PARADIGME DE PROGRAMARE

Curs 8

Modelul contextual de evaluare. Întârzierea evaluării.

Modelul contextual de evaluare – Cuprins

- Context computațional
- Închideri funcționale

Motivație

```
1.    (define (make-counter x)
2.      (lambda () (set! x (+ x 1)) x))
3.    (define c1 (make-counter 10))
4.    (c1)
```

Modelul substituției

- Aplică `(make-counter 10)` și întoarce `(lambda () (set! 10 (+ 10 1)) 10)`
- Dezavantaje majore:
  - Ineficiență: scanează funcția pentru a face toate înlocuirile
  - Incorectitudine: `(set! 10 11)` nu are sens (s-a pierdut variabila care trebuia alterată în viitor)

Motivație

Observație: Modelul substituției este limitat chiar dacă toate funcțiile sunt pure:

- Ineficient
  - O(n) pentru fiecare scanare în vederea efectuării înlocuirilor
- Incapabil să modeleze funcții textual recursive
  - în corpul funcției, substituie la infinit numele funcției cu implementarea sa textuală

Soluția: Modelul contextual:

- Eficient – partajare, nu copiere
  - Textul funcției rămâne nemodificat
  - Funcția are acces la o „filă” pe care sunt notate (și, la nevoie, alterate) asocierile (identificator valoare)
- Capabil să modeleze funcții textual recursive – pointeri, nu text infinit
  - Funcția f este împachetată cu un pointer la „fila” ei (care include asocierea (f valoare-f))

Context computațional

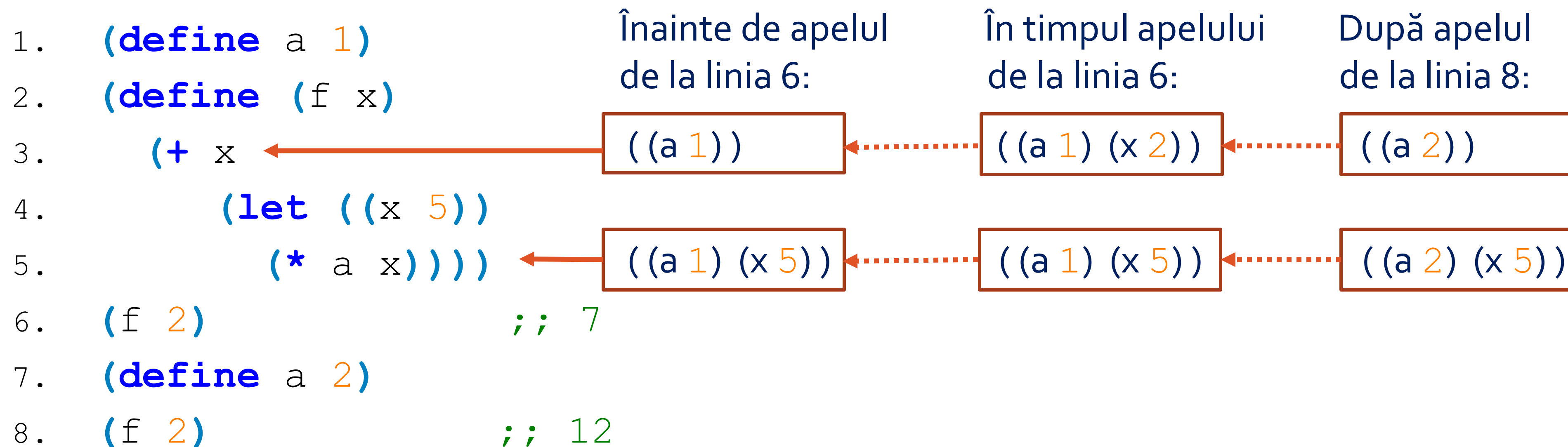
Context computațional al unui punct P din program

= **mulțimea variabilelor care au punctul P în domeniul lor de vizibilitate**
(o mulțime de perechi identificator-valoare)

Observații

- Legare statică: contextul este vizibil imediat ce am scris programul
- Legare dinamică: valoarea contextului depinde de momentul în care se află execuția
  - numai valoarea variabilelor legate dinamic poate să difere de la un moment la altul
- Contextul nu conține informații despre un identificator nelegat
  - punctele din corpul unei funcții conțin informații despre parametrii formali doar în timpul apelului (când este activă legarea parametru formal – parametru efectiv)

Context computațional – Exemplu



Observație: La linia 3, legarea (x 2) există doar pe durata evaluării apelurilor (f 2).

