PARADIGME DE PROGRAMARE

Curs 1

Introducere. Modele de evaluare. Limbajul Racket. Recursivitate.

Administrative

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

- Cursuri, laboratoare, teme
- Catalog
- Exemple de examene și teste
- Regulament

Structura fiecărui curs

- Predare
- Test din cursul anterior (uneori)
- Rezumatul cursului curent

Obiective

Alternative la paradigmele 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, paradigme, limbaje – Cuprins

- Modele de calculabilitate
- Paradigme de programare
- Exemplu rezolvat în diferite paradigme/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

Calculul Lambda

(λx.λy.x+y 2) -> λy.2+y

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

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

Mașina Logică

```
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.

Modele, paradigme, limbaje – Cuprins

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

Exemplu

Să se determine factorialul unui număr natural n folosind paradigmele:

- Imperativă
- Funcțională
- Logică

Rezolvare imperativă

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

← datele sunt reținute în **variabilele** i, factorial

← rezolvarea este o **execuție succesivă** de înmulțiri

← 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)
2.    (if (zero? n)
3.        1
4.        (* n (factorial (- n 1)))))
```

← datele sunt reținute în **funcții** (totul este o funcție)

← caz de bază în **recursivitate** (corespunzător constructorilor nulari/externi)

← 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 **compunere 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).
2.  factorial(N, Result) :-
3.      N > 0,
4.      Prev is N-1,
5.      factorial(Prev, F),
6.      Result is N*F.
```

← datele sunt reținute în **fapte** (ex: factorial de 0 este 1)
și **reguli** (ex: factorial de N este Result, dacă:
N > 0 și
factorial de N-1 este F și
Result este N*F)

De reținut

- Asemănări cu paradigma funcțională: soluție declarativă, ciclarea înlocuită prin recursivitate, atribuirea înlocuită prin noi valori are 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` e_1 e_2 `...` e_n`)`. 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.

$\neq$

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 nestricte **(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 evaluează și reevaluează 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 (≠ 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 ≡	**x**	variabilă
|	**λx.e**	funcție (unară, anonimă) cu parametrul formal x și corpul e
|	**(e1 e2)**	aplicație a expresiei e1 asupra parametrului efectiv e2

Semantica (Modelul substituției)

Pentru a evalua **(λx.e1 e2)** (funcția cu parametrul formal x și corpul e1, aplicată pe e2):

- Peste tot în e1, identificatorul x este înlocuit cu e2
- Se evaluează noul corp e1 și se întoarce rezultatul (se notează $\mathbf{e1}_{[e2/x]}$)

Exemple de λ-expresii

λx.x

(x y)

λx.λy.(x y)

(λx.x a)

(λx.y a)

(λx.x λx.y)

Exemple de λ-expresii

λx.x	funcția identitate
(x y)	
λx.λy.(x y)	
(λx.x a)	
(λx.y a)	
(λx.x λx.y)	

Exemple de λ-expresii

λx.x

(x y) aplicația expresiei x asupra expresiei y

λx.λy.(x y)

(λx.x a)

(λx.y a)

(λx.x λx.y)

Exemple de λ-expresii

λx.x

(x y)

λx.λy.(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

(λx.x a)

(λx.y a)

(λx.x λx.y)

Exemple de λ-expresii

λx.x

(x y)

λx.λy.(x y)

(λx.x a) funcția identitate aplicată asupra lui a (se evaluează la a)

(λx.y a)

(λx.x λx.y)

Exemple de λ-expresii

λx.x

(x y)

λx.λy.(x y)

(λx.x a)

(λx.y a) funcția de parametru x cu corpul y, aplicată asupra lui a (se evaluează la y)

(λx.x λx.y)

Exemple de λ-expresii

λx.x

(x y)

λx.λy.(x y)

(λx.x a)

(λx.y a)

(λx.x λx.y) funcția identitate aplicată asupra funcției λx.y (se evaluează la λx.y)

Funcții anonime în Racket

```
(lambda listă-parametri corp)
```

Exemple

λx.x	`(lambda (x) x) ;; echivalent cu (λ (x) x)`

λx.λy.(x y)	`(lambda (x)`
	`  (lambda (y)`
	`  (x y)))`

(λx.x λx.y)	`((λ (x) x) (λ (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)
>((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 : T1 x T2 -> Pereche // creează o pereche cu punct între orice 2 argumente

Operatori

car : Pereche -> T1 // extrage prima valoare din pereche

cdr : Pereche -> T2 // 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 : T1 x T2 -> Pereche // creează o pereche cu punct între orice 2 argumente

Operatori

car : Pereche -> T1 // extrage prima valoare din pereche

cdr : Pereche -> T2 // 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 : T_1 x .. T_n -> Listă // creează o listă din toate argumentele sale

Operatori

car : Listă -> T

cdr : Listă -> Listă

null? : Listă -> Bool

length : Listă -> Nat

append : Listă x Listă -> Listă

Exemple

```
(car (list 1 'a +))
(cdr '(2 3 4 5))
(null? '())
(length (list))
(append (cons 1 '(2)) '(a b))
```

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 : T_1 x .. T_n -> Listă // creează o listă din toate argumentele sale

Operatori

car : Listă -> T

cdr : Listă -> Listă

null? : Listă -> Bool

length : Listă -> Nat

append : Listă x Listă -> Listă

Exemple

```
(car (list 1 'a +))              ;; 1
(cdr '(2 3 4 5))                 ;; '(3 4 5)
(null? '())                      ;; #t
(length (list))                  ;; 0
(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 evaluează la #f
2.       (+ 1 2)                 ;; NU se evaluează
3.       (- 7 1))                ;; întreg if-ul se evaluează la 6
4.   (if (< 4 10)                ;; se evaluează la #t
5.       'succes                 ;; întreg if-ul se evaluează la ‘succes
6.       (/ 'logica 0))          ;; NU se evaluează (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 evaluează doar condiția la #f
4.    ((null? (cdr L))  (/ 1 0))   ;; se evaluează doar condiția la #f
5.    (else             'other))   ;; întregul cond se evaluează 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

- Atribuiri
- Instrucțiuni de ciclare (for, while)
- Secvență de operații (o funcție se evaluează la o unică valoare și nu are efecte laterale)

Avem

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

Secvență de operații Ciclare Atribuiri

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 neelegant să precizez și comportamentul pentru liste de fix un element
- Orice **în minus e insuficient** și duce la eșecul 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 n L)
```

1) **După ce variabile fac recursivitatea? (ce variabile își schimbă valoarea de la un apel la altul?)**
   - Dacă aș 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 totuna 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 n cât și după L

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 (- n 1) (cdr L)))))
```

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 (λx.e), aplicație ((e1 e2))

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 … 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 $e_1\ e_2 \ldots e_n$)

Sintaxa funcțiilor Racket: (lambda ($x_1\ x_2 \ldots x_n$) corp) sau (define (f $x_1\ x_2 \ldots x_n$) 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 (λx.e), aplicație ((e_1 e_2))

Sintaxa expresiilor Racket: (funcție e_1 e_2 … e_n)

Sintaxa funcțiilor Racket: (lambda (x_1 x_2 … x_n) corp) sau (define (f x_1 x_2 … 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 (λx.e), aplicație ((e_1 e_2))

Sintaxa expresiilor Racket: (funcție e_1 e_2 … e_n)

Sintaxa funcțiilor Racket: (lambda (x_1 x_2 … x_n) corp) sau (define (f x_1 x_2 … x_n) corp)

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

Operatori condiționali: 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 (λx.e), aplicație ((e_1 e_2))

Sintaxa expresiilor Racket: (funcție e_1 e_2 … e_n)

Sintaxa funcțiilor Racket: (lambda (x_1 x_2 … x_n) corp) sau (define (f x_1 x_2 … x_n) corp)

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

Operatori condiționali: 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 iterației)
- 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:** Θ(n) (se efectuează n înmulțiri și stiva se redimensionează de 2*n ori)
- **Spațiu:** Θ(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

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

- **Timp:** Θ(n) (se efectuează n înmulțiri)
- **Spațiu:** Θ(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 iterație
  - 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 neschimbat 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 inteligent care detectează situația în care apelul recursiv este „la coadă” (nu mai participă la calcule ulterioare).

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

rezultatul tuturor celor 4 apeluri este
← rezultatul final al funcției, anume 6

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

(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

(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

(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

(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

(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

(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
1
(fib 0)
1
1
0
1
0
1
0

(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
1
(fib 0)
1
1
0
1
0
1
0

(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
1
0
1
0
1
0

(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
1
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0
1
0

(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
1
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0

(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
1
1
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0

(fib 5)
(fib 4)
(fib 3)
2
(fib 2)
(fib 2)
(fib 1)
1
1
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0

(fib 5)
(fib 4)
(fib 3)
2
(fib 2)
(fib 2)
(fib 1)
1
1
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0

(fib 5)
(fib 4)
(fib 3)
2
(fib 2)
(fib 2)
(fib 1)
1
1
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0

(fib 5)
(fib 4)
(fib 3)
2
(fib 2)
(fib 2)
(fib 1)
1
1
1
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0
1
0

(fib 5)
(fib 4)
(fib 3)
2
(fib 2)
(fib 2)
(fib 1)
1
1
1
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0

(fib 5)
(fib 4)
(fib 3)
2
(fib 2)
(fib 2)
(fib 1)
1
1
1
0
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0

(fib 5)
(fib 4)
(fib 3)
2
1
(fib 2)
(fib 1)
1
1
1
0
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0

(fib 5)
3
(fib 3)
2
1
(fib 2)
(fib 1)
1
1
1
0
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0
1
0
.....etc.

(fib 5)

(fib 4)

(fib 3)

(fib 3)

(fib 2)

(fib 1)

(fib 1) (fib 0) 1

(fib 2)

(fib 1) (fib 0)

1 0

(fib 2)

(fib 1) (fib 0)

1 0

(fib 1)

1 .

1 0

← multitudine de calcule duplicate

Observații – recursivitate arborescentă

- **Timp:** Θ(fib(n+1)) (arborele are 2*fib(n+1)-1 noduri)
- **Spațiu:** Θ(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 (fibo-tail n)
2.    (fibo-tail-helper n 0 1))
3.
4.  (define (fibo-tail-helper n a b)
5.    (if (zero? n)
6.        a
7.        (fibo-tail-helper (- n 1) b (+ a b))))
```

timp = Θ(n)

```
>(fibo-tail-helper 3 0 1)
>(fibo-tail-helper 2 1 1)
>(fibo-tail-helper 1 1 2)
>(fibo-tail-helper 0 2 3)
>2
```

← fib(0), fib(1)
← fib(1), fib(2)
← fib(2), fib(3)
← fib(3), fib(4)

spațiu = Θ(1)

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 aceleași 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 retur
  - singurul apel recursiv e în poziție finală (valoarea sa este valoarea de retur) → 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

Exemple

Ce tip de recursivitate 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)))))
```

Exemple

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

```
(define (f x)
  (cond ((zero? x)    0)
        ((even? x)    (f (/ x 2)))
        (else         (+ 1 (f (- x 1))))))

(define (g L result)
  (cond ((null? L)          result)
        ((list? (car L))    (g (cdr L) (append (g (car L) '()) result)))
        (else               (g (cdr L) (cons (car L) result)))))
```

pe stivă

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

arborescentă

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 retur nu mai este valoarea pe cazul de bază, ci acumulatorul
- Valoarea funcției pe cazul de bază corespunde adesea valorii inițiale a acumulatorului

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 acumulatorul este o listă

Soluții pentru conservarea ordinii

- Inversarea acumulatorului înainte de retur (pe cazul de bază)

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

- Adăugarea fiecărui nou element la sfârșitul acumulatorului (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: $\Theta(n^2)$ (Θ(length(acc)) pentru fiecare append în parte)
  $(0 + 1 + 2 + \ldots + (n-1))$

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

```
(define (reverse L) (rev L '()))
(define (rev L acc)
  (if (null? L)
      acc
      (rev (cdr L) (cons (car L) acc))))
```

