



Propositions, Functions, and Predicates

What is a proposition?

- A proposition is a claim made about the state of a world
 - 4 is even
 - 2 is the only even prime number
 - All odd numbers are prime

```
def isEven (n: ℕ) : Prop := n % 2 = 0
```

```
def fourIsEven := isEven 4
```

```
#check fourIsEven
```

```
def isPrime(n: ℕ): Prop := ¬(∃(m: ℕ), m > 1 ∧ m ≠ n ∧ n % m = 0)
```

```
def twoOnlyEvenPrime := ∀(n: ℕ), (isEven n) ∧ (isPrime n) → (n = 2)
```

```
#check twoOnlyEvenPrime
```

```
def allOddNumbersPrime := ∀(n: ℕ), ¬(isEven n) → (isPrime n)
```

```
#check allOddNumbersPrime
```

Propositions don't
have to be true

Proposition components

- True and False: most basic propositions
- Conjunction: whether two or more propositions are all true
- Disjunction: whether any of two or more propositions are true
- Implication: whether one proposition is true whenever another proposition is true
- Negation: whether a proposition is false
- Universal quantification: whether a proposition is true for all instances of a particular type
- Existential quantification: whether a proposition is true for some instance of a particular type

Negations are actually a bit more subtle than this

What is a function?

- A function is a mapping from one or more inputs to an output
 - The mapping must be unique
 - The square root function can return +2 or -2, but it must be consistently defined
 - Technically, it could also return a *pair* of numbers, but that pair would be the output
- Exercise:
 - Describe the function that takes an x input and returns a y input such that the function draws a circle
 - There is no such function!
- Exercise: Give some examples of interesting functions from nat to bool

```
def positive (n: nat) : bool :=  
  if n > 0 then tt else ff
```

```
def uint32 (n: nat) : bool :=  
  if n >= 0 ∧ n < 2^32 then tt else ff
```

Lambda expressions

- Lambda expressions are basically anonymous functions:

```
λ n : nat,
```

```
  (if n > 0 then tt else ff : bool)
```

- Many ways to create the λ symbol, but please don't use `\lamda` 🙄

```
#check λ n : nat, (if n > 0 then tt else ff : bool)
```

```
#check λ n, (if n > 0 then tt else ff)
```

```
#check λ n, n > 0
```

Alternate Formulations

- There are multiple ways to write a function
- The best way depends on taste, but also sometimes on context

```
def positive (n: nat) : bool :=  
  if n > 0 then tt else ff
```

```
def positive' : nat → bool :=  
  λ(n : nat), (if n > 0 then tt else ff : bool)
```

```
def positive''' := λ n, if n > 0 then tt else ff
```

```
def positive''': ℕ → bool :=  
begin  
  exact λ n, if n > 0 then tt else ff  
end
```

Exercise

- Exercise: Give some examples of interesting functions from \mathbb{N} to \mathbb{N}

```
def double (n:  $\mathbb{N}$ ) := 2 * n
```

```
#check double
```

```
#check double 3
```

```
#reduce double 3
```

```
def square (n:  $\mathbb{N}$ ) := n * n
```

```
#check square
```

```
#reduce square 3
```

Functions as types

- Recall that a function from \mathbb{N} to \mathbb{N} is of type $\mathbb{N} \rightarrow \mathbb{N}$
- If a function has a type, can it be an argument to another function?
- Can a function be the return value of another function?
- Discuss

Function as return value

- Consider the following function:

```
def add(x: N) (y: N) := x + y
#reduce add 3 4 = 7
#check add 3
```

This is an example of currying, one of the most common ways to return a value that is a function

- If add takes two arguments, what happens when we only give it one?

```
def add3(y: N) := 3 + y
```

