

Paradigme de Programare

Mihnea Muraru

`mihnea.muraru@upb.ro`

2022–2023, semestrul 2



Partea I

Introducere



Cuprins

Organizare

Obiective

Exemplu introductiv

Paradigme și limbaje



Cuprins

Organizare

Obiective

Exemplu introductiv

Paradigme și limbaje



Notare

- ▶ Laborator: 1p
- ▶ Teste grilă: 1p (0,3 săptămânale + 0,7 final)
- ▶ Teme: 4p ($3 \times 1.33p$)
- ▶ Examen (*open-book*): 4p
- ▶ Bonus-uri la testele săptămânale, teme și examen

Regulament

Vă rugăm să citiți regulamentul cu atenție!

<https://ocw.cs.pub.ro/courses/pp/23/regulament>



Desfășurarea cursului

- ▶ Recapitularea cursului anterior
+ exercițiu similar subiectelor de examen
- ▶ Predare
- ▶ Feedback despre cursul curent (de acasă)
- ▶ Legătură strânsă între curs, laborator
și etapele temelor!



Cuprins

Organizare

Obiective

Exemplu introductiv

Paradigme și limbaje



Ce vom studia?

1. Modele de calculabilitate:
2. Paradigme de programare:
3. Limbaje de programare:



Ce vom studia?

1. Modele de calculabilitate:

Diverse perspective conceptuale asupra noțiunii de calculabilitate efectivă

2. Paradigme de programare:

3. Limbaje de programare:



Ce vom studia?

1. Modele de calculabilitate:

Diverse perspective conceptuale asupra noțiunii de calculabilitate efectivă

2. Paradigme de programare:

Influența perspectivei alese asupra procesului de modelare și rezolvare a problemelor

3. Limbaje de programare:



Ce vom studia?

1. Modele de calculabilitate:

Diverse perspective conceptuale asupra noțiunii de calculabilitate efectivă

2. Paradigme de programare:

Influența perspectivei alese asupra procesului de modelare și rezolvare a problemelor

3. Limbaje de programare:

Mecanisme expresive, aferente paradigmelor, cu accent pe aspectul comparativ



De ce?

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

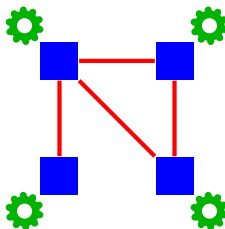
Edsger Dijkstra,
How do we tell truths that might hurt



Descompunerea problemelor

Controlul complexității: descompunere și interfațare

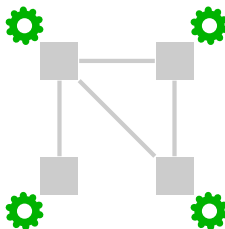
Descompunere	Accent pe	Rezultat



Descompunerea problemelor

Controlul complexității: descompunere și interfațare

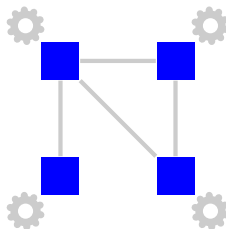
Descompunere	Accent pe	Rezultat
Procedurală	Acțiuni	Proceduri



Descompunerea problemelor

Controlul complexității: descompunere și interfațare

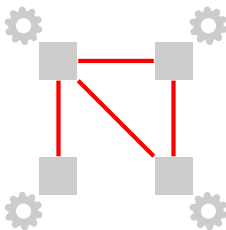
Descompunere	Accent pe	Rezultat
Procedurală	Acțiuni	Proceduri
Orientată obiect	Entități	Clase și obiecte



Descompunerea problemelor

Controlul complexității: descompunere și interfațare

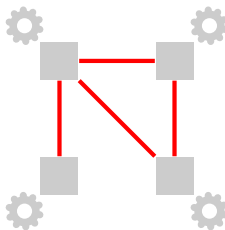
Descompunere	Accent pe	Rezultat
Procedurală	Acțiuni	Proceduri
Orientată obiect	Entități	Clase și obiecte
Funcțională	Relații	Funcții în sens matematic



Descompunerea problemelor

Controlul complexității: descompunere și interfațare

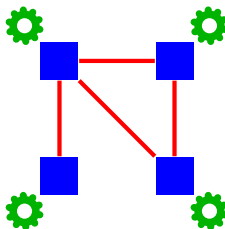
Descompunere	Accent pe	Rezultat
Procedurală	Acțiuni	Proceduri
Orientată obiect	Entități	Clase și obiecte
Funcțională	Relații	Funcții în sens matematic
Logică	Relații	Predicate și propoziții



Descompunerea problemelor

Controlul complexității: descompunere și interfațare

Descompunere	Accent pe	Rezultat
Procedurală	Acțiuni	Proceduri
Orientată obiect	Entități	Clase și obiecte
Funcțională	Relații	Funcții în sens matematic
Logică	Relații	Predicate și propoziții



De ce? (cont.)

- ▶ Lărgirea spectrului de **abordare** a problemelor



De ce? (cont.)

- ▶ Lărgirea spectrului de **abordare** a problemelor
- ▶ Identificarea perspectivei ce permite modelarea **simplă** a unei probleme; alegerea limbajului adecvat



De ce? (cont.)

- ▶ Lărgirea spectrului de **abordare** a problemelor
- ▶ Identificarea perspectivei ce permite modelarea **simplă** a unei probleme; alegerea limbajului adecvat
- ▶ **Exploatarea** mecanismelor oferite de limbajele de programare (v. Dijkstra!)

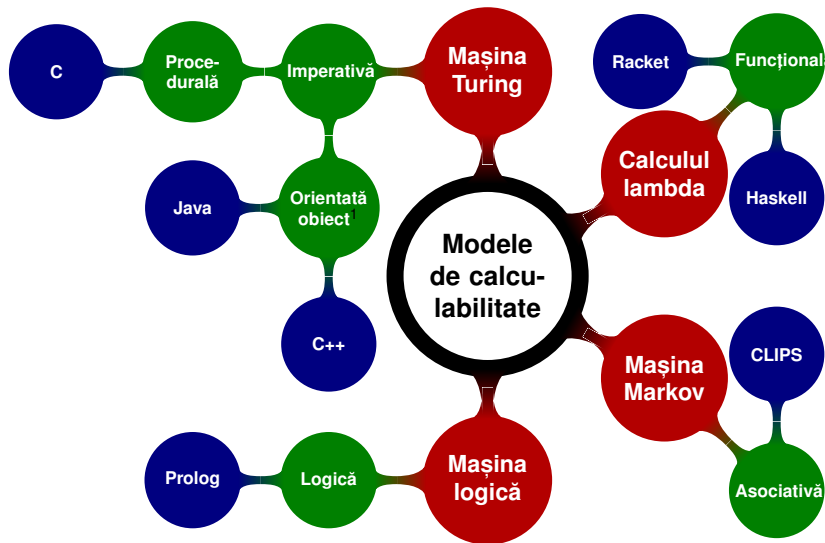


De ce? (cont.)

- ▶ Lărgirea spectrului de **abordare** a problemelor
- ▶ Identificarea perspectivei ce permite modelarea **simplă** a unei probleme; alegerea limbajului adecvat
- ▶ **Exploatarea** mecanismelor oferite de limbajele de programare (v. Dijkstra!)
- ▶ Sporirea capacității de **învățare** a noi limbaje și de **adaptare** la particularitățile și diferențele dintre acestea



Modele, paradigme, limbaje



Modele

Paradigme

Limbaje

¹ Original imperativă, dar se poate combina chiar cu abordarea funcțională



Limitele calculabilității

- ▶ **Teza Church-Turing:**
efectiv calculabil $\equiv$ Turing calculabil



Limitele calculabilității

- ▶ **Teza Church-Turing:**
efectiv calculabil $\equiv$ Turing calculabil
- ▶ **Echivalența** celorlalte modele de calculabilitate,
și a multor alora, cu Mașina Turing



Limitele calculabilității

- ▶ **Teza Church-Turing:**
efectiv calculabil $\equiv$ Turing calculabil
- ▶ **Echivalența** celorlalte modele de calculabilitate,
și a multor altora, cu Mașina Turing
- ▶ Există vreun model **superior** ca forță de calcul?



Cuprins

Organizare

Obiective

Exemplu introductiv

Paradigme și limbaje



O primă problemă

Example 3.1.

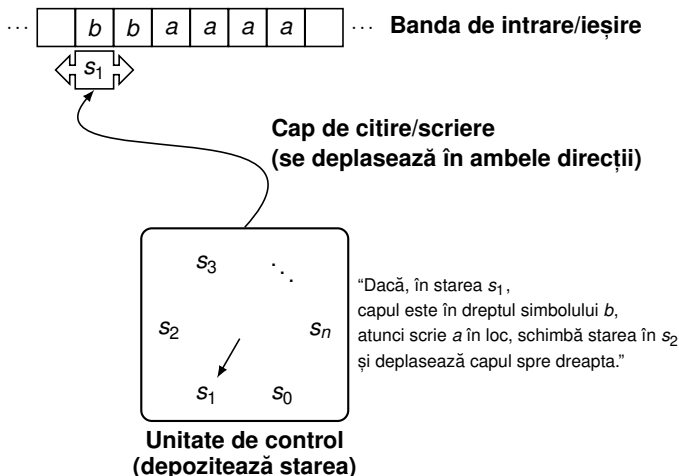
Să se determine elementul minim dintr-un vector.



Abordare imperativă

Modelul

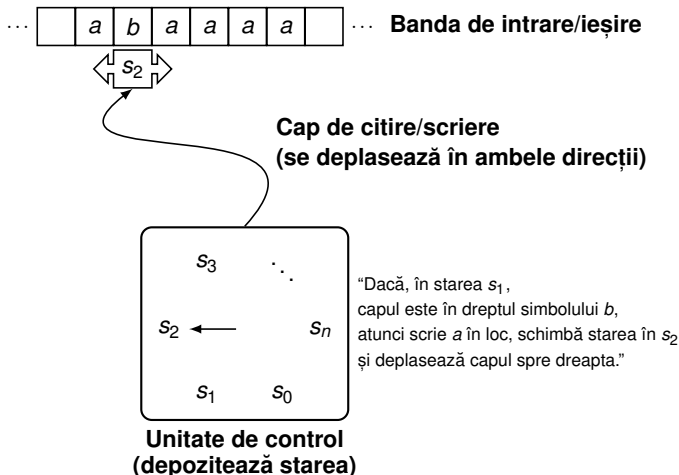
Mașina Turing



Abordare imperativă

Modelul

Mașina Turing



Abordare imperativă (procedurală)

Limbajul

```
1: procedure MINLIST( $L, n$ )
2:    $min \leftarrow L[1]$ 
3:    $i \leftarrow 2$ 
4:   while  $i \leq n$  do
5:     if  $L[i] < min$  then
6:        $min \leftarrow L[i]$ 
7:     end if
8:      $i \leftarrow i + 1$ 
9:   end while
10:  return  $min$ 
11: end procedure
```



Abordare imperativă

Paradigma

- Orientare spre **acțiuni** și **efectele** acestora



Abordare imperativă

Paradigma

- ▶ Orientare spre **acțiuni** și **efectele** acestora
- ▶ **“Cum”** se obține soluția, pașii de urmat



Abordare imperativă

Paradigma

- ▶ Orientare spre **acțiuni** și **efectele** acestora
- ▶ “**Cum**” se obține soluția, pașii de urmat
- ▶ **Atribuirea** ca operație fundamentală



Abordare imperativă

Paradigma

- ▶ Orientare spre **acțiuni** și **efectele** acestora
- ▶ “**Cum**” se obține soluția, pașii de urmat
- ▶ **Atribuirea** ca operație fundamentală
- ▶ Programe cu **stare**



Abordare imperativă

Paradigma

- ▶ Orientare spre **acțiuni** și **efectele** acestora
- ▶ “**Cum**” se obține soluția, pașii de urmat
- ▶ **Atribuirea** ca operație fundamentală
- ▶ Programe cu **stare**
- ▶ **Secvențierea** instrucțiunilor



Abordare funcțională

Modelul

Calculul lambda

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

“Pentru a aplica funcția $\lambda x.x$



Abordare funcțională

Modelul

Calculul lambda

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

“Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual,
 y ,

Abordare funcțională

Modelul

Calculul lambda

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

“Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, y , se indentifică parametrul formal, x ,

Abordare funcțională

Modelul

Calculul lambda

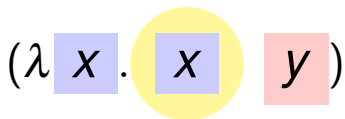
$$(\lambda x. x \ y)$$

“Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, y , se indentifică parametrul formal, x , în corpul funcției, x ,

Abordare funcțională

Modelul

Calculul lambda

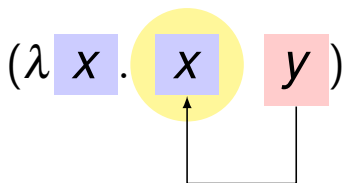
$$(\lambda x. x \quad x \quad y)$$
The diagram shows the lambda expression $(\lambda x. x \quad x \quad y)$. The lambda symbol λ is black. The first x is inside a light blue square. The dot $.$ is black. The second x is inside a light blue square and is also enclosed within a yellow circle. The third x is inside a light blue square. The y is inside a light red square. The parentheses are black.

“Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, y , se indentifică parametrul formal, x , în corpul funcției, x , iar aparițiile primului, x (singura),

Abordare funcțională

Modelul

Calculul lambda

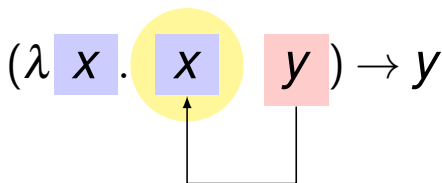


“Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, y , se indentifică parametrul formal, x , în corpul funcției, x , iar aparițiile primului, x (singura), se **substituie** cu parametrul actual,

Abordare funcțională

Modelul

Calculul lambda



“Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, y , se identifică parametrul formal, x , în corpul funcției, x , iar aparițiile primului, x (singura), se **substituie** cu parametrul actual, obținându-se rezultatul unui pas de evaluare.”

Abordare funcțională

Limbajul

► Racket (2 variante):

```
1  (define (minList1 L)
2    (if (= (length L) 1) (car L)
3        (min (car L) (minList1 (cdr L)))))
4
5  (define (minList2 L)
6    (foldl min (car L) (cdr L)))
```



Abordare funcțională

Limbajul

► Racket (2 variante):

```
1  (define (minList1 L)
2    (if (= (length L) 1) (car L)
3        (min (car L) (minList1 (cdr L)))))
4
5  (define (minList2 L)
6    (foldl min (car L) (cdr L)))
```

► Haskell (aceleași 2 variante):

```
1  minList1 [h]      = h
2  minList1 (h : t) = min h (minList1 t)
3
4  minList2 (h : t) = foldl min h t
```



Abordare funcțională

Paradigma

- ▶ **Funcții** matematice, care transformă intrările în ieșiri



Abordare funcțională

Paradigma

- ▶ **Funcții** matematice, care transformă intrările în ieșiri
- ▶ **Absența** atribuirilor și a stării



Abordare funcțională

Paradigma

- ▶ **Funcții** matematice, care transformă intrările în ieșiri
- ▶ **Absența** atribuirilor și a stării
- ▶ Funcții ca **valori** de prim rang (e.g., ca parametri ai altor funcții)



Abordare funcțională

Paradigma

- ▶ **Funcții** matematice, care transformă intrările în ieșiri
- ▶ **Absența** atribuirilor și a stării
- ▶ Funcții ca **valori** de prim rang (e.g., ca parametri ai altor funcții)
- ▶ **Recursivitate**, în locul iterației



Abordare funcțională

Paradigma

- ▶ **Funcții** matematice, care transformă intrările în ieșiri
- ▶ **Absența** atribuirilor și a stării
- ▶ Funcții ca **valori** de prim rang (e.g., ca parametri ai altor funcții)
- ▶ **Recursivitate**, în locul iterației
- ▶ **Compunere** de funcții, în locul secvențierii instrucțiunilor



Abordare funcțională

Paradigma

- ▶ **Funcții** matematice, care transformă intrările în ieșiri
- ▶ **Absența** atribuirilor și a stării
- ▶ Funcții ca **valori** de prim rang (e.g., ca parametri ai altor funcții)
- ▶ **Recursivitate**, în locul iterației
- ▶ **Compunere** de funcții, în locul secvențierii instrucțiunilor
- ▶ **Diminuarea** importanței ordinii de evaluare



Abordare funcțională

Paradigma

- ▶ **Funcții** matematice, care transformă intrările în ieșiri
- ▶ **Absența** atribuirilor și a stării
- ▶ Funcții ca **valori** de prim rang (e.g., ca parametri ai altor funcții)
- ▶ **Recursivitate**, în locul iterației
- ▶ **Compunere** de funcții, în locul secvențierii instrucțiunilor
- ▶ **Diminuarea** importanței ordinii de evaluare
- ▶ Funcții de ordin **superior** (i.e. care iau alte funcții ca parametru, e.g., `foldl`)



Abordare logică

Modelul

Logica cu predicate de ordin I

$muritor(Socrate) \wedge om(Platon) \wedge \forall x. om(x) \Rightarrow muritor(x)$



Abordare logică

Modelul

Logica cu predicate de ordin I

$muritor(Socrate) \wedge om(Platon) \wedge \forall x. om(x) \Rightarrow muritor(x)$

“La ce se poate lega variabila y
astfel încât $muritor(y)$ să fie **satisfăcută**?”



Abordare logică

Modelul

Logica cu predicate de ordin I

$muritor(Socrate) \wedge om(Platon) \wedge \forall x. om(x) \Rightarrow muritor(x)$

“La ce se poate lega variabila y
astfel încât $muritor(y)$ să fie **satisfăcută**?”

$y \leftarrow Socrate$ sau $y \leftarrow Platon$



Abordare logică

Limbajul

- ▶ Axiome:



Abordare logică

Limbajul

- ▶ Axiome:

1. $x \leq y \Rightarrow \min(x, y, x)$



Abordare logică

Limbajul

► Axiome:

1. $x \leq y \Rightarrow \min(x, y, x)$

2. $y < x \Rightarrow \min(x, y, y)$



Abordare logică

Limbajul

► Axiome:

1. $x \leq y \Rightarrow \text{min}(x, y, x)$

2. $y < x \Rightarrow \text{min}(x, y, y)$

3. $\text{minList}([m], m)$



Abordare logică

Limbajul

► Axiome:

1. $x \leq y \Rightarrow \min(x, y, x)$

2. $y < x \Rightarrow \min(x, y, y)$

3. $\minList([m], m)$

4. $\minList([y|t], n) \wedge \min(x, n, m) \Rightarrow \minList([x, y|t], m)$



Abordare logică

Limbajul

► Axiome:

1. $x \leq y \Rightarrow \text{min}(x, y, x)$
2. $y < x \Rightarrow \text{min}(x, y, y)$
3. $\text{minList}([m], m)$
4. $\text{minList}([y|t], n) \wedge \text{min}(x, n, m) \Rightarrow \text{minList}([x, y|t], m)$

► Prolog:

```
1  min(X, Y, X) :- X =< Y.
2  min(X, Y, Y) :- Y < X.
3
4  minList([M], M).
5  minList([X, Y | T], M) :-
6      minList([Y | T], N), min(X, N, M).
```



Abordare logică

Paradigma

- ▶ Formularea **proprietăților** logice ale obiectelor și soluției



Abordare logică

Paradigma

- ▶ Formularea **proprietăților** logice ale obiectelor și soluției
- ▶ Flux de control **implicit**, dirijat de date



Abordările funcțională și logică

Asemănări

- ▶ Formularea **proprietăților** soluției



Abordările funcțională și logică

Asemănări

- ▶ Formularea **proprietăților** soluției
- ▶ “**Ce**” trebuie obținut (vs. “cum” la imperativă)



Abordările funcțională și logică

Asemănări

- ▶ Formularea **proprietăților** soluției
- ▶ “**Ce**” trebuie obținut (vs. “cum” la imperativă)
- ▶ Se subsumează abordării **declarative**, opuse celei imperative



Cuprins

Organizare

Obiective

Exemplu introductiv

Paradigme și limbaje



Ce este o paradigmă de programare?

- ▶ Un set de convenții care dirijează maniera în care **gândim** programele



Ce este o paradigmă de programare?

- ▶ Un set de convenții care dirijează maniera în care **gândim** programele
- ▶ Ea dictează modul în care:



Ce este o paradigmă de programare?

- ▶ Un set de convenții care dirijează maniera în care **gândim** programele
- ▶ Ea dictează modul în care:
 - ▶ reprezentăm **datele**



Ce este o paradigmă de programare?

- ▶ Un set de convenții care dirijează maniera în care **gândim** programele
- ▶ Ea dictează modul în care:
 - ▶ reprezentăm **datele**
 - ▶ **operațiile** prelucrează datele respective



Ce este o paradigmă de programare?

- ▶ Un set de convenții care dirijează maniera în care **gândim** programele
- ▶ Ea dictează modul în care:
 - ▶ reprezentăm **datele**
 - ▶ **operațiile** prelucrează datele respective
- ▶ Abordările anterioare reprezintă paradigme de programare (procedurală, funcțională, logică)



Accepții asupra limbajelor

- ▶ Modalitate de exprimare a **instrucțiunilor** pe care calculatorul le execută



Accepții asupra limbajelor

- ▶ Modalitate de exprimare a **instrucțiunilor** pe care calculatorul le execută
- ▶ Mai important, modalitate de exprimare a unui mod de **gândire**



Accepții asupra limbajelor

*... “computer science” is not a science and [...] its significance has little to do with computers. The computer revolution is a revolution in the way we **think** and in the way we **express** what we think.*

Harold Abelson et al.,
Structure and Interpretation of Computer Programs



Câteva trăsături

- ▶ **Tipare**
 - ▶ Statică/ dinamică
 - ▶ Tare/ slabă



Câteva trăsături

- ▶ **Tipare**
 - ▶ Statică/ dinamică
 - ▶ Tare/ slabă
- ▶ **Ordinea de evaluare** a parametrilor funcțiilor
 - ▶ Aplicativă
 - ▶ Normală



Câteva trăsături

- ▶ **Tipare**
 - ▶ Statică/ dinamică
 - ▶ Tare/ slabă
- ▶ **Ordinea de evaluare** a parametrilor funcțiilor
 - ▶ Aplicativă
 - ▶ Normală
- ▶ **Legarea variabilelor**
 - ▶ Statică
 - ▶ Dinamică



Importanța cunoașterii
paradigmelor și limbajelor de programare,
în scopul identificării celor **convenabile**
pentru modelarea unei probleme particulare

Partea II

Limbajul Racket



Cuprins

Expresii și evaluare

Liste și perechi

Tipare

Omoiconicitate și metaprogramare



Cuprins

Expresii și evaluare

Liste și perechi

Tipare

Omoiconicitate și metaprogramare



Expresii

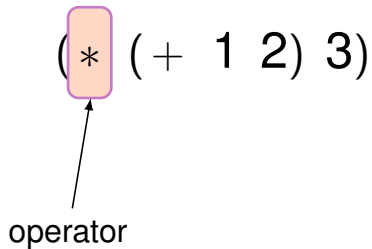
$(* (+ 1 2) 3)$



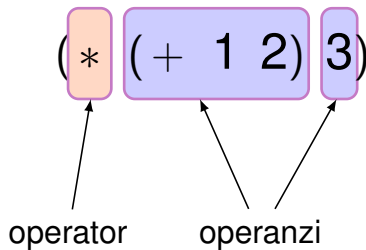
Expresii

$(* (+ 1 2) 3)$

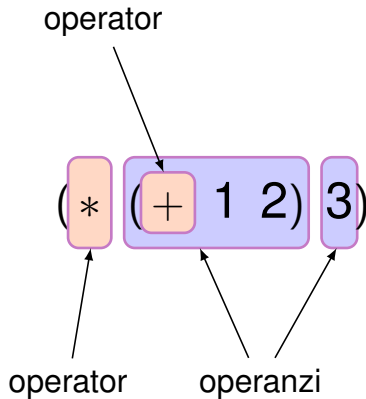
operator

The diagram shows the expression $(* (+ 1 2) 3)$. The asterisk operator $*$ is highlighted with a pink rounded rectangle. An arrow points from the word "operator" below to the asterisk.

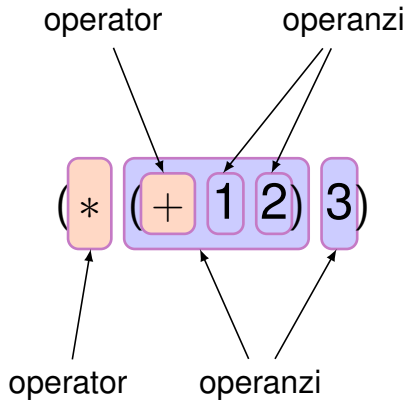
Expresii



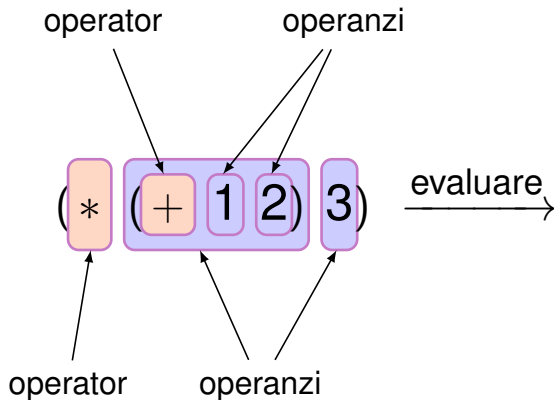
Expresii



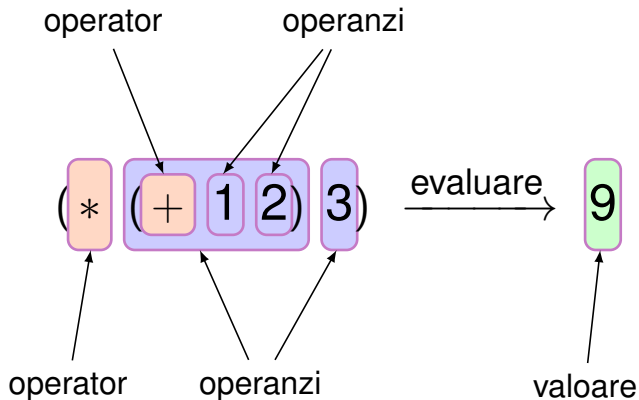
Expresii



Expresii



Expresii



Evaluarea expresiilor primitive

1 (* (+ 1 2) 3)



Evaluarea expresiilor primitive

1. Evaluarea (reducerea) **operanzilor** la valori (argumente)

1 (* (+ 1 2) 3)



Evaluarea expresiilor primitive

1. Evaluarea (reducerea) **operanzilor** la valori (argumente)

1 $(* \text{ (+ 1 2) } 3) \rightarrow (* \text{ 3 } 3)$



Evaluarea expresiilor primitive

1. Evaluarea (reducerea) **operanzilor** la valori (argumente)
2. Aplicarea **operatorului** primitiv asupra argumentelor

1 $(* \ (+ \ 1 \ 2) \ 3) \rightarrow (* \ 3 \ 3)$



Evaluarea expresiilor primitive

1. Evaluarea (reducerea) **operanzilor** la valori (argumente)
2. Aplicarea **operatorului** primitiv asupra argumentelor

1 $(* \ (+ \ 1 \ 2) \ 3) \rightarrow (* \ 3 \ 3) \rightarrow 9$

Evaluarea expresiilor primitive

1. Evaluarea (reducerea) **operandilor** la valori (argumente)
2. Aplicarea **operatorului** primitiv asupra argumentelor

Recursiv pentru subexpresii

1 (* (+ 1 2) 3) → (* 3 3) → 9



Racket stepper

Construcția `define`

Scop

```
1  (define WIDTH 100)
```



Construcția `define`

Scop

```
1 (define WIDTH 100)
```

- *Leagă* o variabilă globală la **valoarea** unei expresii



Construcția `define`

Scop

```
1 (define WIDTH 100)
```

- ▶ *Leagă* o variabilă globală la **valoarea** unei expresii
- ▶ Atenție! Principal, este vorba de **constante**



Construcția `define`

Scop

```
1 (define WIDTH 100)
```

- ▶ *Leagă* o variabilă globală la **valoarea** unei expresii
- ▶ Atenție! Principial, este vorba de **constante**
- ▶ Avantaje:



Construcția `define`

Scop

```
1 (define WIDTH 100)
```

- ▶ *Leagă* o variabilă globală la **valoarea** unei expresii
- ▶ Atenție! Principial, este vorba de **constante**
- ▶ Avantaje:
 - ▶ Lizibilitate (atribuire de **sens** prin numire)



Construcția `define`

Scop

```
1 (define WIDTH 100)
```

- ▶ *Leagă* o variabilă globală la **valoarea** unei expresii
- ▶ Atenție! Principal, este vorba de **constante**
- ▶ Avantaje:
 - ▶ Lizibilitate (atribuire de **sens** prin numire)
 - ▶ Flexibilitate (modificare într-un **singur** loc)



Construcția `define`

Scop

```
1 (define WIDTH 100)
```

- ▶ *Leagă* o variabilă globală la **valoarea** unei expresii
- ▶ Atenție! Principal, este vorba de **constante**
- ▶ Avantaje:
 - ▶ Lizibilitate (atribuire de **sens** prin numire)
 - ▶ Flexibilitate (modificare într-un **singur** loc)
 - ▶ Reutilizare (**evitarea** reproducerii multiple a unei expresii complexe)



Construcția `define`

Evaluare

```
1 (define x (* (+ 1 2) 3) )  
2 (+ x 10)
```



Construcția define

Evaluare

1. La **definire**, se evaluează expresia,

```
1 (define x (* (+ 1 2) 3))  
2 (+ x 10)
```



Construcția define

Evaluare

1. La **definire**, se evaluează expresia, și se *leagă* variabila la **valoarea** ei

```
1 (define x (* (+ 1 2) 3)) ; x <- 9
2 (+ x 10)
```



Construcția `define`

Evaluare

1. La **definire**, se evaluează expresia, și se *leagă* variabila la **valoarea** ei
2. La **utilizare**,

```
1 (define x (* (+ 1 2) 3)) ; x <- 9
2 (+ x 10)
```



Construcția `define`

Evaluare

1. La **definire**, se evaluează expresia, și se *leagă* variabila la **valoarea** ei
2. La **utilizare**, variabila se evaluează la valoarea ei

```
1 (define x (* (+ 1 2) 3)) ; x <- 9
2 (+ x 10) → (+ 9 10)
```



Funcții

Definire

```
1 (define (increment n)
2   (+ n 1) )
```

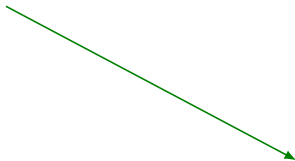
```
1 (define (average x y)
2   (/ (+ x y) 2) )
```



Funcții

Definire

nume



```
1 (define (increment n)
2   (+ n 1) )
```

```
1 (define (average x y)
2   (/ (+ x y) 2) )
```



Funcții

Definire

nume

parametri

```
1 (define (increment n)
2   (+ n 1) )
```

```
1 (define (average x y)
2   (/ (+ x y) 2) )
```



Funcții

Definire

nume

parametri

```
1 (define (increment n)
2   (+ n 1) )
```

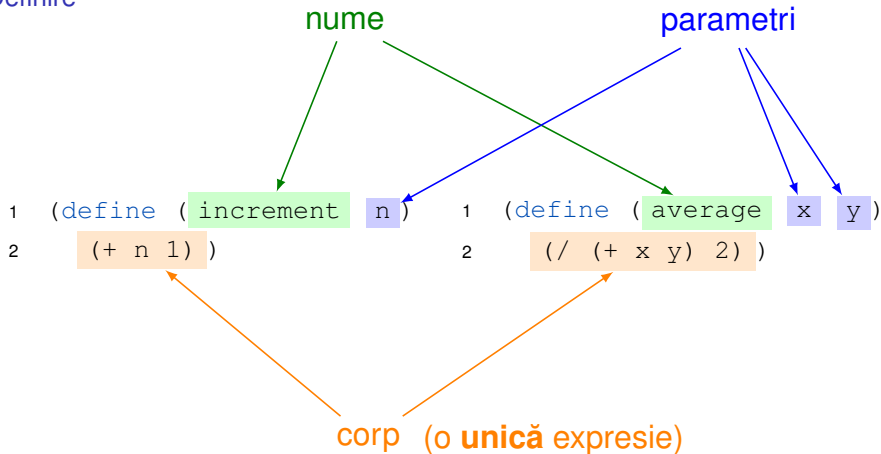
```
1 (define (average x y)
2   (/ (+ x y) 2) )
```

corp (o **unică** expresie)



Funcții

Definire

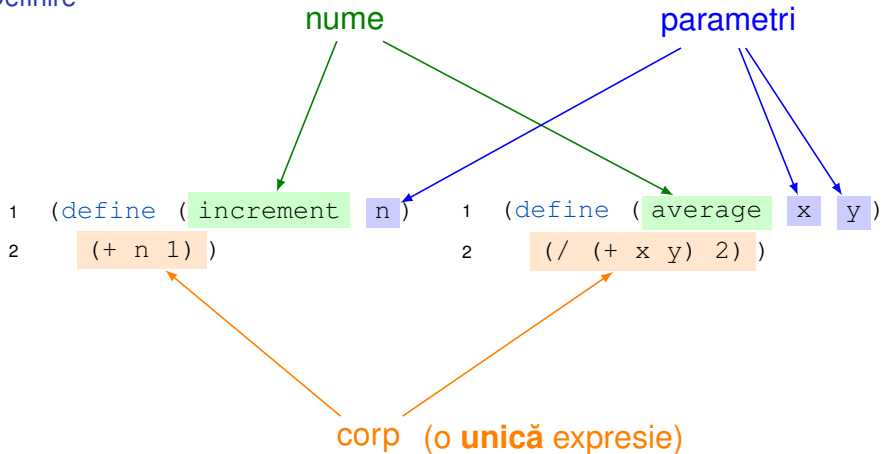


- Accepție matematică a funcțiilor — **valoare** calculată



Funcții

Definire



- ▶ Accepție matematică a funcțiilor — **valoare** calculată
- ▶ **Absența** informației de tip



Funcții

Evaluare

Definire:

```
1 (define (increment x)
2   (+ x 1))
```

Aplicare:

```
1 (increment (+ 1 2))
```



Funcții

Evaluare

Definire:

- ▶ Înregistrarea definiției funcției

```
1 (define (increment x) ; increment <- <functia>  
2   (+ x 1))
```

Aplicare:

```
1 (increment (+ 1 2))
```



Funcții

Evaluare

Definire:

- ▶ Înregistrarea definiției funcției

```
1 (define (increment x) ; increment <- <functia>
2   (+ x 1))
```

Aplicare:

1. Evaluarea (reducerea) operanzilor

```
1 (increment (+ 1 2))
```



Funcții

Evaluare

Definire:

- ▶ Înregistrarea definiției funcției

```
1 (define (increment x) ; increment <- <functia>  
2   (+ x 1))
```

Aplicare:

1. Evaluarea (reducerea) **operandilor** la argumente

```
1 (increment (+ 1 2)) → (increment 3)
```



Funcții

Evaluare

Definire:

- ▶ Înregistrarea definiției funcției

```
1 (define (increment x) ; increment <- <functia>  
2   (+ x 1))
```

Aplicare:

1. Evaluarea (reducerea) **operandilor** la argumente
2. **Substituirea** argumentelor în corpul funcției

```
1 (increment (+ 1 2)) → (increment 3)
```



Funcții

Evaluare

Definire:

- ▶ Înregistrarea definiției funcției

```
1 (define (increment x) ; increment <- <functia>  
2   (+ x 1))
```

Aplicare:

1. Evaluarea (reducerea) **operandilor** la argumente
2. **Substituirea** argumentelor în corpul funcției

```
1 (increment (+ 1 2)) → (increment 3)  
2 → (+ 3 1)
```



Funcții

Evaluare

Definire:

- ▶ Înregistrarea definiției funcției

```
1 (define (increment x) ; increment <- <functia>  
2   (+ x 1))
```

Aplicare:

1. Evaluarea (reducerea) **operandilor** la argumente
2. **Substituirea** argumentelor în corpul funcției
3. Evaluarea expresiei obținute

```
1 (increment (+ 1 2)) → (increment 3)  
2 → (+ 3 1)
```



Funcții

Evaluare

Definire:

- ▶ Înregistrarea definiției funcției

```
1 (define (increment x) ; increment <- <functia>
2   (+ x 1))
```

Aplicare:

1. Evaluarea (reducerea) **operandilor** la argumente
2. **Substituirea** argumentelor în corpul funcției
3. Evaluarea expresiei obținute

```
1 (increment (+ 1 2)) → (increment 3)
2 → (+ 3 1) → 4
```



Construcția `if`

Prezentare

```
1  (if (< 1 2) (+ 3 4) (+ 5 6))
```

- Imaginabilă în forma unei funcții



Construcția `if`

Prezentare

```
1  (if (< 1 2) (+ 3 4) (+ 5 6))
```

- ▶ Imaginabilă în forma unei funcții
- ▶ Ramurile *then* și *else* ca operanzi



Construcția `if`

Prezentare

```
1  (if (< 1 2) (+ 3 4) (+ 5 6))
```

- ▶ Imaginabilă în forma unei **funcții**
- ▶ Ramurile *then* și *else* ca **operandi**
- ▶ De aici, **obligativitatea** prezenței ramurii *else*!



