Wejście - wyjście

Inspiracja

Materiały do ćwiczeń będą oparte w dużej mierze na materiałach dr. Sławomira Bakalarskiego (do tego samego kursu).

Monada IO

Operacje wejścia - wyjścia są realizowane w Haskellu za pomocą *monady IO*. Można o niej myśleć jako o typie danych, który nie tylko może przechowywać dane jakiegoś typu, ale również wykonywać dodatkowe operacje. Bardzo prostym przykładem może być program wypisujący `Hello, world!` na standardowe wyjście. Zapiszmy (np. w pliku `helloworld.hs`) program

```
main = putStrLn "Hello, world!"
```

i skompilujmy za pomocą `ghc helloworld.hs`. Skompilowany program powinien wypisać "Hello, world!" na standardowe wyjście.

putStrLn

Możemy sprawdzić (np. za pomocą `:t` w GHCI), że sygnatura użytej powyżej funkcji to

```
putStrLn :: String -> IO ()
```

Przyjmuje ona `String` i zwraca obiekt typu `IO ()`. Jest to monada IO, która "opakowuje" obiekt typu `()`. W przypadku obiektu `()` jest to jednocześnie jego nazwa i typ, można o nim myśleć jako o pustej wartości (lub analogonie pary (a,b) czy trójki (a,b,c) długości zero).

main

Pisząc własny program, który chcemy skompilować, musimy zawrzeć w nim `main` odpowiadający za akcję wejścia - wyjścia (inaczej GHC zwróci błąd kompilacji). Możemy zdefiniować inne funkcje (lub stałe, `main` w naszym przykładzie jest typu `IO ()`, więc bliżej mu do stałej takiej jak `osiem = 8` niż do funkcji). Nawet jednak, jeśli zdefiniujemy inne funkcje z monadami IO, **nie zostaną one wykonane automatycznie jak main**. Możemy się o tym przekonać dopisując np. `main2` do `helloworld.hs` z innym tekstem, nie zostanie on wypisany. Jeśli chcemy więc, żeby operacja IO zdefiniowana w naszym pliku została wykonana, musi ona w końcu trafić do `main`. Nie oznacza to oczywiście, że musi być w całości w nim zdefiniowana, możemy np. najpierw zdefiniować `main2` wypisujący napis, a potem zdefiniować `main = main2` lub wywołać `main2` w `main`. Jeśli wywołamy polecenie w GHCI, którego zwracaną wartością jest monada IO, zostaną wykonane przypisane jej operacje.

do

Co jeśli chcemy jednak wykonać sekwencję operacji wejścia - wyjścia? Służy do tego blok `do`, w którym możemy zapisać właśnie sekwencję operacji IO, zostaną one wykonane w zadanej kolejności. Przykładowo:

```
main = do
    putStr "Hello,"
    putChar ' '
    putStrLn "world!"
```

również wypisze `Hello, world!`. Użyliśmy przy okazji dwóch nowych funkcji, `putStr`, która wypisuje na standardowe wyjście, ale nie dopisuje znaku nowej linii na koniec oraz `putChar`, która wypisuje pojedynczy znak. Obie zwracają wartość typu `IO ()`. Cały blok `do` jest również formalnie pojedynczą wartością typu `IO` (typ oraz wartość opakowana jest typem i wartością ostatniej operacji IO w bloku).

print

Warto też wspomnieć o funkcji `print`, jest ona po prostu złożeniem `putStrLn.show`, czyli wywołujemy `show`, a na wyniku `putStrLn`. Ponieważ `putStrLn` przyjmuje `String`, to chcąc wypisać np. wartość typu `Integer` wygodniejsze może być użycie `print`. Warto jednak wspomnieć, że funkcja `show` jest zdefiniowana dla typu `String` w taki sposób, że zwraca wartość wraz z otwierającymi i zamykającymi `"`. Program

```
main = print "Hello, world!"
```

wypisze więc na standardowe wyjście `"Hello, world!"` zamiast `Hello, world!`.

getLine

`IO ()` nie jest jedyną możliwością użycia monady IO. Rozpatrzmy "funkcję" (formalnie stałą) wczytującą linię ze standardowego wejścia, ma ona sygnaturę

```
getLine :: IO String
```

czyli `String` opakowany w monadę IO. Rozważmy teraz przykładowy program wchodzący w interakcję z użytkownikiem:

```
przywitajSie :: String -> String
przywitajSie imie = "Czesc " ++ imie ++
 ". Witam serdecznie!"
main = do
         putStrLn "Podaj swoje imie: "
         imie<-getLine
         putStrLn $ przywitajSie imie
         putStrLn $ "Twoje imie ma " ++
          show (length imie) ++ " liter."
```

Przykład

W powyższym przykładzie mamy "zwykłą" funkcję `przywitajSie` oraz `main` w formie bloku `do`. Po wypisaniu na wyjście wiadomości następuje wczytanie linii wejścia do `imie`. Dokładniej, `getLine` zwraca linię z wejścia opakowaną w monadę IO. Przypisujemy wartość wczytanej linii do `imie` za pomocą `<-`. **Uwaga**:

- Operator `<-` nie zadziała w "zwykłej" funkcji (nie będącej częścią IO); GHC nawet zasugeruje, że powinien być wewnątrz bloku `do`, jeśli spróbujemy go niepoprawnie użyć.
- Gdybyśmy wywołali `imie=getLine`, wprowadzilibyśmy tylko alias dla `getLine`, i tak musielibyśmy użyć `<-` aby wydobyć z niego wartość.
- Operatora `<-` możemy też użyć, aby „wydobyć" inne zwracane wartości, nie zawsze jednak ma to sens. Moglibyśmy np. wywołać `puste<-putStrLn "Podaj swoje imie: "`, dostalibyśmy wtedy `puste` z wartością `()`.