- Does format matter?

```
def add'(x y: N) := x + y
#check add' 3
```

Examples of functions as arguments

```
def compose (f:  $\mathbb{N} \rightarrow \mathbb{N}$ ) (g:  $\mathbb{N} \rightarrow \mathbb{N}$ ) (x:  $\mathbb{N}$ ) :  $\mathbb{N}$  :=  
  f (g x)
```

```
#check compose  
#reduce compose double double 3  
#reduce compose square double 3  
#reduce square (double 3)
```

```
def do_twice (f :  $\mathbb{N} \rightarrow \mathbb{N}$ ) (x:  $\mathbb{N}$ ) :  $\mathbb{N}$  := f (f x)
```

```
#check do_twice  
#reduce do_twice square 3
```

More compose examples

```
def compose (g f :  $\mathbb{N} \rightarrow \mathbb{N}$ ) :  $\mathbb{N} \rightarrow \mathbb{N}$  :=  
   $\lambda k : \mathbb{N}, g (f k)$ 
```

```
def compose' : ( $\mathbb{N} \rightarrow \mathbb{N}$ )  $\rightarrow$  ( $\mathbb{N} \rightarrow \mathbb{N}$ )  $\rightarrow$  ( $\mathbb{N} \rightarrow \mathbb{N}$ ) :=  
   $\lambda g : \mathbb{N} \rightarrow \mathbb{N},$   
     $\lambda f : \mathbb{N} \rightarrow \mathbb{N},$   
       $\lambda k : \mathbb{N},$   
         $g(f(k))$ 
```

```
def square :  $\mathbb{N} \rightarrow \mathbb{N}$  :=  $\lambda k, k^2$ 
```

```
def increment :  $\mathbb{N} \rightarrow \mathbb{N}$  :=  $\lambda k, k + 1$ 
```

```
def square_then_increment :  $\mathbb{N} \rightarrow \mathbb{N}$  :=  
  (compose increment square)
```

```
def increment_then_square :  $\mathbb{N} \rightarrow \mathbb{N}$  :=  
  (compose square increment)
```

```
#eval square_then_increment 5  
#eval increment_then_square 5
```

Alternate do_twice representations

```
def do_twice (f :  $\mathbb{N} \rightarrow \mathbb{N}$ ) (x :  $\mathbb{N}$ ) :  $\mathbb{N}$  := f (f x)
```

```
def do_twice' : ( $\mathbb{N} \rightarrow \mathbb{N}$ )  $\rightarrow$   $\mathbb{N} \rightarrow \mathbb{N}$  :=  $\lambda$  f x, f (f x)
```

```
def do_twice'' : ( $\mathbb{N} \rightarrow \mathbb{N}$ )  $\rightarrow$   $\mathbb{N} \rightarrow \mathbb{N}$  :=  
   $\lambda$  f : ( $\mathbb{N} \rightarrow \mathbb{N}$ ),  
     $\lambda$  (x :  $\mathbb{N}$ ), f (f x)
```

```
theorem dt_eq_dt : do_twice = do_twice'' := rfl
```

What do these types mean? (1 of 2)

- $\mathbb{N} \rightarrow \mathbb{N}$
 - Function that takes a natural number as an argument and returns a natural number
- $\mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$
 - Function that takes a natural number as an argument and returns a function that takes a natural number as an argument and returns a natural number
 - E.g.,

```
def constNatFun (n: ℕ) : ℕ → ℕ := λ k, n
def constNatFun' (n: ℕ) (k: ℕ) : ℕ := n
```
- $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$
 - Same as above

What do these types mean? (2 of 2)

- $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$
 - Function that takes a function (that maps naturals to naturals) as an argument and returns a natural number — hard to imagine a useful example
 - ```
def unuseful(f: N → N): N := 3
```
- $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$ 
  - Function that takes a function as an argument and returns another function
  - E.g., `do_twice`
- $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$ 
  - Function that takes a function as an argument and returns a function that takes a function as an argument and returns another argument
  - E.g., `compose`

# Inception

```
def do_twice' (f: (N → N) → N → N) (x: (N → N)) : (N →
N) := \overline{f} (f x)
```

```
#check do_twice'
```

```
#check do_twice' do_twice
```

```
#eval (do_twice' do_twice) square 2
```

- Why not #reduce?

