Wykład 8

Monada IO i moduły w języku Haskell

Motywacja

Monada IO

Akcje pierwotne

Akcje wtórne

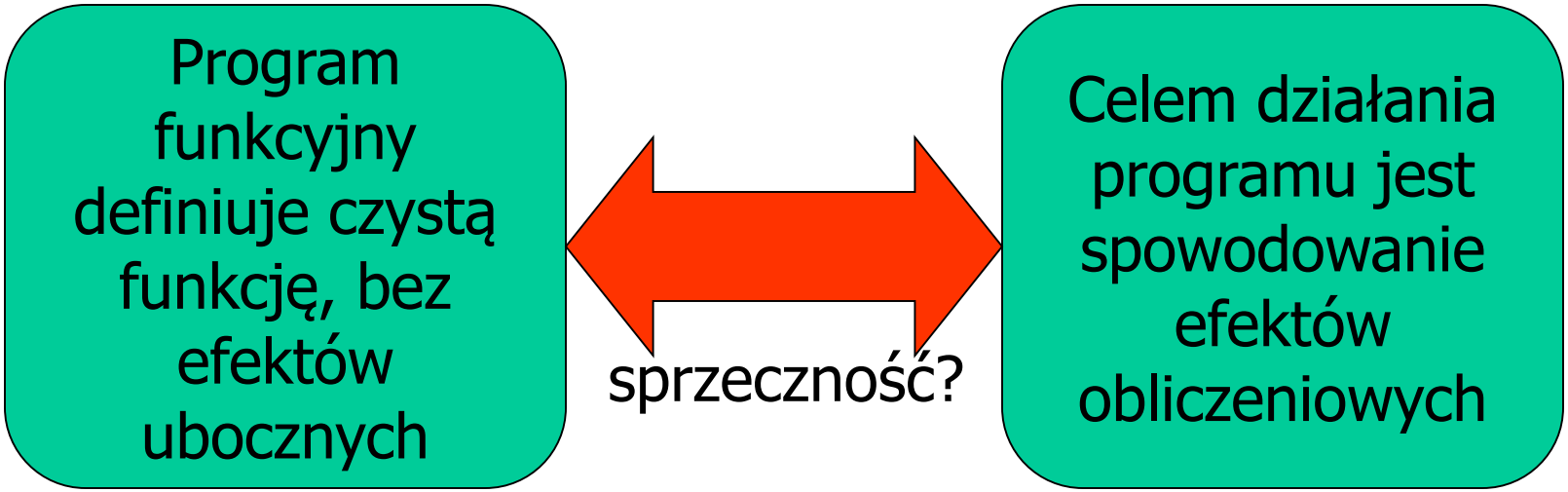
Wyjątki wejścia/wyjścia

Definicja monady

Przykład : zgadywanie słów

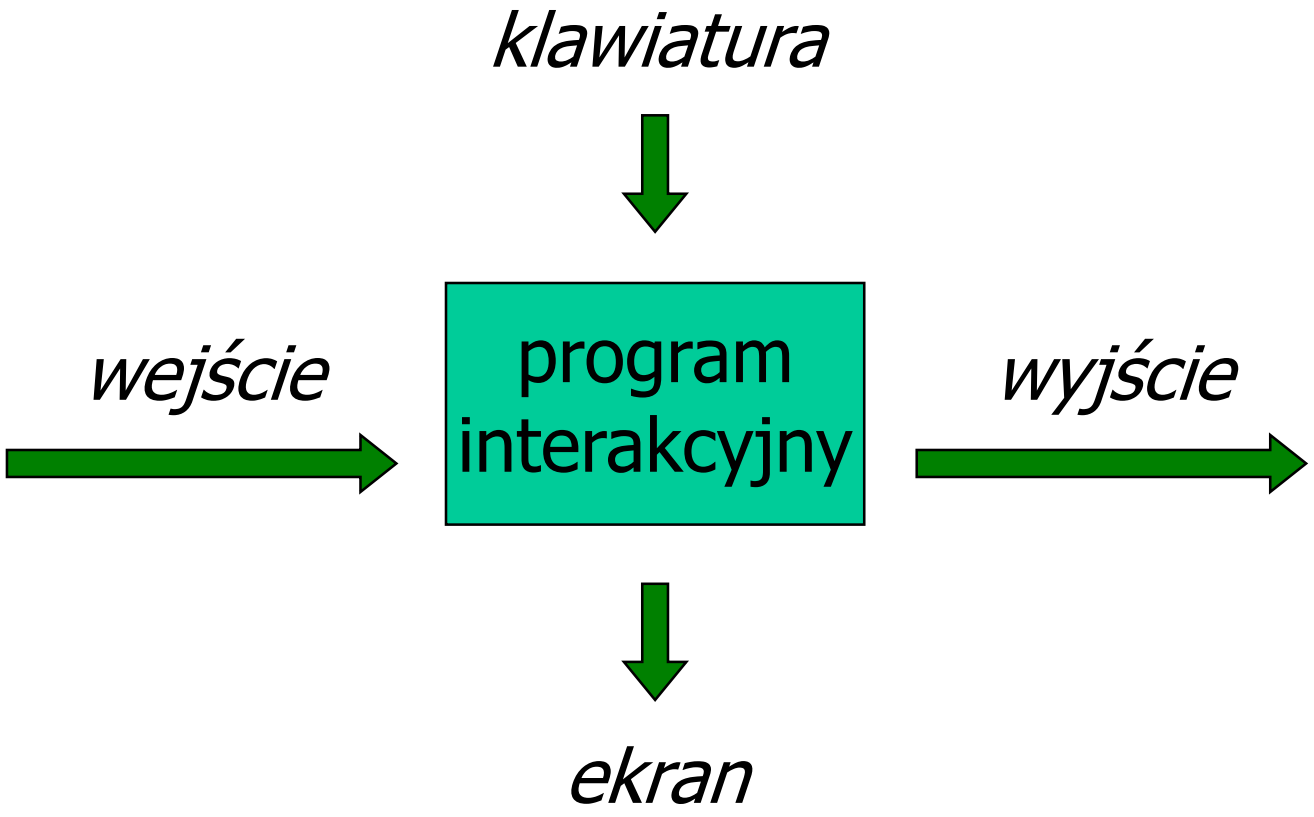
Moduły jako jednostki kompilacj

Problem


Program funkcyjny definiuje czystą funkcję, bez efektów ubocznych
sprzeczność?
Celem działania programu jest spowodowanie efektów obliczeniowych

Do tej pory wszystkie programy w Haskellu były funkcjami. Jednak nawet dla funkcji wyniki były wyświetlane na ekranie monitora.

Co zrobić z programami interakcyjnymi?



Problem

Programy w Haskellu są funkcjami matematycznymi, czyli:

> Programy w Haskellu nie powodują efektów obliczeniowych.

Czytanie z klawiatury i wyświetlanie na ekranie monitora są efektami obliczeniowymi:

> Programy interakcyjne wykorzystują efekty obliczeniowe.

Funkcyjne wejście/wyjście (OCaml)

```
# let witaj()=
  let _ = print_string "Jak masz na imie? "
  in let imie = read_line()
     in print_endline ("Witaj, "^imie^"!");;

val witaj : unit -> unit = <fun>
```

Sekwencja to tylko lukier syntaktyczny.

```
# let witaj'()=
  print_string "Jak masz na imie? ";
  let imie=read_line()
  in print_endline ("Witaj, "^imie^"!");;

val witaj' : unit -> unit = <fun>
```

Rozwiązanie w Haskellu

Programy interakcyjne w Haskellu wymagają użycia specjalnego typu dla odróżnienia „czystych” wyrażeń od „nieczystych” akcji, które mogą powodować efekty obliczeniowe.

```
IO a
```

