Logică și structuri discrete
Funcții

Marius Minea
marius@cs.upt.ro

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

26 septembrie 2016

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*, mai ales în *limbajele de programare*
$\Rightarrow$ cum să *programăm mai bine*

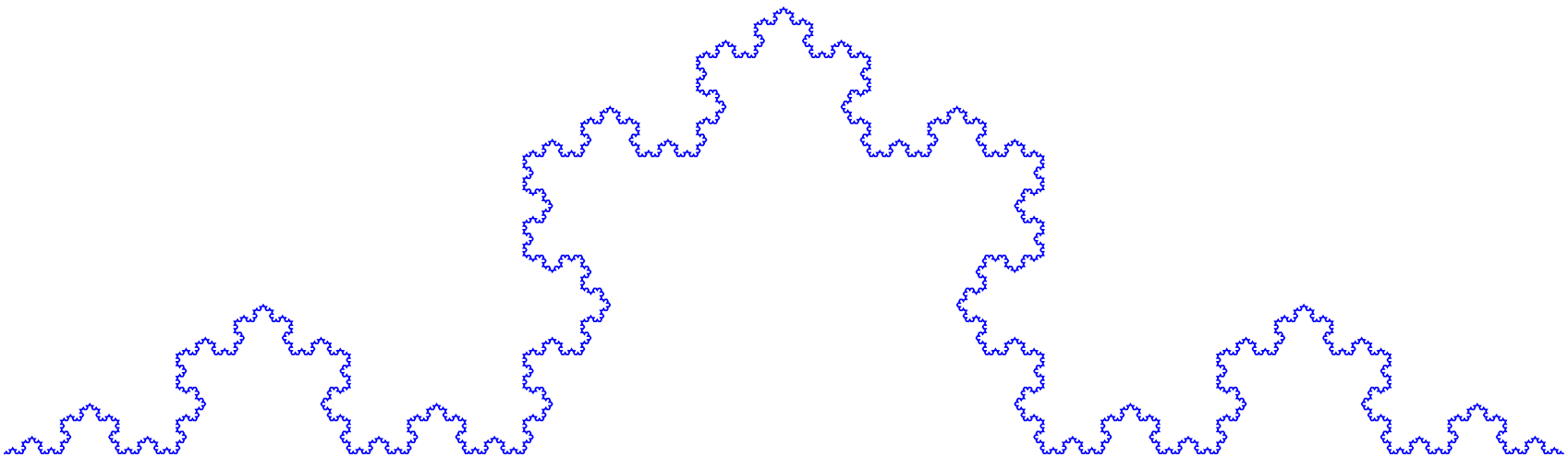
Ce învățăm la acest curs ?

funcții: modulul de bază pentru *calcule*

relații: apar în grafuri, rețele (sociale), calcul paralel, ...

liste, mulțimi: prelucrarea de *colecții* de obiecte

recursivitate: cum să definim *simplu* prelucrări *complexe*
și să rezolvăm probleme prin subprobleme mai mici



Fractalul lui Koch

Ce învățăm la acest curs ?

Logică matematică

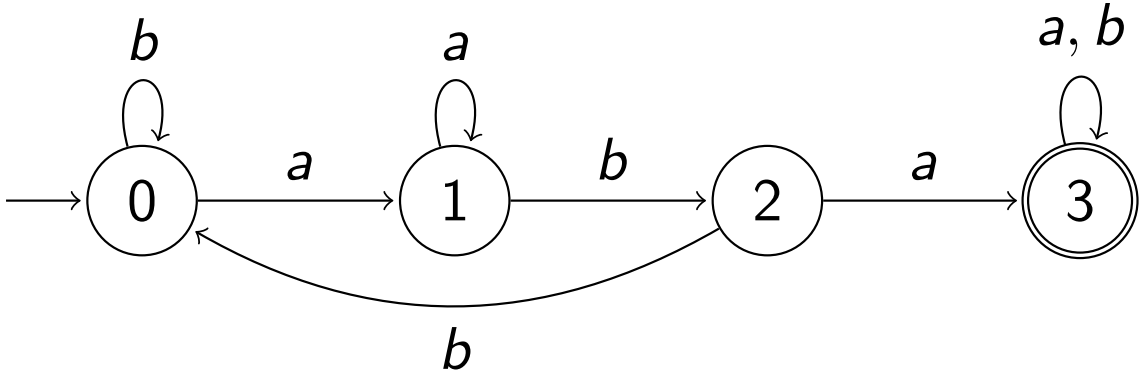
cum exprimăm *precis* afirmații
pentru definiții riguroase, specificații în software, ...

cum *demonstrăm* afirmații
pentru a arăta că un algoritm e corect

cum *prelucrăm* formule logice
pentru a găsi soluții la probleme (ex. programare logică)

Ce învățăm la acest curs ?