Construcția `if`

Evaluare

```
1 (if (< 1 2) (+ 3 4) (+ 5 6))
```



Construcția `if`

Evaluare

1. Evaluarea condiției

```
1  (if (< 1 2) (+ 3 4) (+ 5 6))
```



Construcția `if`

Evaluare

1. Evaluarea condiției

```
1  (if (< 1 2) (+ 3 4) (+ 5 6))
```

```
2  → (if true (+ 3 4) (+ 5 6))
```



Construcția `if`

Evaluare

1. Evaluarea **condiției**
2. Înlocuirea **întregii** expresii `if` cu ramura potrivită

```
1  (if (< 1 2) (+ 3 4) (+ 5 6))
```

```
2  → (if true (+ 3 4) (+ 5 6))
```



Construcția `if`

Evaluare

1. Evaluarea condiției
2. Înlocuirea întregii expresii `if` cu ramura potrivită

1 `(if (< 1 2) (+ 3 4) (+ 5 6))`

2 `→ (if true (+ 3 4) (+ 5 6))`

3 `→ (+ 3 4)`



Construcția `if`

Evaluare

1. Evaluarea **condiției**
2. Înlocuirea **întregii** expresii `if` cu ramura potrivită
3. Evaluarea expresiei obținute

1 `(if (< 1 2) (+ 3 4) (+ 5 6))`

2 `→ (if true (+ 3 4) (+ 5 6))`

3 `→ (+ 3 4)`



Construcția `if`

Evaluare

1. Evaluarea condiției
2. Înlocuirea întregii expresii `if` cu ramura potrivită
3. Evaluarea expresiei obținute

1 `(if (< 1 2) (+ 3 4) (+ 5 6))`

2 `→ (if true (+ 3 4) (+ 5 6))`

3 `→ (+ 3 4) → 7`



Construcția `if`

Evaluare

1. Evaluarea **condiției**
2. Înlocuirea **întregii** expresii `if` cu ramura potrivită
3. Evaluarea expresiei obținute

Ordine **diferită** de evaluare, față de funcțiile obișnuite!

1 `(if (< 1 2) (+ 3 4) (+ 5 6))`

2 $\rightarrow$ `(if true (+ 3 4) (+ 5 6))`

3 $\rightarrow$ `(+ 3 4)` $\rightarrow$ 7



Cuprins

Expresii și evaluare

Liste și perechi

Tipare

Omoiconicitate și metaprogramare



Liste

Literali

- ▶ Aspectul de listă al aplicațiilor operatorilor

(+ 1 2)



Liste

Literali

- ▶ Aspectul de listă al aplicațiilor operatorilor

(+ 1 2)

- ▶ Ce s-ar întâmpla dacă am înlocui + cu 0?

(0 1 2)



Liste

Literali

- ▶ Aspectul de listă al **aplicațiilor** operatorilor

(+ 1 2)

- ▶ Ce s-ar întâmpla dacă am înlocui + cu 0?

(0 1 2)

Eroare! 0 nu este operator!



Liste

Literali

- ▶ Aspectul de listă al **aplicațiilor** operatorilor

`(+ 1 2)`

- ▶ Ce s-ar întâmpla dacă am înlocui `+` cu `0`?

`(0 1 2)`

Eroare! `0` nu este operator!

- ▶ Soluție: **împiedicarea** evaluării, cu `quote`

`(quote (0 1 2))` sau `'(0 1 2)`



Liste

Structură

- ▶ Structură **recursivă**



Liste

Structură

- ▶ Structură **recursivă**
 - ▶ O listă **nouă** se obține prin atașarea unui element (*head*) în fața altei liste (*tail*), **fără** modificarea listei existente!

`(cons 0 ' (1 2)) → ' (0 1 2)`



Liste

Structură

- ▶ Structură **recursivă**

- ▶ O listă **nouă** se obține prin atașarea unui element (*head*) în fața altei liste (*tail*), **fără** modificarea listei existente!

$$(\text{cons } 0 \text{ ' (1 2)}) \rightarrow \text{' (0 1 2)}$$

- ▶ Cazul de bază: lista vidă, **' ()**



Liste

Structură

- ▶ Structură **recursivă**
 - ▶ O listă **nouă** se obține prin atașarea unui element (*head*) în fața altei liste (*tail*), **fără** modificarea listei existente!

`(cons 0 ' (1 2)) → ' (0 1 2)`

- ▶ Cazul de bază: lista vidă, `' ()`
- ▶ Alternativă de construcție: funcția `list`

`(list 0 1 2)`



Liste

Structură

- ▶ Structură **recursivă**

- ▶ O listă **nouă** se obține prin atașarea unui element (*head*) în fața altei liste (*tail*), **fără** modificarea listei existente!

$$(\text{cons } 0 \text{ ' (1 2)}) \rightarrow \text{' (0 1 2)}$$

- ▶ Cazul de bază: lista vidă, **' ()**

- ▶ Alternativă de construcție: funcția `list`

$$(\text{list } 0 \text{ 1 2})$$

- ▶ Selectorii

$$\begin{aligned} (\text{car } \text{' (0 1 2)}) &\rightarrow 0 \\ (\text{cdr } \text{' (0 1 2)}) &\rightarrow \text{' (1 2)} \end{aligned}$$


Liste

Funcții

- ▶ Exploatarea structurii **recursive** de funcțiile pe liste



Liste

Funcții

- ▶ Exploatarea structurii **recursive** de funcțiile pe liste
- ▶ Exemplu: **minimul** unei liste nevide (v. slide-ul 21)



Liste

Funcții

- ▶ Exploatarea structurii **recursive** de funcțiile pe liste
- ▶ Exemplu: **minimul** unei liste nevide (v. slide-ul 21)
 - ▶ **Axiome**, pornind de la un tip de date abstract *List*, cu constructorii de bază *'()* și *cons*:

$$(\text{minList } (\text{cons } e \ '())) = e$$

$$(\text{minList } (\text{cons } e \ L)) = (\text{min } e \ (\text{minList } (\text{cdr } L)))$$



Liste

Funcții

- ▶ Exploatarea structurii **recursive** de funcțiile pe liste
- ▶ Exemplu: **minimul** unei liste nevide (v. slide-ul 21)
 - ▶ **Axiome**, pornind de la un tip de date abstract *List*, cu constructorii de bază *'()* și *cons*:

$$(\text{minList } (\text{cons } e '())) = e$$

$$(\text{minList } (\text{cons } e L)) = (\text{min } e (\text{minList } (\text{cdr } L)))$$

- ▶ Implementare

```
1 (define (minList1 L)
2   (if (= (length L) 1) (car L)
3       (min (car L) (minList1 (cdr L)))))
```



Liste

Funcții

- ▶ Exploatarea structurii **recursive** de funcțiile pe liste
- ▶ Exemplu: **minimul** unei liste nevide (v. slide-ul 21)
 - ▶ **Axiome**, pornind de la un tip de date abstract *List*, cu constructorii de bază *'()* și *cons*:

$$(\text{minList } (\text{cons } e '())) = e$$

$$(\text{minList } (\text{cons } e L)) = (\text{min } e (\text{minList } (\text{cdr } L)))$$

- ▶ Implementare

```
1 (define (minList1 L)
2   (if (= (length L) 1) (car L)
3       (min (car L) (minList1 (cdr L)))))
```

- ▶ Traducere **fidelă** a axiomelor unui TDA într-un program funcțional!



Perechi

- ▶ Intern, listă $\equiv$ pereche *head-tail*



Perechi

- ▶ Intern, listă $\equiv$ pereche *head-tail*
- ▶ `cons`, aplicabil asupra oricăror doi operanzi, pentru generarea unei perechi cu punct (*dotted pair*)

`(cons 0 1) → '(0 . 1)`



Perechi

- ▶ Intern, listă $\equiv$ pereche *head-tail*
- ▶ `cons`, aplicabil asupra oricăror doi operanzi, pentru generarea unei perechi cu punct (*dotted pair*)

$$\begin{aligned} (\text{cons } 0 \ 1) &\rightarrow '(0 \ . \ 1) \\ '(0 \ 1 \ 2) &\equiv '(0 \ . \ (1 \ . \ (2 \ . \ ()))) \end{aligned}$$


Perechi

- ▶ Intern, listă $\equiv$ pereche *head-tail*
- ▶ `cons`, aplicabil asupra oricăror doi operanzi, pentru generarea unei perechi cu punct (*dotted pair*)

$$\begin{aligned} (\text{cons } 0 \ 1) &\rightarrow '(0 \ . \ 1) \\ '(0 \ 1 \ 2) &\equiv '(0 \ . \ (1 \ . \ (2 \ . \ ()))) \end{aligned}$$

- ▶ Toretic, perechi reprezentabile ca funcții!
(vom vedea mai târziu).



Perechi

- ▶ Intern, listă $\equiv$ pereche *head-tail*
- ▶ `cons`, aplicabil asupra oricăror doi operanzi, pentru generarea unei perechi cu punct (*dotted pair*)

$$\begin{aligned} (\text{cons } 0 \ 1) &\rightarrow ' (0 \ . \ 1) \\ ' (0 \ 1 \ 2) &\equiv ' (0 \ . \ (1 \ . \ (2 \ . \ ()))) \end{aligned}$$

- ▶ Toretic, perechi reprezentabile ca funcții!
(vom vedea mai târziu). De fapt, ...



Universalitatea funcțiilor

- ▶ ..., orice limbaj prevăzut **exclusiv** cu funcții și **fără** tipuri predefinite este **la fel** de expresiv ca orice alt limbaj (în limitele tezei Church-Turing)



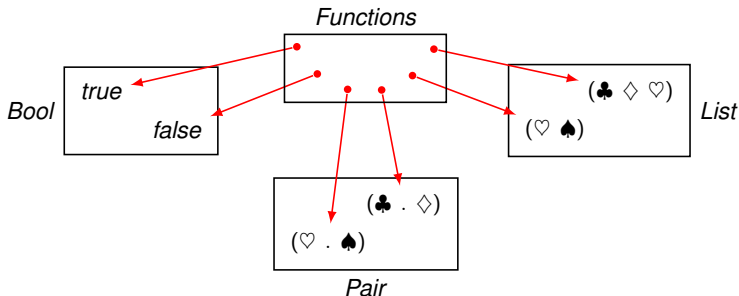
Universalitatea funcțiilor

- ▶ ..., orice limbaj prevăzut **exclusiv** cu funcții și **fără** tipuri predefinite este **la fel** de expresiv ca orice alt limbaj (în limitele tezei Church-Turing)
- ▶ Majoritatea **tipurilor** uzuale, codificabile direct prin intermediul funcțiilor



Universalitatea funcțiilor

- ..., orice limbaj prevăzut **exclusiv** cu funcții și **fără** tipuri predefinite este **la fel** de expresiv ca orice alt limbaj (în limitele tezei Church-Turing)
- Majoritatea **tipurilor** uzuale, codificabile direct prin intermediul funcțiilor



Cuprins

Expresii și evaluare

Liste și perechi

Tipare

Omoiconicitate și metaprogramare



Caracteristici

- ▶ **Tipare** = modalitatea de definire, manipulare și verificare a tipurilor dintr-un limbaj



Caracteristici

- ▶ **Tipare** = modalitatea de definire, manipulare și verificare a tipurilor dintr-un limbaj
- ▶ Existența unor tipuri **predefinite** în Racket (boolean, caracter, număr etc.)



Caracteristici

- ▶ **Tipare** = modalitatea de definire, manipulare și verificare a tipurilor dintr-un limbaj
- ▶ Existența unor tipuri **predefinite** în Racket (boolean, caracter, număr etc.)
- ▶ Întrebări:

Caracteristici

- ▶ **Tipare** = modalitatea de definire, manipulare și verificare a tipurilor dintr-un limbaj
- ▶ Existența unor tipuri **predefinite** în Racket (boolean, caracter, număr etc.)
- ▶ Întrebări:
 - ▶ **Când** se realizează verificarea?



Caracteristici

- ▶ **Tipare** = modalitatea de definire, manipulare și verificare a tipurilor dintr-un limbaj
- ▶ Existența unor tipuri **predefinite** în Racket (boolean, caracter, număr etc.)
- ▶ Întrebări:
 - ▶ **Când** se realizează verificarea?
 - ▶ Cât de **flexibile** sunt regulile de tipare?



Flexibilitatea regulilor

- ▶ Ce produce evaluarea următoarei expresii?

(+ 1 "OK")



Flexibilitatea regulilor

- ▶ Ce produce evaluarea următoarei expresii?

(+ 1 "OK")

- ▶ Criteriu: flexibilitatea în agregarea valorilor de tipuri diferite



Flexibilitatea regulilor

- ▶ Ce produce evaluarea următoarei expresii?

(+ 1 "OK")

- ▶ Criteriu: flexibilitatea în agregarea valorilor de tipuri diferite
- ▶ Racket: verificare rigidă — tipare tare (*strong*)



Flexibilitatea regulilor

- ▶ Ce produce evaluarea următoarei expresii?

(+ 1 "OK")

- ▶ Criteriu: flexibilitatea în agregarea valorilor de tipuri diferite
- ▶ Racket: verificare rigidă — tipare tare (*strong*)
- ▶ Răspuns: eroare!



Flexibilitatea regulilor

- ▶ Ce produce evaluarea următoarei expresii?

(+ 1 "OK")

- ▶ Criteriu: flexibilitatea în agregarea valorilor de tipuri diferite
- ▶ Racket: verificare rigidă — tipare tare (*strong*)
- ▶ Răspuns: eroare!
- ▶ Alternativă în alte limbaje — tipare slabă (*weak*)



Flexibilitatea regulilor

- ▶ Ce produce evaluarea următoarei expresii?

`(+ 1 "OK")`

- ▶ Criteriu: flexibilitatea în agregarea valorilor de tipuri diferite
- ▶ Racket: verificare rigidă — tipare tare (*strong*)
- ▶ Răspuns: eroare!
- ▶ Alternativă în alte limbaje — tipare slabă (*weak*)
 - ▶ Visual Basic: `1 + "23" = 24`



Flexibilitatea regulilor

- ▶ Ce produce evaluarea următoarei expresii?

`(+ 1 "OK")`

- ▶ Criteriu: flexibilitatea în agregarea valorilor de tipuri diferite
- ▶ Racket: verificare rigidă — tipare tare (*strong*)
- ▶ Răspuns: eroare!
- ▶ Alternativă în alte limbaje — tipare slabă (*weak*)
 - ▶ Visual Basic: `1 + "23" = 24`
 - ▶ JavaScript: `1 + "23" = "123"`



Momentul verificării

- ▶ Ce produce evaluarea următoarei expresii?

```
(+ 1 (if condition 2 "OK"))
```



Momentul verificării

- ▶ Ce produce evaluarea următoarei expresii?

```
(+ 1 (if condition 2 "OK"))
```

- ▶ Racket: verificare în momentul **aplicării** unui operator **predefinit** — tipare **dinamică**



Momentul verificării

- ▶ Ce produce evaluarea următoarei expresii?

```
(+ 1 (if condition 2 "OK"))
```

- ▶ Racket: verificare în momentul **aplicării** unui operator **predefinit** — tipare **dinamică**
- ▶ Răspunsul depinde de valoarea lui `condition`:



Momentul verificării

- ▶ Ce produce evaluarea următoarei expresii?

```
(+ 1 (if condition 2 "OK"))
```

- ▶ Racket: verificare în momentul **aplicării** unui operator **predefinit** — tipare **dinamică**
- ▶ Răspunsul depinde de valoarea lui `condition`:
 - ▶ `true`: 3



Momentul verificării

- ▶ Ce produce evaluarea următoarei expresii?

```
(+ 1 (if condition 2 "OK"))
```

- ▶ Racket: verificare în momentul **aplicării** unui operator **predefinit** — tipare **dinamică**
- ▶ Răspunsul depinde de valoarea lui `condition`:
 - ▶ `true`: 3
 - ▶ `false`: Eroare, imposibilitatea adunării unui număr cu un șir



Momentul verificării

- ▶ Ce produce evaluarea următoarei expresii?

```
(+ 1 (if condition 2 "OK"))
```

- ▶ Racket: verificare în momentul **aplicării** unui operator **predefinit** — tipare **dinamică**
- ▶ Răspunsul depinde de valoarea lui `condition`:
 - ▶ `true`: 3
 - ▶ `false`: Eroare, imposibilitatea adunării unui număr cu un șir
- ▶ Posibilitatea evaluării cu succes a unei expresii ce conține subexpresii eronate, cât timp cele din urmă **nu** sunt evaluate



Cuprins

Expresii și evaluare

Liste și perechi

Tipare

Omoiconicitate și metaprogramare



Omoiconicitate și metaprogramare

- ▶ **Corepondență** între sintaxa programului și structura de date fundamentală (lista)



Omoiconicitate și metaprogramare

- ▶ Corepondență între sintaxa programului și structura de date fundamentală (lista)
- ▶ Racket — limbaj **omoiconic**
(*homo* = *aceeași*, *icon* = *reprezentare*)



Omoiconicitate și metaprogramare

- ▶ Corepondență între sintaxa programului și structura de date fundamentală (lista)
- ▶ Racket — limbaj **omoiconic**
(*homo* = *aceeași*, *icon* = *reprezentare*)
- ▶ Manipularea listelor ~ manipularea **codului**



Omoiconicitate și metaprogramare

- ▶ **Corepondență** între sintaxa programului și structura de date fundamentală (lista)
- ▶ Racket — limbaj **omoiconic**
(*homo* = *aceeași*, *icon* = *reprezentare*)
- ▶ Manipularea listelor ~ manipularea **codului**
- ▶ **Metaprogramare**: posibilitatea programului de a se **autorescrie**



Exemplu de metaprogramare

```
1 (define plus (list '+ 3 2))
```



Exemplu de metaprogramare

```
1 (define plus (list '+ 3 2)) ; '(+ 3 2)
```



Exemplu de metaprogramare

```
1 (define plus (list '+ 3 2)) ; '(+ 3 2)
2 (eval plus) ; 5
```



Exemplu de metaprogramare

```
1 (define plus (list '+ 3 2)) ; '(+ 3 2)
2 (eval plus) ; 5
3
4 (define minus (cons '- (cdr plus)))
```



Exemplu de metaprogramare

```
1 (define plus (list '+ 3 2)) ; '(+ 3 2)
2 (eval plus) ; 5
3
4 (define minus (cons '- (cdr plus))) ; '(- 3 2)
```



Exemplu de metaprogramare

```
1 (define plus (list '+ 3 2)) ; '(+ 3 2)
2 (eval plus) ; 5
3
4 (define minus (cons '- (cdr plus))) ; '(- 3 2)
5 (eval minus) ; 1
```



Exemplu de metaprogramare

```
1 (define plus (list '+ 3 2)) ; '(+ 3 2)
2 (eval plus) ; 5
3
4 (define minus (cons '- (cdr plus))) ; '(- 3 2)
5 (eval minus) ; 1
```

Forțarea evaluării de către `eval`



Rezumat

- ▶ Limbaj **omoiconic**
- ▶ Evaluare bazată pe **substituție** textuală
- ▶ Tipare **dinamică** și **tare**

Partea III

Recursivitate



Cuprins

Introducere

Tipuri de recursivitate

Specificul recursivității pe coadă



Cuprins

Introducere

Tipuri de recursivitate

Specificul recursivității pe coadă



Recursivitate

- ▶ Componentă **fundamentală** a paradigmei funcționale



Recursivitate

- ▶ Componentă **fundamentală** a paradigmei funcționale
- ▶ **Substituit** pentru iterarea clasică (*for*, *while* etc.), în **absența** stării



Recursivitate

- ▶ Componentă **fundamentală** a paradigmei funcționale
- ▶ **Substituit** pentru iterarea clasică (*for*, *while* etc.), în **absența** stării
- ▶ Formă de *wishful thinking*: “Consider rezolvată **subproblema** și mă gândesc la cum să rezolv problema”



Cuprins

Introducere

Tipuri de recursivitate

Specificul recursivității pe coadă



Funcția *factorial*

Recursivitate pe stivă, liniară

```
5  (define (fact-stack n)
6    (if (= n 1)
7        1
8        (* n (fact-stack (- n 1)))))

1  (fact-stack 3)
```



Funcția *factorial*

Recursivitate pe stivă, liniară

```
5  (define (fact-stack n)
6    (if (= n 1)
7        1
8        (* n (fact-stack (- n 1)))))

1  (fact-stack 3)
```

3

Stiva procesului



Funcția *factorial*

Recursivitate pe stivă, liniară

```
5 (define (fact-stack n)
6   (if (= n 1)
7       1
8       (* n (fact-stack (- n 1)))))
```

```
1 (fact-stack 3)
2 → (* 3 (fact-stack 2))
```

3

Stiva procesului

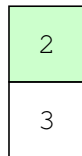


Funcția *factorial*

Recursivitate pe stivă, liniară

```
5  (define (fact-stack n)
6    (if (= n 1)
7        1
8        (* n (fact-stack (- n 1)))))
```

```
1  (fact-stack 3)
2  →  (* 3 (fact-stack 2) )
```



Stiva procesului

Funcția *factorial*

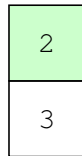
Recursivitate pe stivă, liniară

```
5 (define (fact-stack n)
6   (if (= n 1)
7       1
8       (* n (fact-stack (- n 1)))))
```

```
1 (fact-stack 3)
```

```
2 → (* 3 (fact-stack 2))
```

```
3 → (* 3 (* 2 (fact-stack 1)))
```



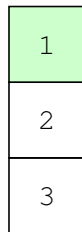
Stiva procesului

Funcția *factorial*

Recursivitate pe stivă, liniară

```
5 (define (fact-stack n)
6   (if (= n 1)
7       1
8       (* n (fact-stack (- n 1)))))
```

```
1 (fact-stack 3)
2 → (* 3 (fact-stack 2) )
3 → (* 3 (* 2 (fact-stack 1) ) )
```



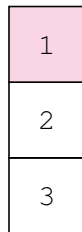
Stiva procesului

Funcția *factorial*

Recursivitate pe stivă, liniară

```
5 (define (fact-stack n)
6   (if (= n 1)
7       1
8       (* n (fact-stack (- n 1)))))
```

```
1 (fact-stack 3)
2 → (* 3 (fact-stack 2) )
3 → (* 3 (* 2 (fact-stack 1) ) )
4 → (* 3 (* 2 1) )
```



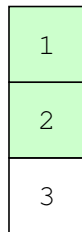
Stiva procesului

Funcția *factorial*

Recursivitate pe stivă, liniară

```
5 (define (fact-stack n)
6   (if (= n 1)
7       1
8       (* n (fact-stack (- n 1)))))
```

```
1 (fact-stack 3)
2 → (* 3 (fact-stack 2) )
3 → (* 3 (* 2 (fact-stack 1) ) )
4 → (* 3 (* 2 1) )
```



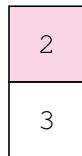
Stiva procesului

Funcția *factorial*

Recursivitate pe stivă, liniară

```
5 (define (fact-stack n)
6   (if (= n 1)
7       1
8       (* n (fact-stack (- n 1)))))
```

```
1 (fact-stack 3)
2 → (* 3 (fact-stack 2) )
3 → (* 3 (* 2 (fact-stack 1) ) )
4 → (* 3 (* 2 1 ) )
5 → (* 3 2 )
```



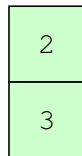
Stiva procesului

Funcția *factorial*

Recursivitate pe stivă, liniară

```
5  (define (fact-stack n)
6    (if (= n 1)
7        1
8        (* n (fact-stack (- n 1)))))
```

```
1  (fact-stack 3)
2  →  (* 3 (fact-stack 2) )
3  →  (* 3 (* 2 (fact-stack 1) ) )
4  →  (* 3 (* 2 1) )
5  →  (* 3 2 )
```



Stiva procesului

Funcția *factorial*

Recursivitate pe stivă, liniară

```
5 (define (fact-stack n)
6   (if (= n 1)
7       1
8       (* n (fact-stack (- n 1)))))
```

```
1 (fact-stack 3)
2 → (* 3 (fact-stack 2) )
3 → (* 3 (* 2 (fact-stack 1) ) )
4 → (* 3 (* 2 1) )
5 → (* 3 2)
6 → 6
```



Stiva procesului

Recursivitate pe stivă, liniară

- ▶ Depunerea pe stivă a unor valori pe **avansul** în recursivitate



Recursivitate pe stivă, liniară

- ▶ Depunerea pe stivă a unor valori pe **avansul** în recursivitate
- ▶ Utilizarea acestora pentru calculul propriu-zis, pe **revenirea** din recursivitate



Recursivitate pe stivă, liniară

- ▶ Depunerea pe stivă a unor valori pe **avansul** în recursivitate
- ▶ Utilizarea acestora pentru calculul propriu-zis, pe **revenirea** din recursivitate
- ▶ **Spațiul** ocupat pe stivă: $\Theta(n)$



Recursivitate pe stivă, liniară

- ▶ Depunerea pe stivă a unor valori pe **avansul** în recursivitate
- ▶ Utilizarea acestora pentru calculul propriu-zis, pe **revenirea** din recursivitate
- ▶ **Spațiul** ocupat pe stivă: $\Theta(n)$
- ▶ Numărul de **operații**: $\Theta(n)$



Recursivitate pe stivă, liniară

- ▶ Depunerea pe stivă a unor valori pe **avansul** în recursivitate
- ▶ Utilizarea acestora pentru calculul propriu-zis, pe **revenirea** din recursivitate
- ▶ **Spațiul** ocupat pe stivă: $\Theta(n)$
- ▶ Numărul de **operații**: $\Theta(n)$
- ▶ Informație “ascunsă”, **implicită**, despre stare



Funcția *factorial*

Iterare clasică

```
1: procedure FACTORIAL( $n$ )  
2:    $product \leftarrow 1$   
3:    $i \leftarrow 1$   
4:   while  $i \leq n$  do  
5:      $product \leftarrow product \cdot i$   
6:      $i \leftarrow i + 1$   
7:   end while  
8:   return  $product$   
9: end procedure
```



Funcția *factorial*

Iterare clasică

```
1: procedure FACTORIAL(n)  
2:   product  $\leftarrow$  1  
3:   i  $\leftarrow$  1  
4:   while  $i \leq n$  do  
5:     product  $\leftarrow$  product · i  
6:     i  $\leftarrow$  i + 1  
7:   end while  
8:   return product  
9: end procedure
```

- **Starea** programului: variabilele *i* și *product*



Funcția *factorial*

Iterare clasică

```
1: procedure FACTORIAL( $n$ )  
2:    $product \leftarrow 1$   
3:    $i \leftarrow 1$   
4:   while  $i \leq n$  do  
5:      $product \leftarrow product \cdot i$   
6:      $i \leftarrow i + 1$   
7:   end while  
8:   return  $product$   
9: end procedure
```

- ▶ **Starea** programului: variabilele i și $product$
- ▶ Spațiu **constant** pe stivă!



Funcția *factorial*

Iterare clasică

```
1: procedure FACTORIAL(n)  
2:   product  $\leftarrow$  1  
3:   i  $\leftarrow$  1  
4:   while  $i \leq n$  do  
5:     product  $\leftarrow$  product · i  
6:     i  $\leftarrow$  i + 1  
7:   end while  
8:   return product  
9: end procedure
```

- ▶ **Starea** programului: variabilele *i* și *product*
- ▶ Spațiu **constant** pe stivă!
- ▶ Cum putem exploata această idee?



