekty, jako je výpis na obrazovku. Je ale možné výraz vyhodnotit na „scénář“ obsahující popis vstupně-výstupních operací: kdy se má načítat vstup, co se má vkládat do kterých souborů apod. Při samotném vyhodnocení se sice nic nestane, ale podle výsledného scénáře pak může vstup a výstup provádět třeba interpret.

Těmto scénářům říkáme *vstupně-výstupní akce*. Hodnota typu `IO a` je pak scénář pro získání `a`. Zde `a` je typ *vnitřního výsledku* akce, který je naplněn po vykonání akce. Vnitřní výsledek (například načtený řetězec) nemůže vstupovat do vyhodnocování výrazů, ale jiné akce jej používat mohou (například ho vypsát na obrazovku).

K takovému řetězení scénářů do větších akcí slouží operátor `>>=`. Povšimněte si jeho typu: levý argument je „scénář pro získání `a`“, pravý zjednodušeně dává „scénář, podle kterého se z `a` vyrobí `b`“¹.

¹Ve skutečnosti spíše „jak z `a` vyrobit scénář pro `b`“. Pro zkrácení ale můžeme např. funkci typu

Akce jako hodnoty jsou pro nás zcela neprůhledné – nemáme možnost zjistit, jaké efekty bude akce mít, aniž ji spustíme (a tedy všechny efekty provedeme). *Spouštění* je kromě spojování přes `>>=` v podstatě jediná zajímavá věc, kterou s akcemi můžeme dělat. Spouští se každá akce, kterou si necháme vyhodnotit v interpretu, a v samostatných spustitelných souborech akce `main :: IO ()`.

Jindřiška varuje: Vstup a výstup je matoucí, pokud důsledně nerozlišujeme mezi pojmy *hodnota*, (*vstupně-výstupní*) *akce* a *vnitřní výsledek akce*. Plést si nesmíme ani *vyhodnocení* a *spuštění*.

7.1 Skládání akcí operátorem `>>=`

- Př. 7.1.1** Pomocí známých funkcí a operátoru `>>=` definujte vstupně-výstupní akci, která po spuštění přečte řádek ze standardního vstupu a:
- » a) vypíše jej beze změny na standardní výstup;
 - » b) vypíše jej pozpátku;
 - » c) vypíše jej, není-li prázdný, jinak vypíše „<empty>“;
 - » d) vypíše jej a také jej uloží jako vnitřní výsledek akce (bude se hodit i operátor `>>`).
- Př. 7.1.2** Bez použití *do*-notace definujte akci `getInteger :: IO Integer`, která ze standardního vstupu načte celé číslo. Využijte knihovni funkci `read :: (Read a) => String -> a`.
- Př. 7.1.3** Bez použití *do*-notace definujte vstupně-výstupní akci `loopecho :: IO ()`, která při spuštění načítá a vypisuje řádky do té doby, než načte prázdný řádek.
- Př. 7.1.4** Napište akci `getSanitized :: IO String`, která načte jeden řádek textu od uživatele a z něj odstraní všechny znaky, které nejsou znaky abecedy. V úkolu použijte funkci `isAlpha` z modulu `Data.Char`.
- Př. 7.1.5** Bez použití *do*-notace definujte akci, která ze standardního vstupu přečte cestu k souboru a následně uživateli oznámí, zda zadaný soubor existuje. Úkol řešte s využitím `doesFileExist` z modulu `System.Directory`.
- Př. 7.1.6** Definujte funkci `(>>)` pomocí funkce `(>>=)`.
- Př. 7.1.7** Napište funkce `runLeft :: [IO a] -> IO ()` a `runRight :: [IO a] -> IO ()`, které spustí všechny akce v zadaném seznamu postupně zleva (respektive zprava).

