

# Paradigme de Programare

Conf. dr. ing. Andrei Olaru

[andrei.olaru@upb.ro](mailto:andrei.olaru@upb.ro) | [cs@andreiolaru.ro](mailto:cs@andreiolaru.ro)  
Departamentul de Calculatoare

2021

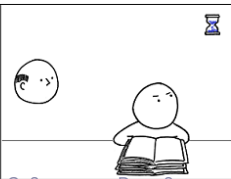
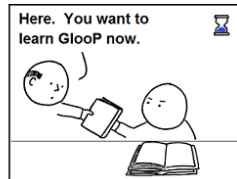
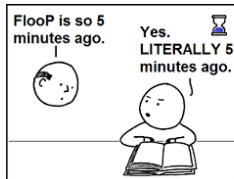
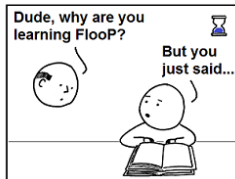
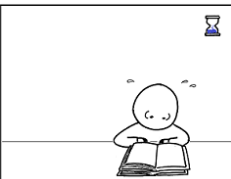
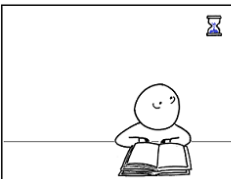
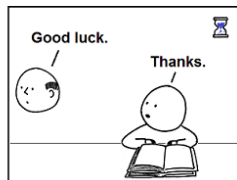
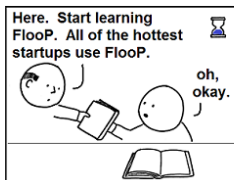
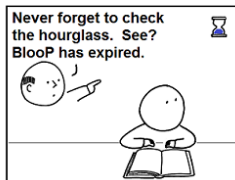
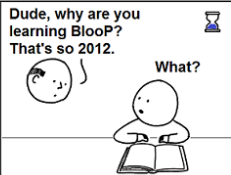
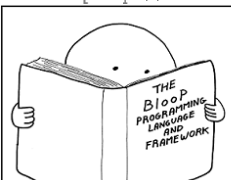
# Cursul 1: Introducere

---

- 1 Exemplu
- 2 Ce studiem la PP?
- 3 De ce studiem această materie?
- 4 Organizare
- 5 Introducere în Racket
- 6 Paradigma de programare
- 7 Istoric: Paradigme și limbaje de programare

# BlooP and FlooP and GloopP

[<http://abstrusegoose.com/503>]



Exemplar  
[(CC) BY-NC abstrusegoose.com]

Organizare  
Introducere

Paradigma

Istoric

# Exemplu



Exemplu

Să se determine dacă un element  $e$  se regăsește într-o listă  $L$  ( $e \in L$ ).

Să se sorteze o listă  $L$ .

## Racket:

```
1 (define memList (lambda (e L)
2   (if (null? L)
3       #f
4       (if (equal? (first L) e)
5           #t
6           (memList e (rest L))
7         ))
8   ))
9
10 (define ins (lambda (x L)
11   (cond ((null? L) (list x))
12         ((< x (first L)) (cons x L))
13         (else (cons (first L) (ins x (rest L))))))
```

## Haskell

```
1 memList x [] = False
2 memList x (e:t) = x == e || memList x t
3
4 ins x [] = [x]
5 ins x l@(h:t) = if x < h then x:l else h : ins x t
```

## Prolog:

```
1 memberA(E, [E|_]) :- !.
2 memberA(E, [_|L]) :- memberA(E, L).
3
4 % elementul, lista, rezultatul
5 ins(E, [], [E]).
6 ins(E, [H | T], [E, H | T]) :- E < H, !.
7 ins(E, [H | T], [H | TE]) :- ins(E, T, TE).
```



# Ce studiem la PP?

- Paradigma funcțională și paradigma logică, în contrast cu paradigma imperativă.
- Racket: introducere în programare funcțională
- Calculul  $\lambda$  ca bază teoretică a paradigmei funcționale
- Racket: întârzierea evaluării și fluxuri
- Haskell: programare funcțională cu o sintaxă avansată
- Haskell: evaluare leneșă și fluxuri
- Haskell: tipuri, sinteză de tip, și clase
- Prolog: programare logică
- LPOI ca bază pentru programarea logică
- Prolog: strategii pentru controlul execuției
- Algorimi Markov: calcul bazat pe reguli de transformare

# De ce studiem această materie?



The first math class.

[(C) Zach Weinersmith,  
Saturday Morning Breakfast  
Cereal]

[[https://www.smbc-comics.com/  
comic/a-new-method](https://www.smbc-comics.com/comic/a-new-method)]

The first math class.

*I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.*

---

The law of instrument – Abraham Maslow

- până acum ați studiat paradigma imperativă (legată și cu paradigma orientată-obiect)

→ **un anumit mod** de a privi procesul de rezolvare al unei probleme și de a căuta soluții la probleme de programare.

- paradigmele declarative studiate oferă o gamă diferită (complementară!) de **unelte** → **alte moduri** de a rezolva anumite probleme.

⇒ o pregătire ce permite accesul la poziții de calificare mai înaltă (arhitect, designer, etc.)

## Sunt aceste paradigme relevante?

---

- **evaluarea leneșă** → prezentă în Python (de la v3), .NET (de la v4)
- **funcții anonime** → prezente în C++ (de la v11), C#/.NET (de la v3.0/v3.5), Dart, Go, Java (de la JDK8), JS/ES, Perl (de la v5), PHP (de la v5.0.1), Python, Ruby, Swift.
- **Prolog și programarea logică** sunt folosite în software-ul modern de A.I., e.g. Watson; automated theorem proving.
- În **industrie** sunt utilizate limbaje puternic funcționale precum Erlang, Scala, F#, Clojure.
- Limbaje **multi-paradigmă** → adaptarea paradigmei utilizate la necesități.

- Developer Survey 2019

[<https://insights.stackoverflow.com/survey/2019/#top-paying-technologies>]

[<https://insights.stackoverflow.com/survey/2019/#salary>]

- Developer Survey 2018

[<https://insights.stackoverflow.com/survey/2018/>

[#technology-what-languages-are-associated-with-the-highest-salaries-worldwide](https://insights.stackoverflow.com/survey/2018/#technology-what-languages-are-associated-with-the-highest-salaries-worldwide)]

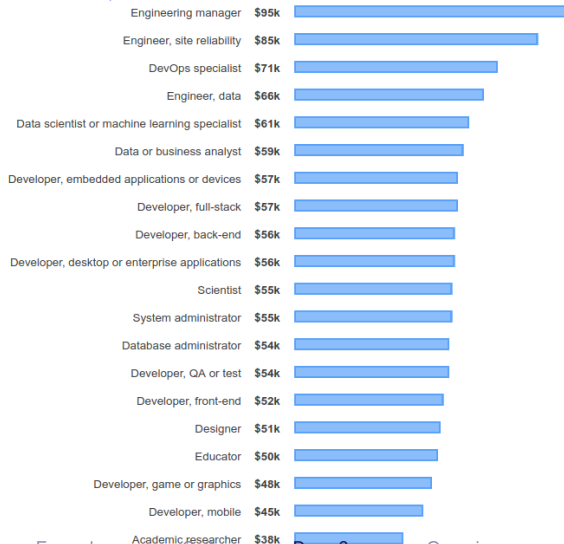
- Developer Survey 2017

[<https://insights.stackoverflow.com/survey/2017/#top-paying-technologies>]



# De ce?

## Cine câștigă cel mai bine? (1)



Exemplu

Ce?

De ce?

Organizare  
Introducere

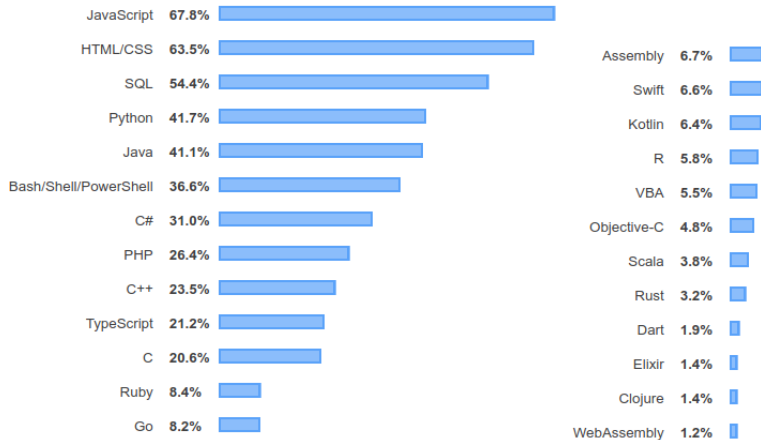
Racket

Paradigmă

Istoric

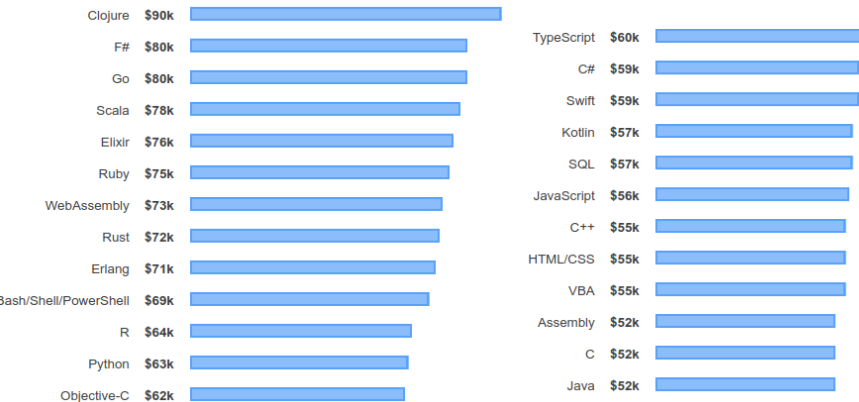
# De ce?

## Cine câștigă cel mai bine?



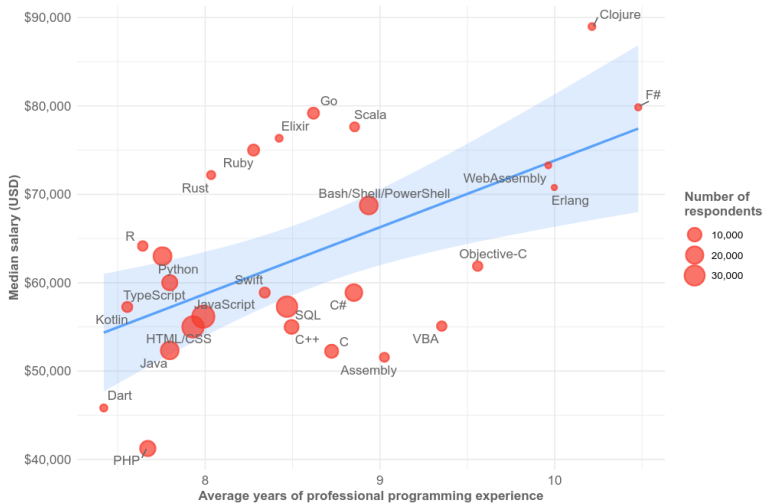
# De ce?

## Cine câștigă cel mai bine?



# De ce?

## Cine câștigă cel mai bine?



Exemplu

Ce?

De ce?

Organizare  
Introducere

Racket

Paradigmă

Istoric

# Organizare

`https://ocw.cs.pub.ro/courses/pp`

Regulament: `https://ocw.cs.pub.ro/courses/pp/21/regulament`

Forumuri: Moodle → 03-ACS-L-A2-S2-PP-CA-CC-CD

`https://curs.upb.ro/course/view.php?id=11557`

Elementele cursului sunt comune la seriile CA, CC și CD.

- Laborator: 1p ← activitate 0.7p + test grilă din laborator 0.3p
- Teme: 4p ( $3 \times 1.33p$ ) ← cu bonusuri, dar în limita a maxim 6p pe parcurs
- Teste la curs: 0.5p ← punctare pe parcurs, la curs
- Test din materia de laborator: 0.5p ← test grilă, cunoaștere de limbajelor
- Examen: 4p ← limbaje + teorie

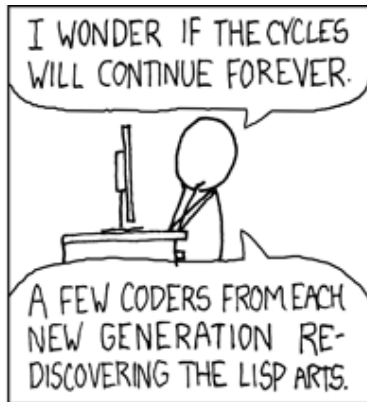
|             |   |    |    |        |
|-------------|---|----|----|--------|
| L           | T | tc | tg | Ex     |
| min parcurs |   |    |    | min ex |

# Introducere în Racket



# Lisp cycles

[<http://xkcd.com/297/>]



[(CC) BY-NC Randall Munroe, xkcd.com]

Exemplu

Ce?

De ce?

Organizare

Racket

Paradigmă

Istoric

1 : 24

Introducere

- funcțional
- dialect de Lisp
- totul este văzut ca o funcție
- constante – expresii neevaluate
- perechi / liste pentru structurarea datelor
- apeluri de funcții – liste de apelare, evaluate
- evaluare aplicativă, funcții stricte, cu anumite excepții

# Paradigma de programare

- aceasta este o diferență între limbaje, dar este influențată și de natura paradigmei
- diferă sintaxa ←
  - mecanisme specifice unei paradigme aduc elemente noi de sintaxă
  - e.g. funcțiile anonime
- diferă modul de construcție ←
  - ce poate reprezenta o expresie, ce operatori putem aplica între expresii.al expresiilor
- diferă structura programului ←
  - ce anume reprezintă programul
  - cum se desfășoară execuția programului

- valorile de prim rang
- modul de construcție a programului
- modul de tipare al valorilor
- ordinea de evaluare (generare a valorilor)
- modul de legare al variabilelor (managementul valorilor)
- controlul execuției

• **Paradigma de programare** este dată de stilul fundamental de construcție al structurii și elementelor unui program.

- 1 Diverse perspective conceptuale asupra noțiunii de calculabilitate efectivă → **modele de calculabilitate**.
- 2 Influența perspectivei alese asupra procesului de modelare și rezolvare a problemelor → **paradigme de programare**.
- 3 **Limbaje de programare** aferente paradigmelor, cu accent pe aspectul comparativ.

|                              |                         |                    |
|------------------------------|-------------------------|--------------------|
| C, Pascal → procedural       | → paradigma imperativă  | → Mașina Turing    |
| J, C++, Py → orientat-obiect |                         |                    |
| Racket, Haskell              | → paradigma funcțională | → Mașina $\lambda$ |
| Prolog                       | → paradigma logică      | → FOL + Resolution |
| CLIPS                        | → paradigma asociativă  | → Mașina Markov    |

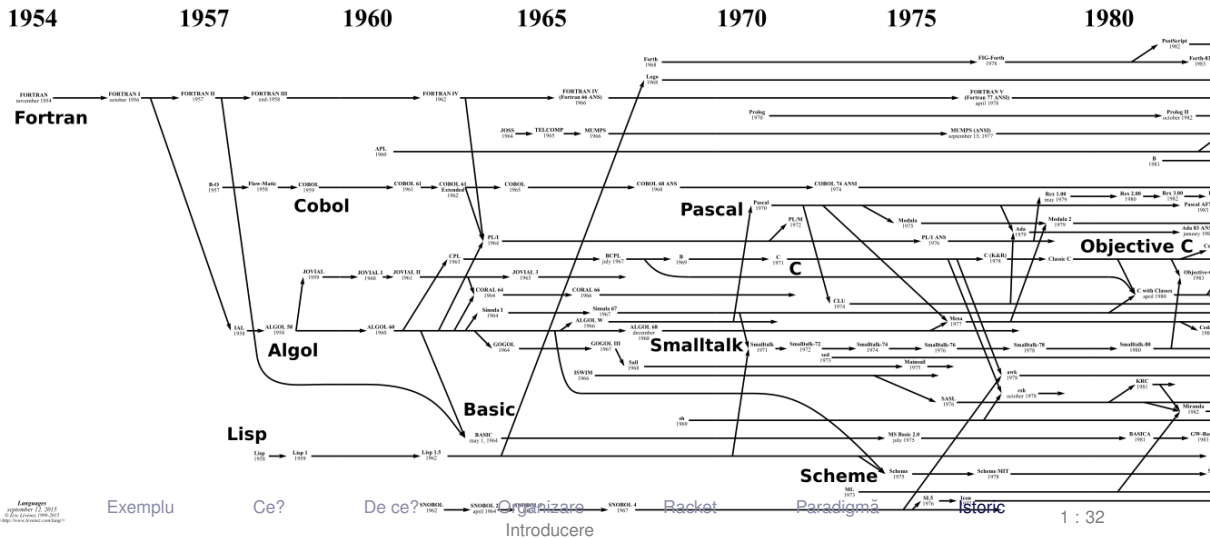
echivalente !

