# Paradigme de Programare

Ș.l. dr. ing. Andrei Olaru

Departamentul Calculatoare

2015 – 2016, semestrul 2



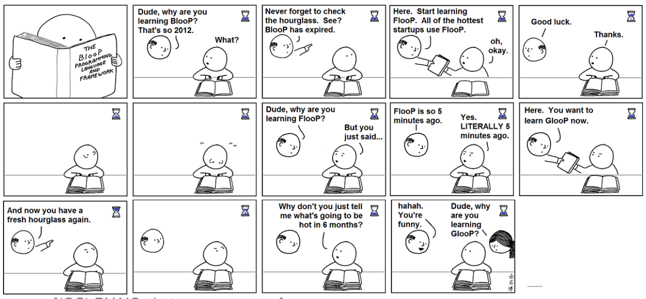
## Cursul 1: Introducere





## BlooP and FlooP and GlooP

[http://abstrusegoose.com/503]




[(CC) BY-NC abstrusegoose.com]



## Ce studiem la PP?



## Elemente pe care le vom studia

λPP

- Paradigma funcțională și paradigma logică, paradigme de tip declarativ, 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



## Exemplu



## Exemplu

λPP

**Ex** Exemplu

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



## Modelare funcţională (1)

λPP

Racket:

```
(define memList (lambda (e L)
    (if (null? L)
        #f
        (if (equal? (car L) e)
            #t
            (memList e (cdr L))
            ))
    ))
```

## Modelare funcțională (2)

Haskell

```
memList x [] = False
memList x (y:t) = x == y || memList x t
```



## Modelare logică

Prolog:

```
memberA(E, [E|_]) :- !.
memberA(E, [_|L]) :- memberA(E, L).
```



De ce studiem această materie?



## De ce?

*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



## De ce?
### Mai concret

· 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 funcțională și logică (și paradigma declarative în general) 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.)



## De ce?
### 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.
- În industrie sunt utilizate limbaje puternic funcționale precum Erlang, Scala, F#, Clojure.
- Limbaje multi-paradigmă → adaptarea paradigmei utilizate la necesități.



## Modele → paradigme → limbaje
### Modele de calculabilitate

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

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



## Ce înseamnă paradigma de programare
### Ce diferă între paradigme?

- diferă sintaxa ← aceasta este o diferență între limbaje
- diferă modul de construcție al expresiilor
- diferă structura programului ← omoiconicitate: structura programului este similară cu structura datelor și a expresiilor individuale

## Ce înseamnă paradigma de programare
### Ce caracterizează o paradigmă?

- modul de tipare al valorilor
- ordinea de evaluare (generare a valorilor)
- modul de legare al variabilelor (managementul valorilor)

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



## Ce vom studia?
### Conținutul cursului

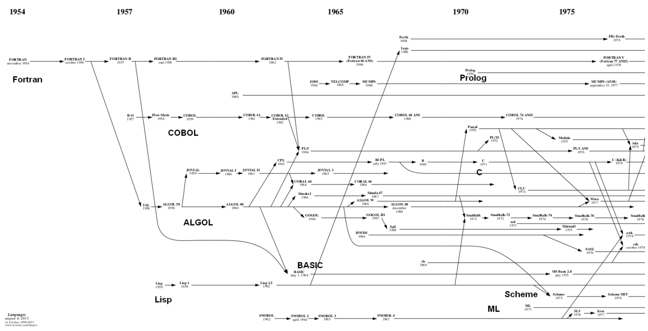
1. Diverse perspective conceptuale asupra noţiunii de calculabilitate efectivă → modele de calculabilitate.
2. Influenţa perspectivei alese asupra procesului de modelare şi rezolvare a problemlor → paradigme de programare.
3. Limbaje de programare aferente paradigmelor, cu accent pe aspectul comparativ.



## Istoric: Paradigme și limbaje de programare

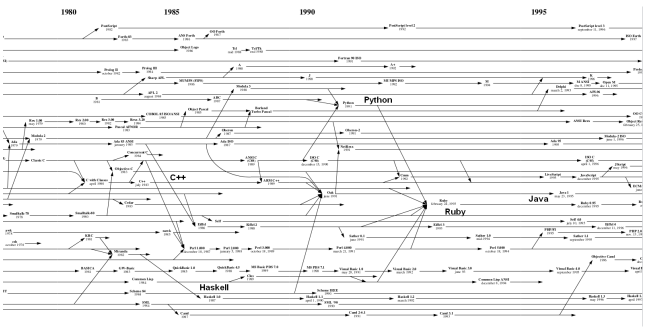


## Istorie
### 1950-1975



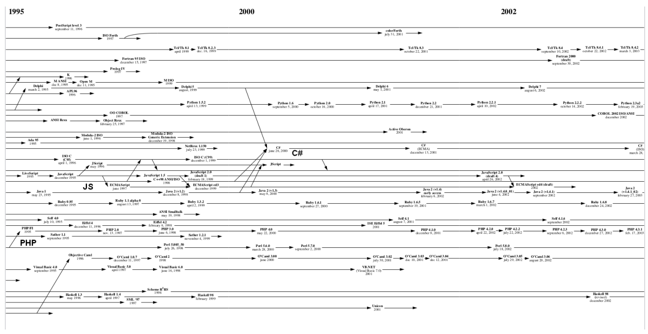



## Istorie
### 1975-1995



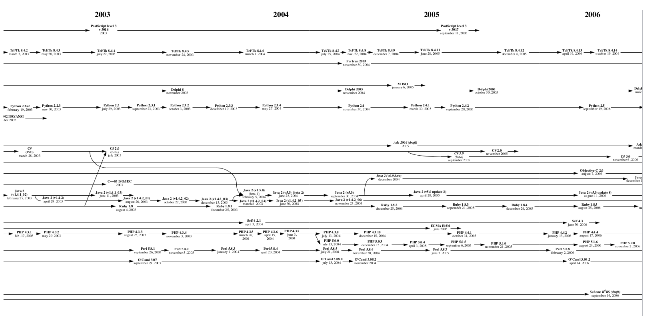



## Istorie
### 1995-2002



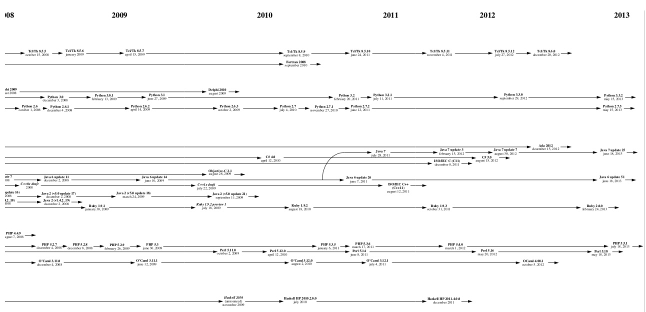



## Istorie
### 2002-2006

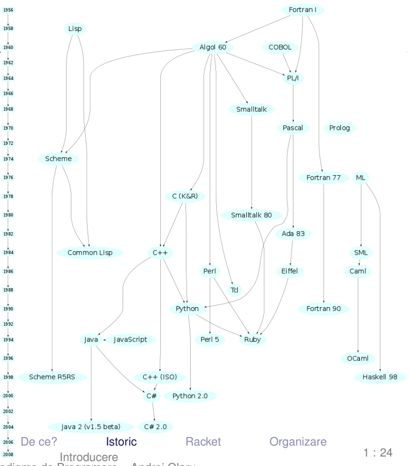





## Istorie
### 2006-2013



## Istorie
'56-'04 pe scurt






## Istorie
Resurse

λP.P

- imagine navigabilă (slides precedente):
  [http://www.levenez.com/lang/]
- poster (până în 2004):
  [http://oreilly.com/pub/a/oreilly/news/languageposter_0504.html]
- arbore din slide precedent și arbore extins:
  [http://rigaux.org/language-study/diagram.html]
- Wikipedia:
  [http://en.wikipedia.org/wiki/Generational_list_of_programming_languages]

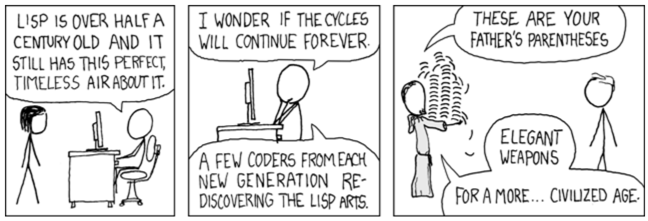


# Introducere în Racket



## Lisp cycles
[http://xkcd.com/297/]

λP.P




[(CC) BY-NC xkcd.com]



## Racket
*din 1975*

λP.P

- 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



# Organizare



## Unde găsesc informații?
Resurse de bază

λP.P

`http://elf.cs.pub.ro/pp/`

Regulament: `http://elf.cs.pub.ro/pp/regulament`

Teme și forumuri: curs.cs → L-2-PP-CA-CC
`http://cs.curs.pub.ro/2015/course/view.php?id=81`

Elementele cursului sunt comune la seriile CA și CC.



## Notare
mai multe la `http://elf.cs.pub.ro/pp/regulament`

λP.P

- Laborator: 1p ← cu bonusuri, dar maxim 1p total (cu extensie până la 1.5 pentru performanță susținută)
- Teme: 4p (3 × 1.33p) ← cu bonusuri, dar în limita a maxim 6p pe parcurs
- Teste la curs: 0.5p ← punctare pe parcurs, la curs, din cursul anterior
- Test din materia de laborator: 0.5p ← test grilă, de cunoaștere a limbajelor
- Examen: 4p ← limbaje + teorie

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

(

[http://xkcd.com/859/]


[(CC) BY-NC xkcd.com]

+ Dați feedback la acest curs aici:
[http://goo.gl/forms/SjDsW06v5J]



## Cursul 2: Programare funcţională în Racket

7. Introducere
8. Discuție despre tipare
9. Legarea variabilelor
10. Evaluare
11. Construcția programelor prin recursivitate



# Introducere



## Racket vs. Scheme

Cum se numește limbajul despre care discutăm?

- Racket este dialect de Lisp/Scheme (așa cum Scheme este dialect de Lisp);
- la nivelul studiat, Racket este identic cu Scheme;
- Racket este derivat din Scheme, oferind instrumente mai puternice;
- Racket (fost PLT Scheme) este interpretat de mediul DrRacket (fost DrScheme);

[http://en.wikipedia.org/wiki/Racket_(programming_language)]
[http://racket-lang.org/new-name.html]



# Discuție despre tipare



## Tipuri în Racket

În Racket avem:

- numere: `1`, `2`, `1.5`
- simboli (literali): `'abcd`, `'andrei`
- valori booleene: `#t`, `#f`
- șiruri de caractere: `"șir de caractere"`
- perechi: `(cons 1 2)` → `'(1 . 2)`
- liste: `(cons 1 (cons 2 '()))` → `'(1 2)`
- funcții: `(λ (e f) (cons e f))` → `#<procedure>`

· Cum sunt gestionate tipurilor valorilor (variabilelor) la compilare (verificare) și la execuție?



## Modalităţi de tipare

· 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



## Tipare statică vs. dinamică

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";
print(x);
```
→ Eroare la *compilare*: `x` este `int`.

## Tipare statică vs. dinamică
### Caracteristici

**Tipare statică**
- La compilare
- Valori şi variabile
- Rulare mai rapidă
- Rigidă: sancţionează orice construcţie
- Debugging mai facil
- Declaraţii explicite sau inferenţe de tip
- Pascal, C, C++, Java, Haskell

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



## Tipare tare vs. slabă
### Exemple

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

**Exemplu – Tipare tare**

`1 + "23"` → Eroare (Haskell)

**Exemplu – Tipare slabă**

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



## Tiparea în Racket

- este dinamică

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

- este tare

```
(+ "1" 2) → Eroare
```

- dar, permite liste cu elemente de tipuri diferite.



## Legarea variabilelor



## Variabile (Nume)
### Proprietăţi generale ale variabilelor

**Proprietăţi**
- identificator
- valoarea legată (la un anumit moment)
- domeniul de vizibilitate (*scope*) + durata de viaţă
- tip

**Stări**
- declarată: cunoaştem identificatorul
- definită: cunoaştem şi valoarea → variabila a fost *legată*

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



## Legarea variabilelor
### Definiţii (1)

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

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

- Este determinat de modalitatea de legare a variabilelor.



## Legarea variabilelor
### Definiţii (2)

**+ Legare statică** – Variabilele din corpul unei expresii sunt extrase din contextul în care aceasta a fost definită.

- Domeniu de vizibilitate determinat prin construcţiile limbajului, putând fi desprins la compilare.

**+ Legare dinamică** – Valorile variabilelor depind de momentul în care o expresie este evaluată.

- Domeniu de vizibilitate (al unei legări) determinat la execuţie.



## Legarea variabilelor în Racket

- Variabile declarate (! și definite) în construcţii interioare → legate static
  - `lambda`
  - `let`
  - `let*`
  - `letrec`
- Variabile *top-level* → legate dinamic
  - `define`

## Construcţia `lambda`
### Definiţie & Exemplu

- Leagă static parametrii formali ai unei funcţii
- Sintaxă:

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

- Domeniul de vizibilitate al parametrului `pk`: mulţimea punctelor din `expr` (care este corpul funcţiei), puncte în care apariţia lui `pk` este liberă.



## Construcţia `lambda`
### Semantică

- Aplicaţie:

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

1. Evaluare aplicativă: se evaluează argumentele `ak`, în ordine aleatoare (nu se garantează o anumită ordine).
2. Se evaluează corpul funcţiei, `expr`, ţinând cont de legările `pk` ← *valoare*(`ak`).
3. Valoarea aplicaţiei este valoarea lui `expr`.



## Construcţia `let`
### Definiţie, Exemplu, Semantică

- Leagă static variabile locale
- Sintaxă:

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

- Domeniul de vizibilitate a variabilei `vk` (cu valoarea `ek`): mulţimea punctelor din `expr` (corp let), în care apariţiile lui `vk` sunt libere.

Exemplu

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

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



## Construcţia `let*`
### Definiţie & Exemplu

- Leagă static variabile locale
- Sintaxă:

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

- Scope pentru variabila `vk` = mulţimea punctelor din
  - restul legărilor (legări ulterioare) şi
  - corp – `expr`

  în care apariţiile lui `vk` sunt libere

Exemplu

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



## Construcţia `let*`
### Semantică

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

echivalent cu

