Logică și structuri discrete

Funcții

Casandra Holotescu

casandra@cs.upt.ro

https://tinyurl.com/lecturesLSD

Ce învățăm la acest curs?

Discret vs. continuu

Nu studiem domeniul *continuu*
numere reale, infinitezimale, limite, ecuații diferențiale
vezi: analiză matematică

Studiem noțiuni/obiecte care iau *valori distincte*, *discrete*
(întregi, valori logice, liste, relații, arbori, grafuri, etc.)

Logică și structuri discrete, sau ...

Matematici discrete *cu aplicații*

folosind *programare funcțională*

Bazele informaticii

noțiunile de bază din știința calculatoarelor

unde și cum se *aplică*

$\Rightarrow$ cum să *programăm mai bine*

Programare funcțională în ML

Vom lucra cu un limbaj în care noțiunea fundamentală e *funcția*

- ilustrează concepte de matematici discrete (liste, mulțimi, etc.)
- *concis* (în câteva linii de cod se pot face multe)
- *fundamentat riguros* $\Rightarrow$ ajută să evităm erori

Programare funcțională în ML

Vom lucra cu un limbaj în care noțiunea fundamentală e *funcția*

- ilustrează concepte de matematici discrete (liste, mulțimi, etc.)
- *concis* (în câteva linii de cod se pot face multe)
- *fundamentat riguros* $\Rightarrow$ ajută să evităm erori

Programarea funcțională

- *complementară* programării imperative (în C)
- vom discuta ce e *comun*, și ce e *diferit* (și de ce)

Caml: un dialect de ML, cu interpretorul și compilatorul OCaml

`http://ocaml.org`

E relevantă programarea funcțională?

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

Alan Perlis

Conceptele din programarea funcțională au influențat alte limbaje:
JavaScript, Python, Scala; F# (.NET) e foarte similar cu ML

E relevantă programarea funcțională?

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

Alan Perlis

Conceptele din programarea funcțională au influențat alte limbaje:
JavaScript, Python, Scala; F# (.NET) e foarte similar cu ML

Exemplu: adoptarea funcțiilor anonime (lambda-expresii)
1930 λ-calcul (Alonzo Church) – pur teoretic
1958: LISP (John McCarthy)
1973: ML (Robin Milner)
2007: C# v3.0
2011: C++11
2014: Java 8

OK, să-i dăm drumul!

Cum demonstrăm o afirmație?

Demonstrația prin reducere la absurd

Contrapozitiva unei afirmații:
negăm premisa și concluzia, și le inversăm.

afirmația	$P \Rightarrow Q$
are contrapozitiva	$\neg Q \Rightarrow \neg P$

În logică, o afirmație e echivalentă cu contrapozitiva ei.

$$P \Rightarrow Q \quad \Leftrightarrow \quad \neg Q \Rightarrow \neg P$$

Demonstrația prin reducere la absurd

Contrapozitiva unei afirmații:
negăm premisa și concluzia, și le inversăm.

afirmația $P \Rightarrow Q$
are contrapozitiva $\neg Q \Rightarrow \neg P$

În logică, o afirmație e echivalentă cu contrapozitiva ei.

$P \Rightarrow Q \quad \Leftrightarrow \quad \neg Q \Rightarrow \neg P$

Demonstrația prin reducere la absurd
– presupunem concluzia falsă

Demonstrația prin reducere la absurd

Contrapozitiva unei afirmații:
negăm premisa și concluzia, și le inversăm.

afirmația $P \Rightarrow Q$
are contrapozitiva $\neg Q \Rightarrow \neg P$

În logică, o afirmație e echivalentă cu contrapozitiva ei.

$$P \Rightarrow Q \quad \Leftrightarrow \quad \neg Q \Rightarrow \neg P$$

Demonstrația prin reducere la absurd
– presupunem concluzia falsă
– arătăm că atunci premisa e falsă $\Rightarrow$ *absurd* (e adevărată)

Demonstrația prin reducere la absurd

Contrapozitiva unei afirmații:
negăm premisa și concluzia, și le inversăm.

afirmația $P \Rightarrow Q$
are contrapozitiva $\neg Q \Rightarrow \neg P$

În logică, o afirmație e echivalentă cu contrapozitiva ei.

$$P \Rightarrow Q \quad \Leftrightarrow \quad \neg Q \Rightarrow \neg P$$

Demonstrația prin reducere la absurd
– presupunem concluzia falsă
– arătăm că atunci premisa e falsă $\Rightarrow$ *absurd* (e adevărată)
– deci concluzia nu poate fi falsă $\Rightarrow$ e adevărată

Demonstrația prin inducție matematică

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

Demonstrația prin inducție matematică

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 $n \geq 0$

$$P(n) \Rightarrow P(n+1)$$

Demonstrația prin inducție matematică

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.

Cum arătăm că o afirmație (universală) e falsă?

E suficient să găsim un *contraexemplu*.

Exemplu:

dacă o propoziție $Q(n)$ depinde de un număr natural n
și pentru $n = 3$, $Q(3)$ e falsă $\Rightarrow Q(n)$ e falsă

Mulțimi – scurt intro

Ce sunt mulțimile?

Definiție informală:

O *mulțime* e o *colecție* de obiecte numite *elementele* mulțimii.

Ce sunt mulțimile?

Definiție informală:

O *mulțime* e o *colecție* de obiecte numite *elementele* mulțimii.

