

PARADIGME DE PROGRAMARE

Curs 1

Introducere. Modele de evaluare. Limbajul Racket. Recursivitate.

Administrative

<https://ocw.cs.pub.ro/courses/pp>

- Cursuri, teme, instrucțiuni de instalare
- Laborator (teorie, exerciții, soluții)
 - teoria înainte / exercițiile la laborator / soluțiile după
- Exemple de examene și teste
- Regulament
 - citiți-l la începutul semestrului

Structura fiecărui curs

- Predare
 - fiți activi, puneți întrebări!
- Test din cursul anterior (uneori)
- Rezumatul cursului curent
 - conceput pentru autotestare, nu doar pentru a fi citit

Obiectivul materiei – programatori mai buni

Alternative la paradigmile imperativă și orientată obiect

- Paradigma funcțională, paradigma logică

Cum sunt proiectate limbajele de programare

- Modele de calculabilitate
- Features: controlul complexității prin lizibilitate și eficiență
- Limbaje multiparadigmă pentru programatori multiparadigmă

Adaptarea rapidă la noi limbaje de programare

- Racket, Haskell, Prolog

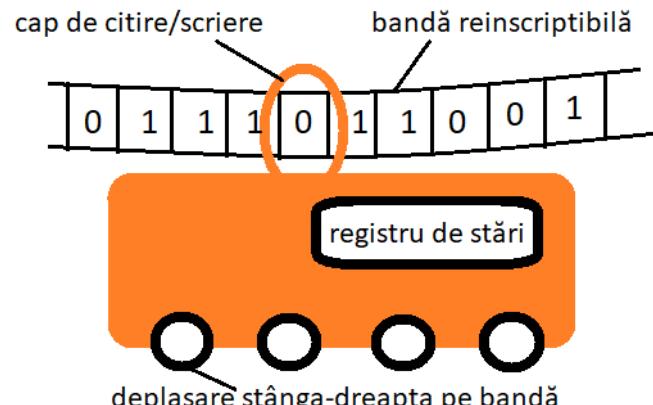
Provocări distractive

Modele, paradigmă, limbaje – Cuprins

- Modele de calculabilitate
- Paradigme de programare
- Exemplu rezolvat în diferite paradigmă/limbaje

Modele de calculabilitate

Mașina Turing



Dacă în starea 5 citesc simbolul 0 pe bandă, atunci scriu 1, mut capul către stânga și trec în starea 3

Mașina Markov

imperativă -> funcțională
rea -> bună
este cea mai bună -> bate tot

Dându-se s = "Programarea imperativă este cea mai rea" și regulile de substituție de mai sus, asupra lui s se aplică succesiv prima regulă aplicabilă, cât timp ea există.

Programarea funcțională este cea mai rea
Programarea funcțională este cea mai bună
Programarea funcțională bate tot

$$(\lambda x. \lambda y. x + y \ 2) \rightarrow \\ \lambda y. 2 + y$$

Dacă aplic funcția cu argumentul **x** și corpul **$\lambda y. x + y$** asupra valorii **2**, obțin funcția cu argumentul **y** și corpul **$2 + y$**

om(alin). om(adina). om(cristi).
place(alin, adina). place(adina, cristi).
fericit(X) dacă om(X), om(Y), place(Y, X).

fericit(Cine).
>>> Cine = adina; Cine = cristi.

Dându-se faptele și regulile anterioare, se încearcă instanțierea variabilelor **Cine**, **X** și **Y** în toate modurile posibile astfel încât să se satisfacă scopul că **Cine** este fericit.

Calculul Lambda

Mașina Logică

Modele, paradigmă, limbaje – Cuprins

- Modele de calculabilitate
- Paradigme de programare
- Exemplu rezolvat în diferite paradigmă/limbaje

Modele, paradigme, limbaje

Model de calculabilitate

- Oferă un model formal al efectuării calculului
- Diferă de alte modele prin CUM se calculează funcțiile, nu prin CE funcții se calculează

Paradigmă de programare

- Stil fundamental de a programa, bazat pe un anumit model de calculabilitate
- Mod de reprezentare a datelor (ex: variabile, funcții, obiecte, fapte, constrângeri)
- Mod de prelucrare a reprezentării (ex: atribuiri, evaluări, fire de execuție)

Limbaj de programare

- Limbaj formal capabil să exprime procesul de rezolvare a problemelor
- Sprijină una sau mai multe paradigme (ex: Scala, F# - POO și PF; Python – imperativ, POO și PF)

Paradigma	Reprezentarea datelor	Structura programului	Execuția programului	Rezultat	Limbaje
Imperativă	Variabile	Succesiune de comenzi	Execuție de comenzi	Stare finală a memoriei	C, Pascal, Fortran
Orientată Obiect	Obiecte	Colecție de clase și obiecte	Transmitere de mesaje între obiecte	Stare finală a obiectelor	Java, C++
Funcțională	Funcții	Colecție de funcții	Evaluare de funcții	Valoarea la care se evaluatează funcția principală	Racket, Haskell
Logică	Fapte, reguli	Axiome și o teoremă care trebuie demonstrată	Demonstrarea teoremei	Reușită sau eșec în demonstrarea teoremei	Prolog

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

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

Abraham Maslow, The law of instrument

The illiterate of the 21st century will not be those who cannot read and write, but those who cannot learn, unlearn, and relearn.

Alvin Toffler



Modele, paradigme, limbaje – Cuprins

- Modele de calculabilitate
- Paradigme de programare
- Exemplu rezolvat în diferite paradigmă/limbaje

Exemplu

Să se determine factorialul unui număr natural n folosind paradigmile:

- Imperativă
- Funcțională
- Logică

Rezolvare imperativă

1. `int i, factorial = 1;` ← datele sunt reținute în **variabilele** i, factorial
2. `for (i = 2; i <= n; i++)` ← rezolvarea este o **execuție succesivă** de înmulțiri
3. `factorial *= i;` ← rezultatul se regăsește în **starea** finală a memoriei
(în zona rezervată variabilei factorial)

De reținut

- Programele imperative au stare
- Starea diferă de la un moment al execuției la altul
- Constructe fundamentale: atribuirea, ciclarea
- Soluție tip „rețetă” (programul descrie CUM se construiește, pas cu pas, rezultatul)

Rezolvare funcțională – Racket

```
1. (define (factorial n)           ← datele sunt reținute în funcții (totul este o funcție)
   (if (zero? n)
       1                           ← caz de bază în recursivitate
       (* n (factorial (- n 1))))) ← apelul recursiv cu o nouă valoare a parametrului n
                                      (corespunzător constructorilor interni)
```

De reținut

- Programele funcționale nu au stare
- **Ciclarea** este înlocuită prin **recursivitate**
- **Atribuirea** este înlocuită printr-un apel recursiv cu **noi valori ale parametrilor funcției**
- **Succesiunea de comenzi** este înlocuită prin **componere de funcții**
- Soluție declarativă (programul descrie CE este, din punct de vedere matematic, rezultatul)

Rezolvare funcțională – Haskell

```
1. factorial 0 = 1  
2. factorial n = n * factorial (n - 1)
```

← totul este o funcție, deci nu este necesar un cuvânt cheie care să spună că urmează o funcție

De observat

- Haskell permite **pattern matching** pe parametrii formali ai funcției (feature existent în Haskell dar nu și în Racket)
- Dacă pattern matching-ul de pe linia 1 reușește, se întoarce rezultatul 1, altfel se încearcă potrivirea cu linia următoare (care, pe codul de mai sus, reușește întotdeauna)

Rezolvări funcționale „avansate”

Racket

```
1. (define (factorial n)
2.       (apply * (range 2 (+ n 1))))
```

Haskell

```
1. factorial n = product [1 .. n]
```

Pentru rezolvări și mai avansate:

<http://www.willamette.edu/~fruehr/haskell/evolution.html>

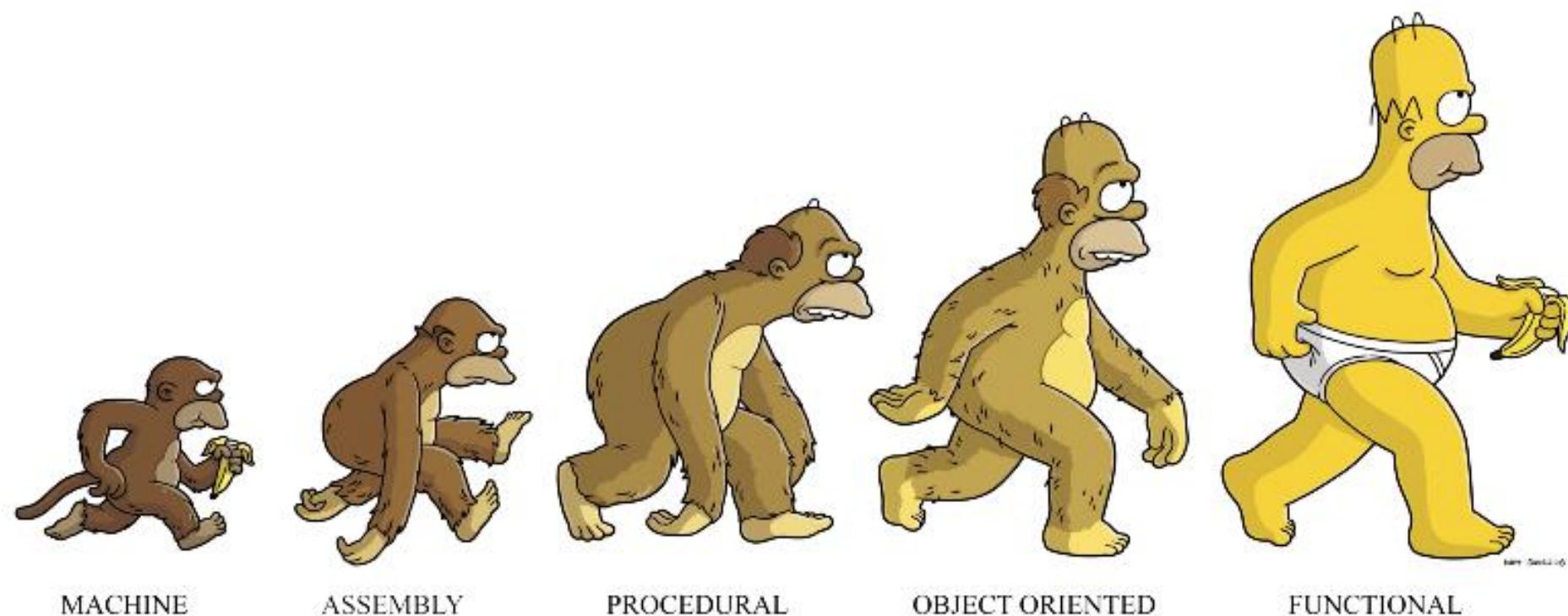
Rezolvare logică – Prolog

```
1. factorial(0, 1).                                ← datele sunt reținute în fapte (ex: factorial de 0 este 1)
2. factorial(N, Result) :-                         și reguli (ex: factorial de N este Result, dacă:
   3. N > 0,                                         N > 0 și
   4. Prev is N-1,                                    factorial de N-1 este F și
   5. factorial(Prev, F),                           Result este N*F)
   6. Result is N*F.
```

De reținut

- Asemănări cu paradigma funcțională: soluție declarativă, ciclarea înlocuită prin recursitate, atribuirea înlocuită prin noi valori ale parametrilor apelului recursiv
- Faptele și regulile sunt axiome, iar o interogare de tip factorial(5, 120) sau factorial(4, F) reprezintă teorema pe care programul încearcă să o demonstreze
- Limbajul demonstrează teorema potrivind-o în toate modurile posibile cu axiomele existente în universul problemei (**backtracking** încorporat în limbajul de programare)

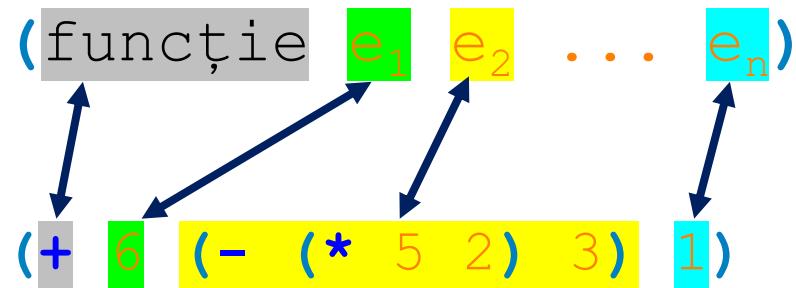
Programare funcțională în Racket



Limbajul Racket – Cuprins

- Expresii și evaluare aplicativă
- Lambda-expresii și funcții
- Perechi și liste
- Operatori condiționali
- Recursivitate

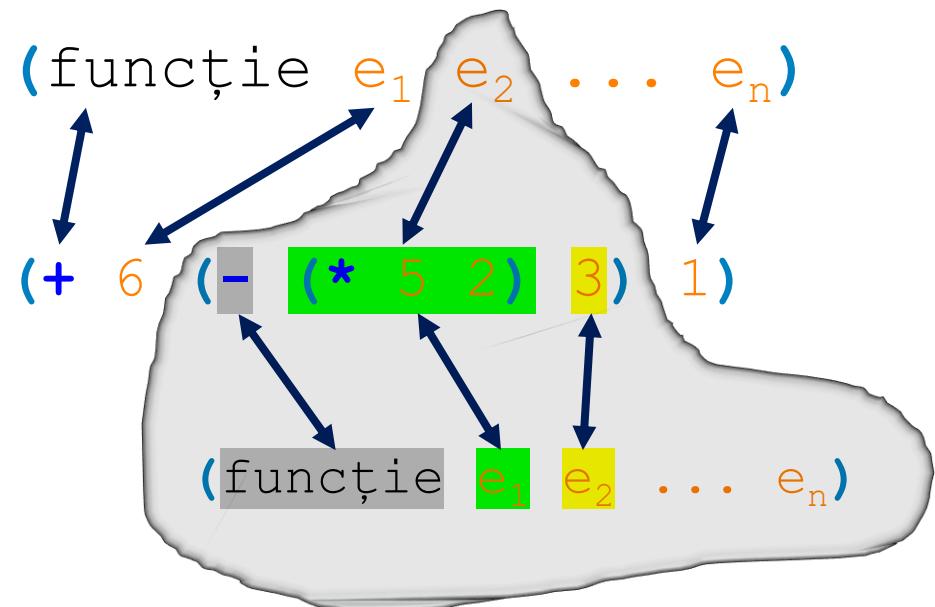
Expresii în Racket



Observație

Fiecare argument al funcției poate fi, la rândul său, o nouă expresie complexă, cu aceeași sintaxă `(funcție e1 e2 ... en)`. Este cazul lui e_2 de mai sus.

Expresii în Racket



Evaluare aplicativă

Evaluare aplicativă (ex: Racket)

Înainte de a aplica o funcție asupra unor subexpresii, evaluează toate aceste subexpresii cât de mult se poate.



Evaluare normală (ex: Calcul Lambda)

Subexpresiile sunt pasate funcției fără a fi evaluate, în expresia finală subexpresiile reductibile se evaluează de la stânga la dreapta.

Exemplu de evaluare a unei expresii Racket

(+ 6 (- (* 5 2) 4)))

(+ 6 (- (* 5 2) 4)))

(+ 6 (- 10 4)))

(+ 6 (- 10 4)))

(+ 6 6)

(+ 6 6)

12

Strategii de evaluare

Reprezintă reguli de evaluare a expresiilor într-un limbaj de programare.

Strategii stricte (argumentele funcțiilor sunt evaluate la apel, înainte de aplicare)

- **Evaluare aplicativă**
- **Call by value** – funcției i se dă o copie a valorii obținută la evaluare (C, Java, Racket)
- **Call by reference** – funcției i se pasează o referință la argument (Perl, Visual Basic)

Strategii nestrictive (argumentele funcțiilor nu sunt evaluate până ce valoarea lor nu e necesară undeva în corpul funcției)

- **Evaluare normală**
- **Call by name** – funcția primește argumentele ca atare, neevaluate, și le evaluatează și reevaluatează de fiecare dată când valoarea e necesară în corpul funcției
- **Call by need** – un call by name în care prima evaluare reține rezultatul într-un cache pentru refolosire (Haskell, R)

Construcția **define**

(define identificator expresie)

- Creează o pereche identificator-valoare (provenită din evaluarea imediată a expresiei), nu alterează o zonă de memorie (\neq atribuire)
- Scopul:
 - lizibilitate (numele documentează semnificația valorii)
 - flexibilitate (la nevoie, valoarea se modifică într-un singur loc)
 - reutilizare (expresiile complexe nu trebuie rescrise integral de fiecare dată)

Exemple

1. **(define** PI 3.14159265)
 2. **(define** r 5)
 3. **(define** area (* PI r r))
- ← identificatorul area se leagă la valoarea 78.53981625,
fără să rețină de unde a provenit aceasta

Limbajul Racket – Cuprins

- Expresii și evaluare aplicativă
- Lambda-expresii și funcții
- Perechi și liste
- Operatori condiționali
- Recursivitate

λ -expresia (în Calculul Lambda)

Sintaxa

$e \equiv$ x variabilă
| $\lambda x.e$ funcție (unară, anonimă) cu parametrul formal x și corpul e
| $(e_1 e_2)$ aplicație a expresiei e_1 asupra parametrului efectiv e_2

Semantica (Modelul substituției)

Pentru a evalua $(\lambda x.e_1 e_2)$ (funcția cu parametrul formal x și corpul e_1 , aplicată pe e_2):

- Peste tot în e_1 , identificatorul x este înlocuit cu e_2
- Se evaluatează noul corp e_1 și se întoarce rezultatul (se notează $e_1_{[e_2/x]}$)

Exemple de λ -expresii

$\lambda x.x$

$(x y)$

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

$(\lambda x.x a)$

$(\lambda x.y a)$

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

Exemple de λ -expresii

$\lambda x.x$ funcția identitate

$(x y)$

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

$(\lambda x.x a)$

$(\lambda x.y a)$

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

Exemple de λ -expresii

$\lambda x.x$

$(x\ y)$

aplicația expresiei x asupra expresiei y

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

$(\lambda x.x\ a)$

$(\lambda x.y\ a)$

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

Exemple de λ -expresii

$\lambda x.x$

$(x\ y)$

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

$(\lambda x.x\ a)$

$(\lambda x.y\ a)$

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

o funcție de un parametru x care întoarce o altă funcție
de un parametru y care îl aplică pe x asupra lui y

Exemple de λ -expresii

$\lambda x.x$

$(x y)$

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

$(\lambda x.x a)$ funcția identitate aplicată asupra lui a (se evaluatează la a)

$(\lambda x.y a)$

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

Exemple de λ -expresii

$\lambda x.x$

$(x y)$

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

$(\lambda x.x a)$

$(\lambda x.y a)$ funcția de parametru x cu corpul y , aplicată asupra lui a (se evaluatează la y)

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

Exemple de λ -expresii

$\lambda x.x$

$(x y)$

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

$(\lambda x.x a)$

$(\lambda x.y a)$

$(\lambda x.x \lambda x.y)$ funcția identitate aplicată asupra funcției $\lambda x.y$ (se evaluatează la $\lambda x.y$)

Funcții anonime în Racket

(**lambda** listă-parametri corp)

Exemple

$\lambda x.x$ (**lambda** (x) x) ; ; echivalent cu $(\lambda (x) x)$

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

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

Funcții cu nume în Racket

O funcție este o expresie și, ca orice expresie, poate primi un nume cu **define**.

1. **(define** arithmetic-mean
2. **(λ (x y)**
3. **(/ (+ x y)**
4. **2)))**
5. **(arithmetic-mean 5 19) ;;** se evaluează la 12

Racket permite și sintaxa **(define (nume-funcție x1 x2 ... xn) corp)**:

1. **(define (arithmetic-mean x y) (/ (+ x y) 2))**

Exemplu de evaluare a unei aplicații de funcție

```
1. (define (sum-of-squares x y)
2.   (+ (sqr x) (sqr y)))
3.
4. (sum-of-squares (+ 1 2) (* 3 5))      ;; înlocuiește numele prin valoare
>((λ (x y) (+ (sqr x) (sqr y))) (+ 1 2) (* 3 5)) ;; evaluare aplicativă
>((λ (x y) (+ (sqr x) (sqr y))) 3 15)           ;; modelul substituției (x<-3, y<-15)
>(+ (sqr 3) (sqr 15))                          ;; evaluare aplicativă
>(+ 9 225)
>(+ 9 225)
>234
```

Limbajul Racket – Cuprins

- Expresii și evaluare aplicativă
- Lambda-expresii și funcții
- Perechi și liste
- Operatori condiționali
- Recursivitate

TDA-ul Pereche

Constructori de bază

cons : $T_1 \times T_2 \rightarrow$ Pereche // creează o pereche cu punct între orice 2 argumente

Operatori

car : Pereche $\rightarrow T_1$ // extrage prima valoare din pereche
cdr : Pereche $\rightarrow T_2$ // extrage a doua valoare din pereche

Exemple

```
(cons (cons 1 2) 'a)  
(cons + 3)  
(car (cons (cons 1 2) 5))  
(cdr '(4 . b))
```

TDA-ul Pereche

Constructori de bază

cons : $T_1 \times T_2 \rightarrow$ Pereche // creează o pereche cu punct între orice 2 argumente

Operatori

car : Pereche $\rightarrow T_1$ // extrage prima valoare din pereche
cdr : Pereche $\rightarrow T_2$ // extrage a doua valoare din pereche

Exemple

```
(cons (cons 1 2) 'a)      ;; '#(1 . 2) . a
(cons + 3)                 ;; '#<procedure:+> . 3
(car (cons (cons 1 2) 5)) ;; '#(1 . 2)
(cdr '(4 . b))            ;; '#b
```

Sintaxa valorilor de tip Pereche

' (1 . 2) echivalent cu (quote (1 . 2))

Explicație

Funcția **quote** își „citează” argumentul, în sensul că previne evaluarea acestuia.

Apostroful este doar o notație prescurtată echivalentă cu funcția **quote**.

Acest artificiu este necesar în reprezentarea valorilor de tip Pereche (sau Listă), pentru că Racket, la întâlnirea unei paranteze deschise, consideră că urmează o funcție și apoi argumentele pe care se aplică aceasta.

Racket va interpreta codul ' (1 2) ca pe lista (1 2), în schimb va da eroare dacă încercăm să rulăm codul (1 2):



*application: not a procedure;
expected a procedure that can be applied to arguments
given: 1
arguments....:*

TDA-ul Listă

Constructori (de bază și nu numai)

null : -> Listă

// creează o listă vidă; echivalent cu valoarea '()

cons : T x Listă -> Listă

// creează o listă prin adăugarea unei valori la începutul unei liste

list : T₁ x .. T_n -> Listă

// creează o listă din toate argumentele sale

Operatori

car : Listă -> T

(**car** (**list** 1 'a +))

cdr : Listă -> Listă

(**cdr** '(2 3 4 5))

null? : Listă -> Bool

(**null?** '())

length : Listă -> Nat

(**length** (**list**))

append : Listă x Listă -> Listă

(**append** (**cons** 1 '(2)) '(a b))

Exemple

TDA-ul Listă

Constructori (de bază și nu numai)

null : -> Listă

// creează o listă vidă; echivalent cu valoarea '()

cons : T x Listă -> Listă

// creează o listă prin adăugarea unei valori la începutul unei liste

list : T₁ x .. T_n -> Listă

// creează o listă din toate argumentele sale

Operatori

car : Listă -> T

Exemple

(**car** (**list** 1 'a +)) ; ; 1

cdr : Listă -> Listă

(**cdr** '(2 3 4 5)) ; ; '(3 4 5)

null? : Listă -> Bool

(**null?** '()) ; ; #t

length : Listă -> Nat

(**length** (**list**)) ; ; 0

append : Listă x Listă -> Listă

(**append** (**cons** 1 '(2)) '(a b)) ; ; '(1 2 a b)

Sintaxa valorilor de tip Listă

' (1 2 3) echivalent cu ' (1 . (2 . (3 . ()))))

Explicație

Listele sunt reprezentate intern ca perechi (cu punct) între primul element și restul listei.

Așadar:

- lista ' (3) este de fapt o pereche între valoarea 3 și lista vidă: ' (3 . ())
- lista ' (2 3) este o pereche între valoarea 2 și lista ' (3): ' (2 . (3 . ()))
- lista ' (1 2 3) este o pereche între valoarea 1 și lista ' (2 3): ' (1 . (2 . (3 . ())))

Observație

Lista este TDA-ul de bază în programarea funcțională.

Orice funcție Racket are structura unei liste și codul Racket poate fi generat, respectiv parsat în Racket folosind constructori și operatori pe liste.

Limbajul Racket – Cuprins

- Expresii și evaluare aplicativă
- Lambda-expresii și funcții
- Perechi și liste
- Operatori condiționali
- Recursivitate

Condiționala **if**

(**if** condiție rezultat-then rezultat-else)

Exemple

1. **(if** (**null?** '1 2)) ;; se evaluatează la #f
2. (+ 1 2) ;; NU se evaluatează
3. (- 7 1)) ;; întreg if-ul se evaluatează la 6
4. **(if** (< 4 10) ;; se evaluatează la #t
5. 'succes ;; întreg if-ul se evaluatează la 'succes
6. (/ 'logica 0)) ;; NU se evaluatează (de aceea nu dă eroare)

Totul este o funcție

`if` se comportă ca o **funcție cu 3 argumente**: condiția, rezultatul pe ramura de `then`, și rezultatul pe ramura de `else`.

Întrucât funcția trebuie să se evalueze mereu la ceva, niciunul din cele 3 argumente nu poate lipsi! (**nu putem avea un if fără else**)

Întrucât unul din argumente nu va fi necesar, `if` nu își evaluează argumentele la apel (este o **funcție nestrictă**). Evaluarea unui `if` se produce astfel:

- Se evaluează condiția (doar primul argument, nu și celelalte două)
- Dacă rezultatul este `true`, întregul `if` este înlocuit cu rezultatul pe ramura de `then`, altfel întregul `if` este înlocuit cu rezultatul pe ramura de `else`
- Se evaluează noua expresie

Condiționala **cond**

(cond (condiție₁ rezultat₁)

...

(condiție_n rezultat_n))

← în loc de ultima condiție putem folosi cuvântul cheie **else**, dar nu e obligatoriu

Exemplu

1. **(define** L ' (1 2 3))
2. **(cond**
3. **((null?** L) 0) ;; se evaluatează doar condiția la #f
4. **((null? (cdr L)) (/ 1 0))** ;; se evaluatează doar condiția la #f
5. **(else** 'other)) ;; întregul cond se evaluatează la 'other

Limbajul Racket – Cuprins

- Expresii și evaluare aplicativă
- Lambda-expresii și funcții
- Perechi și liste
- Operatori condiționali
- Recursivitate

Recursivitate în programarea funcțională

Nu mai avem

- Atribuirile
- Instrucțiunile de ciclare (for, while)
- Secvența de operații (o funcție se evaluatează la o unică valoare și nu are efecte laterale)

Avem

Compunere de funcții recursive cu starea problemei pasată ca parametru în aceste funcții



De la axiomele TDA-ului la recursivitate

Exemplu: Suma elementelor dintr-o listă

Axiome

// Operatorul sum

sum([]) = 0

sum(x:l) = x + sum(l)

Program Racket

(**define** (sum L)

(**if** (**null?** L)

0

(+ (**car** L) (sum (**cdr** L))))))

Observații

Axiomele TDA-ului se traduc direct în cod funcțional

- Trebuie precizat comportamentul funcției pe **toți constructorii de bază**
- Orice **în plus e redundant**
 - ex: e redundant și nelegant să precizez și comportamentul pentru liste de fix un element
- Orice **în minus e insuficient** și duce la eşecul aplicării funcției pe anumite valori
 - ex: factorial(1) = 1; factorial(succ(n)) = succ(n) * factorial(n) => eroare pentru factorial(0)

Abordare generală în scrierea de funcții recursive

- 1) După ce variabile fac recursivitatea? (ce variabile își schimbă valoarea de la un apel la altul?)
- 2) Scrie condiția de oprire pentru fiecare asemenea variabilă (constructori nulari și externi)
- 3) Scrie ce se întâmplă când problema nu este încă elementară (constructori interni, care generează obligatoriu cel puțin un apel recursiv)

Exemplu: Extragerea primelor n elemente dintr-o listă L

```
(define (take L n))
```

- 1) După ce variabile fac recursivitatea? (ce variabile își schimbă valoarea de la un apel la altul?)
 - Dacă aş ști să extrag din (cdr L), m-ar ajuta? (încerc să scad, pe rând, dimensiunea parametrilor)
 - Observ că a lua primele 3 elemente din lista '(1 2 3 4) e totuна cu a lua primele 2 elemente din lista '(2 3 4) și a îl adăuga pe 1 în față
 - Rezultă că subproblema care mă ajută are (cdr L) și (- n 1) ca parametri, deci recursivitatea se face atât după L cât și după n
- 2) Scrie condiția de oprire pentru fiecare asemenea variabilă (constructori nulari și externi)

```
(if (or (zero? n) (null? L))  
    '())
```
- 3) Scrie ce se întâmplă când problema nu este încă elementară (constructori interni, care generează obligatoriu cel puțin un apel recursiv)

```
(cons (car L) (take (cdr L) (- n 1))))
```

Rezumat

Modele de calculabilitate

Paradigme

Strategii de evaluare

Lambda-expresii

Sintaxa expresiilor Racket

Sintaxa funcțiilor Racket

Perechi și Liste

Operatori condiționali

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică

Paradigme

Strategii de evaluare

Lambda-expresii

Sintaxa expresiilor Racket

Sintaxa funcțiilor Racket

Perechi și Liste

Operatori condiționali

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică

Paradigme: imperativă, orientată obiect, funcțională, logică

Strategii de evaluare

Lambda-expresii

Sintaxa expresiilor Racket

Sintaxa funcțiilor Racket

Perechi și Liste

Operatori condiționali

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică

Paradigme: imperativă, orientată obiect, funcțională, logică

Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)

Lambda-expresii

Sintaxa expresiilor Racket

Sintaxa funcțiilor Racket

Perechi și Liste

Operatori condiționali

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică

Paradigme: imperativă, orientată obiect, funcțională, logică

Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)

Lambda-expresii: variabilă (x), funcție ($\lambda x.e$), aplicație ($(e_1 e_2)$)

Sintaxa expresiilor Racket

Sintaxa funcțiilor Racket

Perechi și Liste

Operatori condiționali

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică

Paradigme: imperativă, orientată obiect, funcțională, logică

Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)

Lambda-expresii: variabilă (x), funcție ($\lambda x.e$), aplicație ($(e_1 e_2)$)

Sintaxa expresiilor Racket: (funcție $e_1 e_2 \dots e_n$)

Sintaxa funcțiilor Racket:

Perechi și Liste

Operatori condiționali

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică

Paradigme: imperativă, orientată obiect, funcțională, logică

Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)

Lambda-expresii: variabilă (x), funcție ($\lambda x.e$), aplicație ($(e_1 e_2)$)

Sintaxa expresiilor Racket: (funcție $e_1 e_2 \dots e_n$)

Sintaxa funcțiilor Racket: (lambda ($x_1 x_2 \dots x_n$) corp) sau (define ($f x_1 x_2 \dots x_n$) corp)

Perechi și Liste

Operatori condiționali

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică

Paradigme: imperativă, orientată obiect, funcțională, logică

Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)

Lambda-expresii: variabilă (x), funcție ($\lambda x.e$), aplicație ($(e_1 e_2)$)

Sintaxa expresiilor Racket: (funcție $e_1 e_2 \dots e_n$)

Sintaxa funcțiilor Racket: (lambda ($x_1 x_2 \dots x_n$) corp) sau (define ($f x_1 x_2 \dots x_n$) corp)

Perechi și Liste: '(a . b), '(1 2 3), '(), cons, null, list, car, cdr, null?, length, append

Operatori condiționali

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică

Paradigme: imperativă, orientată obiect, funcțională, logică

Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)

Lambda-expresii: variabilă (x), funcție ($\lambda x.e$), aplicație ($(e_1 e_2)$)

Sintaxa expresiilor Racket: (funcție $e_1 e_2 \dots e_n$)

Sintaxa funcțiilor Racket: (lambda ($x_1 x_2 \dots x_n$) corp) sau (define ($f x_1 x_2 \dots x_n$) corp)

Perechi și Liste: '(a . b), '(1 2 3), '(), cons, null, list, car, cdr, null?, length, append

Operatori conditionali: if, cond

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică

Paradigme: imperativă, orientată obiect, funcțională, logică

Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)

Lambda-expresii: variabilă (x), funcție ($\lambda x.e$), aplicație ($(e_1 e_2)$)

Sintaxa expresiilor Racket: (funcție $e_1 e_2 \dots e_n$)

Sintaxa funcțiilor Racket: (lambda ($x_1 x_2 \dots x_n$) corp) sau (define ($f x_1 x_2 \dots x_n$) corp)

Perechi și Liste: '(a . b), '(1 2 3), '(), cons, null, list, car, cdr, null?, length, append

Operatori conditionali: if, cond

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni: compunere de funcții recursive cu starea problemei pasată ca parametru în aceste funcții

PARADIGME DE PROGRAMARE

Curs 2

Recursivitate pe stivă / pe coadă / arborescentă. Calcul Lambda.

Tipuri de recursivitate – Cuprins

- Importanța recursivității în paradigma funcțională
- Recursivitate pe stivă
- Recursivitate pe coadă
- Recursivitate arborescentă
- Comparație între tipurile de recursivitate
- Transformarea în recursivitate pe coadă

Recursivitate

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive).

Recursivitate

- Singura modalitate de a prelucra date de dimensiune variabilă (în lipsa iterăției)
- Elegantă (derivă direct din specificația formală / din axiome)
- Minimală (cod scurt, ușor de citit)
- Ușor de analizat formal (ex: demonstrații prin inducție structurală)
- Poate fi ineficientă: Se așteaptă rezultatul fiecărui apel recursiv pentru a fi prelucrat în contextul apelului părinte. Astfel, contextul fiecărui apel părinte trebuie salvat pe stivă pentru momentul ulterior în care poate fi folosit în calcul.

Problema

Pentru o lizibilitate sporită și aceeași putere de calcul (v. Teza lui Church), plătim uneori un preț mai mare (consum mare de memorie care poate duce chiar la nefuncționare – stack overflow).

Tipuri de recursivitate – Cuprins

- Importanța recursivității în paradigma funcțională
- Recursivitate pe stivă
- Recursivitate pe coadă
- Recursivitate arborescentă
- Comparație între tipurile de recursivitate
- Transformarea în recursivitate pe coadă

Exemplu – recursivitate pe stivă

```
1. (define (fact-stack n)
2.   (if (zero? n)
3.     1
4.     (* n (fact-stack (- n 1)))))
```

rezultatul apelului recursiv este așteptat
← pentru a fi înmulțit cu n

> (fact-stack 3) ← apelul curent este marcat cu verde
ceea ce tocmai s-a depus pe stivă e marcat cu roz
ceea ce urmează să se scoată de pe stivă e marcat cu mov

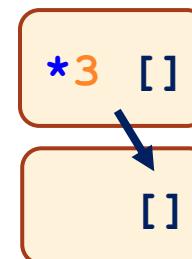
[]

Stiva procesului

Exemplu – recursivitate pe stivă

```
1. (define (fact-stack n)
2.   (if (zero? n)
3.     1
4.     (* n (fact-stack (- n 1)))))
```

```
> (fact-stack 3)
> (* 3 (fact-stack 2))
```

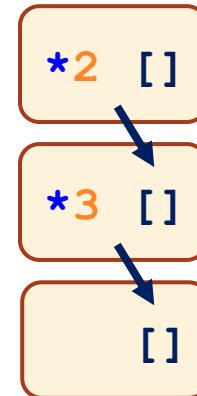


Stiva procesului

Exemplu – recursivitate pe stivă

```
1. (define (fact-stack n)
2.   (if (zero? n)
3.     1
4.     (* n (fact-stack (- n 1)))))
```

```
> (fact-stack 3)
> (* 3 (fact-stack 2))
> (* 3 (* 2 (fact-stack 1)))
```

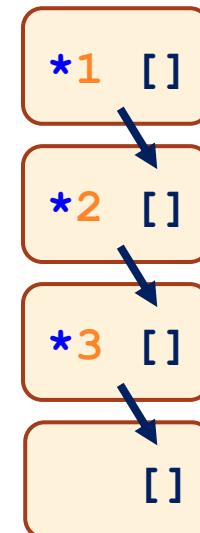


Stiva procesului

Exemplu – recursivitate pe stivă

```
1. (define (fact-stack n)
2.   (if (zero? n)
3.     1
4.     (* n (fact-stack (- n 1))))))

> (fact-stack 3)
> (* 3 (fact-stack 2))
> (* 3 (* 2 (fact-stack 1)))
> (* 3 (* 2 (* 1 (fact-stack 0)))))
```



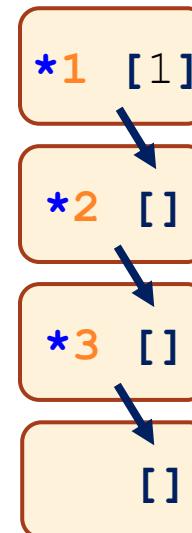
Stiva procesului

Exemplu – recursivitate pe stivă

```
1. (define (fact-stack n)
2.   (if (zero? n)
3.     1
4.     (* n (fact-stack (- n 1))))))
```



```
> (fact-stack 3)
> (* 3 (fact-stack 2))
> (* 3 (* 2 (fact-stack 1)))
> (* 3 (* 2 (* 1 (fact-stack 0))))
> (* 3 (* 2 (* 1 1)))
```

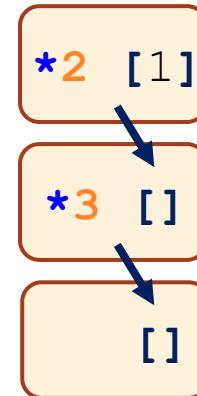


Stiva procesului

Exemplu – recursivitate pe stivă

```
1. (define (fact-stack n)
2.   (if (zero? n)
3.     1
4.     (* n (fact-stack (- n 1))))))
```

```
> (fact-stack 3)
> (* 3 (fact-stack 2))
> (* 3 (* 2 (fact-stack 1)))
> (* 3 (* 2 (* 1 (fact-stack 0))))
> (* 3 (* 2 (* 1 1)))
> (* 3 (* 2 1))
```

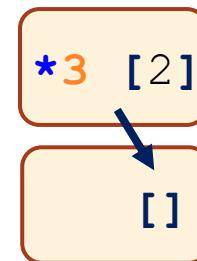


Stiva procesului

Exemplu – recursivitate pe stivă

```
1. (define (fact-stack n)
2.   (if (zero? n)
3.     1
4.     (* n (fact-stack (- n 1))))))
```

```
> (fact-stack 3)
> (* 3 (fact-stack 2))
> (* 3 (* 2 (fact-stack 1)))
> (* 3 (* 2 (* 1 (fact-stack 0))))
> (* 3 (* 2 (* 1 1)))
> (* 3 (* 2 1))
> (* 3 2)
```



Stiva procesului

Exemplu – recursivitate pe stivă

```
1. (define (fact-stack n)
2.   (if (zero? n)
3.     1
4.     (* n (fact-stack (- n 1))))))
```

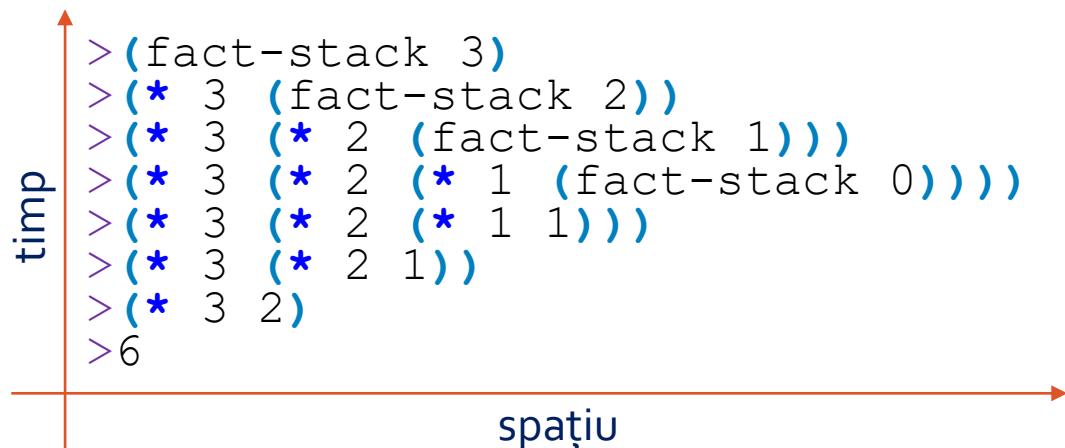
```
> (fact-stack 3)
> (* 3 (fact-stack 2))
> (* 3 (* 2 (fact-stack 1)))
> (* 3 (* 2 (* 1 (fact-stack 0))))
> (* 3 (* 2 (* 1 1)))
> (* 3 (* 2 1))
> (* 3 2)
> 6
```

[6]

Stiva procesului

Observații – recursivitate pe stivă

- **Timp:** $\Theta(n)$ (se efectuează n înmulțiri și stiva se redimensionează de 2^n ori)
- **Spațiu:** $\Theta(n)$ (ocupat de stivă)
- **Calcul:** realizat integral la revenirea din recursivitate
- **Stiva:** reține contextul fiecărui apel părinte, pentru momentul revenirii (starea programului se regăsește în principal în starea stivei)



Comparație cu rezolvarea imperativă

```
1. int i, factorial = 1;  
2. for (i = 2; i <= n; i++)  
3.     factorial *= i;
```

- **Timp:** $\Theta(n)$ (se efectuează n înmulțiri)
- **Spațiu:** $\Theta(1)$ (în orice moment, în memorie sunt reținute doar 3 valori, pentru variabilele i , n , $factorial$)
 - Rezultatul se construiește în variabila `factorial` pe măsură ce avansăm în iterare
 - Putem obține același comportament într-o variantă recursivă?

Tipuri de recursivitate – Cuprins

- Importanța recursivității în paradigma funcțională
- Recursivitate pe stivă
- **Recursivitate pe coadă**
- Recursivitate arborescentă
- Comparație între tipurile de recursivitate
- Transformarea în recursivitate pe coadă

Exemplu – recursivitate pe coadă

```
1. (define (fact-tail n)
2.       (fact-tail-helper n 1))
3.
4. (define (fact-tail-helper n fact)
5.   (if (zero? n)
6.       fact
7.       (fact-tail-helper (- n 1) (* n fact))))
```

rezultatul se construiește în variabila fact
pe măsură ce **avansăm** în recursivitate

rezultatul apelului recursiv este
← rezultatul final al funcției

```
> (fact-tail-helper 3 1)
```

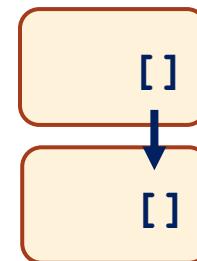
[]

Stiva procesului

Exemplu – recursivitate pe coadă

```
1. (define (fact-tail n)
2.       (fact-tail-helper n 1))
3.
4. (define (fact-tail-helper n fact)
5.       (if (zero? n)
6.           fact
7.           (fact-tail-helper (- n 1) (* n fact)))))
```

```
> (fact-tail-helper 3 1)
> (fact-tail-helper 2 3)
```

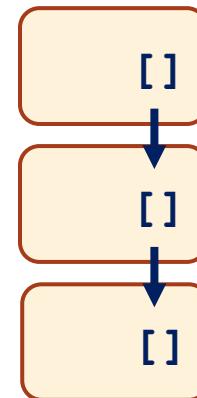


Stiva procesului

Exemplu – recursivitate pe coadă

```
1. (define (fact-tail n)
2.       (fact-tail-helper n 1))
3.
4. (define (fact-tail-helper n fact)
5.       (if (zero? n)
6.           fact
7.           (fact-tail-helper (- n 1) (* n fact)))))

> (fact-tail-helper 3 1)
> (fact-tail-helper 2 3)
> (fact-tail-helper 1 6)
```

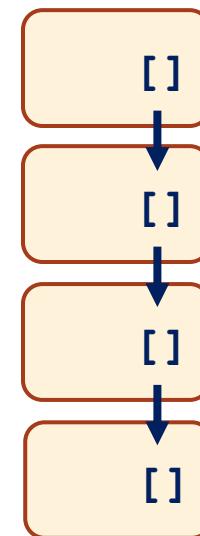


Stiva procesului

Exemplu – recursivitate pe coadă

```
1. (define (fact-tail n)
2.       (fact-tail-helper n 1))
3.
4. (define (fact-tail-helper n fact)
5.       (if (zero? n)
6.           fact
7.           (fact-tail-helper (- n 1) (* n fact)))))

> (fact-tail-helper 3 1)
> (fact-tail-helper 2 3)
> (fact-tail-helper 1 6)
> (fact-tail-helper 0 6)
```



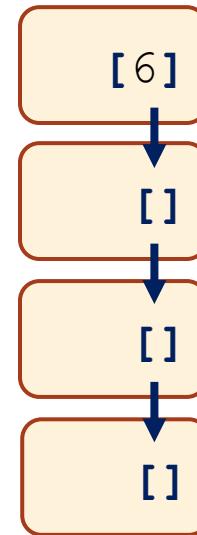
Stiva procesului

Exemplu – recursivitate pe coadă

```
1. (define (fact-tail n)
2.       (fact-tail-helper n 1))
3.
4. (define (fact-tail-helper n fact)
5.       (if (zero? n)
6.           fact
7.           (fact-tail-helper (- n 1) (* n fact)))))

> (fact-tail-helper 3 1)
> (fact-tail-helper 2 3)
> (fact-tail-helper 1 6)
> (fact-tail-helper 0 6)
> 6
```

rezultatul celui mai adânc apel recursiv se va transmite neschimbă către apelul inițial

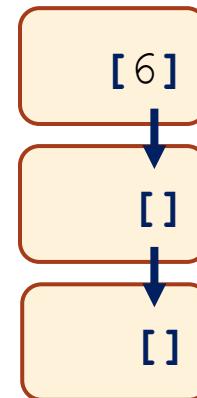


Stiva procesului

Exemplu – recursivitate pe coadă

```
1. (define (fact-tail n)
2.       (fact-tail-helper n 1))
3.
4. (define (fact-tail-helper n fact)
5.       (if (zero? n)
6.           fact
7.           (fact-tail-helper (- n 1) (* n fact)))))

> (fact-tail-helper 3 1)
> (fact-tail-helper 2 3)
> (fact-tail-helper 1 6)
> (fact-tail-helper 0 6)
> 6
```

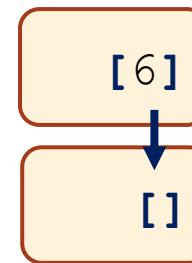


Stiva procesului

Exemplu – recursivitate pe coadă

```
1. (define (fact-tail n)
2.       (fact-tail-helper n 1))
3.
4. (define (fact-tail-helper n fact)
5.       (if (zero? n)
6.           fact
7.           (fact-tail-helper (- n 1) (* n fact)))))
```

```
> (fact-tail-helper 3 1)
> (fact-tail-helper 2 3)
> (fact-tail-helper 1 6)
> (fact-tail-helper 0 6)
> 6
```



Stiva procesului

Exemplu – recursivitate pe coadă

```
1. (define (fact-tail n)
2.       (fact-tail-helper n 1))
3.
4. (define (fact-tail-helper n fact)
5.       (if (zero? n)
6.           fact
7.           (fact-tail-helper (- n 1) (* n fact)))))
```

```
>(fact-tail-helper 3 1)
>(fact-tail-helper 2 3)
>(fact-tail-helper 1 6)
>(fact-tail-helper 0 6)
>6
```

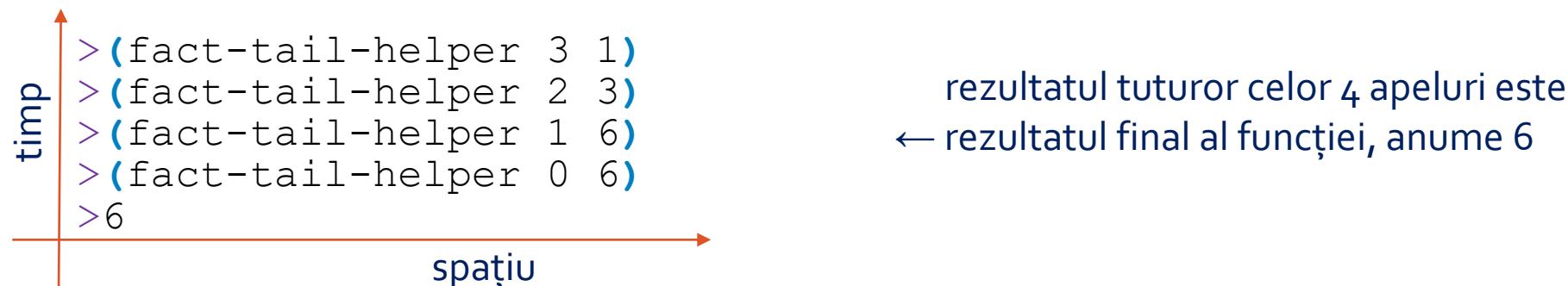
[6]

Stiva procesului

Observații – recursivitate pe coadă

Creșterea stivei nu mai este necesară. Rezultatul unui nou apel recursiv nu mai este așteptat de apelul părinte pentru a participa la un nou calcul, ci este chiar rezultatul final. Astfel, contextul apelului părinte poate fi șters din memorie. Această optimizare se numește **tail-call optimization** și este realizată de un compilator intelligent care detectează situația în care apelul recursiv este „la coadă” (nu mai participă la calcule ulterioare).

- **Timp:** $\Theta(n)$ (se efectuează n înmulțiri)
- **Spațiu:** $\Theta(1)$ (ocupat de variabilele n și $fact$, care rețin starea programului)
- **Calcul:** realizat integral pe avansul în recursivitate



Tipuri de recursivitate – Cuprins

- Importanța recursivității în paradigma funcțională
- Recursivitate pe stivă
- Recursivitate pe coadă
- Recursivitate arborescentă
- Comparație între tipurile de recursivitate
- Transformarea în recursivitate pe coadă

Exemplu – recursivitate arborescentă

```
1. (define (fibo-stack n)
2.   (if (< n 2)
3.     n
4.     (+ (fibo-stack (- n 1)) (fibo-stack (- n 2))))))
```

rezultatele a 2 apeluri recursive sunt așteptate pentru a fi adunate între ele

```
> (fibo-stack 3)
> (fibo-stack 2)
> > (fibo-stack 1)
< <1
> > (fibo-stack 0)
< <0
< 1
> (fibo-stack 1)
< 1
<2
```

← (fibo-stack 1) se calculează de 2 ori, iar numărul de calcule redundante crește exponențial cu n

(fib 5)

(fib 4)

(fib 3)

(fib 3)

(fib 2)

(fib 2)

(fib 1)

(fib 2) (fib 1)

(fib 1) (fib 0)

(fib 1) (fib 0)