→ Θ(length(L))

```
(define (append A B)
  (if (null? A)
      B
      (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

- Apariţiile 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 ≡	**x**	variabilă
|	**λx.e1**	funcție (unară, anonimă) cu parametrul formal x și corpul e
|	**(e1 e2)**	aplicație a expresiei e1 asupra parametrului efectiv e2

Semantica (Modelul substituției)

Pentru a evalua **(λx.e1 e2)** (funcția cu parametrul formal x și corpul e1, aplicată pe e2):

- Peste tot în e1, identificatorul x este înlocuit cu e2
- Se evaluează noul corp e1 și se întoarce rezultatul (se notează $\mathbf{e1}_{[e2/x]}$)

Apariţiile unei variabile într-o λ-expresie

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

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

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

Vom distinge între:

- variabilele al căror nume nu contează (şi ar putea fi oricare altul)
  şi
- 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 = …λx_n.e…**
- **E = …λx.e… 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 (apariția care leagă) — legată (apariții în corpul funcției de parametru x) — legată

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**.

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

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

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

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

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

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

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 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)))

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

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

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

- Apariţiile 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: (x λx.x) din punct de vedere al statutului aparițiilor devine

(x λx.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**.

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

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

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

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

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

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

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**.

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

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

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

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

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

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

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

λx.(y λy.(x (y z)))

(x λx.(λx.y λy.(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)))

Exemple de variabile legate / libere

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

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

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

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

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

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

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

Calcul Lambda – Cuprins

- Apariţiile 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 **(λx.e1 e2)** (funcția cu parametrul formal x și corpul e1, aplicată pe e2):

- ~~Peste tot în e1, identificatorul x este înlocuit cu e2~~
- Peste tot în e1, aparițiile libere ale lui x (libere în e1 !) sunt înlocuite cu e2 (nu are rost să înlocuiesc și aparițiile legate, întrucât numele acestora este irelevant)
- Se evaluează noul corp e1 și se întoarce rezultatul (se notează $\mathbf{e1}_{[e2/x]}$)

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: $(\lambda x.\underbrace{\lambda y.(x\ \ y)}_{e1}\ \underbrace{y}_{e2}) \rightarrow_\beta \lambda y.(x\ \ y)_{[y/x]} = \lambda y.(y\ \ y)$ (greșit!)

Trebuia să obținem

o funcție care îl aplică pe y asupra argumentului său

≠

Am obținut

o funcție care își aplică argumentul asupra lui însuși

Ce a mers rău?

În urma înlocuirii, apariția lui y (care era liberă în e2) s-a trezit legată în e1 (la un argument cu care nu avea nicio legătură).

Generalizare

Conflict de nume între variabilele legate din e1 și variabilele libere din e2 → greșeli în evaluare

Soluția: α-conversia

α-conversie = redenumirea variabilelor legate din corpul unei funcții λx.e1 a.î. ele să nu coincidă cu variabilele libere din parametrul efectiv e2 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 e1 și din e2

Exemplu: $(\lambda x.\underbrace{\lambda y.(x\ y)}_{e1}\ \underbrace{y}_{e2}) \rightarrow_\alpha (\lambda x.\lambda z.(x\ z)\ y) \rightarrow_\beta \lambda z.(x\ z)_{[y/x]} = \lambda z.(y\ z)$ (corect!)

Exemplu de secvență de reducere

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

Exemplu de secvență de reducere

$(\lambda x.(x\ z)\ \ \lambda z.\lambda x.(z\ x)) \rightarrow_\beta$

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

Exemplu de secvență de reducere

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

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

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

Exemplu de secvență de reducere

$(\lambda x.(x\ z)\ \ \lambda z.\lambda x.(z\ x)) \rightarrow_\beta$

$(\lambda z.\lambda x.(z\ x)\ z) \rightarrow_\alpha$

$(\lambda t.\lambda x.(t\ x)\ z) \rightarrow_\beta$

$\lambda x.(z\ x)$

Calcul Lambda – Cuprins

- Apariţiile 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** ⇔ λ-expresie care nu conține niciun β-redex

Întrebări

- Are orice λ-expresie o formă normală?
- Forma normală este unică? (sau secvențe 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: $(\lambda x.(x\ x)\ \lambda x.(x\ x)) \rightarrow_\beta (\lambda x.(x\ x)\ \lambda x.(x\ x)) \rightarrow_\beta (\lambda x.(x\ x)\ \lambda x.(x\ x)) \rightarrow_\beta \ldots$

λ-expresie **reductibilă** ⇔ 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 $\begin{cases} e \to^* a \\ \text{și} \\ e \to^* b \end{cases}$ atunci $\exists$ d a.î. $\begin{cases} a \to^* d \\ \text{și} \\ b \to^* d \end{cases}$ ($\to^*$ = secvență de reducere)

Explicație: Dacă a și b ar fi forme normale distincte, prin definiție a și b nemaiconținând 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: $E_1 = (\lambda x.(x\ x)\ \lambda x.(x\ x))$

$E_2 = (\lambda x.y\ E_1) \rightarrow_\beta y$ ⟵ 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 recursivitate

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 recursivitate

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 recursivitate: 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 recursivitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)

Apariții ale unei variabile într-o expresie: legate (λx.e, λx. …x…), 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 recursivitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)

Apariții ale unei variabile într-o expresie: legate (λx.e, λx. …x…), 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 recursivitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)

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

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

β-redex și β-reducere: $(\lambda x.e1\ e2)$, $(\lambda x.e1\ e2) \rightarrow_\beta e1_{[e2/x]}$

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 recursivitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)

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

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

β-redex și β-reducere: $(\lambda x.e1\ e2)$, $(\lambda x.e1\ e2) \rightarrow_\beta e1_{[e2/x]}$

α-conversie: $(\lambda x. \ldots \underline{\lambda y.e1} \ldots \quad e2) \rightarrow_\alpha (\lambda x. \ldots \underline{\lambda t.e1_{[t/y]}} \ldots \quad e2)$ unde t nu era liberă în e1, e2

Forma normală

Expresie ireductibilă

Teorema normalizării

Rezumat

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

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

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

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

β-redex și β-reducere: $(\lambda x.e1\ e2)$, $(\lambda x.e1\ e2) \rightarrow_\beta e1_{[e2/x]}$

α-conversie: $(\lambda x. \ldots \underline{\lambda y.e1} \ldots \quad e2) \rightarrow_\alpha (\lambda x. \ldots \underline{\lambda t.e1_{[t/y]}} \ldots \quad e2)$ unde t nu era liberă în e1, e2

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 recursivitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)

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

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

β-redex și β-reducere: $(\lambda x.e1\ e2)$, $(\lambda x.e1\ e2) \rightarrow_\beta e1_{[e2/x]}$

α-conversie: $(\lambda x. \ldots \underline{\lambda y.e1} \ldots \quad e2) \rightarrow_\alpha (\lambda x. \ldots \underline{\lambda t.e1_{[t/y]}} \ldots \quad e2)$ unde t nu era liberă în e1, e2

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 recursivitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)

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

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

β-redex și β-reducere: $(\lambda x.e1\ e2)$, $(\lambda x.e1\ e2) \rightarrow_\beta e1_{[e2/x]}$

α-conversie: $(\lambda x. \ldots \underline{\lambda y.e1} \ldots \quad e2) \rightarrow_\alpha (\lambda x. \ldots \underline{\lambda z.e1_{[z/y]}} \ldots \quad e2)$ unde z nu era liberă în e1, e2

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 retur (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 întrucât propunea un model de mașină pe care să se execute algoritmii
- 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 însele (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 retur (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 retur (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 însele

```
1.  (define a +)
2.  (a 2 3 5)
3.
4.  null?
5.
6.  (define fs (list + (λ (x y) (- x y y)) 5))
7.  fs
8.
9.  ((cadr fs) 20 8)
```

← identificatorul a se leagă la valoarea funcției +

← funcția null? se evaluează la ea însăși

← al doilea element al listei fs este o funcție anonimă

Funcții ca valori evaluate la ele însele

```
(define a +)
(a 2 3 5)                                   ;; 10

null?                                       ;; #<procedure:null?>

(define fs (list + (λ (x y) (- x y y)) 5))
fs                                          ;; '(#<procedure:+> #<procedure> 5)

((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 retur

```
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 retur

```
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-curry x)
2.    (λ (y)
3.      (+ x y)))
4.
5.  (plus-curry 2)
6.  ((plus-curry 2) 10)
7.
8.  (define inc (plus-curry 1))
9.  (inc 10)
```

pe rând

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

(plus-uncurry 2 3)

(plus-uncurry 2)
```

deodată

Funcții curry / uncurry - Exemple

```
(define (plus-curry x)
  (λ (y)
    (+ x y)))

(plus-curry 2) ;; #<procedure>
((plus-curry 2) 10) ;; 12

(define inc (plus-curry 1))
(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 retur (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 șablon 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 șablonului 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 retur (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 combinările 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 retur (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 in termeni conceptuali, „dorind” ca aceste concepte să existe în limbaj
    - oferim implementări pentru fiecare abstracțiune procedurală folosită de funcția principală

Abstracțiuni 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 (apropie 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 șablon comun

Exemple la calculator:

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

Observații

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

Recunoaștere → Definire → Reutilizare

Recunoașterea șablonului

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

Definire

- Șablonului identificat la pasul anterior i se atribuie un nume sugestiv, care face programul ușor de înțeles

Reutilizare

- Cu cât șablonul se întâlnește mai frecvent, cu atât este mai important ca el să fie abstractizat (și multiplele sale instanțe să poată fi astfel derivate cu ușurință)

Test

Să se implementeze:

- Maximul elementelor dintr-o listă de numere, cu recursivitate pe coadă (există funcția `max` pentru maximul dintre 2 numere)
- Pătratul elementelor dintr-o listă de numere, cu recursivitate pe stivă (există funcția `sqr` pentru pătratul unui număr)

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 acumulator, rezultând un nou acumulator
- Apoi pe (second L) și noul acumulator … 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 acumulator, rezultând un nou acumulator
- Apoi pe penultimul și noul acumulator … 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))
(apply list -1 0 '(1 2 3))
(apply 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 (opțional există și alte argumente date unul câte unul înainte de cele care vor fi despachetate din L)

Observație

- Funcționalele 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: **selecție**)

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 retur (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

empty-bst? : 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 retur

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 retur

Funcții 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 retur

Funcții 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 retur

Funcții 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 retur

Funcții 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 retur

Funcții 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 retur

Funcții 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: (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 retur

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

Abstractizare la nivel de proces: abstracțiuni procedurale, abstractizare șabloane comune

Abstractizare la nivel de date

Rezumat

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

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

Abstractizare la nivel de proces: abstracțiuni procedurale, abstractizare șabloane comune

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

PARADIGME DE PROGRAMARE

Curs 4

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

Transparență referențială – Cuprins

- Efecte laterale
- Transparență 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

- Contează **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

- Contează **strategia de evaluare**

Comportament bizar al compilatorului care nu are operația definită

Afișează **0 1** → Contează 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

- Contează **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

- Contează **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.  int main() {
14.    cout << add(); cout << b;
```

Efecte laterale

Consecințe

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

- Risc ridicat de **bug-uri**

Transparență referențială – Cuprins

- Efecte laterale
- Transparență referențială

Transparență referențială

Transparență referențială

- Există atunci când toate funcțiile/expresiile 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)))
```

Transparență referențială

Transparență referențială

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

Exemple

- Toate funcțiile Racket implementate de noi până acum — transparente referențial
- `(random)` — opacă referențial
- opacă referențial

```
(define counter 1)
(define (display-and-inc-counter)
   (display counter)
   (set! counter (add1 counter)))
```

Transparența 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 = mulțimea punctelor din program unde asocierea identificator – valoare este vizibilă (și valoarea se poate accesa prin identificator)

Exemple în Calcul Lambda

(x λx.(λx.y λy.(x z)))

(x λx.(λx.y λy.(x z)))

(x λx.(λx.y λy.(x z)))

(x λx.(λx.y λy.(x z)))

Domeniu de vizibilitate al unei variabile

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

Exemple în Calcul Lambda

(x λx.(λx.y λy.(x z)))

(x λx.(λx.y λy.(x z)))

(x λx.(λx.y λy.(x z)))

(x λx.(λx.y λy.(x z)))

Domeniu de vizibilitate al unei variabile

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

Exemple în Calcul Lambda

(x λx.(λx.y λy.(x z)))

(x λx.(λx.y λy.(x z))) – acest x nu mai este vizibil nicăieri în această zonă de program

(x λx.(λx.y λy.(x z)))

(x λx.(λx.y λy.(x z)))

Domeniu de vizibilitate al unei variabile

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

Exemple în Calcul Lambda

(x λx.(λx.y λy.(x z)))

(x λx.(λx.y λy.(x z)))

(x λx.(λx.y λy.(x z))) – practic: zona din corp în care aparițiile lui x sunt libere

(x λx.(λx.y λy.(x z)))

Domeniu de vizibilitate al unei variabile

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

Exemple în Calcul Lambda

(x λx.(λx.y λy.(x z)))

(x λx.(λx.y λy.(x z)))

(x λx.(λx.y λy.(x z)))

(x λx.(λx.y λy.(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 **obscurează** 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)
```

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 evaluează 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 evaluează 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 evaluează **în ordine aleatoare** argumentele arg_1, arg_2, … arg_m (evaluare aplicativă)
- Se realizează legările $var_k \leftarrow valoare(arg_k)$
- Domeniul de vizibilitate al variabilei var_k este corpul lui lambda (exceptând aparițiile legate ale lui var_k în corp)
- Se evaluează în ordine expresiile din corpul funcției ($expr_1$, $expr_2$, … $expr_n$)
- Rezultatul este valoarea lui $expr_n$ (valorile lui $expr_1$, $expr_2$, … $expr_{n-1}$ se pierd)

