1. [20 points] **Lambda Calculus: Reductions**

   There are two kinds of *reduction steps* in the λ-calculus:

   - α-steps, written `e =a> e'`,
   - β-steps, written `e =b> e'`,
   - transitive-steps, written `e =*> e'`.

   We write `e =*> e'` if there is a sequence of 0 or more steps `e => ... => e'` where each `=>` is *either* an α- *or* β-step. For each of the following *pairs* of lambda calculus expressions, of the form `e1 =?> e2`, circle the most appropriate reduction step (or `none` if none of the steps are valid.

   (a) [4 points]

   ```
       (\n f x -> f (n f x)) (\f x -> x)

   =?> (\n f x -> f (n f x)) (\g y -> y)
   ```

   1. `=a>` (α-step)
   2. `=b>` (β-step)
   3. `=*>` (transitive-step)
   4. None (Invalid step)

   (b) [4 points]

   ```
       (\n f x -> f (n f x)) (\f x -> x)

   =?> (\f x -> f ((\f x -> x) f x))
   ```

   1. `=a>` (α-step)
   2. `=b>` (β-step)
   3. `=*>` (transitive-step)
   4. None (Invalid step)

   (c) [4 points]

   ```
       (\n f x -> f (n f x)) (\f x -> x)

   =?> (\n f x -> f (n f x)) (\g y -> g)
   ```

   1. `=a>` (α-step)
   2. `=b>` (β-step)
   3. `=*>` (transitive-step)
   4. None (Invalid step)

(d) [4 points]

```
    (\n f x -> f (n f x)) (\f x -> x)

=?> (\f x -> (f f) x)
```

1. `=a>` (α-step)
2. `=b>` (β-step)
3. `=*>` (transitive-step)
4. None (Invalid step)

(e) [4 points]

```
    (\n f x -> f (n f x)) (\f x -> x)

=?> (\f x -> f (f x))
```

1. `=a>` (α-step)
2. `=b>` (β-step)
3. `=*>` (transitive-step)
4. None (Invalid step)

2. [15 points] **Lambda Calculus: Lists**

As promised in lecture, in this problem you will develop an encoding of *lists* in the λ-calculus.

Lets implement `NIL` – *empty* lists – as just `FALSE` defined in the cheat sheet at the end of the exam.

```
let NIL = FALSE
```

(a) [5 points] **Construct and Destruct**

Fill in the implementations of `CONS`, `HEAD` and `TAIL`.

`CONS h t` should return a list whose "head" is `h` and "tail" is `t`.

```
let CONS = \h t -> ________________________________________
```

`HEAD l` should return the "head" of the list `l`.

```
let HEAD = \l ->   ________________________________________
```

`TAIL l` should return the "tail" of the list `l`.

```
let TAIL = \l ->   ________________________________________
```

When you are done, you should get the following behavior:

```
eval list1 :
  HEAD (CONS apple (CONS banana (CONS cantaloupe NIL)))
  =~> apple

eval list2 :
  HEAD (TAIL (CONS apple (CONS banana (CONS cantaloupe NIL))))
  =~> banana
```

(b) [10 points] **Access**

Recall that we represent the natural number `n` as function that is applied to a base value `n` times, e.g.,

```
let ZERO  = \f x -> x
let ONE   = \f x -> f x
let TWO   = \f x -> f (f x)
let THREE = \f x -> f (f (f x))
```

Fill in the implementation of `GetNth` such that `GetNth n l` returns the `n`-th element of a list `l`

```
let GetNth = \n l -> ___________________________________________
```

When you are done, you should see the following behavior

```
eval nth0 :
  GetNth ZERO (CONS apple (CONS banana (CONS cantaloupe NIL)))
  =~> apple

eval nth1 :
  GetNth ONE (CONS apple (CONS banana (CONS cantaloupe NIL)))
  =~> banana

eval nth2 :
  GetNth TWO (CONS apple (CONS banana (CONS cantaloupe NIL)))
  =~> cantaloupe
```

NOTE: You can assume that in every call `GetNth l n`, that `l` is a list with at least `n` elements.