**T** | **Teza Church-Turing**: efectiv calculabil = Turing calculabil

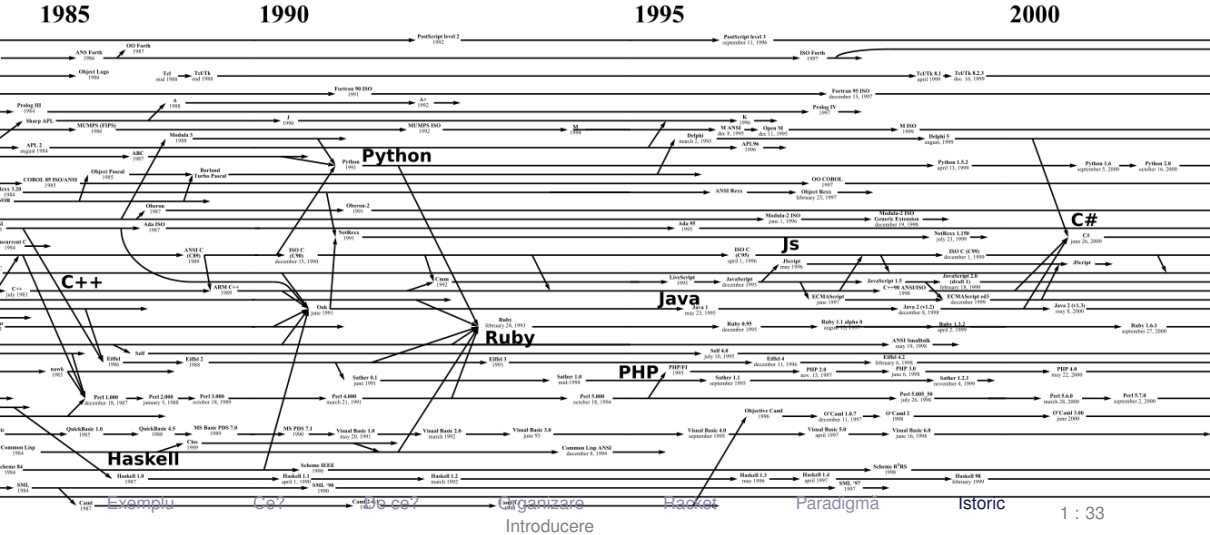
# Istoric: Paradigme și limbaje de programare



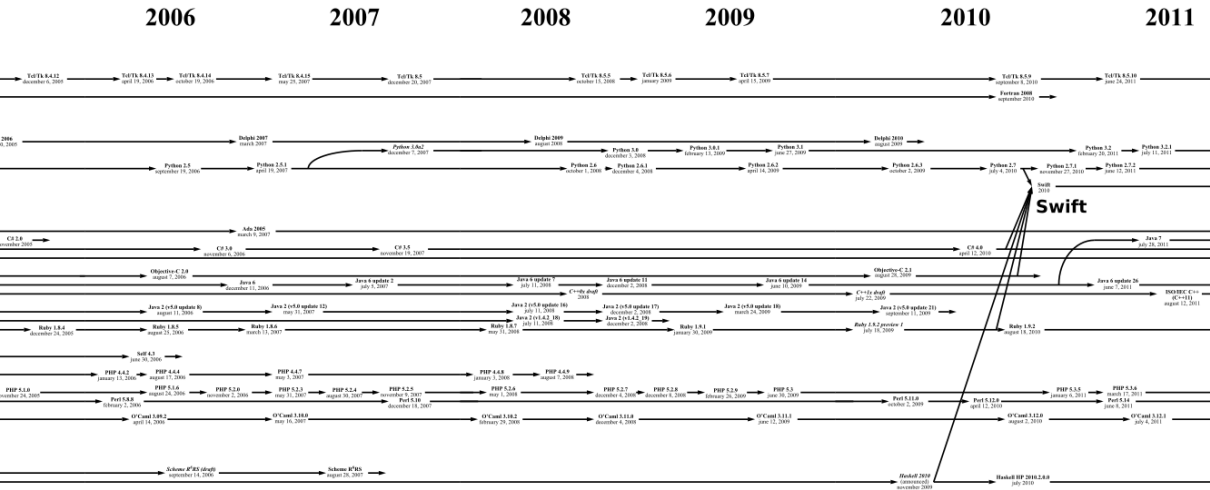
# Istorie 1950-1975



# Istorie 1975-1995







Exemplu

Ce?

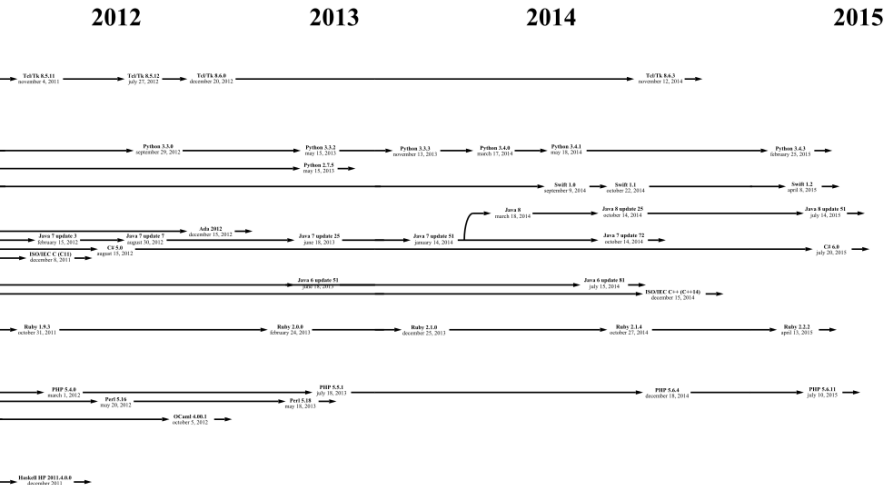
De ce?

Organizare  
Introducere

Racket

Paradigmă

Istoric



Exemplu

Ce?

De ce?

Organizare  
Introducere

Racket

Paradigmă

Istoric

- imagine navigabilă (slides precedente): `[http://www.levenez.com/lang/]`

- Wikipedia:

`[http://en.wikipedia.org/wiki/Generational_list_of_programming_languages]`

`[https://en.wikipedia.org/wiki/Timeline_of_programming_languages]`

(  
[\[http://xkcd.com/859/\]](http://xkcd.com/859/)

---

(AN UNMATCHED LEFT PARENTHESIS  
CREATES AN UNRESOLVED TENSION  
THAT WILL STAY WITH YOU ALL DAY.

[(CC) BY-NC xkcd.com]



- 8 Introducere
- 9 Legarea variabilelor
- 10 Evaluare
- 11 Construcția programelor prin recursivitate
- 12 Discuție despre tipare



# Introducere



- Gestionarea valorilor
  - modul de tipare al valorilor
  - modul de legare al variabilelor
  - valorile de prim rang
- Gestionarea execuției
  - ordinea de evaluare (generare a valorilor)
  - controlul evaluării
  - modul de construcție al programelor

# Legarea variabilelor

### ⋮ Proprietăți

- identificador
- valoarea legată (la un anumit moment)
- domeniul de vizibilitate (*scope*) + durata de viață
- tip

### ⋮ Stări

- declarată: cunoaștem **identificadorul**
- definită: cunoaștem și **valoarea** → variabila a fost *legată*

· în Racket, variabilele (numele) sunt legate *static* prin construcțiile `lambda`, `let`, `let*`, `letrec` și `define`, și sunt vizibile în domeniul construcției unde au fost definite (excepție face `define`).

+ **Legarea variabilelor** – modalitatea de **asociere** a apariției unei variabile cu definiția acesteia (deci cu valoarea).

+ **Domeniul de vizibilitate** – *scope* – mulțimea punctelor din program unde o **definiție** (legare) este vizibilă.

+ **Legare statică** – Valoarea pentru un nume este legată o singură dată, la **declarare**, în contextul în care aceasta a fost definită. Valoarea depinde doar de contextul **static** al variabilei.

- Domeniu de vizibilitate al legării poate fi desprins la **compilare**.

+ **Legare dinamică** – Valorile variabilelor depind de **momentul** în care o expresie este **evaluată**. Valoarea poate fi (re-)legată la variabilă **ulterior** declarării variabilei.

- Domeniu de vizibilitate al unei legări – determinat la **execuție**.



- Variabile definite în construcții interioare → **legate static, local**:
  - `lambda`
  - `let`
  - `let*`
  - `letrec`
- Variabile *top-level* → **legate static, global**:
  - `define`



- Leagă **static** parametrii formali ai unei funcții
- Sintaxă:

```
1 (lambda (p1 ... pk ... pn) expr)
```

- Domeniul de vizibilitate al parametrului  $p_k$ : mulțimea punctelor din `expr` (care este **corpul funcției**), puncte în care apariția lui  $p_k$  este **liberă**.





- Aplicație:

```
1 ((lambda (p1 ... pn) expr)
2  a1 ... an)
```

- 1 Evaluare aplicativă: se evaluează **argumentele**  $a_k$ , în ordine **aleatoare** (nu se garantează o anumită ordine).
- 2 Se evaluează **corpul** funcției,  $expr$ , ținând cont de legările  $p_k \leftarrow \text{valoare}(a_k)$ .
- 3 Valoarea aplicației este **valoarea** lui  $expr$ , evaluată mai sus.



- Leagă **static** variabile locale
- Sintaxă:

```
1  (let ( (v1 e1) ... (vk ek) ... (vn en) )
2      expr)
```

- Domeniul de vizibilitate a variabilei  $v_k$  (cu valoarea  $e_k$ ): mulțimea punctelor din `expr` (**corp let**), în care aparițiile lui  $v_k$  sunt **libere**.

## Exemplu

```
1  (let ((x 1) (y 2)) (+ x 2))
```

- **Atenție!** Construcția `(let ((v1 e1) ... (vn en)) expr)` – **echivalentă** cu `((lambda (v1 ...vn) expr) e1 ...en)`



- Leagă **static** variabile locale

- Sintaxă:

```
1 (let* ((v1 e1) ... (vk ek) ... (vn en))
2   expr)
```

- Scope pentru variabila  $v_k$  = mulțimea punctelor din

- restul **legărilor** (legări ulterioare) și
- **corp** – `expr`

în care aparițiile lui  $v_k$  sunt **libere**.



### Exemplu

```
1 (let* ((x 1) (y x))
2   (+ x 2))
```



```
1  (let* ((v1 e1) ... (vn en))
2      expr)
```

echivalent cu

```
1  (let ((v1 e1))
2      ...
3      (let ((vn en))
4          expr) ... )
```

- Evaluarea expresiilor  $e_i$  se face **în ordine!**



- Leagă `static` variabile locale

- Sintaxă:

```
1 (letrec ((v1 e1) ... (vk ek) ... (vn en))
2      expr)
```

- Domeniul de vizibilitate a variabilei  $v_k$  = mulțimea punctelor din **întreaga** construcție, în care aparițiile lui  $v_k$  sunt **libere**.



### Exemplu

```
1 (letrec ((factorial
2           (lambda (n)
3             (if (zero? n) 1
4                 (* n (factorial (- n 1)))))))
5   (factorial 5))
```



- Leagă **static** variabile **top-level**.
- Avantaje:
  - definirea variabilelor *top-level* în **orice** ordine
  - definirea de funcții **mutual** recursive

### Ex Definiții echivalente:

```
1 (define f1
2   (lambda (x)
3     (add1 x)
4   ))
5
6 (define (f2 x)
7   (add1 x)
8 ))
```

# Evaluare





- Evaluare **aplicativă**: evaluarea parametrilor **înaintea** aplicării funcției asupra acestora (în ordine aleatoare).
- Funcții **stricte** (i.e. cu evaluare aplicativă)
  - Excepții: `if`, `cond`, `and`, `or`, `quote`.



- quote sau '
  - funcție **nestrictă**
  - întoarce parametrul **neevaluat**
- eval
  - funcție **strictă**
  - forțează **evaluarea** parametrului și întoarce valoarea acestuia

## Exemplu

```
1 (define sum '(+ 2 3))
2 sum ; '(+ 2 3)
3 (eval (list (car sum) (cadr sum) (caddr sum))) ; 5
```

# Construcția programelor prin recursivitate



- **Recursivitatea** – element fundamental al paradigmei funcționale
  - Numai prin recursivitate (sau iterare) se pot realiza prelucrări pe date de dimensiuni nedefinite.
- Dar, este eficient să folosim recursivitatea?
  - recursivitatea (pe stivă) poate **încărca stiva**.



- pe stivă:  $factorial(n) = n * factorial(n - 1)$ 
  - timp: liniar
  - spațiu: liniar (ocupat pe stivă)
  - dar, în procedural putem implementa factorialul în spațiu **constant**.
- pe coadă:  
 $factorial(n) = fH(n, 1)$   
 $fH(n, p) = fH(n - 1, p * n)$ ,  $n > 1$ ;  $p$  altfel
  - timp: liniar
  - spațiu: constant
- beneficiu *tail call optimization*

# Discuție despre tipare



În Racket avem:

- numere: 1, 2, 1.5
- simbolii (literali): 'abcd, 'andrei
- valori booleene: #t, #f
- șiruri de caractere: "șir de caractere"

- perechi: (cons 1 2)  $\rightarrow$  '(1 . 2)
- liste: (cons 1 (cons 2 '()))  $\rightarrow$  '(1 2)

- funcții: ( $\lambda$  (e f) (cons e f))  $\rightarrow$  #<procedure>

• Cum sunt gestionate tipurile valorilor (variabilelor) la **compilare** (verificare) și la **execuție**?

· Rolul tipurilor: exprimare a intenției programatorului, abstractizare, documentare, optimizare, verificare

+ **Tipare** – modul de gestionare a tipurilor.

⋮ Clasificare după **momentul** verificării:

- statică
- dinamică

⋮ Clasificare după **rigiditatea** regulilor:

- tare
- slabă



### Ex Tipare dinamică

Exemplu

Javascript:

```
var x = 5;  
if(condition) x = "here";  
print(x); → ce tip are x aici?
```

### Ex Tipare statică

Exemplu

Java:

```
int x = 5;  
if(condition)  
    x = "here"; → Eroare la compilare: x este int.  
print(x);
```

### ⋮ Tipare statică

- La compilare
- Valori și variabile
- Rulare mai rapidă
- Rigidă: sancționează orice construcție
- Debugging mai facil
- Declarații explicite sau inferențe de tip
- Pascal, C, C++, Java, Haskell

### ⋮ Tipare dinamică

- La rulare
- Doar valori
- Rulare mai lentă (necesită verificarea tipurilor)
- Flexibilă: sancționează doar când este necesar
- Debugging mai dificil
- Permite metaprogramare (v. `eval`)
- Python, Scheme/Racket, Prolog, JavaScript, PHP

- Clasificare după **libertatea** de a agrega valori de tipuri **diferite**.

### Ex Tipare tare

Exemplu

$1 + "23" \rightarrow$  **Eroare** (Haskell, Python)

### Ex Tipare slabă

Exemplu

$1 + "23" = 24$  (Visual Basic)

$1 + "23" = "123"$  (JavaScript)



- este **dinamică**

```
1 (if #t 'something (+ 1 #t)) → 'something
```

```
2 (if #f 'something (+ 1 #t)) → Eroare
```

- este **tare**

```
1 (+ "1" 2) → Eroare
```

- dar, permite **liste** cu elemente de tipuri diferite.



- Tipare: dinamică vs. statică, tare vs. slabă;
- Legare: dinamică vs statică;
- Racket: tipare dinamică, tare; domeniu al variabilelor;
- construcții care leagă nume în Racket: `lambda`, `let`, `let*`, `letrec`, `define`;
- evaluare aplicativă;
- construcția funcțiilor prin recursivitate.

- 13 Introducere
- 14 Lambda-expresii
- 15 Reducere
- 16 Evaluare
- 17 Limbajul lambda-0 și incursiune în TDA
- 18 Racket vs. lambda-0

# Introducere

- ne punem problema dacă putem realiza un calcul sau nu → pentru a demonstra trebuie să avem un model simplu al calculului (**cum realizăm calculul**, în mod formal).
- un model de calculabilitate trebuie să fie cât mai simplu, atât ca număr de **operații** disponibile cât și ca mod de **construcție a valorilor**.
- corectitudinea unui program se demonstrează mai ușor dacă limbajul de programare este mai apropiat de mașina teoretică (modelul abstract de calculabilitate).



- **Model de calculabilitate** (Alonzo Church, 1932) – introdus în cadrul cercetărilor asupra fundamentelor matematicii.

[[http://en.wikipedia.org/wiki/Lambda\\_calculus](http://en.wikipedia.org/wiki/Lambda_calculus)]

- sistem formal pentru exprimarea calculului.
- **Echivalent** cu Mașina Turing (v. Teza Church-Turing)
- Axat pe conceptul matematic de **funcție** – totul este o funcție

- Aplicații importante în
  - **programare**
  - demonstrarea formală a **corectitudinii** programelor, datorită modelului simplu de execuție
- Baza teoretică a numeroase **limbaje**:  
LISP, Scheme, Haskell, ML, F#, Clean, Clojure, Scala, Erlang etc.

# Lambda-expresii



Exemplu

- 1  $x \rightarrow$  variabila (numele)  $x$
- 2  $\lambda x.x \rightarrow$  funcția identitate
- 3  $\lambda x.\lambda y.x \rightarrow$  funcție selector
- 4  $(\lambda x.x \ y) \rightarrow$  aplicația funcției identitate asupra parametrului actual  $y$
- 5  $(\lambda x.(x \ x) \ \lambda x.x) \rightarrow ?$



Intuitiv, evaluarea aplicației  $(\lambda x.x \ y)$  presupune substituția textuală a lui  $x$ , în corp, prin  $y \rightarrow$  rezultat  $y$ .

### + $\lambda$ -expresie

- **Variabilă**: o variabilă  $x$  este o  $\lambda$ -expresie;
- **Funcție**: dacă  $x$  este o variabilă și  $E$  este o  $\lambda$ -expresie, atunci  $\lambda x.E$  este o  $\lambda$ -expresie, reprezentând funcția **anonimă**, unară, cu parametrul formal  $x$  și corpul  $E$ ;
- **Aplicație**: dacă  $F$  și  $A$  sunt  $\lambda$ -expresii, atunci  $(F A)$  este o  $\lambda$ -expresie, reprezentând aplicația expresiei  $F$  asupra parametrului actual  $A$ .

$$((\lambda x. \lambda y. x \ z) \ t)$$

$$\left( \boxed{(\lambda x. \lambda y. x \ z)} \ t \right) \quad \leftarrow \begin{array}{l} \text{parametru formal} \\ \text{parametru actual} \end{array}$$

||  
substituție

↓  
 $(\lambda y. z \ t)$

$$\boxed{(\lambda y. z \ t)} \quad \leftarrow \begin{array}{l} \text{parametru formal} \\ \text{parametru actual} \end{array}$$

||  
substituție

# Reducere

- $\beta$ -redex: o  $\lambda$ -expresie de forma:  $(\lambda x. E \ A)$ 
  - $E$  –  $\lambda$ -expresie – este corpul funcției
  - $A$  –  $\lambda$ -expresie – este parametrul actual
- $\beta$ -redexul se reduce la  $E_{[A/x]}$  –  $E$  cu toate aparițiile **libere** ale lui  $x$  din  $E$  înlocuite cu  $A$  prin substituție textuală.



