QUESTION 1. (15 pts)

We want to define a function called `occurcount` in Haskell. This function is intended to count the number of occurrences of a list in another list. A small list occurs in a large list if the elements of the small list occurs in the same order in the large list not necessarily in successive positions. That is, the small list `[1, 2]` occurs in the large list `[2, 1, 3, 2, 1]`. The definition of the function `occurcount` is given. However, it uses another function called `occurfind` that finds if a small list (the first parameter) occurs in the large list (the second parameter) and returns `Nothing` if the small list does not occur in the large list, and returns `Just y` where `y` is the remaining part of the large list after small list is found. Note that, the main function `occurcount` then will check the remaining part of the large list to search for other occurrences of the small list. For example `occurfind[1, 2][2, 1, 3, 2, 1, 3, 2, 3]` returns `Just[1, 3, 2, 3]`. Complete the definition of `occurfind`.

```haskell
occurfind (_:_) [] = Nothing
occurfind [] r = Just r
occurfind (a:ar) (b:br) = if (a==b) then
    occurfind ar br
  else
    occurfind (a:ar) br

occurcount _ [] = 0
occurcount x y =
  let
    q = occurfind x y
  in
    case q of
      Nothing -> 0
      Just qq->1+occurcount x qq
```

-- Examples in hugs
-- Main> occurcount [1,2,3] [3,1,2,1,3,1,2,1,2,1,3,2,4]
-- 2
-- Main> occurcount [1,2,3] [3,1,2,1,3,1,2,1,2,1,3,2,1,2,3,4]
-- 3

Do not define any auxiliary functions in the implementations.
QUESTION 2. (20 pts)

Define higher order \textit{reduce}_2 function that reduces a list of a list to a scalar value. The higher order function \textit{reduce}_2 is similar to the ordinary higher order \textit{reduce} function that takes two functions and two default values and applies the first function and the first default value to reduce inner lists to scalar values. Then, it applies the second function and the second default value to the list obtained from the list of lists and reduces it to a scalar value.

Some examples are given below. Fill the blanks below to answer the questions stated.

-- Functions to be applied to lists. Their types: \textbf{Int}$$\rightarrow$$\textbf{Int}$$\rightarrow$$\textbf{Int}

\begin{align*}
\text{add } x \ y &= x+y \\
\text{mult } x \ y &= x*y \\
\text{sub } x \ y &= x-y
\end{align*}

-- Ordinary reduce function with type: \textbf{(a} $$\rightarrow$$ \textbf{b} $$\rightarrow$$ \textbf{b}) $$\rightarrow$$ \textbf{b} $$\rightarrow$$ \{	extbf{a}\} $$\rightarrow$$ \textbf{b}

-- Reduce applies \textit{f2} as follows: \((\textit{f2} \ b_1 \ (\textit{f2} \ b_2 \ ... \ (\textit{f2} \ b_n \ \textit{n2}) \ ... \ ))

-- where \(b_1, b_2, ... , b_n\) are the elements of a list.

\begin{align*}
\text{reduce } \textit{f2} \ \textit{n2} \ [] &= \textit{n2} \\
\text{reduce } \textit{f2} \ \textit{n2} \ (x:xs) &= (\textit{f2} \ x \ (\text{reduce } \textit{f2} \ \textit{n2} \ xs))
\end{align*}

-- \textit{reduce}_2 applies \((\textit{f1} \ a_1 \ (\textit{f1} \ a_2 \ ... \ (\textit{f1} \ a_n \ \textit{n1}) \ ... \ ))\)

-- where \(a_1, a_2, ... , a_n\) are obtained from each sublists

-- by applying \((\textit{f2} \ b_1 \ (\textit{f2} \ b_2 \ ... \ (\textit{f2} \ b_n \ \textit{n2}) \ ... \ ))\)