```
(let ((v1 e1))
  ...
    (let ((vn en))
      expr) ... )
```

- Evaluarea expresiilor `ei` se face în ordine!



## Construcţia `letrec`
### Definiţie

- Leagă static variabile locale
- Sintaxă:

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

- Domeniul de vizibilitate a variabilei `vk` = mulţimea punctelor din întreaga construcţie, în care apariţiile lui `vk` sunt libere



## Construcţia `letrec`
### Exemplu

Exemplu

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



## Construcţia `define`
### Definiţie & Exemplu

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

Definiţii echivalente:

```
(define f1
    (lambda (x)
        (add1 x)
))

(define (f2 x)
    (add1 x)
))
```

## Construcția `define`
### Exemplu de legare dinamică

- în Scheme (e.g. limbajul Pretty Big), `define` leagă dinamic și permite definiri multiple (în Racket nu mai este acceptat acest comportament):

Exemplu

```
(define x 0)
(define f (lambda () x))
(f)
(define x 1)
(f)
```

Output: `0 1`

- Dezavantaj: codul devine neintuitiv și se corupe transparența referențiale



## Evaluare



## Evaluarea în Racket

- 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`.



## Controlul evaluării

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

Exemplu

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



## Construcția programelor prin recursivitate



## 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.



## Recursivitate
### Tipuri

- 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*



## Sfârșitul cursului 2
### Elemente esențiale

- 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.

+ Dați feedback la acest curs aici:
[http://goo.gl/forms/SjDsW06v5J]

## Cursul 3: Calcul Lambda

λ





## Introducere



## Modele de calculabilitate
### De ce?

λ

- 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 mode 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).



## Calculul Lambda
### $\lambda$

λ

- Model de calculabilitate (Alonzo Church, 1932) – introdus în cadrul cercetărilor asupra fundamentelor matematicii. [http://en.wikipedia.org/wiki/Lambda_calculus]
  - sistem formal pentru exprimarea calculului.
- Echivalent cu Maşina Turing (v. Teza Church-Turing)
- Axat pe conceptul matematic de funcţie – totul este o funcţie



## Aplicaţii
### ale calculului $\lambda$

λ

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



## Lambda-expresii



## $\lambda$-expresii
### Exemple

λ

Ex Exemplu

1. $x$ → variabila (numele) $x$
2. $\lambda x.x$ → funcţia identitate
3. $\lambda x.\lambda y.x$ → funcţie selector
4. $(\lambda x.x\ \ y)$ → aplicaţia funcţiei identitate asupra parametrului actual $y$
5. $(\lambda x.(x\ \ x)\ \ \lambda x.x)$ → ?

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



## $\lambda$-expresii
### Definiţie

λ

**+ $\lambda$-expresie**

- Variabilă: o variabilă $x$ este o $\lambda$-expresie;
- Funcţie: dacă $x$ este o variabilă şi $E$ este o $\lambda$-expresie, atunci $\lambda x.E$ este o $\lambda$-expresie, reprezentând funcţia anonimă, unară, cu parametrul formal $x$ şi corpul $E$;
- Aplicaţie: dacă $F$ şi $A$ sunt $\lambda$-expresii, atunci $(F\ \ A)$ este o $\lambda$-expresie, reprezentând aplicaţia expresiei $F$ asupra parametrului actual $A$.

## Evaluare
Intuitiv

$\lambda$

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

||
substituție
⇓

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

||
substituție
⇓

$$z$$

nu mai este nicio funcție de aplicat

· cum știm ce reducem, cum reducem, în ce ordine, și ce apariții ale variabilelor înlocuim?



# Reducere



## $\beta$-redex
Cum arată (Formal, vedem mai târziu)

$\lambda$

- $\beta$-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.



## Apariţii ale variabilelor
Legate vs libere

$\lambda$

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

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

+ **Apariţie liberă** O apariţie a unei variabile este liberă într-o expresie dacă nu este legată în acea expresie.

- Atenţie! În raport cu o expresie dată!



## Apariţii ale variabilelor
Mod de gândire

$\lambda$

· 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.

- $\underset{<1>}{x}$ $\leftarrow$ apariție liberă
- $(\lambda y.\ \underset{<1>}{x}\ \ z)$ $\leftarrow$ apariție încă liberă, nu o leagă nimeni
- $\lambda\ \underset{<2>}{x}\ .(\lambda y.\ \underset{<1>}{x}\ \ z)$ $\leftarrow \lambda\ \underset{<2>}{x}$ *leagă* apariția $\underset{<1>}{x}$
- $(\lambda\ \underset{<2>}{x}\ .\underbrace{(\lambda y.\ \underset{<1>}{x}\ \ z)}_{\text{corp } \lambda x_2}\ \ \underset{<3>}{x}\ )$ $\leftarrow$ apariția $x_3$ este liberă – este în exteriorul corpului funcției cu parametrul formal $x$ $(\lambda x_2)$
- $\lambda\ \underset{<4>}{x}\ .(\lambda\ \underset{<2>}{x}\ .(\lambda y.\ \underset{<1>}{x}\ \ z)\ \ \underset{<3>}{x}\ )$ $\leftarrow \lambda\ \underset{<4>}{x}$ *leagă* apariția $\underset{<3>}{x}$



## Variabile
Legate vs libere

$\lambda$

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

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

- Atenţie! În raport cu o expresie dată!



## Variabile și Apariţii ale lor
Exemplu 1

$\lambda$

Ex Exemplu

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

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



## Variabile și apariţii ale lor
Exemplu 2

$\lambda$

Ex Exemplu

În expresia $E = (\lambda x.\lambda z.(z\ \ x)\ \ (z\ \ y))$, evidenţiem apariţiile:
$(\lambda\ \underset{<1>}{x}\ .\underbrace{\lambda\ \underset{<1>}{z}\ .(\ \underset{<2>}{z}\ \ \underset{<2>}{x}\ )}_{F}\ (\ \underset{<3>}{z}\ \ \underset{<1>}{y}\ ))$.

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

## Determinarea variabilelor libere şi legate
O abordare formală

λ

### Variabile libere (*free variables*)
- $FV(x) = \{x\}$
- $FV(\lambda x.E) = FV(E) \setminus \{x\}$
- $FV((E_1\ E_2)) = FV(E_1) \cup FV(E_2)$

### Variabile legate (*bound variables*)
- $BV(x) = \emptyset$
- $BV(\lambda x.E) = BV(E) \cup \{x\}$
- $BV((E_1\ E_2)) = BV(E_1) \setminus FV(E_2) \cup BV(E_2) \setminus FV(E_1)$



## Expresii închise

λ

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

Exemplu
- $(\lambda x.x\ \lambda x.\lambda y.x) \rightarrow$ închisă
- $(\lambda x.x\ a) \rightarrow$ deschisă, deoarece $a$ este liberă

- Variabilele libere dintr-o $\lambda$-expresie pot sta pentru alte $\lambda$-expresii – $\lambda x.((+\ x)\ 1)$.
- Înaintea evaluării, o expresie trebuie adusă la forma închisă.
- Procesul de înlocuire trebuie să se termine.



## $\beta$-reducere
Definiţie

λ

+ **$\beta$-reducere:** Evaluarea expresiei $(\lambda x.E\ A)$, cu $E$ și $A$ $\lambda$-expresii, prin substituirea textuală a tuturor apariţiilor libere ale parametrului **formal** al funcţiei, $x$, 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.



## $\beta$-reducere
Exemple

λ

Exemplu
- $(\lambda x.x\ y) \rightarrow_\beta x_{[y/x]} \rightarrow y$
- $(\lambda x.\lambda x.x\ y) \rightarrow_\beta \lambda x.x_{[y/x]} \rightarrow \lambda x.x$
- $(\lambda x.\lambda y.x\ y) \rightarrow_\beta \lambda y.x_{[y/x]} \rightarrow \lambda y.y$ Greşit! Variabila liberă $y$ devine legată, schimbându-şi semnificaţia. $\rightarrow \lambda y^{(a)}.y^{(b)}$

  Care este problema?



## $\beta$-reducere
Coliziuni

λ

- **Problemă**: în expresia $(\lambda x.E\ A)$:
  - variabilele libere din $A$ nu au nume comune cu variabilele legate din $E$: $\mathrm{FV}(A) \cap \mathrm{BV}(E) = \emptyset$
    $\rightarrow$ reducere întotdeauna corectă
  - există variabilele libere din $A$ care au nume comune cu variabilele legate din $E$: $\mathrm{FV}(A) \cap \mathrm{BV}(E) \neq \emptyset$
    $\rightarrow$ reducere potenţial greşită
- Soluţie: redenumirea variabilelor legate din $E$, ce coincid cu cele libere din $A$ $\rightarrow$ $\alpha$-conversie.

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
Definiţie

λ

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

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

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



## $\alpha$-conversie
Exemple

λ

Exemplu
- $\lambda x.(x\ y) \rightarrow_\alpha \lambda z.(z\ y) \rightarrow$ Corect!
- $\lambda x.\lambda x.(x\ y) \rightarrow_\alpha \lambda y.\lambda x.(x\ y) \rightarrow$ Greşit! $y$ este liberă în $\lambda x.(x\ y)$
- $\lambda x.\lambda y.(y\ x) \rightarrow_\alpha \lambda y.\lambda y.(y\ y) \rightarrow$ Greşit! Apariţia 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!



## Reducere
Definiţii

λ

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

+ **Secvenţă de reducere:** Succesiune de zero sau mai mulţi paşi de reducere:
$E_1 \rightarrow^* E_2$.
Reprezintă un element din închiderea reflexiv-tranzitivă a relaţiei $\rightarrow$ .

## Reducere
### Proprietăți

: Reducere

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

Ex Exemplu

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



## Evaluare



## Întrebări
### Pentru construcția unei mașini de calcul

· 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. Comportamentul depinde de secvența de reducere?
3. Dacă mai multe secvențe de reducere se termină, obţinem întotdeauna acelaşi rezultat?
4. Dacă rezultatul este unic, cum îl obţinem?



## 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 conţine $\beta$-redecşi i.e. care nu mai poate fi redusă.

+ **Forma normală funcţională – FNF** este o formă $\lambda x.F$, în care $F$ poate conţine $\beta$-redecşi.

Ex Exemplu

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



## Terminarea reducerii (reductibilitate)
### Exemplu și definiție

Ex Exemplu

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

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

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

· expresia $\Omega$ nu este reductibilă.



## Secvenţe de reducere
### şi terminare

Dar!

Ex Exemplu

$E = (\lambda x.y\ \Omega)$
$\rightarrow y$ *sau*
$\rightarrow E \rightarrow y$ *sau*
$\rightarrow E \rightarrow E \rightarrow y$ *sau...*
$\ldots$
$\stackrel{n}{\rightarrow}{}^* y,\ n \geq 0$
$\stackrel{\infty}{\rightarrow}{}^* \ldots$

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



## Unicitatea formei normale
### Rezultate

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

$$\begin{tikzcd} & E_1 \arrow[dr, "*"] & \\ E \arrow[ur, "*"] \arrow[dr, "*"'] & & E_3 \\ & E_2 \arrow[ur, "*"'] & \end{tikzcd}$$

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



## Unicitatea formei normale
### Exemplu

Ex 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.

## Modalităţi de reducere
### Cum putem *organiza* reducerea?

λ

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

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 şi mai din dreapta $\beta$-redex.

Exemplu

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



## Ce modalitate alegem?

λ

**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.



## Răspunsuri la întrebări

λ

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 secvenţa de reducere?
   → DA.
3. Dacă mai multe secvențe de reducere se termină, obţinem întotdeauna acelaşi rezultat?
   → DA.
4. Dacă rezultatul este unic, cum îl obţinem?
   → Reducere stânga-dreapta.
5. Care este valoarea expresiei?
   → Forma normală [funcțională] (FNF).



## Ordine de evaluare
### Tipuri

λ

- **+ Evaluare aplicativă** (*eager*) – corespunde reducerii dreapta-stânga. Parametrii funcţiilor sunt evaluaţi înaintea aplicării funcţiei.
- **+ Evaluare normală** (*lazy*) – corespunde reducerii stânga-dreapta. Parametrii funcţiilor sunt evaluaţi la cerere.
- **+ Funcţie strictă** – funcţie cu evaluare aplicativă.
- **+ Funcţie nestrictă** – funcţie cu evaluare normală.



## Ordine de evaluare
### În practică

λ

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

Exemplu

`(+ (+ 2 3) (* 2 3))` → `(+ 5 6)` → `11`

- Nevoie de funcţii nestricte, chiar în limbajele aplicative: `if`, `and`, `or` etc.

Exemplu

`(if (< 2 3) (+ 2 3) (* 2 3))` → `(< 2 3)` → `#t` → `(+ 2 3)` → `5`



# Limbajul lambda-0 și incursiune în TDA



## Limbajul $\lambda_0$
### Scop

λ

