Logică și structuri discrete

Funcții

Marius Minea
marius@cs.upt.ro

http://www.cs.upt.ro/~marius/curs/lsd/

23 septembrie 2013

Cum s-ar mai putea numi cursul ?

Matematici discrete

Matematici discrete *cu aplicații*

Bazele informaticii

Matematici discrete cu *programare funcțională*

De ce acest curs ?

Matematicile discrete sunt baza informaticii:

mulțimi, funcții, relații, automate, gramatici, grafuri

logica: limbajul în care formulăm riguros problemele

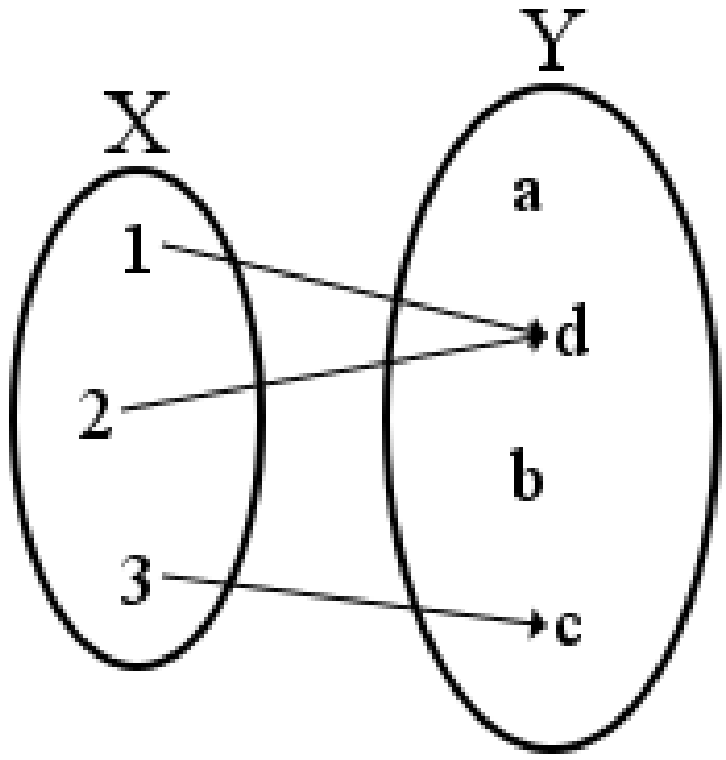
Ce învățăm la acest curs ?

noțiunile de bază din știința calculatoarelor

unde și cum se *aplică* ele, mai ales în *limbajele de programare*
$\Rightarrow$ cum să *programăm mai bine*

Funcții

Fiind date mulțimile A și B, o *funcție* $f : A \to B$ e o asociere care face să corespundă *fiecărui* element din A *un singur* element din B.



Imagine: `http://en.wikipedia.org/wiki/File:Total_function.svg`

Funcții: componentele definiției

Definiția funcției are deci *trei* componente:

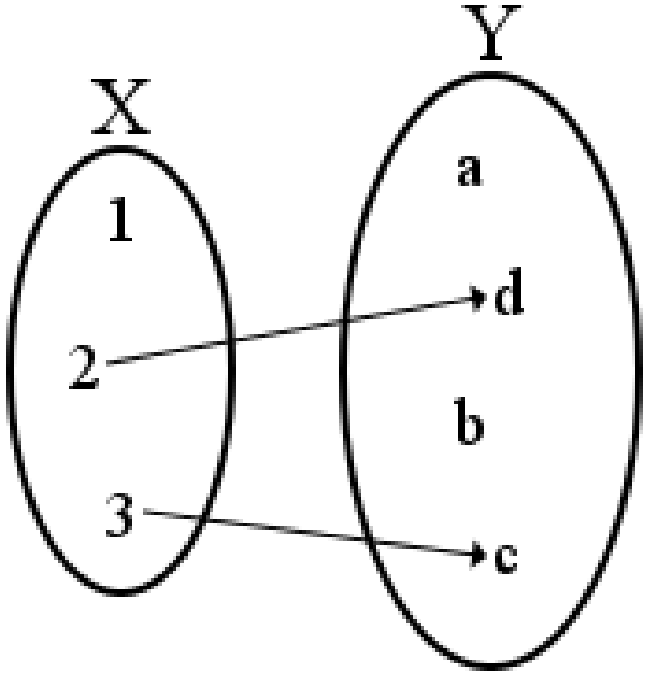
1. *domeniul de definiție* (A)
2. *domeniul de valori* (B)
3. asocierea/corespondența propriu-zisă (legea, regula, etc.)