Construcția `let`

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

La evaluare

- Se evaluează exact ca
  `((lambda (var1 var2 … varm) expr1 expr2 … exprn) e1 e2 … em)`
- Domeniul de vizibilitate al variabilei var_k este corpul lui let (exceptând aparițiile legate ale lui var_k în corp)
- Rezultatul este valoarea lui $expr_n$ (valorile lui $expr_1$, $expr_2$, … $expr_{n-1}$ 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 $var_k \leftarrow valoare(e_k)$
- 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 evaluează în ordine expresiile din corpul funcției ($expr_1$, $expr_2$, … $expr_n$)
- Rezultatul este valoarea lui $expr_n$ (valorile lui $expr_1$, $expr_2$, … $expr_{n-1}$ 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 $var_k \leftarrow valoare(e_k)$
- Domeniul de vizibilitate al variabilei var_k este **întregul letrec** (celelalte 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 evaluează în ordine expresiile din corpul funcției ($expr_1$, $expr_2$, … $expr_n$)
- Rezultatul este valoarea lui $expr_n$ (valorile lui $expr_1$, $expr_2$, … $expr_{n-1}$ 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 var_1, var_2, … var_m, și se și aplică funcția pe argumentele e_1, e_2, … 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

Test

Să se implementeze, folosind funcționale și/sau funcții de bibliotecă (dar nu recursivitate):

- O funcție care primește o listă de numere naturale strict pozitive și întoarce toate numerele x din listă pentru care x^x este mai mare decât o valoare dată val

```
(f '(1 4 3 5 1 2) 30)     ;; '(4 5)
```

Observație: Ridicarea la putere se va face folosind funcționale (nu o funcție de bibliotecă pentru ridicare la putere)!
Util: funcția range (ex: (range 5) produce lista '(0 1 2 3 4))

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 evaluează `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 **transparența 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 formali ai funcției)

Context computațional – Exemplu

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

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:

(a 1)

(f (λx.corp ▪))

text pointer la 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)

Ierarhia de contexte

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

Contextul global:

(a 1)
(f (λx.corp ▪))

Context local adăugat la apelul (f 2):

(x 2) ← dispare odată cu terminarea apelului

Context local adăugat de expresia let:

(x 5) ← dispare odată cu ieșirea din let

La evaluare

- Se caută variabila în contextul curent
- Dacă nu este găsită acolo, se caută în contextul părinte, ș.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 (λn.corp ▪))
(g (λn.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 (λn.corp ▪))
(g (λn.corp ▪))

~~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 (λn.n ▪)) contextul global suprascris de linia 8

(g (λn.corp ▪))

~~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 (λn.n ▪)) contextul global suprascris de linia 8
(g (λn.corp ▪))

~~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) -> ((λ (n) (if (zero? n) 1 (* n (fact (- n 1))))) 4)
      -> (* 4 (fact (- 4 1))) -> (* 4 ((λ (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

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

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 aceleași argumente întorc aceeași valoare; nu au efecte laterale

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 aceleași argumente întorc aceeași valoare; nu au efecte laterale

Transparență 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

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

Transparență 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

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

Transparență 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

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

Transparență 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

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

Transparență 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

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

Transparență 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: (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

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

Transparență 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: (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 nestricte** (if, and, or) utile în:
  - Controlul fluxului prin program
  - Condiții de oprire
    `(or (null? L) (zero? (car L)))` – se evaluează `(car L)` doar dacă L nu e vidă
- **Structuri de date infinite**, din care se evaluează (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)

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

Corpul 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

Implementare cu închideri funcționale

Avantaj

- Capabilă să reprezinte graful infinit

Dezavantaje

- **Ineficientă** din cauza evaluării repetate (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))))
4.      (force a)))
```

(λ () expr) se înlocuiește cu (delay expr)

(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 așa:

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
  valoare ← evaluează expresie
  flag ← #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 .. √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)
2.    (if (> a b)
3.        empty-stream
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)))))
```

interfață foarte asemănătoare celei pentru liste

- **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

Utilizare **liste** | Utilizare **fluxuri**

Constructori / selectori / operatori

Implementare liste (element . **pereche**) | Implementare fluxuri (element . **promisiune**)

Complexitate spațială

- Intervalul [1 .. n] reținut ca **listă: Θ(n)** (n elemente ținute în memorie)
- Intervalul [1 .. n] reținut ca **flux: Θ(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

- (prime? n) cu **liste: Θ(√n)** (generez √n numere, parcurg √n numere ca să le filtrez, aplic null?)
- (prime? n) cu **fluxuri: Θ(distanța până la primul divizor)** (Θ(√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
      ;; Acesta 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 evaluează restul (vezi stream-empty? anterior)
  - Dacă funcția încearcă să acceseze 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	Constructor corespunzător pe fluxuri
'()	empty-stream
cons	stream-cons

Operator pe liste	Operator corespunzător pe fluxuri
null?	stream-empty?
car	stream-first
cdr	stream-rest
map	stream-map
filter	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) '(0 1) (loop 2) '(0 1 2)
- `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
-------------------------------------
 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   ]
-------------------------------------
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   |
-------------------------------------
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 implicite

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 reevaluează porțiunile deja calculate, ci iau rezultatele din cache
  - Închiderile funcționale reevaluează 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 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 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 – Implementare

```
(define (sieve s)
  (let ((p (stream-first s)))
    (stream-cons p (sieve (stream-filter
                            (λ (n) (not (zero? (modulo n p))))
                            (stream-rest s))))))

(define primes
  (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: eficiență: 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: eficiență: 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: eficiență: 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: eficiență: 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), eficiență (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: eficiență: 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), eficiență (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: eficiență: 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), eficiență (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: eficiență: 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), eficiență (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: eficiență: 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), eficiență (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: transformări/operații de alte fluxuri (sau de fluxul însuși)

PARADIGME DE PROGRAMARE

Curs 6

Limbajul Haskell. Tipare tare / slabă / statică / dinamică. Tipuri și expresii de tip.

Programare funcțională în Haskell

Limbajul Haskell – Cuprins

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

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 **corpul** 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 leneșă

TDA-ul Pereche

Constructori de bază

(,) : T1 x T2 -> Pereche // creează o pereche între orice 2 argumente

Operatori

fst : Pereche -> T1 // extrage prima valoare din pereche

snd : Pereche -> T2 // extrage a doua valoare din pereche

Exemple

```
(1, "unu")
a = (('a', 2), "a2")
fst a
snd (fst a)
```

TDA-ul Pereche

Constructori de bază

(,) : T1 x T2 -> Pereche // creează o pereche între orice 2 argumente

Operatori

fst : Pereche -> T1 // extrage prima valoare din pereche

snd : Pereche -> T2 // 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)

[ ] : -> Listă // creează o listă vidă

: : T x Listă -> Listă // creează o listă prin adăugarea unei valori la începutul unei liste

[, , .. ,] : T x .. T -> Listă // creează o listă din toate argumentele sale (de același tip)

Operatori

head : Listă -> T

tail : Listă -> Listă

null : Listă -> Bool

length : Listă -> Nat

++ : 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)

[ ] : -> Listă // creează o listă vidă

: : T x Listă -> Listă // creează o listă prin adăugarea unei valori la începutul unei liste

[, , .. ,] : T x .. T -> Listă // creează o listă din toate argumentele sale (de același tip)

Operatori

head : Listă -> T

tail : Listă -> Listă

null : Listă -> Bool

length : Listă -> Nat

++ : Listă x Listă -> Listă

Exemple

```
head [[2,4],[6],[5]]        -- [2,4]
tail (2:3:[4,5])            -- [3,4,5]
null [[]]                   -- False
length [[]]                 -- 1
[1] ++ [1,2,3] ++ [4,5]     -- [1,1,2,3,4,5]
```

Limbajul Haskell – Cuprins

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

Funcții anonime în Haskell

```
\parametri -> corp
```

Exemple

λx.x	`\x -> x`	
λx.λy.(x y)	`\x -> \y -> x y`	echivalent cu
	`\x y -> x y`	(întrucât funcțiile Haskell sunt **automat curry**)
(λx.x λx.y)	`(\x -> x) (\x -> 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

$f\ x_1\ x_2\ \ldots\ x_n = corp$

generează funcții **curry**, care pot fi aplicate pe oricâți (≤n) parametri la un moment dat.

$f\ e_1\ e_2 \ldots\ e_k$ întoarce o nouă funcție $\backslash x_{k+1}\ \ldots\ x_n \to corp_{[ei/xi]}$.

Pentru a obține comportamentul de funcție **uncurry**, parametrul lui f trebuie să fie un tuplu:

$f\ (x_1,\ x_2\ \ldots,\ x_n) = corp$

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 leneșă

Definirea funcțiilor prin pattern matching

Se descrie comportamentul funcțiilor în funcție de structura parametrilor – ca la scrierea axiomelor TDA-ului.

Exemple

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)

sumL [] = 0
sumL (x:xs) = x + sumL xs

sumP (x,y) = x + y

ordered [] = True
ordered [x] = True
ordered (x:xs@(y:rest)) = x <= y && ordered xs
```

permite crearea unui alias pentru valoarea următoare

La aplicare

```
1.  fib 0 = 0
2.  fib 1 = 1
3.  fib n = fib (n-2) + fib (n-1)

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, ș.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
2.  ordered [x] = True
3.  ordered (x:xs@(y:rest)) = x <= y && ordered xs
```

De aceea a trebuit să specificăm și ce se întâmplă pe lista vidă, și ce se întâmplă pe lista cu un singur element

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 leneșă

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 … 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 … 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 … 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 leneșă

Condiționala `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
      ...
      pattern_n -> rezultat_n
```

Are sens atunci când nu putem folosi pattern matching sau gărzi, de exemplu aici când verificăm structura lui head matrix, fără să o putem verifica pe a lui matrix în loc

Exemplu

```
myTranspose matrix = case (head matrix) of
      [] -> []
      _ -> map head matrix : myTranspose (map tail matrix)
```

Gărzi

```
f parametri
  | condiție₁ = rezultat₁
  | condiție₂ = rezultat₂
  ...
  | condiție_n = rezultat_n
 [| otherwise = rezultat_n]
```

← opțional

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 leneșă

Legarea variabilelor în Haskell – statică

- Doar legare **statică**
- Expresii pentru legare locală: `let legări in expr`, `expr where legări`

Au sens pentru a spori lizibilitatea codului sau pentru a evita apelarea repetată a aceleiași funcții pe aceleași argumente

Exemple

```
myFoldl f acc [] = acc
myFoldl f acc (x:xs) =
        let
                newAcc = f acc x
        in myFoldl f newAcc xs

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 leneșă

Evaluare leneșă

- Toate funcțiile sunt **nestricte**
- **Evaluare leneșă**: **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 => se evaluează de 2 ori
                    -- f 2 + f 2 -> 4 + f 2 -> 4 + 4 -> 8
h (f 2)             -- argumentul se evaluează o dată și se folosește de 3 ori
                    -- (f 2)*(f 2)*(f 2) -> 4*4*4 -> 64
```

Fluxuri

- Evaluare leneșă => toate **listele sunt fluxuri** (se evaluează 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
```

Test

Definiți următoarele fluxuri în Racket:

- `1, 1/2, 1/3, 1/4 ...` – folosind o definiție explicită
- `1, 1/2, 1/6, 1/12, 1/20 ...` – folosind o definiție implicită

(obs: `2=1*2, 6=2*3, 12=3*4` ...)

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 ]
```

Întâi toate rezultatele pentru x=1, apoi toate pentru x=2, apoi toate pentru x=3

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 ]
```

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Tipuri definite de utilizator
- Tipuri parametrizate
- Expresii de tip

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
- Tipuri definite de utilizator
- Tipuri parametrizate
- Expresii de tip

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
- Tipuri definite de utilizator
- Tipuri parametrizate
- Expresii de tip

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

- **(, , ...)** : $MT^n \longrightarrow MT$ (MT = mulțimea tipurilor)
  - $(t_1, t_2, \ldots, t_n)$ = **tuplu** cu elemente de tipurile $t_1, t_2, \ldots, t_n$
  - **Ex:** (Bool, Char) echivalent cu (,) Bool Char
- **[ ]** : $MT \longrightarrow MT$
  - [t] = **listă** cu elemente de tip t
  - **Ex:** [Int] echivalent cu [ ] Int
- **->** : $MT^2 \longrightarrow MT$
  - t_1 -> t_2 = **funcție** de un parametru de tip t_1 care calculează valori de tip t_2
  - **Ex:** Int -> Int echivalent cu (->) 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: `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)`?

```
(add 2) :: Int -> Int
```

- În aceste condiții, care este tipul lui `add`?

