

Paradigme de Programare

S.I. dr. ing. Mihnea Muraru
mmihnea@gmail.com

2015–2016, semestrul 2

Partea I

Introducere

Cuprins

3.33pt

- 1 Organizare
- 2 Obiective
- 3 Exemplu introductiv
- 4 Paradigme și limbaje

Cuprins

3.33pt

- 1 Organizare
- 2 Obiective
- 3 Exemplu introductiv
- 4 Paradigme și limbaje

Notare

- Teste la curs: 0,5
- Test grilă: 0,5
- Laborator: 1
- Teme: 4 (3 × 1,33)
- Examen: 4

Regulament

Vă rugăm să citiți regulamentul cu atenție!

<http://elf.cs.pub.ro/pp/16/regulament>

Desfășurarea cursului

- Recapitularea cursului anterior
- Predare
- Test din cursul anterior
- Feedback despre cursul curent (de acasă)

Cuprins

3.33pt

- 1 Organizare
- 2 Obiective
- 3 Exemplu introductiv
- 4 Paradigme și limbaje

Ce vom studia?

- 1 **Modele de calculabilitate:**
Diverse perspective conceptuale asupra noțiunii de calculabilitate efectivă
- 2 **Paradigme de programare:**
Influența perspectivei alese asupra procesului de modelare și rezolvare a problemelor
- 3 **Limbaje de programare:**
Mecanisme expresive, aferente paradigmelor, cu accent pe aspectul comparativ

De ce?

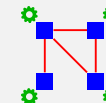
The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

Edsger Dijkstra,
How do we tell truths that might hurt

Descompunerea problemelor

Controlul complexității: descompunere și interfațare

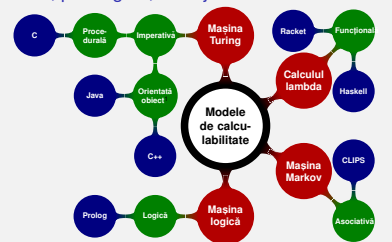
Descompunere	Accent pe	Rezultat
Procedurală	Acțiuni	Proceduri
Orientată obiect	Entități	Clase și obiecte
Funcțională	Relații	Funcții în sens matematic
Logică	Relații	Predicată și propoziții



De ce? (cont.)

- Lărgirea spectrului de abordare a problemelor
- Identificarea perspectivei ce permite modelarea simplă a unei probleme; alegerea limbajului adecvat
- Exploatarea mecanismelor oferite de limbajele de programare (v. Dijkstra!)
- Sporirea capacității de învățare a noi limbaje și de adaptare la particularitățile și diferențele dintre acestea

Modele, paradigme, limbaje



¹Original imperativă, dar se poate combina chiar cu abordarea funcțională

Limitele calculabilității

- Teza Church-Turing:
efectiv calculabil \equiv Turing calculabil
- Echivalența celorlalte modele de calculabilitate, și a multor altora, cu Mașina Turing
- Există vreun model superior ca forță de calcul?

Cuprins

3.33pt

- 1 Organizare
- 2 Obiective
- 3 Exemplu introductiv
- 4 Paradigme și limbaje

Câteva trăsături

- **Tipare**
 - Statică/ dinamică
 - Tare/ slabă
- **Ordinea de evaluare** a parametrilor funcțiilor
 - Aplicativă
 - Normală
- **Legarea variabilelor**
 - Statică
 - Dinamică

32 / 423

Rezumat

Importanța cunoașterii paradigmelor și limbajelor de programare, în scopul identificării celor **convenabile** pentru modelarea unei probleme particulare

33 / 423

Partea II

Limbajul Racket

34 / 423

Cuprins

3.33pt

- 1 Expresii și evaluare
- 2 Liste și perechi
- 3 Tipare
- 4 Omoiconitate și metaprogramare

35 / 423

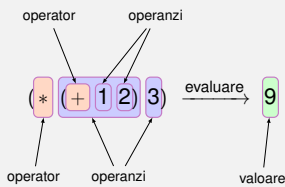
Cuprins

3.33pt

- 1 Expresii și evaluare
- 2 Liste și perechi
- 3 Tipare
- 4 Omoiconitate și metaprogramare

36 / 423

Expresii



37 / 423

Evaluarea expresiilor primitive

- 1 Evaluarea (reducerea) **operanzilor** la valori (argumente)
- 2 Aplicarea **operatorului** primitiv asupra argumentelor

Recursiv pentru subexpresii

```
1 (+ (+ 1 2) 3) → (+ 3 3) → 9
```



38 / 423

Construcția `define`

Scop

```
1 (define WIDTH 100)
```

- **Leagă** o variabilă globală la **valoarea** unei expresii
- **Atenție!** Principal, este vorba de **constante**
- **Avantaje:**
 - Lizibilitate (atribuire de **sens** prin numire)
 - Flexibilitate (modificare într-un **singur** loc)
 - Reutilizare (**evitarea** reproducerii multiple a unei expresii complexe)

39 / 423

Construcția `define`

Evaluare

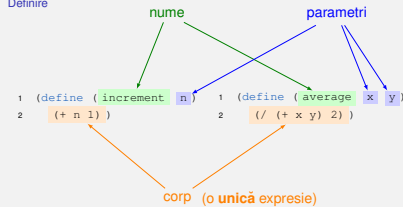
- 1 La **definire**, se evaluează expresia, și se **leagă** variabila la **valoarea** ei
- 2 La **utilizare**, variabila se evaluează la valoarea ei

```
1 (define x (+ (+ 1 2) 3)) ; x ← 9
2 (+ x 10) → (+ 9 10)
```

40 / 423

Funcții

Definire



- Accepție matematică a funcțiilor — **valoare** calculată
- **Absența** informației de tip

41 / 423

Funcții

Evaluare

- Definire:
- 1 Înregistrarea definiției funcției

```
1 (define (increment x) ; increment ← <funcția>
2 (+ x 1))
```

Aplicare:

- 1 Evaluarea (reducerea) **operanzilor** la argumente
- 2 **Substituirea** argumentelor în corpul funcției
- 3 Evaluarea expresiei obținute

```
1 (increment (+ 1 2)) → (increment 3)
2 → (+ 3 1) → 4
```

42 / 423

Construcția `if`

Prezentare

```
1 (if (< 1 2) (+ 3 4) (+ 5 6))
```

- Imaginabilă în forma unei **funcții**
- Ramurile **then** și **else** ca **operanzi**
- De aici, **obligativitatea** prezenței ramurii **else!**

43 / 423

Construcția `if`

Evaluare

- 1 Evaluarea **condiției**
- 2 Înlocuirea **întregii** expresii `if` cu ramura potrivită
- 3 Evaluarea expresiei obținute

Ordine **diferită** de evaluare, față de funcțiile obișnuite!

```
1 (if (< 1 2) (+ 3 4) (+ 5 6))
2 → (if true (+ 3 4) (+ 5 6))
3 → (+ 3 4) → 7
```

44 / 423

Cuprins

3.33pt

- 1 Expresii și evaluare
- 2 Liste și perechi
- 3 Tipare
- 4 Omoiconitate și metaprogramare

45 / 423

Liste

Literali

- 1 Aspectul de listă al **aplicațiilor** operatorilor
`(+ 1 2)`

- 2 Ce s-ar întâmpla dacă am înlocui + cu 0?
`(0 1 2)`

Eroare! 0 nu este operator!

- 3 **Soluție:** **împiedicarea** evaluării, cu `quote`
`(quote (0 1 2))` sau `'(0 1 2)`

46 / 423

Liste

Structură

- 1 **Structură recursivă**
 - O listă se obține prin introducerea unui element (**head**) în vârful altei liste (**tail**)
`(cons 0 '(1 2)) → '(0 1 2)`
 - Cazul de bază: lista vidă, `'()`

- 2 **Alternativă de construcție:** funcția `list`
`(list 0 1 2)`

- 3 **Selectori**
`(car '(0 1 2)) → 0`
`(cdr '(0 1 2)) → '(1 2)`

47 / 423

Liste

Funcții

- Exploatarea structurii **recursive** de funcțiile pe liste
- Exemplu: **minimul** unei liste nevide (v. slide-ul 21)
 - **Axiome**, pornind de la un tip de date abstract *List*, cu constructorii de bază '() și *cons*:
 $(\text{minList } (\text{cons } e ' ())) = e$
 $(\text{minList } (\text{cons } e L)) = (\text{min } e (\text{minList } (\text{cdr } L)))$
- Implementare

```
1 (define (minList1 L)
2   (if (= (length L) 1) (car L)
3       (min (car L) (minList1 (cdr L)))))
```
- Traducere **fidela** a axiomelor unui TDA într-un program funcțional!

48 / 423

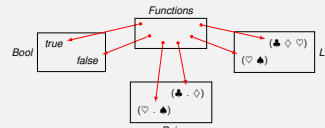
Perechi

- Intern, listă \equiv pereche *head-tail*
- *cons*, aplicabil asupra oricăror doi operanzi, pentru generarea unei perechi cu punct (*dotted pair*)
 $(\text{cons } 0 1) \rightarrow '(0 . 1)$
 $'(0 1 2) \equiv '(0 . (1 . (2 . ())))$
- Toretic, perechi reprezentabile ca **funcții!** (vom vedea mai târziu). De fapt, ...

49 / 423

Universalitatea funcțiilor

- ... orice limbaj prevăzut **exclusiv** cu funcții și **fără** tipuri predefinite este **la fel** de expresiv ca orice alt limbaj (în limitele tezei Church-Turing)
- Majoritatea **tipurilor** uzuale, codificabile direct prin intermediul funcțiilor



50 / 423

Cuprins

3.33pt

- 1 Expresii și evaluare
- 2 Liste și perechi
- 3 **Tipare**
- 4 Omoiconicitate și metaprogramare

51 / 423

Caracteristici

- **Tipare** = modalitatea de definire, manipulare și verificare a tipurilor dintr-un limbaj
- Existența unor tipuri **predefinite** în Racket (boolean, caracter, număr etc.)
- Întrebări:
 - **Când** se realizează verificarea?
 - Cât de **flexibile** sunt regulile de tipare?

52 / 423

Flexibilitatea regulilor

- Ce produce evaluarea următoarei expresii?
 $(+ 1 \text{"OK"})$
- Criteriu: flexibilitatea în agregarea valorilor de tipuri **diferite**
- Racket: verificare **rigidă** — tipare **tare** (*strong*)
- Răspuns: eroare!
- Alternativă în alte limbaje — tipare **slabă** (*weak*)
 - Visual Basic: $1 + "23" = 24$
 - JavaScript: $1 + "23" = "123"$

53 / 423

Momentul verificării

- Ce produce evaluarea următoarei expresii?
 $(+ 1 (\text{if condition } 2 \text{"OK"}))$
- Racket: verificare în momentul **aplicării** unui operator **predefinit** — tipare **dinamică**
- Răspunsul depinde de valoarea lui *condition*:
 - *true*: 3
 - *false*: Eroare, imposibilitatea adunării unui număr cu un șir
- Posibilitatea evaluării cu succes a unei expresii ce conține subexpresii eronate, cât timp cele din urmă **nu** sunt evaluate

54 / 423

Cuprins

3.33pt

- 1 Expresii și evaluare
- 2 Liste și perechi
- 3 Tipare
- 4 **Omoiconicitate și metaprogramare**

55 / 423

Omoiconicitate și metaprogramare

- **Corepondență** între sintaxa programului și structura de date fundamentală (lista)
- Racket — limbaj **omoiconic** (*homo* = aceeași, *icon* = reprezentare)
- Manipularea listelor ~ manipularea **codului**
- **Metaprogramare**: posibilitatea programului de a se **autorecrire**

56 / 423

Exemplu de metaprogramare

```
1 (define plus (list '+ 3 2)) ; '(+ 3 2)
2 (eval plus) ; 5
3
4 (define minus (cons '- (cdr plus))) ; '(- 3 2)
5 (eval minus) ; 1
```

Fortărea evaluării de către *eval*

57 / 423

Rezumat

- Limbaj **omoiconic**
- Evaluare bazată pe **substituție** textuală
- Tipare **dinamică** și **tare**

58 / 423

Partea III Recursivitate

59 / 423

Cuprins

3.33pt

- 9 Introducere
- 10 Tipuri de recursivitate
- 11 **Specificul recursivității pe coadă**

60 / 423

Cuprins

3.33pt

- 9 Introducere
- 10 Tipuri de recursivitate
- 11 **Specificul recursivității pe coadă**

61 / 423

Recursivitate

- Componentă **fundamentală** a paradigmei funcționale
- **Substituit** pentru iterarea clasică (*for*, *while* etc.), în **absența** stării
- Formă de *wishful thinking*: "Consider rezolvată **subproblema** și mă gândesc la cum să rezolv problema"

62 / 423

Cuprins

3.33pt

- 9 Introducere
- 10 Tipuri de recursivitate
- 11 **Specificul recursivității pe coadă**

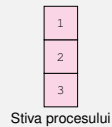
63 / 423

Funcția factorial

Recursivitate pe stivă, liniară

```
5 (define (fact-stack n)
6   (if (= n 1)
7       1
8       (* n (fact-stack (- n 1)))))

1 (fact-stack 3)
2 → (* 3 (fact-stack 2))
3 → (* 3 (* 2 (fact-stack 1)))
4 → (* 3 (* 2 1))
5 → (* 3 2)
6 → 6
```



Stiva procesului

Exemple preluate din: Abelson and Sussman (1996)

64/423

Recursivitate pe stivă, liniară

- Depunerea pe stivă a unor valori pe **avansul** în recursivitate
- Utilizarea acestora pentru calculul propriu-zis, pe **revenirea** din recursivitate
- **Spațiul** ocupat pe stivă: $\Theta(n)$
- Numărul de **operații**: $\Theta(n)$
- Informație "ascunsă", **implicită**, despre stare

65/423

Funcția factorial

Iterare clasică

```
1: procedure FACTORIAL(n)
2:   product ← 1
3:   i ← 1
4:   while i ≤ n do
5:     product ← product · i
6:     i ← i + 1
7:   end while
8:   return product
9: end procedure
```

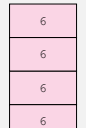
- **Starea** programului: variabilele *i* și *product*
- Spațiu **constant** pe stivă!
- Cum putem exploata această idee?

66/423

Funcția factorial

Recursivitate pe coadă

```
18 (define (fact-tail n)
19   (fact-tail-helper 1 1 n))
20
21 (define (fact-tail-helper product i n)
22   (if (> i n)
23       product
24       (fact-tail-helper (* product i)
25                           (+ i 1)
26                           n)))
27
28
29 (fact-tail-helper 1 1 3)
30 → (fact-tail-helper 1 2 3)
31 → (fact-tail-helper 2 3 3)
32 → (fact-tail-helper 6 4 3)
33 → 6
```



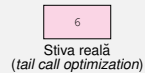
Stiva aparentă

67/423

Recursivitate pe coadă

- Calcul realizat pe **avansul** în recursivitate
- Aparent, **transportarea** neschimbată a valorii celei mai adânci aplicații recursive, către prima
- În realitate, **tail call optimization**: înlocuirea fiecărui apel cu următorul

```
1 (fact-tail-helper 1 1 3)
2 → (fact-tail-helper 1 2 3)
3 → (fact-tail-helper 2 3 3)
4 → (fact-tail-helper 6 4 3)
5 → 6
```



Stiva reală
(tail call optimization)

68/423

Recursivitate pe coadă (cont.)

- Numărul de **operații**: $\Theta(n)$
- **Spațiul** ocupat pe stivă: $\Theta(1)$
- În afară de economisirea spațiului, economisirea timpului necesar **redimensionării** stivei!
- Diferență față de iterarea clasică: transmiterea **explicită** a stării ca parametru

69/423

Funcții și procese

- Funcție: descriere **statică** a unor modalități de transformare
- Proces: Funcție în execuție, aspectul ei **dinamic**
- Posibilitatea unei funcții textual **recursive** (e.g., pe coadă) de a genera un proces **iterativ**!

