Logică și structuri discrete

Funcții

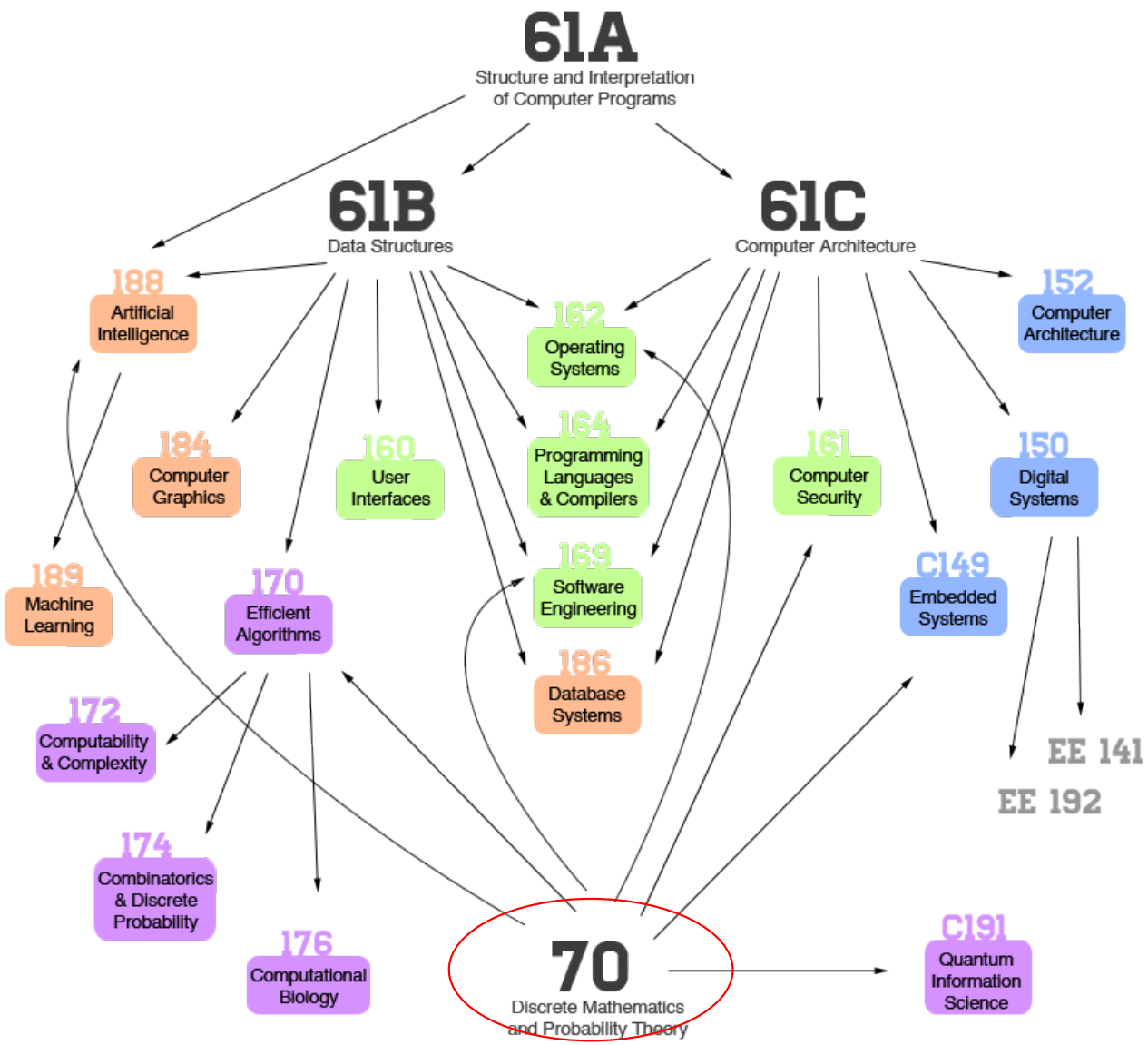
Marius Minea
marius@cs.upt.ro

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

25 septembrie 2017

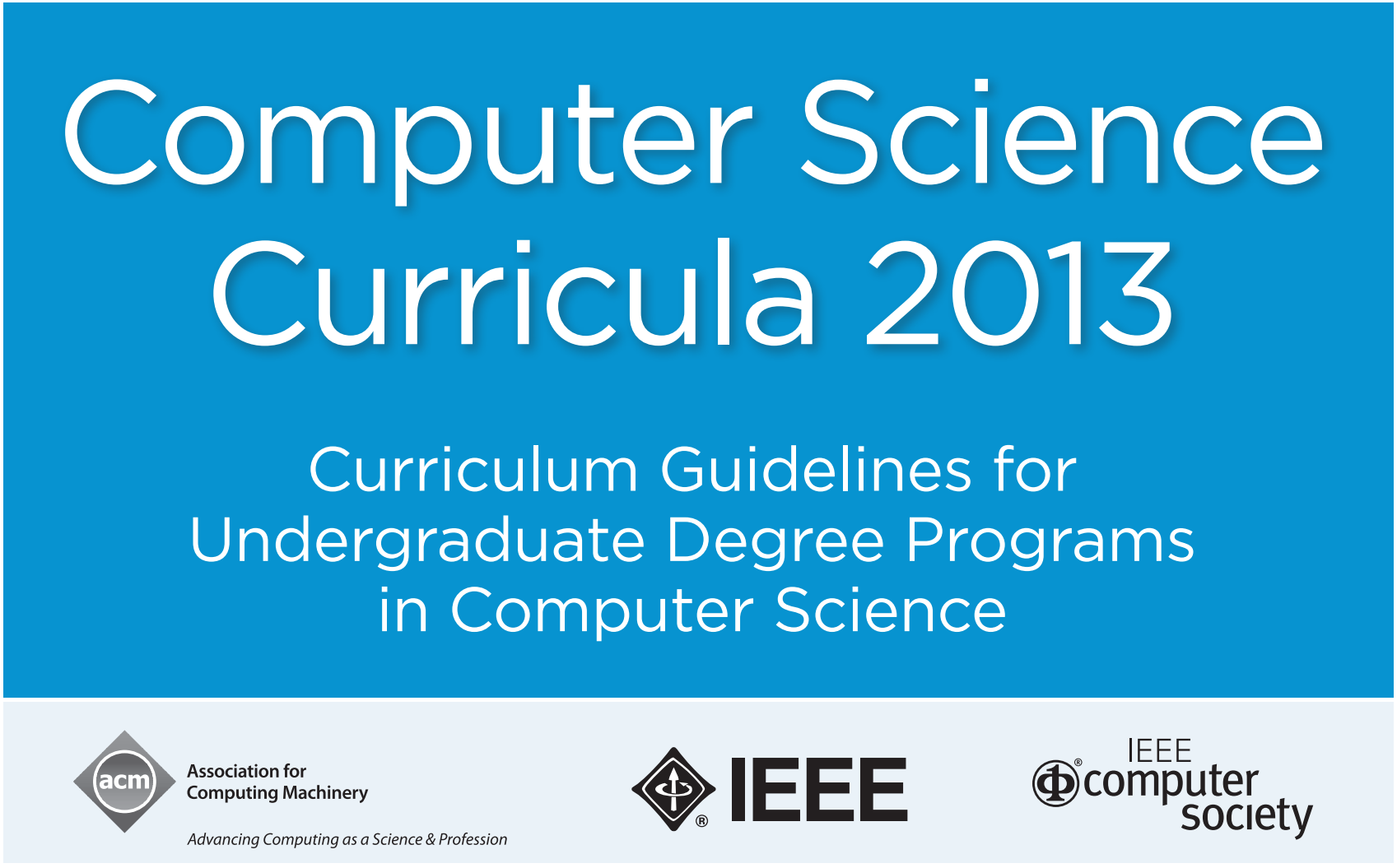
Ce cuprinde domeniul informaticii?

CORE SOFTWARE HARDWARE APPLICATIONS THEORY



Imagine: `https://hkn.eecs.berkeley.edu/courseguides`

Ce se învață în universități?



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