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



## Limbajul $\lambda_0$ – Convenţii
### Evaluare

λ

Scrieri prescurtate:

- $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.E \rightarrow \lambda x_1 x_2 \ldots x_n.E$
- $((\ldots((E\ A_1)\ A_2)\ \ldots)\ A_n) \rightarrow (E\ A_1\ A_2\ \ldots\ A_n)$

Exemplu

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

## Tipuri de date
### Cum reprezentăm datele? Cum interpretăm valorile?

λ

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

  Exemplu

  $T \equiv_{\text{def}} \lambda x.\lambda y.x \equiv \lambda x\,y.x$ $\quad F \equiv_{\text{def}} \lambda x.\lambda y.y \equiv \lambda x\,y.y$

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

  Exemplu

  $not \equiv_{\text{def}} \lambda x.(x\ F\ T)$

  $(not\ T) \to (\lambda x.(x\ F\ T)\ T) \to (T\ F\ T) \to F$



## TDA
### Definiție

λ

**+ 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.



## TDA *Bool*
### Specificare

λ

· Constructori: $\left|\begin{array}{ll} T: & \to Bool \\ F: & \to Bool \end{array}\right.$

· Operatori: $\left|\begin{array}{ll} not: & Bool \to Bool \\ and: & Bool^2 \to Bool \\ or: & Bool^2 \to Bool \end{array}\right.$

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



## TDA *Bool*
### Implementarea constructorilor de bază

λ

Intuiție: selecția între cele două valori, *true* și *false*

- $T \equiv_{\text{def}} \lambda x\,y.x$
- $F \equiv_{\text{def}} \lambda x\,y.y$
- Comportament de selectori:
  - $(T\ a\ b) \to (\lambda x\,y.x\ a\ b) \to a$
  - $(F\ a\ b) \to (\lambda x\,y.y\ a\ b) \to b$



## TDA *Bool*
### Implementarea operatorilor

λ

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



## TDA *Pair*
### Implementare

λ

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



## TDA *List* și *Natural*
### Implementare

λ

Intuiție: listă → pereche (*head*, *tail*)

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

---

Intuiție: număr → 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]



## Absența tipurilor
### Chiar avem nevoie de tipuri? – Rolul tipurilor

λ

- 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;

## Absența tipurilor
### Consecințe asupra reprezentării obiectelor

λ

- 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 → operator!



## Absența tipurilor
### Consecințe asupra corectitudinii calculului

λ

- Incapacitatea Mașinii λ 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.



## Absența tipurilor
### Consecințe pozitive

λ

- 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



## Recursivitate
### Perspective asupra recursivității

λ

· 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.



## Implementare *length*
### Problemă

λ

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



## Combinator de punct fix
### mai multe la
[http://en.wikipedia.org/wiki/Lambda_calculus#Recursion_and_fixed_points]

λ

Exemplu

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

- $(Fix\ F) \rightarrow (\lambda x.(F\ (x\ x))\ \lambda x.(F\ (x\ x))) \rightarrow$
  $(F\ \underline{(\lambda x.(F\ (x\ x))\ \lambda x.(F\ (x\ x)))}) \rightarrow (F\ (Fix\ F))$
- $(Fix\ F)$ este un punct fix al lui $F$.
- *Fix* se numește combinator de punct fix.

- $length \equiv_{def} (Fix\ Length) \rightarrow (Length\ (Fix\ Length)) \rightarrow$
  $\lambda L.(if\ (null\ L)\ zero\ (succ\ ((Fix\ Length)\ (cdr\ L))))$
- Funcție recursivă, fără a fi textual recursivă!



# Racket vs. lambda-0



## Racket vs. $\lambda_0$
### Construcția expresiilor / sintaxă

λ

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

## Racket vs. $\lambda_0$
### Mai precis

- 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 execuţie, în momentul aplicării unei funcţii;
- evaluare aplicativă;
- permite recursivitate textuală;
- avem legări top-level.



## Sfârșitul cursului 3
### Elemente esențiale

- Baza formală a calculului $\lambda$:
- expresie $\lambda$, $\beta$-redex, variabile și apariții legate vs. libere, expresie închisă, $\alpha$-conversie, $\beta$-reducere
- FN și FNF, reducere, reductibilitate, evaluare aplicativă și normală
- TDA și recursivitate pentru calcul lambda

+ Dați feedback la acest curs aici:
[http://goo.gl/forms/SjDsW06v5J]



# Anexă: TDA



## Notație

→ Folosim notația:

- $\lambda x_1.\lambda x_2.\ldots\lambda x_n.E \rightarrow \lambda x_1 x_2 \ldots x_n.E$
- $((\ldots((E\ A_1)\ A_2)\ \ldots)\ A_n) \rightarrow (E\ A_1\ A_2\ \ldots\ A_n)$



## Elemente TDA

→ Pentru definirea unui Tip de Date Abstract avem nevoie de:

- Constructori de bază → un set minimal de funcții, prin aplicarea (eventual, repetată) cărora se poate construi oricare element din mulțimea de valori a tipului.
  - constructorii de bază construiesc valorile.
- Operatori → setul complet de funcții care pot lucra cu valorile din tipul de bază.
  - operatorii arată ce operații putem face cu valorile.
- Axiome → definesc rezultatul operatorilor pentru toate posibilele valori (ne ajutăm de constructorii de bază).
  - axiomele arată ce rezultate obținem din operații.



## TDA *Bool*
### Specificare

· Constructori:
$$\begin{array}{ll} T: & \rightarrow Bool \\ F: & \rightarrow Bool \end{array}$$

· Operatori:
$$\begin{array}{ll} not: & Bool \rightarrow Bool \\ and: & Bool^2 \rightarrow Bool \\ or: & Bool^2 \rightarrow Natural \end{array}$$

· Axiome:
$$\begin{array}{ll} not: & not(T) = F \\ & not(F) = T \\ and: & and(T,a) = a \\ & and(F,a) = F \\ or: & or(T,a) = T \\ & or(F,a) = a \end{array}$$



## TDA *Bool*
### Implementarea constructorilor de bază

**Intuiție:** selecția între cele două valori, *true* și *false*

- $T \equiv_{\text{def}} \lambda x\ y.x$
- $F \equiv_{\text{def}} \lambda x\ y.y$
- Comportament de selectori:
  - $(T\ a\ b) \rightarrow (\lambda x\ y.x\ a\ b) \rightarrow a$
  - $(F\ a\ b) \rightarrow (\lambda x\ y.y\ a\ b) \rightarrow b$



## TDA *Bool*
### Implementarea operatorilor

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

## TDA *Bool*

### Condiționala *if*

λ

· Operator: | $if : Bool \times A \times A \to A$

· Axiome: | $if(T, a, b) = a$
$if(F, a, b) = b$

- Implementare: $if \equiv_{\text{def}} \lambda c\ t\ e.(c\ t\ e)$
  - $(if\ T\ a\ b) \to (\lambda c\ t\ e.(c\ t\ e)\ T\ a\ b) \to (T\ a\ b) \to a$
  - $(if\ F\ a\ b) \to (\lambda c\ t\ e.(c\ t\ e)\ F\ a\ b) \to (F\ a\ b) \to b$
- Funcție nestrictă!



## TDA *Pair*

### Specificare

λ

· Constructori: | $pair : A \times B \to Pair$

· Operatori: | $fst : Pair \to A$
$snd : Pair \to B$

· Axiome: | $snd(pair(a, b)) = b$
$fst(pair(a, b)) = a$



## TDA *Pair*

### Implementare

λ

Intuiție: pereche →funcție ce așteaptă selectorul, pentru a-l aplica asupra membrilor

- $pair \equiv_{\text{def}} \lambda x\ y\ z.(z\ x\ y)$
  - $(pair\ a\ b) \to (\lambda x\ y\ z.(z\ x\ y)\ a\ b) \to \lambda z.(z\ a\ b)$
- $fst \equiv_{\text{def}} \lambda p.(p\ T)$
  - $(fst\ (pair\ a\ b)) \to (\lambda p.(p\ T)\ \lambda z.(z\ a\ b)) \to (\lambda z.(z\ a\ b)\ T) \to (T\ a\ b) \to a$
- $snd \equiv_{\text{def}} \lambda p.(p\ F)$
  - $(snd\ (pair\ a\ b)) \to (\lambda p.(p\ F)\ \lambda z.(z\ a\ b)) \to (\lambda z.(z\ a\ b)\ F) \to (F\ a\ b) \to b$



## TDA *List*

### Specificare

λ

· Constructori: | $nil : \ \to List$
$cons : \ A \times List \to List$

· Operatori: | $car : \ List \setminus \{nil\} \to A$
$cdr : \ List \setminus \{nil\} \to List$
$null? : \ List \to Bool$
$append : \ List^2 \to List$

· Axiome:
$car : \quad car(cons(e, L)) = e$
$cdr : \quad cdr(cons(e, L)) = L$
$null? : \quad null?(nil) = T$
$\quad null?(cons(e, L)) = F$
$append : \quad append(nil, B) = B$
$\quad append(cons(e, A), B) = cons(e, append(A, B))$



## TDA *List*

### Implementare

λ

Intuiție: listă → pereche (*head*, *tail*)

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



## TDA *List*

### Implementare *append*

λ

- $append \equiv_{\text{def}}$
  $\lambda A\ B.(if\ (null?\ A)\ B\ (cons\ (car\ A)\ (append\ (cdr\ A)\ B)))$

· Problemă: expresia nu admite formă închisă! → vezi eliminarea recursivității textuale.



## TDA *Natural*

### Specificare

λ

· Constructori: | $zero : \ \to Natural$
$succ : \ Natural \to Natural$

· Operatori: | $pred : \ Natural \setminus \{zero\} \to Natural$
$zero? : \ Natural \to Bool$
$add : \ Natural^2 \to Natural$

· Axiome: | $pred : \quad pred(succ(n)) = n$
$zero? : \quad zero?(zero) = T$
$\quad zero?(succ(n)) = F$
$add : \quad add(zero, n) = n$
$\quad add(succ(n), m) = succ(add(n, m))$



## TDA *Natural*

### Implementare

λ

Intuiție: număr → listă cu lungimea egală cu valoarea numărului

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

## Recapitulare Calcul $\lambda$



## $\lambda$-expresie
### Sintaxa pentru calcul $\lambda$

· O $\lambda$-expresie poate fi:

- $x$
- $\lambda x.E$ $\quad$ $E$ $\lambda$-expresie
- $(F\ \ A)$ $\quad$ $F, A$ $\lambda$-expresii

Exemple:

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



## $\beta$Redex
### Ce reducem?

· Sursa pentru $\beta$-reducere și pasul de reducere.

· Este o funcție care se poate aplica.

$$(\lambda x.\underbrace{\qquad}_{corp}\ \underbrace{\qquad\qquad}_{parametru\ actual})$$

· $x$: numele parametrului formal.



## $\beta$-reducere
### Cum reducem?

· substituție textuală

$$(\lambda x.\underbrace{\qquad}_{corp}\ \underbrace{\qquad\qquad}_{parametru\ actual}) \rightarrow_\beta \underbrace{\qquad}_{corp}{}_{[parametru\ actual/x]}$$

aparițiile libere ale lui $x$ din corp sunt substituite textual cu parametrul actual



## Substituție
### Ce substituim?

↓ apariție liberă în corp

$$\lambda x.\underbrace{\qquad x \qquad}_{corp}$$

$$\lambda x.\underbrace{\quad \lambda x.\overbrace{\qquad x \qquad}^{corp_2}}_{corp_1}$$

↑ leagă apariția în $corp_1$
(apariția este liberă în $corp_2$)

· O apariție $x$ este legată de cea mai interioară definiție $\lambda x$, care conține apariția în corpul său. Dacă $\lambda x$ care îl leagă este inclus în expresia $E$, apariția este legată în $E$, altfel este liberă în $E$.

· $x$ are o apariție liberă în $E \Rightarrow x$ variabilă liberă în $E$ (altfel legată).

· $\nexists$ variabile libere în $E \Rightarrow E$ închisă.



## Condiții $\beta$-reducere pentru $(\lambda x.E\ \ A)$
### Când este corect să efectuăm substituția?

· Variabilele **libere** din $A$ nu devin **legate** în $E_{[A/x]}$

· Mai precis, numele variabilelor libere din $A$ nu sunt nume de variabile care sunt legate în contextele din $E$ în care apare $x$.

· Exemplu: $(\lambda x.\lambda y.(y\ \ x)\ \ \lambda z.y) \rightarrow$ incorect să efectuăm $\beta$-reducere.



## $\alpha$-conversie în $(\lambda x.E\ \ A)$
### Cum rezolvăm problema anterioară?

· când? → când variabilele din $A$ devin legate în $E_{[A/x]}$

· ce redenumim? → parametri formali ai tuturor funcțiilor din $E$ care conțin apariții libere ale lui $x$ în corp și au ca parametru formal numele unei variabile libere din $A$ (redenumirea parametrilor formali implică folosirea noului nume în toate aparițiile libere ale parametrilor formali în corpurile funcțiilor respective).

· la ce redenumim? → la un nume care nu este nume de variabilă liberă în $A$ sau în propriul corp, și care nu devine legat în corp.



## Pas de reducere
### Cum efectuăm o reducere corectă?

[$\alpha$-conversie] + $\beta$-reducere fără coliziuni

1. avem $\beta$-redex
2. dacă este cazul, efectuăm $\alpha$-conversie
3. efectuăm $\beta$-reducere

## Secvență de reducere

### Cum facem o reducere completă?

λ

Secvență de reducere = $\rightarrow^*$

· Dacă expresia este reductibilă (are o secvență de reducere care se termină), reducerea în ordine stânga-dreapta se va termina cu valoarea expresiei.



## Cursul 4: Programare funcţională în Racket II



## Sfârșitul cursului 4

### Elemente esențiale

- Exemple mai avansate de legare în Racket
- Exemple mai avansate de utilizare Racket

+ Dați feedback la acest curs aici:
[http://goo.gl/forms/SjDsW06v5J]



## Cursul 5: Evaluare leneșă în Racket





# Întârzierea evaluării



## Motivație

### De ce? → Luăm un exemplu

Ex 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.



## Varianta 1

### Încercare → implementare directă

