CHEATSHEET PP 2022

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

- x → variabila (numele) x
- $\lambda x.x$ → funcția identitate
- $\lambda x.\lambda y.x$ → funcție selector
- $(\lambda x.x\ y)$ → aplicația funcției identitate asupra parametrului actual y

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.

$((\boxed{\lambda x.\lambda y.x\ z})\ t)$ ← parametru formal / parametru actual

‖ substituție ⇓

$(\boxed{\lambda y.z\ t})$ ← parametru formal / parametru actual

‖ substituție ⇓

z ← nu mai este nicio funcție de aplicat

β-redex - Lambda expresie de forma -

$$(\lambda x.E\ \ A)$$

E = lambda expresie = corpul funcției
A = lambda expresie = parametrul actual

Beta - redex => $E_{[A/x]}$ => 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:

- $E = \lambda x.F$ sau
- $E = \ldots\ \lambda x_n.F\ \ldots$ sau
- $E = \ldots\ \lambda x.F\ \ldots$ și x_n apare în F.

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

$\underset{<1>}{x}$ ← apariție liberă

$(\lambda y.\ \underset{<1>}{x}\ \ z)$ ← apariție încă liberă, nu o leagă nimeni

$\lambda\ \underset{<2>}{x}\ .(\lambda y.\ \underset{<1>}{x}\ \ z)$ ← $\lambda\ \underset{<2>}{x}$ *leagă* apariția $\underset{<1>}{x}$

$(\lambda\ \underset{<2>}{x}\ .\underbrace{(\lambda y.\ \underset{<1>}{x}\ \ z)}_{\text{corp } \lambda x_2}\ \ \underset{<3>}{x}\ )$ ← apariția x_3 este liberă – este în exteriorul corpului funcției cu parametrul formal x (λx_2)

$\lambda\ \underset{<4>}{x}\ .(\lambda\ \underset{<2>}{x}\ .(\lambda y.\ \underset{<1>}{x}\ \ z)\ \ \underset{<3>}{x}\ )$ ← $\lambda\ \underset{<4>}{x}$ *leagă* apariția $\underset{<3>}{x}$

Variabilă legată - toate aparițiile sale în expresie sunt legate

Variabilă liberă - cel puțin o apariție este liberă

În expresia $E = (\lambda x.x\ \ x)$, evidențiem aparițiile lui x:
$(\lambda\ \underset{<1>}{x}\ .\ \underbrace{\underset{<2>}{x}}_{F}\ \ \underset{<3>}{x}\ )$.

- $\underset{<1>}{x}$, $\underset{<2>}{x}$ legate în E
- $\underset{<3>}{x}$ liberă în E
- $\underset{<2>}{x}$ liberă în F!
- x liberă în E și F

+ | β-reducere: Evaluarea expresiei $(\lambda x.E\ \ A)$, cu E și A λ-expresii, prin substituirea textuală a tuturor aparițiilor libere ale parametrului **formal** al funcției, x, din corpul acesteia, E, cu parametrul **actual**, A:

$$(\lambda x.E\ \ A) \rightarrow_\beta E_{[A/x]}$$

+ | β-redex Expresia $(\lambda x.E\ \ A)$, cu E și A λ-expresii – o expresie pe care se poate aplica β-reducerea.

CHEATSHEET PP 2022

În expresia $E = (\lambda x.\lambda z.(z\ x)\ (z\ y))$, evidențiem aparițiile:

$$(\lambda \underset{<1>}{x} . \underbrace{\lambda \underset{<1>}{z} .(\underset{<2>}{z}\ \ \underset{<2>}{x})}_{F} (\underset{<3>}{z}\ \ \underset{<1>}{y})).$$

- $\underset{<1>}{x}, \underset{<2>}{x}, \underset{<1>}{z}, \underset{<2>}{z}$ legate în E
- $\underset{<1>}{y}, \underset{<3>}{z}$ libere în E
- $\underset{<1>}{z}, \underset{<2>}{z}$ legate în F
- $\underset{<2>}{x}$ liberă în F
- x legată în E, dar liberă în F
- y liberă în E
- z liberă în E, dar legată în F

Variabile libere (*free variables*)

- $FV(x) = \{x\}$
- $FV(\lambda x.E) = FV(E) \setminus \{x\}$
- $FV((E_1\ \ E_2)) = FV(E_1) \cup FV(E_2)$

Variabile legate (*bound variables*)

- $BV(x) = \emptyset$
- $BV(\lambda x.E) = BV(E) \cup \{x\}$
- $BV((E_1\ \ E_2)) = BV(E_1) \setminus FV(E_2) \cup BV(E_2) \setminus FV(E_1)$

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

Exemple beta reducere

- $(\lambda x.x\ \ y) \to_\beta x_{[y/x]} \to y$
- $(\lambda x.\lambda x.x\ \ y) \to_\beta \lambda x.x_{[y/x]} \to \lambda x.x$
- $(\lambda x.\lambda y.x\ \ y) \to_\beta \lambda y.x_{[y/x]} \to \lambda y.y$ Greșit! Variabila liberă y devine legată, schimbându-și semnificația. $\to \lambda y^{(a)}.y^{(b)}$