Funcția *factorial*

Recursivitate pe coadă

```
18 (define (fact-tail n)
19   (fact-tail-helper 1 1 n))
20
21 (define (fact-tail-helper product i n)
22   (if (> i n)
23       product
24       (fact-tail-helper (* product i)
25                           (+ i 1)
26                           n)))
1  (fact-tail-helper 1 1 3)
```



Funcția *factorial*

Recursivitate pe coadă

```
18 (define (fact-tail n)
19   (fact-tail-helper 1 1 n))
20
21 (define (fact-tail-helper product i n)
22   (if (> i n)
23       product
24       (fact-tail-helper (* product i)
25                           (+ i 1)
26                           n)))
1  (fact-tail-helper 1 1 3)
```

1, 1, 3

Stiva aparentă



Funcția *factorial*

Recursivitate pe coadă

```
18 (define (fact-tail n)
19   (fact-tail-helper 1 1 n))
20
21 (define (fact-tail-helper product i n)
22   (if (> i n)
23       product
24       (fact-tail-helper (* product i)
25                           (+ i 1)
26                           n)))
1  (fact-tail-helper 1 1 3)
2  →  (fact-tail-helper 1 2 3)
```

1, 1, 3

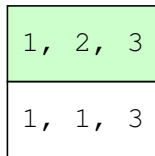
Stiva aparentă



Funcția *factorial*

Recursivitate pe coadă

```
18 (define (fact-tail n)
19   (fact-tail-helper 1 1 n))
20
21 (define (fact-tail-helper product i n)
22   (if (> i n)
23       product
24       (fact-tail-helper (* product i)
25                           (+ i 1)
26                           n)))
27
28 1 (fact-tail-helper 1 1 3)
29 2 → (fact-tail-helper 1 2 3)
```



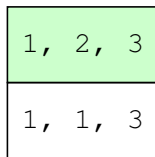
Stiva aparentă



Funcția *factorial*

Recursivitate pe coadă

```
18 (define (fact-tail n)
19   (fact-tail-helper 1 1 n))
20
21 (define (fact-tail-helper product i n)
22   (if (> i n)
23       product
24       (fact-tail-helper (* product i)
25                           (+ i 1)
26                           n)))
27
28 1 (fact-tail-helper 1 1 3)
29 2 → (fact-tail-helper 1 2 3)
30 3 → (fact-tail-helper 2 3 3)
```



Stiva aparentă

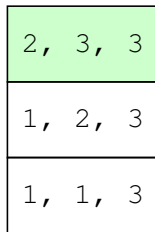


Funcția *factorial*

Recursivitate pe coadă

```
18 (define (fact-tail n)
19   (fact-tail-helper 1 1 n))
20
21 (define (fact-tail-helper product i n)
22   (if (> i n)
23       product
24       (fact-tail-helper (* product i)
25                           (+ i 1)
26                           n)))

1  (fact-tail-helper 1 1 3)
2  →  (fact-tail-helper 1 2 3)
3  →  (fact-tail-helper 2 3 3)
```



Stiva aparentă

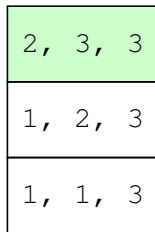


Funcția *factorial*

Recursivitate pe coadă

```
18 (define (fact-tail n)
19   (fact-tail-helper 1 1 n))
20
21 (define (fact-tail-helper product i n)
22   (if (> i n)
23       product
24       (fact-tail-helper (* product i)
25                           (+ i 1)
26                           n)))

1  (fact-tail-helper 1 1 3)
2  →  (fact-tail-helper 1 2 3)
3  →  (fact-tail-helper 2 3 3)
4  →  (fact-tail-helper 6 4 3)
```



Stiva aparentă



Funcția *factorial*

Recursivitate pe coadă

```
18 (define (fact-tail n)
19   (fact-tail-helper 1 1 n))
20
21 (define (fact-tail-helper product i n)
22   (if (> i n)
23       product
24       (fact-tail-helper (* product i)
25                           (+ i 1)
26                           n)))
27
28 1 (fact-tail-helper 1 1 3)
29 2 → (fact-tail-helper 1 2 3)
30 3 → (fact-tail-helper 2 3 3)
31 4 → (fact-tail-helper 6 4 3)
```

6, 4, 3
2, 3, 3
1, 2, 3
1, 1, 3

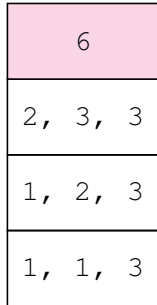
Stiva aparentă



Funcția *factorial*

Recursivitate pe coadă

```
18 (define (fact-tail n)
19   (fact-tail-helper 1 1 n))
20
21 (define (fact-tail-helper product i n)
22   (if (> i n)
23       product
24       (fact-tail-helper (* product i)
25                           (+ i 1)
26                           n)))
27
28 1 (fact-tail-helper 1 1 3)
29 2 → (fact-tail-helper 1 2 3)
30 3 → (fact-tail-helper 2 3 3)
31 4 → (fact-tail-helper 6 4 3)
32 5 → 6
```



Stiva aparentă

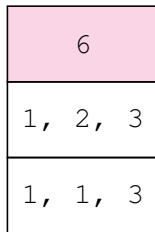


Funcția *factorial*

Recursivitate pe coadă

```
18 (define (fact-tail n)
19   (fact-tail-helper 1 1 n))
20
21 (define (fact-tail-helper product i n)
22   (if (> i n)
23       product
24       (fact-tail-helper (* product i)
25                           (+ i 1)
26                           n)))

1  (fact-tail-helper 1 1 3)
2  →  (fact-tail-helper 1 2 3)
3  →  (fact-tail-helper 2 3 3)
4  →  (fact-tail-helper 6 4 3)
5  →  6
```



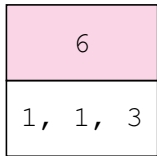
Stiva aparentă



Funcția *factorial*

Recursivitate pe coadă

```
18 (define (fact-tail n)
19   (fact-tail-helper 1 1 n))
20
21 (define (fact-tail-helper product i n)
22   (if (> i n)
23       product
24       (fact-tail-helper (* product i)
25                           (+ i 1)
26                           n)))
27
28 1 (fact-tail-helper 1 1 3)
29 2 → (fact-tail-helper 1 2 3)
30 3 → (fact-tail-helper 2 3 3)
31 4 → (fact-tail-helper 6 4 3)
32 5 → 6
```



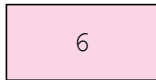
Stiva aparentă



Funcția *factorial*

Recursivitate pe coadă

```
18 (define (fact-tail n)
19   (fact-tail-helper 1 1 n))
20
21 (define (fact-tail-helper product i n)
22   (if (> i n)
23       product
24       (fact-tail-helper (* product i)
25                           (+ i 1)
26                           n)))
27
28 1 (fact-tail-helper 1 1 3)
29 2 → (fact-tail-helper 1 2 3)
30 3 → (fact-tail-helper 2 3 3)
31 4 → (fact-tail-helper 6 4 3)
32 5 → 6
```



Stiva aparentă



Recursivitate pe coadă

- ▶ Calcul realizat pe **avansul** în recursivitate
- ▶ Aparent, **transportarea** neschimbată a valorii celei mai adânci aplicații recursive, către prima
- ▶ În realitate, ***tail call optimization***: înlocuirea fiecărui apel cu următorul

```
1 (fact-tail-helper 1 1 3)
```



Recursivitate pe coadă

- ▶ Calcul realizat pe **avansul** în recursivitate
- ▶ Aparent, **transportarea** neschimbată a valorii celei mai adânci aplicații recursive, către prima
- ▶ În realitate, ***tail call optimization***: înlocuirea fiecărui apel cu următorul

```
1 (fact-tail-helper 1 1 3)
```

1, 1, 3

Stiva reală
(*tail call optimization*)



Recursivitate pe coadă

- ▶ Calcul realizat pe **avansul** în recursivitate
- ▶ Aparent, **transportarea** neschimbată a valorii celei mai adânci aplicații recursive, către prima
- ▶ În realitate, ***tail call optimization***: înlocuirea fiecărui apel cu următorul

```
1 (fact-tail-helper 1 1 3)  
2 → (fact-tail-helper 1 2 3)
```

1, 1, 3

Stiva reală
(*tail call optimization*)



Recursivitate pe coadă

- ▶ Calcul realizat pe **avansul** în recursivitate
- ▶ Aparent, **transportarea** neschimbată a valorii celei mai adânci aplicații recursive, către prima
- ▶ În realitate, **tail call optimization**: înlocuirea fiecărui apel cu următorul

```
1 (fact-tail-helper 1 1 3)
2 → (fact-tail-helper 1 2 3)
```

1, 2, 3

Stiva reală
(*tail call optimization*)



Recursivitate pe coadă

- ▶ Calcul realizat pe **avansul** în recursivitate
- ▶ Aparent, **transportarea** neschimbată a valorii celei mai adânci aplicații recursive, către prima
- ▶ În realitate, **tail call optimization**: înlocuirea fiecărui apel cu următorul

```
1 (fact-tail-helper 1 1 3)
2 → (fact-tail-helper 1 2 3)
3 → (fact-tail-helper 2 3 3)
```

1, 2, 3

Stiva reală
(*tail call optimization*)



Recursivitate pe coadă

- ▶ Calcul realizat pe **avansul** în recursivitate
- ▶ Aparent, **transportarea** neschimbată a valorii celei mai adânci aplicații recursive, către prima
- ▶ În realitate, **tail call optimization**: înlocuirea fiecărui apel cu următorul

```
1 (fact-tail-helper 1 1 3)
2 → (fact-tail-helper 1 2 3)
3 → (fact-tail-helper 2 3 3)
```

2, 3, 3

Stiva reală
(*tail call optimization*)



Recursivitate pe coadă

- ▶ Calcul realizat pe **avansul** în recursivitate
- ▶ Aparent, **transportarea** neschimbată a valorii celei mai adânci aplicații recursive, către prima
- ▶ În realitate, **tail call optimization**: înlocuirea fiecărui apel cu următorul

```
1 (fact-tail-helper 1 1 3)
2 → (fact-tail-helper 1 2 3)
3 → (fact-tail-helper 2 3 3)
4 → (fact-tail-helper 6 4 3)
```

2, 3, 3

Stiva reală
(*tail call optimization*)



Recursivitate pe coadă

- ▶ Calcul realizat pe **avansul** în recursivitate
- ▶ Aparent, **transportarea** neschimbată a valorii celei mai adânci aplicații recursive, către prima
- ▶ În realitate, **tail call optimization**: înlocuirea fiecărui apel cu următorul

```
1 (fact-tail-helper 1 1 3)
2 → (fact-tail-helper 1 2 3)
3 → (fact-tail-helper 2 3 3)
4 → (fact-tail-helper 6 4 3)
```

6, 4, 3

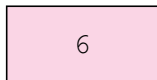
Stiva reală
(*tail call optimization*)



Recursivitate pe coadă

- ▶ Calcul realizat pe **avansul** în recursivitate
- ▶ Aparent, **transportarea** neschimbată a valorii celei mai adânci aplicații recursive, către prima
- ▶ În realitate, **tail call optimization**: înlocuirea fiecărui apel cu următorul

```
1 (fact-tail-helper 1 1 3)
2 → (fact-tail-helper 1 2 3)
3 → (fact-tail-helper 2 3 3)
4 → (fact-tail-helper 6 4 3)
5 → 6
```



Stiva reală
(*tail call optimization*)



Recursivitate pe coadă (cont.)

- ▶ Numărul de operații: $\Theta(n)$



Recursivitate pe coadă (cont.)

- ▶ Numărul de operații: $\Theta(n)$
- ▶ Spațiul ocupat pe stivă: $\Theta(1)$



Recursivitate pe coadă (cont.)

- ▶ Numărul de **operații**: $\Theta(n)$
- ▶ **Spațiul** ocupat pe stivă: $\Theta(1)$
- ▶ În afară de economisirea spațiului, economisirea timpului necesar **redimensionării** stivei!



Recursivitate pe coadă (cont.)

- ▶ Numărul de **operații**: $\Theta(n)$
- ▶ **Spațiul** ocupat pe stivă: $\Theta(1)$
- ▶ În afară de economisirea spațiului, economisirea timpului necesar **redimensionării** stivei!
- ▶ Diferență față de iterarea clasică: transmiterea **explicită** a stării ca parametru



Funcții și procese

- ▶ Funcție: descriere **statică** a unor modalități de transformare



Funcții și procese

- ▶ Funcție: descriere **statică** a unor modalități de transformare
- ▶ Proces: Funcție în execuție, aspectul ei **dinamic**



Funcții și procese

- ▶ Funcție: descriere **statică** a unor modalități de transformare
- ▶ Proces: Funcție în execuție, aspectul ei **dinamic**
- ▶ Posibilitatea unei funcții textual **recursive** (e.g., pe coadă) de a genera un proces **iterativ**!



Funcția *Fibonacci*

Recursivitate pe stivă, arborescentă

```
36 (define (fib-stack n)
37   (cond [(= n 0) 0]
38         [(= n 1) 1]
39         [else (+ (fib-stack (- n 1))
40                  (fib-stack (- n 2)))]))
```



Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă

```
(fib 5)
```



Funcția *Fibonacci* (cont.)

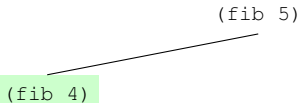
Recursivitate pe stivă, arborescentă

```
(fib 5)
```



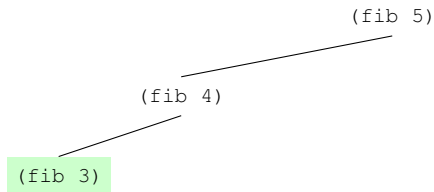
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



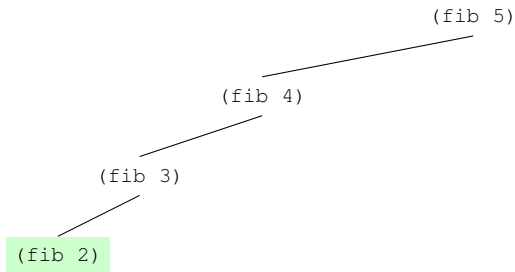
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



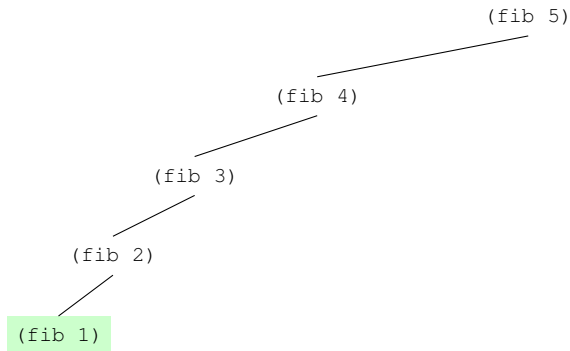
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



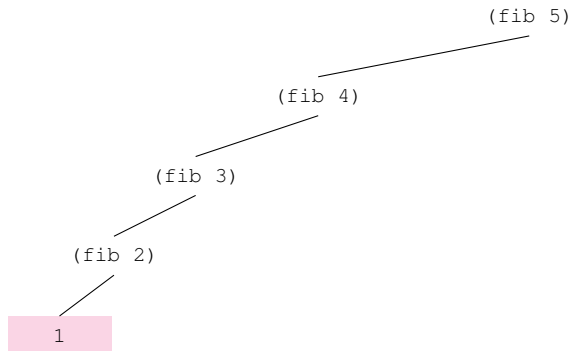
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



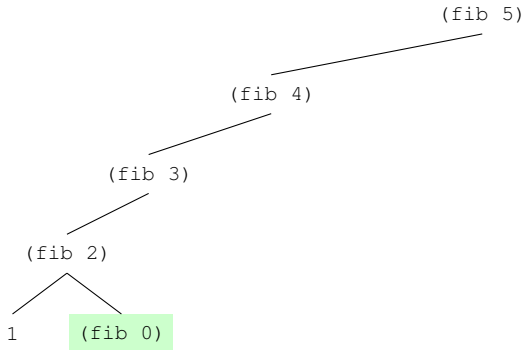
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



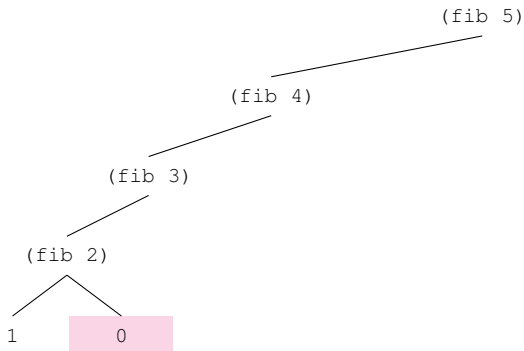
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



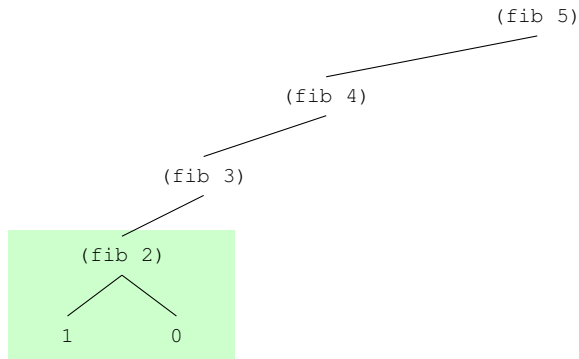
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



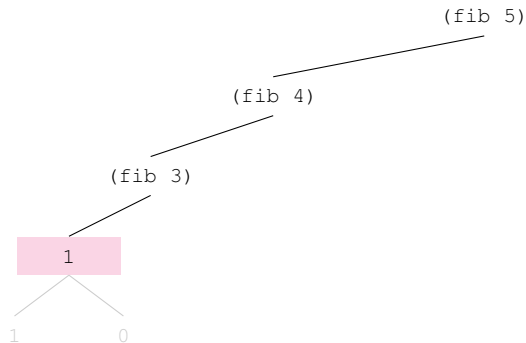
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



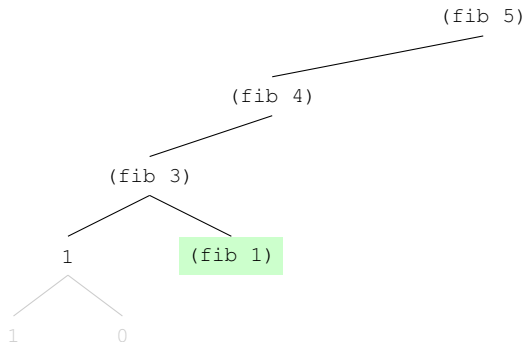
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



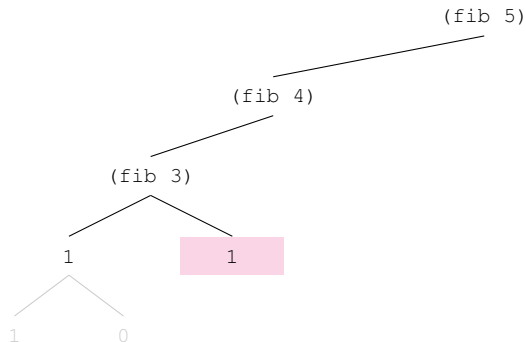
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



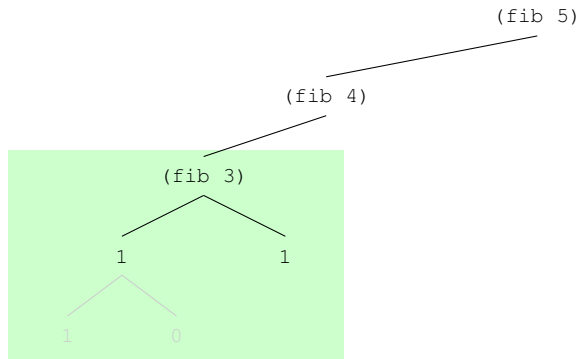
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



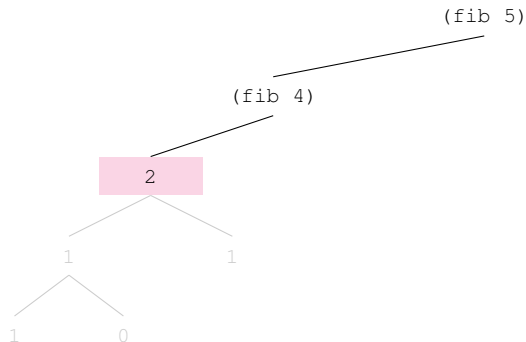
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



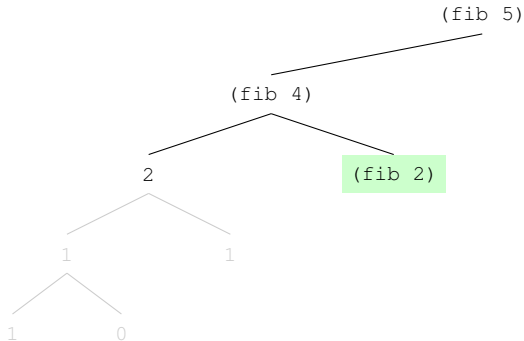
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



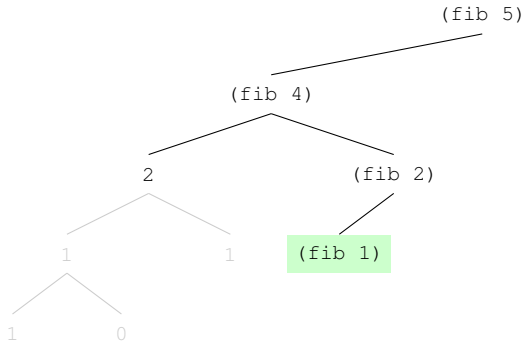
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



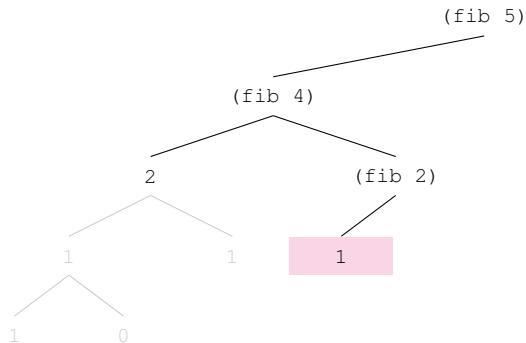
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



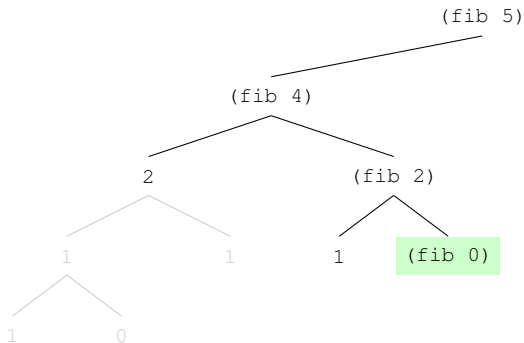
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



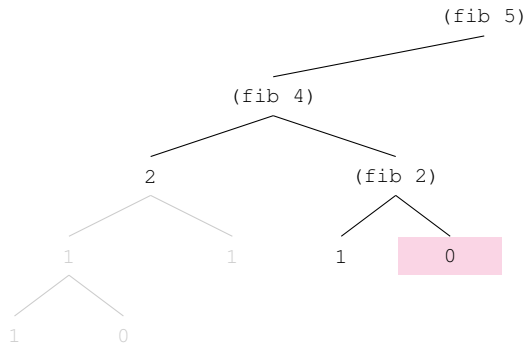
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



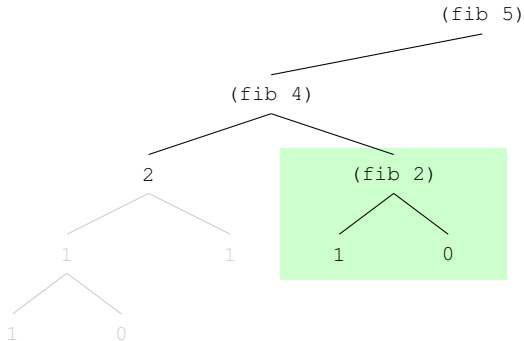
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



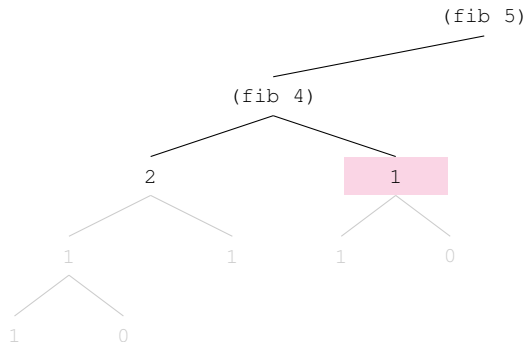
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



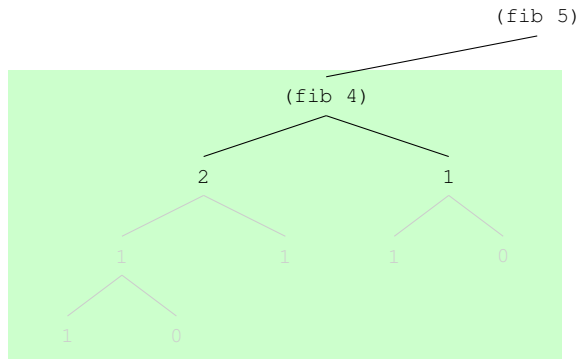
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



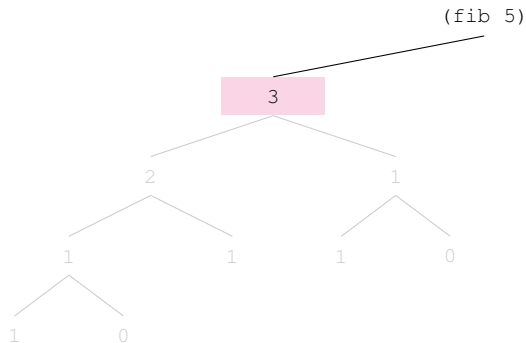
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



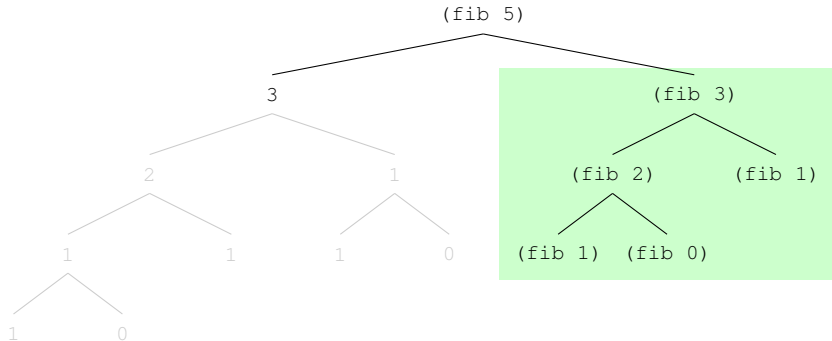
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



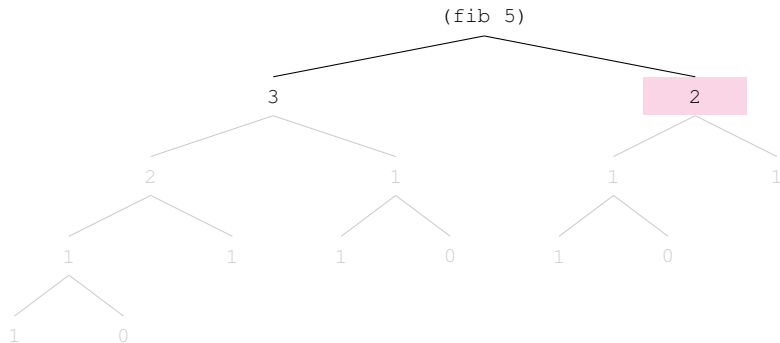
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



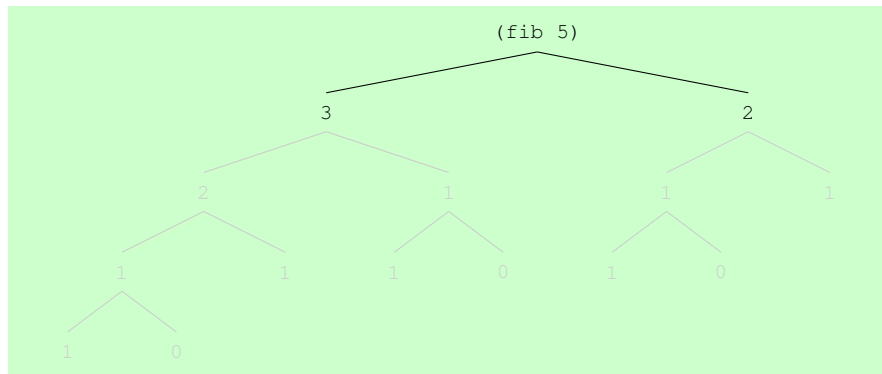
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



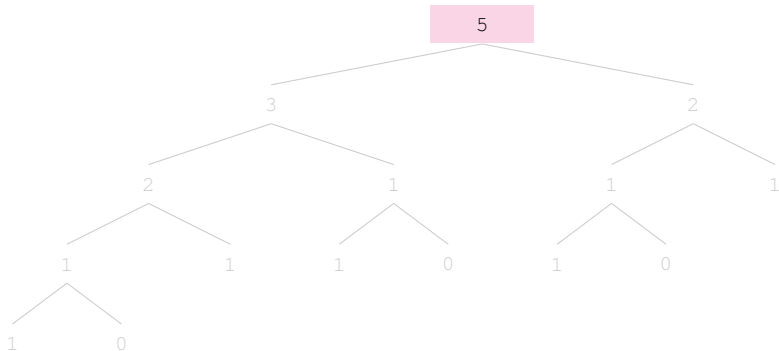
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



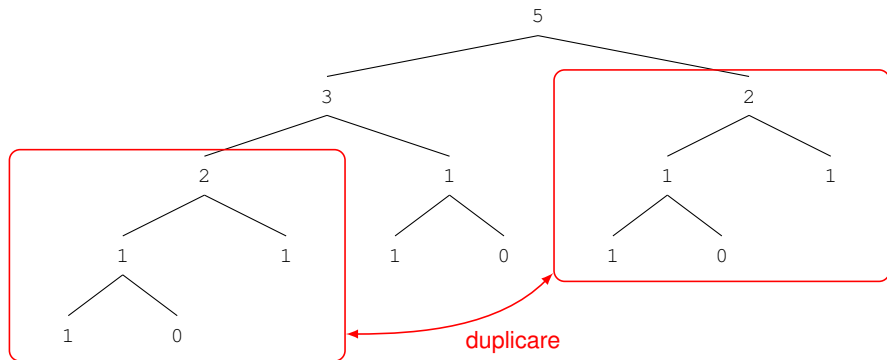
Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



Recursivitate pe stivă, arborescentă

- **Spațiul** ocupat pe stivă: lungimea unei căi din arbore:
 $\Theta(n)$



Recursivitate pe stivă, arborescentă

- ▶ **Spațiul** ocupat pe stivă: lungimea unei căi din arbore:
 $\Theta(n)$
- ▶ În arborele cu rădăcina $fib(n)$:

Recursivitate pe stivă, arborescentă

- ▶ **Spațiul** ocupat pe stivă: lungimea unei căi din arbore:
 $\Theta(n)$
- ▶ În arborele cu rădăcina $fib(n)$:
 - ▶ numărul frunzelor: $fib(n+1)$



Recursivitate pe stivă, arborescentă

- ▶ **Spațiul** ocupat pe stivă: lungimea unei căi din arbore:
 $\Theta(n)$
- ▶ În arborele cu rădăcina $fib(n)$:
 - ▶ numărul frunzelor: $fib(n+1)$
 - ▶ numărul nodurilor: $2fib(n+1) - 1$



Recursivitate pe stivă, arborescentă

- ▶ **Spațiul** ocupat pe stivă: lungimea unei căi din arbore:
 $\Theta(n)$
- ▶ În arborele cu rădăcina $fib(n)$:
 - ▶ numărul frunzelor: $fib(n+1)$
 - ▶ numărul nodurilor: $2fib(n+1) - 1$
- ▶ Numărul de **operații**: $\Theta(fib(n+1)) = \Theta(\phi^n)$
(ϕ — numărul de aur)



Recursivitate pe stivă, arborescentă

- ▶ **Spațiul** ocupat pe stivă: lungimea unei căi din arbore:
 $\Theta(n)$
- ▶ În arborele cu rădăcina $fib(n)$:
 - ▶ numărul frunzelor: $fib(n+1)$
 - ▶ numărul nodurilor: $2fib(n+1) - 1$
- ▶ Numărul de **operații**: $\Theta(fib(n+1)) = \Theta(\phi^n)$
(ϕ — numărul de aur)
- ▶ Creștere **exponențială** a numărului de operații!



Funcția *Fibonacci*

Recursivitate pe coadă

```
50 (define (fib-tail n)
51   (fib-tail-helper 1 0 n))
52
53 (define (fib-tail-helper a b count)
54   (if (= count 0)
55       b
56       (fib-tail-helper (+ a b) a (- count 1))))
```



Recursivitate pe coadă

- ▶ Numărul de operații: $\Theta(n)$



Recursivitate pe coadă

- ▶ Numărul de operații: $\Theta(n)$
- ▶ **Spațiul** ocupat pe stivă: $\Theta(1)$



Recursivitate pe coadă

- ▶ Numărul de operații: $\Theta(n)$
- ▶ **Spațiul** ocupat pe stivă: $\Theta(1)$
- ▶ Diminuarea numărului de operații de la exponențial la **liniar**!



Recursivitate pe stivă vs. pe coadă

Pe stivă, lin./arb.

- ▶ Elegantă, adesea apropiată de specificație

Pe coadă

- ▶ Obscură, necesitând prelucrări specifice

Recursivitate pe stivă vs. pe coadă

Pe stivă, lin./arb.

- ▶ Elegantă, adesea apropiată de specificație
- ▶ Ineficientă spațial și/ sau temporal

Pe coadă

- ▶ Obscură, necesitând prelucrări specifice
- ▶ Eficientă, cel puțin spațial

Recursivitate pe stivă vs. pe coadă

Pe stivă, lin./arb.