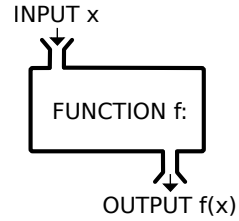
Informatica e mult mai mult decât programare!

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**

Ce învățăm la acest curs?

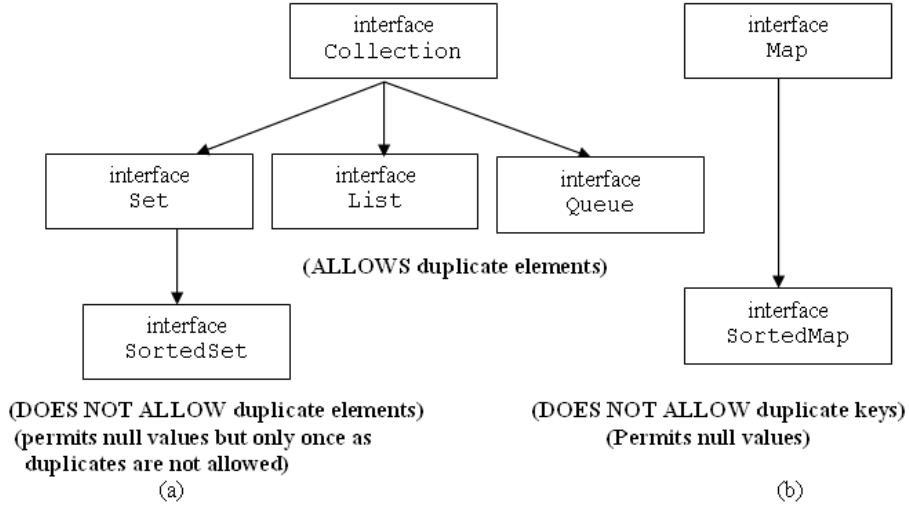
Funcții, mulțimi, relații

funcții: modulul de bază pentru *calcule*



relații: grafuri, rețele (sociale), planificare (relații de precedență), programare concurentă/paralelă (relații de dependență), ...

