Exercise 1 : Typing (10 points)

For each of the following lambda terms,

- give a possible type and indicate whether this type is the *only* possible type, or
- indicate that the term is not typable.

We consider the typing rules of the simply typed lambda calculus (fig. 9-1) with arithmetic expressions including base types `Nat` and `Bool`, sums (11-9) and pairs (11-5), references (13-1) and recursion (11-12).

1. λx. λy. λz. `x (y z)`
   Solution: This term has many possible types, all of the following shape:
   `(A -> B) -> (C -> A) -> C -> B`, where `A`, `B` and `C` stand for any valid type (e.g., `Nat`, `Bool`, `Nat -> Bool`).

2. λx. λy. `(x y) (y x)`
   Solution: untypable

3. λx. λy. `(case x of inl a => !a | inr b => b) !y`
   Solution: This term has many possible types, all of the following shape:
   `(Ref (Y -> A) + (Y -> A))) -> Ref Y -> A`, where `A` and `Y` stand for any valid type (e.g., `Nat`, `Bool`, `Nat -> Bool`).

4. λx. `if x.1 then x.2 else succ x.2`
   Solution: The only possible type is: `Bool` × `Nat -> Nat`.

5. λx. `fix (fix x)`
   Solution: This term has many possible types, all of the following shape:
   `((A -> A) -> (A -> A)) -> A`, where `A` stands for any valid type (e.g., `Nat`, `Bool`, `Nat -> Bool`).

Exercise 2 : Adding call-by-name to the call-by-value simply-typed lambda calculus. (10 points)

Instead of emulating call-by-name evaluation using thunks, we extend the simply typed lambda calculus (call-by-value, no extensions) to provide direct support for call-by-name terms. We introduce two new syntactic forms and a new type:

t	::=	...	**terms**
	|	`delay` t	a thunk that delays the evaluation of t
	|	`force` t	force the evaluation of the thunk t
v	::=	...	**values**
	|	`delay` t	
T	::=	...	**types**
	|	`Delayed` T	type of a term that can be forced to a term of type T

Your task for this assignment is two-fold:

1. Introduce the new evaluation rules and typing rules that follow from the given syntactic extension and the notion of call-by-name evaluation so that the resulting system is sound (and useful).

2. Prove soundness of your system by detailing the new cases of the inductive proofs of preservation and progress. You need only discuss the cases that result from the evaluation and typing rules that you introduced, and, if needed, you may use (without proving them) the standard lemmas of *inversion of typing*, *canonical forms*, and *uniqueness of types*.

   HINT: For both proofs you can use induction on the derivation of $\texttt{t} : \texttt{T}$.

Solution:

$$\frac{\Gamma \vdash \texttt{t} : \texttt{T}}{\Gamma \vdash \texttt{delay t} : \texttt{Delayed T}} \quad \text{(T-Delay)}$$

$$\frac{\Gamma \vdash \texttt{t} : \texttt{Delayed T}}{\Gamma \vdash \texttt{force t} : \texttt{T}} \quad \text{(T-Force)}$$

$$\texttt{force}(\texttt{delay t}) \to \texttt{t} \quad \text{(E-Force-Delay)}$$

$$\frac{\texttt{t} \to \texttt{t}'}{\texttt{force t} \to \texttt{force t}'} \quad \text{(E-Force)}$$

Preservation If $\texttt{t} : \texttt{T}$ and $\texttt{t} \to \texttt{t}'$, then $\texttt{t}' : \texttt{T}$. Two new cases, existing cases remain unchanged:

Case T-Delay: $\texttt{delay t}$ is a value.

Case T-Force: $\texttt{t} = \texttt{force t}_1 \quad \texttt{t}_1 : \texttt{Delayed T}$

Two evaluation rules can apply:

Case E-Force-Delay: $\texttt{t} = \texttt{force}(\texttt{delay t}')$

To prove: from $\texttt{force}(\texttt{delay t}') : \texttt{T}$ follows $\texttt{t}' : \texttt{T}$.
From the assumption of T-Force, we know that $\texttt{delay t}' : \texttt{Delayed T}$.
By inversion of T-Delay follows $\texttt{t}' : \texttt{T}$.