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

- $E = \lambda x.F$  sau
- $E = \dots \lambda x_n.F \dots$  sau
- $E = \dots \lambda x.F \dots$  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.

- **Atenție!** În raport cu o expresie dată!

# Apariții ale variabilelor

 $\lambda$ 

## Mod de gândire

· O apariție **legată în expresie** este o apariție a parametrului formal al unei funcții definite **în** expresie, în corpul funcției; o apariție **liberă** este o apariție a parametrului formal al unei funcții definite **în exteriorul** expresiei, sau nu este parametru formal al niciunei funcții.

•  $x_{<1>}$  ← apariție liberă

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

•  $\lambda x_{<2>}. (\lambda y. x_{<1>} z)$  ←  $\lambda x_{<2>}$  leagă apariția  $x_{<1>}$

•  $(\lambda x_{<2>}. (\lambda y. x_{<1>} z) x_{<3>})$  ← apariția  $x_3$  este liberă – este în exteriorul corpului funcției cu parametrul formal  $x$  ( $\lambda x_2$ )

corp  $\lambda x_2$

•  $\lambda x_{<4>}. (\lambda x_{<2>}. (\lambda y. x_{<1>} z) x_{<3>})$  ←  $\lambda x_{<4>}$  leagă apariția  $x_{<3>}$

Introducere

$\lambda$ -Expresii

Reducere

Evaluare

$\lambda_0$  și TDA

Racket vs.  $\lambda_0$

Calcul Lambda

+ **O variabilă este legată** într-o expresie dacă **toate** aparițiile sale sunt legate în acea expresie.

+ **O variabilă este liberă** într-o expresie dacă nu este legată în acea expresie i.e. dacă **cel puțin o** apariție a sa este liberă în acea expresie.

- Atenție! În raport cu o **expresie** dată!

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

$(\lambda \underset{\langle 1 \rangle}{x} . \underbrace{\underset{\langle 2 \rangle}{x} \ \underset{\langle 3 \rangle}{x}}_F)$ .

- $\underset{\langle 1 \rangle}{x}$ ,  $\underset{\langle 2 \rangle}{x}$  **legate** în  $E$
- $\underset{\langle 3 \rangle}{x}$  **liberă** în  $E$
- $\underset{\langle 2 \rangle}{x}$  **liberă** în  $F$ !
- $x$  **liberă** în  $E$  și  $F$

**Ex**

Exemplu

## Exemplu 2

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

$$(\lambda_{\langle 1 \rangle} x_{\langle 1 \rangle} . \lambda_{\langle 1 \rangle} z_{\langle 1 \rangle} . (\underbrace{z_{\langle 2 \rangle} x_{\langle 2 \rangle}}_F) (z_{\langle 3 \rangle} y_{\langle 1 \rangle})).$$

- $x_{\langle 1 \rangle}, x_{\langle 2 \rangle}, z_{\langle 1 \rangle}, z_{\langle 2 \rangle}$  **legate** în  $E$
- $y_{\langle 1 \rangle}, z_{\langle 3 \rangle}$  **libere** în  $E$
- $z_{\langle 1 \rangle}, z_{\langle 2 \rangle}$  **legate** în  $F$
- $x_{\langle 2 \rangle}$  **liberă** în  $F$
- $x$  **legată** în  $E$ , dar **liberă** în  $F$
- $y$  **liberă** în  $E$
- $z$  **liberă** în  $E$ , dar **legată** în  $F$

Exemplu

### Variable libere (*free variables*)

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

### Variable legate (*bound variables*)

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

+ **O expresie închisă** este o expresie care **nu** conține variabile libere.

## Ex) Exemplu

- $(\lambda x.x \ \lambda x.\lambda y.x) \ \dots \rightarrow$  închisă
- $(\lambda x.x \ a) \ \dots \rightarrow$  deschisă, deoarece  $a$  este liberă
- Variabilele **libere** dintr-o  $\lambda$ -expresie pot sta pentru alte  $\lambda$ -expresii
- Înaintea evaluării, o expresie trebuie adusă la forma **închisă**.
- Procesul de înlocuire trebuie să se **termine**.

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

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

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





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

Care este problema?

- **Problemă:** în expresia  $(\lambda x.E \ A)$ :
  - dacă variabilele libere din  $A$  nu au nume comune cu variabilele legate din  $E$ :  
 $FV(A) \cap BV(E) = \emptyset$   
→ reducere întotdeauna **corectă**
  - dacă există variabilele libere din  $A$  care au nume comune cu variabilele legate din  $E$ :  $FV(A) \cap BV(E) \neq \emptyset$   
→ reducere **potențial greșită**
- **Soluție:** redenumirea variabilelor legate din  $E$ , ce coincid cu cele libere din  $A$  →  **$\alpha$ -conversie**.

### Ex Exemplu

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

+  **$\alpha$ -conversie:** Redenumirea sistematică a variabilelor **legate** dintr-o funcție:  $\lambda x.E \rightarrow_{\alpha} \lambda y.E_{[y/x]}$ . Se impun două condiții.



Exemplu

- $\lambda x.y \rightarrow_{\alpha} \lambda y.y_{[y/x]} \rightarrow \lambda y.y \rightarrow$  **Greșit!**
- $\lambda x.\lambda y.x \rightarrow_{\alpha} \lambda y.\lambda y.x_{[y/x]} \rightarrow \lambda y.\lambda y.y \rightarrow$  **Greșit!**

### : Condiții

- $y$  **nu** este o variabilă liberă, existentă deja în  $E$
- orice apariție liberă în  $E$  **rămâne** liberă în  $E_{[y/x]}$

## Exemple



Exemplu

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

+ **Pas de reducere:** O secvență formată dintr-o  $\alpha$ -conversie și o  $\beta$ -reducere, astfel încât a doua se produce **fără coliziuni**:  
 $E_1 \rightarrow E_2 \equiv E_1 \rightarrow_{\alpha} E_3 \rightarrow_{\beta} E_2$ .

+ **Secvență de reducere:** Succesiune de zero sau mai mulți pași de reducere:  
 $E_1 \rightarrow^* E_2$ .  
Reprezintă un element din închiderea reflexiv-tranzitivă a relației  $\rightarrow$ .

### : Reducere

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



Exemplu

$$\begin{aligned} & ((\lambda x. \lambda y. (y \ x) \ y) \ \lambda x. x) \rightarrow (\lambda z. (z \ y) \ \lambda x. x) \rightarrow (\lambda x. x \ y) \rightarrow y \\ & \implies \\ & ((\lambda x. \lambda y. (y \ x) \ y) \ \lambda x. x) \rightarrow^* y \end{aligned}$$

# Evaluare

· Dacă am vrea să construim o mașină de calcul care să aibă ca program o  $\lambda$ -expresie și să aibă ca operație de bază pasul de reducere, ne punem câteva întrebări:

- 1 Când se **termină** calculul? Se termină **întotdeauna**?
- 2 Dacă mai multe secvențe de reducere se termină, obținem întotdeauna **același** rezultat?
- 3 Comportamentul **depinde** de secvența de reducere?
- 4 Dacă rezultatul este unic, **cum** îl obținem?





Exemplu

$\Omega = (\lambda x.(x\ x)\ \lambda x.(x\ x)) \rightarrow (\lambda x.(x\ x)\ \lambda x.(x\ x)) \rightarrow^* \dots$

$\Omega$  **nu** admite nicio secvență de reducere care se termină.

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

· expresia  $\Omega$  **nu** este reductibilă.

Dar!

$$E = (\lambda x.y \ \Omega)$$

$$\rightarrow y \quad \text{sau}$$

$$\rightarrow E \rightarrow y \quad \text{sau}$$

$$\rightarrow E \rightarrow E \rightarrow y \quad \text{sau} \dots$$

$$\begin{array}{l} \vdots \\ \xrightarrow{n} y, n \geq 0 \\ \vdots \\ \xrightarrow{\infty} \dots \end{array}$$

- $E$  are o secvență de reducere care **nu** se termină;
- dar  $E$  are **forma normală**  $y \Rightarrow E$  este reductibilă;
- lungimea secvențelor de reducere ale  $E$  este **nemărginită**.

Exemplu

# Forme normale

Cum știm că s-a terminat calculul?

- Calculul **se termină** atunci când expresia nu mai poate fi redusă  $\rightarrow$  expresia nu mai conține  $\beta$ -redecși.

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

Este necesar să mergem până la Forma Normală?

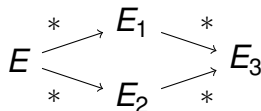
+ **Forma normală funcțională – FNF** este o formă  $\lambda x.F$ , în care  $F$  poate **conține**  $\beta$ -redecși.

Ex Exemplu

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

- FN a unei expresii închise este în mod necesar FNF.
- într-o FNF nu există o necesitate imediată de a **evalua** eventualii  $\beta$ -redecși interiori (funcția nu a fost încă aplicată).

**T | Teorema Church-Rosser / diamantului** Dacă  $E \rightarrow^* E_1$  și  $E \rightarrow^* E_2$ , atunci **există**  $E_3$  astfel încât  $E_1 \rightarrow^* E_3$  și  $E_2 \rightarrow^* E_3$ .



**C | Corolar** Dacă o expresie este reductibilă, forma ei normală este **unică**. Ea corespunde **valorii** expresiei.



Exemplu

$(\lambda x. \lambda y. (x \ y) \ (\lambda x. x \ y))$

- $\rightarrow \lambda z. ((\lambda x. x \ y) \ z) \rightarrow \lambda z. (y \ z) \rightarrow_{\alpha} \lambda a. (y \ a)$
- $\rightarrow (\lambda x. \lambda y. (x \ y) \ y) \rightarrow \lambda w. (y \ w) \rightarrow_{\alpha} \lambda a. (y \ a)$

- Forma normală corespunde unei **clase** de expresii, echivalente sub **redenumiri** sistematice.
  - **Valoarea** este un anumit membru al acestei clase de echivalență.
- $\Rightarrow$  Valorile sunt **echivalente** în raport cu **redenumirea**.

# Modalități de reducere

## Cum putem organiza reducerea?

+ **Reducere stânga-dreapta:** Reducerea celui mai **superficial** și mai din **stânga**  $\beta$ -redex.

Ex Exemplu

$$((\lambda x.x \ \lambda x.y) \ (\lambda x.(x \ x) \ \lambda x.(x \ x))) \rightarrow (\lambda x.y \ \Omega) \rightarrow y$$

+ **Reducere dreapta-stânga:** Reducerea celui mai **adânc** și mai din **dreapta**  $\beta$ -redex.

Ex Exemplu

$$(\lambda x.(\lambda x.x \ \lambda x.y) \ (\lambda x.(x \ x) \ \lambda x.(x \ x))) \rightarrow (\lambda x.(\lambda x.x \ \lambda x.y) \ \Omega) \rightarrow \dots$$

**T | Teorema normalizării** Dacă o expresie este reductibilă, evaluarea **stânga-dreapta** a acesteia se termină.

- Teorema normalizării (normalizare = aducere la forma normală) **nu** garantează terminarea evaluării oricărei expresii, ci doar a celor **reductibile**!
- Dacă expresia este ireductibilă, **nicio** reducere nu se va termina.



- 1 Când se **termină** calculul? Se termină **întotdeauna**?  
→ se termină cu **forma normală [funcțională]**. **NU** se termină decât dacă expresia este **reductibilă**.
- 2 Comportamentul **depinde** de secvența de reducere?  
→ **DA**.
- 3 Dacă mai multe secvențe de reducere se termină, obținem întotdeauna **același** rezultat?  
→ **DA**.
- 4 Dacă rezultatul este unic, **cum** îl obținem?  
→ Reducere **stânga-dreapta**.
- 5 Care este valoarea expresiei?  
→ Forma normală [funcțională] (**FN[F]**).

- **+ | Evaluare aplicativă** (*eager*) – corespunde unei reduceri *mai degrabă dreapta-stânga*. Parametrii funcțiilor sunt evaluați *înaintea* aplicării funcției.
- **+ | Evaluare normală** (*lazy*) – corespunde reducerii *stânga-dreapta*. Parametrii funcțiilor sunt evaluați *la cerere*.
- **+ | Funcție strictă** – funcție cu evaluare *aplicativă*.
- **+ | Funcție nestrictă** – funcție cu evaluare *normală*.

- Evaluarea **aplicativă** prezentă în majoritatea limbajelor: C, Java, Scheme, PHP etc.

### Ex Exemplu

$(+ (+ 2 3) (* 2 3)) \rightarrow (+ 5 6) \rightarrow 11$

- Nevoie de funcții **nestricte**, chiar în limbajele aplicative: if, and, or etc.

### Ex Exemplu

$(\text{if } (< 2 3) (+ 2 3) (* 2 3)) \rightarrow (< 2 3) \rightarrow \#t \rightarrow (+ 2 3) \rightarrow 5$

# Limbajul lambda-0 și incursiune în TDA

- Am putea crea o mașină de calcul folosind calculul  $\lambda$  – mașină de calcul **ipotetică**;
- Mașina folosește limbaajul  $\lambda_0 \equiv$  calcul lambda;
- **Programul**  $\rightarrow$   $\lambda$ -expresie;
  - + Legări top-level de expresii la nume.
- **Datele**  $\rightarrow$   $\lambda$ -expresii;
- Funcționarea mașinii  $\rightarrow$  **reducere** – substituție textuală
  - evaluare normală;
  - terminarea evaluării cu forma normală funcțională;
  - se folosesc numai expresii închise.

# Tipuri de date

Cum reprezentăm datele? Cum interpretăm valorile?

- Putem reprezenta toate datele prin funcții cărora, **convențional**, le dăm o semnificație **abstractă**.

Ex) Exemplu

$$T \equiv_{\text{def}} \lambda x. \lambda y. x \qquad F \equiv_{\text{def}} \lambda x. \lambda y. y$$

- Pentru aceste **tipuri de date abstracte (TDA)** creăm operatori care transformă datele în mod coerent cu interpretarea pe care o dăm valorilor.

Ex) Exemplu

$$\begin{aligned} not &\equiv_{\text{def}} \lambda x. ((x \ F) \ T) \\ (not \ T) &\rightarrow (\lambda x. ((x \ F) \ T) \ T) \rightarrow ((T \ F) \ T) \rightarrow F \end{aligned}$$

+ **Tip de date abstract – TDA** – Model matematic al unei mulțimi de valori și al operațiilor valide pe acestea.

### : Componente

- **constructori de bază**: cum se generează valorile;
- **operatori**: ce se poate face cu acestea;
- **axiome**: cum lucrează operatorii / ce restricții există.

· Constructori:  $\left| \begin{array}{l} T : \rightarrow Bool \\ F : \rightarrow Bool \end{array} \right.$

· Operatori:  $\left| \begin{array}{l} not : Bool \rightarrow Bool \\ and : Bool^2 \rightarrow Bool \\ or : Bool^2 \rightarrow Bool \\ if : Bool \times A \times A \rightarrow A \end{array} \right.$

· Axiome:  $\left| \begin{array}{ll} not : not(T) = F & not(F) = T \\ and : and(T, a) = a & and(F, a) = F \\ or : or(T, a) = T & or(F, a) = a \\ if : if(T, a, b) = a & if(F, a, b) = b \end{array} \right.$





Intuiție

bazat pe comportamentul necesar pentru if: **selecția** între cele două valori

- $T \equiv_{\text{def}} \lambda x. \lambda y. x$
- $F \equiv_{\text{def}} \lambda x. \lambda y. y$

- $if \equiv_{\text{def}} \lambda c. \lambda x. \lambda y. ((c\ x)\ y)$
- $and \equiv_{\text{def}} \lambda x. \lambda y. ((x\ y)\ F)$ 
  - $((and\ T)\ a) \rightarrow ((\lambda x. \lambda y. ((x\ y)\ F)\ T)\ a) \rightarrow ((T\ a)\ F) \rightarrow a$
  - $((and\ F)\ a) \rightarrow ((\lambda x. \lambda y. ((x\ y)\ F)\ F)\ a) \rightarrow ((F\ a)\ F) \rightarrow F$
- $or \equiv_{\text{def}} \lambda x. \lambda y. ((x\ T)\ y)$ 
  - $((or\ T)\ a) \rightarrow ((\lambda x. \lambda y. ((x\ T)\ y)\ T)\ a) \rightarrow ((T\ T)\ a) \rightarrow T$
  - $((or\ F)\ a) \rightarrow ((\lambda x. \lambda y. ((x\ T)\ y)\ F)\ a) \rightarrow ((F\ T)\ a) \rightarrow a$
- $not \equiv_{\text{def}} \lambda x. ((x\ F)\ T)$ 
  - $(not\ T) \rightarrow (\lambda x. ((x\ F)\ T)\ T) \rightarrow ((T\ F)\ T) \rightarrow F$
  - $(not\ F) \rightarrow (\lambda x. ((x\ F)\ T)\ F) \rightarrow ((F\ F)\ T) \rightarrow T$

- Intuiție: pereche  $\rightarrow$  funcție ce așteaptă **selectorul**, pentru a-l aplica asupra membrilor
- $fst \equiv_{\text{def}} \lambda p.(p\ T)$ 
  - $(fst\ ((pair\ a)\ b)) \rightarrow (\lambda p.(p\ T)\ \lambda z.((z\ a)\ b)) \rightarrow (\lambda z.((z\ a)\ b)\ T) \rightarrow ((T\ a)\ b) \rightarrow a$
- $snd \equiv_{\text{def}} \lambda p.(p\ F)$ 
  - $(snd\ ((pair\ a)\ b)) \rightarrow (\lambda p.(p\ F)\ \lambda z.((z\ a)\ b)) \rightarrow (\lambda z.((z\ a)\ b)\ F) \rightarrow ((F\ a)\ b) \rightarrow b$
- $pair \equiv_{\text{def}} \lambda x.\lambda y.\lambda z.((z\ x)\ y)$ 
  - $((pair\ a)\ b) \rightarrow ((\lambda x.\lambda y.\lambda z.((z\ x)\ y)\ a)\ b) \rightarrow \lambda z.((z\ a)\ b)$



**Intuiție:** listă  $\rightarrow$  pereche (*head*, *tail*)

- $nil \equiv_{\text{def}} \lambda x. T$
- $cons \equiv_{\text{def}} pair$ 
  - $((cons\ e)\ L) \rightarrow ((\lambda x. \lambda y. \lambda z. ((z\ x)\ y)\ e)\ L) \rightarrow \lambda z. ((z\ e)\ L)$
- $car \equiv_{\text{def}} fst$        $cdr \equiv_{\text{def}} snd$



**Intuiție:** număr  $\rightarrow$  listă cu lungimea egală cu valoarea numărului

- $zero \equiv_{\text{def}} nil$
- $succ \equiv_{\text{def}} \lambda n. ((cons\ nil)\ n)$
- $pred \equiv_{\text{def}} cdr$

• vezi și [[http://en.wikipedia.org/wiki/Lambda\\_calculus#Encoding\\_datatypes](http://en.wikipedia.org/wiki/Lambda_calculus#Encoding_datatypes)]

- Modalitate de exprimare a **intenției** programatorului;
- **Documentare**: ce operatori acționează asupra căror obiecte;
- Reprezentarea **particulară** a valorilor de tipuri diferite:  
1, “Hello”, #t etc.;
- **Optimizarea** operațiilor specifice;
- Prevenirea **erorilor**;
- Facilitarea verificării **formale**;

- Un număr, o listă sau un arbore, posibil desemnate de **aceeași** valoare!
- Valori și operatori reprezentați de funcții, semnificația fiind dependentă de **context**.
- Valoare **aplicabilă** asupra unei alte valori  $\rightarrow$  operator!

