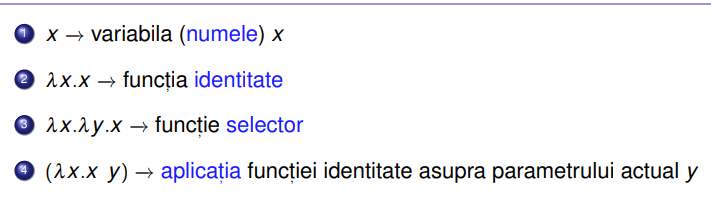
**Calculul Lambda** - axat pe conceptul matematic de funcție => **TOTUL ESTE O FUNCȚIE**

Aplicații

* programare
* demonstrarea formală a corectitudinii programelor

Bază teoretică pentru multe limbaje - LISP, Scheme, Haskell, ML, F#, Clean, Clojure, Scala, Erlang

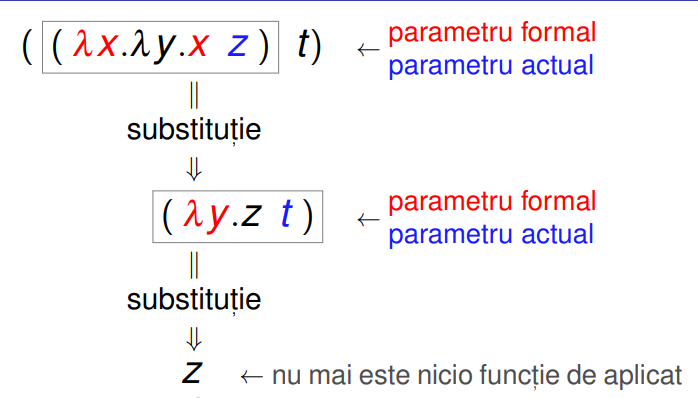
**Lambda expresii**



**Definiții**

**Funcție**: dacă x este o variabila și E este o λ-expresie, atunci λx.E este o λ-expresie, reprezentând funcția anonimă, unară, cu parametrul formal x șii corpul E;

**Aplicație**: dacă F și A sunt λ-expresii, atunci (F A) este o λ-expresie, reprezentând aplicația expresiei F asupra parametrului actual A.



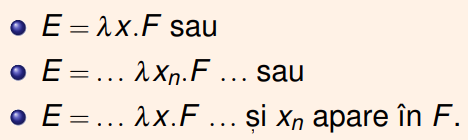
**β-redex** - Lambda expresie de forma -

E = lambda expresie = corpul funcției

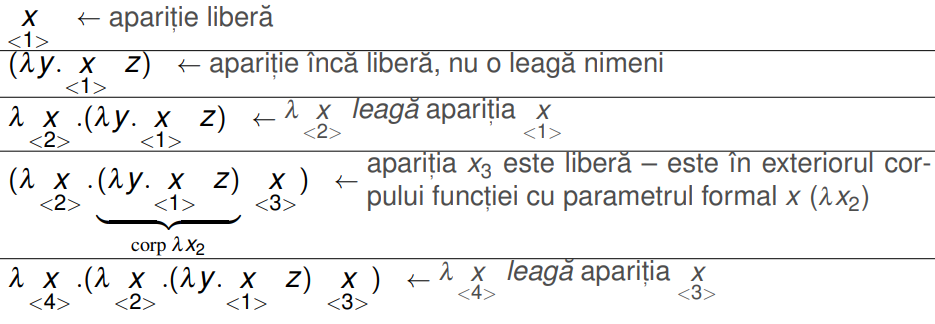
A = lambda expresie = parametrul actual

Beta - redex =>  => E cu toate aparițiile libere ale lui x din E înlocuite cu A prin substituție textuală.

**Apariție legată** - O apariție xn a unei variabile x este legată într-o expresie E dacă:

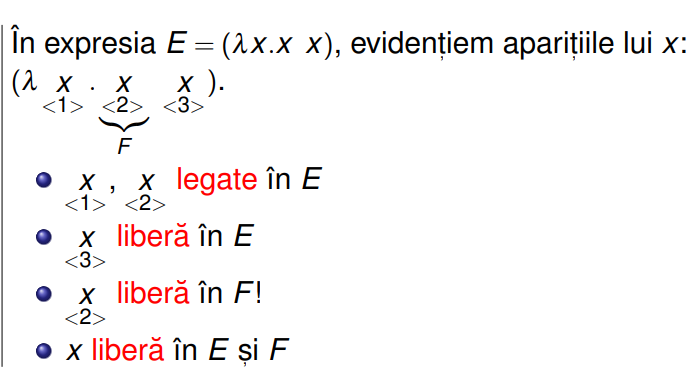


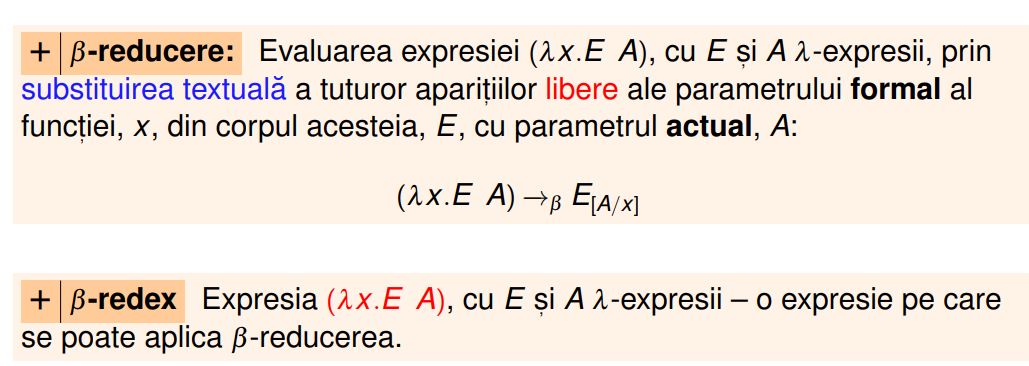
**Apariție liberă** - O apariție a unei variabile este liberă într-o expresie dacă nu este legată în acea expresie.

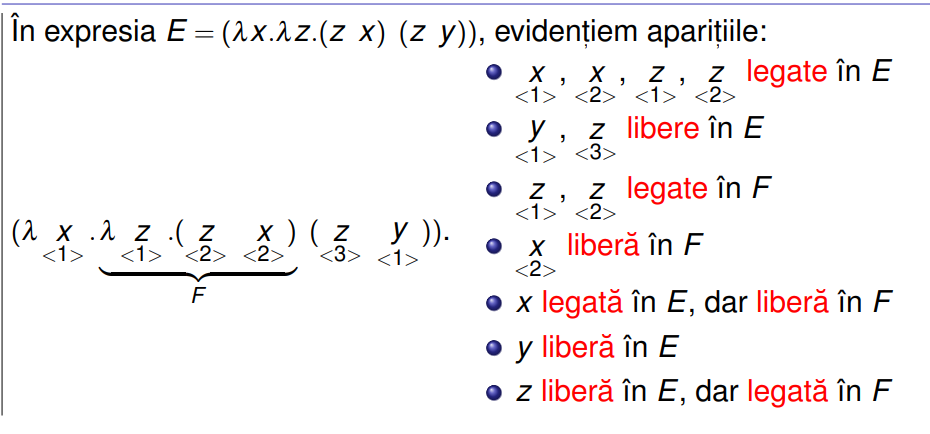


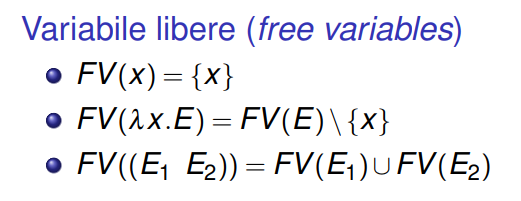
**Variabilă legată** - toate aparițiile sale în expresie sunt legate

