

Wykład 6

Efekty obliczeniowe.

Programowanie imperatywne w języku OCaml.

Efekty obliczeniowe

Komórki i referencje

Referencje jawne i niejawne

Współużytkowanie (aliasowanie)

Równość strukturalna i równość tożsamości

Modyfikowalne struktury danych

Polimorfizm i wartości modyfikowalne. Typy słabe

Imperatywne struktury sterowania

Listy cykliczne

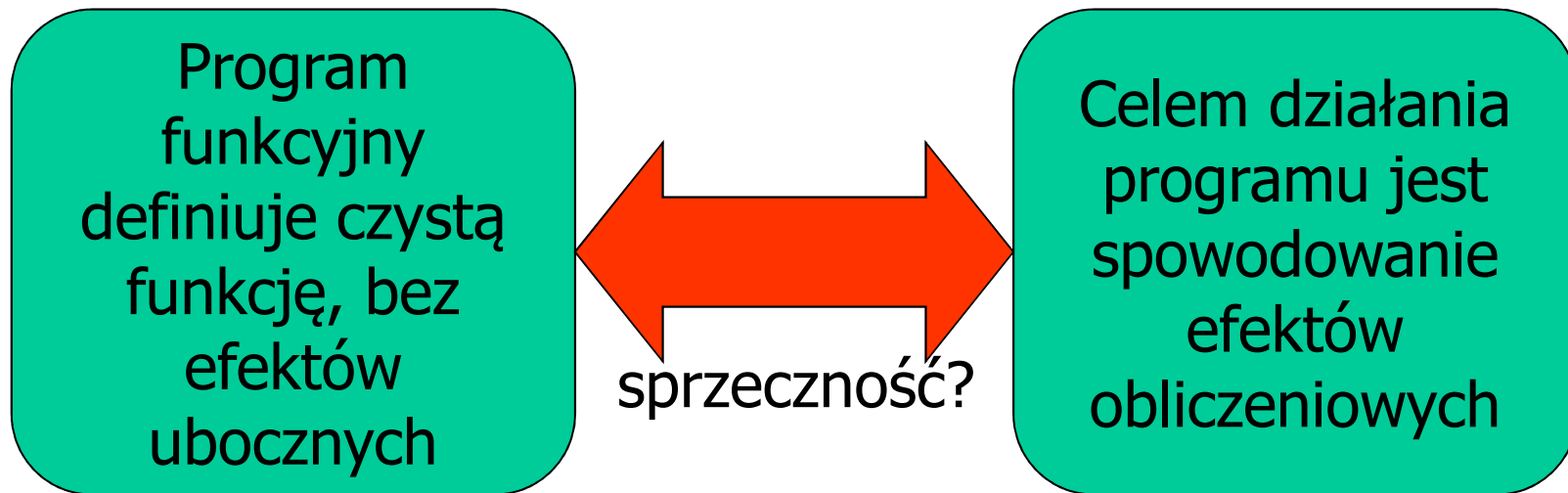
Wejście/wyjście

Przykład: sortowanie szybkie

Współdzielenie czy kopiowanie wartości

Styl funkcyjny i imperatywny: główne różnice

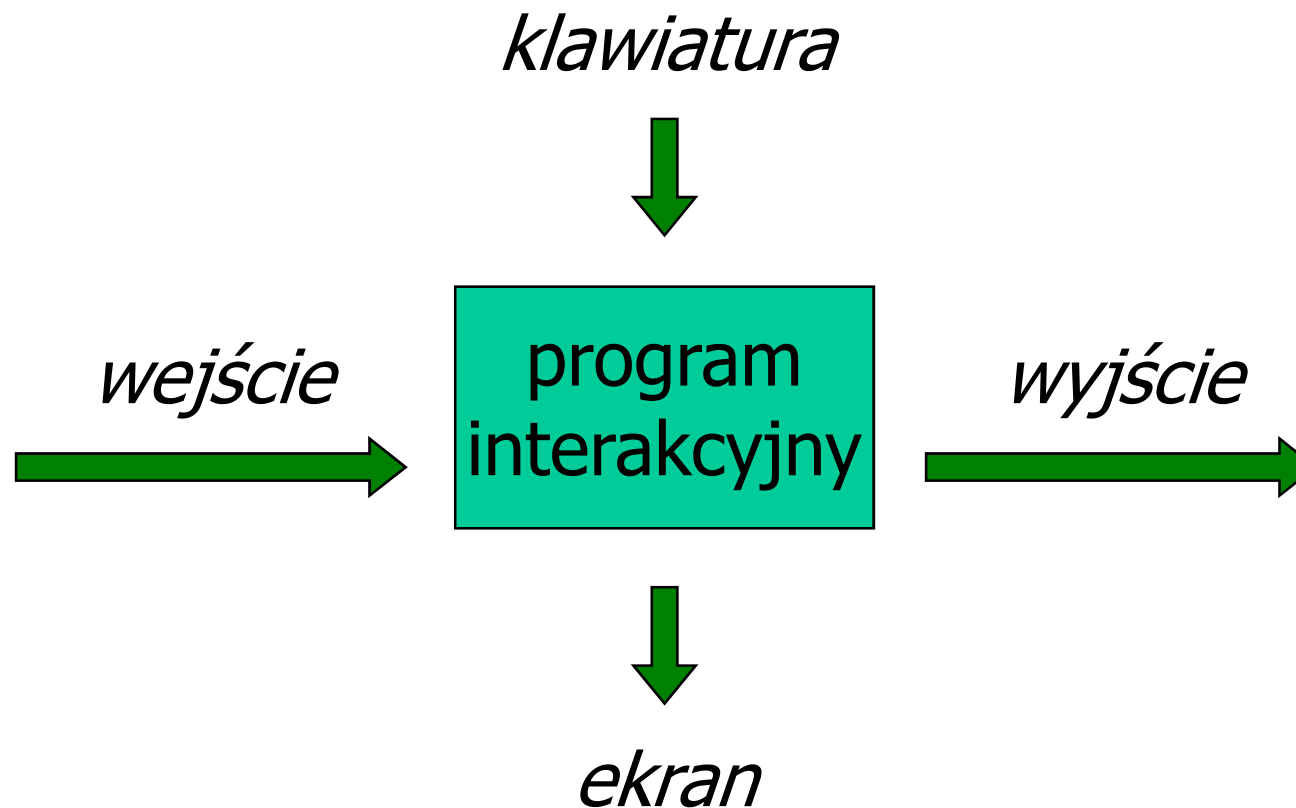
Problem



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



Co zrobić z programami interakcyjnymi?



Efekty obliczeniowe

Każdy kompletny program jako wynik swego działania powinien spowodować określony *efekt* (ang. effect, computational effect).

Efekty mogą być też uboczne (ang. side effect), np. funkcja oblicza wynik, lecz w trakcie działania powoduje dodatkowo pewien efekt (oczywiście zamierzony). Może to być zmiana wartości jakiejś zmiennej modyfikowalnej, tablicy, pola rekordu itp., wykonanie operacji wejścia/wyjścia, wyświetlenie czegoś na ekranie monitora, wykonanie połączenia sieciowego itp.