Două noțiuni distincte: *element* și *mulțime*

$x \in S$: elementul x *aparține* mulțimii S
$x \notin S$: elementul x *nu aparține* mulțimii S

Ce sunt mulțimile?

Definiție informală:

O *mulțime* e o *colecție* de obiecte numite *elementele* mulțimii.

Două noțiuni distincte: *element* și *mulțime*

$x \in S$: elementul x *aparține* mulțimii S
$x \notin S$: elementul x *nu aparține* mulțimii S

Ordinea elementelor *nu* contează $\{1, 2, 3\} = \{1, 3, 2\}$
Un element *nu* apare de mai multe ori $\cancel{\{1, 2, 3, 2\}}$

Submulțimi

A e o *submulțime* a lui B: $\quad A \subseteq B$
dacă fiecare element al lui A e și un element al lui B.

A e o *submulțime proprie* a lui B: $\quad A \subset B$
dacă $A \subseteq B$ și există (măcar) un element $x \in B$ astfel ca $x \notin A$.

Submulțimi

A e o *submulțime* a lui B: $\quad A \subseteq B$
dacă fiecare element al lui A e și un element al lui B.

A e o *submulțime proprie* a lui B: $\quad A \subset B$
dacă $A \subseteq B$ și există (măcar) un element $x \in B$ astfel ca $x \notin A$.

Obs. Ca să demonstrăm $A \not\subseteq B$ e suficient să găsim un element $x \in A$ pentru care $x \notin B$.

Dacă $A \subseteq B$ și $B \subseteq A$, atunci $A = B$ (mulțimile sunt egale)

Cardinalul unei mulțimi

Cardinalul (cardinalitatea) unei mulțimi A e numărul de elemente al mulțimii.

Cardinalul unei mulțimi A se notează $|A|$.

Cardinalul unei mulțimi

Cardinalul (cardinalitatea) unei mulțimi A e numărul de elemente al mulțimii.

Cardinalul unei mulțimi A se notează $|A|$.

Putem avea mulțimi *finite*: $|\{1, 2, 3, 4, 5\}| = 5$
sau *infinite*: $\mathbb{N}$, $\mathbb{R}$, etc.

Care e cardinalul unei mulțimi infinite? $|\mathbb{N}| = |\mathbb{R}| = \infty$?

Cardinalul unei mulțimi

Cardinalul (cardinalitatea) unei mulțimi A e numărul de elemente al mulțimii.

Cardinalul unei mulțimi A se notează $|A|$.

Putem avea mulțimi *finite*: $|\{1,2,3,4,5\}| = 5$
sau *infinite*: $\mathbb{N}$, $\mathbb{R}$, etc.

Care e cardinalul unei mulțimi infinite? $|\mathbb{N}| = |\mathbb{R}| = \infty$?

Nu:

$|\mathbb{N}| = \aleph_0$ — $\aleph_0$ – cel mai mic cardinal infinit

$|\mathbb{R}| = 2^{\aleph_0}$

Tupluri

Un *n-tuplu* e un șir de n elemente $(x_1, x_2, \ldots, x_n)$

Proprietăți:
elementele nu sunt neapărat distincte
ordinea elementelor în tuplu contează

Cazuri particulare:
pereche (a, b),
triplet (x, y, z), etc.

Produs cartezian

Produsul cartezian a două mulțimi e mulțimea perechilor

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

Produsul cartezian a n mulțimi e mulțimea $n-tuplelor$

$$A_1 \times A_2 \times \ldots \times A_n = \{(x_1, x_2, \ldots, x_n) \mid x_i \in A_i, 1 \leq i \leq n\}$$

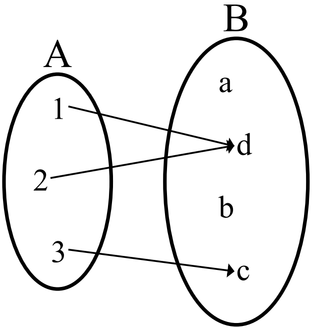
Dacă mulțimile sunt finite, atunci

$$|A_1 \times A_2 \times \ldots \times A_n| = |A_1| \cdot |A_2| \cdot \ldots \cdot |A_n|$$

Funcții – aspect matematic

Funcții

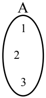
Fiind date mulțimile A și B, o *funcție* $f : A \to B$ e o asociere prin care *fiecărui* element din A îi corespunde *un singur* element din B.



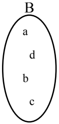
Imagine: `http://en.wikipedia.org/wiki/File:Total_function.svg`

O funcție e definită prin trei componente

1. *domeniul de definiție*

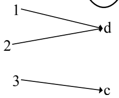


2. *domeniul de valori* (codomeniul)



3. asocierea/corespondența propriu-zisă

   (legea, regula de asociere)

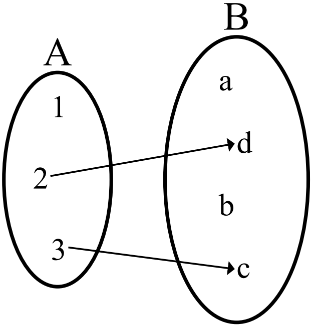


$f : \mathbb{Z} \to \mathbb{Z}$, $f(x) = x + 1$ și
$f : \mathbb{R} \to \mathbb{R}$, $f(x) = x + 1$

sunt funcții distincte!