Problemă: în expresia $(\lambda x.E\ \ A)$:

- dacă variabilele libere din A nu au nume comune cu variabilele legate din E: $FV(A) \cap BV(E) = \emptyset$
  $\to$ reducere întotdeauna corectă
- dacă există variabilele libere din A care au nume comune cu variabilele legate din E: $FV(A) \cap BV(E) \neq \emptyset$
  $\to$ reducere potențial greșită

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

Exemplu

$(\lambda x.\lambda y.x\ \ y) \to_\alpha (\lambda x.\lambda z.x\ \ y) \to_\beta \lambda z.x_{[y/x]} \to \lambda z.y$

Condiții pentru alfa conversia $\lambda x.E \to_\alpha \lambda y.E_{[y/x]}$. sunt:

y nu este o variabilă liberă, existentă deja în E

orice apariție liberă în E rămâne liberă în $E_{[y/x]}$

Exemple alfa-conversie

- $\lambda x.(x\ y) \to_\alpha \lambda z.(z\ y) \to$ Corect!
- $\lambda x.\lambda x.(x\ y) \to_\alpha \lambda y.\lambda x.(x\ y) \to$ Greșit! y este liberă în $\lambda x.(x\ y)$
- $\lambda x.\lambda y.(y\ x) \to_\alpha \lambda y.\lambda y.(y\ y) \to$ Greșit! Apariția liberă a lui x din $\lambda y.(y\ x)$ devine legată, după substituire, în $\lambda y.(y\ y)$
- $\lambda x.\lambda y.(y\ y) \to_\alpha \lambda y.\lambda y.(y\ y) \to$ Corect!

Pas de reducere = alfa-conversie + beta-reducere

$$E_1 \to E_2 \equiv E_1 \to_\alpha E_3 \to_\beta E_2.$$

Secvență de reducere = succesiune de pași de reducere $E_1 \to^* E_2.$

Proprietăți reducere

- $E_1 \to E_2 \implies E_1 \to^* E_2$ – un pas este o secvență
- $E \to^* E$ – zero pași formează o secvență
- $E_1 \to^* E_2 \wedge E_2 \to^* E_3 \Rightarrow E_1 \to^* E_3$ – tranzitivitate

$((\lambda x.\lambda y.(y\ \ x)\ \ y)\ \ \lambda x.x) \to (\lambda z.(z\ \ y)\ \ \lambda x.x) \to (\lambda x.x\ \ y) \to y$
$\Rightarrow$
$((\lambda x.\lambda y.(y\ \ x)\ \ y)\ \ \lambda x.x) \to^* y$

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

+ **Forma normală** a unei expresii este o formă (la care se ajunge prin reducere, care nu mai conține β-redecși i.e. care nu mai poate fi redusă.

+ **Forma normală funcțională – FNF** este o formă $\lambda x.F$, în care F poate conține β-redecși.

Exemplu

$(\lambda x.\lambda y.(x\ \ y)\ \ \lambda x.x) \to_{FNF} \lambda y.(\lambda x.x\ \ y) \to_{FN} \lambda y.y$

- FN expresie fixă este și FNF

într-o FNF nu există o necesitate imediată de a evalua eventualii β-redecși interiori (funcția nu a fost încă aplicată).

Racket

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

```
(define (factorial n)
     (if (= n 0)
         1
         (* n (factorial (- n 1)))))
```

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.

```
(define (tail-recursion n acc)
 (if (= n 0)
    acc
    (tail-recursion (- n 1) (* n acc))))

(define (factorial n)
    (tail-recursion n 1))
```

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.

```
(define (fib n)
 (cond ((= n 0) 0)
       ((= n 1) 1)
       (else (+ (fib (- n 1))
        (fib (- n 2))))))
```

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.

- `foldr` (*right*) poate fi înțeleasă cel mai ușor prin faptul că funcția dată ca parametru se substituie lui `cons`, iar acumulatorul inițial, listei vide de la finalul listei. Prin urmare, elementele listei sunt prelucrate de la dreapta la stânga: `(foldr f acc (list` e_1 `...` e_n`))` $\rightarrow$ `(f` e_1 `(f ... (f` e_n `acc)...))`
- `foldl` (*left*) prelucrează elementele de la stânga la dreapta: `(foldl f acc (list` e_1 `...` e_n`))` $\rightarrow$ `(f` e_n `(f ... (f` e_1 `acc)...))`

```
(foldr + 0 '(1 2 3))   6
```

```
(foldl + 0 '(1 2 3))  6        0  1
(foldr cons '() '(1 2 3))   '(1 2 3)   identitate!
(foldl cons '() '(1 2 3))   '(3 2 1)   inversare
(foldl (lambda (x y acc)          21   2 liste
           (+ x y acc))
         0 '(1 2 3) '(4 5 6))
```

