

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)

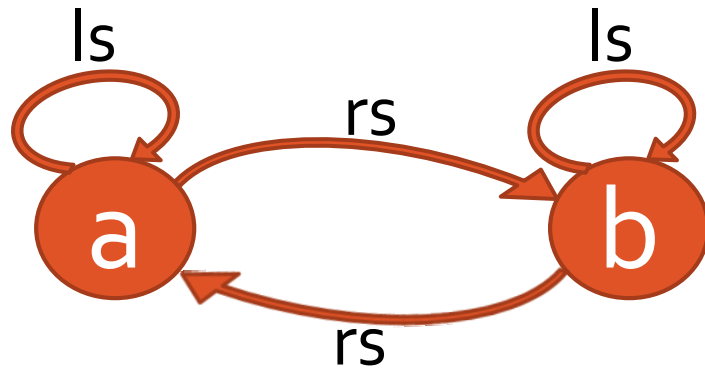
Beneficii

- **Performanță** crescută (se evită calcule inutile – care pot fi multe sau costisitoare)
- Implementare de **funcții nestrict** (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

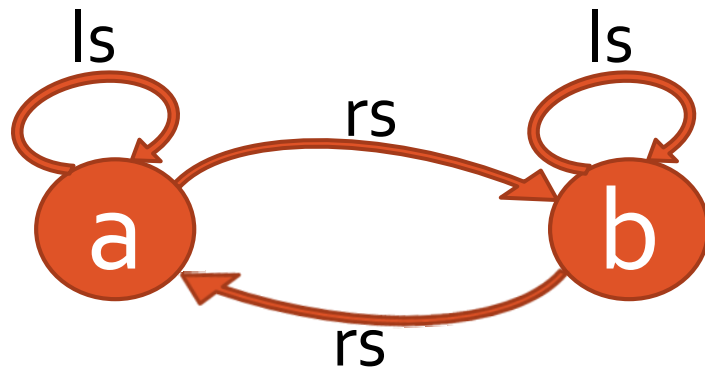
Un graf se reprezintă prin

nodul rădăcină (aici a)

Structură infinită

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

Exemplu – Un graf recursiv (infinit)



Contextul global

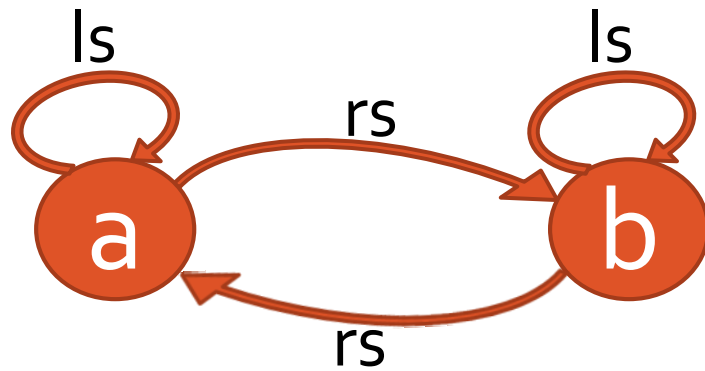
```
(g1 (list 'a  
        (λ().corpa ▪ )  
        (λ().corpb ▪ )))
```

Contextul lui letrec

```
(a (λ().corpa ▪ ))  
(b (λ().corpb ▪ ))
```

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

Exemplu – Un graf recursiv (infini)



Contextul global

```
(g1 (list 'a
         (λ().corpa ▪ )
         (λ().corpb ▪ )))
```

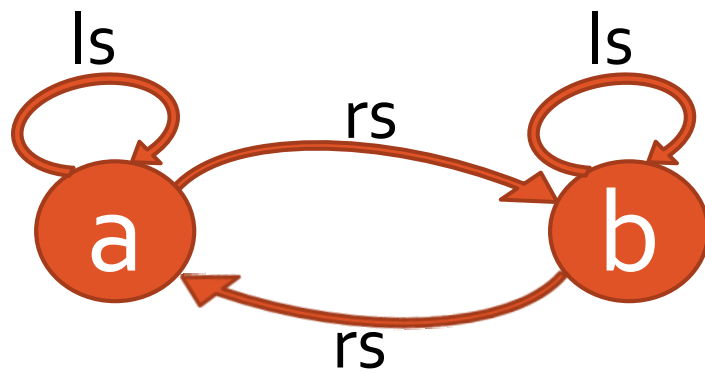
Contextul lui letrec

```
(a (λ().corpa ▪ ))
(b (λ().corpb ▪ ))
```