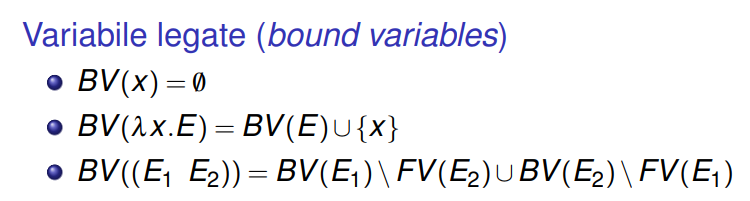
**Variabilă liberă** - cel puțin o apariție este liberă





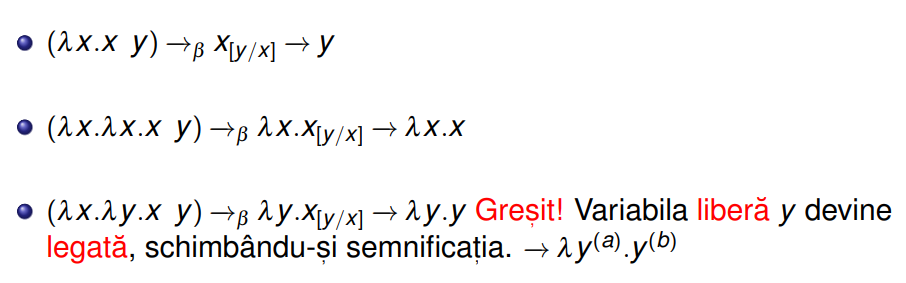


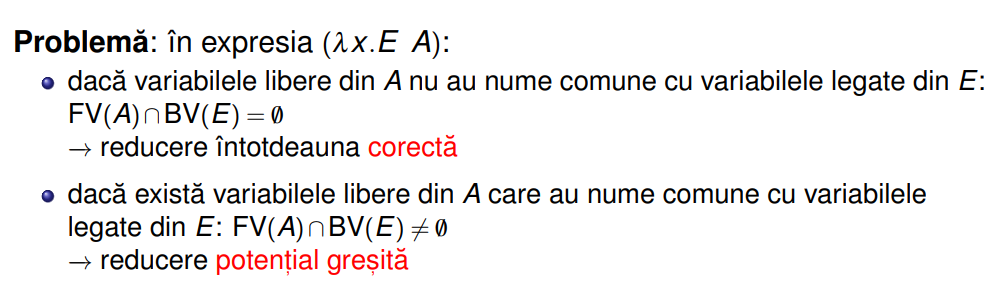




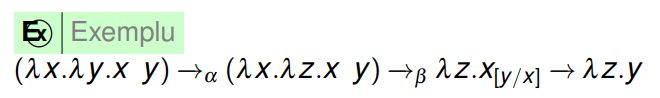
**Expresie închisă** - expresie care nu conține variabile libere. Înaintea evaluării, o expresie trebuie adusă la forma închisă.

**Exemple beta reducere**

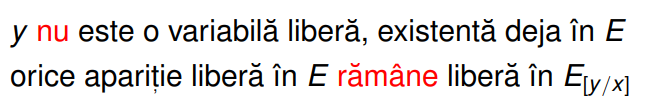




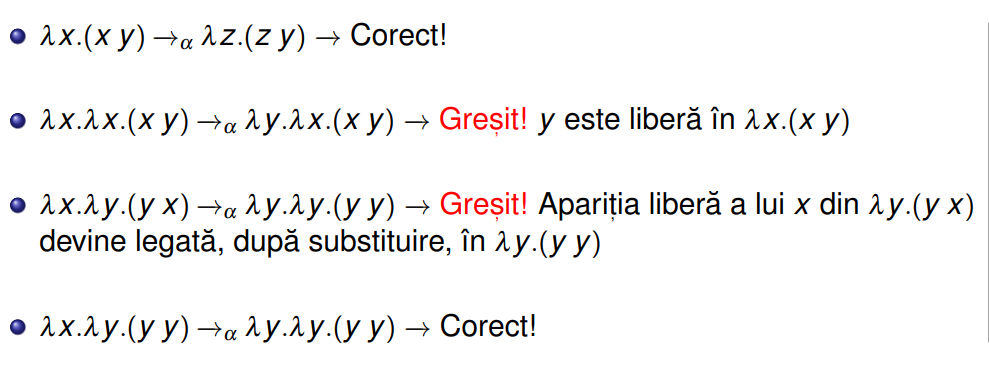
**Alfa conversie** - Redenumirea variabilelor legate din E, ce coincid cu cele libere din A



Condiții pentru alfa conversia  sunt:



Exemple alfa-conversie



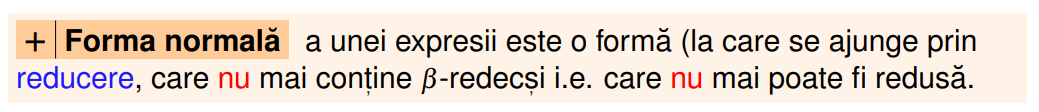
**Pas de reducere** = alfa-conversie + beta-reducere 

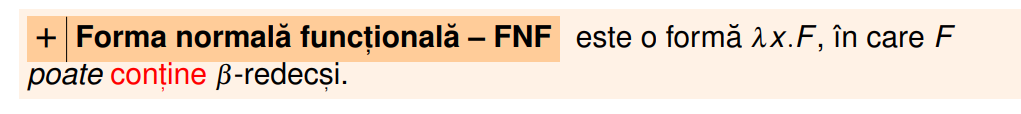
**Secvență de reducere** = succesiune de pași de reducere 

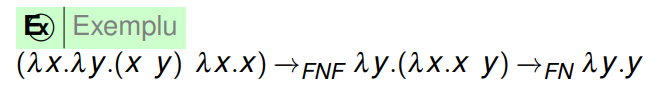
Proprietăți reducere



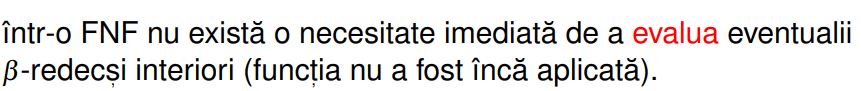
**Expresie reductibilă** - expresie care admite (cel puțin o) secvență de reducere care se termină





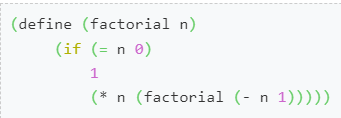


* FN expresie fixă este și FNF



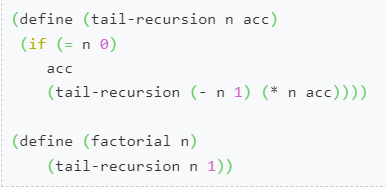
**Racket**

**Recursivitate pe stivă** - apelul recursiv este parte a unei expresii mai complexe, fiind necesară reținerea de informații, pe stivă, pe avansul în recursivitate.



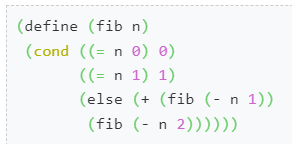
**Recursivitate pe coadă** - valoarea întoarsă de apelul recursiv constituie valoarea de retur a apelului curent, i.e. apelul recursiv este un tail call, nefiind necesară reținerea de informație pe stivă. Nu mai este necesară stocarea stării fiecărei funcții din apelul recursiv, spațiul utilizat fiind O(1).

Metoda de transformare prezentată în laborator constă în utilizarea unui **acumulator**, ca parametru al funcției, în care rezultatul final se construiește treptat, pe avansul în recursivitate, în loc de revenire.



**Diferența** dintre cele două tipuri de funcții constă în necesitatea funcțiilor **recursive pe stivă** de a se întoarce din recursivitate pentru a prelucra rezultatul, respectiv capacitatea funcțiilor **recursive pe coadă** de a genera un rezultat pe parcursul apelului recursiv.

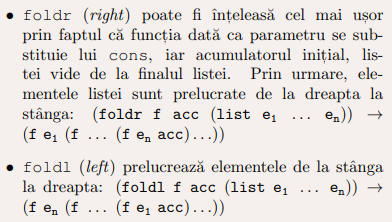
**Recursivitate arborescentă** - în cazul funcțiilor care conțin, în implementare, cel puțin două apeluri recursive care se execută necondiționat.