1

(fib 1) (fib 0) 1

1

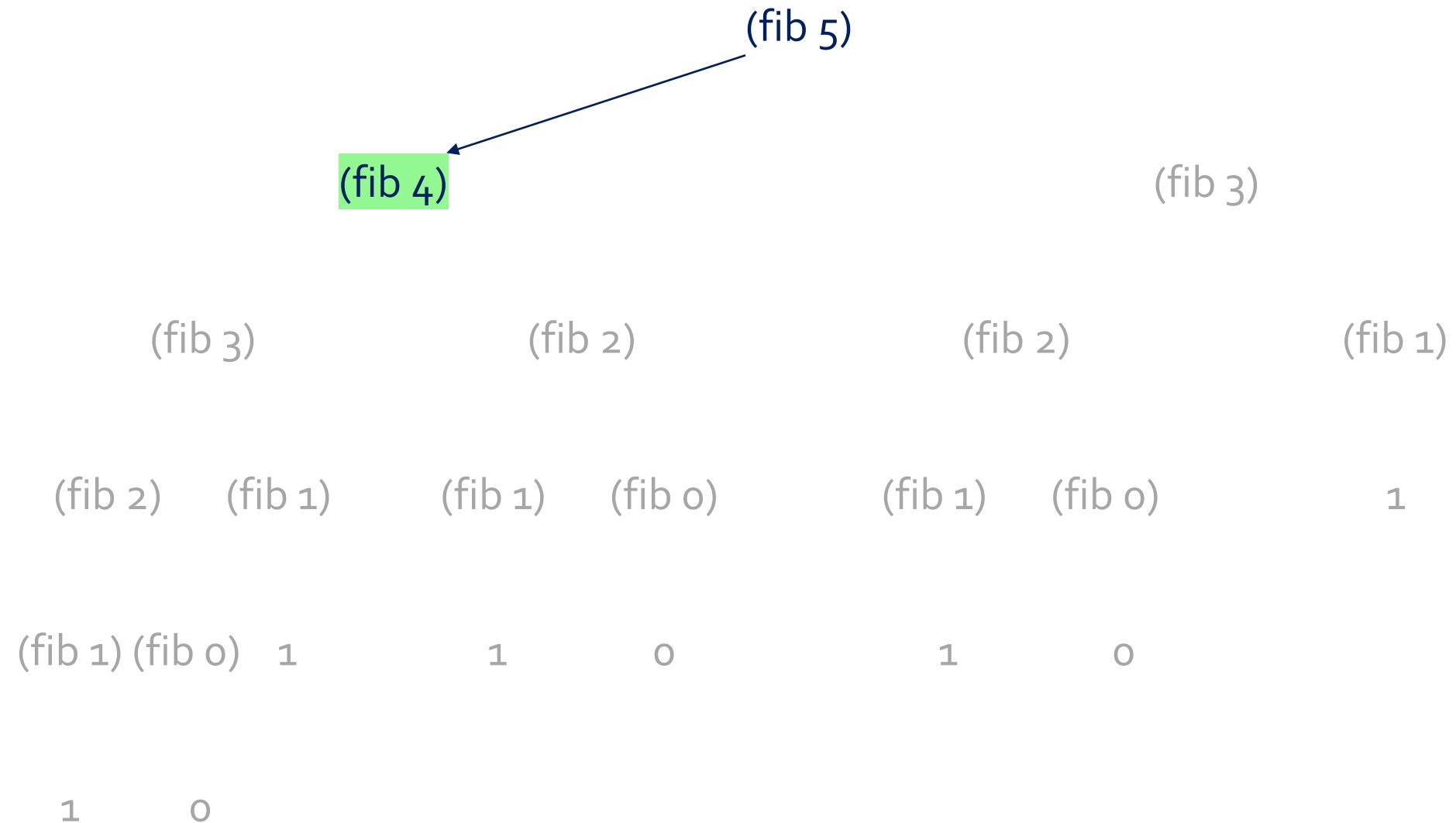
0

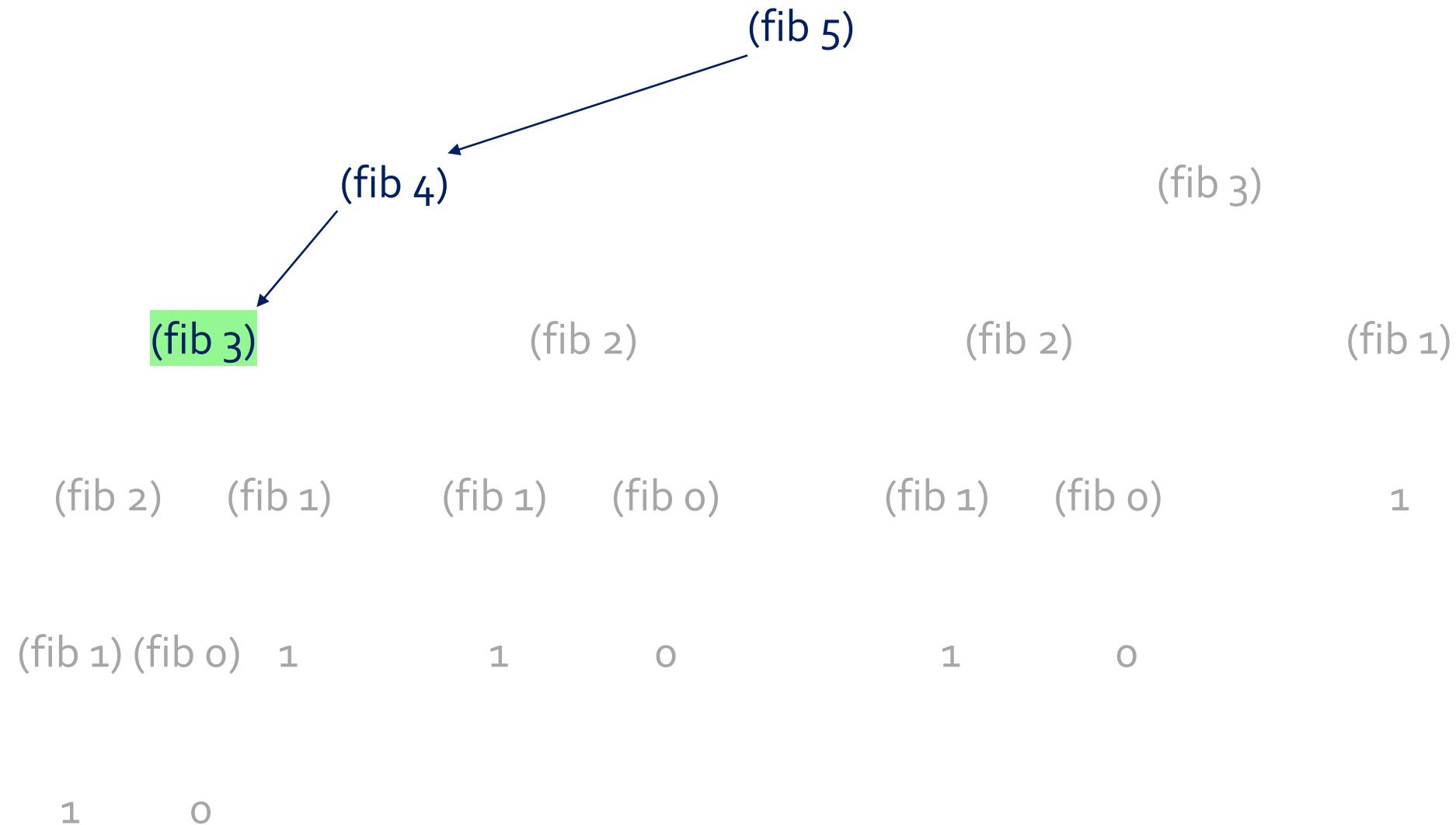
1

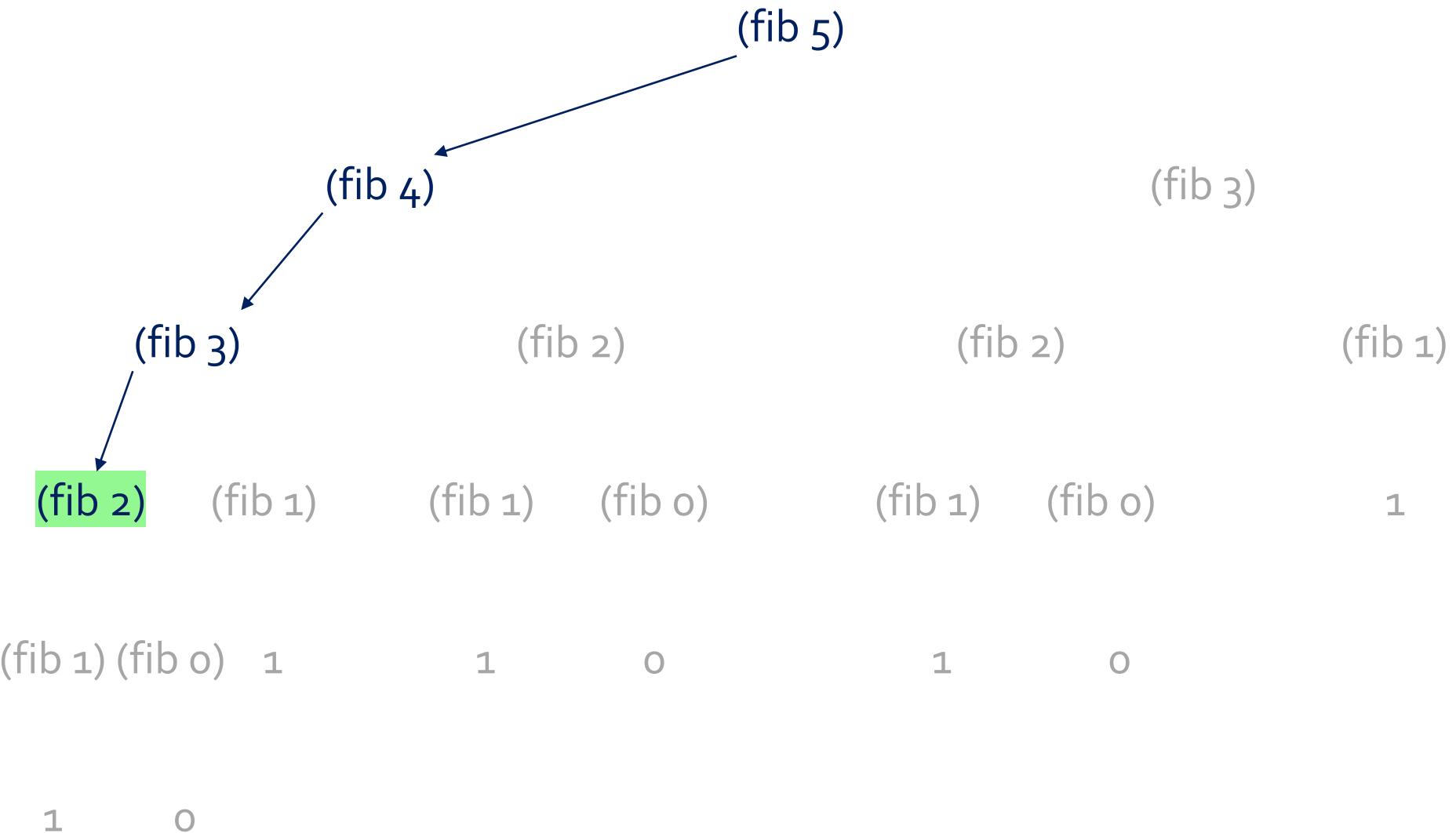
0

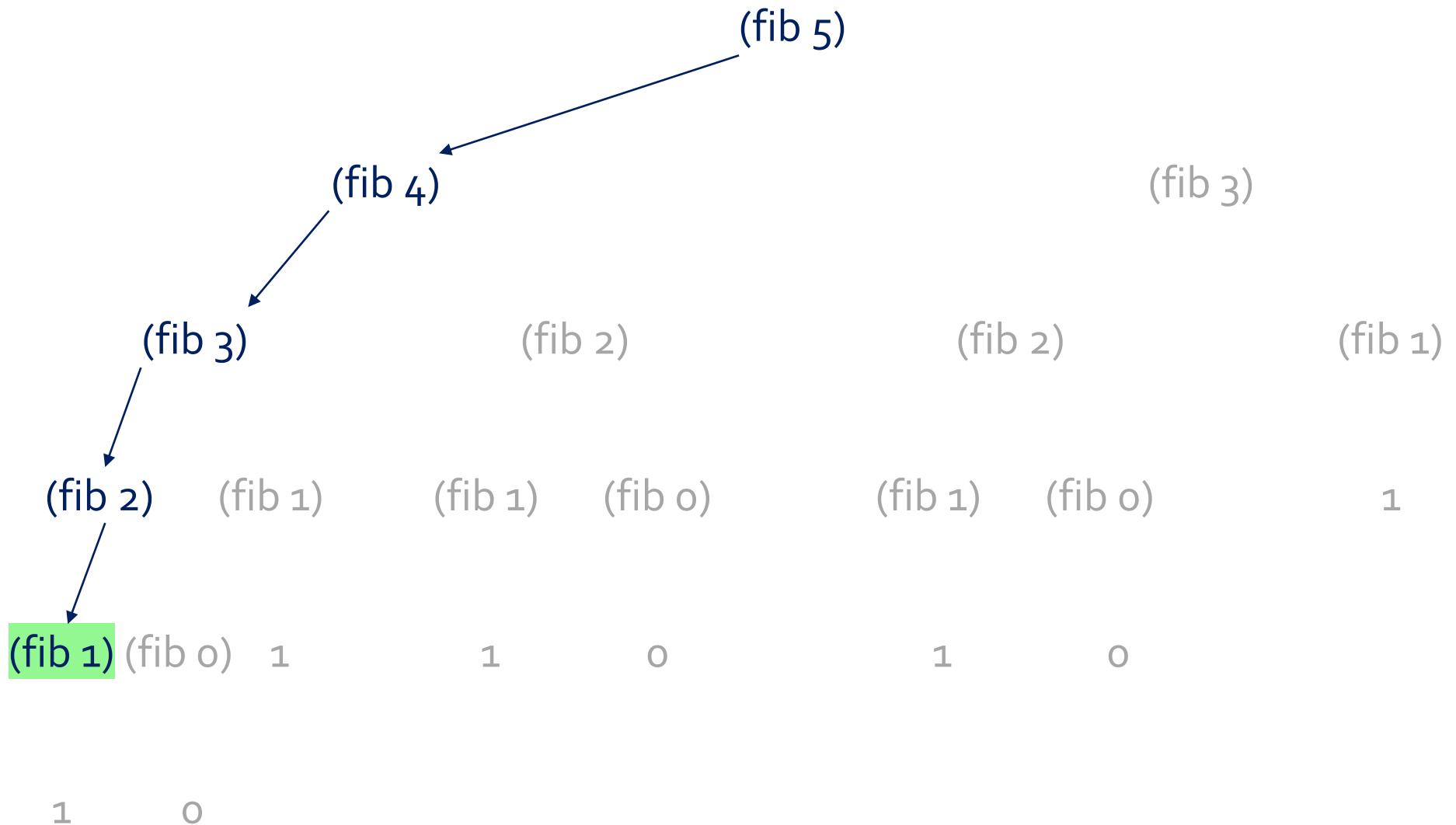
1

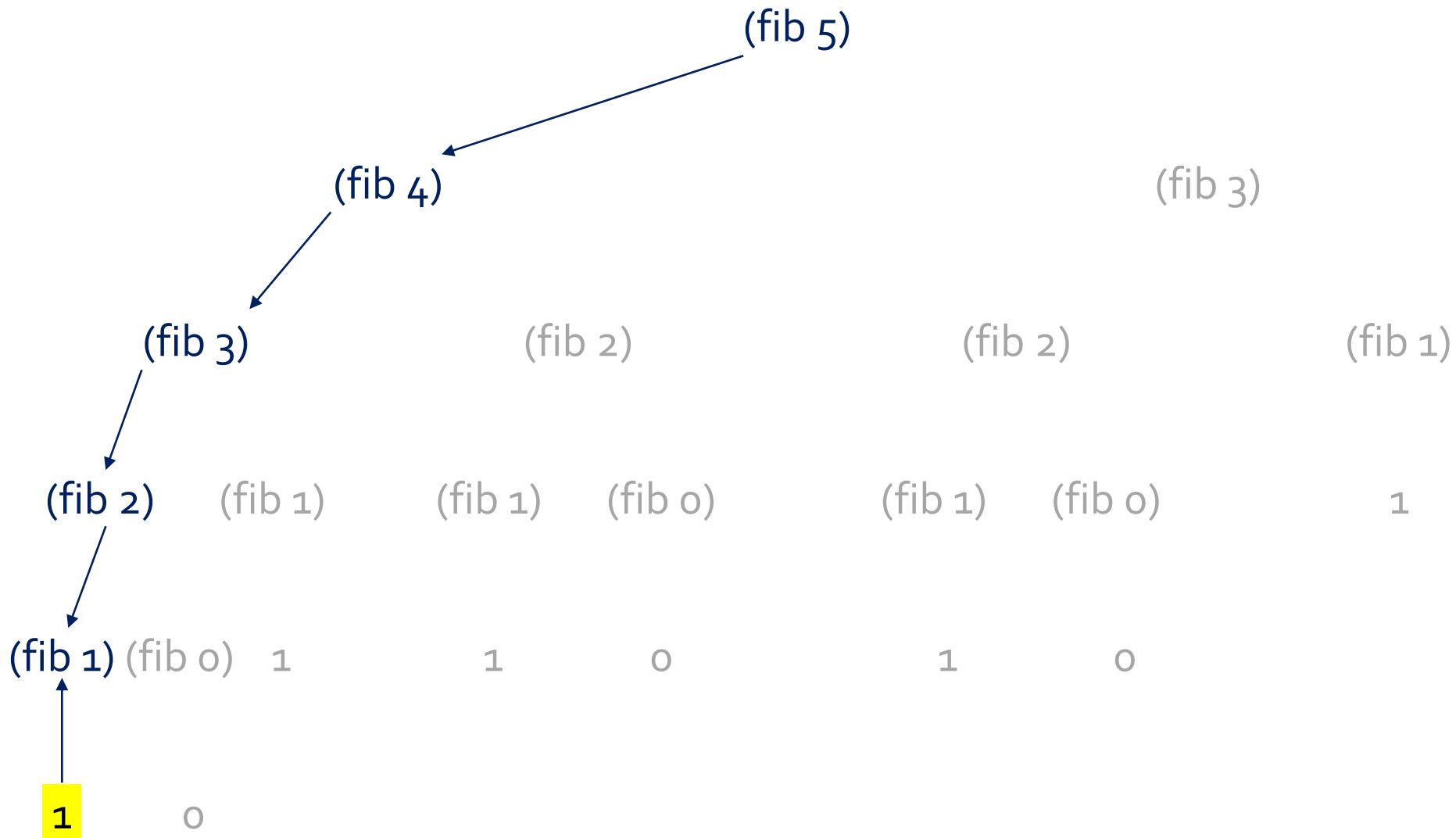
0

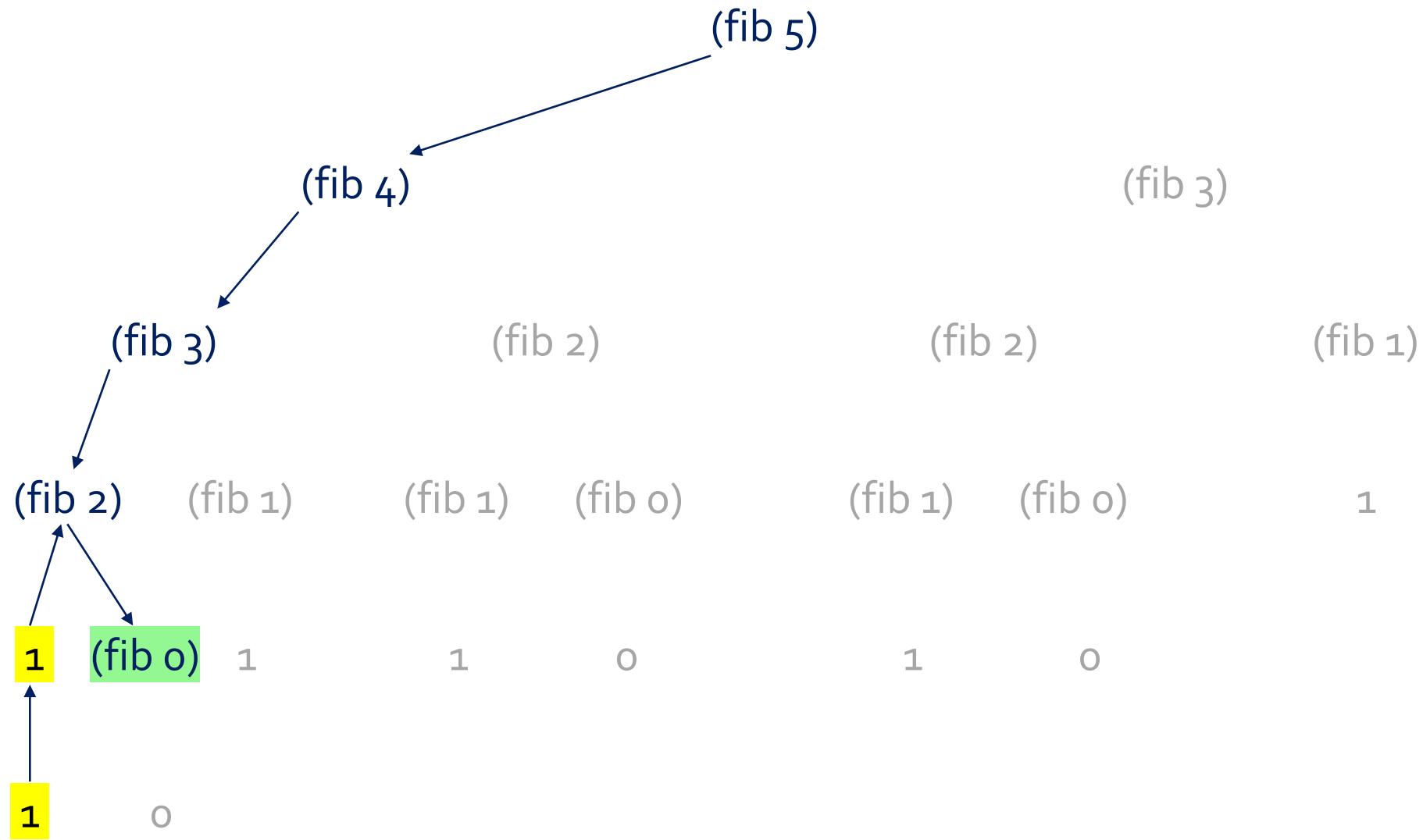


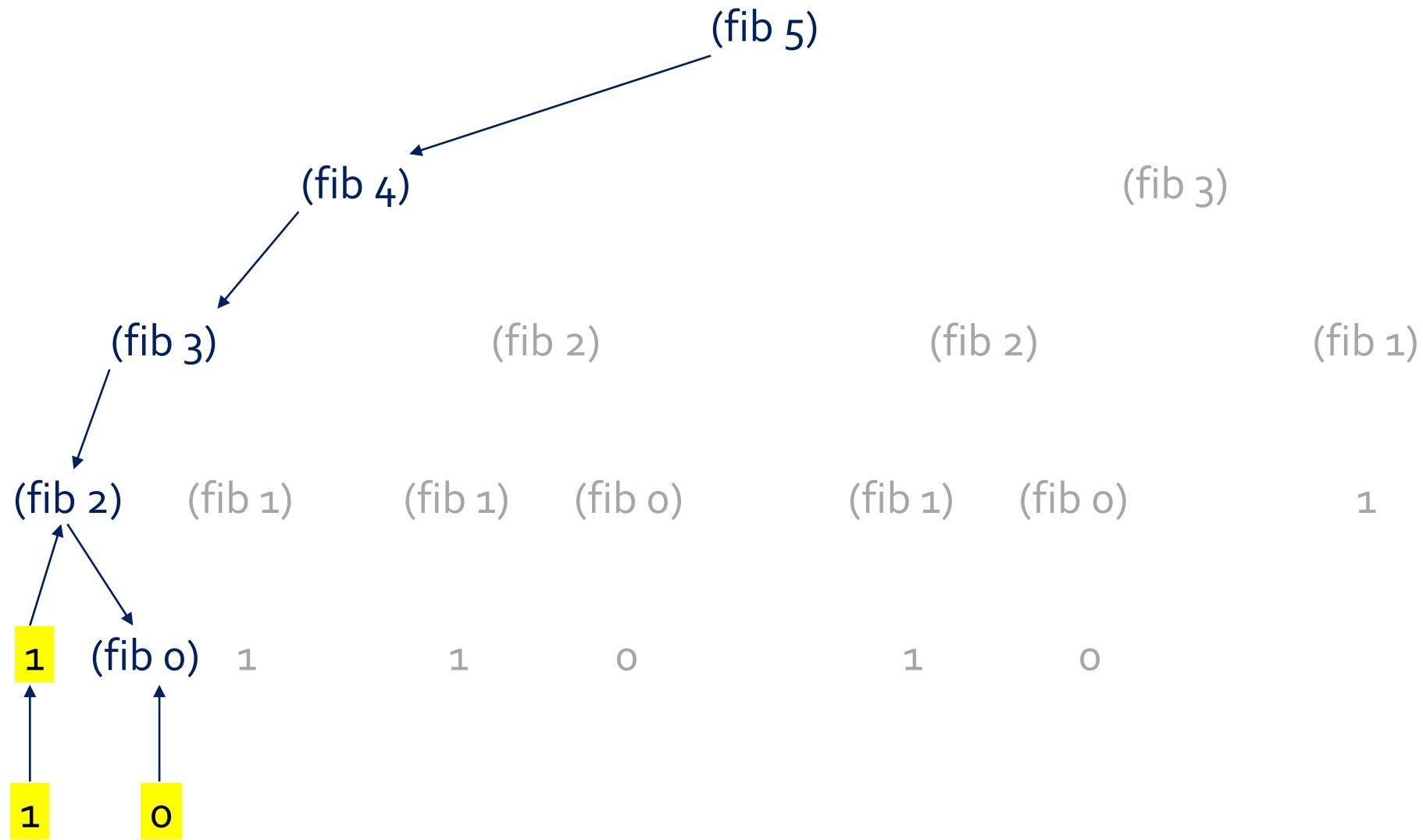


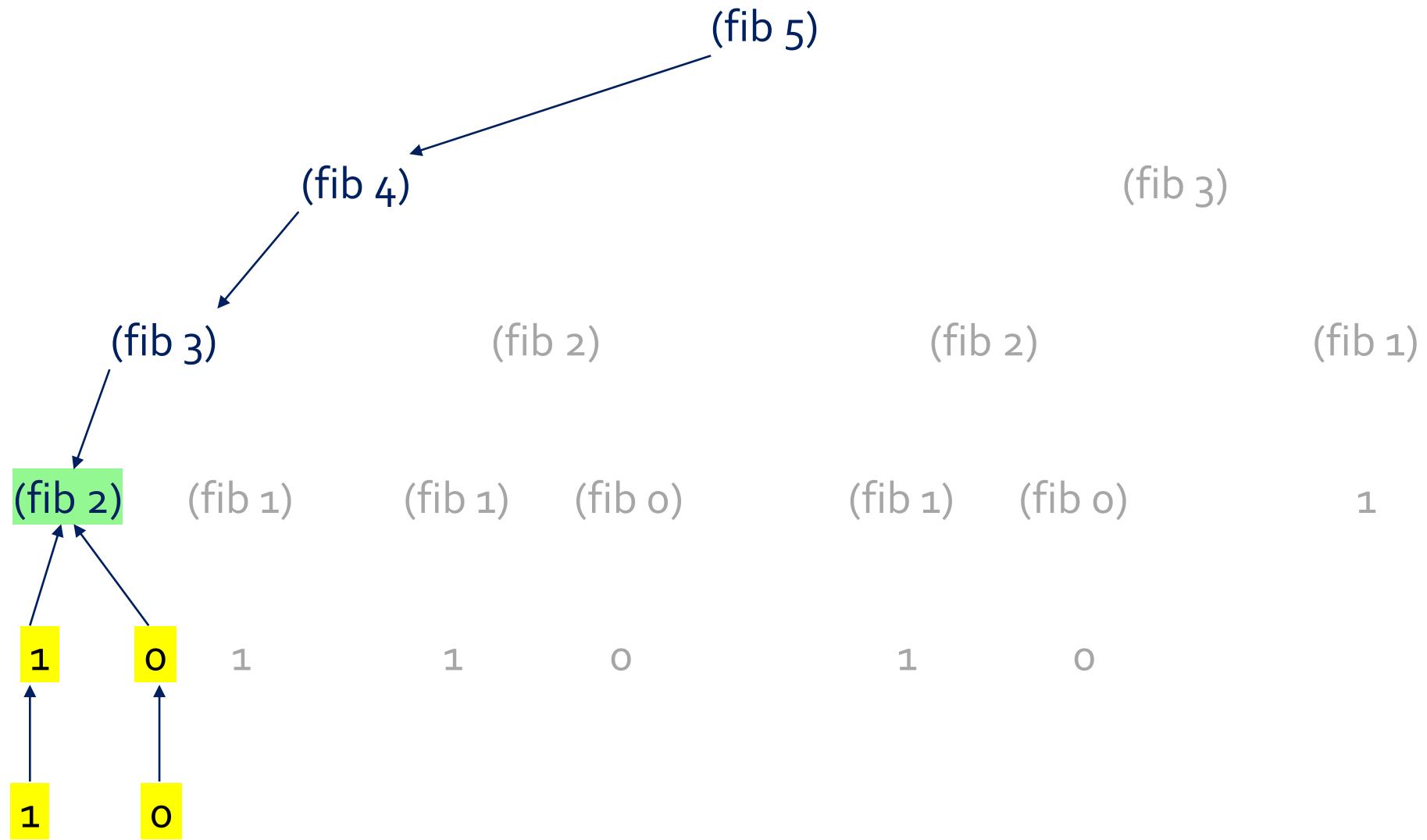


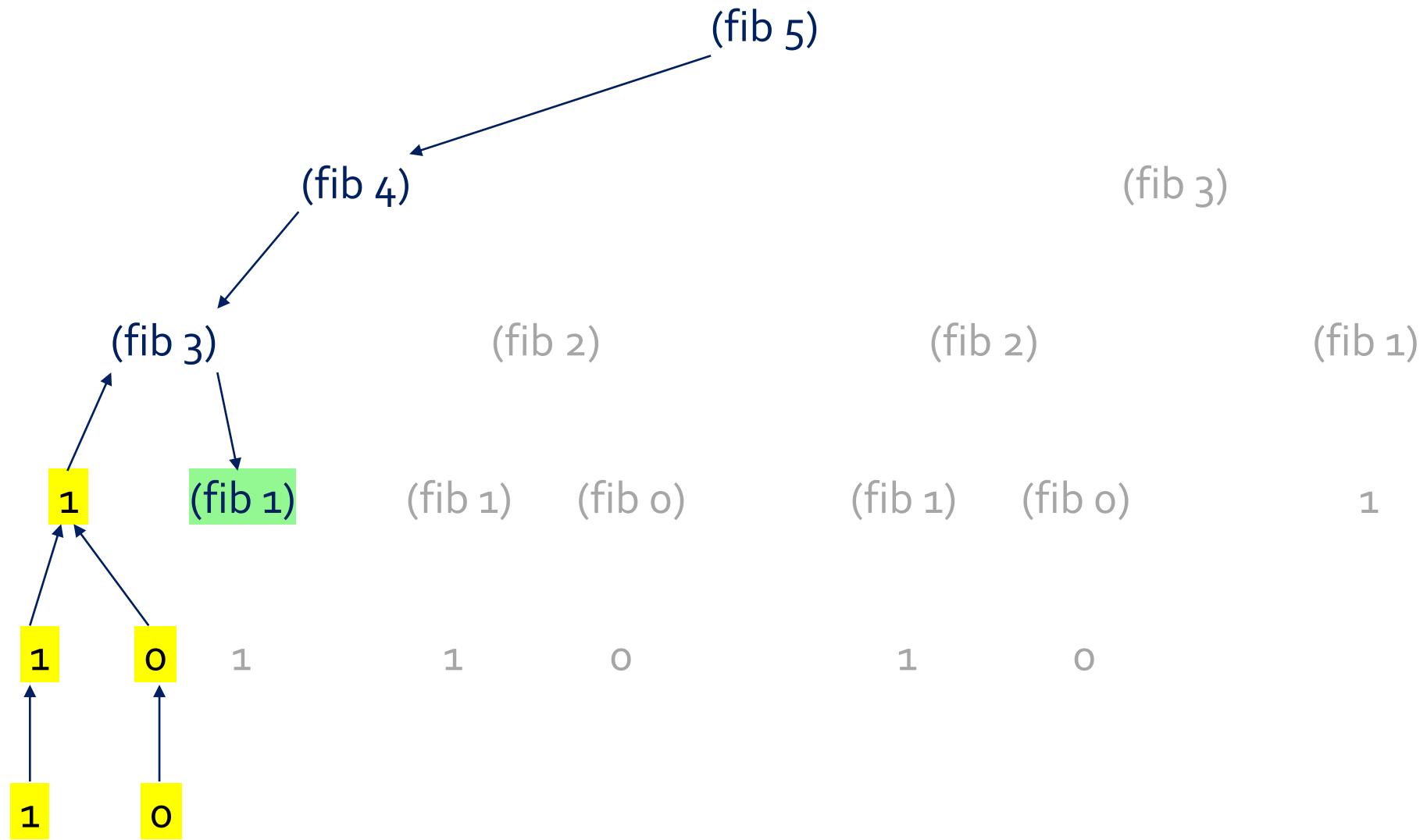


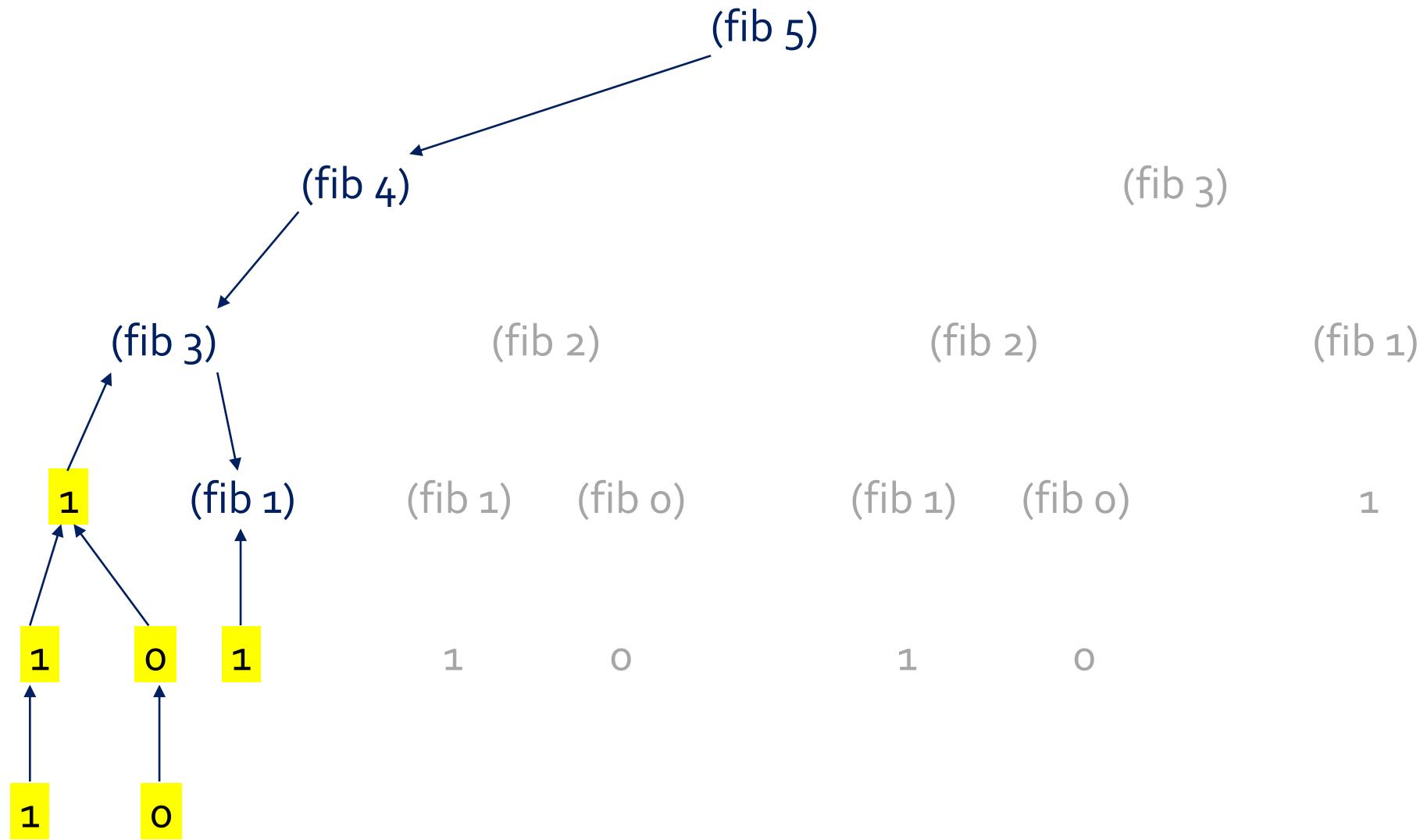


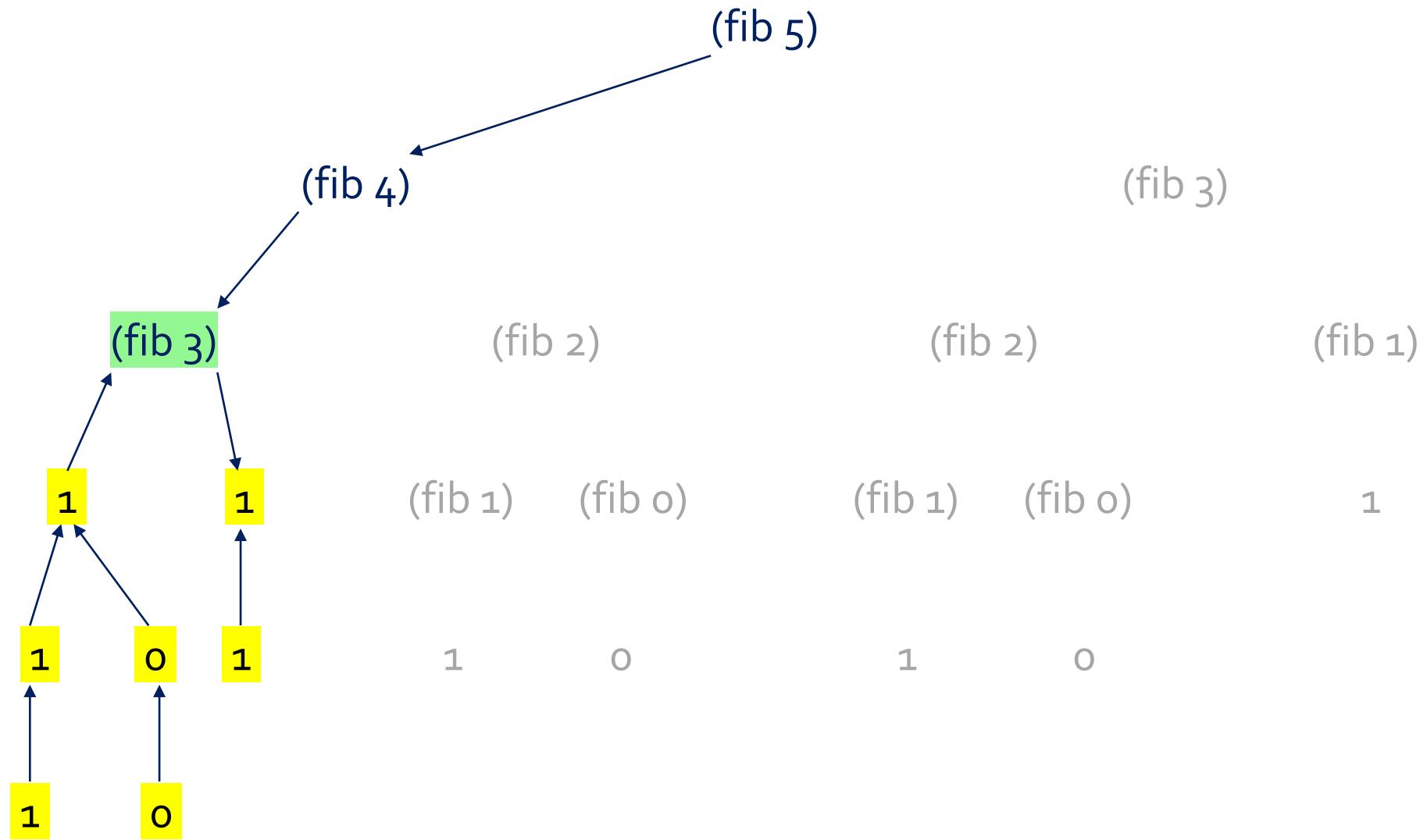


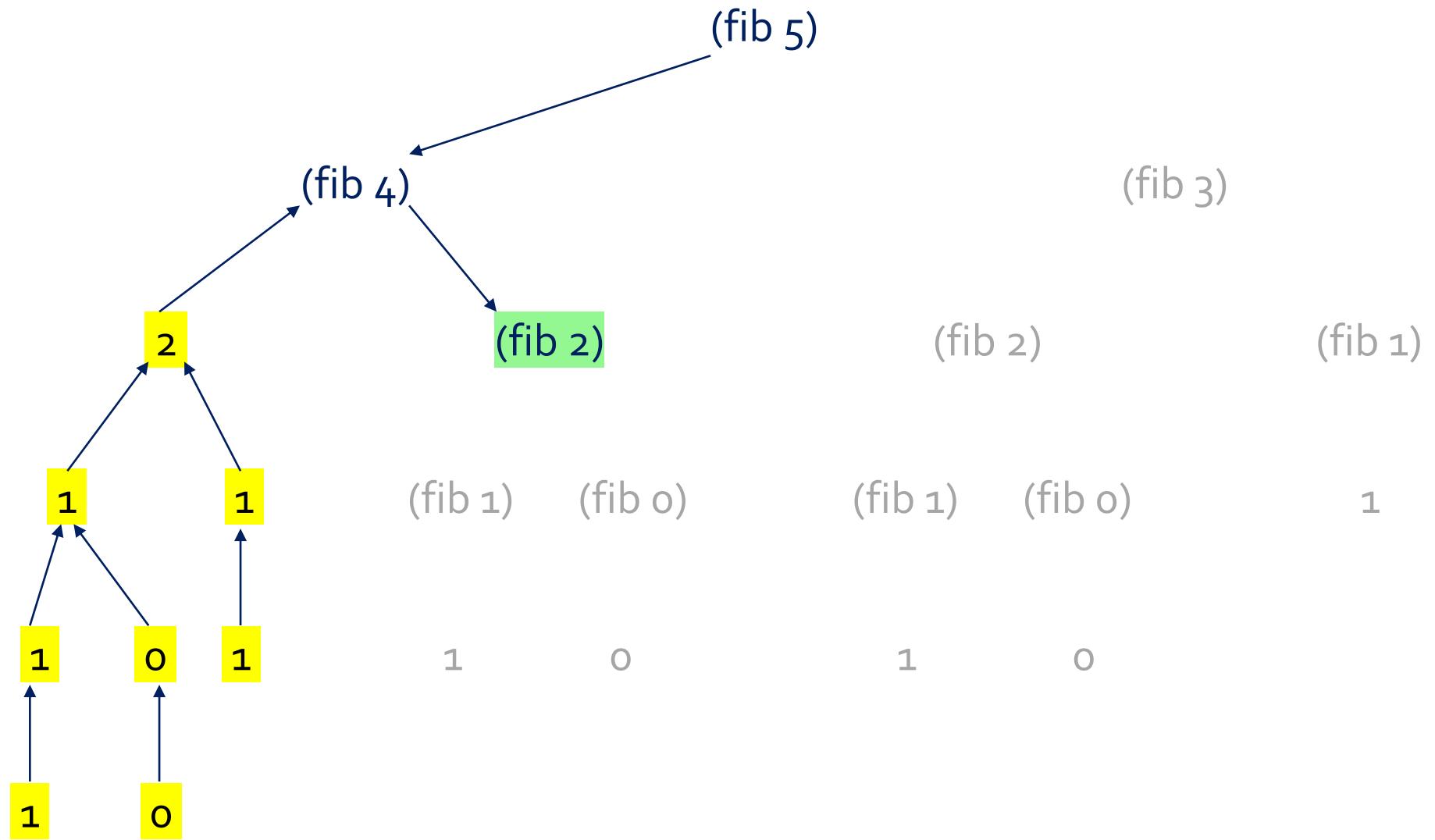


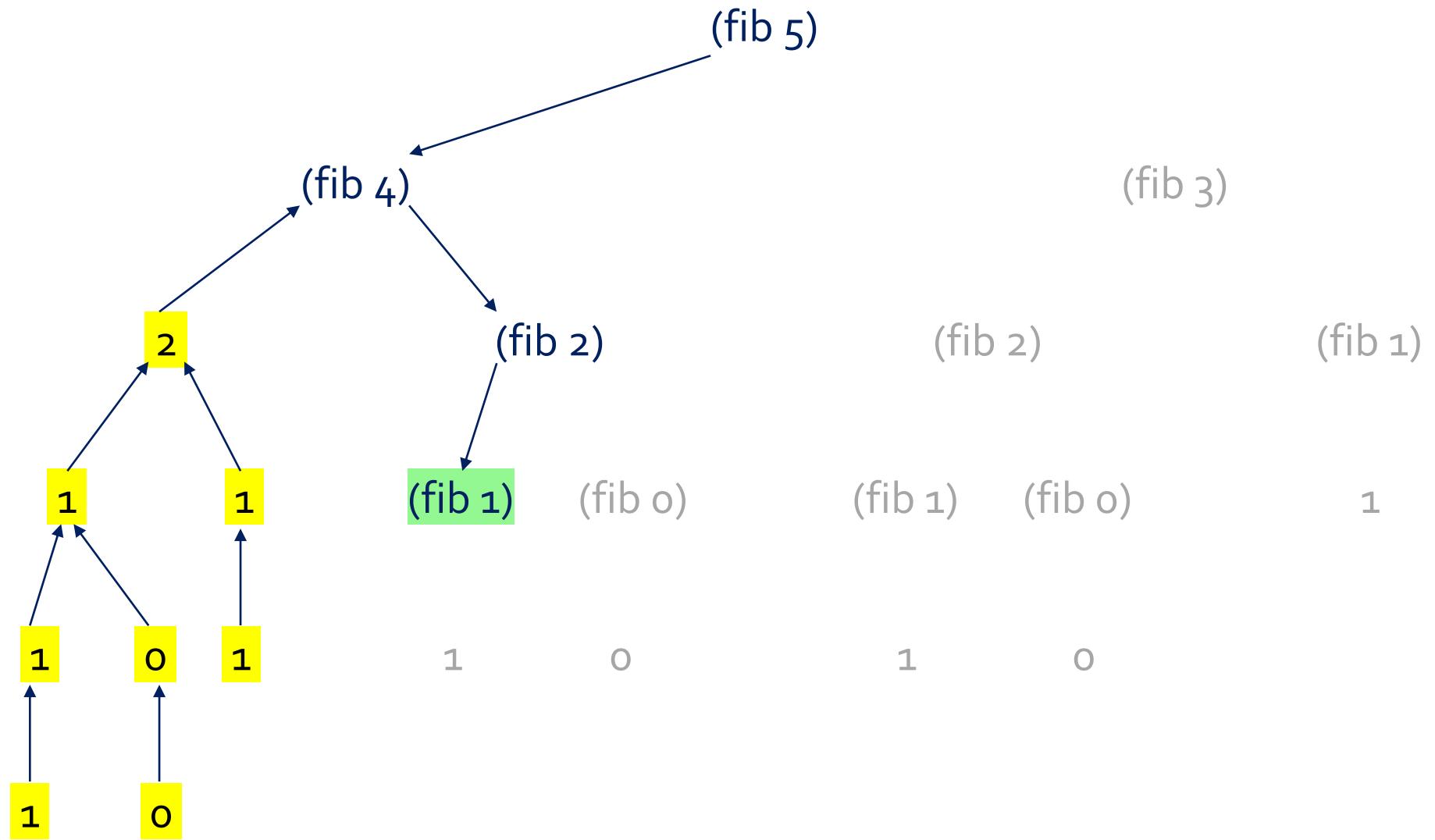


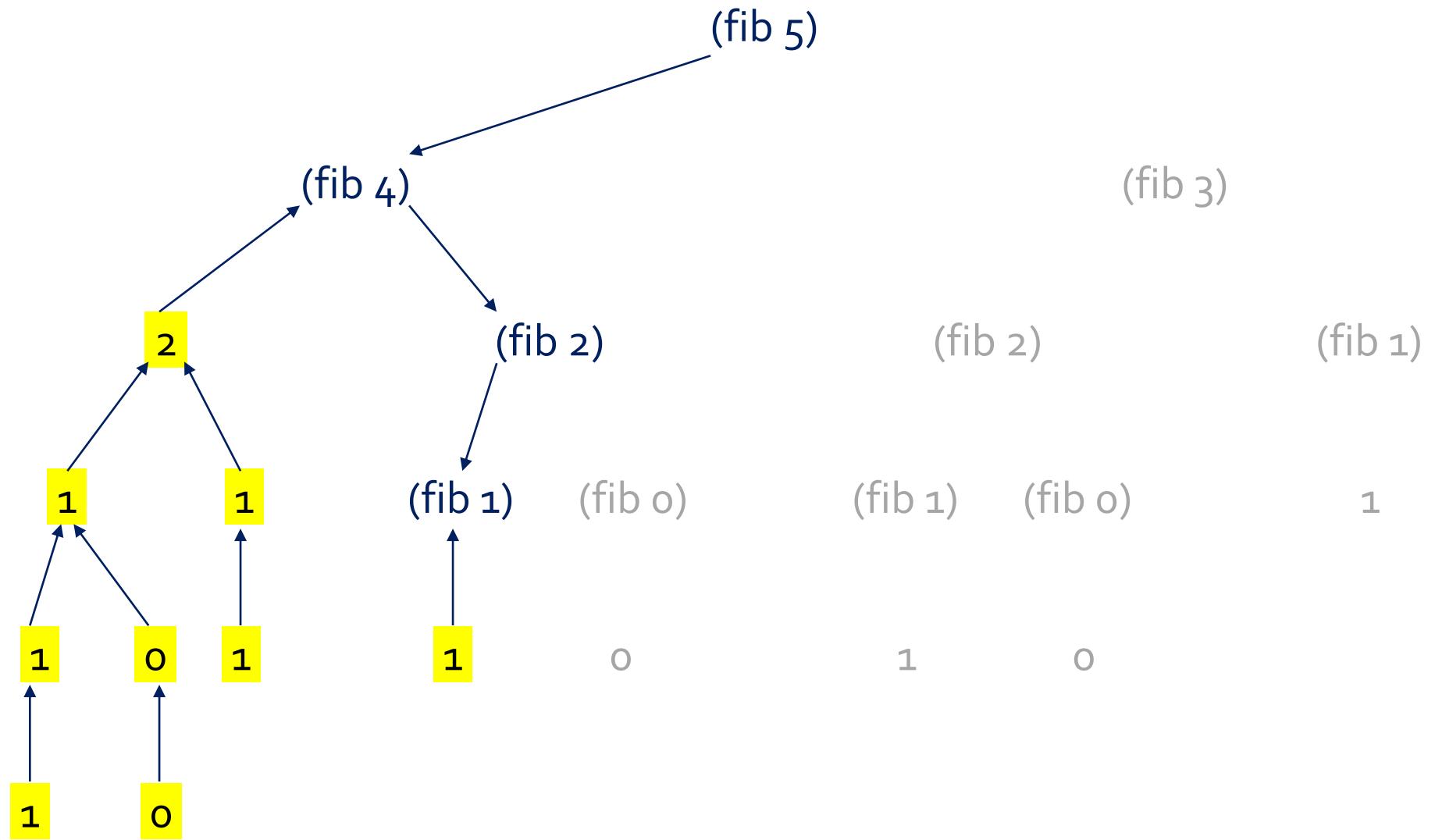


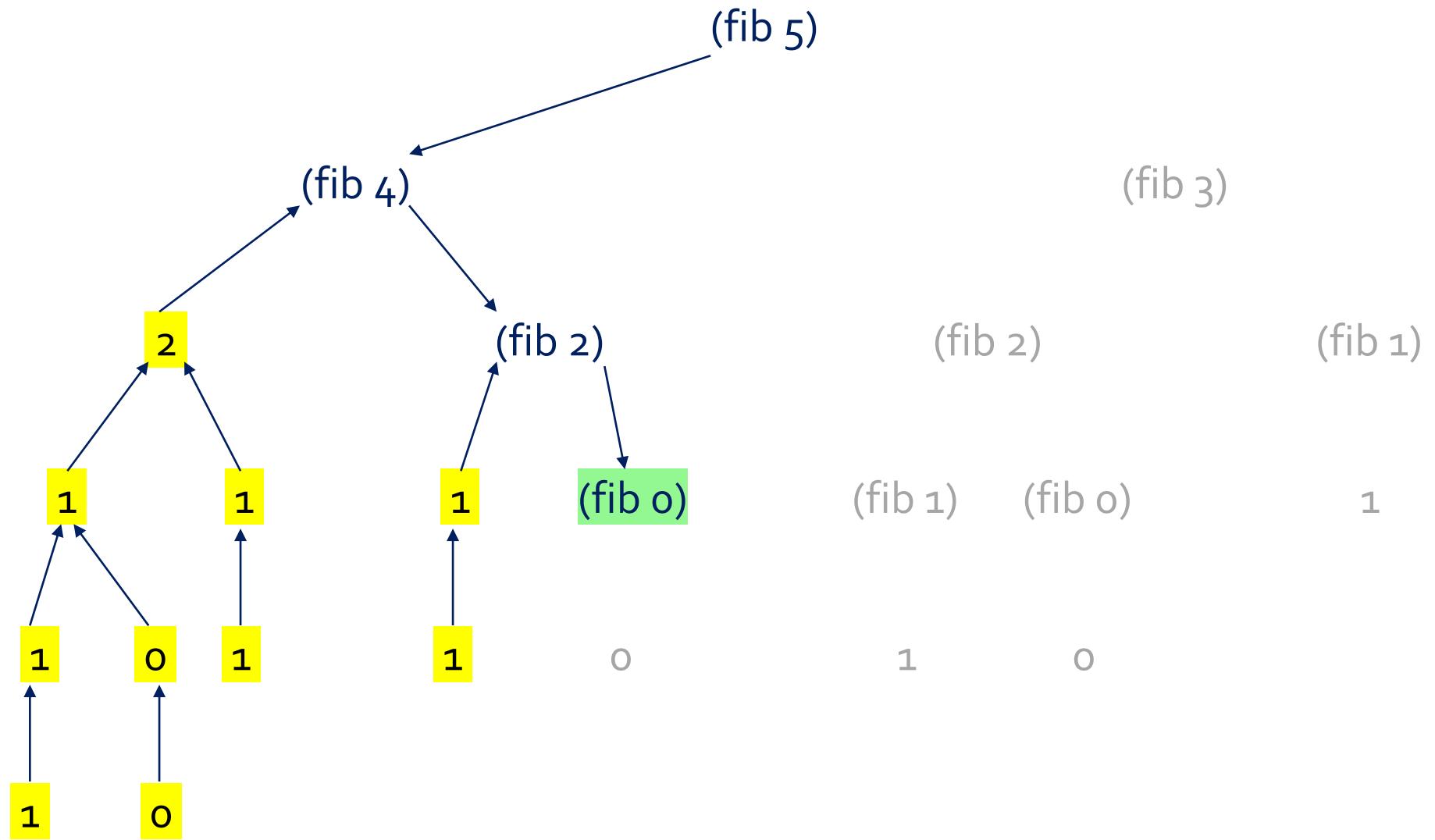


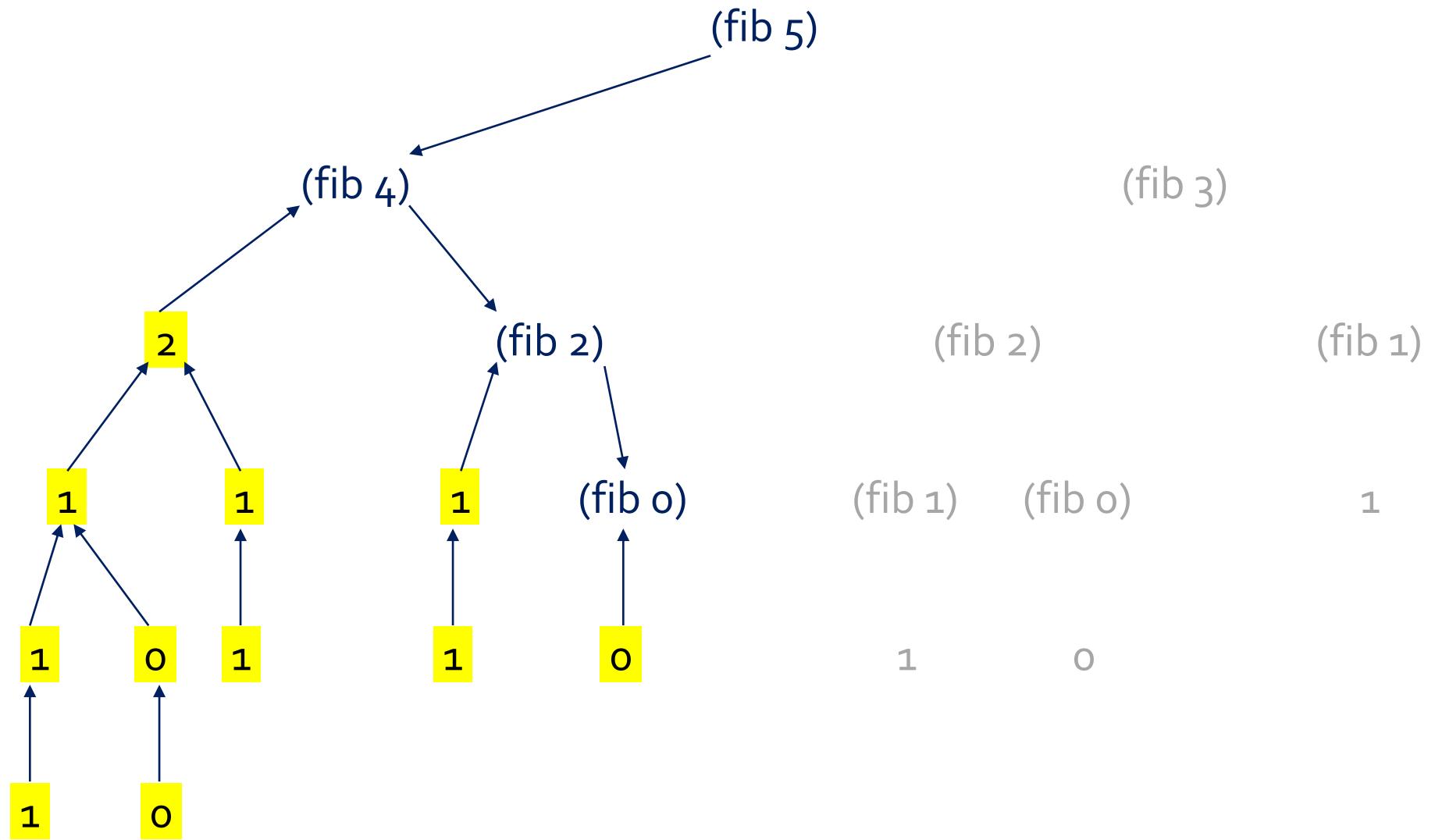


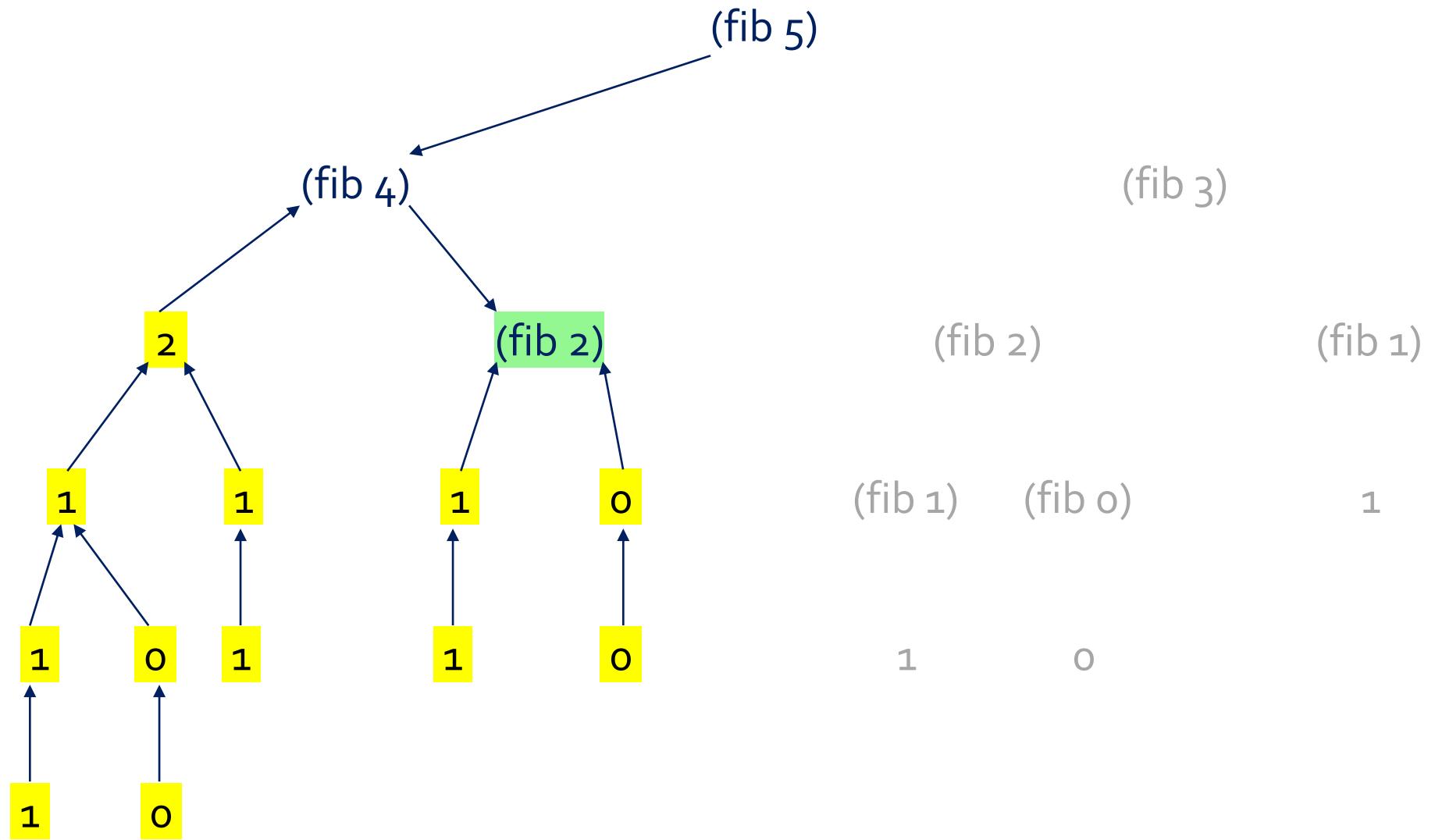


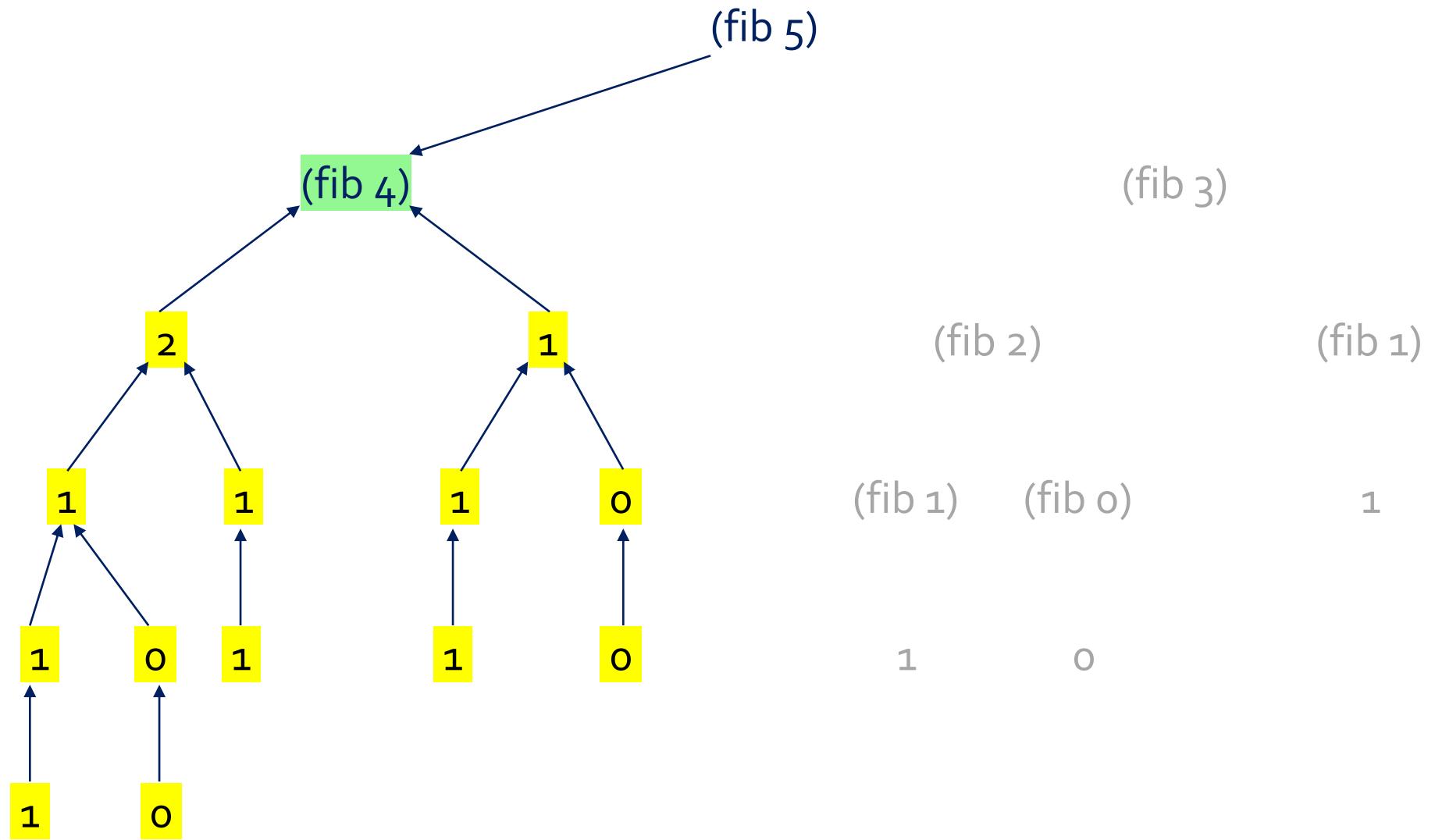


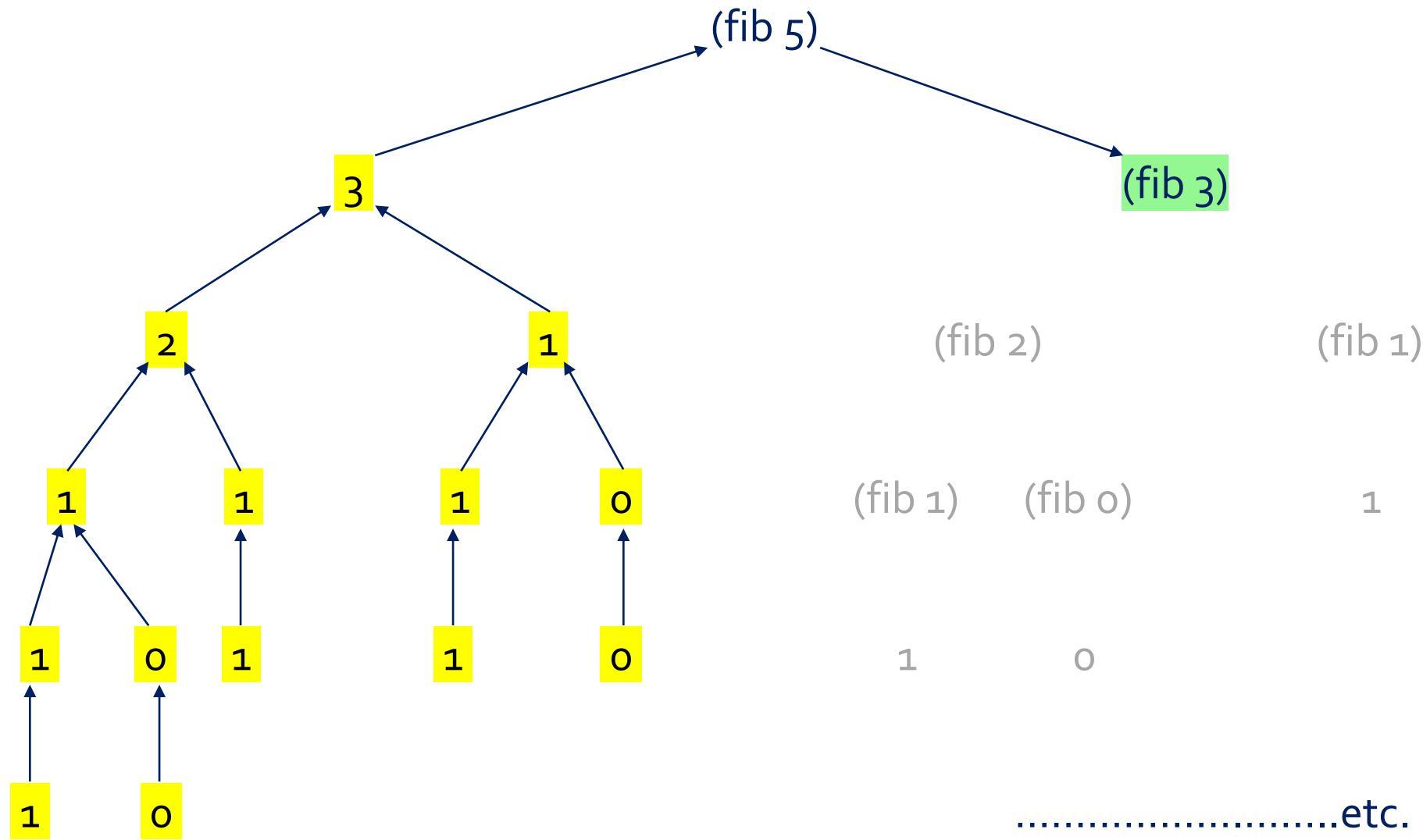


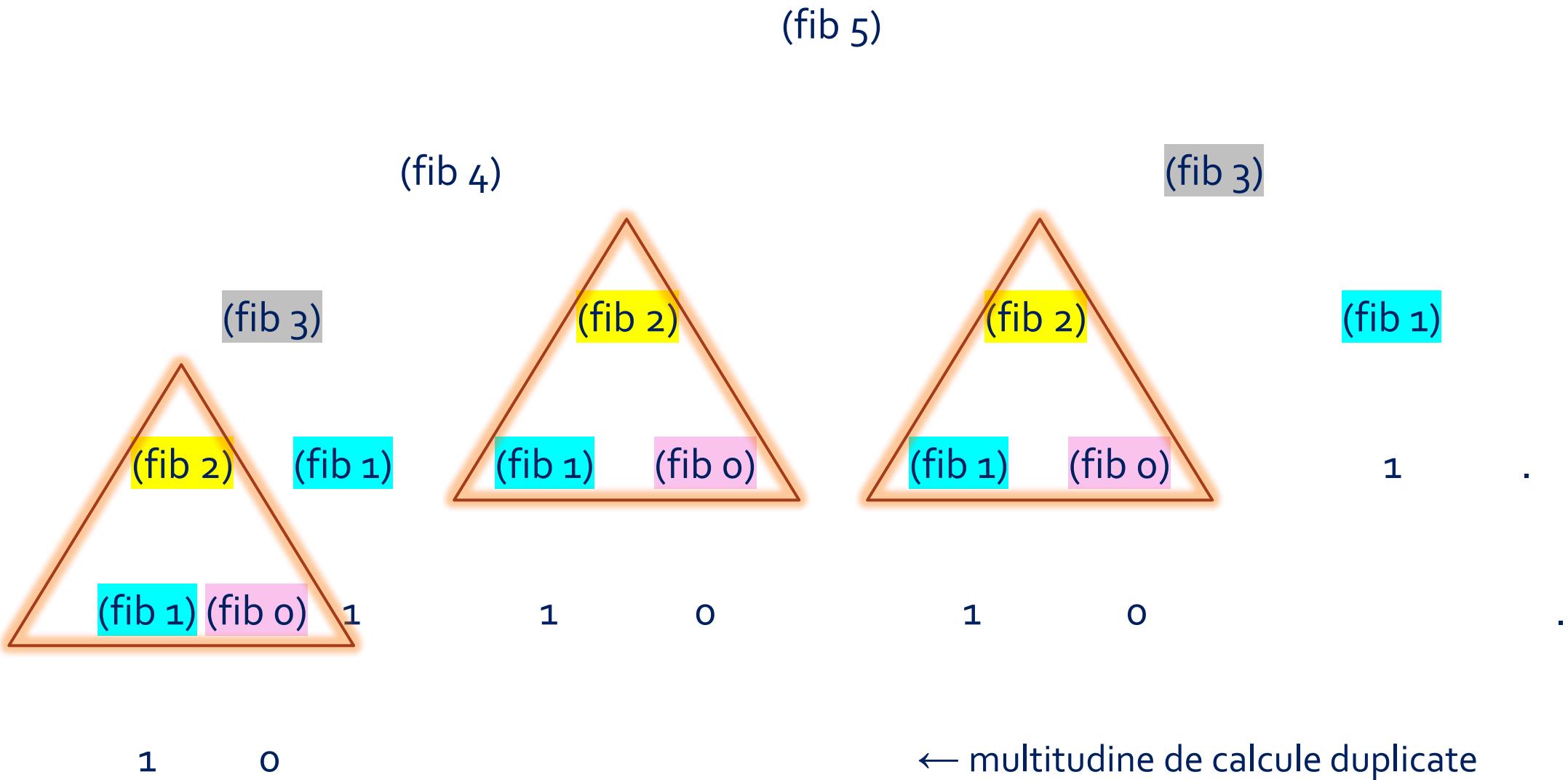






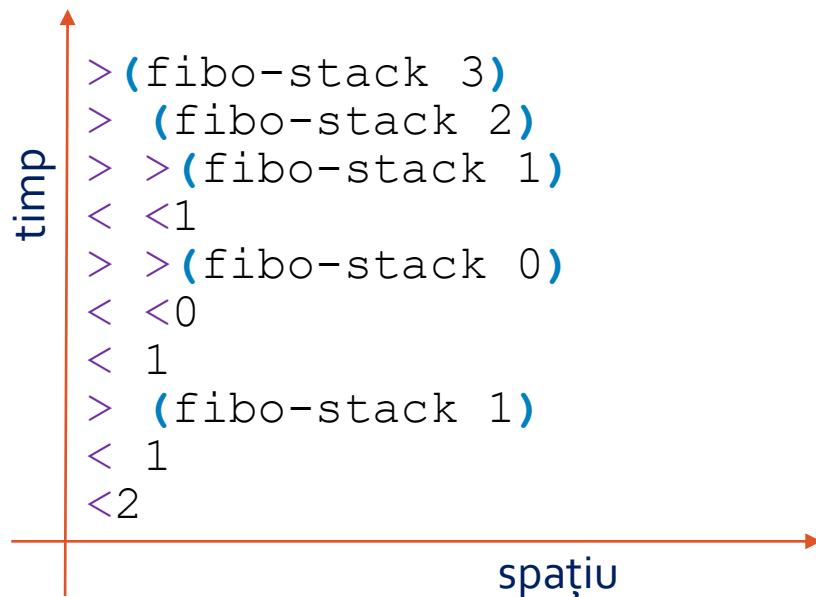






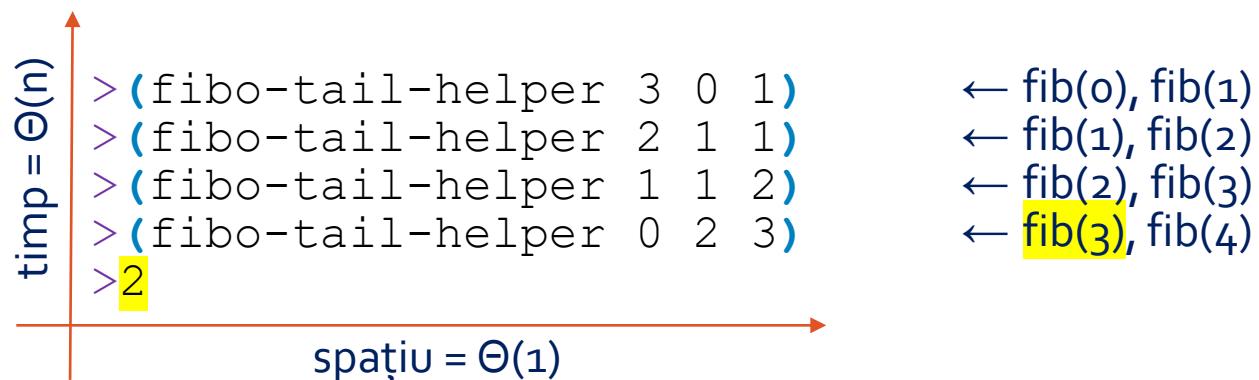
Observații – recursivitate arborescentă

- **Timp:** $\Theta(\text{fib}(n+1))$ (arborele are $2 \cdot \text{fib}(n+1) - 1$ noduri)
- **Spațiu:** $\Theta(n)$ (stiva la un moment dat reprezintă o singură cale în arbore)
- **Calcul:** realizat integral la revenirea din recursivitate



Fibonacci cu recursivitate pe coadă

```
1. (define (fib-tail n)
2.       (fib-tail-helper n 0 1))
3.
4. (define (fib-tail-helper n a b)
5.   (if (zero? n)
6.       a
7.       (fib-tail-helper (- n 1) b (+ a b)))))
```



Tipuri de recursivitate – Cuprins

- Importanța recursivității în paradigma funcțională
- Recursivitate pe stivă
- Recursivitate pe coadă
- Recursivitate arborescentă
- Comparație între tipurile de recursivitate
- Transformarea în recursivitate pe coadă

Comparație

- **Recursivitate pe stivă:** ineficientă spațial din cauza memoriei ocupată de stivă
- **Recursivitate pe coadă:** eficientă spațial și (în general și) temporal
- **Recursivitate arborescentă:** ineficientă spațial din cauza stivei și temporal atunci când aceleși date sunt prelucrate de mai multe noduri din arbore

Atunci când există o soluție iterativă eficientă pentru problemă, aceasta se poate transpune în recursivitate pe coadă. Funcția rezultată va fi:

- Recursivă din punct de vedere textual (funcția se apelează pe ea însăși)
- Iterativă din punct de vedere al procesului generat la execuție
- Mai puțin elegantă decât variantele pe stivă / arborescentă care derivă direct din specificația formală

Cum recunoaștem tipul de recursivitate?

După:

- **numărul de apeluri** recursive pe care un apel le lansează
 - două sau mai multe apeluri → recursivitate arborescentă (care, implicit, folosește și stiva)
 - un singur apel → recursivitate pe stivă sau pe coadă
 - dacă fiecare ramură a unei expresii condiționale lansează maxim un apel recursiv, apelul părinte va lansa maxim un apel recursiv și recursivitatea va fi pe stivă sau pe coadă, nu arborescentă
- **poziția apelurilor** recursive în expresia care descrie valoarea de return
 - singurul apel recursiv e în poziție finală (valoarea sa este valoarea de return) → recursivitate pe coadă (condiția trebuie să fie îndeplinită de fiecare ramură a unei expresii condiționale)
 - există un apel care nu e în poziție finală → recursivitate pe stivă (sau arborescentă)

Exemplu

Ce tip de recursitate au funcțiile f și g de mai jos?

```
1. (define (f x)
2.   (cond ((zero? x) 0)
3.         ((even? x) (f (/ x 2)))
4.         (else (+ 1 (f (- x 1)))))))
5.
6. (define (g L result)
7.   (cond ((null? L) result)
8.         ((list? (car L)) (g (cdr L) (append (g (car L) '()) result)))
9.         (else (g (cdr L) (cons (car L) result))))))
```

Exemplu

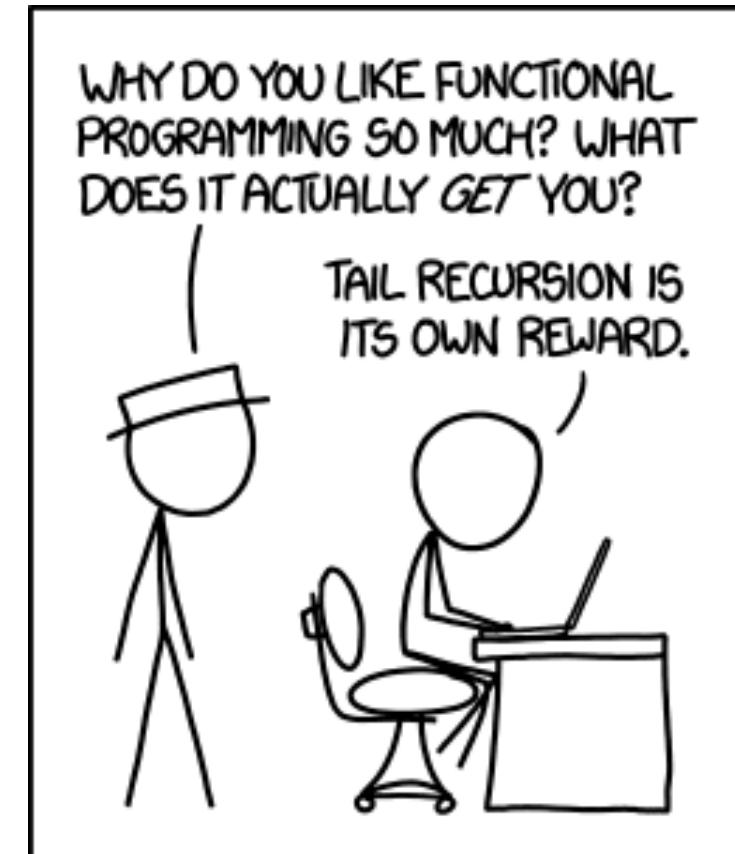
Ce tip de recursivitate au funcțiile f și g de mai jos?

```
1. (define (f x) ← pe stivă
   2.   (cond ((zero? x) 0)
            ((even? x) (f (/ x 2)))
            (else (+ 1 (f (- x 1))))))
   3.
   4.
   5.
   6. (define (g L result) ← arborescentă
      7.   (cond ((null? L) result)
            ((list? (car L)) (g (cdr L) (append (g (car L) '()) result)))
            (else (g (cdr L) (cons (car L) result))))))
```

Existența unei variabile de tip
acumulator nu înseamnă că avem
recursivitate pe coadă!

Tipuri de recursivitate – Cuprins

- Importanța recursivității în paradigma funcțională
- Recursivitate pe stivă
- Recursivitate pe coadă
- Recursivitate arborescentă
- Comparație între tipurile de recursivitate
- Transformarea în recursivitate pe coadă



Transformarea în recursivitate pe coadă

```
(define (fact-stack n)
  (if (zero? n)
      1
      (* n
          (fact-stack (- n 1))))))
```

```
(define (fact-tail n)
  (fact-tail-helper n 1))
(define (fact-tail-helper n fact)
  (if (zero? n)
      fact
      (fact-tail-helper (- n 1)
                        (* n fact)))))
```

- Helper-ul are un argument în plus: **acumulatorul** în care construim rezultatul pe avansul în recursivitate
- Calculul la care urma să participe rezultatul apelului recursiv este **calculul la care participă acumulatorul**
- La ieșirea din recursivitate, valoarea de return nu este **valoarea pe cazul de bază**, ci **acumulatorul**
- Valoarea funcției pe cazul de bază corespunde adesea **valorii inițiale a accumulatorului**

Când acumulatorul este o listă

```
(define (get-odd-stack L)
  (cond
    ((null? L) '())
    ((even? (car L)) (get-odd-stack (cdr L)))
    (else (cons (car L) (get-odd-stack (cdr L)))))))
```

```
>(get-odd-stack '(1 4 6 3))
>(cons 1 (get-odd-stack '(4 6 3)))
>(cons 1 (get-odd-stack '(6 3)))
>(cons 1 (get-odd-stack '(3)))
>(cons 1 (cons 3 (get-odd-stack '())))
>(cons 1 (cons 3 '()))
>(cons 1 '(3))
>'(1 3)
```

```
(define (get-odd-tail L acc)
  (cond
    ((null? L) acc)
    ((even? (car L)) (get-odd-tail (cdr L) acc))
    (else (get-odd-tail (cdr L) (cons (car L) acc))))))
```

```
>(get-odd-tail '(1 4 6 3) '())
>(get-odd-tail '(4 6 3) '(1))
>(get-odd-tail '(6 3) '(1))
>(get-odd-tail '(3) '(1))
>(get-odd-tail '() '(3 1))
>'(3 1)
```

ordine inversă!

Când accumulatorul este o listă

Soluții pentru conservarea ordinii

- Inversarea accumulatorului înainte de return (pe cazul de bază)

```
.... (cond ((null? L) (reverse acc)) ....
```

- Adăugarea fiecărui nou element la sfârșitul accumulatorului (cu append în loc de cons)

```
.... (else (get-odd-tail (cdr L) (append acc (list (car L))))) ....
```

Complexitate

- Inversare: $\Theta(n)$ (dată de complexitatea lui reverse)
- append în loc de cons: $\Theta(n^2)$ ($\Theta(\text{length}(acc))$ pentru fiecare append în parte)
 $(0 + 1 + 2 + \dots + (n-1))$

Complexitate **reverse** și **append**

```
1. (define (reverse L) (rev L '()))
2. (define (rev L acc)
3.   (if (null? L)
4.     acc
5.     (rev (cdr L) (cons (car L) acc)))) → Θ(length(L))
6.
7. (define (append A B)
8.   (if (null? A)
9.     B
10.    (cons (car A) (append (cdr A) B))))) → Θ(length(A))
```

Concluzie: Pentru eficiență folosim inversarea acumulatorului la final, nu adăugarea fiecărui element la sfârșit.