- Paradygmat funkcyjny gwarantuje zachowanie *przezroczystości referencyjnej* (ang. referential transparency), tzn. każde wyrażenie może być w danym kontekście zastąpione przez swoją wartość.
- Efekt może mieć wpływ na wyniki przyszłych obliczeń.

W językach czysto funkcyjnych (Haskell) efekty są ukryte w *monadach* (ang. monads).

W pozostałych językach funkcyjnych efekty też powinny pozostać lokalne, ale odpowiedzialność za to spoczywa na programiście.

Programowanie, korzystające w znacznym stopniu z *przypisań* (ang. assignment), czyli modyfikowania wartości zmiennych, nosi nazwę *programowania imperatywnego* (ang. imperative programming, z łac. imperare = rozkazywać). Program imperatywny składa się z ciągu poleceń (instrukcji, rozkazów), zmieniających stan programu (powodujących efekty).

Komórki i referencje

Efekty w języku programowania muszą być wspierane przez:

- modyfikowalne struktury danych;
- imperatywne struktury sterowania;
- operacje wejścia/wyjścia.

Prawie każdy język programowania (nawet Haskell za pośrednictwem monad) udostępnia pewną formę operacji przypisania, zmieniającej zawartość *modyfikowalnej komórki* (ang. mutable cell), która jest najprostszą modyfikowalną strukturą danych.

Komórka jest kontenerem z tożsamością i zawartością.

- *Tożsamość* jest stałą („nazwą” lub “adresem” komórki). Struktura danych, reprezentująca tożsamość komórki, jest nazywana *referencją* lub *odniesieniem* (ang. reference). Komórka jest dostępna wyłącznie przez referencję.
- *Zawartość* komórki można zmieniać.

Podstawowymi operacjami na referencjach są:

- alokacja (ang. allocation) – przydzielenie pamięci komórce
- przypisanie (ang. assignment, mutation) – zmiana zawartości komórki
- dereferencja (ang. dereferencing) – pobranie zawartości komórki

Referencje jawne

W językach programowania z referencjami jawnymi (ang. explicit references) istnieje specjalny rodzaj wyrażeń dla referencji. Tak jest np. w językach OCaml, SML, Oz. Mechanizm wiązania wartości ze zmienną i mechanizm przypisania są odseparowane. Zachęca to programistę do programowania w stylu funkcyjnym i używania referencji tylko w razie konieczności.

W OCamlu istnieje polimorficzny typ referencyjny `ref`, który można traktować jako typ wskaźnikowy do modyfikowalnych wartości dowolnego typu. Jest on zdefiniowany jako rekord z jednym modyfikowalnym polem: `type 'a ref = {mutable contents: 'a}`.

Konstruktor wartości tego typu jest `ref : 'a -> 'a ref`.

Typ jest wyposażony w funkcję dereferencji `(!) : 'a ref -> 'a` i modyfikacji wartości `(:=) : 'a ref -> 'a -> unit`.

Jedynym celem działania funkcji `(:=)` jest spowodowanie pewnego efektu (zmiana zawartości komórki), więc zwraca wartość jest typu `unit` (z góry znana).

```
# let x = ref 5;;                (* alokacja *)
val x : int ref = {contents = 5}
# !x;;                          (* dereferencja *)
- : int = 5
# x := 7;;                      (* zmiana zawartości komórki *)
- : unit = ()
# !x;;
- : int = 7
```

Współużytkowanie (*aliasowanie*)

Współużytkowanie lub *współdzielenie* (ang. sharing) określane też terminem *aliasowanie* (ang. aliasing) ma miejsce wtedy, gdy dwa identyfikatory, np. *x* i *y*, odwołują się do tej samej komórki. Mówimy wtedy, że *x* jest aliasem *y*, a *y* jest aliasem *x*.

```
# let x = ref 0;;  
val x : int ref = {contents = 0}  
# let y = x;;  
val y : int ref = {contents = 0}  
# y := 10;;  
- : unit = ()  
# !x;;  
- : int = 10
```

Ogólnie, kiedy zmienia się zawartość komórki to analogicznie zmienia się zawartość wszystkich aliasów.

Równość strukturalna

Dwie wartości są równe, jeśli mają tę samą strukturę.

```
# [1;2;3] = [1;2;3];;  
-   : bool = true
```

W takim przypadku mówimy o równości strukturalnej (ang. structure equality). Z takiej równości korzysta się w programowaniu deklaratywnym.

Równość tożsamości

W przypadku komórek rozważa się równość tożsamości (ang. equality of identity). Dwie komórki nie są równe, jeśli mają tę samą zawartość (to się za chwilę może zmienić), ale wtedy, kiedy mają tę samą tożsamość (nazwę, adres), czyli kiedy są w istocie tą samą komórką!

W języku OCaml są dwa rodzaje porównywania: równość strukturalna (operatory = i <=>) oraz równość fizyczna, czyli równość tożsamości (operatory == i !=).

```
# let x = ref 0;;  
val x : int ref = {contents = 0}  
# let y = ref 0;;  
val y : int ref = {contents = 0}  
# x == y;;                                (* równość tożsamości *)  
- : bool = false  
# x = y;;                                (* równość strukturalna *)  
- : bool = true  
# [1;2;3] == [1;2;3];;                    (* równość tożsamości *)  
- : bool = false  
# [1;2;3] = [1;2;3];;                     (* równość strukturalna *)  
- : bool = true
```

Aliasy zawsze mają tę samą tożsamość.

Równość tożsamości, cd.

W OCamlu te dwa operatory równości dają taki sam wynik dla wartości prostych typu: bool, char, int oraz konstruktorów bezargumentowych. Poniższe przykłady pokazują różnice między tymi równością strukturalną i równością tożsamości. W OCamlu liczby zmiennoprzecinkowe oraz napisy są wartościami strukturalnymi. Funkcji nie można porównywać za pomocą równości strukturalnej, ale można to robić za pomocą równości tożsamości.

```
# 1.0 = 1.0;;
- : bool = true
# 1.0 == 1.0;;
- : bool = false
# "OK" = "OK";;
- : bool = true
# "OK" == "OK";;
- : bool = false
# let id = fun x -> x;;
val id : 'a -> 'a = <fun>
# id = id;;
Exception: Invalid_argument "equal: functional value".
# id == id;;
- : bool = true
# id == fun x -> x;;
- : bool = false
```

Referencje niejawne

W językach imperatywnych wszystkie identyfikatory zmiennych odnoszą się do modyfikowalnych komórek, a operacja dereferencji jest niejawna (ang. implicit references).

W języku Scala definicje zmiennych modyfikowalnych są poprzedzone słowem kluczowym `var`, np.

```
var x = 5
```

```
x: Int = 5
```

Podobnie jak definicje zmiennych niemodyfikowalnych (słowo kluczowe `val`), wiążą one identyfikator z wartością, ale ta wartość może być zmieniana w operacji przypisania.

```
x = 7
```

```
x: Int = 7
```

Operacja dereferencji jest niejawna.