liste, mulțimi, dicționare:
lucrul cu *colecții* de obiecte



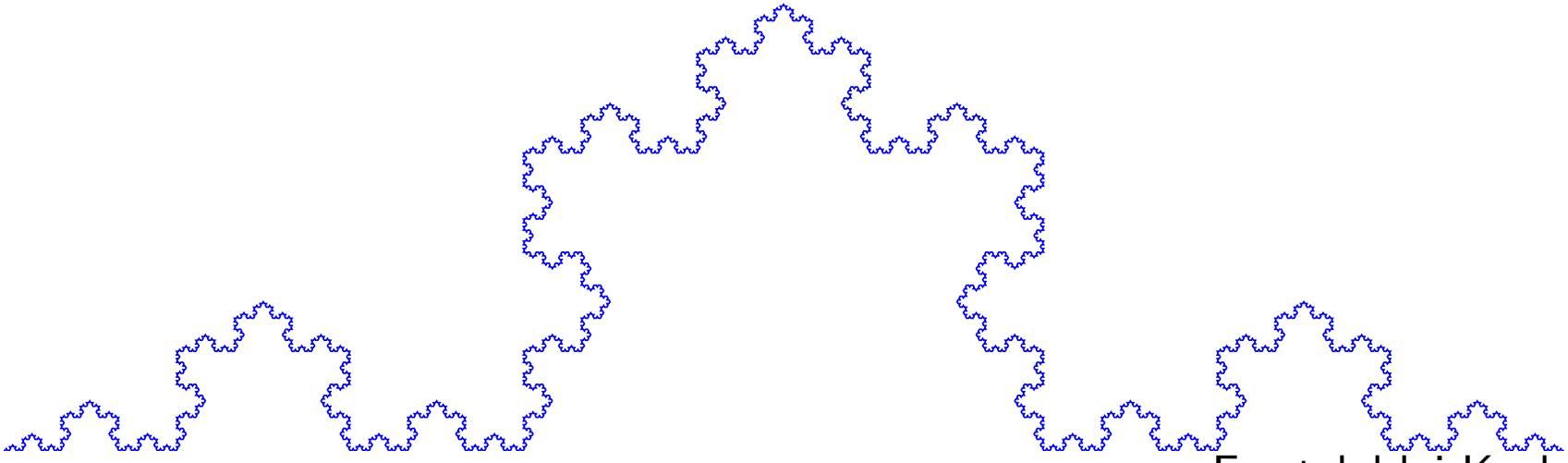
Imagine: `https://way2java.com/collections/java-collections-interfaces-hierarchy/`

Recursivitate: fundamentală în programare

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

$$C_n^k = C_{n-1}^k + C_{n-1}^{k-1} \qquad 0 < k < n$$

expresie: { întreg | *expresie* + *expresie* | *expresie* − *expresie* | *expresie* ∗ *expresie* | *expresie* / *expresie* }



Fractalul lui Koch

Logică matematică

$$p \to q = \neg p \vee q \qquad \neg(a \vee b) = \neg a \vee \neg b$$

$$\exists x P(x) = \neg \forall x \neg P(x)$$

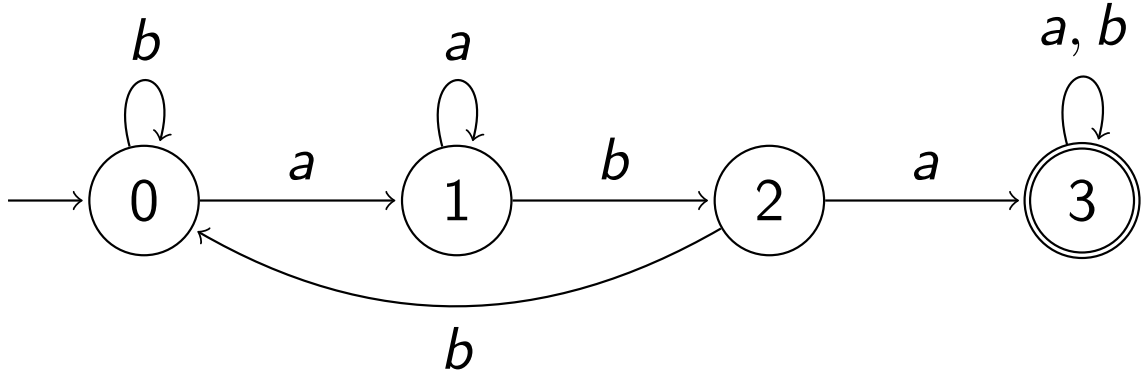
cum exprimăm *precis* afirmații
pentru definiții riguroase, specificații în software, ...

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

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

Limbaje și automate

automate: sisteme cu logică de control simplă



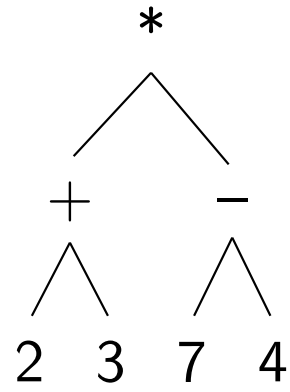
expresii regulate: prelucrări simple de text $(a|b)^*aba(a|b)^*$

