

Wykład 2

Podstawy programowania funkcyjnego c.d.

Środowisko i domknięcie

Organizacja pamięci programu

Rekursja ogonowa

Wzorce (ang. patterns)

Dopasowanie do wzorca w deklaracjach wartości

Wzorzec z wieloznacznikiem (z džokerem, uniwersalny)

Wyrażenie „match”

Kombinacja wzorców

Haskell – ewaluacja leniwa

Współdzielenie danych niemodyfikowalnych

Wzorce warstwowe (ang. as-patterns)

Środowisko i domknięcie

Wyrażenia w językach programowania są wartościowane (ewaluowane) w *środowisku* (ang. environment) składającym się z listy par <identyfikator, wartość>. Z abstrakcyjnego punktu widzenia środowisko jest *słownikiem*.

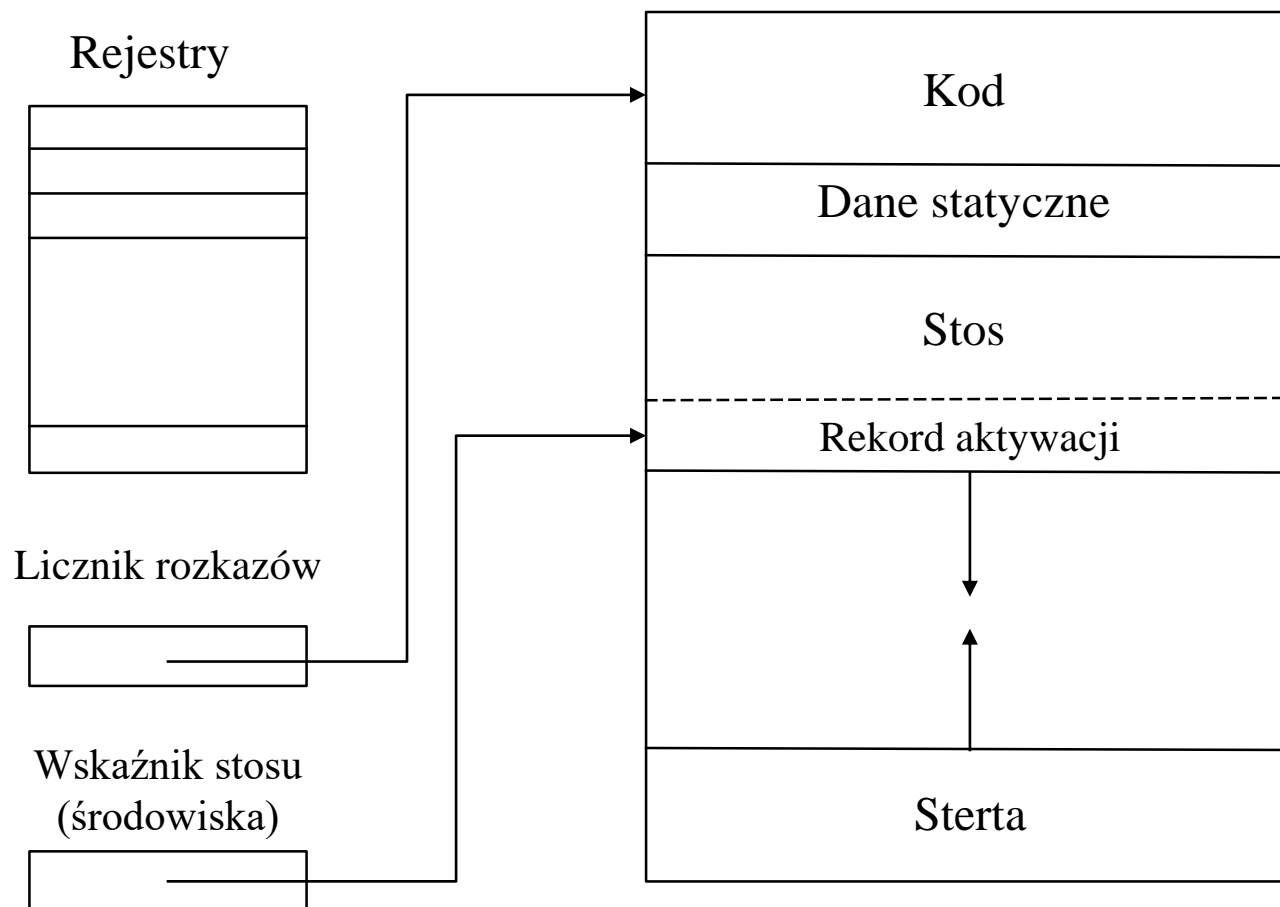
Wartością wyrażenia funkcyjnego jest *domknięcie* (ang. closure), które jest parą <środowisko, wyrażenie funkcyjne>.