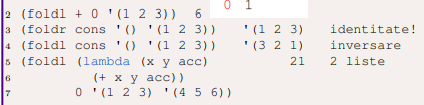
**Funcționale**

**foldl + foldr**

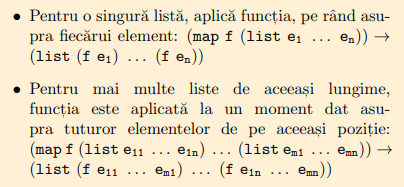
Îmbină toate elementele unei liste pentru a construi o valoare finală, pornind de la un acumulator inițial. Într-un pas, funcția dată ca parametru combină elementul curent din listă cu acumulatorul, întorcând un nou acumulator. Acumulatorul final este întors ca rezultat al funcționalelor fold\*. Acesta poate fi chiar o listă.

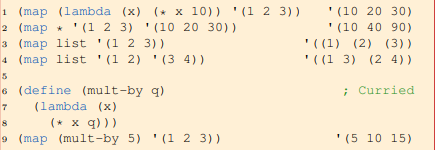




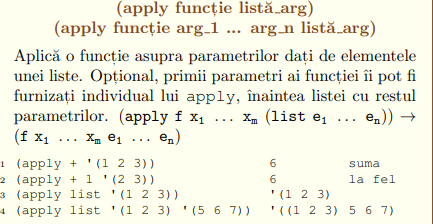


**map**

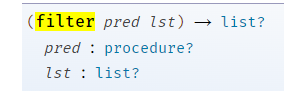




**apply**

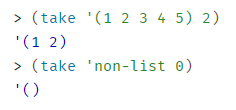


**filter**





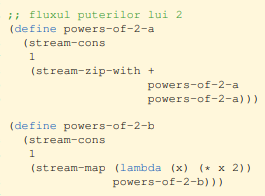
**Alte funcții useful RACKET**

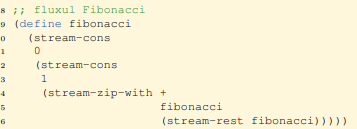


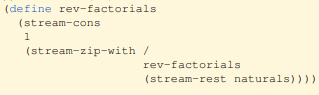
**FLUXURI**



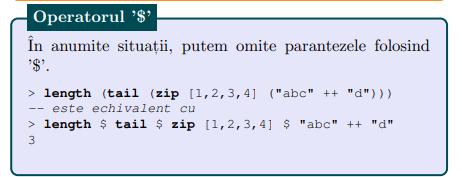
Exemple fluxuri

****

****

****

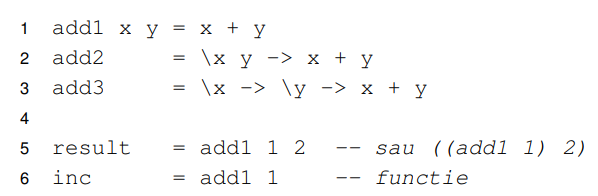
**Haskell**



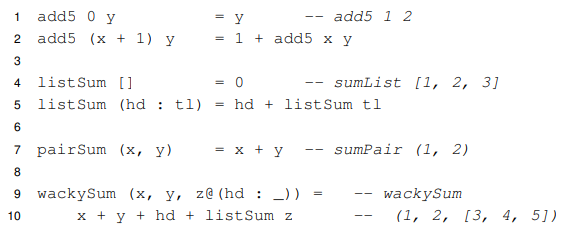
**Operatorul .** -> (f . g)(x) = f(g(x))

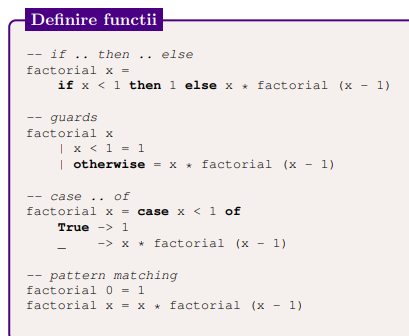


Funcțiile sunt aplicabile asupra oricâtor parametri la un moment dat.

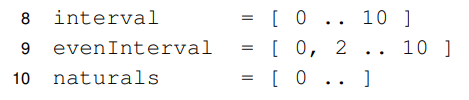


**Pattern matching** - definirea comportamentului funcțiilor pornind de la structura parametrilor

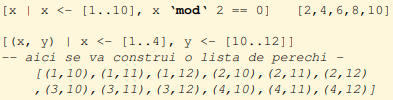




**List comprehensions** - definirea listelor prin proprietățile elementelor, similar unei specificații matematice.

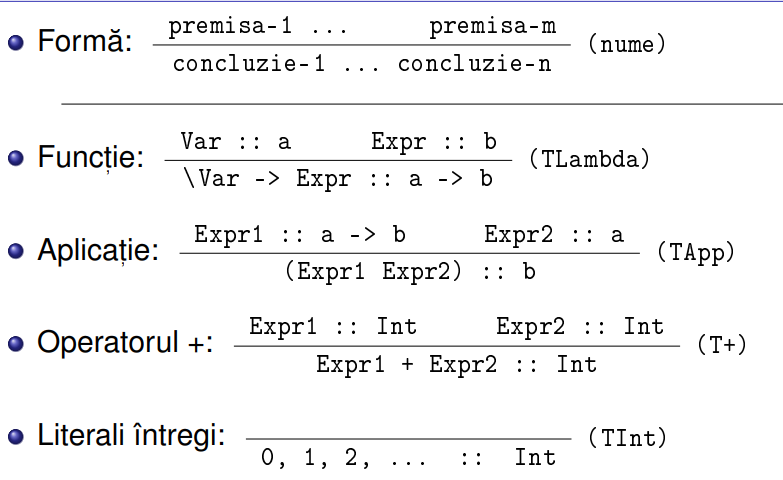


Se generează elemente la infinit. Pentru a putea vedea o porțiune a fluxului folosim funcțiile take și drop.



**Evaluare leneșă** - parametri evaluați la cerere, cel mult o dată, eventual parțial, în cazul obiectelor structurate.

**Sinteză de tip** - Determinarea automată tipului unei expresii, pe baza unor reguli precise



**Funcții Haskell**

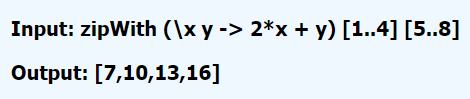
* **zip**

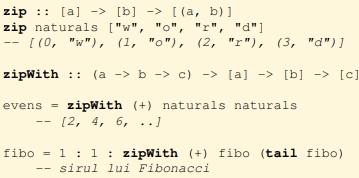
TIP : **zip :: [a] -> [b] -> [(a, b)]**

**zip [x,y,z] [a,b] ≡ [(x,a),(y,b)]**

* **zipWith**

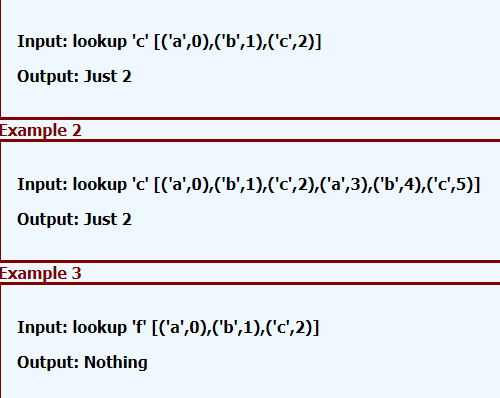
TIP : **(a -> b -> c) -> [a] -> [b] -> [c]**





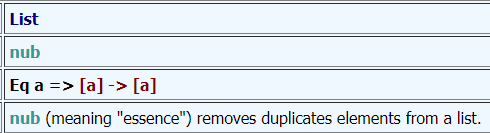
* **lookup**

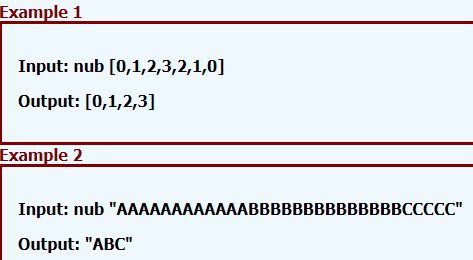
TIP: **Eq a =>** [**a -> [(a,b)] -> Maybe b**](http://zvon.org/other/haskell/Outputprelude/aQ-NQRTa,bUSQ-NQMaybeQb_t.html)