```
(define prod
  (lambda (x y)
    (if x (* y (+ y 1)) 0)))

(define test
  (lambda (x)
    (let ((y 5))
      (prod x (and (display "y␣") y)))))

(test #f)
(test #t)
```

Output: `y 0 | y 30`

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



## Varianta 2

### Încercare → `quote` & `eval`

```
(define prod
  (lambda (x y)
    (if x (* (eval y) (+ (eval y) 1)) 0)))

(define test
  (lambda (x)
    (let ((y 5))
      (prod x (quote (and (display "y␣") y))))))

(test #f)
(test #t)
```

Output: `0 | y undefined`

- `x = #f` → comportament corect: `y` neevaluat
- `x = #t` → eroare: `quote` nu salvează contextul

(

Paranteză!



## Contexte computaţionale
### Definiţie

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

- Legare statică → mulţimea variabilelor care îl conţin pe $P$ în domeniul lexical de vizibilitate
- Legare dinamică → mulţimea variabilelor definite cel mai recent, la momentul $t$, şi referite din $P$



## Contexte computaţionale
### Exemplu

Ex | Exemplu Ce variabile locale conţine contextul computaţional al punctului $P$?

```
(lambda (x y)
  (lambda (z)
    (let ((x (car y)))
      ; ..P..)))
```



## Închideri funcţionale
### Definiţie

**+ Închidere funcţională:** funcţie care îşi salvează contextul, pe care îl va folosi, în momentul aplicării, pentru evaluarea corpului.

· Notaţie: închiderea funcţiei $f$ în contextul $C \rightarrow < f; C >$

Ex | Exemplu

$< \lambda x.z; \{z \leftarrow 2\} >$



)

închidem paranteza



## Varianta 3
### Încercare → închideri funcționale

```
(define prod
  (lambda (x y)
    (if x (* (y) (+ (y) 1)) 0))) ; (y)

(define test
  (lambda (x)
    (let ((y 5))
      (prod x
            (lambda () (and (display "y␣") y))))))

(test #f)
(test #t)
```

Output: `0 | y y 30`

- Comportament corect: `y` evaluat la cerere
- `x = #t` → `y` evaluat de 2 ori → ineficient



## Varianta 4
### Promisiuni: `delay` & `force`

```
(define prod
  (lambda (x y)
    (if x (* (force y) (+ (force y) 1)) 0)))

(define test
  (lambda (x)
    (let ((y 5))
      (prod x
            (delay (and (display "y␣") y))))))

(test #f)
(test #t)
```

Output: `0 | y 30`

- Rezultat corect: `y` evaluat la cerere, o singură dată → evaluare leneșă



## Promisiuni
### Descriere

- 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.

## Promisiuni
### Proprietăți

- 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.



## Evaluare întârziată
### Abstractizare a implementării cu promisiuni

Ex Continuare a exemplului cu funcția `prod`

```
(define-syntax-rule (pack expr) (delay expr))

(define unpack force)


(define prod (lambda (x y)
    (if x (* (unpack y) (+ (unpack y) 1)) 0)))
(define test (lambda (x)
    (let ((y 5))
      (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`

```
(define-syntax-rule (pack expr) (lambda () expr) )

(define unpack (lambda (p) (p)))


(define prod (lambda (x y)
    (if x (* (unpack y) (+ (unpack y) 1)) 0)))
(define test (lambda (x)
    (let ((y 5))
      (prod x (pack (and (display "y␣") y))) )))
```

· utilizarea nu depinde de implementare (același cod ca anterior, altă implementare a funcționalității de evaluare întârziată).



# Fluxuri



## Motivație
### Luăm un exemplu

Ex Determinați suma numerelor pare[1] din intervalul $[a, b]$.

```
(define even-sum-iter ; varianta 1
  (lambda (a b)
    (let iter ((n a)
               (sum 0))
      (cond ((> n b) sum)
            ((even? n) (iter (+ n 1) (+ sum n)))
            (else (iter (+ n 1) sum))))))


(define even-sum-lists ; varianta 2
  (lambda (a b)
    (foldl + 0 (filter even? (interval a b)))))
```

[1] stă pentru o verificare potențial mai complexă, e.g. numere prime



## Motivație
### Observații

- 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



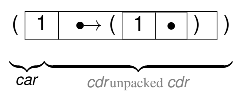
## Fluxuri
### Caracteristici

- 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).



## Fluxuri
### Intuitiv

- 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 cererere.



## Fluxuri
### Operatori: construcție și selecție

- cons, car, cdr, nil, null?

```
(define-macro stream-cons (lambda (head tail)
  `(cons ,head (pack ,tail))))

(define stream-car car)

(define stream-cdr (lambda (s)
  (unpack (cdr s))))

(define stream-nil '())

(define stream-null? null?)
```



## Fluxuri
### Operatori: take și drop

- selecție / eliminare dintr-un flux a $n$ elemente.

```
(define stream-take (lambda (s n)
  (cond ((zero? n) '())
        ((stream-null? s) '())
        (else (cons (stream-car s)
             (stream-take (stream-cdr s) (- n 1))))
)))

(define stream-drop (lambda (s n)
  (cond ((zero? n) s)
        ((stream-null? s) s)
        (else (stream-drop (stream-cdr s) (- n 1)))
)))
```



## Fluxuri
### Operatori: map și filter

- operatori de aplicare și filtrare pe liste.

```
(define stream-map (lambda (f s)
    (if (stream-null? s) s
        (stream-cons (f (stream-car s))
            (stream-map f (stream-cdr s)))
)))

(define stream-filter (lambda (f? s)
    (cond ((stream-null? s) s)
          ((f? (stream-car s))
            (stream-cons (stream-car s)
                (stream-filter f? (stream-cdr s))))
          (else (stream-filter f? (stream-cdr s)))
)))
```



## Fluxuri
### Operatori: zip, append și conversie

```
(define stream-zip (lambda (f s1 s2)
  (if (stream-null? s1) s2
    (stream-cons (f (stream-car s1) (stream-car s2))
     (stream-zip f (stream-cdr s1) (stream-cdr s2)))
)))

(define stream-append (lambda (s1 s2)
  (if (stream-null? s1) s2
    (stream-cons (stream-car s1)
      (stream-append (stream-cdr s1) s2)))))

(define list->stream (lambda (L)
  (if (null? L) stream-null
    (stream-cons (car L) (list->stream (cdr L))))))
```



## Fluxuri – Exemple
### Implementarea unui flux de numere 1

- Definiție cu închideri:

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

- Definiție cu fluxuri:

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

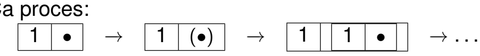
- Definiție cu promisiuni:

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

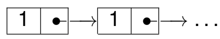


## Fluxuri – Exemple
### Flux de numere 1 – discuție

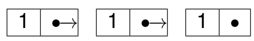
- Ca proces:




- Structural:




- Extinderea se realizează în spațiu constant:






## Fluxul numerelor naturale
### Formulare explicită

```
(define naturals-from (lambda (n)
    (stream-cons n (naturals-from (+ n 1)))))

(define naturals (naturals-from 0))
```

```
(define naturals
  (stream-cons 0
     (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

```
(define even-naturals
  (stream-filter even? naturals))

(define even-naturals
  (stream-zip-with + naturals naturals))
```

## Fluxul numerelor prime
### Metodă

- 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.



## Fluxul numerelor prime
### Implementare

```
(define sieve (lambda (s)
  (if (stream-null? s) s
    (stream-cons (stream-car s)
      (sieve (stream-filter
        (lambda (n) (not (zero?
          (remainder n (stream-car s)))))
        (stream-cdr s)
    )))
)))

(define primes (sieve (naturals-from 2)))
```



## Căutare leneșă în spațiul stărilor



## Spațiul stărilor unei probleme

**+ 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



## Căutare în spațiul stărilor

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



## Căutare în lățime
### *Obișnuită*

```
(define breadth-search-goal
 (lambda (init expand goal?)
  (letrec ((search (lambda (states)
   (if (null? states) '()
    (let ((state (car states)) (states (cdr states)))
         (if (goal? state) state
            (search (append states (expand state)))
   ))))))
   (search (list init)))))
```

- Generarea unei singure soluții
- Cum le obținem pe celelalte, mai ales dacă spațiul e infinit?



## Căutare în lățime
### Leneșă (1) – fluxul stărilor *scop*

```
(define lazy-breadth-search (lambda (init expand)
  (letrec ((search (lambda (states)
           (if (stream-null? states) states
             (let ((state (stream-car states))
                   (states (stream-cdr states)))
               (stream-cons state
                 (search (stream-append states
                                 (expand state)))
           ))))))
      (search (stream-cons init stream-nil))
)))
```

## Căutare în lățime
### Leneșă (2)

```
(define lazy-breadth-search-goal
  (lambda (init expand goal?)
    (stream-filter goal?
      (lazy-breadth-search init expand))
))
```

- Nivel înalt, conceptual: separare între explorarea spațiului și identificarea stărilor *scop*.
- Nivel scăzut, al instrucțiunilor: întrepătrunderea celor două aspecte.
- Aplicații:
  - Palindroame
  - Problema reginelor



## Sfârșitul cursului 5
### Elemente esențiale

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

+ Dați feedback la acest curs aici:
[http://goo.gl/forms/SjDsW06v5J]



## Anexă: Evaluare și legarea variabilelor în Racket



## Legare dinamică și închideri funcționale
### Setup

- Racket nu suportă re-definirea unui simbol prin `define`.
- Pentru rularea acestor exemple, alegeți limbajul Pretty Big pentru execuție.



## Legare dinamică și închideri funcționale
### Exemplul 1

Ex Exemplu

```
(define x 0)
(define f (lambda () x))
(f)
(define x 1)
(f)
```

Output: `0 1`



## Legare dinamică și închideri funcționale
### Exemplul 2 (mai obscur)

Ex Exemplu

```
(define factorial (lambda (n)
    (if (zero? n) 1
        (* n (factorial (- n 1))))))

(factorial 5)

(define g factorial)
(define factorial (lambda (x) x))

(g 5)
```

Output: `120 20`



## Legare dinamică și închideri funcționale
### Exemplul 3

Ex Exemplu

```
(define x 0)
(define f (lambda () x))
(define x 1)

(define g
  (lambda (x)
    (f)))

(g 2)
```

Output: `1`



## Legare dinamică și închideri funcționale
### Exemplul 4

Ex Exemplu

```
(define comp
  (lambda (f) (lambda (g) (lambda (x) (f (g x))))))
(define inc (lambda (x) (+ x 1)))
(define comp-inc (comp inc))

(define double (lambda (x) (* x 2)))
(define comp-inc-double (comp-inc double))
(comp-inc-double 5) ; 11

(define inc (lambda (x) x))
(comp-inc-double 5) ; tot 11
```

## Legare dinamică și închideri funcționale
### Exemplul 4 – comentarii

- *comp-inc* $\equiv < \lambda g.\lambda x.(f\ (g\ x)); \{f \leftarrow \lambda x.(+\ x\ 1)\} >$
- *comp-inc-double* $\equiv$
  $< \lambda x.(f\ (g\ x)); \{f \leftarrow \lambda x.(+\ x\ 1), g \leftarrow \lambda x.(*\ x\ 2)\} >$
- Inutilitatea redefinirii lui `inc`: valoarea sa (o funcție) fusese deja salvată în context, în momentul aplicării



## Anexă: Efecte laterale



## Construcţia `set!`

- Modifică valoarea unei variabile
  Ex Exemplu