- Incapacitatea Mașinii  $\lambda$  de a
  - interpreta **semnificația** expresiilor;
  - asigura **corectitudinea** acestora (dpdv al tipurilor).
- Delegarea celor două aspecte **programatorului**;
- **Orice** operatori aplicabili asupra **oricăror** valori;
- Construcții eronate **acceptate** fără avertisment, dar calcule terminate cu
  - valori **fără** semnificație *sau*
  - expresii care **nu** sunt valori (nu au asociată o semnificație), dar sunt **ireductibile**

→ **instabilitate**.

- **Flexibilitate** sporită în reprezentare;
  - Potrivită în situațiile în care reprezentarea **uniformă** obiectelor, ca liste de simboluri, este convenabilă.
- ... vin cu prețul unei dificultăți sporite în **depanare**, **verificare** și **mentenanță**



- Cum realizăm recursivitatea în  $\lambda_0$ , dacă nu avem nume de funcții?
  - **Textuală**: funcție care se autoapelează, folosindu-și **numele**;
  - **Semantică**: ce **obiect** matematic este desemnat de o funcție recursivă, cu posibilitatea construirii de funcții recursive **anonime**.

- Lungimea unei liste:

*length*  $\equiv_{\text{def}} \lambda L. (\text{if } (\text{null? } L) \text{ zero } (\text{succ } (\text{length } (\text{cdr } L))))$

- Cu ce **înlocuim** zona subliniată, pentru a evita recursivitatea textuală? (expresia pentru *length* nu este închisă!)
- Putem primi ca **parametru** o funcție echivalentă computațional cu *length*?  
*Length*  $\equiv_{\text{def}} \lambda f L. (\text{if } (\text{null? } L) \text{ zero } (\text{succ } (f (\text{cdr } L))))$
- $(\text{Length } \text{length}) = \text{length} \rightarrow \text{length}$  este un **punct fix** al lui *Length*!
- Cum **obținem** punctul fix?



$Fix = \lambda f.(\lambda x.(f (x x)) \ \lambda x.(f (x x)))$

- $(Fix \ F) \rightarrow (\lambda x.(F (x x)) \ \lambda x.(F (x x))) \rightarrow (F \ (\lambda x.(F (x x)) \ \lambda x.(F (x x)))) \rightarrow (F (Fix \ F))$
- $(Fix \ F)$  este un **punct fix** al lui  $F$ .
- $Fix$  se numește **combinator de punct fix**.
- $length \equiv_{\text{def}} (Fix \ Length) \sim (Length \ (Fix \ Length)) \sim \lambda L.(if \ (null? \ L) \ zero \ (succ \ ((Fix \ Length) \ (cdr \ L))))$
- Funcție recursivă, **fără** a fi textual recursivă!

# Racket vs. lambda-0

|                  | $\lambda$                      | Racket                                                  |
|------------------|--------------------------------|---------------------------------------------------------|
| Variabilă/nume   | $x$                            | <code>x</code>                                          |
| Funcție          | $\lambda x. corp$              | <code>(lambda (x) corp)</code>                          |
| uncurry          | $\lambda x y. corp$            | <code>(lambda (x y) corp)</code>                        |
| Aplicare         | $(F A)$                        | <code>(f a)</code>                                      |
| uncurry          | $(F A1 A2)$                    | <code>(f a1 a2)</code>                                  |
| Legare top-level | -                              | <code>(define nume expr)</code>                         |
| Program          | $\lambda$ -expresie<br>închisă | colecție de legări<br>top-level ( <code>define</code> ) |
| Valori           | $\lambda$ -expresii / TDA      | valori de diverse<br>tipuri (numere, liste,<br>etc.)    |

- similar cu  $\lambda_0$ , folosește S-expresii (bază Lisp);
- **tipat** – dinamic/latent
  - variabilele **nu** au tip;
  - valorile **au** tip (3, #f);
  - verificarea se face la **execuție**, în momentul aplicării unei funcții;
- evaluare **aplicativă**;
- permite recursivitate **textuală**;
- avem legări top-level.



19 Întârzierea evaluării

20 Fluxuri

21 Căutare leneșă în spațiul stărilor

# Întârzierea evaluării





Exemplu

Să se implementeze funcția **nestrictă** *prod*, astfel încât al doilea parametru să fie evaluat doar dacă primul este *true*:

- $prod(F, y) = 0$
- $prod(T, y) = y(y + 1)$

**Dar**, evaluarea parametrului *y* al funcției să se facă numai o singură dată.

· Problema de rezolvat: evaluarea **la cerere**.



```
1 (define prod
2   (lambda (x y)
3     (if x (* y (+ y 1)) 0)))
4
5 (define test
6   (lambda (x)
7     (let ((y 5))
8       (prod x (and (display "y ") y))))))
9 (test #f)
10 (test #t)
```

Output: y 0 | y 30

- Implementarea nu respectă **specificația**, deoarece **ambii** parametri sunt evaluați în momentul aplicării



```
1 (define prod
2   (lambda (x y)
3     (if x (* (eval y) (+ (eval y) 1)) 0)))
4
5 (define test
6   (lambda (x)
7     (let ((y 5))
8       (prod x (quote (and (display "y ") y))))))
9 (test #f)
10 (test #t)
```

Output: 0 | y undefined

- $x = \#f \rightarrow$  comportament corect: y neevaluat
- $x = \#t \rightarrow$  eroare: quote nu salvează contextul



+ **Context computațional** Contextul computațional al unui punct  $P$ , dintr-un program, la momentul  $t$ , este mulțimea variabilelor ale căror domenii de vizibilitate îl conțin pe  $P$ , la momentul  $t$ .

- Legare statică  $\rightarrow$  mulțimea variabilelor care îl conțin pe  $P$  în domeniul lexical de vizibilitate
- Legare dinamică  $\rightarrow$  mulțimea variabilelor definite cel mai recent, la momentul  $t$ , și referite din  $P$



**Ex** Exemplu Ce variabile locale conține contextul computațional al punctului *P*?

```
1 (lambda (x y)
2   (lambda (z)
3     (let ((x (car y)))
4       ; ..P..)))
```



+ **Închidere funcțională:** funcție care își salvează **contextul**, pe care îl va folosi, în momentul **aplicării**, pentru evaluarea corpului.

· **Notatie:** închiderea funcției  $f$  în contextul  $C \rightarrow \langle f; C \rangle$

**Ex** Exemplu

$\langle \lambda x.z; \{z \leftarrow 2\} \rangle$



```
1 (define prod
2   (lambda (x y)
3     (if x (* (y) (+ (y) 1)) 0))) ; (y)
4
5 (define test
6   (lambda (x)
7     (let ((y 5))
8       (prod x
9         (lambda () (and (display "y ") y))))))
10 (test #f)
11 (test #t)
```

Output: 0 | y y 30

- Comportament corect: y evaluat **la cerere** (deci leneș)
- x = #t → y evaluat de 2 ori → **ineficient**

# Varianta 4



## Promisiuni: delay & force

```
1 (define prod
2   (lambda (x y)
3     (if x (* (force y) (+ (force y) 1)) 0)))
4
5 (define test
6   (lambda (x)
7     (let ((y 5))
8       (prod x
9         (delay (and (display "y ") y))))))
10 (test #f)
11 (test #t)
```

Output: 0 | y 30

- Rezultat corect:  $y$  evaluat **la cerere**, o **singură dată**  
→ **evaluare leneșă eficientă**





- Rezultatul încă **neevaluat** al unei expresii
- Valori de **prim rang** în limbaj
- `delay`
  - construiește o promisiune;
  - funcție nestrictă.
- `force`
  - forțează respectarea unei promisiuni, evaluând expresia doar la **prima** aplicare, și **salvându-i** valoarea;
  - începând cu a doua invocare, întoarce, direct, valoarea **memorată**.



- Salvarea **contextului computațional** al expresiei a cărei evaluare este întârziată și evaluarea ei ulterioară în **acel** context → asemănător cu închiderile funcționale.
- Salvarea **rezultatului** primei evaluări a expresiei.
- **Distingerea** primei forțări de celelalte → **efect lateral**, dar acceptabil din moment ce legările se fac static – nu pot exista valori care se schimbă *între timp*.



# Evaluare întârziată

Abstractizare a implementării cu **promisiuni**

## Ex | Continuare a exemplului cu funcția prod

```
1 (define-syntax-rule (pack expr) (delay expr))
2
3 (define unpack force)
4
5 (define prod (lambda (x y)
6   (if x (* (unpack y) (+ (unpack y) 1)) 0)))
7 (define test (lambda (x)
8   (let ((y 5))
9     (prod x (pack (and (display "y ") y)))))))
```

· utilizarea nu depinde de implementare (am definit funcțiile `pack` și `unpack` care **abstractizează** implementarea concretă a evaluării întârziate.



# Evaluare întârziată

## Abstractizare a implementării cu închideri

### Ex) Continuare a exemplului cu funcția prod

```
1 (define-syntax-rule (pack expr) (lambda () expr) )
2
3 (define unpack (lambda (p) (p)))
4
5 (define prod (lambda (x y)
6   (if x (* (unpack y) (+ (unpack y) 1)) 0)))
7 (define test (lambda (x)
8   (let ((y 5))
9     (prod x (pack (and (display "y ") y)))) )))
```

· utilizarea nu depinde de implementare (același cod ca și anterior, altă implementare a funcționalității de evaluare întârziată, acum mai puțin eficientă).

# Fluxuri



**Ex** | Determinați suma numerelor pare<sup>1</sup> din intervalul  $[a, b]$ .

```
1 (define even-sum-iter ; varianta 1
2   (lambda (a b)
3     (let iter ((n a)
4                (sum 0))
5       (cond ((> n b) sum)
6             ((even? n) (iter (+ n 1) (+ sum n)))
7             (else (iter (+ n 1) sum))))))
8
9
10 (define even-sum-lists ; varianta 2
11   (lambda (a b)
12     (foldl + 0 (filter even? (interval a b)))))
```

<sup>1</sup>stă pentru o verificare potențial mai complexă, e.g. numere prime

Întârzierea evaluării

Fluxuri

Căutare în spațiul stărilor

Evaluare leneșă în Racket



- Varianta 1 – iterativă (d.p.d.v. proces):
  - **eficientă**, datorită spațiului suplimentar constant;
  - **ne-elegantă** → trebuie să implementăm generarea numerelor.
- Varianta 2 – folosește liste:
  - **ineficientă**, datorită spațiului posibil mare, ocupat la un moment dat – toate numerele din intervalul  $[a, b]$ .
  - **elegantă** și concisă;
- Cum **îmbinăm** avantajele celor 2 abordări? Putem stoca **procesul** fără a stoca **rezultatul** procesului?



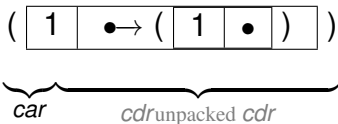
Fluxuri



- Secvențe construite **parțial**, extinse la cerere, ce creează **iluzia** completitudinii structurii;
- Îmbinarea **eleganței** manipulării listelor cu **eficiența** calculului incremental;
- Bariera de abstractizare:
  - componentele **listelor** evaluate la **construcție** (`cons`)
  - componentele **fluxurilor** evaluate la **selecție** (`cdr`)
- Construcție și utilizare:
  - **separate** la nivel conceptual → **modularitate**;
  - **întrepătrunse** la nivel de proces (utilizarea necesită construcția concretă).



- o listă este o **pereche**;
- explorarea listei se face prin operatorii `car` – primul element – și `cdr` – **restul** listei;
- am dori să **generăm** `cdr` algoritmic, dar **la cerere**.





- cons, car, cdr, nil, null?

```
1 (define-syntax-rule (stream-cons head tail)
2   (cons head (pack tail)))
3
4 (define stream-car car)
5
6 (define stream-cdr (lambda (s)
7   (unpack (cdr s))))
8
9 (define stream-nil '())
10
11 (define stream-null? null?)
```



- Definiție cu închideri:

```
(define ones (lambda ()(cons 1 (lambda ()(ones))))))
```

- Definiție cu fluxuri:

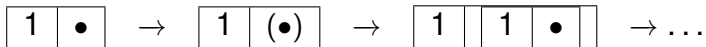
```
1 (define ones (stream-cons 1 ones))  
2 (stream-take 5 ones) ; (1 1 1 1 1)
```

- Definiție cu promisiuni:

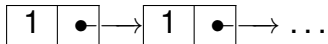
```
(define ones (delay (cons 1 ones)))
```



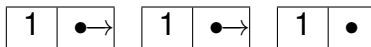
- Ca proces:



- Structural:



- Extinderea se realizează în spațiu constant:





```
1 (define naturals-from (lambda (n)
2   (stream-cons n (naturals-from (+ n 1)))))
3
4 (define naturals (naturals-from 0))

1 (define naturals
2   (stream-cons 0
3     (stream-zip-with + ones naturals)))
```

### · Atenție:

- Închideri: multiple parcurgeri ale fluxului determină **reevaluarea** porțiunilor deja explorate.
- Promisiuni: parcurgerea fluxului determină evaluarea **dincolo** de porțiunile deja explorate.

# Fluxul numerelor pare

## În două variante

---



```
1 (define even-naturals
2   (stream-filter even? naturals))
3
4 (define even-naturals
5   (stream-zip-with + naturals naturals))
```



- Ciurul lui **Eratostene**.
- Pornim de la fluxul numerelor **naturale**, începând cu 2.
- Elementul **curent** din fluxul inițial aparține fluxului numerelor prime.
- **Restul** fluxului generat se obține
  - eliminând **multiplii** elementului curent din fluxul inițial;
  - continuând procesul de **filtrare**, cu elementul următor.



```
1 (define sieve (lambda (s)
2   (if (stream-null? s) s
3       (stream-cons (stream-car s)
4                     (sieve (stream-filter
5                             (lambda (n) (not (zero?
6                                               (remainder n (stream-car s))))))
7                       (stream-cdr s)
8                     )))
9 )))
10
11 (define primes (sieve (naturals-from 2)))
```



# Căutare leneșă în spațiul stărilor



+ **Spațiul stărilor unei probleme** Mulțimea configurațiilor valide din universul problemei.