automate: sisteme cu logică de control simplă



expresii regulate: prelucrări simple de text $(a|b)^*aba(a|b)^*$

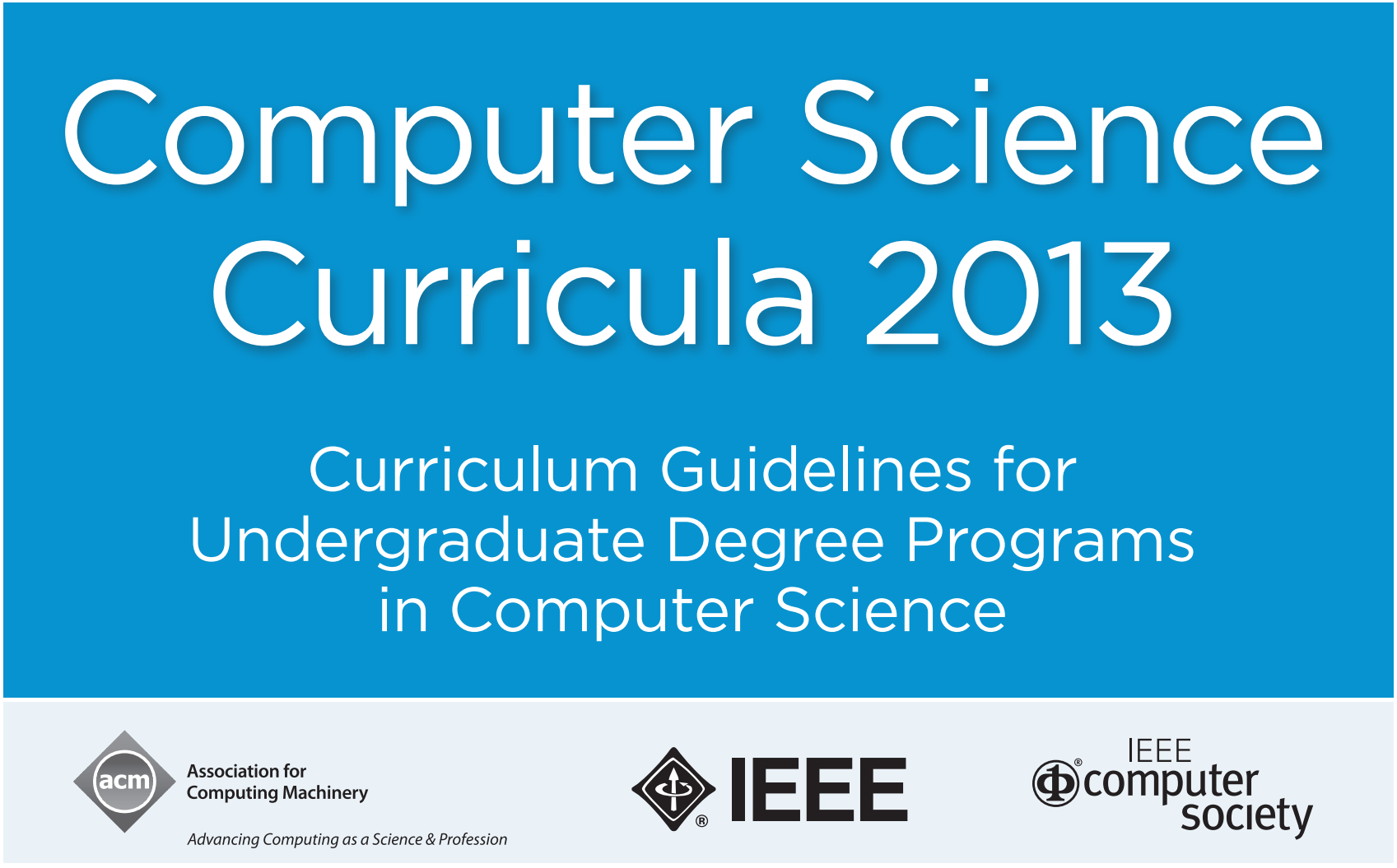
gramatici: sintaxa limbajelor de programare

$$\mathit{IfStmt} ::= \texttt{if}\ (\ \mathit{Expr}\ )\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}$$

arbori: prelucrări de expresii numerice

grafuri: vizualizarea relațiilor

Ce cunoștințe se cer în domeniul calculatoarelor?



http://www.acm.org/education/curricula-recommendations

CS body of knowledge

Knowledge Area	CS2013 Tier1	CS2013 Tier2	CS2008 Core	CC2001 Core
AL-Algorithms and Complexity	19	9	31	31
AR-Architecture and Organization	0	16	36	36
CN-Computational Science	1	0	0	0
DS-Discrete Structures	37	4	43	43
GV-Graphics and Visualization	2	1	3	3
HCI-Human-Computer Interaction	4	4	8	8
IAS-Information Assurance and Security	3	6	--	--
IM-Information Management	1	9	11	10
IS-Intelligent Systems	0	10	10	10
NC-Networking and Communication	3	7	15	15
OS-Operating Systems	4	11	18	18
PBD-Platform-based Development	0	0	--	--
PD-Parallel and Distributed Computing	5	10	--	--
PL-Programming Languages	8	20	21	21
SDF-Software Development Fundamentals	43	0	47	38
SE-Software Engineering	6	22	31	31
SF-Systems Fundamentals	18	9	--	--
SP-Social Issues and Professional Practice	11	5	16	16
Total Core Hours	**165**	**143**	**290**	**280**