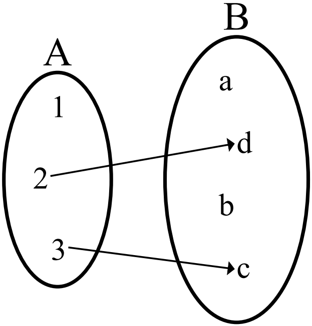
Exemple care NU sunt funcții

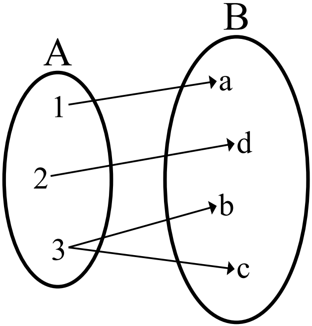
nu asociază o valoare *fiecărui* element



Exemple care NU sunt funcții

nu asociază o valoare *fiecărui* element





asociază *mai multe* valori unui element

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

O definiție alternativă

O funcție $f : A \to B$ este o *mulțime* $f \subseteq A \times B$ a. î. pentru *fiecare* element $a \in A$ există un *unic* element $b \in B$ a. î. $(a, b) \in f$.

Notăm această alegere unică a lui b cu $f(a)$.

O definiție alternativă

O funcție $f : A \to B$ este o *mulțime* $f \subseteq A \times B$ a. î. pentru *fiecare* element $a \in A$ există un *unic* element $b \in B$ a. î. $(a, b) \in f$.

Notăm această alegere unică a lui b cu $f(a)$.

Consecință: putem avea o funcție $f : \emptyset \to \mathbb{N}$?

O definiție alternativă

O funcție $f : A \to B$ este o *mulțime* $f \subseteq A \times B$ a. î. pentru *fiecare* element $a \in A$ există un *unic* element $b \in B$ a. î. $(a, b) \in f$.

Notăm această alegere unică a lui b cu $f(a)$.

Consecință: putem avea o funcție $f : \emptyset \to \mathbb{N}$?

Da: $f \subseteq \emptyset \times \mathbb{N} \Leftrightarrow f \subseteq \emptyset \Leftrightarrow f = \emptyset$

pentru orice $a \in \emptyset$ există un unic $b \in \mathbb{N}$ a. î. $(a, b) \in f$ (adevărat)

$f = \emptyset$ este *funcția vidă*

Proprietăți ale funcțiilor

Funcții injective

O funcție $f : A \to B$ e *injectivă* dacă
pentru orice $x_1, x_2 \in A$, $x_1 \neq x_2 \Rightarrow \; f(x_1) \neq f(x_2)$

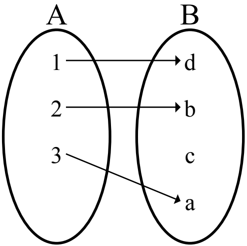
(asociază *valori diferite* la *argumente diferite*)

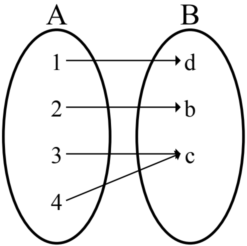
Funcții injective

O funcție $f : A \to B$ e *injectivă* dacă
pentru orice $x_1, x_2 \in A$, $x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2)$

(asociază *valori diferite* la *argumente diferite*)

Exemple: funcție injectivă și neinjectivă





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

Funcții injective (cont.)

În locul condiției $x_1, x_2 \in A$, $x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2)$
putem scrie echivalent:

$$f(x_1) = f(x_2) \Rightarrow x_1 = x_2$$

(dacă valorile sunt egale, atunci argumentele sunt egale)

Funcții injective (cont.)

În locul condiției $x_1, x_2 \in A$, $x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2)$
putem scrie echivalent:

$$f(x_1) = f(x_2) \Rightarrow x_1 = x_2$$

(dacă valorile sunt egale, atunci argumentele sunt egale)

E totuna cu $x_1, x_2 \in A$, $x_1 = x_2 \Rightarrow f(x_1) = f(x_2)$?

Funcții injective (cont.)

În locul condiției $x_1, x_2 \in A,\ x_1 \neq x_2 \Rightarrow\ f(x_1) \neq f(x_2)$
putem scrie echivalent:

$$f(x_1) = f(x_2) \Rightarrow x_1 = x_2$$

(dacă valorile sunt egale, atunci argumentele sunt egale)

E totuna cu $x_1, x_2 \in A,\ x_1 = x_2 \Rightarrow\ f(x_1) = f(x_2)$?

Nu! *Orice funcție* ia aceeași valoare pentru argumente egale!
(e o proprietate de bază a egalității și substituției).

Proprietăți ale funcțiilor injective

Dacă $f : A \to B$ și f e injectivă, atunci $|A| \leq |B|$.

Proprietăți ale funcțiilor injective

Dacă $f : A \to B$ și f e injectivă, atunci $|A| \leq |B|$.

Nu și invers!!

(Pentru orice mulțime A a.î. $|A| > 1$ putem construi f să ducă două elemente din A în aceeași valoare din B)

Proprietăți ale funcțiilor injective

Dacă $f : A \to B$ și f e injectivă, atunci $|A| \leq |B|$.

Nu și invers!!

(Pentru orice mulțime A a.î. $|A| > 1$ putem construi f să ducă două elemente din A în aceeași valoare din B)

Demonstrația: prin reducere la absurd *și* inducție.

1. construim *contrapozitiva*:
   dacă $|A| > |B|$, atunci $f : A \to B$ nu e injectivă
2. prin *inducție* după n, unde $n = |B|$:
   $|A| > |B| = n \Rightarrow f : A \to B$ nu poate fi injectivă.