Exemplu

Fie problema  $Pal_n$ : *Să se determine palindroamele de lungime cel puțin  $n$ , ce se pot forma cu elementele unui alfabet fixat.*

**Stările** problemei → **toate** șirurile generabile cu elementele alfabetului respectiv.

# Specificarea unei probleme

Aplicație pe  $Pal_n$



- Starea **inițială**: șirul vid
- Operatorii de generare a stărilor **succesor** ale unei stări: inserarea unui caracter la începutul unui șir dat
- Operatorul de verificare a proprietății de **scop** a unei stări: palindrom



- Spațiul stărilor ca **graf**:
  - noduri: **stări**
  - muchii (orientate): **transformări** ale stărilor în stări succesori
- Posibile strategii de **căutare**:
  - lățime: **completă** și optimală
  - adâncime: **incompletă** și suboptimală



```
1 (define breadth-search-goal
2   (lambda (init expand goal?)
3     (letrec ((search (lambda (states)
4       (if (null? states) '()
5         (let ((state (car states)) (states (cdr states)))
6           (if (goal? state) state
7             (search (append states (expand state))))
8         )))))
9   (search (list init))))
```

- Generarea unei **singure** soluţii
- Cum le obţinem pe **celelalte**, mai ales dacă spaţiul e **infini**t?



```
1 (define lazy-breadth-search (lambda (init expand)
2   (letrec ((search (lambda (states)
3     (if (stream-null? states) states
4       (let ((state (stream-car states))
5         (states (stream-cdr states)))
6       (stream-cons state
7         (search (stream-append states
8           (expand state))))
9     )))))
10   (search (stream-cons init stream-nil)))
11 )))
```



```
1 (define lazy-breadth-search-goal
2   (lambda (init expand goal?)
3     (stream-filter goal?
4       (lazy-breadth-search init expand)))
5 ))
```

- Nivel înalt, conceptual: **separare** între explorarea spaţiului şi identificarea stărilor *scop*.
- Nivel scăzut, al instrucţiunilor: **întrepătrunderea** celor două aspecte.
- Aplicaţii:
  - Palindroame
  - Problema regiunilor



- Evaluare întârziată → variante de implementare
- Fluxuri → implementare și utilizări
- Căutare într-un spațiu infinit





22 Introducere

23 Sintaxă

24 Evaluaire

# Introducere

- din 1990;
- GHC – Glasgow Haskell Compiler (The Glorious Glasgow Haskell Compilation System)
  - dialect Haskell standard *de facto*;
  - compilează în/folosind C;
- Haskell Stack
- nume dat după logicianul Haskell Curry;
- aplicații: Pugs, Darcs, Linspire, Xmonad, Cryptol, seL4, Pandoc, web frameworks.

| Criteriu                | Racket                          | Haskell             |
|-------------------------|---------------------------------|---------------------|
| Funcții                 | <i>Curry</i> sau <i>uncurry</i> | <i>Curry</i>        |
| Tipare                  | Dinamică, tare (-liste)         | Statică, tare       |
| Legarea variabilelor    | Statică                         | Statică             |
| Evaluare                | Aplicativă                      | Normală (Leneșă)    |
| Transferul parametrilor | <i>Call by sharing</i>          | <i>Call by need</i> |
| Efecte laterale         | set ! *                         | Interzise           |

# Sintaxă

- toate funcțiile sunt *Curry*;
- aplicabile asupra **oricâtor** parametri la un moment dat.

**Ex** | Exemplu : Definiții **echivalente** ale funcției `add`:

```
1  add1          =    \x y -> x + y
2  add2          =    \x -> \y -> x + y
3  add3 x y      =    x + y
4
5  result        =    add1 1 2      -- echivalent, ((add1 1) 2)
6  result2       =    add3 1 2      -- echivalent, ((add3 1) 2)
7  inc           =    add1 1
```

- Aplicabilitatea **parțială** a operatorilor infixati
- **Transformări** operator  $\rightarrow$  funcție și funcție  $\rightarrow$  operator

**Ex** Exemplu Definiții **echivalente** ale funcțiilor `add` și `inc`:

```
1  add4          =      (+)
2  result1       =      (+) 1 2
3  result2       =      1 'add4' 2
4
5  inc1          =      (1 +)
6  inc2          =      (+ 1)
7  inc3          =      (1 'add4 ')
8  inc4          =      ('add4 ' 1)
```

- Definirea comportamentului funcțiilor pornind de la **structura** parametrilor  
→ traducerea axiomelor TDA.

## Ex Exemplu

```
1  add5  0  y      =      y      --  add5  1  2
2  add5  (x + 1) y  =      1 + add5  x  y
3
4  sumList []      =      0      --  sumList  [1,2,3]
5  sumList (hd:tl) =      hd + sumList tl
6
7  sumPair (x, y)   =      x + y      --  sumPair  (1,2)
8
9  sumTriplet (x, y, z@(hd:_)) =      --  sumTriplet
10      x + y + hd + sumList z      --  (1,2,[3,4,5])
```



- Definirea listelor prin **proprietățile** elementelor, ca într-o specificare matematică

## Ex Exemplu

```
1 squares lst      = [x * x | x <- lst]
2
3 quickSort []      = []
4 quickSort (h:t)   = quickSort [x | x <- t, x <= h]
5                   ++ [h]
6                   ++ quickSort [x | x <- t, x > h]
7
8 interval          = [0 .. 10]
9 evenInterval      = [0, 2 .. 10]
10 naturals          = [0 ..]
```

# Evaluare

- Evaluare **leneșă**: parametri evaluați **la cerere**, **cel mult** o dată, eventual **parțial**, în cazul obiectelor structurate
- Transferul parametrilor: *call by need*
- Funcții **nestricte**!



## Exemplu

```
1 f (x, y) z = x + x
```

Evaluare:

```
1 f (2 + 3, 3 + 5) (5 + 8)
```

```
2 → (2 + 3) + (2 + 3)
```

```
3 → 5 + 5      reutilizăm rezultatul primei evaluări!
```

```
4 → 10              ceilalți parametri nu sunt evaluați
```

# Pași în aplicarea funcțiilor

## Exemplu



### Ex Exemplu

```
1  frontSum (x:y:zs)  =  x + y
2  frontSum [x]       =  x
3
4  notNil []          =  False
5  notNil (_:_)       =  True
6
7  frontInterval m n
8      | notNil xs =  frontSum xs
9      | otherwise =  n
10     where
11         xs      =  [m .. n]
```



- 1 **Pattern matching**: evaluarea parametrilor **suficient** cât să se constate (ne-)potrivirea cu *pattern*-ul;
- 2 Evaluarea **gărzilor** ( `|` );
- 3 Evaluarea variabilelor **locale**, **la cerere** (`where`, `let`).

# Pași în aplicarea funcțiilor

## Exemplu – revisited



### Ex execuția exemplului anterior

```
1 frontInterval 3 5
2 ?? notNil xs
3 ?? where
4 ?? xs = [3 .. 5]
5 ?? → 3:[4 .. 5]
6 ?? → notNil (3:[4 .. 5])
7 ?? → True
8 → frontSum xs
9 where
10 xs = 3:[4 .. 5]
11 → 3:4:[5]
12 → frontSum (3:4:[5])
13 → 3 + 4 → 7
```

evaluare pattern

evaluare prima gardă

necesar  $xs \rightarrow$  evaluare where

evaluare valoare gardă

$xs$  deja calculat

- Evaluarea **parțială** a structurilor – liste, tupluri etc.
- Listele sunt, implicit, văzute ca **fluxuri**!

## Exemplu

```
1  ones                =    1 : ones
2
3  naturalsFrom n      =    n : (naturalsFrom (n + 1))
4  naturals1           =    naturalsFrom 0
5  naturals2           =    0 : (zipWith (+) ones naturals2)
6
7  evenNaturals1       =    filter even naturals1
8  evenNaturals2       =    zipWith (+) naturals1 naturals2
9
10 fibo                =    0 : 1 : (zipWith (+) fibo (tail fibo))
```



- Haskell, diferențe față de Racket
- pattern matching și list comprehensions
- evaluare în Haskell





25 Tipare

26 Sinteză de tip

27 TDA

# Tipare

- Tipuri ca **mulțimi** de valori:
  - `Bool = {True, False}`
  - `Natural = {0, 1, 2, ...}`
  - `Char = {'a', 'b', 'c', ...}`
- **Rolul** tipurilor (vezi cursuri anterioare);
- Tipare **statică**:
  - etapa de tipare **anterioară** etapei de evaluare;
  - asocierea **fiecărei** expresii din program cu un tip;
- Tipare **tare**: absența conversiilor **implicite** de tip;
- Expresii de:
  - **program**: `5, 2 + 3, x && (not y)`
  - **tip**: `Integer, [Char], Char -> Bool, a`

### Exemplu

```
1 5 :: Integer
2 'a' :: Char
3 (+1) :: Integer -> Integer
4 [1,2,3] :: [Integer] -- liste de un singur tip !
5 (True, "Hello") :: (Bool, [Char])
6 etc.
```

- Tipurile de bază sunt tipurile elementare din limbaj:

Bool, Char, Integer, Int, Float, ...

- Reprezentare uniformă:

```
1 data Integer = ... | -2 | -1 | 0 | 1 | 2 | ...
2 data Char = 'a' | 'b' | 'c' | ...
```

# Constructori de tip

⇒ tipuri noi pentru valori sau funcții

- **Funcții** de tip, ce **îmbogățesc** tipurile din limbaj.

## Ex Constructori de tip predefiniți

```

1  -- Constructorul de tip functie: ->
2  (-> Bool Bool) ⇒ Bool -> Bool
3  (-> Bool (Bool -> Bool)) ⇒ Bool -> (Bool -> Bool)
4
5  -- Constructorul de tip lista: []
6  ([] Bool) ⇒ [Bool]
7  ([] [Bool]) ⇒ [[Bool]]
8
9  -- Constructorul de tip tuplu: (,...,)
10 ((,) Bool Char) ⇒ (Bool, Char)
11 ((,,) Bool ((,) Char [Bool]) Bool)
12           ⇒ (Bool, (Char, [Bool]), Bool)

```



- Constructorul `->` este asociativ **dreapta**:

`Integer -> Integer -> Integer`

$\equiv$  `Integer -> (Integer -> Integer)`

### Exemplu

```
1 add6      :: Integer -> Integer -> Integer
2 add6 x y  = x + y
3
4 f         :: (Integer -> Integer) -> Integer
5 f g      = (g 3) + 1
6
7 idd      :: a -> a           -- funcție polimorfică
8 idd x    = x                -- a: variabilă de tip!
```

# Sinteză de tip



+ **Sinteză de tip – *type inference*** – Determinarea **automată** a tipului unei expresii, pe baza unor reguli precise.

- Adnotările **explicite** de tip, deși posibile, **necesare** în majoritatea cazurilor
- Dependentă de:
  - **componentele** expresiei
  - **contextul** lexical al expresiei
- Reprezentarea tipurilor → **expresii** de tip:
  - **constante** de tip: tipuri de bază;
  - **variabile** de tip: pot fi legate la orice expresii de tip;
  - **aplicații** ale constructorilor de tip pe expresii de tip.





+ | **Progres** O expresie bine-tipată (căreia i se poate asocia un tip):

- este o **valoare** (nu este o aplicare de funcție) *sau*
- (este aplicarea unei funcții și) poate fi **redușă** (vezi  $\beta$ -redex).

+ | **Conservare** Evaluarea unei expresii bine-tipate produce o expresie **bine-tipată** – de obicei, cu același tip.

- dacă **sinteza de tip** pentru expresia  $E$  dă tipul  $t$ , atunci după reducere, valoarea expresiei  $E$  va fi de tipul  $t$ .

# Exemple de sinteză de tip

## Câteva reguli simplificate de sinteză de tip

---

- Formă: 
$$\frac{\text{premise-1} \dots \text{premise-m}}{\text{concluzie-1} \dots \text{concluzie-n}} \text{ (nume)}$$

---

- Funcție: 
$$\frac{\text{Var} :: a \quad \text{Expr} :: b}{\backslash \text{Var} \rightarrow \text{Expr} :: a \rightarrow b} \text{ (TLambda)}$$
- Aplicație: 
$$\frac{\text{Expr1} :: a \rightarrow b \quad \text{Expr2} :: a}{(\text{Expr1 Expr2}) :: b} \text{ (TApp)}$$
- Operatorul +: 
$$\frac{\text{Expr1} :: \text{Int} \quad \text{Expr2} :: \text{Int}}{\text{Expr1} + \text{Expr2} :: \text{Int}} \text{ (T+)}$$
- Literalii întregi: 
$$\frac{}{0, 1, 2, \dots :: \text{Int}} \text{ (TInt)}$$



### Ex Exemplul 1

1 `f g = (g 3) + 1`

$$\frac{g :: a \quad (g\ 3) + 1 :: b}{f :: a \rightarrow b} \text{ (TLambda)}$$

$$\frac{(g\ 3) :: \text{Int} \quad 1 :: \text{Int}}{(g\ 3) + 1 :: \text{Int}} \text{ (T+)}$$

$$\Rightarrow b = \text{Int}$$

$$\frac{g :: c \rightarrow d \quad 3 :: c}{(g\ 3) :: d} \text{ (TApp)}$$

$$\Rightarrow a = c \rightarrow d, c = \text{Int}, d = \text{Int}$$

$$\Rightarrow f :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$$



### Exemplul 2

```
1 fix f = f (fix f)
```

$$\frac{f :: a \quad f (\text{fix } f) :: b}{\text{fix} :: a \rightarrow b} \quad (\text{TLambda})$$

$$\frac{f :: c \rightarrow d \quad (\text{fix } f) :: c}{(f (\text{fix } f)) :: d} \quad (\text{TApp})$$

$$\Rightarrow a = c \rightarrow d, b = d$$

$$\frac{\text{fix} :: e \rightarrow g \quad f :: e}{(\text{fix } f) :: g} \quad (\text{TApp})$$

$$\Rightarrow a \rightarrow b = e \rightarrow g, a = e, b = g, c = g$$

$$\Rightarrow \text{fix} :: (c \rightarrow d) \rightarrow b = (g \rightarrow g) \rightarrow g$$



### Ex Exemplul 3

1 `f x = (x x)`

$$\frac{x :: a \quad (x \ x) :: b}{f :: a \rightarrow b} \text{ (TLambda)}$$

$$\frac{x :: c \rightarrow d \quad x :: c}{(x \ x) :: d} \text{ (TApp)}$$

Ecuția  $c \rightarrow d = c$  **nu** are soluție ( $\nexists$  tipuri recursive)  
 $\Rightarrow$  funcția **nu** poate fi tipată.



- la baza sintezei de tip: **unificarea** → legarea variabilelor în timpul procesului de sinteză, în scopul **unificării** diverselor formule de tip elaborate.

**+ Unificare** Procesul de identificare a valorilor **variabilelor** din 2 sau mai multe formule, astfel încât **substituirea** variabilelor prin valorile asociate să conducă la **coincidența** formulelor.

**+ Substituție** O substituție este o mulțime de **legări** variabilă - valoare.



- O **variabilă de tip**  $a$  unifică cu o **expresie de tip**  $E$  doar dacă:
  - $E = a$  *sau*
  - $E \neq a$  și  $E$  nu conține  $a$  (*occurrence check*).  
Exemplu:  $a$  unifică cu  $b \rightarrow c$  dar nu cu  $a \rightarrow b$ .
- **2 constante** de tip unifică doar dacă sunt egale;
- **2 aplicații** de tip unifică doar dacă implică același constructor de tip și argumente ce unifică recursiv.

### Exemplu

- Pentru a unifica expresiile de tip:
  - $t1 = (a, [b])$
  - $t2 = (Int, c)$
- putem avea substituțiile (variante):
  - $S1 = \{a \leftarrow Int, b \leftarrow Int, c \leftarrow [Int]\}$
  - $S2 = \{a \leftarrow Int, c \leftarrow [b]\}$
- Forme comune pentru  $s1$  respectiv  $s2$ :
  - $t1/S1 = t2/S1 = (Int, [Int])$
  - $t1/S2 = t2/S2 = (Int, [b])$

+ | ***Most general unifier – MGU*** Cea mai **generală** substituție sub care formulele unifică. Exemplu:  $s2$ .



### Exemplu

- Tipurile:  $t1 = (a, [b])$  ,  $t2 = (Int, c)$ 
  - MGU:  $S = \{a \leftarrow Int, c \leftarrow [b]\}$
  - Tipuri mai particulare (instance):  $(Integer, [Integer]), (Integer, [Char]),$  etc
- Funcția:  $\backslash x \rightarrow x$ 
  - Tipuri corecte:  $Int \rightarrow Int$  ,  $Bool \rightarrow Bool$  ,  $a \rightarrow a$

**+ | Tip principal al unei expresii** – Cel mai **general** tip care descrie **complet** natura expresiei. Se obține prin utilizarea MGU.

# TDA

# Constructorul de tip Natural

## Exemplu de definire TDA 1



### Ex Exemplu

```
1 data Natural      =    Zero
2                   | Succ Natural
3     deriving (Show, Eq)
4
5 unu                =    Succ Zero
6 doi                =    Succ unu
7
8 addNat Zero n      =    n
9 addNat (Succ m) n  =    Succ (addNat m n)
```



- Constructor de `tip`: `Natural`
  - nular;
  - **se confundă** cu tipul pe care-l construiește.
- Constructori de `date`:
  - `Zero`: nular
  - `Succ`: unar
- Constructorii de date ca **funcții**, dar utilizabile în *pattern matching*.

```
1 Zero :: Natural
2 Succ :: Natural -> Natural
```

# Constructorul de tip `Pair`

## Exemplu de definire TDA 2



### Ex Exemplu

```
1 data Pair a b    = P a b
2     deriving (Show, Eq)
3
4 pair1             =    P 2 True
5 pair2             =    P 1 pair1
6
7 myFst (P x y)     =    x
8 mySnd (P x y)     =    y
```



- Constructor de `tip`: `Pair`
  - polimorfic, binar;
  - generează un tip în momentul **aplicării** asupra 2 tipuri.

- Constructor de `date`: `P`, binar:

```
1 P :: a -> b -> Pair a b
```



- tipuri în Haskell
- expresii de tip și construcție de tipuri
- sinteză de tip, unificare



28 Motivație

29 Clase Haskell

30 Aplicații ale claselor



# Motivație

+ **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: `==`.



### Ex Exemplu

Să se definească operația `show`, capabilă să producă reprezentarea oricărui obiect ca șir de caractere. Comportamentul este **specific** fiecărui tip (polimorfism **ad-hoc**).

```
1 show 3 → "3"
2 show True → "True"
3 show 'a' → "'a'"
4 show "a" → "\"a\""
```



# Motivație

## Varianta 1 – Funcții dedicate fiecărui tip

---

```
1 showBool True    = "True"
2 showBool False   = "False"
3
4 showChar c        = "'" ++ [c] ++ "'"
5
6 showString s       = "\" ++ s ++ "\"
```

- Dorim să implementăm funcția `showNewLine`, care adaugă caracterul “linie nouă” la reprezentarea ca șir:

```
1 showNewLine x = (show...? x) ++ "\n"
```

- `showNewLine` **nu** poate fi polimorfică  $\Rightarrow$  avem nevoie de `showNewLineBool`, `showNewLineChar` etc.