gramatici: sintaxa limbajelor de programare

IfStmt ::= `if` `(` *Expr* `)` *Stmt* `else` *Stmt*

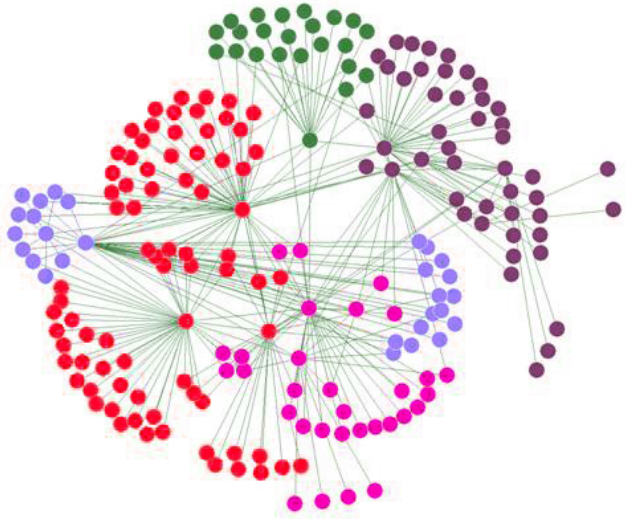
Arbori și grafuri

arbori: reprezentări / prelucrări de expresii



arbori de căutare, etc.

grafuri: vizualizarea relațiilor



parcurgeri, drumuri minime, ...

Imagine: `https://en.wikipedia.org/wiki/Social_network`

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

Programare funcțională în ML

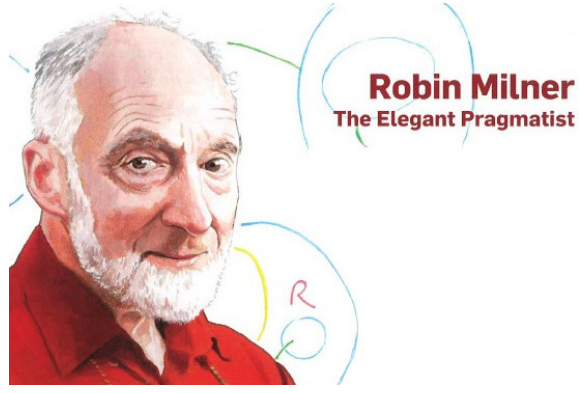
Vom lucra cu un limbaj în care noțiunea fundamentală e *funcția*

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

Cursul e *complementar* celui de programare imperativă (în C)
vom discuta ce e *comun*, și ce e *diferit* (și de ce)



Limbajul ML:
dezvoltat la Univ. Edinburgh (anii '70)
împreună cu un demonstrator de teoreme (logică matematică)

E relevantă programarea funcțională?

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

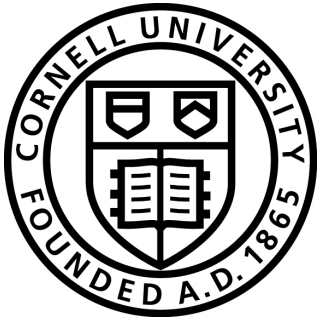
"A language that doesn't affect the way you think about programming, is not worth knowing."

Alan Perlis

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

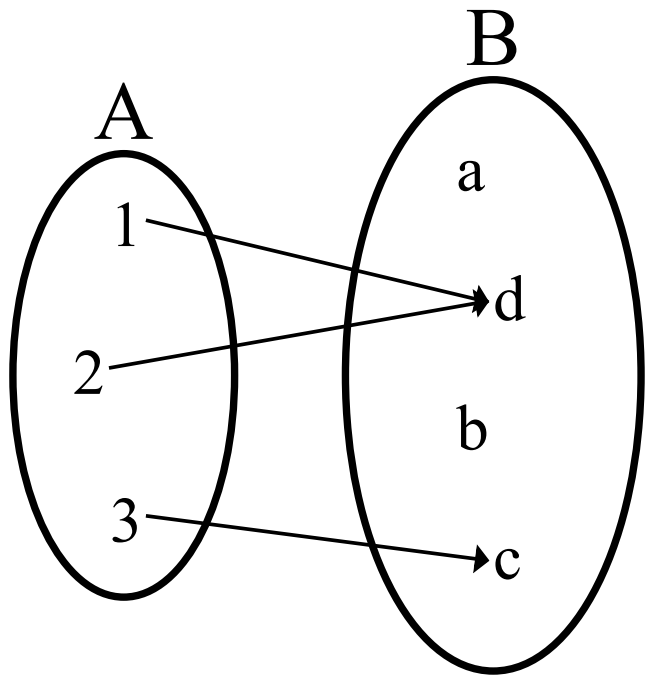
`http://ocaml.org`

Unde se predă programare funcțională?



Funcții

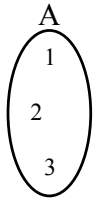
Fiind date mulțimile A și B, o *funcție* $f : A \to B$ e o asociere prin care *fiecărui* element din A îi corespunde *un singur* element din B.