Nazwa pochodzi od „domykania” literału funkcyjnego względem wszystkich zmiennych wolnych tego literału

Na poprzednim wykładzie prosta semantyka języka była oparta na zastępowaniu i przepisywaniu wyrażeń. Implementacja, wykorzystująca bezpośrednio ten model byłaby bardzo nieefektywna. W praktyce wartościowanie wykorzystuje środowiska i domknięcia.

Organizacja pamięci programu

W czasie wykonania w obszarze pamięci programu przechowywany jest jego *kod* (ang. code) i *dane* (ang. data). Obszar danych jest podzielony na *dane statyczne* (ang. static data), *stos* (ang. stack) i *stertę* (ang. heap) z różną strategią rezerwacji pamięci.



Organizacja pamięci programu

- Obszar danych statycznych zawiera wartości zmiennych statycznych, dla których pamięć jest przydzielana w trakcie kompilacji (np. w C i C++ za pomocą słowa kluczowego `static` wewnątrz funkcji). Uwaga: modyfikator `static` w językach Java, C#, C++ w kontekście programowania obiektowego ma inną semantykę!
- Stos (pamięć automatyczna) składa się z *rekordów aktywacji* (ang. activation records), tworzonych automatycznie dla każdego bloku (funkcji) przy wejściu i zdejmowanych ze stosu przy wyjściu z bloku. Rekord aktywacji, nazywany też *ramką stosu* (ang. stack frame) zawiera m.in. zmienne lokalne, łańcze dostępu, argumenty wywołania funkcji, wartość zwracaną (lub jej adres), adres powrotu (adres kolejnej instrukcji w kodzie programu po zakończeniu wykonywania bieżącej funkcji).
- Stermta (pamięć wolna, pamięć dynamiczna) zawiera dane, które mają istnieć dłużej niż wykonanie bieżącego bloku. Pamięć dla nich jest przydzielana dynamicznie np. za pomocą operatora `new`.

Wartością wyrażenia funkcyjnego jest *domknięcie* (ang. closure), które jest parą $\langle \text{środowisko}, \text{wyrażenie funkcyjne} \rangle$. Domknięcie jest implementowane jako para składająca się ze wskaźnika do rekordu aktywacji i wskaźnika do kodu funkcji.

Wskaźnik do rekordu aktywacji wystarczy do reprezentowania środowiska, ponieważ rekord aktywacji dla funkcji zawiera *łańcze dostępu*, nazywane też *łańczem statycznym* (ang. access link, static link), wskazujące na rekord aktywacji najbliższego bloku, w którym funkcja jest zdefiniowana.

Statyczne wiązanie zmiennych globalnych funkcji

Zmienne wolne funkcji są wiązane statycznie z wartościami ze środowiska w momencie deklaracji funkcji. W innych językach programowania wiązanie może być dynamiczne, tzn. dokonywane w czasie wartościowania funkcji.

```
# let p = 10;;  
val p : int = 10  
# let k x = (x, p, x+p);;  
val k : int -> int * int * int = <fun>  
# k p;;  
- : int * int * int = (10, 10, 20)  
# let p = 1000;;  
val p : int = 1000  
# k p;;  
-: int * int * int = (1000, 10, 1010)
```

Wiązanie zmiennych globalnych funkcji

OCaml – wiązanie statyczne

```
# let a = 3  
  in let f = fun x -> x + a  
    and a = 5  
    in a * f 2;;  
- : int = 25  
  
(* 5 * (2 + 3) => 25 *)
```

Lisp – wiązanie dynamiczne

```
➤ let a = 3  
  in let f = proc (x) + (x,a)  
    a = 5  
    in *(a, f(2))  
➔ 35  
  
;; *(5, +(2, 5)) => 35  
;; czyli 5 * (2 + 5) => 35
```

W OCamlu zmienne globalne funkcji są wiązane statycznie z wartościami ze środowiska w momencie **definiowania** funkcji.

W Lispie zmienne globalne funkcji są wiązane dynamicznie z wartościami ze środowiska w czasie **wykonania** funkcji.