-- where \(b_1, b_2, ... , b_n\) are the elements of sublist

-- Determine the type of \textit{reduce}_2:\textbf{(a} $$\rightarrow$$ \textbf{b} $$\rightarrow$$ \textbf{b}) $$\rightarrow$$ \textbf{(c} $$\rightarrow$$ \textbf{a} $$\rightarrow$$ \textbf{a}) $$\rightarrow$$ \textbf{b} $$\rightarrow$$ \textbf{a} $$\rightarrow$$ \{	extbf{[c]}\} $$\rightarrow$$ \textbf{b}

-- Define \textit{reduce}_2 by only using \textit{reduce} & \textit{reduce}_2.

\begin{align*}
\text{reduce}_2 \ \textit{f1} \ \textit{f2} \ \textit{n1} \ \textit{n2} \ [] &= \textit{n1} \\
\text{reduce}_2 \ \textit{f1} \ \textit{f2} \ \textit{n1} \ \textit{n2} \ (x:xs) &= \textit{f1} \ (\text{reduce } \textit{f2} \ \textit{n2} \ x) \ (\text{reduce}_2 \ \textit{f1} \ \textit{f2} \ \textit{n1} \ \textit{n2} \ xs)
\end{align*}

-- Some example applications of \textit{reduce}_2

\begin{align*}
\text{sumprod} &= \text{reduce}_2 \ \textit{add} \ \textit{mult} \ 0 \ 1 \\
\text{sumsub} &= \text{reduce}_2 \ \textit{add} \ \textit{sub} \ 0 \ 0 \\
\text{subsum} &= \text{reduce}_2 \ \textit{sub} \ \textit{add} \ 0 \ 0 \\
\text{subsub} &= \text{reduce}_2 \ \textit{sub} \ \textit{sub} \ 0 \ 0
\end{align*}

-- Determine the types of above functions: \{	extbf{[Int]}\} $$\rightarrow$$ \textbf{Int}

-- Some applications with \textbf{Hugs} interpreter

\begin{align*}
\text{-- Main} & \text{> sumprod} \ \{	extbf{[1,2],[4,1,2],[4,5],[2,3]}\} \\
& \quad = 36 \\
\text{-- Main} & \text{> sumprod} \ \{	extbf{[2,1],[2,1,4],[5,4],[3,2]}\} \\
& \quad = 36 \\
\text{-- Main} & \text{> sumsub} \ \{	extbf{[1,2],[4,1,2],[4,5],[2,3]}\} \\
& \quad = 2 \\
\text{-- Main} & \text{> sumsub} \ \{	extbf{[2,1],[2,1,4],[5,4],[3,2]}\} \\
& \quad = 8
\end{align*}
-- Main> subsum [[1,2],[4,1,2],[4,5],[2,3]]
-- 0
-- Main> subsum [[2,1],[2,1,4],[5,4],[3,2]]
-- 0
-- Main> subsub [[1,2],[4,1,2],[4,5],[2,3]]
-- -6 __________  <-- Determine the result
-- Main> subsub [[2,1],[2,1,4],[5,4],[3,2]]
-- -4 __________  <-- Determine the result

Do not define any auxiliary functions in the implementations.
QUESTION 3. (20 pts)

Show the lifetimes of the variables in the following C program. In order to do this, you need to trace the execution of the program until its termination. Use a time-chart to show when the variables are created and destroyed related to the functions’ executions (i.e., call and return).

```c
#include <stdio.h>

int a=1;

int f2(int*);

int f1(int x)
{
    int b=x;
    int *px;
    printf("ENTER f1 %d\n",x);
    px=(int *) malloc(sizeof(int));
    *px = b-a;
    if ((*px>0) && (*px<3))
        *px=f1(*px);
    if (*px>1)
        b=f2(px);
    return (b);
}

int f2 (int *qx)
{
    static int a=2;
    printf("ENTER f2 %d\n",*qx);
    if (qx)
        free (qx);
    a=f1(a);
    return (a);
}

main()
{
    int a = 3;
    f1(a);
}

// The output of the program is as follows:
// ENTER f1 3
// ENTER f1 2
// ENTER f1 1
// ENTER f2 2
// ENTER f1 2
// ENTER f1 1
```
Answer of Question 3: 5
QUESTION 4. (15 pts)

Assume that, you want to throw a party and keep the guest information in a recursive data structure. A
value of Guest can be either MaleGuest or FemaleGuest. You have concerns to avoid male population in
the group being majority, so you have rules:

- Male guests can bring 0 or more Female friends.
- Male guests cannot bring any Male friends.
- Female guests can bring 0 or 1 Male friends.
- Female guests can bring 0 or more Female friends.
- Each friend brought to the party is a guest. That means he/she can bring his/her own friends
respecting to the rules above. So the number of guests can increase recursively.