70/423

Funcția Fibonacci

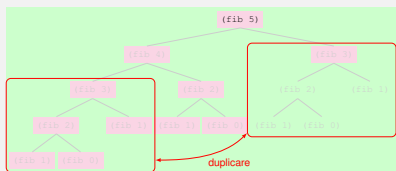
Recursivitate pe stivă, arborescentă

```
36 (define (fib-stack n)
37   (cond [(= n 0) 0]
38         [(= n 1) 1]
39         [else (+ (fib-stack (- n 1))
40                  (fib-stack (- n 2)))]))
```

71/423

Funcția Fibonacci (cont.)

Recursivitate pe stivă, arborescentă



72/423

Recursivitate pe stivă, arborescentă

- **Spațiul** ocupat pe stivă: lungimea unei căi din arbore: $\Theta(n)$
- În arborele cu rădăcina *fib*(*n*):
 - numărul frunzelor: *fib*(*n*+1)
 - numărul nodurilor: $2 \text{fib}(n+1) - 1$
- Numărul de **operații**: $\Theta(\text{fib}(n+1)) = \Theta(\phi^n)$ (ϕ — numărul de aur)
- Creștere **exponentială** a numărului de operații!

73/423

Funcția Fibonacci

Recursivitate pe coadă

```
50 (define (fib-tail n)
51   (fib-tail-helper 1 0 n))
52
53 (define (fib-tail-helper a b count)
54   (if (= count 0)
55       b
56       (fib-tail-helper (+ a b) a (- count 1))))
```

74/423

Recursivitate pe coadă

- Numărul de **operații**: $\Theta(n)$
- **Spațiul** ocupat pe stivă: $\Theta(1)$
- Diminuarea numărului de operații de la exponențial la **liniar**!

75/423

Recursivitate pe stivă vs. pe coadă

Pe stivă, lin./arb.

- **Elegantă**, adesea apropiată de specificație
- **Ineficientă spațial și/sau temporal**

Pe coadă

- **Obscură**, necesitând prelucrări specifice
- **Eficientă**, cel puțin spațial

Câteva cursuri mai târziu — o modalitate de exploatare eficientă a recursivității pe stivă

76/423

Transformarea în recursivitate pe coadă

- De obicei, posibilă, prin introducerea unui **acumulator** ca parametru (v. exemplele anterioare)

- În anumite situații, **imposibilă** direct:

```
1 (define (f x)
2   (if (zero? x)
3       0
4       (g (f (- x 1)))))
5   ; comportamentul lui g depinde
6   ; de parametru
```

77/423

Cuprins

3.33pt

1 Introducere

10 Tipuri de recursivitate

11 Specificul recursivității pe coadă

78/423

Construirea rezultatului

Recursivitate pe stivă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-stack L)
3   (if (null? L)
4       L
5       (cons (* (car L) 10)
6             (mult-stack (cdr L)))))
```

```
1 (mult-stack '(1 2))
2 → (cons 10 (mult-stack '(2)))
3 → (cons 10 (cons 20 (mult-stack '())))
4 → (cons 10 (cons 20 '()))
5 → (cons 10 '(20))
6 → '(10 20); ordinea este corecta
```

79/423

Construirea rezultatului

Recursivitate pe coadă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-tail-helper L Result)
3   (if (null? L)
4       Result
5       (mult-tail-helper (cdr L)
6                         (cons (* (car L) 10)
7                               Result))))
7
8 (mult-tail-helper '(1 2) '())
9 → (mult-tail-helper '(2) '(10))
10 → (mult-tail-helper '() '(20 10))
11 → '(20 10) ; ordinea este inversata
```

80/423

Construirea rezultatului (cont.)

Recursivitate pe coadă

Alternative pentru **conservarea ordinii**:

- **Inversarea listei finale**

```
1 (if (null? L)
2     ...
3     (reverse Result)
4     ...)
```
- **Adăugarea elementului curent la sfârșitul acumului.**

```
1 (if (null? L)
2     ...
3     (mult-all-iter
4       (cdr L)
5       (append Result
6                (list (* (car L) 10)))))
```

81/423

Costul unei concatenări

```
1 (define (app A B) ; recursiva pe stiva
2   (if (null? A)
3       B
4       (cons (car A) (app (cdr A) B))))
```

Număr de operații proporțional cu lungimea **primei** liste!

82/423

Costul concatenărilor repetate

- **Asociere la dreapta:**
 $A ++ (B ++ (C ++ \dots))$
Număr de operații proporțional cu lungimea listei **curente**
- **Asociere la stânga:**
 $(\dots((\dots ++ A) ++ B) ++ C$
Număr de operații proporțional cu lungimea **tuturor** listelor concatenate anterior

83/423

Consecințe asupra recursivității pe coadă

```
1 (define (mult-tail-helper L Result)
2   (if (null? L)
3       Result
4       (mult-tail-helper (cdr L)
5                         (append Result
6                                 (list (* (car L) 10))))))
7
```

```
1 (mult-tail-helper '(1 2 3) '())
2 → (mult-tail-helper '(2 3) (append '() '(10)))
3 → (mult-tail-helper '(3) (append '(10) '(20)))
4 → (mult-tail-helper '() (append '(10 20)
5                                 '(30)))
6 → (mult-tail-helper '() '(10 20 30))
7 → '(10 20 30)
```

84/423

Consecințe asupra recursivității pe coadă (cont.)

- Parcurgerea **întregului** acumulator anterior, pentru construirea celui nou!
- Numărul de elemente parcurse:
 $0 + 1 + \dots + (n-1) = \Theta(n^2)!$
- Astfel, preferabilă varianta **inversării**, și nu cea a adăugării la sfârșit

85/423

Rezumat

- Diverse **tipuri** de recursivitate
 - pe stivă (liniară/ arborescentă)
 - pe coadă
- Recursivitate pe **stivă**: de obicei, ...
 - Eleganță
 - Ineficiență spațial și/ sau temporal
- Recursivitate pe **coadă**: de obicei, ...
 - Mai puțin lizibilă decât cea pe stivă
 - Necesită prelucrări suplimentare (e.g. inversare)
 - Eficiență spațial și/ sau temporal

86/423

Bibliografie

Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition.

87/423

Partea IV

Funcții ca valori de prim rang. Funcționale

88/423

Cuprins

3.33pt

- 12 Motivație
- 13 Funcții ca valori de prim rang
- 14 Funcționale
- 15 Calculul lambda

89/423

Cuprins

3.33pt

- 12 Motivație
- 13 Funcții ca valori de prim rang
- 14 Funcționale
- 15 Calculul lambda

90/423

Abstractizare funcțională

```
1 (define (double n)
2   (* n 2))
3
4 (double 5)
5 → (* 5 2)
6
7 (define (double n)
8   (define (double n)
9     (* n n)))
10
11 (double 10)
12 → (* 10 2)
```

- Generalizare, de la dublarea valorilor particulare, la însuși **conceptul de dublare**
- Rezultat: funcția **double**, **substituibilă** cu orice altă funcție cu același comportament
- Mai precis, **double** = **abstractizare funcțională**

91/423

Un nivel mai sus

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 ;; '(1 2 3) -> '(10 20 30)
3 (define (mult L)
4   (if (null? L)
5       L
6       (cons (* (car L) 10)
7             (mult (cdr L)))))
8
9 ;; Obține paritatea fiecărui număr (true = par)
10 ;; '(1 2 3) -> '(false true false)
11 (define (parities L)
12   (if (null? L)
13       L
14       (cons (even? (car L))
15             (parities (cdr L)))))
```

singura parte
variabilă,
dependentă
de (car L)

rolul lui
(car L)

92/423

Un nivel mai sus (cont.)

Cum putem **izola** transformarea lui (car L)?
Prin funcții!

```
1 ;; map = asociere
2
3 (define (mult-map x)
4   (* x 10))
5
6 (define (parities-map x)
7   (even? x))
```

93/423

Un nivel mai sus (cont.)

```
1 (define (map f L)
2   (if (null? L)
3       L
4       (cons (f (car L))
5             (map f (cdr L)))))
6
7 (define (mult L)
8   (map mult-map L))
9
10 (define (parities L)
11   (map parities-map L))
```

transformarea
lui (car L):
parametru

Generalizare, de la diversele transformări ale listelor, la **conceptul** de transformare element cu element, **independent** de natura acestora — **asociere (mapping)**

94/423

Cuprins

3.33pt

- 12 Motivație
- 13 Funcții ca valori de prim rang
- 14 Funcționale
- 15 Calculul lambda

95/423

Funcții ca valori de prim rang

- În exemplele anterioare: funcții văzute ca **date!**
- Avantaj: sporire considerabilă a **expressivității** limbajului
- Statutul de **valori** de prim rang al funcțiilor, acestea putând fi:
 - create **dinamic** (la execuție)
 - **numite**
 - trimise ca **parametri** unei funcții
 - **intoarse** dintr-o funcție

96/423

Evaluarea funcțiilor

Ca valori, evaluate la ele **însele!**

```
1 > +
2 #<procedure:+>
3
4 > (cons + '(1 2))
5 (#<procedure:+> 1 2)
6
7 > (list + - *)
8 (#<procedure:+> #<procedure:-> #<procedure:*>)
```

97/423

Funcții ca parametru

- În exemplele anterioare, funcții definite separat, deși folosite o **singură dată**:

```
1 (define (mult L)
2   (map mult-map L))
3
4 (define (parities L)
5   (map parities-map L))
```

- Putem defini funcțiile **local** unei expresii?

98/423

Funcții anonime

```
1 (define (mult L)
2   (map (lambda (x) (* x 10)) L))
3
4 (define (parities L)
5   (map (lambda (x) (even? x)) L))
```

De fapt,

```
1 (define (mult-map x)
2   (* x 10))
3
4 (define (parities-map x)
5   (even? x))
```

simpla **legare** a variabilei `mult-map` la o funcție anonimă

99/423

Funcții ca valori de retur

- În exemplul cu funcția `mult`, cum înmulțim toate elementele listei cu un număr **oarecare**, nu neapărat cu 10?

- Posibilă utilizare, pentru înmulțirea cu 5:

```
1 (map (mult-map-by 5) '(1 2 3))
```

- Cum aplicăm `mult-map-by` doar asupra **primului** parametru?

```
1 (define (mult-map-by q x)
2   (* x q))
3
1 (define (mult-map-by q)
2   (lambda (x)
3     (* x q)))
```

simultan (**uncurried**)
pe rând (**curried**)
funcție

100/423

Secvențierea parametrilor

- În loc să afirmăm că `mult-map-by` are un parametru și că întoarce o funcție, ne "prefacem" că primește **doi** parametri, pe rând

- Avantaj: **reutilizare**, prin aplicare **parțială!**

- Funcție **curried**: preia parametrul **pe rând** (aparent)

- Funcție **uncurried**: preia parametrul **simultan**

101/423

Extinderea regulilor de evaluare

- Din moment ce funcțiile sunt valori posibile ale expresiilor, necesitatea evaluării inclusiv a **operatorului** unei aplicații

- Mai departe, evaluarea variabilei → la valoarea ei — funcția de adunare!

```
1 ((if true + -) (+ 1 2) 3)
2 → (+ (+ 1 2) 3)
3 → (#<procedure:+> (+ 1 2) 3)
```

Notă: Pasul de evaluare 2-3 nu transpune la utilizarea *stepper*-ului din Racket, dar este prezent pe slide pentru completitudine.

102/423

Aplicație: compunerea a două funcții

```
1 (define (comp f g)
2   (lambda (x)
3     (f (g x))))
4
5 ((comp car cdr) '(1 2 3)) → 2
```

103/423

Cuprins

3.33pt

- 12 Motivatie
- 13 Funcții ca valori de prim rang
- 14 Funcționale
- 15 Calculul lambda

104/423

Funcționale

- Funcțională = funcție care primește ca parametru și/ sau întoarce o **funcție**

- Surprind metode **generale** de prelucrare

- Funcționale **standard** în majoritatea limbajelor funcționale (prezentate în continuare):

- `map`
- `filter`
- `foldl` (*fold left*)
- `foldr` (*fold right*)

105/423

Funcționala `map`

- Aplicarea unei **transformări** asupra tuturor elementelor unei liste

- Tratată anterior

```
1 (map (lambda (x) (* x 10)) '(1 2 3))
2 → '(10 20 30)
```

106/423

Funcționala `filter`

- Extragerea dintr-o listă a elementelor care **satisfac** un predicat logic

- Funcția primită ca parametru trebuie să întoarcă o valoare **booleană**

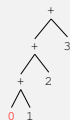
```
1 (filter even? '(1 2 3))
2 → '(2)
```

107/423

Funcționala `foldl`

- Acumularea tuturor elementelor unei liste sub forma unei **singure** valori (posibil tot listă, dar nu exclusiv)
- Pacurgere stânga → dreapta
- Utilizarea unei funcții **binare** element-acumulator
- Pornire cu un acumulator **initial**
- Natural recursivă pe **coadă**

```
1 (foldl + 0 '(1 2 3))
2 → 6
```

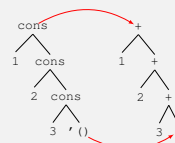


108/423

Funcționala `foldr`

- Similar cu `foldl`
- Pacurgere dreapta → stânga
- Operare pe **structura** listei inițiale
- Natural recursivă pe **stivă**

```
1 (foldr + 0 '(1 2 3))
2 → 6
```



109/423

Universalitatea funcționalelor `fold+`

- **Orice** funcție primitivă recursivă pe liste, implementabilă în termenii funcționalelor `fold+`

- În particular, utilizabile pentru implementarea funcționalelor `map` și `filter`!

110/423

Cuprins

3.33pt

- 12 Motivatie
- 13 Funcții ca valori de prim rang
- 14 Funcționale
- 15 Calculul lambda

111/423

Trăsături

- Model de **calculabilitate** — Alonzo Church, 1932
- Centrat pe conceptul de **funcție**
- Calculul: evaluarea aplicațiilor de funcții, prin **substituție** textuală

112 / 423

Evaluare

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

“Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, y , se indentifică parametrul formal, x , în corpul funcției, x , iar aparițiile primului, x (singura), se **substituie** cu parametrul actual, obținându-se rezultatul unui pas de evaluare.”

113 / 423

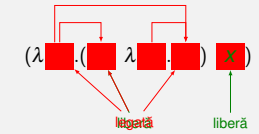
Formalizarea substituției

În expresia $(\lambda x. \lambda x. y)$:

- Aplicarea mecanică a principiului substituției: $\lambda y.y$
- Intuitiv: $\lambda x.y$
- Rezultat **eronat** al abordării mecanice!
- **Ce** ar trebui substituit de fapt?

114 / 423

Apariții libere și legate ale variabilelor



- Apariție **legată** a lui x :
 - După λ
 - În corpul unei funcții de **parametru** x
- Dependentă statutului unei apariții de **expresia** la care ne raportăm!

115 / 423

Formalizarea substituției (cont.)

- Substituirea tuturor aparițiilor parametrului formal, care sunt **libere** în raport cu **corpul**!
- În exemplul anterior, $(\lambda x. \lambda x. y)$:
 - **Absența** aparițiilor libere ale lui x în corpul $\lambda x.y$
 - Producerea **corectă** a corpului nemodificat ca rezultat
- În expresia $(\lambda x. \lambda cons.x cons)$:
 - Apariția din dreapta a lui $cons$ este **liberă**, cu semnificația din Racket
 - Aplicarea mecanică: $\lambda cons.cons$
 - Rezultat eronat, din cauza modificării statutului, din apariție liberă în **legată**

116 / 423

Redenumirea variabilelor legate

$$(\lambda x. \lambda cons.x cons)$$

Aparițiile **legate** din corp, în conflict cu cele **libere** din parametrul actual, **redenumite!**

117 / 423

Formalizarea substituției — concluzie

- Substituirea tuturor aparițiilor parametrului formal, care sunt **libere** în raport cu corpul, **ulterioră** eventualelor **redenumiri** ale aparițiilor **legate** din corpul funcției, care coincid cu aparițiile **libere** din parametrul actual
- În exemplul anterior, $(\lambda x. \lambda z.x cons) \rightarrow \lambda z.cons$
- Rezultat corect, cu păstrarea statutului de apariție **liberă**

118 / 423

Universalitatea funcțiilor

- Posibilitatea reprezentării tuturor valorilor uzuale **exclusiv** prin funcții (v. slide-ul 50)
- Mai devreme, funcții ca date (parametri, valori de retur etc.)
- Acum, date ca funcții!!
- V. sursele atașate slide-urilor

119 / 423

Rezumat

- **Abstractizare** funcțională
- Funcții ca **valori** — sporirea **expressivității** limbajului
- Funcionale — metode **generale** de prelucrare
- Calculul lambda și **universalitatea** funcțiilor

120 / 423

Partea V

Legarea variabilelor. Evaluare contextuală

121 / 423

Cuprins

3.33pt

- 16 Legarea variabilelor
- 17 Contexte, închideri, evaluare contextuală

122 / 423

Cuprins

3.33pt

- 16 Legarea variabilelor
- 17 Contexte, închideri, evaluare contextuală

123 / 423

Variabile

Proprietăți

- Tip: asociate valorilor, **nu** variabilelor
- Identificator
- Valoarea **legată** (la un anumit moment)
- Domeniul de vizibilitate
- Durata de viață

124 / 423

Variabile

Stări

- Declarată: cunoaștem **identificatorul**
- Definită: cunoaștem și **valoarea**

125 / 423

Legarea variabilelor

- Modul de **asociere** a apariției unei variabile cu definiția acesteia
- Domeniul de vizibilitate (*scope*) = mulțimea **punctelor** din program unde o definiție este vizibilă, pe baza modului de **legare**
- Statică (lexicală) / dinamică

126 / 423

Problemă

```
1 def x = 0
2 f () { return x }
3 def x = 1
4 g () { def x = 2 ; return f () }
```

- **Atenție!** Variabilele x sunt **diferite**, nu se reatribuie același x (aceasta este semnificația lui `def`)
- În câte **moduri** poate decurge evaluarea aplicației `g()`, în raport cu variabilele definite?

127 / 423

Legare statică (lexicală)

- Extragerea variabilelor din contextul **definirii** expresiei
- Domeniul de vizibilitate determinat prin **construcțiile** limbajului (lexical), la **compilare** (static)

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
```

g() → 0

128 / 423

Legare statică în calculul lambda

Care sunt domeniile de vizibilitate ale parametrilor formali, în expresia de mai jos?

