Paradigme de Programare

S.I. dr. ing. Andrei Olaru
slides: Mihnea Muraru si Andrei Olaru

Catedra de Calculatoare

2013 – 2013, semestrul 2

Cursul 5

Evaluare leneşă în Scheme

Cuprins

Întârziere

Exemplul 20.1.

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

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

Varianta 1

Încercare – implementare directă

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

Output:

Varianta 1

Încercare – implementare directă

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

Output: `y 0 | y 30`

- Implementare eronată, deoarece ambii parametri sunt evaluaţi în momentul aplicării

Varianta 2

Încercare – `quote` & `eval`

```
1  (define prod
2    (lambda (x y)
3      (if x (* (eval y) (+ (eval y) 1)) 0))) ; eval
4
5  (define test
6    (lambda (x)
7      (let ((y 5))
8        (prod x '(begin (display "y␣") y))))) ; quote
9
10 (test #f)
11 (test #t)
```

Output:

Varianta 2

Încercare – `quote` & `eval`

```
1  (define prod
2    (lambda (x y)
3      (if x (* (eval y) (+ (eval y) 1)) 0))) ; eval
4
5  (define test
6    (lambda (x)
7      (let ((y 5))
8        (prod x '(begin (display "y ") y))))) ; quote
9
10 (test #f)
11 (test #t)
```

Output: `0 | reference to undefined identifier`

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

Varianta 3

Încercare – închideri funcţionale

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

Output:

Varianta 3

Încercare – închideri funcţionale

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

Output: `0 | y y 30`

- Comportament corect: `y` evaluat la cerere
- `x = #t` $\rightarrow$ `y` evaluat de 2 ori – ineficient

Varianta 4

Promisiuni: `delay` & `force`

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

Output:

Varianta 4

Promisiuni: `delay` & `force`

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

Output: `0 | y 30`

- Comportament corect: `y` evaluat la cerere, o singură dată $\rightarrow$ evaluare leneşă

Promisiuni

Descriere

- Rezultatul încă neevaluat al unei expresii
- Exemplu: `(delay (* 5 6))`
- Valori de prim rang în limbaj (v. Definiţia 4.8)
- `delay`
  - construieşte o promisiune
  - funcţie nestrictă
- `force`
  - forţează respectarea unei promisiuni, evaluând expresia doar la prima aplicare, şi salvându-i valoarea
  - începând cu a doua invocare, întoarce, direct, valoarea memorată

Promisiuni

Cerinţe

- Salvarea contextului computaţional al expresiei a cărei evaluare este întârziată şi evaluarea ei ulterioară în acel context $\rightarrow$ asemănător cu închiderile funcţionale.
- Salvarea rezultatului primei evaluări a expresiei
- Distingerea primei forţări de celelalte $\rightarrow$

Promisiuni

Cerinţe

- Salvarea contextului computaţional al expresiei a cărei evaluare este întârziată şi evaluarea ei ulterioară în acel context $\rightarrow$ asemănător cu închiderile funcţionale.
- Salvarea rezultatului primei evaluări a expresiei
- Distingerea primei forţări de celelalte $\rightarrow$ efect lateral.

Promisiuni

Implementare în Scheme (1)

- `p` = o promisiune – expresie care se evaluează numai când este necesar prima oară, și reține valoarea la care s-a evaluat;
- `my-delay` construiește promisiunea;
- `my-force` evaluează promisiunea;

```
1  (define-macro my-delay (lambda (expr)
2      `(make-promise (lambda () ,expr))
3  ))
4
5  (define my-force (lambda (p)
6      (p)
7  ))
```

Promisiuni

Implementare în Scheme (2a)

```
1  (define make-promise (lambda (closure)
2      (let ((ready? #f) (result #f))
3        (lambda ()     ; promisiunea
4          (if (not ready?)
5            (begin (set! ready? #t)
6                   (set! result (closure))))
7          result
8  ))))
```

Dar dacă:

```
1  (define x 1)
2  (define p (my-delay (if (= x 3) 0
3           (begin (set! x (+ x 1)) (my-force p) 100)
4  )))
```

`(my-force p)` returnează 100, deși prima valoare calculată de o promisiune terminată a fost 0 (când `x` a ajuns la 3).