- Alternativ, trimiterea ca **parametru** a funcției `show*` corespunzătoare:

```
1 showNewLine sh x = (sh x) ++ "\n"  
2 showNewLineBool = showNewLine showBool
```

- **Prea general**, fiind posibilă trimiterea unei funcții cu alt comportament, în măsura în care respectă tipul.

- într-un limbaj care suportă supraîncărcarea operatorilor / funcțiilor, aș defini câte o funcție `show` pentru fiecare tip care suportă afișare (cum este `toString` în Java)
- dar cum pot defini în mod coerent tipul lui `showNewLine`?

“`showNewLine` poate primi ca argument orice tip a supraîncărcat funcția `show`.”

⇒ **Clasa** (*mulțimea de tipuri*) `Show`, care necesită implementarea funcției `show`.



# Motivație

## Varianta 2 – Supraîncărcarea funcției → funcție polimorfică ad-hoc

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

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

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

```
1 instance Show Bool where
2     show True  = "True"
3     show False = "False"
4
5 instance Show Char where
6     show c = "'" ++ [c] ++ "'"
```

⇒ Funcția `showNewLine` **polimorfică!**

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

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

```
1 show      :: Show a => a -> String
2 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`.*

- **Context**: constrângeri suplimentare asupra variabilelor din tipul funcției:

$\underbrace{\text{Show } a \Rightarrow}_{\text{context}}$

- **Propagarea** constrângerilor din contextul lui `show` către contextul lui `showNewLine`.





- Contexte utilizabile și la **instanțiere**:

```
1 instance (Show a, Show b) => Show (a, b) where
2     show (x, y) = "(" ++ (show x)
3                   ++ ", " ++ (show y)
4                   ++ ")"
```

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

# Clase Haskell



## Haskell

- **Tipurile** sunt mulțimi de **valori**;
- **Clasele** sunt mulțimi de **tipuri**; tipurile *aderă* la clase;
- **Instanțierea** claselor de către tipuri pentru ca funcțiile definite în clasă să fie disponibile pentru valorile tipului;
- Operațiile specifice clasei sunt implementate în cadrul declarației de instanțiere.

## POO (e.g. Java)

- **Clasele** sunt mulțimi de **obiecte** (**instanțe**);
- **Interfețele** sunt mulțimi de **clase**; clasele *implementează* interfețe;
- **Implementarea** interfețelor de către clase pentru ca funcțiile definite în interfață să fie disponibile pentru instanțele clasei;
- Operațiile specifice interfeței sunt implementate în cadrul definiției clasei.



+ **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. Exemplu: tipul `Bool` în raport cu clasa `Show`.

- *clasa* definește funcțiile **suportate**;
- clasa se definește peste o variabilă care stă pentru **constructorul unui tip**;
- *instanța* definește **implementarea** funcțiilor.

```
1 class Show a where
2     show :: a -> String
3
4 class Eq a where
5     (==), (/=) :: a -> a -> Bool
6     x /= y      = not (x == y)
7     x == y      = not (x /= y)
```

- Posibilitatea scrierii de definiții **implicite** (v. liniile 6–7).
- Necesitatea suprascrierii **cel puțin unuia** din cei 2 operatori ai clasei `Eq` pentru instanțierea corectă.

```
1 class Eq a => Ord a where
2     (<), (<=), (>=), (>) :: a -> a -> Bool
3     ...
```

- contextele – utilizabile și la **definirea** unei clase.
- clasa `Ord` **moștenește** clasa `Eq`, cu preluarea operațiilor din clasa moștenită.
- este **necesară** aderarea la clasa `Eq` în momentul instanțierii clasei `Ord`.
- este **suficientă** supradefinirea lui `(<=)` la instanțiere.



- **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`.
- ```
1 data Alarm = Soft | Loud | Deafening
2     deriving (Eq, Ord, Show)
```
- variabilele de tipul `Alarm` pot fi comparate, testate la egalitate, și afișate.

# Aplicații ale claselor





 invert

Fie constructorii de tip:

```
1 data Pair a = P a a
2
3 data NestedList a
4     =     Atom a
5     |     List [NestedList a]
```

Să se definească operația `invert`, aplicabilă pe valori de tipuri diferite, inclusiv `Pair a` și `NestedList a`, comportamentul fiind **specific** fiecărui tip.



## invert

### Implementare

---

```
1 class Invertible a where
2     invert  ::  a -> a
3     invert  =    id
4
5 instance Invertible (Pair a) where
6     invert (P x y) = P y x
7
8 instance Invertible a => Invertible (NestedList a) where
9     invert (Atom x) =    Atom (invert x)
10    invert (List x) =    List $ reverse $ map invert x
11
12 instance Invertible a => Invertible [a] where
13     invert lst = reverse $ map invert lst
14 instance Invertible Int ...
```

- Necesitatea **contextului**, în cazul tipurilor `[a]` și `NestedList a`, pentru inversarea elementelor **înselor**.



### contents

Să se definească operația `contents`, aplicabilă pe obiecte **structurate**, inclusiv pe cele aparținând tipurilor `Pair a` și `NestedList a`, care întoarce elementele din componență, sub forma unei **liste** Haskell.

```
1 class Container a where
2     contents :: a -> [...?]
```

- `a` este tipul unui **container**, e.g. `NestedList b`
- Elementele listei întoarse sunt cele **din container**
- Cum **precizăm** tipul acestora (`b`)?

## contents

### Varianta 1a

---

```
1 class Container a where
2     contents :: a -> [a]
3
4 instance Container [x] where
5     contents = id
```

Testăm pentru `contents [1,2,3]`:

- Conform definiției clasei:

```
1 contents :: Container [a] => [a] -> [[a]]
```

- Conform supraîncărcării funcției (`id`):

```
1 contents :: Container [a] => [a] -> [a]
```

- Ecuația `[a] = [[a]]` **nu** are soluție  $\Rightarrow$  **eroare**.

```

1 class Container a where
2     contents :: a -> [b]
3
4 instance Container [x] where
5     contents = id

```

Testăm pentru `contents [1,2,3]`:

- Conform definiției clasei:

```

1 contents :: Container [a] => [a] -> [b]

```

- Conform supraîncărcării funcției (`id`):

```

1 contents :: Container [a] => [a] -> [a]

```

- Ecuația `[a] = [b]` **are** soluție pentru `a = b`, dar  
tipul `[a] -> [a]` **insuficient** de general (prea specific) în raport cu `[a] -> [b]`  
`[b] ⇒ eroare!`



**Soluție** clasa primește **constructorul** de tip, și nu tipul container propriu-zis (rezultat după aplicarea constructorului)  $\Rightarrow$  includem tipul conținut de container în expresia de tip a funcției `contents`:

```

1  class Container t where
2      contents :: t a -> [a]
3
4  instance Container Pair where
5      contents (P x y) = [x, y]
6
7  instance Container NestedList where
8      contents (Atom x)    = [x]
9      contents (Seq x)     = concatMap contents x
10
11 instance Container [] where contents = id

```



```
1  fun1 :: Eq a => a -> a -> a -> a
2  fun1 x y z  =    if x == y then x else z
3
4  fun2 :: (Container a, Invertible (a b),
5  Eq (a b)) => (a b) -> (a b) -> [b]
6  fun2 x y      =    if (invert x) == (invert y)
7                      then contents x
8                      else contents y
9
10 fun3 :: Invertible a => [a] -> [a] -> [a]
11 fun3 x y      =    (invert x) ++ (invert y)
12
13 fun4 :: Ord a => a -> a -> a -> a
14 fun4 x y z    =    if x == y then z else
15                      if x > y then x else y
```



- **Simplificarea** contextului lui `fun3`, de la `Invertible [a]` la `Invertible a`.
- **Simplificarea** contextului lui `fun4`, de la `(Eq a, Ord a)` la `Ord a`, din moment ce clasa `Ord` este **derivată** din clasa `Eq`.





- Clase Haskell
- polimorfism ad-hoc, instanțiere de clase
- derivare a unei clase, context



## 31 Introducere în Prolog

# Introducere în Prolog



- introdus în anii 1970 ;
- programul  $\rightarrow$  mulțime de propoziții logice în LPOI;
- mediul de execuție = demonstrator de teoreme care spune:
  - dacă un fapt este adevărat sau fals;
  - în ce condiții este un fapt adevărat.
- Resursă Prolog pe Wikibooks:

`[https://en.wikibooks.org/wiki/Prolog]`



- fundamentare teoretică a procesului de raționament;
- motor de raționament ca unic mod de execuție;
  - modalități limitate de control al execuției.
- căutare automată a valorilor pentru variabilele nelegate (dacă este necesar);
- posibilitatea demonstrațiilor și deducțiilor **simbolice**.



- Introducere în Prolog



32 Procesul de demonstrare

33 Controlul execuției

# Procesul de demonstrare





- 1 Inițializarea **stivei de scopuri** cu scopul solicitat;
- 2 Inițializarea **substituției** (utilizate pe parcursul unificării) cu mulțimea vidă;
- 3 Extragerea scopului din **vârful** stivei și determinarea **primei** clauze din program cu a cărei concluzie **unifică**;
- 4 Îmbogățirea corespunzătoare a **substituției** și adăugarea **premiselor** clauzei în stivă, în ordinea din program;
- 5 Salt la pasul 3.



- ⑥ În cazul **imposibilității** satisfacerii scopului din vârful stivei, **revenirea** la scopul anterior (*backtracking*), și încercarea altei modalități de satisfacere;
- ⑦ **Succes** la **golirea** stivei de scopuri;
- ⑧ **Eșec** la imposibilitatea satisfacerii **ultimului** scop din stivă.

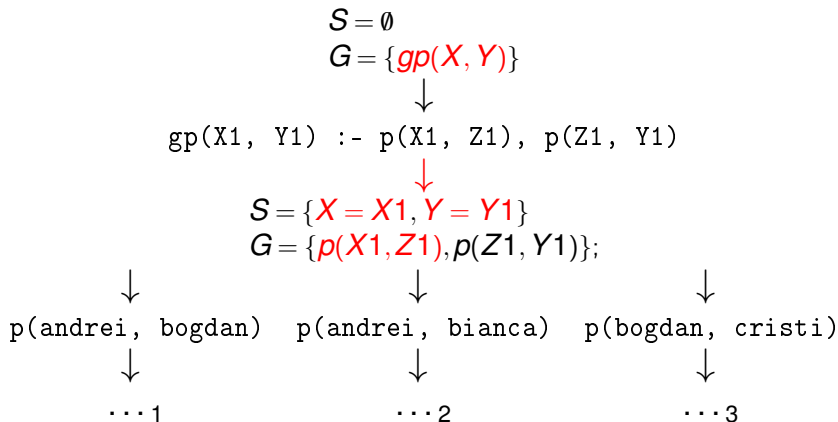


## Exemplu

```
1 parent(andrei, bogdan).
2 parent(andrei, bianca).
3 parent(bogdan, cristi).
4
5 grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

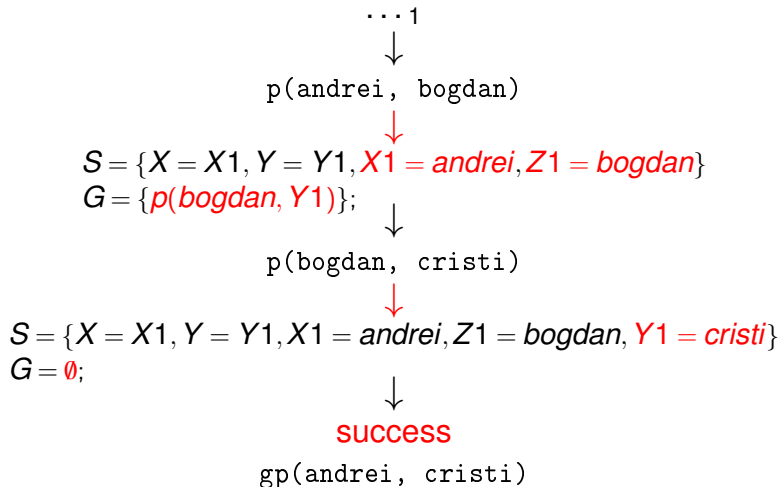
- $true \Rightarrow \text{parent}(\text{andrei}, \text{bogdan})$
- $true \Rightarrow \text{parent}(\text{andrei}, \text{bianca})$
- $true \Rightarrow \text{parent}(\text{bogdan}, \text{cristi})$
- $\forall x. \forall y. \forall z. (\text{parent}(x, z) \wedge \text{parent}(z, y) \Rightarrow \text{grandparent}(x, y))$

# Exemplul genealogic (1)



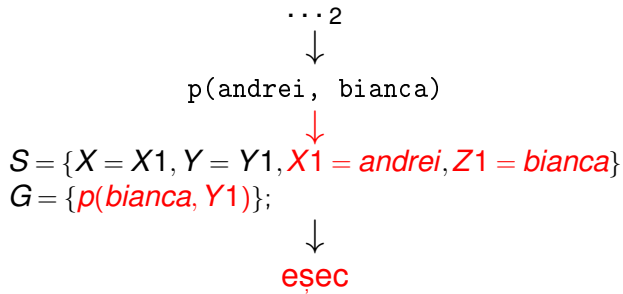
# Exemplul genealogic (2)

## Ramura 1



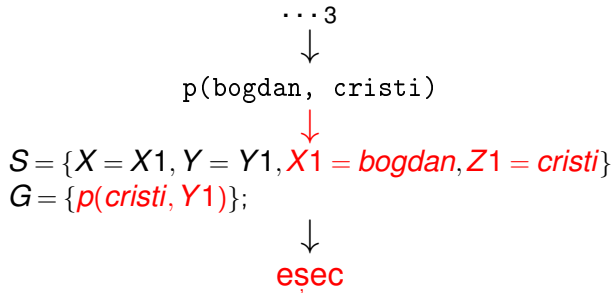
# Exemplul genealogic (3)

## Ramura 2



# Exemplul genealogic (4)

## Ramura 3





- Ordinea evaluării / încercării demonstrării scopurilor
  - Ordinea **clauzelor** în program;
  - Ordinea **premiselor** în cadrul regulilor.
- Recomandare: premisele **mai ușor** de satisfăcut și **mai specifice** primele – exemplu: axiome.





### *Forward chaining (data-driven)*

- Derivarea **tuturor** concluziilor, pornind de la datele inițiale;
- **Oprire** la obținerea scopului (scopurilor);

### *Backward chaining (goal-driven)*

- Utilizarea **exclusivă** a regulilor care pot contribui efectiv la satisfacerea scopului;
- Determinarea regulilor a căror concluzie **unifică** cu scopul;
- Încercarea de satisfacere a **premiselor** acestor reguli ș.a.m.d.

# Strategii de control

## Algoritm Backward chaining



1. **BackwardChaining**(*rules*, *goals*, *subst*)  
lista **regulilor** din program, stiva de **scopuri**, **substituiția** curentă, inițial vidă.  
**returns** satisfiabilitatea scopurilor
2. **if** *goals* =  $\emptyset$  **then**
3.     **return** SUCCESS
4.     *goal*  $\leftarrow$  *head*(*goals*)
5.     *goals*  $\leftarrow$  *tail*(*goals*)
6.     **for-each** *rule*  $\in$  *rules* **do**     // în ordinea din program
7.         **if** *unify*(*goal*, *conclusion*(*rule*), *subst*)  $\rightarrow$  *bindings*
8.             *newGoals*  $\leftarrow$  *premises*(*rule*)  $\cup$  *goals*     // **adâncime**
9.             *newSubst*  $\leftarrow$  *subst*  $\cup$  *bindings*
10.            **if** *BackwardChaining*(*rules*, *newGoals*, *newSubst*)
11.                **then return** SUCCESS
12. **return** FAILURE

# Controlul execuției

# Exemplu – Minimul a două numere

## Cod Prolog



### Ex Minimul a două numere

```
1 min(X, Y, M) :- X =< Y, M is X.
2 min(X, Y, M) :- X > Y, M is Y.
3
4 min2(X, Y, M) :- X =< Y, M = X.
5 min2(X, Y, M) :- X > Y, M = Y.
6
7 % Echivalent cu min2.
8 min3(X, Y, X) :- X =< Y.
9 min3(X, Y, Y) :- X > Y.
```

# Exemplu – Minimul a două numere



## Utilizare

---

```
1  ?- min(1+2, 3+4, M).
2  M = 3 ;
3  false.
4
5  ?- min(3+4, 1+2, M).
6  M = 3.
7
8  ?- min2(1+2, 3+4, M).
9  M = 1+2 ;
10 false.
11
12 ?- min2(3+4, 1+2, M).
13 M = 1+2.
```

# Exemplu – Minimul a două numere



## Observații

- Condiții mutual exclusive:  $X \leq Y$  și  $X > Y \rightarrow$  cum putem **elimina** redundanța?

### Ex Exemplu

```
1 min4(X, Y, X) :- X <= Y.  
2 min4(X, Y, Y).
```

```
1 ?- min4(1+2, 3+4, M).  
2 M = 1+2 ;  
3 M = 3+4.
```

- **Greșit!**

# Exemplu – Minimul a două numere

## Îmbunătățire



- Soluție: **oprirea** recursivității după prima satisfacere a scopului.



### Exemplu

```
1 min5(X, Y, X) :- X =< Y, !.  
2 min5(X, Y, Y).
```

```
1 ?- min5(1+2, 3+4, M).  
2 M = 1+2.
```



- La **prima** întâlnire  $\rightarrow$  **satisfacere**;
- La **a doua** întâlnire în momentul revenirii (*backtracking*)  $\rightarrow$  **eșec**, cu inhibarea **tuturor** căilor ulterioare de satisfacere a scopului care a unificat cu concluzia regulii curente;
- Utilitate în **eficientizarea** programelor.