Calcul Lambda



Freedom
from
state

Calcul Lambda – Cuprins

- Aparițiile unei variabile într-o λ -expresie
- Statutul variabilelor într-o λ -expresie
- Evaluarea unei λ -expresii
- Forma normală a unei λ -expresii

Memento: λ -expresia

Sintaxa

$e \equiv$ **x** variabilă
| **$\lambda x.e_1$** funcție (unară, anonimă) cu parametrul formal x și corpul e_1
| **($e_1 e_2$)** aplicație a expresiei e_1 asupra parametrului efectiv e_2

Semantica (Modelul substituției)

Pentru a evalua **$(\lambda x.e_1 e_2)$** (funcția cu parametrul formal x și corpul e_1 , aplicată pe e_2):

- Peste tot în e_1 , identificatorul x este înlocuit cu e_2
- Se evaluatează noul corp e_1 și se întoarce rezultatul (se notează **$e_1[e_2/x]$**)

Aparițiile unei variabile într-o λ -expresie

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

Variabila x: 3 apariții conform cărora putem rescrie expresia ca $\lambda x_1 . (x_2 \ \lambda y . x_3)$

Variabila y: 1 apariție conform căreia putem rescrie expresia ca $\lambda x . (x \ \lambda y_1 . x)$

Vom distinge între:

- variabilele al căror nume nu contează (și ar putea fi oricare altul)
și
- variabilele ar căror nume este un alias pentru valori din exteriorul expresiei

Apariții legate / libere într-o expresie

Apariția x_n este **legată** în E dacă:

- $E = \dots \lambda x_n . e \dots$
- $E = \dots \lambda x . e \dots$ și x_n apare în e

Variabila de după λ = variabila de legare

Apariția de după λ = apariția care leagă (restul aparițiilor lui x în corpul e)

Altfel, apariția x_n este **liberă** în E.

$$\lambda x_1 . (x_2 \ \lambda y . x_3)$$

↑ ↑ ↑
legată legată legată
(apariția care leagă) (apariții în corpul funcției de parametru x)

Exemple de apariții legate / libere

Vom marca aparițiile **legate** ale fiecărei variabile cu **portocaliu** și pe cele **libere** cu **verde**.

$(x \ \lambda y. x)$

$(y \ \lambda y. x)$

$\lambda z. ((+ z) \ x)$

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

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

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

Exemple de apariții legate / libere

Vom marca aparițiile **legate** ale fiecărei variabile cu **portocaliu** și pe cele **libere** cu **verde**.

(**x** λ **y**.**x**)

(**y** λ **y**.**x**)

λ **z**.((+ **z**) **x**)

(λ **x**. λ **y**.(**x** **y**) **y**)

λ **x**.(**y** λ **y**.(**x** (**y** **z**)))

(**x** λ **x**.(λ **x**.**y** λ **y**.(**x** **z**)))

Exemple de apariții legate / libere

Vom marca aparițiile **legate** ale fiecărei variabile cu **portocaliu** și pe cele **libere** cu **verde**.

$(x \ \lambda y. x)$

$(y \ \lambda y. x)$

$\lambda z. (+ z) x$

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

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

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

Exemple de apariții legate / libere

Vom marca aparițiile **legate** ale fiecărei variabile cu **portocaliu** și pe cele **libere** cu **verde**.

$(x \ \lambda y. x)$

$(y \ \lambda y. x)$

$\lambda z. ((+ z) \ x)$

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

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

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

Exemple de apariții legate / libere

Vom marca aparițiile **legate** ale fiecărei variabile cu **portocaliu** și pe cele **libere** cu **verde**.

$(x \ \lambda y. x)$

$(y \ \lambda y. x)$

$\lambda z. ((+ z) x)$

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

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

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

Exemple de apariții legate / libere

Vom marca aparițiile **legate** ale fiecărei variabile cu **portocaliu** și pe cele **libere** cu **verde**.

$(x \ \lambda y. x)$

$(y \ \lambda y. x)$

$\lambda z. ((+ z) \ x)$

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

$\lambda x. (\textcolor{green}{y} \ \lambda y. (\textcolor{red}{x} \ (\textcolor{red}{y} \ z)))$

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

Exemple de apariții legate / libere

Vom marca aparițiile **legate** ale fiecărei variabile cu **portocaliu** și pe cele **libere** cu **verde**.

$(x \ \lambda y. x)$

$(y \ \lambda y. x)$

$\lambda z. ((+ z) \ x)$

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

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

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

Observații

- O apariție este legată sau liberă **într-o expresie**

$$E = \lambda x. (y \lambda y. (x (y z))) =_{\text{notăție}} \lambda x. e$$

legată în E liberă în e

$$e = (y \lambda y. (x (y z)))$$

- Numele aparițiilor legate ale unei variabile nu contează (le putem redenumi pe toate cu un același nou identificator, semnificația expresiei rămânând aceeași)

$$E = \lambda x. (y \lambda y. (x (y z))) = \lambda a. (y \lambda b. (a (b z)))$$

(valoare externă lui E) acest y nu are nicio legătură cu acest y (parametrul funcției interne)

Calcul Lambda – Cuprins

- Aparițiile unei variabile într-o λ -expresie
- Statutul variabilelor într-o λ -expresie
- Evaluarea unei λ -expresii
- Forma normală a unei λ -expresii

Variabile legate / libere într-o expresie

Variabila x este **legată** în E dacă **toate aparițiile lui x sunt legate în E .**

Altfel, variabila x este **liberă** în E .

Exemplu: $(\text{green } x \lambda \text{ red } x . \text{red } x)$ din punct de vedere al statutului aparițiilor devine

$(\text{green } x \lambda \text{ green } x . \text{red } x)$ din punct de vedere al statutului variabilelor („din cauza” primului x).

Observații

- Ca și în cazul aparițiilor, o variabilă este legată sau liberă **într-o expresie**
- Ca și în cazul aparițiilor, **numele variabilelor legate nu contează**

Exemple de variabile legate / libere

Vom marca variabilele **legate** cu **portocaliu** și pe cele **libere** cu **verde**.

($x \ \lambda y. x$)

($y \ \lambda y. x$)

$\lambda z. ((+ z) x)$

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

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

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

Exemple de variabile legate / libere

Vom marca variabilele **legate** cu **portocaliu** și pe cele **libere** cu **verde**.

(**x** λ **y.****x**)

(**y** λ **y.****x**)

λ **z.**((+ **z**) **x**)

(λ **x.** λ **y.**(**x** **y**) **y**)

λ **x.**(**y** λ **y.**(**x** (**y** **z**)))

(**x** λ **x.**(λ **x.****y** λ **y.**(**x** **z**)))

Exemple de variabile legate / libere

Vom marca variabilele **legate** cu **portocaliu** și pe cele **libere** cu **verde**.

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

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

$\lambda z.((+ z) x)$

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

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

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

Exemple de variabile legate / libere

Vom marca variabilele **legate** cu **portocaliu** și pe cele **libere** cu **verde**.

$(x \lambda y. x)$

$(y \lambda y. x)$

$\lambda z. ((+ z) x)$

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

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

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

Exemple de variabile legate / libere

Vom marca variabilele **legate** cu **portocaliu** și pe cele **libere** cu **verde**.

$(x \lambda y. x)$

$(y \lambda y. x)$

$\lambda z. ((+ z) x)$

$(\lambda x. \lambda y. (x \ y) \ y)$ dar dacă ne limităm la subexpresia $\lambda y. (x \ y)$ statutul variabilelor se inversează!

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

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

Exemple de variabile legate / libere

Vom marca variabilele **legate** cu **portocaliu** și pe cele **libere** cu **verde**.

$(x \lambda y. x)$

$(y \lambda y. x)$

$\lambda z. ((+ z) x)$

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

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

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

Exemple de variabile legate / libere

Vom marca variabilele **legate** cu **portocaliu** și pe cele **libere** cu **verde**.

(**x** λy.x)

(**y** λy.x)

λz.((+ z) x)

(λx. λy.(x y) y)

λx.(y λy.(x (y z)))

(**x** λx.(λx.y λy.(x z)))

Calcul Lambda – Cuprins

- Aparițiile unei variabile într-o λ -expresie
- Statutul variabilelor într-o λ -expresie
- Evaluarea unei λ -expresii
- Forma normală a unei λ -expresii

β -redex și β -reducere

Memento: Semantica λ -expresiilor (Modelul substituției)

Pentru a evalua $(\lambda x. e_1 \ e_2)$ (funcția cu parametrul formal x și corpul e_1 , aplicată pe e_2):

- Peste tot în e_1 , identificatorul x este înlocuit cu e_2
- Peste tot în e_1 , aparițiile libere ale lui x (libere în e_1 !) sunt înlocuite cu e_2 (nu are sens să înlocuiesc și aparițiile legate, întrucât acestea se numesc tot x doar întâmplător)
- Se evaluatează noul corp e_1 și se întoarce rezultatul (se notează $e_1[e_2/x]$)

β -redex = λ -expresie de forma $(\lambda x. e_1 \ e_2)$

β -reducere = efectuarea calculului $(\lambda x. e_1 \ e_2) \rightarrow_{\beta} e_1[e_2/x]$

Greșeli apărute la β -reducere

Exemplu: $(\lambda x. \lambda y. (x \ y) \ y) \rightarrow_{\beta} \lambda y. (x \ y)_{[y/x]} = \lambda y. (y \ y)$ (greșit!)



Trebuia să obținem

o funcție care îl aplică pe y asupra argumentului său

Am obținut

\neq o funcție care își aplică argumentul asupra lui însuși

Ce a mers rău?

În urma înlocuirii, apariția lui y (care era liberă în e_2) s-a trezit legată în e_1 (la un argument cu care nu avea nicio legătură).

Generalizare

Conflict de nume între variabilele legate din e_1 și variabilele libere din $e_2 \rightarrow$ greșeli în evaluare

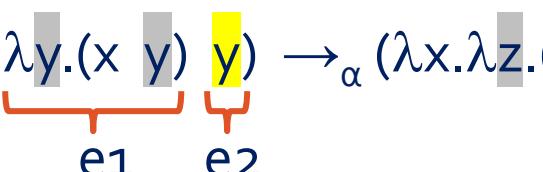
Soluția: α -conversia

α -conversie = redenumirea variabilelor legate din corpul unei funcții $\lambda x.e_1$ a.î. ele să nu coincidă cu variabilele libere din parametrul efectiv e_2 pe care aplicăm funcția

Observații

- Numele variabilelor legate oricum nu contează, deci ele pot fi redenumite
- Noul nume trebuie să nu intre în conflict cu variabilele libere din e_1 și din e_2

Exemplu: $(\lambda x. \lambda y. (x \ y) \ y) \rightarrow_{\alpha} (\lambda x. \lambda z. (x \ z) \ y) \rightarrow_{\beta} \lambda z. (x \ z)_{[y/x]} = \lambda z. (y \ z)$ (corect!)



Exemplu de secvență de reducere

$((\lambda x.\lambda y.\lambda z.(x(yz))\; z)\; y)$

Exemplu de secvență de reducere

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

Exemplu de secvență de reducere

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

Exemplu de secvență de reducere

$$\begin{aligned} & ((\lambda x. \lambda y. \lambda z. (x (y z)) \ z) y) \rightarrow_{\alpha} \\ & ((\lambda x. \lambda y. \lambda t. (x (y t)) \ z) y) \end{aligned}$$

Exemplu de secvență de reducere

$((\lambda x. \lambda y. \lambda z. (x (y z)) \ z) y) \rightarrow_{\alpha}$

$((\lambda x. \lambda y. \lambda t. (x (y t)) \ z) y) \rightarrow_{\beta}$

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

Exemplu de secvență de reducere

$((\lambda x. \lambda y. \lambda z. (x (y z)) \ z) y) \rightarrow_{\alpha}$

$((\lambda x. \lambda y. \lambda t. (x (y t)) \ z) y) \rightarrow_{\beta}$

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

Exemplu de secvență de reducere

$((\lambda x. \lambda y. \lambda z. (x (y z)) \ z) y) \rightarrow_{\alpha}$

$((\lambda x. \lambda y. \lambda t. (x (y t)) \ z) y) \rightarrow_{\beta}$

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

Exemplu de secvență de reducere

$((\lambda x. \lambda y. \lambda z. (x (y z)) \ z) y) \rightarrow_{\alpha}$

$((\lambda x. \lambda y. \lambda t. (x (y t)) \ z) y) \rightarrow_{\beta}$

$(\lambda y. \lambda t. (z (y t)) \ y) \rightarrow_{\beta}$

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

Calcul Lambda – Cuprins

- Aparițiile unei variabile într-o λ -expresie
- Statutul variabilelor într-o λ -expresie
- Evaluarea unei λ -expresii
- Forma normală a unei λ -expresii

Forma normală a unei λ -expresii

λ -expresie în **forma normală** $\Leftrightarrow \lambda$ -expresie care nu conține niciun β -redex

Întrebări

- Are orice λ -expresie o formă normală?
- Forma normală este unică? (sau sevențe distincte de reducere pot duce la forme normale distincte?)
- Dacă o λ -expresie admite o formă normală, se poate garanta găsirea ei?

Observație (ajutătoare pentru întrebările anterioare)

Calculul Lambda este un model de calculabilitate.

λ -expresiile sunt practic programe capabile să ruleze pe o ipotetică Mașină Lambda.

Are orice λ -expresie o formă normală?

NU.

Exemplu: $(\lambda x.(x\ x)\ \lambda x.(x\ x)) \rightarrow_{\beta} (\lambda x.(x\ x)\ \lambda x.(x\ x)) \rightarrow_{\beta} (\lambda x.(x\ x)\ \lambda x.(x\ x)) \rightarrow_{\beta} \dots$

λ -expresie **reductibilă** \Leftrightarrow admite o secvență finită de reducere până la o formă normală

Altfel, λ -expresia este **ireductibilă**.

Forma normală este unică?

DA.

Teorema Church-Rosser

Dacă $\begin{cases} e \rightarrow^* a \\ \text{și} \\ e \rightarrow^* b \end{cases}$ atunci $\exists d \text{ a.î. } \begin{cases} a \rightarrow^* d \\ \text{și} \\ b \rightarrow^* d \end{cases}$ ($\rightarrow^* = \text{secvență de reducere}$)

Explicație: Dacă a și b ar fi forme normale distincte, prin definiție a și b nemaicontinând niciun β -redex, ele nu s-ar putea reduce suplimentar către un același d.

Se poate garanta găsirea formei normale?

DA.

Teorema normalizării

Pentru orice λ -expresie reductibilă, se poate ajunge la forma ei normală aplicând **reducere stânga->dreapta** (reducând mereu cel mai din stânga β -redex, ca la evaluarea normală).

Exemplu: $E_1 = (\lambda x.(x \ x) \ \lambda x.(x \ x))$

$$E_2 = (\lambda x.y \ E_1) \rightarrow_{\beta} y \quad \leftarrow \text{o reducere dreapta->stânga nu s-ar termina niciodată}$$

Concluzie: Evaluarea aplicativă e mai eficientă, dar evaluarea normală e mai sigură.

Rezumat

Teza lui Church

Tipuri de recursitate

Apariții ale unei variabile într-o expresie

Variabile într-o expresie

β -redex și β -reducere

α -conversie

Forma normală

Expresie ireductibilă

Teorema normalizării

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)

Tipuri de recursitate

Apariții ale unei variabile într-o expresie

Variabile într-o expresie

β -redex și β -reducere

α -conversie

Forma normală

Expresie ireductibilă

Teorema normalizării

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)

Tipuri de recursitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)

Apariții ale unei variabile într-o expresie

Variabile într-o expresie

β -redex și β -reducere

α -conversie

Forma normală

Expresie ireductibilă

Teorema normalizării

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)

Tipuri de recursitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)

Apariții ale unei variabile într-o expresie: legate ($\lambda \underline{x}.e$, $\lambda x. \dots \underline{x} \dots$), libere (restul)

Variabile într-o expresie

β -redex și β -reducere

α -conversie

Forma normală

Expresie ireductibilă

Teorema normalizării

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)

Tipuri de recursitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)

Apariții ale unei variabile într-o expresie: legate ($\lambda \underline{x}.e$, $\lambda x. \dots \underline{x} \dots$), libere (restul)

Variabile într-o expresie: legate (toate aparițiile sunt legate), libere (restul)

β -redex și β -reducere

α -conversie

Forma normală

Expresie ireductibilă

Teorema normalizării

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)

Tipuri de recursitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)

Apariții ale unei variabile într-o expresie: legate ($\lambda \underline{x}.e$, $\lambda x. \dots \underline{x} \dots$), libere (restul)

Variabile într-o expresie: legate (toate aparițiile sunt legate), libere (restul)

β -redex și β -reducere: $(\lambda x.e_1 e_2), (\lambda x.e_1 e_2) \rightarrow_{\beta} e_1[e_2/x]$

α -conversie

Forma normală

Expresie ireductibilă

Teorema normalizării

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)

Tipuri de recursitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)

Apariții ale unei variabile într-o expresie: legate ($\lambda \underline{x}.e$, $\lambda x. \dots \underline{x} \dots$), libere (restul)

Variabile într-o expresie: legate (toate aparițiile sunt legate), libere (restul)

β -redex și β -reducere: $(\lambda x.e_1 e_2)$, $(\lambda x.e_1 e_2) \rightarrow_{\beta} e_1[e_2/x]$

α -conversie: $(\lambda x. \dots \underline{\lambda y.e_1} \dots e_2) \rightarrow_{\alpha} (\lambda x. \dots \underline{\lambda t.e_1[t/y]} \dots e_2)$ unde t nu era liberă în e_1, e_2

Forma normală

Expresie ireductibilă

Teorema normalizării

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)

Tipuri de recursitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)

Apariții ale unei variabile într-o expresie: legate ($\lambda \underline{x}.e$, $\lambda x. \dots \underline{x} \dots$), libere (restul)

Variabile într-o expresie: legate (toate aparițiile sunt legate), libere (restul)

β -redex și β -reducere: $(\lambda x.e_1 e_2)$, $(\lambda x.e_1 e_2) \rightarrow_{\beta} e_1[e_2/x]$

α -conversie: $(\lambda x. \dots \underline{\lambda y.e_1} \dots e_2) \rightarrow_{\alpha} (\lambda x. \dots \underline{\lambda t.e_1[t/y]} \dots e_2)$ unde t nu era liberă în e_1, e_2

Forma normală: λ -expresia nu conține niciun β -redex

Expresie ireductibilă

Teorema normalizării

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)

Tipuri de recursitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)

Apariții ale unei variabile într-o expresie: legate ($\lambda \underline{x}.e$, $\lambda x. \dots \underline{x} \dots$), libere (restul)

Variabile într-o expresie: legate (toate aparițiile sunt legate), libere (restul)

β -redex și β -reducere: $(\lambda x.e_1 e_2)$, $(\lambda x.e_1 e_2) \rightarrow_{\beta} e_1[e_2/x]$

α -conversie: $(\lambda x. \dots \underline{\lambda y.e_1} \dots e_2) \rightarrow_{\alpha} (\lambda x. \dots \underline{\lambda t.e_1[t/y]} \dots e_2)$ unde t nu era liberă în e_1, e_2

Forma normală: λ -expresia nu conține niciun β -redex

Expresie ireductibilă: nu poate fi redusă la o formă normală (altfel – reductibilă)

Teorema normalizării

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)

Tipuri de recursitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)

Apariții ale unei variabile într-o expresie: legate ($\lambda \underline{x}.e$, $\lambda x. \dots \underline{x} \dots$), libere (restul)

Variabile într-o expresie: legate (toate aparițiile sunt legate), libere (restul)

β -redex și β -reducere: $(\lambda x.e_1 e_2)$, $(\lambda x.e_1 e_2) \rightarrow_{\beta} e_1[e_2/x]$

α -conversie: $(\lambda x. \dots \underline{\lambda y.e_1} \dots e_2) \rightarrow_{\alpha} (\lambda x. \dots \underline{\lambda z.e_1}_{[z/y]} \dots e_2)$ unde z nu era liberă în e_1, e_2

Forma normală: λ -expresia nu conține niciun β -redex

Expresie ireductibilă: nu poate fi redusă la o formă normală (altfel – reductibilă)

Teorema normalizării: Reducerea stânga->dreapta garantează găsirea formei normale (când aceasta există)

PARADIGME DE PROGRAMARE

Curs 3

Funcții ca valori de ordinul întâi. Funcționale. Abstractizare.

Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- Funcții ca valori ale unor variabile / membri ai unor structuri
- Funcții ca valori de return (funcții curry)
- Funcții ca argumente pentru alte funcții
- Abstractizare
- Abstractizarea la nivel de proces (funcționale)
- Abstractizarea la nivel de date

Calcul Lambda

Istoric

- Inventat de Alonzo Church în 1932 ca un formalism matematic menit să descrie comportamentul computațional al funcțiilor (completând definiția matematică a funcțiilor ca mulțimi de perechi argument-valoare)
- Nu a reușit să înlocuiască teoria mulțimilor ca fundament al matematicii, însă s-a dovedit un model de calculabilitate echivalent cu modelul propus de Turing (Mașina Turing)
- Definiția lui Turing a avut un impact mai mare începând cu momentul în care a propus un model de mașină pe care să se execute algoritmi
- Privit ca **primul limbaj funcțional**, pe care se bazează toate celelalte

Aspecte remarcabile

- **Simplitate:** orice valoare se poate modela cu doar 3 constructori
- **Generalitate:** funcțiile se pot aplica pe ele însăși (imposibil în teoria mulțimilor, ideea de mulțime care se conține pe ea însăși ducând la paradoxuri)

Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- Funcții ca valori ale unor variabile / membri ai unor structuri
- Funcții ca valori de return (funcții curry)
- Funcții ca argumente pentru alte funcții
- Abstractizare
- Abstractizarea la nivel de proces (funcționale)
- Abstractizarea la nivel de date

Valori de ordinul întâi

Date de ordinul întâi – pot fi:

- Valori ale unor variabile
- Membri în structuri compuse
- Trimise ca argumente unor funcții
- Valori returnate de o funcție

Exemple

Lucruri
Substantive
Date



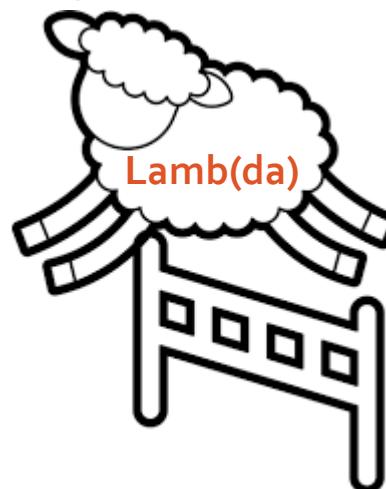
Nu neapărat de ordinul întâi

Acțiuni
Verbe
Funcții

Valori de ordinul întâi

Date de ordinul întâi – în programarea funcțională, avem **doar valori de ordinul întâi!**

- Valori ale unor variabile
- Membri în structuri compuse
- Trimise ca argumente unor funcții
- Valori returnate de o funcție



Exemple

- Lucruri
- Substantive
- Date

Nu neapărat de ordinul întâi

- Acțiuni
- Verbe
- Funcții

Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- **Funcții ca valori ale unor variabile / membri ai unor structuri**
- Funcții ca valori de return (funcții curry)
- Funcții ca argumente pentru alte funcții
- Abstractizare
- Abstractizarea la nivel de proces (funcționale)
- Abstractizarea la nivel de date

Funcții ca valori evaluate la ele însăși

1. `(define a +)` ← identifierul a se leagă la valoarea funcției +
2. `(a 2 3 5)`
- 3.
4. `null?` ← funcția null? se evaluatează la ea însăși
- 5.
6. `(define fs (list + (λ (x y) (- x y y)) 5))`
7. `fs` ← al doilea element al listei fs este o funcție anonimă
- 8.
9. `((cadr fs) 20 8)`

Funcții ca valori evaluate la ele însese

```
1. (define a +)  
2. (a 2 3 5) ; ; 10  
3.  
4. null? ; ; #<procedure:null?>  
5.  
6. (define fs (list + (λ (x y) (- x y y)) 5))  
7. fs ; ; ' (#<procedure:+> #<procedure> 5)  
8.  
9. ((cadr fs) 20 8) ; ; 4
```

Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- Funcții ca valori ale unor variabile / membri ai unor structuri
- **Funcții ca valori de retur (funcții curry)**
- Funcții ca argumente pentru alte funcții
- Abstractizare
- Abstractizarea la nivel de proces (funcționale)
- Abstractizarea la nivel de date

Funcții ca valori de return

1. `(define (f x)`
2. `(if (< x 100) + -))`
3. `) (f 25)`
4. `((f 120) 16 4)`
- 5.
6. `(define (g x)`
7. `(λ (y)`
8. `(cons x y))))`
9. `) (g 2)`
10. `((g 2) ' (5 2)))`

← funcția f (aplicată pe un argument) returnează o funcție de bibliotecă (+ sau -)

← funcția g (aplicată pe un argument) returnează o funcție anonimă

Funcții ca valori de return

```
1.  (define (f x)
2.    (if (< x 100) + -))
3.  (f 25)                                ; ; #<procedure:+>
4.  ((f 120) 16 4)                        ; ; 12
5.
6.  (define (g x)
7.    (λ (y)
8.      (cons x y)))
9.  (g 2)                                ; ; #<procedure>
10. ((g 2) ' (5 2))                      ; ; ' (2 5 2)
```

Funcții curry / uncurry

În Calculul Lambda există doar funcții **unare, anonime** – suficiente pentru a simula orice funcție n-ară

- O funcție binară se poate simula printr-o funcție unară care întoarce o altă funcție unară
- O funcție ternară se poate simula printr-o funcție unară care întoarce o funcție ca mai sus
- ... etc.

Funcția curry

- Își primește argumentele pe rând
- Poate fi aplicată parțial (doar pe o parte din argumente) caz în care întoarce o nouă funcție

Funcția uncurry

- Își primește obligatoriu **toate argumentele deodată**
- Nu poate fi aplicată parțial (o asemenea încercare rezultă într-o eroare)

Funcții curry / uncurry - Exemple

1. `(define (plus-curried x)`

2. `(λ (y) ←`

3. `(+ x y)))`

4.

5. `(plus-curried 2)`

6. `((plus-curried 2) 10)`

7.

8. `(define inc (plus-curried 1))`

9. `(inc 10)`

pe rând

`(define (plus-uncurry x y) ←`

`(+ x y))`

deodată

`(plus-uncurry 2 3)`

`(plus-uncurry 2)`

Funcții curry / uncurry - Exemple

```
1. (define (plus-curried x)
   (lambda (y)
     (+ x y)))
4.
5. (plus-curried 2) ;; #<procedure>
6. ((plus-curried 2) 10) ;; 12
7.
8. (define inc (plus-curried 1))
9. (inc 10) ;; 11
```

```
(define (plus-uncurry x y)
  (+ x y))
```

```
(plus-uncurry 2 3) ;; 5
```

```
(plus-uncurry 2)
  ;; plus-uncurry: arity mismatch;
```

Funcții curry / uncurry

Conduc la **reutilizare de cod**:

- Permit derivarea facilă de funcții din alte funcții (ex: inc din plus-curry)
- Aceste derivări pot avea loc ad-hoc, acolo unde este nevoie de ele
- **Exemplu** rezolvat la calculator: sortarea prin inserție, folosind un comparator oarecare

Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- Funcții ca valori ale unor variabile / membri ai unor structuri
- Funcții ca valori de return (funcții curry)
- **Funcții ca argumente pentru alte funcții**
- Abstractizare
- Abstractizarea la nivel de proces (funcționale)
- Abstractizarea la nivel de date

Funcții ca argumente pentru alte funcții

Exemple la calculator:

- Sortarea prin inserție cu comparator oarecare
- O funcție care aplică o transformare oarecare tuturor numerelor pare dintr-o listă
- Similar pentru numere impare, observând un şablon comun și **abstractizând** funcția astfel încât să se poată ușor folosi atât pentru numere pare cât și pentru numere impare

Observație

- Abstractizarea funcției de mai sus înseamnă o **generalizare**: de la posibilitatea de a transforma numere pare sau impare am generalizat la posibilitatea de a transforma elemente care satisfac o anumită condiție (conceptual, ne-am ridicat la un **nivel mai înalt**, mulțumită observării şablonului comun)

Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- Funcții ca valori ale unor variabile / membri ai unor structuri
- Funcții ca valori de return (funcții curry)
- Funcții ca argumente pentru alte funcții
- Abstractizare
- Abstractizarea la nivel de proces (funcționale)
- Abstractizarea la nivel de date

Programele sunt scrise pentru a fi înțelese

Scop

- Gestiunea complexității intelectuale a programelor (care trebuie întreținute și dezvoltate de oameni, nu doar executate pe niște mașini)

Mijloace

- **Primitive**
 - datele și funcțiile oferite de limbaj
- **Mijloace de combinare**
 - cum se pot pune primitivele împreună și construi obiecte mai complexe din ele
- **Mijloace de abstractizare**
 - cum se pot folosi combinațiile de elemente primitive ca și când ele însele ar fi primitive

Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- Funcții ca valori ale unor variabile / membri ai unor structuri
- Funcții ca valori de return (funcții curry)
- Funcții ca argumente pentru alte funcții
- Abstractizare
- Abstractizarea la nivel de proces (funcționale)
- Abstractizarea la nivel de date

Abstracțiuni procedurale

- Problemele se descompun în mod natural în **subprobleme**
- Fiecare din aceste subprobleme poate fi rezolvată de o **funcție separată**, care este ca o **cutie neagră** care îndeplinește o sarcină
 - din punct de vedere al problemei principale **nu interesează implementarea** funcției ajutătoare (care se poate realiza în diverse moduri)
 - orice funcție capabilă să îndeplinească sarcina este la fel de bună
 - funcția principală se folosește de o **abstracțiune procedurală** mai degrabă decât de o funcție ajutătoare concretă (rezolvarea este descrisă **conceptual** mai degrabă decât intrând în detalii de implementare)
 - spunem că programarea funcțională este de tip **wishful thinking**
 - descriem soluția în termeni conceptuali, „dorind” ca aceste concepte să existe în limbaj
 - oferim implementări pentru fiecare abstracțiune procedurală folosită de funcția principală

Abstracții procedurale - Exemplu

Exemplu: primul număr mai mare sau egal decât start care este palindrom în minim b baze dintre bazele 2, 3, 4, 5, 6, 7, 8, 9, 10.

```
1. (define (first-b-pal start b)
2.   (if (min-b-bases? start b '(2 3 4 5 6 7 8 9 10))
3.       start
4.       (first-b-pal (+ 1 start) b)))
5.
6. (define (min-b-bases? n b Bases)
7.   (cond ((zero? b) #t)
8.         ((null? Bases) #f)
9.         ((palindrome? (num->base n (car Bases))) (min-b-bases? n (- b 1) (cdr Bases)))
10.        (else (min-b-bases? n b (cdr Bases)))))
```

wishful thinking: îmi imaginez că min-b-bases? există
apoi o implementez (se putea implementa și altfel,
de exemplu numărând toate bazele)

Abstracțiuni procedurale – Observații

- O abstracțiune este cu atât mai interesantă cu cât are mai **multe utilizări**
- **Nume sugestive** conduc la programe expresive (aproape programarea de modul în care oamenii gândesc)
- Pentru a folosi o funcție (un feature), utilizatorul trebuie să știe doar **ce face funcția, nu cum** este ea implementată
 - Nu știm cum sunt implementate funcțiile de bibliotecă, le folosim ca pe niște primitive utile (ex: cons, +, equal?)
 - Similar, un dezvoltator al programelor noastre ar trebui să poată folosi funcțiile scrise de noi ca pe niște primitive

Recunoașterea unui şablon comun

Exemple la calculator:

- Adunarea numerelor naturale de la a la b
- Aproximarea lui e conform seriei $e = 1/0! + 1/1! + 1/2! + 1/3! + \dots$
- Aproximarea lui $\pi^2/8$ conform seriei $\pi^2/8 = 1/1^2 + 1/3^2 + 1/5^2 + \dots$

Observații

- Funcțiile au foarte mult cod în comun
- Este util să identificăm un şablon mai abstract din care derivă toate cele 3 funcții
- Pașii de urmat: **Recunoaștere şablon** → **Definire** → **Reutilizare**

Recunoaștere → Definire → Reutilizare

Recunoașterea şablonului

- Reproducem porțiunile de cod comune
- Porțiunile care diferă devin variabile
- Ex: şablonul comun pentru `'(2 2)`, `'(3 3)`, `'(4 4)` este `(list x x)`

Definire

- Şablonului identificat la pasul anterior î se atribuie un nume sugestiv, care face programul uşor de înțeles

Reutilizare

- Cu cât şablonul se întâlneşte mai frecvent, cu atât este mai important ca el să fie abstractizat (şi multiplele sale instanţe să poată fi astfel derivate cu uşurinţă)

Funcționale (funcții de nivel înalt)

Funcționale (numite și **funcții de nivel înalt**)

- **Funcții care primesc ca argumente sau returnează funcții**
- **map, filter, foldl, foldr, apply** – funcționale predefinite în Racket, care abstractizează cele mai comune procese de calcul

Exemple la calculator:

- Maximul elementelor dintr-o listă de numere
- Extragerea inițialelor dintr-o listă de nume
- Extragerea pronumelor dintr-o listă de cuvinte
- Pătratele elementelor dintr-o listă de numere
- Numărul de elemente pare dintr-o listă
- Extragerea numerelor unei liste care sunt mai mici decât o valoare dată

Funcționale (funcții de nivel înalt)

Funcționale (numite și funcții de nivel înalt)

- **Funcții care primesc ca argumente sau returnează funcții**
- **map, filter, foldl, foldr, apply** – funcționale predefinite în Racket, care abstractizează cele mai comune procese de calcul pe liste

Exemple la calculator:

- Maximul elementelor dintr-o listă de numere
- Extragerea inițialelor dintr-o listă de nume
- Extragerea pronumelor dintr-o listă de cuvinte
- Pătratele elementelor dintr-o listă de numere
- Numărul de elemente pare dintr-o listă
- Extragerea numerelor unei liste care sunt mai mici decât o valoare dată

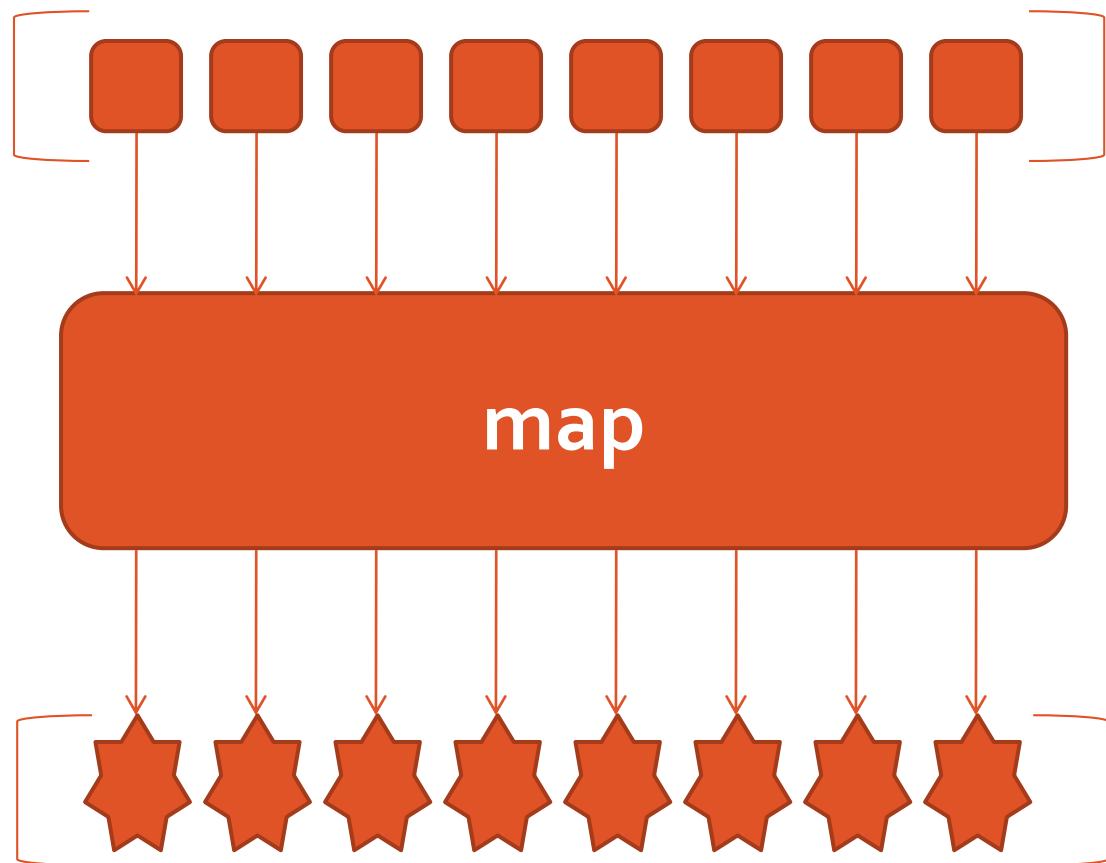
Funcționala map (map f L)

(map add1 (range 2 7))

(map sqr (range 2 7))

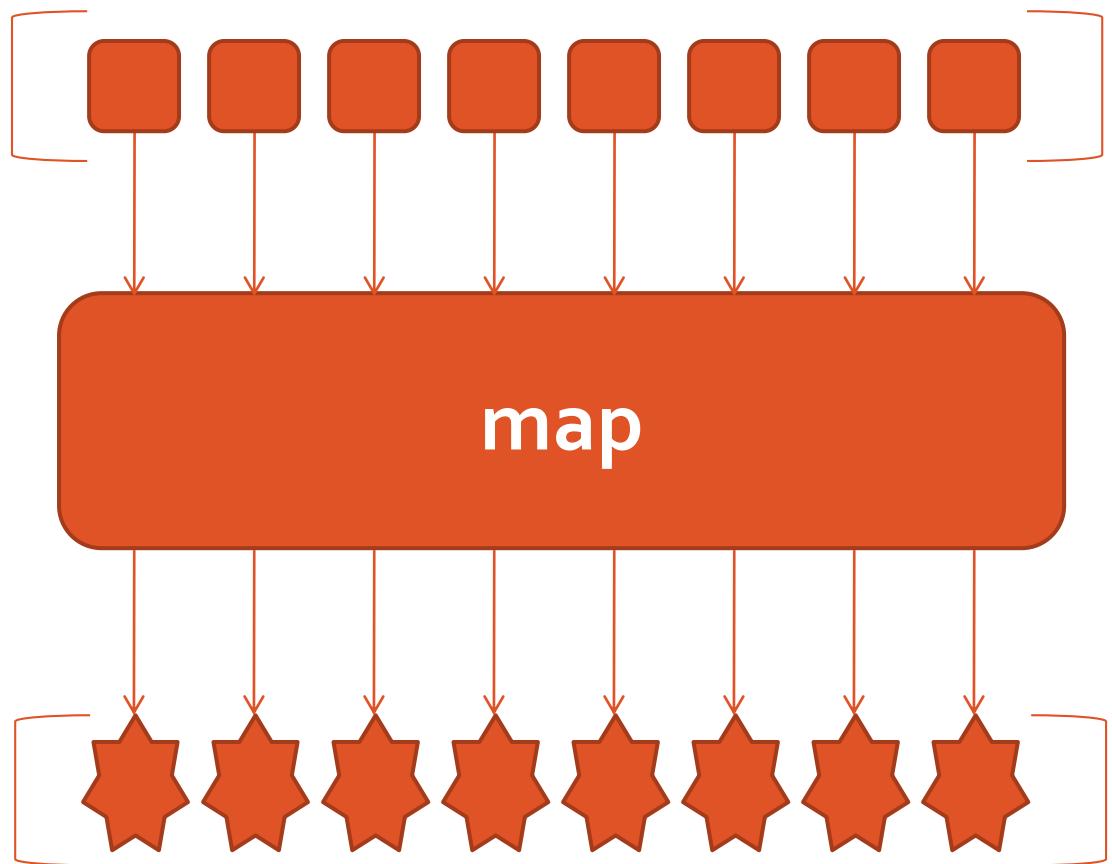
(map even? (range 2 7))

(map random (range 2 7))



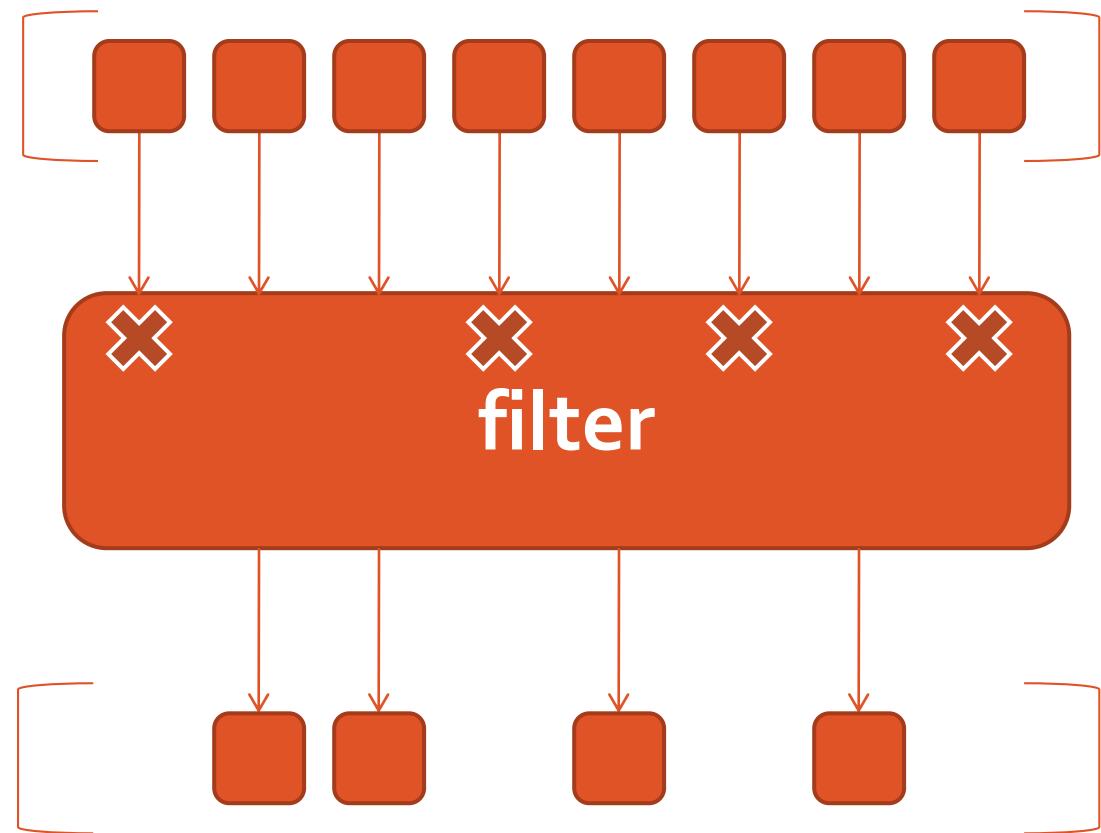
Funcționala map (map f L)

```
(map add1 (range 2 7))  
;; '(3 4 5 6 7)  
  
(map sqr (range 2 7))  
;; '(4 9 16 25 36)  
  
(map even? (range 2 7))  
;; '#t #f #t #f #t  
  
(map random (range 2 7))  
;; surpriză ☺
```



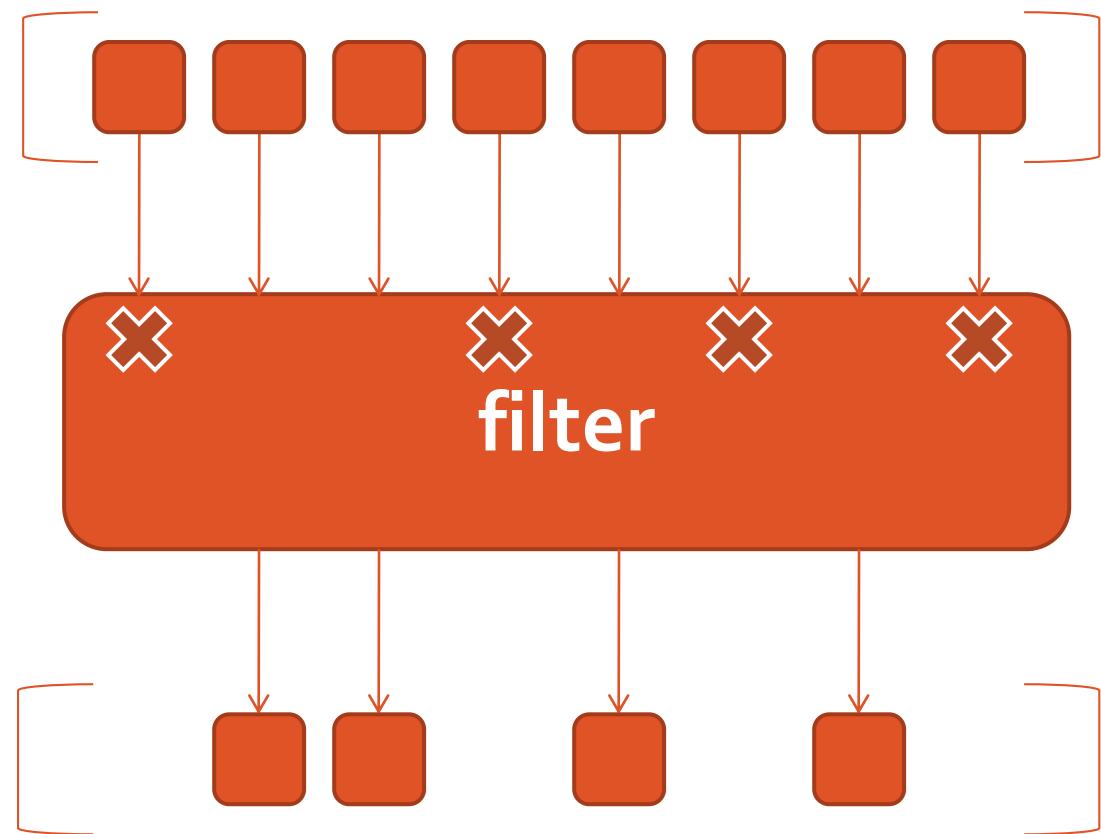
Funcționala filter (filter p L)

```
(filter even? (range 2 7))
```



Funcționala filter (filter p L)

```
(filter even? (range 2 7))  
;; '(2 4 6)
```

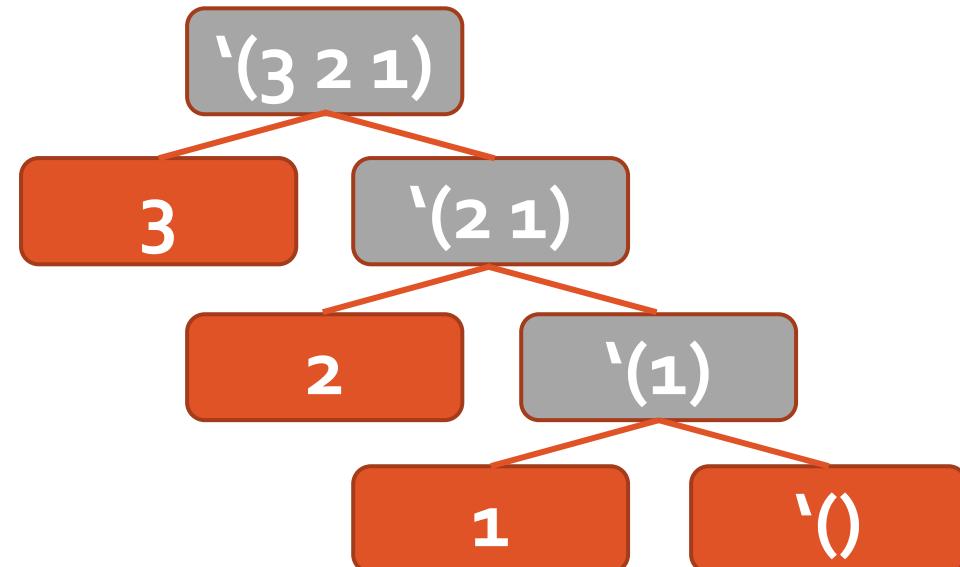


Funcționala foldl (fold left) (foldl f acc L)

- Folosește o **funcție binară element-acumulator**
- Elementele listei sunt prelucrate în ordinea **stânga→dreapta**
- Mai întâi se aplică funcția pe (first L) și accumulator, rezultând un nou accumulator
- Apoi pe (second L) și noul accumulator ... etc.

Exemplu

```
(foldl cons '() '(1 2 3))
```

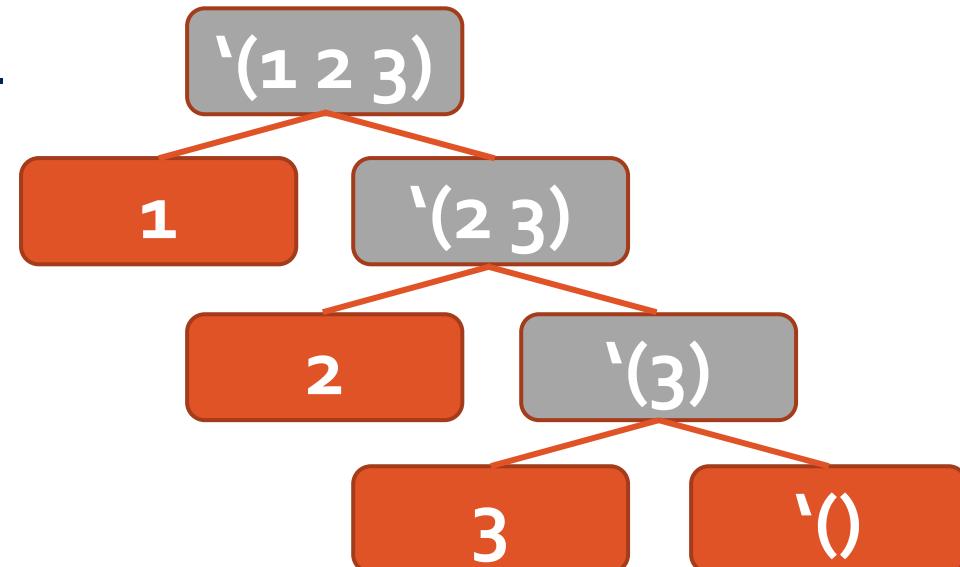


Funcționala foldr (fold right) (foldr f acc L)

- Folosește o **funcție binară element-acumulator**
- Elementele listei sunt prelucrate în ordinea **dreapta→stânga**
- Mai întâi se aplică funcția pe (last L) și accumulator, rezultând un nou accumulator
- Apoi pe penultimul și noul accumulator ... etc.

Exemplu

```
(foldr cons '() '(1 2 3))
```



Funcționala foldl / foldr

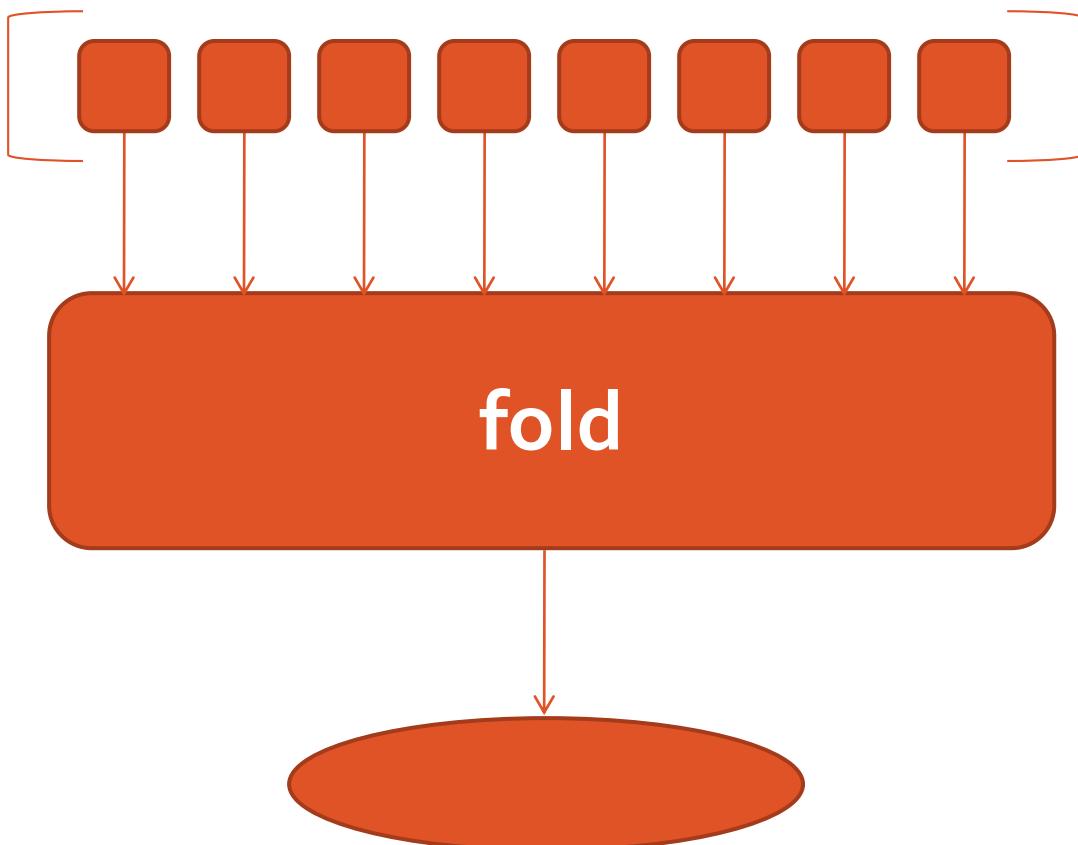
```
(foldl + 0 (range 2 7))
```

```
(foldr max 0 (range 2 7))
```

```
(foldl cons '() (range 2 7))
```

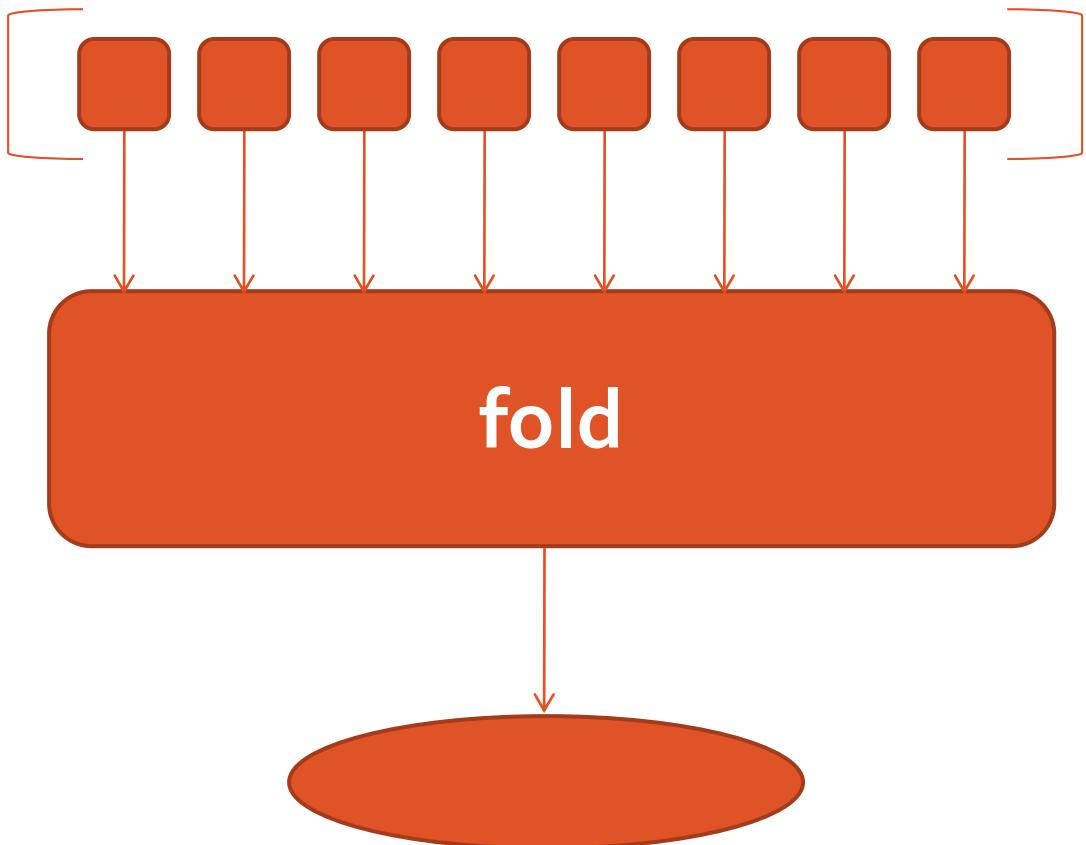
```
(foldr cons '() (range 2 7))
```

```
(foldl list 1 (range 2 7))
```



Funcționala foldl / foldr

```
(foldl + 0 (range 2 7))  
;; 20  
(foldr max 0 (range 2 7))  
;; 6  
(foldl cons '() (range 2 7))  
;; '(6 5 4 3 2)  
(foldr cons '() (range 2 7))  
;; '(2 3 4 5 6)  
(foldl list 1 (range 2 7))  
;; '(6 (5 (4 (3 (2 1)))))
```



Funcționala apply (apply f [...] L)

(apply f L) aplică funcția f pe argumentele din lista L

(apply f x y z L) aplică funcția f pe argumentele x y z și cele venite din lista L ... etc.

Exemple

(apply + (range 2 7))

(+ 2 3 4 5 6)

(apply list -1 0 '(1 2 3))

(list -1 0 1 2 3)

(apply map list '((1 2 3)))

(map list '(1 2 3))

Funcționala apply (apply f [...] L)

(apply f L) aplică funcția f pe argumentele din lista L

(apply f x y z L) aplică funcția f pe argumentele x y z și cele venite din lista L ... etc.

Exemple

```
(apply + (range 2 7)) ; ; 20
(apply list -1 0 '(1 2 3)) ; ; '(-1 0 1 2 3)
(apply map list '((1 2 3))) ; ; '((1) (2) (3))
```

Comparație

(map f L) – aplică o transformare f pe **fiecare element** din lista L

(filter p L) – păstrează **doar elementele** listei L care satisfac condiția (predicatul) p

(foldl/foldr f seed L) – cumulează aplicările funcției f pe **toate elementele** listei L

(apply f [...] L) – aplică funcția f pe argumentele conținute în lista L (optional există și alte argumente date unul câte unul înainte de cele care vor fi despachetate din L)

Observație

- Funcțiunalele de tip fold surprind procesul cel mai general
- **map** și **filter** pot fi implementate cu **fold**
- Pentru expresivitate, este de dorit să folosim
 - **map** când avem de transformat fiecare element (cuvânt cheie: **fiecare**)
 - **filter** când avem de selectat anumite elemente (cuvânt cheie: **selectie**)
- și nu **fold** peste tot.

Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- Funcții ca valori ale unor variabile / membri ai unor structuri
- Funcții ca valori de return (funcții curry)
- Funcții ca argumente pentru alte funcții
- Abstractizare
- Abstractizarea la nivel de proces (funcționale)
- Abstractizarea la nivel de date

Tipuri de date abstracte (TDA)

- Ca și procesele frecvente, tipurile de date frecvent utilizate ar trebui să fie abstractizate
- Un TDA se caracterizează prin **constructori și operatori**, care se comportă în felul așteptat de utilizator (care nu se mai preocupă de implementarea lor internă)
- Limbajul pune la dispoziție o serie de tipuri primitive (ex: numere, perechi, liste) pe care le manipulăm exclusiv prin constructori și operatori
- Tipurile primitive sunt **pietre de cărămidă pentru construcția de tipuri mai complexe**
- Tipurile mai complexe definite de programator sunt pietre de cărămidă pentru eventuale tipuri (concepte) și mai complexe ... etc.
- Astfel **controlăm complexitatea** intelectuală a unui program mare

Abstractizarea la nivel de date

- De fiecare dată când lucrăm cu date compuse (din „cărămizi” mai mici) este bine ca aceste date să fie abstracte
 - În sensul că utilizarea TDA-ului este complet separată (prin ceea ce numim **bariera de abstractizare**) de implementarea sa
 - Se definește o **interfață** (un set de constructori și operatori) astfel încât orice manipulare a valorilor TDA-ului să se poată exprima în termeni de acești constructori și operatori
 - Programele care manipulează aceste date vor funcționa identic în cazul în care implementarea constructorilor și operatorilor se modifică

Exemplu

- Numere complexe

Utilizare numere complexe

make-complex real imag add-c sub-c mul-c div-c abs-c

Implementare numere complexe (ca perechi, dar irelevant la nivel superior)

cons car cdr

Implementare perechi (nu o știm și nu ne interesează)

Exemplu – Tipul BST (Binary Search Tree)

Constructori

empty-bst : -> BST

make-bst : BST x Elem x BST -> BST

Operatori

left : BST -> BST

right : BST -> BST

key : BST -> Elem

bst-empty? : BST -> Bool

insert-bst : Elem x BST -> BST

list->bst : List -> BST

Rezumat

Date de ordinul întâi

Funcții curry / uncurry

Funcționale

Utilizare map

Utilizare filter

Utilizare fold

Utilizare apply

Abstractizare la nivel de proces

Abstractizare la nivel de date

Rezumat

Date de ordinul întâi: valori ale unor variabile, membri în structuri, argumente, valori de returnare

Functii curry / uncurry

Functionale

Utilizare map

Utilizare filter

Utilizare fold

Utilizare apply

Abstractizare la nivel de proces

Abstractizare la nivel de date

Rezumat

Date de ordinul întâi: valori ale unor variabile, membri în structuri, argumente, valori de returnare

Functii curry / uncurry: își primesc argumentele pe rând / își primesc argumentele deodată

Funcționale

Utilizare map

Utilizare filter

Utilizare fold

Utilizare apply

Abstractizare la nivel de proces

Abstractizare la nivel de date

Rezumat

Date de ordinul întâi: valori ale unor variabile, membri în structuri, argumente, valori de returnare

Functii curry / uncurry: își primesc argumentele pe rând / își primesc argumentele deodată

Funcționale: funcții care primesc ca argumente sau returnează funcții

Utilizare map

Utilizare filter

Utilizare fold

Utilizare apply

Abstractizare la nivel de proces

Abstractizare la nivel de date

Rezumat

Date de ordinul întâi: valori ale unor variabile, membri în structuri, argumente, valori de returnare

Functii curry / uncurry: își primesc argumentele pe rând / își primesc argumentele deodată

Funcționale: funcții care primesc ca argumente sau returnează funcții

Utilizare map: (map f L) aplică f pe fiecare element din L

Utilizare filter

Utilizare fold

Utilizare apply

Abstractizare la nivel de proces

Abstractizare la nivel de date

Rezumat

Date de ordinul întâi: valori ale unor variabile, membri în structuri, argumente, valori de returnare

Functii curry / uncurry: își primesc argumentele pe rând / își primesc argumentele deodată

Funcționale: funcții care primesc ca argumente sau returnează funcții

Utilizare map: (map f L) aplică f pe fiecare element din L

Utilizare filter: (filter p L) păstrează doar elementele din L care satisfac predicatul p

Utilizare fold

Utilizare apply

Abstractizare la nivel de proces

Abstractizare la nivel de date

Rezumat

Date de ordinul întâi: valori ale unor variabile, membri în structuri, argumente, valori de returnare

Functii curry / uncurry: își primesc argumentele pe rând / își primesc argumentele deodată

Funcționale: funcții care primesc ca argumente sau returnează funcții

Utilizare map: (map f L) aplică f pe fiecare element din L

Utilizare filter: (filter p L) păstrează doar elementele din L care satisfac predicatul p

Utilizare fold: (foldl/foldr f acc L) parcurge L de la stânga/dreapta, aplicând (f x acc), (f y acc')...

Utilizare apply

Abstractizare la nivel de proces

Abstractizare la nivel de date

Rezumat

Date de ordinul întâi: valori ale unor variabile, membri în structuri, argumente, valori de returnare

Functii curry / uncurry: își primesc argumentele pe rând / își primesc argumentele deodată

Funcționale: funcții care primesc ca argumente sau returnează funcții

Utilizare map: (map f L) aplică f pe fiecare element din L

Utilizare filter: (filter p L) păstrează doar elementele din L care satisfac predicatul p

Utilizare fold: (foldl/foldr f acc L) parurge L de la stânga/dreapta, aplicând (f x acc), (f y acc')...

Utilizare apply: (apply f [...] L) aplică f pe argumentele care urmează, despachetând lista L

Abstractizare la nivel de proces

Abstractizare la nivel de date

Rezumat

Date de ordinul întâi: valori ale unor variabile, membri în structuri, argumente, valori de returnare

Functii curry / uncurry: își primesc argumentele pe rând / își primesc argumentele deodată

Funcționale: funcții care primesc ca argumente sau returnează funcții

Utilizare map: (map f L) aplică f pe fiecare element din L

Utilizare filter: (filter p L) păstrează doar elementele din L care satisfac predicatul p

Utilizare fold: (foldl/foldr f acc L) parurge L de la stânga/dreapta, aplicând (f x acc), (f y acc')...

Utilizare apply: (apply f [...] L) aplică f pe argumentele care urmează, despachetând lista L

Abstractizare la nivel de proces: abstractiuni procedurale, abstractizare sabloane comune

Abstractizare la nivel de date

Rezumat

Date de ordinul întâi: valori ale unor variabile, membri în structuri, argumente, valori de returnare

Functii curry / uncurry: își primesc argumentele pe rând / își primesc argumentele deodată

Funcționale: funcții care primesc ca argumente sau returnează funcții

Utilizare map: (map f L) aplică f pe fiecare element din L

Utilizare filter: (filter p L) păstrează doar elementele din L care satisfac predicatul p

Utilizare fold: (foldl/foldr f acc L) parurge L de la stânga/dreapta, aplicând (f x acc), (f y acc')...

Utilizare apply: (apply f [...] L) aplică f pe argumentele care urmează, despachetând lista L

Abstractizare la nivel de proces: abstractiuni procedurale, abstractizare şabloane comune

Abstractizare la nivel de date: TDA-uri cu utilizare separată complet de implementare

PARADIGME DE PROGRAMARE

Curs 4

Transparentă referențială. Legare statică / dinamică. Modelul contextual de evaluare.