Definiując zmienną, należy zdefiniować wyrażenie, z wartością którego zmienna ma być związana. Jeśli ta wartość w chwili definiowania zmiennej nie jest istotna, można użyć wieloznacznika (podając typ zmiennej):

```
var x:Int = _
```

```
x: Int = 0
```

Kompilator zmienne typów numerycznych inicjuje wartością zero, typu `Boolean` wartością `false`, referencje wartością `null`, a zmienne typu `Char` wartością `\u0000` (tak jak w języku Java).

Referencje niejawne (Scala) i jawne (OCaml)

Scala

```
var z = 0
z: Int = 0

z = 5 // wyrażenie typu Unit
z: Int = 5

val v = z = z+2
v: Unit = ()

z
res0: Int = 7
```

OCaml

```
# let z = ref 0;;
val z : int ref = {contents = 0}

# z := 5;;
- : unit = ()

# let v = z := !z+2;;
val v : unit = ()

# z;;
- : int ref = {contents = 7}

# !z;;
- : int = 7
```

W języku OCaml typ zmiennej referencyjnej różni się od typu wartości, do której się odnosi. Operacja dereferencji jest jawna.

W języku Scala operacja przypisania jest wyrażeniem typu Unit, podobnie jak w OCamlu (unit).

W obu językach imperatywne struktury sterowania są też wyrażeniami typu Unit (unit), co daje możliwość wyboru między funkcyjnym i imperatywnym stylem programowania.

Polimorfizm i wartości modyfikowalne (1)

Połączenie polimorfizmu parametrycznego i wartości modyfikowalnych wymaga dużej ostrożności. Bez dodatkowych zabezpieczeń możliwy byłby taki przebieg **hipotetycznej** sesji OCamlu:

```
# let x = ref [];;  
val x : 'a list ref = {contents=[]}  
# x := 1 :: !x;;  
- : unit = ()  
# x := true :: !x;; ← Błąd wykonania!!!
```

Ten **hipotetyczny** błąd jest spowodowany statyczną typizacją w OCamlu: typ wyrażenia `!x` jest polimorficzny `'a list` i *nie zmienia się w czasie wykonania programu*, więc do takiej listy można dodać element dowolnego typu!

W celu uniknięcia takich sytuacji w OCamlu wprowadzono inną kategorię zmiennych przebiegających typy: słabe zmienne typowe (przebiegające typy). Są one poprzedzane podkreśleniem (`_`).

Prawdziwy przebieg powyższej hipotetycznej sesji będzie więc następujący:

```
# let x = ref [];;  
val x : '_a list ref = {contents=[]}
```

Słaba zmienna typowa nie jest polimorficzna, oznacza ona nieznany typ, który będzie skonkretyzowany podczas pierwszego użycia.

Polimorfizm i wartości modyfikowalne (2)

```
# x := 1 :: !x;;  
- : unit = ()  
# x;;  
- : int list ref = {contents=[1]}
```

Od tego momentu zmienna `x` jest typu `int list ref`. Typ, zawierający słabe zmienne jest więc w rzeczywistości monomorficzny, chociaż nie do końca sprecyzowany.

```
# x := true :: !x;;  
Characters 5-9:  
  x := true :: !x;;  
    ^^^^  
Error: This expression has type bool but is here used with type int
```

Uwaga:

Ta zmiana w systemie typów ma wpływ na programy czysto funkcyjne!

W wyniku aplikacji funkcji polimorficznej do wartości polimorficznej otrzymuje się typ słaby, ponieważ funkcja **może** tworzyć wartości modyfikowalne (system inferencji typów analizuje wyłącznie typy).

Polimorfizm i wartości modyfikowalne (3) - OCaml

```
# let f a b = a;;  
val f : 'a -> 'b -> 'a = <fun>  
# let g = f 1;;  
val g : '_a -> int = <fun>  
# g "ala";;  
- : int = 1  
# g;;  
- : string -> int = <fun>      (* Oj, niedobrze - funkcja monomorficzna!!! *)
```

ale

```
# let g1 x = f 1 x;; (* czyli let g1 = function x -> f 1 x;; *)  
val g1 : 'a -> int = <fun>
```

Ogólnie wyróżnia się wyrażenia syntaktyczne (non-expansive expressions), które są zbyt proste, żeby mogły utworzyć referencje. Są one polimorficzne w zwykłym sensie. Należą do nich:

- stałe;
- identyfikatory (ponieważ odnoszą się do definicji, które już zostały przeanalizowane);
- krotki i rekordy zawierające wartości syntaktyczne;
- wartości zbudowane za pomocą konstruktorów zaaplikowanych do argumentów będących wartościami syntaktycznymi;
- wyrażenia funkcyjne (ponieważ treść funkcji nie jest wykonywana dopóki funkcja nie zostanie zaaplikowana).

Pozostałe wyrażenia są ekspansywne. Są one monomorficzne, ewentualnie ze słabymi zmiennymi typowymi.

Elementy języka, wspomagające programowanie imperatywne

- modyfikowalne struktury danych:
 - referencje,
 - wektory,
 - rekordy z modyfikowalnymi polami;
- struktury sterowania:
 - wyrażenie warunkowe,
 - sekwencje,
 - pętle `for` i `while`,
 - wyjątki;
- operacje wejścia/wyjścia.

Rekordy z modyfikowalnymi polami

W deklaracji typu rekordowego modyfikowalne pola trzeba poprzedzić słowem **mutable**.

type *ident* = { ...; **mutable** et:typ; ...}

```
# type punkt = {wx:float; mutable wy:float};;
type punkt = { wx : float; mutable wy : float; }
# let p = {wx=1.0; wy=0.0};;
val p : punkt = {wx=1.; wy=0.}
```

Wartości modyfikowalnych pól można zmieniać, używając składni: *wyr1.et <- wyr2*.

```
# p.wy <- 3.0;;
- : unit = ()
# p;;
- : punkt = {wx=1.; wy=3.}
# p.wx <- 9.7;;
Characters 0-11:
  p.wx <- 9.7;;
  ^^^^^^^^^^^
Error: The record field wx is not mutable
```

Wektory (*tablice jednowymiarowe*)

```
# let v = [| 2.58; 3.14; 8.73 |];;  
val v : float array = [|2.58; 3.14; 8.73|]
```

Funkcja `Array.make : int -> 'a -> 'a array` bierze liczbę elementów wektora oraz wartość początkową i zwraca utworzony i zainicjowany wektor. Funkcja `Array.length` zwraca długość wektora. Indeksy wektora `v` są zawarte między 0 i `Array.length v - 1`.

```
# let v = Array.make 3 3.14;;  
val v : float array = [|3.14; 3.14; 3.14|]
```

Składnia operacji dostępu do elementów wektora i ich modyfikacji jest następująca:

wyr1.(wyr2)

wyr1.(wyr2) <- wyr3

```
# v.(1);;  
- : float = 3.14  
# v.(0) <- 100.0;;  
- : unit = ()  
# v;;  
- : float array = [|100.; 3.14; 3.14|]  
# v.(-3) +. 5.0;;  
Exception: Invalid_argument "index out of bounds".
```

`t.(i)` jest skrótem notacyjnym dla `Array.get t i`.