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)`?

```
(add 2) :: Int -> Int
```

- În aceste condiții, care este tipul lui `add`?

```
add :: Int -> (Int -> Int)
```

echivalent cu

```
add :: Int -> Int -> Int
```

întrucât **-> este asociativ la dreapta**

- Cum am interpreta tipul `(Int -> 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)`?

```
(add 2) :: Int -> Int
```

- În aceste condiții, care este tipul lui `add`?

```
add :: Int -> (Int -> Int)
```

echivalent cu

```
add :: Int -> Int -> Int
```

întrucât **-> este asociativ la dreapta**

- Cum am interpreta tipul `(Int -> Int) -> 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
- Tipuri definite de utilizator
- Tipuri parametrizate
- Expresii de tip

Tipuri definite de utilizator

- Cuvântul cheie **data** dă utilizatorului posibilitatea definirii unui TDA cu implementare completă (constructori, operatori, axiome)

```
data NumeTip = 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

- Compun noi valori pe baza celor existente (comportament de **funcție**)

  Exemple:
  ```
  unu = Succ Zero
  doi = Succ unu
  ```

- Descompun valori existente în scopul identificării structurii lor (comportament de **pattern**)

  Exemple:
  ```
  addN Zero n = n
  addN (Succ m) n = Succ (addN m n)
  ```

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Tipuri definite de utilizator
- Tipuri parametrizate
- Expresii de tip

Tipuri parametrizate

- **Constructorii de valori** ale tipului (ex: **:**, **Succ**)
  - Pot primi valori ca argumente pentru a produce noi valori

    Exemple:
    ```
    unu = Succ Zero
    lista_unu = 1 : []
    ```

- **Constructorii de tip** (ex: **[ ]**, **(,)**, **->**)
  - Pot primi tipuri ca argumente pentru a produce noi tipuri

    Exemple:
    ```
    [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

    Exemple:
    `[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

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 gasească 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)
6.              _ -> Just lsum
7.      | otherwise = findMaxEvenSum ls
8.          where lsum = sumL l
```

Din cauză că sumL este declarat ca
sumL :: Int -> Int
funcția va întoarce un (Maybe Int)

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Tipuri definite de utilizator
- Tipuri parametrizate
- Expresii de tip

Expresii de tip

- Expresiile reprezintă valori / expresiile de tip reprezintă tipuri

  Exemple: `Char, Int -> Int -> Int, (Char, [Int])`

- **Declararea signaturii** unei funcții (opțională în Haskell) **f :: exprDeTip**
  = asociere între numele funcției și o expresie de tip, cu rol de:
  - **Documentare** (ce ar trebui sa facă funcția)
  - **Abstractizare** (surprinde cel mai general comportament al funcției, funcția percepută 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 a la un tip b
o listă de elemente de tip a

întoarcă: o listă de elemente de tip 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)

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

Perechi și liste
Funcții
. și $
Expresii condiționale
Expresii pentru legare locală
Evaluare leneșă
List comprehensions
Tipare tare/slabă
Tipare statică/dinamică
Constructori de tip
Tipuri definite de utilizator
Tipuri parametrizate
Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții

. și $

Expresii condiționale

Expresii pentru legare locală

Evaluare leneșă

List comprehensions

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $

Expresii condiționale

Expresii pentru legare locală

Evaluare leneșă

List comprehensions

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale

Expresii pentru legare locală

Evaluare leneșă

List comprehensions

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală

Evaluare leneșă

List comprehensions

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă

List comprehensions

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]

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”

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]

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"

Tipuri definite de utilizator: data NumeTip = $Cons_1\ t_{11}\ ..\ t_{1i}$ | ... | $Cons_n\ t_{n1}\ ..\ t_{nk}$

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]

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"

Tipuri definite de utilizator: data NumeTip = $Cons_1\ t_{11}\ ..\ t_{1i}$ | ... | $Cons_n\ t_{n1}\ ..\ t_{nk}$

Tipuri parametrizate: (a,b), [a], a -> b, data ConsTip a b ...

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]

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"

Tipuri definite de utilizator: data NumeTip = $Cons_1\ t_{11}\ ..\ t_{1i}$ | ... | $Cons_n\ t_{n1}\ ..\ t_{nk}$

Tipuri parametrizate: (a,b), [a], a -> b, data ConsTip a b ...

Declararea signaturii: f :: exprTip

PARADIGME DE PROGRAMARE

Curs 7

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 + 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 t are tipul T”
- O expresie e este **bine tipată** ⇔ există un tip T astfel încât e :: T

Regulă de sinteză de tip

$$\frac{P_1, P_2, \ldots P_n}{C} \text{ (id_regulă)}$$

P_i = premise (0 sau mai multe relații de tipare)

C = concluzie (o relație de tipare)

Reguli de sinteză de tip – Exemple

$$\frac{}{\text{True, False} :: \text{Bool}} \quad (TBool)$$

$$\frac{expr1 :: \text{Bool} \quad expr2 :: \text{a} \quad expr3 :: \text{a}}{\text{if } expr1 \text{ then } expr2 \text{ else } expr3 :: \text{a}} \quad (TIf)$$

$$\frac{var :: \text{a} \quad expr :: \text{b}}{\backslash var \rightarrow expr :: \text{a} \rightarrow \text{b}} \quad (TAbs)$$

$$\frac{fun :: \text{a} \rightarrow \text{b} \quad expr :: \text{a}}{fun\ expr :: \text{b}} \quad (TApp)$$

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

$$\frac{expr1 :: \text{Int} \quad expr2 :: \text{Int}}{expr1 + expr2 :: \text{Int}} \quad (T+)$$

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 expr1 :: Int și expr2 :: Int, atunci expr1 + expr2 :: Int
  - De la concluzie către premise
    Exemplu: Dacă expr1 + expr2 :: Int, atunci expr1 :: Int și expr2 :: 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: **f g = g 3 + 1** ⇔ **f = \g -> g 3 + 1**

Constrângeri

$$\frac{var :: \mathrm{a} \quad expr :: \mathrm{b}}{\backslash var \rightarrow expr :: \mathrm{a} \rightarrow \mathrm{b}} \; (TAbs)$$

$$\frac{g :: \mathrm{a} \quad g\ 3 + 1 :: \mathrm{b}}{f :: \mathrm{a} \rightarrow \mathrm{b}} \; (TAbs)$$

$$\frac{expr1 :: \mathrm{Int} \quad expr2 :: \mathrm{Int}}{expr1 + expr2 :: \mathrm{Int}} \; (T+)$$

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

b = Int

$$\frac{fun :: \mathrm{a} \rightarrow \mathrm{b} \quad expr :: \mathrm{a}}{fun\ expr :: \mathrm{b}} \; (TApp)$$

$$\frac{g :: \mathrm{c} \rightarrow \mathrm{Int} \quad 3 :: \mathrm{c}}{g\ 3 :: \mathrm{Int}} \; (TApp)$$

a = c ->Int

$$\frac{}{0, 1, 2\ \ldots :: \mathrm{Int}} \; (TInt)$$

$$\frac{}{1, 3\ :: \mathrm{Int}} \; (TInt)$$

c = Int

Așadar **f :: a -> b** ⇔ **f :: (c -> Int) -> Int** ⇔ **f :: (Int -> Int) -> Int**

Sinteză de tip – Exemple

Care este tipul funcției fix: **fix f = f (fix f)** ⇔ **fix = \f -> f (fix f)**

Constrângeri

$$\frac{var :: \mathrm{a} \quad expr :: \mathrm{b}}{\backslash var \rightarrow expr :: \mathrm{a} \rightarrow \mathrm{b}} \; (TAbs)$$

$$\frac{f :: \mathrm{a} \quad f\ (fix\ f) :: \mathrm{b}}{fix :: \mathrm{a} \rightarrow \mathrm{b}} \; (TAbs)$$

$$\frac{fun :: \mathrm{a} \rightarrow \mathrm{b} \quad expr :: \mathrm{a}}{fun\ expr :: \mathrm{b}} \; (TApp)$$

$$\frac{f :: \mathrm{c} \rightarrow \mathrm{b} \quad (fix\ f) :: \mathrm{c}}{f\ (fix\ f) :: \mathrm{b}} \; (TApp)$$

a = c -> b

$$\frac{fun :: \mathrm{a} \rightarrow \mathrm{b} \quad expr :: \mathrm{a}}{fun\ expr :: \mathrm{b}} \; (TApp)$$

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

a = d, b = c

Așadar **fix :: a -> b** ⇔ **fix :: (c -> b) -> b** ⇔ **fix :: (b -> b) -> 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 ("01" :) . 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 -> a -> a) -> [a] -> [a] -> [a]
unzip (ex: unzip [(1,2), (3,4)] = ([1,3], [2,4]))	[(a,b)] -> ([a], [b])

Nu, nu e necesar ca cele două liste să aibă elemente de același tip.

Ex: zipWith (:) [1,2] [[3],[4]]

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

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

unzip (ex: unzip [(1,2), (3,4)] = ([1,3], [2,4]))	[(a,b)] -> ([a], [b])
[take, drop]	[Int -> [a] -> [a]]

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 signatura 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
(odd . fst)	(Int, b) -> Bool

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 ("01" :) . 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.

Test

Scrieți un list comprehension în Haskell care reprezintă mulțimea (așadar fără duplicate) tuturor tripletelor pitagoreice de numere naturale de maxim 3 cifre.

Obs: Numerele a, b, c formează un triplet pitagoreic dacă pot fi laturile unui triunghi dreptunghic ($a^2 + b^2 = c^2$).

Ex: 3, 4, 5 formează un triplet pitagoreic întrucât 9 + 16 = 25

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 -> [d] și a = Int -> c
  → b -> [d] trebuie să unifice cu Int -> c, altfel sinteza de tip va eșua
- Există S = {Int / b, [d] / c} a.î. b -> [d] și Int -> c se intersectează în Int -> [d]
- Există și S' = {Int / b, [Int] / c, Int / d}, rezultând în Int -> [Int], dar S este **mai generală**!

Mai multe exemple

- e1 = a, e2 = Int -> Bool

- e1 = (a, [a]), e2 = ([b], c)

- e1 = Int, e2 = Float

Mai multe exemple

- e1 = a, e2 = Int -> Bool

S = {Int -> Bool / a}

$e_{unificată}$ = Int -> Bool

- e1 = (a, [a]), e2 = ([b], c)

- e1 = Int, e2 = Float

Mai multe exemple

- e1 = a, e2 = Int -> Bool S = {Int -> Bool / a} $e_{unificată}$ = Int -> Bool

- e1 = (a, [a]), e2 = ([b], c) S = {[b] / a, [[b]] / c} $e_{unificată}$ = ([b], [[b]])

- e1 = Int, e2 = Float

Mai multe exemple

- e1 = a, e2 = Int -> Bool S = {Int -> Bool / a} $e_{unificată}$ = Int -> Bool

- e1 = (a, [a]), e2 = ([b], c) S = {[b] / a, [[b]] / c} $e_{unificată}$ = ([b], [[b]])

- e1 = Int, e2 = Float Nu unifică (nu se fac conversii implicite)!

