ZESTAW ZADAŃ TRENINGOWYCH - SERIA 4

📋 ZADANIA Z FOLD_LIST (10 zadań)

1. `sliding_window : int -> 'a list -> 'a list list`

Stwórz okno przesuwne rozmiaru n używając folda. `sliding_window 3 [1;2;3;4;5]` → `[[1;2;3];[2;3;4];[3;4;5]]`

2. `interleave : 'a list -> 'a list -> 'a list`

Przeplataj dwie listy naprzemiennie używając folda. `interleave [1;3;5] [2;4;6]` → `[1;2;3;4;5;6]`

3. `chunk_by_size : int -> 'a list -> 'a list list`

Podziel listę na kawałki o zadanym rozmiarze używając folda. `chunk_by_size 3 [1;2;3;4;5;6;7]` → `[[1;2;3];[4;5;6];[7]]`

4. `fold_with_index : ('a -> int -> 'b -> 'b) -> 'b -> 'a list -> 'b`

Zaimplementuj fold który przekazuje też indeks elementu.

5. `scan_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a list`

Jak fold_left ale zwraca listę wszystkich pośrednich wyników. `scan_left (+) 0 [1;2;3]` → `[0;1;3;6]`

6. `transpose : 'a list list -> 'a list list`

Transponuj listę list używając folda. `transpose [[1;2;3];[4;5;6]]` → `[[1;4];[2;5];[3;6]]`

7. `run_length_encode : 'a list -> (int * 'a) list`

Kodowanie długości serii używając folda. `[1;1;2;2;2;3]` → `[(2,1);(3,2);(1,3)]`

8. `cartesian_product : 'a list -> 'b list -> ('a * 'b) list`

Iloczyn kartezjański używając folda.

9. `permutations : 'a list -> 'a list list`

Wszystkie permutacje listy używając folda (dla krótkich list).

10. `longest_increasing_subsequence : int list -> int list`

Najdłuższy rosnący podciąg używając folda.

🌳 ZADANIA Z FOLD_TREE (5 zadań)

Dla typu: `type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree`

1. `tree_paths_sum : int tree -> int list`

Dla każdej ścieżki od korzenia do liścia zwróć sumę wartości na ścieżce.

2. `tree_serialize : 'a tree -> 'a list`

Serializuj drzewo do listy (pre-order) tak żeby można było je odtworzyć.

3. `tree_closest_leaf : 'a -> 'a tree -> int option`

Znajdź najkrótszą odległość od wartości do najbliższego liścia.

4. `tree_fold_breadth_first : ('b -> 'a -> 'b) -> 'b -> 'a tree -> 'b`

Fold w kolejności poziomowej (breadth-first).

5. `tree_maximum_path_sum : int tree -> int`

Maksymalna suma na dowolnej ścieżce w drzewie (nie musi iść przez korzeń).

🧮 ZADANIA Z INDUKCJI (5 zadań)

DOWODY (3 zadania):

1. Udowodnij że dla każdej listy `xs` i funkcji `f`, `g`:

`map f (map g xs) = map (fun x -> f (g x)) xs`

2. Udowodnij że dla każdego drzewa `t` i predykatu `p`:

`tree_all p t ∧ tree_any (not ∘ p) t = false`

3. Udowodnij że dla każdych funkcji `f : 'a -> 'b` i `g : 'b -> 'c`:

Jeśli `f` i `g` są surjekcjami, to `g ∘ f` też jest surjekcją.

SFORMUŁOWANIE ZASAD (2 zadania):

4. Sformułuj zasadę indukcji dla typu:

```ocaml
type 'a rose_tree =
  | RLeaf of 'a
  | RNode of 'a * 'a rose_tree list
```

5. Sformułuj zasadę indukcji strukturalnej dla typu:

```ocaml
type lambda_term =
  | Var of string
  | Abs of string * lambda_term
  | App of lambda_term * lambda_term
```

🔤 ZADANIA Z TYPOWANIA (10 zadań - podaj typ)

1. `fun f -> fun g -> fun h -> fun x -> h (f x) (g x)`

2. `fun lst -> List.fold_left (fun acc x -> acc @ [x]) [] lst`

3. `fun f -> fun opt -> match opt with | Some x -> Some (f x) | None -> None`

4. `fun pred -> fun lst -> List.fold_left (fun (yes, no) x -> if pred x then (x::yes, no) else (yes, x::no)) ([], []) lst`

5. `fun f -> fun g -> fun lst -> List.map (fun x -> (f x, g x)) lst`

6. `fun default -> fun lst -> match lst with | [] -> default | x :: _ -> x`

7. `fun cmp -> fun lst -> List.fold_left (fun acc x -> match acc with | None -> Some x | Some y -> Some (if cmp x y < 0 then x else y)) None lst`