CHEATSHEET PP 2022

map

- Pentru o singură listă, aplică funcția, pe rând asupra fiecărui element: (map f (list e_1 … e_n)) → (list (f e_1) … (f e_n))
- Pentru mai multe liste de aceeași lungime, funcția este aplicată la un moment dat asupra tuturor elementelor de pe aceeași poziție: (map f (list e_{11} … e_{1n}) … (list e_{m1} … e_{mn})) → (list (f e_{11} … e_{m1}) … (f e_{1n} … e_{mn}))

```
(map (lambda (x) (* x 10)) '(1 2 3))    '(10 20 30)
(map * '(1 2 3) '(10 20 30))             '(10 40 90)
(map list '(1 2 3))                    '((1) (2) (3))
(map list '(1 2) '(3 4))               '((1 3) (2 4))

(define (mult-by q)                        ; Curried
  (lambda (x)
    (* x q)))
(map (mult-by 5) '(1 2 3))                '(5 10 15)
```

apply

(apply funcție listă_arg)
(apply funcție arg_1 … arg_n listă_arg)

Aplică o funcție asupra parametrilor dați de elementele unei liste. Opțional, primii parametri ai funcției îi pot fi furnizați individual lui apply, înaintea listei cu restul parametrilor. (apply f x_1 … x_m (list e_1 … e_n)) → (f x_1 … x_m e_1 … e_n)

```
(apply + '(1 2 3))                6           suma
(apply + 1 '(2 3))                6           la fel
(apply list '(1 2 3))             '(1 2 3)
(apply list '(1 2 3) '(5 6 7))   '((1 2 3) 5 6 7)
```

filter

```
(filter pred lst) → list?
  pred : procedure?
  lst : list?
```

```
> (filter positive? '(1 -2 3 4 -5))
'(1 3 4)
```

Alte funcții useful RACKET

```
> (take '(1 2 3 4 5) 2)
'(1 2)
> (take 'non-list 0)
'()
```

FLUXURI

```
(stream-map sqr naturals)          fluxul 0, 1, 4..

(stream-filter even? naturals)     fluxul nr pare
```

Exemple fluxuri

```
;; fluxul puterilor lui 2
(define powers-of-2-a
  (stream-cons
   1
   (stream-zip-with +
                    powers-of-2-a
                    powers-of-2-a)))

(define powers-of-2-b
  (stream-cons
   1
   (stream-map (lambda (x) (* x 2))
               powers-of-2-b)))
```

```
;; fluxul Fibonacci
(define fibonacci
  (stream-cons
   0
   (stream-cons
    1
    (stream-zip-with +
                     fibonacci
                     (stream-rest fibonacci)))))
```

```
(define rev-factorials
  (stream-cons
   1
   (stream-zip-with /
                    rev-factorials
                    (stream-rest naturals))))
```

Haskell

Operatorul '$'

În anumite situații, putem omite parantezele folosind '$'.

```
> length (tail (zip [1,2,3,4] ("abc" ++ "d")))
-- este echivalent cu
> length $ tail $ zip [1,2,3,4] $ "abc" ++ "d"
3
```

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

```
> length . tail . zip [1,2,3,4] $ "abc" ++ "d"
3
```

Funcțiile sunt aplicabile asupra oricâtor parametri la un moment dat.

CHEATSHEET PP 2022

```haskell
add1 x y = x + y
add2     = \x y -> x + y
add3     = \x -> \y -> x + y

result   = add1 1 2   -- sau ((add1 1) 2)
inc      = add1 1     -- functie
```

Pattern matching - definirea comportamentului funcțiilor pornind de la structura parametrilor

```haskell
add5 0 y           = y       -- add5 1 2
add5 (x + 1) y     = 1 + add5 x y

listSum []         = 0       -- sumList [1, 2, 3]
listSum (hd : tl)  = hd + listSum tl

pairSum (x, y)     = x + y   -- sumPair (1, 2)

wackySum (x, y, z@(hd : _)) =     -- wackySum
    x + y + hd + listSum z        --  (1, 2, [3, 4, 5])
```

Definire functii

```haskell
-- if .. then .. else
factorial x =
    if x < 1 then 1 else x * factorial (x - 1)

-- guards
factorial x
    | x < 1 = 1
    | otherwise = x * factorial (x - 1)

-- case .. of
factorial x = case x < 1 of
    True -> 1
    _    -> x * factorial (x - 1)

-- pattern matching
factorial 0 = 1
factorial x = x * factorial (x - 1)
```

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

```haskell
interval      = [ 0 .. 10 ]
evenInterval  = [ 0, 2 .. 10 ]
naturals      = [ 0 .. ]
```

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

```haskell
[x | x <- [1..10], x `mod` 2 == 0]    [2,4,6,8,10]

[(x, y) | x <- [1..4], y <- [10..12]]
-- aici se va construi o lista de perechi -
    [(1,10),(1,11),(1,12),(2,10),(2,11),(2,12)
    ,(3,10),(3,11),(3,12),(4,10),(4,11),(4,12)]
```