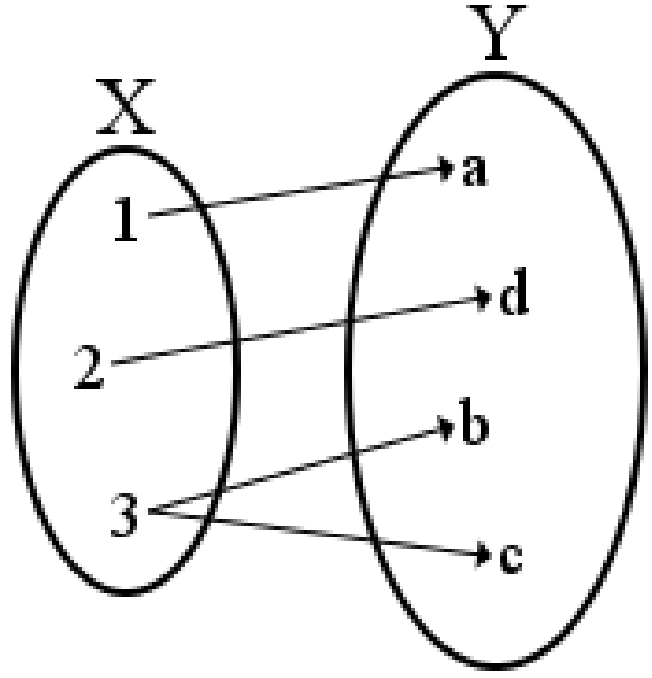
$f : \mathbb{Z} \to \mathbb{Z}$, $f(x) = x + 1$ și $f : \mathbb{R} \to \mathbb{R}$, $f(x) = x + 1$
sunt funcții distincte!

În limbajele de programare, domeniul de definiție și de valori corespund *tipurilor*.

Vom vedea că putem avea funcții care lucrează cu *mai multe* tipuri (*polimorfism*).

Exemple care NU sunt funcții





nu asociază o valoare fiecărui element (fig. 1)

asociază *mai multe* valori unui element (fig. 2)

Imagine: `http://en.wikipedia.org/wiki/File:Partial_function.svg`
`http://en.wikipedia.org/wiki/File:Multivalued_function.svg`

Funcții în limbajele de programare

În limbajele de programare, *funcția* (procedura, metoda)
e entitatea de bază prin care descriem o prelucrare (un calcul).

Acest fapt e cel mai evident în *programarea funcțională*
(și *limbajele de programare funcționale*):

prelucrări complexe prin compunere de funcții simple
(împărțind programul în funcții controlăm complexitatea)

funcțiile pot fi manipulate la fel de simplu ca și alte valori uzuale
(întregi, reali, etc.)

Programare funcțională în ML

De ce ML?

potrivit pentru a ilustra conceptele de matematici discrete

dezvoltat (Robin Milner, Univ. Edinburgh, anii '70) împreună cu un sistem de demonstrare de teoreme (logică matematică)

fundamentat riguros $\Rightarrow$ evită anumite erori (ex. de *tip*)
are concepte care au influențat alte limbaje

E important să învățăm *concepte*, nu doar limbaje!

"A language that doesn't affect the way you think about programming, is not worth knowing."

Alan Perlis

Caml: un dialect de ML, cu interpretorul și compilatorul OCaml
`http://caml.inria.fr`

Funcții în OCaml

`fun x -> x + 1` o *expresie* reprezentând o funcție (fără nume)

```
let f = fun x -> x + 1
```

`let` *leagă identificatorul* (numele) `f` de expresia dată

se scrie mai scurt:

```
let f x = x + 1
```

Interpretorul OCaml răspunde: `val f : int -> int = <fun>`

$\Rightarrow$ matematic: f e o funcție de la întregi la întregi
$\Rightarrow$ în program: `f` e o funcție cu argument de *tip* întreg (`int`)
și rezultat de *tip* întreg (domeniul și codomeniul devin *tipuri*)

În programare, un *tip* de date e o mulțime de valori,
împreună cu niște operații definite pe astfel de valori.