Demonstrație prin inducție

$|A| > |B| = n \Rightarrow f : A \rightarrow B$ nu poate fi injectivă

Cazul de bază: $n = 1$, $B = \{b_1\}$.

$$|A| > |B| \Rightarrow |A| \geq 2$$

$$|A| \geq 2 \Rightarrow f(a_1) = f(a_2) = b_1 \text{ (unica posibilitate)}$$

deci f nu e injectivă.

Cazul inductiv: pres. $P(n)$ adevărat, dem. $P(n) \Rightarrow P(n+1)$

Cazul inductiv: pres. $P(n)$ adevărat, dem. $P(n) \Rightarrow P(n+1)$

fie $|B| = n+1$ și $b_{n+1} \in B$.

dacă $\exists a_1, a_2$ din A, $f(a_1) = f(a_2) = b_{n+1} \Rightarrow$ f nu e injectivă.

Cazul inductiv: pres. $P(n)$ adevărat, dem. $P(n) \Rightarrow P(n+1)$

fie $|B| = n + 1$ și $b_{n+1} \in B$.

dacă $\exists a_1, a_2$ din A, $f(a_1) = f(a_2) = b_{n+1} \Rightarrow$ f nu e injectivă.

altfel, dacă $\exists$ un unic $a_1 \in A$, $f(a_1) = b_{n+1}$
putem elimina a_1 din A și b_{n+1} din B

Cazul inductiv: pres. $P(n)$ adevărat, dem. $P(n) \Rightarrow P(n+1)$

fie $|B| = n + 1$ și $b_{n+1} \in B$.

dacă $\exists a_1, a_2$ din A, $f(a_1) = f(a_2) = b_{n+1} \Rightarrow$ f nu e injectivă.

altfel, dacă $\exists$ un unic $a_1 \in A$, $f(a_1) = b_{n+1}$
putem elimina a_1 din A și b_{n+1} din B

fie $A' = A \setminus \{a_1\}$ și $B' = B \setminus \{b_{n+1}\}$ atunci $|A'| > |B'| = n$

Cazul inductiv: pres. $P(n)$ adevărat, dem. $P(n) \Rightarrow P(n+1)$

fie $|B| = n + 1$ și $b_{n+1} \in B$.

dacă $\exists a_1, a_2$ din A, $f(a_1) = f(a_2) = b_{n+1} \Rightarrow$ f nu e injectivă.

altfel, dacă $\exists$ un unic $a_1 \in A$, $f(a_1) = b_{n+1}$
putem elimina a_1 din A și b_{n+1} din B

fie $A' = A \setminus \{a_1\}$ și $B' = B \setminus \{b_{n+1}\}$ atunci $|A'| > |B'| = n$

$P(n)$: $|A'| > |B'| = n \Rightarrow f : A' \to B'$ nu e injectivă

$\Rightarrow \exists$ două elem. din A' cu valori egale pentru f.

deci $P(n) \Rightarrow P(n+1)$

Cazul inductiv: pres. $P(n)$ adevărat, dem. $P(n) \Rightarrow P(n+1)$

fie $|B| = n + 1$ și $b_{n+1} \in B$.

dacă $\exists a_1, a_2$ din A, $f(a_1) = f(a_2) = b_{n+1} \Rightarrow$ f nu e injectivă.

altfel, dacă $\exists$ un unic $a_1 \in A$, $f(a_1) = b_{n+1}$
putem elimina a_1 din A și b_{n+1} din B

fie $A' = A \setminus \{a_1\}$ și $B' = B \setminus \{b_{n+1}\}$ atunci $|A'| > |B'| = n$

$P(n)$: $|A'| > |B'| = n \Rightarrow f : A' \to B'$ nu e injectivă
$\Rightarrow \exists$ două elem. din A' cu valori egale pentru f.
deci $P(n) \Rightarrow P(n+1)$

(*Principiul lui Dirichlet*: dacă împărțim $n + 1$ obiecte în n categorii există cel puțin o categorie cu mai mult de un obiect)

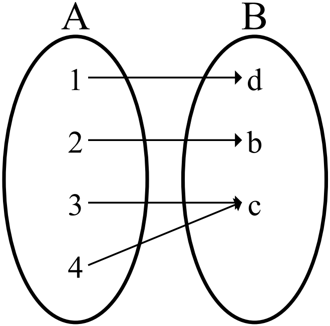
Funcții surjective

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

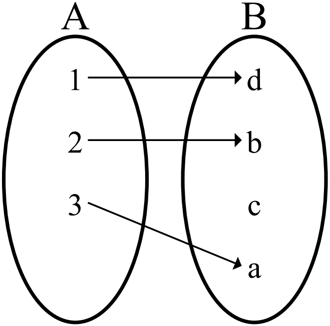
Funcții surjective

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

funcție surjectivă



funcție nesurjectivă



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

Proprietăți ale funcțiilor surjective

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

Proprietăți ale funcțiilor surjective

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

Nu și invers!!
(Putem construi f a. î. să nu ia ca valoare un element anume din B, dacă $|B| > 1$).

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

$f_1 : \mathbb{R} \to \mathbb{R}$, $f_1(x) = x^2$ nu e surjectivă,

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

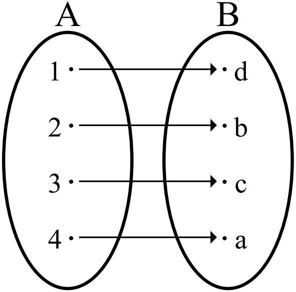
Funcții bijective. Proprietăți

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

Funcții bijective. Proprietăți

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

O funcție bijectivă $f : A \to B$ pune în corespondență *unu la unu* elementele lui A cu cele ale lui B.