Modelul contextual de evaluare – Cuprins

- Context computațional
- Închideri funcționale

Închideri funcționale

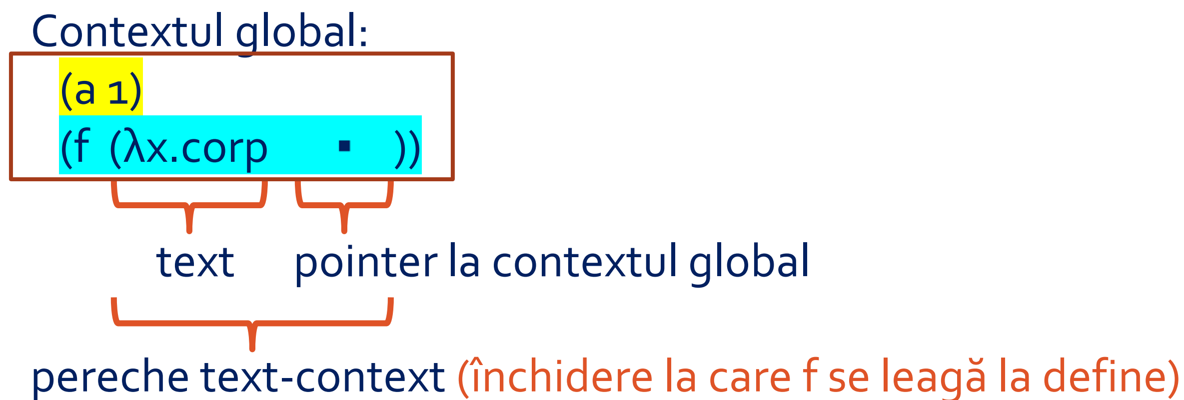
Închidere funcțională

= **pereche text – context (textul funcției și contextul în punctul de definire a funcției**

(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))))
```



Observație: Expresiile de legare statică vor crea noi contexte, locale

- ex: apelul (f 2) pentru legarea (x 2), let 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

Evaluare

- Caută variabila în contextul curent
- Dacă variabila nu este în contextul curent, caută în contextul părinte, ș.a.m.d.

Exemplu “capcană”

```
1.   (define y 5)
2.   (letrec ((x y) (y 1))
3.     x)
```

Contextul global:

(y 5)

Context local adăugat de expresia letrec:

(x ...)
(y ...)

Efectuarea legărilor (în letrec)

- Legările se fac secvențial (stânga-dreapta)
- Pasul 1: x trebuie să se lege la valoarea lui y
  - Se caută variabila y în contextul curent
  - y este în contextul curent dar este nedefinit (nelegat) ⇒ nu va fi căutat în contextul părinte, ⇒ eroare la execuție (nu la compilare)

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)                                  ;; 12
```