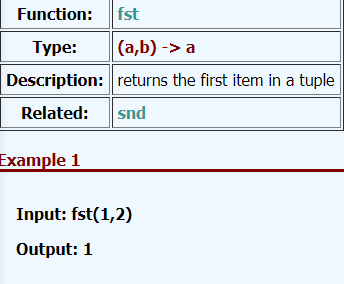
Makes a list, its elements are calculated from the function and the elements of input lists occuring at the same position in both lists. (e de pe un site, mi-a fost lene să traduc)

* **nub**

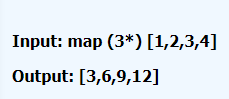
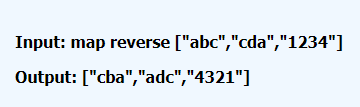




* **fst**



* **map = (a->b) -> [a] -> [b]**



* **filter :: (a -> Bool) -> [a] -> [a]**

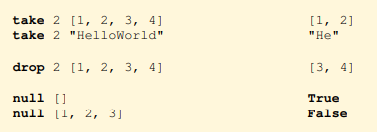
****

* **foldl :: (a -> b -> a) -> a -> [b] -> a**

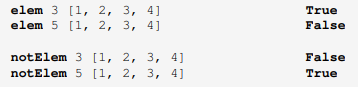
****

* **head - [a] -> a**
* **take, drop, null**

take :: [**Int -> [a] -> [a]**](http://zvon.org/other/haskell/Outputprelude/IntQ-NQRaSQ-NQRaS_t.html)

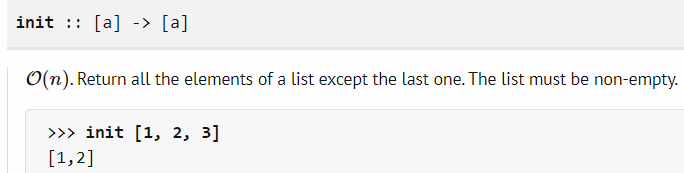


* **elem, notElem**

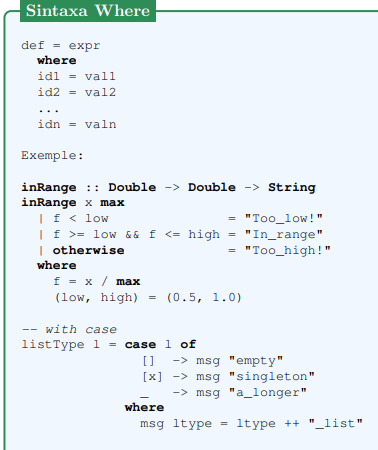


elem :: **Eq a =>** [**a -> [a] -> Bool**](http://zvon.org/other/haskell/Outputprelude/aQ-NQRaSQ-NQBool_t.html)

* init



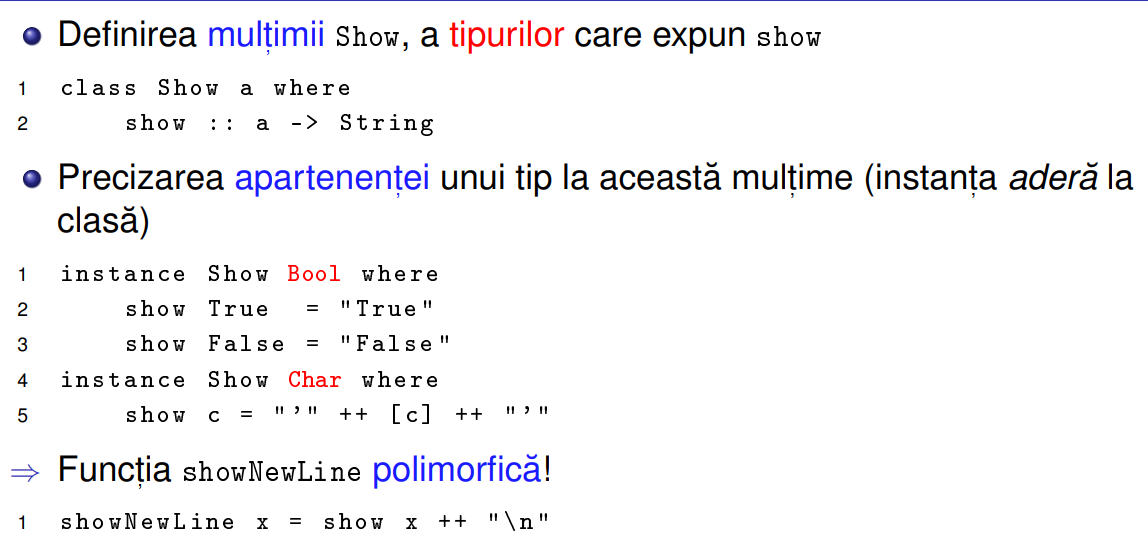
**SINTAXĂ WHERE**

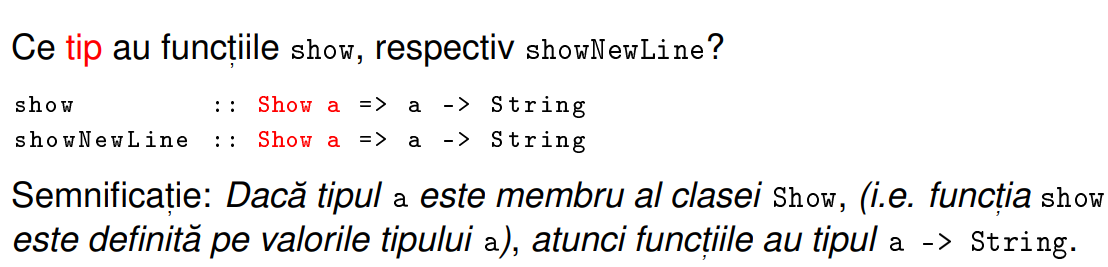


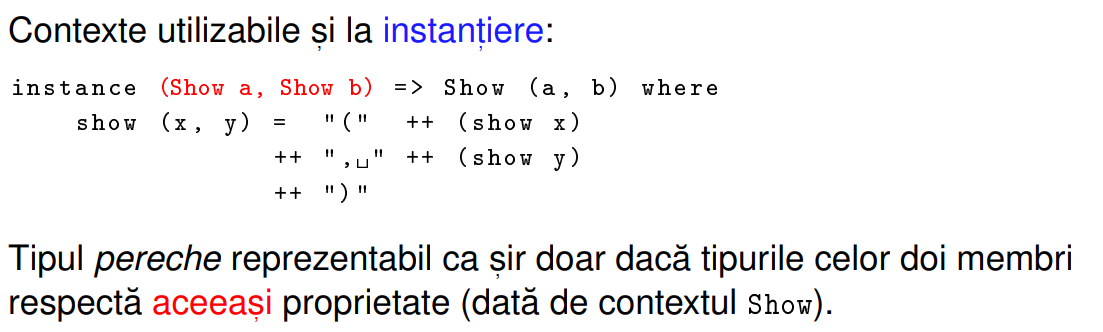
**Polimorfism parametric vs ad-hoc**



**Clasă** - mulțime de tipuri, care necesită implementarea unor funcții.

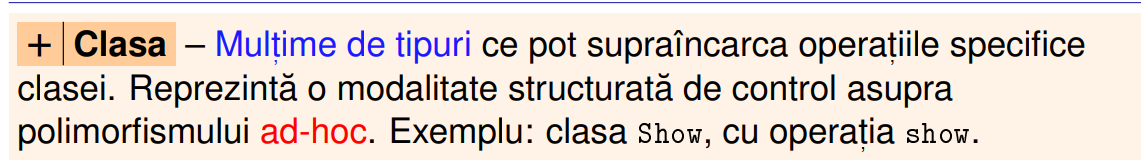




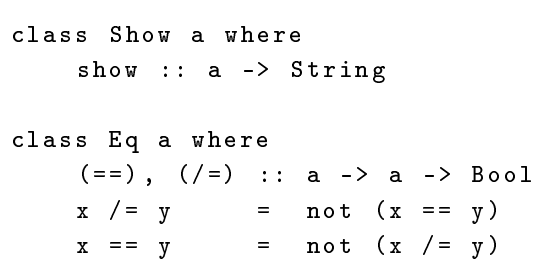


**Tipurile** -> mulțimi de valori

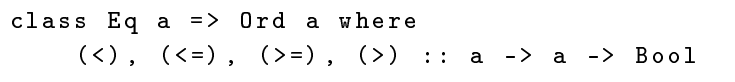
**Instanțierea claselor** de către tipuri pentru ca funcțiile definite în clasa să fie disponibile pentru valorile tipului.