8. `fun f -> fun lst -> List.fold_right (fun x acc -> if f x then x :: acc else acc) lst []`

9. `fun f -> List.fold_left (fun acc x -> f acc x)`

10. `fun lst1 -> fun lst2 -> List.fold_left2 (fun acc x y -> (x, y) :: acc) [] lst1 lst2`

🎯 ZADANIA ODWROTNE - TYP → FUNKCJA (10 zadań)

Napisz funkcję o podanym typie:

1. `('a -> 'b -> 'c) -> ('d -> 'a) -> ('d -> 'b) -> ('d -> 'c)`

2. `('a -> 'b option) -> 'a list -> 'b list`

3. `('a -> 'b -> bool) -> 'a list -> 'b list -> ('a * 'b) list`

4. `int -> ('a -> 'a) -> 'a -> 'a`

5. `('a -> bool) -> ('a -> bool) -> ('a -> bool)`

6. `'a list -> ('a -> 'b) -> ('a -> 'c) -> ('b * 'c) list`

7. `('a option -> 'b) -> 'a list -> 'b list`

8. `(int -> 'a -> 'a) -> int -> 'a -> 'a`

9. `('a -> 'b -> 'c) -> 'a option -> 'b option -> 'c option`

10. `('a -> bool) -> 'a list -> ('a list * 'a list)`

🌲 ZADANIA Z OPERACJI NA DRZEWACH (5 zadań)

Dla typu: `type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree`

1. `tree_insert_sorted : 'a -> 'a tree -> 'a tree`

Wstaw element do BST zachowując porządek.

2. `tree_delete : 'a -> 'a tree -> 'a tree`

Usuń element z BST zachowując porządek.

3. `tree_split : 'a -> 'a tree -> 'a tree * bool * 'a tree`

Podziel BST na części: (mniejsze, czy_był_element, większe).

4. `tree_merge : 'a tree -> 'a tree -> 'a tree`

Połącz dwa BST w jeden (zakładając że wszystkie elementy pierwszego < drugiego).

5. `tree_rebalance : 'a tree -> 'a tree`

Zbalansuj BST używając algorytmu DSW lub podobnego.

📝 ZADANIA Z GRAMATYK BEZKONTEKSTOWYCH (2 zadania)

1. Napisz gramatykę bezkontekstową dla języka:

```
L = {a^i b^j c^k d^l | i+k = j+l, i,j,k,l ≥ 0}
```

(suma a i c równa sumie b i d)

2. Napisz gramatykę dla prostego języka programowania z:

- zmiennymi (identyfikatory)
- przypisaniem x := expr
- if-then-else
- while loops

- wyrażeniami arytmetycznymi z nawiasami
- sekwencją instrukcji

⚙️ ZADANIA Z KOMPILATORÓW (2 zadania)

1. Kompilator z obsługą zmiennych lokalnych:

```
ocaml
```

```
type expr =
  | Int of int
  | Var of string
  | Add of expr * expr
  | Let of string * expr * expr
  | LetRec of string * string * expr * expr

type instr =
  | Push of int
  | Load of int
  | Store of int
  | AddOp
  | EnterScope of int    (* wejście do scope z n zmiennymi *)
  | ExitScope of int     (* wyjście ze scope *)
  | MakeRecClosure of instr list
  | Apply
```

Napisz kompilator który obsługuje zagnieżdżone scope'y.

2. Analiza live variables:

Napisz `live_analysis : instr list -> int list list` która dla każdej instrukcji zwraca listę zmiennych "żywych" w tym punkcie programu.

🔧 ZADANIA Z INTERPRETERÓW (2 zadania)

1. Interpreter z modułami:

Napisz kompletny interpreter dla języka z:

```ocaml
type expr =
  | Int of int | Bool of bool | Var of string
  | Add of expr * expr | Let of string * expr * expr
  | Module of string * (string * expr) list * expr   (* module M = {x=5; f=fun...} in expr *)
  | Access of string * string                         (* M.x *)
  | Open of string * expr                             (* open M in expr *)

type value =
  | VInt of int | VBool of bool
  | VModule of (string * value) list
  | VClosure of string * expr * env
```

2. Interpreter z wyjątkami i finally:

Napisz kompletny interpreter dla języka z:

```ocaml
type expr =
  | Int of int | Bool of bool | Var of string
  | Add of expr * expr | Let of string * expr * expr
  | Throw of expr
  | TryCatch of expr * string * expr
  | TryFinally of expr * expr                         (* try e1 finally e2 *)
  | TryCatchFinally of expr * string * expr * expr    (* try e1 catch x -> e2 finally e3 *)

type value =
  | VInt of int | VBool of bool | VUnit
```

Pamiętaj: finally wykonuje się ZAWSZE, nawet gdy jest wyjątek!