Transparentă referențială – Cuprins

- Efecte laterale
- Transparentă referențială

Efecte laterale

Efecte laterale ale unei funcții

- Efectul principal al oricărei funcții este să întoarcă o valoare
- Efecte laterale = alte efecte asupra stării programului (ex: modificarea unor variabile vizibile în afara funcției) sau asupra „lumii de afară” (ex: scrierea în fișier)

Funcție pură

- Aplicată pe aceleași argumente, întoarce mereu aceeași valoare
- Nu are efecte laterale

Exemplu (C++)

```
i = 7;
```

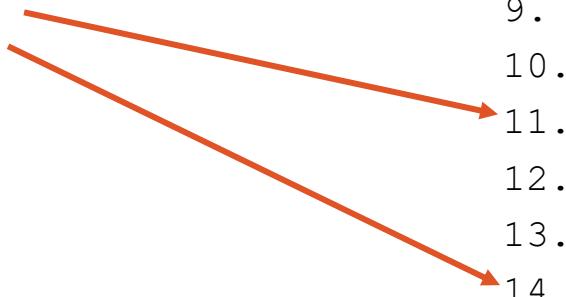
- Expresia întoarce valoarea 7
- Efect lateral: variabila i este setată la valoarea 7

Efecte laterale

Consecințe

- Conțează strategia de evaluare

?



```
1. int x = 0;
2. int f() {
3.     x = 15;
4.     return x;
5. }
6.
7. int main() {
8.     int i = 0;
9.     int a = i-- + ++i;
10.    int b = ++i + i--;
11.    cout << a << " " << b;
12.
13.    int res = x + f();
14.    cout << " " << res << "\n";
```

Efecte laterale

Consecințe

- Conțează strategia de evaluare

Comportament bizar al compilatorului
care nu are operația definită

Afișează 0 1 → Conțează ordinea de
evaluare a argumentelor adunării

Afișează 30 → Cu call by reference s-
ar afișa 30; Cu call by value și evaluare
stânga-dreapta s-ar afișa 15

```
1. int x = 0;
2. int f() {
3.     x = 15;
4.     return x;
5. }
6.
7. int main() {
8.     int i = 0;
9.     int a = i-- + ++i;
10.    int b = ++i + i--;
11.    cout << a << " " << b;
12.
13.    int res = x + f();
14.    cout << " " << res << "\n";
```

Efecte laterale

Consecințe

- Conțează **strategia de evaluare**
- Scade **nivelul de abstractizare**
 - Funcția add() nu mai e ca o cutie neagră → implementări diferite au efecte diferite

```
1. int a = 5, b = 7;
2. int add() {
3.     while (a > 0) {a--; b++;}
4.     return b;
5. }
6. //SAU
7. int add() {
8.     int s = b;
9.     while (a > 0) {a--; s++;}
10.    return s;
11. }
12.
13. int main() {
14.     cout << add() << " " << b;
```

Efecte laterale

Consecințe

- Conținează **strategia de evaluare**
- Scade **nivelul de abstractizare**
 - Funcția add() nu mai e ca o cutie neagră → implementări diferite au efecte diferite

Afișează 12 12 → Funcția are efectul lateral al modificării lui b

Afișează 12 7 → Această implementare nu alterează valoarea lui b

```
1. int a = 5, b = 7;
2. int add() {
3.     while (a > 0) {a--; b++;}
4.     return b;
5. }
6. //SAU
7. int add() {
8.     int s = b;
9.     while (a > 0) {a--; s++;}
10.    return s;
11. }
12.
13.
14. int main() {
15.     cout << add(); cout << b;
```

Efecte laterale

Consecințe

- Conținează **strategia de evaluare**
- Scade **nivelul de abstractizare**



- Risc ridicat de **bug-uri**

Transparentă referențială – Cuprins

- Efecte laterale
- Transparentă referențială

Transparentă referențială

Transparentă referențială

- Există atunci când toate funcțiile/expreziile sunt pure
- O expresie poate fi înlocuită prin valoarea sa (fără să se piardă nimic)

Exemple

- Toate funcțiile Racket implementate de noi până acum
- (random)
- (`define` counter 1)
`(define (display-and-inc-counter)`
 `(display` counter)
 `(set!` counter (add1 counter)))

Transparentă referențială

Transparentă referențială

- Există atunci când toate funcțiile/expreziile sunt pure
- O expresie poate fi înlocuită prin valoarea sa (fără să se piardă nimic)

Exemple

- Toate funcțiile Racket implementate de noi până acum
- (random)
- (**define** counter 1)
(define (display-and-inc-counter)
 (**display** counter)
 (**set!** counter (**add1** counter)))

transparente referențial
opacă referențial
opacă referențial

Transparentă referențială - avantaje

- Programe **elegante** și ușor de **analizat formal**
- Nu contează ordinea de evaluare a expresiilor (se vor evalua oricând la același lucru)
→ compilatorul poate **optimiza** codul prin **reordonarea evaluărilor** și prin **caching**
- Funcțiile nu își afectează execuția una alteia → **paralelizare** ușoară
- Rezultatul expresiilor deja evaluate se poate prelua dintr-un cache (întrucât o nouă evaluare nu va produce altceva) → **call by need**

Legare statică / dinamică – Cuprins

- Variabile
- Domeniu de vizibilitate
- Tipuri de legare a variabilelor
- Expresii pentru legare statică
- Expresii pentru legare dinamică

Variabile

Variabilă = pereche identificator-valoare

Caracteristici

- Domeniu de vizibilitate (zona de program în care valoarea poate fi accesată prin identificator)
- Durată de viață

Observație

- În Racket, tipul este asociat valorilor, nu variabilelor

Legare statică / dinamică – Cuprins

- Variabile
- Domeniu de vizibilitate
- Tipuri de legare a variabilelor
- Expresii pentru legare statică
- Expresii pentru legare dinamică

Domeniu de vizibilitate al unei variabile

Domeniu de vizibilitate = multimea punctelor din program unde asocierea identificator – valoare este vizibila (și valoarea se poate accesa prin identificator)

Exemple în Calcul Lambda

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

$(\textcolor{red}{x} \ \lambda x.(\lambda x.y \ \lambda y.(x z)))$

$(x \ \lambda \textcolor{red}{x}.(\lambda x.y \ \lambda y.(x z)))$

$(x \ \lambda x.(\lambda \textcolor{red}{x}.y \ \lambda y.(x z)))$

Domeniu de vizibilitate al unei variabile

Domeniu de vizibilitate = multimea punctelor din program unde asocierea identificator – valoare este vizibila (și valoarea se poate accesa prin identificator)

Exemple în Calcul Lambda

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

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

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

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

Domeniu de vizibilitate al unei variabile

Domeniu de vizibilitate = multimea punctelor din program unde asocierea identificator – valoare este vizibila (și valoarea se poate accesa prin identificator)

Exemple în Calcul Lambda

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

$(\textcolor{red}{x} \ \lambda x.(\lambda x.y \ \lambda y.(x z)))$ – acest x nu mai este vizibil nicăieri în această zonă de program

$(x \ \lambda \textcolor{red}{x}.(\lambda x.y \ \lambda y.(x z)))$

$(x \ \lambda x.(\lambda \textcolor{red}{x}.y \ \lambda y.(x z)))$

Domeniu de vizibilitate al unei variabile

Domeniu de vizibilitate = multimea punctelor din program unde asocierea identificator – valoare este vizibilă (și valoarea se poate accesa prin identificator)

Exemple în Calcul Lambda

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

$(\textcolor{red}{x} \ \lambda x.(\lambda x.y \ \lambda y.(x z)))$

$(x \ \lambda \textcolor{red}{x}.(\lambda x.y \ \lambda y.(\textcolor{red}{x} z)))$ – practic: zona din corp în care aparițiile lui x sunt libere

$(x \ \lambda x.(\lambda \textcolor{red}{x}.y \ \lambda y.(x z)))$

Domeniu de vizibilitate al unei variabile

Domeniu de vizibilitate = multimea punctelor din program unde asocierea identificator – valoare este vizibila (și valoarea se poate accesa prin identificator)

Exemple în Calcul Lambda

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

$(\textcolor{red}{x} \ \lambda x.(\lambda x.y \ \lambda y.(x z)))$

$(x \ \lambda \textcolor{red}{x}.(\lambda x.y \ \lambda y.(x z)))$

$(x \ \lambda x.(\lambda \textcolor{red}{x}.y \ \lambda y.(x z)))$

Domeniu de vizibilitate al unei variabile

Exemplu în Racket

```
1.  (define a 1)
2.  (define n 5)
3.  (define (fact n)
4.    (if (< n 2)
5.      a
6.      (* n (fact (- n 1))))))
7.  (fact n)
```

Domeniu de vizibilitate al unei variabile

Exemplu în Racket

```
1. (define a 1)
2. (define n 5)
3. (define (fact n)
4.   (if (< n 2)
5.     a
6.     (* n (fact (- n 1))))) )
7. (fact n)
```

Domeniu de vizibilitate al unei variabile

Exemplu în Racket

```
1. (define a 1)
2. (define n 5)
3. (define (fact n)
4.   (if (< n 2)
5.     a
6.     (* n (fact (- n 1))))))
7. (fact n)
```

În corpul funcției fact este vizibilă legarea parametrului n la valoarea pe care este aplicată funcția. Legarea interioară **obscurăză** legarea de la (define n 5).

Domeniu de vizibilitate al unei variabile

Exemplu în Racket

```
1. (define a 1)
2. (define n 5)
3. (define (fact n)
4.   (if (< n 2)
5.     a
6.     (* n (fact (- n 1))))) )
7. (fact n)
```

The diagram illustrates the visibility domain of the variable `n`. Red arrows point from the occurrences of `n` in the code to its definition at line 3. Specifically, the arrows point from the `n` in the condition of the `if` statement, the `n` in the `fact` call, and the `n` in the recursive call to the `n` in the `(fact n)` definition.

Legare statică / dinamică – Cuprins

- Variabile
- Domeniu de vizibilitate
- Tipuri de legare a variabilelor
- Expresii pentru legare statică
- Expresii pentru legare dinamică

Tipuri de legare a variabilelor

Legare = asocierea identificatorului cu valoarea

- Se poate realiza la **define**, la **aplicarea unei funcții** pe argumente, la **let** (vom vedea)
- Felul în care se realizează determină domeniul de vizibilitate al variabilei respective

Legare statică (lexicală)

Domeniul de vizibilitate este

- Controlat textual, prin construcții specifice limbajului (**lambda, let, etc.**)
- Determinat la compilare (**static**)

Legare dinamică

Domeniul de vizibilitate este

- Controlat de timp (se folosește cea mai recentă declarație a variabilei – cel mai recent **define**)
- Determinat la execuție (**dinamic**)

Tipuri de legare a variabilelor

Observații

- Calculul Lambda are doar legare statică
- Racket are legare statică, mai puțin pentru variabilele top-level (definite cu define)

Exemplu de legare dinamică în Racket

```
1. (define (f)
2.   (g 5))
3.
4. (define (g x) 
5.   (* x x))
6. (f)
7.
8. (define (g x) 
9.   (* x x x))
10. (f)
```

Redefinirea nu este posibilă în lang racket,
pentru exemplele de legare dinamică este
necesar să schimbăm limbajul în Pretty Big.

Tipuri de legare a variabilelor

Observații

- Calculul Lambda are doar legare statică
- Racket are legare statică, mai puțin pentru variabilele top-level (definite cu define)

Exemplu de legare dinamică în Racket

```
1. (define (f)
2.   (g 5))
3.
4. (define (g x)
5.   (* x x))
6. (f)           ; ; 25    ← Când se intră în corpul lui f se evaluatează g la definiția cea mai recentă
7.
8. (define (g x)
9.   (* x x x))
10. (f)          ; ; 125   ← Când se intră în corpul lui f se evaluatează g la definiția cea mai recentă
```

Legare statică / dinamică – Cuprins

- Variabile
- Domeniu de vizibilitate
- Tipuri de legare a variabilelor
- Expresii pentru legare statică
- Expresii pentru legare dinamică

Expresii pentru legare statică

- lambda
- let
- let*
- letrec
- named let

exemplificate la calculator

Construcția lambda

```
( (lambda (var1 var2 ... varm)
           expr1
           expr2
           ...
           exprn) arg1 arg2 ... argm)
```

La aplicarea λ-expresiei pe argumente

- Se evaluatează **în ordine aleatoare** argumentele $\text{arg}_1, \text{arg}_2, \dots, \text{arg}_m$ (evaluare aplicativă)
- Se realizează legările $\text{var}_k \leftarrow \text{valoare}(\text{arg}_k)$
- Domeniul de vizibilitate al variabilei var_k este **corpul** lui lambda (exceptând aparițiile legate ale lui var_k în corp)
- Se evaluatează **în ordine** expresiile din **corpul** funcției ($\text{expr}_1, \text{expr}_2, \dots, \text{expr}_n$)
- Rezultatul este **valoarea** lui expr_n (valorile lui $\text{expr}_1, \text{expr}_2, \dots, \text{expr}_{n-1}$ se pierd)

Construcția let

```
(let ((var1 e1) (var2 e2) ... (varm em))  
    expr1  
    expr2  
    ...  
    exprn)
```

La evaluare

- Se evaluatează exact ca

```
((lambda (var1 var2 ... varm) expr1 expr2 ... exprn) e1 e2 ... em)
```

- Domeniul de vizibilitate al variabilei var_k este **corful** lui let (exceptând aparițiile legate ale lui var_k în corp)
- Rezultatul este valoarea lui expr_n (valorile lui expr₁, expr₂, ... expr_{n-1} se pierd)

Construcția let*

```
(let* ((var1 e1) (var2 e2) (var3 e3) ... (varm em))  
    expr1  
    expr2  
    ...  
    exprn)
```

La evaluare

- Se realizează **în ordine (stânga→dreapta)** legările $\text{var}_k \leftarrow \text{valoare}(e_k)$
- Domeniul de vizibilitate al variabilei var_k este **restul textual** (restul legărilor + corp) al lui let* (exceptând aparițiile legate ale lui var_k în acest rest)
- Se evaluatează **în ordine expresiile din corpul funcției** ($\text{expr}_1, \text{expr}_2, \dots, \text{expr}_n$)
- Rezultatul este valoarea lui expr_n (valorile lui $\text{expr}_1, \text{expr}_2, \dots, \text{expr}_{n-1}$ se pierd)

Construcția letrec

```
(letrec ((var1 e1) (var2 e2) (var3 e3) ... (varm em))  
  expr1  
  expr2  
  ...  
  exprn)
```

La evaluare

- Se realizează **în ordine (stânga→dreapta)** legările $\text{var}_k \leftarrow \text{valoare}(e_k)$
- Domeniul de vizibilitate al variabilei var_k este **întregul letrec** (celealte legări + corp) (exceptând aparițiile legate ale lui var_k în această zonă), dar **variabila trebuie să fi fost deja definită atunci când valoarea ei este solicitată** într-o altă zonă din letrec
- Se evaluatează în ordine expresiile din corpul funcției ($\text{expr}_1, \text{expr}_2, \dots, \text{expr}_n$)
- Rezultatul este valoarea lui expr_n (valorile lui $\text{expr}_1, \text{expr}_2, \dots, \text{expr}_{n-1}$ se pierd)

Construcția „named let”

```
(let nume ((var1 e1) (var2 e2) ... (varm em))  
  ...  
  (nume arg1 arg2 ... argm)  
  ...)
```

Semnificație

- Se creează o funcție recursivă (care va fi invocată în corpul named let-ului prin nume) cu parametrii var₁, var₂, ... var_m, și se aplică funcția pe argumentele e₁, e₂, ... e_m
- Domeniul de vizibilitate al variabilei var_k este corpul named let-ului (exceptând aparițiile legate ale lui var_k în corp)
- Ca și la celelalte forme de let, rezultatul este valoarea ultimei expresii evaluate în corp

Legare statică / dinamică – Cuprins

- Variabile
- Domeniu de vizibilitate
- Tipuri de legare a variabilelor
- Expresii pentru legare statică
- Expresii pentru legare dinamică

Construcția **define**

...

(define var expr)

...

La evaluare

- Se evaluatează expr
- Se realizează legarea `var ← valoare(expr)`
- Domeniul de vizibilitate al variabilei `var` este **întregul program**
 - exceptând zonele obscurate de alte legări statice ale lui `var`
 - cu condiția ca la momentul referirii să nu existe o definire mai recentă a lui `var`

Construcția **define** – Consecințe

Avantaje (exemplificate la calculator)

- Funcțiile/expresiile se pot defini în **orice ordine** (cât timp momentul referirii lor le găsește definite)
- Se pot defini **funcții mutual recursive**

Dezavantaje (vezi exemplul de legare dinamică)

- Se pierde **transparenta referențială**

define, let și set! – Comparație

Diferența este în primul rând la nivel **intențional**.

define

- Intenționează **să lege pentru totdeauna** un identificator la o valoare
- De aceea nu e legal în toate zonele de program (nu se poate afla în mijlocul corpului unei funcții, ci doar la început, pentru a defini valori/funcții ajutătoare)

let

- Intenționează **să creeze un context local** pentru anumite variabile
- Nu realizează atribuire, la let variabila nu se modifică ci se naște

set!

- Intenționează **să schimbe valoarea** unei variabile (nu are ce căuta în paradigma funcțională)
- Se poate folosi oriunde în program

Modelul contextual de evaluare – Cuprins

- Context computațional
- Închideri funcționale

Context computațional

Context computațional al unui punct P din program = mulțimea variabilelor care îl au pe P în domeniul lor de vizibilitate (o mulțime de perechi identificator-valoare)

Observații

- Când există doar legare statică
 - contextul unui punct este vizibil imediat ce am scris programul
- Când există și legare dinamică
 - valoarea contextului depinde de momentul în care se află execuția
 - în contextul unui punct P dat, numai valoarea variabilelor legate dinamic poate să difere de la un moment la altul
- Variabilele sunt perechi identificator-valoare, înainte să se realizeze legarea contextul nu conține informații despre identificatorul nelegat (de aceea punctele din corpul unei funcții nu conțin informații despre parametrii formalii ai funcției)

Context computațional – Exemplu

```
1. (define a 1)
2. (define (f x)
3.   (+ x ←
4.     (let ((x 5))
5.       (* a x)))) ; ; 7
6. (f 2)
7. (define a 2)
8. (f 2) ; ; 12
```

Înainte de apelul
de la linia 6:

((a 1))

În timpul apelului
de la linia 6:

((a 1) (x 2))

După apelul
de la linia 8:

((a 2))

((a 1) (x 5))

((a 1) (x 5))

((a 2) (x 5))

Observație: x-ul de la linia 3 se leagă la valoarea 2 abia în momentul în care funcția f este aplicată pe argumentul 2, iar legarea există doar pe durata evaluării apelului.

Modelul contextual de evaluare – Cuprins

- Context computațional
- Închideri funcționale

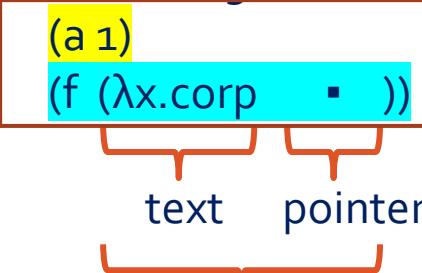
Închideri funcționale

Închidere funcțională = pereche text – context (textul funcției și contextul în punctul de definire a funcției) (cu alte cuvinte: o funcție care știe cine sunt variabilele ei libere)

Exemplu

1. `(define a 1)`
2. `(define (f x)`
3. `(+ x`
4. `(let ((x 5))`
5. `(* a x)))`

Contextul global:



pereche text-context (la asta se evaluează f la define)

La apelul (f 2)

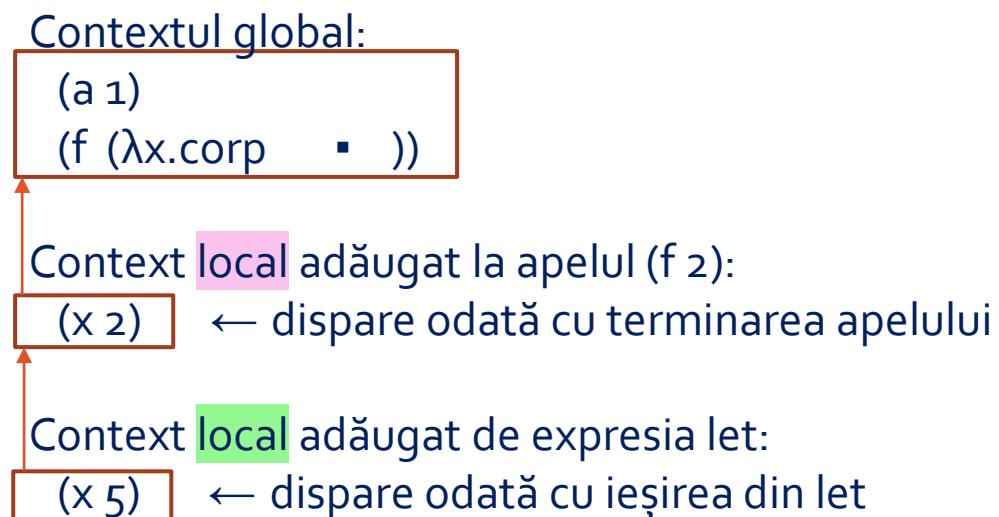
- Se creează un nou context, local, pentru legarea (x 2)
- La evaluarea lui let se creează un nou context, local, pentru legarea (x 5)

Hierarhia de contexte

```
1. (define a 1)
2. (define (f x)
3.   (+ x
4.     (let ((x 5))
5.       (* a x))))
6. (f 2)
```

La evaluare

- Se caută variabila în contextul curent
- Dacă nu este găsită acolo, se caută în contextul părinte, și.a.m.d.



Mai multe exemple

```
1. (define (fact n)
2.   (if (zero? n)
3.     1
4.     (* n (fact (- n 1))))) )
5. (define g fact)
6. (g 4)
7.
8. (define (fact n) n)
9. (g 4)
```

Contextul global:

(fact ($\lambda n.$ corp □))
(g ($\lambda n.$ corp □))

Context local adăugat la apelul (g 4):

(n 4) ← dispare odată cu terminarea apelului
;; 24

Mai multe exemple

```
1. (define (fact n)
2.   (if (zero? n)
3.     1
4.     (* n (fact (- n 1))))) )
5. (define g fact)
6. (g 4)
7.
8. (define (fact n) n)
9. (g 4)
```

Contextul global:

(fact ($\lambda n. corp \quad \cdot \quad$))
(g ($\lambda n. corp \quad \cdot \quad$))

Context local adăugat la apelul (g 4):

(n 4)

← dispare odată cu terminarea apelului

Mai multe exemple

```
1. (define (fact n)
2.   (if (zero? n)
3.     1
4.     (* n (fact (- n 1))))) )
5. (define g fact)
6. (g 4)
7.
8. (define (fact n) n)
9. (g 4)
```

Contextul global:

(fact ($\lambda n. n \quad \cdot \quad \cdot$))
(g ($\lambda n. corp \quad \cdot \quad \cdot$))

contextul global suprascris de linia 8

Context local adăugat la apelul (g 4):

(n 4)

\leftarrow dispare odată cu terminarea apelului

Mai multe exemple

```
1. (define (fact n)
2.   (if (zero? n)
3.     1
4.     (* n (fact (- n 1))))) )
5. (define g fact)
6. (g 4)
7.
8. (define (fact n) n)
9. (g 4)
```

Contextul global:

(fact ($\lambda n. n \quad \cdot \quad \cdot$))
(g ($\lambda n. corp \quad \cdot \quad \cdot$))

contextul global suprascris de linia 8

Context local adăugat la apelul (g 4):

(n 4)

← dispare odată cu terminarea apelului

Context local adăugat la apelul (g 4):

(n 4)

← dispare odată cu terminarea apelului

; ; 12

(g 4) -> (($\lambda (n) (\text{if } (\text{zero? } n) 1 (* n (\text{fact } (- n 1))))$) 4)
-> (* 4 (fact (- 4 1))) -> (* 4 (($\lambda (n) n$) 3)) -> 12

Mai multe exemple

```
1. (define (g x)
2.   (* x x))
3. (define f
4.   (g 5))
5. f
6.
7. (define (g x)
8.   (* x x x))
9. f
```

Mai multe exemple

```
1. (define (g x)
2.   (* x x))
3. (define f          ;; aici f nu e o funcție, și se leagă la 25
4.   (g 5))
5. f              ;; 25
6.
7. (define (g x)
8.   (* x x x))
9. f              ;; tot 25 întrucât asta referă f, nu (g 5)
```

Observație: O închidere funcțională ((f) în loc de f) ar fi amânat evaluarea (g 5). Una din aplicațiile importante ale închiderilor funcționale este **întârzierea evaluării**.

Rezumat

Efecte laterale

Funcții pure

Transparentă referențială

Domeniu de vizibilitate

Legare statică

Legare dinamică

Expresii de legare statică / dinamică

Context computațional

Închidere funcțională

Rezumat

Efecte laterale: alte efecte ale unei funcții în afară de efectul de a întoarce o valoare

Funcții pure

Transparență referențială

Domeniu de vizibilitate

Legare statică

Legare dinamică

Expresii de legare statică / dinamică

Context computațional

Închidere funcțională

Rezumat

Efecte laterale: alte efecte ale unei funcții în afară de efectul de a întoarce o valoare

Funcții pure: aplicate pe aceleasi argumente întorc aceeași valoare; nu au efecte laterale

Transparentă referențială

Domeniu de vizibilitate

Legare statică

Legare dinamică

Expresii de legare statică / dinamică

Context computațional

Închidere funcțională

Rezumat

Efecte laterale: alte efecte ale unei funcții în afară de efectul de a întoarce o valoare

Functii pure: aplicate pe aceleasi argumente întorc aceeași valoare; nu au efecte laterale

Transparentă referențială: toate funcțiile/expresiile sunt pure

Domeniu de vizibilitate

Legare statică

Legare dinamică

Expresii de legare statică / dinamică

Context computațional

Închidere funcțională

Rezumat

Efecte laterale: alte efecte ale unei funcții în afară de efectul de a întoarce o valoare

Functii pure: aplicate pe aceleasi argumente întorc aceeași valoare; nu au efecte laterale

Transparentă referențială: toate funcțiile/expresiile sunt pure

Domeniu de vizibilitate: mulțimea punctelor din program în care variabila e vizibilă

Legare statică

Legare dinamică

Expresii de legare statică / dinamică

Context computațional

Închidere funcțională

Rezumat

Efecte laterale: alte efecte ale unei funcții în afară de efectul de a întoarce o valoare

Functii pure: aplicate pe aceleasi argumente întorc aceeași valoare; nu au efecte laterale

Transparentă referențială: toate funcțiile/expresiile sunt pure

Domeniu de vizibilitate: mulțimea punctelor din program în care variabila e vizibilă

Legare statică: domeniu de vizibilitate controlat textual, determinat la compilare

Legare dinamică

Expresii de legare statică / dinamică

Context computațional

Închidere funcțională

Rezumat

Efecte laterale: alte efecte ale unei funcții în afară de efectul de a întoarce o valoare

Functii pure: aplicate pe aceleasi argumente întorc aceeași valoare; nu au efecte laterale

Transparentă referențială: toate funcțiile/expresiile sunt pure

Domeniu de vizibilitate: mulțimea punctelor din program în care variabila e vizibilă

Legare statică: domeniu de vizibilitate controlat textual, determinat la compilare

Legare dinamică: domeniu de vizibilitate controlat de timp, determinat la execuție

Expresii de legare statică / dinamică

Context computațional

Închidere funcțională

Rezumat

Efecte laterale: alte efecte ale unei funcții în afară de efectul de a întoarce o valoare

Functii pure: aplicate pe aceleasi argumente întorc aceeași valoare; nu au efecte laterale

Transparentă referențială: toate funcțiile/expresiile sunt pure

Domeniu de vizibilitate: mulțimea punctelor din program în care variabila e vizibilă

Legare statică: domeniu de vizibilitate controlat textual, determinat la compilare

Legare dinamică: domeniu de vizibilitate controlat de timp, determinat la execuție

Expresii de legare statică / dinamică: lambda, let, let*, letrec, named let / define

Context computațional

Închidere funcțională

Rezumat

Efecte laterale: alte efecte ale unei funcții în afară de efectul de a întoarce o valoare

Functii pure: aplicate pe aceleasi argumente întorc aceeași valoare; nu au efecte laterale

Transparentă referențială: toate funcțiile/expresiile sunt pure

Domeniu de vizibilitate: mulțimea punctelor din program în care variabila e vizibilă

Legare statică: domeniu de vizibilitate controlat textual, determinat la compilare

Legare dinamică: domeniu de vizibilitate controlat de timp, determinat la execuție

Expresii de legare statică / dinamică: lambda, let, let*, letrec, named let / define

Context computational: (într-un punct P): mulțimea variabilelor care îl au pe P în domeniu

Închidere funcțională

Rezumat

Efecte laterale: alte efecte ale unei funcții în afară de efectul de a întoarce o valoare

Functii pure: aplicate pe aceleasi argumente întorc aceeași valoare; nu au efecte laterale

Transparentă referențială: toate funcțiile/expresiile sunt pure

Domeniu de vizibilitate: mulțimea punctelor din program în care variabila e vizibilă

Legare statică: domeniu de vizibilitate controlat textual, determinat la compilare

Legare dinamică: domeniu de vizibilitate controlat de timp, determinat la execuție

Expresii de legare statică / dinamică: lambda, let, let*, letrec, named let / define

Context computational: (într-un punct P): mulțimea variabilelor care îl au pe P în domeniu

Închidere funcțională: pereche textul funcției – contextul în punctul de definire a funcției

PARADIGME DE PROGRAMARE

Curs 5

Întârzierea evaluării. Închideri funcționale versus promisiuni. Fluxuri.

Întârzierea evaluării – Cuprins

- Importanța evaluării întârziate
- Implementare cu Închideri funcționale
- Implementare cu promisiuni
- Funcțiile delay și force

Evaluare întârziată

Evaluare întârziată

- evaluarea expresiilor este amânată până când valoarea lor este necesară (unui alt calcul)

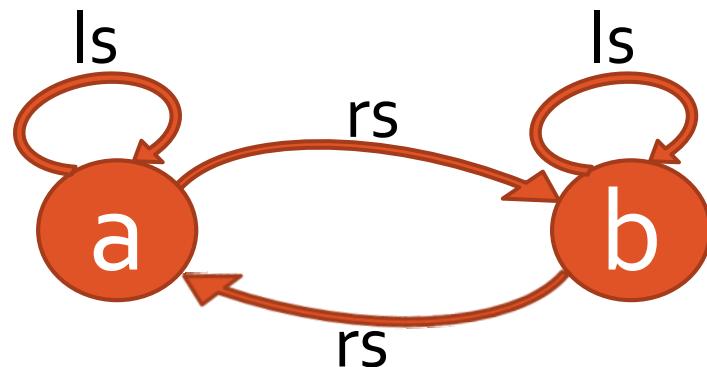
Beneficii

- **Performanță** crescută (se evită calcule inutile – care pot fi multe sau costisitoare)
- Implementare de **funcții nestrictă** (if, and, or) utile în:
 - Controlul fluxului prin program
 - Condiții de oprire
`(or (null? L) (zero? (car L)))` – se evaluatează `(car L)` doar dacă `L` nu e vidă
- **Structuri de date infinite**, din care se evaluatează (la cerere) doar o porțiune finită de lungime necunoscută în prealabil
 - `[0 ..]` – lista infinită de numere naturale (Haskell)
`[0 ..] !! n` – al `n`-lea element al listei

Întârzierea evaluării – Cuprins

- Importanța evaluării întârziate
- Implementare cu Închideri funcționale
- Implementare cu promisiuni
- Funcțiile delay și force

Exemplu – Un graf recursiv (infinit)



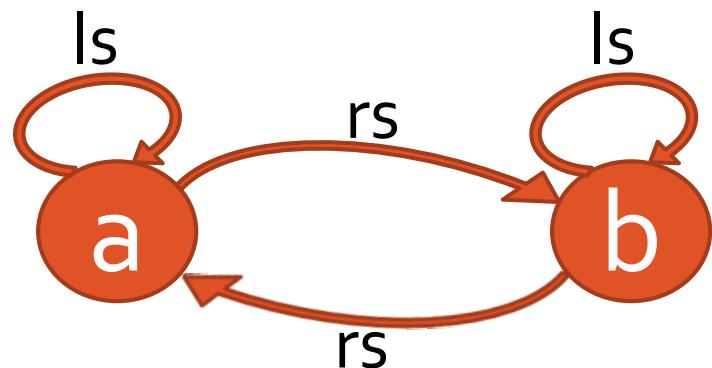
Un nod se reprezintă prin
key = informația din nod
ls = legătura la stânga
rs = legătura la dreapta

Un graf se reprezintă prin
nodul rădăcină (aici a)

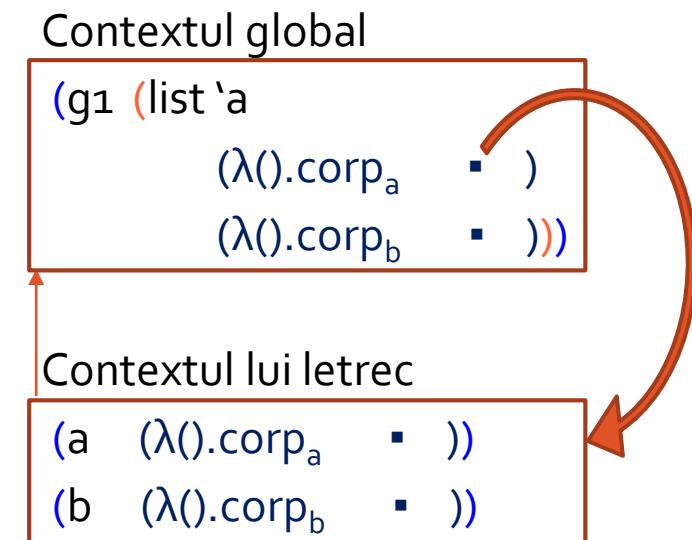
Structură infinită

→ trebuie să împiedicăm cumva evaluarea întregului graf în momentul definirii sale

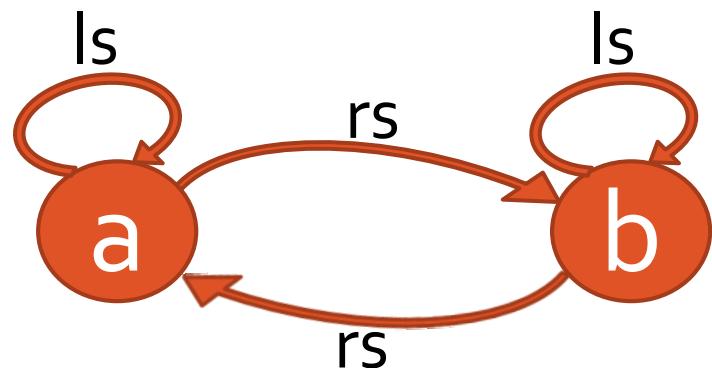
Exemplu – Un graf recursiv (infinit)



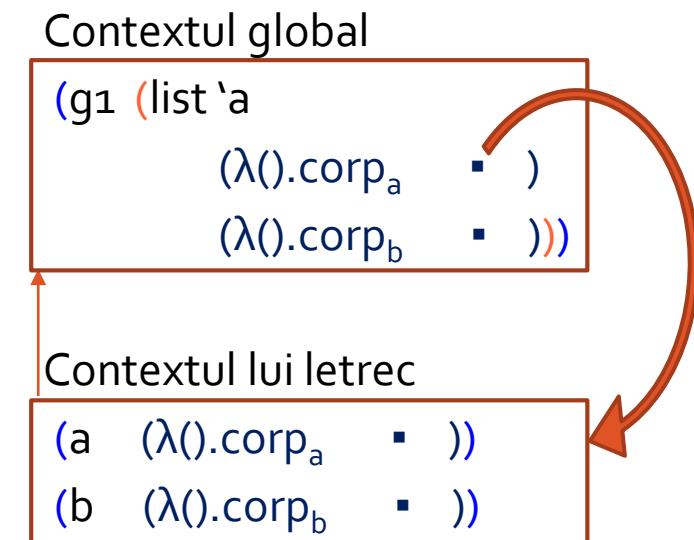
1. `(define g1`
2. `(letrec ((a (λ () (list 'a a b))))`
3. `(b (λ () (list 'b b a))))`
4. `(a))))`



Exemplu – Un graf recursiv (infinit)

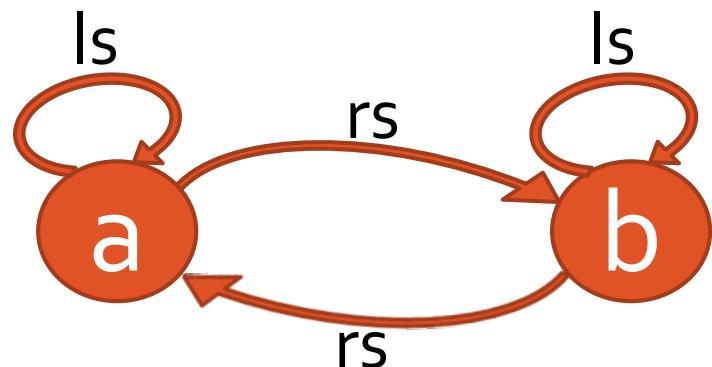


1. `(define key car)`
2. `(define (ls node) ((second node)))` ←
3. `(define (rs node) ((third node)))`



(ls g1) produce o **nouă aplicare** (a) care întoarce tot (list 'a (λ().corp_a ·) (λ().corp_b ·))

Exemplu – Un graf recursiv (infinit)



Corful lui a este evaluat de 3 ori

Nu și în zonele subliniate cu verde, unde **g1** este deja evaluat și se ia din contextul global

```
1. (define g1
2.   (letrec ((a (λ () (list 'a a b)))
3.           (b (λ () (list 'b b a))))
4.         (a)))
5. 
6. (define key car)
7. 
8. (define (ls node) ((second node)))
9. 
10. (define (rs node) ((third node)))
11. 
12. g1
13. (eq? g1 (ls g1))
14. (equal? g1 (ls g1))
```

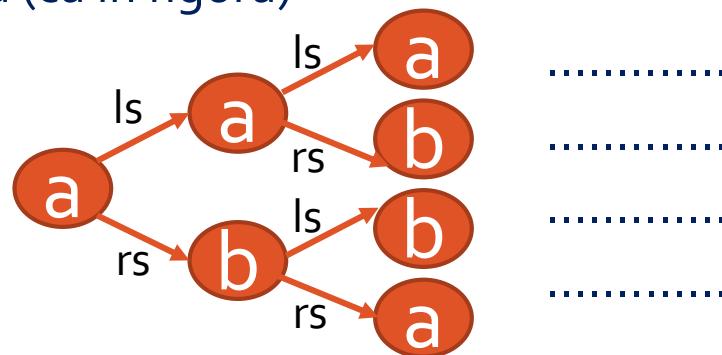
Implementare cu închideri funcționale

Avantaj

- Capabilă să reprezinte graful infinit

Dezavantaje

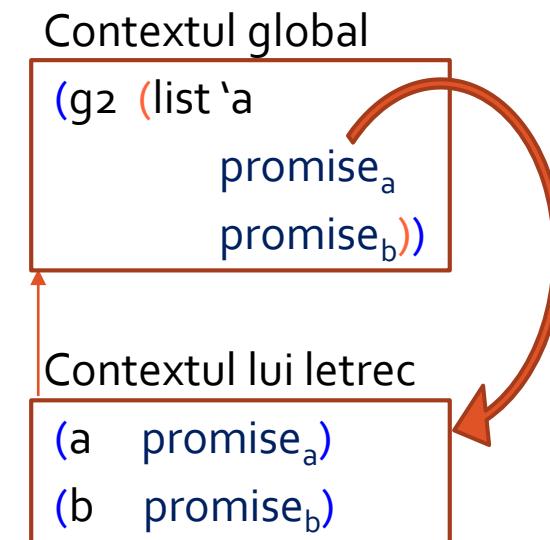
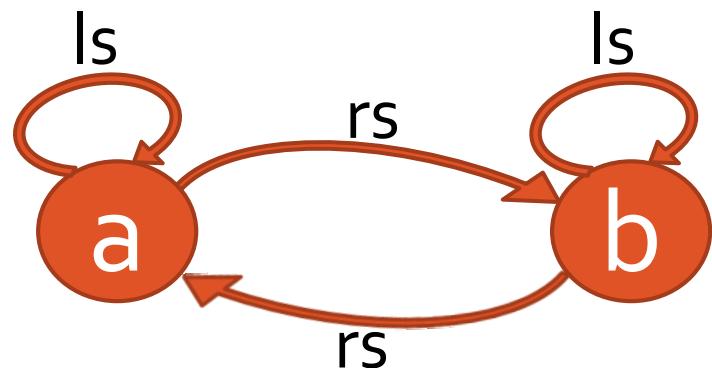
- **Ineficientă** din cauza evaluării repetitive (de fiecare dată când accesăm vecinii unui nod) a unor închideri deja evaluate
- **Probleme „filozofice” de identitate**: vecinul stâng al lui a este chiar a, nu un alt nod identic cu a (ca în figură)



Întârzierea evaluării – Cuprins

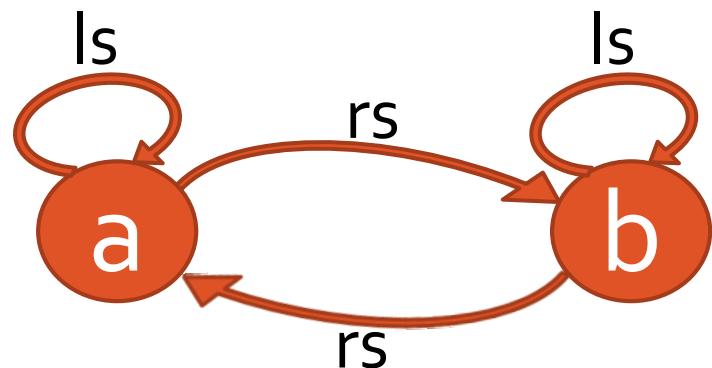
- Importanța evaluării întârziate
- Implementare cu închideri funcționale
- **Implementare cu promisiuni**
- Funcțiile delay și force

Exemplu – Un graf recursiv (infinit)

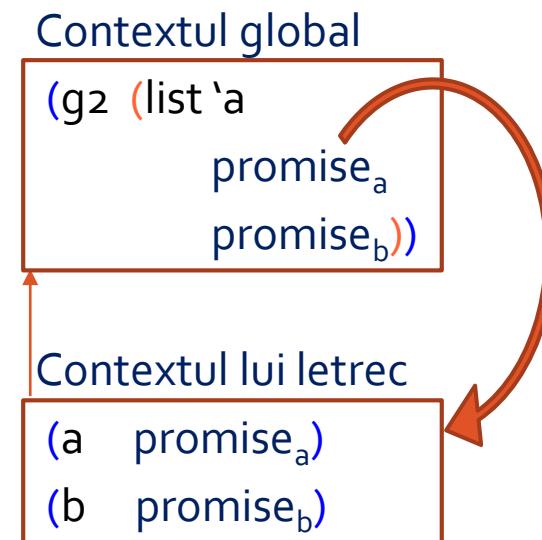


1. `(define g2`
2. `(letrec ((a (delay (list 'a a b))))`
3. `(b (delay (list 'b b a))))` ← $(\lambda () \text{expr})$ se înlocuiește cu (delay expr)
4. `(force a)))` ← (a) se înlocuiește cu (force a)

Exemplu – Un graf recursiv (infinit)



1. `(define key car)`
2. `(define (ls node) (force (second node)))` ←
3. `(define (rs node) (force (third node)))`



(ls g2) produce o **nouă forțare** a lui promise_a care întoarce **rezultatul depus în cache la prima forțare**; are loc **o singură evaluare** a expresiei întârziată cu delay

Implementare cu promisiuni

Avantaje

- Capabilă să reprezinte graful infinit
- **Eficientă**
 - Fiecare din cele 2 promisiuni își evaluează expresia o singură dată
 - Când valoarea unei promisiuni este solicitată ulterior, ea este luată din cache
- **Identitate** a obiectelor întoarse de evaluarea promisiunii la diferite momente de timp
 - Cum oricum nu se produce decât o evaluare a expresiei întârziate, nu se poate întâmpla nimic (de exemplu, legări dinamice) astfel încât două evaluări diferite ale promisiunii să întoarcă valori diferite
 - Acum graful arată și fizic ca cel pe care ne-am propus să îl reprezentăm

Observație

- Atât închiderile funcționale cât și promisiunile sunt **valori de ordinul întâi** (ex: le-am putut stoca în liste)

Întârzierea evaluării – Cuprins

- Importanța evaluării întârziate
- Implementare cu închideri funcționale
- Implementare cu promisiuni
- Funcțiile delay și force

Funcția (nestrictă) **delay**

(delay expr) creează o **promisiune**, care, conceptual, arată cam aşa:



→ expresia întârziată

→ inițial #t – expresia nu a fost încă evaluată

→ inițial nimic – expresia nu a fost încă evaluată

Posibilă implementare pentru `delay`

```
1. (define (delay expr) (memoize (λ () expr)))  
2.  
3. (define (memoize thunk)  
4.   (let ((need-val? #t)  
5.         (val 'whatever))  
6.     (λ ()  
7.       (if need-val?  
8.           (begin  
9.             (set! val (thunk))  
10.            (set! need-val? #f))))  
11.      val)))
```

Ne bazăm tot pe închideri funcționale, dar cu o optimizare importantă

O promisiune este un **obiect cu stare**: prima forțare produce efecte laterale, acceptabile întrucât se află sub bariera de abstractizare (obs: if-ul fără else este posibil și el în Pretty Big)

La forțări ulterioare se întoarce direct val

Funcția `force`

(`force` `promise`) forțează o promisiune, în sensul că solicită valoarea expresiei întârziată în promisiune:

- Dacă expresia a fost deja evaluată (flag #f)
întoarce valoare
- Altfel
 $\text{valoare} \leftarrow \text{evaluează expresie}$
 $\text{flag} \leftarrow \#f$
întoarce valoare

Observație

- Odată ce o promisiune a fost forțată și rezultatul stocat în cache, acest **rezultat nu se mai schimbă** (chiar dacă între timp, din cauza unor legări dinamice sau efecte laterale, o nouă evaluare a expresiei întârziate ar produce altceva)

Fluxuri – Cuprins

- Motivație
- Fluxuri
- Fluxuri definite explicit
- Fluxuri definite implicit
- Ciurul lui Eratostene

Modelarea lumii înconjurătoare

Scopul sistemelor software

- Modelarea lumii înconjurătoare, reducând pe cât posibil:
 - **Complexitatea temporală/spațială**
 - **Complexitatea intelectuală** – identificând module independente sau interdependente și abstractizându-le (pentru a ascunde complexitatea lor în spatele unor interfețe simplu de utilizat)

Viziuni posibile asupra lumii

- Obiecte (module) care își schimbă starea în timp
 - Pentru modularitate, starea e înglobată în interiorul obiectelor (ex: obiectul „promisiune”)
 - Rezultă o programare cu efecte laterale, cu obiecte partajate, cu probleme de sincronizare, etc.
 - Timpului din lumea reală îi corespunde timpul din program
- Un tot (care nu se schimbă) descris prin colecția stadiilor sale de evoluție (ex: funcțiile sin, cos)
 - Rezultă o programare de nivel mai înalt, unde timpul nu trebuie controlat explicit

Exemplu la calculator

Să se determine dacă un număr n este prim.

Definiție

Un număr n este prim dacă și numai dacă nu are divizori în intervalul $[2 .. \sqrt{n}]$.

O soluție modulară



Eleganță versus Eficiență

```
1. (define (interval a b)
2.   (if (> a b)
3.     '()
4.     (cons a (interval (add1 a) b))))
5. (define (prime? n)
6.   (null?
7.     (filter (λ (d) (zero? (modulo n d)))
8.             (interval 2 (sqrt n))))))
```

- Construiește tot intervalul, apoi tot intervalul filtrat, chiar când se găsește din start un divizor
- **Elegant**, dar **ineficient temporal și spațial**

```
1. (define (prime? n)
2.   (let iter ((div 2))
3.     (cond
4.       ((> (* div div) n) #t)
5.       ((zero? (modulo n div)) #f)
6.       (else (iter (add1 div))))))
```

- Nu reține mai mult de 2 variabile la un moment dat (**eficient spațial**)
- Se oprește la primul divizor (**eficient temporal**)
- **Neelegant**: amestecă generarea / filtrarea / testarea

Fluxuri – Cuprins

- Motivație
- Fluxuri
- Fluxuri definite explicit
- Fluxuri definite implicit
- Ciurul lui Eratostene

Eleganță plus Eficiență

```
1. (define (interval-stream a b)                                interfață foarte  
2.   (if (> a b)                                               asemănătoare  
3.     empty-stream                                         celei pentru liste  
4.     (stream-cons a (interval-stream (add1 a) b))))  
5. (define (prime? n)  
6.   (stream-empty?  
7.    (stream-filter (λ (d) (zero? (modulo n d)))  
8.                  (interval-stream 2 (sqrt n))))))
```

- **Elegant și eficient temporal și spațial**, deși, conceptual, este același program cu cel pe liste
- Care este diferența esențială între varianta cu liste și varianta cu fluxuri?

Sub bariera de abstractizare



Complexitate spațială

- Intervalul $[1 .. n]$ reținut ca **listă: $\Theta(n)$** (n elemente ținute în memorie)
- Intervalul $[1 .. n]$ reținut ca **flux: $\Theta(1)$** (un element și o promisiune de a evalua restul fluxului)

1 → (delay (interval-stream 2 b))
(rețin textul expresiei + contextul – cine erau
interval-stream și b)

Complexitate temporală

- $(\text{prime? } n)$ cu **liste: $\Theta(\sqrt{n})$** (generez \sqrt{n} numere, parcurg \sqrt{n} numere ca să le filtrez, aplic null?)
- $(\text{prime? } n)$ cu **fluxuri: $\Theta(\text{distanța până la primul divizor})$** ($\Theta(\sqrt{n})$ pentru n prim)

Exemplu

```
(prime? 115)
(stream-empty? (stream-filter div? '(2 . promise[3-10.72])))
;; 115 : 2? Nu, atunci filtrăm restul intervalului
(stream-empty? (stream-filter div? '(3 . promise[4-10.72])))
;; 115 : 3? Nu, atunci filtrăm restul intervalului
(stream-empty? (stream-filter div? '(4 . promise[5-10.72])))
;; 115 : 4? Nu, atunci filtrăm restul intervalului
(stream-empty? (stream-filter div? '(5 . promise[6-10.72])))
;; 115 : 5? Da, atunci avem un rezultat pentru stream-filter
;; Aceasta i-a cerut intervalului să se deșire până a găsit un divizor
(stream-empty? '(5 . promise[stream-filter div? '(6 . promise[7-10.72])]))
;; #f (stream-empty nu are nevoie să evalueze restul fluxului)
```

Liste versus fluxuri – Comparație

Liste (secvențe complet construite)

- Funcționalele pe liste exprimă succint și elegant o gamă largă de operații
- Eleganța se plătește prin ineficiență: la fiecare pas trebuie copiate și (re)prelucrate listele întregi (care pot fi foarte mari)
- Etapa de **construcție** a listei și etapa de **prelucrare** a ei – **separate conceptual și computațional**

Fluxuri (secvențe parțial construite)

- Sunt **liste cu evaluare întârziată** – au aceleași abstracțiuni elegante ca listele (map, filter, etc.)
- Fiecare flux își **produce elementele unul câte unul, dacă funcția apelantă o cere**
 - Dacă funcției îi este suficient primul element, nu se mai evaluatează restul (vezi stream-empty? anterior)
 - Dacă funcția încearcă să accesize o porțiune de flux încă necalculată, fluxul se extinde automat doar până unde este necesar, păstrând astfel **iluzia că lucrăm cu întregul flux**
- Etapa de **construcție** a fluxului și etapa de **prelucrare** a lui – **separate doar conceptual, computațional construcția este dirijată de nevoia de prelucrare (evitând calcule inutile)**

Liste versus fluxuri – Interfață

Constructor pe liste

'()

cons

Operator pe liste

null?

car

cdr

map

filter

Constructor corespunzător pe fluxuri

empty-stream

stream-cons

Operator corespunzător pe fluxuri

stream-empty?

stream-first

stream-rest

stream-map

stream-filter

Fluxuri – Cuprins

- Motivație
- Fluxuri
- Fluxuri definite explicit
- Fluxuri definite implicit
- Ciurul lui Eratostene

Definiții explicite (generând fiecare element)

Flux = pereche (**element** . **motor**-capabil-să-genereze-restul-fluxului)

Motor: implementat de regulă ca o funcție recursivă, cu parametri pe baza cărora să se poată genera elementul următor din flux

Exemple

```
(define naturals
  (let loop ((n 0)) ;; aici am nevoie de numărul curent în flux
    (stream-cons n (loop (add1 n)))))

(define factorials ;; n-ul cu care urmează să înmulțesc
  (let loop ((n 1) (fact 1)) ;; și factorialul curent
    (stream-cons fact (loop (add1 n) (* n fact)))))
```

Extinderea fluxului la cerere

```
(define naturals
  (let loop ((n 0))
    (stream-cons n (loop (add1 n)))))
```

- Inițial, fluxul își știe doar primul element: naturals = '(0 
- **Extinderea** fluxului (calcularea mai multor elemente) se face **la cerere** (adică la stream-rest): de exemplu, (stream-take naturals 3) va cere încă 2 elemente
- **Evoluția** '(0  loop 1)  loop 2)  loop 3)
- **stream-cons** face un **delay** : sub barieră (stream-cons a b) ~ (cons a (delay b))
- **stream-rest** face un **force** : sub barieră (stream-rest s) ~ (force (cdr s))
- Noi folosim interfața fără să ne pese ce se petrece sub bariera de abstractizare

Fluxuri – Cuprins

- Motivație
- Fluxuri
- Fluxuri definite explicit
- Fluxuri definite implicit
- Ciurul lui Eratostene

Definiții implicite (pe baza altor fluxuri)

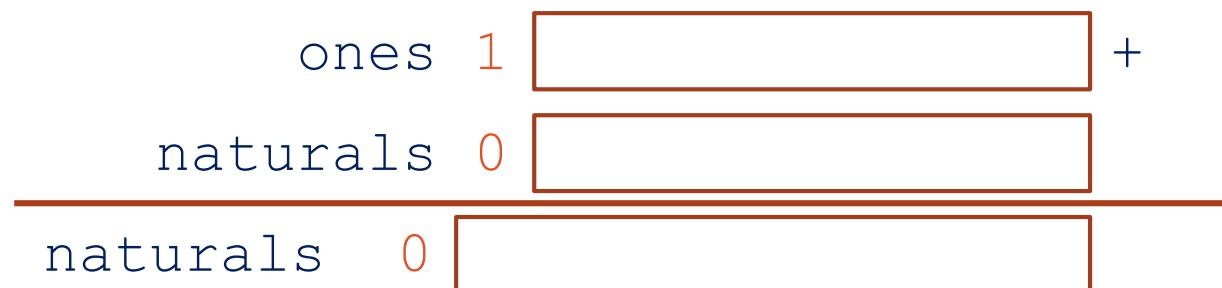
Definiție implicită (fără generator (motor) explicit)

- Profită de evaluarea leneșă pentru a defini fluxul pe baza altor fluxuri (sau a lui însuși)
- Mecanisme uzuale
 - Transformarea (map) sau filtrarea (filter) unui alt flux
 - Operații între două fluxuri (adunare, înmulțire, etc.)
 - Când fluxul este definit inclusiv **pe baza lui însuși**, este esențial să **dăm explicit măcar un element** de la care să pornească, altfel nu are cum să participe la o primă operație

Exemple

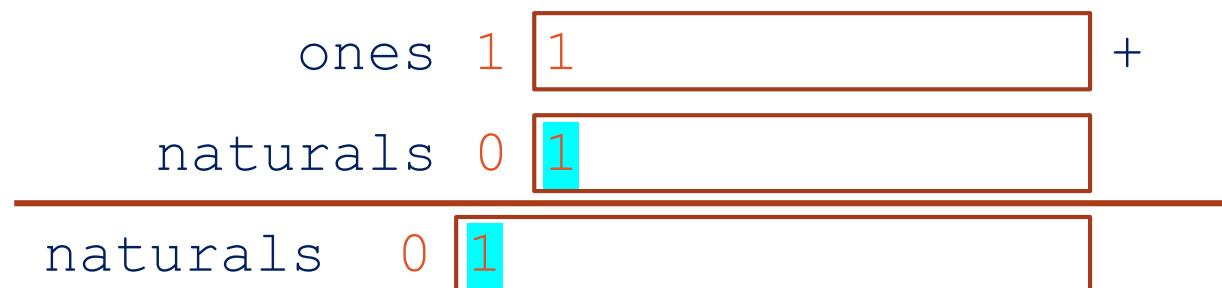
```
(define ones (stream-cons 1 ones))  
  
(define naturals-  
  (stream-cons 0  
    (stream-zip-with + ones naturals-)))
```

Operații între fluxuri – Funcționare



- Pentru a obține al doilea element din `naturals`, e nevoie de primul element din `ones` și primul element din `naturals`, care sunt deja disponibile
- **Observație:** de aceea era necesar să dăm explicit măcar un element din `naturals`, pentru a avea cu ce începe adunările

Operații între fluxuri – Funcționare



- Pentru a obține al treilea element din `naturals`, e nevoie de al doilea element din `ones` și al doilea element din `naturals`, care sunt deja disponibile

Operații între fluxuri – Funcționare

ones	1	1	1	1	1	1	1	1	1	+
naturals	0	1	2	3	4	5	6	7	8	
<hr/>										
naturals	0	1	2	3	4	5	6	7	8

- De fiecare dată avem disponibile exact elementele necesare pentru a calcula elementul următor

Operații între fluxuri – Fibonacci

fibo	0	1 1 2 3 5 8	+
(rest fibo)	1	1 2 3 5 8 13	
<hr/>			
fibo	?	1 2 3 5 8 13 21 . . .	

- Ce trebuie să dăm explicit fluxului `fibo` pentru a putea apoi începe adunările?

Operații între fluxuri – Fibonacci

fibo	0	1	1	2	3	5	8	+			
(rest fibo)	1	1	2	3	5	8	13				
<hr/>											
fibo	0	1	1	2	3	5	8	13	21

- Ce trebuie să dăm explicit fluxului `fibo` pentru a putea apoi începe adunările?
 - Se vede că
 - Din rezultatul adunărilor lipsesc termenii **0 și 1**
 - Este nevoie de primul element din `fibo` și de primul element din `(rest fibo)` pentru a începe adunările
- trebuie să **dăm explicit primii 2 termeni**

Mai multe definiții implice

Exemple la calculator:

- Fluxul numerelor pare
- Fluxul puterilor lui 2
- Fluxul $1/n!$ – cu care se poate aproxima numărul e
- Fluxul sumelor parțiale ale altui flux (atenție la definiția eficientă versus cea ineficientă!)

Reutilizarea fluxului

De ce sunt implementate fluxurile cu promisiuni, nu cu închideri funcționale?

- Diferența de **eficiență** este foarte mare în situațiile în care reutilizăm un flux din care am calculat deja un număr de elemente
 - Promisiunile nu reevaluatează porțiunile deja calculate, ci iau rezultatele din cache
 - Închiderile funcționale reevaluatează tot

Exemplu la calculator: fibonacci cu închideri versus fibonacci cu promisiuni

Fluxuri – Cuprins

- Motivație
- Fluxuri
- Fluxuri definite explicit
- Fluxuri definite implicit
- Ciurul lui Eratostene

Ciurul lui Eratostene

② 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23
~~24~~ 25 ~~26~~ 27 ~~28~~ 29 ~~30~~ 31 ~~32~~ 33 ~~34~~ 35 ~~36~~ 37 ~~38~~ 39 ~~40~~ ...

- Pornim de la fluxul numerelor naturale, începând cu 2
- Primul element p din flux este prim
- Aplicăm același algoritm pe restul fluxului din care eliminăm multiplii lui p

Ciurul lui Eratostene

2 $\cancel{3}$ $\cancel{4}$ $\cancel{5}$ $\cancel{6}$ $\cancel{7}$ $\cancel{8}$ $\cancel{9}$ $\cancel{10}$ $\cancel{11}$ $\cancel{12}$ $\cancel{13}$ $\cancel{14}$ $\cancel{15}$ $\cancel{16}$ $\cancel{17}$ $\cancel{18}$ $\cancel{19}$ $\cancel{20}$ $\cancel{21}$ $\cancel{22}$ $\cancel{23}$
 $\cancel{24}$ $\cancel{25}$ $\cancel{26}$ $\cancel{27}$ $\cancel{28}$ $\cancel{29}$ $\cancel{30}$ $\cancel{31}$ $\cancel{32}$ $\cancel{33}$ $\cancel{34}$ $\cancel{35}$ $\cancel{36}$ $\cancel{37}$ $\cancel{38}$ $\cancel{39}$ $\cancel{40}$...

- Pornim de la fluxul numerelor naturale, începând cu 2
- Primul element p din flux este prim
- Aplicăm același algoritm pe restul fluxului din care eliminăm multiplii lui p

Ciurul lui Eratostene

Diagram illustrating the Sieve of Eratostene. It shows a sequence of numbers from 2 to 40. Prime numbers are circled in blue or red. Non-prime numbers are crossed out with a red or blue diagonal line.

The circled numbers are: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37.

The crossed-out numbers are: 4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25, 26, 27, 28, 30, 32, 33, 34, 35, 36, 38, 39, 40.

- Pornim de la fluxul numerelor naturale, începând cu 2
- Primul element p din flux este prim
- Aplicăm același algoritm pe restul fluxului din care eliminăm multiplii lui p

Ciurul lui Eratostene – Implementare

```
1. (define (sieve s)
2.   (let ((p (stream-first s)))
3.     (stream-cons p (sieve (stream-filter
4.                           (λ (n) (not (zero? (modulo n p)))))
5.                           (stream-rest s))))))
6.
7. (define primes
8.   (sieve (stream-rest (stream-rest naturals)))))
```

Rezumat

Promisiune

Avantaje promisiuni

delay / force

Fluxuri

Avantaje fluxuri

Constructori flux

Operatori flux

Definiții explicite

Definiții implicite

Rezumat

Promisiune: încapsularea unei expresii pentru a fi evaluată mai târziu, la cerere

Avantaje promisiuni

delay / force

Fluxuri

Avantaje fluxuri

Constructori flux

Operatori flux

Definiții explicite

Definiții implicite

Rezumat

Promisiune: încapsularea unei expresii pentru a fi evaluată mai târziu, la cerere

Avantaje promisiuni: eficientă: evaluarea se produce doar la nevoie și o singură dată
delay / force

Fluxuri

Avantaje fluxuri

Constructori flux

Operatori flux

Definiții explicite

Definiții implicite

Rezumat

Promisiune: încapsularea unei expresii pentru a fi evaluată mai târziu, la cerere

Avantaje promisiuni: eficientă: evaluarea se produce doar la nevoie și o singură dată

delay / force: creează o promisiune / solicită valoarea expresiei întârziată într-o promisiune

Fluxuri

Avantaje fluxuri

Constructori flux

Operatori flux

Definiții explicite

Definiții implicite

Rezumat

Promisiune: încapsularea unei expresii pentru a fi evaluată mai târziu, la cerere

Avantaje promisiuni: eficientă: evaluarea se produce doar la nevoie și o singură dată

delay / force: creează o promisiune / solicită valoarea expresiei întârziată într-o promisiune

Fluxuri: secvențe parțial construite (~liste cu evaluare întârziată)

Avantaje fluxuri

Constructori flux

Operatori flux

Definiții explicite

Definiții implicite

Rezumat

Promisiune: încapsularea unei expresii pentru a fi evaluată mai târziu, la cerere

Avantaje promisiuni: eficientă: evaluarea se produce doar la nevoie și o singură dată

delay / force: creează o promisiune / solicită valoarea expresiei întârziată într-o promisiune

Fluxuri: secvențe parțial construite (~liste cu evaluare întârziată)

Avantaje fluxuri: eleganță (modularitate), eficientă (temporală și spațială)

Constructori flux

Operatori flux

Definiții explicite

Definiții implicite

Rezumat

Promisiune: încapsularea unei expresii pentru a fi evaluată mai târziu, la cerere

Avantaje promisiuni: eficientă: evaluarea se produce doar la nevoie și o singură dată

delay / force: creează o promisiune / solicită valoarea expresiei întârziată într-o promisiune

Fluxuri: secvențe parțial construite (~liste cu evaluare întârziată)

Avantaje fluxuri: eleganță (modularitate), eficientă (temporală și spațială)

Constructori flux: empty-stream, stream-cons

Operatori flux

Definiții explicite

Definiții implicite

Rezumat

Promisiune: încapsularea unei expresii pentru a fi evaluată mai târziu, la cerere

Avantaje promisiuni: eficientă: evaluarea se produce doar la nevoie și o singură dată

delay / force: creează o promisiune / solicită valoarea expresiei întârziată într-o promisiune

Fluxuri: secvențe parțial construite (~liste cu evaluare întârziată)

Avantaje fluxuri: eleganță (modularitate), eficientă (temporală și spațială)

Constructori flux: empty-stream, stream-cons

Operatori flux: stream-empty?, stream-first, stream-rest, stream-map, stream-filter

Definiții explicite

Definiții implicite

Rezumat

Promisiune: încapsularea unei expresii pentru a fi evaluată mai târziu, la cerere

Avantaje promisiuni: eficientă: evaluarea se produce doar la nevoie și o singură dată

delay / force: creează o promisiune / solicită valoarea expresiei întârziată într-o promisiune

Fluxuri: secvențe parțial construite (~liste cu evaluare întârziată)

Avantaje fluxuri: eleganță (modularitate), eficientă (temporală și spațială)

Constructori flux: empty-stream, stream-cons

Operatori flux: stream-empty?, stream-first, stream-rest, stream-map, stream-filter

Definiții explicite: generator recursiv care produce fluxul element cu element

Definiții implicite

Rezumat

Promisiune: încapsularea unei expresii pentru a fi evaluată mai târziu, la cerere

Avantaje promisiuni: eficientă: evaluarea se produce doar la nevoie și o singură dată

delay / force: creează o promisiune / solicită valoarea expresiei întârziată într-o promisiune

Fluxuri: secvențe parțial construite (~liste cu evaluare întârziată)

Avantaje fluxuri: eleganță (modularitate), eficientă (temporală și spațială)

Constructori flux: empty-stream, stream-cons

Operatori flux: stream-empty?, stream-first, stream-rest, stream-map, stream-filter

Definiții explicite: generator recursiv care produce fluxul element cu element

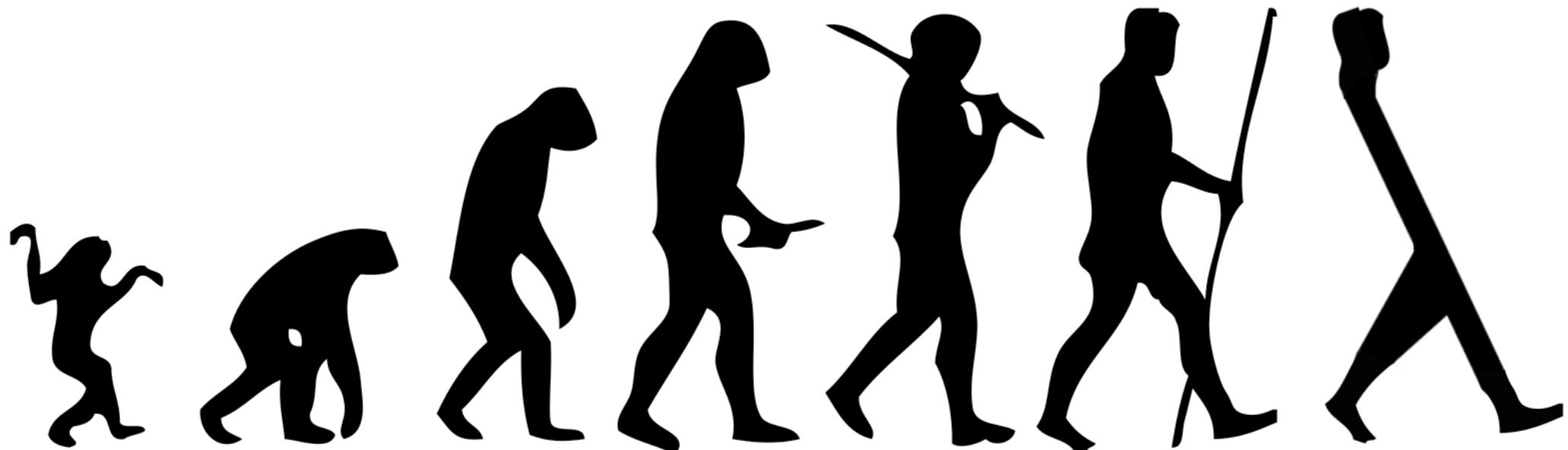
Definiții隐式: transformări/operații de alte fluxuri (sau de fluxul însuși)

PARADIGME DE PROGRAMARE

Curs 6

Limbajul Haskell.

Programare funcțională în Haskell



Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii . și \$
- Expresii condiționale
- Legare statică
- Evaluare lenșă

Sintaxa Haskell

- Notație infixată pentru operatori

`1 + 2, a < 5`

- Notație prefixată pentru funcții

`filter odd [1 .. 7], f 2 + f 5`

- Parantezele se folosesc pentru controlul priorității (nu pentru aplicația de funcție)

`f (1 + 2), f (g x)`

`(f 1) + 2` este echivalent cu `f 1 + 2`, întrucât aplicația de funcție are cea mai mare prioritate

`(f g) x` este echivalent cu `f g x`, întrucât aplicația de funcție este asociativă la stânga

- Indentarea înlocuiește controlul prin separatori ca `{ }` sau ;

Orice cod din **corpu** unei expresii trebuie indentat **mai la dreapta** decât începutul expresiei!

O nouă expresie începe pe **același nivel sau mai la stânga** față de începutul expresiei anterioare!