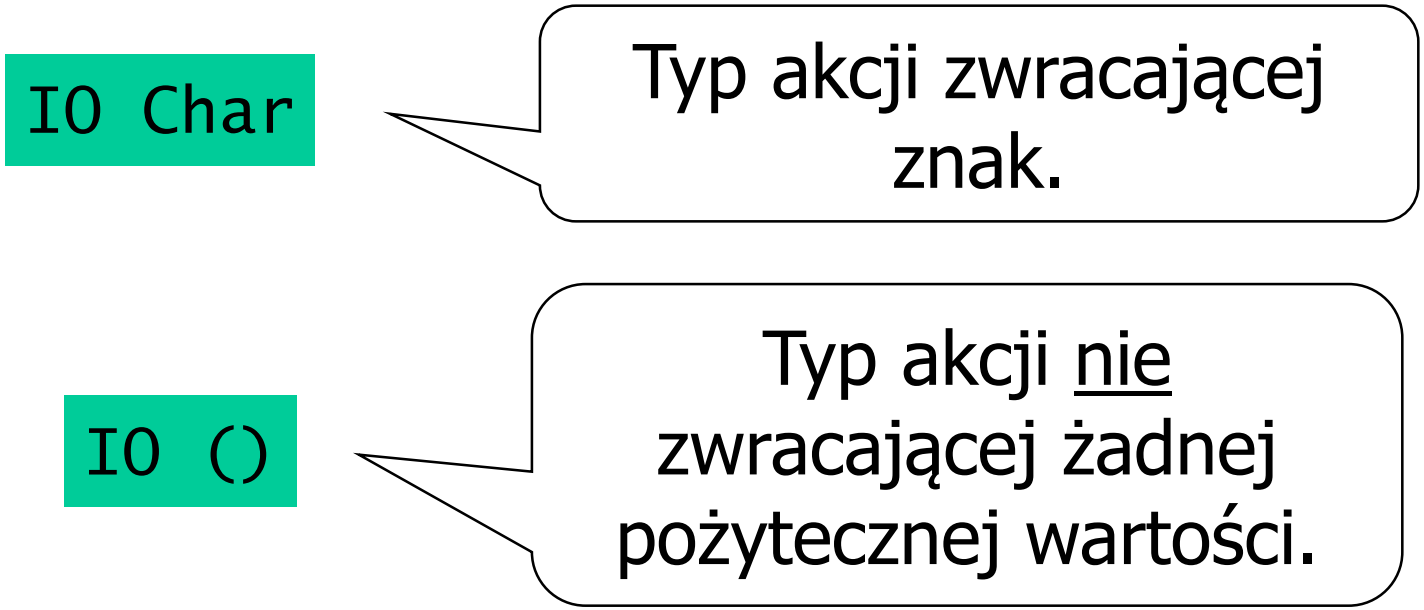
Typ akcji zwracającej wartość typu a.

Monadyczne wejście/wyjście

Wartość typu (IO t) jest „akcją”, która może wykonać operacje we/wy zanim zwróci wynik typu t.

- Monada I/O, wykorzystywana w Haskellu, pośredniczy między wartościami używanymi w języku funkcyjnym, a akcjami charakterystycznymi dla operacji we/wy i ogólniej dla programowania imperatywnego.
- Kolejność ewaluacji wyrażeń w Haskellu jest ograniczona tylko zależnościami między danymi; implementacja ma dużą swobodę w wyborze kolejności ewaluacji.
- W przypadku akcji – w szczególności operacji we/wy – kolejność ich wykonania musi być precyzyjnie określona. Monada I/O w Haskellu pozwala programiście wyspecyfikować kolejność wykonywanych akcji, a implementacja musi ten porządek zachować.

Na przykład:



() jest typem dla „pustej” krotki (bez składowych), czyli

() :: ()

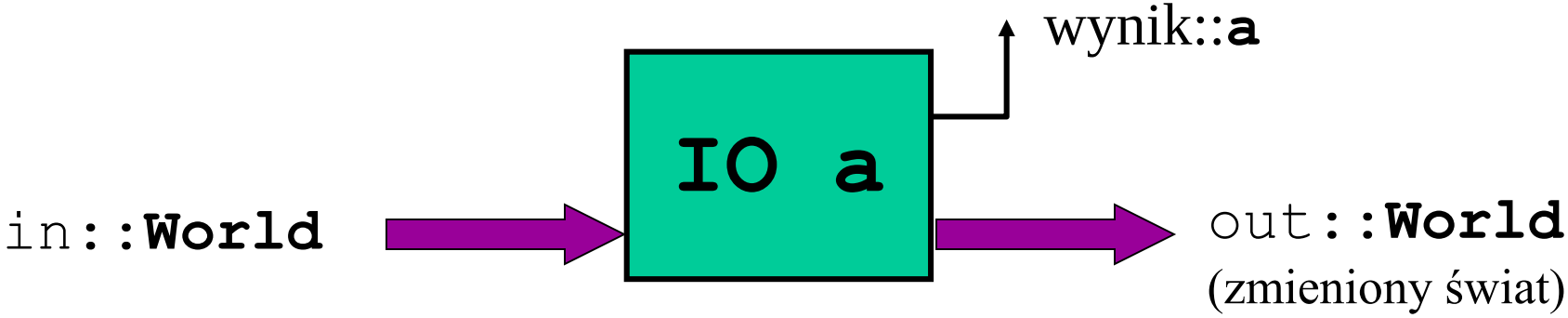
Odpowiednikiem w OCamlu jest typ unit, czyli

() : unit

Poglądowy rysunek

Wartość typu (IO t) jest „akcją”, która może wykonać operacje we/wy zanim zwróci wynik typu t.

```
type IO a = World -> (a, World)
```



Podstawowe akcje

Biblioteka standardowa zawiera wiele akcji, m.in.:

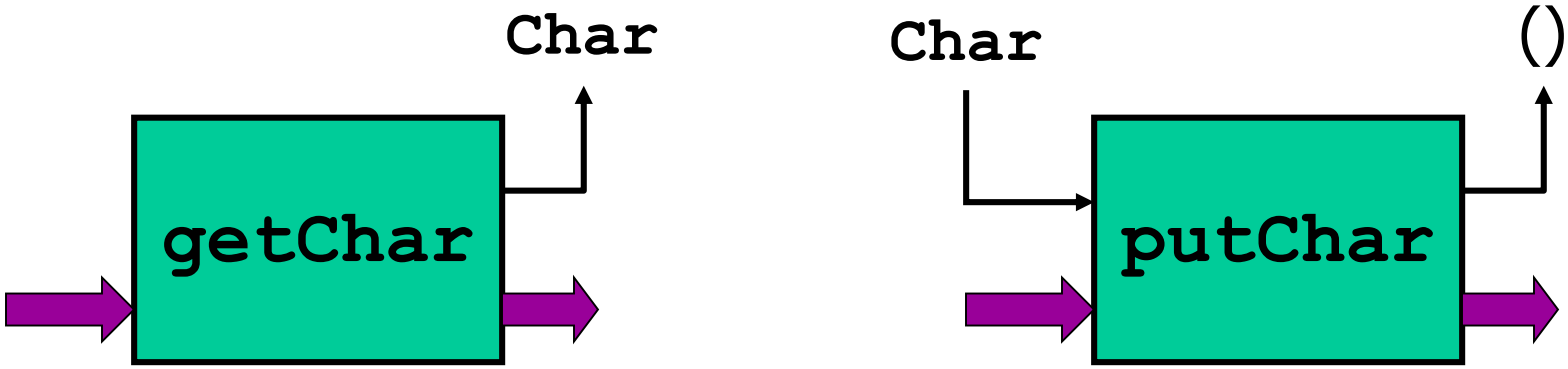
Akcja getChar czyta znak z klawiatury i zwraca go jako wartość wynikową (ale typu IO Char, a nie Char!):

```
getChar :: IO Char
```

Akcja putChar c wyświetla znak c na ekranie, ale nie zwraca żadnej wartości wynikowej (a dokładniej zwraca mało interesującą wartość ()):

```
putChar :: Char → IO ()
```

Proste we/wy

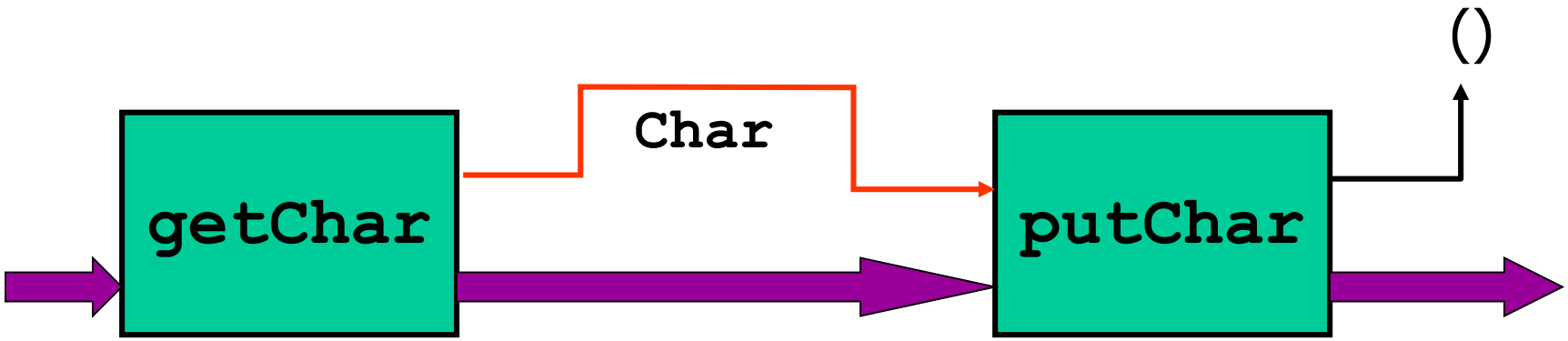



```
getChar :: IO Char
putChar :: Char -> IO ()


main :: IO ()
main = putChar 'x'
```

Kompletny program jest zwykle akcją typu IO ()