Pentru *orice* funcție, din definiție,
la fiecare $x \in A$ corespunde
un *unic* $y \in B$ cu $f(x) = y$

Pentru o funcție *bijectivă*, și invers:
la fiecare $y \in B$ corespunde
un *unic* $x \in A$ cu $f(x) = y$

Dacă există $f : A \to B$ și f e bijectivă, atunci $|A| = |B|$.

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

Compunerea funcțiilor

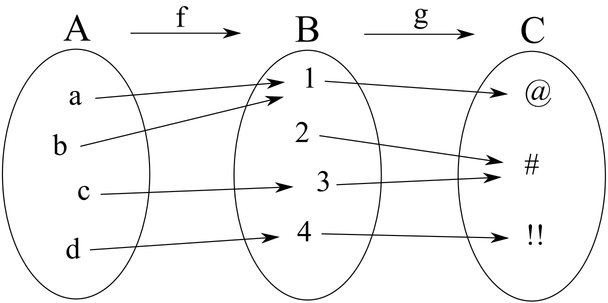
Compunerea funcțiilor

Fie functiile $f : A \to B$ și $g : B \to C$.

Compunerea lor este funcția $g \circ f : A \to C$

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

Putem compune $g \circ f$ doar când *codomeniul* lui f = *domeniul* lui g !



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

Proprietăți ale compunerii funcțiilor

Compunerea a două funcții e *asociativă*:

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

Demonstrație: fie x oarecare din domeniul lui h. Atunci:

$$\begin{aligned} ((f \circ g) \circ h)(x) &= \\ \textit{rescriem } \circ \quad &= (f \circ g)(h(x)) \\ \textit{rescriem } \circ \quad &= f(g(h(x))) \end{aligned}$$

$$\begin{aligned} (f \circ (g \circ h))(x) &= \\ \textit{rescriem } \circ \quad &= f((g \circ h)(x)) \\ \textit{rescriem } \circ \quad &= f(g(h(x))) \end{aligned}$$

Compunerea a două funcții *nu* e neapărat *comutativă*

Puteți da un exemplu pentru care $f \circ g \neq g \circ f$?

Funcții inversabile

Pe orice mulțime A putem defini *funcția identitate*

$id_A : A \to A$

$id_A(x) = x$ (notată adeseori și $\mathbf{1}_A$)

O funcție $f : A \to B$ e *inversabilă* dacă există o funcție

$f^{-1} : B \to A$ astfel încât

$f^{-1} \circ f = id_A$ și

$f \circ f^{-1} = id_B$.

Funcții inversabile

O funcție e inversabilă dacă și numai dacă e *bijectivă*.

Funcții inversabile

O funcție e inversabilă dacă și numai dacă e *bijectivă*. Demonstrăm:

Dacă f e inversabilă:
pentru $y \in B$ oarecare, fie $x = f^{-1}(y)$.

Funcții inversabile

O funcție e inversabilă dacă și numai dacă e *bijectivă*. Demonstrăm:

Dacă f e inversabilă:
pentru $y \in B$ oarecare, fie $x = f^{-1}(y)$.
Atunci $f(x) = f(f^{-1}(y)) = y$, deci f e surjectivă

Funcții inversabile

O funcție e inversabilă dacă și numai dacă e *bijectivă*. Demonstrăm:

Dacă f e inversabilă:
  pentru $y \in B$ oarecare, fie $x = f^{-1}(y)$.
Atunci $f(x) = f(f^{-1}(y)) = y$, deci f e surjectivă
  dacă $f(x_1) = f(x_2)$, atunci $f^{-1}(f(x_1)) = f^{-1}(f(x_2))$,
  deci $x_1 = x_2$ (injectivă)

Funcții inversabile

O funcție e inversabilă dacă și numai dacă e *bijectivă*. Demonstrăm:

Dacă f e inversabilă:
pentru $y \in B$ oarecare, fie $x = f^{-1}(y)$.
Atunci $f(x) = f(f^{-1}(y)) = y$, deci f e surjectivă
dacă $f(x_1) = f(x_2)$, atunci $f^{-1}(f(x_1)) = f^{-1}(f(x_2))$,
deci $x_1 = x_2$ (injectivă)

Reciproc, dacă f e bijectivă:
– f e surjectivă $\Rightarrow$ pentru orice $y \in B$ *există* $x \in A$ cu $f(x) = y$

Funcții inversabile

O funcție e inversabilă dacă și numai dacă e *bijectivă*. Demonstrăm:

Dacă f e inversabilă:
pentru $y \in B$ oarecare, fie $x = f^{-1}(y)$.
Atunci $f(x) = f(f^{-1}(y)) = y$, deci f e surjectivă
dacă $f(x_1) = f(x_2)$, atunci $f^{-1}(f(x_1)) = f^{-1}(f(x_2))$,
deci $x_1 = x_2$ (injectivă)

Reciproc, dacă f e bijectivă:
– f e surjectivă $\Rightarrow$ pentru orice $y \in B$ *există* $x \in A$ cu $f(x) = y$
– f fiind injectivă, dacă $f(x_1) = y = f(x_2)$, atunci $x_1 = x_2$.