Unificare la sinteza de tip – Exemplu

Care este tipul funcției f: **f x = x x** ⇔ **f = \x -> x x**

Constrângeri

$$\frac{var :: \mathrm{a} \quad expr :: \mathrm{b}}{\backslash var \rightarrow expr :: \mathrm{a} \rightarrow \mathrm{b}} \; (TAbs)$$

$$\frac{x :: \mathrm{a} \quad x\ x :: \mathrm{b}}{f :: \mathrm{a} \rightarrow \mathrm{b}} \; (TAbs)$$

$$\frac{fun :: \mathrm{a} \rightarrow \mathrm{b} \quad expr :: \mathrm{a}}{fun\ expr :: \mathrm{b}} \; (TApp)$$

$$\frac{x :: \mathrm{c} \rightarrow \mathrm{b} \quad x :: \mathrm{c}}{x\ x :: \mathrm{b}} \; (TApp)$$

a = c -> b = c

a unifică cu c -> b, dar c -> b nu unifică cu c.
(c = c -> b = c -> b -> b = c -> b -> b -> b … 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: t1 ~ t1 -> 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 ($S = S_1 \cup \ldots \cup S_n$) dacă și numai dacă sunt aplicații ale aceluiași constructor de tip asupra câte n argumente care unifică recursiv (prin substituțiile $S_1 \ldots S_n$)

Unificare la sinteza de tip – Implementare

- Algoritmul de sinteză de tip pe bază de unificare folosește
  - O **stivă de constrângeri** de forma **`e1 = e2`** – se umple în timpul algoritmului de sinteză de tip
  - O **substituție S** (mulțime de asocieri între variabile de tip și expresii de tip) – inițial vidă
- La fiecare iterație a pasului de unificare
  - Se extrage o constrângere `e1 = e2` din stivă
  - Dacă `e1` unifică cu `e2` prin substituția S_{12}, se aplică S_{12}
    - asupra stivei și
    - asupra substituției S
  - Altfel, unificarea eșuează ⟶ sinteza de tip eșuează ⟶ eroare de tip

Exemplu

Stiva	Substituția

`a = b -> c`
`a = d -> d`
`b = Int`	

`b -> c = d -> d` `b = Int`	`a = b -> c`

Exemplu

Stiva	Substituția

`a = b -> c`
`a = d -> d`
`b = Int`	

`b -> c = d -> d` `b = Int`	`a = b -> c`

`b = d`
`c = d`
`b = Int`
	`a = b -> c`

Exemplu

Stiva	Substituția
a = b -> c a = d -> d b = Int	
b -> c = d -> d b = Int	a = b -> c
b = d c = d b = Int	a = b -> c
c = d d = Int	a = d -> c b = d

Exemplu

Stiva	Substituția
a = b -> c a = d -> d b = Int	
b -> c = d -> d b = Int	a = b -> c
b = d c = d b = Int	a = b -> c
c = d d = Int	a = d -> c b = d
d = Int	a = d -> d b = d c = d

Exemplu

Stiva	Substituția
a = b -> c a = d -> d b = Int	
b -> c = d -> d b = Int	a = b -> c
b = d c = d b = Int	a = b -> c
c = d d = Int	a = d -> c b = d
d = Int	a = d -> d b = d c = d
	a = Int -> Int b = Int c = Int d = Int

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") **Constrângeri**

(T+) ⟶ expr :: Int ⇔ length ([]++[1,2]) :: Int, length ([]++"abcd") :: Int

(TLen) ⟶ []++[1,2] :: [a], []++"abcd" :: [b]

(T++) ⟶ [] :: [a], [1,2] :: [a], [] :: [b], "abcd" :: [b]

(T[]) ⟶ 1,2 :: a, 'a','b','c','d' :: b

(TInt, TChar) ⟶ 1,2 :: Int, 'a','b','c','d' :: Char a = Int, b = Char

[] :: [Int] și [] :: [Char] ok ([] e polimorfic – cele 2 liste vide nu reprezintă aceeași valoare)

Etapa de unificare reușește și ea (nu avem două constrângeri asupra aceleiași 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+) ⟶ expr :: Int ⇔ length (xs++[1,2]) :: Int, length (xs++"abcd") :: 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+) ⟶ expr :: Int ⇔ 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]

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]

(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 ⇔ 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 ⇔ 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

Constrângeri

(TApp) ⟶ expr :: b ⟺ curry :: a -> b, id :: a

Determinarea tipului expresiilor / funcțiilor

curry id

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, id :: a

$$\frac{}{\text{curry} :: \big((c, d) \to e\big) \to c \to d \to e} \quad (TCurry)$$

Constrângeri

a = (c,d) -> e, b = c -> d -> e

Determinarea tipului expresiilor / funcțiilor

curry id

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, id :: a

$$\frac{}{\text{curry} :: ((c, d) \to e) \to c \to d \to e} \quad (TCurry)$$

$$\frac{}{\text{id} :: f \to f} \quad (TId)$$

Constrângeri

a = (c,d) -> e, b = c -> d -> e

a = f -> f

Determinarea tipului expresiilor / funcțiilor

curry id

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, id :: a

$$\frac{}{\text{curry} :: ((c, d) \to e) \to c \to d \to e} \quad (TCurry)$$

$$\frac{}{\text{id} :: f \to f} \quad (TId)$$

Constrângeri

a = (c,d) -> e, b = c -> d -> e

a = f -> f

În etapa de unificare: a = (c,d) -> e = f -> f ⟶ e = f = (c,d) ⟶ b = c -> d -> (c,d), deci expresia are tipul c -> d -> (c,d) conform TApp.

Determinarea tipului expresiilor / funcțiilor

uncurry id

Constrângeri

Determinarea tipului expresiilor / funcțiilor

uncurry id

Constrângeri

(TApp) ⟶ expr :: b ⇔ uncurry :: a -> b, id :: a

Determinarea tipului expresiilor / funcțiilor

uncurry id

(TApp) ⟶ expr :: b ⇔ uncurry :: a -> b, id :: a

$$\frac{}{\text{uncurry} :: (c \to d \to e) \to (c, d) \to e} \quad (TUncurry)$$

Constrângeri

a = c -> d -> e, b = (c,d) -> e

Determinarea tipului expresiilor / funcțiilor

uncurry id

(TApp) ⟶ expr :: b ⇔ uncurry :: a -> b, id :: a

$$\frac{}{\text{uncurry} :: (\text{c} \to \text{d} \to \text{e}) \to (\text{c}, \text{d}) \to \text{e}} \quad (TUncurry)$$

$$\frac{}{\text{id} :: \text{f} \to \text{f}} \quad (TId)$$

Constrângeri

a = c -> d -> e, b = (c,d) -> e

a = f -> f

Determinarea tipului expresiilor / funcțiilor

uncurry id

(TApp) ⟶ expr :: b ⇔ uncurry :: a -> b, id :: a

$$\frac{}{\text{uncurry} :: (c \to d \to e) \to (c, d) \to e} \quad (TUncurry)$$

$$\frac{}{\text{id} :: f \to f} \quad (TId)$$

Constrângeri

a = c -> d -> e, b = (c,d) -> e

a = f -> f

În etapa de unificare: a = c -> d -> e = f -> f ⟶ c = f = d -> e ⟶ b = (d -> e, d) -> e, deci expresia are tipul (d -> e, d) -> e conform TApp.

Determinarea tipului expresiilor / funcțiilor

curry (curry id)

Constrângeri

Determinarea tipului expresiilor / funcțiilor

curry (curry id)

Constrângeri

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, curry id :: a

Determinarea tipului expresiilor / funcțiilor

curry (curry id)

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, curry id :: a

(ExAnterior) ⟶ curry id :: f -> g -> (f,g)

Constrângeri

a = f -> g -> (f,g)

Determinarea tipului expresiilor / funcțiilor

curry (curry id)

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, curry id :: a

(ExAnterior) ⟶ curry id :: f -> g -> (f,g)

$$\frac{}{\text{curry} :: \big((c,d) \to e\big) \to c \to d \to e} \quad (TCurry)$$

Constrângeri

a = f -> g -> (f,g)

a = (c,d) -> e, b = c -> d -> e

Determinarea tipului expresiilor / funcțiilor

curry (curry id)

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, curry id :: a

(ExAnterior) ⟶ curry id :: f -> g -> (f,g)

$$\frac{}{\text{curry} :: \big((c,d) \to e\big) \to c \to d \to e} \quad (TCurry)$$

Constrângeri

a = f -> g -> (f,g)

a = (c,d) -> e, b = c -> d -> e

În etapa de unificare: a = f -> g -> (f,g) = (c,d) -> e ⟶ f = (c,d), e = g -> (f,g) = g -> ((c,d),g) ⟶ b = c -> d -> g -> ((c,d), g), deci expresia are tipul c -> d -> g -> ((c,d), g) conform TApp.

Determinarea tipului expresiilor / funcțiilor

curry uncurry

Constrângeri

Determinarea tipului expresiilor / funcțiilor

curry uncurry

Constrângeri

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, uncurry :: a

Determinarea tipului expresiilor / funcțiilor

curry uncurry

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, uncurry :: a

$$\frac{}{\text{uncurry} :: (c \to d \to e) \to (c, d) \to e} \quad (TUncurry)$$

Constrângeri

a = (c -> d -> e) -> (c,d) -> e

Determinarea tipului expresiilor / funcțiilor

curry uncurry

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, uncurry :: a

$$\frac{}{\text{uncurry} :: (c \to d \to e) \to (c, d) \to e} \quad (TUncurry)$$

$$\frac{}{\text{curry} :: \big((f, g) \to h\big) \to f \to g \to h} \quad (TCurry)$$

Constrângeri

a = (c -> d -> e) -> (c,d) -> e

a = (f,g) -> h, b = f -> g -> h

Determinarea tipului expresiilor / funcțiilor

curry uncurry

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, uncurry :: a

$$\frac{}{\text{uncurry} :: (c \to d \to e) \to (c, d) \to e} \quad (TUncurry)$$

$$\frac{}{\text{curry} :: \big((f, g) \to h\big) \to f \to g \to h} \quad (TCurry)$$

Constrângeri

a = (c -> d -> e) -> (c,d) -> e

a = (f,g) -> h, b = f -> g -> h

În etapa de unificare: a = (c -> d -> e) -> (c,d) -> e = (f,g) -> h ⟶ (printre altele) c -> d -> e = (f, g) care nu unifică (sunt aplicații ale unor constructori de tip diferiți), 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 `->'
Înlocuiește cu ::

```
f x = x ++ [1; 2; 3]
```

curs7.hs:1:14: parse error on input `;'
Înlocuiește cu ,

```
f x =
      | null x = 0
      | otherwise = 1
```

curs7.hs:2:9: parse error on input `|'
Șterge = 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 ace(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
      | oterwise = f (a+1) (tail x)
```

Not in scope: `oterwise'
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 (<)

...
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 așteaptă funcția**
  - posibil să fi greșit ordinea argumentelor

```
f :: String -> String
f = head
```

Couldn't match type `Char' with `[Char]'

```
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 gărzi **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 gărzi **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
mySUm (x:xs) = x + mySum xs
```

*Main> mySum [1..5]

*** 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

Tipuri definite de utilizator. Polimorfism. Clase în Haskell.

Tipuri definite de utilizator – Cuprins

- Construcția type
- Construcția newtype
- Tipuri enumerate
- Tipuri înregistrare
- Tipuri recursive
- Tipuri parametrizate

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")]
```

Tipuri definite de utilizator – Cuprins

- Construcția type
- Construcția newtype
- Tipuri enumerate
- Tipuri înregistrare
- Tipuri recursive
- Tipuri parametrizate

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 Person = Person (Name, Age) deriving Show
fc :: Person
fc = Person (("Frederic","Chopin"), 209)
```

Tipuri definite de utilizator – Cuprins

- Construcția type
- Construcția newtype
- Tipuri enumerate
- Tipuri înregistrare
- Tipuri recursive
- Tipuri parametrizate

Tipuri Enumerate

- Numite și tipuri sumă (| face suma/reuniunea valorilor tipului)
- Enumeră toate valorile tipului, sub forma
  data ConsTip = Val_1 | Val_2 | ... | Val_n

Exemple

```
data Dice = S1 | S2 | S3 | S4 | S5 | S6
```

```
*Main> :i Bool

data Bool = False | True          -- Defined in `GHC.Types'
```

Tipuri definite de utilizator – Cuprins

- Construcția type
- Construcția newtype
- Tipuri enumerate
- Tipuri înregistrare
- Tipuri recursive
- Tipuri parametrizate

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, ... câmpₙ :: tipₙ}`
  care este o variantă cu funcții selector pentru definiția
  `data ConsTip = Cons tip₁ ... tipₙ`
- Au un corespondent în majoritatea limbajelor de programare (ex: struct în C++)

Exemplu

```
data Person2 = Person2 {name :: Name, age :: Age}
fc2 :: Person2
fc2 = Person2 ("Frederic","Chopin") 209
composer = name fc2
```

Tipuri definite de utilizator – Cuprins

- Construcția type
- Construcția newtype
- Tipuri enumerare
- Tipuri înregistrare
- Tipuri recursive
- Tipuri parametrizate

Tipuri recursive

- Tipuri pentru care specificăm și cel puțin un constructor intern
  `data ConsTip = .. | Cons_i .. ConsTip .. | ..`

Exemple

```
data Natural = Zero | Succ Natural deriving Show

data IntList = Nil | Cons Int IntList deriving Show
```

Tipuri definite de utilizator – Cuprins

- Construcția type
- Construcția newtype
- Tipuri enumerare
- Tipuri înregistrare
- Tipuri recursive
- Tipuri parametrizate

Tipuri parametrizate

- Constructorul de tip este aplicat pe una sau mai multe variabile de tip, permițând obținerea unor tipuri particulare la instanțiere
  `data ConsTip 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]
```

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Derivarea unei clase
- Instanțierea 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:** length, head, id

Polimorfism ad-hoc = **funcția este supraîncărcată, având comportament diferit în funcție de tipul argumentelor pe care le primește**

- **Ex:** +, *, ==
- `2 + 3 :: Int` întoarce `5`, `2 + 3 :: 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

- **Lizbilitate**
  - 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, permițând 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 distruge 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 :: Num a => [a] -> a`

Tipul a trebuie să supraîncarce operațiile specifice numerelor (+ , - , * ...)