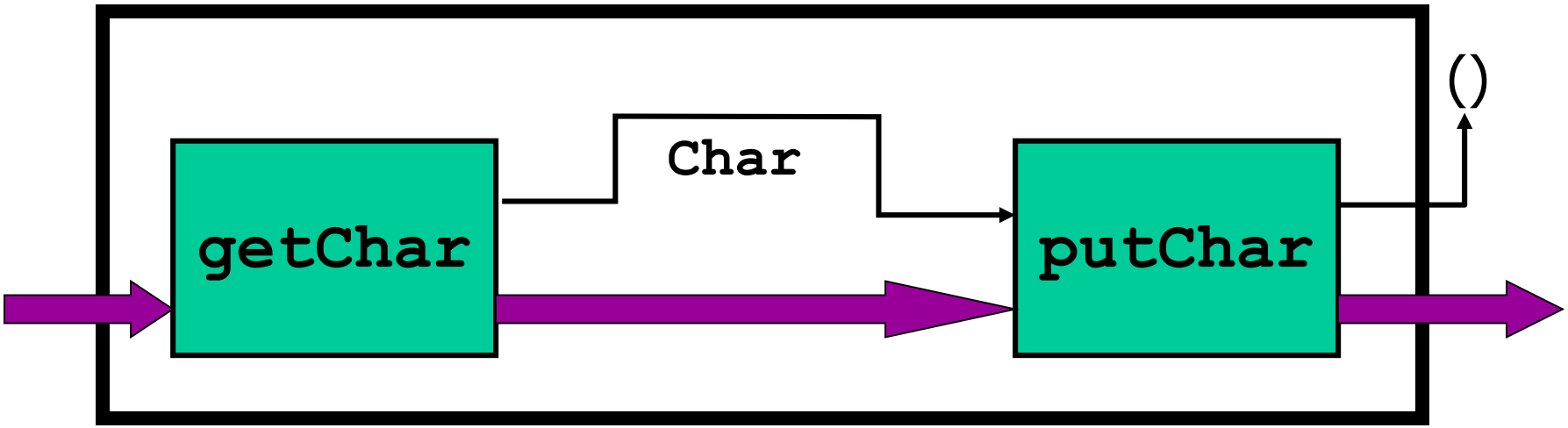
Składanie akcji



Cel: przeczytaj znak a następnie wypisz go

Kombinator (>>=)

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



Dwie akcje zostały połączone w celu utworzenia nowej, większej akcji.

```
getChar :: IO Char
putChar :: Char -> IO ()
```

```
echo :: IO ()
echo = getChar >>= putChar
```

Wyprowadzanie znaku dwukrotnie

```
echoDup :: IO ()
echoDup = getChar   >>= (\c ->
          putChar c >>= (\() ->
          putChar c))
```

- Nawiasy są zbędne

Kombinator (>>)

Można to zrobić prościej:

```
echoDup :: IO ()
echoDup = getChar   >>= \c ->
          putChar c >>
          putChar c
```

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

m >> n  ≡  m >>= (\_ -> n)
```

Pobieranie dwóch znaków

```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar     >>= \c1 ->
              getChar     >>= \c2 ->
              ????
```

Chcemy zwrócić (c1,c2)

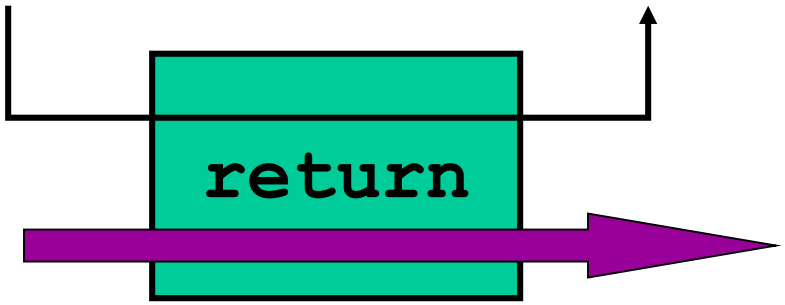
Akcja return v zwraca wartość v, bez przeprowadzania żadnych interakcji:

```
return :: a → IO a
```

Kombinator ***return***

```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar     >>= \c1 ->
              getChar     >>= \c2 ->
              return (c1,c2)
```

```
return :: a -> IO a
```



Abstrakcja lingwistyczna

```
getTwoChars :: IO (Char,Char)
getTwoChars =       getChar>>= \c1 ->
                    getChar>>= \c2 ->
                    return (c1,c2)
```

Notacja „do” to tylko abstrakcja lingwistyczna (wcięcia tylko zwiększają czytelność).

```
getTwoChars :: IO (Char,Char)
getTwoChars =       do {   c1 <- getChar ;
                           c2 <- getChar ;
                           return (c1,c2) }
```

Wykorzystanie wcięć to kolejne ułatwienie syntaktyczne (wcięcia mają tu znaczenie semantyczne).