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii . și \$
- Expresii condiționale
- Legare statică
- Evaluare lenșă

TDA-ul Pereche

Constructori de bază

(,) : $T_1 \times T_2 \rightarrow$ Pereche // creează o pereche între orice 2 argumente

Operatori

fst : Pereche $\rightarrow T_1$ // extrage prima valoare din pereche
snd : Pereche $\rightarrow T_2$ // extrage a doua valoare din pereche

Exemple

```
(1, "unu")  
a = ('a', 2, "a2")  
fst a  
snd (fst a)
```

TDA-ul Pereche

Constructori de bază

(,) : $T_1 \times T_2 \rightarrow$ Pereche // creează o pereche între orice 2 argumente

Operatori

fst : Pereche $\rightarrow T_1$ // extrage prima valoare din pereche
snd : Pereche $\rightarrow T_2$ // extrage a doua valoare din pereche

Exemple

```
(1, "unu") -- (1, "unu")
a = ('a', 2, "a2")
fst a -- ('a', 2)
snd (fst a) -- 2
```

TDA-ul Listă

Constructori (de bază și nu numai)

<code>[]</code> : -> Listă	// creează o listă vidă
<code>: T x Listă -> Listă</code>	// creează o listă prin adăugarea unei valori la începutul unei liste
<code>[, ..,] : T x .. T -> Listă</code>	// creează o listă din toate argumentele sale (de același tip)

Operatori

<code>head</code> : Listă -> T
<code>tail</code> : Listă -> Listă
<code>null</code> : Listă -> Bool
<code>length</code> : Listă -> Nat
<code>++</code> : Listă x Listă -> Listă

Exemple

```
head [[2, 4], [6], [5]]  
tail (2:3:[4, 5])  
null []  
length []  
[1] ++ [1, 2, 3] ++ [4, 5]
```

TDA-ul Listă

Constructori (de bază și nu numai)

<code>[]</code> : -> Listă	// creează o listă vidă
<code>: T x Listă -> Listă</code>	// creează o listă prin adăugarea unei valori la începutul unei liste
<code>[, ..,] : T x .. T -> Listă</code>	// creează o listă din toate argumentele sale (de același tip)

Operatori

<code>head</code> : Listă -> T
<code>tail</code> : Listă -> Listă
<code>null</code> : Listă -> Bool
<code>length</code> : Listă -> Nat
<code>++</code> : Listă x Listă -> Listă

Exemple

<code>head [[2, 4], [6], [5]]</code>	-- [2, 4]
<code>tail (2:3:[4, 5])</code>	-- [3, 4, 5]
<code>null [[]]</code>	-- False
<code>length [[]]</code>	-- 1
<code>[1] ++ [1, 2, 3] ++ [4, 5]</code>	-- [1, 1, 2, 3, 4, 5]

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- **Functii**
- Pattern matching
- Operatorii . și \$
- Expresii condiționale
- Legare statică
- Evaluare lenșă

Funcții anonime în Haskell

\parametri -> corp

Exemple

$\lambda x.x$ $\lambda x \rightarrow x$

$\lambda x.\lambda y.(x y)$ $\lambda x \rightarrow \lambda y \rightarrow x y$ echivalent cu
 $\lambda x y \rightarrow x y$ (întrucât funcțiile Haskell sunt **automat curry**)

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

Funcții cu nume în Haskell

f parametri = corp

Exemple

arithmeticMean = \x -> \y -> (x + y) / 2 echivalent cu

arithmeticMean = \x y -> (x + y) / 2 și cu

arithmeticMean x y = (x + y) / 2

f = arithmeticMean 3 -- se creează \y -> (3 + y) / 2

f 18 -- 10.5

Simularea funcțiilor uncurry

Definițiile de tipul

$f \ x_1 \ x_2 \ \dots \ x_n = \text{corp}$

generează funcții **curry**, care pot fi aplicate pe oricâți ($\leq n$) parametri la un moment dat.

$f \ e_1 \ e_2 \ \dots \ e_k$ întoarce o nouă funcție $\backslash x_{k+1} \ \dots \ x_n \rightarrow \text{corp}_{[e_i/x_i]}$.

Pentru a obține comportamentul de funcție **uncurry**, parametrul lui f trebuie să fie un tuplu:

$f \ (x_1, \ x_2 \ \dots, \ x_n) = \text{corp}$

Exemplu

`arithmeticMean (x, y) = (x + y) / 2`

`arithmeticMean (3, 18)`

Transformări operator – funcție

- (op) face transformarea operator → funcție

```
(-) 3 5           -- -2
(||) (1<2) (5<3) -- True
foldr (+) 0 [1..5] -- 15
(/=) 2 2          -- False
```

- `f` face transformarea funcție → operator

```
5 `mod` 3           -- 2
(div 6) `map` [1,2,3] -- [6,3,2]
((==) 2) `filter` [1,2,3] -- [2]
```

Secțiuni (aplicare parțială a operatorilor)

- Când se dă operandul din stânga, se așteaptă operandul din dreapta

(5/) 2

map (2-) [0..4]

filter (2<) [0..4]

- Când se dă operandul din dreapta, se așteaptă operandul din stânga

(/5) 2

map (-2) [0..4]

map (/2) [0..4]

filter (<2) [0..4]

Secțiuni (aplicare parțială a operatorilor)

- Când se dă operandul din stânga, se așteaptă operandul din dreapta

```
(5/) 2          -- 2.5  
map (2-) [0..4] -- [2,1,0,-1,-2]  
filter (2<) [0..4] -- [3,4]
```

- Când se dă operandul din dreapta, se așteaptă operandul din stânga

```
(/5) 2  
map (-2) [0..4]  
map (/2) [0..4]  
filter (<2) [0..4]
```

Secțiuni (aplicare parțială a operatorilor)

- Când se dă operandul din stânga, se așteaptă operandul din dreapta

```
(5/) 2          -- 2.5  
map (2-) [0..4] -- [2,1,0,-1,-2]  
filter (2<) [0..4] -- [3,4]
```

- Când se dă operandul din dreapta, se așteaptă operandul din stânga

```
(/5) 2          -- 0.4  
map (-2) [0..4] -- eroare, aici -2 e număr, nu funcție  
map (/2) [0..4] -- [0.0,0.5,1.0,1.5,2.0]  
filter (<2) [0..4] -- [0,1]
```

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii . și \$
- Expresii condiționale
- Legare statică
- Evaluare lenșă

Definirea funcțiilor prin pattern matching

Se descrie comportamentul funcțiilor în funcție de structura parametrilor – ca la scrierea axiomelor TDA-ului.

Exemple

1. fib 0 = 0
2. fib 1 = 1
3. fib n = fib (n-2) + fib (n-1)
- 4.
5. sumL [] = 0
6. sumL (x:xs) = x + sumL xs
- 7.
8. sumP (x, y) = x + y
- 9.
10. ordered [] = True
11. ordered [x] = True
12. ordered (x:xs@(y:rest)) = x <= y && ordered xs

permite crearea unui alias pentru valoarea următoare

La aplicare

1. $\text{fib } 0 = 0$
2. $\text{fib } 1 = 1$
3. $\text{fib } n = \text{fib } (n-2) + \text{fib } (n-1)$

$\text{fib } 3$

- Dacă argumentul se potrivește cu parametrul din primul punct (primul pattern)
 - Se folosește definiția din primul punct
 - Se ignoră definițiile următoare
- Altfel
 - Se încearcă potrivirea cu punctul următor, și a.m.d.

Consecință: Ordinea contează!

Pattern-uri exhaustive

Este important să specificăm comportamentul funcției pe toate valorile tipului – ca la scrierea axiomelor TDA-ului.

1. `ordered [] = True`  De aceea trebuie să specificăm și ce se întâmplă pe lista vidă, și ce se întâmplă pe lista cu un singur element
2. `ordered [x] = True` 
3. `ordered (x:xs@(y:rest)) = x <= y && ordered xs`

O definiție alternativă pentru funcția `ordered`:

1. `ordered2 (x:xs@(y:rest)) = x <= y && ordered2 xs`
2. `ordered2 _ = True` 

Se traduce prin „orice altceva”
Unde în definiția lui `ordered` se mai putea folosi?

Când se poate folosi pattern matching

- De fiecare dată **când se leagă variabile**
 - La definirea funcțiilor
 - La crearea de legări locale folosind `let` sau `where` (vom vedea)
- Pattern-urile **nu** se potrivesc și între ele

```
eq x x = True    -- dă eroare Conflicting definitions for `x'  
eq __ = False
```

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii . și \$
- Expresii condiționale
- Legare statică
- Evaluare lenșă

Operatorii . și \$

- **Operatorul . (punct)** realizează **compunere de funcții**

```
myLast = head . reverse -- myLast [1..5] =
```

```
myMin = head . sort      -- myMin [2,4,1,2,3,6,2] =
```

```
myMax = myLast . sort    -- myMax [2,4,1,2,3,6,2] =
```

- **Operatorul \$ (dolar)** realizează **aplicație de funcție**

```
take 4 $ filter (odd . fst) $ zip [1..] [2..]
```

- $f \$ a = f a$ este interesant pentru că \$ are o **prioritate foarte mică**, astfel încât ambele părți vor fi evaluate înainte să se realizeze aplicația de funcție → evităm astfel să folosim foarte multe paranteze
- \$ este **asociativ la dreapta** și este util pentru a rescrie structuri de genul **$f(g(h \dots x))$** , nu poate suplini orice fel de paranteze (vezi (odd . fst) mai sus)

Operatorii . și \$

- **Operatorul . (punct)** realizează **compunere de funcții**

```
myLast = head . reverse -- myLast [1..5] = 5
```

```
myMin = head . sort      -- myMin [2,4,1,2,3,6,2] = 1
```

```
myMax = myLast . sort    -- myMax [2,4,1,2,3,6,2] = 6
```

- **Operatorul \$ (dolar)** realizează **aplicație de funcție**

```
take 4 $ filter (odd . fst) $ zip [1..] [2..]
```

- $f \$ a = f a$ este interesant pentru că \$ are o **prioritate foarte mică**, astfel încât ambele părți vor fi evaluate înainte să se realizeze aplicația de funcție → evităm astfel să folosim foarte multe paranteze
- \$ este **asociativ la dreapta** și este util pentru a rescrie structuri de genul **$f(g(h \dots x))$** , nu poate suplini orice fel de paranteze (vezi `(odd . fst)` mai sus)

Operatorii . și \$

- **Operatorul . (punct)** realizează **compunere de funcții**

```
myLast = head . reverse -- myLast [1..5] = 5
```

```
myMin = head . sort      -- myMin [2,4,1,2,3,6,2] = 1
```

```
myMax = myLast . sort    -- myMax [2,4,1,2,3,6,2] = 6
```

- **Operatorul \$ (dolar)** realizează **aplicație de funcție**

```
take 4 $ filter (odd . fst) $ zip [1..] [2..]
```

```
-- [(1,2), (3,4), (5,6), (7,8)]
```

- $f \$ a = f a$ este interesant pentru că \$ are o **prioritate foarte mică**, astfel încât ambele părți vor fi evaluate înainte să se realizeze aplicația de funcție → evităm astfel să folosim foarte multe paranteze
- \$ este **asociativ la dreapta** și este util pentru a rescrie structuri de genul **$f(g(h \dots x))$** , nu poate suplini orice fel de paranteze (vezi `(odd . fst)` mai sus)

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii . și \$
- Expresii condiționale
- Legare statică
- Evaluare lenșă

Conditionala **if**

if condiție **then** rezultatThen **else** rezultatElse

Exemple

```
if 2<3 then "all normal" else "what did just happen?" -- "all normal"  
uglySum l = if null l then 0  
              else head l + uglySum (tail l)
```

Observație

- Un cod Haskell elegant va folosi pattern matching sau gărzi (vom vedea) înainte de a folosi **if**

Condiționala **case**

```
case expresie of
    pattern1 -> rezultat1
    pattern2 -> rezultat2
    ...
    patternn -> rezultatn
```

Are sens atunci când nu putem folosi pattern matching sau găzzi, de exemplu aici când verificăm structura lui head matrix, mai degrabă decât pe a lui matrix

Exemplu

```
myTranspose matrix = case (head matrix) of
    [] -> []
    _ -> map head matrix : myTranspose (map tail matrix)
```

Gărzi

```
f parametri
| condiție1 = rezultat1
| condiție2 = rezultat2
...
| condițien = rezultatn
[ | otherwise = rezultatn] ← optional
```

Exemplu

```
allEqual a b c
| a==b = b==c
| otherwise = False
```

Au sens atunci când punem condiții asupra variabilelor,
mai degrabă decât să le potrivim cu o anumită structură
(caz în care am folosi pattern matching)

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii . și \$
- Expresii condiționale
- Legare statică
- Evaluare lenșă

Legarea variabilelor în Haskell – statică

- Doar legare statică
- Expresii pentru legare locală: **let** legări **in** expr, expr **where** legări

Exemple

```
myFoldl f acc [] = acc
myFoldl f acc (x:xs) =
    let
        newAcc = f acc x
    in myFoldl f newAcc xs
```

Au sens pentru a spori lizibilitatea codului sau pentru a evita apelarea repetată a aceleiași funcții pe aceleași argumente

```
myFoldr f acc [] = acc
myFoldr f acc (x:xs) = f x rightResult
    where rightResult = myFoldr f acc xs
```

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii . și \$
- Expresii condiționale
- Legare statică
- Evaluare lenșă

Evaluare lenășă

- Toate funcțiile sunt **nestrictă**
- **Evaluare lenășă:** subexpresiile (argumentele) sunt pasate funcției fără a fi evaluate, în corpul funcției ele se vor evalua (eventual parțial) maxim o dată

Exemple

1. $f\ x = 2^x$

2. $g\ x = f\ 2 + f\ 2$

3. $h\ x = x^x^x$

$g\ 5$ -- 2 aplicări distincte ale lui $f \Rightarrow$ se evaluatează de 2 ori

-- $f\ 2 + f\ 2 \rightarrow 4 + f\ 2 \rightarrow 4 + 4 \rightarrow 8$

$h\ (f\ 2)$ -- argumentul se evaluatează o dată și se folosește de 3 ori

-- $(f\ 2)^*(f\ 2)^*(f\ 2) \rightarrow 4*4*4 \rightarrow 64$

Fluxuri

- Evaluare lenășă => toate **listele sunt fluxuri** (se evaluatează în măsura în care e nevoie)

Exemple

```
naturals = let loop n = n : loop (n+1) in loop 0  
ones = 1 : ones  
fibonacci = 0 : 1 : zipWith (+) fibonacci (tail fibonacci)  
evens = filter even naturals
```

Generarea intervalelor

[start..stop] sau [start..]

[start, next..stop] sau [start, next..] --next dă pasul

Exemple

[1..5]

[1,3..10]

[10,7..0]

[20,19.5..]

Generarea intervalelor

[start..stop] sau [start..]

[start, next..stop] sau [start, next..] --next dă pasul

Exemple

[1..5] -- [1, 2, 3, 4, 5]

[1, 3..10] -- [1, 3, 5, 7, 9]

[10, 7..0] -- [10, 7, 4, 1]

[20, 19.5..] -- lista infinită [20, 19.5, 19, 18.5...]

List comprehensions

[expr | generatori, condiții, legări locale]

Exemple

lc1 = [(x, y, z) | x<- [1..3], y<- [1..4], x<y, let z = x+y, odd z]

fibo = 0 : 1 : [x+y | (x, y) <- zip fibo (tail fibo)]

qsort [] = []

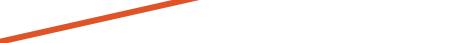
qsort (x:xs) =

 qsort [y | y<-xs, y<=x] ++

 [x] ++

 qsort [y | y<-xs, y>x]

Întâi toate rezultatele pentru x=1, apoi
toate pentru x=2, apoi toate pentru x=3



Rezumat

Perechi și liste

Funcții

. și \$

Expresii condiționale

Expresii pentru legare locală

Evaluare lenășă

List comprehensions

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții

. și \$

Expresii condiționale

Expresii pentru legare locală

Evaluare lenășă

List comprehensions

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Functii: $\lambda x y \rightarrow \text{corp}$, $f x y = \text{corp}$

. și \$

Expresii condiționale

Expresii pentru legare locală

Evaluare lenășă

List comprehensions

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Functii: \x y -> corp, f x y = corp

. și \$: compunere de funcții / aplicație de funcție

Expresii condiționale

Expresii pentru legare locală

Evaluare lenășă

List comprehensions

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Functii: $\lambda x y \rightarrow \text{corp}$, $f x y = \text{corp}$

. și \$: compunere de funcții / aplicație de funcție

Expresii condiționale: if ... then ... else ..., case ... of (... -> ...), (| ... = ...)

Expresii pentru legare locală

Evaluare lenășă

List comprehensions

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Functii: $\lambda x y \rightarrow \text{corp}$, $f x y = \text{corp}$

. și \$: compunere de funcții / aplicație de funcție

Expresii condiționale: if ... then ... else ..., case ... of (... -> ...), (| ... = ...)

Expresii pentru legare locală: let ... in ..., ... where ...

Evaluare lenășă

List comprehensions

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Functii: $\lambda x y \rightarrow \text{corp}$, $f x y = \text{corp}$

. și \$: compunere de funcții / aplicație de funcție

Expresii condiționale: if ... then ... else ..., case ... of (... -> ...), (| ... = ...)

Expresii pentru legare locală: let ... in ..., ... where ...

Evaluare lenășă: argumentele nu se evaluatează la apel, apoi se evaluatează maxim o dată

List comprehensions

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Functii: \x y -> corp, f x y = corp

. și \$: compunere de funcții / aplicație de funcție

Expresii condiționale: if ... then ... else ..., case ... of(... -> ...), (| ... = ...)

Expresii pentru legare locală: let ... in ..., ... where ...

Evaluare lenășă: argumentele nu se evaluatează la apel, apoi se evaluatează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]

... <- ... let ... = ...

PARADIGME DE PROGRAMARE

Curs 7a

Tipare tare / slabă / statică / dinamică. Tipuri și expresii de tip. Tipuri definite de utilizator.

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Expresii de tip
- Tipuri definite de utilizator

Tipare tare / slabă

- **Tipare tare:** nu se permit operații pe argumente care nu au tipul corect (se convertește tipul numai în cazul în care nu se pierde informație la conversie)

Exemplu: `1+"23"` → eroare (Racket, Haskell)

- **Tipare slabă:** nu se verifică corectitudinea tipurilor, se face cast după reguli specifice limbajului

Exemplu: `1+"23" = 24` (Visual Basic)

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

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Expresii de tip
- Tipuri definite de utilizator

Tipare statică / dinamică

- **Tipare statică:** verificarea tipurilor se face la compilare
 - atât variabilele cât și valorile au un tip asociat

Exemple: C++, Java, Haskell, ML, Scala, etc.

- **Tipare dinamică:** verificarea tipurilor se face la execuție
 - numai valorile au un tip asociat

Exemple: Python, Racket, Prolog, Javascript, etc.

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Expresii de tip
- Tipuri definite de utilizator

Tipuri primitive în Haskell

Tip

Bool = [True, False]

Char = .. 'a', 'b', ..]

Int = .. -1, 0, 1, ..]

Tipare expresie (:t expr)

True :: Bool

'a' :: Char

(fib 0) :: Int

Altele: **Integer**, **Float**, **Double**, etc.

Constructori de tip

Constructor de tip = „funcție” care creează un tip compus pe baza unor tipuri mai simple

- $(, , \dots) : MT^n \rightarrow MT$ (MT = mulțimea tipurilor)
 - $(t_1, t_2, \dots, t_n) = \text{tuplu}$ cu elemente de tipurile t_1, t_2, \dots, t_n
 - **Ex:** (Bool, Char) echivalent cu $(,) \text{ Bool Char}$
- $[] : MT \rightarrow MT$
 - $[t] = \text{listă}$ cu elemente de tip t
 - **Ex:** [Int] echivalent cu $[] \text{ Int}$
- $\rightarrow : MT^2 \rightarrow MT$
 - $t_1 \rightarrow t_2 = \text{funcție}$ de un parametru de tip t_1 care calculează valori de tip t_2
 - **Ex:** Int \rightarrow Int echivalent cu $(\rightarrow) \text{ Int Int}$

Tipul funcțiilor n-are

Exemplu: add $x \ y = x + y$ (pentru simplitate, presupunem că + merge doar pe Int)

- Având în vedere că toate funcțiile sunt curry, care este tipul lui (add 2) ?

Tipul funcțiilor n-are

Exemplu: $\text{add } x \ y = x + y$ (pentru simplitate, presupunem că + merge doar pe Int)

- Având în vedere că toate funcțiile sunt curry, care este tipul lui $(\text{add } 2)$?

$(\text{add } 2) :: \text{Int} \rightarrow \text{Int}$

- În aceste condiții, care este tipul lui add ?

Tipul funcțiilor n-are

Exemplu: $\text{add } x \ y = x + y$ (pentru simplitate, presupunem că + merge doar pe Int)

- Având în vedere că toate funcțiile sunt curry, care este tipul lui $(\text{add } 2)$?

$(\text{add } 2) :: \text{Int} \rightarrow \text{Int}$

- În aceste condiții, care este tipul lui add ?

$\text{add} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$ echivalent cu

$\text{add} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ încrucișat **-> este asociativ la dreapta**

- Cum am interpreta tipul $(\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$?

Tipul funcțiilor n-are

Exemplu: $\text{add } x \ y = x + y$ (pentru simplitate, presupunem că + merge doar pe Int)

- Având în vedere că toate funcțiile sunt curry, care este tipul lui $(\text{add } 2)$?

$(\text{add } 2) :: \text{Int} \rightarrow \text{Int}$

- În aceste condiții, care este tipul lui add ?

$\text{add} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$ echivalent cu

$\text{add} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ încrucișat **-> este asociativ la dreapta**

- Cum am interpreta tipul $(\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$?

o funcție care – primește o funcție de la Int la Int
– întoarce un Int

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- **Expresii de tip**
- Tipuri definite de utilizator

Expresii de tip

- Expresiile reprezintă valori / expresiile de tip reprezintă tipuri

Exemple: Char, Int -> Int -> Int, (Char, [Int])

- **Declararea signaturii** unei funcții (optională în Haskell) **f :: exprDeTip**

= asociere între numele funcției și o expresie de tip, cu rol de:

- **Documentare** (ce ar trebui să facă funcția)
- **Abstractizare** (surprinde cel mai general comportament al funcției, funcția percepătă ca operator al unui anumit TDA sau al unei clase de TDA-uri)
- **Verificare** (Haskell generează o eroare dacă intenția (declarația) nu se potrivește cu implementarea)

Exemplu - myMap

myMap :: (a → b) → [a] → [b] trebuie să:

primească: o funcție de la un tip oarecare a la un tip oarecare b

întoarcă: o listă de elemente de același tip oarecare a

întoarcă: o listă de elemente de același tip oarecare b

- Dacă implementăm myMap astfel:

1. myMap :: (a → b) → [a] → [b]
2. myMap f [] = []
3. myMap f (x:xs) = f (f x) : myMap f xs

compilatorul va da eroare, arătând că funcția nu se comportă conform declarației.

- Fără declarația de tip, Haskell ar fi dedus singur tipul lui myMap și ne-ar fi lăsat să continuăm cu o funcție care nu face ceea ce dorim.

Observații

Verificarea strictă a tipurilor înseamnă:

- Mai **multă siguranță** („dacă trece de compilare atunci merge”)
- Mai **puțină libertate**
 - Listele sunt neapărat omogene: **[a]**
 - Contrast cu liste ca `'(1 'a #t)` din Racket
 - Funcțiile întorc mereu valori de un același tip: **f :: ... -> b**
 - Contrast cu funcții ca `member` din Racket (care întoarce o listă sau `#f`)

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Expresii de tip
- Tipuri definite de utilizator

Tipuri definite de utilizator

- Cuvântul cheie **data** dă utilizatorului posibilitatea definirii unui TDA cu implementare completă (constructori, operatori, axiome)

```
data ConsTip = Cons1 t11 .. t1i |  
           Cons2 t21 .. t2j | ... |  
           Consn tn1 .. tnk
```

Numele constructorilor valorilor tipului și tipurile parametrilor acestora (dacă au)

Exemple

```
data RH = Pos | Neg
```

-- doar constructori nulari

```
data ABO = O | A | B | AB
```

-- doar constructori nulari

```
data BloodType = BloodType ABO RH
```

-- constructor extern

Exemplu – Tipul Natural

```
1. data Natural = Zero | Succ Natural -- constructori nular și intern
2.                               deriving Show -- face posibilă afișarea valorilor tipului
3. unu = Succ Zero
4. doi = Succ unu
5. trei = Succ doi
6.
7. addN :: Natural -> Natural -> Natural -- arată exact ca axiomele
8. addN Zero n = n
9. addN (Succ m) n = Succ (addN m n)

addN unu trei                                -- Succ (Succ (Succ (Succ Zero)))
```

Constructorii valorilor unui TDA

DUBLĂ UTILIZARE a constructorilor valorilor unui TDA

- Compon noi valori pe baza celor existente (comportament de **funcție**)

Exemple: $\text{unu} = \text{Succ Zero}$
 $\text{doi} = \text{Succ unu}$

- Descompun valori existente în scopul identificării structurii lor (comportament de **pattern**)

Exemple: $\text{addN Zero n} = n$
 $\text{addN} (\text{Succ m}) n = \text{Succ} (\text{addN m n})$

Variante de tipuri definite de utilizator

- Tipuri enumerate (tipuri sumă)
- Tipuri înregistrare (tipuri produs)
- Tipuri recursive
- Tipuri parametrizate

Tipuri enumerate

- Numite și tipuri sumă ($|$ face suma/reuniunea valorilor tipului)
- Enumeră toate valorile tipului, sub forma
data ConstTip = Val₁ | Val₂ | ... | Val_n

Exemple

```
data Dice = S1 | S2 | S3 | S4 | S5 | S6
```

```
*Main> :i Bool
```

```
data Bool = False | True           -- Defined in `GHC.Types'
```

Tipuri Înregistrare

- Numite și tipuri produs: o valoare a tipului se obține prin combinarea unor valori de alte tipuri, sub forma
data ConsTip = Cons {câmp₁ :: tip₁, ... câmp_n :: tip_n}
care este o variantă cu funcții selector pentru definiția
data ConsTip = Cons tip₁ ... tip_n
- Au un corespondent în majoritatea limbajelor de programare (ex: struct în C++)

Exemplu

```
data Person = Person {name :: (String, String), age :: Int}  
fc :: Person  
fc = Person ("Frederic", "Chopin") 211  
composer = name fc
```

Tipuri recursive

- Tipuri pentru care specificăm și cel puțin un constructor intern
`data ConstTip = .. | Consi .. ConstTip .. | ..`

Exemple

```
data Natural = Zero | Succ Natural deriving Show
```

```
data IntList = Nil | Cons Int IntList deriving Show
```

Tipuri parametrizate – în general

- **Constructorii de valori** ale tipului (ex: `:`, `Succ`)
 - Pot primi valori ca argumente pentru a produce noi valori

Exemplu: `unu = Succ Zero`
`lista_unu = 1 : []`

- **Constructorii de tip** (ex: `[]`, `(,)`, `->`)
 - Pot primi tipuri ca argumente pentru a produce noi tipuri

Exemplu: `[Int]`, `[Char]`, `[[Char]]`
`(Int, Char)`, `([Char], Int, Int)`

- Când TDA-ul sau funcțiile noastre se comportă la fel indiferent de tipul valorilor pe care le manipulează, folosim variabile (parametri) de tip

Exemplu: `[a]` - o listă cu elemente de un tip oarecare a
`(a, b)` - o pereche de un element de un tip oarecare a
și un altul de un tip oarecare b

Tipuri parametrizate – definite de utilizator

- Constructorul de tip este aplicat pe una sau mai multe variabile de tip, permitând obținerea unor tipuri particulare la instanțiere
`data ConstTip a b ... =`
- Nu are rost să avem `IntList`, `CharList`, `PairList`, etc., este de preferat să avem un singur tip parametrizat `List a`, unde `a` se va lega la un tip concret în momentul în care plasăm valori de un tip concret în listă

Exemplu

```
data List a = Nil | Cons a (List a) deriving Show
lst1 = Cons 1 $ Cons 2.5 $ Cons 4 Nil      -- :t lst1 => lst1 :: List Double
lst2 = Cons "Hello " $ Cons "world!" Nil   -- :t lst2 => lst2 :: List [Char]
```

Exemplu – Tipul (Maybe a)

Tipul **(Maybe a)** există în Haskell și este definit astfel:

```
data Maybe a = Nothing | Just a
```

Constructor de tip Parametru de tip Constructori de valori ale tipului

- Se folosește pentru situații când funcția întoarce sau nu un rezultat (de exemplu pentru funcții de căutare care ar putea să găsească sau nu ceea ce caută)
 - În funcție de ce tip de valoare va stoca acest tip de date atunci când are ce stoca, constructorul de tip va produce un `Maybe Int` sau un `Maybe Char`, etc.

Exemplu de instanțiere (Maybe a)

Să se găsească suma pară maximă dintre sumele elementelor listelor unei liste de liste, dacă există (ex: `findMaxEvenSum [[1,2,3,4,5], [2,2], [2,4]] = Just 6`).

```
1. --findMaxEvenSum :: [[Int]] -> Maybe Int
2. findMaxEvenSum [] = Nothing
3. findMaxEvenSum (l:ls)
4.   | even lsum = case findMaxEvenSum ls of
5.     Just s -> Just (max lsum s)           Din cauză că sumL este declarat ca
6.     _ -> Just lsum                      sumL :: [Int] -> Int
7.   | otherwise = findMaxEvenSum ls          funcția va întoarce un (Maybe Int)
8.   where lsum = sumL l
```

Construcția **type**

- Se folosește pentru a crea **sinonime de tip**, în scop de:
 - **Documentare**: este mai clar ce face o variabilă de tip Age decât o variabilă de tip Int
 - **Concizie**: este mai scurt (și mai clar) Name decât (String, String)

Exemple

```
type Age = Int
```

```
type Name = (String, String)
```

```
names :: [Name]
```

```
names = [ ("Frederic", "Chopin"), ("Antonio", "Vivaldi"), ("Maurice", "Ravel") ]
```

Construcția `newtype`

- Se folosește pentru a crea **tipuri noi** (definite de utilizator) folosind **un singur constructor cu un singur parametru**
- Mai **eficient** decât **data**:
 - Pentru valorile tipurilor definite cu **data** trebuie să se facă pattern match pe constructori și apoi să se acceseze valorile închise în aceștia
 - Pentru valorile tipurilor definite cu **newtype**, existând un singur constructor, acesta este șters încă din faza de compilare, și înlocuit cu valoarea închisă în el (care se știe ce tip are)
- Util pentru a defini apoi operații pe tipul respectiv

Exemplu

```
newtype Person2 = Person2 (Name, Age) deriving Show  
fc2 :: Person2  
fc2 = Person2 (("Frederic", "Chopin"), 211)
```

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional		
Funcții		
Pattern matching		
Legare		
Evaluare		
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții		
Pattern matching		
Legare		
Evaluare		
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții	Automat uncurry	Automat curry
Pattern matching		
Legare		
Evaluare		
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții	Automat uncurry	Automat curry
Pattern matching	Nu	Da
Legare		
Evaluare		
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții	Automat uncurry	Automat curry
Pattern matching	Nu	Da
Legare	Locală – statică, top-level - dinamică	Statică
Evaluare		
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții	Automat uncurry	Automat curry
Pattern matching	Nu	Da
Legare	Locală – statică, top-level - dinamică	Statică
Evaluare	Aplicativă	Leneșă
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții	Automat uncurry	Automat curry
Pattern matching	Nu	Da
Legare	Locală – statică, top-level - dinamică	Statică
Evaluare	Aplicativă	Leneșă
Tipare	Tare, dinamică	Tare, statică

Rezumat

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Declararea signaturii

Construcția „data”

Tipuri enumerate

Tipuri înregistrare

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicate de tip

Tipare statică/dinamică

Constructori de tip

Declararea signaturii

Construcția „data”

Tipuri enumerate

Tipuri înregistrare

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicate de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip

Declararea signaturii

Construcția „data”

Tipuri enumerate

Tipuri înregistrare

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicate de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,,), [], ->, tipurile definite cu „data”

Declararea signaturii

Construcția „data”

Tipuri enumerate

Tipuri înregistrare

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicate de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,,), [], ->, tipurile definite cu „data”

Declararea signaturii: f :: exprTip

Construcția „data”

Tipuri enumerate

Tipuri înregistrare

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,,), [], ->, tipurile definite cu „data”

Declararea signaturii: f :: exprTip

Construcția „data”: data ConsTip = Cons₁ t₁₁ .. t_{1i} | ... | Cons_n t_{n1} .. t_{nk}

Tipuri enumerate

Tipuri înregistrare

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,,), [], ->, tipurile definite cu „data”

Declararea signaturii: $f :: \text{exprTip}$

Construcția „data”: $\text{data ConsTip} = \text{Cons}_1 t_{11} \dots t_{1i} | \dots | \text{Cons}_n t_{n1} \dots t_{nk}$

Tipuri enumerate: $\text{data ConsTip} = \text{Val}_1 | \text{Val}_2 | \dots | \text{Val}_n$

Tipuri înregistrare

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,,), [], ->, tipurile definite cu „data”

Declararea signaturii: f :: exprTip

Construcția „data”: data ConsTip = Cons₁ t₁₁ .. t_{1i} | ... | Cons_n t_{n1} .. t_{nk}

Tipuri enumerate: data ConsTip = Val₁ | Val₂ | ... | Val_n

Tipuri înregistrare: data ConsTip = Cons {câmp₁ :: tip₁, ... câmp_n :: tip_n}

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,,), [], ->, tipurile definite cu „data”

Declararea signaturii: $f :: \text{exprTip}$

Construcția „data”: data ConsTip = Cons₁ t₁₁ .. t_{1i} | ... | Cons_n t_{n1} .. t_{nk}

Tipuri enumerate: data ConsTip = Val₁ | Val₂ | ... | Val_n

Tipuri înregistrare: data ConsTip = Cons {câmp₁ :: tip₁, ... câmp_n :: tip_n}

Tipuri recursive: data ConsTip = ... | Cons_i..ConsTip .. | ...

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicate de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,,), [], ->, tipurile definite cu „data”

Declararea signaturii: f :: exprTip

Construcția „data”: data ConsTip = Cons₁ t₁₁ .. t_{1i} | ... | Cons_n t_{n1} .. t_{nk}

Tipuri enumerate: data ConsTip = Val₁ | Val₂ | ... | Val_n

Tipuri înregistrare: data ConsTip = Cons {câmp₁ :: tip₁, ... câmp_n :: tip_n}

Tipuri recursive: data ConsTip = ... | Cons_i.. ConsTip .. | ...

Tipuri parametrizate: (a,b), [a], a -> b, data ConsTip a b ...

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,,), [], ->, tipurile definite cu „data”

Declararea signaturii: $f :: \text{exprTip}$

Construcția „data”: data ConsTip = Cons₁ t₁₁ .. t_{1i} | ... | Cons_n t_{n1} .. t_{nk}

Tipuri enumerate: data ConsTip = Val₁ | Val₂ | ... | Val_n

Tipuri înregistrare: data ConsTip = Cons {câmp₁ :: tip₁, ... câmp_n :: tip_n}

Tipuri recursive: data ConsTip = ... | Cons_i..ConsTip .. | ...

Tipuri parametrizate: (a,b), [a], a -> b, data ConsTip a b ...

Construcția „type”: creează sinonime de tip (nu noi tipuri)

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,,), [], ->, tipurile definite cu „data”

Declararea signaturii: f :: exprTip

Construcția „data”: data ConsTip = Cons₁ t₁₁ .. t_{1i} | ... | Cons_n t_{n1} .. t_{nk}

Tipuri enumerate: data ConsTip = Val₁ | Val₂ | ... | Val_n

Tipuri înregistrare: data ConsTip = Cons {câmp₁ :: tip₁, ... câmp_n :: tip_n}

Tipuri recursive: data ConsTip = ... | Cons_i.. ConsTip .. | ...

Tipuri parametrizate: (a,b), [a], a -> b, data ConsTip a b ...

Construcția „type”: creează sinonime de tip (nu noi tipuri)

Construcția „newtype”: data ConsTip = Cons₁ t₁ (un singur constructor cu un singur parametru)

PARADIGME DE PROGRAMARE

Curs 7b

Sinteză de tip.

Sinteză de tip – Cuprins

- Sistem de tipuri
- Sinteză de tip
- Siguranță de tip
- Unificare
- Exemple
- Mesaje de eroare

Sistem de tipuri

Sistem de tipuri = set de mecanisme și reguli valabile într-un limbaj de programare, privind organizarea, construirea, manevrarea tipurilor de date acceptate în limbaj

Atribuții (nu se ocupă neapărat de toate)

- Asocierea de tipuri constructelor din limbaj
- Definirea de noi tipuri
- Decizia asupra echivalenței / compatibilității tipurilor
- Sinteza (inferența) de tip

Avantaje și limitări

Avantaje

- Detectare timpurie a
 - Erorilor (+ mai mare acuratețe în identificarea sursei)
 - Codului inutil sau invalid
- Declarații de tip – abstractizare, documentare (care rămâne consistentă cu modificările efectuate ulterior)
- Eficiență (se pot optimiza operații pentru anumite tipuri)

Limitări

- Rejectează cod care nu ar produce probleme
- Renunță la anumite facilități (ex: Haskell nu permite liste eterogene)
- Tiparea statică necesită ori adnotări (declarații) de tip, ori sinteză (inferență) de tip

Sinteză de tip – Cuprins

- Sistem de tipuri
- Sinteză de tip
- Siguranță de tip
- Unificare
- Exemple
- Mesaje de eroare

Sinteză (inferență) de tip

- Tipul unei expresii este sintetizat în funcție de
 - **Tipul componentelor** expresiei (ex: tipul expresiei $2.5 + 2$)
 - **Contextul lexical** al expresiei (ex: tipul lui x în contextul $['a' .. x]$)
- Reprezentarea tipului – expresie de tip, care reprezintă o combinație de
 - **Constante de tip** (tipurile primitive: Bool, Char, Int, etc.)
 - **Variabile de tip** (cu semnificația „orice (expresie de) tip”)
 - **Constructori de tip** ((,), [], ->, constructori definiți cu „data”)

Sinteză de tip = determinarea automată a tipului unei expresii, prin combinarea constantelor, variabilelor și constructorilor de tip după reguli bine stabilite (reguli de sinteză de tip)

Reguli de sinteză de tip

Relație de tipare: $e :: T$

- Se citește „expresia e are tipul T”
- O expresie e este **bine tipată** \Leftrightarrow există un tip T astfel încât $e :: T$

Regulă de sinteză de tip

P_1, P_2, \dots, P_n

P_i = premise (0 sau mai multe relații de tipare)

(id_regulă)
C

C = concluzie (o relație de tipare)

Reguli de sinteză de tip – Exemple

$$\frac{}{\text{True}, \text{False} :: \text{Bool}} (TBool)$$

$$\frac{expr1 :: \text{Bool} \quad expr2 :: a \quad expr3 :: a}{\text{if } expr1 \text{ then } expr2 \text{ else } expr3 :: a} (TIf)$$

$$\frac{var :: a \quad expr :: b}{\lambda var \rightarrow expr :: a \rightarrow b} (TAbs)$$

$$\frac{fun :: a \rightarrow b \quad expr :: a}{fun \ expr :: b} (TApp)$$

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

$$\frac{expr1 :: \text{Int} \quad expr2 :: \text{Int}}{expr1 + expr2 :: \text{Int}} (T+)$$

funcții

valori booleene

întregi

Observații

- **Tipul unei expresii este unic**
- Regulile de tipare funcționează în ambele sensuri (vom observa că în sinteza de tip pornim de la concluzie către premise)
 - De la **premise către concluzie**
Exemplu: Dacă $\text{expr1} :: \text{Int}$ și $\text{expr2} :: \text{Int}$, atunci $\text{expr1} + \text{expr2} :: \text{Int}$
 - De la **concluzie către premise**
Exemplu: Dacă $\text{expr1} + \text{expr2} :: \text{Int}$, atunci $\text{expr1} :: \text{Int}$ și $\text{expr2} :: \text{Int}$

Algoritm de sinteză de tip

- **Identifică regula aplicabilă**, prin pattern match între
 - Expresia care trebuie tipată
 - Concluziile regulilor de sinteză de tip
- **Aplică regula**, obținând constrângerile asupra componentelor expresiei
- **Pentru fiecare componentă în parte, reia algoritmul**, până când nu mai este nimic de descompus în părți componente
- **Tipul expresiei rezultă din aplicarea tuturor constrângerilor** obținute la pașii anteriori

Sinteză de tip – Exemple

Care este tipul funcției $f: fg = g \cdot 3 + 1 \Leftrightarrow f = \lambda g \rightarrow g \cdot 3 + 1$

Constrângeri

$$\frac{var :: a \quad expr :: b}{\lambda var \rightarrow expr :: a \rightarrow b} (TAbs)$$

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

$$\frac{expr1 :: Int \quad expr2 :: Int}{expr1 + expr2 :: Int} (T+)$$

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

$b = \text{Int}$

$$\frac{fun :: a \rightarrow b \quad expr :: a}{fun \ expr :: b} (TApp)$$

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

$a = c \rightarrow \text{Int}$

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

$$\frac{}{1, 3 :: \text{Int}} (TInt)$$

$c = \text{Int}$

Așadar $f :: a \rightarrow b \Leftrightarrow f :: (c \rightarrow \text{Int}) \rightarrow \text{Int} \Leftrightarrow f :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$

Sinteză de tip – Exemple

Care este tipul funcției fix: $\text{fix } f = f(\text{fix } f) \Leftrightarrow \text{fix} = \lambda f \rightarrow f(\text{fix } f)$

Constrângeri

$$\frac{\text{var} :: a \quad \text{expr} :: b}{\lambda \text{var} \rightarrow \text{expr} :: a \rightarrow b} \quad (\text{TAbs})$$

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

$$\frac{\text{fun} :: a \rightarrow b \quad \text{expr} :: a}{\text{fun expr} :: b} \quad (\text{TApp})$$

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

$$\frac{\text{fun} :: a \rightarrow b \quad \text{expr} :: a}{\text{fun expr} :: b} \quad (\text{TApp})$$

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

$a = c \rightarrow b$

$a = d, b = c$

Așadar $\text{fix} :: a \rightarrow b \Leftrightarrow \text{fix} :: (c \rightarrow b) \rightarrow b \Leftrightarrow \text{fix} :: (b \rightarrow b) \rightarrow b$

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

(++)

[a] -> [b] -> [c]

zipWith (ex: zipWith (+) [1,2] [3,4] = [4,6])

(a -> a -> a) -> [a] -> [a] -> [a]

unzip (ex: unzip [(1,2), (3,4)] = ([1,3], [2,4]))

[(a,b)] -> ([a], [b])

[take, drop]

[Int -> [a] -> [a]]

words (ex: words "wait for it" = ["wait", "for", "it"])

String -> String

(odd . fst)

(Int, b) -> Bool

map ("o1" :) . reverse

[[Char]] -> [[Char]]

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

(++)

[a] -> [b] -> [c]

zipWith (ex: zipWith (+) [1,2] [3,4] = [4,6])

(a -> a -> a) -> [a] -> [a] -> [a]

Nu, nu pot concatena liste cu elemente de tipuri diferite!

Ex: [1,2,3] ++ [(+),(-)]

Corect: [a] -> [a] -> [a]

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

zipWith (ex: `zipWith (+) [1,2] [3,4] = [4,6]`) $(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow [a] \rightarrow [a]$

unzip (ex: `unzip [(1,2), (3,4)] = ([1,3], [2,4])`) $[(a,b)] \rightarrow ([a], [b])$

Nu, nu e necesar ca cele două liste să aibă elemente de același tip.

Ex: `zipWith (:) [1,2] [[3],[4]]`

Corect: $(a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

<code>unzip</code> (ex: <code>unzip [(1,2), (3,4)] = ([1,3], [2,4])</code>)	<code>[(a,b)] -> ([a], [b])</code>
<code>[take, drop]</code>	<code>[Int -> [a] -> [a]]</code>

Ex: `unzip [(1,'a'), (2,'b'), (3,'c')]`

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

[take, drop]

[Int -> [a] -> [a]]

words (ex: words "wait for it" = ["wait", "for", "it"])

String -> String

Da, atenție la faptul că signatura indică o listă de funcții de același tip.

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă semnatura din dreapta?

Nu, rezultatul este o listă de String-uri, nu o listă de caractere (String ~ [Char]).

Corect: String -> [String]

Alternativ, se putea scrie: words :: [Char] -> [[Char]]

words (ex: words "wait for it" = ["wait", "for", "it"]) String -> String

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

Da:

fst :: (a,b) -> a odd :: Integral a => a -> Bool

Ex: (odd . fst) (1, [])

(odd . fst)

(Int, b) -> Bool

map ("o1" :) . reverse

[[Char]] -> [[Char]]

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

Nu, fiecare element din listă trebuie să fie nu un String, ci o listă de String-uri.

"o1" : element înseamnă că "o1" are tipul a și element are tipul [a].

Ex: (map ("o1":) . reverse) [[["So", "many"], ["Strings"]]]

Corect: [[String]] -> [[String]] sau [[[Char]]] -> [[[Char]]]

map ("o1" :) . reverse

[[Char]] -> [[Char]]

Sinteză de tip – Cuprins

- Sistem de tipuri
- Sinteză de tip
- **Siguranță de tip**
- Unificare
- Exemple
- Mesaje de eroare

Siguranță de tip

Există **siguranță de tip** atunci când expresiile bine tipate „nu o pot lua pe căi greșite” în timpul evaluării, adică atunci când se respectă proprietățile:

- **Progres**
 - O expresie bine tipată e :: T nu se blochează:
 - Ori este o valoare
 - Ori se poate reduce conform unei reguli de evaluare
- **Conservare**
 - Un pas de evaluare asupra unei expresii bine tipate e :: T produce o expresie bine tipată e' :: T

Scop: prevenirea erorilor cauzate de discrepanțele dintre tipurile așteptate și cele procesate.

Sinteză de tip – Cuprins

- Sistem de tipuri
- Sinteză de tip
- Siguranță de tip
- **Unificare**
- Exemple
- Mesaje de eroare

Unificare

Unificarea a 2 expresii de tip = găsirea celei mai generale substituții

$$S = \{e_1/t_1, e_2/t_2, \dots, e_n/t_n\}$$

pentru variabilele de tip t_i din cele 2 expresii astfel încât, în urma substituției, cele 2 expresii de tip devin una și aceeași

- Se folosește în sinteza de tip în pasul de **aplicare a constrângerilor**

Exemplu

- Din sinteza de tip avem constrângerile $a = b \rightarrow [d]$ și $a = \text{Int} \rightarrow c$
→ $b \rightarrow [d]$ trebuie să unifice cu $\text{Int} \rightarrow c$, altfel sinteza de tip va eşua
- Există $S = \{\text{Int} / b, [d] / c\}$ a.î. $b \rightarrow [d]$ și $\text{Int} \rightarrow c$ se intersectează în $\text{Int} \rightarrow [d]$
- Există și $S' = \{\text{Int} / b, [\text{Int}] / c, \text{Int} / d\}$, rezultând în $\text{Int} \rightarrow [\text{Int}]$, dar S este **mai generală!**

Mai multe exemple

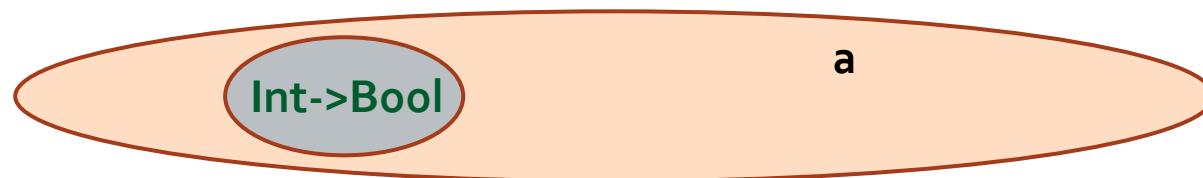
- $e_1 = a, e_2 = \text{Int} \rightarrow \text{Bool}$
- $e_1 = (a, [a]), e_2 = ([b], c)$
- $e_1 = \text{Int}, e_2 = \text{Float}$

Mai multe exemple

- $e_1 = a, e_2 = \text{Int} \rightarrow \text{Bool}$

$$S = \{\text{Int} \rightarrow \text{Bool} / a\}$$

$$e_{\text{unificată}} = \text{Int} \rightarrow \text{Bool}$$



- $e_1 = (a, [a]), e_2 = ([b], c)$

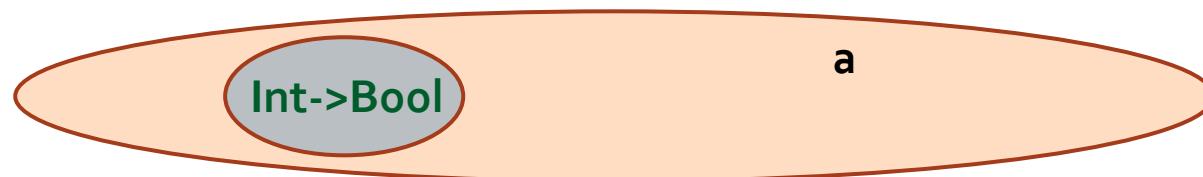
- $e_1 = \text{Int}, e_2 = \text{Float}$

Mai multe exemple

- $e_1 = a, e_2 = \text{Int} \rightarrow \text{Bool}$

$$S = \{\text{Int} \rightarrow \text{Bool} / a\}$$

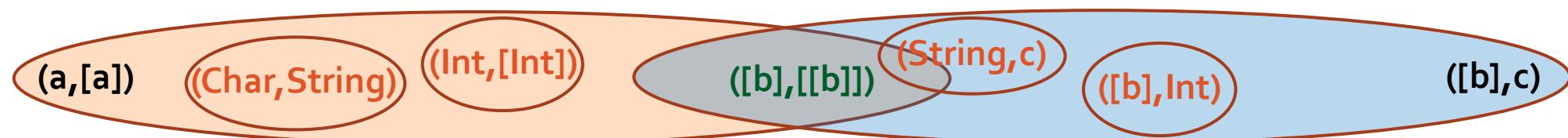
$$e_{\text{unificată}} = \text{Int} \rightarrow \text{Bool}$$



- $e_1 = (a, [a]), e_2 = ([b], c)$

$$S = \{[b] / a, [[b]] / c\}$$

$$e_{\text{unificată}} = ([b], [[b]])$$



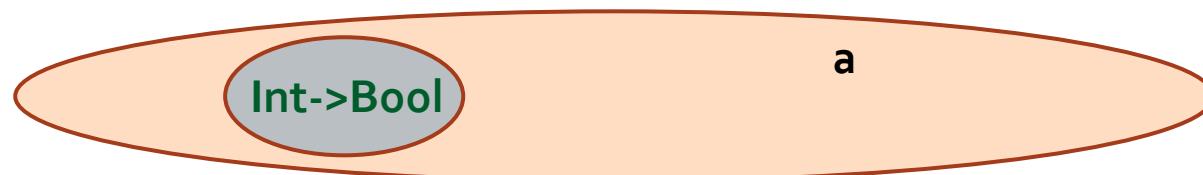
- $e_1 = \text{Int}, e_2 = \text{Float}$

Mai multe exemple

- $e_1 = a, e_2 = \text{Int} \rightarrow \text{Bool}$

$$S = \{\text{Int} \rightarrow \text{Bool} / a\}$$

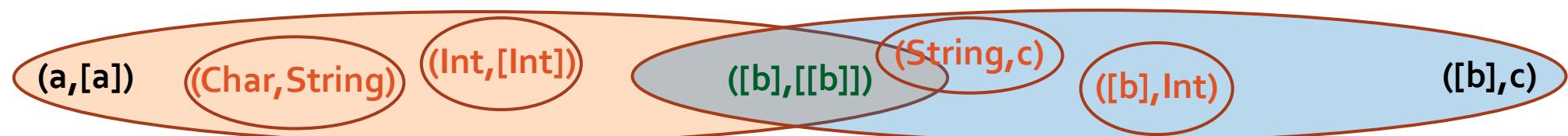
$$e_{\text{unificată}} = \text{Int} \rightarrow \text{Bool}$$



- $e_1 = (a, [a]), e_2 = ([b], c)$

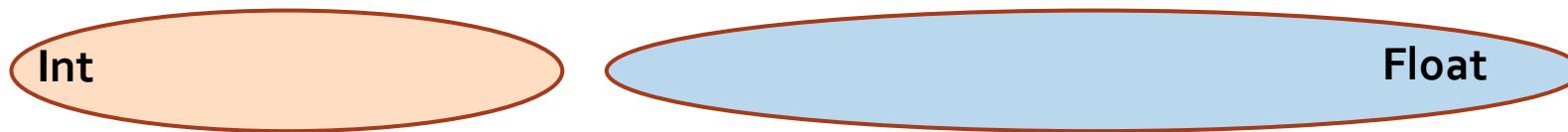
$$S = \{[b] / a, [[b]] / c\}$$

$$e_{\text{unificată}} = ([b], [[b]])$$



- $e_1 = \text{Int}, e_2 = \text{Float}$

Nu unifică (nu se fac conversii implicite)!



Unificare la sinteza de tip – Exemplu

Care este tipul funcției f: $f x = x x \Leftrightarrow f = \lambda x \rightarrow x x$

Constrângeri

$$\frac{var :: a \quad expr :: b}{\lambda var \rightarrow expr :: a \rightarrow b} (TAbs)$$

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

$$\frac{fun :: a \rightarrow b \quad expr :: a}{fun\ expr :: b} (TApp)$$

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

$a = c \rightarrow b = c$

a unifică cu $c \rightarrow b$, dar $c \rightarrow b$ nu unifică cu c.

$(c = c \rightarrow b = c \rightarrow b \rightarrow b = c \rightarrow b \rightarrow b \rightarrow b \dots$ rezultă un tip infinit, care nu poate fi sintetizat)

Întrucât unificarea eșuează (găsește o inconsistență), definiția lui f generează eroare:
Occurs check: cannot construct the infinite type: $t_1 \sim t_1 \rightarrow t$

Reguli folosite în algoritmul de unificare

- O **variabilă de tip** a unifică cu o **expresie de tip** e ($S = \{e / a\}$) dacă și numai dacă
 - $e = a$ sau
 - e nu conține a (occurs check – pentru a evita tipuri infinite)
- **2 constante de tip** unifică ($S = \{\}$) dacă și numai dacă sunt egale
- **2 construcții de tip** unifică ($S = S_1 \cup \dots \cup S_n$) dacă și numai dacă sunt aplicații ale aceluiași constructor de tip asupra către n argumente care unifică recursiv (prin substituțiile $S_1 \dots S_n$)

Unificare la sinteza de tip – Implementare

- Algoritmul de sinteză de tip pe bază de unificare folosește
 - O **stivă de constrângeri** de forma $e_1 = e_2$ – se umple în timpul algoritmului de sinteză de tip
 - O **substituție S** (multime de asocieri între variabile de tip și expresii de tip) – inițial vidă
- La fiecare iterație a pasului de unificare
 - Se extrage o constrângere $e_1 = e_2$ din stivă
 - Dacă e_1 unifică cu e_2 prin substituția S_{12} , se aplică S_{12}
 - asupra stivei și
 - asupra substituției S
 - Altfel, unificarea eşuează → sinteza de tip eşuează → eroare de tip

Exemplu

Stiva	Substituția
<pre>a = b -> c a = d -> d b = Int</pre>	
<pre>b -> c = d -> d b = Int</pre>	<pre>a = b -> c</pre>

Exemplu

Stiva	Substituția
<pre>a = b -> c a = d -> d b = Int</pre>	
<pre>b -> c = d -> d b = Int</pre>	<pre>a = b -> c</pre>
<pre>b = d c = d b = Int</pre>	<pre>a = b -> c</pre>

Exemplu

Stiva	Substituția
<pre>a = b -> c a = d -> d b = Int</pre>	
<pre>b -> c = d -> d b = Int</pre>	<pre>a = b -> c</pre>
<pre>b = d c = d b = Int</pre>	<pre>a = b -> c</pre>
<pre>c = d d = Int</pre>	<pre>a = d -> c b = d</pre>

Exemplu

Stiva	Substituția
<pre>a = b -> c a = d -> d b = Int</pre>	
<pre>b -> c = d -> d b = Int</pre>	<pre>a = b -> c</pre>
<pre>b = d c = d b = Int</pre>	<pre>a = b -> c</pre>
<pre>c = d d = Int</pre>	<pre>a = d -> c b = d</pre>
<pre>d = Int</pre>	<pre>a = d -> d b = d c = d</pre>

Exemplu

Stiva	Substituția
<pre>a = b -> c a = d -> d b = Int</pre>	
<pre>b -> c = d -> d b = Int</pre>	<pre>a = b -> c</pre>
<pre>b = d c = d b = Int</pre>	<pre>a = b -> c</pre>
<pre>c = d d = Int</pre>	<pre>a = d -> c b = d</pre>
<pre>d = Int</pre>	<pre>a = d -> d b = d c = d</pre>
	<pre>a = Int -> Int b = Int c = Int d = Int</pre>

Sinteză de tip – Cuprins

- Sistem de tipuri
- Sinteză de tip
- Siguranță de tip
- Unificare
- Exemple
- Mesaje de eroare

Determinarea tipului expresiilor / funcțiilor

length ([]++[1,2]) + length ([]++"abcd")

Constrângerile

$(T+) \rightarrow \text{expr} :: \text{Int} \Leftrightarrow \text{length} ([]++[1,2]) :: \text{Int}, \text{length} ([]++"abcd") :: \text{Int}$

$(T\text{Len}) \rightarrow []++[1,2] :: [a], []++"abcd" :: [b]$

$(T++) \rightarrow [] :: [a], [1,2] :: [a], [] :: [b], "abcd" :: [b]$

$(T[]) \rightarrow 1,2 :: a, 'a','b','c','d' :: b$

$(T\text{Int}, T\text{Char}) \rightarrow 1,2 :: \text{Int}, 'a','b','c','d' :: \text{Char}$ **a = Int, b = Char**

$[] :: [\text{Int}]$ și $[] :: [\text{Char}]$ ok ($[]$ e polimorfic – cele 2 liste vide nu reprezintă aceeași valoare)

Etapa de unificare reușește și ea (nu avem două constrângerile asupra aceleiasi variabile), deci expresia are **tipul Int conform T+**.

Determinarea tipului expresiilor / funcțiilor

`length(xs++[1,2]) + length(xs++"abcd")`

Constrângeri

Determinarea tipului expresiilor / funcțiilor

length (xs++[1,2]) + length (xs++"abcd")

Constrângeri

$(T+) \rightarrow \text{expr} :: \text{Int} \Leftrightarrow \text{length} (\text{xs++}[1,2]) :: \text{Int}, \text{length} (\text{xs++}"abcd") :: \text{Int}$

Determinarea tipului expresiilor / funcțiilor

length (xs++[1,2]) + length (xs++"abcd")

Constrângeri

(T+) → expr :: Int ⇔ length (xs++[1,2]) :: Int, length (xs++"abcd") :: Int

(TLen) → xs++[1,2] :: [a], xs++"abcd" :: [b]

Determinarea tipului expresiilor / funcțiilor

length (xs++[1,2]) + length (xs++"abcd")

Constrângeri

$(T+) \rightarrow \text{expr} :: \text{Int} \Leftrightarrow \text{length} (\text{xs++}[1,2]) :: \text{Int}, \text{length} (\text{xs++}"abcd") :: \text{Int}$

$(T\text{Len}) \rightarrow \text{xs++}[1,2] :: [a], \text{xs++}"abcd" :: [b]$

$(T++) \rightarrow \text{xs} :: [a], [1,2] :: [a], \text{xs} :: [b], \text{"abcd"} :: [b]$

[a] = [b]

Determinarea tipului expresiilor / funcțiilor

length (xs++[1,2]) + length (xs++"abcd")

Constrângeri

(T+) → expr :: Int \Leftrightarrow **length (xs++[1,2]) :: Int, length (xs++"abcd") :: Int**

(TLen) → xs++[1,2] :: [a], xs++"abcd" :: [b]

(T++) → xs :: [a], [1,2] :: [a], xs :: [b], "abcd" :: [b] **[a] = [b]**

(T[]) → 1,2 :: a, 'a','b','c','d' :: b

Determinarea tipului expresiilor / funcțiilor

length (xs++[1,2]) + length (xs++"abcd") Constrângeri

(T+) → expr :: Int \Leftrightarrow length (xs++[1,2]) :: Int, length (xs++"abcd") :: Int

(TLen) → xs++[1,2] :: [a], xs++"abcd" :: [b]

(T++) → xs :: [a], [1,2] :: [a], xs :: [b], "abcd" :: [b] [a] = [b]

(T[]) → 1,2 :: a, 'a','b','c','d' :: b

(TInt, TChar) → 1,2 :: Int, 'a','b','c','d' :: Char a = Int, b = Char

Determinarea tipului expresiilor / funcțiilor

length (xs++[1,2]) + length (xs++"abcd")

Constrângeri

(T+) → expr :: Int \Leftrightarrow **length (xs++[1,2]) :: Int, length (xs++"abcd") :: Int**

(TLen) → xs++[1,2] :: [a], xs++"abcd" :: [b]

(T++) → xs :: [a], [1,2] :: [a], xs :: [b], "abcd" :: [b]

[a] = [b]

(T[]) → 1,2 :: a, 'a','b','c','d' :: b

(TInt, TChar) → 1,2 :: Int, 'a','b','c','d' :: Char

a = Int, b = Char

În etapa de unificare: **[Int] = [Char] → Int = Char** care nu unifică, deci tipul expresiei nu poate fi sintetizat și vom obține o eroare de tip.

Determinarea tipului expresiilor / funcțiilor

curry id

Constrângeri

Determinarea tipului expresiilor / funcțiilor

curry id

(TApp) → expr :: b \Leftrightarrow curry :: a \rightarrow b, id :: a

Constrângeri

Determinarea tipului expresiilor / funcțiilor

curry id

$(TApp) \rightarrow \text{expr} :: b \Leftrightarrow \text{curry} :: a \rightarrow b, \text{id} :: a$

$$\frac{}{\text{curry} :: ((c, d) \rightarrow e) \rightarrow c \rightarrow d \rightarrow e} (TCurry)$$

Constrângeri

$a = (c, d) \rightarrow e, b = c \rightarrow d \rightarrow e$

Determinarea tipului expresiilor / funcțiilor

curry id

$(TApp) \rightarrow \text{expr} :: b \Leftrightarrow \text{curry} :: a \rightarrow b, \text{id} :: a$

$$\frac{}{\text{curry} :: ((c, d) \rightarrow e) \rightarrow c \rightarrow d \rightarrow e} (TCurry)$$
$$\frac{}{\text{id} :: f \rightarrow f} (TId)$$

Constrângeri

$a = (c, d) \rightarrow e, b = c \rightarrow d \rightarrow e$

$a = f \rightarrow f$

Determinarea tipului expresiilor / funcțiilor

curry id

$(TApp) \rightarrow \text{expr} :: b \Leftrightarrow \text{curry} :: a \rightarrow b, \text{id} :: a$

$$\frac{}{\text{curry} :: ((c, d) \rightarrow e) \rightarrow c \rightarrow d \rightarrow e} (TCurry)$$
$$\frac{}{\text{id} :: f \rightarrow f} (TId)$$

Constrângeri

$a = (c, d) \rightarrow e, b = c \rightarrow d \rightarrow e$

$a = f \rightarrow f$

În etapa de unificare: $a = (c, d) \rightarrow e = f \rightarrow f \rightarrow e = f = (c, d) \rightarrow b = c \rightarrow d \rightarrow (c, d)$, deci expresia are tipul $c \rightarrow d \rightarrow (c, d)$ conform TApp.

Determinarea tipului expresiilor / funcțiilor

uncurry id

Constrângeri

Determinarea tipului expresiilor / funcțiilor

uncurry id

(TApp) → expr :: b \Leftrightarrow uncurry :: a \rightarrow b, id :: a

Constrângeri

Determinarea tipului expresiilor / funcțiilor

uncurry id

(TApp) \rightarrow expr :: b \Leftrightarrow uncurry :: a \rightarrow b, id :: a

$$\frac{}{\text{uncurry} :: (c \rightarrow d \rightarrow e) \rightarrow (c, d) \rightarrow e} \quad (TUncurry)$$

Constrângeri

a = c \rightarrow d \rightarrow e, b = (c, d) \rightarrow e

Determinarea tipului expresiilor / funcțiilor

uncurry id

$(TApp) \rightarrow \text{expr} :: b \Leftrightarrow \text{uncurry} :: a \rightarrow b, \text{id} :: a$

$$\frac{}{\text{uncurry} :: (c \rightarrow d \rightarrow e) \rightarrow (c, d) \rightarrow e} (TUncurry)$$
$$\frac{}{\text{id} :: f \rightarrow f} (TId)$$

Constrângeri

$a = c \rightarrow d \rightarrow e, b = (c, d) \rightarrow e$

$a = f \rightarrow f$

Determinarea tipului expresiilor / funcțiilor

uncurry id

$(TApp) \rightarrow \text{expr} :: b \Leftrightarrow \text{uncurry} :: a \rightarrow b, \text{id} :: a$

$$\frac{}{\text{uncurry} :: (c \rightarrow d \rightarrow e) \rightarrow (c, d) \rightarrow e} (TUncurry)$$
$$\frac{}{\text{id} :: f \rightarrow f} (TId)$$

Constrângeri

$a = c \rightarrow d \rightarrow e, b = (c, d) \rightarrow e$

$a = f \rightarrow f$

În etapa de unificare: $a = c \rightarrow d \rightarrow e = f \rightarrow f \rightarrow c = f = d \rightarrow e \rightarrow b = (d \rightarrow e, d) \rightarrow e$, deci expresia are tipul $(d \rightarrow e, d) \rightarrow e$ conform TApp.

Determinarea tipului expresiilor / funcțiilor

curry (curry id)

Constrângeri

Determinarea tipului expresiilor / funcțiilor

curry (curry id)

Constrângeri

(TApp) \rightarrow expr :: b \Leftrightarrow curry :: a \rightarrow b, curry id :: a

Determinarea tipului expresiilor / funcțiilor

curry (curry id)

(TApp) \rightarrow expr :: b \Leftrightarrow curry :: a \rightarrow b, curry id :: a

(ExAnterior) \rightarrow curry id :: f \rightarrow g \rightarrow (f,g)

Constrângeri

a = f \rightarrow g \rightarrow (f,g)

Determinarea tipului expresiilor / funcțiilor

curry (curry id)

(TApp) \rightarrow expr :: b \Leftrightarrow curry :: a \rightarrow b, curry id :: a

(ExAnterior) \rightarrow curry id :: f \rightarrow g \rightarrow (f,g)

$$\frac{}{\text{curry } :: ((c,d) \rightarrow e) \rightarrow c \rightarrow d \rightarrow e} (TCurry)$$

Constrângeri

a = f \rightarrow g \rightarrow (f,g)

a = (c,d) \rightarrow e, b = c \rightarrow d \rightarrow e

Determinarea tipului expresiilor / funcțiilor

curry (curry id)

(TApp) \rightarrow expr :: b \Leftrightarrow curry :: a \rightarrow b, curry id :: a

(ExAnterior) \rightarrow curry id :: f \rightarrow g \rightarrow (f,g)

Constrângeri

a = f \rightarrow g \rightarrow (f,g)

a = (c,d) \rightarrow e, b = c \rightarrow d \rightarrow e

$$\frac{}{\text{curry} :: ((c,d) \rightarrow e) \rightarrow c \rightarrow d \rightarrow e} (TCurry)$$

În etapa de unificare: a = f \rightarrow g \rightarrow (f,g) = (c,d) \rightarrow e \rightarrow f = (c,d), e = g \rightarrow (f,g) = g \rightarrow ((c,d),g)
 \rightarrow b = c \rightarrow d \rightarrow g \rightarrow ((c,d), g), deci expresia are tipul c \rightarrow d \rightarrow g \rightarrow ((c,d), g) conform TApp.

Determinarea tipului expresiilor / funcțiilor

curry uncurry

Constrângeri

Determinarea tipului expresiilor / funcțiilor

curry uncurry

(TApp) → expr :: b \Leftrightarrow curry :: a -> b, uncurry :: a

Constrângeri

Determinarea tipului expresiilor / funcțiilor

curry uncurry

$(TApp) \rightarrow \text{expr} :: b \Leftrightarrow \text{curry} :: a \rightarrow b, \text{uncurry} :: a$

$$\frac{}{\text{uncurry} :: (c \rightarrow d \rightarrow e) \rightarrow (c, d) \rightarrow e} (TUncurry)$$

Constrângeri

$a = (c \rightarrow d \rightarrow e) \rightarrow (c, d) \rightarrow e$

Determinarea tipului expresiilor / funcțiilor

curry uncurry

$(TApp) \rightarrow \text{expr} :: b \Leftrightarrow \text{curry} :: a \rightarrow b, \text{uncurry} :: a$

$$\frac{}{\text{uncurry} :: (c \rightarrow d \rightarrow e) \rightarrow (c, d) \rightarrow e} (TUncurry)$$
$$\frac{}{\text{curry} :: ((f, g) \rightarrow h) \rightarrow f \rightarrow g \rightarrow h} (TCurry)$$

Constrângeri

$a = (c \rightarrow d \rightarrow e) \rightarrow (c, d) \rightarrow e$

$a = (f, g) \rightarrow h, b = f \rightarrow g \rightarrow h$

Determinarea tipului expresiilor / funcțiilor

curry uncurry

$(TApp) \rightarrow \text{expr} :: b \Leftrightarrow \text{curry} :: a \rightarrow b, \text{uncurry} :: a$

$$\frac{}{\text{uncurry} :: (c \rightarrow d \rightarrow e) \rightarrow (c, d) \rightarrow e} (TUncurry)$$

$$\frac{}{\text{curry} :: ((f, g) \rightarrow h) \rightarrow f \rightarrow g \rightarrow h} (TCurry)$$

Constrângeri

$a = (c \rightarrow d \rightarrow e) \rightarrow (c, d) \rightarrow e$

$a = (f, g) \rightarrow h, b = f \rightarrow g \rightarrow h$

În etapa de unificare: $a = (c \rightarrow d \rightarrow e) \rightarrow (c, d) \rightarrow e = (f, g) \rightarrow h \rightarrow$ (printre altele) $c \rightarrow d \rightarrow e = (f, g)$ care nu unifică (sunt aplicații ale unor constructori de tip diferenți), deci tipul expresiei nu poate fi sintetizat și vom obține o eroare de tip.

