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

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

ocaml

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

ocaml

```ocaml
let rec f x =
  let g y = x + y in
  if g x > g y
  then g (f x) else h 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 y -> x + y > 2`: _____
- `fun x y z -> x y && z`: _____
- `fun x y z -> x z (y z)`: _____
- `fun f -> (fun x -> f x x)(fun x -> f x x)`: _____
- `fun f x -> f (f x) > x`: _____
- `fun f x -> f (f x) && x > 0`: _____

Zadanie 3. (3 pkt)

Zaimplementuj funkcje, których działanie jest opisane słownie poniżej, używając funkcji `List.fold_left` lub `List.fold_right`. Dla każdego z rozwiązywanych problemów postaraj się wybrać lepszą z tych dwóch funkcji (umożliwiającą prostszą lub efektywniejszą implementację).

- Oblicz iloczyn wartości funkcji dla argumentów z pewnej listy liczb całkowitych.

```ocaml
let product f xs = _____________________________________________
```

- Odwróć kolejność elementów w liście wejściowej.

```ocaml
let reverse xs = _____________________________________________
```

- Dla zadanej listy wejściowej utwórz nową listę zawierającą wartości zadanej funkcji f zaaplikowanej do wszystkich elementów listy wejściowej w oryginalnej kolejności.

```ocaml
let map f xs = _____________________________________________
```

Zadanie 4. (2 pkt)

Rozważ poniższą gramatykę języka poprawnie sparowanych nawiasów, w której S to symbol startowy (i jedyny symbol nieterminalny), nawiasy `()` to symbole terminalne, a ε oznacza słowo puste:

```
S → (S)    S → SS    S → ε
```

Gramatyka ta jest niejednoznaczna. Podaj słowo, które ma przynajmniej dwa drzewa wyprowadzenia z tej gramatyki i narysuj te drzewa:

Zadanie 5. (4 pkt)

Załóżmy, że chcemy zaimplementować interpreter języka z funkcjami wyższego rzędu, mutowanym stanem i wyjątkami używając (metacyklicznie) funkcji wyższego rzędu, i stanu, ale nie używając wyjątków. Zaproponuj typ reprezentujący wartości w takim interpreterze.

```ocaml
type value =
  | VInt of int          (* wartości liczbowe *)
  | VRef of ________     (* stan mutowalny *)
  | VExn of string       (* wyjątek z potencjalnie pustym identyfikatorem *)
  | VClosure of ___________
```

Zadanie 6. (5 pkt)

Rozważmy problem budowy kompletnego drzewa BST z listy posortowanej. Drzewo binarne jest kompletne, jeśli na każdym k-tym poziomie drzewa za wyjątkiem ostatniego zawiera ono wszystkie 2^k węzłów. Liczba wierzchołków na ostatnim poziomie musi być mniejsza lub równa 2^k (jeśli drzewo ma 2^n - 1 elementów dla pewnego n, liczba ta wynosi 2^(n-1)). Przykładowo, drzewo o elementach z elementów [1; 2; 3; 4] może wyglądać następująco:

Przyjmijmy następujący typ drzew:

```ocaml
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
```

Zaimplementuj funkcję typu `int list -> int tree`, która dla zadanej posortowanej listy skonstruuje kompletne drzewo binarne. (W przypadku wątpliwości, wystarczy zapewnić, że funkcja będzie działać poprawnie dla list długości postaci 2^n - 1.) Konieczne będzie zdefiniowanie rekurencyjnej funkcji pomocniczej.

```ocaml
let list_to_complete_tree xs =
```

Napisz (jednym słowem) czy Twoja definicja generuje nieużytki: _____

Zadanie 7. (6 pkt)

Rozważmy drzewa binarne zdefiniowane tak, jak w zadaniu 6. Sformułuj zasadę indukcji dla tego typu danych.

Zadanie 8. (6 pkt)

Rozważmy typowany język z rekurencyjnymi funkcjami wyższego rzędu oraz tablicami, którego składnię abstrakcyjną opisujemy następującym typem danych.

```ocaml
type var = string
type expr =
  | Unit
  | Num      of int
  | Var      of var
  | Seq      of expr * expr
  | Fun      of var * tp * expr
  | App      of expr * expr
  | Fix      of var * tp * tp * var * expr
  | ArrayNew of expr * expr
  | ArrayGet of expr * expr
  | ArraySet of expr * expr * expr
```

Konstrukcja `Fix(f, tp1, tp2, x, body)` tworzy anonimową funkcję rekurencyjną gdzie tp1 oraz tp2 to odpowiednio typ argumentu i wartości, zmienne f oraz x reprezentują odpowiednio całą funkcję oraz jej argument i są związane w ciele funkcji (body). Konstrukcje `ArrayNew(len, v)` tworzy nową tablicę długości len, wypełnioną początkowo wartościami v. Konstrukcje `ArrayGet(arr, i)` oraz `ArraySet(arr, i, v)` oznaczają odpowiednio odczytanie i zapisanie i-tego elementu tablicy arr.

Uzupełnij poniższą definicję typów w tym języku.

```ocaml
type tp =
  | TUnit
  | TInt
  | TArrow of tp * tp
  | ____________
```

Następnie uzupełnij poniższą implementację wyprowadzania typów.

ocaml

```ocaml
let rec infer_type env e =
  match e with
  | Unit  -> TUnit
  | Num _ -> TInt
  | Var x -> Env.lookup env x
  | Seq(e1, e2) ->
      check_type env e1 TUnit;
      infer_type env e2
  | Fun(x, tp, body) ->
      TArrow(tp, infer_type (Env.add env x tp) body)
  | App(e1, e2) ->
```

Zadanie 9. (4 pkt)

Poniżej znajduje się definicja prostego języka wyrażeń arytmetycznych w postaci odwrotnej notacji polskiej wraz ze stosową maszyną obliczającą wartość wyrażenia.

ocaml

```ocaml
type op = Add | Sub

type cmd = Int of int
         | Op of op

type prog = cmd list

let eval_op (op : op) : int -> int -> int =
  match op with
  | Add -> (+)
  | Sub -> (-)

let rec eval_vm (p : prog) (stack : int list) : int =
  match p, stack with
  | [], [k] -> k
  | Int k :: p, s -> eval_vm p (k :: s)
  | Op op :: p, e2 :: e1 :: s -> eval_vm p (eval_op op e1 e2 :: s)

let eval (p : prog) : int =
  eval_vm p []
```

Zaimplementuj funkcję `calc_stack_length : prog -> int`, która oblicza, jak dużo przestrzeni kompilator musiałby zarezerwować na stos, żeby obliczyć wartość wyrażenia używając semantyki zadanej przez maszynę (innymi słowy, dla dowolnego programu p, wartością `calc_stack_length p` jest

maksymalna liczba elementów, które znalazły się jednocześnie na stosie maszyny podczas obliczania wartości programu p).

Przykładowo, wartością wyrażenia `calc_stack_length [Int 4; Int 3; Op Add; Int 5; Op Sub]` jest 2. Możesz założyć, że argument zawsze jest poprawnym programem.

```ocaml
let rec calc_stack_length p =
```