Wykład 4

Algebraiczne typy danych

Definicje typów jako aliasy (synonimy)

Algebraiczne typy danych

Przykład: lista „heterogeniczna”

Polimorficzne typy rekurencyjne

Rekordy

Wyjątki: deklarowanie, zgłaszanie, przechwytywanie i obsługa

Przykład: składnia abstrakcyjna i ewaluacja wyrażeń

Typ „opcjonalny”

Rozszerzalne typy wariantowe

Słownik jako funkcja

Drzewa wielokierunkowe. Grafy

Budowanie abstrakcji za pomocą typów danych

• Do tej pory budowaliśmy pojęcia abstrakcyjne za pomocą funkcji i funkcjonałów. Mogliśmy zauważyć, że wiele operacji wykonywanych na listach wykonuje się według podobnych schematów i zapisać je w postaci funkcjonałów *map* czy *fold_left*. Jeśli w języku programowania brak środków językowych do wyrażania takich abstrakcji, to trudniej zauważyć podobieństwo, np. między dodawaniem elementów listy liczb, a „spłaszczaniem" listy list.

• Drugim ważnym sposobem budowania abstrakcji w językach programowania są typy danych. Do tej pory używaliśmy wbudowanych typów strukturalnych: list i krotek. W językach z rodziny ML można budować własne typy (również rekursywne i polimorficzne) za pomocą konstruktorów typów i konstruktorów wartości. Notacja przypomina notację BNF.

Ten mechanizm pozwala na łatwe definiowanie typów danych (np. karty do gry, drzewa, wyrażenia), odpowiednich dla opisu pewnego wycinka rzeczywistości.

Definicje typów danych

Definicje typów danych stanowią frazę języka OCaml.

type *ident* = *def-typu*;;

W odróżnieniu od definicji zmiennych, definicje typów są domyślnie rekurencyjne (nie jest wymagane słowo kluczowe `rec`).

type *ident1* = *def-typu1*
and *ident2* = *def-typu2*
...
and *identn* = *def-typun*;;

Definicje typów mogą być parametryzowane zmiennymi typowymi.

type `'`*a ident* = *def-typu*;;
type (`'`*a1*, ..., `'`*an***)** *ident* = *def-typu*;;

Definicje typów jako aliasy (synonimy) - OCaml

Można nadać nazwę istniejącym typom danych (nie używa się konstruktorów danych).

```
# type 'param para_i_x = int * 'param;;
type 'param para_i_x = int * 'param

# type para_i_f = float para_i_x;;
type para_i_f = float para_i_x

# let x = (3,3.14);;
val x : int * float = (3, 3.14)

# let (x:para_i_f) = (3,3.14);;
val x : para_i_f = (3, 3.14)

# let (x: 'a para_i_x) = (3,3.14);;
val x : float para_i_x = (3, 3.14)
```

Definicje typów jako aliasy (synonimy) - Haskell

Podobnie zachowuje się Haskell.

```
type Para_i_x param = (Int,param)
type Para_i_f  = Para_i_x Float

x = (3,3.14)  -- plik
*Main> :t x
x :: (Integer, Double)
*Main> let x =(3,3.14)::Para_i_f
*Main> :t x
x :: Para_i_f

x :: Para_i_f  -- plik
x = (3,3.14)   -- plik
*Main> :t x
x :: Para_i_f

x :: Para_i_x Float -- plik
x = (3,3.14) – plik
*Main> :t x
x :: Para_i_x Float
```

Algebraiczne typy danych

Algebraiczne typy danych)definiuje się za pomocą ***konstruktorów typów*** i ***konstruktorów wartości***. Konstruktorem typu dla list w języku OCaml jest `list` (w notacji postfiksowej), np. `int list`, `float list` są już typami, a `int` i `float` są argumentami dla konstruktora typu `list`. Istnieją dwa konstruktory wartości dla list: `[]` dla listy pustej i infiksowy konstruktor `::` dla listy niepustej. We współczesnych funkcyjnych językach programowania ze statyczną typizacją istnieją mechanizmy, pozwalające programiście definiować swoje algebraiczne typy danych. Idea jest koncepcyjnie prosta, jak zobaczymy za chwilę na przykładach.

W języku Haskell dla konstruktorów typów stosowana jest notacja prefiksowa. Konstruktory wartości mogą być w postaci rozwiniętej, a konstruktory typów muszą być w takiej postaci.

Warianty (sumy rozłączne) - OCaml

type *ident* =
  $Cons_1$ **of** typ_{11}* ... * typ_{1i}
    ...
| $Cons_n$ of typ_{n1}* ... * typ_{nm} **;;**

Uwaga: nazwy konstruktorów wartości muszą się zaczynać wielką literą.

```
# type kolor = Trefl | Karo | Kier | Pik;;
type kolor = Trefl | Karo | Kier | Pik
# type karta = Blotka of kolor*int | Walet of kolor | Dama of kolor | Krol of kolor | As of kolor;;
type karta =
    Blotka of kolor * int
  | Walet of kolor
  | Dama of kolor
  | Krol of kolor
  | As of kolor
# let k1 = Krol Pik  let k2 = Blotka(Karo,2);;
val k1 : karta = Krol Pik
val k2 : karta = Blotka (Karo, 2)
```

kolor i karta to bezargumentowe konstruktory typów, Trefl, Karo, Kier, Pik to bezargumentowe konstruktory wartości typu kolor, a As, Krol, Dama, Walet i Blotka to jednoargumentowe konstruktory wartości typu karta.