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

- Formă: $$\frac{\texttt{premisa-1 ... premisa-m}}{\texttt{concluzie-1 ... concluzie-n}}\ (\texttt{nume})$$
- Funcție: $$\frac{\texttt{Var :: a} \qquad \texttt{Expr :: b}}{\texttt{\textbackslash Var -> Expr :: a -> b}}\ (\texttt{TLambda})$$
- Aplicație: $$\frac{\texttt{Expr1 :: a -> b} \qquad \texttt{Expr2 :: a}}{\texttt{(Expr1 Expr2) :: b}}\ (\texttt{TApp})$$
- Operatorul +: $$\frac{\texttt{Expr1 :: Int} \qquad \texttt{Expr2 :: Int}}{\texttt{Expr1 + Expr2 :: Int}}\ (\texttt{T+})$$
- Literali întregi: $$\frac{}{\texttt{0, 1, 2, ... :: Int}}\ (\texttt{TInt})$$

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]**

Input: zipWith (\x y -> 2*x + y) [1..4] [5..8]

Output: [7,10,13,16]

```haskell
zip :: [a] -> [b] -> [(a, b)]
zip naturals ["w", "o", "r", "d"]
-- [(0, "w"), (1, "o"), (2, "r"), (3, "d")]

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

evens = zipWith (+) naturals naturals
    -- [2, 4, 6, ..]

fibo = 1 : 1 : zipWith (+) fibo (tail fibo)
    -- sirul lui Fibonacci
```

- **lookup**

TIP: **Eq a => a -> [(a,b)] -> Maybe b**

CHEATSHEET PP 2022

Input: lookup 'c' [('a',0),('b',1),('c',2)]

Output: Just 2

Example 2

Input: lookup 'c' [('a',0),('b',1),('c',2),('a',3),('b',4),('c',5)]

Output: Just 2

Example 3

Input: lookup 'f' [('a',0),('b',1),('c',2)]

Output: Nothing

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**

List
nub
Eq a => [a] -> [a]
nub (meaning "essence") removes duplicates elements from a list.

Example 1

Input: nub [0,1,2,3,2,1,0]

Output: [0,1,2,3]

Example 2

Input: nub "AAAAAAAAAAAABBBBBBBBBBBBBBCCCCC"

Output: "ABC"

- **fst**

Function:	fst
Type:	(a,b) -> a
Description:	returns the first item in a tuple
Related:	snd

Example 1

Input: fst(1,2)

Output: 1

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

Input: map reverse ["abc","cda","1234"]

Output: ["cba","adc","4321"]

Input: map (3*) [1,2,3,4]

Output: [3,6,9,12]

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

```
filter odd [1, 2, 3, 4]        [1, 3]
```

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

```
foldl (+) 0 [1, 2, 3, 4]     10
foldl (-) 0 [1, 2]           -3    (0 - 1) - 2
foldr (-) 0 [1, 2]           -1    1 - (2 - 0)
```

- **head - [a] -> a**

- **take, drop, null**

take :: **Int -> [a] -> [a]**

```
take 2 [1, 2, 3, 4]          [1, 2]
take 2 "HelloWorld"          "He"

drop 2 [1, 2, 3, 4]          [3, 4]

null []                      True
null [1, 2, 3]               False
```

- **elem, notElem**

```
elem 3 [1, 2, 3, 4]              True
elem 5 [1, 2, 3, 4]              False

notElem 3 [1, 2, 3, 4]           False
notElem 5 [1, 2, 3, 4]           True
```

elem :: **Eq a => a -> [a] -> Bool**

- init

```
init :: [a] -> [a]
```

$\mathcal{O}(n)$. Return all the elements of a list except the last one. The list must be non-empty.

```
>>> init [1, 2, 3]
[1,2]
```

CHEATSHEET PP 2022

SINTAXĂ WHERE

Sintaxa Where

```
def = expr
  where
  id1 = val1
  id2 = val2
  ...
  idn = valn

Exemple:

inRange :: Double -> Double -> String
inRange x max
  | f < low                = "Too_low!"
  | f >= low && f <= high = "In_range"
  | otherwise              = "Too_high!"
  where
    f = x / max
    (low, high) = (0.5, 1.0)

-- with case
listType l = case l of
               []  -> msg "empty"
               [x] -> msg "singleton"
               _   -> msg "a_longer"
             where
               msg ltype = ltype ++ "_list"
```

Polimorfism parametric vs ad-hoc

+ Polimorfism parametric Manifestarea aceluiași comportament pentru parametri de tipuri diferite. Exemplu: `id`, `Pair`.

+ Polimorfism ad-hoc Manifestarea unor comportamente diferite pentru parametri de tipuri diferite. Exemplu: `==`.