În ML, tipurile pot fi deduse *automat* (*inferență de tip*):
pentru că la `x` se aplică +, compilatorul deduce că `x` e întreg

De unde vin limbajele funcționale ?

lambda-calcul: cel mai simplu limbaj de programare (Church 1932)

Universal: poate exprima orice funcție calculabilă

O expresie în lambda-calcul e fie:

- o *variabilă* — x
- o *funcție* — $\lambda x\,.\,e$ (funcție de variabilă x)
  în ML: `fun x -> e`
- o *evaluare* de functie — $e_1\ e_2$ (funcția e_1 aplicată argumentului e_2)
  la fel în ML: `f 3` fără paranteze
  asociativă la stânga: `f x y` = `(f x) y`

Teoria lui e fundamentală în studiul limbajelor de programare

Funcția în matematică și programare

Implicit, dar *esențial*:
O funcție matematică, apelată repetat (cu *același* argument),
dă *același* rezultat.

E adevărat și în programarea funcțională *pură*.
vom programa cât mai mult în acest fel
e mai ușor de raționat despre programele scrise

În programarea imperativă nu e întotdeauna așa
(atribuiri la variabile)

Cum putem defini o funcție?
printr-o singură formulă (expresie/regulă)
pe cazuri (mai multe variante/expresii, depinzând de o condiție)
individual (explicit) pentru fiecare valoare

Funcții definite pe cazuri

Fie $abs : \mathbb{Z} \to \mathbb{Z}$ $\quad abs(x) = \begin{cases} x & x \geq 0 \\ -x & x < 0 \end{cases}$

Valoarea funcției nu e dată de o singură expresie, ci de una din două expresii diferite (x sau -x), depinzând de o condiție ($x \geq 0$).

```
let abs x = if x >= 0 then x else - x
```

Construcția `if` $expr_1$ `then` $expr_2$ `else` $expr_3$
e o *expresie condițională*

Dacă *evaluarea* $expr_1$ dă o valoare *adevărată*, valoarea întregii expresii e valoarea lui $expr_2$, altfel e valoarea lui $expr_3$.

Obs.: În alte limbaje (C, Java, etc.) `if` lucrează cu *instrucțiuni*.
În ML, `if` lucrează cu *expresii*.
Mai mult, $expr_2$ și $expr_3$ trebuie să aibe *același tip* (ambele să fie întregi, ambele reale, etc.).

Funcții definite prin tipare

Exemplu: conversie note americane

```
let us_to_GPA = function
  | 'A' -> 4
  | 'B' -> 3
  | 'C' -> 2
  | 'D' -> 1
  | _ -> 0
```

Cuvântul cheie `function` introduce o funcție definită folosind *potriviri de tipare* (engl. pattern matching).
Fiecare variantă are forma *tipar* `->` *rezultat*.
În cazul simplu de mai sus, fiecare tipar e o valoare individuală.
Tiparul `_` acoperă orice valoare (care nu a fost deja acoperită)

Limbajul ne *obligă* să acoperim toate variantele (funcția trebuie să fie definită complet) $\Rightarrow$ reduce numărul de erori.

Funcții injective

Def.: O funcție $f : A \to B$ e *injectivă* dacă asociază valori diferite la argumente (valori) diferite.