Promisiuni

Implementare în Scheme (2b – corect)

```
1  (define make-promise (lambda (closure)
2      (let ((ready? #f) (result #f))
3        ; promisiunea
4        (lambda ()
5          (if ready?
6              result
7              (let ((r (closure)))
8                (if ready?
9                    result
10                   (begin (set! ready? #t)
11                          (set! result r)
12                          result)
13             ))
14         ))
15 )))
```

Promisiuni

Implementare în Scheme – discuție

- Situaţii în care evaluarea expresiei împachetate declanşează, ea însăşi, forţarea promisiunii → a doua verificare a lui `ready?`.
- Promisiuni → obiecte cu stare.
- Prima forţare → efecte laterale.

Observaţii

- Dependenţă între mecanismul de întârziere şi cel de evaluare ulterioară a expresiilor — închideri/aplicaţii (varianta 3), `delay`/`force` (varianta 4) etc.
- Număr mare de modificări la înlocuirea unui mecanism existent, utilizat de un număr mare de funcţii
- Cum se pot diminua dependenţele?

Abstracţii

Abstracţii procedurale

Motivație

- Context:
  - Probleme cu complexitate ridicată.
  - Descompunere în subprobleme, dar până unde?
  - Nevoia restrângerii detaliilor luate în calcul la un anumit moment $\rightarrow$ abstractizare.
  - Lucru la nivel conceptual, al gândirii programatorului, deasupra nivelului operaţiilor elementare din limbaj.

Abstracţii procedurale

Exemplu

Exemplul 21.1.

```
1  (define sum-of-squares
2    (lambda (x y)
3      (+ (square x) (square y))))
4
5  (define square
6    (lambda (x)
7      (* x x)))
```

- `sum-of-squares`: conceptul de sumă a pătratelor, deasupra conceptului de ridicare la pătrat
- `square`: conceptul de ridicare la pătrat, deasupra conceptului de înmulţire

Abstracţii procedurale

Definiție

- Combinarea conceptelor pentru obţinerea de concepte mai complexe, cu propria identitate.
- `square`, din perspectiva `sum-of-squares`, substituibilă cu orice altă funcţie cu acelaşi comportament.

Definiția 21.2 (Abstracţie procedurală).

Funcţionalitate autonomă, independentă de implementare.

Abstracţii procedurale

Cerințe

- Izolarea implementării de utilizare $\rightarrow$ modularitate.
- Reutilizabilitate.
- `square`, `sum-of-squares` $\rightarrow$ generalizare la nivel de numere
- Funcţionale (e.g. `map`, `filter`, `foldl`) $\rightarrow$ generalizare la nivel de comportament !
- Gândirea în termenii diverselor abstracţii răspândite (*patterns*) $\rightarrow$ aplicarea lor în situaţii noi.

Abstracţii de date

Motivație

- Exemplu: cum reprezentăm expresiile cu evaluare întârziată?
- Abordarea din secţiunea precedentă: 1 singur nivel:

funcţii ce operează cu expresii
cu evaluare întârziată:
implementare şi utilizare,
sub formă de închideri sau promisiuni

Abstracţii de date

Soluție

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

funcţii ce operează cu expresii cu evaluare întârziată: utilizare
interfață: `pack`, `unpack`
expresii cu evaluare întârziată, ca închideri funcționale sau promisiuni: implementare

- Bariera:
  - limitează analiza detaliilor
  - elimină dependenţele dintre nivele

Abstracţii de date

Definiție

Definiția 21.3 (Abstracţie de date).

Tehnică de separare a utilizării unei structuri de date de implementarea acesteia.

- Permit *wishful thinking*: utilizarea structurii înaintea implementării acesteia.