- ▶ Elegantă, adesea apropiată de specificație
- ▶ Ineficientă spațial și/ sau temporal

Pe coadă

- ▶ Obscură, necesitând prelucrări specifice
- ▶ Eficientă, cel puțin spațial

Câteva cursuri mai târziu — o modalitate de exploatare eficientă a recursivității pe stivă

Transformarea în recursivitate pe coadă

- ▶ De obicei, posibilă, prin introducerea unui **acumulator** ca parametru (v. exemplele anterioare)



Transformarea în recursivitate pe coadă

- ▶ De obicei, posibilă, prin introducerea unui **acumulator** ca parametru (v. exemplele anterioare)
- ▶ În anumite situații, **imposibilă** direct:

```
1  (define (f x)
2    (if (zero? x)
3        0
4        (g (f (- x 1)))))
5    ; comportamentul lui g depinde
6    ; de parametru
```



Cuprins

Introducere

Tipuri de recursivitate

Specificul recursivității pe coadă



Construirea rezultatului

Recursivitate pe stivă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-stack L)
3   (if (null? L)
4       L
5       (cons (* (car L) 10)
6             (mult-stack (cdr L)))))

1 (mult-stack '(1 2))
```



Construirea rezultatului

Recursivitate pe stivă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-stack L)
3   (if (null? L)
4       L
5       (cons (* (car L) 10)
6             (mult-stack (cdr L)))))

1 (mult-stack '(1 2))
```



Construirea rezultatului

Recursivitate pe stivă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-stack L)
3   (if (null? L)
4       L
5       (cons (* (car L) 10)
6             (mult-stack (cdr L)))))
```

```
1 (mult-stack '(1 2))
```

```
2 → (cons 10 (mult-stack '(2)))
```



Construirea rezultatului

Recursivitate pe stivă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-stack L)
3   (if (null? L)
4       L
5       (cons (* (car L) 10)
6             (mult-stack (cdr L)))))
```

```
1 (mult-stack '(1 2))
2 → (cons 10 (mult-stack '(2)))
```



Construirea rezultatului

Recursivitate pe stivă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-stack L)
3   (if (null? L)
4       L
5       (cons (* (car L) 10)
6             (mult-stack (cdr L)))))
```

```
1 (mult-stack '(1 2))
2 → (cons 10 (mult-stack '(2)))
3 → (cons 10 (cons 20 (mult-stack '()))) )
```



Construirea rezultatului

Recursivitate pe stivă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-stack L)
3   (if (null? L)
4       L
5       (cons (* (car L) 10)
6             (mult-stack (cdr L)))))
```

```
1 (mult-stack '(1 2))
2 → (cons 10 (mult-stack '(2)))
3 → (cons 10 (cons 20 (mult-stack '()))) )
```



Construirea rezultatului

Recursivitate pe stivă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-stack L)
3   (if (null? L)
4       L
5       (cons (* (car L) 10)
6             (mult-stack (cdr L)))))
```

```
1 (mult-stack '(1 2))
2 → (cons 10 (mult-stack '(2)))
3 → (cons 10 (cons 20 (mult-stack '())))
4 → (cons 10 (cons 20 '()))
```



Construirea rezultatului

Recursivitate pe stivă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-stack L)
3   (if (null? L)
4       L
5       (cons (* (car L) 10)
6             (mult-stack (cdr L)))))
```

```
1 (mult-stack '(1 2))
2 → (cons 10 (mult-stack '(2)))
3 → (cons 10 (cons 20 (mult-stack '())))
4 → (cons 10 (cons 20 '()))
```



Construirea rezultatului

Recursivitate pe stivă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-stack L)
3   (if (null? L)
4       L
5       (cons (* (car L) 10)
6             (mult-stack (cdr L)))))
```

```
1 (mult-stack '(1 2))
2 → (cons 10 (mult-stack '(2)))
3 → (cons 10 (cons 20 (mult-stack '())))
4 → (cons 10 (cons 20 '()))
5 → (cons 10 '(20))
```



Construirea rezultatului

Recursivitate pe stivă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-stack L)
3   (if (null? L)
4       L
5       (cons (* (car L) 10)
6             (mult-stack (cdr L)))))
```

```
1 (mult-stack '(1 2))
2 → (cons 10 (mult-stack '(2)))
3 → (cons 10 (cons 20 (mult-stack '())))
4 → (cons 10 (cons 20 '()))
5 → (cons 10 '(20))
```



Construirea rezultatului

Recursivitate pe stivă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-stack L)
3   (if (null? L)
4       L
5       (cons (* (car L) 10)
6             (mult-stack (cdr L)))))
```

```
1 (mult-stack '(1 2))
2 → (cons 10 (mult-stack '(2)))
3 → (cons 10 (cons 20 (mult-stack '())))
4 → (cons 10 (cons 20 '()))
5 → (cons 10 '(20))
6 → '(10 20)
```



Construirea rezultatului

Recursivitate pe stivă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-stack L)
3   (if (null? L)
4       L
5       (cons (* (car L) 10)
6             (mult-stack (cdr L)))))
```

```
1 (mult-stack '(1 2))
2 → (cons 10 (mult-stack '(2)))
3 → (cons 10 (cons 20 (mult-stack '())))
4 → (cons 10 (cons 20 '()))
5 → (cons 10 '(20))
6 → '(10 20) ; ordinea este corecta
```



Construirea rezultatului

Recursivitate pe coadă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-tail-helper L Result)
3   (if (null? L)
4       Result
5       (mult-tail-helper (cdr L)
6                           (cons (* (car L) 10)
7                                   Result))))
1  (mult-tail-helper '(1 2) '())
```



Construirea rezultatului

Recursivitate pe coadă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-tail-helper L Result)
3   (if (null? L)
4       Result
5       (mult-tail-helper (cdr L)
6                          (cons (* (car L) 10)
6                                Result))))
1 (mult-tail-helper '(1 2) '())
```



Construirea rezultatului

Recursivitate pe coadă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-tail-helper L Result)
3   (if (null? L)
4       Result
5       (mult-tail-helper (cdr L)
6                           (cons (* (car L) 10)
7                                   Result))))
```

```
1 (mult-tail-helper '(1 2) '())
```

```
2 → (mult-tail-helper '(2) '(10))
```



Construirea rezultatului

Recursivitate pe coadă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-tail-helper L Result)
3   (if (null? L)
4       Result
5       (mult-tail-helper (cdr L)
6                           (cons (* (car L) 10)
7                                   Result))))
```

```
1 (mult-tail-helper '(1 2) '())
```

```
2 → (mult-tail-helper '(2) '(10))
```



Construirea rezultatului

Recursivitate pe coadă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-tail-helper L Result)
3   (if (null? L)
4       Result
5       (mult-tail-helper (cdr L)
6                           (cons (* (car L) 10)
7                                   Result))))
```

```
1 (mult-tail-helper '(1 2) '())
```

```
2 → (mult-tail-helper '(2) '(10))
```

```
3 → (mult-tail-helper '() '(20 10))
```



Construirea rezultatului

Recursivitate pe coadă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-tail-helper L Result)
3   (if (null? L)
4       Result
5       (mult-tail-helper (cdr L)
6                           (cons (* (car L) 10)
7                                   Result))))
```

```
1 (mult-tail-helper '(1 2) '())
```

```
2 → (mult-tail-helper '(2) '(10))
```

```
3 → (mult-tail-helper '() '(20 10))
```



Construirea rezultatului

Recursivitate pe coadă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-tail-helper L Result)
3   (if (null? L)
4       Result
5       (mult-tail-helper (cdr L)
6                           (cons (* (car L) 10)
7                                   Result))))
```

```
1 (mult-tail-helper '(1 2) '())
2 → (mult-tail-helper '(2) '(10))
3 → (mult-tail-helper '() '(20 10))
4 → '(20 10)
```



Construirea rezultatului

Recursivitate pe coadă

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 (define (mult-tail-helper L Result)
3   (if (null? L)
4       Result
5       (mult-tail-helper (cdr L)
6                           (cons (* (car L) 10)
7                                   Result))))
```

```
1 (mult-tail-helper '(1 2) '())
2 → (mult-tail-helper '(2) '(10))
3 → (mult-tail-helper '() '(20 10))
4 → '(20 10) ; ordinea este inversata
```



Construirea rezultatului (cont.)

Recursivitate pe coadă

Alternative pentru **conservarea** ordinii:



Construirea rezultatului (cont.)

Recursivitate pe coadă

Alternative pentru **conservarea** ordinii:

- ▶ **Inversarea** listei finale

```
1  (if (null? L)
2      (reverse Result)
3      ...)
```



Construirea rezultatului (cont.)

Recursivitate pe coadă

Alternative pentru **conservarea** ordinii:

- **Inversarea** listei finale

```
1  (if (null? L)
2      (reverse Result)
3      ...)
```

- Adăugarea elementului curent la **sfârșitul** acumul.

```
1  (if (null? L)
2      ...
3      (mult-all-iter
4          (cdr L)
5          (append Result
6                  (list (* (car L) 10)))))
```



Costul unei concatenări

```
1 (define (app A B) ; recursiva pe stiva
2   (if (null? A)
3       B
4       (cons (car A) (app (cdr A) B))))
```



Costul unei concatenări

```
1 (define (app A B) ; recursiva pe stiva
2   (if (null? A)
3       B
4       (cons (car A) (app (cdr A) B))))
```

Număr de operații proporțional cu lungimea **primei** liste!



Costul concatenărilor repetate

- ▶ Asociere la **dreapta**:

$A \ ++ \ (B \ ++ \ (C \ ++ \ \dots) \ \dots)$



Costul concatenărilor repetate

- Asociere la dreapta:

$A \mathrel{++} (B \mathrel{++} (C \mathrel{++} \dots) \dots)$

Număr de operații proporțional cu lungimea listei
curente



Costul concatenărilor repetate

- Asociere la **dreapta**:

$A \ ++ \ (B \ ++ \ (C \ ++ \ \dots) \ \dots)$

Număr de operații proporțional cu lungimea listei
curente

- Asociere la **stânga**:

$(\dots (\dots \ ++ \ A) \ ++ \ B) \ ++ \ C$



Costul concatenărilor repetate

- Asociere la **dreapta**:

$$A \ ++ \ (B \ ++ \ (C \ ++ \ \dots) \ \dots)$$

Număr de operații proporțional cu lungimea listei
curente

- Asociere la **stânga**:

$$(\dots (\dots \ ++ \ A) \ ++ \ B) \ ++ \ C$$

Număr de operații proporțional cu lungimea **tuturor**
listelor concatenate anterior



Consecințe asupra recursivității pe coadă

```
1 (define (mult-tail-helper L Result)
2   (if (null? L)
3       Result
4       (mult-tail-helper
5         (cdr L)
6         (append Result
7                 (list (* (car L) 10))))))
```

```
1 (mult-tail-helper '(1 2 3) '())
2 → (mult-tail-helper '(2 3) (append '() '(10)))
3 → (mult-tail-helper '(3) (append '(10) '(20)))
4 → (mult-tail-helper '() (append '(10 20)
5                                  '(30)))
6 → (mult-tail-helper '() '(10 20 30))
7 → '(10 20 30)
```



Consecințe asupra recursivității pe coadă (cont.)

- ▶ Parcurgerea **întregului** acumulator anterior, pentru construirea celui nou!



Consecințe asupra recursivității pe coadă (cont.)

- ▶ Parcurgerea **întregului** acumulator anterior, pentru construirea celui nou!
- ▶ Numărul de elemente parcurse:

$$0 + 1 + \dots + (n - 1) = \Theta(n^2)!$$



Consecințe asupra recursivității pe coadă (cont.)

- ▶ Parcurgerea **întregului** acumulator anterior, pentru construirea celui nou!
- ▶ Numărul de elemente parcurse:

$$0 + 1 + \dots + (n - 1) = \Theta(n^2)!$$

- ▶ Astfel, preferabilă varianta **inversării**, și nu cea a adăugării la sfârșit



Rezumat

- ▶ Diverse **tipuri** de recursivitate
 - ▶ pe stivă (liniară/ arborescentă)
 - ▶ pe coadă
- ▶ Recursivitate pe **stivă**: de obicei, ...
 - ▶ Elegantă
 - ▶ Ineficientă spațial și/ sau temporal
- ▶ Recursivitate pe **coadă**: de obicei, ...
 - ▶ Mai puțin lizibilă decât cea pe stivă
 - ▶ Necesită prelucrări suplimentare (e.g. inversare)
 - ▶ Eficientă spațial și/ sau temporal



Bibliografie

Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition.

Partea IV

Funcții ca valori de prim rang.
Funcționale



Cuprins

Motivație

Funcții ca valori de prim rang

Funcționale

Calculul lambda



Cuprins

Motivație

Funcții ca valori de prim rang

Funcționale

Calculul lambda



Abstractizare funcțională

```
1 (define (double n)
2   (* n 2))
```



Abstractizare funcțională

```
1 (define (double n)
2   (* n 2))
```

(* 5 2) (* 10 2)

- Generalizare, de la dublarea valorilor particulare, la însuși **conceptul** de *dublare*



Abstractizare funcțională

```
1 (define (double n)
2   (* n 2))
```

(* 5 2) (* 10 2)

- ▶ Generalizare, de la dublarea valorilor particulare, la însuși **conceptul** de *dublare*
- ▶ Rezultat: funcția `double`



Abstractizare funcțională

```
1 (define (double n)
2   (* n 2))
```

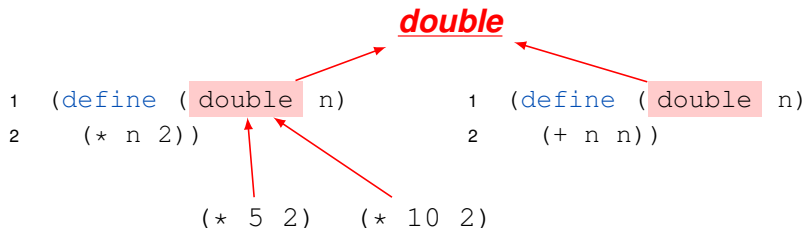
(* 5 2) (* 10 2)

```
1 (define (double n)
2   (+ n n))
```

- ▶ Generalizare, de la dublarea valorilor particulare, la însuși **conceptul** de *dublare*
- ▶ Rezultat: funcția `double`, **substituibilă** cu orice altă funcție cu același comportament



Abstractizare funcțională



- ▶ Generalizare, de la dublarea valorilor particulare, la însuși **conceptul** de *dublare*
- ▶ Rezultat: funcția `double`, **substituibilă** cu orice altă funcție cu același comportament
- ▶ Mai precis, `double` = ***abstractizare funcțională***



Un nivel mai sus

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 ;; '(1 2 3) -> '(10 20 30)
3 (define (mult L)
4   (if (null? L)
5       L
6       (cons (* (car L) 10)
7             (mult (cdr L)))))
```



Un nivel mai sus

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 ;; '(1 2 3) -> '(10 20 30)
3 (define (mult L)
4   (if (null? L)
5       L
6       (cons (* (car L) 10)
7             (mult (cdr L))))))
8
9 ;; Obtine paritatea fiecarui numar (true = par)
10 ;; '(1 2 3) -> '(false true false)
11 (define (parities L)
12   (if (null? L)
13       L
14       (cons (even? (car L))
15             (parities (cdr L)))))
15
```



Un nivel mai sus

```
1 ;; Inmulteste cu 10 toate elementele listei L
2 ;; '(1 2 3) -> '(10 20 30)
3 (define (mult L)
4   (if (null? L)
5       L
6       (cons (* (car L) 10)
7             (mult (cdr L)))))
8
9 ;; Obtine paritatea fiecarui numar (true = par)
10 ;; '(1 2 3) -> '(false true false)
11 (define (parities L)
12   (if (null? L)
13       L
14       (cons (even? (car L))
15             (parities (cdr L)))))
15
```

singura parte
variabilă,
dependentă
de (car L)



Un nivel mai sus (cont.)

Cum putem **izola** transformarea lui $(\text{car } L)$?



Un nivel mai sus (cont.)

Cum putem **izola** transformarea lui (car L) ?
Prin **funcții**!

```
1 ;; map = asociere
2
3 (define (mult-map x)
4   (* x 10))
5
6 (define (parities-map x)
7   (even? x))
```



Un nivel mai sus (cont.)

Cum putem **izola** transformarea lui `(car L)`?
Prin **funcții**!

```
1 ;; map = asociere
```

```
2
```

```
3 (define (mult-map x)
```

```
4   (* x 10))
```

```
5
```

```
6 (define (parities-map x)
```

```
7   (even? x))
```

rolul lui
(car L)



Un nivel mai sus (cont.)

```
1 (define (map f L)
2   (if (null? L)
3       L
4       (cons (f (car L))
5              (map f (cdr L))))))
```



Un nivel mai sus (cont.)

```
1 (define (map f L)
2   (if (null? L)
3       L
4       (cons (f (car L))
5             (map f (cdr L)))))
```

transformarea
lui (car L):
parametru



Un nivel mai sus (cont.)

```
1 (define (map f L)
2   (if (null? L)
3       L
4       (cons (f (car L))
5             (map f (cdr L)))))
6
7 (define (mult L)
8   (map mult-map L))
9
10 (define (parities L)
11   (map parities-map L))
```

transformarea
lui (car L):
parametru



Un nivel mai sus (cont.)

```
1 (define (map f L)
2   (if (null? L)
3       L
4       (cons (f (car L))
5             (map f (cdr L)))))
6
7 (define (mult L)
8   (map mult-map L))
9
10 (define (parities L)
11   (map parities-map L))
```

transformarea
lui (car L):
parametru

Generalizare, de la diversele transformări ale listelor,
la **conceptul** de transformare element cu element,
independent de natura acesteia — *asociere (mapping)*



Cuprins

Motivație

Funcții ca valori de prim rang

Funcționale

Calculul lambda



Funcții ca valori de prim rang

- ▶ În exemplele anterioare: funcții văzute ca **date**!



Funcții ca valori de prim rang

- ▶ În exemplele anterioare: funcții văzute ca **date**!
- ▶ Avantaj: sporire considerabilă a **expresivității** limbajului



Funcții ca valori de prim rang

- ▶ În exemplele anterioare: funcții văzute ca **date**!
- ▶ Avantaj: sporire considerabilă a **expresivității** limbajului
- ▶ Statutul de **valori** de prim rang al funcțiilor, acestea putând fi:
 - ▶ create **dinamic** (la execuție)
 - ▶ **numite**
 - ▶ trimise ca **parametri** unei funcții
 - ▶ **întoarse** dintr-o funcție



Evaluarea funcțiilor

Ca valori, evaluate la ele **însele**!



Evaluarea funcțiilor

Ca valori, evaluate la ele **însele**!

1 > +



Evaluarea funcțiilor

Ca valori, evaluate la ele **însele!**

```
1 > +
```

```
2 #<procedure:+>
```



Evaluarea funcțiilor

Ca valori, evaluate la ele **însele**!

```
1 > +  
2 #<procedure:+>  
3  
4 > (cons + '(1 2))
```



Evaluarea funcțiilor

Ca valori, evaluate la ele **însele**!

```
1 > +  
2 #<procedure: +>  
3  
4 > (cons + '(1 2))  
5 (#<procedure: +> 1 2)
```



Evaluarea funcțiilor

Ca valori, evaluate la ele **însele!**

```
1 > +  
2 #<procedure:+>  
3  
4 > (cons + '(1 2))  
5 (#<procedure:+> 1 2)  
6  
7 > (list + - *)
```



Evaluarea funcțiilor

Ca valori, evaluate la ele **însele**!

```
1 > +
2 #<procedure:+>
3
4 > (cons + '(1 2))
5 (#<procedure:+> 1 2)
6
7 > (list + - *)
8 (#<procedure:+> #<procedure:-> #<procedure:*>)
```



Funcții ca parametru

- ▶ În exemplele anterioare, funcții definite separat, deși folosite o **singură** dată:

```
1  (define (mult L)
2    (map mult-map L))
3
4  (define (parities L)
5    (map parities-map L))
```



Funcții ca parametru

- ▶ În exemplele anterioare, funcții definite separat, deși folosite o **singură** dată:

```
1 (define (mult L)
2   (map mult-map L))
3
4 (define (parities L)
5   (map parities-map L))
```

- ▶ Putem defini funcțiile **local** unei expresii?



Funcții anonime

```
1 (define (mult L)
2   (map (lambda (x) (* x 10)) L))
3
4 (define (parities L)
5   (map (lambda (x) (even? x)) L))
```



Funcții anonime

constructor

```
1 (define (mult L)
2   (map (lambda (x) (* x 10)) L))
3
4 (define (parities L)
5   (map (lambda (x) (even? x)) L))
```



Funcții anonime

constructor

parametru

```
1 (define (mult L)
2   (map (lambda (x) (* x 10)) L))
3
4 (define (parities L)
5   (map (lambda (x) (even? x)) L))
```



Funcții anonime

constructor

parametru

corp

```
1 (define (mult L)
2   (map (lambda (x) (* x 10)) L))
3
4 (define (parities L)
5   (map (lambda (x) (even? x)) L))
```



Funcții anonime

constructor parametru corp

```
1 (define (mult L)
2   (map (lambda (x) (* x 10)) L))
3
4 (define (parities L)
5   (map (lambda (x) (even? x)) L))
```

De fapt,

1 (define (mult-map x)	≡	1 (define mult-map
2 (* x 10))		2 (lambda (x)
3		3 (* x 10)))



Funcții anonime

constructor parametru corp

```
1 (define (mult L)
2   (map (lambda (x) (* x 10)) L))
3
4 (define (parities L)
5   (map (lambda (x) (even? x)) L))
```

De fapt,

1 (define (mult-map x)	≡	1 (define mult-map
2 (* x 10))		2 (lambda (x)
3		3 (* x 10)))

simpla **legare** a variabilei `mult-map` la o funcție anonimă



Funcții ca valori de retur

- ▶ În exemplul cu funcția `mult`, cum înmulțim toate elementele listei cu un număr **oarecare**, nu neapărat cu 10?



Funcții ca valori de retur

- ▶ În exemplul cu funcția `mult`, cum înmulțim toate elementele listei cu un număr **oarecare**, nu neapărat cu 10?
- ▶ Posibilă utilizare, pentru înmulțirea cu 5:

```
1 (map mult-map '(1 2 3))
```

 **funcție**



Funcții ca valori de retur

- ▶ În exemplul cu funcția `mult`, cum înmulțim toate elementele listei cu un număr **oarecare**, nu neapărat cu 10?
- ▶ Posibilă utilizare, pentru înmulțirea cu 5:

```
1 (map (mult-map-by 5) '(1 2 3))
```

↑ funcție



Funcții ca valori de retur

- ▶ În exemplul cu funcția `mult`, cum înmulțim toate elementele listei cu un număr **oarecare**, nu neapărat cu 10?

- ▶ Posibilă utilizare, pentru înmulțirea cu 5:

```
1  (map (mult-map-by 5) '(1 2 3))
```

↑ funcție

- ▶ Cum aplicăm `mult-map-by` doar asupra **primului** parametru?

```
1  (define (mult-map-by q x)
2    (* x q))
3
```



Funcții ca valori de retur

- ▶ În exemplul cu funcția `mult`, cum înmulțim toate elementele listei cu un număr **oarecare**, nu neapărat cu 10?
- ▶ Posibilă utilizare, pentru înmulțirea cu 5:

```
1 (map (mult-map-by 5) '(1 2 3))
```

↑ funcție

- ▶ Cum aplicăm `mult-map-by` doar asupra **primului** parametru?

```
1 (define (mult-map-by q x)
2   (* x q))
3
```

```
1 (define (mult-map-by q)
2   (lambda (x)
3     (* x q)))
```



Funcții ca valori de retur

- ▶ În exemplul cu funcția `mult`, cum înmulțim toate elementele listei cu un număr **oarecare**, nu neapărat cu 10?
- ▶ Posibilă utilizare, pentru înmulțirea cu 5:

```
1 (map (mult-map-by 5) '(1 2 3))
```

↑ funcție

- ▶ Cum aplicăm `mult-map-by` doar asupra **primului** parametru?

```
1 (define (mult-map-by q x)
2   (* x q))
3
```

↑
simultan
(**uncurried**)

```
1 (define (mult-map-by q)
2   (lambda (x)
3     (* x q)))
```



Funcții ca valori de retur

- ▶ În exemplul cu funcția `mult`, cum înmulțim toate elementele listei cu un număr **oarecare**, nu neapărat cu 10?
- ▶ Posibilă utilizare, pentru înmulțirea cu 5:

```
1 (map (mult-map-by 5) '(1 2 3))
```

↑ funcție

- ▶ Cum aplicăm `mult-map-by` doar asupra **primului** parametru?

```
1 (define (mult-map-by q x)
2   (* x q))
3
```

↑
simultan
(*uncurried*)

```
1 (define (mult-map-by q)
2   (lambda (x)
3     (* x q)))
```

↑
pe rând
(*curried*)



Funcții ca valori de retur

- ▶ În exemplul cu funcția `mult`, cum înmulțim toate elementele listei cu un număr **oarecare**, nu neapărat cu 10?
- ▶ Posibilă utilizare, pentru înmulțirea cu 5:

```
1 (map (mult-map-by 5) '(1 2 3))
```

↑ funcție

- ▶ Cum aplicăm `mult-map-by` doar asupra **primului** parametru?

```
1 (define (mult-map-by q x)
2   (* x q))
3
```

simultan
(**uncurried**)

```
1 (define (mult-map-by q)
2   (lambda (x)
3     (* x q)))
```

pe rând
(**curried**)



Secvențierea parametrilor

- ▶ În loc să afirmăm că `mult-map-by` are **un** parametru și că întoarce o funcție, ne “prefacem” că primește **doi** parametri, pe rând



Secvențierea parametrilor

- ▶ În loc să afirmăm că `mult-map-by` are **un** parametru și că întoarce o funcție, ne “prefacem” că primește **doi** parametri, pe rând
- ▶ Avantaj: **reutilizare**, prin aplicare **parțială**!



Secvențierea parametrilor

- ▶ În loc să afirmăm că `mult-map-by` are **un** parametru și că întoarce o funcție, ne “prefacem” că primește **doi** parametri, pe rând
- ▶ Avantaj: **reutilizare**, prin aplicare **parțială**!
- ▶ Funcție *curried*: preia parametrii **pe rând** (aparent)



Secvențierea parametrilor

- ▶ În loc să afirmăm că `mult-map-by` are **un** parametru și că întoarce o funcție, ne “prefacem” că primește **doi** parametri, pe rând
- ▶ Avantaj: **reutilizare**, prin aplicare **parțială**!
- ▶ Funcție *curried*: preia parametrii **pe rând** (aparent)
- ▶ Funcție *uncurried*: preia parametrii **simultan**



Extinderea regulilor de evaluare

```
1 ( (if true + -) (+ 1 2) 3)
```



Extinderea regulilor de evaluare

- Din moment ce funcțiile sunt valori posibile ale expresiilor, necesitatea evaluării inclusiv a **operatorului** unei aplicații

```
1 ( (if true + -) (+ 1 2) 3)
```



Extinderea regulilor de evaluare

- Din moment ce funcțiile sunt valori posibile ale expresiilor, necesitatea evaluării inclusiv a **operatorului** unei aplicații

```
1 ( (if true + -) (+ 1 2) 3)
2 → ( + (+ 1 2) 3)
```



Extinderea regulilor de evaluare

- ▶ Din moment ce funcțiile sunt valori posibile ale expresiilor, necesitatea evaluării inclusiv a **operatorului** unei aplicații
- ▶ Mai departe, evaluarea variabilei **+** la valoarea ei — funcția de adunare!

```
1 ( (if true + -) (+ 1 2) 3)
2 → ( + (+ 1 2) 3)
```



Extinderea regulilor de evaluare

- ▶ Din moment ce funcțiile sunt valori posibile ale expresiilor, necesitatea evaluării inclusiv a **operatorului** unei aplicații
- ▶ Mai departe, evaluarea variabilei **+** la valoarea ei — funcția de adunare!

```
1 ( (if true + -) (+ 1 2) 3)
2 → ( + (+ 1 2) 3)
3 → ( #<procedure:+> (+ 1 2) 3)
```



Extinderea regulilor de evaluare

- ▶ Din moment ce funcțiile sunt valori posibile ale expresiilor, necesitatea evaluării inclusiv a **operatorului** unei aplicații
- ▶ Mai departe, evaluarea variabilei **+** la valoarea ei — funcția de adunare!

```
1 ( (if true + -) (+ 1 2) 3)
2 → ( + (+ 1 2) 3)
3 → ( #<procedure:+> (+ 1 2) 3)
```

Notă: Pasul de evaluare 2–3 nu transpune la utilizarea *stepper*-ului din Racket, dar este prezent pe slide pentru completitudine.



Aplicație: compunerea a două funcții

```
1 (define (comp f g)
2   (lambda (x)
3     (f (g x))))
4
5 ((comp car cdr) '(1 2 3)) → 2
```



Cuprins

Motivație

Funcții ca valori de prim rang

Funcționale

Calculul lambda



Funcționale

- ▶ Funcțională = funcție care primește ca parametru și/ sau întoarce o funcție



Funcționale

- ▶ Funcțională = funcție care primește ca parametru și/ sau întoarce o funcție
- ▶ Surprind metode generale de prelucrare



Funcționale

- ▶ Funcțională = funcție care primește ca parametru și/ sau întoarce o funcție
- ▶ Surprind metode generale de prelucrare
- ▶ Funcționale standard în majoritatea limbajelor funcționale (prezentate în continuare):



Funcționale

- ▶ Funcțională = funcție care primește ca parametru și/ sau întoarce o funcție
- ▶ Surprinz metode generale de prelucrare
- ▶ Funcționale standard în majoritatea limbajelor funcționale (prezentate în continuare):
 - ▶ map



Funcționale

- ▶ Funcțională = funcție care primește ca parametru și/ sau întoarce o funcție
- ▶ Surprinz metode generale de prelucrare
- ▶ Funcționale standard în majoritatea limbajelor funcționale (prezentate în continuare):
 - ▶ `map`
 - ▶ `filter`



Funcționale

- ▶ Funcțională = funcție care primește ca parametru și/ sau întoarce o **funcție**
- ▶ Surprinz metode **generale** de prelucrare
- ▶ Funcționale **standard** în majoritatea limbajelor funcționale (prezentate în continuare):
 - ▶ `map`
 - ▶ `filter`
 - ▶ `foldl` (*fold left*)



Funcționale

- ▶ Funcțională = funcție care primește ca parametru și/ sau întoarce o funcție
- ▶ Surprinz metode generale de prelucrare
- ▶ Funcționale standard în majoritatea limbajelor funcționale (prezentate în continuare):
 - ▶ `map`
 - ▶ `filter`
 - ▶ `foldl` (*fold left*)
 - ▶ `foldr` (*fold right*)



Funcționala `map`

- ▶ Aplicarea unei **transformări** asupra tuturor elementelor unei liste
- ▶ Tratată anterior

```
1 (map (lambda (x) (* x 10)) '(1 2 3))  
2 → '(10 20 30)
```



Funcționala `filter`

- ▶ Extragerea dintr-o listă a elementelor care **satisfac** un predicat logic
- ▶ Funcția primită ca parametru trebuie să întoarcă o valoare **booleană**

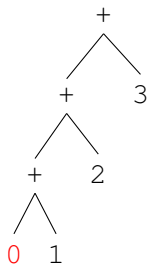
```
1 (filter even? '(1 2 3))  
2 → '(2)
```



Funcționala `foldl`

- ▶ Acumularea tuturor elementelor unei liste sub forma unei **singure** valori (posibil tot listă, dar nu exclusiv)
- ▶ Pacurgere stânga $\rightarrow$ dreapta
- ▶ Utilizarea unei funcții **binare** element-acumulator
- ▶ Pornire cu un acumulator **inițial**
- ▶ Natural recursivă pe ...

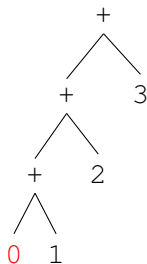
```
1 (foldl + 0 '(1 2 3))  
2  $\rightarrow$  6
```



Funcționala `foldl`

- ▶ Acumularea tuturor elementelor unei liste sub forma unei **singure** valori (posibil tot listă, dar nu exclusiv)
- ▶ Pacurgere stânga $\rightarrow$ dreapta
- ▶ Utilizarea unei funcții **binare** element-acumulator
- ▶ Pornire cu un acumulator **inițial**
- ▶ Natural recursivă pe **coadă**

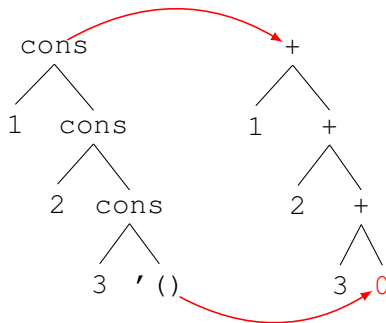
```
1 (foldl + 0 ' (1 2 3))  
2  $\rightarrow$  6
```



Funcționala `foldr`

- ▶ Similar cu `foldl`
- ▶ Pacurgere dreapta $\rightarrow$ stânga
- ▶ Operare pe **structura** listei inițiale
- ▶ Natural recursivă pe ...

