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

Zadanie 1. (3 pkt)

Dla każdego z poniższych typów podaj przykład funkcji, która go realizuje. Funkcja powinna być możliwie prosta, ale poprawna.

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

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

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

Zadanie 2. (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 rec f x y =
  let g = fun z -> x + z in
  if y > 0 then g (f x (y-1))
  else h x
```

```ocaml
fun x ->
  let rec loop y =
    if y = 0 then x
    else loop (y - 1)
  in loop z
```

Zadanie 3. (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 f g x -> f (g x) = g (f x)`: _____
- `fun x -> if x then x else not x`: _____
- `let rec f x = f in f`: _____
- `fun (x, y) -> (y, x, x = y)`: _____

- `fun f -> let x = f 0 in let y = f true in x + y`: _____

Zadanie 4. (6 pkt)

Rozważmy typ list z wartownikiem na końcu:

```ocaml
type 'a slist =
  | SNil of 'a
  | SCons of 'a * 'a slist
```

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

b) (4 pkt) Udowodnij indukcyjnie, że dla dowolnych funkcji `f : 'a -> 'b` i `g : 'b -> 'b -> 'b` oraz dowolnej listy `lst : 'a slist`, zachodzi:

```
smap f (sfold g lst) = sfold g (smap f lst)
```

gdzie:

```ocaml
let rec smap f lst = match lst with
  | SNil x -> SNil (f x)
  | SCons (h, t) -> SCons (f h, smap f t)

let rec sfold g lst = match lst with
  | SNil x -> x
  | SCons (h, t) -> g h (sfold g t)
```

Zadanie 5. (5 pkt)

Rozważmy typ dla monad z dodatkowym efektem zliczania:

```ocaml
type 'a counted = int * 'a
```

a) (3 pkt) Zaimplementuj funkcje `return` i `bind` dla tej monady, gdzie:

- `return` tworzy wartość monadyczną z zerowym licznikiem
- `bind` łączy obliczenia, sumując liczniki

```ocaml
let return (x : 'a) : 'a counted = _____________

let bind (m : 'a counted) (f : 'a -> 'b counted) : 'b counted = _____________
```

b) (2 pkt) Używając powyższych funkcji, zaimplementuj funkcję `count_calls`, która zwiększa licznik o 1:

```ocaml
let count_calls () : unit counted = _____________
```

oraz funkcję `sequence`, która wykonuje listę obliczeń monadycznych w kolejności:

```ocaml
let rec sequence (ms : ('a counted) list) : ('a list) counted = _____________
```

Zadanie 6. (3 pkt)

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

- Funkcja `partition p xs` dzieli listę na parę list `(satisfy, dont_satisfy)`, gdzie pierwsza zawiera elementy spełniające predykat `p`, a druga pozostałe. Zachowaj kolejność elementów.

```ocaml
let partition p xs = _____________
```

- Funkcja `group_consecutive xs` grupuje kolejne identyczne elementy w podlisty. Przykład: `group_consecutive [1;1;2;3;3;3;1]` = `[[1;1];[2];[3;3;3];[1]]`

```ocaml
let group_consecutive xs = _____________
```

Zadanie 7. (4 pkt)

Rozważ poniższą gramatykę dla wyrażeń boolowskich:

```
B ::= true | false | B and B | B or B | not B | (B)
```

a) (1 pkt) Czy gramatyka jest jednoznaczna? Jeśli nie, podaj przykład niejednoznaczności.

b) (3 pkt) Zaproponuj jednoznaczną gramatykę dla tego samego języka, uwzględniając standardowe priorytety operatorów (not > and > or) oraz łączność lewostronną dla operatorów binarnych.

Zadanie 8. (5 pkt)

Rozważmy język Mini-ML z konstruktorem `match`:

```ocaml
type expr =
  | Int of int
  | Bool of bool
  | Var of string
  | If of expr * expr * expr
  | Let of string * expr * expr
  | Fun of string * expr
  | App of expr * expr
  | Pair of expr * expr
  | Match of expr * string * string * expr  (* match e with (x,y) -> body *)
```

Zaimplementuj funkcję sprawdzającą, czy wszystkie zmienne w wyrażeniu są związane:

```ocaml
let rec all_vars_bound (bound_vars : string list) (e : expr) : bool = _____________
```

Zadanie 9. (5 pkt)

Rozważmy maszynę stosową z instrukcjami dla par:

```ocaml
type instr =
  | IConst of int
  | IPair              (* tworzy parę z dwóch wartości na stosie *)
  | IFst               (* wyciąga pierwszy element pary *)
  | ISnd               (* wyciąga drugi element pary *)
  | ISwap              (* zamienia dwa elementy na szczycie stosu *)
  | IDup               (* duplikuje element na szczycie stosu *)

type value = VInt of int | VPair of value * value
type stack = value list
```

a) (3 pkt) Zaimplementuj interpreter dla tej maszyny:

```ocaml
let rec eval (instrs : instr list) (stack : stack) : stack = _______________
```

b) (2 pkt) Napisz funkcję kompilującą wyrażenie `(a, (b, c))` (gdzie a, b, c są stałymi) na ciąg instrukcji:

```ocaml
let compile_nested_pair a b c : instr list = _______________
```