Zestaw zadań - Monady, Interpretery, Kompilatory, Type Checkery

---

Zadanie 1. Monady (40 pkt - 2 pkt za podpunkt)

Dla każdego z poniższych typów zaimplementuj funkcje `return` i `bind`:

a) Option/Maybe

```ocaml
type 'a option = None | Some of 'a
let return x = _____________
let bind m f = _____________
```

b) List

```ocaml
type 'a list (* wbudowany typ *)
let return x = _____________
let bind m f = _____________
```

c) Result/Either

```ocaml
type ('a, 'e) result = Ok of 'a | Error of 'e
let return x = _____________
let bind m f = _____________
```

d) Writer (z konkatenacją stringów)

```ocaml
type 'a writer = 'a * string
let return x = _____________
let bind m f = _____________
```

e) State

ocaml

```ocaml
type ('s, 'a) state = 's -> ('a * 's)
let return x = _____________
let bind m f = _____________
```

f) Reader/Environment

ocaml

```ocaml
type ('e, 'a) reader = 'e -> 'a
let return x = _____________
let bind m f = _____________
```

g) IO (symulowana)

ocaml

```ocaml
type 'a io = unit -> 'a
let return x = _____________
let bind m f = _____________
```

h) Continuation

ocaml

```ocaml
type ('a, 'r) cont = ('a -> 'r) -> 'r
let return x = _____________
let bind m f = _____________
```

i) Pair/Product (z monoidu)

ocaml

```ocaml
type 'a pair = 'a * 'a
let return x = _____________
let bind m f = _____________
```

j) Tree (drzewo binarne)

ocaml

```ocaml
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
let return x = _____________
let bind m f = _____________
```

k) Lazy

```ocaml
type 'a lazy_m = unit -> 'a
let return x = _____
let bind m f = _____
```

l) Identity

```ocaml
type 'a identity = 'a
let return x = _____
let bind m f = _____
```

m) Writer (z listą intów)

```ocaml
type 'a writer_int = 'a * int list
let return x = _____
let bind m f = _____
```

n) Validation (gromadzi wszystkie błędy)

```ocaml
type ('a, 'e) validation = Valid of 'a | Invalid of 'e list
let return x = _____
let bind m f = _____
```

o) Parser (uproszczony)

```ocaml
type 'a parser = string -> ('a * string) option
let return x = _____
let bind m f = _____
```

p) Future/Promise

```ocaml
type 'a future = Pending | Resolved of 'a | Failed of string
let return x = _____
let bind m f = _____
```

q) NonEmpty List

```ocaml
type 'a nelist = 'a * 'a list
let return x = _____________
let bind m f = _____________
```

r) Rose Tree

```ocaml
type 'a rose = Rose of 'a * 'a rose list
let return x = _____________
let bind m f = _____________
```

s) Zipper (dla list)

```ocaml
type 'a zipper = 'a list * 'a * 'a list (* left, focus, right *)
let return x = _____________
let bind m f = _____________
```

t) Distribution/Probability

```ocaml
type 'a dist = ('a * float) list (* wartość * prawdopodobieństwo *)
let return x = _____________
let bind m f = _____________
```

Zadanie 2. Interpreter (20 pkt)

Zaimplementuj interpreter dla poniższego języka:

```ocaml
type expr =
  | Int of int
  | Bool of bool
  | Var of string
  | Add of expr * expr
  | Sub of expr * expr
  | Mul of expr * expr
  | Eq of expr * expr    (* równość *)
  | Lt of expr * expr     (* mniejsze *)
  | If of expr * expr * expr
  | Fun of string * expr
  | App of expr * expr
  | Let of string * expr * expr
  | LetRec of string * expr * expr

type value =
  | VInt of int
  | VBool of bool
  | VClosure of string * expr * env
  | VRecClosure of string * string * expr * env  (* dla funkcji rekurencyjnych *)
and env = (string * value) list

exception Runtime_error of string

let rec eval (env : env) (e : expr) : value = _______________
```

Zadanie 3. Kompilator (20 pkt)

Zaimplementuj kompilator dla prostego języka do maszyny stosowej:

```ocaml
type expr =
  | Const of int
  | Var of string
  | Add of expr * expr
  | Sub of expr * expr
  | Mul of expr * expr
  | Let of string * expr * expr
  | If of expr * expr * expr

type instr =
  | IPushConst of int
  | IPushVar of string
  | IAdd
  | ISub
  | IMul
  | IStore of string    (* zdejmij wartość ze stosu i zapisz w zmiennej *)
  | IPop of string      (* usuń zmienną ze środowiska *)
  | IJumpIfFalse of int (* relatywny skok jeśli 0 na stosie *)
  | IJump of int        (* relatywny skok *)

let rec compile (e : expr) : instr list = _______________
```

Wskazówki:

- `If` kompiluje się do: [kod_warunku, IJumpIfFalse n, kod_then, IJump m, kod_else]
- `Let` kompiluje się do: [kod_expr, IStore x, kod_body, IPop x]

Zadanie 4. Type Checker (20 pkt)

Zaimplementuj type checker dla prostego języka funkcyjnego:

ocaml

```ocaml
type typ =
  | TInt
  | TBool
  | TFun of typ * typ
  | TPair of typ * typ

type expr =
  | EInt of int
  | EBool of bool
  | EVar of string
  | EAdd of expr * expr
  | EMul of expr * expr
  | EEq of expr * expr      (* porównanie, zwraca bool *)
  | EIf of expr * expr * expr
  | EFun of string * typ * expr  (* lambda z anotacją typu *)
  | EApp of expr * expr
  | ELet of string * expr * expr
  | EPair of expr * expr
  | EFst of expr            (* pierwszy element pary *)
  | ESnd of expr            (* drugi element pary *)

type tenv = (string * typ) list

exception Type_error of string

let rec type_check (env : tenv) (e : expr) : typ = _______________
```

Wskazówki:

- `EAdd` i `EMul` wymagają dwóch `TInt` i zwracają `TInt`
- `EEq` wymaga dwóch argumentów tego samego typu i zwraca `TBool`
- `EIf` wymaga `TBool` jako warunku, obie gałęzie muszą mieć ten sam typ
- `EFst` i `ESnd` wymagają typu `TPair`