7.2 IO pomocí *do*-notace, převody mezi notacemi

`String -> IO Integer` označovat jako „akci, která bere řetězec a vrací číslo“, byť formálně správně se jedná o „funkci, která bere řetězec a vrací vstupně-výstupní akci s vnitřním výsledkem typu číslo“.

<i>do</i> -notace	<i>bind</i> -notace
<code>do f g</code>	<code>f >> g</code>
<code>do x <- f g</code>	<code>f >>= \x -> g</code>
<code>do let x = y f</code>	<code>let x = y in f</code>

Př. 7.2.1 Vraťte se k některému ze svých řešení příkladů z předchozí sekce a přepište je na *do*-notaci. Pro ilustraci zkuste přepsat jedno snadné a jedno mírně složitější řešení a srovnejte je s řešením pomocí `>>=`.



Př. 7.2.2 Naprogramujte funkci `leftPadTwo :: IO ()`, která od uživatele načte dva řetězce a pak je vypíše na obrazovku zarovnaný doprava tak, že před kratší z nich vypíše ještě vhodný počet mezer. Použijte notaci *do*.



Př. 7.2.3 Převedte následující program v *do*-notaci na notaci s použitím `>>=`.



```
main = do
  f <- getLine
  s <- getLine
  appendFile f (s ++ "\n")
```

Př. 7.2.4 Následující funkci přepište do tvaru, ve kterém nepoužijete konstrukci *do*. Určete také typ funkce.

```
query question = do putStrLn question
  answer <- getLine
  pure (answer == "ano")
```

Př. 7.2.5 Funkci `query` z předchozího příkladu dále vylepšete tak, aby:




- Rozlišovala kladné i záporné odpovědi a při nekorektní nebo nerozpoznané odpovědi otázku opakovala.
- Akceptovala odpovědi s malými i velkými písmeny, interpunkcí, případně ve více jazycích.

Př. 7.2.6 U každého z následujících výrazů určete typ a význam, případně vysvětlete, proč není korektní:




- | | |
|---|---|
| a) <code>getLine</code> | g) <code>do let x = getLine
pure x</code> |
| b) <code>x = getLine</code> | h) <code>do let x = getLine
x</code> |
| c) <code>let x = getLine in x</code> | i) <code>do x <- getLine
pure x</code> |
| d) <code>let x <- getLine in x</code> | j) <code>do x <- getLine
x</code> |
| e) <code>getLine >>= \x -> pure x</code> | |
| f) <code>getLine >>= \x -> x</code> | |

7.3 Vstupně-výstupní programy


Př. 7.3.1 Vytvořte a spusťte program, který se při chování jako akce `leftPadTwo` z příkladu 7.2.2.  Připomínáme, že zdrojový kód se na spustitelný soubor překládá programem `ghc` a musí mít zdefinovanou akci `main :: IO ()`. Výsledný program se jmenuje jako zdrojový soubor bez přípony `.hs` a je nutné ho spouštět pomocí `./program`.

Př. 7.3.2 Upravte a doplňte následující zdrojový kód tak, aby program vyžadoval a načetl postupně tři celá čísla a o nich určil, zda mohou být délkami hran trojúhelníku. Akce `getInteger` pochází z příkladu 7.1.2. Nezdávejte se kód refaktorovat a vytvořit si další pomocné funkce či akce.

```
main :: IO ()
main = do putStrLn "Enter one number:"
         x <- getInteger
         putStrLn (show (1 + x))
```

Př. 7.3.3 Napište program, který vyzve uživatele, aby zadal jméno souboru, a poté ověří, že zadaný soubor existuje. Pokud existuje, vypíše jeho obsah na obrazovku, pokud ne, informuje o tom uživatele. Úkol řešte s využitím `doesFileExist` z modulu `System.Directory`. 