### Exemplu

```
1  girl(mary).
2  girl(ann).
3
4  boy(john).
5  boy(bill).
6
7  pair(X, Y) :- girl(X), boy(Y).
8  pair(bella, harry).
9
10 pair2(X, Y) :- girl(X), !, boy(Y).
11 pair2(bella, harry).
```



```
1  ?- pair(X, Y).  
2  X = mary,  
3  Y = john ;  
4  X = mary,  
5  Y = bill ;  
6  X = ann,  
7  Y = john ;  
8  X = ann,  
9  Y = bill ;  
10 X = bella,  
11 Y = harry.
```

```
1  ?- pair2(X, Y).  
2  X = mary,  
3  Y = john ;  
4  X = mary,  
5  Y = bill.
```



## Ex Exemplu

```
1 nott(P) :- P, !, fail.  
2 nott(P).
```

- P: atom – exemplu: boy(john)
- dacă P este **satisfiabil**:
  - eșecul **primei** reguli, din cauza lui fail;
  - abandonarea celei **de-a doua** reguli, din cauza lui !;
  - rezultat: nott(P) **nesatisfiabil**.
- dacă P este **nesatisfiabil**:
  - eșecul **primei** reguli;
  - succesul celei **de-a doua** reguli;
  - rezultat: nott(P) **satisfiabil**.



- Prolog: structura unui program, funcționarea unei demonstrații
- ordinea evaluării, algoritmul de control al demonstrației
- tehnici de control al execuției.

- 34 Logica propozițională
- 35 Evaluarea valorii de adevăr
- 36 Logica cu predicate de ordinul întâi
- 37 LPOI – Semantică
- 38 Forme normale
- 39 Unificare și rezoluție

- formalism simbolic pentru reprezentarea faptelor și raționament.
- se bazează pe ideea de **valoare de adevăr** – e.g. *Adevărat* sau *Fals*.
- permite realizarea de argumente (argumentare) și demonstrații – deducție, inducție, rezoluție, etc.

# Logica propozițională

- Cadru pentru:
  - descrierea proprietăților obiectelor, prin intermediul unui limbaj, cu o semantică asociată;
  - deducerea de noi proprietăți, pe baza celor existente.
- Expresia din limbaj: propoziția, corespunzătoare unei afirmații, ce poate fi adevărată sau falsă.
- Exemplu: “Afară este frumos.”
- Accepții asupra unei propoziții:
  - secvența de simboluri utilizate sau
  - înțelesul propriu-zis al acesteia, într-o interpretare.



- 2 categorii de propoziții
  - simple  $\rightarrow$  fapte **atomice**: “Afară este frumos.”
  - compuse  $\rightarrow$  **relații** între propoziții mai simple: “Telefonul sună și câinele latră.”
- Propoziții simple:  $p, q, r, \dots$
- Negații:  $\neg \alpha$
- Conjuncții:  $(\alpha \wedge \beta)$
- Disjuncții:  $(\alpha \vee \beta)$
- Implicații:  $(\alpha \Rightarrow \beta)$
- Echivalențe:  $(\alpha \Leftrightarrow \beta)$

- Scop: dezvoltarea unor mecanisme de prelucrare, aplicabile **independent** de valoarea de adevăr a propozițiilor într-o situație particulară.
- Accent pe **relațiile** între propozițiile compuse și cele constituente.
- Pentru explicitarea propozițiilor → utilizarea conceptului de **interpretare**.

+ **Interpretare** Mulțime de **asocieri** între fiecare propoziție **simplă** din limbaj și o valoare de adevăr.



Exemplu

Interpretarea  $I$ :

- $p^I = false$
- $q^I = true$
- $r^I = false$

Interpretarea  $J$ :

- $p^J = true$
- $q^J = true$
- $r^J = true$

- cum știu dacă  $p$  este adevărat sau fals? Pot ști dacă știu **interpretarea** –  $p$  este doar un *nume* pe care îl dau unei propoziții concrete.

- Sub o interpretare **fixată**  $\rightarrow$  **dependența** valorii de adevăr a unei propoziții compuse de valorile de adevăr ale celor constituente
- **Negație**:  $(\neg \alpha)^I = \begin{cases} true & \text{dacă } \alpha^I = false \\ false & \text{altfel} \end{cases}$
- **Conjunție**:  $(\alpha \wedge \beta)^I = \begin{cases} true & \text{dacă } \alpha^I = true \text{ și } \beta^I = true \\ false & \text{altfel} \end{cases}$
- **Disjuncție**:  $(\alpha \vee \beta)^I = \begin{cases} false & \text{dacă } \alpha^I = false \text{ și } \beta^I = false \\ true & \text{altfel} \end{cases}$

- **Implicație:**  $(\alpha \Rightarrow \beta)^I = \begin{cases} false & \text{dacă } \alpha^I = true \text{ și } \beta^I = false \\ true & \text{altfel} \end{cases}$

- **Echivalență:**

$$(\alpha \Leftrightarrow \beta)^I = \begin{cases} true & \text{dacă } \alpha \Rightarrow \beta \wedge \beta \Rightarrow \alpha \\ false & \text{altfel} \end{cases}$$

# Evaluarea valorii de adevăr

+ **Evaluare** Determinarea **valorii de adevăr** a unei **propoziții**, sub o **interpretare**, prin aplicarea regulilor semantice anterioare.



Exemplu

- Interpretarea  $I$ :
  - $p^I = false$
  - $q^I = true$
  - $r^I = false$
- Propoziția:  $\phi = (p \wedge q) \vee (q \Rightarrow r)$   
 $\phi^I = (false \wedge true) \vee (true \Rightarrow false) = false \vee false = false$

+ **Satisfiabilitate** Proprietatea unei propoziții care este adevărată sub cel puțin o interpretare. Acea interpretare **satisface** propoziția.

+ **Validitate** Proprietatea unei propoziții care este adevărată în toate interpretările. Propoziția se mai numește **tautologie**.

Ex Exemplu Propoziția  $p \vee \neg p$  este **validă**.

+ **Nesatisfiabilitate** Proprietatea unei propoziții care este falsă în toate interpretările. Propoziția se mai numește **contradicție**.

Ex Exemplu Propoziția  $p \wedge \neg p$  este **nesatisfiabilă**.



### Ex) Metoda tabelului de adevăr

| $p$          | $q$          | $r$          | $(p \wedge q) \vee (q \Rightarrow r)$ |
|--------------|--------------|--------------|---------------------------------------|
| <i>true</i>  | <i>true</i>  | <i>true</i>  | <i>true</i>                           |
| <i>true</i>  | <i>true</i>  | <i>false</i> | <i>true</i>                           |
| <i>true</i>  | <i>false</i> | <i>true</i>  | <i>true</i>                           |
| <i>true</i>  | <i>false</i> | <i>false</i> | <i>true</i>                           |
| <i>false</i> | <i>true</i>  | <i>true</i>  | <i>true</i>                           |
| <i>false</i> | <i>true</i>  | <i>false</i> | <i>false</i>                          |
| <i>false</i> | <i>false</i> | <i>true</i>  | <i>false</i>                          |
| <i>false</i> | <i>false</i> | <i>false</i> | <i>false</i>                          |

$\Rightarrow$  Propoziția  $(p \wedge q) \vee (q \Rightarrow r)$  este **satisfiabilă**.

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



Exemplu

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

- Verificabilă prin metoda tabelii de adevăr: **toate** intrările pentru care **premisele** sunt adevărate trebuie să inducă adevărul **concluziei**.

Demonstrăm că  $\{p, p \Rightarrow q\} \models q$ .



Exemplu

| $p$          | $q$          | $p \Rightarrow q$ |
|--------------|--------------|-------------------|
| <i>true</i>  | <i>true</i>  | <i>true</i>       |
| <i>true</i>  | <i>false</i> | <i>false</i>      |
| <i>false</i> | <i>true</i>  | <i>true</i>       |
| <i>false</i> | <i>false</i> | <i>true</i>       |

Singura intrare în care ambele premise,  $p$  și  $p \Rightarrow q$ , sunt adevărate, precizează și adevărul concluziei,  $q$ .

- $\{\phi_1, \dots, \phi_n\} \models \phi$

*sau*

- Propoziția  $\phi_1 \wedge \dots \wedge \phi_n \Rightarrow \phi$  este **validă**

*sau*

- Propoziția  $\phi_1 \wedge \dots \wedge \phi_n \wedge \neg \phi$  este **nesatisfiabilă**

- Creșterea **exponențială** a numărului de interpretări în raport cu numărul de propoziții simple.
- De aici, **diminuarea** valorii practice a metodelor **semantice**, precum cea a tabelii de adevăr.
- Alternativ, metode **sintactice**, care manipulează doar reprezentarea simbolică.
  - Inferență  $\rightarrow$  Derivare **mecanică**  $\rightarrow$  demers de **calcul**, în scopul verificării derivabilității logice.
  - folosind **metodele de inferență**, putem construi o **mașină de calcul**.

+ **Inferența** – Derivarea **mecanică** a concluziilor unui set de premise.

+ **Regulă de inferență** – **Procedură** de calcul capabilă să deriveze concluziile unui set de premise. Derivabilitatea mecanică a concluziei  $\phi$  din mulțimea de premise  $\Delta$ , utilizând **regula de inferență** *inf*, se notează  $\Delta \vdash_{inf} \phi$ .

Ex | **Modus Ponens (MP)** :

$$\frac{\alpha \Rightarrow \beta \quad \alpha}{\beta}$$

Ex | **Modus Tollens** :

$$\frac{\alpha \Rightarrow \beta \quad \neg \beta}{\neg \alpha}$$

+ **Consistență (soundness)** – Regula de inferență determină **numai** propoziții care sunt, într-adevăr, **consecințe logice** ale premiselor.

$$\Delta \vdash_{inf} \phi \Rightarrow \Delta \models \phi.$$

+ **Completitudine (completeness)** – Regula de inferență determină **toate consecințele logice** ale premiselor.  $\Delta \models \phi \Rightarrow \Delta \vdash_{inf} \phi.$

- Ideal, **ambele** proprietăți – “nici în plus, nici în minus” –  $\Delta \models \phi \Leftrightarrow \Delta \vdash_{inf} \phi$
- **Incompletitudinea** regulii *Modus Ponens*, din imposibilitatea scrierii oricărei propoziții ca implicație.

# Logica cu predicate de ordinul întâi



- **Extensie** a logicii propoziționale, cu explicitarea:
  - **obiectelor** din universul problemei;
  - **relațiilor** dintre acestea.
- 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:  $prieten(x, y)$ : “ $x$  este prieten cu  $y$ .”
  - $\forall x. \forall y. (prien(x, y) \Leftrightarrow prieten(y, x))$
  - Aplicare pe cazuri **particulare**.
  - **Transparență** în raport cu obiectele și relațiile referite.

- + **Constante** – obiecte particulare din universul discursului:  $c, d, andrei, bogdan, \dots$
- + **Variabile** – obiecte generice:  $x, y, \dots$
- + **Simboluri funcționale** –  $succesor, +, abs \dots$
- + **Simboluri relaționale** (**predicate**) – relații  $n$ -are peste obiectele din universul discursului:  $prieten = \{(andrei, bogdan), (bogdan, andrei), \dots\}$ ,  $impar = \{1, 3, \dots\}, \dots$
- + **Conectori logici**  $\neg, \wedge, \vee, \Rightarrow, \Leftarrow$
- + **Cuantificatori**  $\forall, \exists$

### + Termeni (obiecte):

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

### Ex) Exemple

- $\text{succesor}(4)$ : succesul lui 4, și anume 5.
- $+(2, x)$ : aplicația funcției de adunare asupra numerelor 2 și  $x$ , și, totodată, suma lor.

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

### Ex Exemple

- $impar(3)$
- $varsta(ion, 20)$
- $= (+ (2, 3), 5)$

+ **Propoziții** (fapte) – dacă  $x$  variabilă,  $A$  atom, și  $\alpha$  și  $\beta$  propoziții, atunci o propoziție are forma:

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

“Sora Ioanei are un prieten deștept”

$$\underbrace{\underbrace{\underbrace{\exists X}_{\text{termen}} \cdot \underbrace{\text{prieten}(X, \underbrace{\text{sora}(\text{ioana}))}_{\text{termen}}})_{\text{atom/propoziție}} \wedge \underbrace{\text{deștept}(X)}_{\text{atom/propoziție}}}_{\text{propoziție}}$$



Exemplu

# LPOI – Semantică

+ **Interpretarea** constă din:

- Un **domeniu** nevid,  $D$ , de concepte (obiecte)
- Pentru fiecare **constantă**  $c$ , un element  $c^I \in D$
- Pentru fiecare simbol **funcțional**,  $n$ -ar  $f$ , o funcție  $f^I : D^n \rightarrow D$
- Pentru fiecare **predicat**  $n$ -ar  $p$ , o funcție  $p^I : D^n \rightarrow \{false, true\}$ .



- Atom:

$$(p(t_1, \dots, t_n))^I = p^I(t_1^I, \dots, t_n^I)$$

- Negație, conectori, implicații: v. logica propozițională

- Cuantificare **universală**:

$$(\forall x. \alpha)^I = \begin{cases} false & \text{dacă } \exists d \in D. \alpha^I_{[d/x]} = false \\ true & \text{altfel} \end{cases}$$

- Cuantificare **existențială**:

$$(\exists x. \alpha)^I = \begin{cases} true & \text{dacă } \exists d \in D. \alpha^I_{[d/x]} = true \\ false & \text{altfel} \end{cases}$$

**Ex** Exemple cu cuantificatori

- 1 “Vrăbia mălai visează.”  $\forall x.(vrăbie(x) \Rightarrow viseaza(x, malai))$
- 2 “Unele vrăbii visează mălai.”  $\exists x.(vrăbie(x) \wedge viseaza(x, malai))$
- 3 “Nu toate vrăbiile visează mălai.”  $\exists x.(vrăbie(x) \wedge \neg viseaza(x, malai))$
- 4 “Nicio vrăbie nu visează mălai.”  $\forall x.(vrăbie(x) \Rightarrow \neg viseaza(x, malai))$
- 5 “Numai vrăbiile visează mălai.”  $\forall x.(viseaza(x, malai) \Rightarrow vrăbie(x))$

- $\forall x.(vrabie(x) \Rightarrow viseaza(x, malai))$   
→ corect: “Toate vrăbiile visează mălai.”
- $\forall x.(vrabie(x) \wedge viseaza(x, malai))$   
→ **greșit**: “Toți sunt vrăbii și toți visează mălai.”
- $\exists x.(vrabie(x) \wedge viseaza(x, malai))$   
→ corect: “Unele vrăbii visează mălai.”
- $\exists x.(vrabie(x) \Rightarrow viseaza(x, malai))$   
→ **greșit**: probabil nu are semnificația pe care o intenționăm. Este adevărată și dacă luăm un  $x$  care nu este vrabie (fals implică orice).

- **Necomutativitate:**

- $\forall x. \exists y. \text{viseaza}(x, y) \rightarrow$  “Toți visează la ceva anume.”
- $\exists x. \forall y. \text{viseaza}(x, y) \rightarrow$  “Există cineva care visează la orice.”

- **Dualitate:**

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

- Satisfiabilitate.
- Validitate.
- Derivabilitate.
- Inferență.

# Forme normale

## Definiții

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

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

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

Ex 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

$$\{\neg A_1, \dots, \neg A_m, B_1, \dots, B_n\}, \Leftrightarrow (A_1 \wedge \dots \wedge A_m) \Rightarrow (B_1 \vee \dots \vee B_n)$$

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

**Ex** Exemplu Transformarea propoziției

$\forall x. vrabie(x) \vee ciocarlie(x) \Rightarrow pasare(x)$  în formă normală, utilizând clauze Horn:

FNC:  $\{\neg vrabie(x), pasare(x)\}, \{\neg ciocarlie(x), pasare(x)\}$



- 1 Eliminarea **implicațiilor** ( $\Rightarrow$ )
- 2 Împingerea **negațiilor** până în fața atomilor ( $\neg$ )
- 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:

$$\begin{aligned} &\forall x.\forall y.\exists z.((p(x) \wedge q(y)) \vee r(z)) \\ &\rightarrow \forall x.\forall y.((p(x) \wedge q(y)) \vee r(f_z(x, y))) \end{aligned}$$

- 6 Eliminarea cuantificatorilor **universali**, considerați, acum, implicați ( $\rightarrow$ ):

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

- 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).

