Oceny (próg): 25 pkt = 3.0, 30 pkt = 3.5, 35 pkt = 4.0, 40 pkt = 4.5, 45 pkt = 5.0

Zadanie 1. (3 pkt)

W poniższych wyrażeniach podkreśl wolne wystąpienia zmiennych. Dla każdego związanego wystąpienia zmiennej, narysuj strzałkę od tego wystąpienia do wystąpienia wiążącego je.

```ocaml
let f = fun x -> x + y in
let g h = h (f z) in
g f
```

```ocaml
fun x ->
  let y = fun z -> x + z in
  let x = 5 in
  y x
```

Zadanie 2. (4 pkt)

Dla poniższych wyrażeń w języku OCaml podaj ich (najogólniejszy) typ, lub napisz „BRAK TYPU", gdy wyrażenie nie posiada typu.

- `fun x y -> if x = y then x else y`: _____
- `fun f x -> let y = f x in (y, y)`: _____
- `fun x -> (x 1, x true, x "hello")`: _____
- `let rec fix f x = f (fix f) x in fix`: _____
- `fun (x :: xs) -> x + List.length xs`: _____

Zadanie 3. (4 pkt)

Napisz DOWOLNĄ funkcję realizującą każdy z poniższych typów (wszystkie da się zrealizować!):

a) `'a -> 'b -> 'b`

b) `('a -> 'b) -> ('b -> 'c) -> 'a -> 'c`

c) `'a list -> ('a -> bool) -> 'a list`

d) `('a * 'b) -> ('a -> 'c) -> ('b -> 'd) -> ('c * 'd)`

Zadanie 4. (5 pkt)

Zaimplementuj funkcje używając `List.fold_left` lub `List.fold_right`:

- `find_index : ('a -> bool) -> 'a list -> int option` - zwraca indeks pierwszego elementu spełniającego predykat

```ocaml
let find_index p xs = _________________
```

- `unzip : ('a * 'b) list -> 'a list * 'b list` - rozdziela listę par na parę list

```ocaml
let unzip pairs = _________________
```

- `count_if : ('a -> bool) -> 'a list -> int` - zlicza elementy spełniające predykat

```ocaml
let count_if p xs = _________________
```

Zadanie 5. (6 pkt)

Zaimplementuj `return` i `bind` dla następujących monad:

a) **Result monad** - obliczenia które mogą się nie udać

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

b) **Reader monad** - obliczenia z dostępem do współdzielonego środowiska

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

c) **Continuation monad** - obliczenia z kontynuacjami

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

Zadanie 6. (7 pkt)

Rozważ typ dla list niepustych:

```ocaml
type 'a nelist = Cons of 'a * 'a nelist | Single of 'a
```

a) (2 pkt) Sformułuj zasadę indukcji dla typu `'a nelist`.

b) (5 pkt) Zdefiniuj funkcje:

```ocaml
let rec append (xs : 'a nelist) (ys : 'a nelist) : 'a nelist = _____________

let rec reverse (xs : 'a nelist) : 'a nelist = _____________
```

Następnie udowodnij indukcyjnie, że dla dowolnych list niepustych `xs` i `ys`:

```
reverse (append xs ys) = append (reverse ys) (reverse xs)
```

Zadanie 7. (7 pkt)

Zaimplementuj prosty type checker dla języka z liczbami, boolami i funkcjami:

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

type expr =
  | EInt of int
  | EBool of bool
  | EVar of string
  | EAdd of expr * expr
  | EEqual of expr * expr  (* porównanie == *)
  | EIf of expr * expr * expr
  | EFun of string * typ * expr  (* funkcja z anotacją typu *)
  | EApp of expr * expr
  | ELet of string * expr * expr

type tenv = (string * typ) list

exception Type_error of string

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

Zadanie 8. (7 pkt)

Zaimplementuj interpreter dla prostego języka imperatywnego:

```ocaml
type expr =
  | Num of int
  | Var of string
  | Add of expr * expr
  | Assign of string * expr  (* x := e *)
  | Seq of expr * expr       (* e1; e2 *)
  | While of expr * expr     (* while e1 do e2 *)
  | Print of expr           (* print e, zwraca wartość e *)

type store = (string * int) list

(* eval zwraca parę: (wartość, nowy stan) *)
let rec eval (s : store) (e : expr) : (int * store) = _______________
```

Wskazówka:

- `Assign` zwraca przypisaną wartość
- `While` zwraca 0
- `Print` możesz zaimplementować używając `Printf.printf "%d\n"`

Zadanie 9. (7 pkt)

Zaimplementuj kompilator i maszynę wirtualną:

a) (4 pkt) Kompilator dla wyrażeń:

```ocaml
type expr =
  | Const of int
  | Var of int          (* indeks zmiennej *)
  | BinOp of op * expr * expr
  | If of expr * expr * expr
and op = Add | Sub | Mul | Lt

type instr =
  | IPush of int
  | IVar of int       (* ładuje zmienną o danym indeksie *)
  | IOp of op
  | IJump of int      (* skok bezwarunkowy *)
  | IJumpIfFalse of int  (* skok jeśli false/0 na stosie *)

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

b) (3 pkt) Maszyna wirtualna:

```ocaml
type value = VInt of int | VBool of bool
type state = {
  code : instr array;
  pc : int;           (* program counter *)
  stack : value list;
  vars : value array; (* zmienne lokalne *)
}

let step (s : state) : state = _______________
```