Funkcje rekurencyjne

let rec *ident arg1 ... argn = wyr;;*

```
# let rec sigma n = if n = 0 then 0 else n + sigma (n-1);;  
val sigma : int -> int = <fun>  
# sigma 5;;  
- : int = 15
```

Funkcje wzajemnie rekurencyjne

```
# let rec even n = if n=0 then true else odd(n-1)  
    and odd n = if n=0 then false else even(n-1);;  
val even : int -> bool = <fun>  
val odd : int -> bool = <fun>  
# even 128;;  
- : bool = true  
# odd 128;;  
-: bool = false
```

Uwaga! Powyższy przykład był oczywiście czysto dydaktyczny. Efektywne definicje funkcji even i odd powinny wyglądać tak:

```
# let even n = n mod 2 = 0    let odd n = n mod 2 <> 0;;  
val even : int -> bool = <fun>  
val odd : int -> bool = <fun>
```

Rekursja ogonowa - przykład

Rekursja ogonowa lub terminalna (ang. tail recursion, terminal recursion) ma miejsce wtedy, kiedy w każdej klauzuli definiującej funkcję wartość funkcji jest obliczana bez wywołania rekurencyjnego lub jest *bezpośrednim* wynikiem tego wywołania.

```
# let rec suc n = if n=0 then 1 else 1 + suc(n-1);;
val suc : int -> int = <fun>
# suc 1000000;;
Stack overflow during evaluation (looping recursion?).

# let succ_tail n =
  let rec succ_aux(n,accum) =
    if n=0 then accum else succ_aux(n-1,accum+1)
  in succ_aux(n,1)
;;
val succ_tail : int -> int = <fun>
# succ_tail 1000000;;
- : int = 1000001
```


Gorliwe wartościowanie rekursji ogonowej

```
let rec factIt (n, ak) = if n=0 then ak else factIt(n-1, n*ak);;
```

factIt(4,1) \Rightarrow factIt(4-1,4*1)
 \Rightarrow factIt(3,4)
 \Rightarrow factIt(3-1,3*4)
 \Rightarrow factIt(2,12)
 \Rightarrow factIt(2-1,2*12)
 \Rightarrow factIt(1,24)
 \Rightarrow factIt(1-1,1*24)
 \Rightarrow factIt(0,24)
 \Rightarrow 24

Dobre kompilatory wykrywają rekursję ogonową i wykonują ją efektywnie (iteracyjnie, bez tworzenia nowych rekordów aktywacji na stosie). Rezultat wywołania rekurencyjnego `factIt(n-1, n*ak)` nie podlega dalszym obliczeniom, lecz jest zwracany jako wartość wywołania `factIt(n, ak)`.

Dodatkowy parametr `ak` nazywany jest akumulatorem, ponieważ służy do „akumulowania” wyniku.

Gorliwe wartościowanie rekursji

```
let rec silnia n = if n=0 then 1 else n*silnia (n-1);;
```

```
silnia 4  ⇒ if 4=0 then 1 else 4*silnia (4-1)
          ⇒ 4*silnia 3
          ⇒ 4*(if 3=0 then 1 else 3*silnia (3-1))
          ⇒ 4*(3* silnia 2)
          ⇒ 4*(3*(if 2=0 then 1 else 2*silnia (2-1)))
          ⇒ 4*(3*(2*silnia 1))
          ⇒ 4*(3*(2*(if 1=0 then 1 else 1*silnia (1-1))))
          ⇒ 4*(3*(2*(1*silnia 0)))
          ⇒ 4*(3*(2*(1*(if 0=0 then 1 else 0*silnia (0-1)))))
          ⇒ 4*(3*(2*(1*1)))
          ⇒ 4*(3*(2*1))
          ⇒ 4*(3*2)
          ⇒ 4*6
          ⇒ 24
```

Rekursja ogonowa w języku Haskell (1)

A jak to wygląda w Haskellu?

Zwykła rekursja:

```
suc :: Int -> Int
suc n = if n == 0 then 1 else 1 + suc (n-1)

suc 1000000000
*** Exception: stack overflow
```

Rekursja ogonowa:

```
sucTail' :: Int -> Int
sucTail' n = sucAux 1 n
    where
        sucAux accum 0 = accum
        sucAux accum n = sucAux (accum+1) (n-1)

sucTail' 1000000000
*** Exception: stack overflow
```

Rekursja ogonowa w języku Haskell (2)

W Haskellu ewaluacja jest leniwa, więc proces obliczania `sucAux` przebiega tak:

```
sucAux 1 3
= sucAux (1+1) 2
= sucAux ((1+1)+1) 1
= sucAux (((1+1)+1)+1) 0
= ((1+1)+1)+1 = (2+1)+1 = 3+1 = 4
```

