# Deklaracje

Deklaracje wiążą nazwy z wartościami. Zadeklarowane identyfikatory są stałymi — nie można zmieniać ich wartości. Z identyfikatorami można wiązać wartości dowolnych typów, w szczególności funkcji.

Oto prosty przykład:

```
# let pi = 4.0 *. atan 1.0;;
val pi : float = 3.14159265359

# let nazwa = "zielony bal" ;;
val nazwa : string = "zielony bal"

# let tez_nazwa = "onik";;
val tez_nazwa : string = "onik"

# let ato = nazwa ^ tez_nazwa ;;
val ato : string = "zielony balonik"
```

Identyfikatory można redefiniować (nie mylić z przypisaniem!!!):

```
# let aa = 3 * 3;;
val aa : int = 9

# let bb = 3 * aa;;
val bb : int = 27

# let aa = not (27 mod 2 = 0);;
val aa : bool = true

# let cc = (aa, bb);;
val cc : bool * int = true, 27
```

Można też definiować wiele identyfikatorów naraz:

```
# let zz = 83 mod 3
  and yy = "??";;
val zz : int = 2
val yy : string = "??"
```

Można dokonywać deklaracji lokalnych:

```
# let d = 10 in
  let ohoho = d * d + 2 * d * 10 in
  let ttt = d * 3 * d + 3 * d + (d / 3)
  in
    ohoho * d + ttt ;;
- : int = 3333

# let str = "333";;
- : string = "333"

# let str = 3 in str * str ;;
- : int = 9

# str ^ str ;;
- : string = "333333"
```

# Dopasowanie wzorców

Dopasowanie wzorców pozwala na „rozbieranie” wartości typów złożonych. Jest to przydatne (np. w przypadku rekordów), a często niezbędne (n–tki uporządkowane).

Przykłady:

```
# let para = (3, "3");;
val para : int * string = 3, "3"

# let (lewy, prawy) = para;;
val lewy : int = 3
val prawy : string = "3"
# let (lewy, prawy) = para in 2 * lewy + 1;;
- : int = 7

# let trojka = (4, 4 = 44, "4");;
val trojka : int * bool * string = 4, false, "4"
# let (_, false, tekst ) = trojka;;
Warning: this pattern-matching
                    is not exhaustive
val tekst : string = "4"
```

Z kolei następujące przykłady są **niepoprawne**:

```
let (l, 4, r) = trojka;;
let (lewy, prawy) = trojka;;
```

a poniższe zawodzą z błędem wykonania:

```
# let (l, true, r) = (32, 32 = 33, 33);;
Warning: this pattern-matching
                    is not exhaustive
Uncaught exception: Match_failure(_)

# let (lewy, 37, prawy) = (33, 34, 35);;
Warning: this pattern-matching
                    is not exhaustive
Uncaught exception: Match_failure(_)
```

Dopasowanie wzorców w przypadku rekordów jest bardzo podobne. Oto przykłady:

```
# type num = {n1 : int; n2 : int};;
type num = { n1 : int; n2 : int }

# type rekord = {st : string; num : num};;
type rekord = { st : string; num : num }

# let r =
    {st = "o" ^ "k";
     num = {n1 = 3 + 10;
            n2 = 9}};;
val r : rekord = {st="ok";
                  num={n1=13; n2=9}}

# let {st = stan; num = liczby} = r;;
val stan : string = "ok"
val liczby : num = {n1=13; n2=9}

# let {num = {n1 = l1; n2 = l2}} = r;;
val l1 : int = 13
val l2 : int = 9
```

Można też definiować wzorce z aliasami (kilkoma nazwami dla tej samej składowej):

```
# let para = ((27, 37), 47);;
val para : (int * int) * int = (27, 37), 47

# let ((ll, rr) as lewy, prawy) = para;;
val ll : int = 27
val rr : int = 37
val lewy : int * int = 27, 37
val prawy : int = 47
```

Nazwy wiązane we wzorcach nie mogą się powtarzać (warunek ten nosi nazwę *warunku liniowości*, a spełniające go wzorce nazywa się wzorcami liniowymi).