Wektory (2)

Funkcje, przeznaczone do manipulowania wektorami są zawarte w module `Array`. Są tam m.in. funkcjonały znane nam już z modułu `List`:

```
map : ('a -> 'b) -> 'a array -> 'b array
```

```
fold_left: ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a
```

```
fold_right: ('a -> 'b -> 'b) -> 'a array -> 'b -> 'b
```

Są także funkcjonały

```
to_list: 'a array -> 'a list
```

```
of_list: 'a list -> 'a array
```

przekształcające odpowiednio tablicę w listę i listę w tablicę.

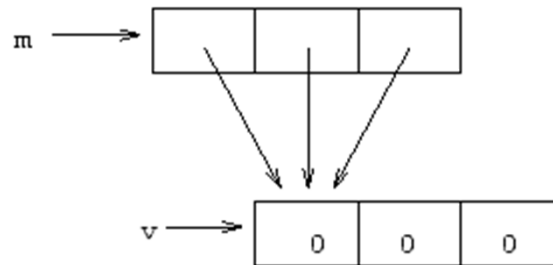
Bezpośrednio w tablicy są przechowywane wartości o długości nie przekraczającej słowa maszynowego: liczby całkowite, znaki, wartości boolowskie i konstruktory bezargumentowe. Pozostałe wartości (strukturalne) są reprezentowane w tablicy przez wskaźnik. W przypadku inicjowania wektorów wartościami strukturalnymi wykorzystywana jest ta sama kopia wartości. Liczby rzeczywiste stanowią przypadek szczególny i są kopiowane przy inicjowaniu wektora.

Java podobnie rozwiązuje problem przechowywania wartości w tablicy. Każdy element tablicy typu prostego (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`) zawiera wartość tego typu, zainicjowaną w czasie przydzielania pamięci dla tablicy. Jeśli tablica ma zawierać obiekty, to każdy element tablicy jest referencją (wskaźnikiem) do obiektu odpowiedniego typu, zainicjowaną początkowo wartością `null`.

Wektory (3)

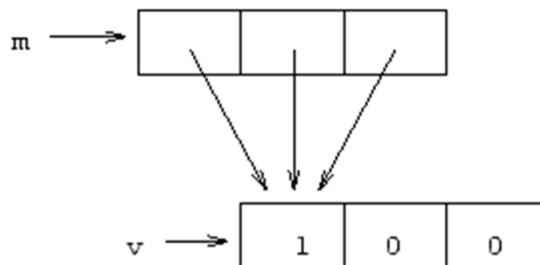
```
# let v = Array.make 3 0;;  
val v : int array = [|0; 0; 0|]  
# let m = Array.make 3 v;;  
val m : int array array = [| [|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|] |]
```

Po wykonaniu powyższych fraz pamięć wygląda następująco.



Modyfikacja wektora *v* powoduje modyfikację wszystkich elementów wektora *m*.

```
# v.(0) <- 1;;  
- : unit = ()  
# m;;  
- : int array array = [| [|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|] |]
```

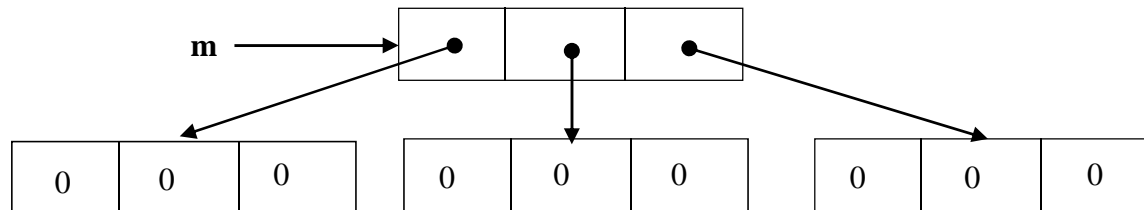


Wektory (4)

Gdybyśmy chcieli uniknąć dzielenia tej samej wartości strukturalnej przez wszystkie elementy wektora (lub inicjować wektor w bardziej skomplikowany sposób), to możemy użyć funkcji `Array.init`.

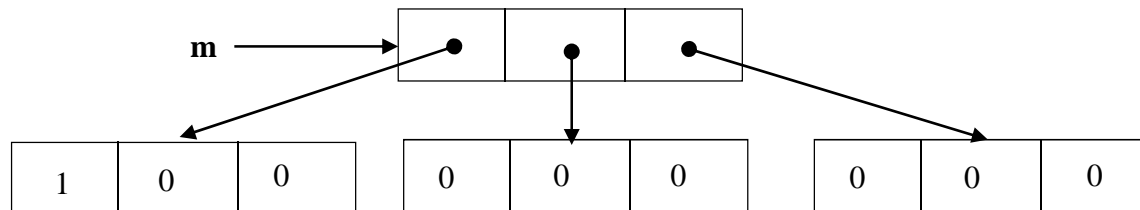
```
# let m = Array.init 3 (function i -> Array.make 3 0);;  
val m : int array array = [| [|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|] |]
```

Po wykonaniu powyższych fraz pamięć wygląda następująco.



co można łatwo sprawdzić:

```
# m.(0).(0) <- 1;;  
- : unit = ()  
# m;;  
- : int array array = [| [|1; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|] |]
```



Wektory (5)

Macierze (wektory wektorów) nie muszą być prostokątne (podobnie jest w Javie).

```
# let t = [| [| 1 |];  
            [| 2; 2 |];  
            [| 3; 3; 3 |]  
          |];;  
val t : int array array = [| [|1|]; [|2; 2|]; [| 3; 3; 3 |] |]  
# t.(1);;  
- : int array = [|2; 2|]
```

Kopiowanie wektorów jest płytke, tj. nie powoduje kopiowania wartości strukturalnych.

Napisy - OCaml

Napisy w języku OCaml są modyfikowalne (co w języku funkcyjnym może być zaskakujące). Napisy mogą być uważane za wektory znaków, jednak ze względu na efektywne wykorzystanie pamięci są one potraktowane oddzielnie.

Składnia operacji dostępu do elementów napisu i ich modyfikacji jest następująca:

wyr1.*[wyr2]*

wyr1.*[wyr2]* <- *wyr3*

```
# let s = "Ala";;  
val s : string = "Ala"  
# s.[2];;  
- : char = 'a'  
# s.[0] <- 'O';;  
- : unit = ()  
# s;;  
-   : string = "Ola"
```

Od wersji 4.02 wprowadzono typ `bytes` dla modyfikowalnych tablic bajtów, natomiast wartości typu `string` są niemodyfikowalne. Ze względu na wsteczną kompatybilność domyślnie te dwa typy są traktowane zamiennie, ale w nowych programach można je odseparować za pomocą odpowiedniej opcji kompilatora (co jest oczywiście zalecane).