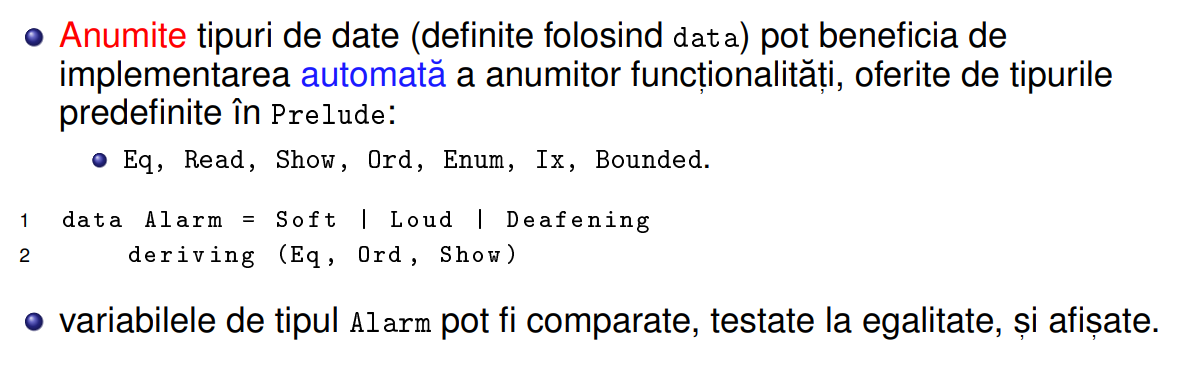




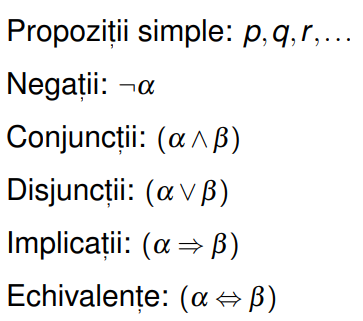
Pentru o instanțiere corectă este necesară suprascrierea cel puțin unui operator (== sau /=).

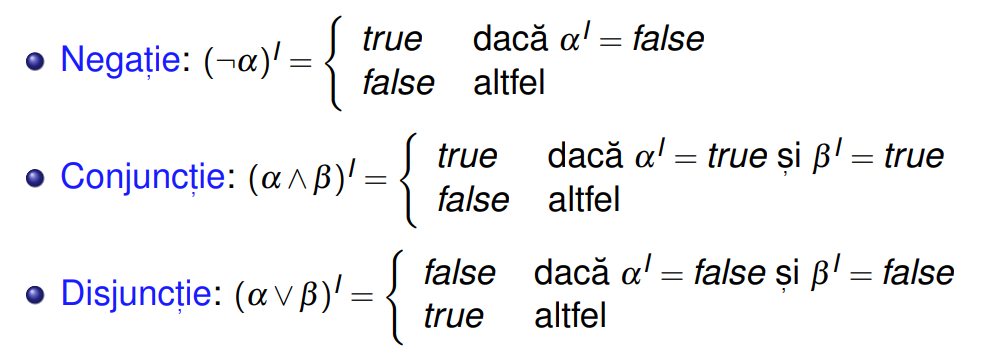


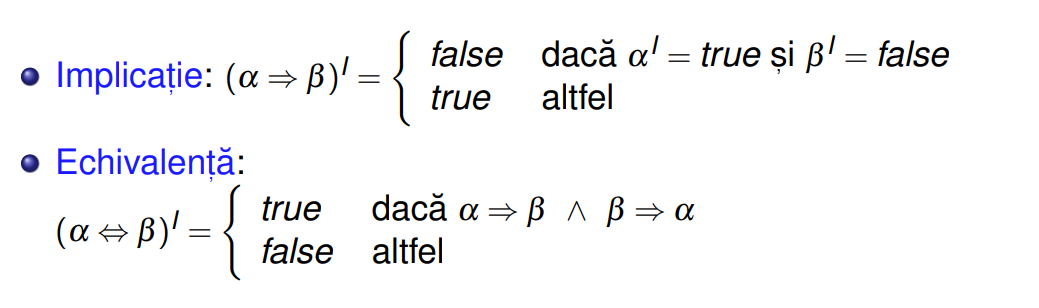
Clasa Ord moștenește clasa Eq, cu preluarea operațiilor din clasa moștenită. Este necesară aderarea la clasa Eq, în momentul instanțierii clasei Ord.



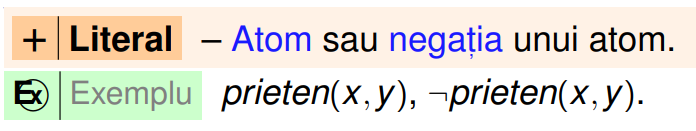
**Logica propozițională**

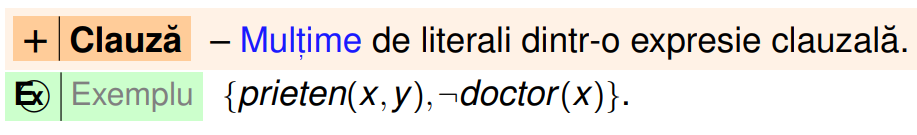
****

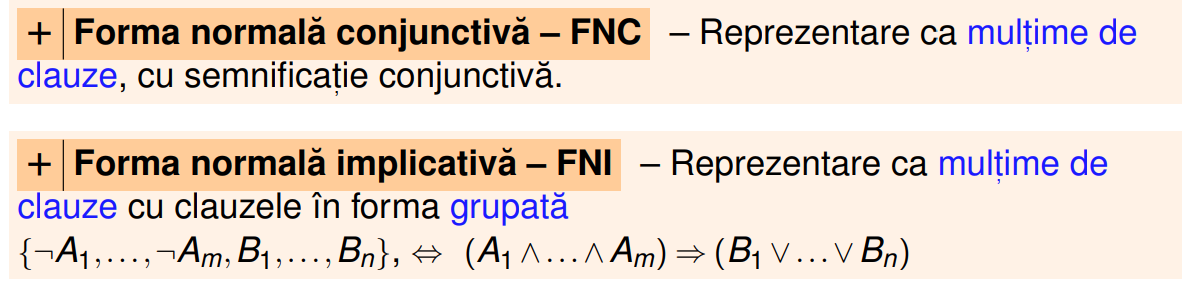
****

****

**Definiții**





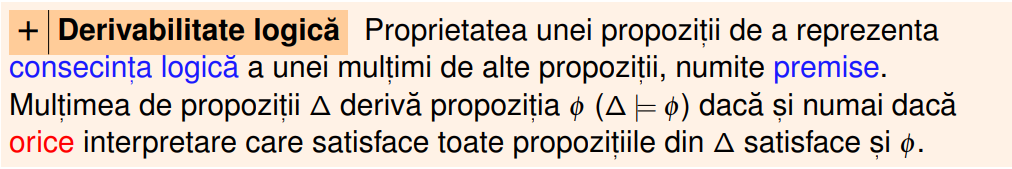


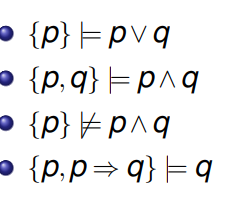
**Satisfiabilitate** - Proprietatea unei propoziții care este adevărată sub cel puțin o interpretare.