- `:t elem`

  `elem :: Eq a => a -> [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
- Instanțierea unei clase
- Context
- Clase uzuale
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

Clase în Haskell

Clasă Haskell = **mulțime 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 implicite

- În general, clasa doar definește funcțiile care trebuie supraîncărcate, nu le și implementează (firesc, întrucât implementările diferă de la tip la tip)
- Excepție: implementări implicite (**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 implicite

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 minimale, 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
- Instanțierea 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
- Instanțierea 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
        f1 = ... -- implementare
        ...
        fn = ... -- implementare
```

Tip concret, nu variabilă de tip ca la definirea clasei

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
- Instanțierea 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
```

- **instanțierea 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` :: Num a => a , `1` :: Num a => a, `(+)` :: Num a => a -> a -> a)
- Atât otherwise cât și == întorc Bool (necesar pentru a folosi gărzi)
- Tipul lui lst trebuie să fie comparabil pentru egalitate (`(==)` :: Eq a => a -> a -> Bool)
- Tipul lui lst trebuie să fie [t] ([] :: [t], `(==)` :: Eq a => a -> a -> Bool)
- Tipul t trebuie să fie comparabil pentru egalitate (instance Eq a => 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` :: Num a => a , `1` :: Num a => a, `(+)` :: Num a => a -> a -> a)
- Atât otherwise cât și == întorc Bool (necesar pentru a folosi gărzi)
- Tipul lui lst trebuie să fie comparabil pentru egalitate (`(==)` :: Eq a => a -> a -> Bool)
- Tipul lui lst trebuie să fie [t] ([] :: [t], `(==)` :: Eq a => a -> a -> Bool)
- Tipul t trebuie să fie comparabil pentru egalitate (instance Eq a => Eq [a])

myLen :: (Num a, Eq t) => [t] -> 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` :: Num a => a , `1` :: Num a => a, `(+)` :: Num a => a -> a -> 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` :: Num a => a , `1` :: Num a => a, `(+)` :: Num a => a -> a -> a)
- Atât otherwise cât și null întorc Bool (necesar pentru a folosi gărzi)
- Tipul lui lst trebuie să fie [t] (null :: [t] -> Bool)

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` :: Num a => a , `1` :: Num a => a, `(+)` :: Num a => a -> a -> a)
- Atât otherwise cât și null întorc Bool (necesar pentru a folosi gărzi)
- Tipul lui lst trebuie să fie [t] (null :: [t] -> Bool)

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
- Instanțierea 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                          max x y
   | x == y =    EQ                      | x >= y    = x
   | x <= y =    LT                      | otherwise = y
   | otherwise = GT

  x <= y = compare x y /= GT           min x y
  x <  y = compare x y == LT            | x <= y    = x
  x >= y = compare x y /= LT            | otherwise = y
  x >  y = compare x y == GT
```

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                          max x y
   | x == y =    EQ                      | x >= y    = x
   | x <= y =    LT                      | otherwise = y
   | otherwise = GT

  x <= y = compare x y /= GT           min x y
  x <  y = compare x y == LT            | x <= y    = x
  x >= y = compare x y /= LT            | otherwise = y
  x >  y = compare x y == GT
```

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 ȘI 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 aceloraşi 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

Pentru a instanția Ord trebuie să instanțiem și Eq, nu e automat

Exemplu (cu verificări la calculator)

```
data Dice = S1 | S2 | S3 | S4 | S5 | S6 deriving (Eq, Ord, Enum)
```