Imperatywne struktury sterowania (1)

Wyrażenie warunkowe

if *warunek* **then** *wyr1*

Jeśli pominięta zostanie część **else** *wyr2*, to kompilator przyjmie, że *wyr2* jest typu unit, czyli *wyr1* też musi być typu unit. Ma to sens tylko wtedy, kiedy *wyr1* powoduje pewien efekt.

Sekwencja

wyr1; ... ; *wyrn*

Wartością sekwencji jest wartość ostatniego wyrażenia *wyrn*, tzn. *wyr1*; *wyr2* \equiv **let** *_* = *wyr1* **in** *wyr2*.

Sekwencje zwykle umieszcza się w nawiasach.

(*sekwencja*) lub **begin** *sekwencja* **end**

Pętle

for *ident* = *wyr1* **to** *wyr2* **do** *wyr3* **done**

for *ident* = *wyr1* **downto** *wyr2* **do** *wyr3* **done**

Wyrażenia *wyr1* **i** *wyr2* **muszą być typu** int.

```
# for i=1 to 10 do print_int i; print_string " " done;
  print_newline() ;;
1 2 3 4 5 6 7 8 9 10
- : unit = ()
```

Imperatywne struktury sterowania (2)

while *warunek* **do** *wyr* **done**

Wyrażenie *warunek* musi być typu `bool`.

```
# let r = ref 1
  in begin
    while !r < 11 do
      print_int !r;
      print_string " ";
      r := !r + 1
    done;
    print_newline()
  end

;;
1 2 3 4 5 6 7 8 9 10
- : unit = ()
```

Pętle są również wyrażeniami o wartości `()` typu `unit`.

Jeśli treść pętli (*wyr3* dla `for` i *wyr* dla `while`) nie jest wyrażeniem typu `unit` to kompilator wyprowadza ostrzeżenie. Można się go pozbyć, wykorzystując standardową funkcję `ignore : 'a -> unit`.

Wyrażenie *let* czy sekwencja

Widzieliśmy, że jeśli program wykorzystuje efekty uboczne, konieczne jest precyzyjne określenie kolejności wartościowania. Można to wykonać używając dwóch stylów. W wyrażeniu $(f\ g)$ OCaml najpierw wartościuje g , następnie f (SML odwrotnie), a potem aplikuje wartość f do wartości g . Kolejność wartościowania można wymusić w obu stylach.

Styl funkcyjny:

`let pom = g in f pom` lub `let pom = f in pom g`

Styl imperatywny:

`pom := g; f !pom` lub `pom := f; (!pom) g`

Listy cykliczne (1)

W liście cyklicznej ostatni węzeł zawiera referencję do pierwszego węzła listy. Węzeł w liście zdefiniujemy jako obiekt typu `lnode`.

```
# type 'a lnode = {item: 'a; mutable next: 'a lnode};;
```

Funkcja `mk_circular_list` tworzy jednoelementową listę cykliczną.

```
# let mk_circular_list e =  
  let rec x = {item=e; next=x}  
  in x;;  
val mk_circular_list : 'a -> 'a lnode = <fun>
```

Dla list wieloelementowych punktem wejścia jest zwykle ostatni węzeł, co umożliwia wstawianie nowych elementów na początku (`insert_head`) i na końcu (`insert_tail`) listy w czasie stałym.

```
# let insert_head e l =  
  let x = {item=e; next=l.next}  
  in l.next <- x; l;;  
val insert_head : 'a -> 'a lnode -> 'a lnode = <fun>  
  
# let insert_tail e l =  
  let x = {item=e; next=l.next}  
  in l.next <- x; x;;  
val insert_tail : 'a -> 'a lnode -> 'a lnode = <fun>
```

Listy cykliczne (2)

Zwracanie wartości pierwszego (`first`) i ostatniego (`last`) elementu listy również odbywa się w czasie stałym.

```
# let first ln = (ln.next).item;;  
val first : 'a lnode -> 'a = <fun>  
# let last ln = ln.item;;  
val last : 'a lnode -> 'a = <fun>
```

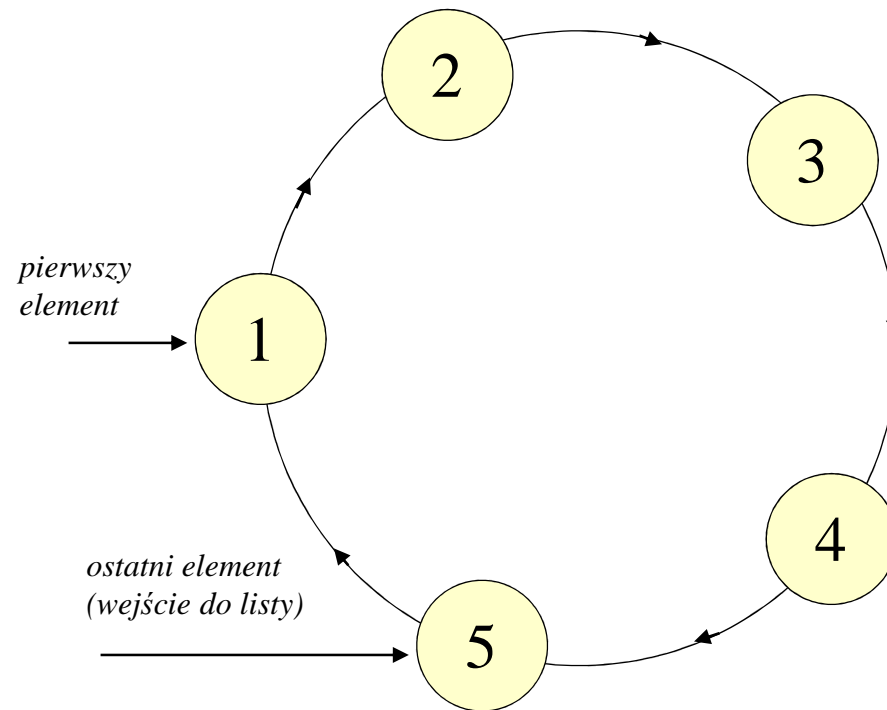
Usunięcie pierwszego węzła wymaga tylko jednego przypisania (zauważ jednak, że ta funkcja nie usuwa węzła z listy jednoelementowej!).

```
# let elim_head l = l.next <- (l.next).next; l;;  
val elim_head : 'a lnode -> 'a lnode = <fun>
```

Jako przykład skonstruujemy cykliczną listę pięcioelementową.