**Validitate** - Proprietatea unei propoziții care este adevărată în toate interpretările.

(tautologie)

**Nesatisfiabilitate** - Proprietate unei propoziții care este falsă în toate interpretările (contradicție)

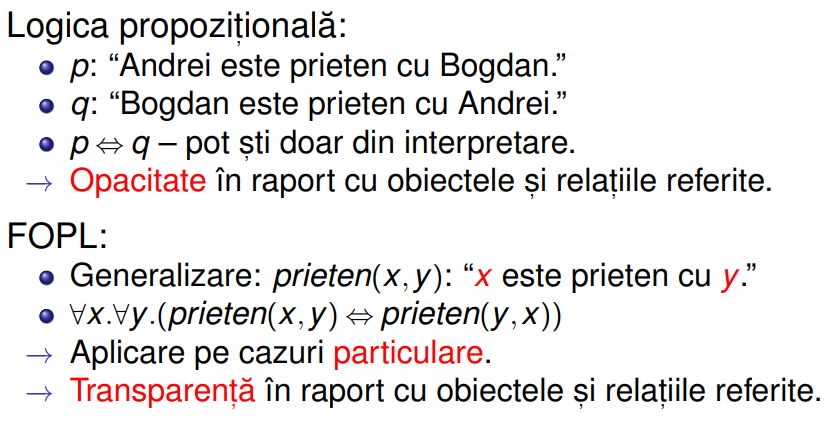


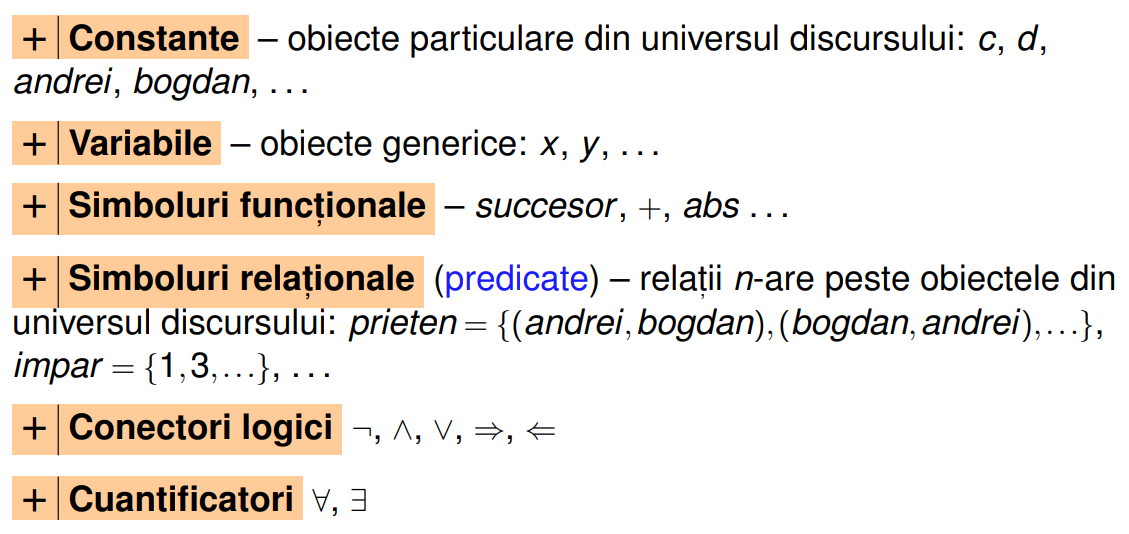


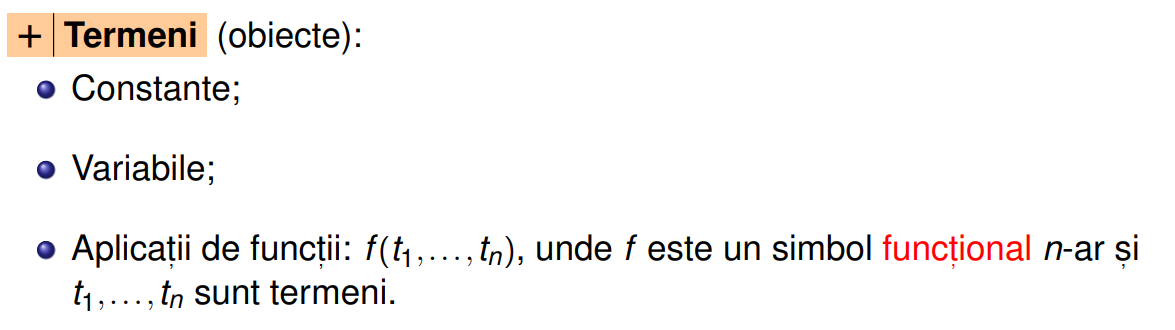
**Inferența** - derivarea mecanică a concluziilor unui set de premise