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Derivarea unei clase
- Instanțierea 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 = Data.Foldable.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 = Data.Foldable.foldl (+) 10 (1,2)      -- 12
```

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Derivarea unei clase
- Instanțierea 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 instanțiată 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
- Instanțierea unei clase
- Context
- Clase uzuale
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

No instance for (<> 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 (<> 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
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

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

Mecanisme pentru definirea tipurilor / sinonimelor de tip

Polimorfism parametric

Polimorfism ad-hoc

Clasă

Definire clasă

Derivare clasă

Instanțiere clasă

Context

Clase uzuale

Clase pentru containere

Rezumat

Mecanisme pentru definirea tipurilor / sinonimelor de tip: newtype, data / type

Polimorfism parametric

Polimorfism ad-hoc

Clasă

Definire clasă

Derivare clasă

Instanțiere clasă

Context

Clase uzuale

Clase pentru containere

Rezumat

Mecanisme pentru definirea tipurilor / sinonimelor de tip: newtype, data / type

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

Mecanisme pentru definirea tipurilor / sinonimelor de tip: newtype, data / type

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

Mecanisme pentru definirea tipurilor / sinonimelor de tip: newtype, data / type

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

Mecanisme pentru definirea tipurilor / sinonimelor de tip: newtype, data / type

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

Mecanisme pentru definirea tipurilor / sinonimelor de tip: newtype, data / type

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

Mecanisme pentru definirea tipurilor / sinonimelor de tip: newtype, data / type

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

Mecanisme pentru definirea tipurilor / sinonimelor de tip: newtype, data / type

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

Mecanisme pentru definirea tipurilor / sinonimelor de tip: newtype, data / type

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

Mecanisme pentru definirea tipurilor / sinonimelor de tip: newtype, data / type

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 propoziţională. 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 neambiguu
  - 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.)* *(**ex:** Bugs e iepure și Leo e broască țestoasă.)*

*(**notație:** p, q, r, etc.)* *(**notație:** ¬P, P ∧ Q, etc.)*

propoziție compusă = negație | conjuncție | disjuncție | implicație | echivalență

(**¬P**) (**P ∧ Q**) (**P ∨ Q**) (**P ⇒ Q**) (**P ⇔ Q**)

Logica propozițională – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Satisfiabilitate și validitate
- Derivabilitate
- Decidabilitate
- Inferență și reguli de inferență
- Demonstrație
- Rezoluție

Semantică

Interpretare = mulțime 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: $p \wedge q$ = true în interpretarea I / $p \wedge q$ = 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

$\neg P_1 = true$	dacă și numai dacă	$P_1 = false$		
$P_1 \wedge P_2 = true$	dacă și numai dacă	$P_1 = true$	și	$P_2 = true$
$P_1 \vee P_2 = true$	dacă și numai dacă	$P_1 = true$	sau	$P_2 = true$
$P_1 \Rightarrow P_2 = true$	dacă și numai dacă	$P_1 = false$	sau	$P_2 = true$
$P_1 \Leftrightarrow P_2 = true$	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

$E = p \wedge (q \vee r)$ în interpretarea $I = \{ p = true, q = true, r = false \}$

$E = true \wedge (true \vee false) = true \wedge true = true$

Logica propozițională – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Satisfiabilitate și validitate
- Derivabilitate
- Decidabilitate
- Inferență și reguli de inferență
- Demonstrație
- Rezoluție

Satisfiabilitate

Propoziția P este **satisfiabilă** ⇔ P este adevărată (true) în cel puțin o interpretare

Propoziția P este **nesatisfiabilă** ⇔ 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)** ⇔ P este adevărată (true) în toate interpretările

Propoziția P este **invalidă** ⇔ P este falsă (false) în cel puțin o interpretare

- Validitatea unei propoziții se determină folosind **tabele de adevăr**

Exemple

- (p ∨ q) ∨ (¬q ∨ r) validă conform tabelei alăturate (întrucât este adevărată în toate interpretările)
- p ∧ (q ∨ r) invalidă conform paginii anterioare (întrucât este falsă în 5 interpretări)

p	q	r	(p ∨ q) ∨ (¬q ∨ 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 **(Δ ⊨ P)** ⇔ orice interpretare care satisface toate propozițiile din Δ satisface și propoziția P

- Derivabilitatea unei propoziții se determină folosind **tabele de adevăr**

Exemplu

- { p, p ⇒ q} ⊨ q conform tabelei alăturate (Singura interpretare care satisface p și p ⇒ q este prima, și prima interpretare satisface și q)

p	q	p ⇒ 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 ⇒ 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**

⇒ 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 mulțimi de premise

Regulă de inferență

$$\frac{P_1, P_2, \ldots P_n}{C}\ (\text{id_regulă})$$

P_i = premise (propoziții considerate adevărate)

C = concluzie (propoziție care derivă din mulțimea de premise)

Notații

$\Delta \models C$ - derivabilitate logică

$\Delta \vdash_{\text{id_regulă}} C$ - derivabilitate mecanică (folosind regula de inferență *id_regulă*)

Reguli de inferență

$$\frac{A \Rightarrow B \qquad A}{B} \; (ModusPonens)$$

$$\frac{A \Rightarrow B \qquad \neg B}{\neg A} \; (ModusTollens)$$

$$\frac{A \vee B \qquad \neg A \vee C}{B \vee C} \; (Rezoluție)$$

$$\frac{}{A \vee \neg A} \; (AxiomaA)$$

Regulile fără premise se numesc **axiome**

Proprietăți ale regulilor de inferență

- **Consistență:** regula determină doar concluzii care derivă logic din premise

Dacă $\Delta \vdash_{id_regulă} C$, atunci $\Delta \vDash C$

- **Completitudine:** regula determină toate concluziile care derivă logic din premise

Dacă $\Delta \vDash C$, atunci $\Delta \vdash_{id_regulă} C$

- 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)

$$\frac{A \Rightarrow B \quad A}{B} \; (ModusPonens)$$

Exemplu

$\{ p \Rightarrow q, q \Rightarrow r \} \models p \Rightarrow r$ (se poate verifica prin tabela de adevăr)

$\{ p \Rightarrow q, q \Rightarrow r \} \nvdash_{ModusPonens} p \Rightarrow r$ (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 ¬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:

$$\frac{A \Rightarrow B \quad A}{B} \; (ModusPonens)$$

Axiome

$$\frac{}{A \Rightarrow (B \Rightarrow A)} \; (IntroImp)$$

$$\frac{}{(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))} \; (DistriImp)$$

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)
  - Oprire 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)

$$\frac{A \vee B \qquad \neg A \vee C}{B \vee C} \; (Rezoluție)$$

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 = mulțime 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 = mulțime 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

$$\frac{\begin{array}{c}\{p, q\} \\ \{\neg p, r\}\end{array}}{\{q, r\}} \ (Rezoluție)$$

Forma generală

$$\frac{\begin{array}{c}\{q_1, \ \dots, p, \dots q_m\} \\ \{r_1, \ \dots, \neg p, \dots r_n\}\end{array}}{\{q_1 \ \dots \ q_m, r_1 \ \dots r_n\}} \ (Rezoluție)$$

Contradicție

$$\frac{\begin{array}{c}\{p\} \\ \{\neg p\}\end{array}}{\{\ \}} \ (Rezoluție)$$

Rezolvent vid: indică existența unei contradicții între premise

Cazuri particulare ale Rezoluției

Modus Ponens

$$\frac{A \Rightarrow B \quad A}{B} \; (ModusPonens)$$

$$\frac{\{\neg p, q\} \quad \{p\}}{\{q\}} \; (Rezoluție)$$

Modus Tollens

$$\frac{A \Rightarrow B \quad \neg B}{\neg A} \; (ModusTollens)$$

$$\frac{\{\neg p, q\} \quad \{\neg q\}}{\{\neg p\}} \; (Rezoluție)$$

Tranzitivitatea implicației

$$\frac{A \Rightarrow B \quad B \Rightarrow C}{A \Rightarrow C} \; (TranziImp)$$

$$\frac{\{\neg p, q\} \quad \{\neg q, r\}}{\{\neg p, r\}} \; (Rezoluție)$$

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 ¬P în premise ← 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 ¬P în premise ← reducere la absurd
  - Derivez clauza vidă
- **P validă**
  - Pornesc cu premisa ¬P
  - Derivez clauza vidă

Se observă că nu avem o metodă de a demonstra satisfiabilitatea / invaliditatea

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

∀X,Y.((iepure(X) ∧ țestoasă(Y)) ⇒ mai_rapid(X,Y)).	*Iepurii sunt mai rapizi decât broaștele țestoase.*
iepure(bugs).	*Bugs e iepure.*
țestoasă(leo).	*Leo e broască țestoasă.*
mai_rapid(bugs, leo).	*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(franța) (calculează o constantă)
- **Predicate:** respectă(porthos,athos)
- **Conective:** ¬, ∧, ∨, ⇒, ⇔ (după cum scade prioritatea)
- **Cuantificatori:** ∀, ∃

  Asociativi la dreapta
- **Egalitate:** =

Sintaxă

termen = constantă | variabilă | funcție(termen,… termen)

propoziție atomică = predicat(termen,… termen) | termen = termen

propoziție = propoziție atomică | (propoziție) | ¬propoziție |
propoziție conectivă_binară propoziție | cuantificator var_1 … 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 = mulțime de asocieri pentru termeni și propoziții
  - Constantelor și funcțiilor li se asociază câte un obiect din D ($c \in D$, $f : D^n \longrightarrow D$)
  - Predicatelor li se asociază o valoare de adevăr (true sau false) ($p : D^n \longrightarrow \{true, false\}$)

Reguli semantice

Cele de la logica propozițională (pentru conective) +

$\forall X.prop = true$ dacă și numai dacă nu există $d \in D$ a.î. $prop_{[d/X]} = false$

$\exists X.prop = true$ dacă și numai dacă există $d \in D$ a.î. $prop_{[d/X]} = true$

Logica cu predicate de ordinul întâi – Cuprins

- Ontologie
- Sintaxă
- Semantică
- Exemple
- Decidabilitate
- Forma clauzală
- Algoritm de aducere la forma clauzală
- Unificare

Exemple

1. Vrabia 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. Vrabia mălai visează.
   $\forall X \bullet (\text{vrabie}(X) \Rightarrow \text{visează}(X,\text{mălai}))$
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. Vrabia mălai visează.
   $\forall X \bullet (vrabie(X) \Rightarrow visează(X,mălai))$
2. Unele vrăbii visează mălai.
   $\exists X \bullet (vrabie(X) \wedge visează(X, mălai))$
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. Vrabia mălai visează.

   $\forall X \bullet (vrabie(X) \Rightarrow visează(X,mălai))$

2. Unele vrăbii visează mălai.

   $\exists X \bullet (vrabie(X) \wedge visează(X, mălai))$

3. Nu toate vrăbiile visează mălai.

$\neg(\forall X \bullet (vrabie(X) \Rightarrow visează(X, mălai)))$ sau $\exists X \bullet (vrabie(X) \wedge \neg visează(X, mălai))$

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. Vrabia mălai visează.

   $\forall X \bullet (vrabie(X) \Rightarrow visează(X,mălai))$

2. Unele vrăbii visează mălai.

   $\exists X \bullet (vrabie(X) \wedge visează(X, mălai))$

3. Nu toate vrăbiile visează mălai.

$\neg(\forall X \bullet (vrabie(X) \Rightarrow visează(X, mălai)))$ sau $\exists X \bullet (vrabie(X) \wedge \neg visează(X, mălai))$

4. Nicio vrabie nu visează mălai.

$\neg(\exists X \bullet (vrabie(X) \wedge visează(X, mălai)))$ sau $\forall X \bullet (vrabie(X) \Rightarrow \neg visează(X, mălai))$

5. Numai vrăbiile visează mălai.

6. Toate și numai vrăbiile visează mălai.

7. Vivi este o vrabie care visează mălai.

Exemple

1. Vrabia mălai visează.

   $\forall X \bullet (vrabie(X) \Rightarrow visează(X,mălai))$

2. Unele vrăbii visează mălai.

   $\exists X \bullet (vrabie(X) \wedge visează(X, mălai))$

3. Nu toate vrăbiile visează mălai.

$\neg(\forall X \bullet (vrabie(X) \Rightarrow visează(X, mălai)))$ sau $\exists X \bullet (vrabie(X) \wedge \neg visează(X, mălai))$

4. Nicio vrabie nu visează mălai.

$\neg(\exists X \bullet (vrabie(X) \wedge visează(X, mălai)))$ sau $\forall X \bullet (vrabie(X) \Rightarrow \neg visează(X, mălai))$

5. Numai vrăbiile visează mălai.

   $\forall X \bullet (visează(X, mălai) \Rightarrow vrabie(X))$

6. Toate și numai vrăbiile visează mălai.

7. Vivi este o vrabie care visează mălai.

Exemple

1. Vrabia mălai visează.

   $\forall X \bullet (vrabie(X) \Rightarrow visează(X,mălai))$

2. Unele vrăbii visează mălai.

   $\exists X \bullet (vrabie(X) \wedge visează(X, mălai))$

3. Nu toate vrăbiile visează mălai.

   $\neg(\forall X \bullet (vrabie(X) \Rightarrow visează(X, mălai)))$ sau $\exists X \bullet (vrabie(X) \wedge \neg visează(X, mălai))$

4. Nicio vrabie nu visează mălai.

   $\neg(\exists X \bullet (vrabie(X) \wedge visează(X, mălai)))$ sau $\forall X \bullet (vrabie(X) \Rightarrow \neg visează(X, mălai))$

5. Numai vrăbiile visează mălai.

   $\forall X \bullet (visează(X, mălai) \Rightarrow vrabie(X))$

6. Toate și numai vrăbiile visează mălai.

   $\forall X \bullet (visează(X, mălai) \Leftrightarrow vrabie(X))$

7. Vivi este o vrabie care visează mălai.

Exemple

1. Vrabia mălai visează.

   $\forall X \bullet (vrabie(X) \Rightarrow visează(X,mălai))$

2. Unele vrăbii visează mălai.

   $\exists X \bullet (vrabie(X) \wedge visează(X, mălai))$

3. Nu toate vrăbiile visează mălai.

   $\neg(\forall X \bullet (vrabie(X) \Rightarrow visează(X, mălai)))$ sau $\exists X \bullet (vrabie(X) \wedge \neg visează(X, mălai))$

4. Nicio vrabie nu visează mălai.

   $\neg(\exists X \bullet (vrabie(X) \wedge visează(X, mălai)))$ sau $\forall X \bullet (vrabie(X) \Rightarrow \neg visează(X, mălai))$

5. Numai vrăbiile visează mălai.

   $\forall X \bullet (visează(X, mălai) \Rightarrow vrabie(X))$

6. Toate și numai vrăbiile visează mălai.

   $\forall X \bullet (visează(X, mălai) \Leftrightarrow vrabie(X))$

7. Vivi este o vrabie care visează mălai.

   $vrabie(vivi) \wedge visează(vivi, mălai)$

Greșeli frecvente

- În general:
  - $\forall$ se folosește împreună cu $\Rightarrow$
  - $\exists$ se folosește împreună cu $\wedge$

- $\forall X \bullet (\text{vrabie}(X) \wedge \text{visează}(X, \text{mălai}))$
  ?

- $\exists X \bullet (\text{vrabie}(X) \Rightarrow \text{visează}(X, \text{mălai}))$
  ?

Greșeli frecvente

- În general:
  - $\forall$ se folosește împreună cu $\Rightarrow$
  - $\exists$ se folosește împreună cu $\wedge$

- $\forall X \bullet$(vrabie(X) $\wedge$ visează(X, mălai))
  înseamnă că toată lumea e vrabie și toată lumea visează mălai

- $\exists X \bullet$(vrabie(X) $\Rightarrow$ visează(X, mălai))
  este o propoziție adevărată și dacă există cineva care nu e vrabie

Proprietăți cuantificatori și conective

- **(Ne)Comutativitate**

  $\forall X.\forall Y$ este totuna cu $\forall Y.\forall X$ și se scrie prescurtat $\forall X,Y.$

  $\exists X.\exists Y$ este totuna cu $\exists Y.\exists X$ și se scrie prescurtat $\exists X,Y.$

  $\exists X.\forall Y$ NU este totuna cu $\forall Y.\exists X$

  - $\exists X.\forall Y.$visează(X,Y) inseamnă că .....
  - $\forall Y.\exists X.$visează(X,Y) inseamnă că .....

- **Dualitate (se pot scrie unul/una în funcție de altul/alta)**

$\forall X.p$	$= \neg(\exists X.\neg p)$	$p \vee q = \neg(\neg p \wedge \neg q)$
$\exists X.p$	$= \neg(\forall X.\neg p)$	$p \wedge q = \neg(\neg p \vee \neg q)$
$\neg(\forall X.p)$	$= \exists X.\neg p$	$\neg(p \vee q) = \neg p \wedge \neg q$
$\neg(\exists X.p)$	$= \forall X.\neg p$	$\neg(p \wedge q) = \neg p \vee \neg q$

Proprietăți cuantificatori și conective

- **(Ne)Comutativitate**

  $\forall X.\forall Y$ este totuna cu $\forall Y.\forall X$ și se scrie prescurtat $\forall X,Y.$

  $\exists X.\exists Y$ este totuna cu $\exists Y.\exists X$ și se scrie prescurtat $\exists X,Y.$

  $\exists X.\forall Y$ NU este totuna cu $\forall Y.\exists X$

  - $\exists X.\forall Y.$visează(X,Y) inseamnă că există cineva care visează la toată lumea
  - $\forall Y.\exists X.$visează(X,Y) inseamnă că la oricine visează măcar cineva

- **Dualitate (se pot scrie unul/una în funcție de altul/alta)**

$\forall X.p$	=	$\neg(\exists X.\neg p)$		$p \vee q = \neg(\neg p \wedge \neg q)$
$\exists X.p$	=	$\neg(\forall X.\neg p)$		$p \wedge q = \neg(\neg p \vee \neg q)$
$\neg(\forall X.p)$	=	$\exists X.\neg p$		$\neg(p \vee q) = \neg p \wedge \neg q$
$\neg(\exists X.p)$	=	$\forall X.\neg p$		$\neg(p \wedge q) = \neg p \vee \neg q$

Definiții moștenite de la logica propozițională

- **Evaluare:** determinarea valorii de adevăr a lui P (într-o interpretare) prin aplicarea regulilor semantice
- **Satisfiabilitate:** P = true în cel puțin o interpretare
- **Validitate:** P = true în toate interpretările
- **Derivabilitate:** P = true în toate interpretările care satisfac premisele
- **Inferență:** derivare prin calcul a concluziilor unei mulțimi 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 ⇒ 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), ¬fricos(X) } echivalent cu prieten(X,porthos) ∨ ¬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**
   $\neg(a \vee b) = \neg a \wedge \neg b$ $\quad$ $\neg(\forall X \bullet p) = \exists X \bullet \neg p$ $\quad$ $\neg(\neg p) = p$
   $\neg(a \wedge b) = \neg a \vee \neg b$ $\quad$ $\neg(\exists X \bullet p) = \forall X \bullet \neg p$

3. **Redenumește variabilele cuantificate a.î. să nu existe cuantificatori diferiți care folosesc același nume de variabilă**
   (ex: $\forall X \bullet p(X) \wedge \forall X \bullet q(X) \vee \exists X \bullet r(X)$ devine $\forall X \bullet p(X) \wedge \forall Y \bullet q(Y) \vee \exists Z \bullet r(Z)$)

4. **Forma prenex: Mută toți cuantificatorii la începutul propoziției, în aceeași ordine**
   (ex: $\forall X \bullet p(X) \wedge \forall Y \bullet q(Y) \vee \exists Z \bullet r(Z)$ devine $\forall X \bullet \forall Y \bullet \exists Z \bullet (p(X) \wedge q(Y) \vee r(Z)$)

Algoritm de aducere la forma clauzală (II)

5. **Skolemizare: Elimină cuantificatorii existențiali prin înlocuire cu**
   - O **constantă** - dacă acest ∃ nu era precedat de niciun ∀
   - O **funcție** de toate variabilele cuantificate universal care îl precedau - altfel
     (ex: $\forall X \bullet \forall Y \bullet \exists Z \bullet (p(X) \wedge q(Y) \vee r(Z))$ devine $\forall X \bullet \forall Y \bullet (p(X) \wedge q(Y) \vee r(f(X,Y)))$)

6. **Elimină cuantificatorii universali (ce rămâne e automat cuantificat universal)**
   (ex: $\forall X \bullet \forall Y \bullet (p(X) \wedge q(Y) \vee r(f(X,Y)))$ devine $p(X) \wedge q(Y) \vee r(f(X,Y))$)

7. **Distribuie ∨ prin ∧**
   $(a \wedge b) \vee c$ 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."

$\boxed{\forall X \bullet (\forall Y \bullet (\text{laborator}(Y) \Rightarrow \text{rezolvă}(X,Y)) \Rightarrow \exists Y \bullet \text{apreciază}(Y,X))}$

1. **Elimină implicațiile**
   $\forall X \bullet (\neg(\forall Y \bullet \neg\text{laborator}(Y) \vee \text{rezolvă}(X,Y)) \vee \exists Y \bullet \text{apreciază}(Y,X))$

2. **Mută negațiile spre interior**
   $\forall X \bullet (\exists Y \bullet \neg(\neg\text{laborator}(Y) \vee \text{rezolvă}(X,Y)) \vee \exists Y \bullet \text{apreciază}(Y,X))$
   $\forall X \bullet (\exists Y \bullet (\text{laborator}(Y) \wedge \neg\text{rezolvă}(X,Y)) \vee \exists Y \bullet \text{apreciază}(Y,X))$

3. **Redenumește variabilele cuantificate**
   $\forall X \bullet (\exists Y \bullet (\text{laborator}(Y) \wedge \neg\text{rezolvă}(X,Y)) \vee \exists Z \bullet \text{apreciază}(Z,X))$

4. **Mută cuantificatorii la început**
   $\forall X \bullet \exists Y \bullet \exists Z \bullet ((\text{laborator}(Y) \wedge \neg\text{rezolvă}(X,Y)) \vee \text{apreciază}(Z,X))$

Aducere la forma clauzală – Exemplu (II)

"Oricine rezolvă toate laboratoarele este apreciat de cineva."

$\forall X \bullet \exists Y \bullet \exists Z \bullet ((\text{laborator}(Y) \wedge \neg\text{rezolvă}(X,Y)) \vee \text{apreciază}(Z,X))$

5. **Elimină cuantificatorii existențiali**
   $\forall X \bullet ((\text{laborator}(f_y(X)) \wedge \neg\text{rezolvă}(X,f_y(X))) \vee \text{apreciază}(f_z(X),X))$

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 ∨ prin ∧**
   $(\text{laborator}(f_y(X)) \vee \text{apreciază}(f_z(X),X)) \wedge (\neg\text{rezolvă}(X,f_y(X)) \vee \text{apreciază}(f_z(X),X))$

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**

$S = \{e_1 / var_1, e_2 / var_2, \ldots e_n / var_n\}$

pentru variabilele din cele 2 expresii astfel încât, în urma substituției, cele 2 expresii devin una și aceeași

Utilitate

$$\frac{\{\neg\text{cal}(X), \text{mănâncă}(X, \text{ovăz})\} \quad \{\text{cal}(X)\}}{\text{mănâncă}(X, \text{ovăz})} \; (Rezoluție)$$

$$\frac{\{\neg\text{cal}(X), \text{mănâncă}(X, \text{ovăz})\} \quad \{\text{cal}(\text{rocinante})\}}{?} \; (Rezoluție)$$

Mulțumită unificării, putem conchide și că Rocinante mănâncă ovăz.

Unificare – Exemple

- P = mănâncă(vivi, X)
  Q = hrănește(vivi, puiu)

- P = mănâncă(vivi, X)
  Q = mănâncă(X, mălai)

- P = mănâncă(Canibal, tata(Canibal))
  Q = mănâncă(tata(X), Y)

Unificare – Exemple

- P = mănâncă(vivi, X)
  Q = hrănește(vivi, puiu)

  Nu unifică!
  (nu au același predicat)

- P = mănâncă(vivi, X)
  Q = mănâncă(X, mălai)

- P = mănâncă(Canibal, tata(Canibal))
  Q = mănâncă(tata(X), Y)

Unificare – Exemple

- P = mănâncă(vivi, X)
  Q = hrănește(vivi, puiu)

  Nu unifică!
  (nu au același predicat)

- P = mănâncă(vivi, X)
  Q = mănâncă(X, mălai)

  Nu unifică!
  (vivi/X, mălai/X, vivi nu unifică cu mălai)

- P = mănâncă(Canibal, tata(Canibal))
  Q = mănâncă(tata(X), Y)

Unificare – Exemple

- P = mănâncă(vivi, X)
  Q = hrănește(vivi, puiu)

  Nu unifică!
  (nu au același predicat)

- P = mănâncă(vivi, X)
  Q = mănâncă(X, mălai)

  Nu unifică!
  (vivi/X, mălai/X, vivi nu unifică cu mălai)

- P = mănâncă(Canibal, tata(Canibal))
  Q = mănâncă(tata(X), Y)

  S = { tata(X) / Canibal, tata(Canibal) / Y } →
  S = { tata(X) / Canibal, tata(tata(X)) / Y }
  subst(S,P) = mănâncă(tata(X), tata(tata(X)))
  subst(S,Q) = mănâncă(tata(X), tata(tata(X)))
  subst(S,P) = mănâncă(tata(X), bunicul(X))

Așa Da
Așa NU

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 evaluează la același obiect
- **2 propoziții atomice** unifică dacă și numai dacă sunt aplicații ale aceluiași predicat asupra câte 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

$$\frac{\begin{array}{c} p_1 \wedge \cdots \wedge p_m \rightarrow \boldsymbol{p} \\ q_1 \wedge \cdots \wedge \boldsymbol{p'} \wedge \cdots \wedge q_n \rightarrow q \\ \text{unificare}(p, p') = S \end{array}}{\text{subst}(S, p_1 \wedge \cdots \wedge p_m, q_1 \wedge \cdots \wedge q_n \rightarrow q)} \; (Rezoluție)$$

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 = 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 = true în cel puțin o interpretare
Validitate: P = 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 = true în cel puțin o interpretare
Validitate: P = true în toate interpretările
Derivabilitate: P = true în toate interpretările care satisfac premisele ($\Delta \vDash P$)
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 = true în cel puțin o interpretare
Validitate: P = true în toate interpretările
Derivabilitate: P = true în toate interpretările care satisfac premisele ($\Delta \vDash P$)
Inferență: derivare prin calcul a concluziilor unei mulțimi de premise ($\Delta \vdash_{id_regulă} P$)
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 = true în cel puțin o interpretare
Validitate: P = true în toate interpretările
Derivabilitate: P = true în toate interpretările care satisfac premisele ($\Delta \vDash P$)
Inferență: derivare prin calcul a concluziilor unei mulțimi de premise ($\Delta \vdash_{id_regulă} P$)
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 = true în cel puțin o interpretare
Validitate: P = true în toate interpretările
Derivabilitate: P = true în toate interpretările care satisfac premisele ($\Delta \vDash P$)
Inferență: derivare prin calcul a concluziilor unei mulțimi de premise ($\Delta \vdash_{id_regulă} P$)
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 = true în cel puțin o interpretare
Validitate: P = true în toate interpretările
Derivabilitate: P = true în toate interpretările care satisfac premisele ($\Delta \vDash P$)
Inferență: derivare prin calcul a concluziilor unei mulțimi de premise ($\Delta \vdash_{id_regulă} P$)
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 = true în cel puțin o interpretare
Validitate: P = true în toate interpretările
Derivabilitate: P = true în toate interpretările care satisfac premisele ($\Delta \vDash P$)
Inferență: derivare prin calcul a concluziilor unei mulțimi de premise ($\Delta \vdash_{id_regulă} P$)
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 = true în cel puțin o interpretare
Validitate: P = true în toate interpretările
Derivabilitate: P = true în toate interpretările care satisfac premisele ($\Delta \vDash P$)
Inferență: derivare prin calcul a concluziilor unei mulțimi de premise ($\Delta \vdash_{id_regulă} P$)
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 = true în cel puțin o interpretare
Validitate: P = true în toate interpretările
Derivabilitate: P = true în toate interpretările care satisfac premisele ($\Delta \vDash P$)
Inferență: derivare prin calcul a concluziilor unei mulțimi de premise ($\Delta \vdash_{id_regulă} P$)
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 = true în cel puțin o interpretare
Validitate: P = true în toate interpretările
Derivabilitate: P = true în toate interpretările care satisfac premisele ($\Delta \vDash P$)
Inferență: derivare prin calcul a concluziilor unei mulțimi de premise ($\Delta \vdash_{id_regulă} P$)
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
Nedemonstrabile prin rezoluție
Unificare

Rezumat

Satisfiabilitate: P = true în cel puțin o interpretare
Validitate: P = true în toate interpretările
Derivabilitate: P = true în toate interpretările care satisfac premisele ($\Delta \vDash P$)
Inferență: derivare prin calcul a concluziilor unei mulțimi de premise ($\Delta \vdash_{id_regulă} P$)
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 = true în cel puțin o interpretare
Validitate: P = true în toate interpretările
Derivabilitate: P = true în toate interpretările care satisfac premisele ($\Delta \vDash P$)
Inferență: derivare prin calcul a concluziilor unei mulțimi de premise ($\Delta \vdash_{id_regulă} P$)
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 = true în cel puțin o interpretare
Validitate: P = true în toate interpretările
Derivabilitate: P = true în toate interpretările care satisfac premisele ($\Delta \vDash P$)
Inferență: derivare prin calcul a concluziilor unei mulțimi de premise ($\Delta \vdash_{id_regulă} P$)
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 => propoziție (se va scrie doar propoziție.)