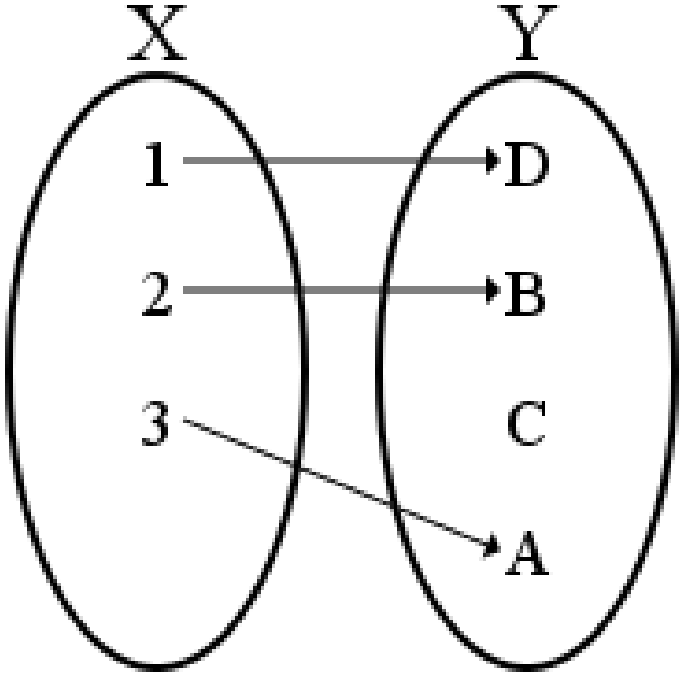
Riguros: pentru orice $x_1, x_2 \in A$, $x_1 \neq x_2 \Rightarrow \ f(x_1) \neq f(x_2)$

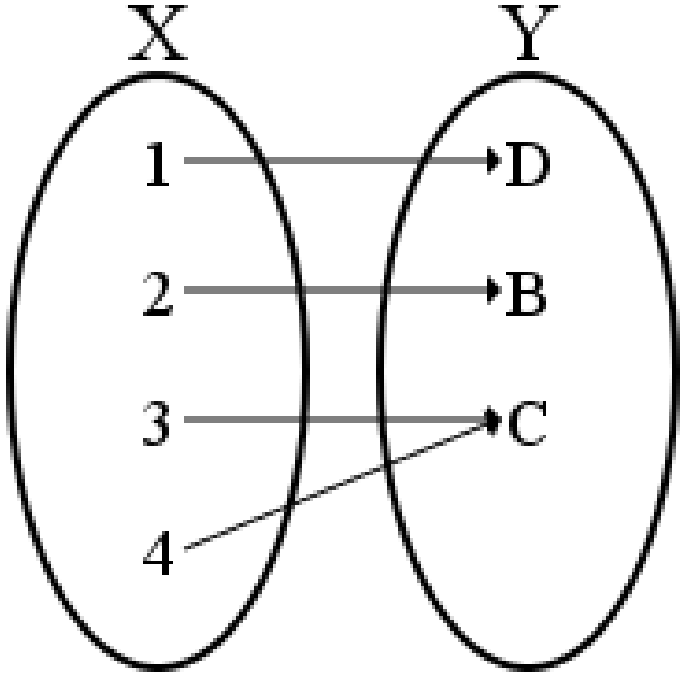
Echivalent: $f(x_1) = f(x_2) \Rightarrow x_1 = x_2$
dacă valorile sunt egale, atunci argumentele sunt egale
(*contrapozitiva* afirmației de mai sus)

În logică, faptul că o afirmație e echivalentă cu contrapozitiva ei ne permite *demonstrație prin reducere la absurd*.

Dacă mulțimile A și B sunt finite, și f e injectivă, atunci $|A| \leq |B|$.

Exemple: funcție injectivă și neinjectivă





Imagine: http://en.wikipedia.org/wiki/File:Injection.svg
http://en.wikipedia.org/wiki/File:Surjection.svg

Funcții injective: aplicații

Să presupunem că vrem să asociem fiecărui student o informație (ex. număr de telefon).

O soluție ar fi să punem studenții într-un tabel.
Dacă putem construi o funcție care din fiecare nume să calculeze numărul intrării de tabel (ex. "Adrian Ionache" -> 15),
găsim ușor informația asociată.