```
(define x 0)
(define f (lambda (p)
    (set! x p)
    x))
(f 3) ; 3
x ; 3
```

- Diferenţă la nivel de intenţie faţă de `let`-uri şi `define`, care urmăresc definirea de variabile noi şi nu modificarea celor existente!



## Atribuiri

- Avantaje:
  - Modelarea obiectelor a căror stare variază în timp
  - Evitarea pasării explicite a fiecărei modificări de stare
- Dezavantaj: pierderea transparenţei referenţiale.



## Cursul 6: Programare funcțională în Haskell





## Introducere



## Haskell
[https://en.wikipedia.org/wiki/Haskell_(programming_language)]

- din 1990;
- GHC – Glasgow Haskell Compiler (The Glorious Glasgow Haskell Compilation System)
  - dialect Haskell standard *de facto*;
  - compilează în/folosind C;
- Haskell Platform
  - include manager de pachete, debugger, generator de parsere, unelte de documentație, legare la cod C, etc.
- nume luat după logicianul Haskell Curry;
- aplicații: Pugs, Darcs, Linspire, Xmonad, Cryptol, seL4, Pandoc, web frameworks.



## Paralelă între limbaje

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

# Sintaxă



## Funcții

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

Exemplu : Definiții echivalente ale funcției `add`:

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

result       =   add1 1 2       -- echivalent, ((add1 1) 2)
result2      =   add3 1 2       -- echivalent, ((add3 1) 2)
inc          =   add1 1
```



## Funcții vs operatori

- Aplicabilitatea parțială a operatorilor infixați
- Transformări operator → funcție și funcție → operator

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

```
add4        =   (+)
result1     =   (+) 1 2
result2     =   1 `add4` 2

inc1        =   (1 +)
inc2        =   (+ 1)
inc3        =   (1 `add4`)
inc4        =   (`add4` 1)
```



## *Pattern matching*

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

Exemplu

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

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

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

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



## *List comprehensions*

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

Exemplu

```
squares lst     =   [x * x | x <- lst]

quickSort []    =   []
quickSort (h:t) =   quickSort [x | x <- t, x <= h]
                ++  [h]
                ++  quickSort [x | x <- t, x > h]

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



# Evaluare



## 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

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

Evaluare:

```
f (2 + 3, 3 + 5) (5 + 8)
→ (2 + 3) + (2 + 3)
→ 5 + 5        reutilizăm rezultatul primei evaluări!
→ 10            ceilalți parametri nu sunt evaluați
```



## Pași în aplicarea funcțiilor
### Exemplu

Exemplu

```
frontSum (x:y:zs)   =   x + y
frontSum [x]        =   x

notNil []       =   False
notNil (_:_)    =   True

frontInterval m n
    | notNil xs =   frontSum xs
    | otherwise =   n
    where
        xs      =   [m .. n]
```

## Pași în aplicarea funcțiilor
### Ordine

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

```
f 3 5
   evaluare pattern
?? notNil xs                          evaluare
   prima gardă
??      where                         necesar xs
   → evaluare where
??          xs = [3 .. 5]
??          → 3:[4 .. 5]
??  → notNil (3:[4 .. 5])
??  → True
→ front xs                            evaluare
   valoare gardă
   where
      xs = 3:[4 .. 5]            xs deja calculat
      → 3:4:[5]
→ front (3:4:[5])
→ 3 + 4 → 7
```



## Consecințe

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

Ex Exemplu

```
ones            =   1 : ones

naturalsFrom n  =   n : (naturalsFrom (n + 1))
naturals1       =   naturalsFrom 0
naturals2       =   0 : (zipWith (+) ones naturals2)

evenNaturals1   =   filter even naturals1
evenNaturals2   =   zipWith (+) naturals1 naturals2

fibo            =   0 : 1 : (zipWith (+) fibo (tail fibo))
```



## Sfârșitul cursului 6
### Elemente esențiale

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

+ Dați feedback la acest curs aici:
[http://goo.gl/forms/SjDsW06v5J]



## Cursul 7: Tipuri în Haskell

28 Tipare

29 Sinteză de tip



## Tipare



## Tipuri
### Pentru toate valorile (inclusiv funcții)

- 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`



## Tipuri
### Exemple de valori

Ex Exemplu

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

- Tipurile de bază sunt tipurile elementare din limbaj:
  `Bool, Char, Integer, Int, Float, ...`
- Reprezentare uniformă:

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

## Constructori de tip
⇒ tipuri noi pentru valori sau funcții

- Funcții de tip, ce îmbogățesc tipurile din limbaj.

**Constructori de tip predefiniți**

```
-- Constructorul de tip functie: ->
(-> Bool Bool) ⇒ Bool -> Bool
(-> Bool (Bool -> Bool)) ⇒ Bool -> (Bool -> Bool)

-- Constructorul de tip lista: []
([] Bool) ⇒ [Bool]
([] [Bool]) ⇒ [[Bool]]

-- Constructorul de tip tuplu: (,...,)
((,) Bool Char) ⇒ (Bool, Char)
((,,) Bool ((,) Char [Bool]) Bool)
          ⇒ (Bool, (Char, [Bool]), Bool)
```



## Constructori de tip
Tipurile funcțiilor

- Constructorul `->` este asociativ dreapta:
  `Integer -> Integer -> Integer`
  `≡ Integer -> (Integer -> Integer)`

**Exemplu**

```
add6        ::  Integer -> Integer -> Integer
add6 x y    =   x + y

f           ::  (Integer -> Integer) -> Integer
f g         =   (g 3) + 1

idd         ::  a -> a        -- functie polimorfica
idd x       =   x             -- a: variabila de tip!
```



## Constructorul de tip `Natural`
Exemplu de definire TDA 1

**Exemplu**

```
data Natural    =   Zero
                |   Succ Natural
    deriving (Show, Eq)

unu                 =   Succ Zero
doi                 =   Succ unu

addNat Zero n       =   n
addNat (Succ m) n   =   Succ (addNat m n)
```



## Constructorul de tip `Natural`
Comentarii

- 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*.

```
Zero :: Natural
Succ :: Natural -> Natural
```



## Constructorul de tip `Pair`
Exemplu de definire TDA 2

**Exemplu**

```
data Pair a b   = P a b
    deriving (Show, Eq)

pair1           =   P 2 True
pair2           =   P 1 pair1

myFst (P x y)   =   x
mySnd (P x y)   =   y
```



## Constructorul de tip `Pair`
Comentarii

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

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



## Polimorfism

**+ 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: `==`.

· mai multe detalii în cursul următor.



# Sinteză de tip

## Sinteza de tip
### Definiție

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

- Adnotările explicite de tip, deși posibile, nenecesare 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.



## Proprietăți induse de tipuri

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



## Exemple de sinteză de tip
### Transformare de funcție

**Ex Exemplul 1**

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

$$\frac{\texttt{g :: a} \qquad \texttt{(g 3) + 1 :: b}}{\texttt{f :: a -> b}}\ (\texttt{TLambda})$$

$$\frac{\texttt{(g 3) :: Int} \qquad \texttt{1 :: Int}}{\texttt{(g 3) + 1 :: Int}}\ (\texttt{T+})$$

$\Rightarrow$ `b = Int`

$$\frac{\texttt{g :: c -> d} \qquad \texttt{3 :: c}}{\texttt{(g 3) :: d}}\ (\texttt{TApp})$$

$\Rightarrow$ `a = c -> d, c = Int, d = Int`

$\Rightarrow$ `f :: (Int -> Int) -> Int`



## Exemple de sinteză de tip
### Combinator de punct fix

**Ex Exemplul 2**

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

$$\frac{\texttt{f :: a} \qquad \texttt{f (fix f) :: b}}{\texttt{fix :: a -> b}}\ (\texttt{TLambda})$$

$$\frac{\texttt{f :: c -> d} \qquad \texttt{(fix f) :: c}}{\texttt{(f (fix f)) :: d}}\ (\texttt{TApp})$$

$\Rightarrow$ `a = c -> d, b = d`

$$\frac{\texttt{fix :: e -> g} \qquad \texttt{f :: e}}{\texttt{(fix f) :: g}}\ (\texttt{TApp})$$

$\Rightarrow$ `a -> b = e -> g, a = e, b = g, c = g`

$\Rightarrow$ `f :: (c -> d) -> b = (g -> g) -> g`



## Exemple de sinteză de tip
### O funcție ne-tipabilă

**Ex Exemplul 3**

```
f x = (x x)
```

$$\frac{\texttt{x :: a} \qquad \texttt{(x x) :: b}}{\texttt{f :: a -> b}}\ (\texttt{TLambda})$$

$$\frac{\texttt{x :: c -> d} \qquad \texttt{x :: c}}{\texttt{(x x) :: d}}\ (\texttt{TApp})$$

Ecuația `c -> d = c` nu are soluție ($\nexists$ tipuri recursive)
$\Rightarrow$ funcția nu poate fi tipată.



## Unificare
### Definiție

- 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.



## Unificare
### Condiții

- 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 -> c` dar nu cu `a -> 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.

## Unificare
### Exemplu

**Exemplu**

- Pentru a unifica expresiile de tip:
  - `t1 = (a, [b])`
  - `t2 = (Int, c)`
- putem avea substituțiile (variante):
  - `S1 = {a ← Int, b ← Int, c ← [Int]}`
  - `S2 = {a ← Int, c ← [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`.



## Tip principal
### Exemplu și definiție

**Exemplu**

- Tipurile: `t1 = (a, [b])` , `t2 = (Int, c)`
  - MGU: `S = {a ← Int, c ← [b]}`
  - Tipuri mai particulare (instanțe): `(Integer, [Integer])`, `(Integer, [Char])`, etc
- Funcția: `\x -> x`
  - Tipuri corecte: `Int -> Int` , `Bool -> Bool` , `a -> a`

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



## Sfârșitul cursului 7
### Elemente esențiale

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

**+** Dați feedback la acest curs aici:
[http://goo.gl/forms/SjDsW06v5J]



## Cursul 8: Clase în Haskell





## Motivație



## Motivație
### Exemplu

**Exemplu**

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

```
show 3 → "3"
show True → "True"
show 'a' → "'a'"
show "a" → "\"a\""
```



## Motivație
### Varianta 1 – Funcții dedicate fiecărui tip

```
showBool True   =   "True"
showBool False  =   "False"

showChar c      =   "'" ++ [c] ++ "'"

showString s    =   "\"" ++ s ++ "\""
```



## Motivație
### Varianta 1 – Funcții dedicate – discuție

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

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

- `showNewLine` nu poate fi polimorfică ⇒ avem nevoie de `showNewLineBool`, `showNewLineChar` etc.
- Alternativ, trimiterea ca parametru a funcției `show*` corespunzătoare:

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

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

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

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

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

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

```
instance Show Bool where
    show True  = "True"
    show False = "False"

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

⇒ Funcția `showNewLine` polimorfică!

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



## Motivație
### Varianta 2 – Supraîncărcare – discuție (1)

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

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

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

- Context: constrângeri suplimentare asupra variabilelor din tipul funcției: $\underbrace{\texttt{Show a =>}}_{\textit{context}}$

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



## Motivație
### Varianta 2 – Supraîncărcare – discuție (2)

- Contexte utilizabile și la instanțiere:

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

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



# Clase Haskell



## Clase Haskell vs. Clase în POO

| Haskell | POO (e.g. Java) |
|---|---|
| Clasele sunt mulțimi de tipuri (superclase); | Clasele sunt mulțimi de obiecte (tipuri); interfețele sunt mulțimi de tipuri; |
| Instanțierea claselor de către tipuri; | Implementarea interfețelor de către clase; |
| Operațiile specifice clasei sunt implementate în cadrul declarației de instanțiere (în afara definiției tipului). | Operațiile specifice interfeței sunt implementate în cadrul definiției tipului (clasei). |



## Clase și instanțe
### Definiții

+ **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.



## Clase predefinite
### `Show`, `Eq`

```
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.



## Clase predefinite
### `Ord`

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

- contextele – utilzabile 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.

## Utilizarea claselor predefinite
### Pentru tipuri de date noi

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

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

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



## Aplicații ale claselor



## `invert`
### Problemă

Ex `invert`

Fie constructorii de tip:

```
data Pair a = P a a

data NestedList a
    =   Atom a
    |   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

```
class Invertible a where
    invert  ::  a -> a
    invert  =   id

instance Invertible (Pair a) where
    invert (P x y) = P y x

instance Invertible a => Invertible (NestedList a) where
    invert (Atom x) =   Atom (invert x)
    invert (List x) =   List $ reverse $ map invert x

instance Invertible a => Invertible [a] where
    invert lst = reverse $ map invert lst
instance Invertible Int ...
```

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



## `contents`
### Problema

Ex `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.

```
class Container a where
    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

```
class Container a where
    contents :: a -> [a]

instance Container [x] where
    contents = id
```

Testăm pentru `contents [1,2,3]`:

- Conform definiției clasei:

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

- Conform supraîncărcării funcției (`id`):

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

- Ecuația `[a] = [[a]]` nu are soluție ⇒ eroare.



## `contents`
### Varianta 1b

```
class Container a where
    contents :: a -> [b]

instance Container [x] where
    contents = id
```

Testăm pentru `contents [1,2,3]`:

- Conform definiției clasei:

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

- Conform supraîncărcării funcției (`id`):

```
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]` ⇒ eroare!



## `contents`
### Varianta 2

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

```
class Container t where
    contents :: t a -> [a]

instance Container Pair where
    contents (P x y) = [x, y]

instance Container NestedList where
    contents (Atom x)   =   [x]
    contents (Seq x)    =   concatMap contents x

instance Container [] where contents = id
```

## Contexte
### Câteva exemple

```haskell
fun1          ::  Eq a => a -> a -> a -> a
fun1 x y z    =   if x == y then x else z

fun2          ::  (Container a, Invertible (a b),
                   Eq (a b)) => (a b) -> (a b) -> [b]
fun2 x y      =   if (invert x) == (invert y)
                      then contents x
                      else contents y

fun3          ::  Invertible a => [a] -> [a] -> [a]
fun3 x y      =   (invert x) ++ (invert y)

fun4          ::  Ord a => a -> a -> a -> a
fun4 x y z    =   if x == y then z else
                      if x > y then x else y
```



## Contexte
### Observații

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



## Sfârșitul cursului 8
### Elemente esențiale

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

+ Dați feedback la acest curs aici:
[http://goo.gl/forms/SjDsW06v5J]



## Cursul 9: Concluzie – Paradigma Funcțională
λPP





## Caracteristici ale paradigmelor de programare



## Paradigma de programare
### Impact în scrierea unui program

λPP

- 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!