```
om(gica). om(ilie). impiedicat(gica).
traverseaza(ilie, santier). %% un șantier anume (identificat prin constanta santier)
sapa_groapa(ilie, gica).    %% o groapă oarecare (nu trebuie identificată)
```

  - **Reguli** de forma propoziție$_1$ ∧ … ∧ propoziție$_n$ => propoziție
    (se folosește scrierea propoziție :- propoziție$_1$, … , 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 ≠ negația logică
    - \+p în Prolog = programul curent nu poate demonstra p

```
bun(X) :- \+sapa_groapa(X,_).          %% false, fiindca sapa_groapa e satisfiabil
om_bun(X) :- om(X), \+sapa_groapa(X,_). %% X = gica
```

    - ¬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` (puține! unitatea de bază e propoziția, nu funcția)
- **Structuri:** (modelează proprietățile și relațiile obiectelor)

```
carte(titlu('Magicianul'), autor('John Fowles'))
sapa_groapa(ilie, gica)
```

- **Conective:** `, ;` (virgulă = ∧, punct și virgulă = ∨)
  (virgula are prioritate mai mare)

Sintaxă

termen = constantă | variabilă | structură
(obiect cu componente)

clauză = fapt | regulă

```
sapa_groapa(ilie, gica).
```

fapt = structură.

```
om_bun(X) :- om(X), \+sapa_groapa(X,_).
```

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).
true.
```

```
?- cade_in_groapa(gica).
false.
```

```
?- cade_in_groapa(X).
X = ilie.
```

Test

Traduceți următoarea propoziție în LPOI:

„Undeva, toți câinii au covrigi în coadă.”

Obs: Atenție la folosirea constantelor: de exemplu, nu vrem ca toți câinii să umble cu același covrig în coadă, este foarte incomod!

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 grație 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** (mulțime 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) :- sapa_groapa(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 **instanțiază variabilele** se pun primele (v. parent și v. mai jos)

```
factorial(0, 1).
factorial(N, F) :-
        N > 0,
        N1 is N-1, factorial(N1, F1),
        F is N * F1.

factorial_1(0, 1).
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)                 (_=1, _=[2,3])
true                  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])
                                              (_=1, _=[3])
                      true                    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]).
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]).
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).
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 (instanțierile 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).
```

X	Rest	L

X	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).
```

Regula se citește: dacă L1 este o listă cu head-ul X și tail-ul Rest, și dacă Rest concatenat cu L dă Rez, atunci L1 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).
2. concat([X|Rest], L, [X|Rez]) :- concat(Rest, L, Rez).
```

X	Rest	L

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 \= 3.`
  - Pentru **atribuire cu evaluarea** operațiilor aritmetice se folosește **is**: `X is 1+2.`
  - Pentru **verificarea egalității / inegalității valorilor** se folosesc **=:=** și **=\=**: `1+2 =\= 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 \= 3. (true)`
  - Pentru **atribuire cu evaluarea** operațiilor aritmetice se folosește **is**: `X is 1+2. (X = 3)`
  - Pentru **verificarea egalității / inegalității valorilor** se folosesc **=:=** și **=\=**: `1+2 =\= 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: oprirea 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 ca 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 altă regulă ulterioară

Negația ca eșec

Revenind pe șantier...

```
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)" ⇔ „Oricare X, X nu sapă groapa nimănui"
  - Funcționare: sapa_groapa(X,_) unifică cu sapa_groapa(ilie, gica)
    combinația !, fail condamnă satisfacerea scopului curent la eșec definitiv

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).
```

```
?- 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 ⇔
  Programe ușor de înțeles, care fac ceea ce ne așteptăm să facă
- Prin urmare, atât cut cât și not trebuie folosite cu grijă 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

Metapredicate. 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

Metapredicate 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^Goal**

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)
- Dacă doresc să nu se țină cont de instanțierile 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)
- Dacă doresc să nu se țină cont de instanțierile 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 5x5.

- 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 intermediar N1:** [1/1, 2/3] (din linia 1 și coloana 1 s-a sărit pe linia 2 și coloana 3)
- **Nod intermediar 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:** (N1, N2) (tranziție legală, spre deosebire de (N1, 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 adecvat 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)
    - metapredicate (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 **tranziție 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

- **Template** (ex pt săritura calului: [1/1, _])
  - Adecvat cazurilor în care cunoaștem diverse detalii despre forma soluției finale
    - Lungimea acesteia
    - Ce se află pe anumite poziții (de exemplu, la soluția unui Sudoku)
  - Câmpurile deja instanțiate opresc automat expandările care nu se potrivesc cu template-ul
- **Regulă de recunoaștere** (ex pt săritura calului: length(Sol, 25))
  - O stare finală este o stare care îndeplinește un set de condiții
  - Înainte de orice expandare, verificăm dacă am ajuns deja într-o stare finală

Observație

- Cu sau fără template, se vor expanda multe căi inutile dacă programatorul nu filtrează încă din predicatul arc sau imediat după ce efectuează o tranziție folosind predicatul 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 paradigma 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 sa 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