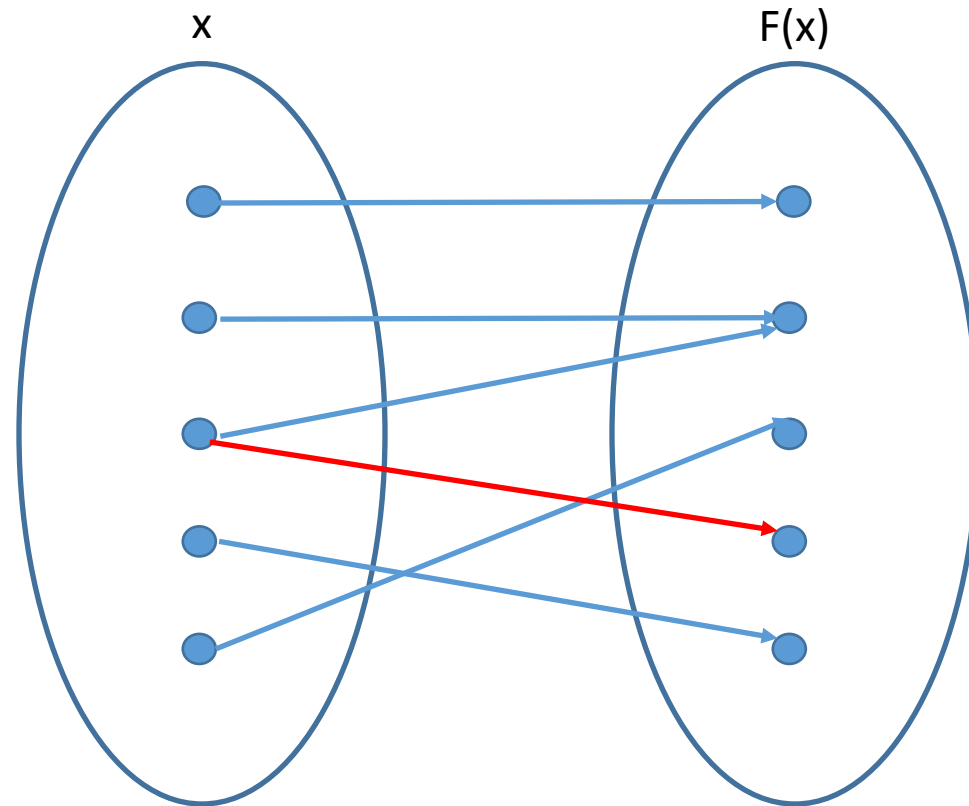
# Important Properties about Functions

- Single-valued (always true of functions)
- Total (all domain values are valid)
- Partial (some domain values are not valid)
- Injective (total and no output is duplicated)
- Surjective (every element of the codomain/range is a valid output of the function)
- Bijective (injective and surjective)
- Extensionality