```
getTwoChars :: IO (Char,Char)
getTwoChars =       do c1 <- getChar
                       c2 <- getChar
                       return (c1,c2)
```

Funkcyjne wejście/wyjście (OCaml)

```
# let witaj()=
  let _ = print_string "Jak masz na imie? "
  in let imie = read_line()
     in print_endline ("Witaj, "^imie^"!");;

val witaj : unit -> unit = <fun>
```

Sekwencja to tylko lukier syntaktyczny.

```
# let witaj'()=
  print_string "Jak masz na imie? ";
  let imie=read_line()
  in print_endline ("Witaj, "^imie^"!");;

val witaj' : unit -> unit = <fun>
```

Monadyczne wejście/wyjście (Haskell)

```
witaj :: IO()
witaj = putStr "Jak masz na imie? " >>
        getLine                     >>=  \imie ->
        putStrLn  ("Witaj, " ++ imie ++ "!")
```

Notacja „do” to tylko abstrakcja lingwistyczna .

```
witaj' :: IO()
witaj' = do { putStr "Jak masz na imie? ";
              imie <- getLine;
              putStrLn  ("Witaj, " ++ imie ++ "!")
            }
```

Wykorzystanie wcięć to kolejne ułatwienie syntaktyczne.

```
witaj'' :: IO()
witaj'' = do putStr "Jak masz na imie? "
             imie <- getLine
             putStrLn  ("Witaj, " ++ imie ++ "!")
```

Standardowy moduł Prelude

Odpowiednik modułu Pervasives w języku OCaml.

Zawiera m.in. standardowe uchwyty (ang. handles)

```
stdin :: Handle
stdout :: Handle
stderr :: Handle
```

Funkcje wejścia/wyjścia z modułu Prelude

Funkcje wyjścia wypisują tekst na standardowym wyjściu (zwykle na terminalu użytkownika).

```
putChar  :: Char -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()  -- dodaje znak końca wiersza
print    :: Show a => a -> IO ()
```

Funkcja print wypisuje wartość dowolnego typu drukowalnego na standardowym wyjściu. Typy drukowalne są instancjami klasy Show; print konwertuje zadaną wartość do napisu i dodaje znak końca wiersza . Np. print ([(n, n^2) | n <- [0..19]]) wypisuje 20 początkowych liczb naturalnych oraz ich kwadraty.

Funkcje wejścia czytają dane ze standardowego wejścia (zwykle z terminala użytkownika).

```
getChar     :: IO Char
getLine     :: IO String
getContents :: IO String
interact    :: (String -> String) -> IO ()
readIO      :: Read a => String -> IO a
readLn      :: Read a => IO a
```

Wyjątki wejścia/wyjścia

Wyjątki w monadzie IO są reprezentowane przez wartości typu IOError. Dla tego typu zdefiniowano kolekcję predykatów, rozpoznających poszczególne rodzaje wyjątków we/wy. Na przykład predykat:

```
IO.isEOFError :: IOError -> Bool
```

sprawdza, czy został zgłoszony wyjątek, sygnalizujący koniec pliku.

W standardowym preludium umieszczono funkcję, która tworzy wyjątek typu IOError:

```
userError :: String -> IOError
```

Wyjątki we/wy są zgłaszane i przechwytywane za pomocą następujących funkcji:

```
ioError :: IOError -> IO a
catch   :: IO a    -> (IOError -> IO a) -> IO a
```

Do zgłoszenia wyjątku we/wy można też użyć funkcji fail, zdefiniowanej następująco:

```
fail s = ioError (userError s)
```

Monoid

Przypomnijmy definicję monoidu:

$\langle S, \bullet, 1 \rangle$, gdzie $\bullet : S \times S \to S$, $1 : S$

$$a \bullet 1 = a$$
$$1 \bullet a = a$$
$$a \bullet (b \bullet c) = (a \bullet b) \bullet c \quad \text{(łączność)}$$

Dwuargumentowa operacja $\bullet$ jest łączna, a 1 jest obustronną jednością (elementem neutralnym).

Monady

Monada składa się z

- Konstruktora typu M

i dwóch operacji:

- bind :: M a -> (a -> M b) -> M b (>>=)

  Łączy wartość z monady (lewy argument) z funkcją (prawy argument).

- unit :: a -> M a (return)

  Wkłada wartość typu *a* do monady i zwraca tę monadę.

Często używana jest operacja

- sequence :: M a -> M b -> M b (>>)

którą można zdefiniować:

```
m >> n  ≡  m >>= (\_ -> n)
```

Aksjomaty dla monad

```
m >>= return   =   m
```

`return` jest prawostronną jednością.