Całe wyrażenie dla sumy w akumulatorze jest konstruowane, zanim zostaną wykonane dodawania. Pamięć zaoszczędzona na stosie została wykorzystana na zapamiętanie długiego wyrażenia. **W tym przypadku wartość akumulatora powinna być obliczana gorliwie.**

Haskell w standardowym preludium ma zdefiniowany **operator gorliwej aplikacji** (ang. `strict application`) z łącznością prawostronną:

$$(\$!) :: (a \rightarrow b) \rightarrow a \rightarrow b$$
$$f \$! x = \dots$$

Operator `$` zachowuje leniwe wartościowanie, zmienia tylko łączność na prawostronną. O ewaluacji leniwej będzie mowa na wykładzie 5.

Rekursja ogonowa w języku Haskell (3)

Funkcja `sucTail`, wykorzystująca operator `$!` wygląda tak:

```
sucTail    :: Int -> Int
sucTail n = sucAux 1 n
           where
               sucAux accum 0 = accum
               sucAux accum n = (sucAux $! (accum+1)) (n-1)

sucTail 100000000
100000001
```

Obliczenie tego wyniku trwa dużo dłużej niż w OCamlu. Szacowanie złożoności obliczeniowej programów z ewaluacją leniwą jest trudniejsze, niż w przypadku ewaluacji gorliwej.

Żadna ze strategii ewaluacji nie jest uniwersalnym panaceum. W OCamlu bywa przydatna ewaluacja leniwa, a w Haskellu – gorliwa. Trzeba rozumieć obie.

Zawężanie typów

Można sprecyzować (zawęzić) typ wyrażenia: *(wyr : typ)*
w deklaracji funkcji.

```
# let makePair x y = (x,y);;
val makePair : 'a -> 'b -> 'a * 'b = <fun>
# let makePair1 (x:int) y = (x,y);;
val makePair1 : int -> 'a -> int * 'a = <fun>
# let makePair2 x y = (x, (y:float));;
val makePair2 : 'a -> float -> 'a * float = <fun>
# [];;
- : 'a list = []
# ([] : char list);;
-: char list = []
```

W Haskellu dobry styl programowania wymaga podania typu funkcji przed jej definicją. Można w nim zawęzić jej najogólniejszy typ.

Wzorce - definicja

Wzorce pozwalają na dekompozycję wartości strukturalnych i związanie identyfikatorów z wartościami ich komponentów.

Wzorzec (ang. pattern) jest zbudowany ze zmiennych i stałych za pomocą konstruktorów wartości. **Zmienne we wzorcach nie są związane z żadnymi wartościami - związanie nastąpi w wyniku dopasowania do wzorca.** Dopasowanie do wzorca (ang. pattern matching) ma sens tylko dla wartości, nie będących funkcjami.

- Każda wartość dopasowuje się do zmiennej, tworząc związanie.
- Stała dopasowuje się tylko do identycznej stałej.
- Do wzorca, posiadającego strukturę, dopasowuje się tylko wartość o takiej samej strukturze.

Dopasowanie do wzorca w definicjach zmiennych

```
# let x = (false, 10);;
val x : bool * int = (false, 10)
```

```
# let (z,y) = x;;
val z : bool = false
val y : int = 10
```

```
# let (false, y) = x;;
Characters 4-14:
  let (false ,y) = x;;
      ^^^^^^^^^^
```

Warning 8: this pattern-matching is not exhaustive.

Here is an example of a value that is not matched:

```
(true, _)
val y : int = 10
```

```
# let (true, y) = x;;
Characters 4-13:
  let (true, y) = x;;
      ^^^^^^^^^^
```

Warning 8: this pattern-matching is not exhaustive.

Here is an example of a value that is not matched:

```
(false, _)
Exception: Match_failure ("//toplevel//", 1, 4).
```


Dopasowanie do wzorca w definicjach zmiennych

```
# let l = ["Ala"; "ma"; "kota"];;  
val l : string list = ["Ala"; "ma"; "kota"]
```

```
# let [x1; x2; x3] = l;;
```

Characters 4-16:

```
let [x1; x2; x3] = l;;
```

```
^^^^^^^^^^^^
```

Warning 8: this pattern-matching is not exhaustive.

Here is an example of a value that is not matched:

```
[]
```

```
val x1 : string = "Ala"
```

```
val x2 : string = "ma"
```

```
val x3 : string = "kota"
```

```
# let h::t = l;;
```

Characters 4-8:

```
let h::t = l;;
```

```
^^^^
```

Warning 8: this pattern-matching is not exhaustive.

