Oceny (próg): 17 pkt = 3.0, 21 pkt = 3.5, 24 pkt = 4.0, 28 pkt = 4.5, 31 pkt = 5.0

Zadanie 1. (2 pkt)

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

```ocaml
let x = 5 in
let f y = x + y + z in
f x
```

```ocaml
fun f ->
  let x = f y in
  fun y -> f x y
```

Zadanie 2. (3 pkt)

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

- `fun x -> x :: [x; x]`: _____
- `fun f x -> if f x then x else x + 1`: _____
- `fun x y -> x y y`: _____
- `fun f -> f f`: _____

Zadanie 3. (3 pkt)

Napisz funkcję realizującą każdy z poniższych typów. Funkcja powinna być poprawna i możliwie prosta.

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

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

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

Zadanie 4. (3 pkt)

Zaimplementuj poniższe funkcje używając **wyłącznie** `List.fold_left` lub `List.fold_right`:

- `last : 'a list -> 'a option` - zwraca ostatni element listy (lub `None` dla pustej listy)

```ocaml
let last xs = _________________
```

- `take_while : ('a -> bool) -> 'a list -> 'a list` - zwraca początkowy fragment listy, którego elementy spełniają predykat

```ocaml
let take_while p xs = _________________
```

Zadanie 5. (4 pkt)

Uzupełnij poniższą implementację drzew AVL (tylko wstawianie, bez usuwania):

ocaml

```ocaml
type 'a avl =
  | Empty
  | Node of 'a avl * 'a * 'a avl * int  (* lewe, wartość, prawe, wysokość *)

let height = function
  | Empty -> 0
  | Node (_, _, _, h) -> h

let make_node l v r =
  Node (l, v, r, 1 + max (height l) (height r))

let balance_factor = function
  | Empty -> 0
  | Node (l, _, r, _) -> height l - height r

let rotate_left = function
  | Node (l, v, Node (rl, rv, rr, _), _) -> _______________
  | t -> t

let rotate_right = function
  | Node (Node (ll, lv, lr, _), v, r, _) -> _______________
  | t -> t

let balance t =
  match balance_factor t with
  | 2 -> (* lewe poddrzewo za wysokie *)
      begin match t with
      | Node (l, _, _, _) when balance_factor l < 0 ->
          (* rotacja LR *) _______________
      | _ -> rotate_right t
      end
  | -2 -> (* prawe poddrzewo za wysokie *)
      begin match t with
      | Node (_, _, r, _) when balance_factor r > 0 ->
          (* rotacja RL *) _______________
      | _ -> rotate_left t
      end
  | _ -> t
```

Zadanie 6. (4 pkt)

Rozważ poniższą gramatykę języka prostych wyrażeń arytmetycznych:

```
E ::= n | E + T | T
T ::= n | T * n | n
```

gdzie `n` oznacza liczbę całkowitą.

a) (1 pkt) Podaj drzewo wyprowadzenia dla wyrażenia `2 + 3 * 4`

b) (3 pkt) Przekształć tę gramatykę do postaci, która może być bezpośrednio użyta w generatorze parserów (usuń lewostronną rekursję).

Zadanie 7. (3 pkt)

Dla typu:

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

a) (1 pkt) Napisz funkcje `return` i `bind` realizujące monadę:

```ocaml
let return x = _______________
let bind m f = _______________
```

b) (2 pkt) Używając tylko `bind` i `return`, napisz funkcję:

```ocaml
let add_options : int option -> int option -> int option = _______________
```

która dodaje dwie liczby w opcjach (zwraca `None` jeśli którakolwiek jest `None`).

Zadanie 8. (5 pkt)

Dany jest typ reprezentujący proste wyrażenia:

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

oraz typ środowiska:

```ocaml
type env = (string * int) list
```

Uzupełnij interpreter:

```ocaml
let rec eval (env : env) (e : expr) : int =
  match e with
  | Const n -> _______________
  | Add (e1, e2) -> _______________
  | Mul (e1, e2) -> _______________
  | Var x -> _______________
  | Let (x, e1, e2) ->
      let v1 = eval env e1 in
      _______________
```

Zadanie 9. (4 pkt)

Uzupełnij implementację sprawdzania typów dla prostego języka z liczbami i funkcjami:

ocaml

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

type expr =
  | EInt of int
  | EBool of bool
  | EVar of string
  | EIf of expr * expr * expr
  | EFun of string * typ * expr
  | EApp of expr * expr
  | EAdd of expr * expr

type tenv = (string * typ) list

let rec type_check (tenv : tenv) (e : expr) : typ =
  match e with
  | EInt _ -> _______________
  | EBool _ -> _______________
  | EVar x -> List.assoc x tenv
  | EIf (e1, e2, e3) ->
      if type_check tenv e1 = TBool then
        let t2 = type_check tenv e2 in
        let t3 = type_check tenv e3 in
        if t2 = t3 then t2 else failwith "if branches have different types"
      else failwith "if condition must be bool"
  | EFun (x, t, body) -> _______________
  | EApp (e1, e2) -> _______________
  | EAdd (e1, e2) ->
      if type_check tenv e1 = TInt && type_check tenv e2 = TInt then
        _______________
      else failwith "add requires two ints"
```