Exemplu: schema curriculară la MIT

project: 1/2+1/2

advanced: 2

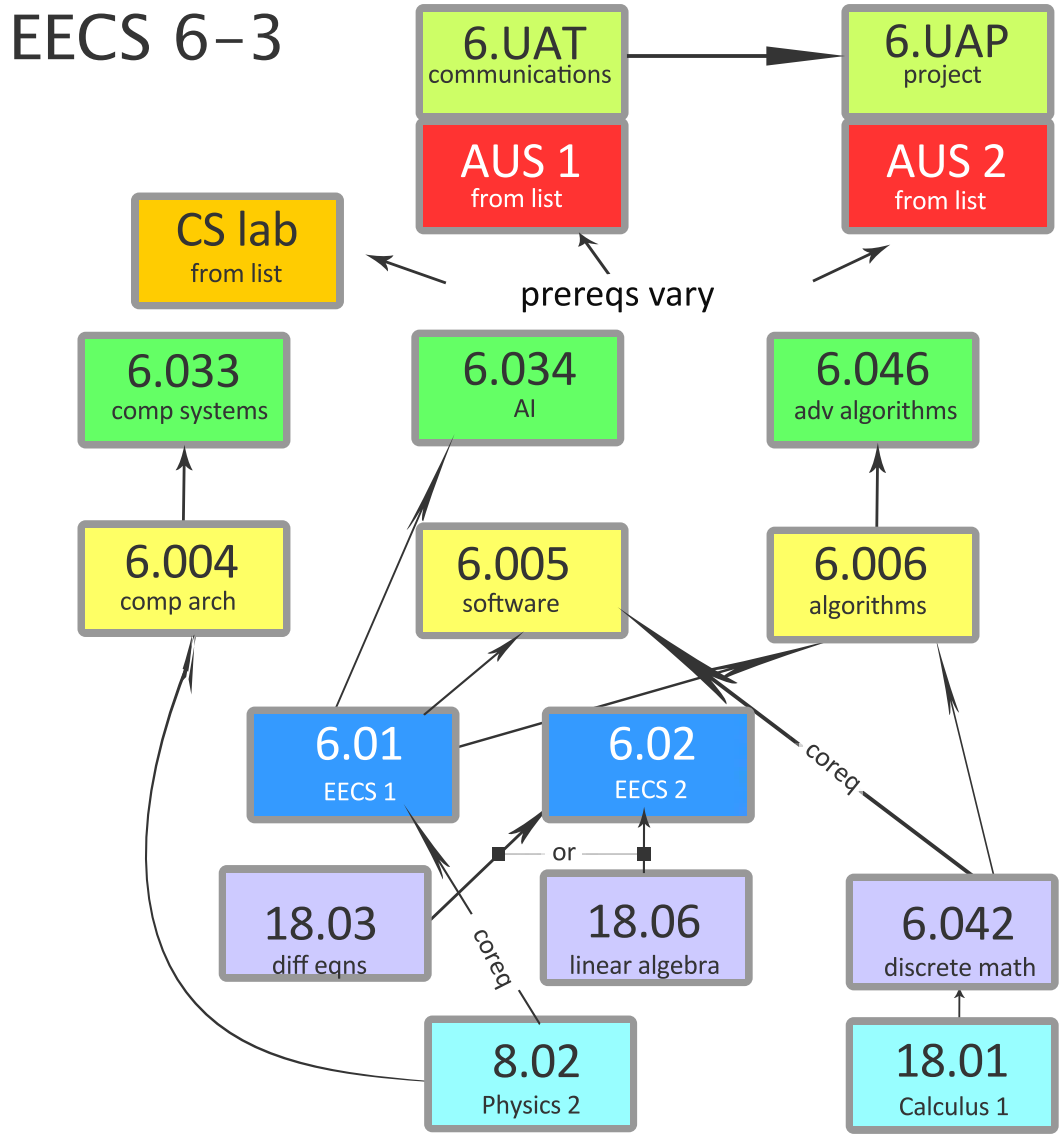
lab: 1

header: 3

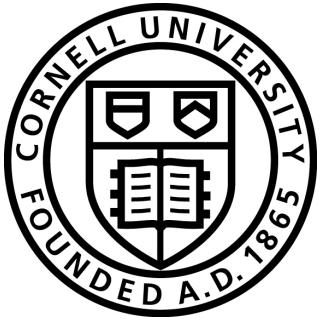
foundational: 3

introductory: 2

math: 2
18.03/06 + 6.042

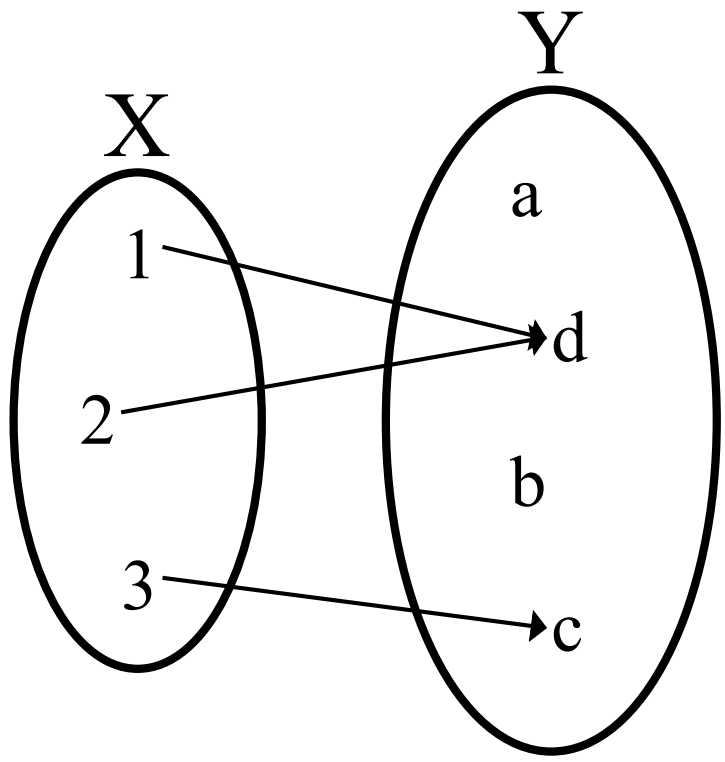


Unde se predă programare funcțională?



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:

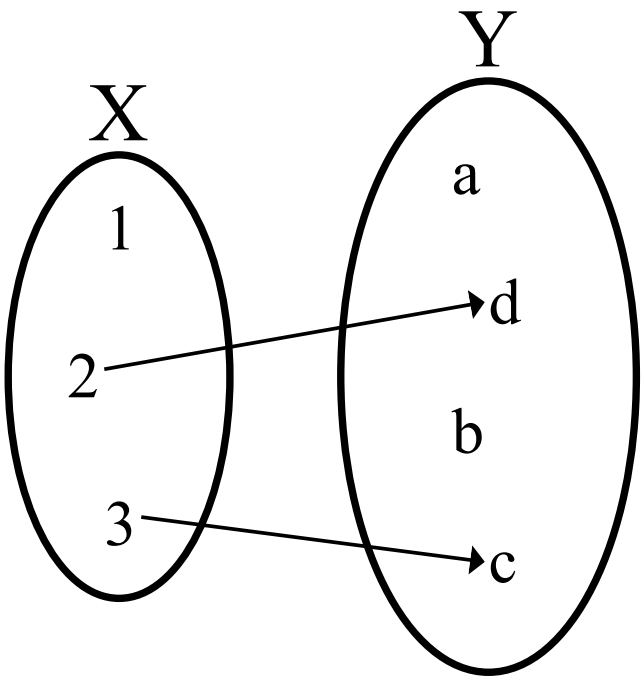
1. *domeniul de definiție* (A)
2. *domeniul de valori* (B)
3. asocierea/corespondența propriu-zisă (legea, regula)

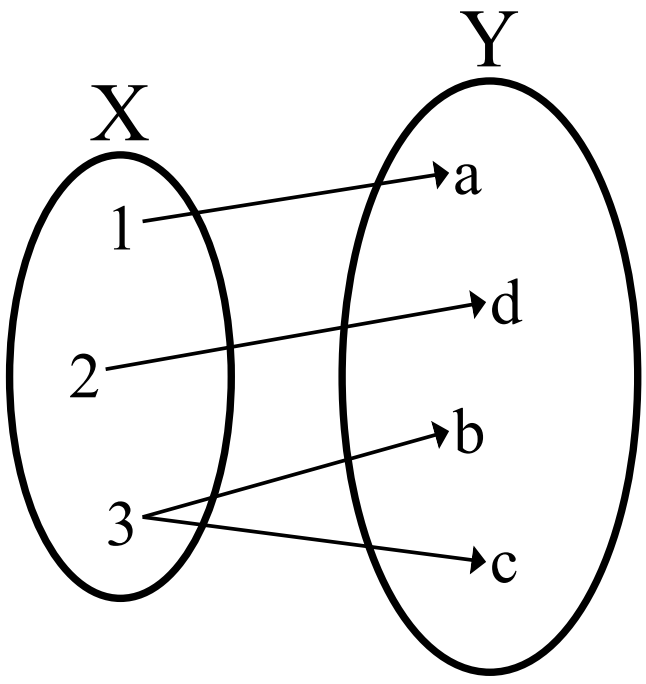
$f : \mathbb{Z} \to \mathbb{Z}$, $f(x) = x + 1$ și $f : \mathbb{R} \to \mathbb{R}$, $f(x) = x + 1$
sunt funcții distincte!

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

Putem avea funcții care lucrează cu *mai multe* tipuri (*polimorfism*).