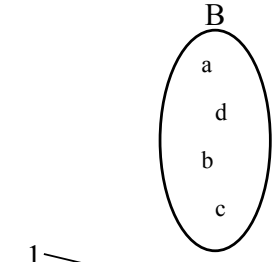
Imagine: `http://en.wikipedia.org/wiki/File:Total_function.svg`

O funcție e definită prin trei componente

1. *domeniul de definiție*

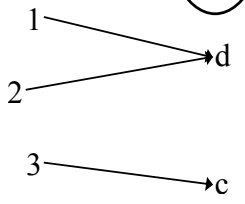


2. *domeniul de valori* (codomeniul)



3. asocierea/corespondența propriu-zisă

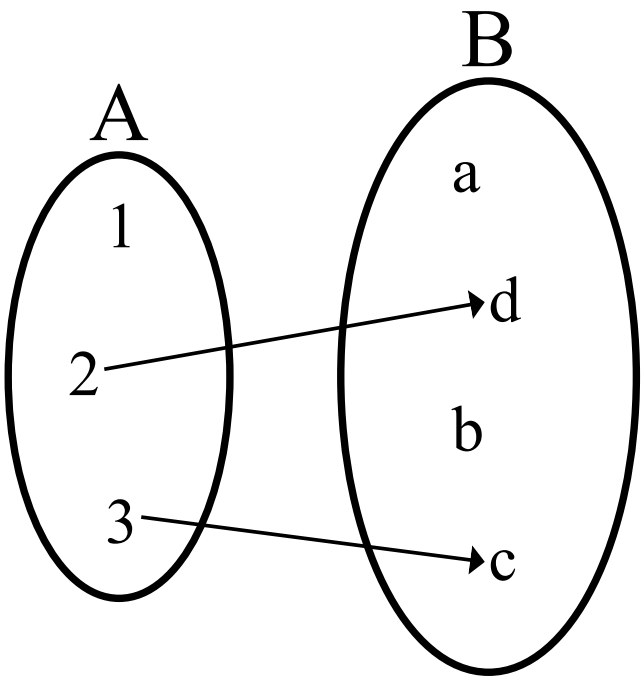
   (legea, regula de asociere)

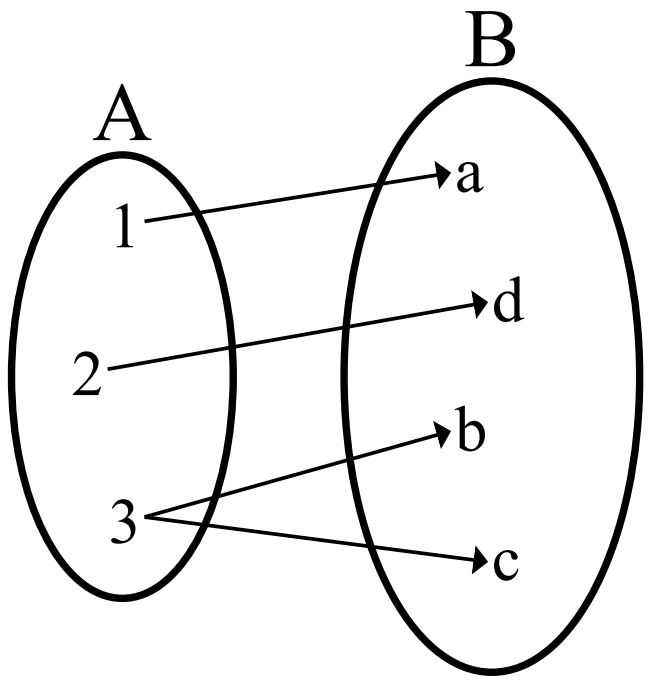


$f : \mathbb{Z} \to \mathbb{Z}$, $f(x) = x + 1$ și $f : \mathbb{R} \to \mathbb{R}$, $f(x) = x + 1$
sunt funcții distincte!

În limbajele de programare, domeniul de definiție și de valori corespund *tipurilor de date* (întregi, reali, booleni, enumerare, ...)

Exemple care NU sunt funcții





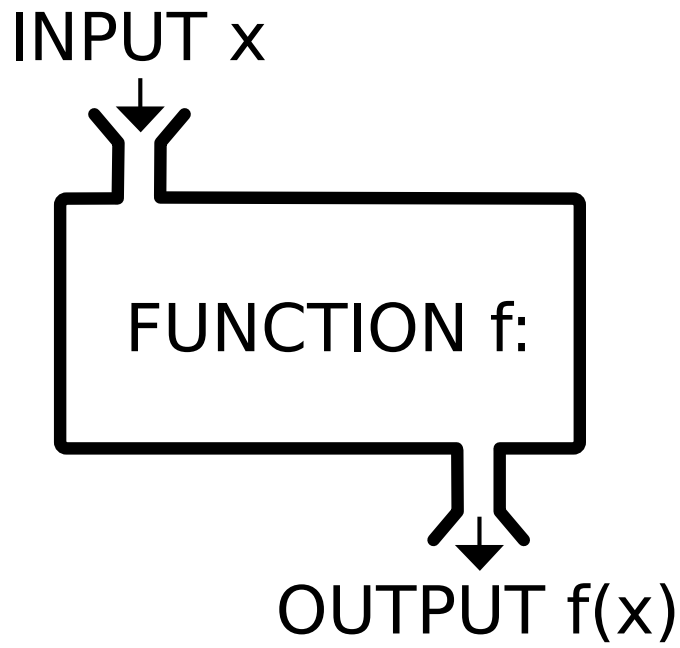
nu asociază o valoare fiecărui element

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 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):

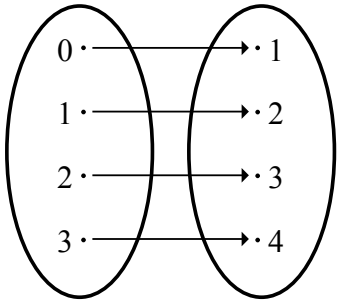
`let y = 3` definește identificatorul y cu valoarea 3 (un întreg)

În general `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.

