\newpage

# 0.1 The natural numbers

## 0.1.1 Definition

ℕ denotes the NATURAL NUMBERS.

## 0.1.2

The idea of a "(natural) number" is part of our culture and daily life, but it is not so simple to find a good formal definition. Let us take a first approach by the principle of simple counting:

> We can count a given amount (say apples, matches or people) by taking a UNIT for each item. The collection of this units is a NATURAL NUMBER.

We "implement" or "formalize" this idea in different ways:

- We could lift one of our fingers for each item and our hands make the number. But this way is obviously limited.
- We can make a stroke "|" for each item on a board "☐", and |||||  is a number (of five items).
- If each unit is a "•" and a number is a bag "{...}", then {•, •, •, •, •} is the number (of five items).
- In Haskell, we can use a list of units "()" to represent a number, e.g. `[(),(),(),(),()]` is a number.

## 0.1.3 Definition

The following definitions must be well defined on the natural numbers:

- **zero** : ℕ the ZERO or EMPTY number
- **isZero** : ℕ ⟶ **Bool** a function to tests if a number is zero
- **succ** : ℕ ⟶ ℕ the INCREASE or SUCCESSOR function, that has a successor for every natural number
- **pred** : ℕ ⟶ ℕ the DECREASE or PREDECESSOR function, which returns a well–defined predecessor for each number $n$, but only if $n$ is not empty.

## 0.1.4 comparison

There is another intuitive idea, the principle of comparison:

> For any two natural numbers $n$ and $m$, either $n$ is LOWER THAN $m$, or $n$ and $m$ are EQUAL, or $n$ is BIGGER THAN $m$.

In nice mathematical symbolism, these three cases are also expressed by

$$n < m \qquad n = m \qquad n > m$$

But instead of implementing the relations $<, =, >$ (and variations like $\leq, \geq, \not<, \not\leq$, etc) separately, the following compare function does all that in one step. A call of "**compare**$nm$" returns the answer, which one of the three cases actually holds.

## 0.1.5 Definition

We define the set

**Ordering** $:= \{$ **LT** , **EQ** , **GT** $\}$

In proper Haskell, that is a type declaration

```
data Ordering = LT | EQ | GT
```

## 0.1.6 Definition

We define the (LINEAR) (ORDER) COMPARISON FUNCTION on the natural numbers

```
compare :: ℕ -> ℕ -> Ordering
```

in terms of **isZero** and **pred** as follows:

```
compare n m =
  if (isZero n)
  then if (isZero m)
       then EQ
       else LT
  else if (isZero m)
       then GT
       else compare (pred n) (pred m)
```

## 0.1.7 Definition

A COMPARE FUNCTION on a given type `a` is a function

```
compare :: a -> a -> Ordering
```

where

```
data Ordering = LT | EQ | GT
```

Every such compare function induces the following derived functions

```
(<), (<=), (==), (/=), (>=), (>) :: a -> a -> Bool
```

***** CONTINUE HERE *****

To be a WELL–DEFINED (LINEAR) (ORDER) COMPARE FUNCTION, the function has to satisfy the following properties

- ***** transitivity *****
- ***** CONTINUE HERE *****

## 0.1.8 Definition

```
add :: Naturals -> Naturals -> Naturals
add n m =
  if (isZero n)
  then m
  else succ (add (pred n) m)

mult :: Naturals -> Naturals -> Naturals
mult n m =
  if (isZero n)
  then zero
  else add m (mult (pred n) m)

power :: Naturals -> Naturals -> Naturals
power n m =
  if (isZero n)
  then (succ zero)
  else mult m (power (pred n) m)
```

\newpage

### 0.1.9 natural operations

Together with the concept of a (natural) number comes a couple of obvious operations.

- There is the ADDITION "+" of two numbers:

  $\{\bullet,\bullet,\bullet\} + \{\bullet,\bullet\}$ is $\{\bullet,\bullet,\bullet,\bullet,\bullet\}$

  |||+|| is |||||

- More simple than the addition is the augmentation or SUCCESSOR operation "**succ**" [1]

  **succ** $\{\bullet,\bullet\}$ is $\{\bullet,\bullet,\bullet\}$

- Then there is the inverse PREDECESSOR operation "**pred**", that takes away one item at a time.

  **pred** $\{\bullet,\bullet\}$ is $\{\bullet\}$

  But this may run into trouble. We can only remove items, if the number bag is not empty. With similar constraints, we have a subraction "−".

- Then there is of course the whole zoo of comparison relations, in particular the EQUALITY "==" (we use two "=" to distinguish equality from assigment) and (linear) ORDER "<=". For example,

  ||| == ||| is True

  || <= ||| is True

  $\{\bullet,\bullet\}$ == $\{\bullet\}$ is False

### 0.1.10 (maybe a digression) the COMPARE function

***** introduce Ordering and compare *****

### 0.1.11

***** flaws of the unary representations; 1. emergence of the Roman and 2. the arabic system *****

### 0.1.12

***** the binary or bit numeral representation *****

### 0.1.13 Exercise

Let us try to implement this idea of natural numbers so far as a Haskell module, called `UnaryNumber`. An obvious choice of an item is the `()` and a list of these items is a representation for a natural number. For example, `[(),(),(),(),()]` is a five. Our basic number type is thus

```
type Number = [()]
```

Try to find correct implementations of the functions

```
zero :: Number
succ, pred :: Number -> Number
(+), (*) :: Number -> Number -> Number
compare :: Number -> Number -> Ordering
(==), (<), (<=), (>), (>=) :: Number -> Number -> Bool
```

so that they work as described.

### 0.1.14 Solution of exercise ??

***** CONTINUE HERE *****

```
module UnaryNumber where

  type Number = [()]

  zero :: Number
  zero = []

  succ :: Number -> Number
  succ n = () : n

  pred :: Number -> Number
  pred [] = error "zero number"
  pred n = tail n

  add :: Number -> Number -> Number
  add = ...

  ...
```

### 0.1.15 Digression/Exercise finite ordinals

***** definition of $\omega$; how $<, \in, \subseteq$ coincide *****

## 0.2 Integers

***** go on with Integers, Rational Numbers, Real number etc *****

[1] C–like programming languages use the "++" operator instead of "succ".