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

```
(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).
O aplicație importantă a închiderilor funcționale este întârzierea evaluării.

Întârzierea evaluării – Cuprins

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

Evaluare întârziată

Evaluare întârziată

- Se amână evaluarea expresiilor până când valoarea lor este necesară continuării execuției

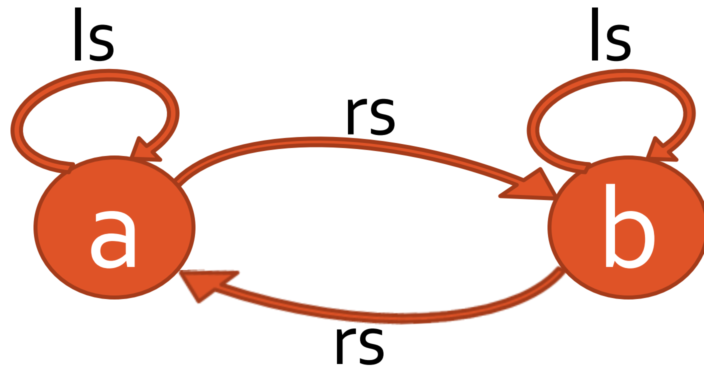
Beneficii

- Performanță: evită calcule inutile – care pot fi multe sau costisitoare
- Funcții nestricte (ex: if, and, or) utile în:
  - Controlul fluxului prin program
  - Condiții de oprire
    `(or (null? L) (zero? (car L)))` – evaluează `(car L)` doar dacă L nu e vidă
- Structuri de date infinite, din care se evaluează (la cerere) doar porțiunea necesară
  - `[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
- Tabel comparativ

Exemplu – Un graf recursiv (infinit)

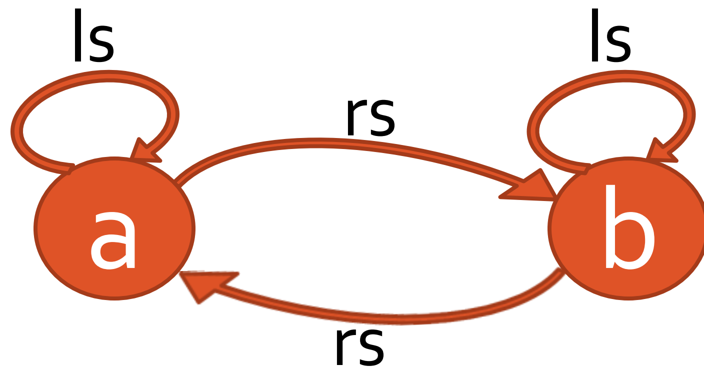


Un nod se reprezintă prin
key = informația din nod
ls = legătura la stânga
rs = legătura la dreapta

Un graf se reprezintă prin
nodul rădăcină (aici a)

Structură infinită
⟶ trebuie să împiedicăm evaluarea întregului graf în momentul definirii sale

Exemplu – Un graf recursiv (infinit)



Contextul global

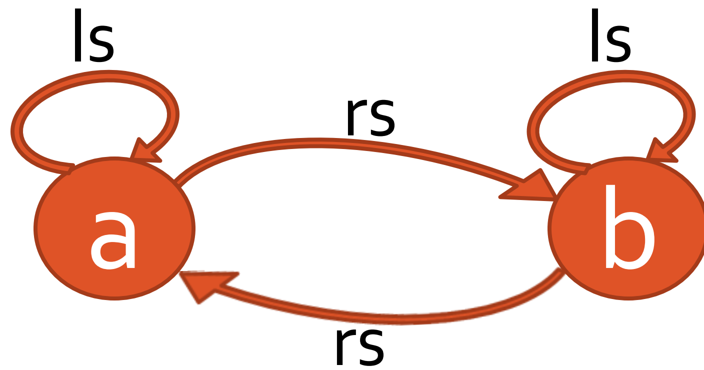
(g1 (list 'a
(λ().corp$_a$ ▪)
(λ().corp$_b$ ▪)))

Contextul lui letrec

(a	(λ().corp$_a$	▪))
(b	(λ().corp$_b$	▪))

```
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)



Contextul global

(g1 (list 'a
(λ().corp$_a$ ▪)
(λ().corp$_b$ ▪)))

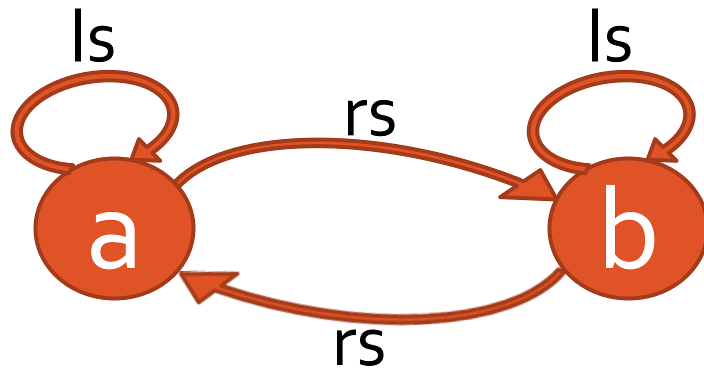
Contextul lui letrec

(a (λ().corp$_a$ ▪))
(b (λ().corp$_b$ ▪))