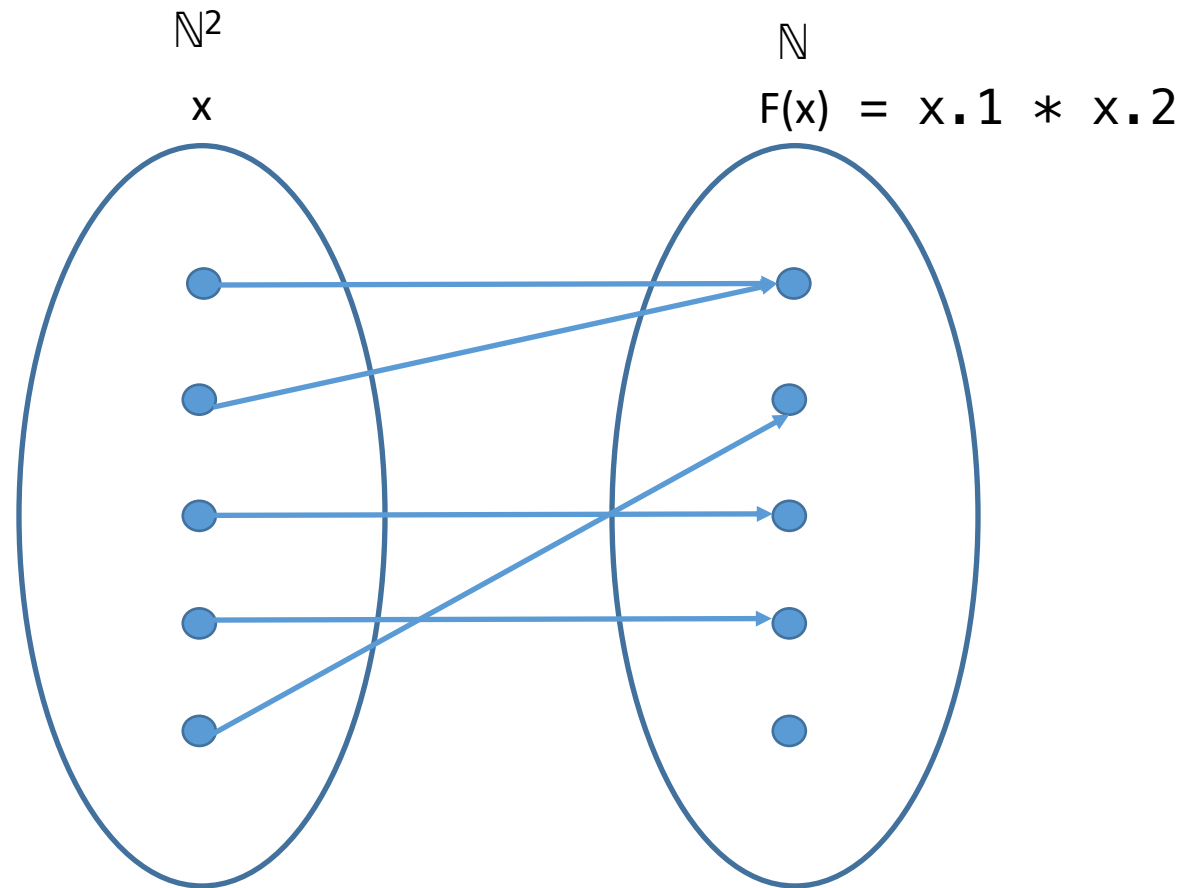
# Single-valued

- The value of  $x$  uniquely determines the value of  $F(x)$



# Total

- All possible values for  $x$  yield a valid result for  $F(x)$