Abstracţii de date

Implementări diferite, aceeași utilizare; v1: promisiuni

Exemplul 21.4 (Continuare a exemplului 20.1).

```
1  (define-macro pack (lambda (expr)
2      `(delay ,expr)))
3
4  (define unpack force)
5
6  (define prod (lambda (x y)
7      (if x (* (unpack y) (+ (unpack y) 1)) 0)))
8
9  (define test (lambda (x)
10     (let ((y 5))
11       (prod x (pack (begin (display "y ") y))) )))
```

Abstracţii de date

Implementări diferite, aceeași utilizare; v2: inchideri

Exemplul 21.5 (Continuare a exemplului 20.1).

```
1  (define-macro pack (lambda (expr)
2      `(lambda () ,expr) ))
3
4  (define unpack (lambda (p) (p)))
5
6  (define prod (lambda (x y)
7      (if x (* (unpack y) (+ (unpack y) 1)) 0)))
8
9  (define test (lambda (x)
10     (let ((y 5))
11       (prod x (pack (begin (display "y ") y))) )))
```

Fluxuri

Motivaţie

Exemplul 22.1.

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

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

Observaţii

- Varianta 1 – iterativă (d.p.d.v. proces): eficientă, datorită spaţiului suplimentar constant
- Varianta 2 – folosește liste:
  - elegantă şi concisă
  - ineficientă, datorită spaţiului posibil mare, ocupat la un moment dat – toate numerele din intervalul $[a, b]$
- Cum îmbinăm avantajele celor 2 abordări?

Fluxuri

Caracteristici

- Secvenţe construite parţial, extinse la cerere, ce creează iluzia completitudinii structurii
- Îmbinarea eleganţei manipulării listelor cu eficienţa calculului incremental
- Bariera de abstractizare:
  - componentele listelor evaluate la construcţie (`cons`)
  - componentele fluxurilor evaluate la selecţie (`cdr`)
- Construcţie şi utilizare:
  - separate la nivel conceptual $\rightarrow$ modularitate
  - întrepătrunse la nivel de proces

Fluxuri

Operatori: construcție și selecție

- `cons, car, cdr, nil, null?.`

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

Fluxuri

Operatori: `take` și `drop`

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

```
1  (define stream-take (lambda (n s)
2    (cond ((zero? n) '())
3          ((stream-null? s) '())
4          (else (cons (stream-car s)
5                      (stream-take (- n 1) stream-cdr s)))
6  )))
7
8  (define stream-drop (lambda (n s)
9    (cond ((zero? n) s)
10         ((stream-null? s) s)
11         (else (stream-drop (- n 1) (stream-cdr s)))
12 )))
```

Fluxuri

Operatori: `map` și `filter`

- operatori de aplicare și filtrare pe liste.

```
1  (define stream-map (lambda (f s)
2      (if (stream-null? s) s
3          (stream-cons (f (stream-car s))
4              (stream-map f (stream-cdr s)))
5  )))
6
7  (define stream-filter (lambda (f? s)
8      (cond ((stream-null? s) s)
9            ((f? (stream-car s))
10             (stream-cons (stream-car s)
11                 (stream-filter f? (stream-cdr s))))
12           (else (stream-filter f? (stream-cdr s)))
13 )))
```

Fluxuri

Operatori: `zip`, `append` și conversie

```
1  (define stream-zip (lambda (f s1 s2)
2    (if (stream-null? s1) s2
3      (stream-cons (f (stream-car s1) (stream-car s2))
4       (stream-zip f (stream-cdr s1) (stream-cdr s2)))
5  )))
6
7  (define stream-append (lambda (s1 s2)
8    (if (stream-null? s1) s2
9      (stream-cons (stream-car s1)
10       (stream-append (stream-cdr s1) s2)))))
11
12 (define list->stream (lambda (L)
13   (if (null? L) stream-null
14     (stream-cons (car L) (list->stream (cdr L))))))
```

Fluxuri – Exemple

Implementarea unui flux de numere 1

- Definiție cu închideri:

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

- Definiție cu fluxuri:

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

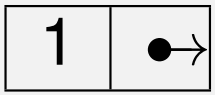
- Definiție cu promisiuni:

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

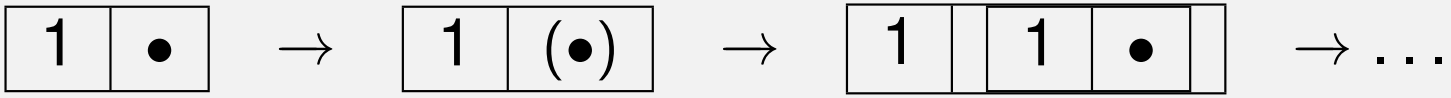
Fluxuri – Exemple

Flux de numere 1 – discuție

- Extinderea se realizează în spaţiu constant:



- Ca proces:



- Structural:

Fluxul numerelor naturale

Formulare explicită

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

- Închideri: multiple parcurgeri ale fluxului determină reevaluarea porţiunilor deja explorate.
- Promisiuni: multiple parcurgeri ale fluxului determină evaluarea dincolo de porţiunile deja explorate.