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().corpa ▪) (λ().corpb ▪))`

Exemplu – Un graf recursiv (infini)



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

```
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))
```

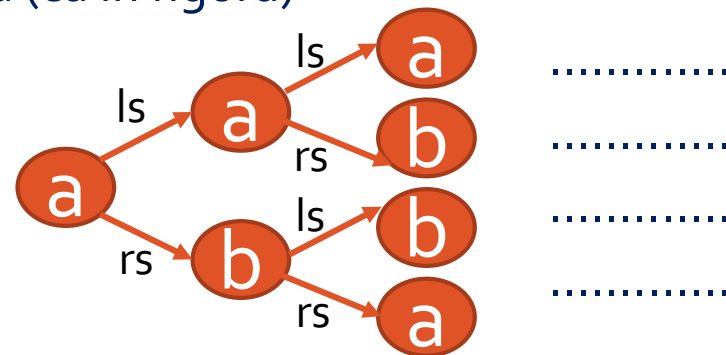

Implementare cu închideri funcționale

Avantaj

- Capabilă să reprezinte graful infinit

Dezavantaje

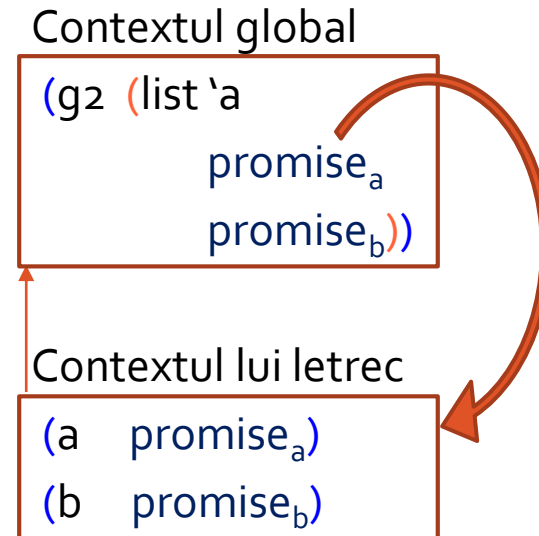
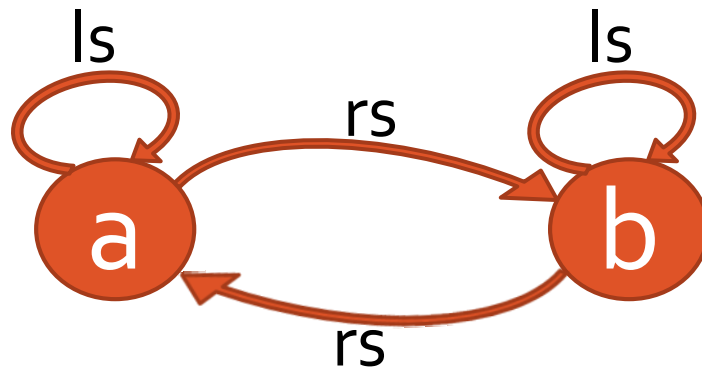
- **Ineficientă** din cauza evaluării repetate (de fiecare dată când accesăm vecinii unui nod) a unor închideri deja evaluate
- **Probleme „filozofice” de identitate**: vecinul stâng al lui a este chiar a, nu un alt nod identic cu a (ca în figură)



Întârzierea evaluării – Cuprins

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

Exemplu – Un graf recursiv (infini)