Warianty (sumy rozłączne)

```
# let rec przedzial  a b = if a>b then []
                           else b::(przedzial a (b-1));;
val przedzial : int -> int -> int list = <fun>

# let wszystkieKarty kol =
    let figury = [As kol; Krol kol; Dama kol; Walet kol]
      and blotki = List.map (function n -> Blotka(kol,n)) (przedzial 2 10)
    in figury @ blotki
;;
val wszystkieKarty : kolor -> karta list = <fun>

# let kiery = wszystkieKarty Kier;;
val kiery : karta list =
  [As Kier; Krol Kier; Dama Kier; Walet Kier; Blotka (Kier, 10);
   Blotka (Kier, 9); Blotka (Kier, 8); Blotka (Kier, 7);
   Blotka (Kier, 6); Blotka (Kier, 5); Blotka (Kier, 4); Blotka (Kier, 3); Blotka (Kier, 2)]
```

Przykład: lista „heterogeniczna”(1)

Listy w językach programowania z typizacją stayczną są homogeniczne, tzn. typy elementów listy muszą być identyczne. Jeśli elementy listy muszą być heterogeniczne, to możemy je potraktować jako wartości zdefiniowanego przez nas typu.

```
# type ('a,'b) ab = A of 'a | B of 'b;;
type ('a, 'b) ab = A of 'a | B of 'b

# let ls = ["Ala "; "ma "; "kota"] and lint = [1;2;3];;
val ls : string list = ["Ala "; "ma "; "kota"]
val lint : int list = [1; 2; 3]

# let lsint = (List.map (function x -> A x) ls) @ (List.map (function x -> B x) lint);;
val lsint : (string, int) ab list =
              [A "Ala "; A "ma "; A "kota"; B 1; B 2; B 3]
```

Przykład: lista „heterogeniczna” (2)

Przetwarzanie takiej listy „heterogenicznej” łatwo zorganizować, wykorzystując dopasowanie wzorca. Funkcja concat_and_add konkatenuje napisy i dodaje liczby w zdefiniowanej powyżej liście „heterogenicznej”.

```
# let rec concat_and_add l =
  match l with
      [] -> ("",0)
    | h::t -> match (h, concat_and_add t) with
                  (A str, (s,n)) -> (str^s, n)
                | (B num, (s,n)) -> (s, num+n);;
val concat_and_add : (string, int) ab list -> string * int = <fun>

# concat_and_add lsint;;
- : string * int = ("Ala ma kota", 6)
```

Spłaszczanie list zagnieżdżonych (1)

Bardziej skomplikowany przykład list heterogenicznych stanowią listy zagnieżdżone, których elementy mogą być tego samego typu, co cała lista.

W poniższym przykładzie należy spłaszczyć taką listę, tzn. skonstruować listę, której elementy nie są listami.

```
# type 'a nestedList = Nil
                     | NN of 'a nestedList * 'a nestedList
                     | N of 'a * 'a nestedList;;
  type 'a nestedList =
    Nil
  | NN of 'a nestedList * 'a nestedList
  | N of 'a * 'a nestedList

# let rec flatten = function
    Nil -> []
  | NN(x,xr) -> (flatten x)@(flatten xr)
  | N(x,xr) -> x::flatten xr;;
  val flatten : 'a nestedList -> 'a list = <fun>
```

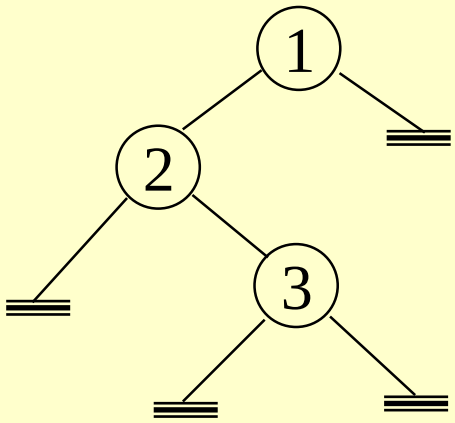
Spłaszczanie list zagnieżdżonych (2)

```
# let nl = NN(NN(N(1,N(2,Nil)),
              N(3,Nil)),
            N(4,
              NN(N(5,N(6,Nil)),
                 NN(NN(NN(N(7,
                          N(8,Nil)),
                       Nil),
                    Nil),
                 Nil)
               )
              )
            );;
 val nl : int nestedList =
  NN (NN (N (1, N (2, Nil)), N (3, Nil)),
   N (4,
    NN (N (5, N (6, Nil)), NN (NN (NN (N (7, N (8, Nil)), Nil), Nil),
          Nil))))

# flatten nl;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8]
```

Polimorficzne typy rekurencyjne - drzewa binarne

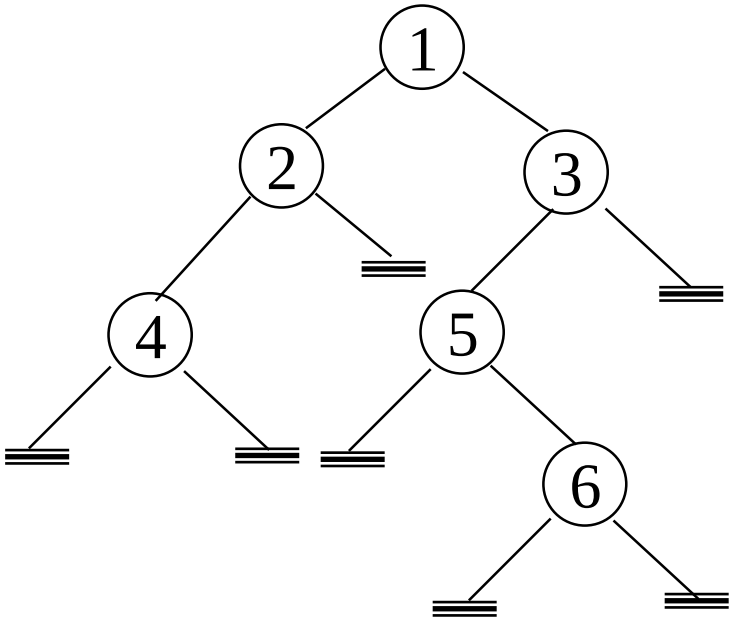
1. Pusta struktura jest drzewem pustym.
2. Jeśli *e* jest elementem typu *E* i t_1, t_2 są drzewami binarnymi, to struktura <*e*, t_1, t_2> jest drzewem binarnym.
3. Drzewami binarnymi są wyłącznie struktury definiowane za pomocą reguł 1 i 2.

```
# type 'a bt = Empty | Node of 'a * 'a bt * 'a bt;;
type 'a bt = Empty | Node of 'a * 'a bt * 'a bt

# let t = Node(1,Node(2,Empty,Node(3,Empty,Empty)),Empty);;
val t : int bt = Node (1, Node (2, Empty, Node (3,Empty,
                                                Empty)), Empty)
# let rec nodes = function
      Empty -> 0
    | Node(_,t1,t2) -> 1 + nodes t1 + nodes t2;;
val nodes : 'a bt -> int = <fun>

# nodes t;;
- : int = 3
```