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

- Definirea mulțimii `Show`, a tipurilor care expun `show`

```
class Show a where
    show :: a -> String
```

- Precizarea apartenenței unui tip la această mulțime (instanța *aderă* la clasă)

```
instance Show Bool where
    show True  = "True"
    show False = "False"
instance Show Char where
    show c = "'" ++ [c] ++ "'"
```

⇒ Funcția `showNewLine` polimorfică!

```
showNewLine x = show x ++ "\n"
```

Ce tip au funcțiile `show`, respectiv `showNewLine`?

```
show        :: Show a => a -> String
showNewLine :: Show a => a -> String
```

Semnificație: *Dacă tipul* `a` *este membru al clasei* `Show`, *(i.e. funcția* `show` *este definită pe valorile tipului* `a`*), atunci funcțiile au tipul* `a -> String`.

Contexte utilizabile și la instanțiere:

```
instance (Show a, Show b) => Show (a, b) where
    show (x, y) = "(" ++ (show x)
                ++ ", " ++ (show y)
                ++ ")"
```

Tipul *pereche* reprezentabil ca șir doar dacă tipurile celor doi membri respectă aceeași proprietate (dată de contextul `Show`).

Tipurile -> mulțimi de valori

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

+ Clasa – Mulțime de tipuri ce pot supraîncarca operațiile specifice clasei. Reprezintă o modalitate structurată de control asupra polimorfismului ad-hoc. Exemplu: clasa `Show`, cu operația `show`.

+ Instanță a unei clase – Tip care supraîncarcă operațiile clasei.

```
class Show a where
    show :: a -> String

class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y      =  not (x == y)
    x == y      =  not (x /= y)
```

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

```
class Eq a => Ord a where
    (<), (<=), (>=), (>) :: a -> a -> Bool
```

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

- Anumite tipuri de date (definite folosind `data`) pot beneficia de implementarea automată a anumitor funcționalități, oferite de tipurile predefinite în `Prelude`:
  - `Eq, Read, Show, Ord, Enum, Ix, Bounded.`

```
data Alarm = Soft | Loud | Deafening
    deriving (Eq, Ord, Show)
```

- variabilele de tipul `Alarm` pot fi comparate, testate la egalitate, și afișate.

CHEATSHEET PP 2022

Logica propozițională

Propoziții simple: $p, q, r, \ldots$

Negații: $\neg\alpha$

Conjuncții: $(\alpha \wedge \beta)$

Disjuncții: $(\alpha \vee \beta)$

Implicații: $(\alpha \Rightarrow \beta)$

Echivalențe: $(\alpha \Leftrightarrow \beta)$

- Negație: $(\neg\alpha)^I = \begin{cases} \textit{true} & \text{dacă } \alpha^I = \textit{false} \\ \textit{false} & \text{altfel} \end{cases}$
- Conjuncție: $(\alpha \wedge \beta)^I = \begin{cases} \textit{true} & \text{dacă } \alpha^I = \textit{true} \text{ și } \beta^I = \textit{true} \\ \textit{false} & \text{altfel} \end{cases}$
- Disjuncție: $(\alpha \vee \beta)^I = \begin{cases} \textit{false} & \text{dacă } \alpha^I = \textit{false} \text{ și } \beta^I = \textit{false} \\ \textit{true} & \text{altfel} \end{cases}$
- Implicație: $(\alpha \Rightarrow \beta)^I = \begin{cases} \textit{false} & \text{dacă } \alpha^I = \textit{true} \text{ și } \beta^I = \textit{false} \\ \textit{true} & \text{altfel} \end{cases}$
- Echivalență:
  $(\alpha \Leftrightarrow \beta)^I = \begin{cases} \textit{true} & \text{dacă } \alpha \Rightarrow \beta \ \wedge \ \beta \Rightarrow \alpha \\ \textit{false} & \text{altfel} \end{cases}$

Definiții

+ **Literal** – Atom sau negația unui atom.

Exemplu $\textit{prieten}(x, y)$, $\neg\textit{prieten}(x, y)$.

+ **Clauză** – Mulțime de literali dintr-o expresie clauzală.

Exemplu $\{\textit{prieten}(x, y), \neg\textit{doctor}(x)\}$.

+ **Forma normală conjunctivă – FNC** – Reprezentare ca mulțime de clauze, cu semnificație conjunctivă.

+ **Forma normală implicativă – FNI** – Reprezentare ca mulțime de clauze cu clauzele în forma grupată $\{\neg A_1, \ldots, \neg A_m, B_1, \ldots, B_n\}$, $\Leftrightarrow$ $(A_1 \wedge \ldots \wedge A_m) \Rightarrow (B_1 \vee \ldots \vee B_n)$

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)

+ **Derivabilitate logică** Proprietatea unei propoziții de a reprezenta consecința logică a unei mulțimi de alte propoziții, numite premise. Mulțimea de propoziții Δ derivă propoziția ϕ ($\Delta \models \phi$) dacă și numai dacă orice interpretare care satisface toate propozițiile din Δ satisface și ϕ.