Funcții inversabile

O funcție e inversabilă dacă și numai dacă e *bijectivă*. Demonstrăm:

Dacă f e inversabilă:
pentru $y \in B$ oarecare, fie $x = f^{-1}(y)$.
Atunci $f(x) = f(f^{-1}(y)) = y$, deci f e surjectivă
dacă $f(x_1) = f(x_2)$, atunci $f^{-1}(f(x_1)) = f^{-1}(f(x_2))$,
deci $x_1 = x_2$ (injectivă)

Reciproc, dacă f e bijectivă:
– f e surjectivă $\Rightarrow$ pentru orice $y \in B$ *există* $x \in A$ cu $f(x) = y$
– f fiind injectivă, dacă $f(x_1) = y = f(x_2)$, atunci $x_1 = x_2$.

Deci $f^{-1} : B \to A$, $f^{-1}(y) =$ acel x a. î. $f(x) = y$
e o funcție bine definită, $f^{-1}(f(x)) = x$, și $f(f^{-1}(y)) = y$.

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 $f(S)$.

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 $f^{-1}(T)$.

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 $f^{-1}(T)$.

$$f^{-1}(f(S)) \supseteq S$$

Aplicând întâi funcția și apoi inversa ei se pierde precizie.
(*nu orice calcul e reversibil*).

Probleme de numărare

Câte funcții există de la A la B ?

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

(în fiecare element din B se poate mapa orice element din A)

Demonstrație: prin *inducție matematică* după $|A|$

Mulțimea funcțiilor $f : A \to B$ se notează uneori B^A

Notația ne amintește că numărul acestor funcții e $|B|^{|A|}$.

Câte funcții injective există de la A la B ?

Dacă A și B sunt mulțimi finite și $f : A \to B$ injectivă
$\Rightarrow |f(A)| = |A|$ (imaginea lui f va avea $|A|$ elemente).

Câte funcții injective există de la A la B ?

Dacă A și B sunt mulțimi finite și $f : A \to B$ injectivă
$\Rightarrow |f(A)| = |A|$ (imaginea lui f va avea $|A|$ elemente).

Ordinea în care alegem elementele *contează* !
(ordini diferite $\Rightarrow$ funcții diferite)
... deci avem aranjamente de $|B|$ luate câte $|A|$

Câte funcții injective există de la A la B ?

Dacă A și B sunt mulțimi finite și $f : A \to B$ injectivă
$\Rightarrow |f(A)| = |A|$ (imaginea lui f va avea $|A|$ elemente).

Ordinea în care alegem elementele *contează* !
(ordini diferite $\Rightarrow$ funcții diferite)
... deci avem aranjamente de $|B|$ luate câte $|A|$

$\Rightarrow$ există $A_{|B|}^{|A|} = \dfrac{|B|!}{(|B| - |A|)!}$ funcții injective

Câte funcții bijective există de la A la B ?

Dacă A și B sunt mulțimi finite și $f : A \to B$ bijectivă
$\Rightarrow |f(A)| = |A| = |B|$ (imaginea lui f va avea $|A|$ elemente).

Ordinea în care alegem elementele *contează* !
... deci avem permutări de $|A|$ elemente

Câte funcții bijective există de la A la B ?

Dacă A și B sunt mulțimi finite și $f : A \to B$ bijectivă
$\Rightarrow |f(A)| = |A| = |B|$ (imaginea lui f va avea $|A|$ elemente).

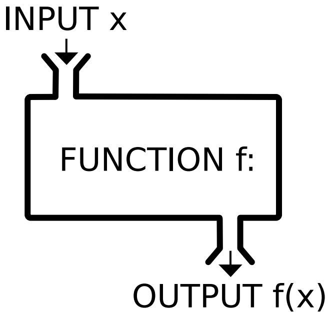
Ordinea în care alegem elementele *contează* !
... deci avem permutări de $|A|$ elemente

$\Rightarrow$ există $P_{|A|} = |A|!$ funcții bijective

Funcții – aspect computațional

Funcții: aspectul computațional

În limbajele de programare, o funcție exprimă un *calcul*:
primește o *valoare* (*argumentul*) și produce ca *rezultat* altă *valoare*



Imagine: `http://en.wikipedia.org/wiki/File:Function_machine2.svg`

Funcții în OCaml

Cel mai simplu, definim funcții astfel:

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

“fie funcția f de argument x, cu valoarea $x + 1$”

Putem defini și identificatori cu alte valori (de ex. numerice):

`let y = 3`   definește identificatorul `y` cu valoarea 3 (un întreg)

Funcții în OCaml

Cel mai simplu, definim funcții astfel:

`let f x = x + 1`
"fie funcția f de argument x, cu valoarea $x + 1$"

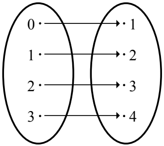
Putem defini și identificatori cu alte valori (de ex. numerice):

`let y = 3` definește identificatorul `y` cu valoarea 3 (un întreg)

`let` *nume* = *expresie*
leagă (asociază) *identificatorul* *nume* cu valoarea expresiei date

Funcțiile sunt și ele valori

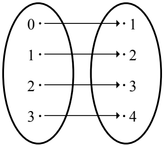
În diagrame, funcțiile nu au neapărat nume:



funcția care asociază 1 lui 0, etc.

Funcțiile sunt și ele valori

În diagrame, funcțiile nu au neapărat nume:



funcția care asociază 1 lui 0, etc.