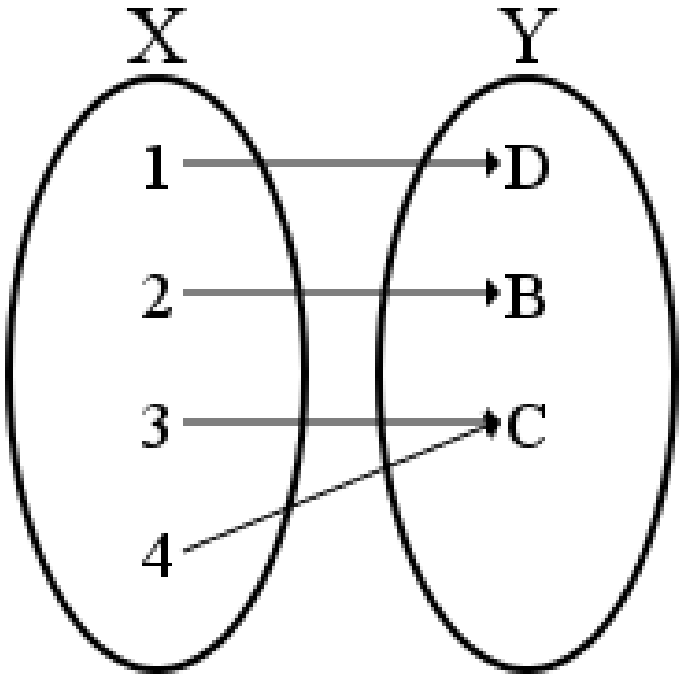
Funcția care dă indicele din tabel trebuie să fie *injectivă*
– altfel se suprapun intrările pentru doi studenți.

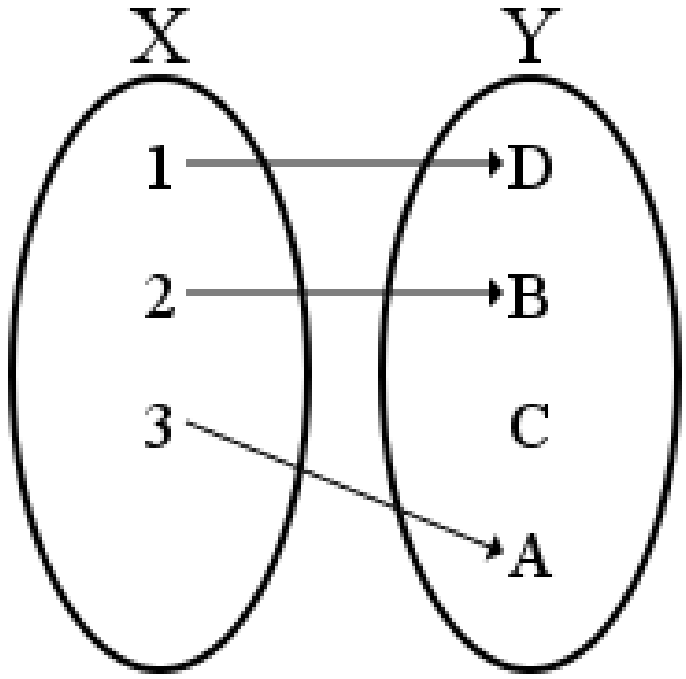
În programe, se folosesc *tabele de dispersie* (engl. hashtables).
Funcțiile (hash functions) care calculează indicele nu pot fi întotdeauna injective (de ce ?)
(numărul posibil de șiruri e mai mare decât tabelul).
Dar dorim o probabilitate cât mai mică de *coliziuni* (argumente diferite cu valori egale), și trebuie tratat cazul coliziunilor.

Funcții surjective

O funcție $f : A \to B$ e *surjectivă* dacă pentru fiecare $y \in B$ există un $x \in A$ cu $f(x) = y$.

Exemple: funcție surjectivă și nesurjectivă





Imagine: http://en.wikipedia.org/wiki/File:Surjection.svg

Imagine: http://en.wikipedia.org/wiki/File:Injection.svg

Funcții surjective: discuție

Dacă A și B sunt mulțimie finite, și $f : A \to B$ e surjectivă, atunci $|A| \geq |B|$.

Putem transforma o funcție ne-surjectivă într-una surjectivă prin restrângerea domeniului de valori:

$f_1 : \mathbb{R} \to \mathbb{R}$, $f_1(x) = x^2$ nu e surjectivă,
dar $f_2 : \mathbb{R} \to [0, \infty)$ (restrânsă la valori nenegative) este.