Case E-FORCE: $\texttt{t} = \texttt{force t}_1$

To prove: from $\texttt{force t}_1 : \texttt{T}$ follows $\texttt{force t}_1' : \texttt{T}$
From the assumption of T-Force, we know that $\texttt{t}_1 : \texttt{Delayed T}$
And by ind. hyp. thus $\texttt{t}_1' : \texttt{Delayed T}$
By applying T-Force we get $\texttt{force t}_1' : \texttt{T}$.

□

Progress By induction on the derivation of $\texttt{t} : \texttt{T}$. Two new cases, existing cases remain unchanged:

- T-delay: already a value.
- T-force: apply induction hypothesis, split it in two cases:
  - t is a value (from canonical forms it must be `delay t` since its type is `Delayed t`), thus E-force-delay applies
  - t can take a step: E-force applies

□

Exercise 3 : Hacking in the untyped call-by-value lambda calculus. (10 points)

Pierce describes encoding lists using fold on p. 63 and pp. 350-352. Let's use a slightly different encoding: we represent a list by what it means to pattern match on it.

More precisely, a list is encoded as a function that takes two functions as its arguments. It applies the first argument (let's call it `fNil`) to some unspecified value if the list is empty, else it applies the second function (`fCons`) to the head and the (encoding of the) tail of the list. In Scala notation, where `xs` is the list to be encoded, the corresponding function is:

```
fNil => fCons => xs match {
  case Nil => fNil()
  case a :: as => fCons(a)(as)
}
```

Formally, the encoding from a Scala list to the corresponding function in lambda calculus is given by the function [| `xs` |] that takes a Scala list `xs` to its lambda term:

[| `List()` |] = λ`fNil.` λ`fCons.` `fNil unit`
[| `x :: xs` |] = λ`fNil.` λ`fCons.` `fCons x` [| `xs` |]

As an example, assuming the constants `A`, `B`, and `C` are defined (only for the purpose of this example):

[| `List(A, B, C)`|] = λ`fNil.` λ`fCons.` `fCons A (`λ`fNil.` λ`fCons.` `fCons B (`
λ`fNil.` λ`fCons.` `fCons C (`λ`fNil.` λ`fCons.` `fNil unit)))`

Your task is to program the following functions in the untyped call-by-value lambda calculus (as described on the lectures and in Chapter 5 of the book) using the above encoding. More specifically, your answer should consist of 4 lambda terms (one for each of the functions described below):

NOTE: You may assume `unit`, `zero`, `succ`, and `fix` are defined for you.

- **head**: return the first element of a given list
  Solution: `let head =` λ`xs. xs (`λ`u. u) (`λ`x.` λ`xs. x)`
- **tail**: returns the tail of a given list (i.e., the list without its first element)
  Solution: `let tail =` λ`xs. xs (`λ`u. u) (`λ`x.` λ`xs. xs)`
- **len**: return the length of a given list (where the length is a Church-encoded natural)
  Solution: `let len = fix` λ`me.` λ`xs. xs (`λ`x. zero) (`λ`x.` λ`xs. succ (me xs))`
- **rev**: return the reverse of a given list.
  Solution: `let rev = (fix` λ`me.` λ`acc.` λ`xs. xs (`λ`x. acc) (`λ`x.` λ`xs. me (`λ`fNil.`
  λ`fCons. fCons x acc) xs) (`λ`fNil.` λ`fCons. fNil unit)`

```
// In Untyped Scala:
def head(xs: ?) =
  xs(
    (u: ?) => u )(
    (x: ?) => (xs: ?) => x
  )

def tail(xs: ?) =
  xs(
    (u: ?) => u )(
    (x: ?) => (xs: ?) => xs
  )

def len(xs: ?) = {
  val len0 = (me: ?) =>
         (xs: ?) =>
    xs(
      (x: ?) => zero )(
      (x: ?) => (xs: ?) => succ(me(xs))
    )
  fix(len0)(xs)
}

def rev(xs: List[Int]): List[Int] = {
  def rev0(xs: List[Int], acc: List[Int]): List[Int] = xs match {
    case Nil => acc
    case y :: ys => rev0(ys, y :: acc)
  }
  rev0(xs, List())
}

def rev(xs: ?) = {
  val rev0 = (me: ?) =>
         (xs: ?) => (acc: ?) =>
      xs(
        (_: ?) => acc )( // case Nil
        (y: ?) => (ys: ?) => me(ys)(cons(y, acc)) // case y :: ys
      )
  fix(rev0)(xs)(nil)
}
```