Exercițiu: Încercați și curry curry, uncurry curry, uncurry uncurry (nu toate dau eroare).

Sinteză de tip – Cuprins

- Sistem de tipuri
- Sinteză de tip
- Siguranță de tip
- Unificare
- Exemple
- Mesaje de eroare

Parse error

- programul nu respectă **sintaxa Haskell**
- se indică locul în care a apărut un simbol neașteptat
- eroarea e la locul indicat sau undeva înainte

```
f : Int -> [a] -> Int  
f a b = a + length b
```

curs7.hs:1:9: parse error on input '->'

```
f x = x ++ [1; 2; 3]
```

curs7.hs:1:14: parse error on input ';'

```
f x =  
| null x = 0  
| otherwise = 1
```

curs7.hs:2:9: parse error on input '|'

Parse error

- programul nu respectă **sintaxa Haskell**
- se indică locul în care a apărut un simbol neașteptat
- eroarea e la locul indicat sau undeva înainte

```
f : Int -> [a] -> Int  
f a b = a + length b
```

curs7.hs:1:9: parse error on input '->'
Înlocuieste cu ::

```
f x = x ++ [1; 2; 3]
```

curs7.hs:1:14: parse error on input ';'
Înlocuieste cu ,
curs7.hs:2:9: parse error on input '|'

```
f x =  
| null x = 0  
| otherwise = 1
```

Şterge = dinainte de |

Not in scope

- compilatorul **nu recunoaște o variabilă / o funcție / un constructor**
- fie am folosit o variabilă / o funcție / un constructor care nu există (posibil typo)
- fie trebuie să importăm modulul în care se află ace(a)sta

```
f a x
  | null x = cons a x
  | otherwise = f (a+1) (tail x)
```

Not in scope: 'cons'

```
f a x
  | null x = a
  | oterwise = f (a+1) (tail x)
```

Not in scope: 'oterwise'
Perhaps you meant 'otherwise'

```
f x = sort (take 10 x)
```

Not in scope: 'sort'

Not in scope

- compilatorul **nu recunoaște o variabilă / o funcție / un constructor**
- fie am folosit o variabilă / o funcție / un constructor care nu există (posibil typo)
- fie trebuie să importăm modulul în care se află acea(a)sta

```
f a x
| null x = cons a x
| otherwise = f (a+1) (tail x)
```

Not in scope: 'cons'
Înlocuiește cu (:)

```
f a x
| null x = a
| otherwise = f (a+1) (tail x)
```

Not in scope: 'otherwise'
Perhaps you meant 'otherwise'

```
f x = sort (take 10 x)
```

Not in scope: 'sort'

Caută funcția (ex: hoogle.haskell.org) și vezi ce modul trebuie importat (aici `import Data.List`)

The function <> is applied to <> arguments

- am aplicat o funcție pe mai multe sau mai puține argumente decât așteaptă
- fie am pus ceva în plus / minus
- fie n-am pus parantezele bine

```
import Data.List  
f x = sort (take 10 x) (<)
```

The function 'sort' is applied to two arguments,
but its type '[a] -> [a]' has only one

```
f x = if null x then 0 else 1 + f tail x
```

...

Probable cause: 'tail' is applied to too few arguments

The function <> is applied to <> arguments

- am aplicat o funcție pe mai multe sau mai puține argumente decât așteaptă
- fie am pus ceva în plus / minus
- fie n-am pus parantezele bine

```
import Data.List
f x = sort (take 10 x) (<)
f x = if null x then 0 else 1 + f tail x
```

The function 'sort' is applied to two arguments,
but its type '[a] -> [a]' has only one
Șterge (<)

+ f tail x

...

Probable cause: 'tail' is applied to too few arguments
Pune paranteze la tail: f (tail x)

Couldn't match expected type with actual type

- fie nu suntem în concordanță cu signatura (ori corectăm funcția, ori signatura)
- fie **am aplicat o funcție pe tipuri incompatibile cu ce aşteaptă funcția**
 - posibil să fi greșit ordinea argumentelor

```
f :: String -> String    Couldn't match type 'Char' with '[Char]'
```

```
f = head
```

```
f = 'A' ++ "BC"    Couldn't match expected type '[Char]' with actual type 'Char'
```

```
f = (:) "BC" 'A'    Couldn't match expected type '[[Char]]' with actual type 'Char'  
In the second argument of '(:)', namely 'A'
```

Couldn't match expected type with actual type

- fie nu suntem în concordanță cu signatura (ori corectăm funcția, ori signatura)
- fie **am aplicat o funcție pe tipuri incompatibile cu ce așteaptă funcția**
 - posibil să fi greșit ordinea argumentelor

```
f :: String -> String  
f = head
```

Couldn't match type 'Char' with '[Char]'
Fie f :: String -> Char, fie f = (: []) . head

```
f = 'A' ++ "BC"
```

Couldn't match expected type '[Char]' with actual type 'Char'
Fie 'A' : "BC", fie "A" ++ "BC"

```
f = (:) "BC" 'A'
```

Couldn't match expected type '[[Char]]' with actual type 'Char'
In the second argument of '(:)', namely 'A'
Schimb ordinea: (:) 'A' "BC"

Non-exhaustive patterns

- o funcție definită cu pattern matching sau găzzi **nu acoperă toate valorile tipului**
- fie am uitat anumite valori
- fie avem un typo la numele funcției (și Haskell pur și simplu consideră că este altă funcție)

```
f x
| x < 0 = -1
| x > 0 = 1
```

*Main> f 0
*** Exception: curs7.hs:(1,1)-(3,19): Non-exhaustive patterns in function f
(dacă avem warning-urile activate, primim warning la compilare)

```
mySum [] = 0
mySum (x:xs) = x + mySum xs
```

*Main> mySum [1..5]
*** Exception: ... Non-exhaustive patterns ... mySum

Non-exhaustive patterns

- o funcție definită cu pattern matching sau găzzi **nu acoperă toate valorile tipului**
- fie am uitat anumite valori
- fie avem un typo la numele funcției (și Haskell pur și simplu consideră că este altă funcție)

```
f x
| x < 0 = -1
| x > 0 = 1
```

*Main> f 0

*** Exception: curs7.hs:(1,1)-(3,19): Non-exhaustive patterns in function f
(dacă avem warning-urile activate, primim warning la compilare)

Adaugă | otherwise = ...

```
mySum [] = 0
```

*Main> mySum [1..5]

```
mySUM (x:xs) = x + mySum xs
```

*** Exception: ... Non-exhaustive patterns ... mySum

Corectează mySUM în mySum

Rezumat

Sinteză de tip

Regulă de sinteză de tip

Algoritm de sinteză de tip

Siguranță de tip

Unificare

O variabilă unifică cu
2 constante unifică
2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip
Regulă de sinteză de tip

Algoritm de sinteză de tip

Siguranță de tip

Unificare

O variabilă unifică cu
2 constante unifică
2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip

Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)

Algoritm de sinteză de tip

Siguranță de tip

Unificare

O variabilă unifică cu
2 constante unifică
2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip

Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)

Algoritm de sinteză de tip: potrivește concluziile relațiilor de sinteză de tip cu componentele expresiei până ajunge la componente elementare

Siguranță de tip

Unificare

O variabilă unifică cu

2 constante unifică

2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip

Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)

Algoritm de sinteză de tip: potrivește concluziile relațiilor de sinteză de tip cu componentele expresiei până ajunge la componente elementare

Siguranță de tip: progres (expresia bine tipată nu se blochează), conservare (rămâne bine tipată după evaluare)

Unificare

O variabilă unifică cu

2 constante unifică

2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip

Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)

Algoritm de sinteză de tip: potrivește concluziile relațiilor de sinteză de tip cu componentele expresiei până ajunge la componente elementare

Siguranță de tip: progres (expresia bine tipată nu se blochează), conservare (rămâne bine tipată după evaluare)

Unificare: cea mai generală substituție conform căreia 2 expresii de tip devin identice

O variabilă unifică cu

2 constante unifică

2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip

Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)

Algoritm de sinteză de tip: potrivește concluziile relațiilor de sinteză de tip cu componentele expresiei până ajunge la componente elementare

Siguranță de tip: progres (expresia bine tipată nu se blochează), conservare (rămâne bine tipată după evaluare)

Unificare: cea mai generală substituție conform căreia 2 expresii de tip devin identice

O variabilă unifică cu: orice, cât timp occurs check este OK

2 constante unifică

2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip

Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)

Algoritm de sinteză de tip: potrivește concluziile relațiilor de sinteză de tip cu componentele expresiei până ajunge la componente elementare

Siguranță de tip: progres (expresia bine tipată nu se blochează), conservare (rămâne bine tipată după evaluare)

Unificare: cea mai generală substituție conform căreia 2 expresii de tip devin identice

O variabilă unifică cu: orice, cât timp occurs check este OK

2 constante unifică: dacă sunt identice

2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip

Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)

Algoritm de sinteză de tip: potrivește concluziile relațiilor de sinteză de tip cu componentele expresiei până ajunge la componente elementare

Siguranță de tip: progres (expresia bine tipată nu se blochează), conservare (rămâne bine tipată după evaluare)

Unificare: cea mai generală substituție conform căreia 2 expresii de tip devin identice

O variabilă unifică cu: orice, cât timp occurs check este OK

2 constante unifică: dacă sunt identice

2 construcții unifică: dacă folosesc același constructor de tip pe argumente care unifică la rândul lor

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip

Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)

Algoritm de sinteză de tip: potrivește concluziile relațiilor de sinteză de tip cu componentele expresiei până ajunge la componente elementare

Siguranță de tip: progres (expresia bine tipată nu se blochează), conservare (rămâne bine tipată după evaluare)

Unificare: cea mai generală substituție conform căreia 2 expresii de tip devin identice

O variabilă unifică cu: orice, cât timp occurs check este OK

2 constante unifică: dacă sunt identice

2 construcții unifică: dacă folosesc același constructor de tip pe argumente care unifică la rândul lor

Structuri folosite de algoritmul de unificare: stiva de constrângeri, substituția

PARADIGME DE PROGRAMARE

Curs 8

Polimorfism. Clase în Haskell.

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Derivarea unei clase
- Instantierea unei clase
- Context
- Clase uzuale
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

Polimorfism

- Se referă la furnizarea unei interfețe comune (în cazul nostru, o aceeași funcție) pentru tipuri diferite
- 2 tipuri de polimorfism

Polimorfism parametric = funcția se comportă la fel pentru argumente de tipuri diferite

- Ex: (:), head, id

Polimorfism ad-hoc = funcția este supraîncărcată, având comportament diferit în funcție de tipul argumentelor pe care le primește

- Ex: +, *, ==
- $2 + 3 :: \text{Int}$ întoarce 5, $2 + 3 :: \text{Double}$ întoarce 5.0

Observație: operațiile aritmetice se comportă diferit în funcție de context (la sinteza de tip se deduce că rezultatul operației trebuie să aibă un anumit tip)

Supraîncărcare

Avantaje

- **Lizibilitate**

- Mai clar $x == y, a == b, p == q$ decât `eqInt x y, eqChar a b, eqBool p q`

- **Reutilizare**

- Mai bine o singură funcție polimorfică `myElem` (reimplementarea lui `elem` din Haskell)

```
myElem _ [] = False  
myElem a (x:xs) = a == x || myElem a xs
```

decât câte un `myElem` pentru fiecare tip de date care poate fi căutat într-o listă

`myElemInt` care folosește `eqInt`

`myElemChar` care folosește `eqChar`

`myElemBool` care folosește `eqBool` ...

Alternative (inferioare) la supraîncărcare

- 1) Funcții diferite pentru fiecare tip (myElemInt, myElemChar, myElemBool...)
- 2) Pasarea funcției al cărei comportament diferă ca parametru

```
--myElem2 :: (a -> b -> Bool) -> a -> [b] -> Bool --tipul dedus
--myElem2 :: (a -> a -> Bool) -> a -> [a] -> Bool --tipul dat explicit
myElem2 _ _ [] = False
myElem2 eq a (x:xs) = eq a x || myElem2 eq a xs
```

dar, chiar dacă declarăm noi tipul funcției, acesta este mai general decât ne-am dori, permitând și alte funcții decât cele care testează pentru egalitate

Observație

Haskell nu permite definiții multiple (cu signaturi diferite) pentru un același nume de funcție. O asemenea facilitate ar distrugе mecanismul foarte puternic de sinteză de tip.

Supraîncărcare și sinteză de tip

- Tipul funcției trebuie să **restrângă utilizarea ei la tipurile care supraîncarcă o anumită operație**
 - Astfel prevenim și erori rezultate din aplicarea funcției pe tipuri care nu au operația respectivă
 - **Ex:** Este posibil să avem liste de funcții în Haskell:

```
*Main> zipWith (\f x -> f x) [(+1), (2/)] [3..]  
[4.0, 0.5]
```

dar nu este posibil să căutăm o funcție în ele, întrucât nu există un algoritm pentru a determina dacă 2 funcții sunt egale (au același comportament):

```
*Main> elem (+1) [(+1), (2/)]  
...  
No instance for (Eq (a0 -> a0)) arising from a use of `elem'
```

Funcțiile nu pot fi comparate pentru egalitate, dar argumentele lui elem trebuie să poată

Supraîncărcare și sinteză de tip

Exemple (signaturi pentru funcții polimorfice ad-hoc)

- :t **sum**

sum :: (Foldable t, Num a) => t a -> a

Tipul a trebuie să supraîncarce operațiile specifice numerelor (+, -, *, ...)

- :t **elem**

elem :: (Foldable t, Eq a) => a -> t a -> Bool

Tipul a trebuie să fie comparabil pentru egalitate (să supraîncarce ==)

- :t **Data.List.sort**

Data.List.sort :: Ord a => [a] -> [a]

Tipul a trebuie să fie ordonabil (să supraîncarce <, <=, >, >= ...)

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Derivarea unei clase
- Instantierea unei clase
- Context
- Clase uzuale
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

Clase în Haskell

Clasă Haskell = multime de tipuri care supraîncarcă operațiile specifice clasei

- mecanismul Haskell de a implementa polimorfismul ad-hoc
- și un mod de a documenta cum se comportă tipurile (ex: Int este Bounded, Integer nu)

Exemple

Show – clasa tipurilor afișabile (prin funcția show = un fel de toString din Java)

- Membri: toate tipurile în afară de IO și funcții

Num – clasa tipurilor numerice

- Membri: Int, Integer, Float, Double

Bounded – clasa tipurilor ale căror valori sunt limitate inferior și superior

- Membri: Int, Char, Bool, tupluri, Ordering

Definirea unei clase

```
class NumeClasă t where
```

```
    f1 :: signatura1
```

```
    ...
```

```
    fn :: signaturan
```

Variabilă de tip care reprezintă un tip membru al clasei

Signaturile folosesc variabila de tip (pentru că prin definiție descriem o întreagă clasă de tipuri)

Exemplu

```
class Eq a where
```

```
    (==) :: a -> a -> Bool
```

```
    (/=) :: a -> a -> Bool
```

Semnificația: Pentru ca un tip a să aparțină clasei Eq, trebuie ca el să implementeze funcțiile (==) și (/=) (respectând signaturile date)

Implementări implicate

- În general, clasa doar definește funcțiile care trebuie supraîncărcate, nu le și implementează (firesc, încă că implementările diferă de la tip la tip)
- Exceptie: implementări implicate (**definiții circulare** ale funcțiilor, care permit ca un tip membru să redefinească doar o parte din funcții iar restul să se comporte corect)
- **Minimal complete definition:** un set minimal de funcții ale clasei care **trebuie redefinite** la instanțiere astfel încât toate funcțiile să se comporte apoi corect pe tipul respectiv

Exemplu

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

Dacă un tip specifică o implementare pentru (==), (/=) se deduce automat din implementarea implicită

Dacă specifică implementarea pentru (/=), atunci (==) este cel care se deduce automat

Minimal complete definition: (==) SAU (/=)

Implementări implicate

Avantaje

- **Efort minim:** Nu trebuie să redefinim toate funcțiile
 - Din rațiuni de performanță, uneori le vom redefini pe toate (Ex: poate că tipul are o metodă mai bună de a detecta inegalitatea decât să eșueze în verificarea egalității)
- **Ușurință la instanțiere:** Există mai multe definiții complete minime, o putem alege pe cea mai convenabilă
 - Ex: Uneori e mai ușor să definesc `(==)`, alteori e mai ușor să definesc `(/=)`

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Derivarea unei clase
- Instantierea unei clase
- Context
- Clase uzuale
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

Derivarea unei clase

- Așa cum funcțiile depind de apartenența unui tip la o anumită clasă, la fel și clasele pot necesita ca tipul lor membru să aparțină deja altei clase

Derivarea clasei = impunerea condiției ca un tip să fie deja membru al altei clase (clasa părinte) în momentul în care el devine instanță a clasei copil

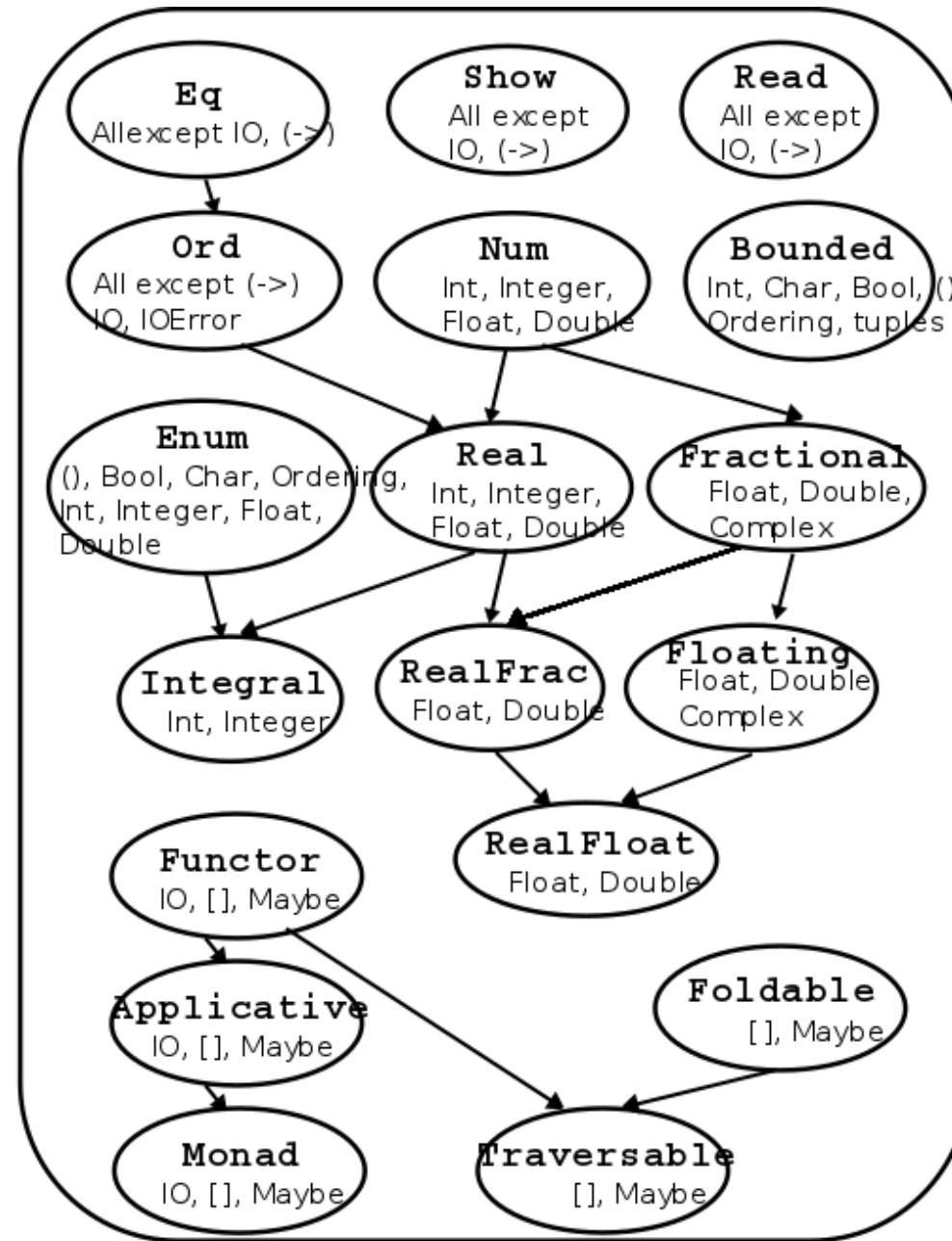
- Clasa copil nu moștenește de la clasa părinte decât promisiunea că instanțele sale vor fi implementat anumite funcții
 - Cele două instanțieri (a clasei părinte și a clasei copil) sunt separate

Exemplu

```
class Eq a => Ord a where  
... -- alte funcții decât (==), (/=)
```

Semnificația: Pentru ca un tip a să fie membru al clasei Ord, el trebuie să fie deja membru al clasei Eq și să implementeze funcțiile specificate în clasa Ord

Ierarhia de clase în Haskell



Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Derivarea unei clase
- Instantierea unei clase
- Context
- Clase uzuale
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

Instanțierea unei clase

Instanță a unei clase = tip care supraîncarcă toate funcțiile clasei (tip membru)

instance NumeClasă Tip where → Tip concret, nu variabilă de tip ca la definirea clasei
f₁ = ... -- implementare
...
f_n = ... -- implementare → Implementări care respectă signaturile din definiția clasei

Exemplu

```
instance Show Dice where
    show S1 = "[ · ]"
    ...
    show S6 = "[::::]"
```

Instanțierea unei clase definită de utilizator

Exemplu

```
class Valuable a where
    value :: a -> Int

instance Valuable Dice where
    value S1 = 1
    ...
    value S6 = 6
```

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Derivarea unei clase
- Instantierea unei clase
- Context
- Clase uzuale
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

Context

Context = mulțimea constrângerilor (de apartenență la diverse clase) asupra variabilelor de tip din

- **signatura unei funcții**

```
rollSum :: (Valuable a, Valuable b) => (a, b) -> Int  
rollSum (x, y) = value x + value y
```

- **declarația unei clase**

```
class Eq a => Ord a where
```

- **instantierea unei clase**

```
instance Eq a => Eq [a] where  
  [] == [] = True  
  (x:xs) == (y:ys) = x == y && xs == ys  
  _ == _ = False
```

Se folosește un tuplu pentru constrângeri multiple

Propagarea contextului la sinteza de tip

Să se determine tipul următoarelor 2 funcții:

```
myLen lst
| lst == [] = 0
| otherwise = 1 + myLen (tail lst)
```

```
myLen2 lst
| null lst = 0
| otherwise = 1 + myLen2 (tail lst)
```

Propagarea contextului la sinteza de tip

Să se determine tipul următoarelor 2 funcții:

```
myLen lst
| lst == [] = 0
| otherwise = 1 + myLen (tail lst)
```

- Rezultatul este un număr ($0 :: \text{Num } a \Rightarrow a$, $1 :: \text{Num } a \Rightarrow a$, $(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$)
- Atât otherwise cât și == întorc Bool (necesar pentru a folosi găzzi)
- Tipul lui lst trebuie să fie comparabil pentru egalitate ($(==) :: \text{Eq } a \Rightarrow a \rightarrow a \rightarrow \text{Bool}$)
- Tipul lui lst trebuie să fie $[t]$ ($[] :: [t]$, $(==) :: \text{Eq } a \Rightarrow a \rightarrow a \rightarrow \text{Bool}$)
- Tipul t trebuie să fie comparabil pentru egalitate (instance $\text{Eq } a \Rightarrow \text{Eq } [a]$)

Propagarea contextului la sinteza de tip

Să se determine tipul următoarelor 2 funcții:

```
myLen lst
| lst == [] = 0
| otherwise = 1 + myLen (tail lst)
```

- Rezultatul este un număr ($0 :: \text{Num } a \Rightarrow a$, $1 :: \text{Num } a \Rightarrow a$, $(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$)
- Atât otherwise cât și == întorc Bool (necesar pentru a folosi găzzi)
- Tipul lui lst trebuie să fie comparabil pentru egalitate ($(==) :: \text{Eq } a \Rightarrow a \rightarrow a \rightarrow \text{Bool}$)
- Tipul lui lst trebuie să fie $[t]$ ($[] :: [t]$, $(==) :: \text{Eq } a \Rightarrow a \rightarrow a \rightarrow \text{Bool}$)
- Tipul t trebuie să fie comparabil pentru egalitate (instance $\text{Eq } a \Rightarrow \text{Eq } [a]$)

$\text{myLen} :: (\text{Num } a, \text{Eq } t) \Rightarrow [t] \rightarrow a$

Propagarea contextului la sinteza de tip

Să se determine tipul următoarelor 2 funcții:

```
myLen2 lst
| null lst = 0
| otherwise = 1 + myLen2 (tail lst)
```

- Rezultatul este un număr ($0 :: \text{Num } a \Rightarrow a$, $1 :: \text{Num } a \Rightarrow a$, $(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$)
- Atât otherwise cât și null întorc Bool (necesar pentru a folosi gărzi)

Propagarea contextului la sinteza de tip

Să se determine tipul următoarelor 2 funcții:

```
myLen2 lst
| null lst = 0
| otherwise = 1 + myLen2 (tail lst)
```

- Rezultatul este un număr ($0 :: \text{Num } a \Rightarrow a$, $1 :: \text{Num } a \Rightarrow a$, $(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$)
- Atât otherwise cât și null întorc Bool (necesar pentru a folosi gărzi)
- Tipul lui lst trebuie să fie $[t]$ ($\text{tail} :: [t] \rightarrow [t]$)

Propagarea contextului la sinteza de tip

Să se determine tipul următoarelor 2 funcții:

```
myLen2 lst
| null lst = 0
| otherwise = 1 + myLen2 (tail lst)
```

- Rezultatul este un număr ($0 :: \text{Num } a \Rightarrow a$, $1 :: \text{Num } a \Rightarrow a$, $(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$)
- Atât otherwise cât și null întorc Bool (necesar pentru a folosi gărzi)
- Tipul lui lst trebuie să fie $[t]$ ($\text{tail} :: [t] \rightarrow [t]$)

myLen2 :: Num a => [t] -> a

*Main> myLen2 [(+)]

1

*Main> myLen [(+)]

eroare

Simplificarea contextului

- Constrângerile din context se pot aplica **doar pe variabile de tip** (nu pe expresii de tip mai complexe decât atât)
 - `myLen :: (Num a, Eq [t]) => [t] -> a` – eroare fiindcă `[t]` nu e variabilă de tip
 - `myLen :: (Num a, Eq t) => [t] -> a` – corect
- Constrângerile impun apartenența la clasa copil **fără să impună explicit apartenența la toate clasele părinte** (aceasta se subînțelege)
 - `myMax :: (Eq a, Ord a) => a -> a -> a` – redundant
 - `myMax :: Ord a => a -> a -> a` – corect

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Derivarea unei clase
- Instantierea unei clase
- Context
- Clase uzuale
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

Clasa Ord

```
class Eq a => Ord a where
    compare :: a -> a -> Ordering
    (<), (≤), (≥), (>) :: a -> a -> Bool
    max, min :: a -> a -> a
```

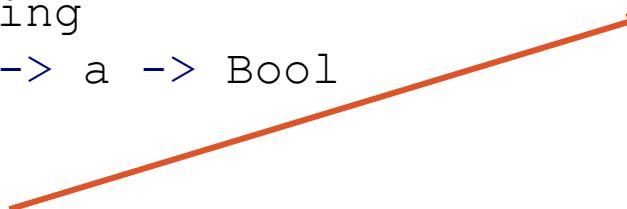
```
compare x y
| x == y = EQ
| x <= y = LT
| otherwise = GT
```

```
x ≤ y = compare x y /= GT
x < y = compare x y == LT
x ≥ y = compare x y /= LT
x > y = compare x y == GT
```

```
max x y
| x >= y = x
| otherwise = y
```

```
min x y
| x <= y = x
| otherwise = y
```

Minimal complete definition:
?



Clasa Ord

```
class Eq a => Ord a where
    compare :: a -> a -> Ordering
    (<), (≤), (≥), (>) :: a -> a -> Bool
    max, min :: a -> a -> a
```

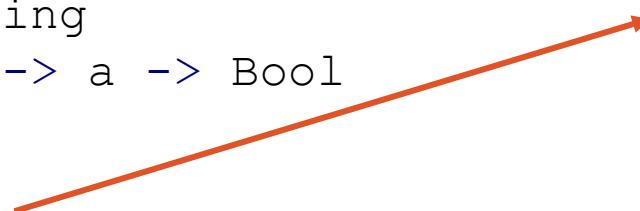
```
compare x y
| x == y = EQ
| x <= y = LT
| otherwise = GT
```

```
x <= y = compare x y /= GT
x < y = compare x y == LT
x ≥ y = compare x y /= LT
x > y = compare x y == GT
```

```
max x y
| x >= y = x
| otherwise = y
```

```
min x y
| x <= y = x
| otherwise = y
```

Minimal complete definition:
(≤) SAU compare



Clasa Enum

```
class Enum a where
    succ, pred      :: a -> a
    toEnum          :: Int -> a
    fromEnum        :: a -> Int
    enumFrom        :: a -> [a]           -- [n..]
    enumFromThen   :: a -> a -> [a]       -- [n, n'..]
    enumFromTo     :: a -> a -> [a]       -- [n..m]
    enumFromThenTo :: a -> a -> a -> [a]  -- [n, n'..m]

    succ            = toEnum . (+1) . fromEnum
    pred            = toEnum . (subtract 1) . fromEnum
    enumFrom x      = map toEnum [fromEnum x ..]
    enumFromTo x y = map toEnum [fromEnum x .. fromEnum y]
    enumFromThen x y = map toEnum [fromEnum x, fromEnum y ..]
    enumFromThenTo x y z = map toEnum [fromEnum x, fromEnum y .. fromEnum z]
```

Minimal complete definition:
toEnum <--> fromEnum

Clasa Bounded

```
class Bounded a where
    minBound      :: a
    maxBound      :: a
```

```
*Main> minBound :: Int
-9223372036854775808
*Main> maxBound :: Int
9223372036854775807
*Main> maxBound :: Integer
```

```
<interactive>:149:1:
No instance for (Bounded Integer) arising from a use of `maxBound'
```

Cuvântul cheie **deriving**

- Multe din clasele predefinite sunt **derivabile**, adică **funcțiile lor pot fi implementate automat** (rudimentar) pentru un nou tip care solicită asta folosind **deriving**
- **Show** – afișează o valoare ca pe o aplicare succesivă de constructori
Eq – două valori sunt egale dacă se obțin prin aplicarea acelorași constructori pe aceleași valori în aceeași ordine
Ord și Enum – folosesc ordinea în care sunt definiți constructorii de date
Ex: False < True pentru că data Bool = False | True

Exemplu (cu verificări la calculator)

```
data Dice = S1 | S2 | S3 | S4 | S5 | S6 deriving (Eq, Ord, Enum)
```

Pentru a instanția Ord trebuie să instanțiem și Eq, nu e automat

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Derivarea unei clase
- Instantierea unei clase
- Context
- Clase uzuale
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

Containere

- Să presupunem că vrem să definim **clasa Container** (care nu există în Haskell) pentru tipuri (existente sau definite de noi) care conțin elemente (ex: [a], Maybe a, List a, BSTree a) și că această clasă oferă **funcția contents** care întoarce o listă cu toate elementele din structură

```
class Container t where  
    contents :: t -> ??
```

t = tipul containerului, de exemplu BSTree a

Problema: trebuie să întoarcem [a] și a nu este accesibil în acest punct

Containere

```
class Container t where  
    contents :: t -> ??
```

t = tipul containerului, de exemplu BSTree a

Problema: trebuie să întoarcem [a] și a nu este accesibil în acest punct

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

Soluția: variabila t să reprezinte constructorul de tip (ex: BSTree), nu întreg tipul parametrizat (BSTree a)

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

Constructorul de tip ([]), nu întreg tipul parametrizat ([a])

Exercițiu la calculator: instanțierea pentru List a

Clase Haskell pentru containere

- Clasa Container nu există în Haskell, dar există 2 clase ale căror operații sunt dedicate containerelor
- **Functor** (pentru **abstractizarea operațiilor de tip map**)

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

- **Foldable** (pentru **abstractizarea operațiilor de tip fold**)

```
class Foldable t where
    ...
    foldr   :: (a -> b -> b) -> b -> t a -> b
    ...
    foldl   :: (a -> b -> a) -> a -> t b -> a
    ...
```

Exerciții

La ce se evaluează ex1, ex2, ex3, ex4, cunoscând:

instance Functor Maybe -- Defined in 'Data.Maybe'

instance Functor [] -- Defined in 'GHC.Base'

instance Functor ((->) r) -- Defined in 'GHC.Base'

instance Functor ((,) a) -- Defined in 'GHC.Base'

instance Foldable ((,) a) -- Defined in 'Data.Foldable'

```
ex1 = fmap (+1) (Just 5)
```

```
ex2 = fmap (+1) (+1) 2
```

```
ex3 = fmap (+1) (1, 2)
```

```
ex4 = foldl (+) 10 (1, 2)
```

Exerciții

La ce se evaluează ex1, ex2, ex3, ex4, cunoscând:

instance Functor Maybe -- Defined in 'Data.Maybe'

instance Functor [] -- Defined in 'GHC.Base'

instance Functor ((->) r) -- Defined in 'GHC.Base'

instance Functor ((,) a) -- Defined in 'GHC.Base'

instance Foldable ((,) a) -- Defined in 'Data.Foldable'

```
ex1 = fmap (+1) (Just 5)           -- Just 6
ex2 = fmap (+1) (+1) 2             -- 4
ex3 = fmap (+1) (1, 2)             -- (1, 3)
ex4 = foldl (+) 10 (1, 2)          -- 12
```

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Derivarea unei clase
- Instantierea unei clase
- Context
- Clase uzuale
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

Clase Haskell versus clase și interfețe POO

Clase Haskell ≠ Clase POO

- O clasă Haskell este o mulțime de tipuri
- O clasă POO este un singur tip (mulțimea valorilor de acel tip)

Clase Haskell ~ Interfețe POO

- Clasa Haskell este instantiată de diverse tipuri
- Interfața POO este implementată de diverse clase (care sunt ca niște tipuri)
- Ambele doar precizează operațiile pe care tipul trebuie să le aibă, nu le și implementează

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Derivarea unei clase
- Instantierea unei clase
- Context
- Clase uzuale
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

No instance for (\neq a)

- **signatura** furnizată este **incompletă**: trebuie să-i adăugăm contextul

```
eq :: a -> a -> Bool  
eq x y = x == y
```

No instance for (Eq a) arising from a use of `=='

Possible fix:

add (Eq a) to the context of
the type signature for eq :: a -> a -> Bool
In the expression: x == y

No instance for (\neq a)

- **signatura** furnizată este **incompletă**: trebuie să-i adăugăm contextul

```
eq :: Eq a => a -> a -> Bool  
eq x y = x == y
```

No instance for (Eq a) arising from a use of `=='

Possible fix:

add (Eq a) to the context of
the type signature for eq :: a -> a -> Bool

In the expression: x == y

Could not deduce (b ~ a)

- din sinteza de tip rezultă că **a = b**, dar **signaturile furnizate de programator nu garantează** acest lucru
- rigid type variable înseamnă că signatura a fost fixată de programator și Haskell nu e liber să unifice a și b

eq :: (Eq a, Eq b) => a -> b -> Bool Could not deduce (b ~ a)

eq x y = x == y from the context (Eq a, Eq b)

...

'b' is a rigid type variable bound by
the type signature for eq :: (Eq a, Eq b) => a -> b -> Bool

'a' is a rigid type variable bound by
the type signature for eq :: (Eq a, Eq b) => a -> b -> Bool

Could not deduce (b ~ a)

- din sinteza de tip rezultă că **a = b**, dar **signaturile furnizate de programator nu garantează** acest lucru
- rigid type variable înseamnă că signatura a fost fixată de programator și Haskell nu e liber să unifice a și b

```
eq :: Eq a => a -> a -> Bool  
eq x y = x == y
```

Could not deduce (b ~ a)
from the context (Eq a, Eq b)

...

'b' is a rigid type variable bound by
the type signature for eq :: (Eq a, Eq b) => a -> b -> Bool

'a' is a rigid type variable bound by
the type signature for eq :: (Eq a, Eq b) => a -> b -> Bool

No instance for (Num <>)

- numerele în Haskell sunt polimorfice (în funcție de context, 1 este Int / Float / Double / ...)
- în orice context în care apare un întreg (ex: 1), acesta este înlocuit cu fromInteger 1 (care îl transformă în tipul numeric așteptat)
- fromInteger :: Num a => Integer -> a
- eroarea spune că **am folosit un număr pe poziția pe care se aștepta un tip nenumeric**

```
f = False || 1
```

No instance for (Num Bool) arising from the literal `1'

Explicație: aștept Bool, înseamnă că fromInteger 1 :: Bool = a, înseamnă că Num a adică Num Bool (dar asta nu se întâmplă)

No instance for (Num <>)

- numerele în Haskell sunt polimorfice (în funcție de context, 1 este Int / Float / Double / ...)
- în orice context în care apare un întreg (ex: 1), acesta este înlocuit cu fromInteger 1 (care îl transformă în tipul numeric așteptat)
- fromInteger :: Num a => Integer -> a
- eroarea spune că **am folosit un număr pe poziția pe care se aștepta un tip nenumeric**

```
f = False || True
```

No instance for (Num Bool) arising from the literal `1'

Explicație: aștept Bool, înseamnă că fromInteger 1 :: Bool = a, înseamnă că Num a adică Num Bool (dar asta nu se întâmplă)

Rezumat

Polimorfism parametric

Polimorfism ad-hoc

Clasă

Definire clasă

Derivare clasă

Instanțiere clasă

Context

Clase uzuale

Clase pentru containere

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc

Clasă

Definire clasă

Derivare clasă

Instanțiere clasă

Context

Clase uzuale

Clase pentru containere

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite

Clasă

Definire clasă

Derivare clasă

Instanțiere clasă

Context

Clase uzuale

Clase pentru containere

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite

Clasă: mulțime de tipuri care supraîncarcă operațiile specifice clasei

Definire clasă

Derivare clasă

Instanțiere clasă

Context

Clase uzuale

Clase pentru containere

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite

Clasă: mulțime de tipuri care supraîncarcă operațiile specifice clasei

Definire clasă: class <Clasă> t where <declarații de tip>

Derivare clasă

Instanțiere clasă

Context

Clase uzuale

Clase pentru containere

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite

Clasă: mulțime de tipuri care supraîncarcă operațiile specifice clasei

Definire clasă: class <Clasă> t where <declarații de tip>

Derivare clasă: class <Clasă'> t => Clasă t where <declarații de tip>

Instanțiere clasă

Context

Clase uzuale

Clase pentru containere

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite

Clasă: mulțime de tipuri care supraîncarcă operațiile specifice clasei

Definire clasă: class <Clasă> t where <declarații de tip>

Derivare clasă: class <Clasă'> t => Clasă t where <declarații de tip>

Instanțiere clasă: instance <Clasă> <Tip> where <implementări>

Context

Clase uzuale

Clase pentru containere

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite

Clasă: mulțime de tipuri care supraîncarcă operațiile specifice clasei

Definire clasă: class <Clasă> t where <declarații de tip>

Derivare clasă: class <Clasă'> t => Clasă t where <declarații de tip>

Instanțiere clasă: instance <Clasă> <Tip> where <implementări>

Context: mulțimea constrângerilor de apartenență a variabilelor de tip la diverse clase

Clase uzuale

Clase pentru containere

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite

Clasă: mulțime de tipuri care supraîncarcă operațiile specifice clasei

Definire clasă: class <Clasă> t where <declarații de tip>

Derivare clasă: class <Clasă'> t => Clasă t where <declarații de tip>

Instanțiere clasă: instance <Clasă> <Tip> where <implementări>

Context: mulțimea constrângerilor de apartenență a variabilelor de tip la diverse clase

Clase uzuale: Eq, Ord, Read, Show, Enum, Bounded

Clase pentru containere

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite

Clasă: mulțime de tipuri care supraîncarcă operațiile specifice clasei

Definire clasă: class <Clasă> t where <declarații de tip>

Derivare clasă: class <Clasă'> t => Clasă t where <declarații de tip>

Instanțiere clasă: instance <Clasă> <Tip> where <implementări>

Context: mulțimea constrângerilor de apartenență a variabilelor de tip la diverse clase

Clase uzuale: Eq, Ord, Read, Show, Enum, Bounded

Clase pentru containere: Functor, Foldable

PARADIGME DE PROGRAMARE

Curs 9

Logica propozițională. Logica cu predicate de ordinul întâi.

Context

Scop: modelarea raționamentelor logice ca procese de calcul efectuate pe mașini de calcul

Abordare

- Descrierea proprietăților obiectelor din universul problemei într-un limbaj neambigu
 - Sintactic
 - Semantic
- Reguli pentru deducerea (calculul) de noi proprietăți din cele existente

Formalisme logice

- Logica propozițională
- Logica cu predicate de ordinul întâi

Logica propozițională – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Satisfiabilitate și validitate
- Derivabilitate
- Decidabilitate
- Inferență și reguli de inferență
- Demonstrație
- Rezoluție

Ontologie

- Universul problemei este descris prin propoziții declarative, numite **fapte**
Exemple

Iepurii sunt mai rapizi decât broaștele țestoase.

Bugs e iepure.

Leo e broască țestoasă.

Bugs e mai rapid decât Leo.

- **Faptele nu sunt structurate** astfel încât să surprindă obiectele din acest univers și relațiile dintre ele
 - În logica propozițională, ultima propoziție nu este o consecință logică a celorlalte, pentru că logica propozițională nu poate modela relațiile dintre aceste fapte

Logica propozițională – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Satisfiabilitate și validitate
- Derivabilitate
- Decidabilitate
- Inferență și reguli de inferență
- Demonstrație
- Rezoluție

Sintaxă

Expresie în logica propozițională = propoziție care poate fi adevărată sau falsă

propoziție = propoziție simplă (atomică) | propoziție compusă

(*ex: Bugs e iepure.*)

(*notatie:* $p, q, r, \text{etc.}$)

(*ex: Bugs e iepure și Leo e broască țestoasă.*)

(*notatie:* $\neg P, P \wedge Q, \text{etc.}$)

propoziție compusă = negație | conjuncție | disjuncție | implicație | echivalență
 $(\neg P)$ $(P \wedge Q)$ $(P \vee Q)$ $(P \Rightarrow Q)$ $(P \Leftrightarrow Q)$

Logica propozițională – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Satisfiabilitate și validitate
- Derivabilitate
- Decidabilitate
- Inferență și reguli de inferență
- Demonstrație
- Rezoluție

Semantică

Interpretare = multime de asocieri între fiecare propoziție atomică din limbaj și o valoare de adevăr

- **Exemplu:** Interpretarea I = { p = true, q = true, r = false }
Interpretarea J = { p = false, q = true, r = false }

- Valoarea de adevăr a unei propoziții se stabilește
 - În raport cu o interpretare (ex: $p \wedge q = \text{true}$ în interpretarea I / $p \wedge q = \text{false}$ în interpretarea J)
 - Conform unor **reguli semantice** care determină valoarea de adevăr a unei propoziții compuse din valorile de adevăr ale propozițiilor constituente

Reguli semantice pentru propoziții compuse

$\neg P_1 = \text{true}$	dacă și numai dacă	$P_1 = \text{false}$		
$P_1 \wedge P_2 = \text{true}$	dacă și numai dacă	$P_1 = \text{true}$	și	$P_2 = \text{true}$
$P_1 \vee P_2 = \text{true}$	dacă și numai dacă	$P_1 = \text{true}$	sau	$P_2 = \text{true}$
$P_1 \Rightarrow P_2 = \text{true}$	dacă și numai dacă	$P_1 = \text{false}$	sau	$P_2 = \text{true}$
$P_1 \Leftrightarrow P_2 = \text{true}$	dacă și numai dacă	$P_1 = P_2$		

Evaluare

Evaluarea unei expresii (propoziții) = determinarea valorii de adevăr a propoziției, într-o interpretare, prin aplicarea regulilor semantice

Exemplu

$$E = p \wedge (q \vee r) \quad \text{în interpretarea } I = \{ p = \text{true}, q = \text{true}, r = \text{false} \}$$

$$E = \text{true} \wedge (\text{true} \vee \text{false}) = \text{true} \wedge \text{true} = \text{true}$$

Logica propozițională – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Satisfiabilitate și validitate
- Derivabilitate
- Decidabilitate
- Inferență și reguli de inferență
- Demonstrație
- Rezoluție

Satisfiabilitate

Propoziția P este **satisfiabilă** $\Leftrightarrow P$ este adevărată (true) în cel puțin o interpretare

Propoziția P este **nesatisfiabilă** $\Leftrightarrow P$ este falsă (false) în toate interpretările

- Satisfiabilitatea unei propoziții se determină folosind **tabele de adevăr**

Exemple

- $p \wedge (q \vee r)$ satisfiabilă conform tabelei alăturate (întrucât este adevărată în 3 interpretări)
- $p \wedge \neg p$ nesatisfiabilă (se determină similar)

p	q	r	$p \wedge (q \vee r)$
true	true	true	true
true	true	false	true
true	false	true	true
true	false	false	false
false	true	true	false
false	true	false	false
false	false	true	false
false	false	false	false

Validitate

Propoziția P este **validă (tautologie)** \Leftrightarrow P este adevărată (true) în toate interpretările

Propoziția P este **invalidă** \Leftrightarrow P este falsă (false) în cel puțin o interpretare

- Validitatea unei propoziții se determină folosind **tabele de adevăr**

Exemple

- $(p \vee q) \vee (\neg q \vee r)$ validă conform tabelei alăturate (întrucât este adevărată în toate interpretările)
- $p \wedge (q \vee r)$ invalidă conform paginii anterioare (întrucât este falsă în 5 interpretări)

p	q	r	$(p \vee q) \vee (\neg q \vee r)$
true	true	true	true
true	true	false	true
true	false	true	true
true	false	false	true
false	true	true	true
false	true	false	true
false	false	true	true
false	false	false	true

Logica propozițională – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Satisfiabilitate și validitate
- Derivabilitate
- Decidabilitate
- Inferență și reguli de inferență
- Demonstrație
- Rezoluție

Derivabilitate

Mulțimea de propoziții Δ **derivă** propoziția P ($\Delta \vDash P$) \Leftrightarrow orice interpretare care satisface toate propozițiile din Δ satisface și propoziția P

- Derivabilitatea unei propoziții se determină folosind **tabele de adevăr**

Exemplu

- $\{ p, p \Rightarrow q \} \vDash q$ conform tabelei alăturate
(Singura interpretare care satisface p și $p \Rightarrow q$ este prima, și prima interpretare satisface și q)

p	q	$p \Rightarrow q$
true	true	true
true	false	false
false	true	true
false	false	true

Logica propozițională – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Satisfiabilitate și validitate
- Derivabilitate
- Decidabilitate
- Inferență și reguli de inferență
- Demonstrație
- Rezoluție

Decidabilitate

Logica propozițională este **decidabilă**: există o procedură care se termină pentru a decide dacă o propoziție este validă

- n propoziții atomice $\Rightarrow 2^n$ interpretări
- Pentru fiecare interpretare se evaluează propoziția și dacă rezultatul este mereu true, atunci propoziția este validă
- Întrucât trebuie să trecem prin 2^n interpretări, problema determinării validității unei propoziții este **NP-completă**
 \Rightarrow Pentru eficiență, se vor prefera metode bazate pe **reguli de inferență** (în dauna celor bazate pe tabele de adevăr)

Logica propozițională – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Satisfiabilitate și validitate
- Derivabilitate
- Decidabilitate
- Inferență și reguli de inferență
- Demonstrație
- Rezoluție

Inferență

Inferență = derivarea prin calcul a concluziilor unei multimi de premise

Regulă de inferență

P_1, P_2, \dots, P_n

P_i = premise (propoziții considerate adevărate)

_____ (id_regulă)
 C

C = concluzie (propoziție care derivă din multimea de premise)

Notări

$\Delta \models C$ - derivabilitate logică

$\Delta \vdash_{\text{id_regulă}} C$ - derivabilitate mecanică (folosind regula de inferență *id_regulă*)

Reguli de inferență

$$\frac{A \Rightarrow B}{\frac{A}{B}} \text{ (ModusPonens)}$$

$$\frac{A \vee B}{\frac{\neg A \vee C}{B \vee C}} \text{ (Rezoluție)}$$

$$\frac{A \Rightarrow B}{\frac{\neg B}{\neg A}} \text{ (ModusTollens)}$$

$$\frac{}{A \vee \neg A} \text{ (AxiomaA)}$$

Regulile fără premise se numesc **axiome**

Proprietăți ale regulilor de inferență

- **Consistență:** regula determină doar concluzii care derivă logic din premise
Dacă $\Delta \vdash_{\text{id_regulă}} C$, atunci $\Delta \vDash C$
- **Completitudine:** regula determină toate concluziile care derivă logic din premise
Dacă $\Delta \vDash C$, atunci $\Delta \vdash_{\text{id_regulă}} C$
- O regulă de inferență consistentă și completă ar fi suficientă pentru a deduce toate adevărurile din universul problemei (nici mai multe, nici mai puține)

Proprietăți Modus Ponens

- **Consistentă** (vezi tabela de adevăr)
- **Incompletă** (vezi exemplul de mai jos)

$$\frac{A \Rightarrow B}{\frac{A}{B}} \text{ (ModusPonens)}$$

Exemplu

$\{ p \Rightarrow q, q \Rightarrow r \} \models p \Rightarrow r$ (se poate verifica prin tabela de adevăr)

$\{ p \Rightarrow q, q \Rightarrow r \} \not\models_{\text{ModusPonens}} p \Rightarrow r$ (nu am informație despre vreo propoziție atomică)

Consecință: Modus Ponens se folosește în demonstrații împreună cu un set de axiome

Logica propozițională – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Satisfiabilitate și validitate
- Derivabilitate
- Decidabilitate
- Inferență și reguli de inferență
- Demonstrație
- Rezoluție

Demonstrație

Teoremă = propoziție care se dorește demonstrată

Demonstrație (a unei teoreme) = secvență de propoziții adevărate, încheiată cu teorema, conținând:

- Premise
- Instanțe ale axiomelor
- Rezultate ale aplicării regulilor de inferență asupra propozițiilor anterioare în secvență

Demonstrație prin reducere la absurd = secvență de propoziții considerate adevărate, încheiată cu o propoziție nesatisfiabilă, conținând

- Aceleași elemente de mai sus +
- La premise se adaugă $\neg C$ (negația teoremei, singurul posibil neadevăr, cauza pentru care o propoziție falsă a fost determinată ca fiind adevărată)

Demonstrație cu Modus Ponens - Exemplu

Să se demonstreze $\{p \Rightarrow q, q \Rightarrow r\} \vdash p \Rightarrow r$, folosind regulile de inferență:

$$\boxed{\begin{array}{c} A \Rightarrow B \\ \frac{A}{B} \text{ (ModusPonens)} \end{array}}$$

$$\boxed{\text{Axiome}} \quad \boxed{\frac{}{(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))} \text{ (DistriImp)}}$$

1. $p \Rightarrow q$ (Premisă)
2. $q \Rightarrow r$ (Premisă)
3. $(q \Rightarrow r) \Rightarrow (p \Rightarrow (q \Rightarrow r))$ (IntroImp)
4. $p \Rightarrow (q \Rightarrow r)$ (ModusPonens 2,3)
5. $(p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r))$ (DistriImp)
6. $(p \Rightarrow q) \Rightarrow (p \Rightarrow r)$ (ModusPonens 4,5)
7. $p \Rightarrow r$ (ModusPonens 1,6)

Strategii de control

- În timpul unei demonstrații, se întâmplă să existe mai multe reguli aplicabile simultan
Strategia de control dictează care reguli au prioritate în asemenea situații.

Strategii de control uzuale

- **Forward chaining**

- Se pleacă de la premise către scop (ceea ce trebuie demonstrat)
- Se derivează toate concluziile posibile (în orice ordine)
- Oprit la satisfacerea scopului

- **Backward chaining**

- Se pleacă de la scop către premise
- Se folosesc doar regulile care pot contribui la satisfacerea scopului
- Premisele acestor reguli devin noi scopuri, etc.

Logica propozițională – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Satisfiabilitate și validitate
- Derivabilitate
- Decidabilitate
- Inferență și reguli de inferență
- Demonstrație
- Rezoluție

Proprietăți Rezoluție

- **Consistentă** (vezi tabela de adevăr)
- **Completă** (demonstrația nu face obiectul cursului)
 - în sensul că pot determina valoarea de adevăr a unei propoziții date, dar nu pot genera toate propozițiile adevărate)

$$\frac{A \vee B \\ \neg A \vee C}{B \vee C} \text{ (Rezoluție)}$$

Consecință: Un demonstrator de teoreme se poate baza pe rezoluție.

Observație: Pentru a folosi rezoluția în demonstrații, propozițiile sunt aduse la o formă normală numita **forma clauzală** (acest lucru este mereu posibil).

Forma clauzală a unei propoziții

Literal = $p \mid \neg p$ (propoziții atomice sau negațiile lor)

Clauză = multime de literali aflați în relația de disjuncție

- $\{ p, q, \neg r \}$ echivalent cu $p \vee q \vee \neg r$

Clauză Horn = clauză în care un singur literal este nenegat

- $\{ p, \neg q, \neg r \}$ echivalent cu $p \vee \neg q \vee \neg r$ și cu $q \wedge r \Rightarrow p$

Propoziție în formă clauzală = multime de clauze aflate în relația de conjuncție

- $\{ \{p\}, \{q, \neg r\}, \{\neg p, r\} \}$ echivalent cu $p \wedge (q \vee \neg r) \wedge (\neg p \vee r)$

Rezoluție în forma clauzală

Forma simplă

$$\frac{\{p, q\} \quad \{\neg p, r\}}{\{q, r\}} \text{ (Rezoluție)}$$

Forma generală

$$\frac{\{q_1, \dots, p, \dots q_m\} \quad \{r_1, \dots, \neg p, \dots r_n\}}{\{q_1 \dots q_m, r_1 \dots r_n\}} \text{ (Rezoluție)}$$

Contradicție

$$\frac{\{p\} \quad \{\neg p\}}{\{\}} \text{ (Rezoluție)}$$

Rezolvent vid: indică existența unei contradicții între premise

Cazuri particulare ale Rezoluției

Modus Ponens

$$\frac{\{\neg p, q\}}{\{p\}} \quad (Rezoluție)$$

$$A \Rightarrow B$$

$$\frac{A}{B}$$

(*ModusPonens*)

Modus Tollens

$$\frac{\{\neg p, q\}}{\{\neg q\}} \quad (Rezoluție)$$

$$A \Rightarrow B$$

$$\frac{\neg B}{\neg A}$$

(*ModusTollens*)

Tranzitivitatea implicației

$$\frac{\{\neg p, q\}}{\{\neg q, r\}} \quad (Rezoluție)$$

$$A \Rightarrow B$$

$$\frac{B \Rightarrow C}{A \Rightarrow C}$$

(*TranziImp*)

Diferența de notație (A, B vs. p, q) indică faptul că în cazul formei clauzale lucrăm doar cu propoziții atomice și negațiile lor

Demonstrații bazate pe Rezoluție

Fie P o propoziție în formă clauzală. Pentru a demonstra

- P nesatisfiabilă
- P derivabilă din premise
- P validă

Demonstrații bazate pe Rezoluție

Fie P o propoziție în formă clauzală. Pentru a demonstra

- **P nesatisfiabilă**
 - Pornesc cu premisa P
 - Derivez clauza vidă
- **P derivabilă din premise**
- **P validă**

Demonstrații bazate pe Rezoluție

Fie P o propoziție în formă clauzală. Pentru a demonstra

- **P nesatisfiabilă**
 - Pornesc cu premisa P
 - Derivez clauza vidă
- **P derivabilă din premise**
 - Introduc $\neg P$ în premise \leftarrow reducere la absurd
 - Derivez clauza vidă
- **P validă**

Demonstrații bazate pe Rezoluție

Fie P o propoziție în formă clauzală. Pentru a demonstra

- **P nesatisfiabilă**

- Pornesc cu premisa P
- Derivez clauza vidă

- **P derivabilă din premise**

- Introduc $\neg P$ în premise
- Derivez clauza vidă

← reducere la absurd

Se observă că nu avem o metodă de a demonstra satisfaabilitatea / invaliditatea

- **P validă**

- Pornesc cu premisa $\neg P$
- Derivez clauza vidă

Demonstrație cu Rezoluție – Exemplu

Să se demonstreze $\{p \Rightarrow q, q \Rightarrow r\} \vdash p \Rightarrow r$, folosind doar Rezoluția.

Demonstrație prin reducere la absurd

- Introduc în premise $\neg(p \Rightarrow r) = \neg(\neg p \vee r) = p \wedge \neg r = \{\{p\}, \{\neg r\}\}$

1. $\{\neg p, q\}$ *(Premisă)*
2. $\{\neg q, r\}$ *(Premisă)*
3. $\{p\}$ *(Concluzie negată)*
4. $\{\neg r\}$ *(Concluzie negată)*
5. $\{q\}$ *(Rezoluție 1,3)*
6. $\{r\}$ *(Rezoluție 2,5)*
7. $\{\}$ *(Rezoluție 4,6)*

Logica cu predicate de ordinul întâi – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Exemple
- Decidabilitate
- Forma clauzală
- Algoritm de aducere la forma clauzală
- Unificare

Ontologie

- Universul problemei este descris prin **obiecte, proprietățile lor și relațiile** dintre ele

Exemple

$\forall X, Y. ((iepure(X) \wedge \text{țestoasă}(Y)) \Rightarrow \text{mai_rapid}(X, Y)).$
iepure(bugs).
țestoasă(leo).
mai_rapid(bugs, leo).

Iepurii sunt mai rapizi decât broaștele țestoase.
Bugs e iepure.
Leo e broască țestoasă.
Bugs e mai rapid decât Leo.

- Propozițiile sunt structurate** astfel încât să surprindă obiectele din acest univers și relațiile dintre ele

- În logica propozițională, ultima propoziție nu era o consecință logică a celorlalte
- În logica cu predicate de ordinul întâi este

Logica cu predicate de ordinul întâi – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Exemple
- Decidabilitate
- Forma clauzală
- Algoritm de aducere la forma clauzală
- Unificare

Elemente de sintaxă

- **Constante:** athos, porthos, aramis (cu litere mici)
 - **Variabile:** Mușchetar, Cardinal (cu litere mari)
 - **Funcții:** regină(frânță) (calculează o constantă)
 - **Predicate:** respectă(porthos,athos)
 - **Coneective:** \neg , \wedge , \vee , \Rightarrow , \Leftrightarrow (după cum scade prioritatea)
 - **Cuantificatori:** \forall , \exists
 - **Egalitate:** =
- } Asociativi la dreapta

Sintaxă

termen = constantă | variabilă | funcție(termen,... termen)

propoziție atomică = predicat(termen,... termen) | termen = termen

propoziție = propoziție atomică | (propoziție) | \neg propoziție |
propoziție conectivă_binară propoziție | cuantificator var₁... var_n.propoziție

- Termenii reprezintă obiecte
- Propozițiile reprezintă proprietăți și relații

Logica cu predicate de ordinul întâi – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Exemple
- Decidabilitate
- Forma clauzală
- Algoritm de aducere la forma clauzală
- Unificare

Semantică

Interpretare = pereche (D, A) , unde

- D = domeniu (specifică obiectele din problemă)
- A = multime de asocieri pentru termeni și propoziții
 - Constantelor și funcțiilor li se asociază câte un obiect din D ($c \in D, f : D^n \rightarrow D$)
 - Predicatelor li se asociază o valoare de adevăr (true sau false) ($p : D^n \rightarrow \{\text{true}, \text{false}\}$)

Reguli semantice

Cele de la logica propozițională (pentru conective) +

- | | | |
|---------------------------------------|--------------------|---|
| $\forall X.\text{prop} = \text{true}$ | dacă și numai dacă | nu există $d \in D$ a.î. $\text{prop}_{[d/X]} = \text{false}$ |
| $\exists X.\text{prop} = \text{true}$ | dacă și numai dacă | există $d \in D$ a.î. $\text{prop}_{[d/X]} = \text{true}$ |

Logica cu predicate de ordinul întâi – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Exemple
- Decidabilitate
- Forma clauzală
- Algoritm de aducere la forma clauzală
- Unificare

Exemple



1. Vribia mălai visează.
2. Unele vrăbii visează mălai.
3. Nu toate vrăbiile visează mălai.
4. Nicio vrabie nu visează mălai.
5. Numai vrăbiile visează mălai.
6. Toate și numai vrăbiile visează mălai.
7. Vivi este o vrabie care visează mălai.

Exemple



1. Vribia mălai visează.
 $\forall X \cdot (\text{vrabie}(X) \Rightarrow \text{visează}(X, \text{mălai}))$
2. Unele vrăbii visează mălai.
3. Nu toate vrăbiile visează mălai.
4. Nicio vrabie nu visează mălai.
5. Numai vrăbiile visează mălai.
6. Toate și numai vrăbiile visează mălai.
7. Vivi este o vrabie care visează mălai.

Exemple



1. Vribia mălai visează.
 $\forall X \cdot (\text{vrabie}(X) \Rightarrow \text{visează}(X, \text{mălai}))$
2. Unele vrăbii visează mălai.
 $\exists X \cdot (\text{vrabie}(X) \wedge \text{visează}(X, \text{mălai}))$
3. Nu toate vrăbiile visează mălai.
4. Nicio vrabie nu visează mălai.
5. Numai vrăbiile visează mălai.
6. Toate și numai vrăbiile visează mălai.
7. Vivi este o vrabie care visează mălai.

Exemple



1. Vribia mălai visează.
 $\forall X \cdot (\text{vrabie}(X) \Rightarrow \text{visează}(X, \text{mălai}))$
2. Unele vrăbii visează mălai.
 $\exists X \cdot (\text{vrabie}(X) \wedge \text{visează}(X, \text{mălai}))$
3. Nu toate vrăbiile visează mălai.
 $\neg(\forall X \cdot (\text{vrabie}(X) \Rightarrow \text{visează}(X, \text{mălai})))$ sau $\exists X \cdot (\text{vrabie}(X) \wedge \neg \text{visează}(X, \text{mălai}))$
4. Nicio vrabie nu visează mălai.
5. Numai vrăbiile visează mălai.
6. Toate și numai vrăbiile visează mălai.
7. Vivi este o vrabie care visează mălai.

Exemple



1. Vribia mălai visează.
 $\forall X \cdot (\text{vrabie}(X) \Rightarrow \text{visează}(X, \text{mălai}))$
2. Unele vrăbii visează mălai.
 $\exists X \cdot (\text{vrabie}(X) \wedge \text{visează}(X, \text{mălai}))$
3. Nu toate vrăbiile visează mălai.
 $\neg(\forall X \cdot (\text{vrabie}(X) \Rightarrow \text{visează}(X, \text{mălai})))$ sau $\exists X \cdot (\text{vrabie}(X) \wedge \neg \text{visează}(X, \text{mălai}))$
4. Nicio vrabie nu visează mălai.
 $\neg(\exists X \cdot (\text{vrabie}(X) \wedge \text{visează}(X, \text{mălai})))$ sau $\forall X \cdot (\text{vrabie}(X) \Rightarrow \neg \text{visează}(X, \text{mălai}))$
5. Numai vrăbiile visează mălai.
6. Toate și numai vrăbiile visează mălai.
7. Vivi este o vrabie care visează mălai.

Exemple



1. Vribia mălai visează.
 $\forall X \cdot (\text{vrabie}(X) \Rightarrow \text{visează}(X, \text{mălai}))$
2. Unele vrăbii visează mălai.
 $\exists X \cdot (\text{vrabie}(X) \wedge \text{visează}(X, \text{mălai}))$
3. Nu toate vrăbiile visează mălai.
 $\neg(\forall X \cdot (\text{vrabie}(X) \Rightarrow \text{visează}(X, \text{mălai})))$ sau $\exists X \cdot (\text{vrabie}(X) \wedge \neg \text{visează}(X, \text{mălai}))$
4. Nicio vrabie nu visează mălai.
 $\neg(\exists X \cdot (\text{vrabie}(X) \wedge \text{visează}(X, \text{mălai})))$ sau $\forall X \cdot (\text{vrabie}(X) \Rightarrow \neg \text{visează}(X, \text{mălai}))$
5. Numai vrăbiile visează mălai.
 $\forall X \cdot (\text{visează}(X, \text{mălai}) \Rightarrow \text{vrabie}(X))$
6. Toate și numai vrăbiile visează mălai.
7. Vivi este o vrabie care visează mălai.

Exemple



1. Vribia mălai visează.
 $\forall X \cdot (\text{vrabie}(X) \Rightarrow \text{visează}(X, \text{mălai}))$
2. Unele vrăbii visează mălai.
 $\exists X \cdot (\text{vrabie}(X) \wedge \text{visează}(X, \text{mălai}))$
3. Nu toate vrăbiile visează mălai.
 $\neg(\forall X \cdot (\text{vrabie}(X) \Rightarrow \text{visează}(X, \text{mălai})))$ sau $\exists X \cdot (\text{vrabie}(X) \wedge \neg \text{visează}(X, \text{mălai}))$
4. Nicio vrabie nu visează mălai.
 $\neg(\exists X \cdot (\text{vrabie}(X) \wedge \text{visează}(X, \text{mălai})))$ sau $\forall X \cdot (\text{vrabie}(X) \Rightarrow \neg \text{visează}(X, \text{mălai}))$
5. Numai vrăbiile visează mălai.
 $\forall X \cdot (\text{visează}(X, \text{mălai}) \Rightarrow \text{vrabie}(X))$
6. Toate și numai vrăbiile visează mălai.
 $\forall X \cdot (\text{visează}(X, \text{mălai}) \Leftrightarrow \text{vrabie}(X))$
7. Vivi este o vrabie care visează mălai.

Exemple



1. Vribia mălai visează.
 $\forall X \cdot (\text{vrabie}(X) \Rightarrow \text{visează}(X, \text{mălai}))$
2. Unele vrăbii visează mălai.
 $\exists X \cdot (\text{vrabie}(X) \wedge \text{visează}(X, \text{mălai}))$
3. Nu toate vrăbiile visează mălai.
 $\neg(\forall X \cdot (\text{vrabie}(X) \Rightarrow \text{visează}(X, \text{mălai})))$ sau $\exists X \cdot (\text{vrabie}(X) \wedge \neg \text{visează}(X, \text{mălai}))$
4. Nicio vrabie nu visează mălai.
 $\neg(\exists X \cdot (\text{vrabie}(X) \wedge \text{visează}(X, \text{mălai})))$ sau $\forall X \cdot (\text{vrabie}(X) \Rightarrow \neg \text{visează}(X, \text{mălai}))$
5. Numai vrăbiile visează mălai.
 $\forall X \cdot (\text{visează}(X, \text{mălai}) \Rightarrow \text{vrabie}(X))$
6. Toate și numai vrăbiile visează mălai.
 $\forall X \cdot (\text{visează}(X, \text{mălai}) \Leftrightarrow \text{vrabie}(X))$
7. Vivi este o vrabie care visează mălai.
 $\text{vrabie}(\text{vivi}) \wedge \text{visează}(\text{vivi}, \text{mălai})$

Greșeli frecvente

- În general:
 - \forall se folosește împreună cu \Rightarrow
 - \exists se folosește împreună cu \wedge
- $\forall X . (vrabie(X) \wedge visează(X, mălai))$
?
- $\exists X . (vrabie(X) \Rightarrow visează(X, mălai))$
?