# Partial

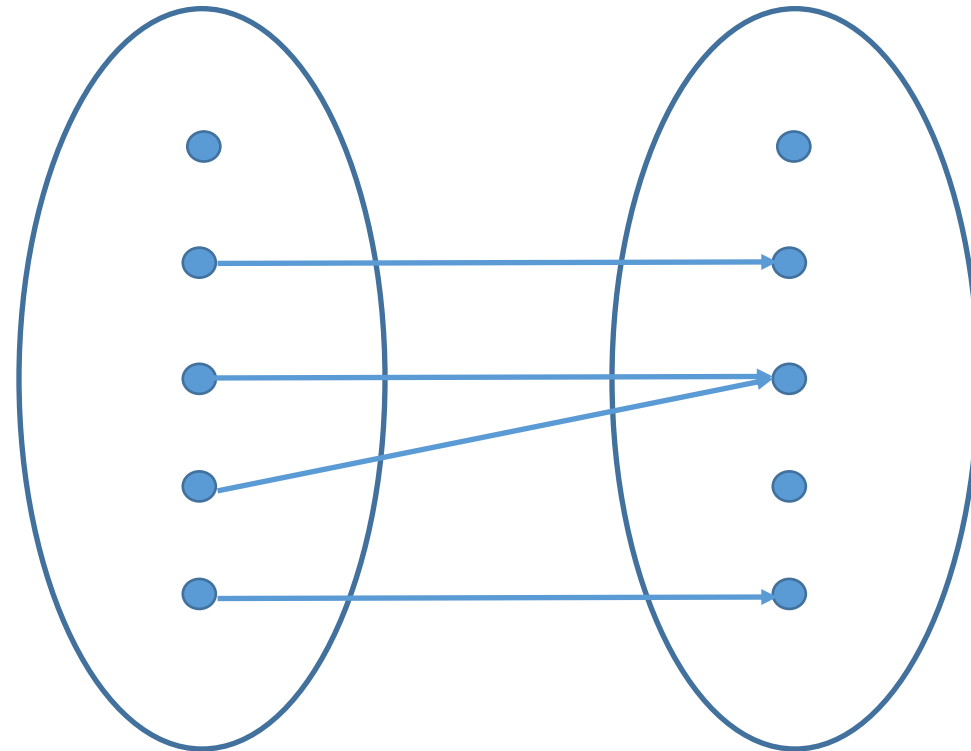
- There are some values of  $x$  (e.g., 0) that do not yield a valid value for  $F(x)$
- Is subtraction over the naturals total or partial?