Here is an example of a value that is not matched:

```
[]
```

```
val h : string = "Ala"
```

```
val t : string list = ["ma"; "kota"]
```

Dopasowanie do wzorca w definicjach zmiennych - Haskell

```
Prelude> let x = (False,10)
```

```
x :: (Bool, Integer)
```

```
Prelude> let (z,y)=x
```

```
y :: Num t => t
```

```
z :: Bool
```

```
Prelude> y
```

```
10
```

```
it :: Num t => t
```

```
Prelude> z
```

```
False
```

```
it :: Bool
```

```
Prelude> let (False,y)=x
```

```
y :: Num t => t
```

```
Prelude> y
```

```
10
```

```
it :: Num t => t
```

```
Prelude> let (True,y)=x
```

```
y :: Num t => t    -- skompiluje się i wykona (leniwa ewaluacja!)
```

```
Prelude> y        -- dopiero teraz jest potrzebna wartość y, co wymaga ewaluacji prawej strony,  
                  -- dopasowania wzorca i w efekcie spowoduje błąd wykonania
```

```
*** Exception: <interactive>:1:4-13: Irrefutable pattern failed for pattern (True, y)
```

Wzorzec z wieloznacznikiem (z džokerem, uniwersalny)

Wieloznacznik (lub *dżoker*, ang. wildcard) podobnie jak zmienna dopasowuje się z każdą wartością ale nie tworzy związania.

```
# let (z, _ ) = (false, 10);;  
val z : bool = false
```

Wzorce nie muszą być „płaskie”.

```
# let x = (("Smith", 25), true);;  
val x : (string * int) * bool = (("Smith", 25), true)  
# let ((n,w),b) = x;;  
val n : string = "Smith"  
val w : int = 25  
val b : bool = true
```

Wyrażenie „match” (1)

match *wyr* **with**

wl **->** *wyrl*

⋮

| *wn* **->** *wywn*

```
let imply1 pb =  
  match pb with  
    (true, true) -> true  
  | (true, false) -> false  
  | (false, true) -> true  
  | (false, false) -> true;;
```

```
let imply2 pb =  
  match pb with  
    (true, x) -> x  
  | (false, x) -> true;;
```

Wyrażenie „match” (2)

```
let imply3 pb =  
  match pb with  
    (true, x) -> x  
  | (false, _) -> true;;  
let imply4 pb =  
  match pb with  
    (true, false) -> false  
  | _ -> true;;
```

```
let imply5 =  
  function  
    (true, false) -> false  
  | _ -> true;;
```

```
let (=>) b1 b2 =  
  match (b1, b2) with  
    (true, false) -> false  
  | _ -> true;;  
(* # false => true;;  
- : bool = true *)
```

(* postać rozwinięta, operator infiksowy *)

Kombinacja wzorców

wl / ... / wn

wi muszą zawierać te same zmienne tych samych typów!

```
# let isLatinVowel v =  
  match v with  
    'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true  
    | _ -> false;;  
val isLatinVowel : char -> bool = <fun>  
# isLatinVowel 's';;  
- : bool = false  
# isLatinVowel 'i';;  
- : bool = true
```

Wzorce dozorowane (ang. guarded matches)

```
# let srednia p =  
  match p with  
    (x,x) -> x                                (* zmienna x występuje we wzorcu dwukrotnie *)  
  | (x,y) -> (x +. y) /. 2.0;;
```

Characters 40-41:

```
(x,x) -> x                                (* zmienna x występuje we wzorcu dwukrotnie *)  
  ^
```

Error: Variable x is bound several times in this matching

(* Można jednak użyć wzorców dozorowanych *)

```
# let srednia p =  
  match p with  
    (x,y) when x=y -> x                        (* guarded match *)  
  | (x,y)          -> (x+.y)/.2.0;;  
  val srednia : float * float -> float = <fun>
```

```
# srednia (5.,5.);;
```

```
-   : float = 5.
```

```
-   (* W tym przykładzie najlepiej tak: *)
```

```
let srednia(x,y) = if x=y then x else (x+.y)/.2.0;;
```