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

129 / 423

Legare dinamică

- Extragerea variabilelor din contextul **evaluării** expr.
- Domeniul de vizibilitate determinat la **execuție**

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
5 ...
```

← f() → 0
← f() → 1
← f() → 2 <- g()
← f() → 1

Atenție! x-ul **portocaliu**, vizibil:

- spațial: în **întregul** program
- temporal: doar pe durata evaluării **corpului** lui g()

130 / 423

Legare mixtă

- Variabile locale, **static**
- Variabile globale, **dinamic**

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
5 ...
```

← f() → 0
← f() → 1
← f() → 1 <- g()
← f() → 1

Atenție! x-ul **portocaliu**, **invizibil** în corpul lui f!

131 / 423

Legarea variabilelor în Racket

- Variabile declarate sau definite în expresii: **static**:

- lambda
- let
- let*
- letrec

- Variabile **top-level**: **dinamic**:

- define

132 / 423

Construcția lambda

Definiție

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

- Sintaxă:

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

- Domeniul de vizibilitate a parametrului p_k = mulțimea punctelor din **corpul** funcției, $expr$, în care aparițiile lui p_k sunt **libere** (v. slide-ul 128)

133 / 423

Construcția lambda

Exemplu

```
1 (lambda (x)
2   (x (lambda (y) y)))
```

134 / 423

Construcția lambda

Semantică

- Aplicație:

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

- Se evaluează **operandii** a_k , în ordine aleatoare (evaluare aplicativă)
- Se evaluează **corpul** funcției, $expr$, ținând cont de legăturile $p_k \leftarrow valoare(a_k)$
- Valoarea** aplicației este valoarea lui $expr$

135 / 423

Construcția let

Definiție

- Leagă **static** variabile locale

- Sintaxă:

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

- Domeniul de vizibilitate a variabilei v_k = mulțimea punctelor din **corp**, $expr$, în care aparițiile lui v_k sunt **libere** (v. slide-ul 128)

136 / 423

Construcția let

Exemplu

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

137 / 423

Construcția let

Semantică

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

echivalent cu

```
1 ((lambda (v1 ... vn)
2   expr) e1 ... en)
```

138 / 423

Construcția let*

Definiție

- Leagă **static** variabile locale

- Sintaxă:

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

- Domeniul de vizibilitate a variabilei v_k = mulțimea punctelor din
 - restul **legăturilor** și
 - corp**, $expr$,în care aparițiile lui v_k sunt **libere** (v. slide-ul 128)

139 / 423

Construcția let*

Exemplu

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

140 / 423

Construcția let*

Semantică

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

echivalent cu

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

Evaluarea expresiilor se face **în ordine!**

141 / 423

Construcția letrec

Definiție

- Leagă **static** variabile locale

- Sintaxă:

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

- Domeniul de vizibilitate a variabilei v_k = mulțimea punctelor din **întreaga** construcție, în care aparițiile lui v_k sunt **libere** (v. slide-ul 128)

142 / 423

Construcția letrec

Exemplu

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

143 / 423

Construcția define

Definiție

- Leagă **dinamic** variabile *top-level* (de obicei)
- Sintaxă:

```
1 (define v expr)
```
- Domeniul de vizibilitate a variabilei $v =$ **întregul program**, presupunând că:
 - legarea a fost făcută, în timpul **execuției**
 - nicio o altă** legare, statică sau dinamică, a lui v , nu a fost făcută ulterior

144/423

Construcția define

Exemple

```
1 (define x 0)
2 (define f (lambda () x))
3 (f) ; 0
4 (define x 1)
5 (f) ; 1
```

145/423

Construcția define

Exemple

```
1 (define factorial
2   (lambda (n)
3     (if (zero? n) 1
4         (* n (factorial (- n 1))))))
5
6 (factorial 5) ; 120
7
8 (define g factorial)
9 (define factorial (lambda (x) x))
10
11 (g 5) ; 20
```

146/423

Construcția define

Semantică

- Se evaluează **expresia**, $expr$
- Valoarea** lui v este valoarea lui $expr$
- Avantaje:
 - definirea variabilelor *top-level* în **orice** ordine
 - definirea funcțiilor **mutual** recursive
- Dezavantaj: efect de **atribuire**

147/423

Exemplu mixt

Codificarea secvenței de pe slide-ul 131

```
1 (define x 0)
2 (define f (lambda () x))
3 (define x 1)
4
5 (define g
6   (lambda (x)
7     (f)))
8
9 (g 2) ; 1
```

148/423

Aplicație pentru legarea variabilelor

```
79 (define (app A B)
80   (if (null? A)
81       B
82       (cons (car A) (app (cdr A) B))))
```

Problemă: B este trimis **nemodificat** fiecărei aplicații recursive. Rescriem:

```
87 (define (app2 A B)
88   (letrec ([internal
89             (lambda (L)
90               (if (null? L) B
91                   (cons (car L)
92                           (internal (cdr L))))))]
93     (internal A)))
```

149/423

Cuprins

3.33pt

- 13 Legarea variabilelor
- 17 Contexte, închideri, evaluare contextuală

150/423

Modelul de evaluare bazat pe substituție

- Ineficient**
- Tratament special pentru **coliziunile** dintre variabile libere ale parametrului actual și cele legate ale corpului funcției aplicate
- Imposibil** de aplicat, în prezența unor eventuale reatribuiri ale variabilelor

151/423

Alternativă la substituția textuală

$(\lambda x.x y) \rightarrow y$
→ $(x; \{x \leftarrow y\})$ ← închidere
↑
expresie context

- Asocierea unei expresii cu un dicționar de variabile libere: **context** de evaluare
- Căutarea** unei variabile utilizate în procesul de evaluare, în contextul asociat
- Perechea: **închidere**, i.e. formă pseudoînchisă a expresiei, obținută prin legarea variabilelor libere

152/423

Context computațional

- Mulțime de **variabile**, alături de **valorile** acestora
- Dependent de **punctul** din program și de momentul de **timp**
- Legare **statică** — mulțimea variabilelor care conțin punctul conform structurii **lexicale** a programului
- Legare **dinamică** — mulțimea variabilelor definite cel mai **recent**

```
1 (let ([x 1])
2   (+ x (let ([y 2])
3         (+ x y)))) {x ← 1, y ← 2}
```

153/423

Închideri

Definiție

- Închidere: **pereche** expresie-context
- Semnificația** unei închideri:
 (e, C)
este valoarea expresiei e , în contextul C
- Închidere **funcțională**:
 $(\lambda x.e, C)$
este o funcție care își salvează contextul, pe care îl utilizează, în momentul aplicării, pentru evaluarea corpului
- Utilizate pentru legare **statică!**

154/423

Închideri

Construcție

- Construcție prin evaluarea unei expresii **lambda**, într-un context dat
- Legarea** variabilelor *top-level*, în contextul global, prin **define**

```
1 (define y 0)
2 (define sum (lambda (x) (+ x y)))
```

$y \leftarrow 0$
 $sum \leftarrow (\lambda x.(+ x y))$ Contextul global

Pointer către contextul global

155/423

Închideri

Aplicare

- Legarea parametrilor formali, într-un **nou** context, la valorile parametrilor actuali
- Mostenirea** contextului din închidere de către cel nou
- Evaluarea **corpului** închiderii în noul context

```
1 (sum (+ 1 2))
```

G $y \leftarrow 0$
 $sum \leftarrow (\lambda x.(+ x y))$ Contextul global

↑ Mostenire

C $x \leftarrow 3$

Contextul în care se evaluează corpul $(+ x y)$

156/423

Ierarhia de contexte

- Arbore** având contextul global drept rădăcină
- În cazul **absenței** unei variabile din contextul curent, căutarea acesteia în contextul **părinte** s.a.m.d.
- Pe slide-ul 156:
 - x : identificat în C
 - y : absent din C , dar identificat în G , părintele lui C

157/423

Închideri funcționale

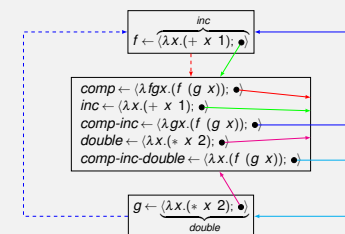
Exemplu

```
1 (define comp
2   (lambda (f)
3     (lambda (g)
4       (lambda (x)
5         (f (g x))))))
6
7 (define inc (lambda (x) (+ x 1)))
8 (define comp-inc (comp inc))
9
10 (define double (lambda (x) (* x 2)))
11 (define comp-inc-double (comp-inc double))
12
13 (comp-inc-double 5) ; 11
14
15 (define inc (lambda (x) x))
16 (comp-inc-double 5) ; tot 11!
```

158/423

Închideri funcționale

Explicația exemplului



159/423

Rezumat

- Legare **statică/ dinamică** a variabilelor
- Contexte de evaluare, închideri, evaluare contextuală

160/423

Partea VI

Întârzierea evaluării

161/423

Cuprins

3.33pt

- 18 Mecanisme
- 19 Abstractizare de date
- 20 Fluxuri
- 21 Rezolvarea problemelor prin căutare leneșă în spațiul stărilor

162/423

Cuprins

3.33pt

- 18 Mecanisme
- 19 Abstractizare de date
- 20 Fluxuri
- 21 Rezolvarea problemelor prin căutare leneșă în spațiul stărilor

163/423

Motivație

- Să se implementeze funcția *prod*:
 - $prod(false, y) = 0$
 - $prod(true, y) = y(y+1)$
- Se presupune că evaluarea lui *y* este costisitoare, și că ar trebui efectuată doar dacă este necesar.

164/423

Varianta 1

Implementare directă

```
1 (define (prod x y)
2   (if x (* y (+ y 1)) 0))
3
4 (define (test x)
5   (let ([y 5])
6     (prod x (begin (display "y") y))))
7
8 (test #f) ; y 0
9 (test #t) ; y 30
```

Implementare **eronată**, deoarece **ambii** parametri sunt evaluați în momentul aplicării!

165/423

Varianta 2

quote & eval

```
1 (define (prod x y)
2   (if x (* (eval y) (+ (eval y) 1)) 0))
3
4 (define (test x)
5   (let ([y 5])
6     (prod x '(begin (display "y") y))))
7
8 (test #f) ; 0
9 (test #t) ; y y: undefined
```

- $x = \#f$ — comportament corect, *y* neevaluat
- $x = \#t$ — **eroare**, quote **nu** salvează contextul

166/423

Varianta 3

Închideri funcționale

```
1 (define (prod x y)
2   (if x (* (y) (+ (y) 1)) 0))
3
4 (define (test x)
5   (let ([y 5])
6     (prod x (lambda ()
7               (begin (display "y") y))))))
8
9 (test #f) ; 0
10 (test #t) ; yy 30
```

- Comportament corect: *y* evaluat **la cerere**
- $x = \#t$ — *y* evaluat de 2 ori, **ineficient**

167/423

Varianta 4

Promisiuni: *delay* & *force*

```
1 (define (prod x y)
2   (if x (* (force y) (+ (force y) 1)) 0))
3
4 (define (test x)
5   (let ([y 5])
6     (prod x (delay (begin (display "y") y)))))
7
8 (test #f) ; 0
9 (test #t) ; y 30
```

Comportament corect: *y* evaluat **la cerere**, o **singură dată** — evaluare **leneșă**

168/423

Promisiuni

Descriere

- Rezultatul încă **neevaluat** al unei expresii
- Exemplu: `(delay (* 5 6))`
- Valori de **prim rang** în limbaj (v. slide-ul 96)
- *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**

169/423

Observații

- **Dependență** între mecanismul de întârziere și cel de evaluare ulterioară a expresiilor — închideri/ aplicații (varianta 3), *delay/force* (varianta 4) etc.
- Număr **mare** de modificări la **înlocuirea** unui mecanism existent, utilizat de un număr mare de funcții
- Cum se pot **diminua** dependențele?

170/423

Cuprins

3.33pt

- 18 Mecanisme
- 19 Abstractizare de date
- 20 Fluxuri
- 21 Rezolvarea problemelor prin căutare leneșă în spațiul stărilor

171/423

Abstractizare de date I

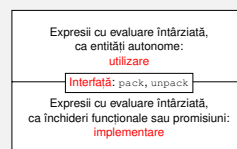
- Cum **reprezentăm** expresiile cu evaluare întârziată?
- Abordarea din secțiunea precedentă: **1** singur nivel

Expresii cu evaluare întârziată:
utilizare și implementare,
sub formă de închideri sau promisiuni

172/423

Abstractizare de date II

- Alternativ: **2** nivele, separate de o **barieră** de abstractizare



- **Bariera:**
 - **limitează** analiza detaliilor
 - **elimină** dependențele dintre nivele

173/423

Abstractizare de date III

- Tehnică de **separare** a utilizării unei structuri de date de implementarea acesteia.
- Permite **wishful thinking**: utilizarea structurii **înainte** implementării acesteia

174/423

Abstractizare de date IV

```
1 (define-syntax-rule (pack expr)
2   (delay expr)) ; sau (lambda () expr)
3
4 (define (unpack force) ; sau (lambda (p) (p))
5
6 (define (prod x y)
7   (if x (* (unpack y) (+ (unpack y) 1)) 0))
8
9 (define (test x)
10  (let ([y 5])
11    (prod x (pack (begin (display "y") y)))))
```

175/423

Cuprins

3.33pt

Mecanisme

Abstractizare de date

Fluxuri

Rezolvarea problemelor prin căutare leasă în spațiul stărilor

176 / 423

Motivație

Să se determine suma numerelor pare din intervalul [a, b].

```
1 (define (even-sum-iter a b)
2   (let iter (([n a]
3               [sum 0]))
4     (cond [(> n b) sum]
5           [(even? n) (iter (+ n 1) (+ sum n))]
6           [else (iter (+ n 1) sum)])))
7
8 (define (even-sum-lists a b)
9   (foldl + 0 (filter even? (interval a b))))
```

177 / 423

Comparație

- Varianta iterativă (d.p.d.v. proces):
 - **eficientă**, datorită spațiului suplimentar constant
 - **nu** foarte lizibilă
- Varianta pe liste:
 - **elegantă și concisă**
 - **ineficientă**, datorită
 - spațiului posibil mare ocupat la un moment dat — **toate** numerele din intervalul [a, b]
 - parcurgerii **repetate** a intervalului (interval, filter, foldl)
- Cum **îmbinăm** avantajele celor două abordări?