$$\mathbb{N} \times \{a: \mathbb{N}^2 // a \neq 0\}$$

$x$

$\mathbb{N}$

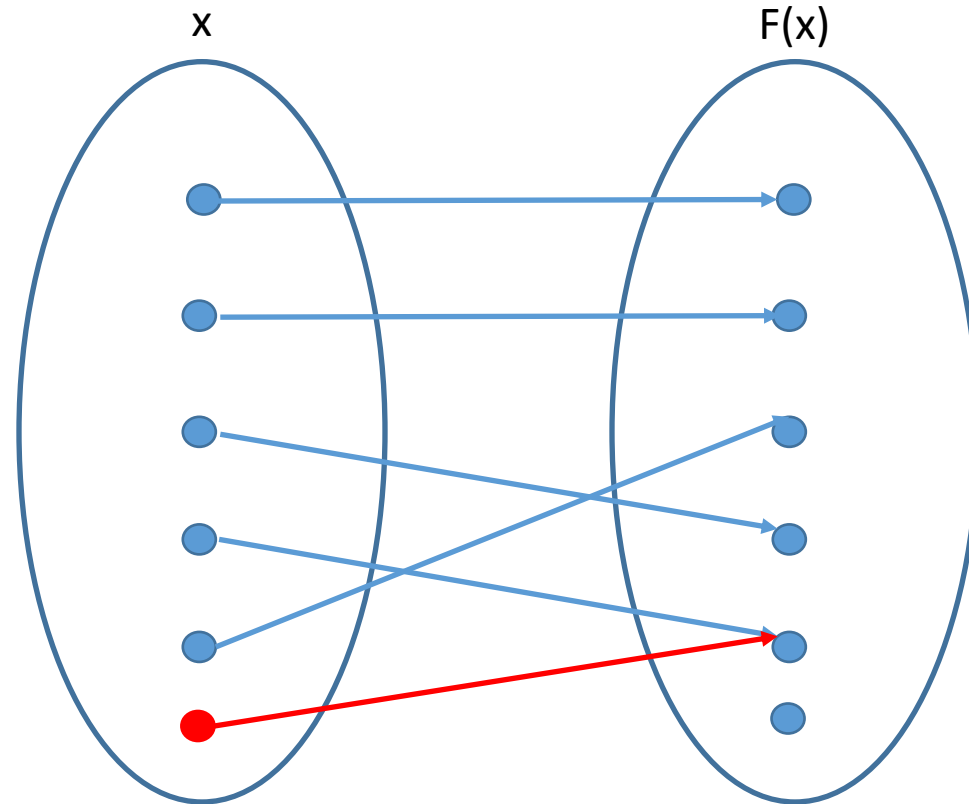
$$F(x) = x.1 / x.2$$



# Injective

- Total and no output is duplicated
- Inverse is valid function

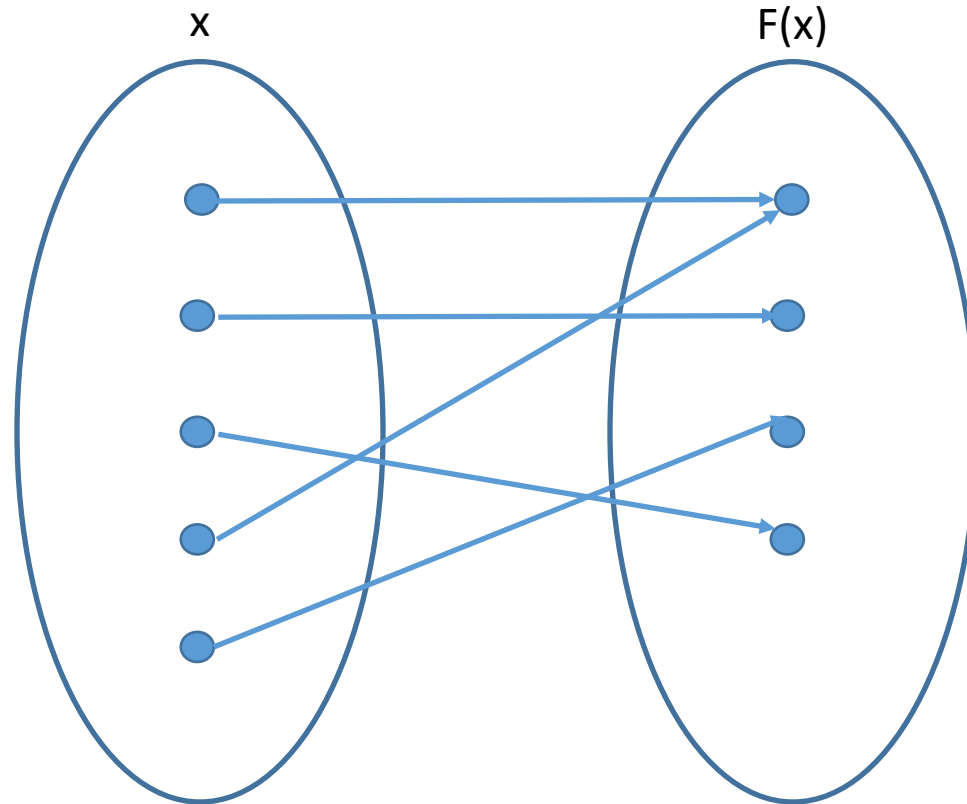
```
def injective(d: Type) (c: Type)
 (F: d → c): Prop :=
 ∀ (x y: d),
 ((F x) = (F y)) → (x = y)
```