Przykład drzewa binarnego

```
#let tt = Node(1,
               Node(2,
                    Node(4,
                         Empty,
                         Empty
                        ),
                    Empty
                   ),
               Node(3,
                    Node(5,
                         Empty,
                         Node(6,
                              Empty,
                              Empty
                             )
                        ),
                    Empty
                   )
              );;
```



Obejście drzew binarnych wszerz

```
# let breadth t =
  let rec breadth_aux = function
      []                -> []
    | Empty::t          -> breadth_aux t
    | Node(e,lt,rt)::t  -> e::breadth_aux (t @ [lt; rt])
  in breadth_aux [t]  (* ta lista reprezentuje kolejkę *)
;;
val breadth : 'a bt -> 'a list = <fun>

# breadth t;;
- : int list = [1; 2; 3]

# breadth tt;;
- : int list = [1; 2; 3; 4; 5; 6]
```

Obejście drzew binarnych w głąb prefiksowo

```
# let rec preorder = function
    Node(v,l,r) -> v ::(preorder l) @ (preorder r)
  | Empty -> []
;;
val preorder : 'a bt -> 'a list = <fun>
# preorder tt;;
-: int list = [1; 2; 4; 3; 5; 6]
```

Mimo dużej czytelności ta funkcja nie jest zbyt efektywna. Dla drzew ukośnych jej złożoność może być kwadratowa ze względu na użycie @ . Efektywniejsza wersja wykorzystuje tylko jedną operację :: dla każdego węzła.

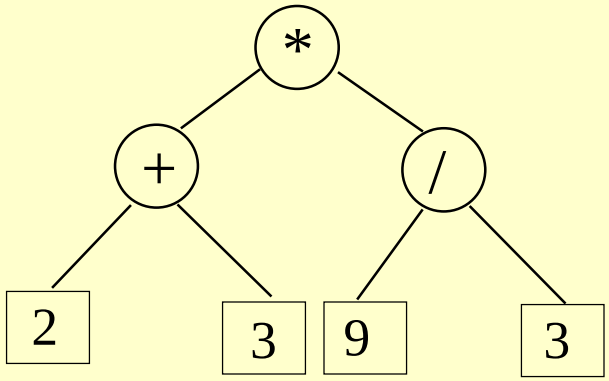
```
#let preorder' t =
  let rec preord = function
      (Empty, labels) -> labels
    | (Node(v,t1,t2), labels) -> v :: preord (t1, preord (t2, labels))
  in preord (t,[]);;
val preorder' : 'a bt -> 'a list = <fun>
# preorder' tt;;
- : int list = [1; 2; 4; 3; 5; 6]
```

Lista –> drzewo poszukiwań binarnych

```
# let rec list2bst l =
  let rec insert2bst = function
      (k, Node (r,lt,rt)) ->
       if k<r then Node(r,insert2bst(k,lt),rt) else
         if k>r then Node(r,lt,insert2bst(k,rt))
         else failwith "duplicated key"
    | (k, Empty) -> Node(k,Empty,Empty)
  in
    match l with
       h::t ->  insert2bst(h, list2bst t)
      | [] -> Empty;;
val list2bst : 'a list -> 'a bt = <fun>
# list2bst [6;4;9;2;5];;
- : int bt =
Node (5, Node (2, Empty, Node (4, Empty, Empty)),
 Node (9, Node (6, Empty, Empty), Empty))
```

Uogólnione drzewa binarne

```
# type ('a,'b) bt = Leaf of 'a | Node of 'b*('a,'b) bt * ('a,'b) bt;;
type ('a, 'b) bt = Leaf of 'a | Node of 'b * ('a, 'b) bt * ('a, 'b) bt
# let rec nrOfLeaves: ('a,'b) bt -> int =
   function
      Leaf _ -> 1
   |  Node (_,l,r) -> nrOfLeaves l + nrOfLeaves r;;
val nrOfLeaves : ('a, 'b) bt -> int = <fun>
# let t1 =
    Node('*', Node('+',Leaf 2,Leaf 3), Node('/',Leaf 9,Leaf 3));;
val t1 : (int, char) bt =
    Node ('*', Node ('+', Leaf 2, Leaf 3), Node ('/', Leaf 9, Leaf 3))
# nrOfLeaves t1;;
- : int = 4
```