Př. 7.3.4 Vysvětlete význam a rizika rekurzivního použití akce `main` v následujícím programu.


```
 main :: IO ()
main = do putStr "Enter string: "
         s <- getLine
         if null s then putStrLn "You shall not pass!"
         else do putStrLn (reverse s)
               main
```

Př. 7.3.5 V dokumentaci naleznete vhodnou funkci typu `Read a => String -> Maybe a` a s jejím využitím napište funkci `requestInteger :: String -> IO (Maybe Integer)` použitelnou na chytřejší načítání čísel. Akce `requestInteger delim` od uživatele čte řádky tak dlouho, než dostane řetězec `delim` (potom vrátí `Nothing`), nebo celé číslo (které vrátí zabalené v `Just`). Po každém neúspěšném pokusu by měl uživatel dostat výzvu k opětovnému zadání čísla.

Př. 7.3.6 S využitím akce z předchozího příkladu napište program, který která ze standardního vstupu čte řádky s čísly, dokud nenarazí na prázdný řádek, potom vypíše jejich aritmetický průměr. Vyřešte úlohu:

- s ukládáním čísel do seznamu;
- bez použití seznamu.

Př. 7.3.7 Napište program `leftPad`, který je obecnější variantou akce z příkladu 7.2.2 a zarovnává libovolný počet řádků na vstupu, dokud nenalezne řádek obsahující jen tečku.² Řádky jsou opět zarovnány doprava na délku nejdelšího.  Následně proveďte drobnou optimalizaci: prázdné řádky nechte prázdnými, tj. bez zbytečných výplňových mezer.

Př. 7.3.8 Upravte program `07_guess.hs` tak, aby parametry funkce `guess` četl z příkazové řádky.  Vhod může přijít modul `System.Environment`.

²Proč zrovna tečku, ptáte se? Bylo nebylo... [https://en.wikipedia.org/wiki/ed_\(text_editor\)](https://en.wikipedia.org/wiki/ed_(text_editor)), https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol#SMTP_transport_example

Př. 7.3.9 Vymyslete a naprogramujte několik triviálních programků manipulujících s textovými soubory:

»=

- počítání řádků
- výpis konkrétního řádku podle zadaného indexu
- vypsání obsahu pozpátku
- seřazení řádků, ...

Definice alternativně přepište s a bez pomoci syntaktické konstrukce `do`.

Př. 7.3.10 Představme si bohy zatracený svět, v němž neexistuje konstruktor `IO` ani vstupně-výstupní akce, které je nutné spouštět. Místo toho k vedlejším efektům dochází rovnou při vyhodnocování výrazů. Např. `getLine' :: String` se vyhodnotí na řádek vstupu. Vyhodnocovací strategie ale funguje stejně, jak ji v Haskellu známe. Co bude výsledkem úplného vyhodnocení výrazu `(getLine', getLine')`?



*
**

Na konci cvičení byste měli zvládnout:

napsat v Haskellu jednoduchý program pracující se vstupem od uživatele a vypisující informace na výstup;

za pomoci dokumentace napsat jednoduchý program pracující se soubory;

převádět mezi konstrukcí `do` a operátory `>>=` a `>>`.

Zdá se, že vstup a výstup vyžaduje velmi odlišný přístup, speciální operátory a syntaxi a celé to může působit velmi nehaskellovsky. Ve skutečnosti jsou ale podobné konstrukce Haskellu vlastní a běžně se používají i mimo `IO`! Napovídat tomu může typ operátoru `>>=`, který zmiňuje jakousi `Monad`. *Monáda* je velmi abstraktní, avšak fascinující koncept, o němž se můžete více dozvědět v jarním semestru v navazujících předmětech IB016 *Seminář z funkcionálního programování* a IA014 *Advanced Functional Programming*.

