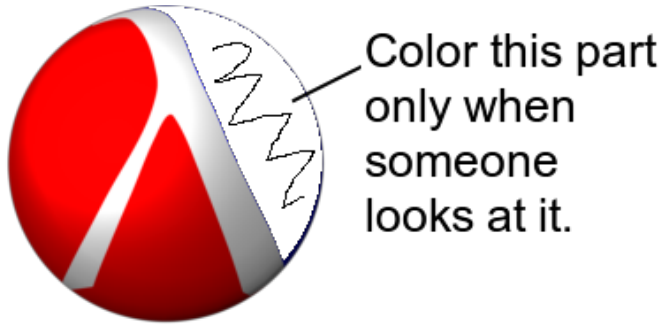
Paradigme de Programare

Ș.l. dr. ing. Andrei Olaru

Departamentul Calculatoare

2015 – 2016, semestrul 2

Cursul 5: Evaluare leneșă în Racket



(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113 127 131 137 139 149
151 157 163 167 173 179 181 191 193 197 199 211 223
227 229 233 239 241 251 257 263 269 271 277 281 283
293 307 311 313 317 331 337 347 349 353 359 367 373
379 383 389 397 401 409 419 421 431 433 439 443 449
457 461 463 467 479 487 491 499 503 509 521 523 541 . . .

Cursul 5: Evaluare lenesă în Racket

Întârzierea evaluării

Motivație
De ce? → Luăm un exemplu

Exemplu

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

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

Dar, evaluarea parametrului y al funcției să se facă numai o singură dată.

· Problema de rezolvat: evaluarea la cerere.

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 (and (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 (and (display "y ") y)))))
9
10 (test #f)
11 (test #t)
```

Output: `y 0 | y 30`

- Implementarea nu respectă specificația, deoarece ambii parametri sunt evaluați în momentul aplicării

Varianta 2

Încercare → `quote` & `eval`

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

Output: `0 | y undefined`

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

(

Paranteză!

Contexte computaţionale
Definiţie

+ | Context computaţional Contextul computaţional al unui punct P, dintr-un program, la momentul t, este mulţimea variabilelor ale căror domenii de vizibilitate îl conţin pe P, la momentul t.

- Legare statică $\rightarrow$ mulţimea variabilelor care îl conţin pe P în domeniul lexical de vizibilitate
- Legare dinamică $\rightarrow$ mulţimea variabilelor definite cel mai recent, la momentul t, şi referite din P

Contexte computaţionale
Exemplu

Ex | Exemplu Ce variabile locale conţine contextul computaţional al punctului *P*?

```
1 (lambda (x y)
2   (lambda (z)
3     (let ((x (car y)))
4       ; ..P..)))
```

Contexte computaţionale
Exemplu

Ex | Exemplu Ce variabile locale conţine contextul computaţional al punctului *P*?

```
1 (lambda (x y)
2   (lambda (z)
3     (let ((x (car y)))
4       ; ..P..)))
```

Închideri funcţionale
Definiţie

+ | Închidere funcţională: funcţie care îşi salvează contextul, pe care îl va folosi, în momentul aplicării, pentru evaluarea corpului.

· Notaţie: închiderea funcţiei f în contextul $C \rightarrow < f; C >$

Ex | Exemplu

$< \lambda x.z; \{z \leftarrow 2\} >$

)

închidem paranteza

Varianta 3

Încercare → închideri funcționale

```
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 () (and (display "y ") y))))))
10
11 (test #f)
12 (test #t)
```

Output:

Varianta 3
Încercare → închideri funcționale

```
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 () (and (display "y␣") y))))))
10
11 (test #f)
12 (test #t)
```

Output: `0 | y y 30`

- Comportament corect: `y` evaluat la cerere
- `x = #t` → `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 (and (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 (and (display "y ") y))))))
10
11 (test #f)
12 (test #t)
```

Output: `0 | y 30`

- Rezultat corect: `y` evaluat la cerere, o singură dată
  → evaluare lenesă

Promisiuni
Descriere

- Rezultatul încă neevaluat al unei expresii
- Valori de prim rang în limbaj
- `delay`
  - construiește o promisiune;
  - funcție nestrictă.
- `force`
  - forțează respectarea unei promisiuni, evaluând expresia doar la prima aplicare, și salvându-i valoarea;
  - începând cu a doua invocare, întoarce, direct, valoarea memorată.