În programare, e util să definim funcția cu tipul rezultatului cât mai precis (dacă e posibil, surjectivă).
Astfel, când raționăm despre program, știm deja din tipul funcției ce valori poate returna, fără a trebui să-i analizăm codul.

Exercițiu: cum putem defini funcția semn ?

Câte funcții există de la A la B ?

Dacă A și B sunt mulțimi finite există $|B|^{|A|}$ funcții de la A la B.

Notație: $|A|$ = cardinalul lui A (numărul de elemente)

Demonstrație: prin *inducție matematică*

Principiul inducției matematice

Dacă o propoziție $P(n)$ depinde de un număr natural n, și

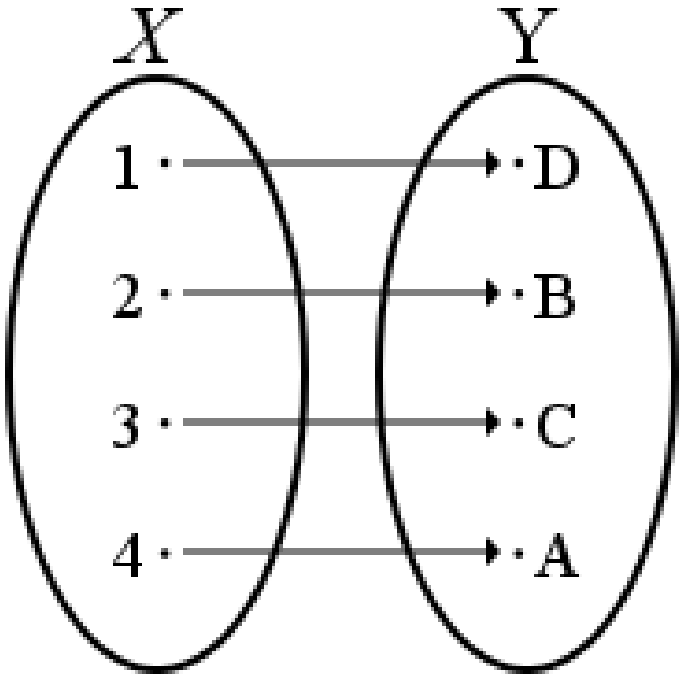
1) (*cazul de bază*) P(0) e adevărată

2) (*pasul inductiv*) pentru orice $n \geq 0$, $P(n) \Rightarrow P(n+1)$

atunci $P(n)$ e adevărată pentru orice n.

Funcții bijective

O funcție care e injectivă și surjectivă se numește *bijectivă*.



Dacă mulțimile A și B sunt finite, și f e bijectivă, atunci $|A| = |B|$.

Imagine: `http://en.wikipedia.org/wiki/File:Bijection.svg`

Imagine și preimagine

Fie $f : A \to B$.

Dacă $S \subseteq A$, mulțimea elementelor $f(x)$ cu $x \in S$ se numește *imaginea* lui S prin f, notată $f(S)$.

Dacă $T \subseteq B$, mulțimea elementelor x cu $f(x) \in T$ se numește *preimaginea* lui T prin f, notată $f^{-1}(T)$.

În general, $f^{-1}(f(S)) \supseteq S$ (aplicând întâi funcția, și apoi revenind la preimagine, se pierde precizie).

Funcții inversabile

O funcție $f : A \to B$ e inversabilă dacă pentru orice $y \in B$ există o *unică* valoare $x \in A$ cu $f(x) = y$.

O funcție e inversabilă dacă și numai dacă e bijectivă:
pentru orice $y \in B$ *există* $x \in A$ cu $f(x) = y$: f e surjectivă
x cu $f(x) = y$ e *unic*: f e injectivă.

Inversa lui $f : A \to B$ se notează $f^{-1} : B \to A$.

Cât de ușor se poate calcula inversa?

Faptul că o funcție e inversabilă nu înseamnă neapărat că inversa e *ușor calculabilă*.

Considerăm mulțimea $\mathbb{Z}_p^*$ a resturilor nenule modulo p, cu p prim. Ea formează un *grup multiplicativ* cu operația de înmulțire mod p.

Teorema lui Fermat: $a^{p-1} = 1 \mod p$ pentru orice $a \in \mathbb{Z}_p^*$.