Řešení

Řeš. 7.1.1 a) `f = getLine >>= \s -> putStrLn s`
`f' = getLine >>= putStrLn`
b) `f = getLine >>= putStrLn . reverse`
c) `f = getLine >>= \s -> putStrLn $ if null s then "<empty>" else s`
d) `f = getLine >>= \s -> putStrLn s >> pure s`

Řeš. 7.1.2 `getInteger :: IO Integer`
`getInteger = getLine >>= \num -> pure (read num :: Integer)`

Řeš. 7.1.3 `loopecho = getLine >>= \s ->`
`if null s then pure ()`
`else putStrLn s >>`
`loopecho`

Řeš. 7.1.4 `import Data.Char`
`getSanitized :: IO String`
`getSanitized = getLine >>= pure . filter isAlpha`

Řeš. 7.1.5 `import System.Directory`
`checkFile :: IO ()`
`checkFile = putStr "File: " >> getLine >>=`
`doesFileExist >>= \b ->`
`putStrLn $ if b then "File exists." else "No such file."`

Řeš. 7.1.6 `x >> f = x >>= _ -> f`

Řeš. 7.1.7 `runLeft = foldl (\a b -> a >> b) (pure ())`
`runLeft' = foldr (\a b -> a >> b) (pure ())`
`runRight = foldl (\a b -> b >> a) (pure ())`
`runRight' = foldr (\a b -> b >> a) (pure ())`

Všimněte si, že nezáleží na výběru akumulární funkce. Aplikací funkcí `foldl` a `foldr` sice vzniknou jiné stromy aplikací operátorů `>>`, ale pořadí listů zůstává stejné a operace `>>` je asociativní. Pro lepší názornost si nakreslete obrázek příslušných katamorfizmů.

Řeš. 7.2.2 `leftPadTwo :: IO ()`
`leftPadTwo = do`
`str1 <- getLine`
`let l1 = length str1`
`str2 <- getLine`
`let l2 = length str2`
`let m = max l1 l2`
`putStrLn (replicate (m-l1) ' ' ++ str1)`
`putStrLn (replicate (m-l2) ' ' ++ str2)`

Řeš. 7.2.3 `main = getLine >>= \f ->`
`getLine >>= \s ->`

```
appendFile f (s ++ "\n")
```

Řeš. 7.2.4 `query' :: String -> IO Bool`
`query' question =`
 `putStrLn question >> getLine >>= \answer -> pure (answer == "ano")`

Nebo lze upravit vzniklou λ -abstrakci na pointfree tvar:

```
query'' :: String -> IO Bool
query'' question =
  putStrLn question >> getLine >>= pure . (== "ano")
```

Řeš. 7.2.5 a) `query2 :: String -> IO Bool`
`query2 question = do`
 `putStrLn question`
 `answer <- getLine`
 `if answer == "ano"`
 `then pure True`
 `else if answer == "ne"`
 `then pure False`
 `else query2 question`

b) `import Data.Char`
`query3 :: String -> IO Bool`
`query3 question = do`
 `putStrLn question`
 `answer <- getLine`
 `let lcAnswer = map toLower answer`
 `if lcAnswer `elem` ["ano", "áno", "yes"] then`
 `pure True`
 `else`
 `if lcAnswer `elem` ["ne", "nie", "no"] then`
 `pure False`
 `else`
 `query3 question`

- Řeš. 7.2.6 a) Typ `IO String`; akce, která při spuštění načte řádek vstupu, jenž se stane vnitřním výsledkem.
- b) Není výrazem. Při zadání do interpretu nebo do souboru se takto zadefinuje nová akce `x :: IO String`, která se chová stejně jako `getLine`.
- c) Ekvivalentní `getLine`; `x` je lokální akce zadefinovaná jako výše.
- d) Syntakticky nesprávné; konstrukci `<-` je lze použít pouze v `do`-bloku nebo intensionálním zápisu seznamu.
- e) Ekvivalentní `getLine`; `x :: String` představuje načtený řádek.
- f) Typově nesprávné; na pravé straně `>>=` má v tomto případě stát funkce typu `String -> IO a`.
- g) Může to být překvapivé, ale výraz je správně utvořený. Má ale poněkud děsivý typ `Monad m => m (IO String)`, což pro účely tohoto cvičení můžeme číst jako `IO (IO String)`. Je to akce, jejímž vnitřním výsledkem je po spuštění akce `getLine`.
- h) Ekvivalentní `getLine`; `x` je opět lokálně zadefinovaná akce.