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

o nouă aplicare (a) (ex: `(ls g1)`) produce o reevaluare 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

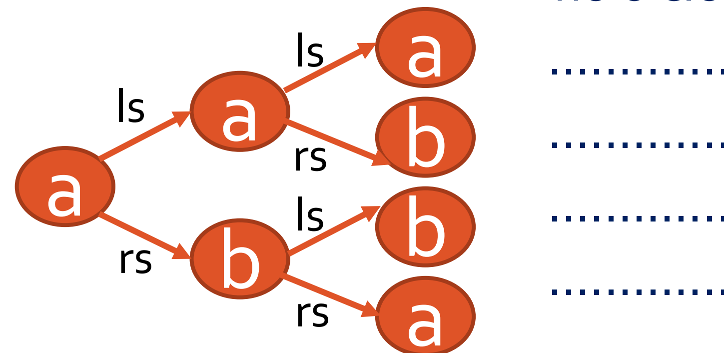
Implementare cu închideri funcționale

Avantaj

- Reprezintă graful infinit

Dezavantaje

- Ineficiență: evaluări repetate ale unor închideri deja evaluate
  (de fiecare dată când accesăm vecinii unui nod)
- Probleme „filozofice” de identitate: vecinul stâng al lui a este chiar a,
  nu o clonă a lui a (ca în figură)



………….

………….

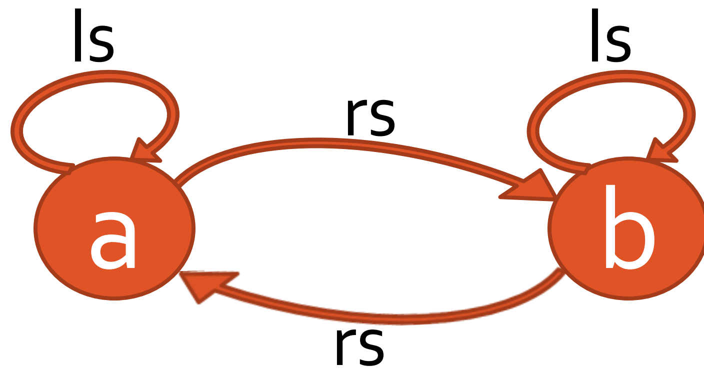
………….

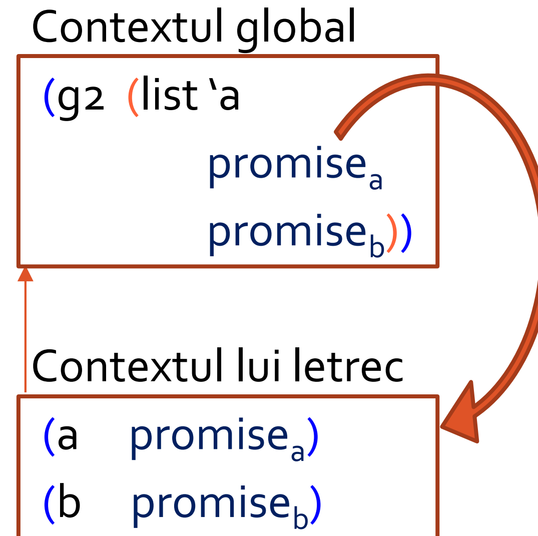
………….