178 / 423

Caracteristicile fluxurilor

- Secvențe construite **parțial**, extinse la cerere, ce creează **iluzia** completitudinii structurii
- Îmbinarea **elegantei** manipulării listelor cu **eficiența** calculului incremental
- Bariera de abstractizare:
 - componentele listelor evaluate la **construcție** (cons)
 - ale fluxurilor la **selectie** (cdr)
- Construcția și utilizarea:
 - **separate** la nivel conceptual — **modularitate**
 - **întrepărunse** la nivel de proces

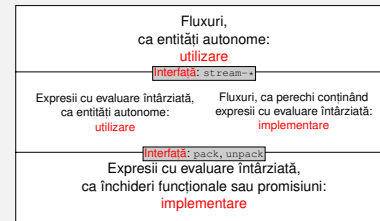
179 / 423

Operatori

```
3 (define-syntax-rule (stream-cons head tail)
4   (cons head (pack tail)))
5
6 (define stream-first car)
7
8 (define stream-rest (compose unpack cdr))
9
10 (define empty-stream '())
11
12 (define stream-empty? null?)
```

180 / 423

Barierile de abstractizare

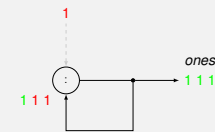


181 / 423

Fluxul de numere 1

Implementare

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



- Linii continue: fluxuri
- Linii întrerupte: intrări scalare, utilizate o singură dată
- Cifre: **intrări / ieșiri**

182 / 423

Fluxul de numere 1

Utilizarea memoriei

Atât cu închideri, cât și cu promisiuni, extinderea se realizează în spațiu constant:



Alternativ: (define ones (pack (cons 1 ones)))

- închideri:
- promisiuni:

183 / 423

Fluxul numerelor naturale

Formulare explicită

```
10 (define (naturals-from n)
11   (stream-cons n (naturals-from (+ n 1))))
12
13 (define naturals (naturals-from 0))
```

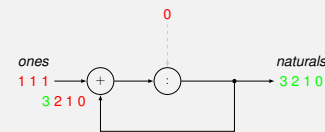
- Închideri: multiple parcurgeri ale fluxului determină **reevaluarea** porțiunilor deja explorate
 - Explorarea 1, cu 3 elemente: 0 1 2
 - Explorarea 2, cu 5 elemente: 0 1 2 3 4
- Promisiuni: multiple parcurgeri ale fluxului determină evaluarea **dincolo** de porțiunile deja explorate
 - Explorarea 1, cu 3 elemente: 0 1 2
 - Explorarea 2, cu 5 elemente: 0 1 2 3 4

184 / 423

Fluxul numerelor naturale

Formulare implicită

```
17 (define naturals
18   (stream-cons 0
19               (stream-zip-with +
20                               ones
21                               naturals)))
```



185 / 423

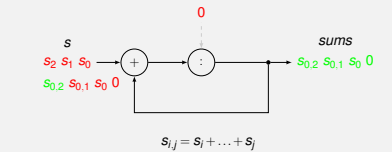
Fluxul numerelor pare

```
25 (define even-naturals-1
26   (stream-filter even? naturals))
27
28 (define even-naturals-2
29   (stream-zip-with + naturals naturals))
```

186 / 423

Fluxul sumelor parțiale ale altui flux

```
33 (define (sums s)
34   (letrec ([out (stream-cons
35                0
36                (stream-zip-with + s out))])
37     out))
```

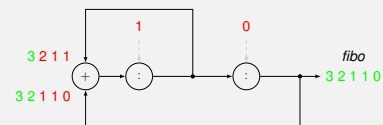


187 / 423

Fluxul numerelor Fibonacci

Formulare implicită

```
43 (define fibo
44   (stream-cons 0
45               (stream-cons 1
46                           (stream-zip-with +
47                                           fibo
48                                           (stream-rest fibo)))))
```



188 / 423

Fluxul numerelor prime I

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

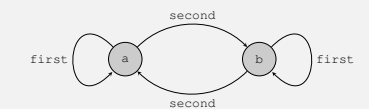
189 / 423

Fluxul numerelor prime II

```
52 (define (sieve s)
53   (if (stream-empty? s) s
54       (stream-cons
55         (stream-first s)
56         (sieve
57          (stream-filter
58            (lambda (n)
59              (not (zero? (remainder
60                          n
61                          (stream-first s))))))
62          (stream-rest s)))))
63
64 (define primes (sieve (naturals-from 2)))
```

190 / 423

Grafuri ciclice I



- Fiecare nod conține:
- cheia: key
 - legăturile către două noduri: first, second

191 / 423

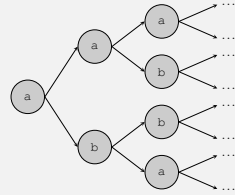
Grafuri ciclice II

```
3 (define-syntax-rule (node key fst snd)
4   (pack (list key fst snd)))
5
6 (define key car)
7 (define fst (compose unpack cadr))
8 (define snd (compose unpack caddr))
9
10 (define graph
11   (letrec ([a (node 'a a b)]
12            [b (node 'b b a)]))
13   (unpack a)))
14
15 (eq? graph (fst graph)) ; similar cu == din Java
16 ; #f pentru inchideri, #t pentru promisiuni
```

192 / 423

Grafuri ciclice III

- Explorarea grafului în cazul **închiderilor**: nodurile sunt **regenerate** la fiecare vizitare



193 / 423

Cuprins

3.33pt

- 13 Mecanisme
- 15 Abstractizare de date
- 18 Fluxuri
- 21 Rezolvarea problemelor prin căutare leneșă în spațiul stărilor

194 / 423

Spațiul stărilor unei probleme

Multimea configurațiilor valide din universul problemei

195 / 423

Problema palindroamelor

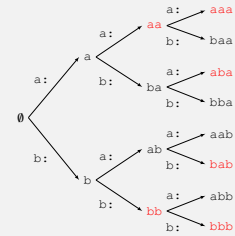
Definiție

- Pal_n : Să se determine palindroamele de lungime cel puțin n , care se pot forma cu elementele unui alfabet fixat.
- Stările problemei: **toate** șirurile generabile cu elementele alfabetului respectiv

196 / 423

Problema palindroamelor

Spațiul stărilor lui Pal_2



197 / 423

Problema palindroamelor

Specificare Pal_n

- Starea **inițială**: șirul vid
- Operatorii de generare a stărilor **succesoare** alea: inserarea unui caracter la începutul unui șir dat
- Operatorul de verificare a proprietății de **soluție** pentru o stare: palindrom, de lungime cel puțin n

198 / 423

Căutare în spațiul stărilor

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

199 / 423

Căutare în lățime

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

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

200 / 423

Căutare leneșă în lățime I

Fluxul stărilor soluție

```
3 (define (lazy-breadth-search init expand)
4   (let search
5     ([states (stream-cons init empty-stream)]
6      (if (stream-empty? states) states
9        (let ([state (stream-first states)]
10              [states (stream-rest states)])
11          (stream-cons
12            state
13            (search (stream-append
14                    states
15                    (expand state))))))))))
14
15 (define (lazy-breadth-search-goal
16   init expand goal?)
17   (stream-filter goal?
```

201 / 423

Căutare leneșă în lățime II

Fluxul stărilor soluție

```
18 (lazy-breadth-search init
19   expand))
```

- La nivel înalt, conceptual: **separare** între explorarea spațiului și identificarea stărilor soluție
- La nivelul scăzut, al instrucțiunilor: **întrepătrunderea** celor două aspecte

202 / 423

Aplicații

- Palindroame
- Problema reginelor

203 / 423

Problema reginelor

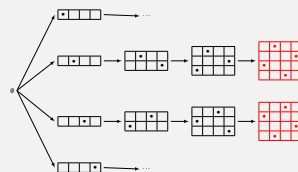
Definiție

- $Queens_n$: Să se determine toate modurile de amplasare a n regine pe o tablă de șah de dimensiune n , astfel încât oricare două să nu se atace.
- Stările problemei: configurațiile, eventual parțiale, ale **tablei**

204 / 423

Problema reginelor

Spațiul stărilor lui $Queens_4$



205 / 423

Rezumat

Evaluarea leneșă permite un stil de programare de nivel înalt, prin separarea aparentă a diverselor aspecte — de exemplu, construcția și accesarea listelor.

206 / 423

Bibliografie

Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition.

207 / 423

Partea VII

Limbajul Haskell

208 / 423

Cuprins

3.33pt

- 1 Introducere
- 2 Evaluare
- 3 Tipare
- 4 Sinteza de tip

209 / 423

Cuprins

3.33pt

- 1 Introducere
- 2 Evaluare
- 3 Tipare
- 4 Sinteza de tip

210 / 423

Paralelă între limbaje

Criteriu	Scheme	Haskell
Funcții	Curried / uncurried	Curried
Evaluare	Aplicativă	Lenesă
Tipare	Dinamică, tare	Statică, tare
Legarea variabilelor	Locale → statică, top-level → dinamică	Statică

211 / 423

Funcții

- Curried
- Aplicabile asupra oricărui parametri la un moment dat

```
1 add1 x y = x + y
2 add2 = \x y -> x + y
3 add3 = \x -> \y -> x + y
4
5 result = add1 1 2 -- sau ((add1 1) 2)
6 inc = add1 1 -- functie
```

212 / 423

Funcții și operatori

- Aplicabilitatea parțială a operatorilor infixati (secțiuni)
- Transformări operator → funcție și funcție → operator

```
1 add4 = (+)
2
3 result1 = (+) 1 2 -- operator ca functie
4 result2 = 1 `add4` 2 -- functie ca operator
5
6 inc1 = (1 +) -- secțiuni
7 inc2 = (+ 1)
8 inc3 = (1 `add4`)
9 inc4 = (`add4` 1)
```

213 / 423

Pattern matching

Definirea comportamentului funcțiilor pornind de la structura parametrilor — traducerea axiomelor TDA

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

214 / 423

List comprehensions

Definirea listelor prin proprietățile elementelor, similar unei specificații matematice

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

215 / 423

Cuprins

3.33pt

- 1 Introducere
- 2 Evaluare
- 3 Tipare
- 4 Sinteza de tip

216 / 423

Evaluare

- Evaluare lenesă: parametri evaluați la cerere, cel mult o dată, eventual parțial, în cazul obiectelor structurate
- Funcții restricte!

```
1 f (x, y) z = x + x
2
3 f (2 + 3, 3 + 5) (5 + 8)
4 → (2 + 3) + (2 + 3)
5 → 5 + 5 -- reutilizam rezultatul primei evaluari
6 → 10
```

217 / 423

Pași în aplicarea funcțiilor I

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

Exemplu preluat din Thompson (1999)

218 / 423

Pași în aplicarea funcțiilor II

- Pattern matching: evaluarea parametrilor suficient cât să se constate (ne-)potrivirea cu pattern-ul
- Evaluarea gărzilor (!)
- Evaluarea variabilelor locale, la cerere (where, let)

219 / 423

Pași în aplicarea funcțiilor III

```
1 f 3 5
2 ?? notNil xs
3 ?? where
4 ?? xs = [3 .. 5]
5 ?? → 3 : [4 .. 5]
6 ?? → notNil (3 : [4 .. 5])
7 ?? → True
8 → front xs
9 where
10 xs = 3 : [4 .. 5]
11 → 3 : 4 : [5]
12 → front (3 : 4 : [5])
13 → 3 + 4
14 → 7
```

220 / 423

Consecințe

- Evaluarea parțială a obiectelor structurate (liste etc.)
- Liste, implicit, ca fluxuri!

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

221 / 423

Cuprins

3.33pt

- 1 Introducere
- 2 Evaluare
- 3 Tipare
- 4 Sinteza de tip

222 / 423

Tipuri

- Tipuri ca mulțimi de valori:
 - Bool = {True, False}
 - Natural = {0, 1, 2, ...}
 - Char = {'a', 'b', 'c', ...}
- 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

223 / 423

Exemple de tipuri

```
1 5           :: Integer
2 'a'        :: Char
3 inc        :: Integer -> Integer
4 [1,2,3]    :: [Integer]
5 (True, "Hello") :: (Bool, (Char))
```

224 / 423

Tipuri de bază

- Tipurile ale căror valori **nu** pot fi descompuse

Exemple:

- Bool
- Char
- Integer
- Int
- Float

225 / 423

Constructorii de tip

"Funcții" de tip, care generează tipuri noi pe baza celor existente

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

226 / 423

Tipurile funcțiilor

Constructorul "->" asociativ la dreapta:
Integer -> Integer -> Integer
≡ Integer -> (Integer -> Integer)

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

227 / 423

Polimorfism

- Parametric:** manifestarea **aceluiași** comportament pentru parametri de tipuri **diferite**. Exemplu: idd

- Ad-hoc:** manifestarea unor comportamente **diferite** pentru parametri de tipuri **diferite**. Exemplu: (==)

228 / 423

Constructorul de tip Natural I

Definit de utilizator

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

229 / 423

Constructorul de tip Natural II

Definit de utilizator

- Constructor de tip: Natural
 - nular
 - se contundă cu tipul pe care-l construiești

Constructorii de date:

- Zero: nular
- Succ: unar

- Constructorii de date ca **funcții**, utilizabile în **pattern matching**

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

230 / 423

Constructorul de tip Pair I

Definit de utilizator

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

231 / 423

Constructorul de tip Pair II

Definit de utilizator

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

- Constructor de date: P, binar

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

232 / 423

Uniformitatea reprezentării tipurilor

```
1 data Integer = ... | -2 | -1 | 0 | 1 | 2 | ...
2
3 data Char = 'a' | 'b' | 'c' | ...
4
5 data [a] = [] | a : [a]
6
7 data (a, b) = (a, b)
```

233 / 423

Cuprins

3.33pt

23 Introducere

23 Evaluare

23 Tipare

25 Sinteza de tip

234 / 423

Sinteza de tip

- Definiție: determinarea **automată** a tipului unei expresii, pe baza unor reguli precise
- Adnotările **explicite** de tip, deși posibile, **necesare** în majoritatea cazurilor
- Dependentă de:
 - componentele expresiei
 - contextul lexical al expresiei
- Reprezentarea tipurilor prin **expresii** de tip:
 - constante de tip: tipuri de bază (Int)
 - variabile de tip: pot fi legate la orice expresii de tip (a)
 - aplicații ale constructorilor de tip asupra expresiilor de tip ([a])

235 / 423

Reguli simplificate de sinteză de tip I

- Forma generală:

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

- Funcție:

$$\frac{\text{Var} :: a \quad \text{Expr} :: b}{\backslash \text{Var} \rightarrow \text{Expr} :: a \rightarrow b} \quad (\text{TLambda})$$

- Aplicație:

$$\frac{\text{Expr1} :: a \rightarrow b \quad \text{Expr2} :: a}{(\text{Expr1 Expr2}) :: b} \quad (\text{TApp})$$

236 / 423

Reguli simplificate de sinteză de tip II

- Operatorul +:

$$\frac{\text{Expr1} :: \text{Int} \quad \text{Expr2} :: \text{Int}}{\text{Expr1} + \text{Expr2} :: \text{Int}} \quad (\text{T+})$$

- Literalii întregi:

$$0, 1, 2, \dots :: \text{Int} \quad (\text{TInt})$$

237 / 423

Exemple de sinteză de tip I

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

g :: a -> (g 3) + 1 :: b
f :: a -> b                (TLambda)