Przykład: funkcje „zip” i „unzip”

```
# let rec zip pl =          (* zip ([x1, ... ,xn], [y1, ... ,yn]) = [(x1,y1), ... ,(xn,yn)] *)
  match pl with
    (h1::t1,h2::t2) -> (h1,h2)::zip(t1,t2)
  | _ -> [];
val zip : 'a list * 'b list -> ('a * 'b) list = <fun>
# zip ([1;2;3], [4;5;6]);;
- : (int * int) list = [(1, 4); (2, 5); (3, 6)]
# zip ([1;2;3], [4;5]);;
- : (int * int) list = [(1, 4); (2, 5)]

# let rec unzip ps =      (* unzip [(x1,y1), ... ,(xn,yn)] = ([x1, ... ,xn], [y1, ... ,yn]) *)
  match ps with
    [] -> ([], [])
  | (h1,h2)::t -> let (l1,l2) = unzip t in (h1::l1, h2::l2);;
val unzip : ('a * 'b) list -> 'a list * 'b list = <fun>
# unzip [(1,2);(3,4);(5,6);(7,8)];;
- : int list * int list = ([1; 3; 5; 7], [2; 4; 6; 8])
```


Haskell – ewaluacja leniwa

Leniwa ewaluacja może spowodować subtelne efekty w definiowanych funkcjach.

```
ff 0 _ = True
ff _ 0 = True
ff _ _ = False

ff1 _ 0 = True
ff1 0 _ = True
ff1 _ _ = False
```

Funkcje `ff` i `ff1` dla argumentów o określonych wartościach zachowują się identycznie.

Dla argumentów, których ewaluacja nie doprowadza do obliczenia wartości, ich zachowanie może być różne.

```
*Main> ff 0 (error "Och!")
True
*Main> ff1 0 (error "Och!")
*** Exception: Och!
*Main> ff (error "Och!") 0
*** Exception: Och!
*Main> ff1 (error "Och!") 0
True
```

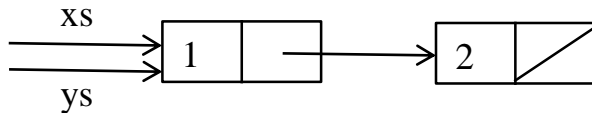
Funkcja `ff` jest "bardziej zdefiniowana" względem swojego drugiego argumentu, natomiast funkcja `ff1` jest "bardziej zdefiniowana" względem swojego pierwszego argumentu.

Współdzielenie danych niemodyfikowalnych

W językach funkcyjnych współdzielone są wszystkie dane, które można bezpiecznie współdzielić przy założeniu, że są one niemodyfikowalne, np.

```
# let xs = [1;2];;  
val xs : int list = [1; 2]  
# let ys = xs;;  
val ys : int list = [1; 2]
```

Jak wygląda reprezentacja wewnętrzna tych list?



Na wykładzie 6 dowiemy się, jak można to sprawdzić.

Współdzielenie i kopiowanie danych niemodyfikowalnych

W języku OCaml konstruktor wartości z argumentem zawsze tworzy nową wartość, np.

```
# let ys = let h::t = xs in h::t;;
```

Characters 13-17:

```
let ys = let h::t = xs in h::t;;
```

^^^^

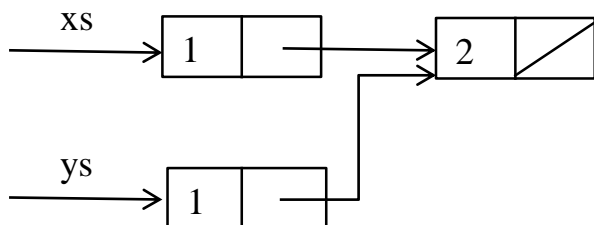
Warning 8: this pattern-matching is not exhaustive.

Here is an example of a value that is not matched:

```
[]
```

```
val ys : int list = [1; 2]
```

Jak teraz wygląda reprezentacja wewnętrzna tych list?



W języku Haskell jest podobnie, z wyjątkiem typów definiowanych przy użyciu słowa kluczowego `newtype` (patrz wykład 4).

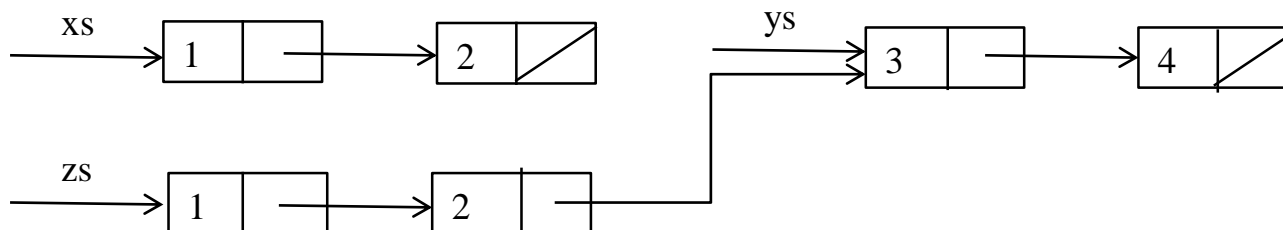
Współdzielenie danych niemodyfikowalnych

