1. Building derivations

1.1

Using the operational semantics of Nano at the end of this document, build a derivation of `E, f 5 => ?, ?` where `E = [f -> <[], \x y -> x>]`

1.2

Using the operational semantics of Nano at the end of this document, build a derivation of `E, (\x y -> x) 5 => ?, ?` where `E = [f -> <[], \x y -> x>]`

1.3

Using the type system of Nano at the end of this document, build a derivation of `[] |- \x y -> x :: Int -> Int -> Int`

1.3

Using the type system of Nano at the end of this document, build a derivation of `G |- f 5 :: Int -> Int` where `G = [f : forall a b. a -> b -> a]`

2. Negation Normal Form

```haskell
type Id = String

data Formula = Var Id
  | Not Formula
  | And Formula Formula
  | Or Formula Formula
  deriving Show

type Env = [(Id, Bool)]
```

Implement a recursive function `nnf` that converts a formula to negation normal form:

```
nnf :: Formula -> Formula
nnf f = ???
```

3. Folds

Convert the function `append` into an equivalent function that uses a fold instead of recursion:

```
append :: [a] -> [a] -> [a]
append [] ys = ys
append (x:xs) ys = x:(append xs ys)

-- This one shouldn't use recursion!
append' :: [a] -> [a] -> [a]
append' xs ys = ???
```

Syntax and Semantics of Nano2

Expression syntax:

```
e ::= n | x | e1 + e2 | let x = e1 in e2 | \x -> e | e1 e2
```

Operational semantics:

```
[Var]      E, x => E, E[x]           if x in dom(E)

[Add]      E, n1 + n2 => E, n        where n == n1 + n2

                E, e1 => E', e1'
[Add-L]    --------------------------
           E, e1 + e2 => E', e1' + e2

                E, e2 => E', e2'
[Add-R]    --------------------------
           E, n1 + e2 => E', n1 + e2'

[Let]      E, let x = v in e2 => E[x->v], e2

                         E, e1 => E', e1'
[Let-Def]  --------------------------------------------
           E, let x = e1 in e2 => E', let x = e1' in e2

[Abs]      E, \x -> e => E, <E,\x -> e>

[App]      E, <E1,\x -> e> v => E1[x->v], e

              E, e1 => E', e1'
[App-L]    ----------------------
           E, e1 e2 => E', e1' e2

             E, e => E', e'
[App-R]    ------------------
           E, v e => E', v e'
```

Syntax of types:

```
T ::= Int | T1 -> T2 | a
S ::= T | forall a . S
```

Typing rules:

```
[T-Num]  G |- n :: Int

         G |- e1 :: Int   G |- e2 :: Int
[T-Add]  -------------------------------
               G |- e1 + e2 :: Int

[T-Var]  G |- x :: S          if x:S in G

           G, x:T1 |- e :: T2
[T-Abs]  ------------------------
         G |- \x -> e :: T1 -> T2

         G |- e1 :: T1 -> T2   G |- e2 :: T1
[T-App]  -----------------------------------
                  G |- e1 e2 :: T2

         G |- e1 :: S   G, x:S |- e2 :: T
[T-Let]  --------------------------------
            G |- let x = e1 in e2 :: T

         G |- e :: forall a . S
[T-Inst] ----------------------
          G |- e :: [a / T] S

               G |- e :: S
[T-Gen]  ---------------------- if not (a in FTV(G))
         G |- e :: forall a . S
```

Here $n \in \mathbb{N}$ is natural number, $v \in \text{Val}$ is a value, $x \in \text{Var}$ is a variable, $e \in \text{Expr}$ is an expression, $E \in \text{Var} \to \text{Val}$ is an environment, $a \in \text{TVar}$ is a type variable, $T \in \text{Type}$ is a type, $S \in \text{Poly}$ is a type scheme (a poly-type), $G \in \text{Var} \to \text{Poly}$ is a type environment (a context).