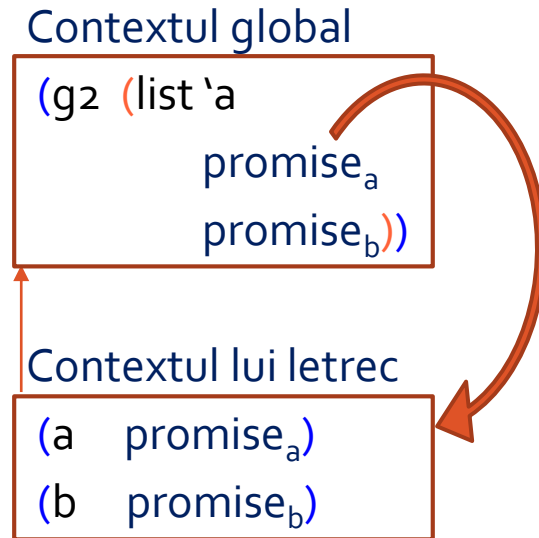
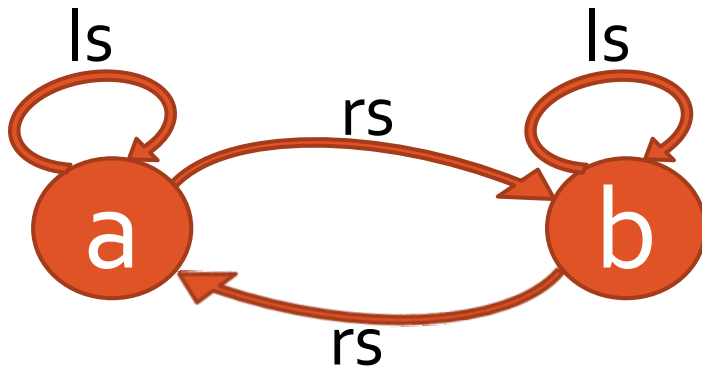


```
1. (define g2
2.   (letrec ((a (delay (list 'a a b)))
3.           (b (delay (list 'b b a))))
4.     (force a)))
```

← (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 deus 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

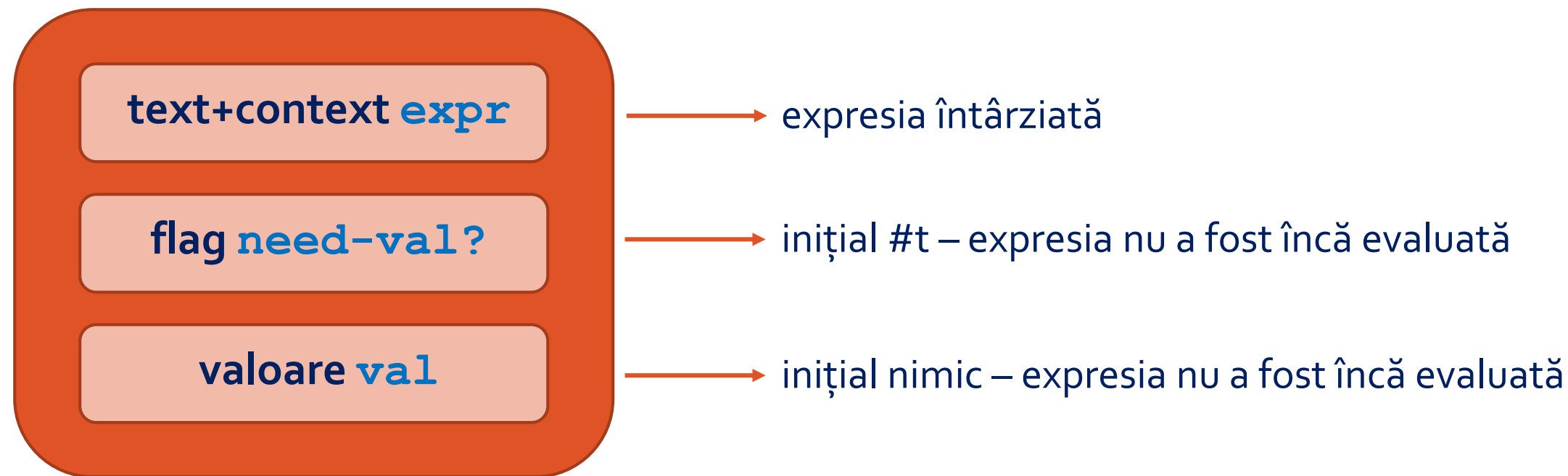
- Atât închiderile funcționale cât și promisiunile sunt **valori de ordinul întâi** (ex: le-am putut compara cu equal?)

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 întârziată 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 simple de utilizat)

Viziuni posibile asupra lumii

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

Exemplu la calculator

Să se determine dacă un număr n este prim.