(g 3) :: Int    1 :: Int
(g 3) + 1 :: Int (T+, TInt)

b = Int

g :: c -> d    3 :: c
(g 3) :: d    (TApp)

a -> c -> d, c = Int, d = Int

f :: (Int -> Int) -> Int
```

238 / 423

Exemple de sinteză de tip II

```
fix f = f (fix f)

f :: a -> f (fix f) :: b
fix :: a -> b                (TLambda)

f :: c -> d    (fix f) :: c
f (fix f) :: d (TApp)

a -> c -> d, b = d

fix :: e -> g    f :: e
(fix f) :: g    (TApp)

a -> b -> e -> g, a = e, b = g, c = g

f :: (c -> d) -> b -> (g -> g) -> g
```

239 / 423

Exemple de sinteză de tip III

$f\ x = (x\ x)$

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

Ecuatia $c \rightarrow d = c$ **nu** are soluție, deci funcția **nu** poate fi tipată.

240 / 423

Unificare I

- Sinteza de tip presupune **legarea** variabilelor de tip în scopul **unificării** diverselor expresii de tip obținute
- Unificare = procesul de identificare a valorilor **variabilelor** din 2 sau mai multe expresii, astfel încât **substituirea** variabilelor prin valorile asociate să conducă la **coincidenta** expresiilor
- Substituție = mulțime de **legări** variabilă-valoare

241 / 423

Unificare II

Exemplu:

- **Expresii:**
 - $t1 = (a, [b])$
 - $t2 = (Int, c)$
- **Substituții:**
 - $S1 = \{a \leftarrow Int, b \leftarrow Int, c \leftarrow [Int]\}$
 - $S2 = \{a \leftarrow Int, c \leftarrow [b]\}$
- **Forme comune:**
 - $t1/S1 = t2/S1 = (Int, [Int])$
 - $t1/S2 = t2/S2 = (Int, [b])$

Most general unifier (MGU) = cea mai **generală** substituție sub care expresiile unifică. Exemplu: $S2$.

242 / 423

Unificare III

- **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*).
- **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.

243 / 423

Tip principal

Exemplu:

- **Funcție:** $\backslash x \rightarrow x$
- **Tipuri corecte:**
 - $Int \rightarrow Int$
 - $Bool \rightarrow Bool$
 - $a \rightarrow a$
- Unele tipuri se obțin prin **instanțierea** altora.

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

244 / 423

Rezumat

- Evaluare leneșă
- Tipare statică și tare, anterioară evaluării

245 / 423

Bibliografie

Thompson, S. (1999). *Haskell: The Craft of Functional Programming*. Ediția a doua. Addison-Wesley.

246 / 423

Partea VIII

Evaluare leneșă în Haskell

247 / 423

Cuprins

3.33pt

248 / 423

Suma pătratelor

Suma pătratelor numerelor naturale până la n , ca sumă a elementelor unei **liste**:

```
1 sum (map (^2) [1..n])
2 → sum (map (^2) 1 : [2..n])
3 → sum (1^2 : (map (^2) [2..n]))
4 → 1^2 + sum (map (^2) [2..n])
5 → 1 + sum (map (^2) [2..n])
6 ...
7 → 1 + (4 + sum (map (^2) [3..n]))
8 ...
9 → 1 + (4 + (9 + ... + n^2))
```

Nicio listă **nu** este efectiv construită în timpul evaluării.

249 / 423

Elementul minim al unei liste I

Elementul minim al unei liste, drept prim element al acesteia, după **sortarea** prin inserție (Thompson, 1999):

```
34 ins x [] = [x]
35 ins x (h : t)
36 | x <= h = x : h : t
37 | otherwise = h : (ins x t)
38
39 isort [] = []
40 isort (h : t) = ins h (isort t)
41
42 minList 1 = head (isort 1)
```

250 / 423

Elementul minim al unei liste II

```
45 minList [3, 2, 1]
46 = head (isort [3, 2, 1])
47 = head (isort (3 : [2, 1]))
48 = head (ins 3 (isort [2, 1]))
49 = head (ins 3 (isort (2 : [1])))
50 = head (ins 3 (ins 2 (isort [1])))
51 = head (ins 3 (ins 2 (isort (1 : []))))
52 = head (ins 3 (ins 2 (ins 1 (isort []))))
53 = head (ins 3 (ins 2 (ins 1 [])))
54 = head (ins 3 (ins 2 (1 : [])))
55 = head (ins 3 (1 : ins 2 []))
56 = head (1 : (ins 3 (ins 2 [])))
57 = 1
```

Lista **nu** este efectiv sortată, minimul fiind, pur și simplu, adus în fața acesteia și întors.

251 / 423

Accesibilitatea într-un graf orientat

Accesibilitatea între două noduri dintr-un graf \Leftrightarrow existența elementelor în mulțimea **tuturor** căilor dintre cele două noduri (Thompson, 1999):

```
75 routes source dest graph explored
76 | source == dest = [[source]]
77 | otherwise = [ source : path
78 | neighbor <- neighbors source
79 | path <- routes neighbor dest
80 | graph (source : explored)
81 ]
82 accessible source dest graph =
83 (routes source dest graph []) /- []
```

Backtracking desfășurat doar până la determinarea **primului** element al listei de căi.

252 / 423

Evaluarea leneșă

- **Programare orientată spre date:** exprimarea unor prelucrări în termenii unor operații pe **structuri de date**, posibil **niciodată** generate complet (suma pătratelor, sortare)
- **Backtracking eficient:** găsirea unui obiect cu o anumită proprietate, prin generarea aparentă a **tuturor** celor care îndeplinesc proprietatea respectivă (accesibilitatea în graf)
- **Pilon al modularității eficiente** — prelucrări **distincte** ale unei structuri, aplicate într-o **singură** parcurgere!

253 / 423

Studiu de caz

Biblioteca de parsare (Thompson, 1999)

254 / 423

Bibliografie

Thompson, S. (1999). *Haskell: The Craft of Functional Programming*. Ediția a doua. Addison-Wesley.

255 / 423

Partea IX

Clase în Haskell

256 / 423

Cuprins

3.33pt

- Clase
- Aplicație pentru clase

257 / 423

Cuprins

3.33pt

- Clase
- Aplicație pentru clase

258 / 423

Motivație

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

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

259 / 423

Varianta 1 I

Funcții dedicate fiecărui tip

```
1 show4Bool True  - "True"
2 show4Bool False - "False"
3
4 show4Char c     - "[" ++ [c] ++ "]"
5
6 show4String s   - "\"" ++ s ++ "\""
```

260 / 423

Varianta 1 II

Funcții dedicate fiecărui tip

- Funcția `showNewLine`, care adaugă caracterul "linie nouă" la reprezentarea ca șir:

```
1 showNewLine x = (show... x) ++ "\n"
```
- `showNewLine` **nu** poate fi polimorfică
→ `showNewLine4Bool`, `showNewLine4Char` etc.
- Alternativ, trimiterea ca **parametru** a funcției `show*`, corespunzătoare:

```
1 showNewLine sh x = (sh x) ++ "\n"
2 showNewLine4Bool = showNewLine show4Bool
```
- Prea general**, fiind posibilă trimiterea unei funcții cu alt comportament, în măsura în care respectă tipul

261 / 423

Varianta 2 I

Supraîncărcarea funcției

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

```
1 class Show a where
2   show :: a -> String
3   ...
```
- Precizarea **aderenței** unui tip la această mulțime:

```
1 instance Show Bool where
2   show True  - "True"
3   show False - "False"
4
5 instance Show Char where
6   show c - "[" ++ [c] ++ "]"
```
- Funcția `showNewLine` **polimorfică!**

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

262 / 423

Varianta 2 II

Supraîncărcarea funcției

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

```
1 show      :: Show a => a -> String
2 showNewLine :: Show a => a -> String
```
- "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: `Show a`
- Propagarea** constrângerilor din contextul lui `show` către contextul lui `showNewLine`

263 / 423

Varianta 2 III

Supraîncărcarea funcției

- Contexte utilizabile și la **instanțiere**:

```
1 instance (Show a, Show b) => Show (a, b) where
2   show (x, y) = "(" ++ (show x)
3               ++ "," ++ (show y)
4               ++ ")"
```
- "Ori de câte ori tipurile `a` și `b` aparțin clasei `Show`, tipul `(a, b)` îi aparține de asemenea."

264 / 423

Clase

- Clasă = **mulțime** de tipuri ce supraîncarcă operațiile specifice clasei
- Modalitate structurată de control al polimorfismului **ad-hoc**
- Exemplu: clasa `Show`, cu operația `show`

265 / 423

Instanțe ale claselor

- Instanță = **tip** care supraîncarcă operațiile clasei
- Exemplu: tipul `Bool`, în raport cu clasa `Show`

266 / 423

Clase predefinite I

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

- Possibilitatea scrierii de definiții **implicit** (v. liniile 7–8)
- Necesitatea suprascrierii **cel puțin unuia** dintre cei doi operatori ai clasei `Eq`, pentru instanțierea corectă

267 / 423

Clase predefinite II

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

- Contexte utilizabile și la **definirea** unei clase
- Moștenirea** claselor, cu preluarea operațiilor din clasa moștenită
- Necesitatea** aderenței la clasa `Eq` în momentul instanțierii clasei `Ord`
- Suficienta** supradefinirii lui `(<=)` la instanțiere

268 / 423

Clase Haskell vs. POO

Haskell

- Mulțimi de **tipuri**
- Instanțierea** claselor de către tipuri
- Implementarea operațiilor **în afara** definiției tipului

POO

- Mulțimi de **obiecte**: *tipuri*
- Implementarea** interfețelor de către clase
- Implementarea operațiilor **în cadrul** definiției tipului

Clase Haskell ~ Interfețe Java

269 / 423

Cuprins

3.33pt

- Clase
- Aplicație pentru clase

270 / 423

invert I

Fie constructorii de tip:

```
3 data Pair a = P a a
4
5 data NestedList a
6   = Atom a
7   | List [NestedList a]
```

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

271 / 423

invert II

```
5 class Invert a where
6   invert :: a -> a
7   invert - id
8
9 instance Invert (Pair a) where
10  invert (P x y) = P y x
11
12 instance Invert a => Invert (NestedList a) where
13   invert (Atom x) = Atom (invert x)
14   invert (List x) = List $ reverse $ map invert x
15
16 instance Invert a => Invert [a] where
17   invert lst = reverse $ map invert lst
```

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

272 / 423

contents I

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

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

- a este tipul unui **container**, ca `NestedList b`
- Elementele listei întoarse sunt cele din **container**
- Cum **precizăm** tipul acestora, b?

273 / 423

contents II

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

- Conform definiției clasei:
1 contents :: Container [a] => [a] -> [[a]]
- Conform supraîncărcării funcției (`id`):
1 contents :: Container [a] => [a] -> [a]
- Ecuația [a] = [[a]] **nu are soluție — eroare!**

274 / 423

contents III

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

- Conform definiției clasei:
1 contents :: Container [a] => [a] -> [b]
- Conform supraîncărcării funcției (`id`):
1 contents :: Container [a] => [a] -> [a]
- Ecuația [a] = [b] **are soluție** pentru a = b
- Dar, [a] -> [a] **insuficient de general** în raport cu [a] -> [b] — **eroare!**

275 / 423

contents IV

Soluție: clasa primește **constructorul** de tip, și nu tipul container propriu-zis

```
5 class Container t where
6   contents :: t a -> [a]
7
8 instance Container Pair where -- nu (Pair a)!
9   contents (P x y) = [x, y]
10
11 instance Container NestedList where
12   contents (Atom x) = [x]
13   contents (List l) = concatMap contents l
14
15 instance Container [] where
16   contents = id
```

276 / 423

Contexte I

```
6 fun1 :: Eq a => a -> a -> a -> a
7 fun1 x y z = if x == y then x else z
8
9 fun2 :: (Container a, Invert (a b), Eq (a b))
10 => (a b) -> (a b) -> [b]
11 fun2 x y = if (invert x) == (invert y)
12 then contents x
13 else contents y
14
15 fun3 :: Invert a => [a] -> [a] -> [a]
16 fun3 x y = (invert x) ++ (invert y)
17
18 fun4 :: Ord a => a -> a -> a -> a
19 fun4 x y z = if y == y
20 then z
21 else if x > y
22 then x
23 else y
```

277 / 423

Contexte II

- **Simplificarea** contextului lui `fun3`, de la `Invert [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`

278 / 423

Rezumat

- **Clase** = multimi de tipuri care supraîncărcă anumite operații
- Formă de polimorfism **ad-hoc**: tipuri diferite, comportamente diferite
- **Instanțierea** unei clase = aderarea unui tip la o clasă
- **Derivarea** unei clase = impunerea condiției ca un tip să fie deja membru al clasei părinte, în momentul instanțierii clasei copil, și moștenirea operațiilor din clasa părinte
- **Context** = mulțimea constrângerilor asupra tipurilor din semnatura unei funcții, în termenii aderenței la diverse clase

279 / 423

Partea X

Paradigma funcțională vs. paradigma imperativă

280 / 423

Cuprins

3.33pt

- 28 Efecte laterale și transparență referențială
- 28 Aspecte comparative
- 30 Aplicații ale programării funcționale

281 / 423

Cuprins

3.33pt

- 28 Efecte laterale și transparență referențială
- 28 Aspecte comparative
- 30 Aplicații ale programării funcționale

282 / 423

Efecte laterale (side effects)

Definiție

- În expresia $2 + (i = 3)$, subexpresia $(i = 3)$:
 - produce **valoarea** 3, conducând la rezultatul 5 pentru întreaga expresie
 - are **efectul lateral** de inițializare a lui i cu 3
- Inerente în situațiile în care programul interacționează cu exteriorul — **I/O!**

283 / 423

Efecte laterale (side effects)

Consecințe

- 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 $x + (x + 1) = 0 + 1 = 1$
- Adunare **necomutativă!**
- Importanța **ordinii de evaluare!**
- Dependente **implicite**, dificil de desprins și posibile generatoare de bug-uri

284 / 423

Transparență referențială

- Zeus la greci ≡ Jupiter la romani (Woodridge și Jennings, 1995)

- 1 Cazul 1:
 - "Zeus este fiul lui Cronos"
 - "Jupiter este fiul lui Cronos"
 - **aceeași** semnificație
- 2 Cazul 2:
 - "Ionel știe că Zeus este fiul lui Cronos"
 - "Ionel știe că Jupiter este fiul lui Cronos"
 - **altă** semnificație

- **Transparență referențială** = **independența** înțelesului unei propoziții în raport cu modul de desemnare a obiectelor — cazul 1.

285 / 423

Expresii transparente referențial

One of the most useful properties of expressions is [...] **referential transparency**. In essence this means that if we wish to find the value of an expression which contains a sub-expression, the only thing we need to know about the sub-expression is its **value**. Any other features of the sub-expression, such as its internal structure, the number and nature of its components, the order in which they are evaluated or the colour of the ink in which they are written, are **irrelevant** to the value of the main expression.

Christopher Strachey,
Fundamental Concepts in Programming Languages

286 / 423

Expresii transparente referențial

The only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value. Moreover, the value of an expression is, within certain limits, the **same** whenever it occurs.

Joseph Stoy,
Denotational semantics: the Scott-Strachey approach to programming language theory

287 / 423

Expresii transparente referențial

- Expresii (ne)transparente referențial:
 - `x-- + ++x`: nu, valoarea depinde de ordinea de evaluare
 - `x = x + 1; nu`, două evaluări consecutive vor produce rezultate diferite
 - `x`: da, presupunând că `x` nu este modificată în altă parte
- Efecte laterale => opacitate referențială!

288 / 423

Funcții transparente referențial

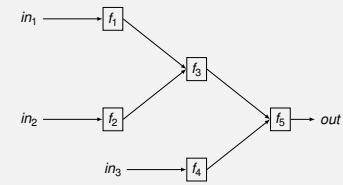
Funcție transparentă referențial: rezultatul întors depinde **exclusiv** de parametri