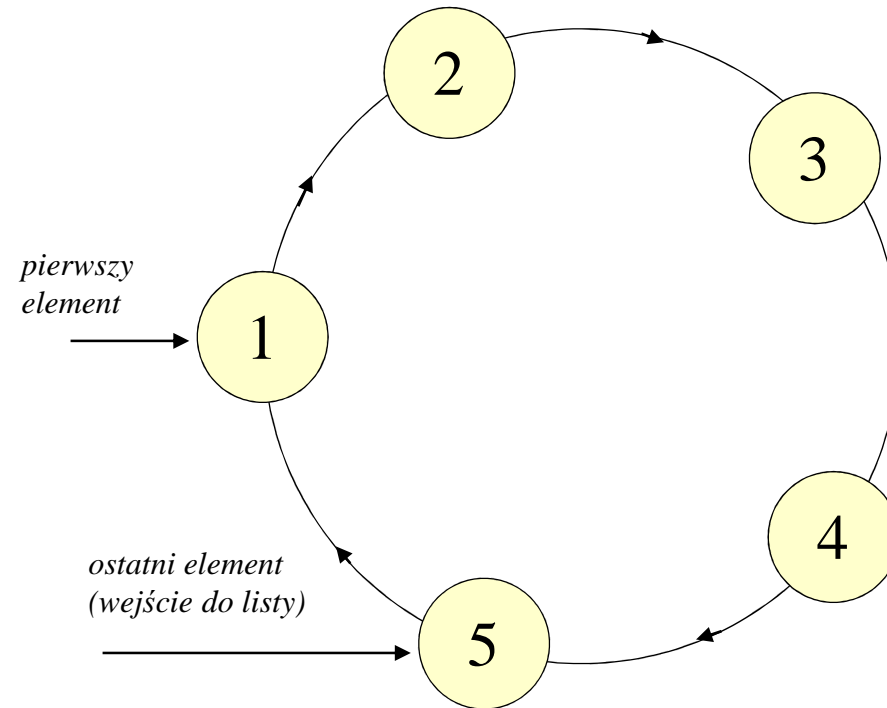
Listy cykliczne (3)

```
# #print_length 13;;  
# let ll =  
  let l = mk_circular_list 1  
  in List.fold_right insert_tail [5;4;3;2] l;;  
val ll : int lnode =  
{item = 5;  
 next =  
  {item = 1;  
   next =  
    {item = 2;  
     next =  
      {item = 3;  
       next =  
        {item = 4;  
         next =  
          {item = 5;  
           next =  
            {item =  
              ...;  
             next =  
              ...}}}}}}}}}
```



Listy cykliczne (4)

```
# let l2 = (* inny sposób generowania takiej samej listy cyklicznej *)
  let l=ref(mk_circular_list 1)
  in for i=2 to 5 do l := insert_tail i !l done; !l;;
val l2 : int lnode =
{item = 5;
 next =
 {item = 1;
  next =
   {item = 2;
    next =
     {item = 3;
      next =
       {item = 4;
        next =
         {item = 5;
          next =
           {item =
            ...;
            next =
             ...}}}}}}}}}}
# l1==l2;;
- : bool = false
(* l1 = l2;; próba strukturalnego porównania list cyklicznych powoduje
   zapętlenie *)
```



Wejście/wyjście (1)

Funkcje wejścia/wyjścia w OCamlu powodują efekty uboczne w postaci zmian środowiska. Funkcje, powodujące efekty uboczne, są czasem nazywane komendami, ponieważ nie są to funkcje w sensie matematycznym. Poniższe identyfikatory są zdefiniowane w module `Pervasives`.

Typy: `in_channel` i `out_channel`

Wyjątek: `End_of_file`.

Kanały standardowe:

```
stdin: in_channel  
stdout: out_channel  
stderr: out_channel
```

Tworzenie kanałów:

```
open_in: string -> in_channel    (* źródłem jest plik o podanej nazwie *)  
open_out: string -> out_channel  (* odbiornikiem jest plik o podanej nazwie *)
```


Wejście/wyjście (2)

Operacje na kanałach wejściowych:

```
read_line : unit -> string
read_int  : unit -> int
read_float : unit -> float
input_char : in_channel -> char
input_line : in_channel -> string
input      : in_channel -> string -> int -> int -> int
close_in   : in_channel -> unit
```

Operacje na kanałach wyjściowych:

```
print_char : char -> unit
print_string : string -> unit
print_int   : int -> unit
print_float : float -> unit
print_endline : string -> unit
print_newline : unit -> unit
flush        : out_channel -> unit
output_char  : out_channel -> char -> unit
output_string : out_channel -> string -> unit
output       : out_channel -> string -> int -> int -> unit
close_out    : out_channel -> unit
```

Przykład: funkcja input_string

```
# let isspace ch = (ch = ' ') || (ch = '\t') || (ch = '\n') || (ch = '\r') ;;
val isspace : char -> bool = <fun>

# let input_string channel =
  let s = ref "" and ch = ref (input_char channel)
  in
    begin
      while isspace (!ch) do ch := input_char channel done;
      while not (isspace (!ch)) do
        s := !s^(String.make 1 !ch);
        ch := input_char channel
      done;
      !s
    end
;;
val input_string : in_channel -> string = <fun>
```

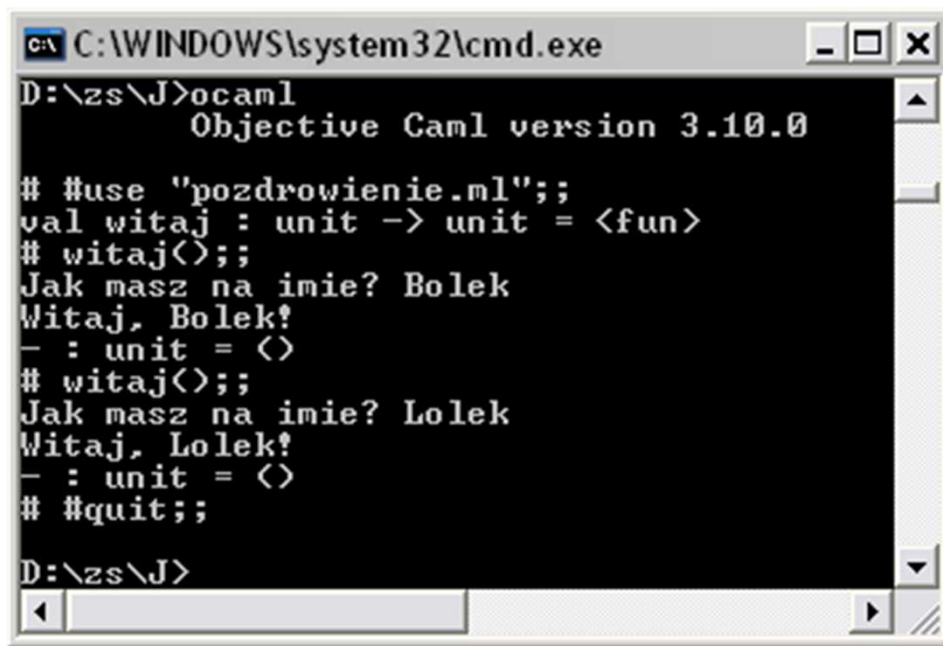
Uruchamianie programów, czytających z kanałów

W środowisku Emacs+Tuareg dane, czytane przez kanał wejściowy, należy kończyć klawiszami Esc+Enter lub (lewy) Alt+Enter. Niestety, na końcu są dołączane dwa średniki.

Inny sposób to uruchamianie programu z okna terminala.