Równość i porządek dla definiowanych typów - OCaml

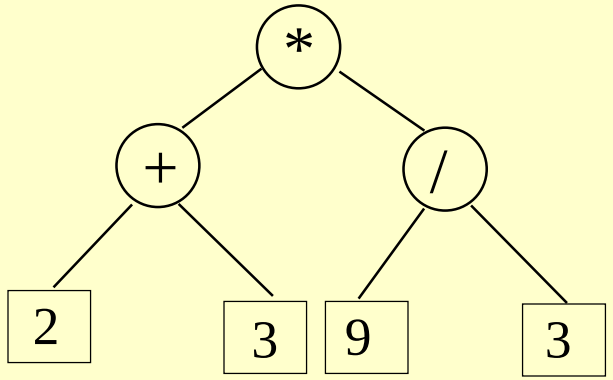
W języku OCaml operatory porównania (równości i porządku) są przeciążane dla nowo definiowanych algebraicznych typów danych. **Istotna jest kolejność konstruktorów wartości i ich argumentów.** Porównanie wartości funkcyjnych powoduje zgłoszenie wyjątku Invalid_argument. Porównanie struktur cyklicznych może spowodować zapętlenie.

```
type kolor = Trefl | Karo | Kier | Pik;;
type 'a tree = Leaf of 'a | Branch of 'a tree * 'a tree;;

# Trefl < Kier;;
- : bool = true
# min Pik Karo;;
- : kolor = Karo
# Leaf 4 < Branch (Leaf 0, Leaf 1);;
- : bool = true
# Branch (Leaf 0, Leaf 2) <=  Branch (Leaf 0, Leaf 1);;
- : bool = false
```

Definicje typów danych - Haskell

Do definiowania typów danych w Haskellu używa się słowa kluczowego `data`. Konstruktory typów danych muszą być w postaci rozwiniętej, a konstruktory wartości mogą być w postaci rozwiniętej. Konstruktory wartości mogą być używane jak zwykłe funkcje.

```
data BT a b = Leaf a | Node b (BT a b) (BT a b)
            deriving (Eq,Show,Read)

nrOfLeaves :: BT a b -> Int
nrOfLeaves (Leaf _) = 1
nrOfLeaves (Node n l r) =  nrOfLeaves l + nrOfLeaves r

t1 =
    Node '*' (Node '+' (Leaf 2) (Leaf 3)) (Node '/' (Leaf 9) (Leaf 3))

*Main> nrOfLeaves t1
4
```



Równość i porządek dla definiowanych typów - Haskell

W języku Haskell domyślna równość i porządek (jak w OCamlu) są generowane automatycznie, jeśli zostanie użyta klauzula deriving. **Istotna jest kolejność konstruktorów wartości i ich argumentów.** Można je zdefiniować inaczej za pomocą własnych klas typów, o których będzie mowa później.

```
data Kolor = Trefl | Karo | Kier | Pik
           deriving (Eq, Ord, Show)
data Tree a = Leaf a | Branch (Tree a) (Tree a)
            deriving (Eq, Ord, Show)

*Main> Trefl < Kier
True
*Main> min Pik Karo
Karo
*Main> Leaf 4 < Branch (Leaf 0) (Leaf 1)
True
*Main> Branch (Leaf 0)(Leaf 2) <=  Branch (Leaf 0)(Leaf 1)
False
```

Definicje typów są generatywne - Haskell

Każda definicja typu tworzy nowy typ nawet wtedy, kiedy wcześniej zdefiniowano identyczny typ.

```
data Bool = False|True
          deriving(Eq,Ord,Show,Read)
*Main> not True
<interactive>:1:4: error:
    Ambiguous occurrence 'True'
    It could refer to either 'Prelude.True',
                             imported from `Prelude' at w4.hs:1:1
                           (and originally defined in 'GHC.Types')
                         or  'Main.True', defined at w4.hs:1:19

*Main> not Main.True
<interactive>:1:4: error:
    Couldn't match expected type `Prelude.Bool'
                with actual type `Main.Bool'
    In the first argument of 'not', namely 'Main.True'
    In the expression: not Main.True
*Main> not Prelude.True
False
```

Definicje typów są generatywne - OCaml

OCaml zachowuje się podobnie, ale komunikaty o błędach są mniej czytelne.

```
# type bool = false | true;;
type bool = false | true

# not true;;
- : bool = false        (* wbudowany typ bool *)

# not (true:bool);;     (* zdefiniowany typ bool *)
Characters 5-9:
  not (true:bool);;
       ^^^^
Error: This expression has type bool/1017
       but an expression was expected of type bool/5

# not;;
- : bool -> bool = <fun>
```

Negacja `not` oczekuje argumentu wbudowanego typu `bool`. Po definicji został wygenerowany nowy typ `bool`, który jest traktowany jako różny od istniejących typów.

Definicje typów są generatywne - OCaml

Taka sama nazwa konstruktora wartości może być użyta w różnych typach. Są to jednak dwa różne konstruktory o różnych typach. OCaml „stara się” użyć konstruktora odpowiedniego typu, jeśli to możliwe. Przeanalizuj poniższy przykład.

```
# type t1 = A | B;;
type t1 = A | B
# let f1 x = if x=A then A else A;;
val f1 : t1 -> t1 = <fun>
# f1 A;;
- : t1 = A
# type t2 = A | C;;
type t2 = A | C
# let f2 x = if x=A then A else A;;
val f2 : t2 -> t2 = <fun>
# f2 A;;
- : t2 = A
# f1 A;;
- : t1 = A
# f1 (f2 A);;
Characters 3-9:
  f1 (f2 A);;
     ^^^^^^
Error: This expression has type t2 but an expression was expected of type t1
```