Întârzierea evaluării – Cuprins

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

Exemplu – Un graf recursiv (infinit)



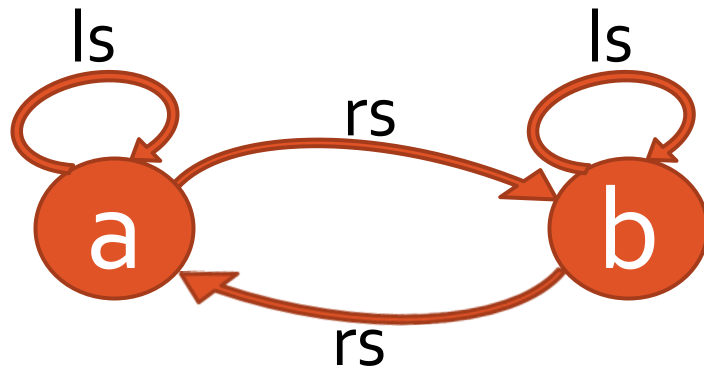


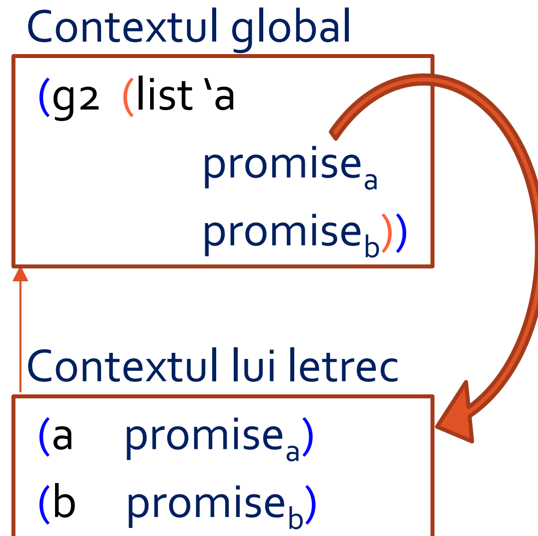

```
1.   (define g2
2.     (letrec ((a (delay (list 'a a b)))
3.              (b (delay (list 'b b a))))
4.       (force a)))
```