## Paradigma de programare
### Legătura cu mașina de calcul

λPP

- paradigmele sunt legate teoretic de o mașină de calcul în care prelucrările caracteristice paradigmei se fac la nivelul mașinii;
- dar putem executa orice program, scris în orice paradigmă, pe orice mașină.



## Paradigma de programare
### Ce o definește

λPP

· 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 a 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, transparență referențială, modul de transfer al parametrilor pentru elementele de prelucrare a datelor.

## Variabile și valori de prim rang



## Variabile
### Nume date unor valori



- în majoritatea limbajelor există variabile, ca NUME date unor valori – rezultatul anumitor procesări (calcule, 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).



## Funcții ca valori de prim rang
### Definiție



+ **Valoare de prim rang** – O valoare care poate fi:
  - creată dinamic
  - stocată într-o variabilă
  - trimisă ca parametru unei funcții
  - întoarsă dintr-o funcție

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



## Funcții ca valori de prim rang: `Compose`
### C



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

- 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



## Funcții ca valori de prim rang:
### Java



```
abstract class Func<U, V> {
    public abstract V apply(U u);

    public <T> Func<T, V> compose(final Func<T, U> f) {
        final Func<U, V> outer = this;

        return new Func<T, V>() {
            public V apply(T t) {
                return outer.apply(f.apply(t));
            }
        };
    }
}
```

- 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.



## Funcții ca valori de prim rang: `Compose`
### Racket & Haskell



- Racket:

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))
```

- Haskell:

```
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.



## Legarea variabilelor



## Legarea variabilelor
### Impactul asupra programului



· două posibilități esențiale:

- un nume este întotdeauna legat (într-un anumit context) la aceeași valoare / la același calcul ⇒ numele stă pentru un calcul;
  - legare statică.
- un nume poate fi legat la mai multe valori pe parcursul execuției ⇒ numele stă pentru un spațiu de stocare – fiecare element de stocare fiind identificat printr-un nume;
  - legare dinamică.

## Efecte laterale (*side effects*)
### Definiție

λPP

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!



## Efecte laterale (*side effects*)
### Consecințe

λPP

Ex În expresia `x-- + ++x`, cu `x = 0`:

- evaluarea stânga → dreapta produce `0 + 0 = 0`
- evaluarea dreapta → 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.



## Efecte laterale (*side effects*)
### Consecințe asupra programării leneșe

λPP

- 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
### Pentru expresii

λPP

**+ Transparență referențială** Confundarea unui obiect ("valoare") cu referința la acesta.

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

Ex Exemplu

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



## Transparență referențială
### Pentru funcții

λPP

+ Funcție transparentă referențial: rezultatul întors depinde exclusiv de parametri.

Ex 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.



## Transparență referențială
### Avantaje

λPP

- 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



## Evaluare
### Mod de evaluare și execuția programelor

λPP

- 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.

## Transferul 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*



## *Call by value*
### În evaluarea aplicativă



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



## *Call by sharing*
### În evaluarea aplicativă



- 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;



## *Call by reference*
### În evaluarea aplicativă



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



## *Call by name*
### În evaluarea normală



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



## *Call by need*
### În evaluarea normală



- 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.



## Sfârșitul cursului 9
### Elemente esențiale



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

+ Dați feedback la acest curs aici:
[http://goo.gl/forms/SjDsW06v5J]



## Cursul 10: Prolog și logica cu predicate de ordinul I

Introducere în Prolog



## Prolog
### Limbaj de programare logică

- introdus în anii 1970 ;
- programul → 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]



## Prolog
### Caracteristici

- 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.



## Logică

- 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.



## Programare logică

- program scris folosind propoziții logice (clauze Horn pentru Prolog);
- mediul de execuție poate folosi propozițiile pentru a demonstra teoreme sau pentru a deduce fapte.
- pentru a înțelege cum funcționează programele scrise într-un limbaj de programare logică trebuie să înțelegem
  - ce sunt propozițiile, ce înseamnă și cum pot fi ele reprezentate;
  - cum funcționează procesele teoretice pe care se bazează mediul de execuție.



Logica propozițională



## Logica propozițională $P\vee\overline{P}$
### Context și elemente principale

- 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.



## Logica propozițională $P\vee\overline{P}$
### Sintaxă

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

## Logica propozițională
### Semantică
$P \vee \overline{P}$

- 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.



## Semantică
### Interpretare
$P \vee \overline{P}$

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

Exemplu

| Interpretarea $I$: | Interpretarea $J$: |
|---|---|
| $p^I = false$ | $p^J = true$ |
| $q^I = true$ | $q^J = true$ |
| $r^I = false$ | $r^J = true$ |

- 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.



## Semantică
### Propoziții compuse (1)
$P \vee \overline{P}$

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



## Semantică
### Propoziții compuse (2)
$P \vee \overline{P}$

- 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
### Cum determinăm valoarea de adevăr?
$P \vee \overline{P}$

**+ 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: $\phi = (p \wedge q) \vee (q \Rightarrow r)$
  $\phi^I = (false \wedge true) \vee (true \Rightarrow false) = false \vee false = false$



## Valoarea de adevăr în afara interpretării
### Satisfiabilitate, Valididate, Nesatisfiabilitate
$P \vee \overline{P}$

**+ 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.

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

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

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



## Valoarea de adevăr în afara interpretării
### Metoda tabelei de adevăr
$P \vee \overline{P}$

Exemplu Metoda tabelei de adevăr

| $p$ | $q$ | $r$ | $(p \wedge q) \vee (q \Rightarrow r)$ |
|---|---|---|---|
| true | true | true | true |
| true | true | false | true |
| true | false | true | true |
| true | false | false | true |
| false | true | true | true |
| false | true | false | false |
| false | false | true | false |
| false | false | false | false |

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

## Derivabilitate
### Definiție

$P \vee \overline{P}$

**+ 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

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



## Derivabilitate
### Verificare

$P \vee \overline{P}$

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

Exemplu

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

| $p$ | $q$ | $p \Rightarrow q$ |
|---|---|---|
| *true* | *true* | *true* |
| *true* | *false* | *false* |
| *false* | *true* | *true* |
| *false* | *false* | *true* |

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



## Derivabilitate
### Formulări echivalente

$P \vee \overline{P}$

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

*sau*

- Propoziția $\phi_1 \wedge \ldots \wedge \phi_n \Rightarrow \phi$ este validă

*sau*

- Propoziția $\phi_1 \wedge \ldots \wedge \phi_n \wedge \neg\phi$ este nesatisfiabilă



## Inferența
### Motivație

$P \vee \overline{P}$

- Creșterea exponențială a numărului de interpretări în raport cu numărul de propoziții simple.
- De aici, diminuarea valorii practice a metodelor semantice, precum cea a tabelei 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
### Definiție

$P \vee \overline{P}$

**+ 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 : $$\frac{\alpha \Rightarrow \beta \quad \neg\beta}{\neg\alpha}$$



## Inferența
### Proprietăți ale regulilor

$P \vee \overline{P}$

**+ 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”.
- Incompletitudinea regulii *Modus Ponens*, din imposibilitatea scrierii oricărei propoziții ca implicație.



# Rezoluția



## Rezoluție
### O regulă de inferență completă și consistentă

$P \vee \overline{P}$

- 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:
  - propoziție = mulțime de clauze (semnificație conjunctivă)
  - clauză = mulțime de literali (semnificație disjunctivă)
  - literal = atom sau atom negat
  - atom = propoziție simplă

## Forma clauzală
### Definiții


**+ Literal** – Propoziție simplă sau negația ei.

Ex: $p$ și $\neg p$.

**+ Expresie clauzală** – Literal sau disjuncție de literali.

Ex: $p \vee \neg q \vee r$.

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

Ex: $\{p, \neg q, r\}$.

**+ Forma clauzală – CNF** – Reprezentarea unei propoziții sub forma unei mulțimi de clauze, implicit legate prin conjuncții.



## Forma clauzală
### Exemplu


Ex: FNC

Forma clauzală a propoziției

$p \wedge (\neg q \vee r) \wedge (\neg p \vee \neg r)$

este

$\{p\}, \{\neg q, r\}, \{\neg p, \neg r\}$.



## Forma clauzală
### Obținere


- Orice propoziție convertibilă în această formă astfel:

1. Eliminarea implicațiilor:
   $\alpha \Rightarrow \beta \;\rightarrow\; \neg\alpha \vee \beta$
2. Avansarea negațiilor până la literali:
   $\neg(\alpha \wedge \beta) \;\rightarrow\; \neg\alpha \vee \neg\beta,$
   $\neg(\alpha \vee \beta) \;\rightarrow\; \neg\alpha \wedge \neg\beta,$
   $\neg(\neg\alpha) \;\rightarrow\; \alpha$
3. Distribuirea lui $\vee$ față de $\wedge$:
   $\alpha \vee (\beta \wedge \gamma) \;\rightarrow\; (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$
4. Transformarea expresiilor în clauze:
   $\phi_1 \vee \ldots \vee \phi_n \;\rightarrow\; \{\phi_1, \ldots, \phi_n\}$
   $\phi_1 \wedge \ldots \wedge \phi_n \;\rightarrow\; \{\phi_1\}, \ldots, \{\phi_n\}$



## Forma clauzală
### Obținere – Exemplu


Ex: Exemplu

Transformăm propoziția $p \wedge (q \Rightarrow r)$ în formă clauzală.

1. $p \wedge (\neg q \vee r)$ Eliminare implicații
2. $\{p\}, \{\neg q, r\}$ Formare clauze

Ex: Exemplu

Transformăm propoziția $\neg(p \wedge (q \Rightarrow r))$ în formă clauzală.

1. $\neg(p \wedge (\neg q \vee r))$ Eliminare implicații
2. $\neg p \vee \neg(\neg q \vee r)$ Împinge negații (1)
3. $\neg p \vee (q \wedge \neg r)$ Împinge negații (2,3)
4. $(\neg p \vee q) \wedge (\neg p \vee \neg r)$ Distribuire $\vee$
5. $\{\neg p, q\}, \{\neg p, \neg r\}$ Formare clauze



## Rezoluție
### Principiu de bază → pasul de rezoluție


Ideea :

$$\frac{\begin{array}{l}\{p \Rightarrow q\}\\ \{\neg p \Rightarrow r\}\end{array}}{\{q, r\}}$$

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

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



## Rezoluție
### Cazuri speciale


- Clauza vidă → indicator de contradicție între premise

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

- Mai mult de 2 rezolvenți posibili → se alege doar unul:

$$\frac{\begin{array}{l}\{p, q\}\\ \{\neg p, \neg q\}\end{array}}{\begin{array}{l}\{p, \neg p\} \textit{ sau}\\ \{q, \neg q\}\end{array}}$$



## Rezoluție
### Alte reguli de inferență → cazuri particulare ale rezoluției


- *Modus Ponens*:

$$\frac{\begin{array}{l}p \Rightarrow q\\ p\end{array}}{q} \quad \sim \quad \frac{\begin{array}{l}\{\neg p, q\}\\ \{p\}\end{array}}{\{q\}}$$

- *Modus Tollens*

$$\frac{\begin{array}{l}p \Rightarrow q\\ \neg q\end{array}}{\neg p} \quad \sim \quad \frac{\begin{array}{l}\{\neg p, q\}\\ \{\neg q\}\end{array}}{\{\neg p\}}$$

- Tranzitivitatea implicației:

$$\frac{\begin{array}{l}p \Rightarrow q\\ q \Rightarrow r\end{array}}{p \Rightarrow r} \quad \sim \quad \frac{\begin{array}{l}\{\neg p, q\}\\ \{\neg q, r\}\end{array}}{\{\neg p, r\}}$$



## Rezoluție
### Demonstrare


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

## Rezoluție
### Demonstrare – algoritm

$P \vee \overline{P}$

0. Am premisele $\phi_1, \ldots, \phi_n$ și concluzia dorită $\phi$
1. Transform $\phi_1, \ldots, \phi_n$ și $\neg\phi$ în FNC
   $\Rightarrow$ mulțime de clauze $\phi_1, \ldots, \phi_n, \neg\phi$
2. Aleg două clauze și aplic pasul de rezoluție
3. **Dacă** rezultatul pasului de rezoluție este clauza vidă ($\emptyset$)
4. **atunci** am terminat demonstrația cu succes
5. **altfel** merg la pasul 2



## Rezoluție
### Exemplu

$P \vee \overline{P}$

Exemplu

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.

| | | |
|---|---|---|
| 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 |



## Rezoluție
### Consistență și completitudine

$P \vee \overline{P}$

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.