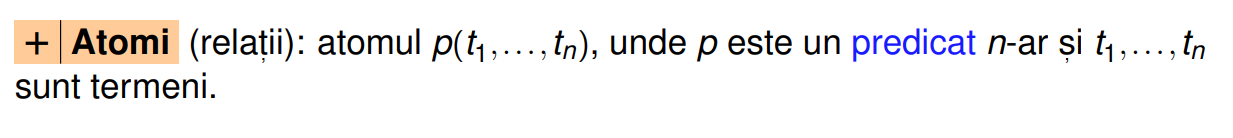
**FOPL - First Order Predicate Logic**

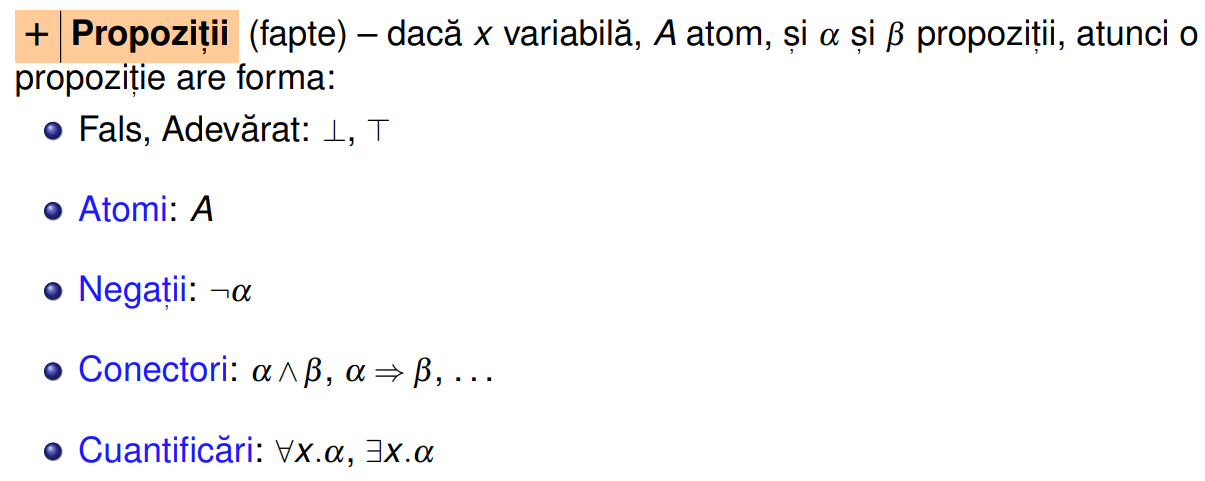
**Logica cu predicate de ordinul I**

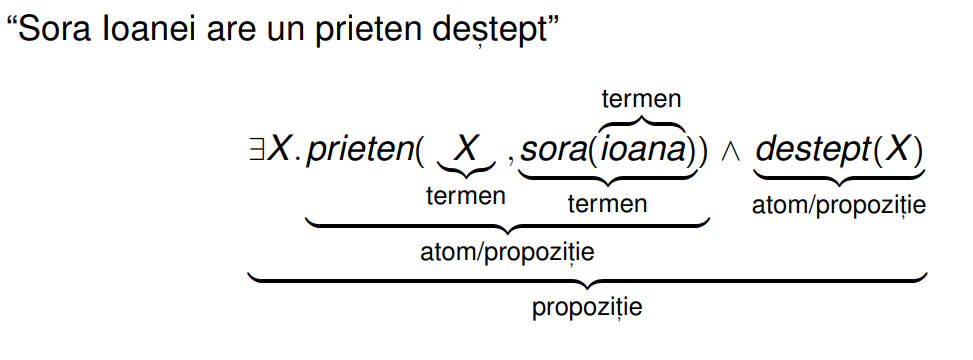


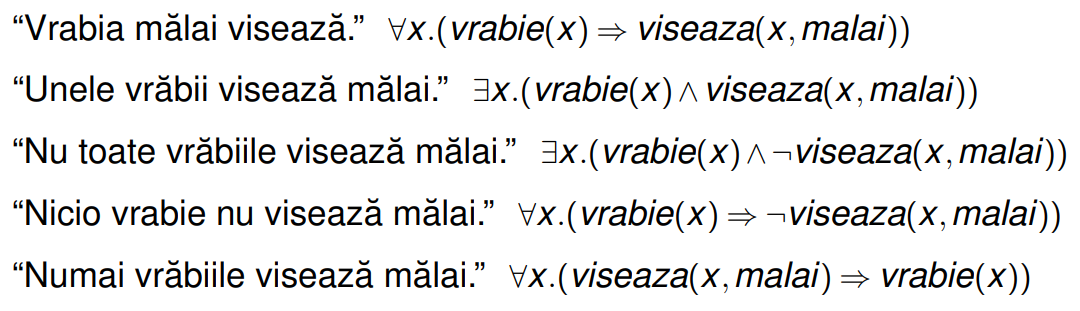


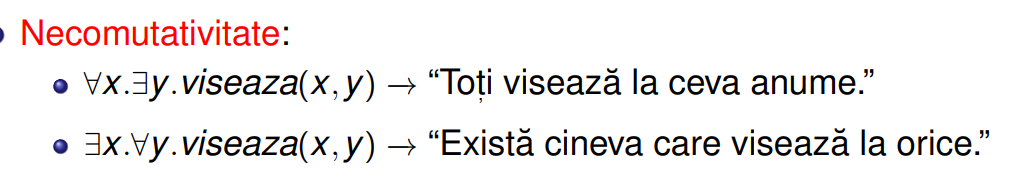


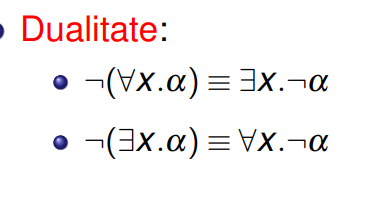


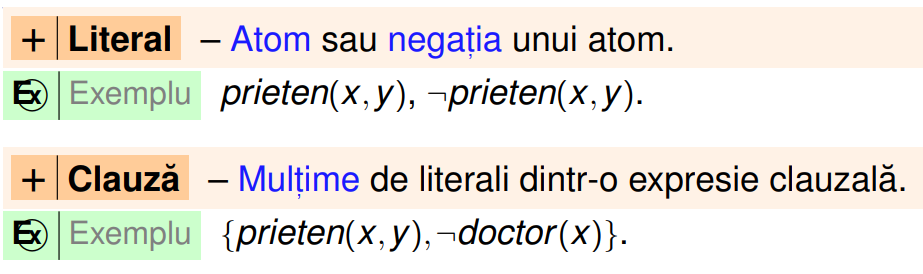


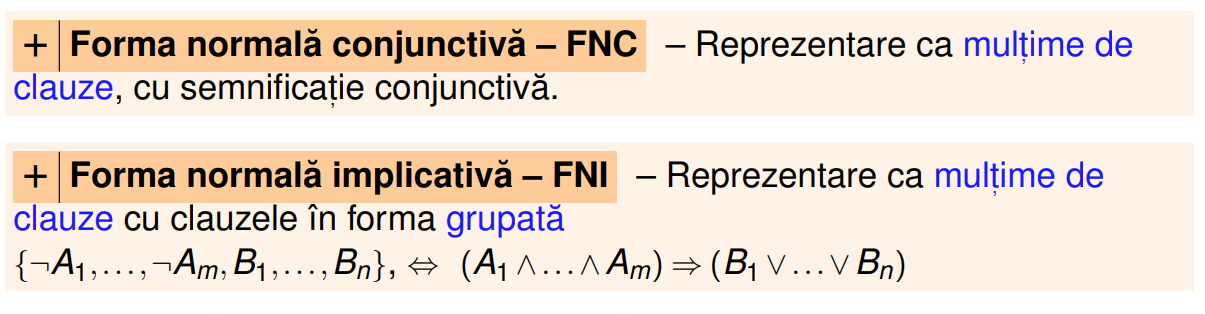


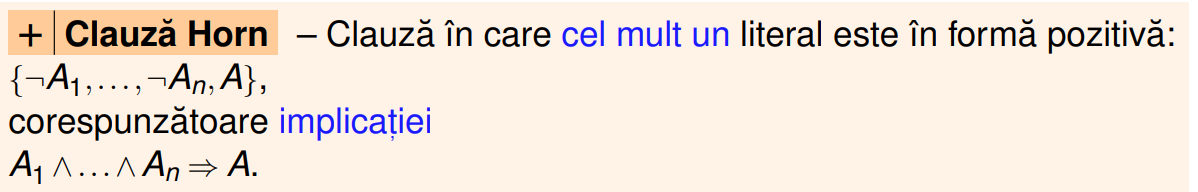


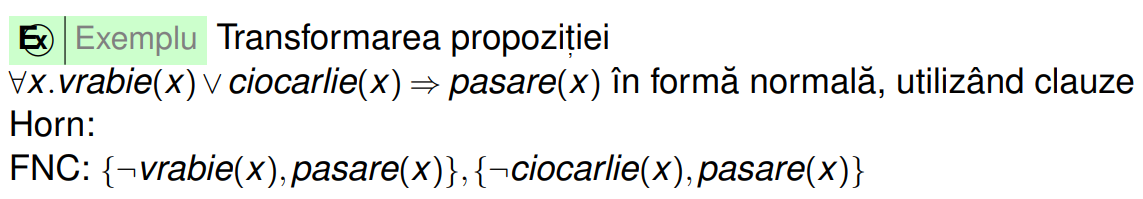




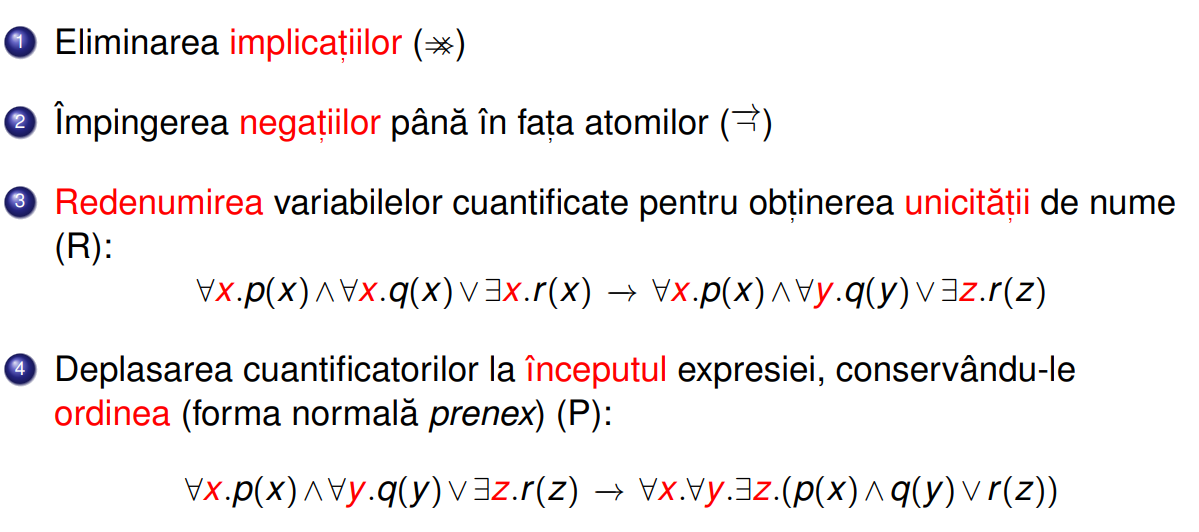


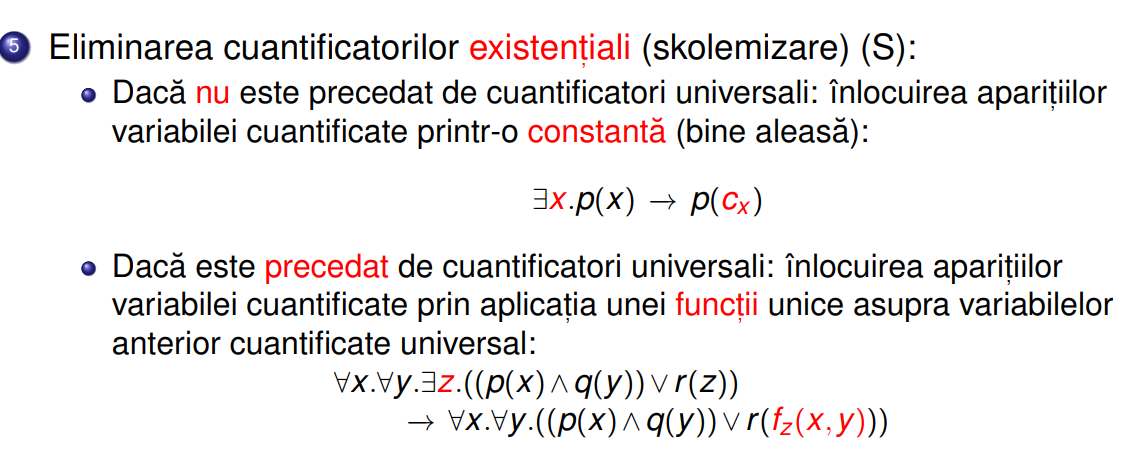


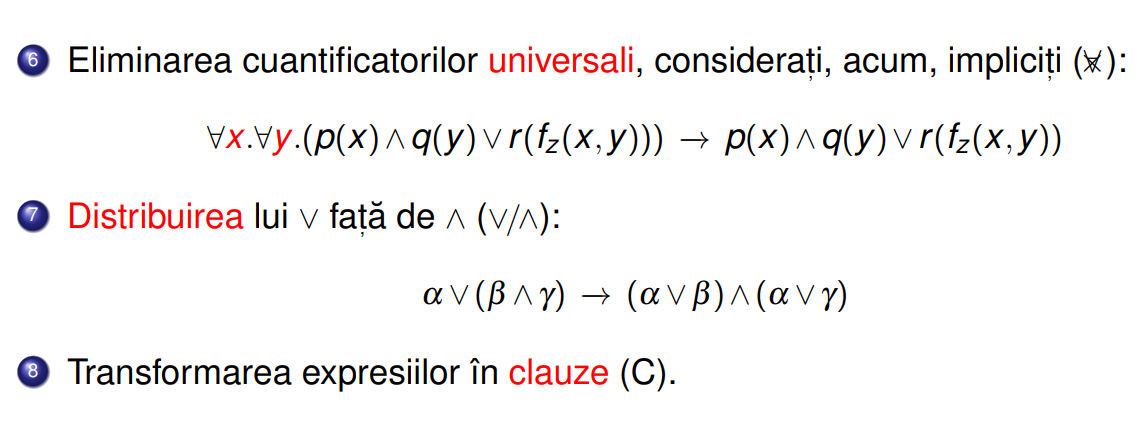