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`

Ultima scriere e mai concisă și apropiată celei din matematică.
Prima scriere ne arată că o funcție e și ea o *valoare* (ca și întregii, realii, etc.) și poate fi folosită la fel cu alte valori.

Apelul de funcție

Dacă am definit o funcție: `let f x = x + 3`
o apelăm (calculăm valoarea) scriind funcția, apoi argumentul: `f 2`

Interpretorul răspunde: `- : int = 5`
avem o valoare fără nume (–), care e un întreg, și are valoarea 5

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.

Putem scrie f(2), la fel ca ((2))+(3), dar e inutil și derutant.
Diverse limbaje au reguli de scris diferite (sintaxa).

Putem apela direct și o funcție anonimă: `(fun x -> x + 3) 2`
(cu paranteze pentru a grupa expresia funcției anonime)

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*)

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

`int -> int` e tot un tip, al 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 $abs : \mathbb{Z} \to \mathbb{Z}$ $\quad abs(x) = \begin{cases} x & \text{dacă } x \geq 0 \\ -x & \text{altfel } (x < 0) \end{cases}$

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

```
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*.

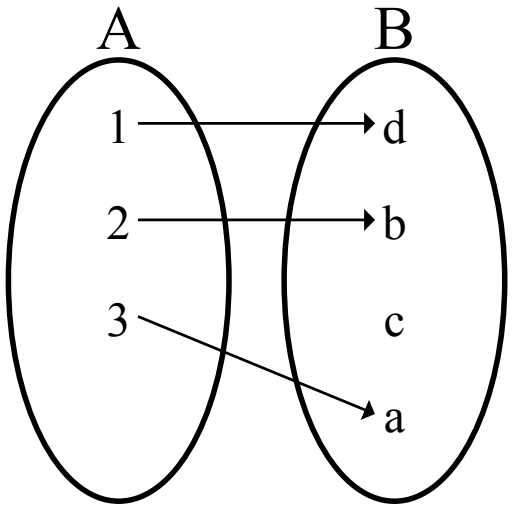
În ML, `if` e o *expresie*. ML nu are instrucțiuni, ci doar *expresii* (care sunt evaluate), și *definiții* (`let`) care dau nume unor valori.
Vom lucra mai târziu și cu definiții de tipuri și de module.

Funcții injective

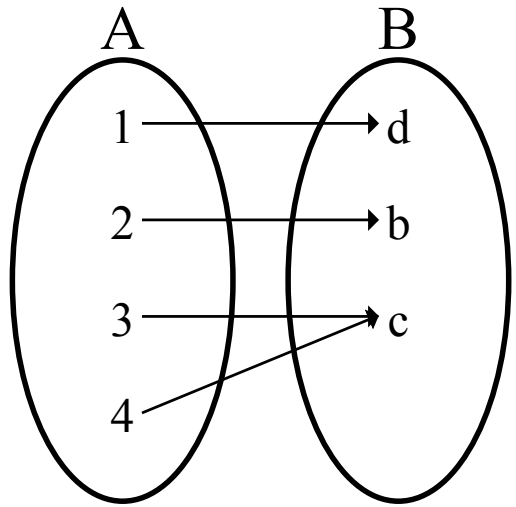
Def.: O funcție $f : A \to B$ e *injectivă* dacă asociază valori diferite la argumente diferite.

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

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

contrapozitiva afirmației de mai sus: negăm premisa și concluzia, și le inversăm: $P \Rightarrow Q \quad \Leftrightarrow \quad \neg Q \Rightarrow \neg P$