Definicje newtype - Haskell

W Haskellu do tworzenia nowych typów można też użyć słowa kluczowego `newtype`. Definiowany typ musi mieć dokładnie jeden konstruktor wartości z dokładnie jednym argumentem. Konstruktor typu może mieć dowolnie wiele argumentów. Konstruktor wartości takiego typu jest wykorzystywany do jawnej koercji typów. Ta informacja wykorzystywana jest wyłącznie w czasie kompilacji i nie powoduje żadnego narzutu w czasie wykonania.

```
newtype IntX x y = D (Int,x,y) deriving (Show)
ixy = D(5,'x',[5])
D(i,x,y) = ixy    -- dopasowanie do wzorca

*Main> ixy
D (5,'x',[5])
it :: IntX Char [Integer]
*Main> i
5
it :: Int
*Main> x
'x'
it :: Char
*Main> y
[5]
it :: [Integer]
```

Rekordy w języku Haskell

Rekordy w Haskellu są lukrem syntaktycznym. Pozwalają nazywać pola i generują akcesory, czyli funkcje, umożliwiające dostęp do pól. Mogą być użyte jako argumenty konstruktorów wartości w definicjach typów. Konstruktory wartości z argumentami rekordowymi mogą być używane ze zwykłą składnią zarówno przy tworzeniu wartości jak i we wzorcach (trzeba tylko zachować odpowiednią kolejność argumentów). Nie można używać tych samych nazw pól w różnych rekordach.

```
data Complex = Complex {
  re :: Float,
  im :: Float
  } deriving (Eq,Show,Read)

add_complex ::  Complex ->  Complex ->  Complex  -- z akcesorami
add_complex c1 c2 = Complex {re=re c1 + re c2, im=im c1 + im c2}

add_complex' ::  Complex ->  Complex ->  Complex -- z dopasowaniem do wzorca
add_complex' (Complex x1 y1) (Complex x2 y2) = Complex {re=x1+x2, im=y1+y2}

let c = Complex {re = 2.0, im = 3.0}
let c' = Complex 2.0 3.0 -- można użyć tradycyjnej składni
*Main> c == c'
True
*Main> c == Complex {im = 3.0, re = 2.0} -- kolejność pól jest nieistotna
True
```

Rekordy w języku OCaml (1)

Rekordy są krotkami, w których każdy element posiada etykietę. Rekord w języku OCaml odpowiada zawsze definicji nowego typu.

type *ident* **=** **{** *pole1* **:** *typ1***;** ... **;** *polen* **:** *typn* **}** **;;**

```
# type complex = {re : float; im : float};;
type complex = { re : float; im : float; }
```

Utworzenie wartości typu rekordowego wymaga nadania wartości wszystkim polom rekordu (w dowolnej kolejności).

{ *pole1* **=** *wyr1***;** ... **;** *polen* **=** *wyrn* **}** **;;**

```
# let c = {re=2.; im=3.0};;
val c : complex = {re = 2.; im = 3.}
# c = {im=3.; re=2.};;
- : bool = true
# let d = {im=4.};;
Characters 8-15:
  let d = {im=4.};;
          ^^^^^^^
Error: Some record field are undefined: re
```

Rekordy w języku OCaml (2)

Dostęp do pól rekordu można uzyskać używając notacji kropkowej:

wyr.pole gdzie *wyr* musi być typu rekordowego, posiadającego pole *pole*,

lub wykorzystując wzorce:

{*pole1* **=** *w1***; ... ;** *polek* **=** *wk***}** gdzie *wi* są wzorcami (w takim wzorcu nie muszą wystąpić wszystkie pola rekordu).

```
# let add_complex  c1 c2 = {re = c1.re +. c2.re; im = c1.im +. c2.im};;
val add_complex : complex -> complex -> complex = <fun>

# add_complex c c;;
- : complex = {re = 4.; im = 6.}

# let mult_complex {re=x1; im=y1} {re=x2; im=y2} =
                                        {re=x1*.x2-.y1*.y2; im=x1*.y2+.x2*.y1};;
val mult_complex : complex -> complex -> complex = <fun>

# mult_complex c c;;
- : complex = {re = -5.; im = 12.}
```

Rekordy w języku OCaml (3)

We wzorcu rekordu nie muszą wystąpić wszystkie jego pola. Typ wzorca jest typem ostatnio zadeklarowanego rekordu z polami, wymienionymi we wzorcu.

```
 # type aRec = {p1:bool; re:float};;
type aRec = { p1 : bool; re : float; }
# let rec1 = {re=5.6; p1=false};;
val rec1 : aRec = {p1 = false; re = 5.6}
# let h {re=x} = x;;
val h : aRec -> float = <fun>
# h rec1;;
- : float = 5.6
```

Istnieje wygodna konstrukcja języka, pozwalająca na tworzenie rekordów różniących się od istniejących tylko wartościami niektórych pól. Jest ona wykorzystywana w pracy z rekordami o dużej liczbie pól.

{*ident* **with** *pole1* **=** *wyr1***; ... ;** *polek* **=** *wyrk***}**

```
# let rec2 = {rec1 with re=0.};;
val rec2 : aRec = {p1 = false; re = 0.}
```

Wyjątki

Mogliśmy się już zetknąć z wyjątkami w języku OCaml, np.

```
# 1/0;;
Exception: Division_by_zero.

# List.hd [];;
Exception: Failure "hd".

# List.nth [1;2] (-3);;
Exception: Invalid_argument "List.nth".
```

Wyjątki w języku OCaml są konstruktorami wartości wbudowanego typu `exn`. Zbiór konstruktorów wartości typu `exn` może być rozszerzany. Ponieważ wyjątki są konstruktorami wartości, ich nazwy muszą się zaczynać od dużej litery.