Exemple care NU sunt funcții





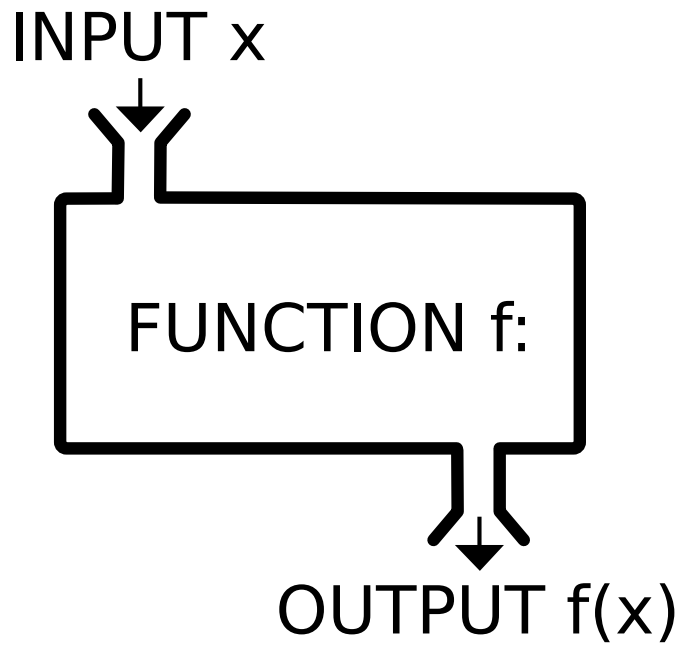
nu asociază o valoare fiecărui element

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`

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 limbajele de programare

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

Aceasta se vede cel mai clar în *limbajele de programare funcționale*:

lucrăm cu funcții la fel de simplu ca și cu alte valori uzuale
(întregi, reali, etc.)

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

Programare funcțională în ML

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

ilustrează bine conceptele de matematici discrete (liste, mulțimi)

concis (în câteva linii de cod se pot face multe)

fundamentat riguros $\Rightarrow$ evită anumite erori (ex. de *tip*)

conceptele din programarea funcțională au influențat alte limbaje
(JavaScript, lambda-expresii în C# și Java 8, Python, Scala...)
F# (platforma .NET) e foarte similar cu ML

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://ocaml.org`

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

Lambda-calcul: originea limbajelor funcționale

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

Universal: poate exprima orice funcție calculabilă
(orice poate fi calculat printr-un program)

O expresie în λ-calcul e:

- o *variabilă* — x
- o *funcție* — $\lambda x\,.\,e$ funcție de variabilă x cu expresia e
  în ML: `fun x -> e`
- o *evaluare* de functie — $e_1\ e_2$ funcția e_1 aplicată argumentului e_2
  la fel în ML: `(fun x -> x+1) 3`