- i) Ekvivalentní `getLine`; `x :: String` představuje načtený řádek. Analogické k e).
- j) Typově nesprávné; poslední výraz `do`-bloku musí být typu „akce“. Analogické k f).

Řeš. 7.3.1 Do souboru s akcí `leftPadTwo`, necht se jmenuje třeba `Cv07.hs`, přidáme:

```
main :: IO ()
main = putStrLn "Always two lines there are:" >> leftPadTwo
```

Soubor přeložíme a spustíme:

```
$ ghc Cv07.hs
[1 of 1] Compiling Main          ( Cv07.hs, Cv07.o )
Linking Cv07 ...
$ ./Cv07
Always two lines there are:
They're taking the hobbits to Isengard!
Tell me where is Gandalf, for I much desire to speak with him.
      They're taking the hobbits to Isengard!
Tell me where is Gandalf, for I much desire to speak with him.
$
```

Řeš. 7.3.2 `biggest :: Integer -> Integer -> Integer -> Integer`
`biggest x y z = max (max x y) z`

```
lowerTwo :: Integer -> Integer -> Integer -> Integer
lowerTwo x y z = x + y + z - biggest x y z
```

```
getInteger :: IO Integer
getInteger = getLine >>= \num -> pure (read num :: Integer)
```

```
main :: IO ()
main = do
  putStrLn "Enter first number:"
  x <- getInteger
  putStrLn "Enter second number:"
  y <- getInteger
  putStrLn "Enter third number:"
  z <- getInteger
  if biggest x y z < lowerTwo x y z
    then putStrLn "ANO"
    else putStrLn "NE"
```

Řeš. 7.3.3 `import System.Directory`

```
main :: IO ()
main = do putStrLn "Enter filename: "
  fileName <- getLine
  fileExists <- doesFileExist fileName
  if fileExists then do
    fileContents <- readFile fileName
    putStrLn fileContents
  else putStrLn "Impossible. Perhaps the archives are incomplete."
```


Řeš. 7.3.4 Obecně, rekurze v kontextu **IO** umožňuje opakované vykonávání akcí. V tomto případě vidíme, že od první akce, která je součástí `main'`, až po její další výskyt se opakuje načtení řetězce a výpis jeho převrácené podoby. Dobře použitá rekurze musí být vhodným způsobem ukončitelná. Význam kódu měl nejspíš být takový, že zadáme-li prázdný řetězec, dojde k ukončení rekurze a akce se nebude znovu opakovat. Autor zde však rekurzi neuhlídal a `main'` se volá pokaždé; akci tedy není jak ukončit. Náprava je jednoduchá: stačí poslední řádek odsadit tak, aby byl součástí vnořeného `do`-bloku.



V takto jednoduchém případě se místo vnořeného `do`-bloku dá bez ztráty čitelnosti použít `>>`:

```
main :: IO ()
main = do putStr "Enter string: "
          s <- getLine
          if null s then putStrLn "You shall not pass!"
          else putStrLn (reverse s) >> main
```

Řeš. 7.3.5 `import Text.Read`

```
requestInteger :: String -> IO (Maybe Integer)
requestInteger delim = do
  str <- getLine
  if str == delim then pure Nothing else f (readMaybe str)
  where
    f Nothing = requestInteger delim
    f mayInt  = pure mayInt
```

Řeš. 7.3.7 `leftPad :: IO ()`
`leftPad = leftPad' []`

```
leftPad' :: [String] -> IO ()
leftPad' lines = do
  line <- getLine
  if line == "."
    then printLines (maximum (map length lines)) (reverse lines)
    else leftPad' (line:lines)
```

```
printLines :: Int -> [String] -> IO ()
printLines _ [] = pure ()
printLines maxlen (line:xs) = do
  let prefix = replicate (maxlen - length line) ' '
      putStrLn (prefix ++ line)
      printLines maxlen xs
```