Deklarowanie i zgłaszanie wyjątków

Deklarowanie wyjątków.

exception *ident*;; lub **exception** *ident* **of** *typ*;;

Poniższe wyjątki są zdefiniowane w języku OCaml .

```
exception Failure of string;;
exception Invalid_argument of string;;
exception Not_found;;
```

Zgłaszanie wyjątków.

raise *wyj* lub **raise** (*wyj wyr)*

Poniższe funkcje są zdefiniowane w module Pervasives.

```
let failwith s = raise(Failure s);;
let invalid_arg s = raise(Invalid_argument s);;
```

Zgłaszanie wyjątków – przykłady (moduł List)

```
let hd = function
   [] -> failwith "hd"
  | a::l -> a

let rec nth l n =
   match l with
     [] -> failwith "nth"
   | a::l ->
       if n = 0 then a
       else
          if n > 0 then nth l (n-1)
          else invalid_arg "List.nth"

let rec assoc x = function
    [] -> raise Not_found
  | (a,b)::l -> if a = x then b else assoc x l
```

Przechwytywanie i obsługa wyjątków

try *wyr* **with**
 | *w1 -> wyr1*
 :
 | *wn -> wyrn*

```
# let l = [(1, "Alu"); (2, "Olu")];;
val l : (int * string) list = [(1, "Alu"); (2, "Olu")]
# "Witaj, " ^ List.assoc 2 l;;
- : string = "Witaj, Olu"
# "Witaj, " ^ List.assoc 3 l;;
Exception: Not_found.
val szukaj : 'a -> ('a * string) list -> string = <fun>
# let szukaj klucz slownik =
  try  List.assoc klucz slownik with Not_found ->"niepowodzenie";;
  val szukaj : 'a -> ('a * string) list -> string = <fun>
# "Witaj, " ^ szukaj 1 l;;
- : string = "Witaj, Alu"
# "Witaj, " ^ szukaj 3 l;;
- : string = "Witaj, niepowodzenie"
```

Przykład: składnia abstrakcyjna

```
# type expression =                    (* abstract syntax *)
    Const of float
  | Var of string
  | Sum of expression * expression     (* e1 + e2 *)
  | Diff of expression * expression    (* e1 - e2 *)
  | Prod of expression * expression    (* e1 * e2 *)
  | Quot of expression * expression    (* e1 / e2 *)
;;
type expression =
   Const of float
 | Var of string
 | Sum of expression * expression
 | Diff of expression * expression
 | Prod of expression * expression
 | Quot of expression * expression
```

Przykład: ewaluacja wyrażeń

Wyrażenia są ewaluowane w zadanym środowisku *env*, zadającym wartości zmiennych wolnych wyrażenia *exp*.

```
# exception Unbound_variable of string;;
exception Unbound_variable of string

# let rec eval env exp =
  match exp with
    Const c -> c
  | Var v -> (try List.assoc v env
              with Not_found -> raise(Unbound_variable v))
  | Sum(a,b) -> eval env a +. eval env b
  | Diff(a,b) -> eval env a -. eval env b
  | Prod(a,b) -> eval env a *. eval env b
  | Quot(a,b) -> eval env a /. eval env b
;;
val eval : (string * float) list -> expression -> float = <fun>
                                              (*   (x +. 2.0) *. y   *)
# eval [("x", 1.0); ("y", 3.14)] (Prod(Sum(Var "x", Const 2.0), Var "y"));;
    -      : float = 9.42
                                              (*   (x +. 2.0) *. y   *)
# eval [("x", 1.0); ("z", 3.14)] (Prod(Sum(Var "x", Const 2.0), Var "y"));;
Exception: Unbound_variable "y".
```

Typ „opcjonalny”

W języku OCaml jest zdefiniowany następujący typ:

```
type 'a option = None | Some of 'a;;
```

Może on być wykorzystany w wielu sytuacjach zamiast wyjątków. Funkcja

```
# List.assoc;;
-: 'a -> ('a * 'b) list -> 'b = <fun>
```

mogłaby mieć typ `'a -> ('a * 'b) list -> 'b option` i w przypadku niepowodzenia zwracać wartość `None`, zamiast wyjątku `Not_found`. Ostatecznie sytuacja, kiedy nie znajdujemy czegoś w słowniku, nie jest taka wyjątkowa. Jeśli w słowniku znajdzie się para `(klucz, info)`, to zwracana byłaby wartość `Some info`, skąd łatwo wydobyć `info` używając dopasowania do wzorca.

Odpowiednikiem w Haskellu jest typ:

```
data Maybe a = Nothing | Just a
             deriving(Eq,Ord,Show,Read)
```

Typ „opcjonalny”

Typ opcjonalny można wykorzystywać jako bezpieczną alternatywę dla użycia wyjątków lub wartości null. Z tego powodu pojawia się on (pod różnymi nazwami) we wszystkich popularnych językach programowania.

Patrz:

Wikipedia

https://en.wikipedia.org/wiki/Option_type

Także:

https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

Rozszerzalne typy wariantowe

Od wersji 4.02 w OCamlu można używać rozszerzalnych typów wariantowych. Umożliwiają one dodawanie nowych konstruktorów do istniejących typów wariantowych, co było od początku możliwe dla wbudowanego typu exn.
Rozszerzalny typ wariantowy należy najpierw zdefiniować z wykorzystaniem dwóch kropek .. .
Nowe konstruktory są dodawane za pomocą += .

```
type color = ..
# type color += Black;;
type color += Black
# type color +=
      | White
      | Blue;;
type color += White | Blue
# let color2string = function
    | Black -> "black"
    | Blue ->  "blue"
    | White -> "white"
    | _ -> "unknown color";;
val color2string : color -> string = <fun>
```

W celu uniknięcia ostrzeżenia o niekompletności wzorców należy użyć wieloznacznika, do którego dopasują się później zdefiniowane konstruktory.