  asociativă la stânga: `f x y` = `(f x) y`

lambda-calculul e fundamental în studiul limbajelor de programare

Funcția în matematică și programare

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
```

`if` *expr*$_1$ `then` *expr*$_2$ `else` *expr*$_3$ e o *expresie condițională*

Dacă *evaluarea* lui *expr*$_1$ dă valoarea *adevărat* (*true*)
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*. În ML nu avem instrucțiuni, ci doar *expresii* (care sunt evaluate), și *definiții* de valori sau funcții (cu `let`).
(Vom lucra ulterior și cu definiții de tipuri și de module.)

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 de mai sus, fiecare tipar e o valoare individuală (caracter).
Tiparul _ acoperă orice valoare (care nu a fost deja acoperită)

Limbajul ne *obligă* să acoperim toate variantele (o funcție 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 $x_1, x_2 \in A$, $x_1 \neq x_2 \Rightarrow \ f(x_1) \neq f(x_2)$

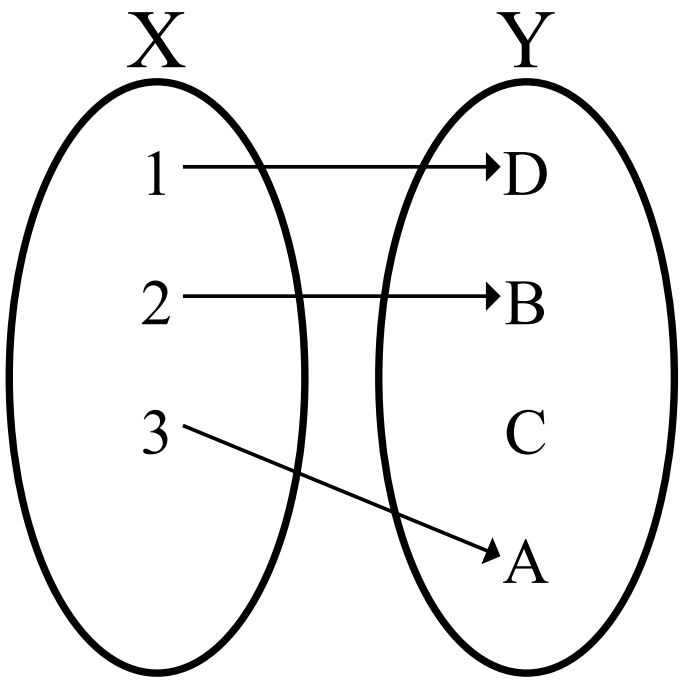
Echivalent: $f(x_1) = f(x_2) \Rightarrow x_1 = x_2$
dacă valorile sunt egale, atunci argumentele sunt egale
(*contrapozitiva* afirmației de mai sus)

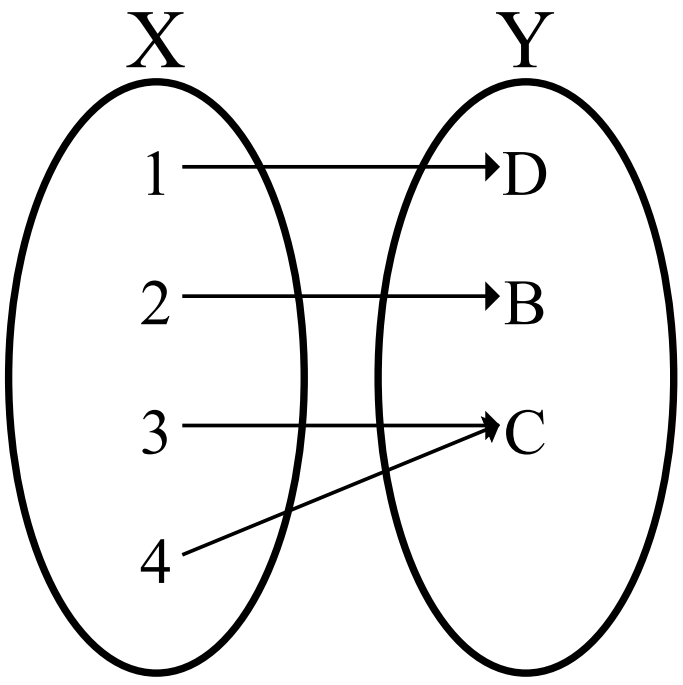
În logică, faptul că o afirmație e echivalentă cu contrapozitiva ei ne permite *demonstrația prin reducere la absurd*.

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

Nu neapărat invers!! (oricum ar fi $|A| > 1$ putem construi f să ducă două elemente din A în aceeași valoare din B).