Putem scrie și în OCaml:

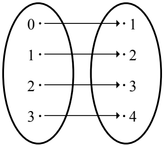
`fun x -> x + 1` o *expresie* reprezentând o funcție *anonimă*

Ca la orice expresie, putem asocia un nume cu valoarea expresiei:

`let f = fun x -> x + 1` e la fel ca `let f x = x + 1`

Funcțiile sunt și ele valori

În diagrame, funcțiile nu au neapărat nume:



funcția care asociază 1 lui 0, etc.

Putem scrie și în OCaml:

`fun x -> x + 1` o *expresie* reprezentând o funcție *anonimă*

Ca la orice expresie, putem asocia un nume cu valoarea expresiei:

`let f = fun x -> x + 1` e la fel ca `let f x = x + 1`

O funcție e și ea o *valoare* (ca întregii, realii, etc.) și poate fi folosită la fel ca orice valoare (dată ca parametru, returnată, etc.)

Apelul de funcție

Dacă am definit o funcție:

```
let f x = x + 3
```

o apelăm scriind numele funcției, apoi argumentul:

```
f 2
```

Putem apela direct și o funcție anonimă:

```
(fun x -> x + 3) 2
```

Apelul de funcție

Dacă am definit o funcție:

```
let f x = x + 3
```

o apelăm scriind numele funcției, apoi argumentul:

```
f 2
```

Putem apela direct și o funcție anonimă:

```
(fun x -> x + 3) 2
```

Interpretorul răspunde, calculând valoarea:

```
- : int = 5
```

avem o valoare fără nume (–), care e un întreg, și are valoarea 5

Apelul de funcție

Apel de funcție în ML:

```
f 2
```

Apelul de funcție

Apel de funcție în ML:

```
f 2
```

<u>În ML, funcțiile se apelează fără paranteze!</u>

În matematică, folosim paranteze:

ca să grupăm calcule care se fac întâi: $(2 + 3) * (7 - 3)$

ca să identificăm argumentele funcțiilor: f(2)

Apelul de funcție

Apel de funcție în ML:

```
f 2
```

În ML, funcțiile se apelează fără paranteze!

În matematică, folosim paranteze:

ca să grupăm calcule care se fac întâi: $(2+3)*(7-3)$

ca să identificăm argumentele funcțiilor: f(2)

În ML, folosim paranteze doar pentru a grupa (sub)expresii:

```
f (5+7)
(fun x -> x + 3) 2
```

Diverse limbaje au reguli de scris diferite (sintaxa).

Tipuri de date

Dacă definim

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

interpretorul OCaml *evaluează* definiția și răspunde:

```
val f : int -> int = <fun>
```

Tipuri de date

Dacă definim

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

interpretorul OCaml *evaluează* definiția și răspunde:

```
val f : int -> int = <fun>
```

Matematic:

f e o funcție de la întregi la întregi

În program:

`f` e o funcție cu argument de *tip* întreg (`int`)

și rezultat de *tip* întreg (*domeniul* și *codomeniul* devin *tipuri*)

Tipuri de date

```
val f : int -> int = <fun>
```

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

```
int -> int
```

e tipul funcțiilor de *argument întreg* cu *valoare întreagă*.

Tipuri de date

```
val f : int -> int = <fun>
```

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

`int -> int`

e tipul funcțiilor de *argument întreg* cu *valoare întreagă*.

În ML, tipurile pot fi deduse *automat* (*inferență de tip*):

pentru că la `x` se aplică `+`, compilatorul deduce că `x` e întreg

Pentru reali, am scrie `let f x = x +. 1.`

cu punct zecimal pentru reali, și în operatori: `+.`, `*.` etc.

Funcții definite pe cazuri

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

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

Funcții definite pe cazuri

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

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

În ML:

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

Funcții definite pe cazuri

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

`if` $expr_1$ `then` $expr_2$ `else` $expr_3$

e o *expresie condițională*

Funcții definite pe cazuri

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

`if` $expr_1$ `then` $expr_2$ `else` $expr_3$

e o *expresie condițională*

Dacă *evaluarea* lui $expr_1$ dă valoarea *true* (*adevărat*)
valoarea expresiei e valoarea lui $expr_2$,
altfel e valoarea lui $expr_3$.

$expr_2$ și $expr_3$ trebuie să aibe *același tip* (ambele întregi, reale, ...)

Funcții definite pe cazuri

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

`if` $expr_1$ `then` $expr_2$ `else` $expr_3$

e o *expresie condițională*

Dacă *evaluarea* lui $expr_1$ dă valoarea *true* (*adevărat*)
valoarea expresiei e valoarea lui $expr_2$,
altfel e valoarea lui $expr_3$.

$expr_2$ și $expr_3$ trebuie să aibe *același tip* (ambele întregi, reale, ...)

În alte limbaje (C, Java, etc.) `if` și ramurile lui sunt *instrucțiuni*.

În ML, `if` e o *expresie*. ML *nu are* instrucțiuni, ci doar *expresii* (care sunt evaluate), și *definiții* (`let`) care dau nume unor valori.

Funcții cu mai multe argumente

Matematic:

$f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, \quad f(x, y) = 2x + y - 1$

În ML, enumerăm doar argumentele (fără paranteze, făra virgule):

```
let f x y = 2*x + y - 1
```

Funcții cu mai multe argumente

Matematic:

$f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, \quad f(x, y) = 2x + y - 1$

În ML, enumerăm doar argumentele (fără paranteze, făra virgule):

```
let f x y = 2*x + y - 1
```

iar interpretorul răspunde

```
val f : int -> int -> int = <fun>
```

`f` e o funcție care ia un *întreg* și încă un *întreg*
și *returnează un întreg*.

