

# PARADIGME DE PROGRAMARE

## Curs 4

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

1

## Transparență referențială – Cuprins

- Efecte laterale
- Transparență referențială

2

2

## 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

3

3

## 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";

```

4

4

## 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";

```

5

5

## 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;

```

6

6

## 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;

```

7

7

## Efecte laterale

### Consecințe

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

- Risc ridicat de **bug-uri**

8

8

## Transparență referențială – Cuprins

- Efecte laterale
- **Transparență referențială**

9

9

## 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)))`

10

10

## 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)))`
- transparente referențial  
opacă referențial  
opacă referențial

11

11

## 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**

12

12

## Legare statică / dinamică – Cuprins

- Variabile
- Domeniu de vizibilitate
- Tipuri de legare a variabilelor
- Expresii pentru legare statică
- Expresii pentru legare dinamică

13

13

## Variabile

**Variabilă** = pereche identificador-valoare

### Caracteristici

- Domeniu de vizibilitate (zona de program în care valoarea poate fi accesată prin identificador)
- Durată de viață

### Observație

- În Racket, tipul este asociat valorilor, nu variabilelor

14

14

## Legare statică / dinamică – Cuprins

- Variabile
- Domeniu de vizibilitate
- Tipuri de legare a variabilelor
- Expresii pentru legare statică
- Expresii pentru legare dinamică

15

15

## Domeniu de vizibilitate al unei variabile

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

### 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)))

16

16

## 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

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

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

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

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

17

17

## 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

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

$(x \lambda x.(\lambda x.y \lambda y.(x z)))$  – acest x nu mai este vizibil nicăieri în această zonă de program

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

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

18

18

## 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

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

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

$(x \lambda x.(\lambda x.y \lambda y.(x z)))$  – practic: zona din corp în care aparițiile lui x sunt libere

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

19

19

## 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

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

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

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

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

20

20

## 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)

```

21

21

## 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)

```

22

22

## 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 **obscurăază** legarea de la (define n 5).

23

23

## 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)

```

24

24

## Legare statică / dinamică – Cuprins

- Variabile
- Domeniu de vizibilitate
- Tipuri de legare a variabilelor
- Expresii pentru legare statică
- Expresii pentru legare dinamică

25

25

## 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)

26

26

## 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.

27

27

## 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)
```

;; 25 ← Când se intră în corpul lui f se evaluează g la definiția cea mai recentă

;; 125 ← Când se intră în corpul lui f se evaluează g la definiția cea mai recentă

28

28

## Legare statică / dinamică – Cuprins

- Variabile
- Domeniu de vizibilitate
- Tipuri de legare a variabilelor
- Expresii pentru legare statică
- Expresii pentru legare dinamică

29

29

## Expresii pentru legare statică

- lambda
  - let
  - let\*
  - letrec
  - named let
- } exemplificate la calculator

30

30

## Construcția lambda

```
( (lambda (var1 var2 ... varm)
  expr1
  expr2
  ...
  exprn) arg1 arg2 ... argn)
```

### La aplicarea λ-expresiei pe argumente

- Se evaluează în ordine aleatoare argumentele  $arg_1, arg_2, \dots, arg_m$  (evaluare aplicativă)
- Se realizează legările  $var_k \leftarrow \text{valoare}(arg_k)$
- 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 ( $expr_1, expr_2, \dots, expr_n$ )
- Rezultatul este valoarea lui  $expr_n$  (valorile lui  $expr_1, expr_2, \dots, expr_{n-1}$  se pierd)

31

31

## Construcția let

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

### La evaluare

- Se evaluează exact ca  
 $((\text{lambda } (var_1 \text{ } var_2 \dots var_m) \text{ } expr_1 \text{ } expr_2 \dots expr_n) e_1 e_2 \dots e_m)$
- 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  $expr_1, expr_2, \dots, expr_{n-1}$  se pierd)

32

32



## Construcția **let\***

```
(let* ((var1 e1) (var2 e2) (var3 e3) ... (varm em))
  expr1
  expr2
  ...
  exprn)
```

### La evaluare

- Se realizează **în ordine (stânga→dreapta)** legările  $var_k \leftarrow \text{valoare}(e_k)$
- 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 ( $expr_1, expr_2, \dots, expr_n$ )
- Rezultatul este valoarea lui  $expr_n$  (valorile lui  $expr_1, expr_2, \dots, expr_{n-1}$  se pierd)

33

33

## Construcția **letrec**

```
(letrec ((var1 e1) (var2 e2) (var3 e3) ... (varm em))
  expr1
  expr2
  ...
  exprn)
```

### La evaluare

- Se realizează **în ordine (stânga→dreapta)** legările  $var_k \leftarrow \text{valoare}(e_k)$
- 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 ( $expr_1, expr_2, \dots, expr_n$ )
- Rezultatul este valoarea lui  $expr_n$  (valorile lui  $expr_1, expr_2, \dots, expr_{n-1}$  se pierd)

34

34

## 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 parametri  $var_1, var_2, \dots, var_m$ , și se și aplică funcția pe argumentele  $e_1, e_2, \dots, e_m$
- 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

35

35

## 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))

36

36

## Legare statică / dinamică – Cuprins

- Variabile
- Domeniu de vizibilitate
- Tipuri de legare a variabilelor
- Expresii pentru legare statică
- Expresii pentru legare dinamică

37

37

## 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`

38

38

## 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**

39

39

## **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 ajutoare)

### **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

40

40

## Modelul contextual de evaluare – Cuprins

- Context computațional
- Închideri funcționale

41

41

## 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)

42

42

## Context computațional – Exemplu

<pre> 1. (define a 1) 2. (define (f x) 3.   (+ x 4.     (let ((x 5)) 5.       (* a x)))) 6. (f 2) 7. (define a 2) 8. (f 2) </pre>	<p>Înainte de apelul de la linia 6:</p> <p>În timpul apelului de la linia 6:</p> <p>După apelul de la linia 8:</p>	
<pre> ((a 1)) ((a 1) (x 5)) </pre>	<pre> ((a 1) (x 2)) ((a 1) (x 5)) </pre>	<pre> ((a 2)) ((a 2) (x 5)) </pre>

**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.

43

43

## Modelul contextual de evaluare – Cuprins

- Context computațional
- Închideri funcționale

44

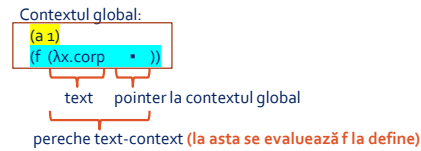
44

## 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))))
```



### 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)

45

45

## 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) ← dispăre odată cu terminarea apelului
```

Context local adăugat de expresia let:

```
(x 5) ← dispăre 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.

46

46

## 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) ← dispăre odată cu terminarea apelului
;; 24
```

47

47

## 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) ← dispăre odată cu terminarea apelului
```

48

48

## 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) ← dispăre odată cu terminarea apelului~~

49

49

## 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) ← dispăre odată cu terminarea apelului~~

Context local adăugat la apelul (g 4):  
 (n 4) ← dispăre 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

```

50

50

## 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

```

51

51

## 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**.

52

52

## Rezumat

- Efecte laterale
- Funcții pure
- Transparență referențială
- Domeniu de vizibilitate
- Legare statică
- Legare dinamică
- Expresii de legare statică / dinamică
- Context computațional
- Închidere funcțională

53

53

## 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

54

54

## 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

55

55

## 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

56

56

## 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

57

57

## 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

58

58

## 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

59

59

## 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

60

60

## 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

**Transparentă 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

61

61

## 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

**Transparentă 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

62

62