Exemple: funcție injectivă și neinjectivă



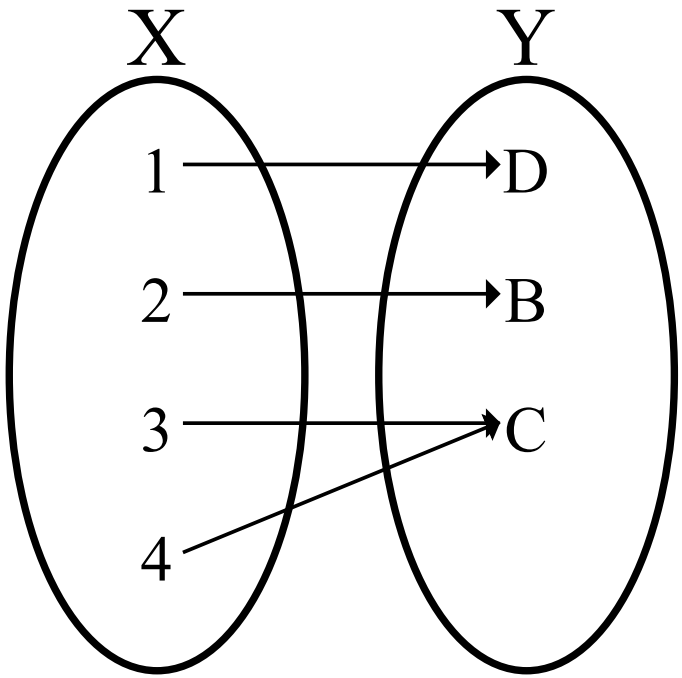


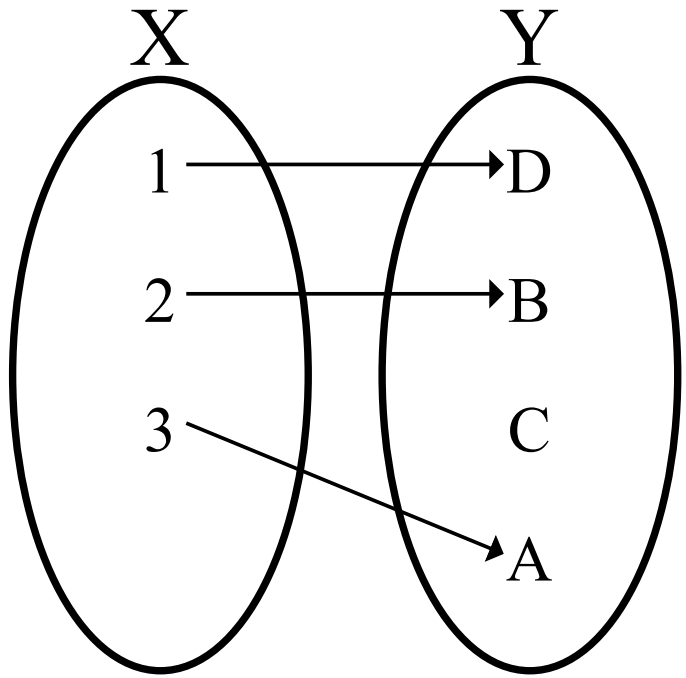
Imagine: `http://en.wikipedia.org/wiki/File:Injection.svg`
`http://en.wikipedia.org/wiki/File:Surjection.svg`

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

Nu neapărat invers! (construim f să nu ia ca valoare un element anume din B, dacă $|B| > 1$).

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* după $|A|$

Principiul inducției matematice

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

1) (*cazul de bază*) P(0) e adevărată

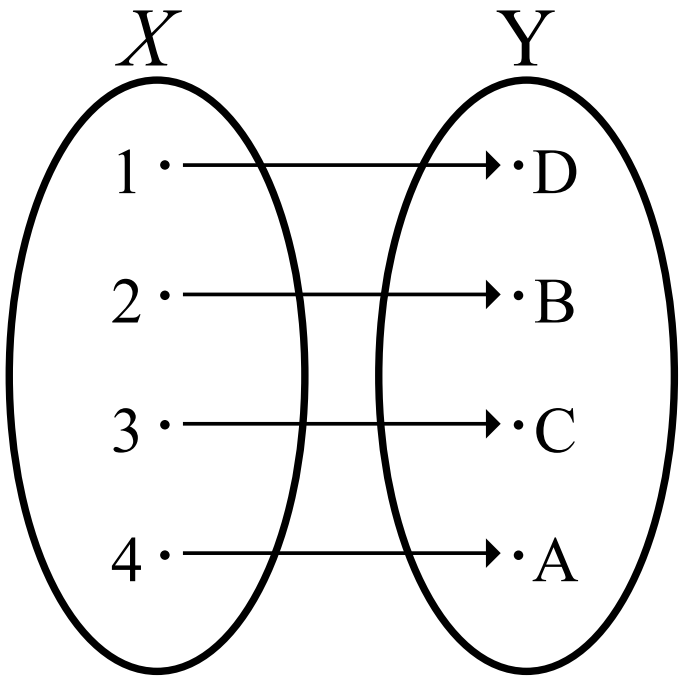
2) (*pasul inductiv*) pentru orice $n \geq 0$, $P(n) \Rightarrow P(n+1)$

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

Funcții bijective

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

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 $f(x) = y$

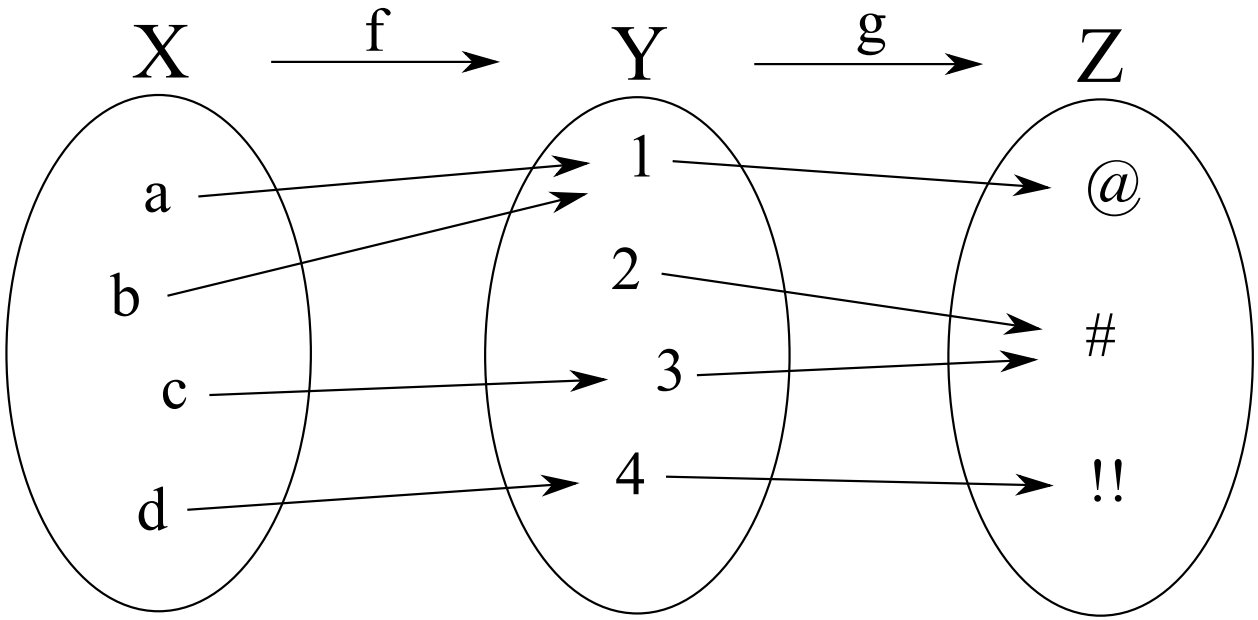
Dacă mulțimile A și B sunt finite, și f e bijectivă, atunci $|A| = |B|$.

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

Compunerea funcțiilor

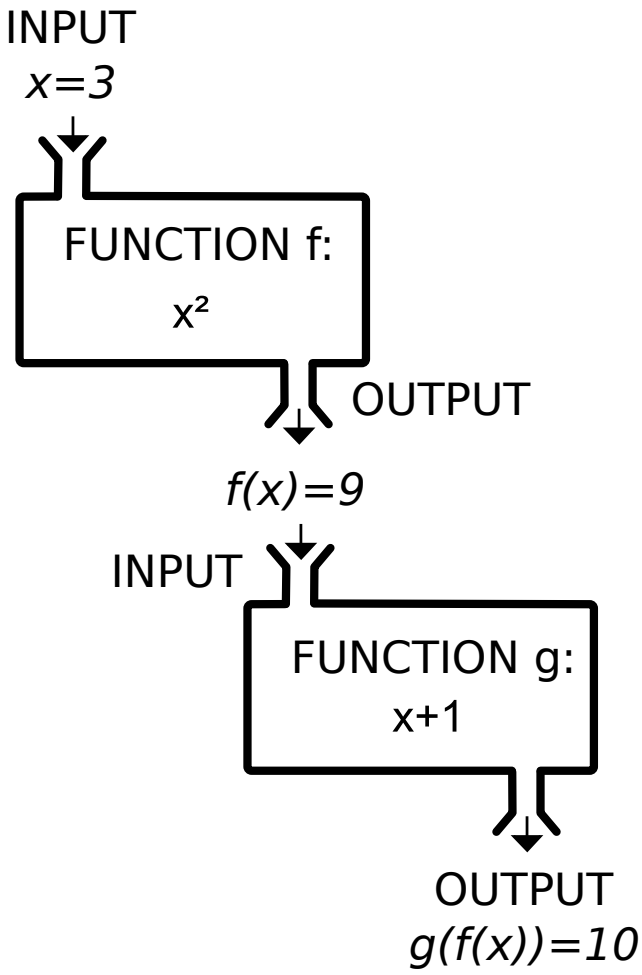
Fie functiile $f : A \to B$ și $g : B \to C$. Compunerea lor este funcția $g \circ f : A \to C$, $(g \circ f)(x) = g(f(x))$.

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



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

Compunerea funcțiilor - ilustrare computațională



Rezultatul funcției f devine argument pentru funcția g

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

Proprietăți ale 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

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

Funcții inversabile

Pe orice mulțime A definim *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 $f \circ f^{-1} = id_B$.

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 $x_1 = x_2$

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 există x_1, x_2 cu $f(x_1) = y = f(x_2)$, atunci $x_1 = x_2$.
Deci $f^{-1} : B \to A$, $f^{-1}(y) =$ acel x astfel încât $f(x) = y$
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 $f(S)$.

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).

Nu orice inversă e ușor calculabilă

Pentru o funcție inversabilă, inversa nu e neapărat *ușor calculabilă*.

Fie 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 $\mathbb{Z}_p^*$.

De exemplu, 3 e generator în Z_7^*: șirul $3^k \mod 7$ e $3, 2, 6, 4, 5, 1$

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 când p e mare (problema logaritmului discret) $\Rightarrow$ e folosită în criptografie.

Funcții cu mai multe argumente în ML

Când scriem `f(x, y) = ...` în ML, f nu are două argumente, ci *un* argument, *perechea* `(x, y)`.

Dacă $x \in A$ și $y \in B$, f e definită pe *produsul cartezian* $A \times B$:
$f : A \times B \to C$