Promisiuni
Proprietăți

- Salvarea contextului computațional al expresiei a cărei evaluare este întârziată și evaluarea ei ulterioară în acel context $\rightarrow$ asemănător cu închiderile funcționale.
- Salvarea rezultatului primei evaluări a expresiei
- Distingerea primei forțări de celelalte $\rightarrow$

Promisiuni
Proprietăți

- Salvarea contextului computațional al expresiei a cărei evaluare este întârziată și evaluarea ei ulterioară în acel context $\rightarrow$ asemănător cu închiderile funcționale.
- Salvarea rezultatului primei evaluări a expresiei
- Distingerea primei forțări de celelalte $\rightarrow$ efect lateral.

Evaluare întârziată

Abstractizare a implementării cu promisiuni

Ex Continuare a exemplului cu funcția `prod`

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

- utilizarea nu depinde de implementare (am definit funcțiile `pack` și `unpack` care abstractizează implementarea concretă a evaluării întârziate.

Evaluare întârziată

Abstractizare a implementării cu închideri

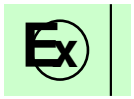
Ex Continuare a exemplului cu funcția `prod`

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

- utilizarea nu depinde de implementare (același cod ca anterior, altă implementare a funcționalității de evaluare întârziată).

Fluxuri

Motivație
Luăm un exemplu



Determinați suma numerelor pare[1] 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
10 (define even-sum-lists ; varianta 2
11   (lambda (a b)
12     (foldl + 0 (filter even? (interval a b)))))
```

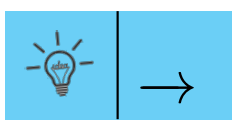
[1] stă pentru o verificare potențial mai complexă, e.g. numere prime

Motivație
Observații

- Varianta 1 – iterativă (d.p.d.v. proces):
  - eficientă, datorită spațiului suplimentar constant;
  - ne-elegantă $\rightarrow$ trebuie să implementăm generarea numerelor.
- Varianta 2 – folosește liste:
  - ineficientă, datorită spațiului posibil mare, ocupat la un moment dat – toate numerele din intervalul $[a, b]$.
  - elegantă și concisă;
- Cum îmbinăm avantajele celor 2 abordări? Putem stoca procesul fără a stoca rezultatul procesului?

Motivație
Observații

- Varianta 1 – iterativă (d.p.d.v. proces):
  - eficientă, datorită spațiului suplimentar constant;
  - ne-elegantă $\rightarrow$ trebuie să implementăm generarea numerelor.
- Varianta 2 – folosește liste:
  - ineficientă, datorită spațiului posibil mare, ocupat la un moment dat – toate numerele din intervalul $[a, b]$.
  - elegantă și concisă;
- Cum îmbinăm avantajele celor 2 abordări? Putem stoca procesul fără a stoca rezultatul procesului?



$\rightarrow$ Fluxuri

Fluxuri
Caracteristici

- Secvențe construite parțial, extinse la cerere, ce creează iluzia completitudinii structurii;
- Îmbinarea eleganței manipulării listelor cu eficiența calculului incremental;
- Bariera de abstractizare:
  - componentele listelor evaluate la construcție (`cons`)
  - componentele fluxurilor evaluate la selecție (`cdr`)
- Construcție și utilizare:
  - separate la nivel conceptual $\rightarrow$ modularitate;
  - întrepătrunse la nivel de proces (utilizarea necesită construcția concretă).

Fluxuri

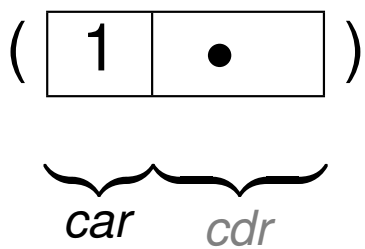
Intuitiv

- o listă este o pereche;
- explorarea listei se face prin operatorii `car` – primul element – și `cdr` – restul listei;
- am dori să generăm cdr algoritmic, dar la cererere.

Fluxuri

Intuitiv

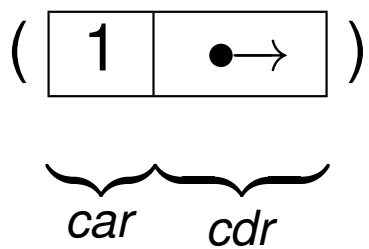
- o listă este o pereche;
- explorarea listei se face prin operatorii `car` – primul element – și `cdr` – restul listei;
- am dori să generăm cdr algoritmic, dar la cererere.



Fluxuri

Intuitiv

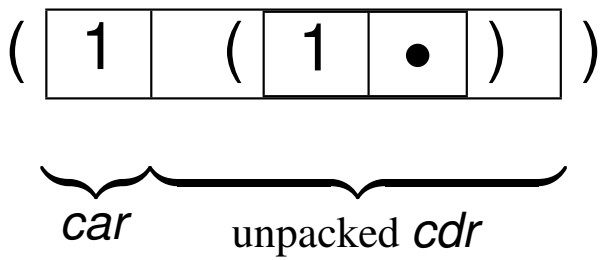
- o listă este o pereche;
- explorarea listei se face prin operatorii `car` – primul element – și `cdr` – restul listei;
- am dori să generăm cdr algoritmic, dar la cererere.



Fluxuri

Intuitiv

- o listă este o pereche;
- explorarea listei se face prin operatorii `car` – primul element – și `cdr` – restul listei;
- am dori să generăm cdr algoritmic, dar la cererere.



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-nil '())
10
11 (define stream-null? null?)
```