**Ex** Exemplu “Cine rezolvă toate laboratoarele este apreciat de cineva.”

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

$$\not\equiv \forall x.(\neg \forall y.(\neg lab(y) \vee rezolva(x, y)) \vee \exists y.apreciaza(y, x))$$

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

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

$$R \quad \forall x.(\exists y.(lab(y) \wedge \neg rezolva(x, y)) \vee \exists z.apreciaza(z, x))$$

$$P \quad \forall x.\exists y.\exists z.((lab(y) \wedge \neg rezolva(x, y)) \vee apreciaza(z, x))$$

$$S \quad \forall x.((lab(f_y(x)) \wedge \neg rezolva(x, f_y(x))) \vee apreciaza(f_z(x), x))$$

$$\not\equiv (lab(f_y(x)) \wedge \neg rezolva(x, f_y(x))) \vee apreciaza(f_z(x), x)$$

$$\vee/\wedge \quad (lab(f_y(x)) \vee apr(f_z(x), x)) \wedge (\neg rez(x, f_y(x)) \vee apr(f_z(x), x))$$

$$C \quad \{lab(f_y(x)), apr(f_z(x), x)\}, \{\neg rez(x, f_y(x)), apr(f_z(x), x)\}$$

# Unificare și rezoluție

- **Pasul de rezoluție**: regulă de inferență foarte puternică.
- Baza unui demonstrator de teoreme **consistent și complet**.
- Spațiul de căutare mai mic decât în alte sisteme.
- Se bazează pe lucrul cu propoziții în **forma clauzală** (clauze):
  - propoziție = mulțime de **clauze** (semnificație conjunctivă)
  - clauză = mulțime de **literali** (semnificație disjunctivă)
  - literal = **atom** sau **atom negat**
  - atom = **propoziție simplă**



**Ideea** (în LP):

$$\frac{\begin{array}{l} \{p \Rightarrow q\} \\ \{\neg p \Rightarrow r\} \end{array}}{\{q, r\}} \rightarrow \text{"Anularea" lui } p$$

- $p$  falsă  $\rightarrow \neg p$  adevărată  $\rightarrow r$  adevărată
- $p$  adevărată  $\rightarrow q$  adevărată
- $p \vee \neg p \Rightarrow$  Cel puțin una dintre  $q$  și  $r$  adevărată ( $q \vee r$ )
- Forma generală a pasului de rezoluție:

$$\frac{\begin{array}{l} \{p_1, \dots, r, \dots, p_m\} \\ \{q_1, \dots, \neg r, \dots, q_n\} \end{array}}{\{p_1, \dots, p_m, q_1, \dots, q_n\}}$$

- Clauza **vidă**  $\rightarrow$  indicator de **contradicție** între premise

$$\frac{\begin{array}{c} \{\neg p\} \\ \{p\} \end{array}}{\{\} = \emptyset}$$

- **Mai mult de 2** rezolvenți posibili  $\rightarrow$  se alege doar unul:

$$\frac{\begin{array}{c} \{p, q\} \\ \{\neg p, \neg q\} \end{array}}{\begin{array}{c} \{p, \neg p\} \text{ sau} \\ \{q, \neg q\} \end{array}}$$



- Demonstrarea **nesatisfiabilității**  $\rightarrow$  derivarea clauzei **vide**.
- Demonstrarea **derivabilității** concluziei  $\phi$  din premisele  $\phi_1, \dots, \phi_n \rightarrow$  demonstrarea **nesatisfiabilității** propoziției  $\phi_1 \wedge \dots \wedge \phi_n \wedge \neg \phi$ .
- Demonstrarea **validității** propoziției  $\phi \rightarrow$  demonstrarea **nesatisfiabilității** propoziției  $\neg \phi$ .

Demonstrăm că  $\{p \Rightarrow q, q \Rightarrow r\} \vdash p \Rightarrow r$ ,  
i.e. mulțimea  $\{p \Rightarrow q, q \Rightarrow r, \neg(p \Rightarrow r)\}$  conține o **contradicție**.



Exemplu

1.  $\{\neg p, q\}$  Premisă
2.  $\{\neg q, r\}$  Premisă
3.  $\{p\}$  Concluzie negată
4.  $\{\neg r\}$  Concluzie negată
5.  $\{q\}$  Rezoluție 1, 3
6.  $\{r\}$  Rezoluție 2, 5
7.  $\{\}$  Rezoluție 4, 6  $\rightarrow$  clauza vidă

**T | Teorema Rezoluției:** Rezoluția propozițională este **consistentă și completă**, i.e.  $\Delta \models \phi \Leftrightarrow \Delta \vdash_{rez} \phi$ .

- **Terminare garantată** a procedurii de aplicare a rezoluției: număr **finit** de clauze  $\rightarrow$  număr **finit** de concluzii.

- Utilizată pentru rezoluția în LPOI
- vezi și sinteza de tip în Haskell



cum știm dacă folosind ipoteza  $om(Marcel)$  și propoziția  $\forall x.om(x) \Rightarrow are\_inima(x)$  putem demonstra că  $are\_inima(Marcel) \rightarrow$  unificând  $om(Marcel)$  și  $\forall om(x)$ .

- reguli:
  - o propoziție unifică cu o propoziție de aceeași formă
  - două predicate unifică dacă au același nume și parametri care unifică ( $om$  cu  $om$ ,  $x$  cu  $Marcel$ )
  - o constantă unifică cu o constantă cu același nume
  - o variabilă unifică cu un termen ce nu conține variabila ( $x$  cu  $Marcel$ )

- Problemă **NP-completă**;
- Posibile legări **ciclice**;
- Exemplu:  
 $prieten(x, coleg\_banca(x))$  și  
 $prieten(coleg\_banca(y), y)$   
MGU:  $S = \{x \leftarrow coleg\_banca(y), y \leftarrow coleg\_banca(x)\}$   
 $\Rightarrow x \leftarrow coleg\_banca(coleg\_banca(x)) \rightarrow$  **imposibil!**
- Soluție: verificarea apariției unei variabile în **valoarea** la care a fost legată (*occurrence check*);

- Rezoluția pentru clauze **Horn**:

$$A_1 \wedge \dots \wedge A_m \Rightarrow A$$

$$B_1 \wedge \dots \wedge A' \wedge \dots \wedge B_n \Rightarrow B$$

$$\text{unificare}(A, A') = S$$

$$\hline \text{subst}(S, A_1 \wedge \dots \wedge A_m \wedge B_1 \wedge \dots \wedge B_n \Rightarrow B)$$

- $\text{unificare}(\alpha, \beta) \rightarrow$  substituția sub care unifică propozițiile  $\alpha$  și  $\beta$ ;
- $\text{subst}(S, \alpha) \rightarrow$  propoziția rezultată în urma aplicării substituției  $S$  asupra propoziției  $\alpha$ .



Exemplu

### Horses and hounds

- 1 Horses are faster than dogs.
- 2 There is a greyhound that is faster than any rabbit.
- 3 Harry is a horse and Ralph is a rabbit.
- 4 Is Harry faster than Ralph?

## Exemplu Horses and Hounds

- 1  $\forall x. \forall y. \text{horse}(x) \wedge \text{dog}(y) \Rightarrow \text{faster}(x, y) \rightarrow \neg \text{horse}(x) \vee \neg \text{dog}(y) \vee \text{faster}(x, y)$
- 2  $\exists x. \text{greyhound}(x) \wedge (\forall y. \text{rabbit}(y) \Rightarrow \text{faster}(x, y))$   
 $\rightarrow \text{greyhound}(\text{Greg}) \quad ; \quad \neg \text{rabbit}(y) \vee \text{faster}(\text{Greg}, y)$
- 3  $\text{horse}(\text{Harry}) \quad ; \quad \text{rabbit}(\text{Ralph})$
- 4  $\neg \text{faster}(\text{Harry}, \text{Ralph})$  (concluzia negată)
- 5  $\neg \text{greyhound}(x) \vee \text{dog}(x)$  (common knowledge)
- 6  $\neg \text{faster}(x, y) \vee \neg \text{faster}(y, z) \vee \text{faster}(x, z)$  (tranzitivitate)
- 7  $1 + 3a \rightarrow \neg \text{dog}(y) \vee \text{faster}(\text{Harry}, y)$  (cu  $\{\text{Harry}/x\}$ )
- 8  $2a + 5 \rightarrow \text{dog}(\text{Greg})$  (cu  $\{\text{Greg}/x\}$ )
- 9  $7 + 8 \rightarrow \text{faster}(\text{Harry}, \text{Greg})$  (cu  $\{\text{Greg}/y\}$ )
- 10  $2b + 3b \rightarrow \text{faster}(\text{Greg}, \text{Ralph})$  (cu  $\{\text{Ralph}/y\}$ )
- 11  $6 + 9 + 10 \rightarrow \text{faster}(\text{Harry}, \text{Ralph}) \quad \{\text{Harry}/x, \text{Greg}/y, \text{Ralph}/z\}$
- 12  $11 + 4 \rightarrow \square$  q.e.d.



- sintaxa și semantica în LPOI
- Forme normale, Unificare, Rezoluție în LPOI

- 40 Caracteristici ale paradigmelor de programare
- 41 Variabile și valori de prim rang
- 42 Tipare a variabilelor
- 43 Legarea variabilelor
- 44 Modul de evaluare

# Caracteristici ale paradigmelor de programare

- **Paradigma de programare** – un mod de a:
  - aborda rezolvarea unei probleme printr-un program;
  - structura un program;
  - reprezenta datele dintr-un program;
  - implementa diversele aspecte dintr-un program (**cum** prelucrăm datele);
- Un limbaj poate include caracteristici dintr-una sau mai multe paradigme;
  - în general există o paradigmă dominantă;
- **Atenție!** Paradigma nu are legătură cu sintaxa limbajului!

- paradigmele sunt legate de o **mașină de calcul** teoretică în care prelucrările caracteristice paradigmei se fac la nivelul mașinii;
- **dar** putem executa orice program, scris în orice paradigmă, pe orice mașină (după o eventuală translație).

- În principal, paradigma este definită de
  - elementele principale din sintaxa limbajului – e.g. existența și semnificația **variabilelor**, semnificația **operatorilor** asupra datelor, modul de construire a programului;
  - modul de construire al **tipurilor** variabilelor;
  - modul de definire și statutul **operatorilor** – elementele principale de prelucrare a datelor din program (e.g. obiecte, funcții, predicate);
  - **legarea** variabilelor, efecte laterale, transparentă referențială, modul de transfer al parametrilor pentru elementele de prelucrare a datelor.

# Variable și valori de prim rang

- în majoritatea limbajelor există variabile, ca **NUME** date unor valori – rezultatul anumitor procesări (calcul, inferențe, substituții);
- variabilele pot fi o **referință** pentru un spațiu de memorie sau pentru un rezultat abstract;
- elementele de procesare a datelor pot sau nu să fie **valori de prim rang** (să poată fi asociate cu variabile).



+ **Valoare de prim rang** – O valoare care poate fi:

- creată dinamic
- stocată într-o variabilă
- trimisă ca parametru unei funcții
- întoarsă dintr-o funcție

 Să se scrie funcția `compose`, ce primește ca parametri alte 2 funcții, `f` și `g`, și întoarce funcția obținută prin compunerea lor,  $f \circ g$ .

```
1 int compose(int (*f)(int), int (*g)(int), int x) {  
2     return (*f)((*g)(x));  
3 }
```

- în C, funcțiile **nu** sunt valori de prim rang;
- pot scrie o funcție care compune două funcții pe o anumită valoare (ca mai sus)
- pot întoarce pointer la o funcție existentă
- dar nu pot crea o referință (pointer) la o funcție **nouă**, care să fie folosit apoi ca o funcție obișnuită

```
1  abstract class Func<U, V> {
2      public abstract V apply(U u);
3
4      public <T> Func<T, V> compose(final Func<T, U> f) {
5          final Func<U, V> outer = this;
6
7          return new Func<T, V>() {
8              public V apply(T t) {
9                  return outer.apply(f.apply(t));
10             }
11         };
12     }
13 }
```

- În Java, funcțiile **nu** sunt valori de prim rang – pot crea rezultatul dar este complicat, și rezultatul nu este o funcție obișnuită, ci un obiect.

- Racket:

```
1 (define compose
2   (lambda (f g)
3     (lambda (x)
4       (f (g x))))))
```

- Haskell:

```
1 compose = (.)
```

- În Racket și Haskell, funcțiile **sunt** valori de prim rang.
- mai mult, ele pot fi **aplicate parțial**, și putem avea **funcționale** – funcții care iau alte funcții ca parametru.

# Tipare a variabilelor

· Rolul tipurilor: exprimare a intenției programatorului, abstractizare, documentare, optimizare, verificare

+ **Tipare** – modul de gestionare a tipurilor.

⋮ Clasificare după **momentul** verificării:

- statică
- dinamică

⋮ Clasificare după **rigiditatea** regulilor:

- tare
- slabă

## Exemplu

### Ex Tipare dinamică

Exemplu

Javascript:

```
var x = 5;  
if(condition) x = "here";  
print(x); → ce tip are x aici?
```

### Ex Tipare statică

Exemplu

Java:

```
int x = 5;  
if(condition)  
    x = "here"; → Eroare la compilare: x este int.  
print(x);
```

### ⋮ Tipare statică

- La compilare
- Valori și variabile
- Rulare mai rapidă
- Rigidă: sancționează orice construcție
- Debugging mai facil
- Declarații explicite sau inferențe de tip
- Pascal, C, C++, Java, Haskell

### ⋮ Tipare dinamică

- La rulare
- Doar valori
- Rulare mai lentă (necesită verificarea tipurilor)
- Flexibilă: sancționează doar când este necesar
- Debugging mai dificil
- Permite metaprogramare (v. `eval`)
- Python, Scheme/Racket, Prolog, JavaScript, PHP



- Clasificare după **libertatea** de a agrega valori de tipuri **diferite**.

### Ex Tipare tare

Exemplu  $1 + "23" \rightarrow$  **Eroare** (Haskell, Python)

### Ex Tipare slabă

Exemplu  $1 + "23" = 24$  (Visual Basic)  
 $1 + "23" = "123"$  (JavaScript)

# Legarea variabilelor

· două posibilități esențiale:

- un nume este întotdeauna legat (într-un anumit context) la aceeași valoare / la același calcul  $\Rightarrow$  numele **stă pentru un calcul**;
  - legare **statică**.
  - în Prolog, putem utiliza variabila și înainte de a fi legată (în anumite condiții).
- un nume (aceeași variabilă) poate fi legat la mai multe valori pe parcursul execuției  $\Rightarrow$  numele **stă pentru un spațiu de stocare** – fiecare element de stocare fiind identificat printr-un nume;
  - legare **dinamică**.

 **Exemplu** În expresia  $2 + (i = 3)$ , subexpresia  $(i = 3)$ :

- produce **valoarea** 3, conducând la rezultatul 5 al întregii expresii;
- are **efectul lateral** de inițializare a lui  $i$  cu 3.

**+** **Efect lateral** Pe lângă valoarea pe care o produce, o expresie sau o funcție poate **modifica** starea globală.

- Inerente în situațiile în care programul interacționează cu exteriorul → **I/O!**



În expresia  $x-- + ++x$ , cu  $x = 0$ :

- evaluarea stânga  $\rightarrow$  dreapta produce  $0 + 0 = 0$
- evaluarea dreapta  $\rightarrow$  stânga produce  $1 + 1 = 2$
- dacă înlocuim cele două subexpresii cu valorile pe care le reprezintă, obținem
$$x + (x + 1) = 0 + 1 = 1$$
- Importanța **ordinii de evaluare**!
- Dependențe **implicite**, puțin lizibile și posibile generatoare de bug-uri.

- În prezența efectelor laterale, programarea leneșă devine foarte dificilă;
- Efectele laterale pot fi gestionate corect numai atunci când **secvența** evaluării este garantată → garanție inexistentă în programarea leneșă.
  - nu știm când anume va fi **nevoie** de valoarea unei expresii.

+ | **Transparență referențială** Confundarea unui obiect (“valoare”) cu referința la acesta.

+ | **Expresie transparentă referențială**: posedă o unică valoare, cu care poate fi substituită, **păstrând** semnificația programului.

## Ex | Exemplu

- $x-- + ++x \rightarrow$  **nu**, valoarea depinde de ordinea de evaluare
- $x = x + 1 \rightarrow$  **nu**, două evaluări consecutive vor produce rezultate diferite
- $x \rightarrow$  ar putea fi, în funcție de statutul lui  $x$  (globală, statică etc.)

**+** **Funcție transparentă referențială:** rezultatul întors depinde **exclusiv** de parametri.

### Exemplu

```
int transparent(int x) {  
    return x + 1;  
}
```

```
int g = 0;
```

```
int opaque(int x) {  
    return x + ++g;  
}
```

- `opaque(3) - opaque(3) != 0!`
- **Funcții transparente:** `log`, `sin` etc.
- **Funcții opace:** `time`, `read` etc.



- **Lizibilitatea** codului;
- Demonstrarea formală a **corectitudinii** programului – mai ușoară datorită lipsei **stării**;
- **Optimizare** prin reordonarea instrucțiunilor de către compilator și prin caching;
- **Paralelizare** masivă, prin eliminarea modificărilor concurente.

# Modul de evaluare

- modul de evaluare al expresiilor dictează modul în care este executat programul;
- este legat de funcționarea **mașinii teoretice** corespunzătoare paradigmei;
- ne interesează în special ordinea în care expresiile se evaluează;
- în final, întregul program se evaluează la o valoare;
- important în modul de evaluare este modul de **evaluare / transfer a parametrilor**.

- Evaluare **aplicativă** – parametrii sunt evaluați înainte de evaluarea corpului funcției.
  - *Call by value*
  - *Call by sharing*
  - *Call by reference*
- Evaluare **normală** – funcția este evaluată fără ca parametrii să fie evaluați înainte.
  - *Call by name*
  - *Call by need*

### Exemplu

```
1 // C sau Java
2 void f(int x) {
3     x = 3;
4 }
```

```
1 // C
2 void g(struct str s) {
3     s.member = 3;
4 }
```

- Efectul liniilor 3 este **invizibil** la apelant.
  - Evaluarea parametrilor **înaintea** aplicației funcției și transferul unei **copii** a valorii acestuia
  - Modificări locale **invizibile** la apelant
  - C, C++, tipurile primitive Java

- Variantă a *call by value*;
- Trimiterea unei **referințe** la obiect;
- Modificări locale asupra **referinței** invizibile la apelant;
- Modificări locale asupra **obiectului** referit vizibile la apelant;
- Racket, Java;

- Trimiterea unei **referințe** la obiect;
- Modificări locale asupra referinței și obiectului referit **vizibile** la apelant;
- Folosirea “&” în C++.

- Argumente **neevaluate** în momentul aplicării funcției → substituție directă (textuală) în corpul funcției;
- Evaluare parametrilor la cerere, de **fiecare** dată când este nevoie de valoarea acestora;
- în calculul  $\lambda$ .



- Variantă a *call by name*;
- Evaluarea unui parametru doar la **prima** utilizare a acestuia;
- **Memorarea** valorii unui parametru deja evaluat și returnarea acesteia în cazul utilizării repetate a aceluiași parametru (datorită transparenței referențiale, o aceeași expresie are întotdeauna aceeași valoare) – **memoizare**;
- în Haskell.

- caracteristicile unei paradigme;
- variabile, funcții ca valori de prim rang;
- legare, efecte laterale, transparență referențială;
- evaluare și moduri de transfer al parametrilor.