Słowniki

Słownikiem (ang. dictionary) nazywamy abstrakcyjny typ danych z operacjami wstawiania elementu do zbioru (insert), usuwania elementu ze zbioru (delete), oraz wyszukiwania elementu w zbiorze (lookup, search). Często przyjmuje się założenie, że klucze słownika należą do zbioru liniowo uporządkowanego. Czasem dla słownika używa się nazw lista lub tablica asocjacyjna. Efektywnymi strukturami służącymi do reprezentowania słowników są tablice z haszowaniem.

W przykładach z tego wykładu słownik był reprezentowany przez listę asocjacyjną. W językach funkcyjnych słownik można reprezentować za pomocą funkcji. **Nie jest to reprezentacja efektywna**, ale pokazuje siłę wyrazu funkcji.

Słownik jako funkcja

```
# exception Duplicated_key;;
exception Duplicated_key
# let emptyD key = None;;
val emptyD : 'a -> 'b option = <fun>
# let insertD dict (key,item) =
     if dict key <> None then raise Duplicated_key
     else function k -> if k=key then Some item else dict k;;
val insertD : ('a -> 'b option) -> 'a * 'b -> 'a -> 'b option = <fun>
# let lookupD dict key = dict key;;
val lookupD : ('a -> 'b) -> 'a -> 'b = <fun>
# let deleteD dict key = function k ->
                        if k = key then None
                        else dict k;;
val deleteD : ('a -> 'b option) -> 'a -> 'a -> 'b option = <fun>
# let updateD dict (key,item) =
      function k -> if k=key then Some item else dict k;;
val updateD : ('a -> 'b option) -> 'a * 'b -> 'a -> 'b option = <fun>
# let concatD dict1 dict2 =
      function key -> et res = dict1 key in if res = None then dict2 key else res;;
val concatD : ('a -> 'b option) -> ('a -> 'b option) -> 'a -> 'b option= <fun>
```

Słownik jako funkcja - użycie

```
# let (<==)  dict (key,item) = insertD  dict (key,item);;
val ( <== ) : ('a -> 'b option) -> 'a * 'b -> 'a -> 'b option = <fun>
# let d = emptyD;;
val d : 'a -> 'b option = <fun>
# lookupD d 5;;
- : 'a option = None
# let d = d <== (3,"three") <== (5,"five") <== (1,"one");;
val d : int -> string option = <fun>
# lookupD d 5;;
- : string option = Some "five"
# let d = deleteD d 5;;
val d : int -> string option = <fun>
# lookupD d 5;;
- : 'a option = None
# let d = concatD d (emptyD <== (0,"zero") <== (8,"eight"));;
val d : int -> string option = <fun>
# lookupD d 8;;
- : string option = Some "eight"
# let d = updateD d (8,"osiem");;
val d : int -> string option = <fun>
# lookupD d 8;;
- : string option = Some "osiem"
```

Drzewa wielokierunkowe

W *drzewach wielokierunkowych* (ang. multiway trees, finitely branching trees) węzeł może mieć dowolną liczbę potomków. *Las* (ang. forest) jest uporządkowanym zbiorem drzew. Typ dla drzew wielokierunkowych można zdefiniować, wykorzystując wzajemną rekursję:

```
# type 'a mtree = MNode of 'a * 'a forest
  and 'a forest = EmptyForest | Forest of 'a mtree * 'a forest;;
type 'a mtree = MNode of 'a * 'a forest
type 'a forest = EmptyForest | Forest of 'a mtree * 'a forest
```

Las drzew można też reprezentować za pomocą listy, co prowadzi do następującej definicji:

```
type 'a mtree_lst = MTree of 'a * ('a mtree_lst) list;;
```

Grafy

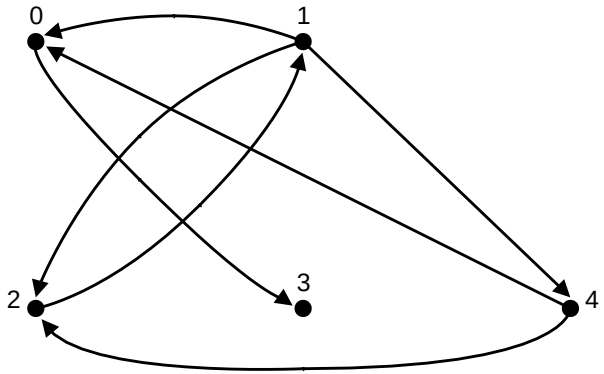
Do najczęściej stosowanych komputerowych reprezentacji grafów należą listy sąsiedztwa i macierze sąsiedztwa. Wykorzystując funkcje możemy zdefiniować następujący typ danych dla grafów:

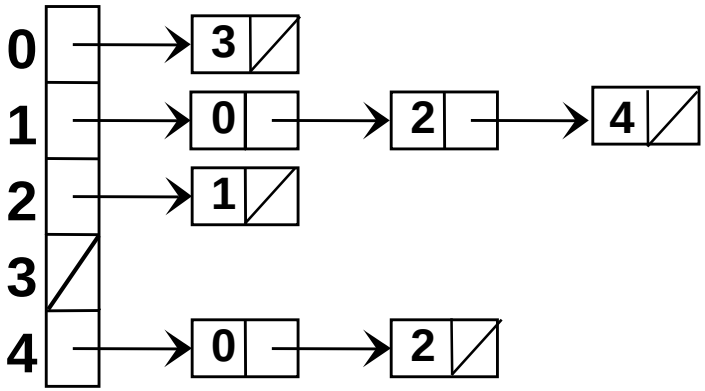
```
# type 'a graph = Graph of ('a -> 'a list);;
type 'a graph = Graph of ('a -> 'a list)
```