```
1 int transparent(int x) { 5 int g = 0;
2   return x + 1;        6
3 }                      7 int opaque(int x) {
                        8   return x + ++g;
                        9 }
11 // opaque(3) != opaque(3)
```

- Funcții transparente: `log`, `sin` etc.
- Funcții opace: `time`, `read` etc.

289 / 423

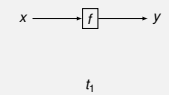
Înlănțuirea funcțiilor



290 / 423

Calcul fără stare

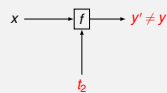
Dependența ieșirii de intrare, nu și de timp



291 / 423

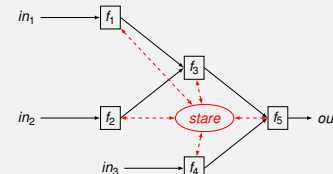
Calcul cu stare

Dependența ieșirii de intrare, și de timp



292 / 423

Calcul cu stare



Stare = mulțimea valorilor variabilelor, la un anumit moment, ce pot influența rezultatul evaluării aceleiași expresii.

293 / 423

Avantajele transparenței referențiale

- Lizibilitatea codului
- Demonstrarea formală a corectitudinii programului
- Optimizare prin reordonarea instrucțiunilor de către compilator, și prin *caching*
- Paralelizare masivă, în urma eliminării modificărilor concurente
- Evaluare leneșă, imposibilă în absența unei garanții despre menținerea valorii unei expresii, la momente diferite!

294 / 423

Cuprins

3.33pt

- Efecte laterale și transparență referențială
- Aspecte comparative
- Aplicații ale programării funcționale

295 / 423

Explicitarea sensului programelor

```
1: procedure MINLIST(L, n)
2:   min ← L[1]
3:   i ← 2
4:   while i ≤ n do
5:     if L[i] < min then
6:       min ← L[i]
7:     end if
8:     i ← i + 1
9:   end while
10:  return min
11: end procedure
```

```
1 minList [h] - h
2 minList (h : t) - min h $ minList t
```

296 / 423

Verificarea programelor

Funcțional

- Definiția unei funcții = proprietate pe care o îndeplinește
- Aplicabilitatea directă a metodelor, e.g inducție structurală

Imperativ

- Necesitatea adnotării programelor cu descriptori de stare
- Necesitatea aplicării de metode indirecte, bazate pe adnotări

297 / 423

Funcții și variabile

Funcțional

- Funcții cu aceleași valori pentru aceiași parametri
- Variabile nemodificabile

Imperativ

- Funcții cu valori diferite pentru aceiași parametri
- Variabile modificabile

298 / 423

Evaluare leneșă

- Posibilă doar în absența efectelor laterale
- Modularitate eficientă, separație producător-consumator
- Fluxuri

299 / 423

Problema expresivității

	Extinderea tipurilor	Extinderea operațiilor
Funcțional	Dificilă	Ușoară
OO	Ușoară	Dificilă

300 / 423

Alte aspecte

- Funcționale ca structuri de control
- Tipuri algebrice
- Polimorfism

301 / 423

Cuprins

3.33pt

- Efecte laterale și transparență referențială
- Aspecte comparative
- Aplicații ale programării funcționale

302 / 423

Aplicații ale programării funcționale I

- PureScript, translator Haskell → JavaScript: (<http://www.purescript.org/>)
- Yesod Web Framework for Haskell (<http://www.yesodweb.com/>)
- Back-end Haskell pentru Android (<https://wiki.haskell.org/Android>)
- Yampa, EDSL în Haskell pentru Functional Reactive Programming (FRP) (<https://wiki.haskell.org/Yampa>)

303 / 423

Aplicații ale programării funcționale II

- Programare paralelă
(<http://chimera.labs.oreilly.com/books/1230000000929>)
- Utilizare Haskell la Google și Facebook:
(<https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/>)
- Construcții lambda și funcționale, introduse în C++, Java 8, Swift
(<https://developer.apple.com/swift/>)

304/423

Bibliografie

- Thompson, S. (2011). *Haskell: The Craft of Functional Programming*. Ediția a treia. Addison-Wesley.
- Wooldridge, M. și Jennings, N. R. (1995). Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10:115–152.

305/423

Partea XI

Limbajul Prolog

306/423

Cuprins

3.33pt

- 1 Axiome și reguli
- 2 Procesul de demonstrare
- 3 Controlul execuției
- 4 Caracteristici

307/423

Cuprins

3.33pt

- 1 Axiome și reguli
- 2 Procesul de demonstrare
- 3 Controlul execuției
- 4 Caracteristici

308/423

Un prim exemplu

```
1 % constante -> litera mica
2 parent(andrei, bogdan).
3 parent(andrei, bianca).
4 parent(bogdan, cristi).
5
6 % variabile -> litera mare
7 grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

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

309/423

Interogări

```
1 ?- parent(andrei, bogdan).
2 true.
3
4 ?- parent(andrei, cristi).
5 false.
6
7 ?- parent(andrei, X).
8 X = bogdan ;
9 X = bianca.
10
11 ?- grandparent(X, Y).
12 X = andrei,
13 Y = cristi ;
14 false.
```

- "?" → oprire după **primul** răspuns
- ";" → solicitarea **următorului** răspuns

310/423

Concatenarea a două liste

```
1 :- append(L1, L2, Res).
2 append([], L, L).
3 append([_:_], L, [_:Res]) :- append(L, Res).
```

Calcul

```
1 ?- append([1], [2], Res).
2 Res = [1, 2].
```

Generare

```
1 ?- append(L1, L2, [1, 2]).
2 L1 = [],
3 L2 = [1, 2] ;
4 L1 = [1],
5 L2 = [2] ;
6 L1 = [1, 2],
7 L2 = [] ;
8 false.
```

Estomparea granițelor dintre "intrare" și "ieșire"

311/423

Cuprins

3.33pt

- 1 Axiome și reguli
- 2 Procesul de demonstrare
- 3 Controlul execuției
- 4 Caracteristici

312/423

Exemplul genealogic I

```
S = 0
G = {gp(X, Y)}
      ↓
gp(X1, Y1) :-
  p(X1, Z1), p(Z1, Y1)
      ↓
S = {X = X1, Y = Y1}
G = {p(X1, Z1), p(Z1, Y1)}
      ↓
p(andrei, bogdan) p(andrei, bianca) p(bogdan, cristi)
      ↓
      ↓
      ↓
```

313/423

Exemplul genealogic II

```
...
      ↓
p(andrei, bogdan)
      ↓
S = {X = X1, Y = Y1, X1 = andrei, Z1 = bogdan}
G = {p(bogdan, Y1)}
      ↓
p(bogdan, cristi)
      ↓
S = {X = X1, Y = Y1, X1 = andrei, Z1 = bogdan, Y1 = cristi}
G = 0
      ↓
succes
```

314/423

Exemplul genealogic III

```
...
      ↓
p(andrei, bianca)
      ↓
S = {X = X1, Y = Y1, X1 = andrei, Z1 = bianca}
G = {p(bianca, Y1)}
      ↓
eșec
```

315/423

Exemplul genealogic IV

```
...
      ↓
p(bogdan, cristi)
      ↓
S = {X = X1, Y = Y1, X1 = bogdan, Z1 = cristi}
G = {p(cristi, Y1)}
      ↓
eșec
```

316/423

Pași în demonstrare I

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

317/423

Pași în demonstrare II

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

318/423

Observații

- Ordinea **clauzelor** în program
- Ordinea **premiselor** în cadrul regulilor
- Recomandare: premisele **mai ușor** de satisfăcut, primele — exemplu: axiome

319/423

Strategii de control

Forward chaining (data-driven)

- Premise → scop
- Derivarea **tuturor** concluziilor posibile
- **Oprire** la obținerea scopului (scopurilor)

Backward chaining (goal-driven)

- Scop → premise
- Utilizarea **exclusivă** a regulilor care pot contribui efectiv la satisfacerea scopului
- Satisfacerea **premiselor** acestor reguli ș.a.m.d.

320 / 423

Cuprins

3.33pt

- 18 Axiome și reguli
- 19 Procesul de demonstrare
- 18 Controlul execuției
- 58 Caracteristici

321 / 423

Minimul a două numere I

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

322 / 423

Minimul a două numere II

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

323 / 423

Minimul a două numere III

Condiții mutual exclusive: $X < Y$ și $X > Y$ — cum putem **elimina** redundanța?

```
12 min4(X, Y, X) :- X < Y.
13 min4(X, Y, Y).
1
2 ?- min4(1+2, 3+4, M).
3 M = 3+4.
```

Gresit!

324 / 423

Minimul a două numere IV

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

```
15 min5(X, Y, X) :- X < Y, !.
16 min5(X, Y, Y).
1
2 ?- min5(1+2, 3+4, M).
3 M = 1+2.
```

325 / 423

Operatorul cut I

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

326 / 423

Operatorul cut II

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

Backtracking doar la **dreapta** operatorului

327 / 423

Operatorul cut III

```
1 ?- pair(X, Y).
2 X = mary,
3 Y = john ;
4 X = mary,
5 Y = bill ;
6 X = ann,
7 Y = john ;
8 X = ann,
9 Y = bill ;
10 X = bella,
11 Y = harry.
1
2 ?- pair2(X, Y).
3 X = mary,
4 Y = john ;
5 Y = bill.
```

328 / 423

Cuprins

3.33pt

- 18 Axiome și reguli
- 19 Procesul de demonstrare
- 18 Controlul execuției
- 58 Caracteristici

329 / 423

Programare logică

- Reprezentare **simbolică**
- Stil **declarativ**
- **Separarea** datelor de procesul de inferență, încorporat în limbaj
- **Uniformitatea** reprezentării axiomelor și a regulilor de derivare
- Reprezentarea **modularizată** a cunoștințelor
- Posibilitatea modificării **dinamice** a programelor, prin adăugarea și retragerea axiomelor și a regulilor

330 / 423

Prolog I

- Bazat pe logica cu predicate de ordin 1, **restricționată**
- "Calculul": satisfacerea de scopuri, prin **reducere la absurd**
- Regula de inferență: **rezoluția**
- Strategia de control, în evoluția demonstrațiilor:
 - **backward chaining**: de la scop către axiome
 - parcurgere în **adâncime**, în arborele de derivare
- Parcurgerea în **adâncime**:
 - pericolul coborârii pe o cale infinită, ce nu conține soluția — strategie **incompletă**
 - **eficiență** sporită în utilizarea **spațiului**

331 / 423

Prolog II

- Exclusiv clauze **Horn**:

$$A_1 \wedge \dots \wedge A_n \Rightarrow A \quad (\text{Regulă})$$
$$true \Rightarrow B \quad (\text{Axiomă})$$

- Absența **negațiilor** explicite — desprinderea falsității pe baza imposibilității de a demonstra
- Ipoteza lumii **închise** (*closed world assumption*): ceea ce nu poate fi demonstrat este **fals**
- Prin opoziție, ipoteza lumii **deschise** (*open world assumption*): nu se poate afirma **nimic** despre ceea ce nu poate fi demonstrat

332 / 423

Negația ca eșec

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

- $P \rightarrow$ atom — exemplu: boy(john)
- **P satisfiabil**:
 - eșecul **primei** reguli, din cauza lui fail
 - abandonarea celei **de-a doua** reguli, din cauza lui !
 - rezultat: nott(P) **nesatisfiabil**
- **P nesatisfiabil**:
 - eșecul **primei** reguli
 - succesul celei **de-a doua** reguli
 - rezultat: nott(P) **satisfiabil**

333 / 423

Rezumat

- Date: clauze **Horn**
- Regula de inferență: **rezoluție**
- Strategia de căutare: **backward chaining**, dinspre concluzie spre ipoteze
- Posibilități **generative**, pe baza unui anumit stil de scriere a regulilor

334 / 423

Partea XII

Logica propozițională și logica cu predicate de ordinul I

335 / 423

Cuprins

3.33pt

- Introducere
- Logica propozițională
 - Sintaxă și semantică
 - Satisfiabilitate și validitate
 - Derivabilitate
 - Inferență și demonstrație
 - Rezoluție
- Logica cu predicate de ordinul I
 - Sintaxă și semantică
 - Forma clauzală
 - Unificare

336 / 423

Cuprins

3.33pt

- Introducere
- Logica propozițională
 - Sintaxă și semantică
 - Satisfiabilitate și validitate
 - Derivabilitate
 - Inferență și demonstrație
 - Rezoluție
- Logica cu predicate de ordinul I
 - Sintaxă și semantică
 - Forma clauzală
 - Unificare

337 / 423

Logică

- Scop: reducerea efectuării de raționamente la **calcul**
- Problemele de **decidabilitate** din logică: stimulente pentru dezvoltarea modelelor de calculabilitate
- Împrumuturi **reciproce** între domeniile logicii și calculabilității:
 - proiectarea și verificarea programelor → logică
 - principiile logice → proiectarea limbajelor de programare

(Harrison, 2009)

338 / 423

Rolurile logicii

- Descrierea** proprietăților obiectelor, într-o manieră neambiguă, prin intermediul unui **limbaj**, cu următoarele componente:
 - sintaxă**: modalitatea de construcție a expresiilor
 - semantică**: semnificația expresiilor construite
- Deducerea** de noi proprietăți, pe baza celor existente

339 / 423

Cuprins

3.33pt

- Introducere
- Logica propozițională
 - Sintaxă și semantică
 - Satisfiabilitate și validitate
 - Derivabilitate
 - Inferență și demonstrație
 - Rezoluție
- Logica cu predicate de ordinul I
 - Sintaxă și semantică
 - Forma clauzală
 - Unificare

340 / 423

Logica propozițională

- Expresia din limbaj: **propoziția**, corespunzătoare unei afirmații, ce poate fi adevărată sau falsă
- Exemplu: "Telefonul sună și câinele latră."
- Accepții** asupra unei propoziții:
 - secvența de **simboluri** utilizate sau
 - înțelesul** propriu-zis al acesteia, într-o **interpretare**
- Valoarea de adevăr** a unei propoziții — determinată de valorile de adevăr ale propozițiilor **constituente**

(Genesereth, 2010)

341 / 423

Cuprins

3.33pt

- Introducere
- Logica propozițională
 - Sintaxă și semantică
 - Satisfiabilitate și validitate
 - Derivabilitate
 - Inferență și demonstrație
 - Rezoluție
- Logica cu predicate de ordinul I
 - Sintaxă și semantică
 - Forma clauzală
 - Unificare

342 / 423

Sintaxă

- 2 categorii de propoziții
 - simple: fapte **atomice**: "Telefonul sună.", "Câinele latră."
 - compușe: **relați** între propoziții mai simple: "Telefonul sună și câinele latră."
- Propoziții simple: p, q, r, \dots
- Negații: $\neg \alpha$
- Conjunții: $(\alpha \wedge \beta)$
- Disjunții: $(\alpha \vee \beta)$
- Implicații: $(\alpha \Rightarrow \beta)$
- Echivalente: $(\alpha \Leftrightarrow \beta)$

343 / 423

Semantică I

- Atribuirea de **valori de adevăr** propozițiilor
- Accent pe **relațiile** dintre propozițiile compuse și cele constituente
- Pentru explicitarea legăturilor, utilizarea conceptului de **interpretare**

344 / 423

Semantică II

- Interpretare** = mulțime de **asocieri** între fiecare propoziție **simplă** din limbaj și o valoare de adevăr
- Exemplu:

Interpretarea I :	Interpretarea J :
<ul style="list-style-type: none"> $p^I = false$ $q^I = true$ $r^I = false$ 	<ul style="list-style-type: none"> $p^J = true$ $q^J = true$ $r^J = true$
- Sub o interpretare fixată, **dependența** valorii de adevăr a unei propoziții compuse de valorile de adevăr ale celor constituente

345 / 423

Semantică III

- Negație:

$$(\neg \alpha)^I = \begin{cases} true & \text{dacă } \alpha^I = false \\ false & \text{altfel} \end{cases}$$
- Conjunție:

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

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

346 / 423

Semantică IV

- Implicație:

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

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

347 / 423

Evaluare

- Evaluare** = determinarea **valorii de adevăr** a unei propoziții, sub o interpretare, prin aplicarea regulilor semantice anterioare
- Exemplu:

Interpretarea I :	Propoziția: $\phi = (p \wedge q) \vee (q \Rightarrow r)$
<ul style="list-style-type: none"> $p^I = false$ $q^I = true$ $r^I = false$ 	$\phi^I = (false \wedge true) \vee (true \Rightarrow false) = false \vee false = false$

348 / 423

Cuprins

3.33pt

- Introducere
- Logica propozițională
 - Sintaxă și semantică
 - Satisfiabilitate și validitate
 - Derivabilitate
 - Inferență și demonstrație
 - Rezoluție
- Logica cu predicate de ordinul I
 - Sintaxă și semantică
 - Forma clauzală
 - Unificare

349 / 423

Satisfiabilitate

- Satisfiabilitate** = proprietatea unei propoziții **adevărate** în **cel puțin o** interpretare

- Metoda tabeli 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	true
false	false	false	false

350 / 423

Validitate

- Validitate** = proprietatea unei propoziții adevărate în **toate** interpretările (**tautologie**)

- Exemplu: $p \vee \neg p$

- Verificabilă prin metoda tabeli de adevăr

351 / 423

Nesatisfiabilitate

- **Nesatisfiabilitate** = proprietatea unei propoziții **false** în **toate** interpretările (**contradicție**)
- Exemplu: $p \Leftrightarrow \neg p$
- Verificabilă prin metoda tabelii de adevăr

352 / 423

Cuprins

3.33pt

Introducere

Logica propozițională

- Sintaxă și semantică
- Satisfiabilitate și validitate
- **Derivabilitate**
- Inferență și demonstrație
- Rezoluție

Logica cu predicate de ordinul I

- Sintaxă și semantică
- Forma clauzală
- Unificare

353 / 423

Derivabilitate I

- **Derivabilitate logică** = proprietatea unei propoziții de a reprezenta **consecința logică** a unei mulțimi de alte propoziții, numite **premise**
- Mulțimea de propoziții Δ derivă propoziția ϕ , dacă și numai dacă **orice** interpretare care satisface toate propozițiile din Δ satisface și ϕ :

$$\Delta \models \phi$$

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

354 / 423

Derivabilitate II

- Verificabilă prin metoda tabelii 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 .