# Surjective

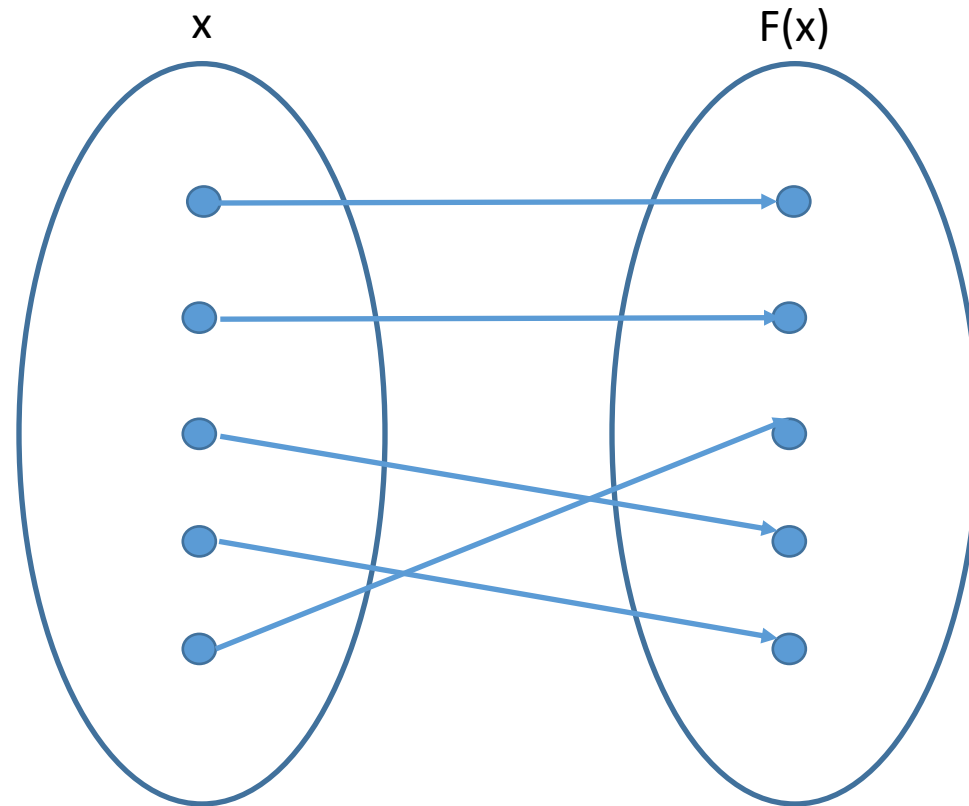
- Codomain of  $F(x)$  is covered

```
def surjective(d: Type) (c: Type)
 (F: d → c): Prop :=
 ∀(y: c),
 ∃(x: d),
 (F x) = y
```



# Bijjective

- Injective and surjective



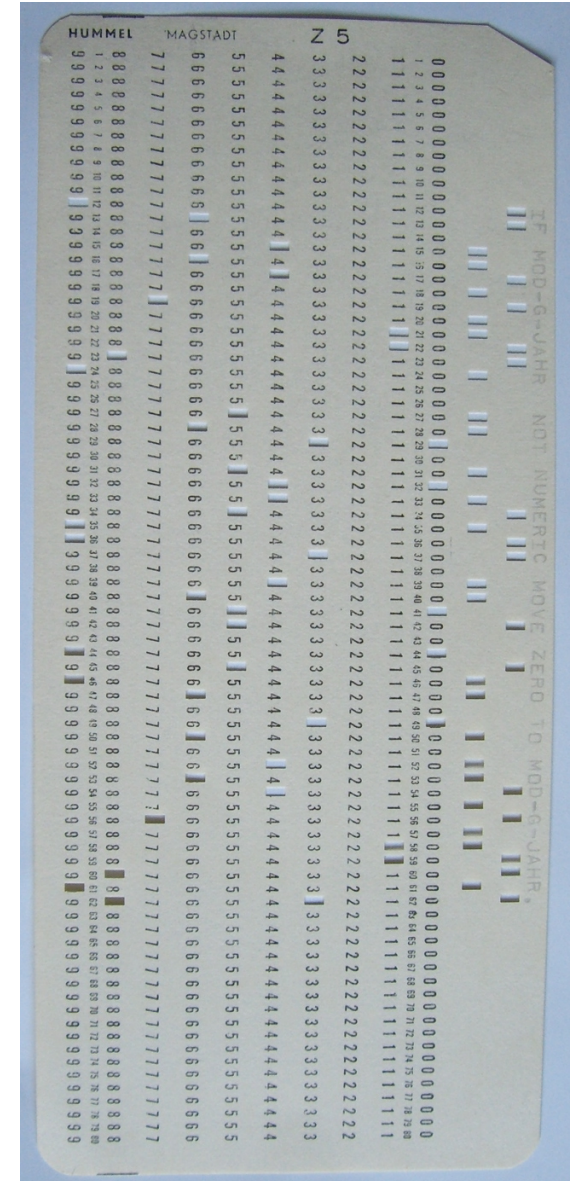
# Extensionality

- If  $f$  and  $g$  have the same domain, and they both map to the same values for all members in that domain, then  $f$  and  $g$  are equivalent

$$\forall (S \ T: \text{Type}) , \ \forall (f \ g: S \rightarrow T) , \ \forall (x: S) , \\ ( (f \ x) = (g \ x) ) \leftrightarrow (f = g)$$

# Story time

- It's important to get your functions correct
- Omitting an “overbar” from an equation in the Mariner 1 software (1962) caused the guidance system to interpret normal movement in the spacecraft as something that needed to be compensated for
- Mariner 1 had to be destroyed 293 seconds into its mission
- How could this have been prevented?