Fluxul numerelor naturale

Formulare implicită

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

- Porţiunea deja explorată din flux poate fi utilizată pentru explorarea porţiunii următoare

Fluxul numerelor pare

În două variante

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

Fluxul numerelor prime

Metodă

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

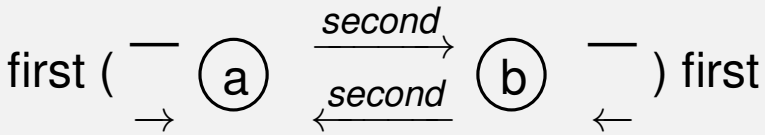
Fluxul numerelor prime

Implementare

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

Grafuri ciclice

Concept



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

Grafuri ciclice

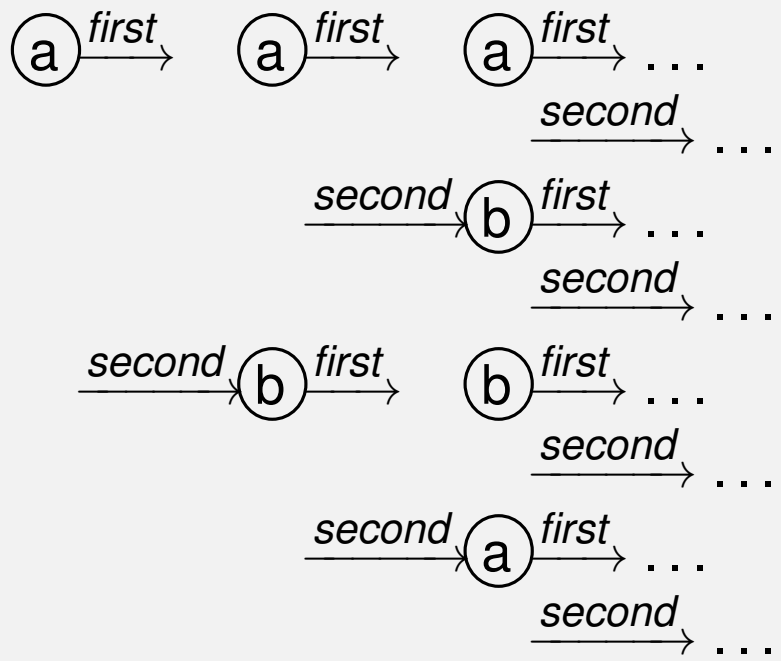
Implementare

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

Grafuri ciclice

Explorare

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



Căutare

Spaţiul stărilor unei probleme

Definiția 23.1 (Spaţiul stărilor unei probleme).

Mulţimea configuraţiilor valide din universul problemei.

Exemplul 23.2.

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

Stările problemei $\rightarrow$toate şirurile generabile cu elementele alfabetului respectiv.

Specificarea unei probleme prin spaţiul stărilor

Aplicaţie pe Pal_n

- Starea iniţială: şirul vid
- Operatorii de generare a stărilor succesor ale unei stări: inserarea unui caracter la începutul unui şir dat
- Operatorul de verificare a proprietăţii de *scop* a unei stări: palindrom

Căutare în spaţiul stărilor

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

Căutare în lăţime

Obișnuită

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

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

Căutare în lăţime

Lenesă (1) – fluxul stărilor *scop*

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

Căutare în lăţime

Lenesă (2)

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

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

Sfârșitul cursului 5

Ce am învățat

· Aplicații ale evaluării întârziate, abstractizare procedurală, fluxuri, căutare în spațiul stărilor.