```
return x >>= f   =   f x
```

Operacja `>>=` „wyłuskuje” z monady (lewego argumentu) wartość i aplikuje funkcję (prawy argument) do tej wartości, zwracając jako wynik nową monadę.

```
m1 >>= (λx->m2 >>= (λ y->m3))
=
(m1 >>= λ x->m2) >>= (λ y->m3)
jeśli x nie jest zmienną wolną w m3
```

Łączność operacji `>>=` z uwzględnieniem faktu, że prawy argument jest funkcją.

Aksjomaty dla monad

Dla kombinatora >> trzeci aksjomat upraszcza się (łączność >>):

```
m1 >> (m2 >> m3)
=
(m1 >> m2) >> m3
```

Oba argumenty oraz wynik operacji >> są monadami.

Akcje wtórne

Czytanie napisu z klawiatury (jest w preludium):

```
getLine :: IO String
getLine  = do x ← getChar
              if x == '\n' then
                 return []
               else
                 do xs ← getLine
                    return (x:xs)
```

Wyświetlenie napisu na ekranie (Prelude):

```
putStr        :: String → IO ()
putStr []      = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

Wyświetlenie napisu na ekranie i przejście do nowego wiersza (Prelude) :

```
putStrLn   :: String → IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

Przykład : zgadywanie słów

Oto wersja anglosaskiej gry hangman w zgadywanie słów:

Jeden z graczy zapisuje sekretne słowo.

Drugi z graczy stara się odgadnąć zapisane słowo.

Dla każdej próby komputer wskazuje litery z zapisanego słowa występujące w propozycji drugiego gracza (wraz z ich pozycjami w zapisanym słowie).

Gra kończy się po odgadnięciu zapisanego słowa.

Przy tworzeniu programu zastosujemy metodę programowania zstępującego („top down”).

```
hangman :: IO ()
hangman  =
   do putStrLn "Think of a word: "
      word ← sgetLine
      putStrLn "Try to guess it:"
      guess word
```

Akcja sgetLine czyta wiersz z klawiatury, wyświetlając każdy znak w postaci myślnika:

```
sgetLine :: IO String
sgetLine  = do x ← getChar
               if x == '\n' then
                  do putChar x
                     return []
                else
                  do putChar '-'
                     xs ← sgetLine
                     return (x:xs)
```

Funkcja guess zawiera główną pętlę, w której gracz wprowadza słowo i otrzymuje odpowiedź.

```
guess     :: String → IO ()
guess word =
   do putStr "> "
      xs ← getLine
      if xs == word then
         putStrLn "You got it!"
       else
         do putStrLn (diff word xs)
            guess word
```

Funkcja diff pokazuje, które znaki z pierwszego napisu występują w drugim napisie:

```
diff      :: String → String → String
diff xs ys =
   [if elem x ys then x else '-' | x ← xs]
```

Na przykład:

```
> diff "haskell" "pascal"

"-as--ll"
```

Moduły w języku Haskell

Kompletny program w Haskellu składa się z modułów (które są jednostkami kompilacji), z których jeden (o nazwie Main) musi eksportować wartość main. Plik źródłowy w Haskellu zawiera definicję jednego modułu. Nazwy pliku (bez rozszerzenia) i modułu muszą być identyczne. W module definiowane są wartości, typy danych, klasy itp. Moduły są wykorzystywane do kontroli przestrzeni nazw. W odróżnieniu od modułów w OCamlu moduły w Haskellu **nie** są wartościami pierwszej kategorii. Wartością programu jest wartość, związana z identyfikatorem main.

W deklaracji modułu po słowie kluczowym module podaje się jego nazwę (pierwsza litera musi być wielka), a następnie w nawiasach listę eksportowanych wartości. Jeśli lista (wraz z nawiasami) zostanie pominięta, to moduł eksportuje wszystkie identyfikatory. Po słowie kluczowym where umieszczana jest treść modułu, np.

```
module Stack (T, create, push, top, pop, isEmpty) where
```

Moduł może odwoływać się do innych modułów, zadeklarowanych w grupie na początku modułu, zaraz po jego deklaracji, np.