Se mai știe că dacă p e prim, grupul $\mathbb{Z}_p^*$ are cel puțin un *generator*, adică un element g astfel încât șirul $g, g^2, g^3, ..., g^{p-1}$ parcurge toată mulțimea $\mathbb{Z}_p^*$.

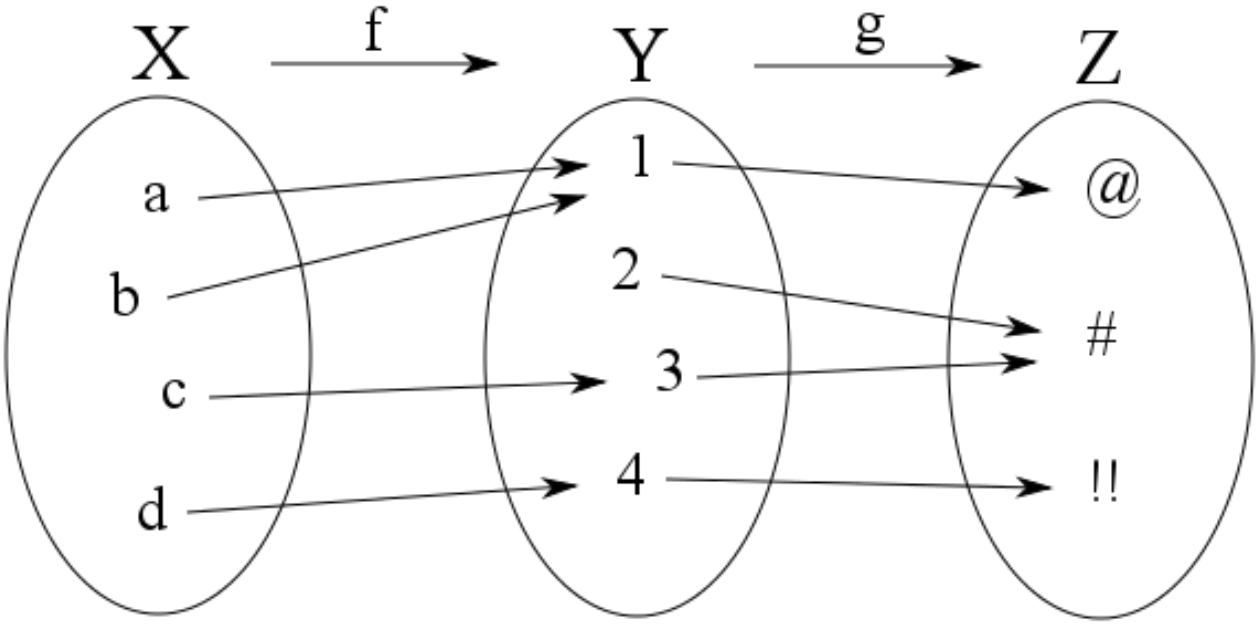
Înseamnă că funcția $f : \mathbb{Z}_p^* \to \mathbb{Z}_p^*$, $f(x) = g^x \mod p$ e o *bijecție* (și inversabilă).

Nu se cunoaște însă un mod eficient de a o inversa (problema logaritmului discret) $\Rightarrow$ e folosită în criptografie.

Compunerea funcțiilor

Fie functiile $f : A \to B$ și $g : B \to C$. Compunerea lor este funcția $g \circ f : A \to C$, $(g \circ f)(x) = g(f(x))$.

Compunerea ne permite să construim funcții mai complicate din funcții mai simple.



Imagine: `http://en.wikipedia.org/wiki/File:Compufun.svg`

Proprietăți are compunerii funcțiilor

Compunerea a două funcții e asociativă:

$$(f \circ g) \circ h = f \circ (g \circ h)$$

Compunerea a două funcții nu e neapărat comutativă

$$f \circ g \neq g \circ f \qquad \text{(în general)}$$

Compunerea și inversa

Pentru orice funcție inversabilă $f : A \to B$ avem

$$f \circ f^{-1} = id_B \quad \text{și} \quad f^{-1} \circ f = id_A$$

unde $id_A : A \to A$, $id_A(x) = x$ e *funcția identitate*