```
let f1 (x, y) = 2*x + y - 1
```
și interpretorul răspunde
```
val f1 : int * int -> int = <fun>
```
(tipul lui `f1` e funcție de o pereche de întregi cu valoare întreagă)

Alternativ, putem defini

```
let f2 x y = 2*x + y - 1
```
și interpretorul răspunde
```
val f2 : int -> int -> int = <fun>
```

`f2` e de fapt o funcție cu *un singur* argument x, care returnează o *funcție*. Aceasta ia argumentul y și returnează rezultatul numeric.

În programarea funcțională se preferă scrierea în acest mod (numit *currying*, după Haskell Curry)
permite *evaluarea parțială* a funcției, având primul argument.

Funcții cu mai multe argumente (cont.)

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

Notația ne amintește că numărul acestor funcții e $|B|^{|A|}$
(dacă A, B sunt finite).

Numărul funcțiilor $f : A \times B \to C$ este $|C|^{|A \times B|} = |C|^{|A| \cdot |B|}$.

Rescriind prin *currying* (`f a b` în loc de `f(a, b)` în ML) obținem mulțimea funcțiilor $f : A \to C^B$ (funcții care cu un argument din A produc o funcție de la B la C, adică din C^B).

Numărul acestora e tot $(|C|^{|B|})^{|A|} = |C|^{|A| \cdot |B|}$.
Aceasta ne indică faptul că există o corespondență 1:1 (bijecție) între scrierea cu un argument pereche și scrierea cu 2 argumente.

Operatorii sunt funcții

Operatorii (ex. matematici, +, *, etc.) sunt tot niște 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` paranteza deosebește 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