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

– presupunem concluzia falsă
– arătăm că atunci premisa e falsă, absurd (știm că e adevărată)
– deci concluzia nu poate fi falsă, e adevărată

E la fel să scriem $x_1, x_2 \in A,\ x_1 = x_2 \Rightarrow\ f(x_1) = f(x_2)$?
Nu! *Oricare ar fi funcția*, dând același argument ia aceeași valoare! (e o proprietate de bază a egalității și substituției).

Proprietăți ale funcțiilor injective

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).

Demonstrăm prin inducție după n că dacă $|A| > |B| = n$, $f : A \to B$ nu poate fi injectivă.

Cazul de bază: $n = 1$, $B = \{b_1\}$. Cum A are cel puțin 2 elemente, avem $f(a_1) = f(a_2) = b_1$ (unica posibilitate), deci f nu e injectivă.

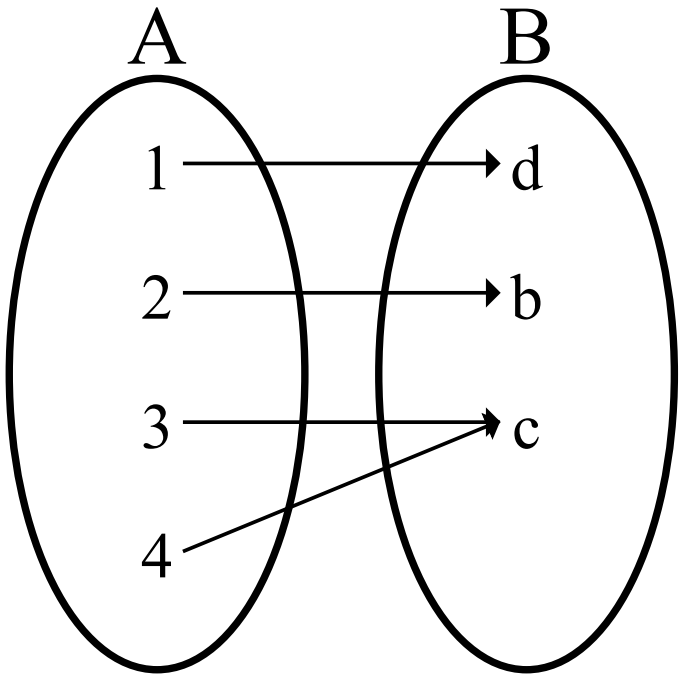
Cazul inductiv: fie $|B| = n + 1$ și $b_{n+1} \in B$. Dacă pentru cel puțin 2 elemente din A, f ia valoarea b_{n+1}, f nu e injectivă.
Altfel, eliminăm din A aceste elemente (cel mult 1), și codomeniul devine $B' = B \setminus \{b_{n+1}\}$. Rămânem cu $|A'| > |B'| = n$. Din ipoteza inductivă, cel puțin două elemente din A' au valori egale pentru f.

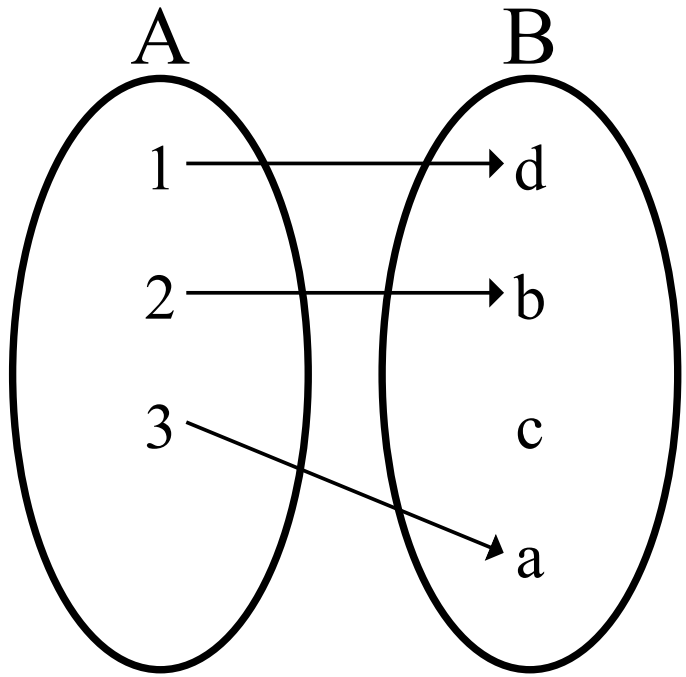
Similar, *principiul lui Dirichlet*: dacă împărțim $n + 1$ obiecte în n categorii există cel puțin o categorie cu mai mult de un obiect.

Funcții surjective

Def.: 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.

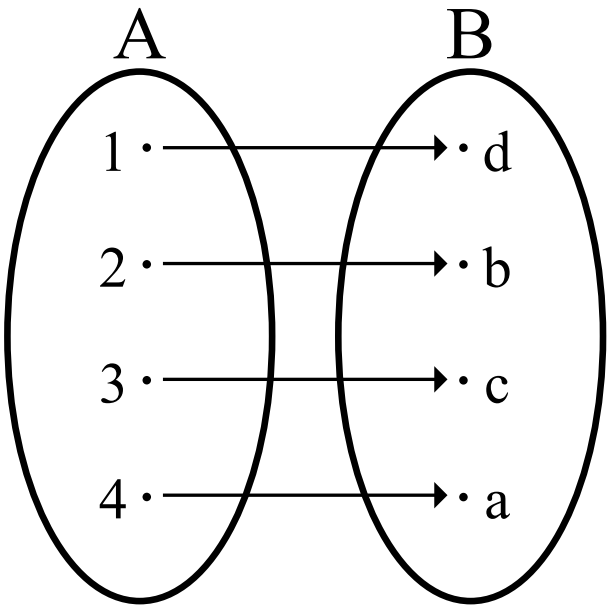
În programare, e util să definim funcția cu tipul rezultatului cât mai precis (dacă e posibil, surjectivă).

Astfel, când citim un program, știm deja din tipul funcției ce valori poate returna, fără a trebui să-i examinăm codul.

Funcții bijective

Def.: 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`

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.

