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 rec f x =
  let rec g y =
    if x > y then f (g x)
    else h (x + y)
  in g (f z)
```

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

Zadanie 2. (4 pkt)

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

- `fun f g x -> f (g x) (g x)`: _____
- `fun f -> (f true, f 0)`: _____
- `let rec f x y = f y x in f`: _____
- `fun (f, g) -> fun x -> (f (g x), g (f x))`: _____
- `fun x -> let rec f y = if y = 0 then x else f (f (y-1)) in f`: _____

Zadanie 3. (4 pkt)

Napisz DOWOLNĄ funkcję realizującą każdy z poniższych typów:

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

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

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

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

Zadanie 4. (5 pkt)

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

- `split_at : int -> 'a list -> 'a list * 'a list` - dzieli listę na pozycji n

```ocaml
let split_at n xs = _______________
```

- `group_by : ('a -> 'a -> bool) -> 'a list -> 'a list list` - grupuje sąsiednie elementy spełniające relację Przykład: `group_by (=) [1;1;2;3;3;1;1]` = `[[1;1];[2];[3;3];[1;1]]`

```ocaml
let group_by eq xs = _______________
```

- `scan_left : ('b -> 'a -> 'b) -> 'b -> 'a list -> 'b list` - zwraca listę kolejnych akumulatorów Przykład: `scan_left (+) 0 [1;2;3]` = `[0;1;3;6]`

```ocaml
let scan_left f init xs = _______________
```

Zadanie 5. (6 pkt)

Rozważ następujące typy monadyczne. Dla każdego zaimplementuj `return` i `bind`:

a) **Writer monad** - obliczenia z logowaniem

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

b) **State monad** - obliczenia ze stanem

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

c) **List monad** - obliczenia niedeterministyczne

```ocaml
type 'a list_m = 'a list
let return x = _______________
let bind m f = _______________
```

Zadanie 6. (6 pkt)

Zdefiniuj typ dla drzew binarnych z wartościami tylko w liściach:

```ocaml
type 'a leaf_tree = Leaf of 'a | Branch of 'a leaf_tree * 'a leaf_tree
```

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

b) (4 pkt) Udowodnij indukcyjnie, że dla dowolnej funkcji `f : 'a -> 'b` i dowolnego drzewa `t : 'a leaf_tree`:

```
count_leaves (map_tree f t) = count_leaves t
```

gdzie:

```ocaml
let rec map_tree f = function
  | Leaf x -> Leaf (f x)
  | Branch (l, r) -> Branch (map_tree f l, map_tree f r)

let rec count_leaves = function
  | Leaf _ -> 1
  | Branch (l, r) -> count_leaves l + count_leaves r
```

Zadanie 7. (8 pkt)

Zaimplementuj interpreter dla języka z funkcjami, aplikacjami i operacjami arytmetycznymi:

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

type value =
  | VInt of int
  | VClosure of string * expr * env
and env = (string * value) list

exception Runtime_error of string

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

Zadanie 8. (8 pkt)

Zaimplementuj type checker dla języka z let-polimorfizmem:

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

type scheme = Forall of string list * typ

type expr =
  | EInt of int
  | EBool of bool
  | EVar of string
  | EAdd of expr * expr
  | EIf of expr * expr * expr
  | EFun of string * expr
  | EApp of expr * expr
  | ELet of string * expr * expr
  | EPair of expr * expr
  | EFst of expr
  | ESnd of expr

type type_env = (string * scheme) list

let rec type_infer (env : type_env) (e : expr) : typ = _______________
```

Wskazówka: możesz założyć, że masz dostęp do funkcji:

- `fresh_type_var () : string` - generuje świeżą zmienną typową
- `instantiate : scheme -> typ` - instancjuje schemat typowy
- `generalize : type_env -> typ -> scheme` - generalizuje typ do schematu

Zadanie 9. (6 pkt)

Zaimplementuj kompilator prostych wyrażeń arytmetycznych do maszyny stosowej:

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

type instr =
  | IPush of int
  | ILoad of string
  | IStore of string
  | IAdd
  | IMul
  | IPop

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

Wskazówka: `Let(x, e1, e2)` powinno:

1. Obliczyć `e1` (wynik na stosie)
2. Zapisać wynik do zmiennej `x` używając `IStore`
3. Obliczyć `e2`
4. Usunąć zmienną lokalną ze stosu używając `IPop`