We can finally begin counting! Let's start with some preliminary remarks.

We will denote ...

- the initial object **undefined** as 0
- the terminal object () as 1
- functions $a \to b$ as b^a
- tuples (a, b) as $a * b$
- co-products **Either a b** as $a + b$

Additionally, we will denote covariant focus by the term **forall** or $\forall$ interchangeably, and contravariant focus by explicitly stating **exists** or $\exists$.

Why do we even care to count?

Inhabitants

An inhabitant of a type T is any expression $e : T$ of type T. It is of importance to be able to ascertain the number of such inhabitants for a variety of types in pure functional languages since:

- Knowing that a complicated polymorphic type has only a small number of inhabitants (e.g.$\forall a.a \to a$ having only one inhabitant) means that we can partially or fully understand its behavior based on the type alone without referring to a particular implementation.
- Knowing that a T has only one inhabitant $e : T$ allows us to simplify any complicated $complicated\_expression : T$ and replace it with $e : T$.
- If we can find the number of inhabitants, we can likely also enumerate them.
- Proving that a type P has inhabitants is equivalent to proving the corresponding theorem P via the Curry-Howard-Lambek Correspondence.

Q: There might be some confusion regarding what exactly we mean by "inhabitants", e.g. are $1 + 1$ and 2 different inhabitants of type *Int*?

Q: There might be some confusion regarding what exactly we mean by "inhabitants", e.g. are $1 + 1$ and 2 different inhabitants of type *Int*?

A: We consider isomorphism classes, and modulo isomorphisms, they are the same. This is a rather deep topic, so for now we will assume no edge cases, and that if two seemingly equivalent witnesses of type a are not isomorphic, then there exists some sub-object classifier of $f : a \to Bool$ which distinguishes the two.

Exercise

Exercise: How can we show that

$$\text{Either } ()() \cong \text{Bool} \cong \mathbf{2}?$$

We can certainly show it explicitly:

```
to :: Either () () -> Bool
to (Right ()) = True
to (Left ())  = False

from :: Bool -> Either () ()
from True = Right ()
from False = Left ()
```