Cvičení 6: Vstup a výstup

Před šestým 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 vypsat 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]“.

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 ()`.

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

> **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*.

6.1 Skládání akcí operátorem >>=

Př. 6.1.1 >>= 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. 6.1.2 >>= Bez použití `do`-notace definujte akci `getInteger :: IO Integer`, která ze standardního vstupu načte celé číslo. Využijte knihovní funkci `read :: (Read a) => String -> a`.

Př. 6.1.3 >>= 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. 6.1.4 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. 6.1.5 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. 6.1.6 Definujte funkci `(>>)` pomocí funkce `(>>=)`.

Př. 6.1.7 Napište funkce `runLeft :: [IO a] -> IO ()` a `runRight :: [IO a] -> IO ()`, které spustí všechny akce v zadaném seznamu postupně zleva (respektive zprava).

6.2 IO pomocí do-notace, převody mezi notacemi

do-notace	***bind*-notace**
`do f` `  g`	`f >> g`
`do x <- f` `  g`	`f >>= \x -> g`
`do let x = y` `  f`	`let x = y in f`

Př. 6.2.1 >>= 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. 6.2.2 Naprogramujte funkci `leftPadTwo :: IO ()`, která od uživatele načte dva řetězce a pak je vypíše na obrazovku zarovnány doprava tak, že před kratší z nich vypíše ještě vhodný počet mezer. Použijte notaci `do`.

Př. 6.2.3 Převeďte následující program v `do`-notaci na notaci s použitím `>>=`.

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

Př. 6.2.4 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. 6.2.5 Funkci `query` z předchozího příkladu dále vylepšete tak, aby:

a) Rozlišovala kladné i záporné odpovědi a při nekorektní nebo nerozpoznané odpovědi otázku opakovala.

b) Akceptovala odpovědi s malými i velkými písmeny, interpunkcí, případně ve více jazycích.

Př. 6.2.6 U každého z následujících výrazů určete typ a význam, případně vysvětlete, proč není korektní:

a) `getLine`

b) `x = getLine`

c) `let x = getLine in x`

d) `let x <- getLine in x`

e) `getLine >>= \x -> pure x`

f) `getLine >>= \x -> x`

g)
```
do let x = getLine
   pure x
```

h)
```
do let x = getLine
   x
```

i)
```
do x <- getLine
   pure x
```

j)
```
do x <- getLine
   x
```

6.3 Vstupně-výstupní programy

Př. 6.3.1 Vyrobte a spusťte program, který se při chová jako akce `leftPadTwo` z příkladu 6.2.2. Připomínáme, že zdrojový kód se na spustitelný soubor překládá programem `ghc` a musí mít zadefinovanou akci `main :: IO ()`. Výsledný program se jmenuje jako zdrojový soubor bez přípony `.hs` a je nutné ho spouštět pomocí *`./program`*.

Př. 6.3.2 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 6.1.2. Nezdráhejte 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. 6.3.3 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. 6.3.4 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. 6.3.5 V dokumentaci nalezněte 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. 6.3.6 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:

a) s ukládáním čísel do seznamu;
b) bez použití seznamu.

Př. 6.3.7 Napište program `leftPad`, který je obecnější variantou akce z příkladu 6.2.2 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. 6.3.8 Upravte program `06_guess.hs` tak, aby parametry funkce `guess` četl z příkazové řádky. Vhod může přijít modul `System.Environment`.

Př. 6.3.9 Vymyslete a naprogramujte několik triviálních prográmků 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. 6.3.10 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')`?

*
**

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

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. 6.1.1 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. 6.1.2
```
getInteger :: IO Integer
getInteger = getLine >>= \num -> pure (read num :: Integer)
```

Řeš. 6.1.3
```
loopecho = getLine >>= \s ->
           if null s then pure ()
                     else putStrLn s >>
                          loopecho
```

Řeš. 6.1.4
```
import Data.Char
getSanitized :: IO String
getSanitized = getLine >>= pure . filter isAlpha
```

Řeš. 6.1.5
```
import System.Directory

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

Řeš. 6.1.6
```
x >> f = x >>= \_ -> f
```

Řeš. 6.1.7
```
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 akumulační 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. 6.2.2
```
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. 6.2.3
```
main = getLine >>= \f ->
       getLine >>= \s ->
```

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

Řeš. 6.2.4

```
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. 6.2.5

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. 6.2.6

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. 6.3.1 Do souboru s akcí `leftPadTwo`, nechť se jmenuje třeba `Cv06.hs`, přidáme:

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

Soubor přeložíme a spustíme:

```
$ ghc Cv06.hs
[1 of 1] Compiling Main             ( Cv06.hs, Cv06.o )
Linking Cv06 ...
$ ./Cv06
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. 6.3.2

```
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. 6.3.3

```
import System.Directory

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

Řeš. 6.3.4 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. 6.3.5

```
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. 6.3.7

```
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. 6.3.10 Dvojice obsahující v obou složkách týž 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 byt 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.

06_guess.hs

```haskell
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
```