(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)))
```

o nouă forțare a lui promise$_a$ (ex: `(ls g2)`) întoarce **rezultatul salvat la prima forțare**;

expresia întârziată se evaluează o singură dată

Implementare cu promisiuni

Avantaje

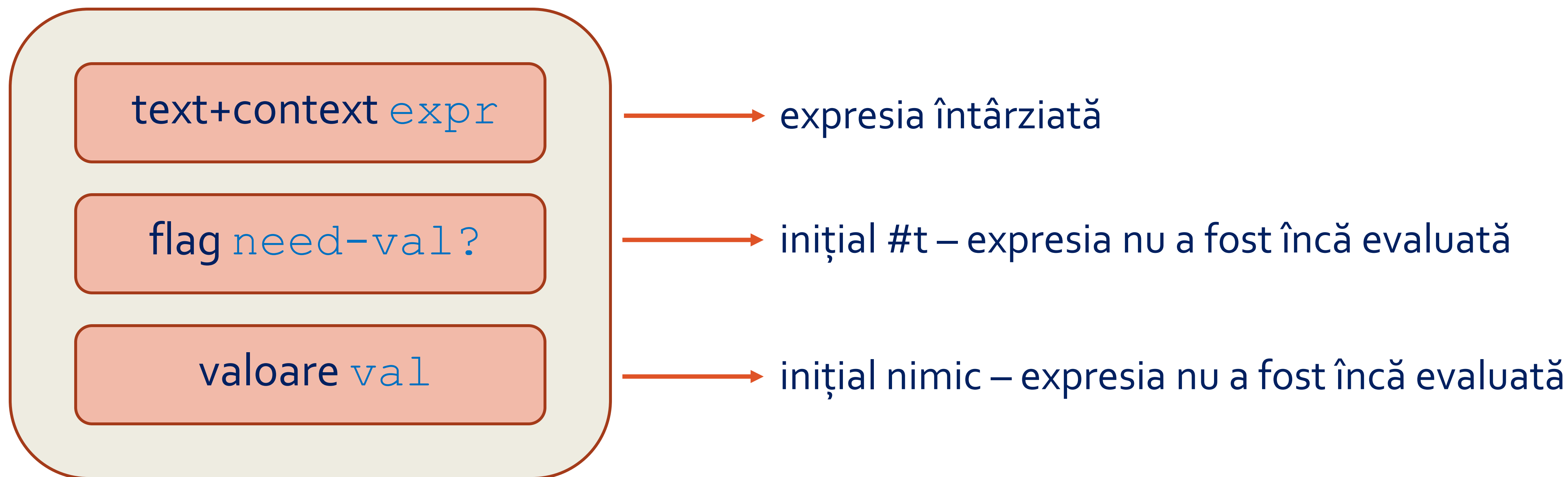
- Reprezintă graful infinit
- Eficiență
  - Ambele promisiuni își evaluează expresia o singură dată
  - La solicitări ulterioare, se ia valoarea expresiei din cache-ul promisiunii
- Identitate a obiectelor întoarse de evaluări distincte ale promisiunii
  - Închideri funcționale: evaluări distincte pot produce rezultate diferite (v. transparența referențială)
    - Rezultate potențial diferite ⇒ obiecte diferite
  - Promisiuni: evaluări distincte produc același rezultat (evaluat prima dată, apoi luat din cache)
    - Același rezultat ⇒ același obiect

Întârzierea evaluării – Cuprins

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

Funcția (nestrictă) `delay`

`(delay expr)` creează o promisiune, care, conceptual, arată astfel:



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 fără else este valid în limbajul Pretty Big)

Forțările ulterioare întorc direct `val`

Funcția **force**

`(force promise)` solicită valoarea expresiei întârziate în promisiune:

- Dacă expresia a fost deja evaluată (flag #f)
  întoarce valoare
- Altfel
  valoare ← evaluează expresie
  flag ← #f
  întoarce valoare

Memento

- Rezultatul primei forțări nu se schimbă niciodată
  (chiar dacă, din cauza unor efecte laterale,
  o nouă evaluare a expresiei întârziate ar produce un rezultat diferit)

Întârzierea evaluării – Cuprins

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

Închideri funcționale versus promisiuni

Criteriu	Închidere funcțională	Promisiune
Creare	(λ () expr) ⟶ f	(delay expr) ⟶ p
Declanșare	(f)	(force p)
Evaluare	Reevaluare la fiecare apel, evaluări diferite pot da rezultate diferite	O singură evaluare (+ memoizare), toate forțările întorc același rezultat
Scop	Capturarea variabilelor din punctul de definire	Întârzierea evaluării

Rezumat

Context computațional

Închidere funcțională

Promisiune

Avantaj promisiuni

(delay expr)

(force p)

Rezumat

Context computațional: (într-un punct P): mulțimea variabilelor vizibile în punctul P

Închidere funcțională

Promisiune

Avantaj promisiuni

(delay expr)

(force p)

Rezumat

Context computațional: (într-un punct P): mulțimea variabilelor vizibile în punctul P

Închidere funcțională: pereche (text funcție, context în punctul de definire a funcției)

Promisiune

Avantaj promisiuni

(delay expr)

(force p)

Rezumat

Context computațional: (într-un punct P): mulțimea variabilelor vizibile în punctul P

Închidere funcțională: pereche (text funcție, context în punctul de definire a funcției)

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

Avantaj promisiuni

(delay expr)

(force p)

Rezumat

Context computațional: (într-un punct P): mulțimea variabilelor vizibile în punctul P

Închidere funcțională: pereche (text funcție, context în punctul de definire a funcției)

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

Avantaj promisiuni: eficiență: evaluarea se produce doar la nevoie și o singură dată

(delay expr)

(force p)

Rezumat

Context computațional: (într-un punct P): mulțimea variabilelor vizibile în punctul P

Închidere funcțională: pereche (text funcție, context în punctul de definire a funcției)

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

Avantaj promisiuni: eficiență: evaluarea se produce doar la nevoie și o singură dată

(delay expr): creează o promisiune pe baza expresiei expr

(force p)

Rezumat

Context computațional: (într-un punct P): mulțimea variabilelor vizibile în punctul P

Închidere funcțională: pereche (text funcție, context în punctul de definire a funcției)

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

Avantaj promisiuni: eficiență: evaluarea se produce doar la nevoie și o singură dată

(delay expr): creează o promisiune pe baza expresiei expr

(force p): solicită valoarea expresiei întârziate în promisiunea p