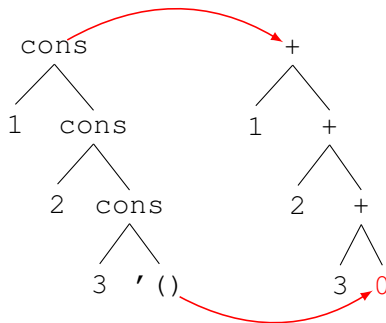
```
1 (foldr + 0 '(1 2 3))  
2  $\rightarrow$  6
```



Funcționala `foldr`

- ▶ Similar cu `foldl`
- ▶ Pacurgere dreapta $\rightarrow$ stânga
- ▶ Operare pe **structura** listei inițiale
- ▶ Natural recursivă pe **stivă**

```
1 (foldr + 0 '(1 2 3))  
2  $\rightarrow$  6
```



Universalitatea funcționalelor `fold*`

- ▶ **Orice** funcție primitiv recursivă pe liste,
implementabilă în termenii funcționalelor `fold*`



Universalitatea funcționalelor `fold*`

- ▶ **Orice** funcție primitiv recursivă pe liste, implementabilă în termenii funcționalelor `fold*`
- ▶ În particular, utilizabile pentru implementarea funcționalelor `map` și `filter`!



Cuprins

Motivație

Funcții ca valori de prim rang

Funcționale

Calculul lambda



Trăsături

- ▶ Model de **calculabilitate** — Alonzo Church, 1932
- ▶ Centrat pe conceptul de **funcție**
- ▶ Calculul: evaluarea aplicațiilor de funcții, prin **substituție** textuală



Evaluare

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

“Pentru a aplica funcția $\lambda x.x$

Evaluare

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

“Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual,
 y ,

Evaluare

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

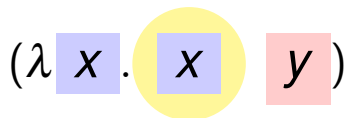
“Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, y , se identifică parametrul formal, x ,

Evaluare

$$(\lambda \boxed{x} . \bigcirc x \boxed{y})$$

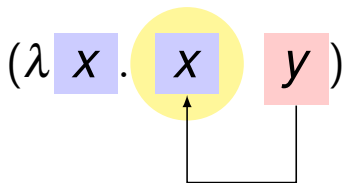
“Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, y , se indentifică parametrul formal, x , în corpul funcției, x ,

Evaluare

$$(\lambda x. x \ y)$$
The diagram shows the lambda expression $(\lambda x. x \ y)$. The lambda symbol λ is black. The first x is enclosed in a light blue square. The dot $.$ is black. The second x is enclosed in a light blue square and is also inside a larger yellow circle. The space between the two x 's contains a black dot. The y is enclosed in a light red square. The closing parenthesis $)$ is black.

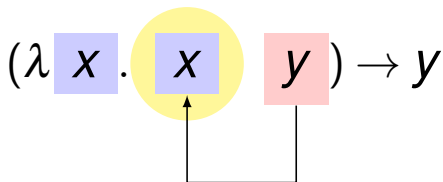
“Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, y , se indentifică parametrul formal, x , în corpul funcției, x , iar aparițiile primului, x (singura),

Evaluare



“Pentru a aplica funcția $\lambda x. x$ asupra parametrului actual, y , se indentifică parametrul formal, x , în corpul funcției, x , iar aparițiile primului, x (singura), se **substituie** cu parametrul actual,

Evaluare



“Pentru a aplica funcția $\lambda x. x$ asupra parametrului actual, y , se identifică parametrul formal, x , în corpul funcției, x , iar aparițiile primului, x (singura), se **substituie** cu parametrul actual, obținându-se rezultatul unui pas de evaluare.”

Formalizarea substituției

În expresia $(\lambda x. \lambda x. y \ y)$:



Formalizarea substituției

În expresia $(\lambda x. \lambda \boxed{x}. y \boxed{y})$:



- Aplicarea mecanică a principiului substituției: $\lambda \textcolor{red}{y}. y$

Formalizarea substituției

În expresia $(\lambda x. \lambda \boxed{x}. y \boxed{y})$:



- ▶ Aplicarea mecanică a principiului substituției: $\lambda \textcolor{red}{y}. y$
- ▶ Intuitiv: $\lambda \textcolor{red}{x}. y$

Formalizarea substituției

În expresia $(\lambda x. \lambda \boxed{x}. y \boxed{y})$:



- ▶ Aplicarea mecanică a principiului substituției: $\lambda y. y$
- ▶ Intuitiv: $\lambda x. y$
- ▶ Rezultat **eronat** al abordării mecanice!

Formalizarea substituției

În expresia $(\lambda x. \lambda \boxed{x}. y \boxed{y})$:



- ▶ Aplicarea mecanică a principiului substituției: $\lambda y. y$
- ▶ Intuitiv: $\lambda x. y$
- ▶ Rezultat **eronat** al abordării mecanice!
- ▶ **Ce** ar trebui substituit de fapt?

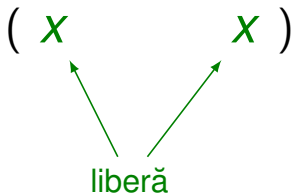
Apariții libere și legate ale variabilelor

x

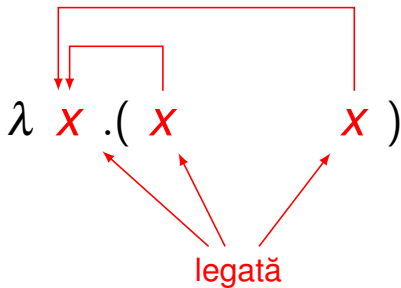
Apariții libere și legate ale variabilelor

x
↑
liberă

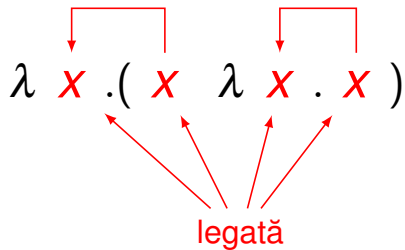
Apariții libere și legate ale variabilelor



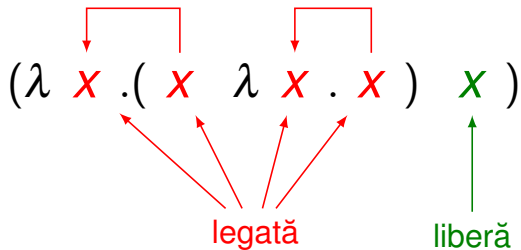
Apariții libere și legate ale variabilelor



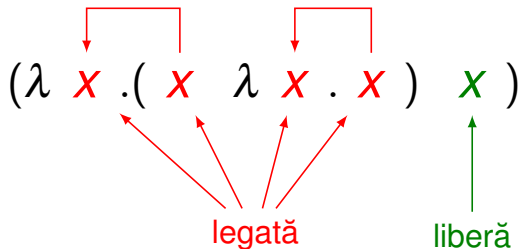
Apariții libere și legate ale variabilelor



Apariții libere și legate ale variabilelor

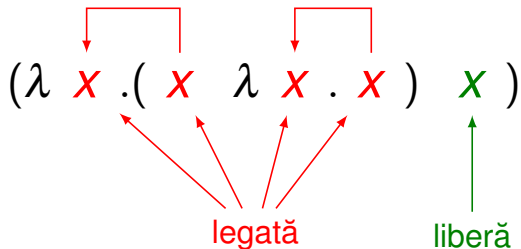


Apariții libere și legate ale variabilelor



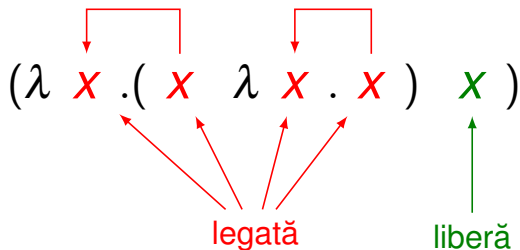
- Apariție **legată** a lui x :

Apariții libere și legate ale variabilelor



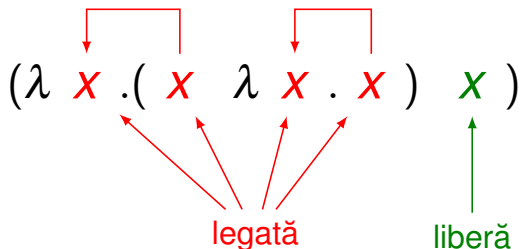
- ▶ Apariție **legată** a lui x :
 - ▶ După λ

Apariții libere și legate ale variabilelor



- ▶ Apariție **legată** a lui x :
 - ▶ După λ
 - ▶ În corpul unei funcții de **parametru** x

Apariții libere și legate ale variabilelor



- ▶ Apariție **legată** a lui x :
 - ▶ După λ
 - ▶ În corpul unei funcții de **parametru** x
- ▶ Dependența statutului unei apariții de **expresia** la care ne raportăm!



Formalizarea substituției (cont.)

- ▶ Substituirea tuturor aparițiilor parametrului formal, care sunt **libere** în raport cu **corpul**!



Formalizarea substituției (cont.)

- ▶ Substituirea tuturor aparițiilor parametrului formal, care sunt **libere** în raport cu **corpul**!
- ▶ În exemplul anterior, $(\lambda x.\lambda x.y\ y)$:

Formalizarea substituției (cont.)

- ▶ Substituirea tuturor aparițiilor parametrului formal, care sunt **libere** în raport cu **corpul**!
- ▶ În exemplul anterior, $(\lambda x.\lambda x.y\ y)$:
 - ▶ **Absența** aparițiilor libere ale lui x în corpul $\lambda x.y$

Formalizarea substituției (cont.)

- ▶ Substituirea tuturor aparițiilor parametrului formal, care sunt **libere** în raport cu **corpul**!
- ▶ În exemplul anterior, $(\lambda x.\lambda x.y\ y)$:
 - ▶ **Absența** aparițiilor libere ale lui x în corpul $\lambda x.y$
 - ▶ Producerea **corectă** a corpului nemodificat ca rezultat



Formalizarea substituției (cont.)

- ▶ Substituirea tuturor aparițiilor parametrului formal, care sunt **libere** în raport cu **corpul**!
- ▶ În exemplul anterior, $(\lambda x.\lambda x.y\ y)$:
 - ▶ **Absența** aparițiilor libere ale lui x în corpul $\lambda x.y$
 - ▶ Producerea **corectă** a corpului nemodificat ca rezultat
- ▶ În expresia $(\lambda x.\lambda cons.x\ cons)$:



Formalizarea substituției (cont.)

- ▶ Substituirea tuturor aparițiilor parametrului formal, care sunt **libere** în raport cu **corpul**!
- ▶ În exemplul anterior, $(\lambda x.\lambda x.y\ y)$:
 - ▶ **Absența** aparițiilor libere ale lui x în corpul $\lambda x.y$
 - ▶ Producerea **corectă** a corpului nemodificat ca rezultat
- ▶ În expresia $(\lambda x.\lambda cons.x\ cons)$:
 - ▶ Apariția din dreapta a lui $cons$ este **liberă**, cu semnificația din Racket



Formalizarea substituției (cont.)

- ▶ Substituirea tuturor aparițiilor parametrului formal, care sunt **libere** în raport cu **corpul**!
- ▶ În exemplul anterior, $(\lambda x.\lambda x.y\ y)$:
 - ▶ **Absența** aparițiilor libere ale lui x în corpul $\lambda x.y$
 - ▶ Producerea **corectă** a corpului nemodificat ca rezultat
- ▶ În expresia $(\lambda x.\lambda cons.x\ cons)$:
 - ▶ Apariția din dreapta a lui $cons$ este **liberă**, cu semnificația din Racket
 - ▶ Aplicarea mecanică: $\lambda cons.cons$



Formalizarea substituției (cont.)

- ▶ Substituirea tuturor aparițiilor parametrului formal, care sunt **libere** în raport cu **corpul**!
- ▶ În exemplul anterior, $(\lambda x.\lambda x.y\ y)$:
 - ▶ **Absența** aparițiilor libere ale lui x în corpul $\lambda x.y$
 - ▶ Producerea **corectă** a corpului nemodificat ca rezultat
- ▶ În expresia $(\lambda x.\lambda cons.x\ cons)$:
 - ▶ Apariția din dreapta a lui $cons$ este **liberă**, cu semnificația din Racket
 - ▶ Aplicarea mecanică: $\lambda cons.cons$
 - ▶ Rezultat eronat, din cauza modificării statutului, din apariție liberă în **legată**



Redenumirea variabilelor legate

$(\lambda x. \lambda cons. x \ cons)$

Redenumirea variabilelor legate

$(\lambda x. \lambda \text{cons}. x \text{ cons})$

Aparițiile **legate** din corp,

Redenumirea variabilelor legate

$$(\lambda x. \lambda \textcolor{red}{cons}. x \textcolor{green}{cons})$$

Aparițiile **legate** din corp,
în conflict cu cele **libere** din parametrul actual,

Redenumirea variabilelor legate

$$(\lambda x. \lambda z \quad .x \text{ *cons*})$$

Aparițiile **legate** din corp,
în conflict cu cele **libere** din parametrul actual,
redenumite!



Formalizarea substituției — concluzie

- ▶ Substituirea tuturor aparițiilor parametrului formal, care sunt **libere** în raport cu **corpul**, ulterioară eventualelor **redenumiri** ale aparițiilor **legate** din corpul funcției, care coincid cu aparițiile **libere** din parametrul actual

Formalizarea substituției — concluzie

- ▶ Substituirea tuturor aparițiilor parametrului formal, care sunt **libere** în raport cu **corpul**, ulterioară eventualelor **redenumiri** ale aparițiilor **legate** din corpul funcției, care coincid cu aparițiile **libere** din parametrul actual
- ▶ În exemplul anterior, $(\lambda x. \lambda z. x \text{ cons}) \rightarrow \lambda z. \text{cons}$



Formalizarea substituției — concluzie

- ▶ Substituirea tuturor aparițiilor parametrului formal, care sunt **libere** în raport cu **corpul**, ulterioară eventualelor **redenumiri** ale aparițiilor **legate** din corpul funcției, care coincid cu aparițiile **libere** din parametrul actual
- ▶ În exemplul anterior, $(\lambda x. \lambda z. x \text{ cons}) \rightarrow \lambda z. \text{cons}$
- ▶ Rezultat corect, cu păstrarea statutului de apariție **liberă**



Universalitatea funcțiilor

- ▶ Posibilitatea reprezentării tuturor valorilor uzuale **exclusiv** prin funcții (v. slide-ul 49)
- ▶ Mai devreme, funcții ca date (parametri, valori de retur etc.)
- ▶ Acum, date ca funcții!!
- ▶ V. sursele atașate slide-urilor



Rezumat

- ▶ **Abstractizare** funcțională
- ▶ Funcții ca **valori** — sporirea **expresivității** limbajului
- ▶ Funcționale — metode **generale** de prelucrare
- ▶ Calculul lambda și **universalitatea** funcțiilor



Partea V

Legarea variabilelor.
Evaluare contextuală



Cuprins

Legarea variabilelor

Contexte, închideri, evaluare contextuală



Cuprins

Legarea variabilelor

Contexte, închideri, evaluare contextuală



Variabile

Proprietăți

- ▶ Tip: asociate valorilor, **nu** variabilelor



Variabile

Proprietăți

- ▶ Tip: asociate valorilor, **nu** variabilelor
- ▶ Identificator



Variabile

Proprietăți

- ▶ Tip: asociate valorilor, **nu** variabilelor
- ▶ Identificator
- ▶ Valoarea legată (la un anumit moment)



Variabile

Proprietăți

- ▶ Tip: asociate valorilor, **nu** variabilelor
- ▶ Identificator
- ▶ Valoarea legată (la un anumit moment)
- ▶ Domeniul de vizibilitate



Variable

Proprietăți

- ▶ Tip: asociate valorilor, **nu** variabilelor
- ▶ Identificator
- ▶ Valoarea legată (la un anumit moment)
- ▶ Domeniul de vizibilitate
- ▶ Durata de viață



Variabile

Stări

- ▶ Declarată: cunoaştem **identificatorul**



Variabile

Stări

- ▶ Declarată: cunoaştem **identificatorul**
- ▶ Definită: cunoaştem şi **valoarea**



Legarea variabilelor

- ▶ Modul de **asociere** a apariției unei variabile cu definiția acesteia



Legarea variabilelor

- ▶ Modul de **asociere** a apariției unei variabile cu definiția acesteia
- ▶ Domeniu de vizibilitate (*scope*) = mulțimea **punctelor** din program unde o definiție este vizibilă, pe baza modului de **legare**



Legarea variabilelor

- ▶ Modul de **asociere** a apariției unei variabile cu definiția acesteia
- ▶ Domeniu de vizibilitate (*scope*) = mulțimea **punctelor** din program unde o definiție este vizibilă, pe baza modului de **legare**
- ▶ Statică (lexicală) / dinamică



Problemă

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
```



Problemă

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
```

- **Atenție!** Variabilele `x` sunt **diferite**, nu se reatribuie același `x` (aceasta este semnificația lui `def`)



Problemă

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
```

- ▶ Atenție! Variabilele `x` sunt **diferite**, nu se reatribuie același `x` (aceasta este semnificația lui `def`)
- ▶ În câte **moduri** poate decurge evaluarea aplicației `g()`, în raport cu variabilele definite?



Legare statică (lexicală)

- ▶ Extragerea variabilelor din contextul **definirii** expresiei

```
1 def x = 0
2   f() { return x }
3 def x = 1
4   g() { def x = 2 ; return f() }
```



Legare statică (lexicală)

- ▶ Extragerea variabilelor din contextul **definirii** expresiei
- ▶ Domeniu de vizibilitate determinat prin **construcțiile** limbajului (lexical), la **compilare** (static)

```
1 def x = 0
2   f() { return x }
3 def x = 1
4   g() { def x = 2 ; return f() }
```



Legare statică (lexicală)

- ▶ Extragerea variabilelor din contextul **definirii** expresiei
- ▶ Domeniu de vizibilitate determinat prin **construcțiile** limbajului (lexical), la **compilare** (static)

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
```



Legare statică (lexicală)

- ▶ Extragerea variabilelor din contextul **definirii** expresiei
- ▶ Domeniu de vizibilitate determinat prin **construcțiile** limbajului (lexical), la **compilare** (static)

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
```



Legare statică (lexicală)

- ▶ Extragerea variabilelor din contextul **definirii** expresiei
- ▶ Domeniu de vizibilitate determinat prin **construcțiile** limbajului (lexical), la **compilare** (static)

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
```



Legare statică (lexicală)

- ▶ Extragerea variabilelor din contextul **definirii** expresiei
- ▶ Domeniu de vizibilitate determinat prin **construcțiile** limbajului (lexical), la **compilare** (static)

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
```



Legare statică (lexicală)

- ▶ Extragerea variabilelor din contextul **definirii** expresiei
- ▶ Domeniu de vizibilitate determinat prin **construcțiile** limbajului (lexical), la **compilare** (static)

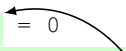
```
1 def x = 0
2   f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
```



Legare statică (lexicală)

- ▶ Extragerea variabilelor din contextul **definirii** expresiei
- ▶ Domeniu de vizibilitate determinat prin **construcțiile** limbajului (lexical), la **compilare** (static)

```
1  def x = 0
2  f() { return x }
3  def x = 1
4  g() { def x = 2 ; return f() }
```



Legare statică (lexicală)

- ▶ Extragerea variabilelor din contextul **definirii** expresiei
- ▶ Domeniu de vizibilitate determinat prin **construcțiile** limbajului (lexical), la **compilare** (static)

```
1  def x = 0
2  f() { return x }
3  def x = 1
4  g() { def x = 2 ; return f() }
```

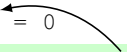
`g()` →



Legare statică (lexicală)

- ▶ Extragerea variabilelor din contextul **definirii** expresiei
- ▶ Domeniu de vizibilitate determinat prin **construcțiile** limbajului (lexical), la **compilare** (static)

```
1  def x = 0
2  f() { return x }
3  def x = 1
4  g() { def x = 2 ; return f() }
```



`g()` $\rightarrow$ 0



Legare statică în calculul lambda

Care sunt domeniile de vizibilitate ale parametrilor formali, în expresia de mai jos?

$$\lambda \underline{x} . \lambda \underline{y} . (\lambda \underline{x} . x \quad y)$$

Legare statică în calculul lambda

Care sunt domeniile de vizibilitate ale parametrilor formali, în expresia de mai jos?

$$\lambda \underline{x} . \lambda \underline{y} . (\lambda \underline{x} . x \quad y)$$

Legare statică în calculul lambda

Care sunt domeniile de vizibilitate ale parametrilor formali, în expresia de mai jos?

$$\lambda \underline{x} . \lambda \underline{y} . (\lambda \boxed{\underline{x}} . \boxed{x} \ y)$$

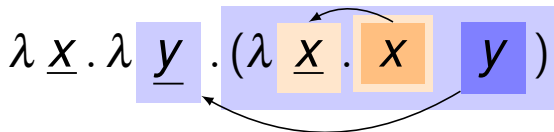
Legare statică în calculul lambda

Care sunt domeniile de vizibilitate ale parametrilor formali, în expresia de mai jos?

$$\lambda \underline{x} . \lambda \underline{y} . (\lambda \underline{x} . \underline{x} \ y)$$

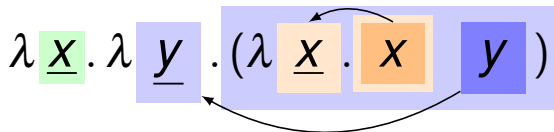
Legare statică în calculul lambda

Care sunt domeniile de vizibilitate ale parametrilor formali, în expresia de mai jos?



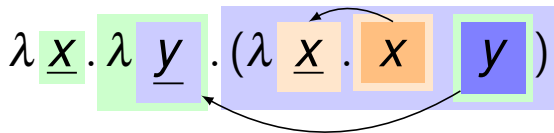
Legare statică în calculul lambda

Care sunt domeniile de vizibilitate ale parametrilor formali, în expresia de mai jos?



Legare statică în calculul lambda

Care sunt domeniile de vizibilitate ale parametrilor formali, în expresia de mai jos?



Legare dinamică

- ▶ Extragerea variabilelor din contextul **evaluării** expr.

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
5 ...
```



Legare dinamică

- ▶ Extragerea variabilelor din contextul **evaluării** expr.
- ▶ Domeniu de vizibilitate determinat la **execuție**

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
5 ...
```



Legare dinamică

- ▶ Extragerea variabilelor din contextul **evaluării** expr.
- ▶ Domeniu de vizibilitate determinat la **execuție**

```
1  def x = 0
2  f() { return x }
3  def x = 1
4  g() { def x = 2 ; return f() }
5  ...
```

◁ f() -> 0

Legare dinamică

- ▶ Extragerea variabilelor din contextul **evaluării** expr.
- ▶ Domeniu de vizibilitate determinat la **execuție**

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
5 ...
```

f() -> 0
◁ f() -> 1



Legare dinamică

- ▶ Extragerea variabilelor din contextul **evaluării** expr.
- ▶ Domeniu de vizibilitate determinat la **execuție**

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
5 ...
```

Diagram illustrating dynamic scoping resolution:

- `f()` → 0
- `f()` → 1
- `f()` → 2 ← `g()`

An arrow points from the `x` in line 2 to the `x` in line 4, indicating that the function `f` resolves the variable `x` to the binding in the scope where `g` was called.



Legare dinamică

- ▶ Extragerea variabilelor din contextul **evaluării** expr.
- ▶ Domeniu de vizibilitate determinat la **execuție**

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
5 ...
```

Diagram illustrating dynamic scoping resolution:

- `f()` -> 0
- `f()` -> 1
- `f()` -> 2 <- `g()`
- < `f()` -> 1

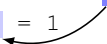
Note: An arrow points from the `x` in line 2 to the `x` in line 4, indicating that `f()` resolves to the binding in `g()` due to dynamic scoping.



Legare dinamică

- ▶ Extragerea variabilelor din contextul **evaluării** expr.
- ▶ Domeniu de vizibilitate determinat la **execuție**

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
5 ...
```



```
f() -> 0
f() -> 1
f() -> 2 <- g()
f() -> 1
```

Atenție! x-ul **portocaliu**, vizibil:



Legare dinamică

- ▶ Extragerea variabilelor din contextul **evaluării** expr.
- ▶ Domeniu de vizibilitate determinat la **execuție**

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
5 ...
```

```
f() -> 0
f() -> 1
f() -> 2 <- g()
f() -> 1
```

Atenție! **x**-ul **portocaliu**, vizibil:

- ▶ spațial: în **întregul** program



Legare dinamică

- ▶ Extragerea variabilelor din contextul **evaluării** expr.
- ▶ Domeniu de vizibilitate determinat la **execuție**

1	def	x	=	0	
2	f()	{	return	x	}
3	def	x	=	1	
4	g()	{	def	x	= 2 ; return f() }
5	...				

f() -> 0
f() -> 1
f() -> 2 <- g()
f() -> 1

Atenție! x-ul **portocaliu**, vizibil:

- ▶ spațial: în **întregul** program
- ▶ temporal: doar pe durata evaluării **corpului** lui g()



Legare mixtă

- ▶ Variabile locale, **static**
- ▶ Variabile globale, **dinamic**

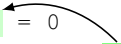
```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
5 ...
```



Legare mixtă

- ▶ Variabile locale, **static**
- ▶ Variabile globale, **dinamic**

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
5 ...
```



◁ f() -> 0



Legare mixtă

- ▶ Variabile locale, **static**
- ▶ Variabile globale, **dinamic**

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
5 ...
```

f() -> 0
◁ f() -> 1



Legare mixtă

- ▶ Variabile locale, **static**
- ▶ Variabile globale, **dinamic**

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
5 ...
```

Diagram illustrating variable binding:

- A curved arrow points from the `x` in line 2 to the `x` in line 3, indicating that the function `f()` captures the value of `x` from the scope where it was defined (line 3).
- A small triangle points to the `x` in line 4, indicating that this is a new local binding for `x` within the scope of `g()`.

Execution flow and results:

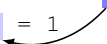
- `f()` -> 0 (Initial value of `x`)
- `f()` -> 1 (Value of `x` after line 3)
- `f()` -> 1 <- `g()` (Value of `x` after line 4, as `g()` returns the result of `f()` where `x` is 1)



Legare mixtă

- ▶ Variabile locale, **static**
- ▶ Variabile globale, **dinamic**

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
5 ...
```



```
f() -> 0
f() -> 1
f() -> 1 <- g()
◁ f() -> 1
```



Legare mixtă

- ▶ Variabile locale, **static**
- ▶ Variabile globale, **dinamic**

```
1 def x = 0
2 f() { return x }
3 def x = 1
4 g() { def x = 2 ; return f() }
5 ...
```

```
f() -> 0
f() -> 1
f() -> 1 <- g()
f() -> 1
```

Atenție! x-ul portocaliu, **invizibil** în corpul lui `f`!



Legarea variabilelor în Racket

- ▶ Variabile declarate sau definite în expresii: **static**:
 - ▶ `lambda`
 - ▶ `let`
 - ▶ `let*`
 - ▶ `letrec`



Legarea variabilelor în Racket

- ▶ Variabile declarate sau definite în expresii: **static**:
 - ▶ `lambda`
 - ▶ `let`
 - ▶ `let*`
 - ▶ `letrec`
- ▶ Variabile *top-level*: **dinamic**:
 - ▶ `define`



Construcția `lambda`

Definiție

- ▶ Leagă **static** parametrii formali ai unei funcții



Construcția `lambda`

Definiție

- ▶ Leagă **static** parametrii formali ai unei funcții
- ▶ Sintaxă:

```
1  (lambda (p1 ... pk ... pn)
2    expr )
```



Construcția `lambda`

Definiție

- ▶ Leagă **static** parametrii formali ai unei funcții

- ▶ Sintaxă:

```
1  (lambda (p1 ... pk ... pn)  
2    expr )
```

- ▶ Domeniul de vizibilitate a parametrului p_k = mulțimea punctelor din **corpul** funcției, `expr`, în care aparițiile lui p_k sunt **libere** (v. slide-ul 127)



Construcția `lambda`

Exemplu

```
1  (lambda (x)
2    (x (lambda (y) y)) )
```



Construcția `lambda`

Exemplu

```
1  (lambda (x)
2    (x (lambda (y) y)))
```



Construcția `lambda`

Semantică

► Aplicație:

```
1  ((lambda (p1 ... pn)
2      expr) a1 ... an)
```



Construcția `lambda`

Semantică

- Aplicație:

```
1  ((lambda (p1 ... pn)
2      expr) a1 ... an)
```

- Se evaluează **operanzii** a_k , în ordine aleatoare (evaluare aplicativă)



Construcția `lambda`

Semantică

- ▶ Aplicație:

```
1  ((lambda (p1 ... pn)
2      expr) a1 ... an)
```

- ▶ Se evaluează **operanzii** a_k , în ordine aleatoare (evaluare aplicativă)
- ▶ Se evaluează **corpul** funcției, `expr`, ținând cont de legările $p_k \leftarrow \text{valoare}(a_k)$



Construcția λ lambda

Semantică

- ▶ Aplicație:

```
1  ((lambda (p1 ... pn)
2      expr) a1 ... an)
```

- ▶ Se evaluează **operanzii** a_k , în ordine aleatoare (evaluare aplicativă)
- ▶ Se evaluează **corpul** funcției, $expr$, ținând cont de legările $p_k \leftarrow valoare(a_k)$
- ▶ **Valoarea** aplicației este valoarea lui $expr$



Construcția `let`

Definiție

- ▶ Leagă **static** variabile locale



Construcția `let`

Definiție

- ▶ Leagă **static** variabile locale

- ▶ Sintaxă:

```
1  (let ([v1 e1] ... [ vk ek] ... [vn en])  
2      expr )
```



Construcția `let`

Definiție

- ▶ Leagă **static** variabile locale

- ▶ Sintaxă:

```
1  (let ([v1 e1] ... [vk ek] ... [vn en])  
2    expr)
```

- ▶ Domeniul de vizibilitate a variabilei v_k = mulțimea punctelor din **corp**, `expr`, în care aparițiile lui v_k sunt **libere** (v. slide-ul 127)



Construcția `let`

Exemplu

```
1  (let ([x 1] [y 2]))  
2      (+ x 2) )
```



Construcția `let`

Exemplu

```
1  (let ([x 1] [y 2]))  
2    (+ x 2) )
```



Construcția `let`

Semantică

```
1  (let ([v1 e1] ... [vn en])  
2    expr)
```

echivalent cu



Construcția `let`

Semantică

```
1  (let ([v1 e1] ... [vn en])  
2    expr)
```

echivalent cu

```
1  ((lambda (v1 ... vn)  
2    expr) e1 ... en)
```



Construcția `let*`

Definiție

- ▶ Leagă **static** variabile locale



Construcția `let*`

Definiție

- ▶ Leagă **static** variabile locale

- ▶ Sintaxă:

```
1  (let* ([v1 e1] ... [vk ek] ... [vn en] )  
2      expr )
```



Construcția `let*`

Definiție

- ▶ Leagă **static** variabile locale

- ▶ Sintaxă:

```
1  (let* ([v1 e1] ... [vk ek] ... [vn en] )  
2    expr )
```

- ▶ Domeniul de vizibilitate a variabilei v_k = mulțimea punctelor din

în care aparițiile lui v_k sunt **libere** (v. slide-ul 127)



Construcția `let*`

Definiție

- ▶ Leagă **static** variabile locale

- ▶ Sintaxă:

```
1  (let* ([v1 e1] ... [vk ek] ... [vn en] )  
2    expr )
```

- ▶ Domeniul de vizibilitate a variabilei v_k = mulțimea punctelor din
 - ▶ restul **legărilor** și

în care aparițiile lui v_k sunt **libere** (v. slide-ul 127)



Construcția `let*`

Definiție

- ▶ Leagă **static** variabile locale

- ▶ Sintaxă:

```
1  (let* ([v1 e1] ... [vk ek] ... [vn en] )  
2    expr )
```

- ▶ Domeniul de vizibilitate a variabilei v_k = mulțimea punctelor din
 - ▶ restul **legărilor** și
 - ▶ **corp**, `expr`,

în care aparițiile lui v_k sunt **libere** (v. slide-ul 127)



Construcția `let*`

Exemplu

```
1  (let* ([x 1] [y x])  
2      (+ x 2) )
```



Construcția `let*`

Exemplu

```
1  (let* ([x 1] [y x])  
2    (+ x 2) )
```



Construcția `let*`

Semantică

```
1  (let* ([v1 e1] ... [vn en])  
2    expr)
```

echivalent cu

Construcția `let*`

Semantică

```
1  (let* ([v1 e1] ... [vn en])  
2      expr)
```

echivalent cu

```
1  (let ([v1 e1])  
2      ...  
3      (let ([vn en])  
4          expr) ...)
```

Evaluarea expresiilor se face **în ordine!**



Construcția `letrec`

Definiție

- ▶ Leagă **static** variabile locale



Construcția `letrec`

Definiție

- ▶ Leagă **static** variabile locale

- ▶ Sintaxă:

```
1  (letrec ( [v1 e1] ... [ vk ek] ... [vn en] )  
2      expr )
```



Construcția `letrec`

Definiție

- ▶ Leagă **static** variabile locale

- ▶ Sintaxă:

```
1  (letrec ( [v1 e1] ... [vk ek] ... [vn en] )  
2      expr )
```

- ▶ Domeniul de vizibilitate a variabilei v_k = mulțimea punctelor din **întreaga** construcție, în care aparițiile lui v_k sunt **libere** (v. slide-ul 127)