## Sfârșitul cursului 10
### Elemente esențiale

$P \vee \overline{P}$

- Introducere în Prolog
- Bazele logicii propoziționale, Sintaxă și Semantică
- Inferență, Rezoluție, Forme normale

+ Dați feedback la acest curs aici:
[http://goo.gl/forms/SjDsW06v5J]



## Cursul 11: Logica cu predicate de ordinul I

$P \vee \overline{P}$





## Introducere



## Logica cu predicate de ordinul I
### *First Order Predicate Logic* (FOL sau FOPL) – Context

$P \vee \overline{P}$

- 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."
  - $p \Leftrightarrow q$
  
  $\rightarrow$ Opacitate în raport cu obiectele și relațiile referite.
- FOL:
  - Generalizare: $prieten(x, y)$: "$x$ este prieten cu $y$."
  - $\forall x.\forall y.(prieten(x, y) \Leftrightarrow prieten(y, x))$
  
  $\rightarrow$ Aplicare pe cazuri particulare.
  
  $\rightarrow$ Transparență în raport cu obiectele și relațiile referite.



## Sintaxă

## Sintaxă
### Simboluri utilizate

$P \vee \overline{P}$

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



## Sintaxă
### Termeni

$P \vee \overline{P}$

**+ Termeni** (obiecte):

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

**Ex Exemple**

- $succesor(4)$: succesorul lui 4, și anume 5.
- $+(2, x)$: aplicația funcției de adunare asupra numerelor 2 și *x*, și, totodată, suma lor.



## Sintaxă
### Atomi

$P \vee \overline{P}$

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

**Ex Exemple**

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



## Sintaxă
### Propoziții

$P \vee \overline{P}$

**+ 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: $\alpha \wedge \beta$, $\alpha \Rightarrow \beta$, ...
- Cuantificări: $\forall x.\alpha$, $\exists x.\alpha$



## Sintaxă
### Exemplu

$P \vee \overline{P}$

**Ex Exemplu**

"Sora Ioanei are un prieten deștept"

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



# Semantică



## Semantică
### Interpretare

$P \vee \overline{P}$

**+ 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 $f^I : D^n \to D$
- Pentru fiecare predicat *n*-ar *p*, o funcție $p^I : D^n \to \{false, true\}$.



## Semantică
### Elemente

$P \vee \overline{P}$

- Atom: $(p(t_1, \ldots, t_n))^I = p^I(t_1^I, \ldots, t_n^I)$
- 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: $(\exists x.\alpha)^I = \begin{cases} true & \text{dacă } \exists d \in D \,.\, \alpha^I_{[d/x]} = true \\ false & \text{altfel} \end{cases}$

## Semantică
### Cuantificatori

**Exemple cu cuantificatori**

1. "Vrabia mălai visează."
   $\forall x.(vrabie(x) \Rightarrow viseaza(x, malai))$
2. "Unele vrăbii visează mălai."
   $\exists x.(vrabie(x) \wedge viseaza(x, malai))$
3. "Nu toate vrăbiile visează mălai."
   $\exists x.(vrabie(x) \wedge \neg viseaza(x, malai))$
4. "Nicio vrabie nu visează mălai."
   $\forall x.(vrabie(x) \Rightarrow \neg viseaza(x, malai))$
5. "Numai vrăbiile visează mălai."
   $\forall x.(viseaza(x, malai) \Rightarrow vrabie(x))$





## Cuantificatori
### Greșeli frecvente

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





## Cuantificatori
### Proprietăți

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





## Aspecte legate de propoziții
### Analoage logicii propoziționale

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



# Forme normale





## Forme normale
### Definiții

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

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

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

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

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

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





## Forme normale
### Clauze Horn

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

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





## Conversia propozițiilor în FNC (1)
### Eliminare implicații, împingere negații, redenumiri

1. Eliminarea implicațiilor ($\not\Rightarrow$)
2. Împingerea negațiilor până în fața literalilor ($\vec{\neg}$)
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):
   $$\forall x.p(x) \wedge \forall y.q(y) \vee \exists z.r(z) \rightarrow \forall x.\forall y.\exists z.(p(x) \wedge q(y) \vee r(z))$$

## Conversia propozițiilor în FNC (2)
### Skolemizare

$P \vee \overline{P}$

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

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

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

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



## Conversia propozițiilor în FNC (3)
### Cuantificatori universali, Distribuire ∨, Clauze

$P \vee \overline{P}$

6. Eliminarea cuantificatorilor universali, considerați, acum, impliciți ($\cancel{\forall}$):

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

7. Distribuirea lui ∨ față de ∧ (∨/∧):

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

8. Transformarea expresiilor în clauze (C).



## Conversia propozițiilor în FNC – Exemplu

$P \vee \overline{P}$

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

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

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



# Unificare și rezoluție



## Unificare

$P \vee \overline{P}$

- Utilizată pentru rezoluție
- vezi și sinteza de tip în Haskell

cum știm dacă folosind ipoteza $om(Marcel)$ și propoziția $\forall om(x) \Rightarrow are\_inima(x)$ putem demonstra că $are\_inima(Marcel)$ → 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 care nu conține variabila (*x* cu *Marcel*)



## Unificare
### Observații

$P \vee \overline{P}$

- 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*);



## Unificare
### Rolul în rezoluție

$P \vee \overline{P}$

- Rezoluția pentru clauze Horn:

$$\frac{\begin{array}{l} A_1 \wedge \ldots \wedge A_m \Rightarrow A \\ B_1 \wedge \ldots \wedge A' \wedge \ldots \wedge B_n \Rightarrow B \\ unificare(A, A') = S \end{array}}{subst(S, A_1 \wedge \ldots \wedge A_m \wedge B_1 \wedge \ldots \wedge B_n \Rightarrow B)}$$

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



## Rezoluție
### Exemplu

$P \vee \overline{P}$

Ex 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?

# Rezoluție

## Exemplu Horses and Hounds

$P \vee \overline{P}$

1. $\forall x.\forall y.horse(x) \wedge dog(y) \Rightarrow faster(x,y)$
   $\rightarrow \neg horse(x) \vee \neg dog(y) \vee faster(x,y)$
2. $\exists x.greyhound(x) \wedge (\forall y.rabbit(y) \Rightarrow faster(x,y))$
   $\rightarrow greyhound(Greg) \quad ; \quad \neg rabbit(y) \vee faster(Greg,y)$
3. $horse(Harry) \quad ; \quad rabbit(Ralph)$
4. $\neg faster(Harry, Ralph)$ (concluzia negată)
5. $\neg greyhound(x) \vee dog(x)$ (common knowledge)
6. $\neg faster(x,y) \vee \neg faster(y,z) \vee faster(x,z)$ (tranzitivitate)

7. 1 + 3a $\rightarrow \neg dog(y) \vee faster(Harry,y)$ (cu $\{Harry/x\}$)
8. 2a + 5 $\rightarrow dog(Greg)$ (cu $\{Greg/x\}$)
9. 7 + 8 $\rightarrow faster(Harry, Greg)$ (cu $\{Greg/y\}$)
10. 2b + 3b $\rightarrow faster(Greg, Ralph)$ (cu $\{Ralph/y\}$)
11. 6 + 9 + 10 $\rightarrow faster(Harry, Ralph)$ $\{Harry/x, Greg/y, Ralph/z\}$
12. 11 + 4 $\rightarrow \square$ q.e.d.



# Sfârșitul cursului 11(a)

## Ce am învățat

$P \vee \overline{P}$

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

+ Dați feedback la acest curs aici:
[http://goo.gl/forms/SjDsW06v5J]



# Cursul 12: Programare logică în Prolog





# Procesul de demonstrare



# Pași în demonstrare (1)

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.



# Pași în demonstrare (2)

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.



# Un exemplu de program Prolog

Exemplu

```
parent(andrei, bogdan).
parent(andrei, bianca).
parent(bogdan, cristi).

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

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



# Exemplul genealogic (1)

$S = \emptyset$
$G = \{gp(X,Y)\}$

$\downarrow$

`gp(X1, Y1) :- p(X1, Z1), p(Z1, Y1)`

$\downarrow$

$S = \{X = X1, Y = Y1\}$
$G = \{p(X1, Z1), p(Z1, Y1)\};$

$\downarrow \qquad \downarrow \qquad \downarrow$

`p(andrei, bogdan)` `p(andrei, bianca)` `p(bogdan, cristi)`

$\downarrow \qquad \downarrow \qquad \downarrow$

$\cdots 1 \qquad \cdots 2 \qquad \cdots 3$

## Exemplul genealogic (2)
### Ramura 1

$\cdots 1$

$\downarrow$

`p(andrei, bogdan)`

$\downarrow$

$S = \{X = X1, Y = Y1, X1 = andrei, Z1 = bogdan\}$
$G = \{p(bogdan, Y1)\};$

$\downarrow$

`p(bogdan, cristi)`

$\downarrow$

$S = \{X = X1, Y = Y1, X1 = andrei, Z1 = bogdan, Y1 = cristi\}$
$G = \emptyset;$

$\downarrow$

success

`gp(andrei, cristi)`



## Exemplul genealogic (3)
### Ramura 2

$\cdots 2$

$\downarrow$

`p(andrei, bianca)`

$\downarrow$

$S = \{X = X1, Y = Y1, X1 = andrei, Z1 = bianca\}$
$G = \{p(bianca, Y1)\};$

$\downarrow$

eșec



## Exemplul genealogic (4)
### Ramura 3

$\cdots 3$

$\downarrow$

`p(bogdan, cristi)`

$\downarrow$

$S = \{X = X1, Y = Y1, X1 = bogdan, Z1 = cristi\}$
$G = \{p(cristi, Y1)\};$

$\downarrow$

eșec



## Observații

- 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.



## Strategii de control
### Ale demonstrațiilor

*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 I
### Algoritm Backward chaining

**BackwardChaining**(*rules*, *goals*, *subst*)
  lista regulilor din program, stiva de scopuri, substituția curentă, inițial vidă.
  **returns** satisfiabilitatea scopurilor
**if** $goals = \emptyset$ **then**
  **return** SUCCESS
$goal \leftarrow head(goals)$
$goals \leftarrow tail(goals)$
**for-each** $rule \in rules$ **do**    // în ordinea din program
  **if** $unify(goal, conclusion(rule), subst) \rightarrow bindings$
    $newGoals \leftarrow premises(rule) \cup goals$    // adâncime
    $newSubst \leftarrow subst \cup bindings$
    **if** $BackwardChaining(rules, newGoals, newSubst)$
      **then return** SUCCESS
**return** FAILURE



## Controlul execuției



## Exemplu – Minimul a două numere
### Cod Prolog

Ex Minimul a două numere

```
min(X, Y, M) :- X =< Y, M is X.
min(X, Y, M) :- X > Y,  M is Y.

min2(X, Y, M) :- X =< Y, M = X.
min2(X, Y, M) :- X > Y,  M = Y.

% Echivalent cu min2.
min3(X, Y, X) :- X =< Y.
min3(X, Y, Y) :- X > Y.
```

## Exemplu – Minimul a două numere
### Utilizare

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

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

?- min2(1+2, 3+4, M).
M = 1+2 ;
false.

?- min2(3+4, 1+2, M).
M = 1+2.
```



## Exemplu – Minimul a două numere
### Observații

- Condiții mutual exclusive: `X =< Y` și `X > Y` → cum putem elimina redundanța?

Exemplu

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

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

- Greșit!



## Exemplu – Minimul a două numere
### Îmbunătățire

- Soluție: oprirea recursivității după prima satisfacere a scopului.

Exemplu

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

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



## Operatorul *cut*
### Definiție

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



## Operatorul *cut*
### Exemplu

Exemplu

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

boy(john).
boy(bill).

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

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



## Operatorul *cut*
### Utilizare

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

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



## Negația ca eșec

Exemplu

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



## Sfârșitul cursului 12
### Elemente esențiale

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

+ Dați feedback la acest curs aici:
[http://goo.gl/forms/SjDsW06v5J]

# Cursul 13: Mașina algoritmică Markov





## Introducere



## Mașina algoritmică Markov

- Model de calculabilitate efectivă, echivalent cu Mașina Turing și Calculul Lambda;
- Principiul de funcționare: *pattern matching* + substituție;
- Fundamentul teoretic al paradigmei asociative și al limbajelor bazate pe reguli (de forma *dacă-atunci*).



## Paradigma asociativă
### Caracteristici

- Potrivită mai ales în cazul problemelor ce nu admit o soluție precisă algoritmică (ieftină);
- Codificarea cunoștințelor specifice unui domeniu și aplicarea lor într-o manieră euristică;
- Descrierea proprietăților soluției, prin contrast cu pașii care trebuie realizați pentru obținerea acesteia (*ce* trebuie obținut vs. *cum*);
- Absența unui flux explicit de control, deciziile fiind determinate, implicit, de cunoștințele valabile la un anumit moment → *data-driven control*.



## Mașina algoritmică Markov



## Mașina algoritmică Markov
### Exemple de implementare

(implementări fără variabile generice)

- online: [http://utilitymill.com/utility/markov_rewriter]
- Windows: [http://yad-studio.github.io/]
- mai multe: [http://en.wikipedia.org/wiki/Markov_algorithm#External_links]



## Structura Mașinii Markov
### Perspectivă generală

RD [ S ]
↑
[ UC ]
↑
SA [ AM ]

- Registrul de date, RD, cu secvența de simboluri, S
  - RD nemărginit la dreapta
  - $S \in (A_b \cup A_l)^*, A_b \cap A_l = \emptyset$ – alfabet de bază și de lucru
- Unitatea de control, UC
- Spațiul de stocare a algoritmului, SA, ce conține algoritmul Markov, AM
  - format din reguli.