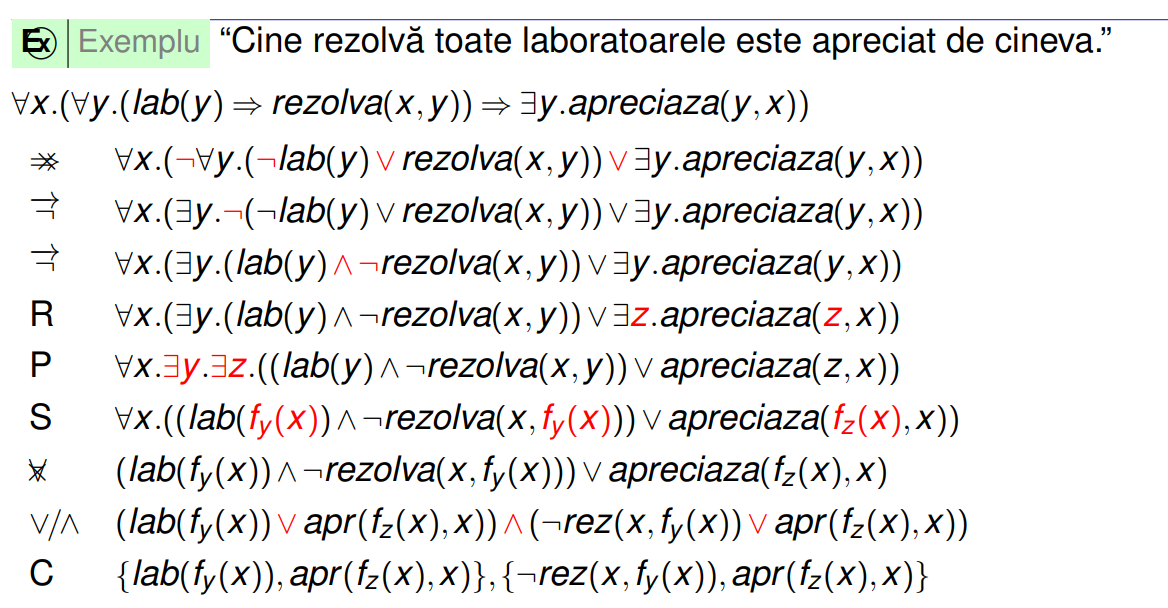
**Conversia propozițiilor în FNC**





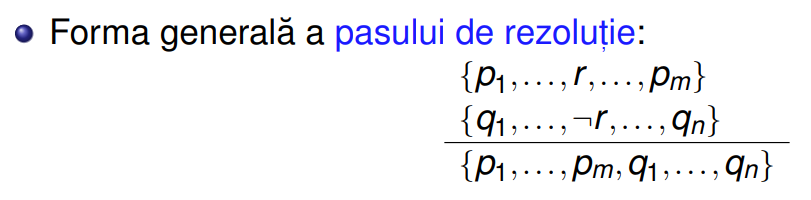


Exemplu



**REZOLUȚIE**

Principiu de bază −→ pasul de rezoluție

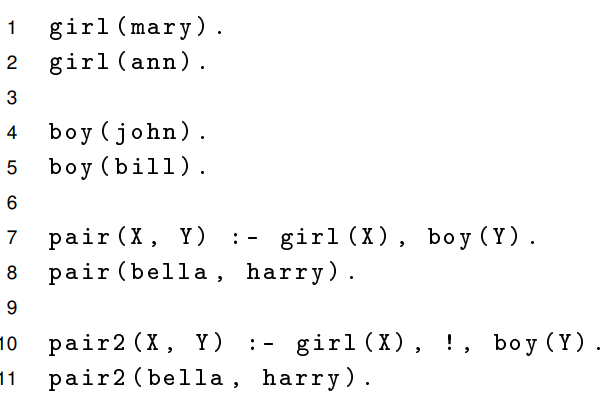


**PROLOG**

**Operatorul Cut = !**

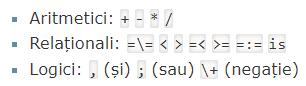
Prima întâlnire -> satisfacere

A doua întâlnire în momentul revenirii (backtracking) -> eșec





Operatori



Operatorii **=:=** și **is** forțează evaluarea unei expresii, pe când **=** verifica doar egalitatea structurală.

**is** și **=** pot primi variabile neinstanțiate pe care le instanțiază (is doar în partea stângă).