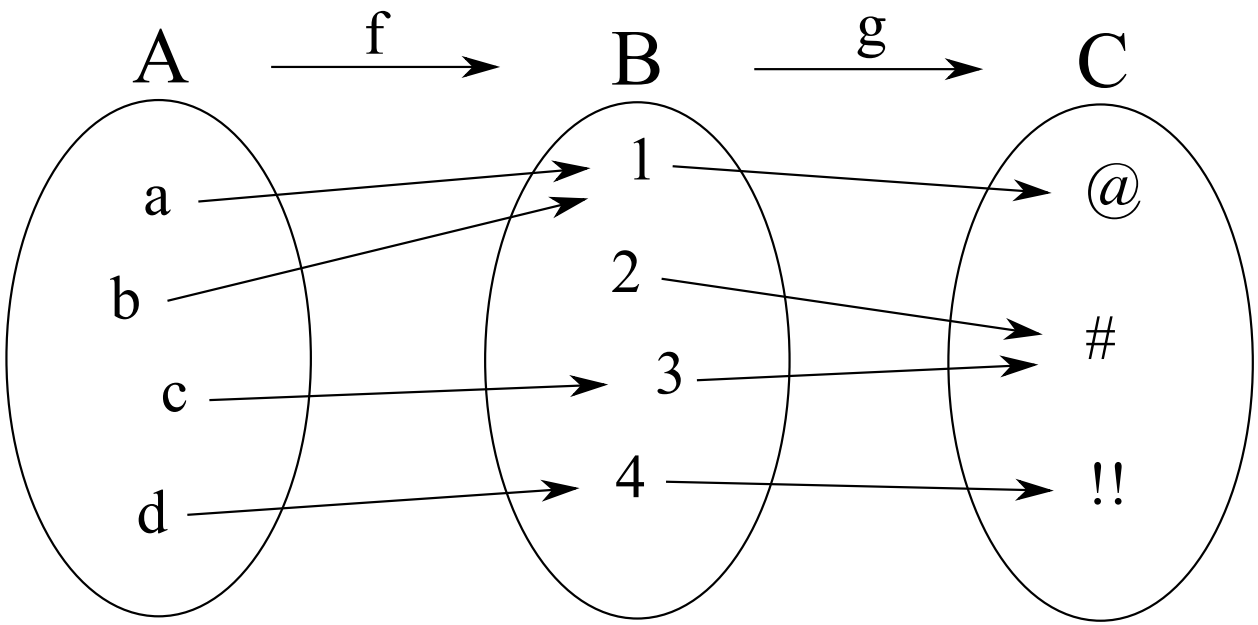
Mulțimea funcțiilor $f : A \to B$ se notează uneori B^A

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

Compunerea funcțiilor

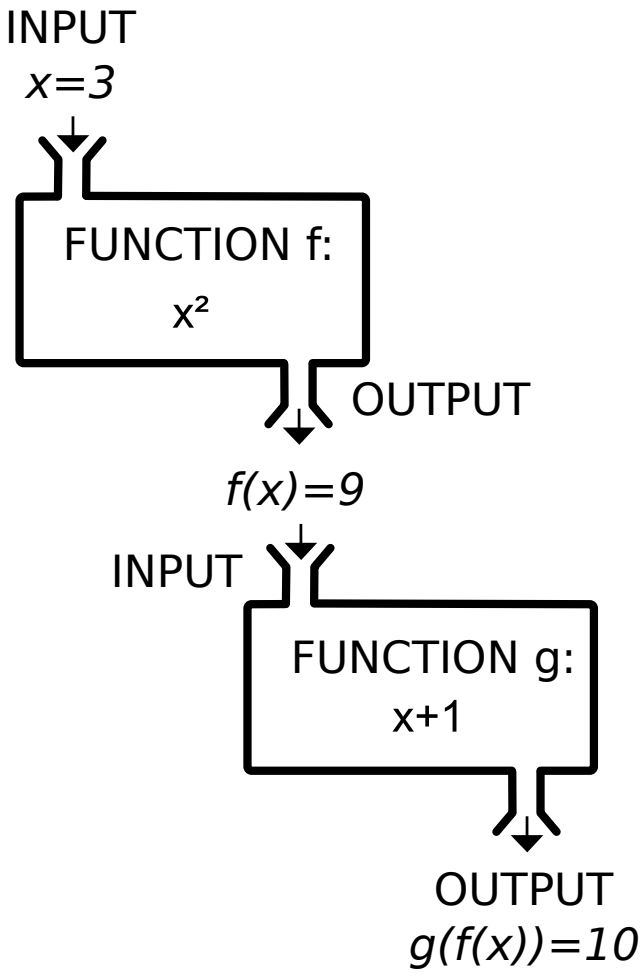
Fie functiile $f : A \to B$ și $g : B \to C$.
Compunerea lor este funcția $g \circ f : A \to C$, $(g \circ f)(x) = g(f(x))$.

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



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

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`

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:

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

$\Rightarrow$ Prelucrăm formulele atent, rescriindu-le conform definiției.

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

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

Funcții cu mai multe argumente

Matematic, scriem de ex. $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, $f(x, y) = 2x + y - 1$

În ML, enumerăm doar argumentele (fără paranteze, fără 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 dă un întreg.

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

adică

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

Interpretorul indică

```
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 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`

Funcții inversabile

Pe orice mulțime A definim *funcția identitate* $id_A : A \to A$,
$id_A(x) = x$ (notată adeseori și $\mathbf{1}_A$)

Def.: 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 $f(x_1) = y = f(x_2)$, 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 preimaginea (încercând să aflăm din ce argument a provenit) se pierde precizie
(în general, nu orice calcul e reversibil).

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.

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