## Structura Mașinii Markov
### Reguli

- Unitatea de bază a unui algoritm Markov → regula asociativă de substituție:

  `șablon identificare (LHS) → șablon substituție (RHS)`

- Exemplu: `ag₁c -> ac`
- șabloanele → secvențe de simboluri:
  - constante: simboluri din $A_b$
  - variabile locale: simboluri din $A_l$
  - variabile generice: simboluri speciale, din mulțimea $G$, legați la simboluri din $A_b$
- Dacă `RHS` este "." → regulă terminală, ce încheie execuția mașinii (halt).

## Structura Mașinii Markov
### Variabile generice

- De obicei, notate cu `g`, urmat de un indice;
- Mulțimea valorilor pe care le poate lua o variabilă → domeniul variabilei – $\texttt{Dom(g)} \subseteq A_b \cup A_l$;
- Legate la exact un simbol la un moment dat;
- Durata de viață → timpul aplicării regulii – sunt legate la identificarea șablonului și legarea se pierde după înlocuirea șablonului de identificare cu cel de substituție;
- Utilizabile în `RHS` doar în cazul apariției în `LHS`.



## Structura Mașinii Markov
### Algoritm Markov

- Mulțime ordonată de reguli, îmbogățite cu declarații:
  - de partiționare a mulțimii $A_b$
  - de variabile generice

Ex | Exemplu Eliminarea din dintr-un șir de simboluri din mulțimea $A \cup B$ simbolurilor ce aparțin mulțimii $B$:

```
setDiff1(A, B); A g1; B g2;
    ag2 -> a;
    ag1 -> g1a;
    a -> .;
    -> a;
end
```

```
setDiff2(A, B); B g2;
    g2 -> ;
    -> .;
end
```

- $\texttt{A,B} \subseteq A_b$
- $g_1,\ g_2$ → variabile generice
- `a` nedeclarată → variabilă locală ($\texttt{a} \in A_l$)



## Reguli
### Aplicabilitate

**+ | Aplicabilitatea unei reguli** Regula $r : a_1 \ldots a_n \rightarrow b_1 \ldots b_m$ este aplicabilă dacă și numai dacă există un subșir $c_1 \ldots c_n$, în RD, astfel încât $\forall i = \overline{1,n}$ exact 1 condiție din cele de mai jos este îndeplinită:

- $a_i \in A_b \cup A_l \ \wedge \ a_i = c_i$
- $a_i \in G \ \wedge \ c_i \in \mathrm{Dom}(a_i) \ \wedge \ (\forall j = \overline{1,n}\ .\ \ a_j = a_i \ \Rightarrow \ c_j = c_i)$,
- oriunde mai apare aceeași variabilă generică în șablonul de identificare, în poziția corespunzătoare din subșir avem același simbol.



## Reguli
### Aplicare

**+ | Aplicarea regulii**
$r : a_1 \ldots a_n \rightarrow b_1 \ldots b_m$ asupra unui subșir $s : c_1 \ldots c_n$, în raport cu care este aplicabilă, constă în substituirea lui $s$ prin subșirul $q_1 \ldots q_m$, calculat astfel încât pentru $\forall i = \overline{1,n}$:

- $b_i \in A_b \cup A_l \ \Rightarrow \ q_i = b_i$
- $b_i \in G \ \wedge \ (\exists j = \overline{1,n}\ .\ \ b_i = a_j) \ \Rightarrow \ q_i = c_j$



## Reguli
### Exemplu de aplicare

Ex | Exemplu

- $A_b = \{1,\ 2,\ 3\}$
- $A_l = \{x,\ y\}$
- $\mathrm{Dom}(g_1) = \{2\}$
- $\mathrm{Dom}(g_2) = A_b$
- S = 1111112x2y31111
- $r : 1g_1xg_1yg_2 \rightarrow 1g_2x$

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| S | = | 11111 | 1 | 2 | x | 2 | y | 3 | 1111 |
| r | : | | 1 | $g_1$ | x | $g_1$ | y | $g_2$ | $\rightarrow 1g_2x$ |
| S' | = | 1111113x1111 | | | | | | | |

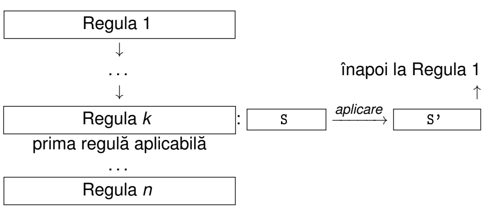


## Unitatea de control
### Aplicabilitate vs. aplicare

- Cazuri speciale: aplicabilitatea:
  - unei reguli pentru mai multe subșiruri;
  - mai multor reguli pentru același subșir.
- La un anumit moment, este posibilă aplicarea propriu-zisă a unei singure reguli asupra unui singur subșir;
- Nedeterminism inerent, ce trebuie exploatat, sau rezolvat;
- Convenție care poate fi făcută:
  - aplicarea primei reguli aplicabile, asupra
  - celui mai din stânga subșir asupra căreia este aplicabilă



## Unitatea de control
### Funcționare






## Exemplu
### Ex | Inversarea intrării

- Ideea: mutarea, pe rând, a fiecărui element în poziția corespunzătoare. Mutarea se face prin pași incrementali de interschimbare a elementelor învecinate.

```
Reverse(A); A g1, g2;
    ag1g2 -> g2ag1;
    ag1 -> bg1;
    abg1 -> g1a;
    a -> .;
    -> a;
end
```

- $\texttt{DOP} \xrightarrow{6} \texttt{aDOP} \xrightarrow{2} \texttt{OaDP} \xrightarrow{2} \texttt{OPaD} \xrightarrow{3} \texttt{OPbD} \xrightarrow{6} \texttt{aOPbD} \xrightarrow{2} \texttt{PaObD} \xrightarrow{3} \texttt{PbObD} \xrightarrow{6} \texttt{aPbObD} \xrightarrow{3} \texttt{bPbObD} \xrightarrow{6} \texttt{abPbObD} \xrightarrow{4} \texttt{PabObD} \xrightarrow{4} \texttt{POabD} \xrightarrow{4} \texttt{PODa} \xrightarrow{5} .$

# Aplicații



## CLIPS

- "C Language Integrated Production System";
- Sistem bazat pe reguli → "producție" = regulă;
- Principiu de funcționare similar cu al mașinii Markov;
- Dezvoltat la NASA în anii 1980;



## CLIPS
### Exemplu: Minimul a două numere – reprezentare individuală

Exemplu

```
(deffacts numbers
    (number 1)
    (number 2))

(defrule min
    (number ?m)
    (number ?x)
    (test (< ?m ?x))
  =>
    (assert (min ?m)))
```



## CLIPS
### Fapte

- Reprezentarea datelor prin fapte → similare simbolurilor mașinii Markov;
- Afirmații despre atributele obiectelor;
- Date simbolice, construite conform unor șabloane;
- Mulțimea de fapte → baza de cunoștințe (*factual knowledge base*)

```
> (facts)
f-0     (initial-fact)
f-1     (number 1)
f-2     (number 2)
For a total of 3 facts.
```



## CLIPS
### Reguli

- Similare regulilor mașinii Markov;
- Șablon de identificare → secvență de fapte parametrizate (vezi variabilele generice ale algoritmilor Markov) și restricții;
- Șablon de acțiune → secvență acțiuni (`assert`, `retract`);
- *Pattern matching* secvențial pe faptele din șablonul de identificare;
- Domeniul de vizibilitate a unei variabile → restul regulii, după prima apariție a variabilei, în șablonul de identificare.



## Înregistrări de activare
### Definiție

- Tuplul (regulă, fapte asupra cărora este aplicabilă) → înregistrare de activare (*activation record*);
- Reguli posibil aplicabile asupra diferitelor porțiuni ale acelorași fapte;
- Muțimea înregistrărilor de activare → agenda.



## Înregistrări de activare
### Exemplu – reluat de mai devreme: minimul a 2 numere

```
> (facts)
f-0     (initial-fact)
f-1     (number 1)
f-2     (number 2)
For a total of 3 facts.

> (agenda)
0      min: f-1,f-2
For a total of 1 activation.

> (run)
FIRE    1 min: f-1,f-2
==> f-3     (min 1)
```



## Terminarea programelor

- Principiul refracției:
  - Aplicarea unei reguli o singură dată asupra acelorași fapte și acelorași porțiuni ale acestora;
  - Altfel, programe care nu s-ar termina.
- Terminare:
  - Aplicarea unui număr maxim de reguli → (`run` *n*);
  - Întâlnirea acțiunii `(halt)`;
  - Golirea agendei.

## CLIPS – Exemple

### Minimul a două numere – Reprezentare agregată (1)

Exemplu

```
(deffacts numbers
    (numbers 1 2))

(defrule min
    (numbers $? ?m $?)
    (numbers $? ?x $?)
    (test (< ?m ?x))
  =>
    (assert (min ?m)))
```

- Observați utilizarea `$?` pentru potrivirea unei secvențe, potențial vidă.



## CLIPS – Exemple

### Minimul a două numere – Reprezentare agregată (2)

```
> (facts)
f-0     (initial-fact)
f-1     (numbers 1 2)
For a total of 2 facts.

> (agenda)
0       min: f-1,f-1
For a total of 1 activation.
```



## CLIPS – Exemple

### Suma oricâtor numere (1)

Exemplu

```
(deffacts numbers (numbers 1 2 3 4 5))

(defrule init
    ; implicit, (initial-fact)
  =>
    (assert (sum 0)))

(defrule sum
    ?f <- (sum ?s)
    (numbers $? ?x $?)
  =>
    (retract ?f)
    (assert (sum (+ ?s ?x))))
```



## CLIPS – Exemple

### Suma oricâtor numere (2) – Interogare

```
> (facts)
f-0     (initial-fact)
f-1     (numbers 1 2 3 4 5)
For a total of 2 facts.

> (agenda)
0       init: *
For a total of 1 activation.

> (run 1)
FIRE    1 init: *
==> f-2     (sum 0)
```



## CLIPS – Exemple

### Suma oricâtor numere (3) – Interogare

```
> (agenda)
0       sum: f-2,f-1
0       sum: f-2,f-1
0       sum: f-2,f-1
0       sum: f-2,f-1
0       sum: f-2,f-1
For a total of 5 activations.

> (run)
ciclează!
```



## CLIPS – Exemple

### Suma oricâtor numere (4) – Observații

- Eroarea: adăugarea unui nou fapt `sum` induce aplicabilitatea repetată a regulii, asupra elementelor deja însumate;
- Corect: consultarea primului număr din listă și eliminarea acestuia.



## CLIPS – Exemple

### Suma oricâtor numere (5) – Implementare corectă

Exemplu

```
(deffacts numbers (numbers 1 2 3 4 5))
(defrule init
  =>
    (assert (sum 0)))

(defrule sum
    ?f <- (sum ?s)
    ?g <- (numbers ?x $?rest)
  =>
    (retract ?f)
    (assert (sum (+ ?s ?x)))
    (retract ?g)
    (assert (numbers $?rest)))
```



## CLIPS – Exemple

### Suma oricâtor numere (6) – Interogare pe implementarea corectă

```
> (run)
FIRE    1 init: *
==> f-2     (sum 0)
FIRE    2 sum: f-2,f-1
<== f-2     (sum 0)
==> f-3     (sum 1)
<== f-1     (numbers 1 2 3 4 5)
==> f-4     (numbers 2 3 4 5)
FIRE    3 sum: f-3,f-4
<== f-3     (sum 1)
==> f-5     (sum 3)
<== f-4     (numbers 2 3 4 5)
==> f-6     (numbers 3 4 5)
```

## CLIPS – Exemple
### Suma oricâtor numere (7) – Interogare pe implementarea corectă

```
FIRE    4 sum: f-5,f-6
<== f-5       (sum 3)
==> f-7       (sum 6)
<== f-6       (numbers 3 4 5)
==> f-8       (numbers 4 5)
FIRE    5 sum: f-7,f-8
<== f-7       (sum 6)
==> f-9       (sum 10)
<== f-8       (numbers 4 5)
==> f-10      (numbers 5)
FIRE    6 sum: f-9,f-10
<== f-9       (sum 10)
==> f-11      (sum 15)
<== f-10      (numbers 5)
==> f-12      (numbers)
```



## XSLT
### Transformarea fișierelor XML – Exemplu

Exemplu

```
<?xml version="1.0" ?>
<persons>
  <person username="JS1">
    <name>John</name>
    <family-name>Smith</family-name>
  </person>
  <person username="MI1">
    <name>Morka</name>
    <family-name>Ismincius</family-name>
  </person>
</persons>
```

⇓ XSLT ⇓

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <name username="JS1">John</name>
  <name username="MI1">Morka</name>
</root>
```



## XSLT
### Transformarea fișierelor XML – Exemplu: sursa

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://..." version="1.0">
  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/persons">
    <root>
      <xsl:apply-templates select="person"/>
    </root>
  </xsl:template>

  <xsl:template match="person">
    <name username="{@username}">
      <xsl:value-of select="name" />
    </name>
  </xsl:template>

</xsl:stylesheet>
```



## Sfârșitul cursului 13
### Ce am învățat

- Ce este și cum funcționează mașina algoritmică Markov: structură, variabile, reguli, algoritmul unității de control.
- Introducere în CLIPS – fapte, reguli, execuție.
- Exemplu de fișier XSLT.

+| Nu uitați să dați feedback la curs.