Należy uruchomić interakcyjną pętlę OCaml komendą `ocaml`, następnie za pomocą dyrektywy `#use "file-name";;` wczytać, skompilować i wykonać frazy z zadanego pliku. Dyrektywa `#quit;;` powoduje zakończenie wykonywania komendy `ocaml`. Niech plik "pozdrawienie.ml" zawiera poniższą funkcję.

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



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The prompt is at "D:\zs\J>". The user has entered "ocaml", which has started the "Objective Caml version 3.10.0" interpreter. The user then enters "#use \"pozdrawienie.ml\";;", which loads the file. The interpreter then prompts "Jak masz na imie? Bolek", and the user enters "Bolek". The interpreter then prompts "Witaj, Bolek!", and the user enters "Witaj, Bolek!". The interpreter then prompts "Jak masz na imie? Lolek", and the user enters "Lolek". The interpreter then prompts "Witaj, Lolek!", and the user enters "Witaj, Lolek!". Finally, the user enters "#quit;;", which ends the session. The prompt returns to "D:\zs\J>".

Obok przedstawiono przykład postępowania.

Idea algorytmu sortowania szybkiego

Algorytm sortowania szybkiego (ang. quicksort) został zaproponowany przez C.A.R. Hoare'a w 1962 r. W praktyce jest najszybszym algorytmem sortowania dla „losowych” ciągów. Jego konstrukcja wykorzystuje strategię „dziel i zwyciężaj”.

Idea sortowania tablicy $t[l..r]$ jest następująca:

1. Jeśli $r-l \leq 1$ to tablica t jest posortowana.
2. Wybierz dowolny element rozdzielający $v \in t$ (ang. pivot).
3. **Dziel** tablicę $t[l..r]$ na dwie podtablice $t[l..k]$ i $t[k+1..r]$, takie że $\forall 1 \leq i \leq k. t[i] \leq v$ oraz $\forall k+1 \leq i \leq r. v \leq t[i]$.
4. **Zwyciężaj**: sortuj w taki sam sposób obie podtablice $t[l..k]$ i $t[k+1..r]$.

Dla efektywności algorytmu kluczowy jest dobry wybór elementu rozdzielającego. W kroku 3 obie podtablice powinny mieć zbliżoną długość. Jako elementu dzielącego *nie należy* używać pierwszego ani ostatniego elementu tablicy! Można wybierać element środkowy, medianę z małej próbki, np. trzech wybranych elementów, bądź dokonywać wyboru losowo.

Możliwe usprawnienia algorytmu sortowania szybkiego:

1. Zamiast włączać element rozdzielający do jednej z podtablic, wstawić go zawsze na właściwe miejsce na granicy obu podtablic i wykluczyć z dalszego sortowania.
2. Zastąpić rekursję iteracją (z jawnym użyciem stosu).
3. Sortować natychmiast krótszą podtablicę, a na stos odkładać żądanie posortowania dłuższej podtablicy. Dzięki temu wielkość stosu można ograniczyć do $\lg n$.
4. Dla krótkich tablic, pozostałych do posortowania w ostatnim etapie działania algorytmu (są one w znacznym stopniu posortowane), używać algorytmu sortowania przez wstawianie.

Sortowanie szybkie (1)

```
# let swap tab i j =  
    let aux = tab.(i) in tab.(i) <- tab.(j); tab.(j) <- aux;;  
val swap : 'a array -> int -> int -> unit = <fun>  
  
# let choose_pivot tab m n = tab.((m+n)/2);;  
val choose_pivot : 'a array -> int -> int -> 'a = <fun>  
  
# let partition tab l r =  
    let i=ref l and j=ref r and pivot=choose_pivot tab l r  
    in while !i <= !j do  
        while tab.(!i) < pivot do incr i done;  
        while pivot < tab.(!j) do decr j done;  
        if !i <= !j  
        then begin swap tab !i !j; incr i; decr j end  
    done;  
    (!i,!j)  
;;  
val partition : 'a array -> int -> int -> int * int = <fun>
```

Sortowanie szybkie (2)

```
# let rec quick tab l r =
  if l < r then
    let (i,j) = partition tab l r
    in if j-l < r-i
       then let _ = quick tab l j in quick tab i r
       else let _ = quick tab i r in quick tab l j
  else ();;
val quick : 'a array -> int -> int -> unit = <fun>

#let quicksort tab = quick tab 0 ((Array.length tab)-1);;
val quicksort : 'a array -> unit = <fun>

# let t1 = [|4;8;1;12;7;3;1;9|];;
val t1 : int array = [|4; 8; 1; 12; 7; 3; 1; 9|]
# quicksort t1;;
- : unit = ()
# t1;;
- : int array = [|1; 1; 3; 4; 7; 8; 9; 12|]
# let t2 = [|"kobyła";"ma";"mały";"bok"|];;
val t2 : string array = [|"kobyła"; "ma"; "mały"; "bok"|]
# quicksort t2;;
- : unit = ()
# t2;;
- : string array = [|"bok"; "kobyła"; "ma"; "mały"|]
```

W funkcjach `swap` i `partition` wykorzystaliśmy styl imperatywny, natomiast w funkcji `quick` styl funkcyjny.

Współdzielenie czy kopiowanie wartości (1)

Dopóki przetwarzane wartości nie są modyfikowalne, nie jest ważne, czy są one współdzielone, czy nie.

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# let a = [ 1; 2; 3 ];;
val a : int list = [1; 2; 3]
# let b = id a;;
val b : int list = [1; 2; 3]
# a == b;;
- : bool = true
```

Nie jest istotne, czy b jest kopią listy a, czy jest tą samą (fizycznie) listą, ponieważ listy liczb całkowitych są wartościami stałymi. Gdyby elementami listy były wartości modyfikowalne, musielibyśmy wiedzieć, czy modyfikacja elementu jednej listy pociąga za sobą modyfikację elementu drugiej listy, tzn. czy elementy są współdzielone.

Implementacja polimorfizmu w OCamlu powoduje kopiowanie wartości bezpośrednich i współdzielenie wartości strukturalnych. Argumenty są przekazywane do funkcji zawsze przez wartość (czyli są kopiowane), ale w przypadku wartości strukturalnych argumentami są wskaźniki (podobnie zachowuje się Java).

Ilustruje to poniższy przykład.

Współdzielenie czy kopiowanie wartości (2)