- $\{p\} \models p \vee q$
- $\{p, q\} \models p \wedge q$
- $\{p\} \not\models p \wedge q$
- $\{p, p \Rightarrow q\} \models q$

Inferența - derivarea mecanică a concluziilor unui set de premise

FOPL - First Order Predicate Logic
Logica cu predicate de ordinul I

Logica propozițională:
- p: "Andrei este prieten cu Bogdan."
- q: "Bogdan este prieten cu Andrei."
- $p \Leftrightarrow q$ – pot ști doar din interpretare.
- → Opacitate în raport cu obiectele și relațiile referite.

FOPL:
- Generalizare: $\textit{prieten}(x, y)$: "x este prieten cu y."
- $\forall x.\forall y.(\textit{prieten}(x, y) \Leftrightarrow \textit{prieten}(y, x))$
- → Aplicare pe cazuri particulare.
- → Transparență în raport cu obiectele și relațiile referite.

+ **Constante** – obiecte particulare din universul discursului: c, d, $\textit{andrei}$, $\textit{bogdan}$, ...

+ **Variabile** – obiecte generice: x, y, ...

+ **Simboluri funcționale** – $\textit{succesor}$, $+$, $\textit{abs}$ ...

+ **Simboluri relaționale** (predicate) – relații n-are peste obiectele din universul discursului: $\textit{prieten} = \{(\textit{andrei}, \textit{bogdan}), (\textit{bogdan}, \textit{andrei}), \ldots\}$, $\textit{impar} = \{1, 3, \ldots\}$, ...

+ **Conectori logici** $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$

+ **Cuantificatori** $\forall$, $\exists$

+ **Termeni** (obiecte):
- Constante;
- Variabile;
- Aplicații de funcții: $f(t_1, \ldots, t_n)$, unde f este un simbol funcțional n-ar și $t_1, \ldots, t_n$ sunt termeni.

+ **Atomi** (relații): atomul $p(t_1, \ldots, t_n)$, unde p este un predicat n-ar și $t_1, \ldots, t_n$ sunt termeni.

CHEATSHEET PP 2022

+ | Propoziții (fapte) – dacă x variabilă, A atom, și α și β propoziții, atunci o propoziție are forma:

- Fals, Adevărat: $\bot$, $\top$
- Atomi: A
- Negații: $\neg\alpha$
- Conectori: $\alpha \wedge \beta$, $\alpha \Rightarrow \beta$, ...
- Cuantificări: $\forall x.\alpha$, $\exists x.\alpha$

"Sora Ioanei are un prieten deștept"

$$\underbrace{\exists X.\underbrace{prieten(\underbrace{X}_{\text{termen}}, \underbrace{sora(\overbrace{ioana}^{\text{termen}})}_{\text{termen}})}_{\text{atom/propoziție}} \wedge \underbrace{destept(X)}_{\text{atom/propoziție}}}_{\text{propoziție}}$$

"Vrabia mălai visează." $\forall x.(vrabie(x) \Rightarrow viseaza(x, malai))$

"Unele vrăbii visează mălai." $\exists x.(vrabie(x) \wedge viseaza(x, malai))$

"Nu toate vrăbiile visează mălai." $\exists x.(vrabie(x) \wedge \neg viseaza(x, malai))$

"Nicio vrabie nu visează mălai." $\forall x.(vrabie(x) \Rightarrow \neg viseaza(x, malai))$

"Numai vrăbiile visează mălai." $\forall x.(viseaza(x, malai) \Rightarrow vrabie(x))$

- Necomutativitate:
  - $\forall x.\exists y.viseaza(x, y) \rightarrow$ "Toți visează la ceva anume."
  - $\exists x.\forall y.viseaza(x, y) \rightarrow$ "Există cineva care visează la orice."

- Dualitate:
  - $\neg(\forall x.\alpha) \equiv \exists x.\neg\alpha$
  - $\neg(\exists x.\alpha) \equiv \forall x.\neg\alpha$

+ | Literal – Atom sau negația unui atom.

Exemplu: $prieten(x, y)$, $\neg prieten(x, y)$.

+ | Clauză – Mulțime de literali dintr-o expresie clauzală.

Exemplu: $\{prieten(x, y), \neg doctor(x)\}$.

+ | Forma normală conjunctivă – FNC – Reprezentare ca mulțime de clauze, cu semnificație conjunctivă.

+ | Forma normală implicativă – FNI – Reprezentare ca mulțime de clauze cu clauzele în forma grupată
$\{\neg A_1, \ldots, \neg A_m, B_1, \ldots, B_n\}$, $\Leftrightarrow$ $(A_1 \wedge \ldots \wedge A_m) \Rightarrow (B_1 \vee \ldots \vee B_n)$

+ | Clauză Horn – Clauză în care cel mult un literal este în formă pozitivă:
$\{\neg A_1, \ldots, \neg A_n, A\}$,
corespunzătoare implicației
$A_1 \wedge \ldots \wedge A_n \Rightarrow A$.