Zatem następujące wzorce są niepoprawne:

```
let (x, x) = (2, 2);;
let (z, z) = ("???", "!!!");;
```

# Definiowanie funkcji

Oto przykład definicji prostej funkcji:

```
# let pi = 4.0 * atan 1.0 ;;
val pi : float = 3.14159265359

# let square x = x *. x ;;
val square : float -> float = <fun>

# square (sin pi) +. square (cos pi) ;;
- : float = 1
```

Funkcje wieloargumentowe definiujemy tak:

```
# let dodaj a b = a + b ;;
val dodaj : int -> int -> int = <fun>

# dodaj 10 20;;
- : int = 30
```

Tak definiujemy funkcje rekurencyjne:

```
# let rec fib n =
    if n < 2 then 1
    else fib (n - 1) + fib (n - 2) ;;
val fib : int -> int = <fun>

# fib 10;;
- : int = 89
```

A tak definiujemy funkcje wzajemnie rekurencyjne:

```
# let rec f1 n =
    if n < 1 then  n
    else 1 + f2 (n - 1)
  and
    if n < 1 then n
    else 2 * f1 (n - 1)
  ;;
val f1 : int -> int = <fun>
val f2 : int -> int = <fun>
```

A to jest bardziej sensowny przykład wieloargumentowej funkcji rekurencyjnej — obliczanie liczb Fibonacciego w czasie liniowym.

```
# let fib2 n =
    let rec fib' n_1 n_2 ile =
      if ile <= 0 then n_1
      else fib' (n_1 + n_2) n_1 (ile - 1)
    in
      fib' 1 1 (n - 2);;
val fib2 : int -> int = <fun>

# fib2 35;;
- : int = 9227465
# fib2 7000;;
- : int = 347838379
```

Ta definicja jest równoważna naiwnej, ale liczy się dużo szybciej, a poza tym jest to definicja ogonowo–rekurencyjna, czyli nie powoduje przystania liczby rekordów aktywacji na stosie.

Funkcje mogą brać argumenty dowolnego typu i dawać dowolne typy w wyniku (np. n–tki lub rekordy):

```
# let
    div (m, n) =
           truncate
            (floor (float m /. float n)) ;;
val div : int * int -> int = <fun>
# div (7, 3) ;;
- : int = 2

#type rr =
   {jeden : string;
    drugi : string} ;;
# let
    wybierz (b, {jeden = s1; drugi = s2})
        = "oj" ^ (if b then s1 else s2) ;;
val wybierz : bool * rr -> string = <fun>

# wybierz (div (9, 2) > 2,
           {jeden = "maly";
            drugi = "duzy"}) ;;
- : string = "ojmaly"
```

# Funkcje jako wartości

W Caml-u funkcje są równie dobrymi wartościami jak wszystkie inne. Można je podawać jako argument wywołań funkcji, dawać w wyniku, w dowolny sposób używać w wyrażeniach.

Oto przykłady:

```
(* funkcja jako wynik wyrażenia *)
# (if 1261 > 100 then cos else sin)
  3.14159265;;
- : float = -1

(* funkcje lokalne *)
# let czy_rozwiazywalne =
    let delta a b c =
          b *. b -. 4.0 *. a *. c in
    let rozw a b c = delta a b c > 0.0
    in
      rozw;;
val czy_rozwiazywalne
      : float -> float ->
             float -> bool = <fun>
```

Funkcji nie trzeba nazywać — można tworzyć *funkcje anonimowe*.

```
# function n -> (n mod 2 = 0);;
- : int -> bool = <fun>
# (function n -> (n mod 2 = 0)) 32;;
- : bool = true

# function (n, m) ->
     (n mod 3) + ((n + m) / 13);;
- : int * int -> int = <fun>

# type czlowiek =
    {imie : string;
     nazwisko : string };;
# fun {imie = a; nazwisko = b}
                    -> a ^ " " ^ b ;;
- : czlowiek -> string = <fun>

# fun () -> 127 ;;
- : unit -> int = <fun>
# fun a b -> a + b;;
- : int -> int -> int = <fun>
```

Funkcje (często anonimowe, ale niekoniecznie) są świetnymi argumentami dla innych funkcji (podobnie jak w Pascalu).

```
# let rozniczka f dx =
    fun x -> (f (x +. dx) -. f x) /. dx;;
val rozniczka :
     (float -> float) ->
           float -> float -> float = <fun>

# let cos' = rozniczka sin 1e-6;;
val cos' : float -> float = <fun>
# cos' 3.14159265359 ;;
- : float = -1.00000000014

(* mozna nawet zdefiniować składanie *)
# let zloz f g = fun x -> f (g x) ;;
val zloz : ('a -> 'b) ->
           ('c -> 'a)
                 -> 'c -> 'b = <fun>

# let cos2 = zloz (fun x -> x *. x) cos ;;
val cos2 : float -> float = <fun>
```