Definiție

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

O soluție modulară



Eleganță versus Eficiență

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

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

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

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

Fluxuri – Cuprins

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

Eleganță plus Eficiență

```
1. (define (interval-stream a b)
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
cele 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



Complexitate spațială

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

1

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

Complexitate temporală

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

Exemplu

```
(prime? 115)
(stream-empty? (stream-filter div? '(2 . promise[3-10.72])))
;; 115 : 2? Nu, atunci filtrăm restul intervalului
(stream-empty? (stream-filter div? '(3 . promise[4-10.72])))
;; 115 : 3? Nu, atunci filtrăm restul intervalului
(stream-empty? (stream-filter div? '(4 . promise[5-10.72])))
;; 115 : 4? Nu, atunci filtrăm restul intervalului
(stream-empty? (stream-filter div? '(5 . promise[6-10.72])))
;; 115 : 5? Da, atunci avem un rezultat pentru stream-filter
;; 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

'()

cons

Constructor corespunzător pe fluxuri

empty-stream

stream-cons

Operator pe liste

null?

car

cdr

map

filter

Operator corespunzător pe fluxuri

stream-empty?

stream-first

stream-rest

stream-map

stream-filter

Fluxuri – Cuprins

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

Definiții explicite (generând fiecare element)

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

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





Exemple

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

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

Extinderea fluxului la cerere

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

- Inițial, fluxul își știe doar primul element: naturals = '(0 )
- **Extinderea** fluxului (calcularea mai multor elemente) se face **la cerere** (adică la stream-rest): de exemplu, (stream-take naturals 3) va cere încă 2 elemente
- **Evoluția** '(0 ) (loop 1) '(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))
```

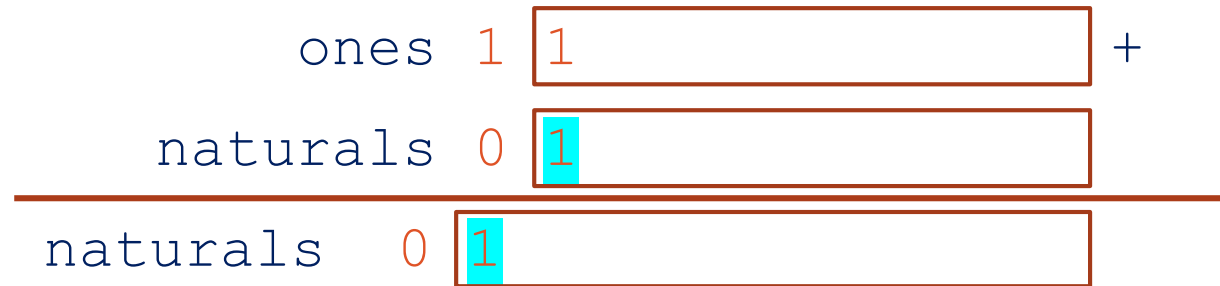
```
(define naturals-  
  (stream-cons 0  
    (stream-zip-with + ones naturals-)))
```


Operații între fluxuri – Funcționare

$$\begin{array}{r} \text{ones } 1 \quad \boxed{} + \\ \text{naturals } 0 \quad \boxed{} \\ \hline \text{naturals } 0 \quad \boxed{} \end{array}$$

- 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

$$\begin{array}{r} \text{ones } 1 \quad \boxed{1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1} \quad + \\ \text{naturals } 0 \quad \boxed{1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8} \\ \hline \text{naturals } 0 \quad \boxed{1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ \dots} \end{array}$$

- 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

$$\begin{array}{r} \text{fibonacci } 0 \quad \boxed{1 \ 1 \ 2 \ 3 \ 5 \ 8} \quad + \\ (\text{rest fibonacci}) \quad 1 \quad \boxed{1 \ 2 \ 3 \ 5 \ 8 \ 13} \\ \hline \text{fibonacci} \quad 0 \ 1 \quad \boxed{1 \ 2 \ 3 \ 5 \ 8 \ 13 \ 21 \ \dots} \end{array}$$

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

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

Rezumat

Promisiune

Avantaje promisiuni

delay / force

Fluxuri

Avantaje fluxuri

Constructorii flux

Operatorii 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

Constructorii flux

Operatorii 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

Constructorii flux

Operatorii 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

Constructorii flux

Operatorii 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

Constructorii flux

Operatorii 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)

Constructorii 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)

Constructorii flux: empty-stream, stream-cons

Operatorii 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)

Constructorii flux: empty-stream, stream-cons

Operatorii 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)

Constructorii flux: empty-stream, stream-cons

Operatorii 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)