Exemplu: Transformarea propoziției $\forall x.vrabie(x) \vee ciocarlie(x) \Rightarrow pasare(x)$ în formă normală, utilizând clauze Horn:
FNC: $\{\neg vrabie(x), pasare(x)\}, \{\neg ciocarlie(x), pasare(x)\}$

Conversia propozițiilor în FNC

1. Eliminarea implicațiilor ($\not\Rightarrow$)
2. Împingerea negațiilor până în fața atomilor ($\overrightarrow{\neg}$)
3. Redenumirea variabilelor cuantificate pentru obținerea unicității de nume (R):

$$\forall x.p(x) \wedge \forall x.q(x) \vee \exists x.r(x) \rightarrow \forall x.p(x) \wedge \forall y.q(y) \vee \exists z.r(z)$$

4. Deplasarea cuantificatorilor la începutul expresiei, conservându-le ordinea (forma normală *prenex*) (P):

$$\forall x.p(x) \wedge \forall y.q(y) \vee \exists z.r(z) \rightarrow \forall x.\forall y.\exists z.(p(x) \wedge q(y) \vee r(z))$$

5. Eliminarea cuantificatorilor existențiali (skolemizare) (S):
   - Dacă nu este precedat de cuantificatori universali: înlocuirea aparițiilor variabilei cuantificate printr-o constantă (bine aleasă):

   $$\exists x.p(x) \rightarrow p(c_x)$$

   - Dacă este precedat de cuantificatori universali: înlocuirea aparițiilor variabilei cuantificate prin aplicația unei funcții unice asupra variabilelor anterior cuantificate universal:

   $$\forall x.\forall y.\exists z.((p(x) \wedge q(y)) \vee r(z)) \rightarrow \forall x.\forall y.((p(x) \wedge q(y)) \vee r(f_z(x, y)))$$

6. Eliminarea cuantificatorilor universali, considerați, acum, impliciți ($\not\forall$):

$$\forall x.\forall y.(p(x) \wedge q(y) \vee r(f_z(x, y))) \rightarrow p(x) \wedge q(y) \vee r(f_z(x, y))$$

7. Distribuirea lui $\vee$ față de $\wedge$ ($\vee/\wedge$):

$$\alpha \vee (\beta \wedge \gamma) \rightarrow (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$$

8. Transformarea expresiilor în clauze (C).

Exemplu

Exemplu: "Cine rezolvă toate laboratoarele este apreciat de cineva."

$\forall x.(\forall y.(lab(y) \Rightarrow rezolva(x, y)) \Rightarrow \exists y.apreciaza(y, x))$

$\not\Rightarrow$	$\forall x.(\neg\forall y.(\neg lab(y) \vee rezolva(x, y)) \vee \exists y.apreciaza(y, x))$
$\overrightarrow{\neg}$	$\forall x.(\exists y.\neg(\neg lab(y) \vee rezolva(x, y)) \vee \exists y.apreciaza(y, x))$
$\overrightarrow{\neg}$	$\forall x.(\exists y.(lab(y) \wedge \neg rezolva(x, y)) \vee \exists y.apreciaza(y, x))$
R	$\forall x.(\exists y.(lab(y) \wedge \neg rezolva(x, y)) \vee \exists z.apreciaza(z, x))$
P	$\forall x.\exists y.\exists z.((lab(y) \wedge \neg rezolva(x, y)) \vee apreciaza(z, x))$
S	$\forall x.((lab(f_y(x)) \wedge \neg rezolva(x, f_y(x))) \vee apreciaza(f_z(x), x))$
$\not\forall$	$(lab(f_y(x)) \wedge \neg rezolva(x, f_y(x))) \vee apreciaza(f_z(x), x)$
$\vee/\wedge$	$(lab(f_y(x)) \vee apr(f_z(x), x)) \wedge (\neg rez(x, f_y(x)) \vee apr(f_z(x), x))$
C	$\{lab(f_y(x)), apr(f_z(x), x)\}, \{\neg rez(x, f_y(x)), apr(f_z(x), x)\}$

REZOLUȚIE

Principiu de bază –→ pasul de rezoluție

- Forma generală a pasului de rezoluție:

$$\frac{\begin{array}{c}\{p_1, \ldots, r, \ldots, p_m\} \\ \{q_1, \ldots, \neg r, \ldots, q_n\}\end{array}}{\{p_1, \ldots, p_m, q_1, \ldots, q_n\}}$$

PROLOG

Operatorul Cut = !