Construcția `letrec`

Exemplu

```
1  (letrec ([factorial
2            (lambda (n)
3              (if (zero? n) 1
4                  (* n (factorial (- n 1))))))]
5    (factorial 5))
```



Construcția `letrec`

Exemplu

```
1 (letrec ([factorial
2           (lambda (n)
3             (if (zero? n) 1
4                 (* n (factorial (- n 1))))))]
5  (factorial 5))
```



Construcția `define`

Definiție

- ▶ Leagă **dinamic** variabile *top-level* (de obicei)



Construcția `define`

Definiție

- ▶ Leagă **dinamic** variabile *top-level* (de obicei)
- ▶ Sintaxă:

```
1  (define v expr)
```



Construcția `define`

Definiție

- ▶ Leagă **dinamic** variabile *top-level* (de obicei)

- ▶ Sintaxă:

```
1  (define v expr)
```

- ▶ Domeniul de vizibilitate a variabilei v =
întregul program, presupunând că:



Construcția `define`

Definiție

- ▶ Leagă **dinamic** variabile *top-level* (de obicei)

- ▶ Sintaxă:

```
1  (define v expr)
```

- ▶ Domeniul de vizibilitate a variabilei v =
întregul program, presupunând că:
 - ▶ legarea a fost făcută, în timpul **execuției**



Construcția `define`

Definiție

- ▶ Leagă **dinamic** variabile *top-level* (de obicei)

- ▶ Sintaxă:

```
1  (define v expr)
```

- ▶ Domeniul de vizibilitate a variabilei v =
întregul program, presupunând că:
 - ▶ legarea a fost făcută, în timpul **execuției**
 - ▶ **nicio o altă** legare, statică sau dinamică, a lui v ,
nu a fost făcută ulterior



Construcția define

Exemple

```
1 (define x 0)
2 (define f (lambda () x))
3 (f) ; 0
4 (define x 1)
5 (f) ; 1
```



Construcția define

Exemple

```
1
2 (define f (lambda () x))
3
4 (define x 1)
5 (f) ; 1
```



Construcția define

Exemple

```
1 (define factorial
2   (lambda (n)
3     (if (zero? n) 1
4         (* n (factorial (- n 1))))))
5
6 (factorial 5)
7
8 (define g factorial)
9 (define factorial (lambda (x) x))
10
11 (g 5)
```



Construcția define

Exemple

```
1 (define factorial
2   (lambda (n)
3     (if (zero? n) 1
4         (* n (factorial (- n 1))))))
5
6 (factorial 5) ; 120
7
8 (define g factorial)
9 (define factorial (lambda (x) x))
10
11 (g 5)
```



Construcția define

Exemple

```
1 (define factorial
2   (lambda (n)
3     (if (zero? n) 1
4         (* n (factorial (- n 1))))))
5
6 (factorial 5) ; 120
7
8 (define g factorial)
9 (define factorial (lambda (x) x))
10
11 (g 5) ; 20
```



Construcția define

Semantică

- ▶ Se evaluează **expresia**, `expr`



Construcția `define`

Semantică

- ▶ Se evaluează **expresia**, `expr`
- ▶ **Valoarea** lui `v` este valoarea lui `expr`



Construcția define

Semantică

- ▶ Se evaluează **expresia**, $expr$
- ▶ **Valoarea** lui v este valoarea lui $expr$
- ▶ Avantaje:



Construcția `define`

Semantică

- ▶ Se evaluează **expresia**, `expr`
- ▶ **Valoarea** lui `v` este valoarea lui `expr`
- ▶ Avantaje:
 - ▶ definirea variabilelor *top-level* în **orice** ordine



Construcția `define`

Semantică

- ▶ Se evaluează **expresia**, `expr`
- ▶ **Valoarea** lui `v` este valoarea lui `expr`
- ▶ Avantaje:
 - ▶ definirea variabilelor *top-level* în **orice** ordine
 - ▶ definirea funcțiilor **mutual** recursive



Construcția `define`

Semantică

- ▶ Se evaluează **expresia**, `expr`
- ▶ **Valoarea** lui `v` este valoarea lui `expr`
- ▶ Avantaje:
 - ▶ definirea variabilelor *top-level* în **orice** ordine
 - ▶ definirea funcțiilor **mutual** recursive
- ▶ Dezavantaj: efect de **atribuire**



Exemplu mixt

Codificarea secvenței de pe slide-ul 130

```
1 (define x 0)
2 (define f (lambda () x))
3 (define x 1)
4
5 (define g
6   (lambda (x)
7     (f)))
8
9 (g 2)
```



Exemplu mixt

Codificarea secvenței de pe slide-ul 130

```
1 (define x 0)
2 (define f (lambda () x))
3 (define x 1)
4
5 (define g
6   (lambda (x)
7     (f)))
8
9 (g 2) ; 1
```



Aplicație pentru legarea variabilelor

```
79 (define (app A B)
80   (if (null? A)
81       B
82       (cons (car A) (app (cdr A) B))))
```



Aplicație pentru legarea variabilelor

```
79 (define (app A B)
80   (if (null? A)
81       B
82       (cons (car A) (app (cdr A) B))))
```

Problemă: B este trimis **nemodificat** fiecărei aplicații recursive. Rescriem:

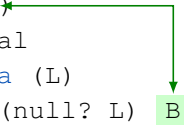


Aplicație pentru legarea variabilelor

```
79 (define (app A B)
80   (if (null? A)
81       B
82       (cons (car A) (app (cdr A) B))))
```

Problemă: B este trimis **nemodificat** fiecărei aplicații recursive. Rescriem:

```
87 (define (app2 A B)
88   (letrec ([internal
89             (lambda (L)
90               (if (null? L) B
91                   (cons (car L)
92                         (internal (cdr L))))))]
93     (internal A)))
```



Cuprins

Legarea variabilelor

Contexte, închideri, evaluare contextuală



Modelul de evaluare bazat pe substituție

- ▶ Ineficient



Modelul de evaluare bazat pe substituție

- ▶ Ineficient
- ▶ Tratatament special pentru coliziunile dintre variabilele libere ale parametrului actual și cele legate ale corpului funcției aplicate

Modelul de evaluare bazat pe substituție

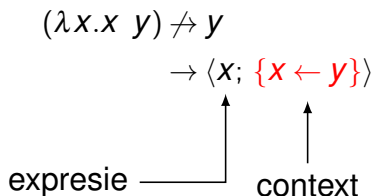
- ▶ Ineficient
- ▶ Tratament special pentru coliziunile dintre variabilele libere ale parametrului actual și cele legate ale corpului funcției aplicate
- ▶ Imposibil de aplicat, în prezența unor eventuale reatribuiri ale variabilelor

Alternativă la substituția textuală

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

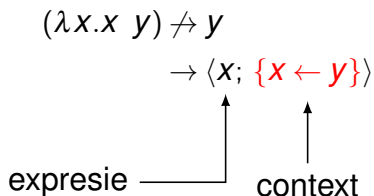


Alternativă la substituția textuală



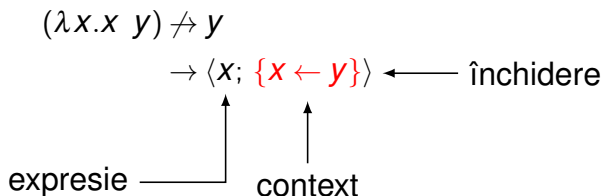
- Asocierea unei expresii cu un dicționar de variabile libere: **context** de evaluare

Alternativă la substituția textuală



- ▶ Asocierea unei expresii cu un dicționar de variabile libere: **context** de evaluare
- ▶ **Căutarea** unei variabile utilizate în procesul de evaluare, în contextul asociat

Alternativă la substituția textuală



- ▶ Asocierea unei expresii cu un dicționar de variabile libere: **context** de evaluare
- ▶ **Căutarea** unei variabile utilizate în procesul de evaluare, în contextul asociat
- ▶ Perechea: **închidere**, i.e. formă pseudoînchisă a expresiei, obținută prin legarea variabilelor libere



Context computațional

- ▶ Mulțime de **variabile**, alături de **valorile** acestora



Context computațional

- ▶ Mulțime de **variabile**, alături de **valorile** acestora
- ▶ Dependent de **punctul** din program și de momentul de **timp**



Context computațional

- ▶ Mulțime de **variabile**, alături de **valorile** acestora
- ▶ Dependent de **punctul** din program și de momentul de **timp**
- ▶ Legare **statică** — mulțimea variabilelor care conțin punctul conform structurii **lexicale** a programului

```
1  (let ([x 1])  
2      (+ x (let ([y 2])  
3              (* x y) )))
```



Context computațional

- ▶ Mulțime de **variabile**, alături de **valorile** acestora
- ▶ Dependent de **punctul** din program și de momentul de **timp**
- ▶ Legare **statică** — mulțimea variabilelor care conțin punctul conform structurii **lexicale** a programului

```
1  (let ([x 1])  
2      (+ x (let ([y 2])  
3              (* x y) )))
```



Context computațional

- ▶ Mulțime de **variabile**, alături de **valorile** acestora
- ▶ Dependent de **punctul** din program și de momentul de **timp**
- ▶ Legare **statică** — mulțimea variabilelor care conțin punctul conform structurii **lexicale** a programului

```
1  (let ([x 1])  
2    (+ x (let ([y 2])  
3          (* x y) )))
```



Context computațional

- ▶ Mulțime de **variabile**, alături de **valorile** acestora
- ▶ Dependent de **punctul** din program și de momentul de **timp**
- ▶ Legare **statică** — mulțimea variabilelor care conțin punctul conform structurii **lexicale** a programului

```
1  (let ([x] 1))  
2    (+ x (let ([y] 2))  
3          (* x y) )))
```



Context computațional

- ▶ Mulțime de **variabile**, alături de **valorile** acestora
- ▶ Dependent de **punctul** din program și de momentul de **timp**
- ▶ Legare **statică** — mulțimea variabilelor care conțin punctul conform structurii **lexicale** a programului

```
1  (let ([x] 1))  
2    (+ x (let ([y] 2))  
3          (* x y)))
```



Context computațional

- ▶ Mulțime de **variabile**, alături de **valorile** acestora
- ▶ Dependent de **punctul** din program și de momentul de **timp**
- ▶ Legare **statică** — mulțimea variabilelor care conțin punctul conform structurii **lexicale** a programului

```
1  (let ([x] 1))
2  (+ x (let ([y] 2))
3      (* x y)))
```

$\{x \leftarrow 1\}$

Context computațional

- ▶ Mulțime de **variabile**, alături de **valorile** acestora
- ▶ Dependent de **punctul** din program și de momentul de **timp**
- ▶ Legare **statică** — mulțimea variabilelor care conțin punctul conform structurii **lexicale** a programului

```
1  (let ([x] 1))
2  (+ x (let ([y] 2))
3      (* x y)))
```

$\{x \leftarrow 1\}$

$\{x \leftarrow 1, y \leftarrow 2\}$

Context computațional

- ▶ Mulțime de **variabile**, alături de **valorile** acestora
- ▶ Dependent de **punctul** din program și de momentul de **timp**
- ▶ Legare **statică** — mulțimea variabilelor care conțin punctul conform structurii **lexicale** a programului

```
1  (let ([x] 1))
2  (+ x (let ([y] 2))
3      (* x y)))
```

$\{x \leftarrow 1\}$

$\{x \leftarrow 1, y \leftarrow 2\}$

- ▶ Legare **dinamică** — mulțimea variabilelor definite cel mai **recent**



Închideri

Definiție

- ▶ Închidere: **pereche** expresie-context

Închideri

Definiție

- ▶ Închidere: **pereche** expresie-context
- ▶ **Semnificația** unei închideri:

$$\langle e; C \rangle$$

este valoarea expresiei e , în contextul C

Închideri

Definiție

- ▶ Închidere: **pereche** expresie-context
- ▶ **Semnificația** unei închideri:

$$\langle e; C \rangle$$

este valoarea expresiei e , în contextul C

- ▶ Închidere **funcțională**:

$$\langle \lambda x. e; C \rangle$$

este o funcție care își salvează contextul, pe care îl utilizează, în momentul aplicării, pentru evaluarea corpului



Închideri

Definiție

- ▶ Închidere: **pereche** expresie-context
- ▶ **Semnificația** unei închideri:

$$\langle e; C \rangle$$

este valoarea expresiei e , în contextul C

- ▶ Închidere **funcțională**:

$$\langle \lambda x.e; C \rangle$$

este o funcție care își salvează contextul, pe care îl utilizează, în momentul aplicării, pentru evaluarea corpului

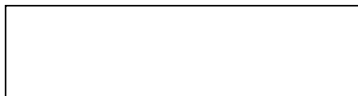
- ▶ Utilizate pentru legare **statică**!



Închideri

Construcție

```
1 (define y 0)
2 (define sum (lambda (x) (+ x y)))
```



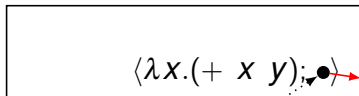
Contextul global

Închideri

Construcție

1. Construcție prin evaluarea unei expresii `lambda`, într-un context dat

```
1 (define y 0)
2 (define sum (lambda (x) (+ x y)))
```



Contextul global

Pointer către contextul global

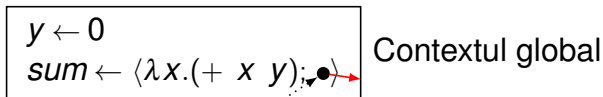


Închideri

Construcție

1. Construcție prin evaluarea unei expresii `lambda`, într-un context dat
2. **Legarea** variabilelor *top-level*, în contextul global, prin `define`

```
1 (define y 0)
2 (define sum (lambda (x) (+ x y)))
```



Pointer către contextul global

1 (sum (+ 1 2))

G $\boxed{\begin{array}{l} y \leftarrow 0 \\ \text{sum} \leftarrow \langle \lambda x. (+ x y); \bullet \rangle \end{array}}$ Contextul global

Închideri

Aplicare

1. Legarea parametrilor formali, într-un **nou** context, la valorile parametrilor actuali

1 (sum (+ 1 2))

G $\boxed{\begin{array}{l} y \leftarrow 0 \\ \text{sum} \leftarrow \langle \lambda x. (+ \ x \ y); \bullet \rangle \end{array}}$ Contextul global

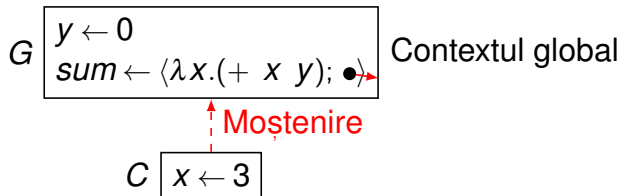
C $\boxed{x \leftarrow 3}$

Închideri

Aplicare

1. Legarea parametrilor formali, într-un **nou** context, la valorile parametrilor actuali
2. **Moștenirea** contextului din închidere de către cel nou

1 (sum (+ 1 2))

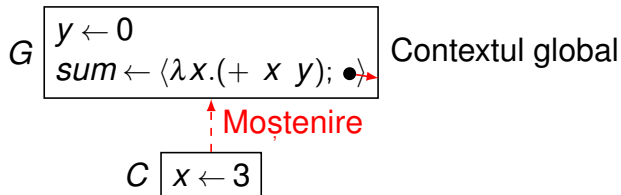


Închideri

Aplicare

1. Legarea parametrilor formali, într-un **nou** context, la valorile parametrilor actuali
2. **Moștenirea** contextului din închidere de către cel nou
3. Evaluarea **corpului** închiderii în noul context

1 (sum (+ 1 2))



Contextul în care se evaluează corpul (+ x y)



Ierarhia de contexte

- ▶ **Arbore** având contextul global drept rădăcină



Ierarhia de contexte

- ▶ **Arbore** având contextul global drept rădăcină
- ▶ În cazul **absenței** unei variabile din contextul curent, căutarea acesteia în contextul **părinte** ș.a.m.d.

Ierarhia de contexte

- ▶ **Arbore** având contextul global drept rădăcină
- ▶ În cazul **absenței** unei variabile din contextul curent, căutarea acesteia în contextul **părinte** ș.a.m.d.
- ▶ Pe slide-ul 155:

Ierarhia de contexte

- ▶ **Arbore** având contextul global drept rădăcină
- ▶ În cazul **absenței** unei variabile din contextul curent, căutarea acesteia în contextul **părinte** ș.a.m.d.
- ▶ Pe slide-ul 155:
 - ▶ x : identificat în C



Ierarhia de contexte

- ▶ **Arbore** având contextul global drept rădăcină
- ▶ În cazul **absenței** unei variabile din contextul curent, căutarea acesteia în contextul **părinte** ș.a.m.d.
- ▶ Pe slide-ul 155:
 - ▶ x : identificat în C
 - ▶ y : absent din C , dar identificat în G , părintele lui C

Închideri funcțională

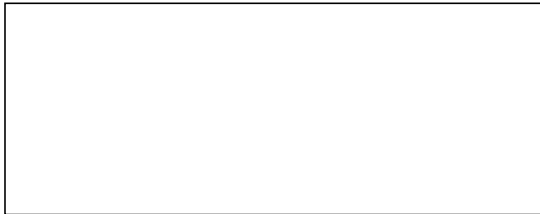
Exemplu

```
1  (define comp
2    (lambda (f)
3      (lambda (g)
4        (lambda (x)
5          (f (g x))))))
6
7  (define inc (lambda (x) (+ x 1)))
8  (define comp-inc (comp inc))
9
10 (define double (lambda (x) (* x 2)))
11 (define comp-inc-double (comp-inc double))
12
13 (comp-inc-double 5) ; 11
14
15 (define inc (lambda (x) x))
16 (comp-inc-double 5) ; tot 11!
```



Închideri funcționale

Explicația exemplului



Închideri funcționale

Explicația exemplului

$comp \leftarrow \langle \lambda fgx.(f (g x)); \bullet \rangle$



Închideri funcționale

Explicația exemplului

$comp \leftarrow \langle \lambda fgx.(f (g x)); \bullet \rangle$

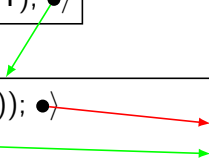
$inc \leftarrow \langle \lambda x.(+ x 1); \bullet \rangle$

Închideri funcțională

Explicația exemplului

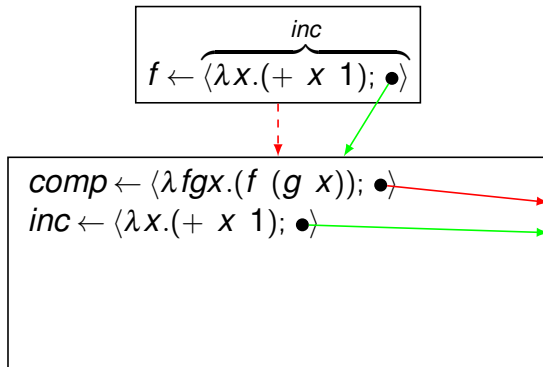
$$f \leftarrow \langle \overbrace{\lambda x. (+ \ x \ 1)}^{inc}; \bullet \rangle$$

$$comp \leftarrow \langle \lambda fgx. (f \ (g \ x)); \bullet \rangle$$

$$inc \leftarrow \langle \lambda x. (+ \ x \ 1); \bullet \rangle$$


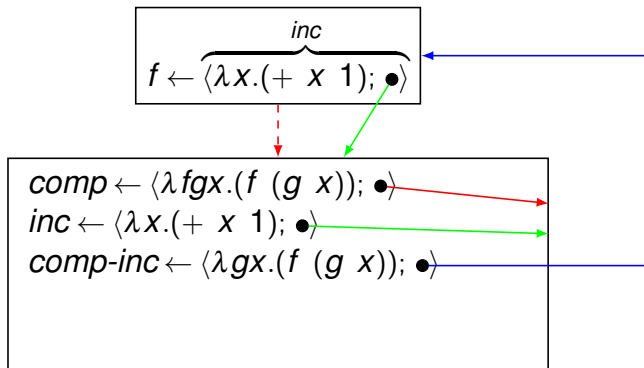
Închideri funcțională

Explicația exemplului



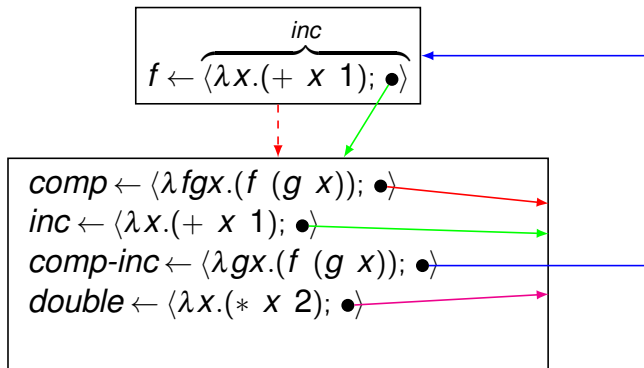
Închideri funcțională

Explicația exemplului



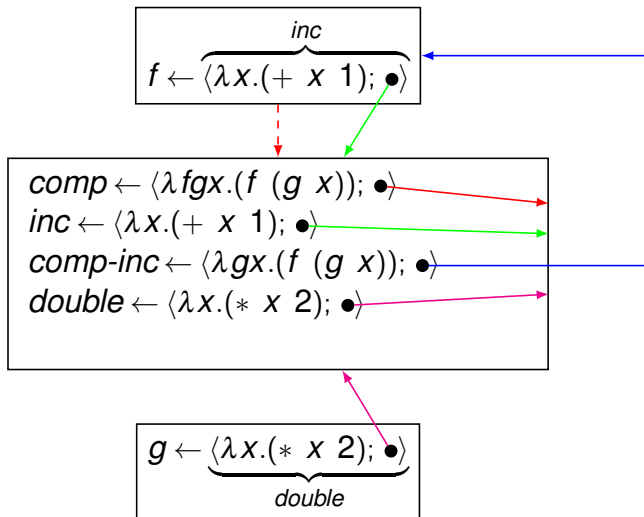
Închideri funcționale

Explicația exemplului



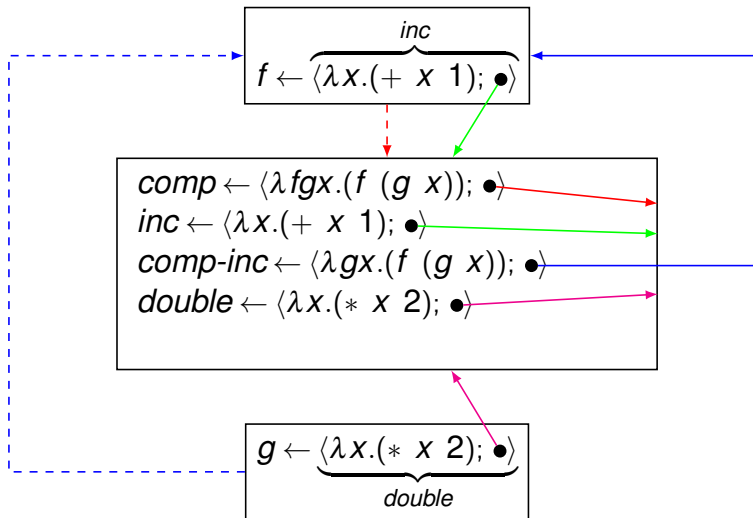
Închideri funcțională

Explicația exemplului



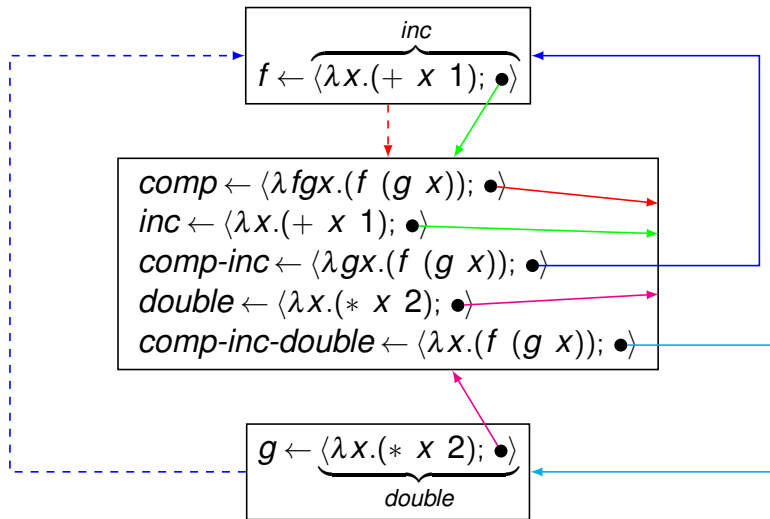
Închideri funcționale

Explicația exemplului



Închideri funcționale

Explicația exemplului



Rezumat

- ▶ Legare **statică/ dinamică** a variabilelor
- ▶ Contexte de evaluare, închideri, evaluare contextuală



Partea VI

Întârzierea evaluării



Cuprins

Mecanisme

Abstractizare de date

Fluxuri

Rezolvarea problemelor prin căutare leneșă
în spațiul stărilor



Cuprins

Mecanisme

Abstractizare de date

Fluxuri

Rezolvarea problemelor prin căutare leneșă
în spațiul stărilor



Motivație

- ▶ Să se implementeze funcția *prod*:
 - ▶ $prod(false, y) = 0$
 - ▶ $prod(true, y) = y(y + 1)$



Motivație

- ▶ Să se implementeze funcția *prod*:
 - ▶ $prod(false, y) = 0$
 - ▶ $prod(true, y) = y(y + 1)$
- ▶ Se presupune că evaluarea lui y este costisitoare, și că ar trebui efectuată doar dacă este necesar.



Varianta 1

Implementare directă

```
1 (define (prod x y)
2   (if x (* y (+ y 1)) 0))
3
4 (define (test x)
5   (let ([y 5])
6     (prod x (begin (display "y") y))))
7
8 (test #f) ; y 0
9 (test #t) ; y 30
```



Varianta 1

Implementare directă

```
1 (define (prod x y)
2   (if x (* y (+ y 1)) 0))
3
4 (define (test x)
5   (let ([y 5])
6     (prod x (begin (display "y") y))))
7
8 (test #f) ; y 0
9 (test #t) ; y 30
```

Implementare **eronată**, deoarece **ambii** parametri sunt evaluați în momentul aplicării!



Varianta 2

quote & eval

```
1 (define (prod x y)
2   (if x (* (eval y) (+ (eval y) 1)) 0))
3
4 (define (test x)
5   (let ([y 5])
6     (prod x '(begin (display "y") y))))
7
8 (test #f) ; 0
9 (test #t) ; y y: undefined
```



Varianta 2

quote & eval

```
1 (define (prod x y)
2   (if x (* (eval y) (+ (eval y) 1)) 0))
3
4 (define (test x)
5   (let ([y 5])
6     (prod x '(begin (display "y") y))))
7
8 (test #f) ; 0
9 (test #t) ; y y: undefined
```

- $x = \#f$ — comportament corect, y neevaluat



Varianta 2

quote & eval

```
1 (define (prod x y)
2   (if x (* (eval y) (+ (eval y) 1)) 0))
3
4 (define (test x)
5   (let ([y 5])
6     (prod x '(begin (display "y") y))))
7
8 (test #f) ; 0
9 (test #t) ; y y: undefined
```

- ▶ `x = #f` — comportament corect, `y` neevaluat
- ▶ `x = #t` — **eroare**, `quote` **nu** salvează contextul



Varianta 3

Închideri funcționale

```
1 (define (prod x y)
2   (if x (* (y) (+ (y) 1)) 0))
3
4 (define (test x)
5   (let ([y 5])
6     (prod x (lambda ()
7               (begin (display "y") y))))))
8
9 (test #f) ; 0
10 (test #t) ; yy 30
```



Varianta 3

Închideri funcționale

```
1 (define (prod x y)
2   (if x (* (y) (+ (y) 1)) 0))
3
4 (define (test x)
5   (let ([y 5])
6     (prod x (lambda ()
7               (begin (display "y") y))))))
8
9 (test #f) ; 0
10 (test #t) ; yy 30
```

- Comportament corect: y evaluat la cerere



Varianta 3

Închideri funcționale

```
1 (define (prod x y)
2   (if x (* (y) (+ (y) 1)) 0))
3
4 (define (test x)
5   (let ([y 5])
6     (prod x (lambda ()
7               (begin (display "y") y))))))
8
9 (test #f) ; 0
10 (test #t) ; yy 30
```

- Comportament corect: y evaluat la cerere
- $x = \#t$ — y evaluat de 2 ori, **ineficient**



Varianza 4

Promisiuni: `delay` & `force`

```
1 (define (prod x y)
2   (if x (* (force y) (+ (force y) 1)) 0))
3
4 (define (test x)
5   (let ([y 5])
6     (prod x (delay (begin (display "y") y)))))
7
8 (test #f) ; 0
9 (test #t) ; y 30
```



Varianta 4

Promisiuni: `delay` & `force`

```
1 (define (prod x y)
2   (if x (* (force y) (+ (force y) 1)) 0))
3
4 (define (test x)
5   (let ([y 5])
6     (prod x (delay (begin (display "y") y)))))
7
8 (test #f) ; 0
9 (test #t) ; y 30
```

Comportament corect: `y` evaluat **la cerere**, o **singură** dată



Varianța 4

Promisiuni: `delay` & `force`

```
1 (define (prod x y)
2   (if x (* (force y) (+ (force y) 1)) 0))
3
4 (define (test x)
5   (let ([y 5])
6     (prod x (delay (begin (display "y") y)))))
7
8 (test #f) ; 0
9 (test #t) ; y 30
```

Comportament corect: `y` evaluat **la cerere**, o **singură** dată
— evaluare **leneșă**



Promisiuni

Descriere

- ▶ Rezultatul încă **neevaluat** al unei expresii

Promisiuni

Descriere

- ▶ Rezultatul încă **neevaluat** al unei expresii
- ▶ Exemplu: `(delay (* 5 6))`



Promisiuni

Descriere

- ▶ Rezultatul încă **neevaluat** al unei expresii
- ▶ Exemplu: `(delay (* 5 6))`
- ▶ Valori de **prim rang** în limbaj (v. slide-ul 95)



Promisiuni

Descriere

- ▶ Rezultatul încă **neevaluat** al unei expresii
- ▶ Exemplu: (`delay` (* 5 6))
- ▶ Valori de **prim rang** în limbaj (v. slide-ul 95)
- ▶ `delay`



Promisiuni

Descriere

- ▶ Rezultatul încă **neevaluat** al unei expresii
- ▶ Exemplu: `(delay (* 5 6))`
- ▶ Valori de **prim rang** în limbaj (v. slide-ul 95)
- ▶ `delay`
 - ▶ construiește o promisiune



Promisiuni

Descriere

- ▶ Rezultatul încă **neevaluat** al unei expresii
- ▶ Exemplu: `(delay (* 5 6))`
- ▶ Valori de **prim rang** în limbaj (v. slide-ul 95)
- ▶ `delay`
 - ▶ construiește o promisiune
 - ▶ funcție nestrictă



Promisiuni

Descriere

- ▶ Rezultatul încă **neevaluat** al unei expresii
- ▶ Exemplu: `(delay (* 5 6))`
- ▶ Valori de **prim rang** în limbaj (v. slide-ul 95)
- ▶ `delay`
 - ▶ construiește o promisiune
 - ▶ funcție nestrictă
- ▶ `force`



Promisiuni

Descriere

- ▶ Rezultatul încă **neevaluat** al unei expresii
- ▶ Exemplu: `(delay (* 5 6))`
- ▶ Valori de **prim rang** în limbaj (v. slide-ul 95)
- ▶ `delay`
 - ▶ construiește o promisiune
 - ▶ funcție nestrictă
- ▶ `force`
 - ▶ forțează respectarea unei promisiuni, evaluând expresia doar la **prima** aplicare, și **salvându-i** valoarea



Promisiuni

Descriere

- ▶ Rezultatul încă **neevaluat** al unei expresii
- ▶ Exemplu: `(delay (* 5 6))`
- ▶ Valori de **prim rang** în limbaj (v. slide-ul 95)
- ▶ `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**



Observații

- ▶ **Dependență** între mecanismul de întârziere și cel de evaluare ulterioară a expresiilor — închideri/ aplicații (varianta 3), `delay/ force` (varianta 4) etc.
- ▶ Număr **mare** de modificări la **înlocuirea** unui mecanism existent, utilizat de un număr mare de funcții
- ▶ Cum se pot **diminua** dependențele?