W języku OCaml funkcja konkatencji list jest zdefiniowana następująco:

```
let rec ( @ ) l1 l2 = (* identyfikator symboliczny, można wywoływać infiksowo *)
  match l1 with
  | [] -> l2
  | hd :: tl -> hd :: (tl @ l2)

# let xs = [1;2] and ys = [3;4] let zs = xs @ ys;;
val xs : int list = [1; 2]
val ys : int list = [3; 4]
val zs : int list = [1; 2; 3; 4]
```

Jak wygląda reprezentacja wewnętrzna tych list?



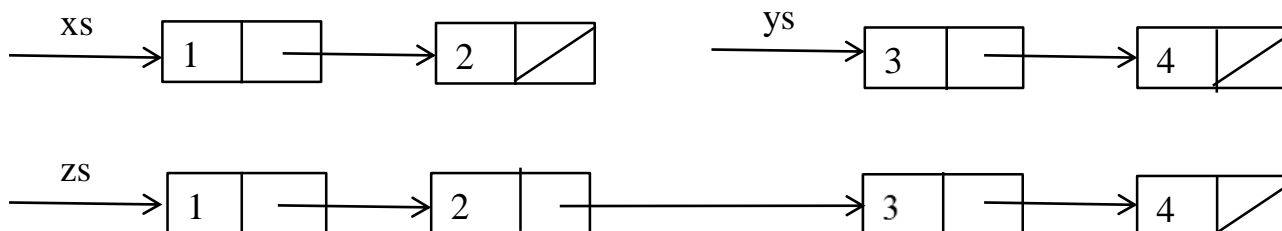
Kopiuwane jest tylko to, co konieczne; współdzielone jest to, co możliwe (przy założeniu, że dane są niemodyfikowalne). Złożoność obliczeniowa (czasowa i pamięciowa) jest liniowa względem długości pierwszej listy.

Współdzielenie i kopiowanie danych niemodyfikowalnych

Funkcję konkatenacji można też zdefiniować za pomocą rekursji po obu listach:

```
let rec badAppend l1 l2 =  
  match (l1, l2) with  
  | ([], []) -> []  
  | ([], h2::t2) -> h2:: badAppend [] t2  
  | (h1 :: t1, []) -> h1 :: badAppend t1 []  
  | (h1 :: t1, h2::t2) -> h1 :: badAppend t1 l2;;  
  
# let xs = [1;2] and ys = [3;4]   let zs = badAppend xs ys;;  
val xs : int list = [1; 2]  
val ys : int list = [3; 4]  
val zs : int list = [1; 2; 3; 4]
```

Jak teraz wygląda reprezentacja wewnętrzna tych list?



Skopiowane zostały obie listy. Złożoność obliczeniowa (czasowa i pamięciowa) jest liniowa względem sumy długości obu list. Jednak jeśli dopuszczalna byłaby modyfikacja wartości elementów list, to takie rozwiązanie może być preferowane.

Wzorce warstwowe (*ang. as-patterns , layered patterns*)

pat **as** *ident*

```
# let ((n,w) as l, b) = x;;  
val n : string = "Smith"  
val w : int = 25  
val l : string * int = ("Smith", 25)  
val b : bool = true  
  
# let f1 = fun ((x,y),z) -> (x,y,(x,y),z);;  
val f1 : ('a * 'b) * 'c -> 'a * 'b * ('a * 'b) * 'c = <fun>  
  
# let f2 = fun ((x,y) as p,z) -> (x,y,p,z);;  
val f2 : ('a * 'b) * 'c -> 'a * 'b * ('a * 'b) * 'c = <fun>
```

Wykorzystanie wzorców warstwowych może polepszyć czytelność oraz efektywność czasową i pamięciową. Funkcja f2 jest nie tylko czytelniejsza, ale też efektywniejsza niż f1. Będzie o tym mowa na wykładzie 6.

Dopasowanie do wzorca - korzyści

```
let rec sqr_list l =  
  match l with  
    [] -> []  
  | h::t -> h*h :: sqr_list t;;
```

- Czytelność, porównaj:

```
let rec sqr_list l =  
  if l = [] then []  
  else let h = List.hd l in h*h :: sqr_list (List.tl l);;
```

- Efektywność, porównaj:

```
let rec sqr_list l =  
  if l = [] then []  
  else (List.hd l)*(List.hd l) :: sqr_list (List.tl l);;
```

- Bezpieczeństwo – kompilator statycznie sprawdza kompletność wzorca.