Ostatnia wartość w bloku do

Mimo, że możemy przypisywać wartości z monad IO wewnątrz bloku do nawet, jeśli nie mamy zamiaru ich używać, to **nie możemy tego zrobić w ostatnim wyrażeniu w bloku** (tym, od którego cały blok do bierze typ). Ponownie, jeśli spróbujemy, GHC zwróci nam błąd. Nie stanowi to jednak w praktyce problemu, i tak "nie zdążylibyśmy" skorzystać z przypisanej wartości, ponieważ jest to ostatnia operacja w bloku.

return

Spójrzmy na przykładowy kod funkcji, która wypisuje podaną liczbę razy ciąg znaków:

```
displayString :: Int->String->IO ()
displayString 0 _ = return ()
displayString n str = do
                        putStrLn str
                        displayString (n-1) str
main = displayString 5 "ala ma kota"
```

Zwróćmy uwagę na występującą tam funkcję `return`. W przeciwieństwie do funkcji o takiej samej nazwie występującej w C++ czy Javie, ta funkcja nie ma nic wspólnego z kończeniem wykonania. W programie wypisującym `Hello, world!` moglibyśmy dopisać w bloku `do` dowolną liczbę funkcji `return` z różnymi wartościami, a zachowanie programu nie zmieniłoby się w zauważalny sposób. Zwraca ona po prostu wartość podaną w argumencie opakowaną w monadę.

Przykład

Poniższy kod wypisze po prostu `Hello, world!`:

```
main = do
          return "nic"
          return ()
          return (5+6)
          napis<-return "Hello, world!"
          putStrLn napis
          return "SLYCHAC MNIE?"
```

Operacje na plikach

Funkcje odpowiadające za operacje na plikach są w module `System.IO`, który możemy zaimportować poleceniem `import`. Przykładowo:

- Aby otworzyć plik używamy funkcji `openFile :: FilePath -> IOMode -> IO Handle` zwracającej opakowany w monadę IO uchwyt do pliku, gdzie
  - `FilePath` to ścieżka do pliku, absolutna lub względem katalogu bieżącego
  - `IOMode` to tryb otwarcia pliku, możliwe opcje to `ReadMode`, `WriteMode`, `AppendMode`, `ReadWriteMode`.
- Po zakończeniu operacji na pliku zamykamy go za pomocą funkcji `hClose :: Handle -> IO()`.

Operacje na plikach

Mamy dostępne analogiczne funkcje do używanych przy wejściu - wyjściu, tzn.:

- `hPutStrLn :: Handle -> String -> IO ()`,
- `hPutStr :: Handle -> String -> IO ()`,
- `hPutChar :: Handle -> Char -> IO ()`,
- `hPrint :: Show a => Handle -> a -> IO ()`,
- `hGetLine :: Handle -> IO String`.

Wszystkie działają dokładnie analogicznie do już poznanych. Do operowania na plikach przydatna będzie jeszcze jedna, która sprawdza, czy koniec pliku został już osiągnięty: `hIsEOF :: Handle -> IO Bool`. Spójrzmy na przykładowy program operujący na plikach, który wypisuje na standardowe wyjście plik podany jako pierwszy argument.

Program wypisujący plik podany w argumencie

```
import System.IO;
import System.Environment; -- dla funkcji getArgs

showFile :: Handle -> IO ()
showFile handle = do
                    eof<-hIsEOF handle
                    if eof then return ()
                    else do
                           line<-hGetLine handle
                           putStrLn line
                           showFile handle
main = do
         (firstArg:_) <-getArgs
         fileHandle <-openFile firstArg ReadMode
         showFile fileHandle
         hClose fileHandle
```

read

Funkcją pomocną w operowaniu na plikach oraz standardowym wyjściu - wejściu jest

```
read :: Read a => String -> a
```

Służy ona do konwersji typów, których mamy reprezentację w ciągu znaków do oryginalnego typu (np. liczb). Przykładowo, jeśli wiemy, że dany `String` zawiera liczbę naturalną i chcemy ją wysłać do funkcji, która operuje na liczbach naturalnych, możemy użyć `read`. Jeśli automatyczne dopasowanie przez Haskell typu nie wystarczy, możemy napisać wprost, jakiej konwersji oczekujemy, np. `read "5" :: Integer` zwróci 5 typu `Integer`.