Cuprins

Mecanisme

Abstractizare de date

Fluxuri

Rezolvarea problemelor prin căutare leneșă
în spațiul stărilor



Abstractizare de date I

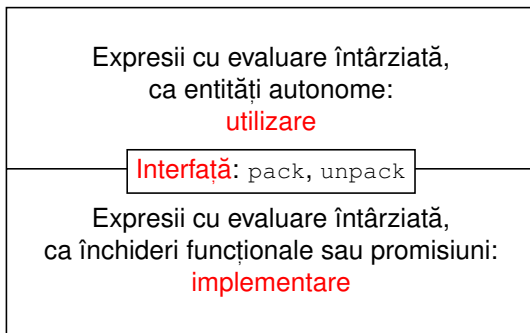
- ▶ Cum **reprezentăm** expresiile cu evaluare întârziată?
- ▶ Abordarea din secțiunea precedentă: **1** singur nivel

Expresii cu evaluare întârziată:
utilizare și **implementare**,
sub formă de închideri sau promisiuni



Abstractizare de date II

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



- ▶ Bariera:
 - ▶ **limitează** analiza detaliilor
 - ▶ **elimină** dependențele dintre nivele



Abstractizare de date III

- ▶ Tehnică de **separare** a utilizării unei structuri de date de implementarea acesteia.
- ▶ Permit *wishful thinking*: utilizarea structurii **înaintea** implementării acesteia



Abstractizare de date IV

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



Cuprins

Mecanisme

Abstractizare de date

Fluxuri

Rezolvarea problemelor prin căutare leneșă
în spațiul stărilor



Motivație

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

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



Comparație

- ▶ Varianta iterativă (d.p.d.v. proces):



Comparație

- ▶ Varianta iterativă (d.p.d.v. proces):
 - ▶ **eficientă**, datorită spațiului suplimentar constant



Comparație

- ▶ Varianta iterativă (d.p.d.v. proces):
 - ▶ **eficientă**, datorită spațiului suplimentar constant
 - ▶ **nu** foarte lizibilă



Comparație

- ▶ Varianta iterativă (d.p.d.v. proces):
 - ▶ **eficientă**, datorită spațiului suplimentar constant
 - ▶ **nu** foarte lizibilă
- ▶ Varianta pe liste:



Comparație

- ▶ Varianta iterativă (d.p.d.v. proces):
 - ▶ **eficientă**, datorită spațiului suplimentar constant
 - ▶ **nu** foarte lizibilă
- ▶ Varianta pe liste:
 - ▶ **elegantă** și concisă



Comparație

- ▶ Varianta iterativă (d.p.d.v. proces):
 - ▶ **eficientă**, datorită spațiului suplimentar constant
 - ▶ **nu** foarte lizibilă
- ▶ Varianta pe liste:
 - ▶ **elegantă** și concisă
 - ▶ **ineficientă**, datorită



Comparație

- ▶ Varianta iterativă (d.p.d.v. proces):
 - ▶ **eficientă**, datorită spațiului suplimentar constant
 - ▶ **nu** foarte lizibilă
- ▶ Varianta pe liste:
 - ▶ **elegantă** și concisă
 - ▶ **ineficientă**, datorită
 - ▶ spațiului posibil mare ocupat la un moment dat
— **toate** numerele din intervalul $[a, b]$



Comparație

- ▶ Varianta iterativă (d.p.d.v. proces):
 - ▶ **eficientă**, datorită spațiului suplimentar constant
 - ▶ **nu** foarte lizibilă
- ▶ Varianta pe liste:
 - ▶ **elegantă** și concisă
 - ▶ **ineficientă**, datorită
 - ▶ spațiului posibil mare ocupat la un moment dat
— **toate** numerele din intervalul $[a, b]$
 - ▶ parcurgerii **repetate** a intervalului
(`interval`, `filter`, `foldl`)



Comparație

- ▶ Varianta iterativă (d.p.d.v. proces):
 - ▶ **eficientă**, datorită spațiului suplimentar constant
 - ▶ **nu** foarte lizibilă
- ▶ Varianta pe liste:
 - ▶ **elegantă** și concisă
 - ▶ **ineficientă**, datorită
 - ▶ spațiului posibil mare ocupat la un moment dat
— **toate** numerele din intervalul $[a, b]$
 - ▶ parcurgerii **repetate** a intervalului
(`interval`, `filter`, `foldl`)
- ▶ Cum **îmbinăm** avantajele celor două abordări?



Caracteristicile fluxurilor

- ▶ Secvențe construite **parțial**, extinse la cerere, ce creează **iluzia** completitudinii structurii

Caracteristicile fluxurilor

- ▶ Secvențe construite **parțial**, extinse la cerere, ce creează **iluzia** completitudinii structurii
- ▶ Îmbinarea **eleganței** manipulării listelor cu **eficiența** calculului incremental



Caracteristicile fluxurilor

- ▶ Secvențe construite **parțial**, extinse la cerere, ce creează **iluzia** completitudinii structurii
- ▶ Îmbinarea **eleganței** manipulării listelor cu **eficiența** calculului incremental
- ▶ Bariera de abstractizare:



Caracteristicile fluxurilor

- ▶ 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`)



Caracteristicile fluxurilor

- ▶ 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`)
 - ▶ ale fluxurilor la **selecție** (`cdr`)



Caracteristicile fluxurilor

- ▶ 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`)
 - ▶ ale fluxurilor la **selecție** (`cdr`)
- ▶ Construcția și utilizarea:



Caracteristicile fluxurilor

- ▶ 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`)
 - ▶ ale fluxurilor la **selecție** (`cdr`)
- ▶ Construcția și utilizarea:
 - ▶ **separate** la nivel conceptual — **modularitate**



Caracteristicile fluxurilor

- ▶ 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`)
 - ▶ ale fluxurilor la **selecție** (`cdr`)
- ▶ Construcția și utilizarea:
 - ▶ **separate** la nivel conceptual — **modularitate**
 - ▶ **întrepătrunse** la nivel de proces



Operatori

```
3 (define-syntax-rule (stream-cons head tail)
4   (cons head (pack tail)))
5
6 (define stream-first car)
7
8 (define stream-rest (compose unpack cdr))
9
10 (define empty-stream '())
11
12 (define stream-empty? null?)
```



Barierele de abstractizare

Fluxuri,
ca entități autonome:
utilizare

Interfață: `stream-*`

Expresii cu evaluare întârziată,
ca entități autonome:
utilizare

Fluxuri, ca perechi conținând
expresii cu evaluare întârziată:
implementare

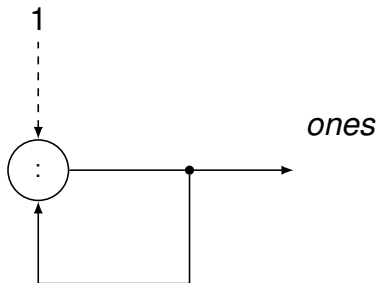
Interfață: `pack, unpack`

Expresii cu evaluare întârziată,
ca închideri funcționale sau promisiuni:
implementare

Fluxul de numere 1

Implementare

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



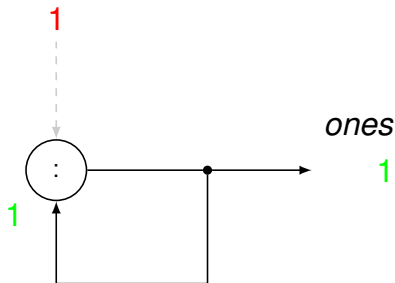
- ▶ Linii continue: fluxuri
- ▶ Linii întrerupte: intrări scalare, utilizate o singură dată
- ▶ Cifre: **intrări** / ieșiri



Fluxul de numere 1

Implementare

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



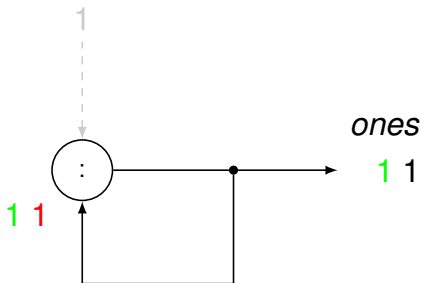
- ▶ Linii continue: fluxuri
- ▶ Linii întrerupte: intrări scalare, utilizate o singură dată
- ▶ Cifre: **intrări** / ieșiri



Fluxul de numere 1

Implementare

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



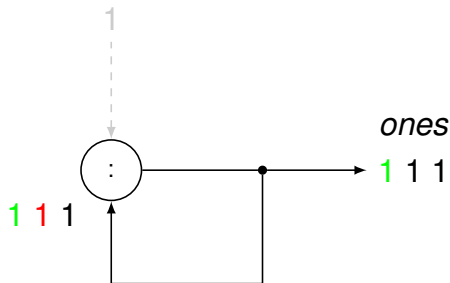
- ▶ Linii continue: fluxuri
- ▶ Linii întrerupte: intrări scalare, utilizate o singură dată
- ▶ Cifre: **intrări** / ieșiri



Fluxul de numere 1

Implementare

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



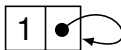
- ▶ Linii continue: fluxuri
- ▶ Linii întrerupte: intrări scalare, utilizate o singură dată
- ▶ Cifre: **intrări** / ieșiri



Fluxul de numere 1

Utilizarea memoriei

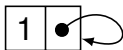
Atât cu închideri, cât și cu promisiuni, extinderea se realizează în spațiu constant:



Fluxul de numere 1

Utilizarea memoriei

Atât cu închideri, cât și cu promisiuni, extinderea se realizează în spațiu constant:

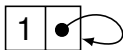


Alternativ: `(define ones (pack (cons 1 ones)))`

Fluxul de numere 1

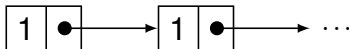
Utilizarea memoriei

Atât cu închideri, cât și cu promisiuni, extinderea se realizează în spațiu constant:



Alternativ: `(define ones (pack (cons 1 ones)))`

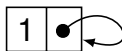
► închideri:



Fluxul de numere 1

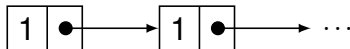
Utilizarea memoriei

Atât cu închideri, cât și cu promisiuni, extinderea se realizează în spațiu constant:

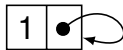


Alternativ: `(define ones (pack (cons 1 ones)))`

► închideri:



► promisiuni:



Fluxul numerelor naturale

Formulare explicită

```
10 (define (naturals-from n)
11     (stream-cons n (naturals-from (+ n 1))))
12
13 (define naturals (naturals-from 0))
```



Fluxul numerelor naturale

Formulare explicită

```
10 (define (naturals-from n)
11   (stream-cons n (naturals-from (+ n 1))))
12
13 (define naturals (naturals-from 0))
```

- Închideri: multiple parcurgeri ale fluxului determină **reevaluarea** porțiunilor deja explorate



Fluxul numerelor naturale

Formulare explicită

```
10 (define (naturals-from n)
11   (stream-cons n (naturals-from (+ n 1))))
12
13 (define naturals (naturals-from 0))
```

- ▶ Închideri: multiple parcurgeri ale fluxului determină **reevaluarea** porțiunilor deja explorate
 - ▶ Explorarea 1, cu 3 elemente: 0 1 2



Fluxul numerelor naturale

Formulare explicită

```
10 (define (naturals-from n)
11   (stream-cons n (naturals-from (+ n 1))))
12
13 (define naturals (naturals-from 0))
```

- ▶ Închideri: multiple parcurgeri ale fluxului determină **reevaluarea** porțiunilor deja explorate
 - ▶ Explorarea 1, cu 3 elemente: 0 1 2
 - ▶ Explorarea 2, cu 5 elemente: 0 1 2 3 4



Fluxul numerelor naturale

Formulare explicită

```
10 (define (naturals-from n)
11   (stream-cons n (naturals-from (+ n 1))))
12
13 (define naturals (naturals-from 0))
```

- ▶ Închideri: multiple parcurgeri ale fluxului determină **reevaluarea** porțiunilor deja explorate
 - ▶ Explorarea 1, cu 3 elemente: 0 1 2
 - ▶ Explorarea 2, cu 5 elemente: 0 1 2 3 4
- ▶ Promisiuni: multiple parcurgeri ale fluxului determină evaluarea **dincolo** de porțiunile deja explorate



Fluxul numerelor naturale

Formulare explicită

```
10 (define (naturals-from n)
11   (stream-cons n (naturals-from (+ n 1))))
12
13 (define naturals (naturals-from 0))
```

- ▶ Închideri: multiple parcurgeri ale fluxului determină **reevaluarea** porțiunilor deja explorate
 - ▶ Explorarea 1, cu 3 elemente: 0 1 2
 - ▶ Explorarea 2, cu 5 elemente: 0 1 2 3 4
- ▶ Promisiuni: multiple parcurgeri ale fluxului determină evaluarea **dincolo** de porțiunile deja explorate
 - ▶ Explorarea 1, cu 3 elemente: 0 1 2



Fluxul numerelor naturale

Formulare explicită

```
10 (define (naturals-from n)
11   (stream-cons n (naturals-from (+ n 1))))
12
13 (define naturals (naturals-from 0))
```

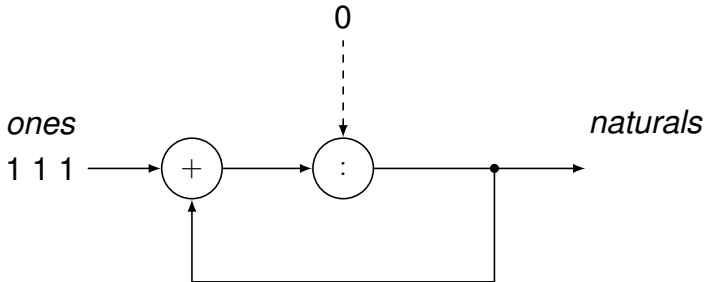
- ▶ Închideri: multiple parcurgeri ale fluxului determină **reevaluarea** porțiunilor deja explorate
 - ▶ Explorarea 1, cu 3 elemente: 0 1 2
 - ▶ Explorarea 2, cu 5 elemente: 0 1 2 3 4
- ▶ Promisiuni: multiple parcurgeri ale fluxului determină evaluarea **dincolo** de porțiunile deja explorate
 - ▶ Explorarea 1, cu 3 elemente: 0 1 2
 - ▶ Explorarea 2, cu 5 elemente: 0 1 2 3 4



Fluxul numerelor naturale

Formulare implicită

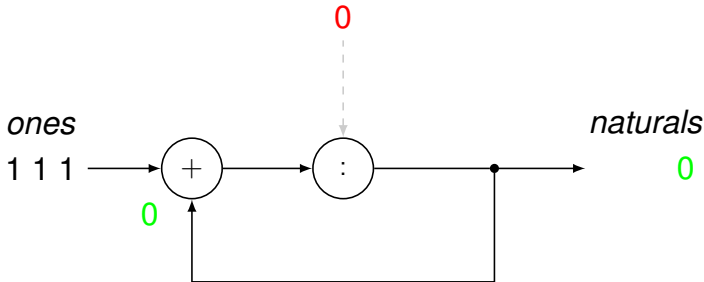
```
17 (define naturals
18   (stream-cons 0
19               (stream-zip-with +
20                               ones
21                               naturals)))
```



Fluxul numerelor naturale

Formulare implicită

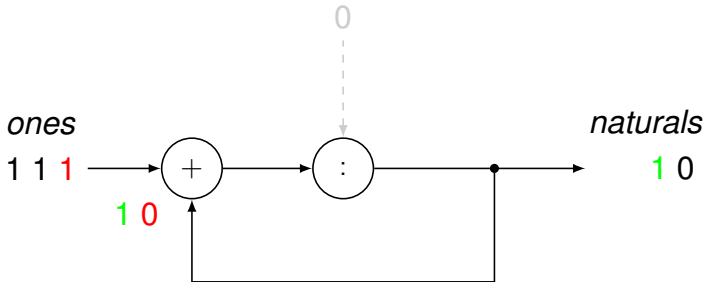
```
17 (define naturals
18   (stream-cons 0
19               (stream-zip-with +
20                               ones
21                               naturals)))
```



Fluxul numerelor naturale

Formulare implicită

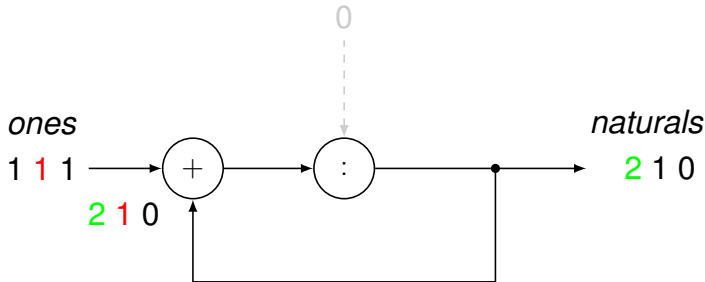
```
17 (define naturals
18   (stream-cons 0
19     (stream-zip-with +
20                       ones
21                       naturals)))
```



Fluxul numerelor naturale

Formulare implicită

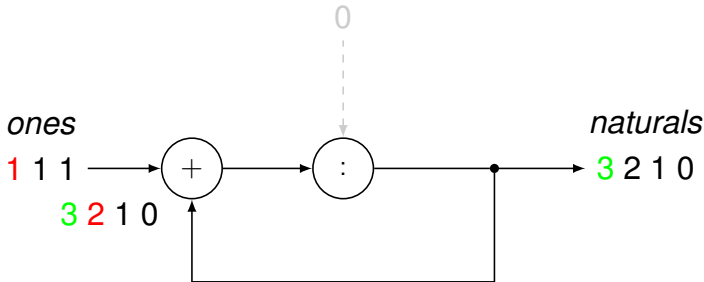
```
17 (define naturals
18   (stream-cons 0
19               (stream-zip-with +
20                               ones
21                               naturals)))
```



Fluxul numerelor naturale

Formulare implicită

```
17 (define naturals
18   (stream-cons 0
19               (stream-zip-with +
20                               ones
21                               naturals)))
```



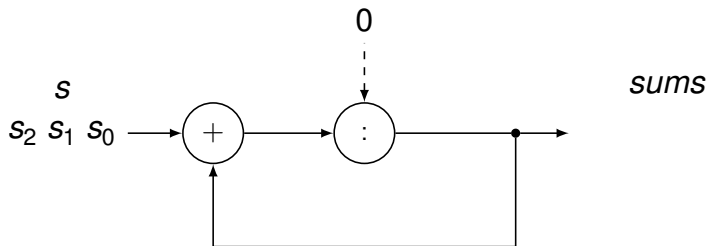
Fluxul numerelor pare

```
25 (define even-naturals-1
26   (stream-filter even? naturals))
27
28 (define even-naturals-2
29   (stream-zip-with + naturals naturals))
```



Fluxul sumelor parțiale ale altui flux

```
33 (define (sums s)
34   (letrec ([out (stream-cons
35               0
36               (stream-zip-with + s out))])
37     out))
```

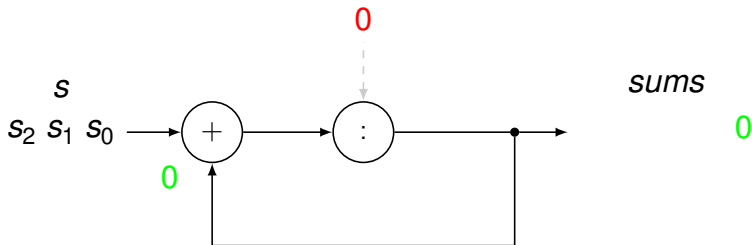


$$s_{i,j} = s_i + \dots + s_j$$



Fluxul sumelor parțiale ale altui flux

```
33 (define (sums s)
34   (letrec ([out (stream-cons
35               0
36               (stream-zip-with + s out))])
37     out))
```

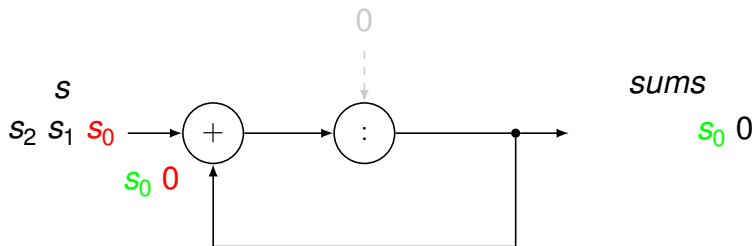


$$s_{i,j} = s_i + \dots + s_j$$



Fluxul sumelor parțiale ale altui flux

```
33 (define (sums s)
34   (letrec ([out (stream-cons
35               0
36               (stream-zip-with + s out))])
37     out))
```

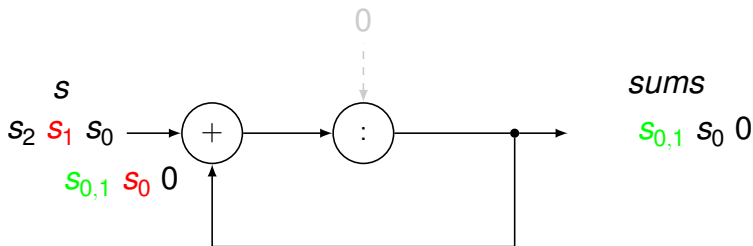


$$s_{i,j} = s_i + \dots + s_j$$



Fluxul sumelor parțiale ale altui flux

```
33 (define (sums s)
34   (letrec ([out (stream-cons
35               0
36               (stream-zip-with + s out))])
37     out))
```

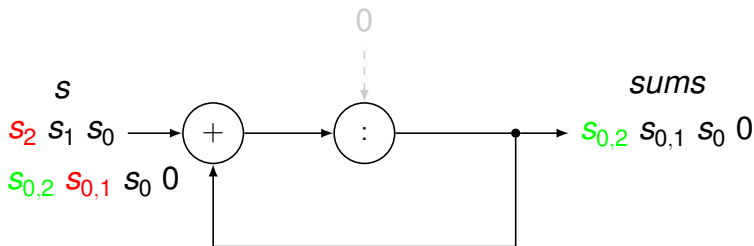


$$s_{i,j} = s_i + \dots + s_j$$



Fluxul sumelor parțiale ale altui flux

```
33 (define (sums s)
34   (letrec ([out (stream-cons
35               0
36               (stream-zip-with + s out))])
37     out))
```



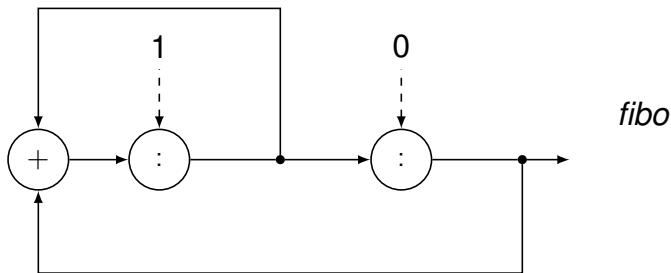
$$s_{i,j} = s_i + \dots + s_j$$



Fluxul numerelor Fibonacci

Formulare implicită

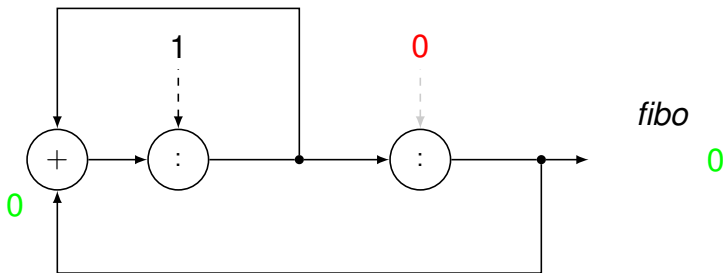
```
43 (define fibo
44   (stream-cons 0
45     (stream-cons 1
46       (stream-zip-with +
47         fibo
48         (stream-rest fibo))))))
```



Fluxul numerelor Fibonacci

Formulare implicită

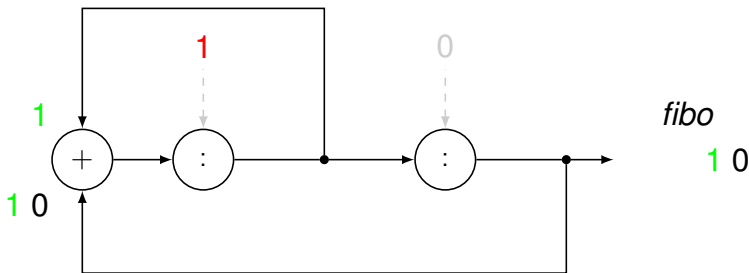
```
43 (define fibo
44   (stream-cons 0
45     (stream-cons 1
46       (stream-zip-with +
47         fibo
48         (stream-rest fibo))))))
```



Fluxul numerelor Fibonacci

Formulare implicită

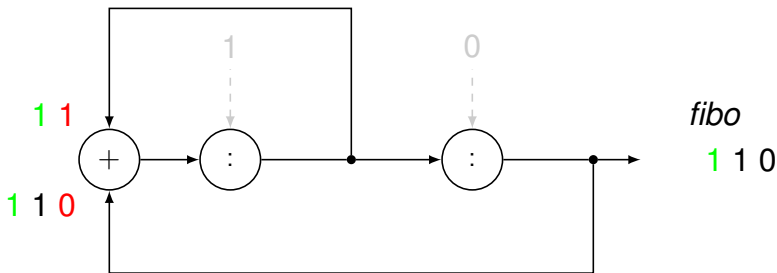
```
43 (define fibo
44   (stream-cons 0
45     (stream-cons 1
46       (stream-zip-with +
47         fibo
48         (stream-rest fibo))))))
```



Fluxul numerelor Fibonacci

Formulare implicită

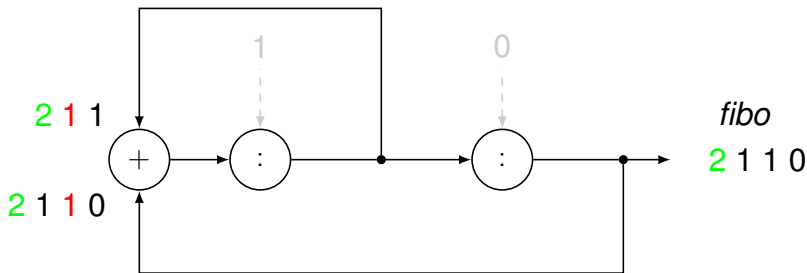
```
43 (define fibo
44   (stream-cons 0
45     (stream-cons 1
46       (stream-zip-with +
47         fibo
48         (stream-rest fibo))))))
```



Fluxul numerelor Fibonacci

Formulare implicită

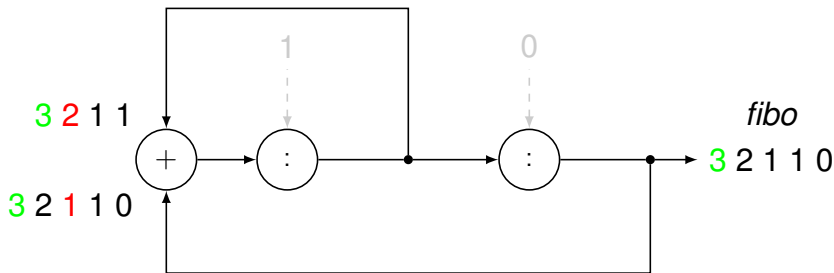
```
43 (define fibo
44   (stream-cons 0
45     (stream-cons 1
46       (stream-zip-with +
47         fibo
48         (stream-rest fibo))))))
```



Fluxul numerelor Fibonacci

Formulare implicită

```
43 (define fibo
44   (stream-cons 0
45     (stream-cons 1
46       (stream-zip-with +
47         fibo
48         (stream-rest fibo))))))
```



Fluxul numerelor prime I

- ▶ Ciurul lui **Eratostene**
- ▶ Pornim de la fluxul numerelor **naturale**, începând cu 2
- ▶ Elementul **curent** din fluxul inițial aparține fluxului numerelor prime
- ▶ **Restul** fluxului se obține
 - ▶ eliminând **multiplii** elementului curent din fluxul inițial
 - ▶ continuând procesul de **filtrare**, cu elementul următor

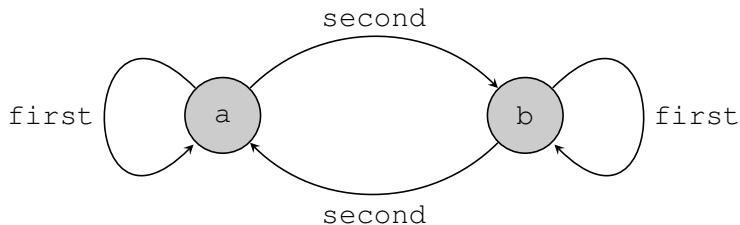


Fluxul numerelor prime II

```
52 (define (sieve s)
53   (if (stream-empty? s) s
54       (stream-cons
55         (stream-first s)
56         (sieve
57          (stream-filter
58           (lambda (n)
59             (not (zero? (remainder
60                          n
61                          (stream-first s))))))
62          (stream-rest s))))))
63
64 (define primes (sieve (naturals-from 2)))
```



Grafuri ciclice I



Fiecare nod conține:

- ▶ cheia: `key`
- ▶ legăturile către două noduri: `first`, `second`

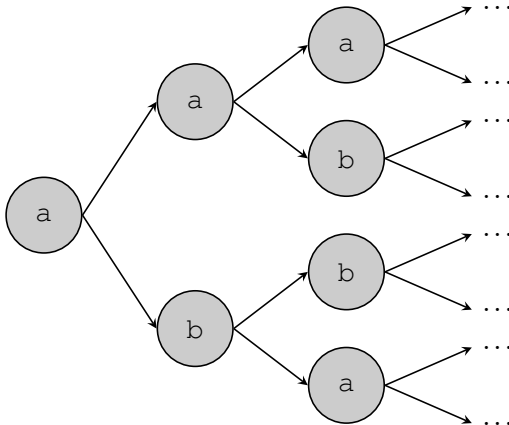
Grafuri ciclice II

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



Grafuri ciclice III

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



Cuprins

Mecanisme

Abstractizare de date

Fluxuri

Rezolvarea problemelor prin căutare leneșă
în spațiul stărilor



Spațiul stărilor unei probleme

Mulțimea configurațiilor valide din universul problemei



Problema palindroamelor

Definiție

- ▶ *Pal_n : Să se determine palindroamele de lungime cel puțin n , care se pot forma cu elementele unui alfabet fixat.*



Problema palindroamelor

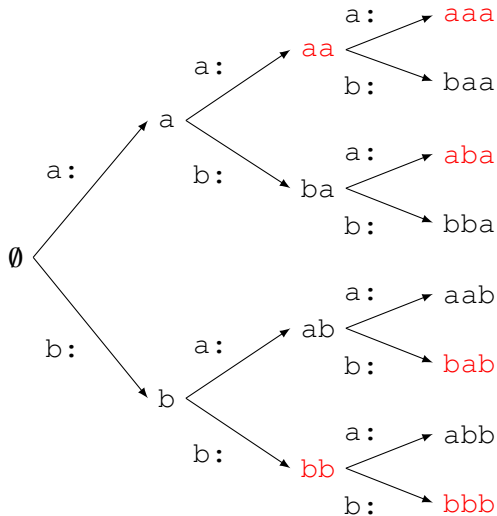
Definiție

- ▶ *Pal_n: Să se determine palindroamele de lungime cel puțin n , care se pot forma cu elementele unui alfabet fixat.*
- ▶ **Stările** problemei: **toate** șirurile generabile cu elementele alfabetului respectiv



Problema palindroamelor

Spațiul stărilor lui Pal_2



Problema palindroamelor

Specificare Pal_n

- Starea **inițială**: șirul vid



Problema palindroamelor

Specificare Pal_n

- ▶ Starea **inițială**: șirul vid
- ▶ Operatorii de generare a stărilor **succesoare** alteia:
inserarea unui caracter la începutul unui șir dat



Problema palindroamelor

Specificare Pal_n

- ▶ Starea **inițială**: șirul vid
- ▶ Operatorii de generare a stărilor **succesoare** alteia: inserarea unui caracter la începutul unui șir dat
- ▶ Operatorul de verificare a proprietății de **soluție** pentru o stare: palindrom, de lungime cel puțin n



Căutare în spațiul stărilor

- ▶ Spațiul stărilor ca **graf**:



Căutare în spațiul stărilor

- ▶ Spațiul stărilor ca graf:
 - ▶ noduri: stări



Căutare în spațiul stărilor

- ▶ Spațiul stărilor ca **graf**:
 - ▶ noduri: **stări**
 - ▶ muchii (orientate): **transformări** ale stărilor în stări succesori



Căutare în spațiul stărilor

- ▶ Spațiul stărilor ca **graf**:
 - ▶ noduri: **stări**
 - ▶ muchii (orientate): **transformări** ale stărilor în stări succesori
- ▶ Posibile strategii de **căutare**:



Căutare în spațiul stărilor

- ▶ Spațiul stărilor ca **graf**:
 - ▶ noduri: **stări**
 - ▶ muchii (orientate): **transformări** ale stărilor în stări succesori
- ▶ Posibile strategii de **căutare**:
 - ▶ lățime: **completă** și optimală



Căutare în spațiul stărilor

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



Căutare în lăţime

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



Căutare în lățime

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

- Generarea unei **singure** soluții



Căutare în lățime

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

- ▶ Generarea unei **singure** soluții
- ▶ Cum le obținem pe **celelalte**, mai ales dacă spațiul este **infin**it?