```
module Main(main) where  -- eksportuje tylko main
--module Main where        -- exportuje wszystko
  import qualified Stack as S
  import Data.Array
  import Control.Exception
  import System.IO
```

Stos jako moduł

```
-- plik: Stack.hs
module Stack (T, create, push, top, pop, isEmpty) where
  data T a = EmptyStack | Push a (T a)
--          deriving (Read, Show)
  create :: () -> T a
  create() = EmptyStack
  push e s = Push e s

  top (Push e _) = e
  top EmptyStack = error "module Stack: top"

  pop (Push _ s) = s
  pop EmptyStack = EmptyStack

  isEmpty EmptyStack = True
  isEmpty _ = False
```

Moduł Main do testowania stosu (1)

```
-- plik: Main.hs
module Main(main) where  -- eksportuje tylko  main
--module Main where          -- eksportuje wszystko
import qualified Stack as S
import Data.Array
import Control.Exception
import System.IO

-- Ten moduł celowo jest wzorowany na module stackTest.ml
-- z poprzedniego wykładu,
-- ale wykorzystuje stos niemodyfikowalny

main = repeatTest (True, S.create())

menuItems = array (0,5) (zip [0..5] ["Stack Operations","push", "top", "pop", "isEmpty", "quit testing"])

doA :: (Monad m, Ix k) => (Array k t -> k -> m a) -> m b -> Array k t -> [k] -> m b
doA  rep finally a (i:is) =
   do rep a i
      doA rep finally a is
doA   _  finally _   _    =  finally
```

Moduł Main do testowania stosu (2)

```
menu :: Array Int String -> IO Int
menu opt =
  do putStr "\n\n================================================ \n„
     putStrLn (opt ! 0)
     doA  (\a i -> putStrLn ((show i)++". "++(opt ! i)))
            (putStr "\nSelect an option: " >> System.IO.hFlush stdout)
            opt (tail $ indices opt)
     choice <- getLine
     return (read choice)


{- można też tak:
menu opt =
  do putStr "\n\n================================================ \n„
       putStrLn (opt ! 0)
       putStrLn ((show 1)++". "++(opt ! 1))
       putStrLn ((show 2)++". "++(opt ! 2))
       putStrLn ((show 3)++". "++(opt ! 3))
       putStrLn ((show 4)++". "++(opt ! 4))
       putStrLn ((show 5)++". "++(opt ! 5))
       putStr "\nSelect an option: „
       System.IO.hFlush stdout
       choice <- getLine
       return (read choice)
-}
```

Moduł Main do testowania stosu (3)

```
repeatTest :: (Bool, S.T Int) -> IO()
repeatTest (False, _) = return ()
repeatTest (True, s) =
  do choice <- menu menuItems
     case choice of
      1 -> do putStr "Stack item = "
              System.IO.hFlush stdout
              item <- getLine
              repeatTest (True, S.push (read item) s)
      2 -> do Control.Exception.catch (print $ S.top s)
                                      (\msg -> putStrLn $ "Exception: " ++ show (msg::Control.Exception.SomeException))
              repeatTest (True, s)
      3 -> do putStrLn "popped"
              repeatTest (True, S.pop s)
      4 -> do putStrLn $ "Stack is "++(if S.isEmpty s then "" else "not ")++"empty."
              repeatTest (True, s)
      5 -> repeatTest (False, s)
```

Kompilacja modułów

Dystrybucja GHC Haskella oprócz interpretera ghci zawiera również kompilator ghc.
Kompilacja naszego przykładowego programu (w wierszu poleceń) przebiega następująco:

```
ghc  -c  Stack.hs
```

W wyniku powstają dwa pliki: Stack.hi (z interfejsem) oraz Stack.o (plik z kodem obiektowym).

Kompilacja i wygenerowanie kodu wykonywalnego wygląda następująco:

```
ghc –o  stackTest  Main.hs
```

W wyniku powstają pliki: Main.hi (z interfejsem) oraz Main.o (plik z kodem obiektowym), a także plik wykonywalny stackTest (w systemie Windows stackTest.exe).

Zadanie kontrolne

Zaimplementuj grę nim w Haskellu. Reguły gry są następujace:

Przykładowa plansza składa się z pięciu wierszy, zawierających gwiazdki:

```
1: * * * * *
2: * * * *
3: * * *
4: * *
5: *
```

Dwaj gracze (jeden z nich to komputer) naprzemiennie usuwają dowolną liczbę gwiazdek (co najmniej jedną) z jednego, wybranego wiersza.

Wygrywa ten z graczy, który usunął z planszy ostatnią gwiazdkę (lub gwiazdki).

Wskazówka:

Reprezentuj planszę jako listę pięciu liczb całkowitych, podających liczbę gwiazdek w każdym wierszu. Tutaj przykładowa plansza początkowa jest reprezentowana przez listę [5,4,3,2,1].