- Functions, not procedures, are fundamental to mathematics and programming
- It is useful to look at the definition of functions to put things in perspective
- Axiomatic set theory is a fundamental corner stone of all of modern mathematics
- Set theoretically functions are quite simple, yet powerful things, with some surprising properties compared to the usual algebraic treatment.

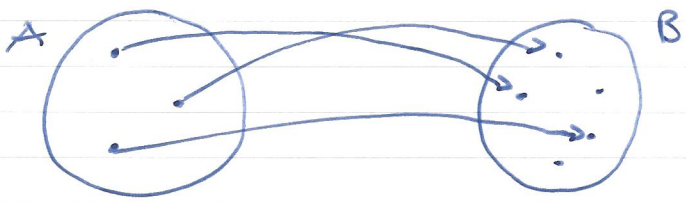
- A function can be defined as a triple: (A, B, f) where A and B are sets and f is a rule or recipe. f can be defined as a set of pairs $\{x \in A, y \in B : (x, y)\}$ for every $x \in A$. The rule f is TRUE for every pair in the set, FALSE otherwise.
- f can also be viewed as a table

A	$f(x) \ni B, x \in A$
a_i	b_i
a_j	b_j
a_k	b_k

- we can think of f as a mapping from A to B:

$$f : A \to B \quad \text{or} \quad y = f(x)$$

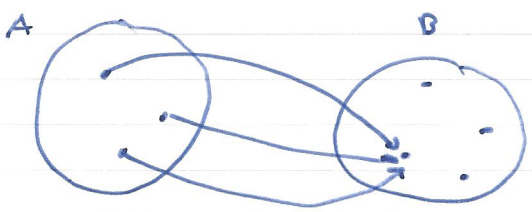
- The set A is called the <u>domain</u> of f and B is called <u>codomain</u>



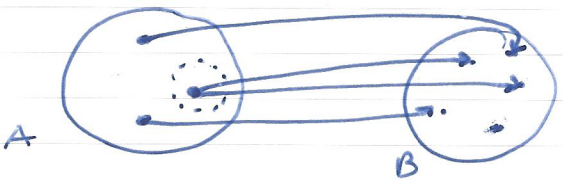
- Observation 1: A function is just a table!
  function = data!

- A function <u>must</u> map every element in its <u>domain</u>, but does <u>not</u> have to cover the entire codomain.

Ex. the constant function $f(x) = c$

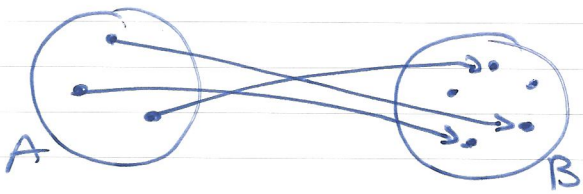


- A function may <u>not</u> map one element in A to <u>more</u> than one element in B!



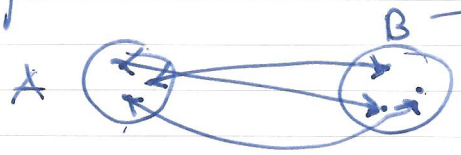
wrong!

- A function which maps elements in a one-to-one relation is <u>injective</u> (or monomorphic)



- A function which covers the whole codomain is <u>surjective</u> (or epimorphic)

- If a function is both injective and surjective it is a <u>bijection</u>. Bijections are <u>invertable</u> $f \leftrightarrow f^{-1}$



- A monotone bijection is order preserving and is an <u>isomorphism!</u>

Examples:

id : $f(x) = x$ (inj, sur, bijective, iso)

const : $f(x) = c$

x^2 : $f(x) = x^2$ $A \subseteq R^+ \Rightarrow$ iso

$A \subseteq R \Rightarrow$ sur (R^+)

- Functions returning functions:

$(A, (B, C, g), f)$

$f : A \to \{(B, C, g)\} = f : A \to (g : B \to C)$

$\Rightarrow f : A \to B \to C$!

A	f(A)	C
a_i	$g_i(B)$	c_i
a_j	$g_j(B)$	c_j

- Function of a function to a value:

$((A, B, g), C, f) = f : (g : A \to B) \to C$

- But wait! How does g get applied? Where does A come from?!

$\Rightarrow$ <u>closure</u>!

Ex. $f(g) = g(5) + 1$

- Function of a function returning a function:

$((A, B, g), (C, D, h), f) = f : (g : A \to B) \to (h : C \to D)$

$\Rightarrow$ functions are easy, it's just table lookup upon table lookup upon ...

- Partial application:

$f : A \to B \to C \quad [x := a_i] \Rightarrow f_{a_i} : B \to C$

$f(x, y) = x^2 + y$

$f(5, y) = 5^2 + y = f(y) = y + 25$

5^2: closure of x; $y + 25$: β-equivalent

- A multivariate function:

  $f : A \to B \to C$

  is really just a table lookup from a pair (A, B) to C:

$(x,y) \in A * B$	$f(x,y)$
(a_i, b_i)	c_i
⋮	⋮

- There is an isomorfism between a function of multiple variables and a function of a single tuple:

  $f : a \to b \to c \iff f : (a,b) \to c$

- Converting <u>from</u> the tupled form is called <u>currying</u>.
- Converting <u>to</u> the tupled form is called <u>uncurrying</u>.

- We now have all the pieces to study λ-functions.

- A λ-function is a normal function, with a slightly special syntax and nomenclature:

$\lambda x.\ t$ ← λ-term (variable (term)) } λ-abstraction

λ-function (term)

$(\lambda x.\ t)\ s = t[x := s]$ (x: bound, t: free)

λ-application (β-reduction)

- All λ-function have the same "name": λ. They are thus "anonymous" (unless we name them explicitly (e.g. let $f = \lambda x.\ t$ in ...))

- We can rename any bound variable, as long as we do it throughout its scope. This is called α-conversion.

- η-conversion: $\lambda x.\ f\ x \Leftrightarrow f$

- Free variables are defined in an outer scope!