Fluxuri

Operatori: `take` și `drop`

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

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

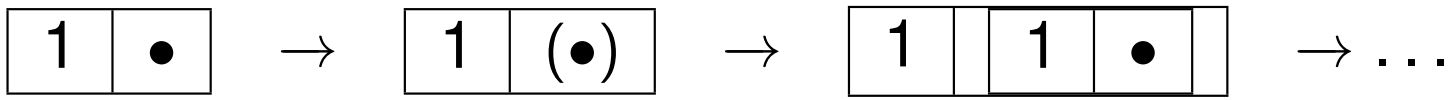
```
1 (define ones (stream-cons 1 ones))
2 (stream-take 5 ones) ; (1 1 1 1 1)
```

- Definiție cu promisiuni:

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

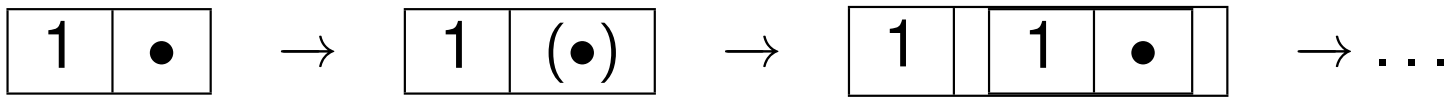
Fluxuri – Exemple
Flux de numere 1 – discuție

- Ca proces:

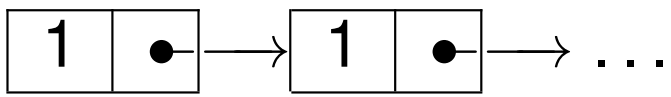


Fluxuri – Exemple
Flux de numere 1 – discuție

- Ca proces:

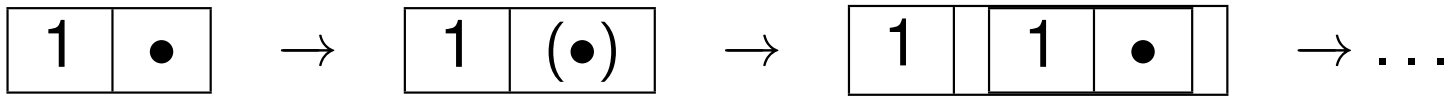


- Structural:

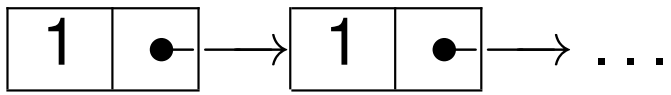


Fluxuri – Exemple
Flux de numere 1 – discuție

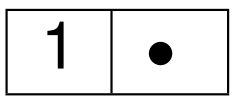
- Ca proces:



- Structural:

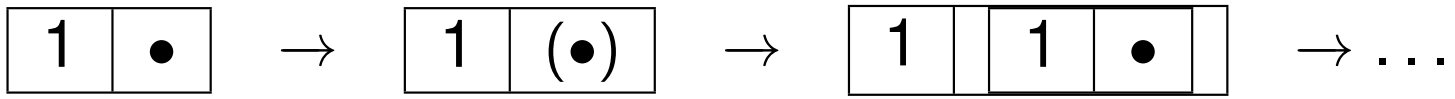


- Extinderea se realizează în spațiu constant:

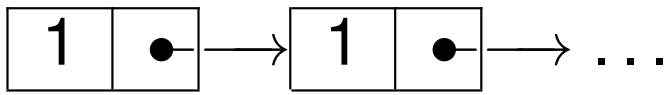


Fluxuri – Exemple
Flux de numere 1 – discuție

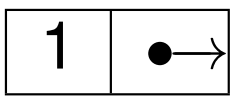
- Ca proces:



- Structural:

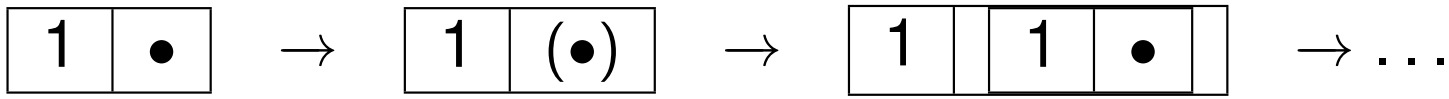


- Extinderea se realizează în spațiu constant:

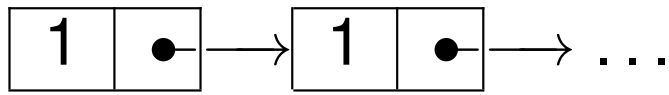


Fluxuri – Exemple
Flux de numere 1 – discuție

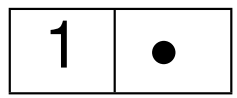
- Ca proces:



- Structural:

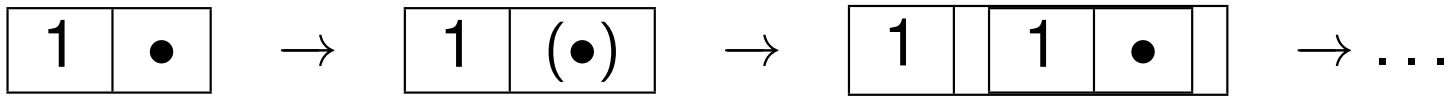


- Extinderea se realizează în spațiu constant:

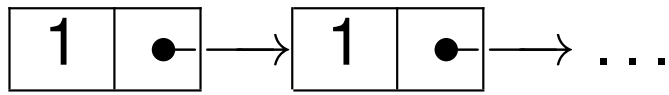


Fluxuri – Exemple
Flux de numere 1 – discuție

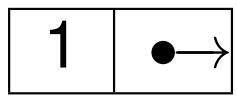
- Ca proces:



- Structural:

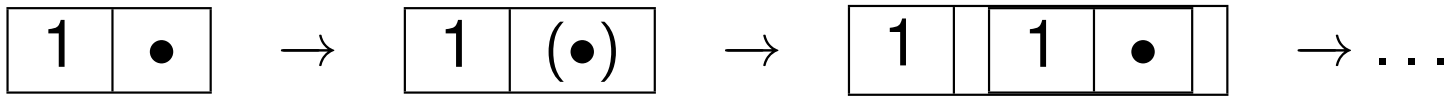


- Extinderea se realizează în spațiu constant:

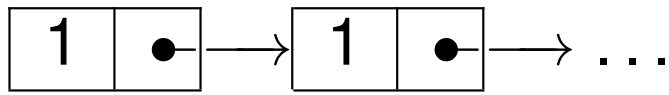


Fluxuri – Exemple
Flux de numere 1 – discuție

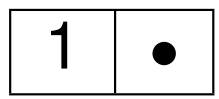
- Ca proces:



- Structural:



- Extinderea se realizează în spațiu constant:



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))
```

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

- Atenție:
  - Închideri: multiple parcurgeri ale fluxului determină reevaluarea porțiunilor deja explorate.
  - Promisiuni: parcurgerea fluxului determină evaluarea dincolo de porțiunile deja explorate.

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 inițial aparține fluxului numerelor prime.
- Restul fluxului generat se obține
  - eliminând multiplii elementului curent din fluxul inițial;
  - 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)))
```

Căutare lenesă în spațiul stărilor

Spațiul stărilor unei probleme

+ | Spațiul stărilor unei probleme Mulțimea configurațiilor valide din universul problemei.

Ex | Exemplu

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

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

Specificarea unei probleme
Aplicație pe Pal_n

- Starea inițială: șirul vid
- Operatorii de generare a stărilor succesor ale unei stări: inserarea unui caracter la începutul unui șir dat
- Operatorul de verificare a proprietății de *scop* a unei stări: palindrom

Căutare în spațiul stărilor

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

Căutare în lățime
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 soluții
- Cum le obținem pe celelalte, mai ales dacă spațiul e infinit?

Căutare în lățime

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-nil))
11 )))
```

Căutare în lățime
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 spațiului și identificarea stărilor *scop*.
- Nivel scăzut, al instrucțiunilor: întrepătrunderea celor două aspecte.
- Aplicații:
  - Palindroame
  - Problema reginelor

Sfârșitul cursului 5
Elemente esențiale

- Evaluare întârziată $\rightarrow$ variante de implementare
- Fluxuri $\rightarrow$ implementare și utilizări
- Căutare într-un spațiu de stări infinit

+ Dați feedback la acest curs aici:

[http://goo.gl/forms/SjDsW06v5J]