Exercise 1: **Functional Inlining**

Perform function inlining on the following program.

```haskell
let f y =
    let z = y + 1 in
    5 + y
in
let x = 5 in
f x + f 7
```

Exercise 2: **Specialization**

Consider the following program:

```haskell
let rec fold = fun f -> fun a -> fun l ->
    match l with
    | [] => a
    | x :: xs => f a (fold f a xs)
in
let f = fun x -> fun s -> x + s in
let sum = fold f 0
```

Perform function specialization and inlining.

Exercise 3: **Deforestation**

Let `comp`, `id`, `map`, `foldl`, and `tabulate` be defined as in the lecture. Consider the following code:

```haskell
let sq x = x * x
let sumSquares = comp (foldl (+) 0) (comp (map sq) (tabulate id))
```

Simplify the function `sumSquares` as much as possible.

Exercise 4: **Strictness Analysis**

Consider the following algorithm:

```haskell
let rec ea x y =
    if x = 0 then y else ea (y mod x) x
```

Use the strictness analysis to detect if the function is strict on both arguments.