```
# let a = [| 1; 2; 3 |];;
val a : int array = [|1; 2; 3|]
# let b = id a;;
val b : int array = [|1; 2; 3|]
# a == b;;
-: bool = true
# a.(1) <- 4;;
- : unit = ()
# a;;
- : int array = [|1; 4; 3|]
# b;;
-: int array = [|1; 4; 3|]
```

Wybór efektywniejszej (w konkretnym przypadku) reprezentacji danych należy do programisty. Z jednej strony, wybór wartości modyfikowalnych umożliwia efektywną manipulację tymi wartościami w miejscu (bez konieczności alokowania pamięci); z drugiej strony wykorzystanie wartości niemodyfikowalnych pozwala na bezpieczne ich współdzielenie.

Styl funkcyjny i imperatywny: główne różnice

Języki programowania funkcyjnego i imperatywnego różnią się głównie sposobem wykonania programu i zarządzaniem pamięcią.

- Program funkcyjny jest wyrażeniem. W wyniku wykonania programu otrzymywana jest wartość tego wyrażenia. Kolejność wykonywania operacji i fizyczna reprezentacja danych nie wpływają na otrzymany wynik. Zarządzaniem pamięcią zajmuje się system, wykorzystując program odśmiecania (ang. garbage collector).
- Program imperatywny jest ciągiem instrukcji, modyfikujących stan pamięci. Na każdym etapie wykonania kolejność wykonywania instrukcji jest ściśle określona (**Uwaga. Jak dotąd wszystkie implementacje języka OCaml ewaluowały argumenty od prawej do lewej**). Programy imperatywne częściej manipulują wskaźnikami lub referencjami do danych (wartości), niż samymi wartościami. Zwykle wymagają one od użytkownika zarządzania pamięcią, co prowadzi często do błędów; jednak nic nie stoi na przeszkodzie w wykorzystaniu odśmieczacza pamięci.

Języki imperatywne dają większą kontrolę nad wykonaniem programu i reprezentacją danych. Są bliższe architektury komputera, dzięki czemu programy mogą być efektywniejsze, ale też bardziej podatne na błędy.

Języki funkcyjne pozwalają programować na wyższym poziomie abstrakcji, dzięki czemu dają większą pewność poprawności (w sensie zgodności ze specyfikacją) i bezpieczeństwa (braku błędów wykonania) programu; statyczne systemy typizacji tę pewność zwiększają.

Poprawność i bezpieczeństwo programu jest coraz częściej dominującym kryterium. Język Java został stworzony zgodnie z zasadą, że efektywność nie może dominować nad bezpieczeństwem i otrzymany produkt jest pochodną tych założeń. Te kryteria są w coraz większym stopniu brane pod uwagę przez producentów oprogramowania.

Kryteria wyboru stylu

W programach czysto funkcyjnych zabronione jest używanie efektów ubocznych, co uniemożliwia wykorzystywanie wartości modyfikowalnych, wyjątków (i operacji wejścia/wyjścia). Mniej restrykcyjna definicja pozwala na użycie funkcji, które nie modyfikują środowiska zewnętrznego, nawet jeśli wewnątrz funkcja używa wartości modyfikowalnych, a więc jest napisana w stylu imperatywnym. Dozwolone jest też zgłaszanie wyjątków. Taka funkcja widziana z zewnątrz jako „czarna skrzynka” jest nie do odróżnienia od funkcji napisanej w stylu czysto funkcyjnym (z wyjątkiem możliwości zgłoszenia wyjątku).

Z drugiej strony, program w stylu imperatywnym korzysta ze wszystkich udogodnień oferowanych przez język OCaml: bezpieczeństwo statycznej typizacji, automatyczne zarządzanie pamięcią, mechanizm wyjątków, parametryczny polimorfizm i inferencja typów.

Wybór stylu funkcyjnego bądź imperatywnego zależy od implementowanej aplikacji. Przy wyborze należy uwzględnić poniższe kryteria.

- Możliwość wyboru struktur danych (struktury modyfikowalne lub niemodyfikowalne).
- Narzucone struktury danych (modyfikowalne, rekursywne).
- Efektywność.
- Szybkość tworzenia i czytelność kodu.

Styl funkcyjny pozwala na *szybsze* stworzenie *czytelniejszego* kodu; po zidentyfikowaniu sekcji krytycznych można je przepisać *efektywniej*, używając stylu imperatywnego.

Styl funkcyjny, a styl imperatywny

Odersky, Spoon i Venners w książce „Programming in Scala” piszą tak:

Scala allows you to program in an imperative style, but encourages you to adopt a more functional style. [...] If you come from an imperative background, we believe that learning to program in a functional style will not only make you a better Scala programmer, it will expand your horizons and make you a better programmer in general.

[...]

If you're coming from an imperative background, such as Java, C++, or C#, you may think of `var` as a regular variable and `val` as a special kind of variable. On the other hand, if you're coming from a functional background, such as Haskell, OCaml, or Erlang, you might think of `val` as a regular variable and `var` as akin to blasphemy. The Scala perspective, however, is that `val` and `var` are just two different tools in your toolbox, both useful, neither inherently evil. Scala encourages you to lean towards `vals`, but ultimately reach for the best tool given the job at hand.

[...]

A balanced attitude for Scala programmers

Prefer `vals`, immutable objects, and methods without side effects.

Reach for them first. Use `vars`, mutable objects, and methods with side effects when you have a specific need and justification for them.

Zadania kontrolne

1. Napisz funkcję `echo: unit -> unit`, która wyświetla na ekranie znaki pobierane ze standardowego strumienia wejściowego (kolejny wiersz jest wyświetlany po naciśnięciu klawisza Enter). Funkcja kończy działanie po przeczytaniu kropki ('.'). Znaki po kropce nie są już wyświetlane. Należy napisać dwie wersje tej funkcji: jedna wykorzystuje pętlę `while`, a druga rekursję ogonową. Wykorzystaj potrzebne funkcje wejścia/wyjścia z modułu `Pervasives`.
2. Napisz funkcję `zgadnij: unit -> unit`, która generuje losowo (patrz moduł `Random`) liczbę całkowitą z przedziału `[0,100]`, następnie w pętli prosi użytkownika o podanie liczby i odpowiada „moja jest większa” lub „moja jest mniejsza”, a po odgadnięciu liczby „Zgadles. Brawo!” i kończy działanie. Ewentualne funkcje pomocnicze powinny być zdefiniowane lokalnie w funkcji `zgadnij`.
3. Napisz funkcję `sortuj_plik: unit -> unit`, która pyta o nazwę pliku wejściowego, czyta z pierwszego wiersza pliku liczbę elementów, czyta do tablicy elementy do sortowania (liczby rzeczywiste), sortuje je niemalejąco (w dowolny sposób), pyta o nazwę pliku wyjściowego i zapisuje posortowany ciąg do tego pliku. Wykorzystaj potrzebne funkcje biblioteczne.

Zadania kontrolne

4. Problem Józefa definiuje się następująco (Cormen et al. str. 340). Niech n osób stoi w okręgu oraz niech dana będzie liczba $m \leq n$. Rozpoczynając od wskazanej osoby, przebiegamy po okręgu, usuwając co m -tą osobę. Po usunięciu każdej kolejnej osoby odliczanie odbywa się w nowo powstałym okręgu. Proces ten postępuje, aż zostaną usunięte wszystkie osoby. Porządek, w którym osoby stojące początkowo w okręgu są z niego usuwane, definiuje permutację Józefa typu (n,m) liczb $1, 2, \dots, n$. Na przykład permutacją Józefa typu $(7,3)$ jest $\langle 3,6,2,7,5,1,4 \rangle$. Napisz funkcję typu $int \rightarrow int \rightarrow int\ list$, która dla danych n oraz m zwraca listę z permutacją Józefa typu (n,m) . Należy wykorzystać listę cykliczną.