Řeš. 7.3.10 Dvojice obsahující v obou složkách též řádek vstupu. Začne se vyhodnocovat první složka dvojice; vyhodnocení `getLine'` způsobí přečtení řádku vstupu, který se stane výsledkem výrazu. Kvůli línému vyhodnocování se už ve druhé složce nemá co vyhodnocovat (a tedy se nenačte další řádek), protože výraz `getLine'` už vyhodnocený je.



Příklad ilustruje jeden z důvodů, proč je v Haskellu potřeba řešit vstup a výstup poněkud neintuitivním způsobem. Dalším důvodem je pořadí spuštění – u vstupně-výstupních akcí ho většinou chceme přesně stanovené. To je u normální redukční strategie obtížnější dosáhnout. Řekněme, že výrazy s vedlejšími efekty se vždy vyhodnocují znovu (aby vůbec došlo k těm

efektům). Například vyhodnocení `getLine'` vždy způsobí, že se načte řádek. Vezměme si hypotetickou funkci `writeFile' :: FilePath -> String -> ()`. Budeme chtít napsat program, který načte název souboru a obsah, který do něj uloží. Které řešení je vezme v jakém pořadí?

```
main =      writeFile' getLine' getLine'
main = flip writeFile' getLine' getLine'
```

Odpověď zní: nevíme, ale v obou bude stejné. Záleží na tom, co `writeFile'` bude chtít načíst první (dá se očekávat, že to bude název souboru). Zároveň s tím `flip` nic neudělá, protože oba argumenty jsou nevyhodnocený výraz (s vedlejšími efekty), a byť už máme zajištěno, že se oba výskyty vyhodnotí zvlášť, redukční strategie stále nevynucuje, aby se to událo před voláním `writeFile'`.



Chceme-li vynutit pořadí vykonávání akcí, můžeme mezi nimi uměle zavést závislost. Například první funkce by vracela kromě načteného řádku i nějaké nahodilé číslo, které by se jako štafetový kolík předalo druhé funkci. Ta by tak mohla být vyhodnocena až poté, co by došlo k vyhodnocení první. Typ by se přitom změnil na `getLine'' :: Integer -> (String, Integer)`. Implementace řetězení funkcí je jistě zřejmá. A protože k takovému řetězení by docházelo často, vymysleli bychom si na něj nějaký hezký operátor a nazvali jej třeba `>>`.

Článek https://wiki.haskell.org/IO_inside názorně krok po kroku ukazuje, že myšlenka štafetového kolíku je už poměrně blízko skutečné implementaci **IO**. Všem nebojácným zájemcům o Haskell ho Pan Fešák doporučuje si přečíst.

Přiložený kód

Pan Sazeč upozorňuje: Kopírování kódu ze souboru PDF nezachovává odsazení! Soubory jsou ale vloženy i jako přílohy tohoto dokumentu; s rozumným prohlížečem je můžete stáhnout kliknutím na název souboru, ať už zde, či v příslušných příkladech. Přílohy jsou k nalezení také ve studijních materiálech v ISu.

 07_guess.hs

```
import Data.Char

main = guess 1 10

query ot = do putStr ot
              ans <- getLine
              pure (ans == "ano")

guess :: Int -> Int -> IO ()
guess m n = do putStrLn ("Mysli si cele cislo od " ++ show m
                        ++ " do " ++ show n ++ ".")
              kv m n

kv m n = do
  if m == n
  then putStrLn ("Je to " ++ show m ++ ".")
  else do o <- query $ "Je tve cislo vetsi nez " ++ show k ++ "? "
         if o then kv (k + 1) n else kv m k
  where k = (m + n) `div` 2
```