355 / 423

Formulări echivalente ale derivabilității

- $\{\phi_1, \dots, \phi_n\} \models \phi$
- Propoziția $\phi_1 \wedge \dots \wedge \phi_n \Rightarrow \phi$ este **validă**
- Propoziția $\phi_1 \wedge \dots \wedge \phi_n \wedge \neg \phi$ este **nesatisfiabilă**

356 / 423

Cuprins

3.33pt

Introducere

Logica propozițională

- Sintaxă și semantică
- Satisfiabilitate și validitate
- Derivabilitate
- **Inferență și demonstrație**
- Rezoluție

Logica cu predicate de ordinul I

- Sintaxă și semantică
- Forma clauzală
- Unificare

357 / 423

Motivație

- Derivabilitate **logică**: proprietate a propozițiilor
- Derivare **mecanică** (inferență): demers de **calcul**, în scopul verificării derivabilității logice
- Creșterea **exponentială** a numărului de interpretări în raport cu numărul de propoziții simple
- De aici, **diminuarea** valorii practice a metodelor **semantice**, precum cea a tabelii de adevăr
- Alternativ, metode **sintactice**, care manipulează doar reprezentarea simbolică

358 / 423

Inferență

- **Inferență** = derivarea **mecanică** a concluziilor unei mulțimi de premise
- **Regulă de inferență** = **procedură** de calcul capabilă să deriveze concluziile unei mulțimi de premise
- Derivabilitatea mecanică a concluziei ϕ din mulțimea de premise Δ , utilizând regula de inferență **inf**:

$$\Delta \vdash_{inf} \phi$$

359 / 423

Reguli de inferență

- Sabloane **parametrizate** de raționament, formate dintr-o mulțime de **premise** și o mulțime de **concluzii**
- **Modus Ponens** (MP):
$$\frac{\alpha \Rightarrow \beta \quad \alpha}{\beta}$$
- **Modus Tollens**:
$$\frac{\alpha \Rightarrow \beta \quad \neg \beta}{\neg \alpha}$$

360 / 423

Proprietăți ale regulilor de inferență

- **Consistență** (*soundness*): regula de inferență determină **doar** 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

361 / 423

Axiome

- Exemplu: verificarea că $\{p \Rightarrow q, q \Rightarrow r\} \models p \Rightarrow r$
- Caz în care premisele sunt **insuficiente** pentru aplicarea regulilor de inferență
- Soluția: adăugarea de **axiome**, reguli de inferență fără premise
- **Introducerea** implicației (II):
$$\alpha \Rightarrow (\beta \Rightarrow \alpha)$$
- **Distribuirea** implicației (DI):
$$(\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma))$$

362 / 423

Demonstrații I

- **Demonstrație** = **secvență** de propoziții, finalizată cu o concluzie, și conținând:
 - **premise**
 - instanțe ale **axiomelor**
 - rezultate ale aplicării **regulilor de inferență** asupra elementelor precedente din secvență
- **Teoremă** = **concluzia** cu care se încheie o demonstrație

363 / 423

Demonstrații II

- **Procedură de demonstrare** = mecanism de demonstrare, constând din:
 - o mulțime de **reguli de inferență**
 - o **strategie de control**, ce dictează ordinea aplicării regulilor

364 / 423

Demonstrații III

Exemplu: demonstrăm că $\{p \Rightarrow q, q \Rightarrow r\} \vdash p \Rightarrow r$.

1	$p \Rightarrow q$	Premisă
2	$q \Rightarrow r$	Premisă
3	$(q \Rightarrow r) \Rightarrow (p \Rightarrow (q \Rightarrow r))$	II
4	$p \Rightarrow (q \Rightarrow r)$	MP 3, 2
5	$(p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r))$	DI
6	$(p \Rightarrow q) \Rightarrow (p \Rightarrow r)$	MP 5, 4
7	$p \Rightarrow r$	MP 6, 1

365 / 423

Demonstrații IV

- Rezultat: existența unui sistem de inferență **consistent și complet**, bazat pe:
- **axiomele** de mai devers, îmbogățite cu altele
 - regula de inferență **Modus Ponens**

$$\Delta \models \phi \Leftrightarrow \Delta \vdash \phi$$

366 / 423

Cuprins

3.33pt

Introducere

Logica propozițională

- Sintaxă și semantică
- Satisfiabilitate și validitate
- Derivabilitate
- Inferență și demonstrație
- **Rezoluție**

Logica cu predicate de ordinul I

- Sintaxă și semantică
- Forma clauzală
- Unificare

367 / 423

Rezoluție

- **Regulă de inferență** foarte puternică
- Baza unui demonstrator de teoreme **consistent și complet**
- Spațiul de căutare mult mai **mic** ca în abordarea standard (v. subsecțiunea anterioară)
- Lucrul cu propoziții în **forma clauzală**

368 / 423

Forma clauzală I

- **Literal** = propoziție **simplică** (p) sau **negatia** ei ($\neg p$)
- **Expresie clauzală** = **literal** sau **disjuncție** de literali, e.g. $p \vee \neg q \vee r \vee p$
- **Clauză** = **mulțime** de literali dintr-o expresie clauzală, e.g. $\{p, \neg q, r\}$

369 / 423

Forma clauzală II

- **Forma clauzală (forma normală conjunctivă, FNC)** = reprezentarea unei propoziții sub forma unei **mulțimi de clauze**, implicit legate prin conjuncții
- Exemplu: forma clauzală a propoziției $p \wedge (\neg q \vee r) \wedge (\neg p \vee \neg r)$ este $\{\{p\}, \{\neg q, r\}, \{\neg p, \neg r\}\}$.
- Posibilitatea **convertirii** oricărei propoziții în această formă, prin algoritmul următor

370 / 423

Transformarea în formă clauzală I

- Eliminarea **implicațiilor** (I):
$$\alpha \Rightarrow \beta \rightarrow \neg \alpha \vee \beta$$
- Introducerea **negatiilor** în paranteze (N):
$$\neg(\alpha \wedge \beta) \rightarrow \neg \alpha \vee \neg \beta \text{ etc.}$$
- **Distribuirea lui \vee față de \wedge** (D):
$$\alpha \vee (\beta \wedge \gamma) \rightarrow (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$$
- Transformarea expresiilor în **clauze** (C):
$$\phi_1 \vee \dots \vee \phi_n \rightarrow \{\phi_1, \dots, \phi_n\}$$
$$\phi_1 \wedge \dots \wedge \phi_n \rightarrow \{\phi_1\}, \dots, \{\phi_n\}$$

371 / 423

Transformarea în formă clauzală II

- Exemplu: $p \wedge (q \Rightarrow r)$
I $p \wedge (\neg q \vee r)$
C $\{p\}, \{\neg q, r\}$
- Exemplu: $\neg(p \wedge (q \Rightarrow r))$
I $\neg(p \wedge (\neg q \vee r))$
N $\neg p \vee \neg(\neg q \vee r)$
D $\neg p \vee (q \wedge \neg r)$
N $\neg p \vee q \wedge (\neg p \vee \neg r)$
C $\{\neg p, q\}, \{\neg p, \neg r\}$

372 / 423

Rezoluție I

- Ideea:
$$\frac{\{p, q\} \quad \{\neg p, r\}}{\{q, r\}}$$
- "Anularea" lui p cu $\neg p$
- p **adevărată**, $\neg p$ falsă, deci r adevărată
- p **falsă**, deci q adevărată
- **Cel puțin una** dintre q și r adevărată
- Forma generală:
$$\frac{\{p_1, \dots, r, \dots, p_m\} \quad \{q_1, \dots, \neg r, \dots, q_n\}}{\{p_1, \dots, p_m, q_1, \dots, q_n\}}$$

373 / 423

Rezoluție II

- Rezolvent **vid** — **contradicție** între premise:
$$\frac{\{\neg p\} \quad \{p\}}{\{\}}$$
- **Mai mult de 2** rezolvenți posibili — se alege doar unul:
$$\frac{\{p, q\} \quad \{\neg p, \neg q\}}{\{p, \neg p\}} \quad \frac{\{p, q\} \quad \{q, \neg q\}}{\{q, \neg q\}}$$

374 / 423

Rezoluție III

- **Modus Ponens** — caz particular al rezoluției:
$$\frac{p \Rightarrow q \quad \{p\}}{q} \quad \frac{\{p, q\} \quad \{\neg p\}}{\{q\}}$$
- **Modus Tollens** — caz particular al rezoluției:
$$\frac{p \Rightarrow q \quad \{\neg q\}}{\neg p} \quad \frac{\{p, q\} \quad \{\neg q\}}{\{\neg p\}}$$
- **Tranzitivitatea** implicației:
$$\frac{p \Rightarrow q \quad q \Rightarrow r}{p \Rightarrow r} \quad \frac{\{p, q\} \quad \{\neg p, q\}}{\{\neg q, r\}} \quad \frac{\{p, q\} \quad \{\neg q, r\}}{\{\neg p, r\}}$$

375 / 423

Rezoluție IV

- Demonstrarea **nesatisfiabilității** — derivarea clauzei **vide**
- Demonstrarea **derivabilității** concluziei ϕ din premisele ϕ_1, \dots, ϕ_n — demonstrarea **nesatisfiabilității** propoziției $\phi_1 \wedge \dots \wedge \phi_n \wedge \neg \phi$ (reducere la absurd)
- Demonstrarea **validității** propoziției ϕ — demonstrarea **nesatisfiabilității** propoziției $\neg \phi$
- Rezoluția incompletă **generativ**, i.e. concluziile **nu** pot fi derivate direct, răspunsul fiind dat în raport cu o "întrebare" fixată

376 / 423

Rezoluție V

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

- | | | |
|---|-----------------|------------------|
| 1 | $\{p, q\}$ | Premisă |
| 2 | $\{\neg q, r\}$ | Premisă |
| 3 | $\{p\}$ | Concluzie negată |
| 4 | $\{\neg r\}$ | Concluzie negată |
| 5 | $\{q\}$ | 1, 3 |
| 6 | $\{r\}$ | 2, 5 |
| 7 | $\{\}$ | 4, 6 |

377 / 423

Rezoluție VI

- **Teorema rezoluției**: rezoluția propozițională este **consistentă și completă** (nu generativ, v. slide-ul 368):
$$\Delta \models \phi \Leftrightarrow \Delta \vdash \phi$$
- **Terminarea** garantată a procedurii de aplicare a rezoluției: număr **finit** de clauze, număr **finit** de concluzii

378 / 423

Cuprins

3.33pt

- Introducere
- Logica propozițională
 - Sintaxă și semantică
 - Satisfiabilitate și validitate
 - Derivabilitate
 - Inferență și demonstrație
 - Rezoluție
- Logica cu predicate de ordinul I
 - Sintaxă și semantică
 - Forma clauzală
 - Unificare

379 / 423

Logica cu predicate de ordinul I

- Logica propozițională:
 - p : "Andrei este prieten cu Bogdan."
 - q : "Bogdan este prieten cu Andrei."
 - $p \Leftrightarrow q$
 - **Opacitate** în raport cu obiectele și relațiile referite
- **First-order logic** (FOL) = **extensie** a logicii propoziționale, cu explicarea:
 - **obiectelor** din universul problemei
 - **relațiilor** dintre acestea
- FOL:
 - Generalizare: $prieten(x, y)$: " x este prieten cu y ."
 - $\forall x, \forall y. (prien(x, y) \Leftrightarrow prieten(y, x))$
 - Aplicare pe cazuri **particulare**
 - **Transparentă** în raport cu obiectele și relațiile referite (Genesereth, 2010)

380 / 423

Cuprins

3.33pt

- Introducere
- Logica propozițională
 - Sintaxă și semantică
 - Satisfiabilitate și validitate
 - Derivabilitate
 - Inferență și demonstrație
 - Rezoluție
- Logica cu predicate de ordinul I
 - Sintaxă și semantică
 - Forma clauzală
 - Unificare

381 / 423

Sintaxă

Simboluri utilizate

- **Constante**: obiecte particulare din universul discursului: $c, d, andrei, bogdan, \dots$
- **Variable**: obiecte generice: x, y, \dots
- Simboluri **funcționale**: $sucesor(x), +(x, y), \dots$
- Simboluri **relationale** (**predicate**): relații n -are peste obiectele din universul discursului: $divide(x, y), impar(x), \dots$
- **Conectori** logici: \neg, \wedge, \dots
- **Cuantificatori**: \forall, \exists

382 / 423

Sintaxă I

Termeni, atomi, propoziții

- **Termeni** (obiecte):
 - Constante
 - Variabile
 - Aplicații de funcții: $f(t_1, \dots, t_n)$, unde f este un simbol **funcțional** n -ar și t_1, \dots, t_n sunt termeni. Exemple:
 - $sucesor(4)$: succesorul lui 4
 - $+(2, x)$: suma simbolurilor 2 și x

383 / 423

Sintaxă II

Termeni, atomi, propoziții

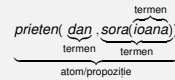
- **Atomi** (relatii): $p(t_1, \dots, t_n)$, unde p este un **predicat** n -ar și t_1, \dots, t_n sunt termeni. Exemple:
 - $impar(3)$
 - $varsta(ion, 20)$
 - $= (+(2, 3), 5)$
- **Propoziții** (fapte) — x variabilă, A atom, α propoziție:
 - Fals, adevărat: \perp, \top
 - Atomi: A
 - Negatii: $\neg\alpha$
 - ...
 - Cuantificări: $\forall x. \alpha, \exists x. \alpha$

384 / 423

Sintaxă III

Termeni, atomi, propoziții

Exemplu: "Dan este prieten cu sora loanei":



- Simplificare: **legarea** tuturor variabilelor, prin cuantificatori universali sau existențiali
- **Zona de acțiune** a unui cuantificator: restul propoziției (v. simbolul λ în calculul lambda)