# What is a predicate?

- A predicate is a *function* that returns a *proposition*
  - E.g., `fromCharlottesville(p: Person): Prop := ...`
  - E.g., `onSegment(s: Segment)(p: Point): Prop := ...`
- Note that in Lean, predicates have a return type of `Prop` and not `bool`
  - In other languages (e.g., PVS), this distinction is not meaningful
- Consider the example `fromCharlottesville(p: Person)`
  - `fromCharlottesville(Maya)` is the proposition that Maya is from Charlottesville
  - `fromCharlottesville(Jamal)` is the proposition that Jamal is from Charlottesville
  - `fromCharlottesville(Bao)` is the proposition that Bao is from Charlottesville
  - Not every proposition derived from a predicate will be true

# Examples

- What predicates have we already seen in these slides?

```
def isEven(n: ℕ) : Prop := n % 2 = 0
```

```
def isPrime(n: ℕ) : Prop := ¬(∃ (m: ℕ), m > 1 ∧ n % m = 0)
```

- Technically, these are not predicates:

```
def positive(n: ℕ) : bool :=
 if n > 0 then tt else ff
```

```
def uint32(n: ℕ) : bool :=
 if n >= 0 ∧ n < 2^32 then tt else ff
```

# Exercises

- Write a predicate that takes a number  $n$ , and returns the proposition that  $n$  is positive

```
def positive(n: ℕ): Prop :=
 if n > 0 then true else false
```

```
def positive'(n: ℕ): Prop := n > 0
```

- Write a predicate that takes two numbers,  $n$  and  $m$ , and returns the proposition that  $n$  is evenly divisible by  $m$  (i.e., that  $m$  divides  $n$ )

```
def isDivisible(n m: ℕ): Prop := n % m = 0
```

# Exercises

- What are other properties of natural numbers that could be expressed as predicates?
- Define a predicate that is true for every natural number (i.e., is *trivial*).
  - `def is_absorbed_by_zero(n: ℕ) : Prop := n * 0 = 0`
- Define a predicate that is false for every natural number (i.e., is *unsatisfiable*)
  - `def equals_self_plus_one(n: ℕ) : Prop := n = n + 1`

# Inductive types

```
inductive day : Type
```

```
| Monday
```

```
| Tuesday
```

```
| Wednesday
```

```
| Thursday
```

```
| Friday
```

```
| Saturday
```

```
| Sunday
```

```
#check day.Tuesday
```

```
open day -- no longer need to day. prefix
```

```
#check Tuesday
```

# Predicate for our type

```
def isWeekend : day → Prop :=
 λ d, d = Saturday ∨ d = Sunday
```

- **Proof that Saturday is part of the weekend**

```
theorem satIsWeekend: isWeekend Saturday :=
begin
 unfold isWeekend, -- unfold tactic
 apply or.intro_left, -- backwards reasoning
 apply rfl -- finally, equality
end
```

# Relations

- Predicates can have more than one argument
  - E.g., `onSegment: Segment → Point → Prop := λ (seg: Segment)(pt: Point),...`
- Predicates of multiple arguments can be used to specify properties of *tuples* (e.g., pairs) of values
- Properties of tuples are called *relations*
- Properties of pairs are called *binary relations*
  - E.g., `=`, `<`, `≥`, ... (infix notation)
  - We could also define a predicate `areEqual` that takes two arguments
    - E.g., `areEqual 2 3` (prefix notation)



Fin