Prima întâlnire -> satisfacere

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

CHEATSHEET PP 2022

```
girl(mary).
girl(ann).

boy(john).
boy(bill).

pair(X, Y) :- girl(X), boy(Y).
pair(bella, harry).

pair2(X, Y) :- girl(X), !, boy(Y).
pair2(bella, harry).
```

```
?- pair(X, Y).
X = mary,
Y = john ;
X = mary,
Y = bill ;
X = ann,
Y = john ;
X = ann,
Y = bill ;
X = bella,
Y = harry.
```

```
?- pair2(X, Y).
X = mary,
Y = john ;
X = mary,
Y = bill.
```

Operatori

- Aritmetici: `+` `-` `*` `/`
- Relaționali: `=\=` `<` `>` `=<` `>=` `=:=` `is`
- Logici: `,` (și) `;` (sau) `\+` (negație)

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).

```
?- 1 + 2 =:= 2 + 1.
true.

?- 1 + 2 = 2 + 1.
false.

?- X = 2 + 1.
X = 2+1.

?- X is 2 + 1.
X = 3.

?- X =:= 2 + 1.
ERROR: =:=/2: Arguments are not sufficiently instantiated
```

Negație

Operatorul unar `\+` folosit pentru un operand reprezintă faptul că nu se poate demonstra că operandul este adevărat. Dacă operandul conține variabile, `\+` denotă că nu există nicio legare pentru variabile astfel încât operandul să fie adevărat. Operatorul `\+` trebuie în mod necesar să fie urmat de spațiu (sau paranteză deschisă)

Aflarea tuturor soluțiilor pentru satisfacerea unui scop

findall/3

```
findall(+Template, +Goal, -Bag)
```

Predicatul findall creează o listă de instanțieri ale lui Template care satisfac Goal și apoi unifică rezultatul cu Bag

```
higherThan(Numbers, Element, Result):-
    findall(X, (member(X, Numbers), X > Element), Result).
?- higherThan([1, 2, 7, 9, 11], 5, X).
X = [7, 9, 11]

?- findall([X, SqX], (member(X, [1,2,7,9,15]), X > 5, SqX is X **
     2), Result). % în argumentul Template putem construi
    structuri mai complexe
Result = [[7, 49], [9, 81], [15, 225]].
```

Aflarea tuturor soluțiilor pentru satisfacerea unui scop

forall/2

```
forall(+Cond, +Action)
```

Predicatul forall verifică dacă pentru orice legare din Cond, care reprezintă un domeniu ce conține legări de variabile, se pot îndeplini condițiile din Action.

```
?- forall(member(X,[2, 4, 6]), X mod 2 =:= 0).
true.

?- forall(member(X,[2, 4, 3, 6]), X mod 2 =:= 0).
false.

?- forall(member(X, [6, 12, 18]), (X mod 2 =:= 0, X mod 3 =:= 0))
    .
true.
```

Aflarea tuturor soluțiilor pentru satisfacerea unui scop

bagof/3

```
bagof(+Template, +Goal, -Bag)
```

Predicatul bagof este asemănător cu predicatul findall, cu excepția faptului că predicatul bagof construiește câte o listă separată pentru fiecare instanțiere diferită a variabilelor din Goal (fie că ele sunt numite sau sunt înlocuite cu underscore.

```
are(andrei,laptop,1). are(andrei,pix,5). are(andrei,ghiozdan,2).
are(radu,papagal,1). are(radu,ghiozdan,1). are(radu,laptop,2).
are(ana, telefon, 3). are(ana, masina, 1).

?- findall(X, are(_, X, _), Bag).
Bag = [laptop, pix, ghiozdan, papagal, ghiozdan, laptop, telefon,
     masina]. % laptop şi ghiozdan apar de două ori pentru că
    sunt două posibile legări pentru persoană şi pentru cantitate

?- bagof(X, are(andrei, X, _), Bag).
Bag = [laptop] ;
Bag = [ghiozdan] ;
Bag = [pix].
% bagof creează câte o soluţie pentru fiecare posibilă legare
    pentru cantitate.  Putem aici folosi operatorul existenţial ^
?- bagof(X, C^are(andrei, X, C), Bag).
Bag = [laptop, pix, ghiozdan]. % am cerut lui bagof să pună toate
    soluţiile indiferent de legarea lui C în acelaşi grup

?- bagof(X, C^are(P, X, C), Bag).
P = ana, Bag = [telefon, masina] ;
P = andrei, Bag = [laptop, pix, ghiozdan] ;
P = radu, Bag = [papagal, ghiozdan, laptop].
```

Dacă aplicăm operatorul existențial pe toate variabilele libere din scop, rezultatul este identic cu cel al lui `findall`.

```
?- bagof(X, X^P^C^are(P, X, C), Bag).
Bag = [laptop, pix, ghiozdan, papagal, ghiozdan, laptop, telefon,
     masina].
```

Aflarea tuturor soluțiilor pentru satisfacerea unui scop

setof/3

```
setof(+Template, +Goal, -Bag)
```

Predicatul setof este asemănător cu bagof, dar sortează rezultatul (și elimină duplicatele) folosind sort/2.

```
?- setof(X, C^are(P, X, C), Bag).
P = ana, Bag = [masina, telefon] ; %se observă sortarea
P = andrei, Bag = [ghiozdan, laptop, pix] ;
P = radu, Bag = [ghiozdan, laptop, papagal].

?- setof(X, P^C^are(P, X, C), Bag).% setof elimină duplicatele
Bag = [ghiozdan, laptop, masina, papagal, pix, telefon].
```