385 / 423

Semantică I

O **interpretare** constă din:

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

386 / 423

Semantică II

- Atom:
 $(p(t_1, \dots, t_n))^I = p^I(t_1^I, \dots, t_n^I)$
- Negatie etc. (v. logica propozițională)
- Cuantificare **universală**:
 $(\forall x. \alpha)^I = \begin{cases} false & \text{dacă există } d \in D \text{ cu } \alpha^I_{d/x_1} = false \\ true & \text{altfel} \end{cases}$
- Cuantificare **existențială**:
 $(\exists x. \alpha)^I = \begin{cases} true & \text{dacă există } d \in D \text{ cu } \alpha^I_{d/x_1} = true \\ false & \text{altfel} \end{cases}$

387 / 423

Exemple

- "Vrabia mălai visează."
 $\forall x. (vrabie(x) \Rightarrow viseaza(x, malai))$
- "Unele vrăbii visează mălai."
 $\exists x. (vrabie(x) \wedge viseaza(x, malai))$
- "Nu toate vrăbiile visează mălai."
 $\exists x. (vrabie(x) \wedge \neg viseaza(x, malai))$
- "Nicio vrabie nu visează mălai."
 $\forall x. (vrabie(x) \Rightarrow \neg viseaza(x, malai))$
- "Numai vrăbiile visează mălai."
 $\forall x. (viseaza(x, malai) \Rightarrow vrabie(x))$
- "Toate și numai vrăbiile visează mălai."
 $\forall x. (viseaza(x, malai) \Leftrightarrow vrabie(x))$

388 / 423

Cuantificatori

Greșeli frecvente

- $\forall x. (vrabie(x) \Rightarrow viseaza(x, malai))$
→ corect: "Toate vrăbiile visează mălai."
- $\forall x. (vrabie(x) \wedge viseaza(x, malai))$
→ **greșit**: "Toți sunt vrăbii care visează mălai."
- $\exists x. (vrabie(x) \wedge viseaza(x, malai))$
→ corect: "Unele vrăbii visează mălai."
- $\exists x. (vrabie(x) \Rightarrow viseaza(x, malai))$
→ **greșit**: adevărată și dacă există cineva care nu este vrabie

389 / 423

Cuantificatori

Proprietăți

- **Necomutativitate**:
 - $\forall x. \exists y. viseaza(x, y)$: "Toți visează la ceva **particular**."
 - $\exists y. \forall x. viseaza(x, y)$: "Toți visează la **același** lucru."
- **Dualitate**:
 - $\neg(\forall x. \alpha) \equiv \exists x. \neg\alpha$
 - $\neg(\exists x. \alpha) \equiv \forall x. \neg\alpha$

390 / 423

Aspecte legate de propoziții

Analoage logicii propozițională:

- Satisfiabilitate
- Validitate
- Derivabilitate
- Inferență
- Demonstrație

391 / 423

Cuprins

3.33pt

- Introducere
- Logica propozițională
 - Sintaxă și semantică
 - Satisfiabilitate și validitate
 - Derivabilitate
 - Inferență și demonstrație
 - Rezoluție
- Logica cu predicate de ordinul I
 - Sintaxă și semantică
 - Forma clauzală
 - Unificare

392 / 423

Forma clauzală

- **Literal**: **atom** ($prieten(x, y)$) sau **negatia** lui ($\neg prieten(x, y)$)
- **Expresie clauzală** = **literal** sau **disjuncție** de literali, e.g. $prieten(x, y) \vee \neg doctor(x)$
- **Clauză** = **multime** de literali dintr-o expresie clauzală, e.g. $\{prieten(x, y), \neg doctor(x)\}$
- **Clauză Horn** = clauză în care un **singur** literal este în formă pozitivă, e.g. $\{\neg A_1, \dots, \neg A_n, A\}$, corespunzătoare **implicației** $A_1 \wedge \dots \wedge A_n \Rightarrow A$

393 / 423

Transformarea în formă clauzală I

- Eliminarea **implicațiilor** (I)
- Introducerea **negatiilor** în interiorul expresiilor (N)
- **Redenumirea** variabilelor cuantificate pentru obținerea **unicității** de nume (R):
 $\forall x. p(x) \wedge \forall x. q(x) \vee \exists z. r(x) \rightarrow \forall x. p(x) \wedge \forall y. q(y) \vee \exists z. r(z)$
- **Deplasarea** cuantificatorilor la **începutul** expresiei, conservându-le **ordinea** (forma normală **prenex**) (P):
 $\forall x. p(x) \wedge \forall y. q(y) \vee \exists z. r(z) \rightarrow \forall x. \forall y. \exists z. (p(x) \wedge q(y) \vee r(z))$

394 / 423

Transformarea în formă clauzală II

- Eliminarea cuantificatorilor **existențiali** (skolemizare) (S):
 - Dacă **nu** este precedat de cuantificatori universali: înlocuirea aparițiilor variabilei cuantificate printr-o **constantă**:
 $\exists x. p(x) \rightarrow p(c_x)$
 - Dacă este **precedat** de cuantificatori universali: înlocuirea aparițiilor variabilei cuantificate prin aplicația unei **funcții** unice asupra variabilelor anterior cuantificate universal:
 $\forall x. \forall y. \exists z. (p(x) \wedge q(y) \vee r(z)) \rightarrow \forall x. \forall y. (p(x) \wedge q(y) \vee r(f_z(x, y)))$

395 / 423

Transformarea în formă clauzală III

- Eliminarea cuantificatorilor **universali**, considerați acum impliciți (U):
 $\forall x. \forall y. (p(x) \wedge q(y) \vee r(f_z(x, y))) \rightarrow p(x) \wedge q(y) \vee r(f_z(x, y))$
- **Distribuirea** lui \vee față de \wedge (D):
 $\alpha \vee (\beta \wedge \gamma) \rightarrow (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$
- Transformarea expresiilor în **clauze** (C)

396 / 423

Transformarea în formă clauzală IV

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

- $\forall x. (\forall y. (lab(y) \Rightarrow rezolva(x, y)) \Rightarrow \exists y. apreciaza(y, x))$
- I $\forall x. (\neg \forall y. (\neg lab(y) \vee rezolva(x, y)) \vee \exists y. apreciaza(y, x))$
- N $\forall x. (\exists y. (\neg lab(y) \vee rezolva(x, y)) \vee \exists y. apreciaza(y, x))$
- N $\forall x. (\exists y. (lab(y) \wedge \neg rezolva(x, y)) \vee \exists y. apreciaza(y, x))$
- 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 $\forall x. ((lab(f_y(x)) \wedge \neg rezolva(x, f_y(x))) \vee apreciaza(f_z(x), x))$
- U $(lab(f_y(x)) \wedge \neg rezolva(x, f_y(x))) \vee apreciaza(f_z(x), x)$
- D $(lab(f_y(x)) \vee apreciaza(f_z(x), x))$
- $\wedge (\neg rezolva(x, f_y(x)) \vee apreciaza(f_z(x), x))$
- C $\{lab(f_y(x)), apreciaza(f_z(x), x)\}, \{\neg rezolva(x, f_y(x)), apreciaza(f_z(x), x)\}$

397 / 423

Cuprins

3.33pt

- Introducere
- Logica propozițională
 - Sintaxă și semantică
 - Satisfiabilitate și validitate
 - Derivabilitate
 - Inferență și demonstrație
 - Rezoluție
- Logica cu predicate de ordinul I
 - Sintaxă și semantică
 - Forma clauzală
 - Unificare

398 / 423

Motivație

- Rezoluție:
 $\frac{\{prieten(x, mama(y)), doctor(x)\}}{\{\neg prieten(mama(z), z)\}} \quad ?$
- Cum aplicăm rezoluția?
- Soluția: **unificare** (v. sinteza de tip, slide-ul 241)
- MGU: $S = \{x \leftarrow mama(z), z \leftarrow mama(y)\}$
- Forma **comună** a celor doi atomi: $prieten(mama(mama(y)), mama(y))$
- **Rezolvent**: $doctor(mama(mama(y)))$

399 / 423

Unificare I

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

400 / 423

Unificare II

- Rezoluția pentru clauze **Horn**:

$$\frac{A_1 \wedge \dots \wedge A_m \Rightarrow A \quad B_1 \wedge \dots \wedge A' \wedge \dots \wedge B_n \Rightarrow B}{unificare(A, A') = S} \\ subst(S, A_1 \wedge \dots \wedge A_m \wedge B_1 \wedge \dots \wedge B_n \Rightarrow B)$$

- **unificare** (α, β) : **substituția** sub care unifică propozițiile α și β
- $subst(S, \alpha)$: propoziția rezultată în urma **aplicării** substituției S asupra propoziției α

401 / 423

Rezumat

- Expresivitatea superioară a logicii cu predicate de ordinul 1, față de cea propozițională
- Propoziții satisfiabile, valide, nesatisfiabile
- Derivabilitate logică: proprietatea unei propoziții de a reprezenta consecința logică a altora
- Derivabilitate mecanică (inferență): posibilitatea unei propoziții de a fi determinată drept consecință a altora, în baza unei proceduri de calcul (de inferență)
- Rezoluție: procedura de inferență consistentă și completă (nu generativ)

402 / 423

Bibliografie

Harrison, J. (2009). *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press.
Genesereth, M. (2010). *CS157: Computational Logic*, curs Stanford.
<http://logic.stanford.edu/classes/cs157/2010/cs157.html>

403 / 423

Partea XIII

Mașina algoritmică Markov

404 / 423

Cuprins

3.33pt

186 Introducere

189 Mașina algoritmică Markov

405 / 423

Cuprins

3.33pt

186 Introducere

189 Mașina algoritmică Markov

406 / 423

Mașina algoritmică Markov

- Model de calculabilitate efectivă, **echivalent** cu mașina Turing și cu calculul lambda
- Principiul de funcționare: identificare de șabloane (eng. *pattern matching*) și substituție
- Fundamentul teoretic al paradigmei **asociative** și al limbajelor bazate pe **reguli**

407 / 423

Paradigma asociativă

- Potrivită mai ales în cazul problemelor ce **nu** admit o soluție precisă, algoritmică
- 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*

408 / 423

Cuprins

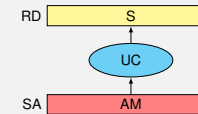
3.33pt

186 Introducere

189 Mașina algoritmică Markov

409 / 423

Structură



- Registrul de **date**, RD, cu secvența de simbolii, S
- Unitatea de **control**, UC
- Spațiul de stocare a **algoritmului**, SA, ce conține algoritmul Markov, AM

410 / 423

Algoritmi

```
1 setDiff1(A, B); A g1; B g2;  
2 ag2 -> a;  
3 ag1 -> g1a;  
4 a -> .;  
5 -> a;  
6 end
```

```
1 setDiff2(A, B); B g2;  
2 g2 -> .;  
3 -> .;  
4 end
```

411 / 423

Registrul de date

- **Nemărginit** la dreapta
- Simboli din alfabetul $A_0 \cup A_1$:
 - A_0 : alfabetul **de bază**
 - A_1 : alfabetul **local** / de lucru
 - $A_0 \cap A_1 = \emptyset$
- Șirurile **inițial** și **final**, formate doar cu simbolii din A_0
- Simbolii din A_1 , utilizabili exclusiv în timpul **execuției**
- Șirul de simbolii, posibil **vid**

412 / 423

Reguli

- Unitatea de bază a unui algoritm Markov: **regula** asociativă de substituție:
șablon de **identificare** (LHS) \rightarrow
șablon de **substituție** (RHS)
- Exemplu: $ag_1c \rightarrow ac$
- **Sabloanele**: secvențe de simbolii:
 - **constante**: simbolii din A_0
 - **variabile locale**: simbolii din A_1
 - **variabile generice**: simbolii speciali, din mulțimea G , legați la simbolii din A_0
- Pentru RHS = "." — regula **terminală**, ce încheie execuția mașinii

413 / 423

Variabile generice

- Legate la exact **un simbol**
- De obicei, **notate** cu g , urmat de un indice
- Mulțimea valorilor pe care le poate lua o variabilă: **domeniul** variabilei, $\text{Dom}(g)$
- Utilizabile în RHS **doar** în cazul apariției în LHS

414 / 423

Algoritmi (detaliu)

- Mulțimi **ordonate** de **reguli**, îmbogățite cu **declarații** de
 - partiționare a mulțimii A_0
 - variabile generice

- Exemplu: **eliminarea** simbolilor ce aparțin mulțimii B:

```
1 setDiff1(A, B); A g1; B g2;      1 setDiff2(A, B); B g2;  
2 ag2 -> a;                        2 g2 -> .;  
3 ag1 -> g1a;                       3 -> .;  
4 a -> .;                            4 end  
5 -> a;  
6 end
```

- $A, B \subseteq A_0$
- g_1, g_2 : variabile generice
- a : **nedeclarată**, variabilă locală ($a \in A_1$)

415 / 423

Aplicabilitatea regulilor

Regula $r : a_1 \dots a_n \rightarrow b_1 \dots b_m$ este aplicabilă dacă și numai dacă există un **subsir** $c_1 \dots c_n$, în RD, astfel încât, pentru orice $i = \overline{1, n}$, **exact o** condiție de mai jos este îndeplinită:

- $a_i \in A_b \wedge a_i = c_i$
- $a_i \in A_l \wedge a_i = c_i$
- $a_i \in G \wedge (\forall j = \overline{1, n} \bullet a_j = a_i \Rightarrow c_j \in \text{Dom}(a_i) \wedge c_j = c_i)$, i.e. variabila a_i este legată la o valoare **unică**, obținută prin potrivirea dintre șablon și subsir.

416/423

Aplicarea regulilor

Aplicarea regulii $r : a_1 \dots a_n \rightarrow b_1 \dots b_m$ asupra unui subsir $s = c_1 \dots c_n$, în raport cu care este **aplicabilă**, constă în **substituirea** lui s prin subsirul $q_1 \dots q_m$, calculat astfel:

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

417/423

Exemplu de aplicare

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

```
s = 11111 1 2 x 2 y 3 1111
r :      1 g1 x g1 y g2 -> 1g2x
s' = 1111113x1111
```

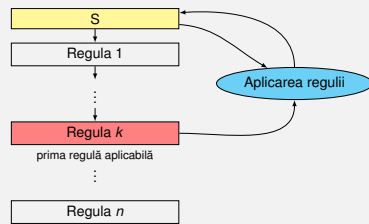
418/423

Aplicabilitate vs. aplicare

- Aplicabilitatea potențială a
 - unei reguli pe **mai multe** subșiruri
 - **mai multor** reguli pe același subșir
- La un anumit moment, aplicarea propriu-zisă a unei **singure** reguli asupra unui **singur** subșir
- **Nedeterminism** inerent, ce trebuie rezolvat
- Convenție:
 - aplicarea **primei** reguli aplicabile, în ordinea definiției,
 - asupra celui mai din **stânga** subșir asupra căreia este aplicabilă

419/423

Unitatea de control I



420/423

Unitatea de control II

- Analogie cu o **sită** pe mai multe nivele, ce corespund regulilor
- **Aplicabilitatea** testată secvențial
- Etape:
 1. determinarea **primei** reguli aplicabile
 2. **aplicarea** acesteia
 3. actualizarea **RD**
 4. salt la pasul 1

421/423

Inversarea intrării

Ideea: mutarea **pe rând**, a fiecărui element în poziția corespunzătoare, prin interschimbarea elementelor **adiacente**

```
1 Reverse(A); A g1, g2;
2   ag1g2 -> g2ag1;
3   ag1 -> bg1;
4   abg1 -> g1a;
5   a -> .;
6   -> a;
7 end
```

```
DOP → aDOP → OaDP → OPaD → OPbd → aOPbd
 2 → PaObD → PbObD → aPbObD → bPbObD → abPbObD
 4 → PabObD → POabb → POda .
```

422/423

Rezumat

Masina Markov: model de calculabilitate, bazat pe identificări spontane de șabloane și pe substituție

423/423