Funcții cu mai multe argumente

```
let f x y = 2*x + y - 1
val f : int -> int -> int = <fun>
```

Să fixăm primul argument, de ex. $x = 2$:
$f(2, y) = 2 \cdot 2 + y - 1$

Am obținut o funcție de un argument (y), singurul rămas nelegat.

Funcții cu mai multe argumente

```
let f x y = 2*x + y - 1
val f : int -> int -> int = <fun>
```

Să fixăm primul argument, de ex. $x = 2$:
$f(2, y) = 2 \cdot 2 + y - 1$

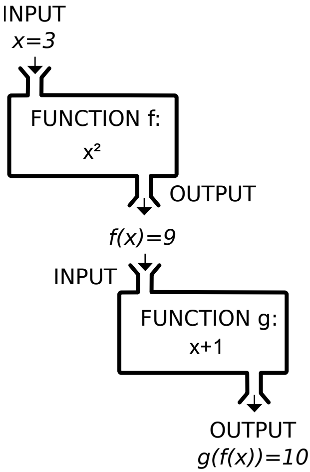
Am obținut o funcție de un argument (y), singurul rămas nelegat.

În ML, evaluând
`f 2` (fixând x = 2)

interpretorul răspunde:
`- : int -> int = <fun>`.

Deci, `f` e de fapt o funcție cu *un* argument x, care returnează o *funcție*. Aceasta ia argumentul y și returnează rezultatul numeric.

Compunerea funcțiilor - ilustrare computațională



Rezultatul funcției f devine argument pentru funcția g

Prin compunere, construim funcții complexe din funcții mai simple.

Imagine: `http://en.wikipedia.org/wiki/File:Function_machine5.svg`

Compunerea funcțiilor în ML

Definim o funcție `comp` care compune două funcții:

```
let comp f g x = f (g x)
```

Echivalent, puteam scrie:

```
let comp f g = fun x -> f (g x)
```

`comp f g`

e funcția care primind argumentul x returnează $f(g(x))$

Compunerea funcțiilor în ML

Definim o funcție `comp` care compune două funcții:

```
let comp f g x = f (g x)
```

Echivalent, puteam scrie:

```
let comp f g = fun x -> f (g x)
```

```
comp f g
```

e funcția care primind argumentul x returnează $f(g(x))$

Interpretorul indică

```
val comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Compunerea funcțiilor în ML

```
val comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Tipurile `'a`, `'b`, `'c` pot fi *oarecare*.

Compunerea funcțiilor în ML

```
val comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Tipurile `'a`, `'b`, `'c` pot fi *oarecare*.

Argument cu argument:

`'c` e tipul lui x

`'c -> 'a` e tipul lui g: duce pe x în tipul `'a`

`'a -> 'b` e tipul lui `f`: duce tipul `'a` în tipul `'b`
(codomeniul lui g e domeniul lui f)

`'b` e tipul rezultatului

Compunerea funcțiilor în ML

```
val comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Tipurile `'a`, `'b`, `'c` pot fi *oarecare*.

Argument cu argument:

`'c` e tipul lui x

`'c -> 'a` e tipul lui g: duce pe x în tipul `'a`

`'a -> 'b` e tipul lui `f`: duce tipul `'a` în tipul `'b`

(codomeniul lui g e domeniul lui f)

`'b` e tipul rezultatului

Putem apela:

```
comp (fun x -> 2*x) (fun x -> x + 1) 3
```

care dă `2 * (x + 1)` pentru $x = 3$, adică 8.

Operatorii sunt funcții

Operatorii (ex. matematici, +, *, etc.) sunt tot *funcții*:
ei calculează un rezultat din valorile operanzilor (argumentelor).

Diferența e doar de *sintaxă*:
scriem operatorii *între* operanzi (*infix*),
iar numele funcției *înaintea* argumentelor (*prefix*).

Operatorii sunt funcții

Operatorii (ex. matematici, +, ∗, etc.) sunt tot *funcții*:
ei calculează un rezultat din valorile operanzilor (argumentelor).

Diferența e doar de *sintaxă*:
scriem operatorii *între* operanzi (*infix*),
iar numele funcției *înaintea* argumentelor (*prefix*).

Putem scrie în ML operatorii și prefix:

`(+) 3 4` parantezele deosebesc de operatorul + unar

```
let add1 = (+) 1
add1 3
```
la fel ca: `(+) 1 3`

`add1` e funcția care adaugă 1 la argument, deci `fun x -> x + 1`

Rezumat

Prin *funcții* exprimăm calcule în programare.
Operatorii sunt cazuri particulare de funcții.

Domeniile de definiție și valori corespund *tipurilor* din programare.

Când scriem/compunem funcții, *tipurile* trebuie *să se potrivească*.

În limbajele funcționale, funcțiile pot fi manipulate ca orice *valori*.
Funcțiile pot fi *argumente* și *rezultate* de funcții.

Funcțiile de mai multe argumente (sau de tuple) pot fi rescrise
ca funcții de un singur argument care returnează funcții.

De știut

Să *raționăm* despre funcții injective, surjective, bijective, inversabile

Să *construim* funcții cu anumite proprietăți

Să *numărăm* funcțiile definite pe mulțimi finite (cu proprietăți date)

Să *compunem* funcții simple pentru a rezolva probleme

Să identificăm *tipul* unei funcții