Exercises

- Infer the type (if possible) of `\f x -> x (f x)`.

```
                                                     ──────────────────────────────── [H = B
]    ──────────────────────────────── [I = D]
                                                     f : B, x : D ⊢ f : H
     f : B, x : D ⊢ x : I
──────────────────────────────── [F = B]     ─────────────────────────────────────────
──────────────────────────────────── [H = I → G]
f : B, x : D ⊢ x : F                                 f : B, x : D ⊢ f x : G
───────────────────────────────────────────────────────────────────────────── [F =
 G → E]
f : B, x : D ⊢ x (f x) : E
────────────────────────────────────── [C = D → E]
f : B ⊢ \ x → x (f x) : C
───────────────────────────────────── [A = B → C]
⊢ \ f x → x (f x) : A
```

We collect the equations:

```
A = B → C
C = D → E
F = G → E
F = B
H = I → G
H = B
```

```
I = D
```

Now apply the type inference algorithm (TODO).

- Prove that the following are isomorphisms:
  - `Prod (B -> A) (C -> A) ~ Sum B C -> A`

```
q2a-to : {A B C : Set} -> Prod (B -> A) (C -> A) -> Sum B C -> A
q2a-to (pair f g) (left b) = f b
q2a-to (pair f g) (right c) = g c

q2a-from : {A B C : Set} -> (Sum B C -> A) -> Prod (B -> A) (C -> A)
q2a-from fg = pair (\ b -> fg (left b)) (\ c -> fg (right c))
```

To check that these are mutually inverse, let `f : B -> A` and `g : C -> A` be arbitrary functions. We need `q2a-from (q2a-to (pair f g)) = pair f g`. We check this by computation:

```
q2a-from (q2a-to (pair f g)) = pair (\ b -> (q2a-to pair f g) (left b)) (\ c -> (q2a-to pair f g) (right c))
                             = pair (\ b -> f b) (\ c -> g c)
                             = pair f g
```

We also need to check the opposite order, so suppose `fg : Sum B C -> A` is an arbitrary function. We need `q2a-to (q2a-from fg) = fg`. Now `fg` itself is a function, so to check for equality, let `bc : Sum B C` be an arbitrary element. We need to check that `q2a-to (q2a-from fg) bc = fg bc`. To check this, we consider both cases for `bc`:

```
q2a-to (q2a-from fg) (left b) = q2a-to (pair (\ b -> fg (left b)) (\ c -> fg (right c))) (left b)
                            = (\ b -> fg (left b))
                            = fg (left b)

q2a-to (q2a-from fg) (right c) = q2a-to (pair (\ b -> fg (left b)) (\ c -> fg (right c))) (left b)
                             = (\ c -> fg (right c))
                             = fg (right c)
```

- `C -> Prod A B ~ Prod (C -> A) (C -> B)`

```
q2b-to : {A B C : Set} -> (C -> Prod A B) -> Prod (C -> A) (C -> B)
q2b-to fg = pair (\ c -> proj1 (fg c)) (\ c -> proj2 (fg c))

q2b-from : {A B C : Set} -> Prod (C -> A) (C -> B) -> C -> Prod A B
q2b-from (pair f g) = \ c -> pair (f c) (g c)
```

To check that these are inverse in one direction, let `fg : C -> Prod A B` be an arbitrary function. We need that `q2b-from (q2b-to fg) = fg`. Since `fg` is a function, we check this for every input `c : C`, so we need that `q2b-from (q2b-to fg) c = fg c`.

```
q2b-from (q2b-to fg) c = q2b-from (pair (\ c -> proj1 (fg c)) (\ c -> proj2 (fg c))) c
                       = (\ c -> pair ((\ c -> proj1 (fg c)) c) ((\ c -> proj2 (fg c)) c)) c
                       = pair ((\ c -> proj1 (fg c)) c) ((\ c -> proj2 (fg c)) c)
                       = pair (proj1 (fg c)) (proj2 (fg c))
                       = fg c
```

Now for the opposite order, let `f : C -> A` and `g : C -> B`. We need that `q2b-to (q2b-from (pair f g)) = pair f g`. To check that these two elements are equal, considering that it is a pair, we need to check that the coordinates are equal, that is, that `proj1 (q2b-to (q2b-from (pair f g))) = proj1 (pair f g)` and `proj2 (q2b-to (q2b-from (pair f g))) = proj2 (pair f g)`. In other words, we need to check that `proj1 (q2b-to (q2b-from (pair f g))) = f` and `proj2 (q2b-to (q2b-from (pair f g))) = g`. Now both of these coordinates are functions, respectively with types `C -> A` and `C -> B`. So to check that the coordinates are equal, we need to check that for every input, they have the same output. So let `c : C` be an arbitrary element. We check, by computation, that `proj1 (q2b-to (q2b-from (pair f g)))` `c = f c`, and that `proj2 (q2b-to (q2b-from (pair f g))) c = g c`

```
proj1 (q2b-to (q2b-from (pair f g))) c = proj1 (pair (\ c -> proj1 ((q2b-from (pair f g)) c)) (\ c -> proj2 ((q2b-from (pair f g)) c)))
                                       = (\ c -> proj1 ((q2b-from (pair f g)) c)) c
                                       = proj1 ((q2b-from (pair f g)) c)
                                       = proj1 ((\ c -> (pair (f c) (g c))) c)
                                       = proj1 (pair (f c) (g c))
                                       = f c

proj2 (q2b-to (q2b-from (pair f g))) c = proj2 (pair (\ c -> proj2 ((q2b-from (pair f g)) c)) (\ c -> proj2 ((q2b-from (pair f g)) c)))
                                       = (\ c -> proj2 ((q2b-from (pair f g)) c)) c
                                       = proj2 ((q2b-from (pair f g)) c)
                                       = proj2 ((\ c -> (pair (f c) (g c))) c)
                                       = proj2 (pair (f c) (g c))
                                       = g c
```

- (**Advanced**): Extend the algorithm of type inference to STLC with `Sum` and `Prod`.
- Define the *multiplication* operation, as *repeated addition* (defined in the lecture).

  Hint:

  `m * 1 = m`

  `m * (1 + n) = m * 1 + m * n = m + (m * n)`

```
mult : Nat -> Nat -> Nat
mult n 0 = 0
mult n (suc k) = add n (mult n k)
```

- Define the function `max : Nat -> Nat -> Nat` which returns the maximum of two `Nat`s

```
max : Nat -> Nat -> Nat
max zero k          = k
max (suc n) zero    = suc n
max (suc n) (suc k) = suc (max n k)
```

- Define the function `maxl : List Nat -> Nat` which returns the maximum element of a list, or `zero` if the list is empty.

```
maxl : List Nat -> Nat
maxl nil = zero
maxl (cons n ns) = max n (maxl ns)
```

- Define the function `zip : {A B : Set} -> List A -> List B`

`-> List (Pair A B)` which converts two lists into a list of pairs of elements. If one of the lists is longer than the other the extra elements are ignored.

```
zip : {A B : Set} -> List A -> List B -> List (Pair A B)
zip (cons a as) (cons b bs) = cons (pair a b) (zip as bs)
zip _ _ = nil
```

- Define the function `intl : {A : Set} -> List A -> List A -> List A` which *interleaves* two lists `xxxx...` and `yyyy....` into `xyxyxy...`. If one of the lists is longer than the other the extra elements are ignored.

```
intl : {A : Set} -> List A -> List A -> List A
intl (cons a as) (cons b bs) = cons a (cons b (intl as bs)
)
intl _ _ = nil
```