Zadanie 1. (2 pkt)
W poniższych wyrażeniach w języku OCaml podkreśl wolne wystąpienia zmiennych. Dla każdego związania zmiennej, narysuj strzałkę od tego wystąpienia do odpowiadającego mu wiążącego wyrażenia.

```
let x = 3 in
x
```

```
let x = 3 in
let y = x + y in
x + y
```

```
fun f x -> f (x + y)
```

```
let x = z in
x - y
```

```
let x = z in
fun z -> f z x
```

```
fun z -> x - z
```

Zadanie 2. (3 pkt)
W poniższych trzech definicjach funkcji pod literami A, ..., F ukryto wszystkie wywołania funkcji List.map, List.cons (::) i List.append (@). Odszukaj, które z tych funkcji powinny się znaleźć w oznaczonych miejscach tak, aby funkcje zachowywały się zgodnie z nazwą i opisem.

1. Zwróć listę list powstałych przez wstawienie elementu x na każdą pozycję w liście xs:

```
let rec inserts x xs =
  A (B x xs)
    (match xs with
     | [] -> []
     | x' :: xs' ->
       (List.map (fun ys -> C x' ys)
         (inserts x xs')))
```

Uzupełnij:
A = _____
B = _____
C = _____

2. Dla danej listy list zwróć listę będącą konkatenacją wszystkich list:

```
let rec concat xs =
  match xs with
  | [] -> []
  | x' :: xs'
    (D x' (concat xs'))
```

Uzupełnij:
D = _____

3. Zwróć listę, która dla każdego elementu listy xs zawiera dwuelementową listę z tym elementem i wynikiem wywołania f na nim:

```
let rec map_dict f xs =
  match xs with
```

```
  | [] -> []
  | x' :: xs' ->
    (E (F x' (f x'))
       (map_dict f xs'))
```

Uzupełnij:
E = _____
F = _____

Zadanie 3 (5 pkt)
Zdefiniujmy typ reprezentujący liście oraz węzły drzewa przeszukiwań binarnych:

```
type 'a bst =
  | Leaf
  | Node of 'a bst * 'a * 'a bst
```

1. Zaimplementuj funkcję insert_bst, wstawiającą zadaną wartość x
do drzewa przeszukiwań binarnych, zachowując jego porządek:

t.z. insert_bst : 'a -> 'a bst -> 'a bst

2. Korzystając z funkcji insert_bst, zaimplementuj funkcję list_to_bst,
która konwertuje listę na drzewo BST.
Nie używaj jawnej rekurencji — użyj List.fold_left.

t.z. list_to_bst : 'a list -> 'a bst

3. Zaimplementuj funkcję bst_to_list, która dla zadanego drzewa
przeszukiwań binarnych zwraca listę jego elementów w kolejności in-order.

t.z. bst_to_list : 'a bst -> 'a list

4. Czy Twoja funkcja bst_to_list generuje nieużytki?
(jednym słowem): ____

Zadanie 4 (6 pkt)
Rozważmy formuły rachunku zdań zbudowane ze zmiennych zdaniowych,
stałej fałszu oraz spójnika implikacji.
Możemy je opisać następującym typem danych:

```
type 'a form =
  | VarF of 'a
  | BotF
  | ImpF of 'a form * 'a form
```

1. Sformułuj zasadę indukcji dla tego typu danych.

Okazuje się, że taki typ jest monadą gdzie,
funkcje returnM oraz bindM zdefiniowane są nastepujaco

```
let returnM x = VarF x

let rec bindM m f =
  match m with
  | VarF x        -> f x
  | BotF          -> BotF
```

```
  | ImpF (m1, m2)   -> ImpF (bindM m1 f, bindM m2 f)
```

2. Pokaż, że funkcje returnM bindM spełniają poniższe równoważności oczekiwane od monady.

bindM (returnM a) f = f a

bindM m returnM = m

bindM (bindM m f) g = bindM m (fun a -> bindM (f a) g)

W tej ostatniej równoważnosci możesz założyć, że funkcja f dla dowolnego argumentu poprawnie się oblicza do wartości bez efektów ubocznych.

Zadanie 5 (2pkt)

Mając typ Form oraz funkcję BindM z poprzedniego zadania definiujemy poniższą funkcję.

```
let foo m =
  bindM m (fun a -> a)
```

1. Napisz jaki jest typ tej funkcji.

2. Napisz (jednym zdaniem) czym różnią się kontrakty od typów.

Zadanie 6 (2 pkt)

Zaproponuj gramatykę opisującą składnie konkretną formuł z zadania 4.
Zadbaj o to, by gramatyka była jednoznaczna, a implikacja wiązała w prawo.

Zadanie 7 (10 pkt)

Rozważamy prosty język funkcyjny Egz, którego składnia abstrakcyjna wyrażeń zawiera:
- stałą numeryczną,
- binarną operację dodawania,
- zmienną,
- funkcję wieloargumentową (w tym 0 argumentów),
- aplikację wieloargumentową (w tym 0 argumentów),
- wyrażenie let z wiązaniem dowolnej liczby zmiennych (w tym 0).

1. Zdefiniuj typ danych exp reprezentujący składnię abstrakcyjną języka Egz.

type exp = ___

2. W celu zdefiniowania semantyki języka Egz zakładamy,
że dany jest typ środowiska Env : (string * val) map wraz z funkcjami:

Env.add : string -> val -> Env -> Env
Env.find : string -> Env -> val

Zdefinuj typ danych val reprezantujący wartości wyrażeń języka Egz (2 pkt)
type val = ___

3. Zdefiniuj funkcję, stanowiącą interpreter wyrażeń języka Egz (4 pkt)

let rec eval (e : exp) (env : env) : val = ___

4. Napisz funkcję, która usuwa wyrażenia let, zastępując je innymi semantycznie równoważnymi wyrażeniami z języka Egz (3 pkt)

let rec desugar (e : exp) : exp = ___