Greșeli frecvente

- În general:
 - \forall se folosește împreună cu \Rightarrow
 - \exists se folosește împreună cu \wedge
- $\forall X \cdot (\text{vrabie}(X) \wedge \text{visează}(X, \text{mălai}))$
înseamnă că toată lumea e vrabie și toată lumea visează mălai
- $\exists X \cdot (\text{vrabie}(X) \Rightarrow \text{visează}(X, \text{mălai}))$
este o propoziție adevărată și dacă există cineva care nu e vrabie

Proprietăți cuantificatori și conective

- **(Ne)Comutativitate**

$\forall X \cdot \forall Y$ este totuna cu $\forall Y \cdot \forall X$ și se scrie prescurtat $\forall X, Y$.

$\exists X \cdot \exists Y$ este totuna cu $\exists Y \cdot \exists X$ și se scrie prescurtat $\exists X, Y$.

$\exists X \cdot \forall Y$ NU este totuna cu $\forall Y \cdot \exists X$

- $\exists X \cdot \forall Y$ visează (X, Y) înseamnă că

- $\forall Y \cdot \exists X$ visează (X, Y) înseamnă că

- **Dualitate (se pot scrie unul/una în funcție de altul/alta)**

$$\forall X \cdot p = \neg(\exists X \cdot \neg p) \quad p \vee q = \neg(\neg p \wedge \neg q)$$

$$\exists X \cdot p = \neg(\forall X \cdot \neg p) \quad p \wedge q = \neg(\neg p \vee \neg q)$$

$$\neg(\forall X \cdot p) = \exists X \cdot \neg p \quad \neg(p \vee q) = \neg p \wedge \neg q$$

$$\neg(\exists X \cdot p) = \forall X \cdot \neg p \quad \neg(p \wedge q) = \neg p \vee \neg q$$

Proprietăți cuantificatori și conective

- **(Ne)Comutativitate**

$\forall X \cdot \forall Y$ este totuna cu $\forall Y \cdot \forall X$ și se scrie prescurtat $\forall X, Y$.

$\exists X \cdot \exists Y$ este totuna cu $\exists Y \cdot \exists X$ și se scrie prescurtat $\exists X, Y$.

$\exists X \cdot \forall Y$ NU este totuna cu $\forall Y \cdot \exists X$

- $\exists X \cdot \forall Y \cdot \text{visează}(X, Y)$ inseamnă că există cineva care visează la toată lumea
- $\forall Y \cdot \exists X \cdot \text{visează}(X, Y)$ inseamnă că la oricine visează măcar cineva

- **Dualitate (se pot scrie unul/una în funcție de altul/alta)**

$$\forall X \cdot p = \neg(\exists X \cdot \neg p) \quad p \vee q = \neg(\neg p \wedge \neg q)$$

$$\exists X \cdot p = \neg(\forall X \cdot \neg p) \quad p \wedge q = \neg(\neg p \vee \neg q)$$

$$\neg(\forall X \cdot p) = \exists X \cdot \neg p \quad \neg(p \vee q) = \neg p \wedge \neg q$$

$$\neg(\exists X \cdot p) = \forall X \cdot \neg p \quad \neg(p \wedge q) = \neg p \vee \neg q$$

Definiții moștenite de la logica propozițională

- **Evaluare:** determinarea valorii de adevăr a lui P (într-o interpretare) prin aplicarea regulilor semantice
- **Satisfiabilitate:** $P = \text{true}$ în cel puțin o interpretare
- **Validitate:** $P = \text{true}$ în toate interpretările
- **Derivabilitate:** $P = \text{true}$ în toate interpretările care satisfac premisele
- **Inferență:** derivare prin calcul a concluziilor unei multimi de premise
- **Demonstrație:** secvență de propoziții adevărate încheiată cu propoziția care se dorea demonstrată (teorema)

Logica cu predicate de ordinul întâi – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Exemple
- Decidabilitate
- Forma clauzală
- Algoritm de aducere la forma clauzală
- Unificare

Decidabilitate

Logica cu predicate de ordinul întâi este **semidecidabilă**

- O infinitate de domenii posibile \Rightarrow o infinitate de interpretări (nu le putem enumera)
 - Se poate demonstra validitatea (folosind, de exemplu, Rezoluția)
 - Nu se poate demonstra invaliditatea
- Compromis între decidabilitate și puterea de reprezentare
 - **Logica propozițională:** decidabilă / putere de reprezentare redusă
 - **Logica cu predicate de ordinul întâi:** semidecidabilă / putere de reprezentare sporită

Logica cu predicate de ordinul întâi – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Exemple
- Decidabilitate
- Forma clauzală
- Algoritm de aducere la forma clauzală
- Unificare

Forma clauzală a unei propoziții

Literal, Clauză, Clauză Horn, Propoziție în formă clauzală

- se definesc ca în cazul logicii propoziționale
(doar că diferă ceea ce se înțelege prin propoziție atomică)

Exemplu de clauză

{ prieten(X,porthos), \neg fricos(X) } echivalent cu prieten(X,porthos) V \neg fricos(X)

Observație: forma clauzală se mai numește și **forma normal conjunctivă**

Logica cu predicate de ordinul întâi – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Exemple
- Decidabilitate
- Forma clauzală
- Algoritm de aducere la forma clauzală
- Unificare

Algoritm de aducere la forma clauzală (I)

1. Elimină implicațiile

$a \Rightarrow b$ devine $\neg a \vee b$

2. Mută negațiile spre interior

$$\neg(a \vee b) = \neg a \wedge \neg b$$

$$\neg(a \wedge b) = \neg a \vee \neg b$$

$$\neg(\forall X \cdot p) = \exists X \cdot \neg p$$

$$\neg(\exists X \cdot p) = \forall X \cdot \neg p$$

$$\neg(\neg p) = p$$

3. Redenumește variabilele cuantificate a.î. să nu existe cuantificatori diferenți care folosesc același nume de variabilă

(ex: $\forall X \cdot p(X) \wedge \forall X \cdot q(X) \vee \exists X \cdot r(X)$ devine $\forall X \cdot p(X) \wedge \forall Y \cdot q(Y) \vee \exists Z \cdot r(Z)$)

4. Forma prenex: Mută toți cuantificatorii la începutul propoziției, în aceeași ordine

(ex: $\forall X \cdot p(X) \wedge \forall Y \cdot q(Y) \vee \exists Z \cdot r(Z)$ devine $\forall X \cdot \forall Y \cdot \exists Z \cdot (p(X) \wedge q(Y) \vee r(Z))$)

Algoritm de aducere la forma clauzală (II)

5. **Skolemizare: Elimină cuantificatorii existențiali prin înlocuire cu**
 - O **constantă** - dacă acest \exists nu era precedat de niciun \forall
 - O **funcție** de toate variabilele cuantificate universal care îl precedau - altfel
(ex: $\forall X \cdot \forall Y \cdot \exists Z \cdot (p(X) \wedge q(Y) \vee r(Z))$ devine $\forall X \cdot \forall Y \cdot (p(X) \wedge q(Y) \vee r(f(X, Y)))$)
6. **Elimină cuantificatorii universali (ce rămâne e automat cuantificat universal)**
(ex: $\forall X \cdot \forall Y \cdot (p(X) \wedge q(Y) \vee r(f(X, Y)))$ devine $p(X) \wedge q(Y) \vee r(f(X, Y))$)
7. **Distribuie \vee prin \wedge**
 $(a \wedge b) \vee c$ devine $(a \vee c) \wedge (b \vee c)$
8. **Transformă expresiile în clauze**

Aducere la forma clauzală – Exemplu (I)

“Oricine rezolvă toate laboratoarele este apreciat de cineva.”

$$\forall X. (\forall Y. (\text{laborator}(Y) \Rightarrow \text{rezolvă}(X, Y)) \Rightarrow \exists Y. \text{apreciază}(Y, X))$$

1. Elimină implicațiile

$$\forall X. (\neg(\forall Y. \neg\text{laborator}(Y) \vee \text{rezolvă}(X, Y)) \vee \exists Y. \text{apreciază}(Y, X))$$

2. Mută negațiile spre interior

$$\forall X. (\exists Y. \neg(\neg\text{laborator}(Y) \vee \text{rezolvă}(X, Y)) \vee \exists Y. \text{apreciază}(Y, X))$$

$$\forall X. (\exists Y. (\text{laborator}(Y) \wedge \neg\text{rezolvă}(X, Y)) \vee \exists Y. \text{apreciază}(Y, X))$$

3. Redenumește variabilele cuantificate

$$\forall X. (\exists Y. (\text{laborator}(Y) \wedge \neg\text{rezolvă}(X, Y)) \vee \exists Z. \text{apreciază}(Z, X))$$

4. Mută cuantificatorii la început

$$\forall X. \exists Y. \exists Z. ((\text{laborator}(Y) \wedge \neg\text{rezolvă}(X, Y)) \vee \text{apreciază}(Z, X))$$

Aducere la forma clauzală – Exemplu (II)

“Oricine rezolvă toate laboratoarele este apreciat de cineva.”

$$\forall X \cdot \exists Y \cdot \exists Z \cdot ((\text{laborator}(Y) \wedge \neg \text{rezolvă}(X, Y)) \vee \text{apreciază}(Z, X))$$

5. **Elimină cuantificatorii existențiali**

$$\forall X \cdot (\text{laborator}(f_y(X)) \wedge \neg \text{rezolvă}(X, f_y(X))) \vee \text{apreciază}(f_z(X), X)$$

6. **Elimină cuantificatorii universali**

$$(\text{laborator}(f_y(X)) \wedge \neg \text{rezolvă}(X, f_y(X))) \vee \text{apreciază}(f_z(X), X)$$

7. **Distribuie \vee prin \wedge**

$$(\text{laborator}(f_y(X)) \vee \text{apreciază}(f_z(X), X)) \wedge (\neg \text{rezolvă}(X, f_y(X)) \vee \text{apreciază}(f_z(X), X))$$

8. **Transformă expresiile în clauze**

$$\{ \text{laborator}(f_y(X)), \text{apreciază}(f_z(X), X) \}$$

$$\{ \neg \text{rezolvă}(X, f_y(X)), \text{apreciază}(f_z(X), X) \}$$

Logica cu predicate de ordinul întâi – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Exemple
- Decidabilitate
- Forma clauzală
- Algoritm de aducere la forma clauzală
- Unificare

Unificare

Unificarea a 2 expresii = găsirea celei mai generale substituții

$$S = \{e_1 / \text{var}_1, e_2 / \text{var}_2, \dots e_n / \text{var}_n\}$$

pentru variabilele din cele 2 expresii astfel încât, în urma substituției, cele 2 expresii devin una și aceeași

Utilitate

$$\frac{\begin{array}{c} \{\neg \text{cal}(X), \text{măncă}(X, \text{ovăz})\} \\ \{\text{cal}(X)\} \end{array}}{\text{măncă}(X, \text{ovăz})} \quad (\text{Rezoluție})$$

$$\frac{\begin{array}{c} \{\neg \text{cal}(X), \text{măncă}(X, \text{ovăz})\} \\ \{\text{cal}(\text{rocinante})\} \end{array}}{?} \quad (\text{Rezoluție})$$

Mulțumită unificării, putem conchide și că Rocinante măncă ovăz.

Unificare – Exemple

- $P = \text{măncă}(\text{vivi}, X)$
 $Q = \text{hrănește}(\text{vivi}, \text{puiu})$
- $P = \text{măncă}(\text{vivi}, X)$
 $Q = \text{măncă}(X, \text{mălai})$
- $P = \text{măncă}(\text{Canibal}, \text{tata}(\text{Canibal}))$
 $Q = \text{măncă}(\text{tata}(X), Y)$

Unificare – Exemple

- $P = \text{măncă}(\text{vivi}, X)$
 $Q = \text{hrănește}(\text{vivi}, \text{puiu})$
- $P = \text{măncă}(\text{vivi}, X)$
 $Q = \text{măncă}(X, \text{mălai})$
- $P = \text{măncă}(\text{Canibal}, \text{tata}(\text{Canibal}))$
 $Q = \text{măncă}(\text{tata}(X), Y)$

Nu unifică!
(nu au același predicat)

Unificare – Exemple

- $P = \text{măncă}(\text{vivi}, X)$
 $Q = \text{hrănește}(\text{vivi}, \text{puiu})$

Nu unifică!
(nu au același predicat)

- $P = \text{măncă}(\text{vivi}, X)$
 $Q = \text{măncă}(X, \text{mălai})$

Nu unifică!
(vivi/X, mălai/X, vivi nu unifică cu mălai)

- $P = \text{măncă}(\text{Canibal}, \text{tata}(\text{Canibal}))$
 $Q = \text{măncă}(\text{tata}(X), Y)$

Unificare – Exemple

- $P = \text{măncă}(\text{vivi}, X)$
 $Q = \text{hrănește}(\text{vivi}, \text{puiu})$

Nu unifică!
(nu au același predicat)

- $P = \text{măncă}(\text{vivi}, X)$
 $Q = \text{măncă}(X, \text{mălai})$

Nu unifică!
(vivi/X, mălai/X, vivi nu unifică cu mălai)

- $P = \text{măncă}(\text{Canibal}, \text{tata}(\text{Canibal}))$
 $Q = \text{măncă}(\text{tata}(X), Y)$

Aşa Da
Aşa NU

$S = \{ \text{tata}(X) / \text{Canibal}, \text{tata}(\text{Canibal}) / Y \} \rightarrow$
 $S = \{ \text{tata}(X) / \text{Canibal}, \text{tata}(\text{tata}(X)) / Y \}$
 $\text{subst}(S, P) = \text{măncă}(\text{tata}(X), \text{tata}(\text{tata}(X)))$
 $\text{subst}(S, Q) = \text{măncă}(\text{tata}(X), \text{tata}(\text{tata}(X)))$
 $\text{subst}(S, P) = \text{măncă}(\text{tata}(X), \text{bunicul}(X))$

Reguli folosite în algoritmul de unificare

- O **variabilă** X unifică cu un **termen** t ($S = \{t / X\}$) dacă și numai dacă
 - $t = X$ sau
 - t nu conține X (occurs check – pentru a evita legări ciclice)
- **2 constante / 2 funcții / o funcție și o constantă** unifică ($S = \{ \}$) dacă și numai dacă se evaluatează la același obiect
- **2 propoziții atomice** unifică dacă și numai dacă sunt aplicații ale aceluiași predicat asupra către n termeni care unifică recursiv

Observație: Rezoluția lucrează doar cu propoziții atomice și negațiile lor, nu avem nevoie să unificăm și propoziții compuse

Rezoluție pentru clauze Horn

$$\frac{p_1 \wedge \dots \wedge p_m \rightarrow p \\ q_1 \wedge \dots \wedge p' \wedge \dots \wedge q_n \rightarrow q \\ \text{unificare}(p, p') = S}{\text{subst}(S, p_1 \wedge \dots \wedge p_m, q_1 \wedge \dots \wedge q_n \rightarrow q)} \quad (\text{Rezoluție})$$

Observație: Pentru a simplifica demonstrațiile, propozițiile în Prolog sunt restricționate la clauze Horn

Comparație LP – LPOI

	Logica propozițională	Logica cu predicate de ordinul întâi
Ontologie		
Sintaxă		
Semantică		
Inferență		
Complexitate		

Comparație LP – LPOI

	Logica propozițională	Logica cu predicate de ordinul întâi
Ontologie	Fapte (propoziții declarative)	Obiecte, proprietăți, relații
Sintaxă		
Semantică		
Inferență		
Complexitate		

Comparație LP – LPOI

	Logica propozițională	Logica cu predicate de ordinul întâi
Ontologie	Fapte (propoziții declarative)	Obiecte, proprietăți, relații
Sintaxă	Atomi, conective	Atomi, conective, termeni structurați, variabile cuantificate
Semantică		
Inferență		
Complexitate		

Comparație LP – LPOI

	Logica propozițională	Logica cu predicate de ordinul întâi
Ontologie	Fapte (propoziții declarative)	Obiecte, proprietăți, relații
Sintaxă	Atomi, conective	Atomi, conective, termeni structurați, variabile cuantificate
Semantică	Interpretări (în tabele de adevăr)	Interpretări (complexe)
Inferență		
Complexitate		

Comparație LP – LPOI

	Logica propozițională	Logica cu predicate de ordinul întâi
Ontologie	Fapte (propoziții declarative)	Obiecte, proprietăți, relații
Sintaxă	Atomi, conective	Atomi, conective, termeni structurați, variabile cuantificate
Semantică	Interpretări (în tabele de adevăr)	Interpretări (complexe)
Inferență	Rezoluție	Rezoluție + unificare
Complexitate		

Comparație LP – LPOI

	Logica propozițională	Logica cu predicate de ordinul întâi
Ontologie	Fapte (propoziții declarative)	Obiecte, proprietăți, relații
Sintaxă	Atomi, conective	Atomi, conective, termeni structurați, variabile cuantificate
Semantică	Interpretări (în tabele de adevăr)	Interpretări (complexe)
Inferență	Rezoluție	Rezoluție + unificare
Complexitate	Decidabilă dar NP-completă	Semidecidabilă

Rezumat

Satisfiabilitate

Validitate

Derivabilitate

Inferență

Reguli de inferență

Regulă consistentă

Regulă completă

Demonstrație

Strategii de control

Formă clauzală

Clauză Horn

Demonstrabile prin rezoluție

Nedemonstrabile prin rezoluție

Unificare

Rezumat

Satisfiabilitate: $P = \text{true}$ în cel puțin o interpretare

Validitate

Derivabilitate

Inferență

Reguli de inferență

Regulă consistentă

Regulă completă

Demonstrație

Strategii de control

Formă clauzală

Clauză Horn

Demonstrabile prin rezoluție

Nedemonstrabile prin rezoluție

Unificare

Rezumat

Satisfiabilitate: $P = \text{true}$ în cel puțin o interpretare

Validitate: $P = \text{true}$ în toate interpretările

Derivabilitate

Inferență

Reguli de inferență

Regulă consistentă

Regulă completă

Demonstrație

Strategii de control

Formă clauzală

Clauză Horn

Demonstrabile prin rezoluție

Nedemonstrabile prin rezoluție

Unificare

Rezumat

Satisfiabilitate: $P = \text{true}$ în cel puțin o interpretare

Validitate: $P = \text{true}$ în toate interpretările

Derivabilitate: $P = \text{true}$ în toate interpretările care satisfac premisele ($\Delta \models P$)

Inferență

Reguli de inferență

Regulă consistentă

Regulă completă

Demonstrație

Strategii de control

Formă clauzală

Clauză Horn

Demonstrabile prin rezoluție

Nedemonstrabile prin rezoluție

Unificare

Rezumat

Satisfiabilitate: $P = \text{true}$ în cel puțin o interpretare

Validitate: $P = \text{true}$ în toate interpretările

Derivabilitate: $P = \text{true}$ în toate interpretările care satisfac premisele ($\Delta \models P$)

Inferență: derivare prin calcul a concluziilor unei multimi de premise ($\Delta \vdash_{\text{id_regulă}} P$)

Reguli de inferență

Regulă consistentă

Regulă completă

Demonstrație

Strategii de control

Formă clauzală

Clauză Horn

Demonstrabile prin rezoluție

Nedemonstrabile prin rezoluție

Unificare

Rezumat

Satisfiabilitate: $P = \text{true}$ în cel puțin o interpretare

Validitate: $P = \text{true}$ în toate interpretările

Derivabilitate: $P = \text{true}$ în toate interpretările care satisfac premisele ($\Delta \models P$)

Inferență: derivare prin calcul a concluziilor unei multimi de premise ($\Delta \vdash_{\text{id_regulă}} P$)

Reguli de inferență: Modus Ponens, Modus Tollens, Rezoluție, axiome

Regulă consistentă

Regulă completă

Demonstrație

Strategii de control

Formă clauzală

Clauză Horn

Demonstrabile prin rezoluție

Nedemonstrabile prin rezoluție

Unificare

Rezumat

Satisfiabilitate: $P = \text{true}$ în cel puțin o interpretare

Validitate: $P = \text{true}$ în toate interpretările

Derivabilitate: $P = \text{true}$ în toate interpretările care satisfac premisele ($\Delta \models P$)

Inferență: derivare prin calcul a concluziilor unei multimi de premise ($\Delta \vdash_{\text{id_regulă}} P$)

Reguli de inferență: Modus Ponens, Modus Tollens, Rezoluție, axiome

Regulă consistentă: determină doar concluzii care derivă logic din premise

Regulă completă

Demonstrație

Strategii de control

Formă clauzală

Clauză Horn

Demonstrabile prin rezoluție

Nedemonstrabile prin rezoluție

Unificare

Rezumat

Satisfiabilitate: $P = \text{true}$ în cel puțin o interpretare

Validitate: $P = \text{true}$ în toate interpretările

Derivabilitate: $P = \text{true}$ în toate interpretările care satisfac premisele ($\Delta \models P$)

Inferență: derivare prin calcul a concluziilor unei mulțimi de premise ($\Delta \vdash_{\text{id_regulă}} P$)

Reguli de inferență: Modus Ponens, Modus Tollens, Rezoluție, axiome

Regulă consistentă: determină doar concluzii care derivă logic din premise

Regulă completă: determină toate concluziile care derivă logic din premise

Demonstrație

Strategii de control

Formă clauzală

Clauză Horn

Demonstrabile prin rezoluție

Nedemonstrabile prin rezoluție

Unificare

Rezumat

Satisfiabilitate: $P = \text{true}$ în cel puțin o interpretare

Validitate: $P = \text{true}$ în toate interpretările

Derivabilitate: $P = \text{true}$ în toate interpretările care satisfac premisele ($\Delta \models P$)

Inferență: derivare prin calcul a concluziilor unei multimi de premise ($\Delta \vdash_{\text{id_regulă}} P$)

Reguli de inferență: Modus Ponens, Modus Tollens, Rezoluție, axiome

Regulă consistentă: determină doar concluzii care derivă logic din premise

Regulă completă: determină toate concluziile care derivă logic din premise

Demonstrație: secvență de propoziții adevărate (premise, axiome, concluzii ale regulilor)

Strategii de control

Formă clauzală

Clauză Horn

Demonstrabile prin rezoluție

Nedemonstrabile prin rezoluție

Unificare

Rezumat

Satisfiabilitate: $P = \text{true}$ în cel puțin o interpretare

Validitate: $P = \text{true}$ în toate interpretările

Derivabilitate: $P = \text{true}$ în toate interpretările care satisfac premisele ($\Delta \models P$)

Inferență: derivare prin calcul a concluziilor unei multimi de premise ($\Delta \vdash_{\text{id_regulă}} P$)

Reguli de inferență: Modus Ponens, Modus Tollens, Rezoluție, axiome

Regulă consistentă: determină doar concluzii care derivă logic din premise

Regulă completă: determină toate concluziile care derivă logic din premise

Demonstrație: secvență de propoziții adevărate (premise, axiome, concluzii ale regulilor)

Strategii de control: forward chaining, backward chaining

Formă clauzală

Clauză Horn

Demonstrabile prin rezoluție

Nedemonstrabile prin rezoluție

Unificare

Rezumat

Satisfiabilitate: $P = \text{true}$ în cel puțin o interpretare

Validitate: $P = \text{true}$ în toate interpretările

Derivabilitate: $P = \text{true}$ în toate interpretările care satisfac premisele ($\Delta \models P$)

Inferență: derivare prin calcul a concluziilor unei multimi de premise ($\Delta \vdash_{\text{id_regulă}} P$)

Reguli de inferență: Modus Ponens, Modus Tollens, Rezoluție, axiome

Regulă consistentă: determină doar concluzii care derivă logic din premise

Regulă completă: determină toate concluziile care derivă logic din premise

Demonstrație: secvență de propoziții adevărate (premise, axiome, concluzii ale regulilor)

Strategii de control: forward chaining, backward chaining

Formă clauzală: conjuncție de disjuncții de literali

Clauză Horn

Demonstrabile prin rezoluție

Nedemonstrabile prin rezoluție

Unificare

Rezumat

Satisfiabilitate: $P = \text{true}$ în cel puțin o interpretare

Validitate: $P = \text{true}$ în toate interpretările

Derivabilitate: $P = \text{true}$ în toate interpretările care satisfac premisele ($\Delta \models P$)

Inferență: derivare prin calcul a concluziilor unei multimi de premise ($\Delta \vdash_{\text{id_regulă}} P$)

Reguli de inferență: Modus Ponens, Modus Tollens, Rezoluție, axiome

Regulă consistentă: determină doar concluzii care derivă logic din premise

Regulă completă: determină toate concluziile care derivă logic din premise

Demonstrație: secvență de propoziții adevărate (premise, axiome, concluzii ale regulilor)

Strategii de control: forward chaining, backward chaining

Formă clauzală: conjuncție de disjuncții de literali

Cluză Horn: cluză cu un singur literal nenegat

Demonstrabile prin rezoluție

Nedemonstrabile prin rezoluție

Unificare

Rezumat

Satisfiabilitate: $P = \text{true}$ în cel puțin o interpretare

Validitate: $P = \text{true}$ în toate interpretările

Derivabilitate: $P = \text{true}$ în toate interpretările care satisfac premisele ($\Delta \models P$)

Inferență: derivare prin calcul a concluziilor unei multimi de premise ($\Delta \vdash_{\text{id_regulă}} P$)

Reguli de inferență: Modus Ponens, Modus Tollens, Rezoluție, axiome

Regulă consistentă: determină doar concluzii care derivă logic din premise

Regulă completă: determină toate concluziile care derivă logic din premise

Demonstrație: secvență de propoziții adevărate (premise, axiome, concluzii ale regulilor)

Strategii de control: forward chaining, backward chaining

Formă clauzală: conjuncție de disjuncții de literali

Clauză Horn: clauză cu un singur literal nenegat

Demonstrabile prin rezoluție: validitate, derivabilitate, nesatisfiabilitate

Nedemonstrabile prin rezoluție

Unificare

Rezumat

Satisfiabilitate: $P = \text{true}$ în cel puțin o interpretare

Validitate: $P = \text{true}$ în toate interpretările

Derivabilitate: $P = \text{true}$ în toate interpretările care satisfac premisele ($\Delta \models P$)

Inferență: derivare prin calcul a concluziilor unei multimi de premise ($\Delta \vdash_{\text{id_regulă}} P$)

Reguli de inferență: Modus Ponens, Modus Tollens, Rezoluție, axiome

Regulă consistentă: determină doar concluzii care derivă logic din premise

Regulă completă: determină toate concluziile care derivă logic din premise

Demonstrație: secvență de propoziții adevărate (premise, axiome, concluzii ale regulilor)

Strategii de control: forward chaining, backward chaining

Formă clauzală: conjuncție de disjuncții de literali

Clauză Horn: clauză cu un singur literal nenegat

Demonstrabile prin rezoluție: validitate, derivabilitate, nesatisfiabilitate

Nedemonstrabile prin rezoluție: invaliditate, satisfiabilitate

Unificare

Rezumat

Satisfiabilitate: $P = \text{true}$ în cel puțin o interpretare

Validitate: $P = \text{true}$ în toate interpretările

Derivabilitate: $P = \text{true}$ în toate interpretările care satisfac premisele ($\Delta \models P$)

Inferență: derivare prin calcul a concluziilor unei multimi de premise ($\Delta \vdash_{\text{id_regulă}} P$)

Reguli de inferență: Modus Ponens, Modus Tollens, Rezoluție, axiome

Regulă consistentă: determină doar concluzii care derivă logic din premise

Regulă completă: determină toate concluziile care derivă logic din premise

Demonstrație: secvență de propoziții adevărate (premise, axiome, concluzii ale regulilor)

Strategii de control: forward chaining, backward chaining

Formă clauzală: conjuncție de disjuncții de literali

Clauză Horn: clauză cu un singur literal nenegat

Demonstrabile prin rezoluție: validitate, derivabilitate, nesatisfiabilitate

Nedemonstrabile prin rezoluție: invaliditate, satisfiabilitate

Unificare: cea mai generală substituție conform căreia 2 expresii devin identice

PARADIGME DE PROGRAMARE

Curs 10

Limbajul Prolog.

Programare logică în Prolog



Limbajul Prolog – Cuprins

- Descriere generală
- Sintaxă
- Semantică
- Inferență
- Unificare și instanțiere
- Mecanisme de control

Descriere generală (I)

Prolog = logică cu predicate de ordinul întâi + restricții

- **Propoziții = clauze Horn** – particularizate în

- **Fapte** de forma $true \Rightarrow \text{propoziție}$ (se va scrie doar **propoziție.**)

```
om(gica). om(ilie). impiedicat(gica).
```

```
traverseaza(ilie, santier). %% un şantier anume (identificat prin constanta santier)
```

```
sapa_groapa(ilie, gica). %% o groapă oarecare (nu trebuie identificată)
```

- **Reguli** de forma $\text{propoziție}_1 \wedge \dots \wedge \text{propoziție}_n \Rightarrow \text{propoziție}$
(se folosește scrierea **propoziție :- propoziție₁, ..., propoziție_n.**)

```
cade_in_groapa(X) :- impiedicat(X), traverseaza(X, santier).
```

```
cade_in_groapa(X) :- sapa_groapa(X, Y), X \= Y.
```

Descriere generală (II)

Prolog = logică cu predicate de ordinul întâi + restricții

- **Ipoteza lumii închise** = există adevăruri doar în program
(tot ce nu poate fi demonstrat în program este fals)

- Se rezolvă problema semidecidabilității

- Negația din Prolog \neq negația logică

- $\text{\textbackslash}+p$ în Prolog = programul curent nu poate demonstra p

bun(**X**) :- \+sapa_groapa(**X**,_). %% false, fiindcă sapa_groapa e satisfiabil

om_bun(**X**) :- om(**X**), \+sapa_groapa(**X**,_). %% X = gica

- $\neg p$ în LPOI = p este fals

Limbajul Prolog – Cuprins

- Descriere generală
- Sintaxă
- Semantică
- Inferență
- Unificare și instanțiere
- Mecanisme de control

Elemente de sintaxă

- **Constante:** 1, 2, ilie, gica (valoare sau identificator care începe cu literă mică)
- **Variabile:** X, List, Carte, _ (identificator care începe cu majusculă sau _)
- **Funcții:** +, -, mod, div, abs (putine! unitatea de bază e propoziția, nu funcția)
- **Structuri:** (modelează proprietățile și relațiile obiectelor)
carte(titlu('Magicianul'), autor('John Fowles'))
sapa_groapa(ilie, gica)
- **Coneective:** , ; (virgulă = \wedge , punct și virgulă = \vee)
(virgula are prioritate mai mare)

Sintaxă

termen = constantă | variabilă | structură
(obiect cu componente)

clauză = fapt | regulă

sapa_groapa(ilie, gica).

om_bun(**x**) :- om(**x**) , \+sapa_groapa(**x**, _).

fapt = structură.

regulă = antet :- corp.

antet = structură

corp = structură | structură, corp

Atenție: Faptele și regulile trebuie să se termine cu semnul „.” (punct)

Sintaxă liste

- [] - lista vidă
- [X | Rest] - lista cu head-ul X și tail-ul Rest
- [_ | Rest] - lista cu un head a cărui valoare e irelevantă și tail-ul Rest
- [X, Y | Rest] - lista formată din 2 elemente X și Y urmate de lista Rest
- [a, B, c] - lista cu 3 elemente dintre care primul și ultimul sunt fixate la constantele a și c

Limbajul Prolog – Cuprins

- Descriere generală
- Sintaxă
- Semantică
- Inferență
- Unificare și instanțiere
- Mecanisme de control

Semantică

Propoziții adevărate

- **Faptele** din program
- **Regulile** din program
- **Propozițiile derivabile din fapte și reguli** (via reducere la absurd folosind Rezoluție)

Scop = propoziție (cu sau fără variabile) care trebuie demonstrată

Interogare = solicitare de satisfacere a unui scop
(cu precizarea eventualelor variabile legate în acest proces)

Exemple

```
?- cade_in_groapa(ilie).    ?- cade_in_groapa(gica).    ?- cade_in_groapa(X).  
true.                      false.                     X = ilie.
```

Limbajul Prolog – Cuprins

- Descriere generală
- Sintaxă
- Semantică
- Inferență
- Unificare și instanțiere
- Mecanisme de control

Inferență

- Programul este doar o colecție de fapte și reguli
- Procesul de inferență este **încorporat în limbaj (ascuns de utilizator)** și constă în
 - **Backward chaining** – folosește doar propozițiile care pot duce la satisfacerea scopului
 - **DFS** – (sub)scopurile nou adăugate în stivă sunt primele pe care încearcă să le satisfacă
 - **Backtracking** – scopurile vizitate de DFS (prin găsirea unui mod de a le satisface și adăugarea subscopurilor rezultate în stivă) sunt revizitate (se explorează toate modurile diferite de a satisface un anumit scop, nu doar primul)
 - **Unificare** – o cale de satisfacere a unui scop este marcată printr-o serie de unificări ale (sub)scopurilor cu faptele și concluziile regulilor

Exemplu (la calculator)

```
?- titlu(X).  
X = 'Razboi si pace' ; %% o primă satisfacere a scopului titlu(X)  
X = 'Sonata Kreutzer' ; %% ; cere o resatisfacere (se obține gratuit bkt)  
X = 'Magicianul'. %% . de la utilizator cere oprirea aici  
%% . de la Prolog înseamnă că s-au terminat soluțiile
```

Inferență – Idei de implementare

- Algoritmul de inferență pe bază de unificare folosește
 - O **stivă Scopuri** – pornește cu scopul (scopurile) din interogare
 - O **substituție Legări** (multime de legări pentru variabilele din scopuri și subscopuri) – inițial vidă
- La fiecare iterație a algoritmului
 - Se extrage un scop din stivă
 - Dacă acesta unifică cu concluzia vreunei reguli (sau vreun fapt) prin substituția S
 - se adaugă S la Legări
 - se adaugă premisele regulii la Scopuri
 - se continuă cu noi scopuri din stivă până se golește stiva (succes) sau unificarea eșuează
 - Backtracking pentru a încerca alte variante

Inferență – Algoritm

backward_chaining(Clauze, Scopuri, Legări)

if Scopuri == []

 success //adaugă la soluții

 return

scop = head(Scopuri); Scopuri = tail(Scopuri)

for_each clauză in Clauze //bkt

 if unifică(scop, antet(clauză), Legări, S) //S = substituția întoarsă de unificare

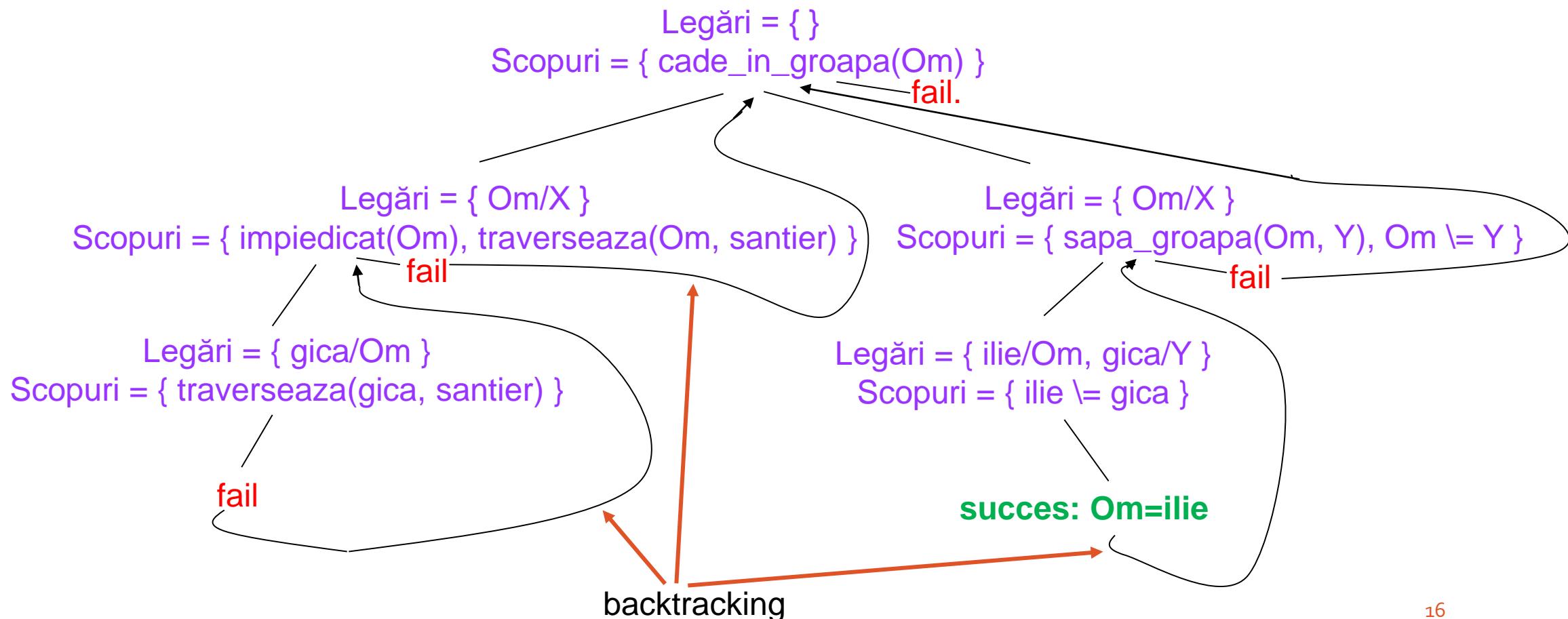
 Subst = Legări U S

 Stivă = append(corp(clauză), Scopuri) // DFS (noile scopuri la începutul stivei)

backward_chaining(Clauze, Stivă, Subst)

Exemplu

```
impiedicat(gica).  
traverseaza(ilie, santier).  
sapa_groapa(ilie, gica).  
cade_in_groapa(X) :- impiedicat(X), traverseaza(X, santier).  
cade_in_groapa(X) :- sapagroapa(X, Y), X \= Y.
```



Declarativ versus procedural

```
predecessor(Parent, Child) :- parent(Parent, Child).
```

```
predecessor(Pred, Succ) :- parent(Pred, Child), predecessor(Child, Succ).
```

Semnificația declarativă

- Interesează obiectele și relațiile definite în program
Pred este predecesorul lui Succ dacă Pred este părintele lui Child și Child este predecesorul lui Succ.
- Determină care va fi rezultatul
- Nu contează ordinea clauzelor și a premiselor în reguli

Semnificația procedurală

- Interesează pașii care sunt urmați de Prolog în evaluarea obiectelor și relațiilor
Pentru a arăta că Pred este predecesorul lui Succ, arată întâi că Pred e părintele lui Child, apoi că Child e predecesorul lui Succ.
- Determină cum se obține rezultatul
- Contează ordinea clauzelor și a premiselor în reguli

Ordinea contează

```
pred(P, C) :- parent(P, C).
```

```
pred(P, S) :- parent(P, C), pred(C, S).
```

%% clauze invers

```
pred1(P, S) :- parent(P, C), pred1(C, S).
```

```
pred1(P, C) :- parent(P, C).
```

%% premise invers

```
pred2(P, C) :- parent(P, C).
```

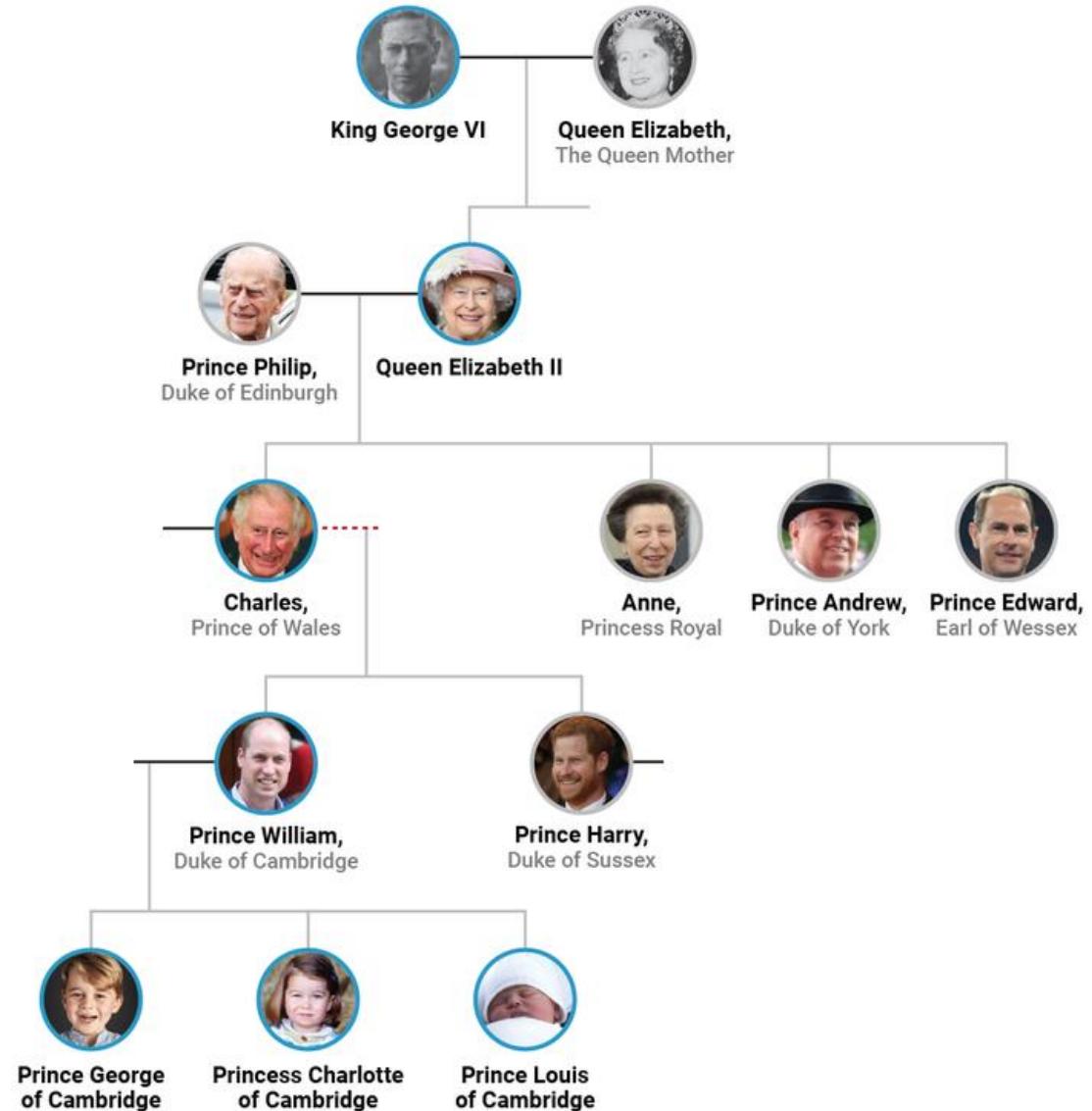
```
pred2(P, S) :- pred2(C, S), parent(P, C).
```

%% clauze și premise invers

```
pred3(P, S) :- pred3(C, S), parent(P, C).
```

```
pred3(P, C) :- parent(P, C).
```

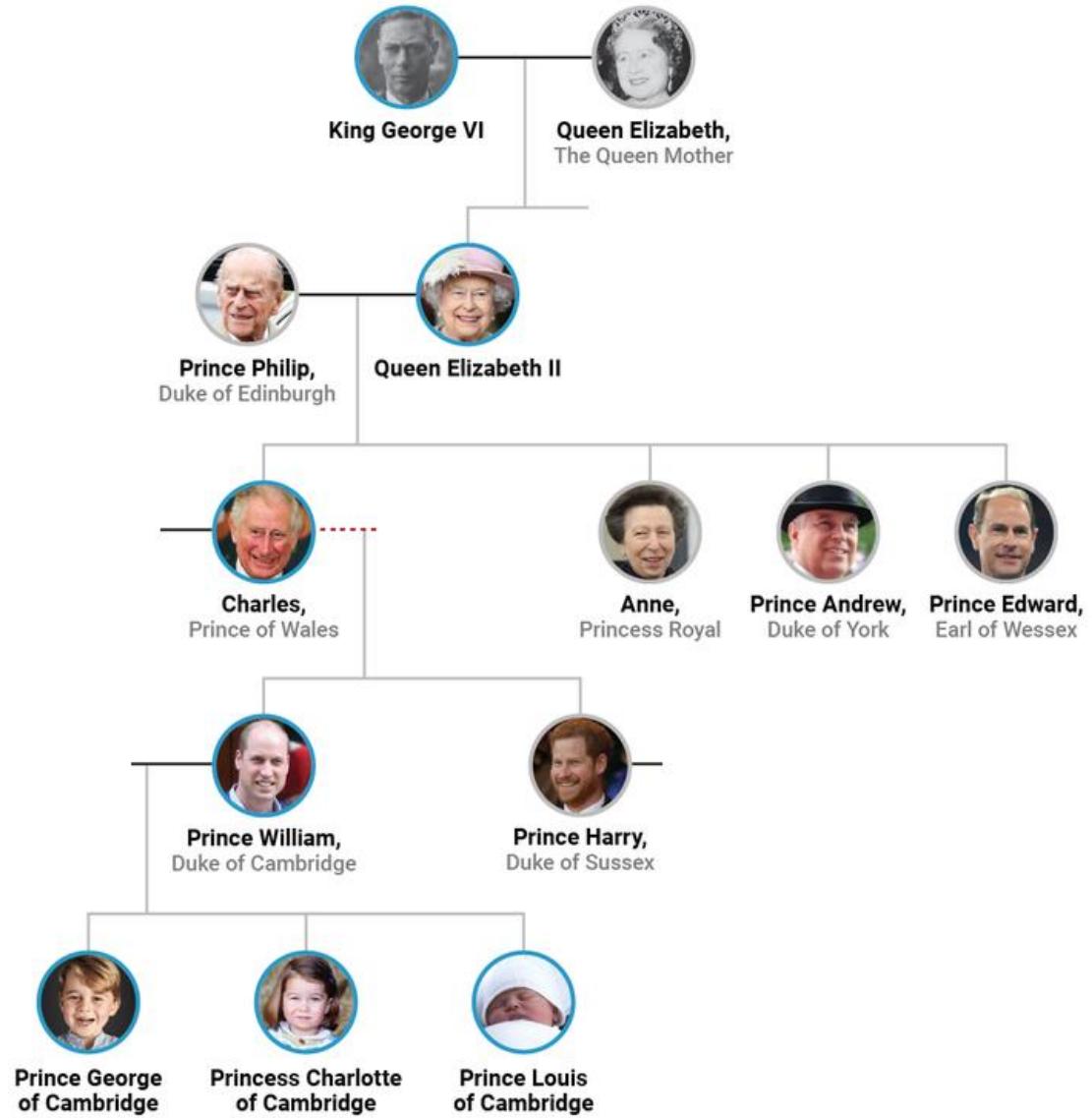
```
?- pred(X, william).
```



Exemplu

```
pred2 (P, C) :- parent (P, C) .  
pred2 (P, S) :- pred2 (C, S) , parent (P, C) .  
?- pred2 (X, william) .
```

- parent(charles, william) -> X = charles
- pred2(C,william), parent(X,C)
 - C=charles -> X = philip, X = elizabeth2
 - pred2(C',william), parent(C,C')
 - C'=charles->C...-> X = george6, X = elizabeth
 - pred2(C'',william), parent(C',C'')
 - C''=charles->fail
 - pred2(C''',william), parent(C'',C''')
 - C'''=charles->fail
 - pred2(C''',william), parent(C''',C''')
etc



Ordinea contează – Concluzii

- Contează ordinea clauzelor în program (se încearcă unificarea în ordine)
- Contează ordinea premiselor în corpul regulilor (se încearcă satisfacerea lor în ordine)
 - **Eficiență:** premisele a căror satisfacere **reduce mult spațiul de căutare** se pun primele
 - **Funcționalitate:** premisele care **instantiază variabilele** se pun primele (v. parent și v. mai jos)

```
1. factorial(0, 1).  
2. factorial(N, F) :-  
   N > 0,  
   N1 is N-1, factorial(N1, F1),  
   F is N * F1.  
6.  
7. factorial_1(0, 1).  
8. factorial_1(N, F) :-  
   N > 0,  
   factorial_1(N1, F1), N1 is N-1,  
   F is N*F1.
```

Pentru interogarea factorial(5,X),
programul știe să încerce apoi să
satisfacă scopul factorial(4,X), etc.

Pentru interogarea factorial(5,X),
programul încearcă apoi satisfacerea
lui factorial(N1,X), și dă o eroare tip
Arguments are not sufficiently instantiated

Limbajul Prolog – Cuprins

- Descriere generală
- Sintaxă
- Semantică
- Inferență
- Unificare și instanțiere
- Mecanisme de control

Exemplu – elem

```
1. elem(X, [X | _]).
```

%% pt că member există

```
2. elem(X, [_ | Rest]) :- elem(X, Rest).
```

```
?- elem(1, [1,2,3]).
```

```
?- elem(1, [1,2,1,3]).
```

```
?- elem(1, [1,2,X,Y] ).
```

```
?- elem(X, [1,2,X,Y] ).
```

```
?- elem(X, L).
```

Exemplu – elem

```
1. elem(X, [X | _]).
```

%% pt că member există

```
2. elem(X, [_ | Rest]) :- elem(X, Rest).
```

```
?- elem(1, [1,2,3]).
```

(_=1)

true

(_=1, _=[2,3])

elem(1, [2,3])

(_=1, _=[3])

elem(1, [3])

(_=1, _=[])

elem(1, [])

false

```
?- elem(1, [1,2,3]).
```

true ;

false.

Exemplu – elem

```
1. elem(X, [X | _]).
```

%% pt că member există

```
2. elem(X, [_ | Rest]) :- elem(X, Rest).
```

```
?- elem(1, [1,2,1,3]).
```

```
(_=1)           (_=1, _=[2,1,3])  
true             elem(1,[2,1,3])
```

```
                  (_=1, _=[1,3])
```

```
                  elem(1, [1,3])
```

```
true
```

```
                  (_=1, _=[3])
```

```
                  elem(1, [3])
```

.....

false

```
?- elem(1, [1,2,1,3]).
```

true ;

true ;

false.

Exemplu – elem

```
1. elem(X, [X | _]).
```

%% pt că member există

```
2. elem(X, [_ | Rest]) :- elem(X, Rest).
```

```
?- elem(1, [1,2,X,Y]).
```

```
(_=1)           (_=1, _=[2,X,Y])  
true
```

```
elem(1, [2,X,Y])
```

```
(_=1, _=[X,Y])  
elem(1, [X,Y])
```

```
X=1           (_=1, _=[Y])  
elem(1, [Y])
```

```
Y=1           (_=1, _=[ ])  
elem(1, [])
```

false

```
?- elem(1, [1,2,X,Y]).
```

true ;

X = 1 ;

Y = 1 ;

false.

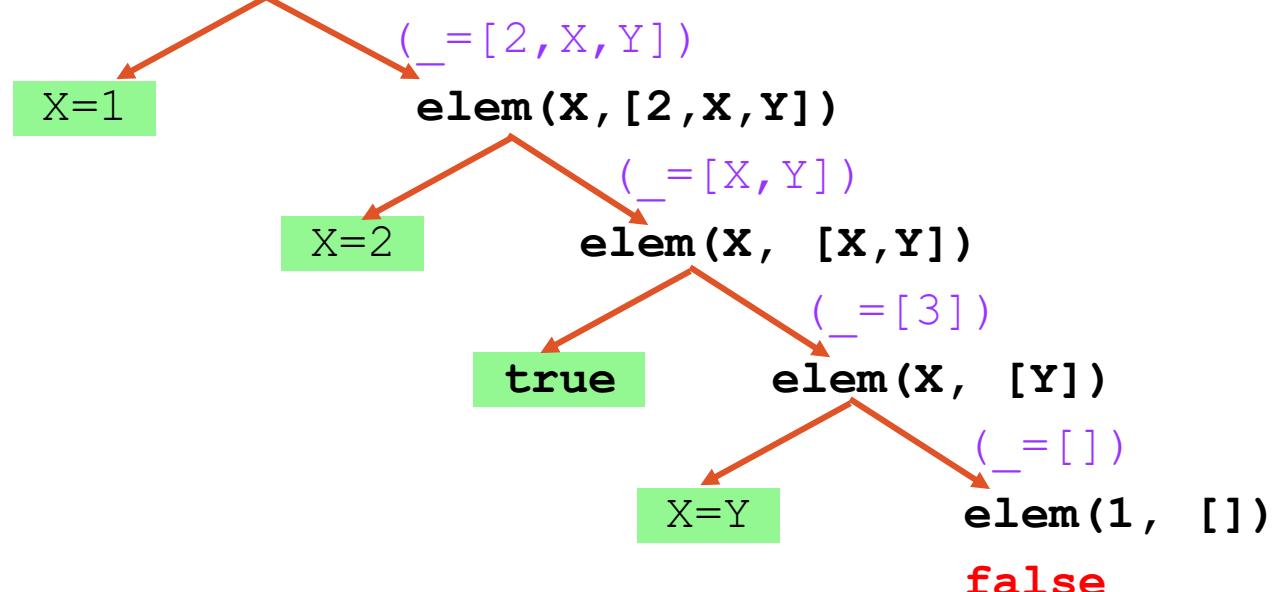
Exemplu – elem

1. elem(**X**, [**X** | _]) .

%% pt că member există

2. elem(**X**, [_ | **Rest**]) :- elem(**X**, **Rest**) .

?- elem(**X**, [1, 2, **X**, **Y**]) .



?- elem(**X**, [1, 2, **X**, **Y**]) .

X = 1 ;

X = 2 ;

true ;

X = Y ;

false.

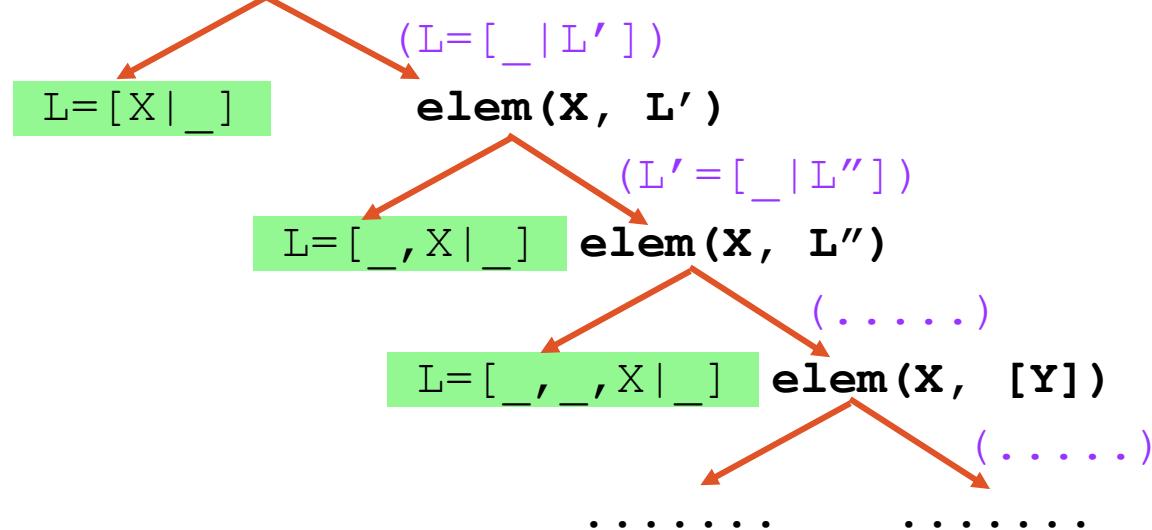
Exemplu – elem

1. elem(**X**, [**X|_**]) .

%% pt că member există

2. elem(**X**, [_|Rest]) :- elem(**X**, Rest) .

?- elem(X, L) .



?- elem(X, L) .

L = [X|_4056] ;

L = [_4054, X|_4062] ;

L = [_4054, _4060, X|_4068] .

Observații – elem

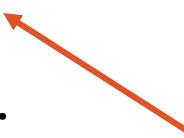
- Structurile Prolog **elimină separarea între datele de intrare și rezultate** (ieșire)
 - Atât intrările cât și ieșirile sunt termeni în structură (statutul de intrare/ieșire nu depinde nicicum de poziția în structură)
 - Spre deosebire de funcții, care separă clar intrarea (argumente) de ieșire (rezultat)
- Mulțumită mecanismului de unificare, elem poate fi folosit pentru
 - a **verifica** apartenența unui element dat la o listă dată: `elem(1, [1,2,3]).`
 - a **afla condițiile (instantierile de variabile)** în care un element este membru într-o listă: `elem(1, [1,2,X,Y]).`
 - a **genera** elementele unei liste date: `elem(X, [1,2,3,4]).`
 - a genera liste care conțin un anumit element: `elem(1, L).`
- Nu contează doar satisfacerea scopului, ci satisfacerea scopului în toate modurile posibile: `elem(1, [1,2,1,3]).`

Exercițiu – concat

```
1. concat([], L, L).  
2. concat([X|Rest], L, [X|Rez]) :- concat(Rest, L, Rez).
```

```
?- concat([1,2],[3,4],[1,2,3,4]).  
?- concat([1,2],[3,4],[1,2,X,4]).  
?- concat([1,2],[3,4],[1,2,3]).  
?- concat(X,[3,4],[1,2,3,4]).  
?- concat([1,2],Y,[1,2,3,4]).  
?- concat([1,2],[3,4],Z).  
?- concat(X,Y,[1,2,3,4]).  
?- concat(X,[3,4],Z).  
?- concat([1,2],Y,Z).  
?- concat(X,Y,Z).
```

X	Rest	L
X	Rez	



Regula se citește: dacă L_1 este o listă cu head-ul X și tail-ul $Rest$, și dacă $Rest$ concatenat cu L dă Rez , atunci L_1 concatenat cu L dă o listă cu head-ul X și tail-ul Rez .

Atenție: În Prolog, spre deosebire de Haskell, pattern-urile se potrivesc și între ele. Prolog știe că în $[X|Rest]$ și $[X|Rez]$ folosesc aceeași valoare X .

Exercițiu – concat

1. `concat([], L, L).`

X	Rest	L
---	------	---

2. `concat([X|Rest], L, [X|Rez]) :- concat(Rest, L, Rez).`

X		Rez
---	--	-----

?- `concat([1,2],[3,4],[1,2,3,4]).`

true.

?- `concat([1,2],[3,4],[1,2,X,4]).`

X = 3.

?- `concat([1,2],[3,4],[1,2,3]).`

false.

?- `concat(X,[3,4],[1,2,3,4]).`

X = [1,2]; **false.**

?- `concat([1,2],Y,[1,2,3,4]).`

Y = [3,4].

?- `concat([1,2],[3,4],Z).`

Z = [1,2,3,4].

?- `concat(X,Y,[1,2,3,4]).` X = [], Y = [1,2,3,4]; X = [1], Y = [2,3,4]; ...

?- `concat(X,[3,4],Z).` X = [], Z = [3,4]; X = [_582], Z = [_582,3,4]; ...

?- `concat([1,2],Y,Z).`

Z = [1,2|Y].

?- `concat(X,Y,Z).`

X = [], Y = Z; X = [_588], Z = [_588|Y]; ...

Exerciții

Să se implementeze elem folosind doar concat.

Să se implementeze last folosind doar concat.

Să se șteargă primele și ultimele 2 elemente dintr-o listă folosind doar concat.

Exerciții

Să se implementeze elem folosind doar concat.

```
elem_(X, L) :- concat(_, [X|_], L).
```

Semnificație declarativă sporită
Semnificație procedurală diminuată

Să se implementeze last folosind doar concat.

```
last_(L, X) :- concat(_, [X], L).
```

Să se șteargă primele și ultimele 2 elemente dintr-o listă folosind doar concat.

```
del22(L, Del) :- concat([_, _|Del], [_, _], L).
```

Unificare, atribuire, evaluare

- **Unificarea** se poate realiza și explicit folosind operatorul **=**
 - Nu se produce niciun fel de evaluare, unificarea reușește doar dacă există o instanțiere a variabilelor în urma căreia cei 2 termeni devin identici

?- $1+2 = 1+2.$

?- $1+2 = 1+x.$

?- $1+2 = x+1.$

?- $x = 1+2.$

- Operatorul **\=** înseamnă „**nu unifică**”: $1+2 \neq 3.$
- Pentru **atribuire cu evaluarea** operațiilor aritmetice se folosește **is**: $x \leftarrow 1+2.$
- Pentru **verificarea egalității / inegalității valorilor** se folosesc **=:=** și **=\=:** $1+2 =\neq 3.$
- Pentru **verificarea egalității / inegalității structurale** se folosesc **==** și **\==:**

$1+x == 1+2.$

$1+2 == 1+2.$

Unificare, atribuire, evaluare

- **Unificarea** se poate realiza și explicit folosind operatorul **=**

- Nu se produce niciun fel de evaluare, unificarea reușește doar dacă există o instanțiere a variabilelor în urma căreia cei 2 termeni devin identici

?- $1+2 = 1+2$. **true**.

?- $1+2 = 1+x$. $x = 2$.

?- $1+2 = x+1$. **false**.

?- $x = 1+2$. $x = 1+2$.

Unificarea este în general
însoțită de instanțieri

- Operatorul **\=** înseamnă „**nu unifică**”: $1+2 \neq 3$. (**true**)
- Pentru **atribuire cu evaluarea** operațiilor aritmetice se folosește **is**: $x \text{ is } 1+2$. ($x = 3$)
- Pentru **verificarea egalității / inegalității valorilor** se folosesc **=:=** și **=\=**: $1+2 =\neq 3$. (**false**)
- Pentru **verificarea egalității / inegalității structurale** se folosesc **==** și **\==**:

$1+x == 1+2$. (**false**)

$1+2 == 1+2$. (**true**)

Limbajul Prolog – Cuprins

- Descriere generală
- Sintaxă
- Semantică
- Inferență
- Unificare și instanțiere
- Mecanisme de control

Oprirea backtracking-ului

Fie analogul lui compare din Haskell:

1. `comp(X, Y, 'LT') :- X < Y.`
2. `comp(X, Y, 'EQ') :- X =:= Y.`
3. `comp(X, Y, 'GT') :- X > Y.`

`?- comp(2+3, 3+1, Ord).`

`?- comp(2+3, 3+1+1, Ord).`

`?- comp(2, 3+1, Ord).`

Oprirea backtracking-ului

Fie analogul lui compare din Haskell:

1. `comp(X, Y, 'LT') :- X < Y.`
2. `comp(X, Y, 'EQ') :- X =:= Y.`
3. `comp(X, Y, 'GT') :- X > Y.`

Cele 3 reguli sunt **mutual exclusive**. Cum putem să îi spunem Prolog-ului ca, dacă s-a putut aplica o regulă, să nu le mai încerce pe celelalte?

?- `comp(2+3, 3+1, Ord).`

Ord = 'GT'.

?- `comp(2+3, 3+1+1, Ord).`

Ord = 'EQ' ; **false**.

?- `comp(2, 3+1, Ord).`

Ord = 'LT' ; **false**.

Predicatul ! (cut)

Rol cut: opirea backtracking-ului la prima satisfacere a unui anumit scop

- În sensul că se vor încerca în continuare toate soluțiile care se pot obține din acest punct în dreapta, dar nu vom încerca să resatisfacem vreun scop din trecut

Funcționare cut

- Prima dată, cut reușește (este satisfiabil)
- Când se revine prin backtracking la cut, cut eșuează
- Toate regulile următoare cu același fel de antet (același predicat) sunt ignorate

1. comp (**X**, **Y**, '**LT**') :- **X** < **Y**, !.
2. comp (**X**, **Y**, '**EQ**') :- **X** =:= **Y**, !.
3. comp (**_**, **_**, '**GT**') .

Nu încercați să scrieți fără cut, ca în Haskell.
Se va face backtracking și veți obține că toate X și Y sunt în relația 'GT'!

Aplicabilitate cut

- **Eficientizarea programelor** (nu doar cazul regulilor mutual exclusive)

Exemplu

Să presupunem că vrem doar funcționalitatea de predicat a lui elem (testarea apartenenței)

```
1. elemP(X, [X | _]) :- !.  
2. elemP(X, [_ | Rest]) :- elemP(X, Rest).
```

```
?- elemP(1, [1,2,3]).  
?- elemP(1, [1,2,1,3]).  
?- elemP(1, [1,2,X,Y]).  
?- elemP(X, [1,2,X,Y]).  
?- elemP(X, L).
```

Aplicabilitate cut

- **Eficientizarea programelor** (nu doar cazul regulilor mutual exclusive)

Exemplu

Să presupunem că vrem doar funcționalitatea de predicat a lui elem (testarea apartenenței)

```
1. elemP(X, [X | _]) :- !.  
2. elemP(X, [_ | Rest]) :- elemP(X, Rest).  
  
?- elemP(1, [1,2,3]).          true.  
?- elemP(1, [1,2,1,3]).        true.  
?- elemP(1, [1,2,X,Y]).       true.  
?- elemP(X, [1,2,X,Y]).      X = 1.  
?- elemP(X, L).               L = [X | _ 6714].
```

Se pierde abilitatea generativă a lui elem

Semnificația procedurală devine mai importantă decât cea declarativă (ea dictează acum care va fi rezultatul)

Backtracking doar la dreapta lui cut

```
sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.
```

```
sibling_(X, Y) :- parent(P, X), !, parent(P, Y), X \= Y.
```

```
?- sibling(X, anne).
```

```
?- sibling(anne, X).
```

```
?- sibling_(X, anne).
```

```
?- sibling_(anne, X).
```

Backtracking doar la dreapta lui cut

```
sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.  
sibling_(X, Y) :- parent(P, X), !, parent(P, Y), X \= Y.
```

```
?- sibling(X, anne).
```

```
X = charles ;
```

```
X = andrew ;
```

```
X = edward ;
```

```
X = charles ;
```

```
X = andrew ;
```

```
X = edward ;
```

false.

```
?- sibling_(X, anne).
```

false.

parent(george6, elizabeth) leagă P la
george6 fără posibilitate de revenire, după
care parent(george6, anne) eșuează

Resatisfacerea
lui parent(P, X)
produce duplicate

```
?- sibling_(anne, X).
```

```
X = charles ;
```

```
X = andrew ;
```

```
X = edward ;
```

```
X = charles ;
```

```
X = andrew ;
```

```
X = edward.
```

```
?- sibling_(anne, X).
```

```
X = charles ;
```

```
X = andrew ;
```

```
X = edward.
```

Mecanisme de control

- **true** – predicat care reușește întotdeauna
- **fail** – predicat care eșuează întotdeauna
- **cut** – predicat care oprește backtracking-ul pe structurile anterioare și regulile ulterioare
- **once(P)** – permite lui P să reușească o singură dată (**once(P) :- P, !.**)
- **not(P)** – reușește dacă P nu e satisfiabil (**not(P) :- P, !, fail ; true.**)

(se preferă scrierea **\+** în loc de **not**, pentru a sugera că nu este vorba de negație logică)

true după sau **(;)** nu este atins, este ca și cum s-ar afla într-o regulă ulterioară

Negația ca eșec

Revenind pe şantier...

```
bun(X) :- \+sapa_groapa(X, _).
```

```
?- bun(X).  
false.
```

- Interogarea `sapa_groapa(X, _)` se poate citi și „Există X care sapă groapa cuiva?”
- Interogarea `\+sapa_groapa(X, _)` **nu** se poate citi și „Există X care nu sapă groapa cuiva?”
 - Ea se citește ca un eșec al variantei afirmative:
„not(Există X care sapă groapa cuiva)” \Leftrightarrow „Oricare X, X nu sapă groapa nimănu”
 - Funcționare: `sapa_groapa(X, _)` unifică cu `sapa_groapa(ilie, gica)`
combinația `!`, fail condamnă satisfacerea scopului curent la eșec definitiv

Negăția ca eșec – Exercițiu

1. `om_bun(X) :- om(X), \+sapa_groapa(X,_) .`
2. `om_bun_(X) :- \+sapa_groapa(X,_), om(X) .`

`?- om_bun(X) .`

`?- om_bun_(X) .`

Negația ca eșec – Exercițiu

1. `om_bun(X) :- om(X), \+sapa_groapa(X,_) .`
2. `om_bun_(X) :- \+sapa_groapa(X,_), om(X) .`

```
?- om_bun(X) .  
X = gica ;  
false.
```

```
?- om_bun_(X) .  
false.
```

Observații:

- Se recomandă evitarea predicatului cut atunci când acesta distruge corespondența între semnificația declarativă și cea procedurală
- Semnificație declarativă = semnificație procedurală \Leftrightarrow
Programe ușor de înțeles, care fac ceea ce ne așteptăm să facă
- Prin urmare, atât cut cât și not trebuie folosite cu grija și numai cu un motiv bun

Rezumat

Propoziții Prolog

Ipoteza lumii închise

Regula de inferență

Strategia de căutare

Algoritmi folosiți în procesul de inferență

Semnificații ale programului

Reguli reversibile

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise

Regula de inferență

Strategia de căutare

Algoritmi folosiți în procesul de inferență

Semnificații ale programului

Reguli reversibile

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență

Strategia de căutare

Algoritmi folosiți în procesul de inferență

Semnificații ale programului

Reguli reversibile

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență: Rezoluția

Strategia de căutare

Algoritmi folosiți în procesul de inferență

Semnificații ale programului

Reguli reversibile

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență: Rezoluția

Strategia de căutare: backward chaining

Algoritmi folosiți în procesul de inferență

Semnificații ale programului

Reguli reversibile

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență: Rezoluția

Strategia de căutare: backward chaining

Algoritmi folosiți în procesul de inferență: DFS, backtracking, unificare

Semnificații ale programului

Reguli reversibile

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență: Rezoluția

Strategia de căutare: backward chaining

Algoritmi folosiți în procesul de inferență: DFS, backtracking, unificare

Semnificații ale programului: declarativă (ce este soluția), procedurală (cum se obține soluția)

Reguli reversibile

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență: Rezoluția

Strategia de căutare: backward chaining

Algoritmi folosiți în procesul de inferență: DFS, backtracking, unificare

Semnificații ale programului: declarativă (ce este soluția), procedurală (cum se obține soluția)

Reguli reversibile: forma relațională permite funcționarea în diverse sensuri (alternând ce este intrare și ce este rezultat)

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență: Rezoluția

Strategia de căutare: backward chaining

Algoritmi folosiți în procesul de inferență: DFS, backtracking, unificare

Semnificații ale programului: declarativă (ce este soluția), procedurală (cum se obține soluția)

Reguli reversibile: forma relațională permite funcționarea în diverse sensuri (alternând ce este intrare și ce este rezultat)

Operatori de unificare/atribuire/verificare egalitate: = / is / :=, ==

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență: Rezoluția

Strategia de căutare: backward chaining

Algoritmi folosiți în procesul de inferență: DFS, backtracking, unificare

Semnificații ale programului: declarativă (ce este soluția), procedurală (cum se obține soluția)

Reguli reversibile: forma relațională permite funcționarea în diverse sensuri (alternând ce este intrare și ce este rezultat)

Operatori de unificare/atribuire/verificare egalitate: = / is / :=, ==

Mecanisme de control: true, fail, cut, once, not

PARADIGME DE PROGRAMARE

Curs 11

Metapredicte. Probleme de căutare în spațiul stărilor. Probleme de satisfacere a constrângerilor.