Funkcja (nazwijmy ją funkcją sąsiedztwa), będąca argumentem konstruktora wartości daje listę następników zadanego węzła. Jest to bardzo abstrakcyjne spojrzenie na grafy. Inne reprezentacje grafów (np. listy sąsiedztwa lub macierze sąsiedztwa) mogą być wykorzystane w implementacji funkcji sąsiedztwa. Użycie funkcji pozwala również na reprezentowanie grafów nieskończonych. Poniższą funkcję sąsiedztwa możemy wykorzystać do reprezentowania skierowanego (zorientowanego) grafu nieskończonego, w którym sąsiadami węzła o numerze *n* (*n*≥1) są wszystkie węzły, których numer jest podzielnikiem *n*, różnym od 1 i od *n*.

```
# let succ_mod n =
    if n<2 then []
    else let rec divisors(num,list) =
              if num<2 then list
              else if n mod num = 0 then divisors(num-1,num::list)
                   else divisors(num-1,list)
         in divisors(n/2,[]);;
val succ_mod : int -> int list = <fun>

# let g_mod = Graph succ_mod;;
val g_mod : int graph = Graph <fun>
```

Graf zorientowany i jego reprezentacje





Listy sąsiedztwa

Funkcja sąsiedztwa

```
# let g = Graph
  (function
      0 -> [3]
    | 1 -> [0;2;4]
    | 2 -> [1]
    | 3 -> []
    | 4 -> [0;2]
    | n -> failwith ("Graph g: node "^string_of_int n^" doesn't exist")
 );;
val g : int graph = Graph <fun>

# depthSearch g 4;;
- : int list = [4; 0; 3; 2; 1]
# breadthSearch g 4;;
- : int list = [4; 0; 2; 3; 1]
```

Przeszukiwanie grafu w głąb i wszerz

Poniższe funkcje obchodzą zadany graf w głąb i wszerz zaczynając od zadanego wierzchołka i zwracają zawartość zwizytowanych węzłów w postaci listy.

```
# let depthSearch (Graph succ) startNode =
    let rec search visited = function
        [] -> []
      | h::t -> if List.mem h visited then search visited t
                else h::search (h::visited) (succ h @ t)
    in search [] [startNode];;
val depthSearch : 'a graph -> 'a -> 'a list = <fun>

# let breadthSearch (Graph succ) startNode =
    let rec search visited = function
        [] -> []
      | h::t -> if List.mem h visited then search visited t
                else h::search (h::visited) (t @ succ h)
    in search [] [startNode];;
val breadthSearch : 'a graph -> 'a -> 'a list = <fun>
```

Obie funkcje są bardzo podobne. Pierwszy argument funkcji `search` jest listą, zawierającą zwizytowane wierzchołki, a drugi argument jest listą wierzchołków oczekujących na zwizytowanie. W funkcji `depthSearch` nowe wierzchołki są dołączane na początek tej listy (stos!), a w funkcji `breadthSearch` na koniec (kolejka!).

Warianty polimorficzne - OCaml

W języku OCaml można również definiować polimorficzne typy wariantowe (ang. polymorphic variant types). Pozwalają one złagodzić ograniczenia, wynikające z generatywności zwykłych typów wariantowych.

Zadania kontrolne

Polimorficzne drzewa binarne są zdefiniowane następująco:

```
type 'a binTree = Node of 'a binTree * 'a * 'a binTree | Empty;;
```

1. Poniższe funkcje tworzą listy etykiet drzewa, wykorzystując dwie standardowe techniki obejścia drzew.

```
let rec inorder = function
          Node(l,v,r) -> (inorder l) @ [v] @ (inorder r)
        | Empty -> [];;
let rec postorder = function
          Node(l,v,r) -> (postorder l) @ (postorder r) @ [v]
        | Empty -> [];;
```

Mimo dużej czytelności nie są one zbyt efektywne. Dla drzew ukośnych ich złożoność może być kwadratowa ze względu na użycie @ . Napisz efektywniejsze wersje, wykorzystujące tylko jedną operację :: dla każdego węzła.

Zadania kontrolne

2. Zdefiniuj funkcję

   `mapBinTree :('a -> 'b) -> 'a binTree -> 'b binTree`,

   aplikującą daną funkcję do obiektów we wszystkich węzłach drzewa.

3. Wykorzystując `mapBinTree`, zdefiniuj następujące funkcje:

   `f :(int list) binTree -> int binTree`, zastępującą w każdym węźle listę liczb całkowitych ich sumą;

   `g :int binTree -> (int list) binTree`, zastępującą w każdym węźle liczbę całkowitą listą jej cyfr.

4. W *regularnym drzewie binarnym* każdy z węzłów jest bądź liściem, bądź ma stopień dwa (patrz Cormen i in. §5.5.3). Zdefiniujmy typ:

   ```
   type 'a regBT = RLeaf | RNode of 'a regBT * 'a * 'a regBT;;
   ```

   Długość ścieżki wewnętrznej regularnego drzewa binarnego jest sumą, po wszystkich węzłach wewnętrznych drzewa, głębokości każdego węzła. Długość ścieżki zewnętrznej jest sumą, po wszystkich liściach drzewa, głębokości każdego liścia.

   Napisz dwie możliwie efektywne funkcje, obliczające odpowiednio długość ścieżki wewnętrznej i zewnętrznej zadanego regularnego drzewa binarnego.

Zadania kontrolne

5. Napisz możliwie efektywne funkcje obejścia wszerz i prefiksowego obejścia w głąb drzew wielokierunkowych dla obu definicji podanych na wykładzie.

6. Wzorując się na funkcji przeszukiwania grafu z wykładu napisz funkcję o typie `'a graph-> 'a ->int -> 'a option`, która bierze graf (można założyć, że graf jest spójny), wierzchołek startowy, liczbę naturalną *n* i znajduje wierzchołek, posiadający co najmniej *n* sąsiadów i osiągalny z wierzchołka startowego.