You are asked to define the necessary Haskell data type declarations to define these Guest values. Your
value should guarantee that the rules above are not violated (i.e. no male guest can bring a male friend).
Use the following naming conventions and constraints:

- A Guest is either Male tagged value of a male guest or Female tagged value of a female guest.
- A female guest is FG tagged value of a cartesian product of a string for name, male friend information,
  and female friend information.
- A male guest is MG tagged value of a cartesian product of a string for name, and female friend
  information.
- You can define other required types for restricted values.
- You can use list data type of Haskell in your type definitions ([], :).
- Remember Haskell type definitions are global, it is possible to define mutually recursive types.

a) Give the Haskell definitions required to define this data type:

```haskell
data NoneorOne a = None | One a

data FemaleGuest = FG (String, NoneorOne MaleGuest, [FemaleGuest])
data MaleGuest = MG (String, [FemaleGuest])

data Guest = Male MaleGuest | Female FemaleGuest
```

b) Give mathematical description of all these types in set notation (Use operators like ×, +, no tags. You
can use α List as a predefined list type)

\[
\text{FemaleGuest} = \text{String} \times (\text{Unit} + \text{MaleGuest}) \times (\text{FemaleGuest List})
\]

\[
\text{MaleGuest} = \text{String} \times (\text{FemaleGuest List})
\]

\[
\text{Guest} = \text{MaleGuest} + \text{FemaleGuest}
\]

with Guest data type.

```haskell
Female (FG ("Ayse", One (MG ("Ali", [FG ("Haticce", None, [])]),
                  [FG ("Oya", None, []),
                   FG ("Fatma", One (MG ("Hasan", []), []))])))
```
QUESTION 5. (20 points)

Assume you have a C version with nested function definitions allowed. (A function definition nested in a function body has a local scope of that function, similar to a local variable)

```c
struct Coord { int x, int y};
int a;
int h();

int f(int t) {
    ...
}  // Coord->typename,a->int, h->int func, f->int func, t->int
int g(double a) {
    double f(int u) {
        ....  // Coord->typename,a->double,u->int,h->int func,f->double func,g int func
    }
    ...
    h();  // Coord->typename,a->double, h->int func, f->double func, g int func
}
int h() {
    ...
}  // Coord->typename,a->int, h->int func, f->int func, g->int func

int main() {
    double z;  // Coord->typename,a->int,h->int func,f->int func,g->int func,main->int func,z->double
    int c;
    ...
}  // Coord->typename,a->int,h->int func,f->int func,g->int func,main->int func,z->double,c->int
```

a) Assuming this version of C uses **static scope** (static binding), fill in the environment of the corresponding lines (the ... positions in the code) above. Give the environment as a set of all possible bindings. Give name ↦ type pairs where type is like “double, int, typename, int func, double func”.

b) Assuming this version of C uses **dynamic scope** (dynamic binding), and at the instance `main()` calls `g(...)` and `g()` calls `h()`, what is the environment in `h()` (forth environment above).
QUESTION 6. (20 points)

What is the output of the following program if the parameter passing mechanism is:

a) reference mechanism, variable (call by reference)
b) copy mechanism, copy-in (call by value)
c) copy mechanism, copy-in-copy-out (value-return technique)

d) Assume x=5 in main() and you call notcalled(++x). What is the value of x after the call. Assuming:

Eager evaluation, copy-in (pass by value): 6

Normal order evaluation (pass by name): 6