Metapredicate

Metapredicat = predicat care primește scopuri ca argumente

- corespondentul funcționalelor din programarea funcțională

Metapredicate pentru colectarea soluțiilor (de satisfacere a unui scop)

- findall
- bagof
- setof

Metapredicale de tip for

- forall

Metapredicatul **findall**

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

- Pentru fiecare variantă de satisfacere a scopului Goal, instanțierea corespunzătoare a lui Template este depusă în Bag

Exemplu

scop compus

```
?- findall(X-Y, (numlist(3,12,NL),member(X,NL),member(Y,NL),X>Y,0 =:= mod(X,Y)), L).
```

Metapredicatul **findall**

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

- Pentru fiecare variantă de satisfacere a scopului Goal, instanțierea corespunzătoare a lui Template este depusă în Bag

Exemplu

scop compus

```
?- findall(X-Y, (numlist(3,12,NL),member(X,NL),member(Y,NL),X>Y,0 =:= mod(X,Y)), L).  
L = [6-3, 8-4, 9-3, 10-5, 12-3, 12-4, 12-6].
```

Metapredicatul **bagof**

bagof (+Template, :Goal, -Bag)

- La fel ca findall, dar se construiește câte un Bag pentru fiecare instanțiere diferită a variabilelor libere din Goal (libere = care nu apar în Template)

Exemple

```
?- bagof(X-Y, (numlist(3,12,NL), member(X,NL), member(Y,NL), X>Y, 0 =:= mod(X,Y))), L).
```

```
?- bagof(X, (numlist(3,12,NL), member(X,NL), member(Y,NL), X>Y, 0 =:= mod(X,Y))), L).
```

Metapredicatul **bagof**

bagof (+Template, :Goal, -Bag)

- La fel ca findall, dar se construiește câte un Bag pentru fiecare instanțiere diferită a variabilelor libere din Goal (libere = care nu apar în Template)

Exemple

```
?- bagof(X-Y, (numlist(3,12,NL),member(X,NL),member(Y,NL),X>Y,0 =:= mod(X,Y)), L).  
NL = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12], L = [6-3, 8-4, 9-3, 10-5, 12-3, 12-4, 12-6].  
?- bagof(X, (numlist(3,12,NL),member(X,NL),member(Y,NL),X>Y,0 =:= mod(X,Y)), L).  
NL = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12], Y = 3, L = [6, 9, 12] ;  
NL = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12], Y = 4, L = [8, 12] ;  
NL = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12], Y = 5, L = [10] ;  
NL = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12], Y = 6, L = [12].
```

Metapredicatul **bagof**

bagof (+**Template**, :**Goal**, -**Bag**)

- La fel ca findall, dar se construiește câte un Bag pentru fiecare instanțiere diferită a variabilelor libere din Goal (libere = care nu apar în Template)
- Dacă doresc să nu se țină cont de instanțierile diferite ale unei variabile libere Y folosesc notația **Y^Goal**

Exemplu

```
?- bagof(X, Y^(numlist(3,12,NL), member(X,NL), member(Y,NL), X>Y, 0 =:= mod(X,Y))), L).
```

Metapredicatul **bagof**

bagof (+Template, :Goal, -Bag)

- La fel ca findall, dar se construiește câte un Bag pentru fiecare instanțiere diferită a variabilelor libere din Goal (libere = care nu apar în Template)
- Dacă doresc să nu se țină cont de instanțierile diferite ale unei variabile libere Y folosesc notația **Y^AGoal**

Exemplu

```
?- bagof(X, Y^(numlist(3,12,NL), member(X,NL), member(Y,NL), X>Y, 0 =:= mod(X,Y))), L).  
NL = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12], L = [6, 8, 9, 10, 12, 12, 12].
```

Metapredicatul **setof**

setof (+Template, :Goal, -Bag)

- La fel ca bagof, dar fiecare Bag este o mulțime (nu conține duplicate) sortată
- Dacă doresc să nu se țină cont de instantierile diferite ale unei variabile libere Y folosesc notația **Y^Goal** (ca la bagof)

Exemplu

```
?- setof(X, Y^(numlist(3,12,NL), member(X,NL), member(Y,NL), X>Y, 0 =:= mod(X,Y))), L.
```

Metapredicatul **setof**

setof (+Template, :Goal, -Bag)

- La fel ca bagof, dar fiecare Bag este o mulțime (nu conține duplicate) sortată
- Dacă doresc să nu se țină cont de instantierile diferite ale unei variabile libere Y folosesc notația **Y^Goal** (ca la bagof)

Exemplu

```
?- setof(X, Y^(numlist(3,12,NL), member(X,NL), member(Y,NL), X>Y, 0 =:= mod(X,Y))), L.  
NL = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12], L = [6, 8, 9, 10, 12].
```

Metapredicatul **forall**

`forall (:Cond, :Action)`

- Reușește dacă pentru fiecare satisfacere a lui Cond se poate demonstra Action

Exemple

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

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

Metapredicatul **forall**

`forall (:Cond, :Action)`

- Reușește dacă pentru fiecare satisfacere a lui Cond se poate demonstra Action

Exemple

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

Căutare în spațiul stărilor – Cuprins

- Spațiul stărilor
- Strategii de căutare
- Reprezentarea datelor
- Abstractizare – o bibliotecă pentru căutare

Spațiul stărilor

- Modelat ca **graf orientat**
 - **Noduri** = stări intermediare (soluții parțiale)
 - **Stare inițială** = soluție neexpandată (configurație de pornire)
 - **Stări finale** = soluții complete (configurații care respectă toate cerințele problemei)
 - **Arce** = tranziții legale între stări (de la o soluție parțială la o soluție parțială extinsă cu un pas)

Căutare în spațiul stărilor = explorarea spațiului, pornind de la o stare inițială, cu scopul de a găsi una sau mai multe stări finale

Spațiul stărilor – Exemplu

Fie problema săriturii calului pe o tablă de șah 5×5 .

- Soluția presupune să plecăm din poziția $1/1$ și să acoperim tabla prin sărituri de cal.

Exemple (de elemente în graf)

- **Stare inițială I:** $[1/1]$ (calul încă nu a efectuat nicio săritură)
- **Nod intermedian N₁:** $[1/1, 2/3]$ (din linia 1 și coloana 1 s-a sărit pe linia 2 și coloana 3)
- **Nod intermedian N₂:** $[1/1, 2/3, 1/5]$
- **Stare finală F:** $[1/1, 2/3, 1/5, 3/4, 1/3, 2/1, 4/2, 5/4, 3/5, 1/4, 2/2, 4/1, 3/3, 2/5, 4/4, 5/2, 3/1, 1/2, 2/4, 4/5, 5/3, 3/2, 5/1, 4/3, 5/5]$
- **Arc:** (N_1, N_2) (tranzitie legală, spre deosebire de (N_1, F) care nu e arc în graf)

Căutare în spațiul stărilor – Cuprins

- Spațiul stărilor
- Strategii de căutare
- Reprezentarea datelor
- Abstractizare – o bibliotecă pentru căutare

Strategii de căutare

- **Generare și testare**
 - foarte ineficient (presupune generarea tuturor permutărilor)
- **Backtracking**
 - foarte adekvat Prolog-ului (care are backtracking încorporat în motorul de inferență)
- **DFS / BFS / Iterative deepening / A***
 - ușor de modelat folosind
 - liste (pe post de stive și cozi)
 - metapredicale (pentru a colecta vecinii unui anumit nod în spațiul stărilor)

Observație: Ne vom concentra pe mecanismul de backtracking.

Căutare în spațiul stărilor – Cuprins

- Spațiul stărilor
- Strategii de căutare
- Reprezentarea datelor
- Abstractizare – o bibliotecă pentru căutare

Reprezentarea datelor

Avem de reprezentat

- Noduri
 - Care este (fapt) sau cum se generează (regulă) o **stare inițială**
 - Care este (fapt) sau cum se recunoaște (regulă) o **stare finală**
- Arce
 - Ce reprezintă o **tranzitie legală** dintr-o stare în alta (regulă)

Reprezentări tipice

- Noduri = căi între configurația inițială și cea curentă
= liste (ex: lista ordonată a pozițiilor pe care calul a sărit deja)
 - Stare inițială = listă vidă (sau conținând o situație inițială)
- Arce = scopuri arc(+Sursă, -Destinație) unde Destinația e o expandare a Sursei

Reprezentarea stării finale

Observatie

- Cu sau fără template, se vor expanda multe căi inutile dacă programatorul nu filtrează încă din prediciul arc sau imediat după ce efectuează o tranziție folosind prediciul arc

Căutare în spațiul stărilor – Cuprins

- Spațiul stărilor
- Strategii de căutare
- Reprezentarea datelor
- Abstractizare – o bibliotecă pentru căutare

Exemplu – săritura calului

Rezolvări (la calculator)

- Varianta cu template
- Varianta fără template

Discuție

- Ce putem abstractiza astfel încât să obținem o bibliotecă utilă și în rezolvarea altor probleme?

Abstractizarea procesului de căutare

```
%% rezolvare = explorare pornind de la starea inițială
solve(Sol) :- initial_state(State), search(State, Sol).
```

```
%% explorarea se oprește când atingem o stare finală
search(State, Sol) :- final_state(State), !, reverse(State, Sol).
```

```
%% altfel:
```

```
search([S | State], Sol) :-
    arc(S, Next),                                %% alege o configurație următoare
    \+member(Next, State),                      %% evitând ciclurile
    search([Next, S | State], Sol).          %% și continuă explorarea
```

Satisfacerea constrângerilor – Cuprins

- Descrierea problemei
- Strategii de rezolvare
- Puzzle-uri logice
- Abstractizare

Problemă de satisfacere a constrângerilor

Date de intrare

- O mulțime de variabile
- Domeniile din care variabilele pot lua valori
- Constrângeri asupra variabilelor

Ieșire

- O mulțime de asocieri variabilă-valoare care să satisfacă toate condițiile de mai sus

Exemplu

Intrare

- $X_1, X_2, X_3, X_4, Y_1, Y_2, Y_3, Y_4 \in [2..9]$
- $X_1 * 2 = Y_3, Y_4 * 2 = X_2, Y_1 = X_1 + X_3, X_1 + X_2 + X_3 + X_4 > Y_1 + Y_2 + Y_3 + Y_4$
- Toate variabilele au valori diferite între ele

Ieșire

$X_i: 2 \ 6 \ 7 \ 8$
 $Y_i: 9 \ 5 \ 4 \ 3$

Satisfacerea constrângerilor – Cuprins

- Descrierea problemei
- Strategii de rezolvare
- Puzzle-uri logice
- Abstractizare

Strategii de rezolvare

- Strategiile de căutare deja menționate
- Backtracking + Algoritm de arc-consistență

Algoritm de arc-consistență (într-o rețea de constrângeri)

- Rețea de constrângeri = graf orientat
- Nod = variabilă
- Arc = constrângere care implică cele 2 variabile
 - (X,Y) arc-consistent dacă pentru orice valoare din domeniul lui X există o valoare în domeniul lui Y astfel încât constrângerea este respectată
- Funcționare: se examinează toate arcele pentru arc-consistență, eliminând valori din domeniile variabilelor atunci când este necesar (ceea ce duce la un număr finit de reexaminări)

Satisfacerea constrângerilor – Cuprins

- Descrierea problemei
- Strategii de rezolvare
- **Puzzle-uri logice**
- Abstractizare

Puzzle-uri logice

Puzzle logic = problemă de deducție, adecvată rezolvării în paradigmă logică

- Strategia de rezolvare = **generare și testare + optimizări** (într-un sens larg, asta sunt toate strategiile menționate anterior)
- Posibilități
 - **Generare ușoară** (soluția se alege dintre puțini candidați, care pot fi enumerați și apoi testați)
 - Se pornește cu setul complet de candidați
 - Pentru fiecare constrângere, se scrie o regulă care produce un set actualizat **pe baza celui anterior**
 - **Generare dificilă** (foarte mulți candidați, durează foarte mult să fie generați toți)
 - Se pornește cu un template care descrie soluția
 - Se instanțiază acest template începând cu alegerile implicate în cât mai multe constrângeri, astfel încât să se reducă mult și rapid spațiul candidaților
 - Nu există o regulă clară despre ordinea care va da cele mai bune rezultate, și de la un punct încolo recurgem tot la generare de permutări (însă intuiția este să o facem cât mai „târziu”)

Generare ușoară – leul și unicornul

Leul și unicornul: Ce zi e azi?

Leul minte în fiecare luni, marți și miercuri și în rest spune adevărul.

Unicornul minte în fiecare joi, vineri și sâmbătă și în rest spune adevărul.

Leul spune: Ieri mințeam.

Unicornul spune: Și eu.

- Tipul problemei
 - **generare ușoară** (avem doar 7 candidați – cele 7 zile ale săptămânii)
- Rezolvare (la calculator)
 - Scrie câte o regulă de filtrare a setului de candidați pentru fiecare constrângere
 - Cea impusă de afirmația leului
 - Cea impusă de afirmația unicornului

Generare ușoară – ziua lui Cheryl

- Enunț și rezolvare – la calculator

Generare dificilă – devoratoarele de Pizza

În jurul unei pizza cu 6 felii stau:

- 6 prietene: mara, moira, maya, marla, myra, marva
- născute în orașele: austin, dallas, san_antonio, galveston, woodlands, houston
- și au ca topping: mushrooms, sausage, pepperoni, pepper, meatballs, broccoli

Va trebui să aflăm cum sunt așezate ele, unde e născută și ce mănâncă fiecare, ținând cont de o serie de indicii (constrângeri).

- Tipul problemei
 - **generare dificilă** (foarte multe combinații posibile)
- Rezolvare (la calculator, alături de restul enunțului)
 - Pornim de la un template care descrie soluția
 - Generăm și testăm toate variantele, apoi schimbăm ordinea pentru a constata efectul asupra performanței

Satisfacerea constrângerilor – Cuprins

- Descrierea problemei
- Strategii de rezolvare
- Puzzle-uri logice
- Abstractizare

Biblioteca clpfd

- **CLP(FD)**: Constraint Logic Programming over Finite Domains
 - :- use_module(library(clpfd)).
- Algoritm de arc-consistență încorporat – pentru constrângeri asupra numerelor întregi
 - Pentru cei interesați: <https://github.com/triska/clpfd>

PARADIGME DE PROGRAMARE

Curs 12

Antrenament pentru testul grilă.

Fuççnctionale

Se dă următorul program Racket:

```
(define L '(1 2 3 4 5 6 7 8))  
(foldl (lambda (elem acc) (if (odd? elem) acc (cons elem acc))) '() L)  
(foldr (lambda (elem acc) (if (even? elem) (cons elem acc) acc)) '() L)  
(filter odd? L)  
(map (lambda (x) (if (even? x) x)) L)
```

Aplicaça cârei funcâii se va evalua la lista '(2 4 6 8)?

- (a) foldl
- (b) filter
- (c) foldr
- (d) map

Fuççnctionale

Se dă următorul program Racket:

```
(define L '(1 2 3 4 5 6 7 8))  
(foldl (lambda (elem acc) (if (odd? elem) acc (cons elem acc))) '() L)  
(foldr (lambda (elem acc) (if (even? elem) (cons elem acc) acc)) '() L)  
(filter odd? L)  
(map (lambda (x) (if (even? x) x)) L)
```

Aplicaça cârei funcâii se va evalua la lista '(2 4 6 8)?

- (a) foldl
- (b) filter
- (c) foldr
- (d) map

Funcționale

La ce se va evalua următoarea expresie Haskell?

```
filter id $ zipWith (<) [1, 2, 3, 4] [4, 3, 2, 1]
```

- (a) [True, True, False, False]
- (b) [True, True]
- (c) [False, False]
- (d) [False, False, True, True]

Funcționale

La ce se va evalua următoarea expresie Haskell?

```
filter id $ zipWith (<) [1, 2, 3, 4] [4, 3, 2, 1]
```

- (a) [True, True, False, False]
- (b) [True, True]
- (c) [False, False]
- (d) [False, False, True, True]

Funcționale

La ce se va evalua următoarea expresie Haskell?

```
take 4 $ iterate (\x -> (head x + 1) : x) [1]
```

- (a) [[1], [2,1], [3,2,1], [4,3,2,1]]
- (b) [[1], [2], [3], [4]]
- (c) [[1], [1,2], [1,2,3], [1,2,3,4]]
- (d) [1,2,3,4]

Funcționale

La ce se va evalua următoarea expresie Haskell?

```
take 4 $ iterate (\x -> (head x + 1) : x) [1]
```

- (a) [[1], [2,1], [3,2,1], [4,3,2,1]]
- (b) [[1], [2], [3], [4]]
- (c) [[1], [1,2], [1,2,3], [1,2,3,4]]
- (d) [1,2,3,4]

Funcționale

La ce se va evalua următoarea expresie Racket?

```
(foldl (lambda (x y) (x y (x y))) 2 (list * +))
```

- (a) 16
- (b) 8
- (c) 2
- (d) 6

Funcționale

La ce se va evalua următoarea expresie Racket?

```
(foldl (lambda (x y) (x y (x y))) 2 (list * +))
```

- (a) 16
- (b) 8
- (c) 2
- (d) 6

Funcționale

La ce se va evalua următoarea expresie Racket?

`(apply map append ' (((1 2)) ((3 4))))`

- (a) (1 3 2 4)
- (b) ((1 3) (2 4))
- (c) ((1 2 3 4))
- (d) (1 2 3 4)

Funcționale

La ce se va evalua următoarea expresie Racket?

```
(apply map append ' (((1 2)) ((3 4))))
```

- (a) (1 3 2 4)
- (b) ((1 3) (2 4))
- (c) ((1 2 3 4))
- (d) (1 2 3 4)

Domeniul de vizibilitate a variabilelor

Ce va afișa următorul program Racket?

```
(define x 1)
(define y 2)
(let* ([x 42] [y (+ y 1)] [z (+ x 1)])
  (+ x y z))
```

- (a) Programul intră în buclă infinită
- (b) 6
- (c) 47
- (d) 88

Domeniul de vizibilitate a variabilelor

Ce va afișa următorul program Racket?

```
(define x 1)
(define y 2)
(let* ([x 42] [y (+ y 1)] [z (+ x 1)])
  (+ x y z))
```

- (a) Programul intră în buclă infinită
- (b) 6
- (c) 47
- (d) 88

Domeniul de vizibilitate a variabilelor

Ce va afișa următorul program Racket?

```
(define x 2)
(let ([x 1] [y 2] [f (delay (lambda (y) (+ x y)))])
  (let ([x 5])
    ((force f) x)))
```

- (a) 3
- (b) 4
- (c) 5
- (d) 7

Domeniul de vizibilitate a variabilelor

Ce va afișa următorul program Racket?

```
(define x 2)
(let ([x 1] [y 2] [f (delay (lambda (y) (+ x y)))])
  (let ([x 5])
    ((force f) x)))
```

- (a) 3
- (b) 4
- (c) 5
- (d) 7

Închideri funcționale

Ce va întoarce următorul program Racket?

```
(define f (lambda () f))  
(f)
```

- (a) O eroare
- (b) O promisiune
- (c) O funcție
- (d) Programul intră în buclă infinită

Închideri funcționale

Ce va întoarce următorul program Racket?

```
(define f (lambda () f))  
(f)
```

- (a) O eroare
- (b) O promisiune
- (c) O funcție
- (d) Programul intră în buclă infinită

Închideri / promisiuni / funcții nestrictate

Care dintre următoarele expresii va produce eroare în Racket?

- (a) (or #f (/ 1 0) #t)
- (b) (lambda (x) (car cdr))
- (c) (delay (/ 1 0))
- (d) (lambda () (/ 1 0))

Închideri / promisiuni / funcții nestrictate

Care dintre următoarele expresii va produce eroare în Racket?

- (a) (or #f (/ 1 0) #t)
- (b) (lambda (x) (car cdr))
- (c) (delay (/ 1 0))
- (d) (lambda () (/ 1 0))

Închideri funcționale

Fie definițiile următoarelor funcții în Racket:

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

Care este asemănarea/diferența dintre apelurile $(f\ 10)$ și $(g\ 10)$?

- (a) Ambele apeluri vor cicla la infinit
- (b) Primul va cicla la infinit, iar al doilea va întoarce o funcție
- (c) Primul va întoarce o funcție, iar al doilea va cicla la infinit
- (d) Ambele apeluri vor întoarce o funcție

Închideri funcționale

Fie definițiile următoarelor funcții în Racket:

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

Care este asemănarea/diferența dintre apelurile `(f 10)` și `(g 10)`?

- (a) Ambele apeluri vor cicla la infinit
- (b) Primul va cicla la infinit, iar al doilea va întoarce o funcție
- (c) Primul va întoarce o funcție, iar al doilea va cicla la infinit
- (d) Ambele apeluri vor întoarce o funcție

Promisiuni

Ce rezultat va întoarce următorul program Racket:

```
(define x 10)
(define pr (lambda (y) (delay (+ x y))))
(+ (force (pr 2)) (force (pr 3)) (force (pr 5)))
```

- (a) 36
- (b) 40
- (c) Eroare
- (d) 10

Promisiuni

Ce rezultat va întoarce următorul program Racket:

```
(define x 10)
(define pr (lambda (y) (delay (+ x y))))
(+ (force (pr 2)) (force (pr 3)) (force (pr 5)))
```

- (a) 36
- (b) 40
- (c) Eroare
- (d) 10

Promisiuni

Ce va afișa următorul program Racket:

```
(define x 10)
(define y (delay (+ x 10)))
(let ([x 20])
  (force y))
(define x 30)
(force y)
```

- (a) 20 40
- (b) 30 40
- (c) 30 30
- (d) 20 20

Promisiuni

Ce va afișa următorul program Racket:

```
(define x 10)
(define y (delay (+ x 10)))
(let ([x 20])
  (force y))
(define x 30)
(force y)
```

- (a) 20 40
- (b) 30 40
- (c) 30 30
- (d) 20 20

Evaluare

Care din următoarele expresii NU va produce eroare?

Racket: (define x (/ (/ 10 2) (let ([z 0]) z)))

Haskell: let x = (10 `div` 2) / 0

Prolog: X = Y / 0, Y is 10 / 2.

- (a) Doar cea scrisă în Haskell
- (b) Cele scrise în Prolog și Haskell
- (c) Cele scrise în Racket și Haskell
- (d) Cele scrise în Racket și Prolog

Evaluare

Care din următoarele expresii NU va produce eroare?

Racket: (define x (/ (/ 10 2) (let ([z 0]) z)))

Haskell: let x = (10 `div` 2) / 0

Prolog: X = Y / 0, Y is 10 / 2.

- (a) Doar cea scrisă în Haskell
- (b) Cele scrise în Prolog și Haskell
- (c) Cele scrise în Racket și Haskell
- (d) Cele scrise în Racket și Prolog

Secțiuni

Având programul următor în Haskell (prima linie poate fi ignorată):

```
{ -# LANGUAGE FlexibleInstances #-}  
instance (Eq a, Num a) => Eq (a -> a) where  
    f == g = (f 1) == (g 1)
```

Ce va afișa apelul următor: map (== (+ 1)) [(+ 1), (+ 2), (/ 2), (* 2)]

- (a) [True, False, False, False]
- (b) [True, False, False, True]
- (c) Eroare
- (d) [False, False, False, False]

Secțiuni

Având programul următor în Haskell (prima linie poate fi ignorată):

```
{ -# LANGUAGE FlexibleInstances #-}  
instance (Eq a, Num a) => Eq (a -> a) where  
    f == g = (f 1) == (g 1)
```

Ce va afișa apelul următor: map (== (+ 1)) [(+ 1), (+ 2), (/ 2), (* 2)]

- (a) [True, False, False, False]
- (b) [True, False, False, True]
- (c) Eroare
- (d) [False, False, False, False]

Secțiuni

Ce rezultat are codul Haskell de mai jos?

(+) ((10 /) (-5)) \\$ (2-) 1

- (a) 2.5
- (b) -2.5
- (c) 5
- (d) -1

Secțiuni

Ce rezultat are codul Haskell de mai jos?

```
(+) ((10 /) (-5)) \$ (2-) 1
```

- (a) 2.5
- (b) -2.5
- (c) 5
- (d) -1

Punct și dolar

Fie următoarele definiții Haskell:

```
f1 = filter odd . map (+1)
f2 = map (+1) . filter even
f3 = filter odd $ map (+1)
f4 = map (+1) $ filter even
```

Care din următoarele afirmații este adevărată?

- (a) $f_1[2,5..10] == f_2[2,5..10]$
- (b) $f_1[2,5..10] == f_3[2,5..10]$ și $f_2[2,5..10] == f_4[2,5..10]$
- (c) $f_1[2,5..10] == f_2[3,6..11]$ și $f_3[2,5..10] == f_4[3,6..11]$
- (d) $f_1[2,5..10] == f_2[1,4..9]$ și $f_3[2,5..10] == f_4[1,4..9]$

Punct și dolar

Fie următoarele definiții Haskell:

```
f1 = filter odd . map (+1)
f2 = map (+1) . filter even
f3 = filter odd $ map (+1)
f4 = map (+1) $ filter even
```

Care din următoarele afirmații este adevărată?

- (a) $f_1[2,5..10] == f_2[2,5..10]$
- (b) $f_1[2,5..10] == f_3[2,5..10]$ și $f_2[2,5..10] == f_4[2,5..10]$
- (c) $f_1[2,5..10] == f_2[3,6..11]$ și $f_3[2,5..10] == f_4[3,6..11]$
- (d) $f_1[2,5..10] == f_2[1,4..9]$ și $f_3[2,5..10] == f_4[1,4..9]$

TDA

Se dă tipul de date Haskell:

```
data MyUnion = UInt Int | UFloat Float | Complex Float Float deriving (Show, Eq)
```

Care din următoarele expresii va genera o eroare la compilare?

- (a) x = UInt 42
- (b) x = UFloat 42.0
- (c) x = MyUnion
- (d) x = Complex 42.0

TDA

Se dă tipul de date Haskell:

```
data MyUnion = UInt Int | UFloat Float | Complex Float Float deriving (Show, Eq)
```

Care din următoarele expresii va genera o eroare la compilare?

- (a) x = UInt 42
- (b) x = UFloat 42.0
- (c) x = MyUnion
- (d) x = Complex 42.0

Scheme de tip

Ce tip are următoarea funcție în Haskell?

$f \ x \ y \ z = fst \ (z, [x, y])$

- (a) $a \rightarrow b \rightarrow c \rightarrow c$
- (b) $a \rightarrow a \rightarrow a \rightarrow a$
- (c) $a \rightarrow a \rightarrow b \rightarrow b$
- (d) $a \rightarrow b \rightarrow a \rightarrow b$

Scheme de tip

Ce tip are următoarea funcție în Haskell?

$f \ x \ y \ z = fst \ (z, [x, y])$

- (a) $a \rightarrow b \rightarrow c \rightarrow c$
- (b) $a \rightarrow a \rightarrow a \rightarrow a$
- (c) $a \rightarrow a \rightarrow b \rightarrow b$
- (d) $a \rightarrow b \rightarrow a \rightarrow b$

Scheme de tip

Ce tip are expresia Add . Inv?

```
data Point = Point Int Int | Add Point Point | Inv Point
```

- (a) Point -> Point
- (b) Point -> Point -> Point
- (c) Int -> Int -> Point
- (d) Eroare de sinteză de tip

Scheme de tip

Ce tip are expresia Add . Inv?

```
data Point = Point Int Int | Add Point Point | Inv Point
```

- (a) Point -> Point
- (b) Point -> Point -> Point
- (c) Int -> Int -> Point
- (d) Eroare de sinteză de tip

Scheme de tip

Care este tipul următoarei expresii în Haskell?

map (+) [1, 2, 3]

- (a) Num a => [a -> a]
- (b) Num a => [a]
- (c) Num a => a
- (d) Eroare de sinteză de tip

Scheme de tip

Care este tipul următoarei expresii în Haskell?

map (+) [1, 2, 3]

- (a) Num a => [a -> a]
- (b) Num a => [a]
- (c) Num a => a
- (d) Eroare de sinteză de tip

Scheme de tip

Care este tipul următoarei funcții în Haskell?

$f \ x \ y \ z = x . y \$ z$

- (a) $(b \rightarrow c) \rightarrow (s \rightarrow b) \rightarrow s \rightarrow c$
- (b) Eroare de sinteză de tip
- (c) $(b \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow b \rightarrow c$
- (d) $(b \rightarrow s) \rightarrow (c \rightarrow b) \rightarrow s \rightarrow c$

Scheme de tip

Care este tipul următoarei funcții în Haskell?

$f \ x \ y \ z = x . y \$ z$

- (a) $(b \rightarrow c) \rightarrow (s \rightarrow b) \rightarrow s \rightarrow c$
- (b) Eroare de sinteză de tip
- (c) $(b \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow b \rightarrow c$
- (d) $(b \rightarrow s) \rightarrow (c \rightarrow b) \rightarrow s \rightarrow c$

Scheme de tip

Care este tipul următoarei funcții în Haskell?

$f \ x \ y \ z = map \ x \ . \ y \ \$ \ z$

- (a) $(c \rightarrow b) \rightarrow (d \rightarrow a \rightarrow c) \rightarrow d \rightarrow [a] \rightarrow [b]$
- (b) $(c \rightarrow a \rightarrow b) \rightarrow (d \rightarrow c) \rightarrow d \rightarrow [a] \rightarrow [b]$
- (c) $(c \rightarrow b) \rightarrow (a \rightarrow c) \rightarrow [a] \rightarrow [b]$
- (d) $(a \rightarrow b) \rightarrow (c \rightarrow [a]) \rightarrow c \rightarrow [b]$

Scheme de tip

Care este tipul următoarei funcții în Haskell?

$f \ x \ y \ z = map \ x \ . \ y \ \$ \ z$

- (a) $(c \rightarrow b) \rightarrow (d \rightarrow a \rightarrow c) \rightarrow d \rightarrow [a] \rightarrow [b]$
- (b) $(c \rightarrow a \rightarrow b) \rightarrow (d \rightarrow c) \rightarrow d \rightarrow [a] \rightarrow [b]$
- (c) $(c \rightarrow b) \rightarrow (a \rightarrow c) \rightarrow [a] \rightarrow [b]$
- (d) $(a \rightarrow b) \rightarrow (c \rightarrow [a]) \rightarrow c \rightarrow [b]$

Polimorfism și clase

Care este semnificația codului următor?

class Num a => Fractional a **where**

- (a) Fractional a este un tip de date care aparține clasei Num a
- (b) Clasa Num a poate fi instantiată numai de tipuri de numere fracționare
- (c) Dacă tipul a este membru al clasei Num, atunci el este și membru al clasei Fractional
- (d) Pentru ca tipul a să fie membru al clasei Fractional, el trebuie să fie membru al clasei Num

Polimorfism și clase

Care este semnificația codului următor?

class Num a => Fractional a **where**

- (a) Fractional a este un tip de date care aparține clasei Num a
- (b) Clasa Num a poate fi instanțiată numai de tipuri de numere fracționare
- (c) Dacă tipul a este membru al clasei Num, atunci el este și membru al clasei Fractional
- (d) Pentru ca tipul a să fie membru al clasei Fractional, el trebuie să fie membru al clasei Num

Polimorfism și clase

Ce fel de polimorfism utilizează următoarea funcție în Haskell?

```
f :: Ord a => a -> a -> b -> b -> b  
f x y a b = if x < y then a else b
```

- (a) Exclusiv parametric
- (b) Exclusiv ad-hoc
- (c) Atât parametric cât și ad-hoc
- (d) Funcția nu este polimorfică

Polimorfism și clase

Ce fel de polimorfism utilizează următoarea funcție în Haskell?

```
f :: Ord a => a -> a -> b -> b -> b  
f x y a b = if x < y then a else b
```

- (a) Exclusiv parametric
- (b) Exclusiv ad-hoc
- (c) Atât parametric cât și ad-hoc
- (d) Funcția nu este polimorfică

Ce întoarce codul?

Care din următoarele expresii Haskell generează lista numerelor impare de la 1 la 10?

- (a) `[x | x <- [1 ..], x < 11, x `mod` 2 == 1]`
- (b) `[x | x <- [1 .. 10], not $ even x]`
- (c) `[odd x | x <- [1 .. 10]]`
- (d) `[x | x > 0, x < 11, odd x]`

Ce întoarce codul?

Care din următoarele expresii Haskell generează lista numerelor impare de la 1 la 10?

- (a) `[x | x <- [1 ..], x < 11, x `mod` 2 == 1]`
- (b) `[x | x <- [1 .. 10], not $ even x]`
- (c) `[odd x | x <- [1 .. 10]]`
- (d) `[x | x > 0, x < 11, odd x]`

Ce afișează codul?

Ce afișează (prin funcția `display`) codul de mai jos? (fără rezultatul final)

```
(define (f n)
  (cond [(< n 2) 1]
        [else (display n) (+ (f (- n 2)) (f (- n 4))))]))
```

(f 8)

- (a) 8 6 4 2 2 4 2
- (b) 8 6 6 4 4 4 2 2 2 2
- (c) 8 6 4 4 2 2 2
- (d) 8 6 4 2 4 2 2

Ce afișează codul?

Ce afișează (prin funcția `display`) codul de mai jos? (fără rezultatul final)

```
(define (f n)
  (cond [(< n 2) 1]
        [else (display n) (+ (f (- n 2)) (f (- n 4))))]))
```

(f 8)

(a) 8 6 4 2 2 4 2

(b) 8 6 6 4 4 4 2 2 2 2

(c) 8 6 4 4 2 2 2

(d) 8 6 4 2 4 2 2

Unificare

Câte soluții va avea următoarea interogare în Prolog?

```
append([x | y], z, [1, 2, 3]).
```

- (a) O infinitate
- (b) 4
- (c) 1
- (d) 3

Unificare

Câte soluții va avea următoarea interogare în Prolog?

```
append([x | y], z, [1, 2, 3]).
```

- (a) O infinitate
- (b) 4
- (c) 1
- (d) 3

Unificare și instanțiere

Fie următoarele definiții în Prolog pentru funcția de maxim:

```
max(X, Y, X) :- X > Y, !.  
max(_, Y, Y).
```

Ce afișează următoarele două interogări?

```
max(3, 2, 2).  
max(2, 3, 2).
```

- (a) false și false
- (b) true și true
- (c) false și true
- (d) true și false

Unificare și instanțiere

Fie următoarele definiții în Prolog pentru funcția de maxim:

```
max(X, Y, X) :- X > Y, !.  
max(_, Y, Y).
```

Ce afișează următoarele două interogări?

```
max(3, 2, 2).  
max(2, 3, 2).
```

- (a) false și false
- (b) true și true
- (c) false și true
- (d) true și false

Unificare și instanțiere

Câte soluții întoarce Prolog pentru interogarea următoare și ce valori are variabila Z?

```
member(X, [1,2,3]), Y == X, !, Z is X + Y.
```

- (a) Trei: 2, 4, 6
- (b) Zero (eșuează)
- (c) Una: 2
- (d) Nouă: 2, 3, 4, 3, 4, 5, 4, 5, 6

Unificare și instanțiere

Câte soluții întoarce Prolog pentru interogarea următoare și ce valori are variabila Z?

```
member(X, [1,2,3]), Y == X, !, Z is X + Y.
```

- (a) Trei: 2, 4, 6
- (b) Zero (eșuează)
- (c) Una: 2
- (d) Nouă: 2, 3, 4, 3, 4, 5, 4, 5, 6

Unificare și instanțiere

Care dintre următoarele două interogări vor fi satisfăcute (vor întoarce true)?

?- $X == Y, X = Y.$

?- $X = Y, X == Y.$

- (a) A doua
- (b) Ambele
- (c) Niciuna
- (d) Prima

Unificare și instanțiere

Care dintre următoarele două interogări vor fi satisfăcute (vor întoarce true)?

?- $X == Y, X = Y.$

?- $X = Y, X == Y.$

- (a) A doua
- (b) Ambele
- (c) Niciuna
- (d) Prima

Cut

Care este diferența dintre:

p :- a, b.

p :- c.

și

p :- a, !, b.

p :- c.

- (a) Niciuna
- (b) Primul este $p = a \wedge b \wedge c$, al doilea este $p = a \wedge (b \vee c)$
- (c) Primul este $p = (a \wedge b) \vee c$, al doilea este $p = (a \wedge b) \vee (\neg a \wedge c)$
- (d) Primul este $p = (a \wedge b) \vee c$, al doilea este $p = (a \wedge b) \vee (b \wedge c)$

Cut

Care este diferența dintre:

p :- a, b.

p :- c.

și

p :- a, !, b.

p :- c.

- (a) Niciuna
- (b) Primul este $p = a \wedge b \wedge c$, al doilea este $p = a \wedge (b \vee c)$
- (c) **Primul este $p = (a \wedge b) \vee c$, al doilea este $p = (a \wedge b) \vee (\neg a \wedge c)$**
- (d) Primul este $p = (a \wedge b) \vee c$, al doilea este $p = (a \wedge b) \vee (b \wedge c)$

Cut

Considerând următoarele fapte Prolog:

```
p1(1). p1(2). p1(3). p1(4).
```

```
p2(2). p2(3).
```

Ce va afișa (exceptând false) interogarea

```
p1(X), p2(Y), write(X), write(Y), X >= Y, !, fail.
```

(a) 123223323423

(b) 1122

(c) 12322

(d) 121322

Cut

Considerând următoarele fapte Prolog:

p1(1). p1(2). p1(3). p1(4).

p2(2). p2(3).

Ce va afișa (exceptând false) interogarea

p1(X), p2(Y), write(X), write(Y), X >= Y, !, fail.

(a) 123223323423

(b) 1122

(c) 12322

(d) 121322

Cut

Cu ce se va solda încercarea de satisfacere a scopului $p(X)$?

$p(\textcolor{violet}{X}) :- !, q(\textcolor{violet}{X}), r(\textcolor{violet}{X}).$

$p(1).$ $p(2).$

$q(3).$ $q(4).$

$r(5).$

(a) $X = 1; X = 2.$

(b) $X = 1.$

(c) $\text{false}.$

(d) $X = 2.$

Cut

Cu ce se va solda încercarea de satisfacere a scopului $p(X)$?

$p(\textcolor{violet}{X}) :- !, q(\textcolor{violet}{X}), r(\textcolor{violet}{X}).$

$p(1).$ $p(2).$

$q(3).$ $q(4).$

$r(5).$

(a) $X = 1; X = 2.$

(b) $X = 1.$

(c) false.

(d) $X = 2.$

Cut

Cu ce se va solda încercarea de satisfacere a scopului $p(X)$?

$p(X) :- !, q(X), r(X).$

$q(1).$ $q(2).$

$r(1).$ $r(2).$

(a) $X = 1; X = 2.$

(b) $X = 1.$

(c) $\text{false}.$

(d) $X = 2.$

Cut

Cu ce se va solda încercarea de satisfacere a scopului $p(X)$?

$p(X) :- !, q(X), r(X).$

$q(1).$ $q(2).$

$r(1).$ $r(2).$

(a) $X = 1; X = 2.$

(b) $X = 1.$

(c) $\text{false}.$

(d) $X = 2.$

Negăția ca eșec

Câte rezultate va avea următoarea interogare? (de câte ori va fi satisfăcută?)

?- L1 = [1,2,3,4,5], L2 = [1,3,5], \+ (member(X, L2), \+ member(X, L1)).

- (a) Niciodată
- (b) O dată
- (c) De două ori
- (d) De trei ori

Negăția ca eșec

Câte rezultate va avea următoarea interogare? (de câte ori va fi satisfăcută?)

?- L1 = [1,2,3,4,5], L2 = [1,3,5], \+ (member(X, L2), \+ member(X, L1)).

- (a) Niciodată
- (b) O dată
- (c) De două ori
- (d) De trei ori

Metapredicate

Fie următoarele fapte în cadrul unui program Prolog:

```
p(1). p(2). p(4).  
q(1). q(4). q(8).
```

Ce va afișa următoarea interogare?

```
?- setof(X/Y, (p(X), q(Y), not(q(X))), L).
```

- (a) $L = [2/1, 2/4, 2/8].$
- (b) $L = [2.0, 0.5, 0.25].$
- (c) $L = [1/4, 1/8, 2/1, 2/4, 2/8, 4/1, 4/8].$
- (d) $L = [0.25, 0.125, 2.0, 0.5, 4.0].$

Metapredicate

Fie următoarele fapte în cadrul unui program Prolog:

```
p(1). p(2). p(4).  
q(1). q(4). q(8).
```

Ce va afișa următoarea interogare?

```
?- setof(X/Y, (p(X), q(Y), not(q(X))), L).
```

- (a) $L = [2/1, 2/4, 2/8].$
- (b) $L = [2.0, 0.5, 0.25].$
- (c) $L = [1/4, 1/8, 2/1, 2/4, 2/8, 4/1, 4/8].$
- (d) $L = [0.25, 0.125, 2.0, 0.5, 4.0].$

Metapredicate

Fie următoarele fapte și reguli în cadrul unui program Prolog:

```
foo(2, 3, 1). foo(2, 3, 6).
```

```
foo(1, 2, 3). foo(1, 3, 4). foo(1, 2, 5).
```

```
p(X) :- bagof(C, (foo(A, B, C), C mod 2 > 0), [X, Y]).
```

Ce va afișa interogarea p(X)?

- (a) X = 3; X = 5.
- (b) X = 1; X = 3.
- (c) X = 3.
- (d) X = 1.

Metapredicate

Fie următoarele fapte și reguli în cadrul unui program Prolog:

```
foo(2, 3, 1). foo(2, 3, 6).
```

```
foo(1, 2, 3). foo(1, 3, 4). foo(1, 2, 5).
```

```
p(X) :- bagof(C, (foo(A, B, C), C mod 2 > 0), [X, Y]).
```

Ce va afișa interogarea p(X)?

- (a) X = 3; X = 5.
- (b) X = 1; X = 3.
- (c) X = 3.
- (d) X = 1.

Metapredicate

Fie următoarele fapte în cadrul unui program Prolog:

a(1). b(1).

a(2). b(2).

a(3). b(3).

Ce va afișa următoarea interogare?

?- findall(X/Y, (a(X), !, b(Y); (a(Y), b(X))), L).

(a) L = [1/1, 2/2, 3/3].

(b) L = [1/1, 2/2, 3/3, 1/1, 2/2, 3/3].

(c) L = [1/2, 1/3, 2/1, 2/3, 3/1, 3/2].

(d) L = [1/1, 1/2, 1/3].

Metapredicate

Fie următoarele fapte în cadrul unui program Prolog:

a(1). b(1).

a(2). b(2).

a(3). b(3).

Ce va afișa următoarea interogare?

?- findall(X/Y, (a(X), !, b(Y); (a(Y), b(X))), L).

(a) L = [1/1, 2/2, 3/3].

(b) L = [1/1, 2/2, 3/3, 1/1, 2/2, 3/3].

(c) L = [1/2, 1/3, 2/1, 2/3, 3/1, 3/2].

(d) L = [1/1, 1/2, 1/3].

PARADIGME DE PROGRAMARE

Curs 13

Rezumat.

Cuprins

- Rezolvare de probleme
- Dezvoltarea unui program bun
- Controlul complexității intelectuale
- Controlul complexității temporale / spațiale
- Controlul corectitudinii programului
- Concluzii

Rezolvarea problemelor în viața de zi cu zi

"Everyone in this country should learn to program a computer, because it teaches you to think." – Steve Jobs

- Abilitatea de a rezolva probleme se învață din experiență
- Strategia: să dezvolti o metodă de a aborda rezolvarea problemelor și să o practici
 - Rezolvând cât mai **multe microprobleme**
 - Nu doar probleme familiare, ci probleme care solicită noi și **noi moduri de a gândi**
- Contează **calitatea soluției** găsite (un lucru mai puțin neglijat în programare decât în viața de zi cu zi)
 - Mai ales în cazul problemelor recurente

Programele modeleză lumea

- Sistemele software sunt modele ale lumii înconjurătoare și sunt dezvoltate pentru a rezolva probleme din viața de zi cu zi
- Primul pas în dezvoltarea unui sistem bun este o bună **modelare a problemei**
 - Fiecare paradigmă de programare propune un mod particular de reprezentare / prelucrare
 - Programatorul este responsabil să aleagă modul cel mai adecvat particularităților problemei
 - Obiecte multe, operații puține: POO
 - Obiecte puține, operații multe: PF
 - Multe prelucrări numerice: programare procedurală
 - Îndeplinirea unui anumit scop: PL (programare logică)
 - Lumea este în continuă schimbare, iar programele trebuie să permită **extinderea ușoară** a universului problemei
 - Noi operații: ușor în PF, greu în POO
 - Noi obiecte: ușor în POO, greu în PF

Cuprins

- Rezolvare de probleme
- **Dezvoltarea unui program bun**
- Controlul complexității intelectuale
- Controlul complexității temporale / spațiale
- Controlul corectitudinii programului
- Concluzii

Caracteristicile unui program bun

- Corectitudine și robustețe
- Ușurință în utilizare și dezvoltare
 - Interactivitate
 - Lizibilitate
 - Extensibilitate
- Eficiență
- Documentare

Aspecte discutate în acest curs

- Abordarea sarcinii de a dezvolta un program pentru o anumită problemă
 - +
- Controlul complexității intelectuale a programelor
- Controlul complexității temporale / spațiale a programelor
- O disciplină care să asigure corectitudinea programelor
- Un stil de programare care facilitează buna documentare

Abordarea scrierii unui program

1) Înțelege problema (fii capabil să o explici în cuvinte)

- "If you can't explain something in simple terms, you don't understand it." —Richard Feynman

2) Planifică

- Aspecte legate de modelare și de o strategie de ansamblu (formulare top-level)

3) Divide (problema în subprobleme)

- **Wishful thinking**: formularea top-level a rezolvării identifică la nivel conceptual niște subprobleme mari, pe care îi le imaginezi gata rezolvate și pe baza cărora scrii programul
 - **Recursivitatea** ca formă de wishful thinking: presupui că există o procedură care rezolvă problema pentru versiuni mai mici decât cea curentă
- Fiecare subproblemă este **rafinată succesiv** în același mod; păstrând fiecare pas de rafinare (detaliere) mic, și sub control complexitatea intelectuală a întregului sistem
 - Rafinarea prezentă în orice paradigmă: la nivel de structuri de date, de funcții (PF), de relații (PL)

Cuprins

- Rezolvare de probleme
- Dezvoltarea unui program bun
- **Controlul complexității intelectuale**
- Controlul complexității temporale / spațiale
- Controlul corectitudinii programului
- Concluzii

Controlul complexității intelectuale

- Ciclul primitive – mijloace de combinare – abstractizare
 - Unealtă de bază pentru design-ul, întreținerea și extinderea sistemelor software
- Polimorfism
- Pattern matching
- Stil declarativ versus stil procedural

Elementele unui limbaj de programare

- **Primitive** – numere, stringuri, valori booleene, operatori aritmetici / relaționali / logici
- **Mijloace de combinare** (obțin structuri mai complexe pe baza celor simple)
 - Date compuse (liste, perechi, structuri (PL)), aplicația de funcție, predicate (PL)
- **Mijloace de abstractizare** (ascund detaliile și tratează lucrurile complexe ca pe primitive)
 - Posibilitatea de a da nume funcțiilor și expresiilor (și a le folosi apoi ca pe niște primitive)
 - Funcții/predicate ca niște cutii negre: izolarea utilizării unei proceduri de implementarea ei
 - **Modularitate**: Izolează componentele unui sistem și furnizează o interfață (input/output) pentru conectarea părților
 - **Tipuri de date abstracte** – ascund felul în care sunt grupate componente și tratează fiecare unitate ca pe o colecție abstractă
 - **Proceduri de nivel înalt**: funcționale (PF), metapredicale (PL) – surprind şablonane frecvente de calcul (care pot avea ca parametri alte şablonane)

Bariera de abstractizare

- TDA = colecție de componente controlabilă printr-o interfață
 - **Constructorii** pun componentele împreună
 - **Selectori** desfac întregul pentru a accesa componentele
 - În programele cu efecte laterale: funcții speciale (**mutator**) pentru alterarea componentelor
- Această interfață permite existența barierei de abstractizare – care ascunde de utilizator detaliile de implementare a tipului
 - **Barieră slabă**: utilizatorul cunoaște implementarea internă a tipului și e responsabilitatea lui să scrie cod care nu calcă bariera (ex: știe că listele sunt implementare ca perechi, sau știe că numerele complexe sunt implementate ca perechi)
 - **Barieră puternică**: utilizatorul nu cunoaște implementarea internă a tipului și nu poate să manipuleze valorile decât folosind interfața (ex: nu știe cum sunt implementate perechile)
 - Disciplina impusă este mai sigură decât cea auto-impusă ☺

Şabloane frecvente de calcul

- Transformarea fiecărui element dintr-o listă: **map**, **maplist** (Prolog)
- Găsirea tuturor elementelor care îndeplinesc o condiție: **filter**, **findall** / **bagof** / **setof**
- Verificarea că toate elementele îndeplinesc o condiție: **forall**
- Acumularea (eventual combinată cu prelucrarea) tuturor elementelor: **fold**

Efecte

- Programe mai scurte și mai ușor de citit
- Crește **puterea expresivă** a limbajului (aceste şabloane devin blocuri pe care le putem trata ca pe primitive)
- Se creează disciplina de a abstractiza și denumi sugestiv și alte şabloane frecvente

Planificarea programului

Design-ul programului presupune:

- **Design-ul structurilor / tipurilor de date**
 - ce informație este natural să fie grupată și ce interfață (care ascunde detaliile) trebuie creată pentru ea
- **Design-ul modulelor de calcul**
 - Se abstractizează: calculele des utilizate, concepte (pași în calcul care îndeplinesc un rol care poate fi ușor numit și înțeles ca un bloc independent)
 - Fiecare modul se poate implementa și testa separat
- **Design-ul interfețelor între module**
 - Se identifică input-ul și output-ul fiecărui modul
 - Se modeleză fluxul informației prin sistem

Polimorfism

Polimorfism parametric = funcția se comportă la fel pentru argumente de tipuri diferite

Polimorfism ad-hoc = funcția se comportă diferit în funcție de tipul argumentelor

- **Claritate** sporită a codului
 - Putem folosi un același nume sugestiv pentru a descrie operații pe diferite tipuri
 - **Contează conceptul** de egalitate / lungime / apartenență etc. mai mult decât implementarea lui specifică
- **Reutilizare:** folosind aceeași funcție pentru diverse tipuri avem un cod mai scurt, mai ușor de întreținut

Pattern matching

- Constructorii unui TDA: închid într-o abstracțiune mai multe informații
- Selectorii: desfac constructorii pentru a extrage una din aceste informații
 - Alternativ, se poate folosi pattern-matching pe constructori – metodă naturală, similară cu experiența umană de recunoaștere a semnificației unui anumit obiect
 - **Facilitat de tiparea statică** – se pot identifica argumentele constructorilor și lega variabilele la valori – v. Haskell, care nu face nicio altă verificare suplimentară
 - **Facilitat de algoritmul de unificare** – capabil să verifice identitatea oricăror 2 structuri, fără ca acestea să fie instanțe ale unui TDA bine definit în program sau în limbaj – v. Prolog, în care pattern matching-ul este mult mai puternic, permite atât verificări cât și crearea unor noi structuri care se potrivesc cu forma dorită

Stil declarativ versus stil procedural

Stil declarativ

- Natural pentru a reprezenta și a raționa despre ceea ce este adevărat
- Nu oferă o metodă computațională de a deduce noi informații
- Mai ușor de înțeles, dar necesită mecanisme specifice care să sprijine acest stil de programare
 - ex: backtracking încorporat în Prolog, evaluare leneșă pentru a folosi fluxuri, etc.
- Adesea este necesar un compromis între eleganță și eficiență

Stil procedural

- Natural pentru a reprezenta metode de calcul (secvențe de pași care transformă o mulțime de fapte într-o nouă mulțime de fapte de interes)
- Nu se aseamănă cu metoda umană de a reprezenta și deduce noi informații



Cuprins

- Rezolvare de probleme
- Dezvoltarea unui program bun
- Controlul complexității intelectuale
- **Controlul complexității temporale / spațiale**
- Controlul corectitudinii programului
- Concluzii

Controlul complexității temporale / spațiale

- Complexitatea operațiilor primitive
- Funcții și procesele generate de acestea
- Tipuri de evaluare: aplicativă, normală, lenesă
- Fluxuri

Complexitatea operațiilor primitive

- Ciclul primitive – mijloace de combinare – abstractizare
 - Face să putem trata operații complexe ca pe niște primitive
 - Atrage atenția asupra faptului că operațiile primitive (în limbaj) au o implementare în spate – care adesea nu e în complexitate constantă
 - Ex: length – complexitate $O(n)$ în Racket
– complexitate $O(1)$ în Python
- Alte exemple (aspecte de care trebuie ținut cont la alegerea limbajului)
 - Algoritmii de sortare implicați în diverse limbaje diferă (merge-sort, quick-sort)
 - Felul în care sunt implementate structurile de date standard (stive, cozi, vectori, hash-uri, heap-uri)

Funcții și procese

- Un anumit mod de a scrie funcțiile (de a descompune problema) duce la un anumit proces (secvență de pași efectuată în calculator pentru transformarea datelor)
- Tipurile de procese se caracterizează prin **complexitatea lor temporală / spațială**
 - Recursivitate **pe stivă** (consumă spațiu suplimentar din cauza alocărilor pe stivă)
 - Recursivitate **pe coadă** (funcția este recursivă, dar procesul generat este iterativ)
 - Recursivitate **arborescentă** (de tip divide and be conquered, duce la timpi exponențiali)
 - Recursivitate de tip **divide et impera** (timp logaritmice)
- **Exemplu:** pentru exponențiere (a^n), când n este par, putem descompune problema:
 - $a^n = (a^2)^{n/2}$
 - $a^n = (a^{n/2})^2$

Cum e mai bine?

Tipuri de evaluare

Evaluare aplicativă = evaluează argumentele înainte de apel

- Eficientă (expresia se poate înlocui prin valoarea ei), dar nu garantează terminarea calculului

Evaluare normală = evaluează subexpresiile în expresia finală, de la stânga la dreapta, când e nevoie de ele

Evaluare leneşă = ca evaluarea normală, dar salvează rezultatul evaluărilor pentru a nu reevalua o aceeași subexpresie de mai multe ori

- Evaluare normală + memoizare (necesară mecanisme speciale)
- Câștigul se vede în posibilitatea de a lucra cu structuri leneșe (indefinit de lungi)

Fluxuri

- Liste cu evaluare lenășă
- Decuplarea ordinii aparente a acțiunilor de ordinea reală:
 - Programe elegante, declarative (asemenea programelor pe liste)
 - Procese eficiente (asemenea proceselor iterative)
- Elimină compromisul eleganță – eficiență

Cuprins

- Rezolvare de probleme
- Dezvoltarea unui program bun
- Controlul complexității intelectuale
- Controlul complexității temporale / spațiale
- Controlul corectitudinii programului
- Concluzii

Controlul corectitudinii programului

- Efecte laterale și transparentă referențială
- Tipuri de legare a variabilelor
- Tipare
- Demonstrații de corectitudine

Efecte laterale și transparentă referențială

Efecte laterale = alte efecte ale unei funcții decât cel de a întoarce o valoare

- Ex: atribuirি, operațiї de intrare/ieșire
- Compromis între ușurințа de a obține anumite rezultate și riscul de efecte neprevăzute (bug-uri)
- Știind să programăm fără efecte laterale, limităm uzul acestora la situațiile în care este necesar

Transparentă referențială = funcții pure (aplicate pe aceleași argumente întorc mereu aceeași valoare), expresii perfect substituibile cu valoarea lor

- Valori imutabile, procedurile sunt funcții matematice
- Programe atemporale, ușor paralelizabile (fire paralele nu vor modifica o resursă comună), permite evaluare leneșă (expresiile odată evaluate nu trebuie reevaluate în caz că vor produce altă valoare)
- Nu contează ordinea de evaluare, nu contează strategia de evaluare, este ușor de prezis ce va întoarce o funcție (folosind modelul substituției) → risc scăzut de bug-uri

Efectele programării cu atribuiriri

Modelul substituției = se înlocuiesc parametrii formali ai funcției cu argumentele și se evaluează corpul

Modelul contextual = o funcție este un text și un context (care poate fi suprascris în timp)

Introducerea atribuirilor

- Modelul substituției trebuie înlocuit cu modelul contextual (mai complicat) pentru a include noțiunea de timp
- Expresii cu sintaxă identică pot avea semantică diferită (depinde de contextul în care sunt evaluate)
- O variabilă nu mai e un nume pentru o valoare, ci o locație în memorie → probleme de identitate (indică două variabile același obiect sau doar au aceeași valoare?)
- O procedură nu mai e o funcție, ci un obiect cu un context care ne spune cum să interpretăm simbolurile din calculul descris de procedură
- Necesitatea sincronizării firelor paralele, pentru cazurile în care mai multe fire vor să altereze aceeași zonă de memorie

Tipuri de legare a variabilelor

Legare statică = domeniu de vizibilitate controlat textual (determinat la compilare)

- Varianta implicită în majoritatea limbajelor de programare

Legare dinamică = domeniu de vizibilitate controlat dinamic (în funcție de timp) (determinat la execuție)

- Permite suprascrierea contextului moștenit de o funcție la definirea ei
- Risc de bug-uri ca de fiecare dată când introducem noțiunea de timp în programele noastre (☺)

Tipare

Tipare statică = variabilele și valorile au un tip asociat, tipuri verificate la compilare

- Ajută la design-ul programelor
 - Exprimăm funcțiile în funcție de tipul input-urilor și tipul output-ului
 - Ne asigurăm că se pasează valori valide de la o funcție la alta și că avem toate modulele necesare pentru a circula informația prin sistem
 - Inconsistențele vor fi semnalate la compilare

Tipare dinamică = doar valorile au un tip asociat, tipuri verificate la execuție

- Erorile nu vor fi detectate decât dacă le generează o secvență de execuție
- Flexibilitate în scrierea funcțiilor – programatorul are libertatea să își impună sau nu disciplina consistenței tipurilor
- Preferată în programele mici, problematică în sisteme complexe întreținute de un număr mare de programatori

Design ajutat de tipare - Exemplu

```
1. -- o funcție care primește o altă funcție fun și un număr n și
2. -- întoarce funcția care reprezintă aplicarea repetată de n ori a lui fun
3. {- 
4. repeated :: (a -> a) -> Int -> (a -> a)
5. repeated _ 0 = \x -> x
6. repeated fun n = \x -> fun (repeated fun (n-1) x)
7. -}
8. repeated :: (a -> a) -> Int -> (a -> a)
9. repeated _ 0 x = x
10. repeated fun n x = fun (repeated fun (n-1) x)
```

Observație: puteam folosi cunoșințe de tipuri și pentru a ghida design-ul funcției în Racket

Tipuri definite de utilizator

- Când realizăm abstractizare la nivel de date în Racket sau Prolog – de fapt definim un tip fără disciplina unui sistem de tipuri
 - Apoi depinde de noi să prelucrăm valorile tipului doar cu operațiile destinate lor
 - Alternativă (disciplină auto-impusă): etichetăm manual datele de un anumit tip și verificăm eticheta fiecărei valori înainte de a opera asupra ei

Exemple

```
(define (make-tree root left right) (list 'tree root left right))
(define (tree? T) (equal? 'tree (first T)))
(define root second)
(define left third)
(define right fourth)

root(tree(Root, _, _), Root).
left(tree(_, Left, _), Left).
right(tree(_, _, Right), Right).
```

Tipuri definite de utilizator

- Haskell impune această disciplină
 - TDA-urile definite de utilizator sunt gata „etichetate” cu constructorii specifici fiecărui tip (nu pot avea același nume de constructor pentru tipuri diferite)
 - doar date prelucrate cu operații specifice lor (**data directed programming**)
 - Aplicarea funcțiilor specifice unui tip necesită pattern matching pe constructori și vom genera o eroare (la compilare) dacă încercăm aplicarea pe un tip necorespunzător
 - doar operații aplicate pe tipul așteptat (**defensive programming**)

Exemplu

```
data Tree a = Nil | Node a (Tree a) (Tree a)
root  (Node r _ _) = r
left   (Node _ l _) = l
right  (Node _ _ r) = r
```

Alte argumente pro tipare

- Îmbunătățește **documentarea** programelor
 - Informații despre natura input-urilor și output-urilor fiecărei funcții
 - Constrângeri asupra argumentelor (ex: Eq, Ord, Num)
 - Documentarea trebuie completată cu informații despre scopul funcției și rolul fiecărui parametru
- Îmbunătățește **productivitatea**
 - Mulțumită IDE-urilor care oferă informații despre tipurile așteptate de fiecare funcție
 - Mai ales în sistemele cu inferență de tip (programatorul este asistat, nu forțat să adnoteze singur)
- Crește **eficiența**
 - Compilatorul optimizează reprezentarea / prelucrarea valorilor în funcție de particularitățile tipului

Demonstrații de corectitudine

- Cu atât mai ușoare (și de încredere) cu cât codul este mai apropiat de specificația formală
- Programarea funcțională se bazează pe funcții recursive, a căror corectitudine se poate demonstra prin **inducție** (inducție matematică, inducție structurală)
 - Demonstrează corectitudinea cazului de bază
 - Demonstrează cum dacă o versiune mai simplă a funcției este corectă, atunci prelucrarea ei cu operații adiționale duce la un răspuns corect pentru problema extinsă
 - Metoda inducției coincide cu metoda design-ului funcției recursive!
- Programarea logică se bazează pe aplicarea rezoluției (regulă de inferență consistentă și completă) asupra adevărurilor din program
 - Programele sunt corecte atât timp cât specificația formală este corectă

Cuprins

- Rezolvare de probleme
- Dezvoltarea unui program bun
- Controlul complexității intelectuale
- Controlul complexității temporale / spațiale
- Controlul corectitudinii programului
- Concluzii

Ce ați dobândit

- Experiență în rezolvarea de micro-probleme variate
- Experiență în felul în care gândiți calculul, independent de detaliile unui limbaj de programare, în scopul controlului complexității intelectuale a problemei
- Cultură generală privind alegerile care stau la baza design-ului limbajelor de programare
- O imagine a efectelor alegerilor voastre de design (efekte laterale, procese recursive sau iterative, abstractizare, disciplina tipurilor, etc.)
- Relația dintre diversele paradigmă de programare și formalismele matematice pe care ele se bazează

Alte idei

- Un limbaj facilitează / impune o paradigmă, dar puteți folosi disciplina unei paradigme și în alte limbaje

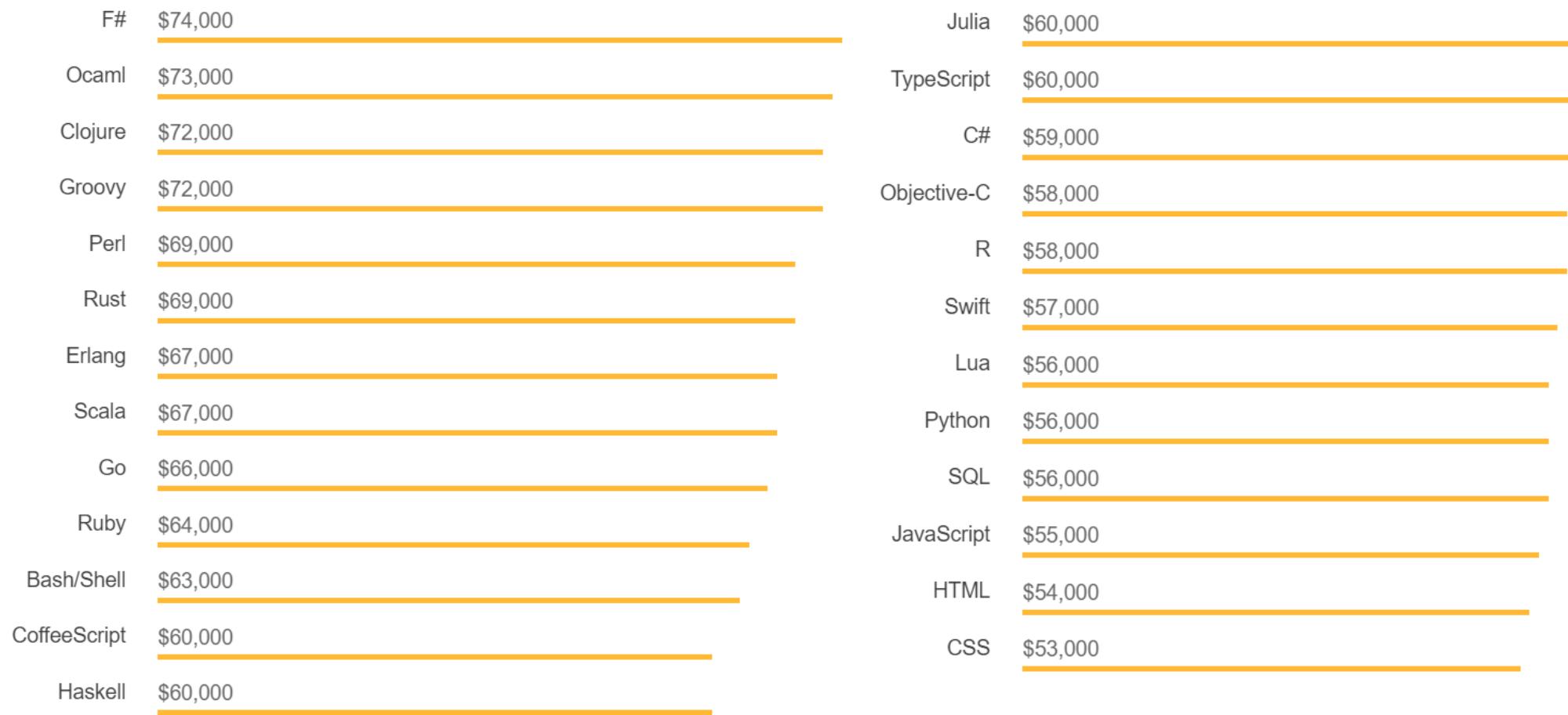
*„When you choose technology, you have to **ignore what other people are doing**, and consider only what will work the best. You can't trust the opinions of the others, because they're satisfied with whatever language they happen to use, because it dictates the way they think about programs.*

*Lisp is so great not because of some magic quality visible only to devotees, but because it is simply the most powerful language available. And the reason everyone doesn't use it is that **programming languages are not merely technologies, but habits of mind as well, and nothing changes slower.***

If you think of using Lisp in a startup, you shouldn't worry that it isn't widely understood. You should hope that it stays that way. Back when I was writing books about Lisp, I used to wish everyone understood it. But when we started Viaweb I found that changed: I wanted everyone to understand Lisp except our competitors.” – Paul Graham

Popular versus bine plătit

(<https://insights.stackoverflow.com/survey/2018/#top-paying-technologies>)



La revedere și...

Time for closure.

(λ ()
' Sesiune_plăcută!)