Zadania kontrolne (1)

1. Liczby Fibonacciego są zdefiniowane następująco:

$$f(0) = 0$$

$$f(1) = 1$$

$$f(n+2) = f(n) + f(n+1)$$

Napisz dwie funkcje, które dla danego n znajdują n -tą liczbę Fibonacciego: pierwszą opartą bezpośrednio na powyższej definicji i drugą, wykorzystującą rekursję ogonową. Porównaj ich szybkość wykonania, obliczając np. 42-gą liczbę Fibonacciego.

2. Dla zadanej liczby rzeczywistej a oraz dokładności ε można znaleźć pierwiastek trzeciego stopnia z a wyliczając kolejne przybliżenia x_i tego pierwiastka (metoda Newtona-Raphsona):

$$x_0 = a/3 \quad \text{dla } a \geq 1$$

$$x_0 = a \quad \text{dla } a < 1$$

$$x_{i+1} = x_i + (a/x_i^2 - x_i)/3$$

Dokładność jest osiągnięta, jeśli $|x_i^3 - a| \leq \varepsilon * |a|$.

Napisz efektywną (wykorzystującą rekursję ogonową) funkcję *root3*, która dla zadanej liczby a znajduje pierwiastek trzeciego stopnia z dokładnością 1e-15.

3. Napisz dowolną funkcję f typu ' $a \rightarrow b$ '.

Zadania kontrolne (2)

4. Zwiąż zmienną x z wartością 0 konstruując wzorce, do których mają się dopasować następujące wyrażenia:

a) $[-2; -1; 0; 1; 2]$

b) $[(1,2); (0,1)]$

Np. dla wyrażenia $(\text{true}, \text{"hello"}, 0)$ wymaganym wzorcem jest $(_, _, x)$.

5. Zdefiniuj funkcję *initsegment* : $'a \text{ list} * 'a \text{ list} \rightarrow \text{bool}$ sprawdzającą w czasie liniowym, czy pierwsza lista stanowi początkowy segment drugiej listy. Każda lista jest swoim początkowym segmentem, lista pusta jest początkowym segmentem każdej listy.

6. Zdefiniuj funkcję *replace_nth* : $'a \text{ list} * \text{int} * 'a \rightarrow 'a \text{ list}$, zastępującą n -ty element listy podaną wartością (pierwszy element ma numer 0), np.

$\text{replace_nth}([1;2;3], 1, 5) = [1;5;3]$

7. Zdefiniuj następujące funkcje:

a) *split* : $'a \text{ list} \rightarrow ('a \text{ list} * 'a \text{ list}) \text{ list}$, która dla danej listy tworzy listę wszystkich par list, dających po skoncatenowaniu listę początkową, np.

$\text{split} [1;2;3] = [([1;2;3], []); ([1;2], [3]); ([1], [2;3]), ([], [1;2;3])]]$

Zadania kontrolne (3)

b) $insert1 : 'a \rightarrow 'a\ list \rightarrow ('a\ list)\ list$, która dla danego elementu a i listy l wstawia a do l na wszystkich możliwych pozycjach i zwraca listę takich list, np.

$insert1\ 0\ [1;2;3] = [[0;1;2;3]; [1;0;2;3]; [1;2;0;3]; [1;2;3;0]]$

c) $insert2 : 'a \rightarrow ('a\ list)\ list \rightarrow ('a\ list)\ list$, która dla danego elementu a i listy list ll wstawia a na wszystkich możliwych pozycjach we wszystkich listach i zwraca listę tak otrzymanych list, np.

$insert2\ 0\ [[1]; [2;3]; []] = [[0;1]; [1;0]; [0;2;3]; [2;0;3]; [2;3;0]; [0]]$

d) $permut : 'a\ list \rightarrow ('a\ list)\ list$, która dla danej listy tworzy listę wszystkich jej permutacji, np.

$permut\ [1;2;3] = [[3;2;1]; [2;3;1]; [2;1;3]; [3;1;2]; [1;3;2]; [1;2;3]]$

Wskazówka. Każda z funkcji jest przydatna w definicji kolejnej funkcji.

Uwaga! Przypominam, że operacja konkatencji list $@$ i funkcja $List.length$ mają złożoność liniową!