Căutare leneșă în lățime I

Fluxul stărilor soluție

```
3 (define (lazy-breadth-search init expand)
4   (let search
5     ([states (stream-cons init empty-stream)])
6     (if (stream-empty? states) states
7         (let ([state (stream-first states)]
8               [states (stream-rest states)])
9           (stream-cons
10            state
11            (search (stream-append
12                    states
13                    (expand state))))))))
14
15 (define (lazy-breadth-search-goal
16         init expand goal?)
17   (stream-filter goal?
```



Căutare leneșă în lățime II

Fluxul stărilor soluție

```
18                                     (lazy-breadth-search init
19                                     expand) ) )
```

- ▶ La nivel înalt, conceptual: **separare** între explorarea spațiului și identificarea stărilor soluție
- ▶ La nivelul scăzut, al instrucțiunilor: **întrepătrunderea** celor două aspecte



Aplicații

- ▶ Palindroame
- ▶ Problema reginelor



Problema reginelor

Definiție

- ▶ *Queens_n: Să se determine toate modurile de amplasare a n regine pe o tablă de șah de dimensiune n , astfel încât oricare două să nu se atace.*



Problema reginelor

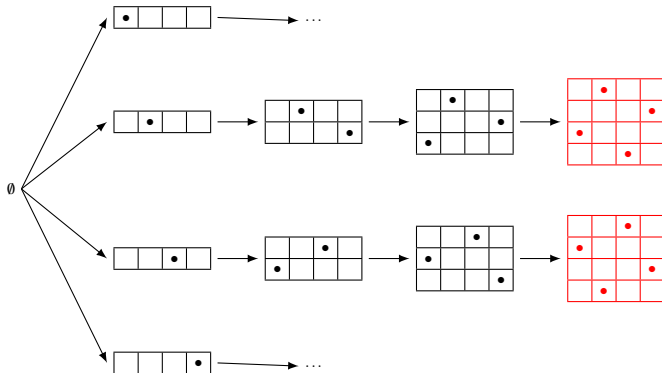
Definiție

- ▶ *Queens_n: Să se determine toate modurile de amplasare a n regine pe o tablă de șah de dimensiune n , astfel încât oricare două să nu se atace.*
- ▶ Stările problemei: configurațiile, eventual parțiale, ale **tablei**



Problema reginelor

Spațiul stărilor lui *Queens*₄



Rezumat

Evaluarea leneșă permite un stil de programare de **nivel înalt**, prin separarea aparentă a diverselor aspecte — de exemplu, construcția și accesarea listelor.



Bibliografie

Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition.



Partea VII

Limbajul Haskell



Cuprins

Introducere

Evaluare

Tipare

Sinteza de tip



Cuprins

Introducere

Evaluare

Tipare

Sinteza de tip



Paralelă între limbaje

Criteriu	Racket	Haskell
Funcții	<i>Curried / uncurried</i>	<i>Curried</i>
Evaluare	Aplicativă	Leneșă
Tipare	Dinamică, tare	Statică, tare
Legarea variabilelor	Locale $\rightarrow$ statică, (<i>top-level</i> $\rightarrow$ dinamică)	Statică

Funcții

- ▶ *Curried*
- ▶ Aplicabile asupra **oricâtor** parametri la un moment dat

```
1 add1 x y = x + y
2 add2      = \x y -> x + y
3 add3      = \x -> \y -> x + y
4 add4 x    = \y -> x + y
5
6 result    = add1 1 2    -- sau ((add1 1) 2)
7 inc       = add1 1      -- functie
```



Funcții și operatori

- ▶ Aplicabilitatea **parțială** a operatorilor infixati (secțiuni)
- ▶ **Transformări** operator→funcție și funcție→operator

```
1 add4      = (+)
2
3 result1 = (+) 1 2      -- operator ca functie
4 result2 = 1 `add4` 2   -- functie ca operator
5
6 inc1      = (1 +)      -- sectiuni
7 inc2      = (+ 1)
8 inc3      = (1 `add4`)
9 inc4      = (`add4` 1)
```



Pattern matching

Definirea comportamentului funcțiilor pornind de la **structura** parametrilor — traducerea axiomelor TDA

```
1  add5 0 y          = y          -- add5 1 2
2  add5 (x + 1) y    = 1 + add5 x y
3
4  listSum []         = 0          -- sumList [1, 2, 3]
5  listSum (hd : tl) = hd + listSum tl
6
7  pairSum (x, y)     = x + y     -- sumPair (1, 2)
8
9  wackySum (x, y, z@(hd : _)) =    -- wackySum
10     x + y + hd + listSum z      -- (1, 2, [3, 4, 5])
```



List comprehensions

Definirea listelor prin **proprietățile** elementelor, similar unei specificații matematice

```
1 squares lst    = [ x * x | x <- lst ]
2
3 qSort []       = []
4 qSort (h : t) =  qSort [ x | x <- t, x <= h ]
5                ++  [h]
6                ++  qSort [ x | x <- t, x > h ]
7
8 interval       = [ 0 .. 10 ]
9 evenInterval   = [ 0, 2 .. 10 ]
10 naturals      = [ 0 .. ]
```



Cuprins

Introducere

Evaluare

Tipare

Sinteza de tip



Evaluare

- ▶ Evaluare **leneșă**: parametri evaluați **la cerere**, **cel mult** o dată, eventual **parțial**, în cazul obiectelor structurate



Evaluare

- ▶ Evaluare **leneșă**: parametri evaluați **la cerere**, **cel mult** o dată, eventual **parțial**, în cazul obiectelor structurate
- ▶ Funcții **nestricte**!



Evaluare

- ▶ Evaluare **leneșă**: parametri evaluați **la cerere**, **cel mult** o dată, eventual **parțial**, în cazul obiectelor structurate
- ▶ Funcții **nestricte**!

```
1  f (x, y) z = x + x
2
3  f (2 + 3, 3 + 5) (5 + 8)
```



Evaluare

- ▶ Evaluare **leneșă**: parametri evaluați **la cerere**, **cel mult** o dată, eventual **parțial**, în cazul obiectelor structurate
- ▶ Funcții **nestricte**!

```
1 f (x, y) z = x + x
```

```
2
```

```
3 f (2 + 3, 3 + 5) (5 + 8)
```



Evaluare

- ▶ Evaluare **leneșă**: parametri evaluați **la cerere**, **cel mult** o dată, eventual **parțial**, în cazul obiectelor structurate
- ▶ Funcții **nestricte**!

```
1 f (x, y) z = x + x
```

```
2
```

```
3 f (2 + 3, 3 + 5) (5 + 8)
```

```
4 → (2 + 3) + (2 + 3)
```


Evaluare

- ▶ Evaluare **leneșă**: parametri evaluați **la cerere**, **cel mult** o dată, eventual **parțial**, în cazul obiectelor structurate
- ▶ Funcții **nestricte**!

```
1  f (x, y) z = x + x
2
3  f (2 + 3, 3 + 5) (5 + 8)
4  →  (2 + 3) + (2 + 3)
```



Evaluare

- ▶ Evaluare **leneșă**: parametri evaluați **la cerere**, **cel mult** o dată, eventual **parțial**, în cazul obiectelor structurate
- ▶ Funcții **nestricte**!

```
1  f (x, y) z = x + x
```

```
2
```

```
3  f (2 + 3, 3 + 5) (5 + 8)
```

```
4  → (2 + 3) + (2 + 3)
```

```
5  → 5 + 5    -- reutilizam rezultatul primei evaluari
```



Evaluare

- ▶ Evaluare **leneșă**: parametri evaluați **la cerere**, **cel mult** o dată, eventual **parțial**, în cazul obiectelor structurate
- ▶ Funcții **nestricte**!

```
1  f (x, y) z = x + x
```

```
2
```

```
3  f (2 + 3, 3 + 5) (5 + 8)
```

```
4  →   (2 + 3)  +  (2 + 3)
```

```
5  →   5 + 5      -- reutilizam rezultatul primei evaluari
```



Evaluare

- ▶ Evaluare **leneșă**: parametri evaluați **la cerere**, **cel mult** o dată, eventual **parțial**, în cazul obiectelor structurate
- ▶ Funcții **nestricte**!

```
1  f (x, y) z = x + x
```

```
2
```

```
3  f (2 + 3, 3 + 5) (5 + 8)
```

```
4  →  (2 + 3)  +  (2 + 3)
```

```
5  →  5 + 5      -- reutilizam rezultatul primei evaluari
```

```
6  →  10
```



Pași în aplicarea funcțiilor I

```
1 front (x : y : zs) = x + y
2 front [x]           = x
3
4 notNil []           = False
5 notNil (_ : _)      = True
6
7 f m n
8     | notNil xs      = front xs
9     | otherwise      = n
10 where
11     xs               = [m .. n]
```

Exemplu preluat din Thompson (1999)



Pași în aplicarea funcțiilor II

1. *Pattern matching*: evaluarea parametrilor **suficient** cât să se constate (ne-)potrivirea cu *pattern*-ul
2. Evaluarea **gărzilor** (`|`)
3. Evaluarea variabilelor **locale**, **la cerere** (`where`, `let`)



Pași în aplicarea funcțiilor III

```
1  f 3 5
2  ??  notNil xs
3  ??    where
4  ??      xs = [3 .. 5]
5  ??      → 3 : [4 .. 5]
6  ??  → notNil (3 : [4 .. 5])
7  ??  → True
8  → front xs
9      where
10     xs = 3 : [4 .. 5]
11     → 3 : 4 : [5]
12 → front (3 : 4 : [5])
13 → 3 + 4
14 → 7
```



Consecințe

- ▶ Evaluarea **parțială** a obiectelor structurate (liste etc.)
- ▶ Liste, implicit, ca **fluxuri**!

```
1  ones                = 1 : ones
2
3  naturalsFrom n      = n : (naturalsFrom (n + 1))
4  naturals1           = naturalsFrom 0
5  naturals2           = 0 : (zipWith (+) ones naturals2)
6
7  evenNaturals1       = filter even naturals1
8  evenNaturals2       = zipWith (+) naturals1 naturals2
9
10 fibo                = 0 : 1 :
11                     (zipWith (+) fibo (tail fibo))
```



Cuprins

Introducere

Evaluare

Tipare

Sinteza de tip



Tipuri

- ▶ Tipuri ca **mulțimi** de valori:
 - ▶ `Bool = {True, False}`
 - ▶ `Natural = {0, 1, 2, ...}`
 - ▶ `Char = {'a', 'b', 'c', ...}`



Tipuri

- ▶ Tipuri ca **mulțimi** de valori:
 - ▶ `Bool = {True, False}`
 - ▶ `Natural = {0, 1, 2, ...}`
 - ▶ `Char = {'a', 'b', 'c', ...}`
- ▶ Tipare **statică**:
 - ▶ etapa de tipare **anterioară** etapei de evaluare
 - ▶ asocierea fiecărei **expresii** din program cu un tip



Tipuri

- ▶ Tipuri ca **mulțimi** de valori:
 - ▶ `Bool = {True, False}`
 - ▶ `Natural = {0, 1, 2, ...}`
 - ▶ `Char = {'a', 'b', 'c', ...}`
- ▶ Tipare **statică**:
 - ▶ etapa de tipare **anterioară** etapei de evaluare
 - ▶ asocierea fiecărei **expresii** din program cu un tip
- ▶ Tipare **tare**: **absența** conversiilor implicite de tip



Tipuri

- ▶ Tipuri ca **mulțimi** de valori:
 - ▶ `Bool = {True, False}`
 - ▶ `Natural = {0, 1, 2, ...}`
 - ▶ `Char = {'a', 'b', 'c', ...}`
- ▶ Tipare **statică**:
 - ▶ etapa de tipare **anterioară** etapei de evaluare
 - ▶ asocierea fiecărei **expresii** din program cu un tip
- ▶ Tipare **tare**: **absența** conversiilor implicite de tip
- ▶ Expresii de:
 - ▶ **program**: `5, 2 + 3, x && (not y)`
 - ▶ **tip**: `Integer, [Char], Char -> Bool, a`



Exemple de tipuri

```
1  5                :: Integer
2  'a'              :: Char
3  inc              :: Integer -> Integer
4  [1,2,3]          :: [Integer]
5  (True, "Hello") :: (Bool, [Char])
```



Tipuri de bază

- ▶ Tipurile ale căror valori **nu** pot fi descompuse
- ▶ Exemple:
 - ▶ Bool
 - ▶ Char
 - ▶ Integer
 - ▶ Int
 - ▶ Float



Constructori de tip

“Funcții” de tip, care generează tipuri noi pe baza celor existente

```
1  -- Constructorul de tip lista: []
2  ([] Bool) ⇒ [Bool]
3  ([] [Bool]) ⇒ [[Bool]]
4
5  -- Constructorul de tip tuplu: (,...,)
6  ((,) Bool Char) ⇒ (Bool, Char)
7  ((,,) Bool ((,) Char [Bool]) Bool)
8      ⇒ (Bool, (Char, [Bool]), Bool)
9
10 -- Constructorul de tip functie: ->
11 (-> Bool Bool) ⇒ Bool -> Bool
12 (-> Bool (Bool -> Bool)) ⇒ Bool -> (Bool -> Bool)
```



Tipurile funcțiilor

Constructorul “->” asociativ la dreapta:

`Integer -> Integer -> Integer`
`≡ Integer -> (Integer -> Integer)`

```
1  add6      :: Integer -> Integer -> Integer
2  add6 x y =  x + y
3
4  f         :: (Integer -> Integer) -> Integer
5  f g      =  (g 3) + 1
6
7  idd       :: a -> a    -- functie polimorfica
8  idd x     =  x        -- a: variabila de tip!
```



Polimorfism

- ▶ *Parametric*: manifestarea **aceluiași** comportament pentru parametri de tipuri **diferite**. Exemplu: `idd`
- ▶ *Ad-hoc*: manifestarea unor comportamente **diferite** pentru parametri de tipuri **diferite**. Exemplu: `(==)`



Constructorul de tip `Natural` I

Definit de utilizator

```
1 data Natural
2     = Zero
3     | Succ Natural
4     deriving (Show, Eq)
5
6 unu           = Succ Zero
7 doi           = Succ unu
8
9 addNat Zero n = n
10 addNat (Succ m) n = Succ (addNat m n)
```



Constructorul de tip `Natural` II

Definit de utilizator

- ▶ Constructor de **tip**: `Natural`
 - ▶ nular
 - ▶ **se confundă** cu tipul pe care-l construiește
- ▶ Constructori de **date**:
 - ▶ `Zero`: nular
 - ▶ `Succ`: unar
- ▶ Constructorii de date ca **funcții**, utilizabile în *pattern matching*

```
1 Zero :: Natural
2 Succ :: Natural -> Natural
```



Constructorul de tip `Pair` I

Definit de utilizator

```
1 data Pair a b
2     = P a b
3     deriving (Show, Eq)
4
5 pair1          = P 2 True
6 pair2          = P 1 pair1
7
8 myFst (P x y)  = x
9 mySnd (P x y)  = y
```



Constructorul de tip `Pair` II

Definit de utilizator

- ▶ Constructor de **tip**: `Pair`
 - ▶ polimorfic, binar
 - ▶ generează un tip în momentul **aplicării** asupra 2 tipuri
- ▶ Constructor de **date**: `P`, binar

```
1 P :: a -> b -> Pair a b
```



Uniformitatea reprezentării tipurilor

```
1 data Integer    =    ... | -2 | -1 | 0 | 1 | 2 | ...
2
3 data Char       =    'a' | 'b' | 'c' | ...
4
5 data [a]        =    [] | a : [a]
6
7 data (a, b)     =    (a, b)
```



Cuprins

Introducere

Evaluare

Tipare

Sinteza de tip



Sinteza de tip

- ▶ Definiție: determinarea **automată** a tipului unei expresii, pe baza unor reguli precise

Sinteza de tip

- ▶ Definiție: determinarea **automată** a tipului unei expresii, pe baza unor reguli precise
- ▶ Adnotările **explicite** de tip, deși posibile, **necesare** în majoritatea cazurilor

Sinteza de tip

- ▶ Definiție: determinarea **automată** a tipului unei expresii, pe baza unor reguli precise
- ▶ Adnotările **explicite** de tip, deși posibile, **necesare** în majoritatea cazurilor
- ▶ Dependentă de:

Sinteza de tip

- ▶ Definiție: determinarea **automată** a tipului unei expresii, pe baza unor reguli precise
- ▶ Adnotările **explicite** de tip, deși posibile, **necesare** în majoritatea cazurilor
- ▶ Dependentă de:
 - ▶ **componentele** expresiei

Sinteza de tip

- ▶ Definiție: determinarea **automată** a tipului unei expresii, pe baza unor reguli precise
- ▶ Adnotările **explicite** de tip, deși posibile, **necesare** în majoritatea cazurilor
- ▶ Dependentă de:
 - ▶ **componentele** expresiei
 - ▶ **contextul** lexical al expresiei

Sinteza de tip

- ▶ Definiție: determinarea **automată** a tipului unei expresii, pe baza unor reguli precise
- ▶ Adnotările **explicite** de tip, deși posibile, **necesare** în majoritatea cazurilor
- ▶ Dependentă de:
 - ▶ **componentele** expresiei
 - ▶ **contextul** lexical al expresiei
- ▶ Reprezentarea tipurilor prin **expresii** de tip:

Sinteza de tip

- ▶ Definiție: determinarea **automată** a tipului unei expresii, pe baza unor reguli precise
- ▶ Adnotările **explicite** de tip, deși posibile, **necesare** în majoritatea cazurilor
- ▶ Dependentă de:
 - ▶ **componentele** expresiei
 - ▶ **contextul** lexical al expresiei
- ▶ Reprezentarea tipurilor prin **expresii** de tip:
 - ▶ **constante** de tip: tipuri de bază (`Int`)



Sinteza de tip

- ▶ Definiție: determinarea **automată** a tipului unei expresii, pe baza unor reguli precise
- ▶ Adnotările **explicite** de tip, deși posibile, **necesare** în majoritatea cazurilor
- ▶ Dependentă de:
 - ▶ **componentele** expresiei
 - ▶ **contextul** lexical al expresiei
- ▶ Reprezentarea tipurilor prin **expresii** de tip:
 - ▶ **constante** de tip: tipuri de bază (`Int`)
 - ▶ **variabile** de tip: pot fi legate la orice expresii de tip (`a`)



Sinteza de tip

- ▶ Definiție: determinarea **automată** a tipului unei expresii, pe baza unor reguli precise
- ▶ Adnotările **explicite** de tip, deși posibile, **necesare** în majoritatea cazurilor
- ▶ Dependentă de:
 - ▶ **componentele** expresiei
 - ▶ **contextul** lexical al expresiei
- ▶ Reprezentarea tipurilor prin **expresii** de tip:
 - ▶ **constante** de tip: tipuri de bază (`Int`)
 - ▶ **variabile** de tip: pot fi legate la orice expresii de tip (`a`)
 - ▶ **aplicații** ale constructorilor de tip asupra expresiilor de tip (`[a]`)



Reguli simplificate de sinteză de tip I

► Forma generală:

$$\frac{\text{premise-1} \dots \text{premise-m}}{\text{concluzie-1} \dots \text{concluzie-n}} \quad (\text{nume})$$

► Funcție:

$$\frac{\text{Var} :: a \quad \text{Expr} :: b}{\backslash \text{Var} \rightarrow \text{Expr} :: a \rightarrow b} \quad (\text{TLambda})$$

► Aplicație:

$$\frac{\text{Expr1} :: a \rightarrow b \quad \text{Expr2} :: a}{(\text{Expr1 Expr2}) :: b} \quad (\text{TApp})$$



Reguli simplificate de sinteză de tip II

► Operatorul +:

$$\frac{\text{Expr1} :: \text{Int} \quad \text{Expr2} :: \text{Int}}{\text{Expr1} + \text{Expr2} :: \text{Int}} \quad (\text{T+})$$

► Literalii întregi:

$$\frac{}{0, 1, 2, \dots :: \text{Int}} \quad (\text{TInt})$$



Exemple de sinteză de tip I

$f\ g = (g\ 3) + 1$

$$\frac{g :: a \quad (g\ 3) + 1 :: b}{f :: a \rightarrow b} \quad (\text{TLambda})$$
$$\frac{(g\ 3) :: \text{Int} \quad 1 :: \text{Int}}{(g\ 3) + 1 :: \text{Int}} \quad (\text{T+}, \text{TInt})$$

$b = \text{Int}$

$$\frac{g :: c \rightarrow d \quad 3 :: c}{(g\ 3) :: d} \quad (\text{TApp})$$

$a = c \rightarrow d, c = \text{Int}, d = \text{Int}$

$f :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$



Exemple de sinteză de tip II

`fix f = f (fix f)`

$$\frac{f :: a \quad f \text{ (fix f) } :: b}{\text{fix} :: a \rightarrow b} \quad (\text{TLambda})$$

$$\frac{f :: c \rightarrow d \quad (\text{fix f}) :: c}{f \text{ (fix f) } :: d} \quad (\text{TApp})$$

`a = c -> d, b = d`

$$\frac{\text{fix} :: e \rightarrow g \quad f :: e}{(\text{fix f}) :: g} \quad (\text{TApp})$$

`a -> b = e -> g, a = e, b = g, c = g`

`f :: (c -> d) -> b = (g -> g) -> g`



Exemple de sinteză de tip III

$$f \ x = (x \ x)$$

$$\frac{x :: a \quad (x \ x) :: b}{f :: a \rightarrow b} \quad (\text{TLambda})$$

$$\frac{x :: c \rightarrow d \quad x :: c}{(x \ x) :: d} \quad (\text{TApp})$$

Ecuatia $c \rightarrow d = c$ **nu** are soluție,
deci funcția **nu** poate fi tipată.



Unificare I

- ▶ Sinteza de tip presupune **legarea** variabilelor de tip în scopul **unificării** diverselor expresii de tip obținute
- ▶ Unificare = procesul de identificare a valorilor **variabilelor** din 2 sau mai multe expresii, astfel încât **substituirea** variabilelor prin valorile asociate să conducă la **coincidența** expresiilor
- ▶ Substituție = mulțime de **legări** variabilă-valoare

Unificare II

Exemplu:

- ▶ Expresii:

- ▶ $t1 = (a, [b])$

- ▶ $t2 = (Int, c)$

- ▶ Substituții:

- ▶ $S1 = \{a \leftarrow Int, b \leftarrow Int, c \leftarrow [Int]\}$

- ▶ $S2 = \{a \leftarrow Int, c \leftarrow [b]\}$

- ▶ Forme comune:

- ▶ $t1/S1 = t2/S1 = (Int, [Int])$

- ▶ $t1/S2 = t2/S2 = (Int, [b])$

Most general unifier (MGU) = cea mai **generală** substituție sub care expresiile unifică. Exemplu: $S2$.



Unificare III

- ▶ O **variabilă** de tip, a , unifică cu o **expresie** de tip, E , doar dacă:
 - ▶ $E = a$ sau
 - ▶ $E \neq a$ și E nu conține a (*occurrence check*).
- ▶ **2 constante** de tip unifică doar dacă sunt egale.
- ▶ **2 aplicații** de tip unifică doar dacă implică același constructor de tip și argumente ce unifică recursiv.



Tip principal

Exemplu:

- ▶ Funcție: $\lambda x. x$
- ▶ Tipuri corecte:
 - ▶ $\text{Int} \rightarrow \text{Int}$
 - ▶ $\text{Bool} \rightarrow \text{Bool}$
 - ▶ $a \rightarrow a$
- ▶ Unele tipuri se obțin prin **instanțierea** altora.

Tip principal al unei expresii = cel mai **general** tip care descrie **complet** natura expresiei. Se obține prin utilizarea MGU.



Rezumat

- ▶ Evaluare leneșă
- ▶ Tipare statică și tare, anterioară evaluării



Bibliografie

Thompson, S. (1999). *Haskell: The Craft of Functional Programming*. Ediția a doua. Addison-Wesley.



Partea VIII

Evaluare leneșă în Haskell



Cuprins



Suma pătratelor

Suma pătratelor numerelor naturale până la n ,
ca sumă a elementelor unei **liste**:



Suma pătratelor

Suma pătratelor numerelor naturale până la n ,
ca sumă a elementelor unei **liste**:

```
1 sum (map (^2) [1 .. n])
```



Suma pătratelor

Suma pătratelor numerelor naturale până la n ,
ca sumă a elementelor unei **liste**:

```
1 sum (map (^2) [1 .. n])  
2 → sum (map (^2) 1 : [2 .. n])
```



Suma pătratelor

Suma pătratelor numerelor naturale până la n ,
ca sumă a elementelor unei **liste**:

```
1 sum (map (^2) [1 .. n])  
2 → sum (map (^2) 1 : [2 .. n])  
3 → sum (1^2 : (map (^2) [2 .. n]))
```



Suma pătratelor

Suma pătratelor numerelor naturale până la n ,
ca sumă a elementelor unei **liste**:

```
1 sum (map (^2) [1 .. n])  
2 → sum (map (^2) 1 : [2 .. n])  
3 → sum (1^2 : (map (^2) [2 .. n]))  
4 → 1^2 + sum (map (^2) [2 .. n])
```



Suma pătratelor

Suma pătratelor numerelor naturale până la n ,
ca sumă a elementelor unei **liste**:

```
1 sum (map (^2) [1 .. n])
2 → sum (map (^2) 1 : [2 .. n])
3 → sum (1^2 : (map (^2) [2 .. n]))
4 → 1^2 + sum (map (^2) [2 .. n])
5 → 1 + sum (map (^2) [2 .. n])
```



Suma pătratelor

Suma pătratelor numerelor naturale până la n ,
ca sumă a elementelor unei **liste**:

```
1 sum (map (^2) [1 .. n])
2 → sum (map (^2) 1 : [2 .. n])
3 → sum (1^2 : (map (^2) [2 .. n]))
4 → 1^2 + sum (map (^2) [2 .. n])
5 → 1 + sum (map (^2) [2 .. n])
6 ...
7 → 1 + (4 + sum (map (^2) [3 .. n]))
```



Suma pătratelor

Suma pătratelor numerelor naturale până la n ,
ca sumă a elementelor unei **liste**:

```
1 sum (map (^2) [1 .. n])
2 → sum (map (^2) 1 : [2 .. n])
3 → sum (1^2 : (map (^2) [2 .. n]))
4 → 1^2 + sum (map (^2) [2 .. n])
5 → 1 + sum (map (^2) [2 .. n])
6 ...
7 → 1 + (4 + sum (map (^2) [3 .. n]))
8 ...
9 → 1 + (4 + (9 + ... + n^2))
```



Suma pătratelor

Suma pătratelor numerelor naturale până la n ,
ca sumă a elementelor unei **liste**:

```
1 sum (map (^2) [1 .. n])
2 → sum (map (^2) 1 : [2 .. n])
3 → sum (1^2 : (map (^2) [2 .. n]))
4 → 1^2 + sum (map (^2) [2 .. n])
5 → 1 + sum (map (^2) [2 .. n])
6 ...
7 → 1 + (4 + sum (map (^2) [3 .. n]))
8 ...
9 → 1 + (4 + (9 + ... + n^2))
```

Nicio listă **nu** este efectiv construită în timpul evaluării.



Elementul minim al unei liste I

Elementul minim al unei liste, drept prim element al acesteia, după **sortarea** prin inserție (Thompson, 1999):

```
34 ins x []          = [x]
35 ins x (h : t)
36     | x <= h      = x : h : t
37     | otherwise = h : (ins x t)
38
39 isort []          = []
40 isort (h : t)     = ins h (isort t)
41
42 minList l = head (isort l)
```



Elementul minim al unei liste II

```
45      minList [3, 2, 1]
46      = head (isort [3, 2, 1])
47      = head (isort (3 : [2, 1]))
48      = head (ins 3 (isort [2, 1]))
49      = head (ins 3 (isort (2 : [1])))
50      = head (ins 3 (ins 2 (isort [1])))
51      = head (ins 3 (ins 2 (isort (1 : []))))
52      = head (ins 3 (ins 2 (ins 1 (isort []))))
53      = head (ins 3 (ins 2 (ins 1 [])))
54      = head (ins 3 (ins 2 (1 : [])))
55      = head (ins 3 (1 : ins 2 []))
56      = head (1 : (ins 3 (ins 2 [])))
57      = 1
```

Lista **nu** este efectiv sortată, minimul fiind, pur și simplu, adus în fața acesteia și întors.



Accesibilitatea într-un graf orientat

Accesibilitatea între două noduri dintr-un graf $\Leftrightarrow$ existența elementelor în mulțimea **tuturor** căilor dintre cele două noduri (Thompson, 1999):

```
75 routes source dest graph explored
76     | source == dest = [[source]]
77     | otherwise      = [ source : path
78                           | neighbor <- neighbors source
79                           | graph \\ explored
80                           | path <- routes neighbor dest
81                           | graph (source : explored)
82                           ]
83 accessible source dest graph =
84     (routes source dest graph []) /= []
```

Backtracking desfășurat doar până la determinarea **primului** element al listei de căi.



Evaluarea leneșă

- ▶ *Programare orientată spre date*: exprimarea unor prelucrări în termenii unor operații pe **structuri de date**, posibil **niciodată** generate complet (suma pătratelor, sortare)
- ▶ Backtracking eficient: găsirea unui obiect cu o anumită proprietate, prin generarea aparentă a **tuturor** celor care îndeplinesc proprietatea respectivă (accesibilitatea în graf)
- ▶ Pilon al **modularității** eficiente — prelucrări **distincte** ale unei structuri, aplicate într-o **singură** parcurgere!



Studiu de caz

Biblioteca de parsare (Thompson, 1999)



Bibliografie

Thompson, S. (1999). *Haskell: The Craft of Functional Programming*. Ediția a doua. Addison-Wesley.



Partea IX

Polimorfism ad-hoc în Haskell



Cuprins

Clase

Aplicație pentru clase



Cuprins

Clase

Aplicație pentru clase



Motivație

Să se definească operația `show`, capabilă să producă reprezentarea oricărui obiect ca șir de caractere. Comportamentul este **specific** fiecărui tip.

```
1 show 3 → "3"  
2 show True → "True"  
3 show 'a' → "'a'"  
4 show "a" → "\"a\""
```



Varianta 1 I

Funcții dedicate fiecărui tip

```
1 show4Bool True  = "True"
2 show4Bool False = "False"
3
4 show4Char c      = "'" ++ [c] ++ "'"
5
6 show4String s    = "\"" ++ s ++ "\""
```



Varianta 1 II

Funcții dedicate fiecărui tip

- ▶ Funcția `showNewLine`, care adaugă caracterul “linie nouă” la reprezentarea ca șir:

```
1 showNewLine x = (show... x) ++ "\n"
```

- ▶ `showNewLine` **nu** poate fi polimorfică
→ `showNewLine4Bool`, `showNewLine4Char` etc.

- ▶ Alternativ, trimiterea ca **parametru** a funcției `show*`, corespunzătoare:

```
1 showNewLine sh x = (sh x) ++ "\n"  
2 showNewLine4Bool = showNewLine show4Bool
```

- ▶ **Prea general**, fiind posibilă trimiterea unei funcții cu alt comportament, în măsura în care respectă tipul



Varianta 2 I

Supraîncărcarea funcției

- Definirea **mulțimii** `Show`, a tipurilor care expun `show`:

```
1 class Show a where  
2     show :: a -> String  
3     ...
```

- Precizarea **aderenței** unui tip la această mulțime:

```
1 instance Show Bool where  
2     show True  = "True"  
3     show False = "False"  
4  
5 instance Show Char where  
6     show c = "'" ++ [c] ++ "'"
```

- Funcția `showNewLine` **polimorfică**!

```
1 showNewLine x = (show x) ++ "\n"
```



Varianta 2 II

Supraîncărcarea funcției

- ▶ Ce **tip** au funcțiile `show`, respectiv `showNewLine`?

```
1 show           :: Show a => a -> String  
2 showNewLine :: Show a => a -> String
```

- ▶ “Dacă tipul `a` este membru al clasei `Show`, i.e. funcția `show` este definită pe valorile tipului `a`, atunci funcțiile au tipul `a -> String`.”
- ▶ **Context**: constrângeri suplimentare asupra variabilelor din tipul funcției: `Show a`
- ▶ **Propagarea** constrângerilor din contextul lui `show` către contextul lui `showNewLine`



Varianta 2 III

Supraîncărcarea funcției

- Contexte utilizabile și la **instanțiere**:

```
1 instance (Show a, Show b) => Show (a, b) where
2     show (x, y) = "(" ++ (show x)
3                   ++ ", " ++ (show y)
4                   ++ ") "
```

- “**Ori de câte ori** tipurile *a* și *b* aparțin clasei `Show`,
tipul `(a, b)` îi aparține de asemenea.”



Clase

- ▶ Clasă = **mulțime** de tipuri ce supraîncarcă operațiile specifice clasei
- ▶ Modalitate structurată de control al polimorfismului **ad-hoc**
- ▶ Exemplu: clasa `Show`, cu operația `show`



Instanțe ale claselor

- ▶ Instanță = **tip** care supraîncarcă operațiile clasei
- ▶ Exemplu: tipul `Bool`, în raport cu clasa `Show`



Clase predefinite I

```
1 class Show a where
2     show :: a -> String
3     ...
4
5 class Eq a where
6     (==), (/=) :: a -> a -> Bool
7     x /= y      = not (x == y)
8     x == y      = not (x /= y)
```

- Posibilitatea scrierii de definiții **implicite** (v. liniile 7–8)
- Necesitatea suprascrierii **cel puțin unuia** dintre cei doi operatori ai clasei `Eq`, pentru instanțierea corectă



Clase predefinite II

```
1 class Eq a => Ord a where
2     (<), (<=), (>=), (>) :: a -> a -> Bool
3     ...
```

- ▶ Contexte utilizabile și la **definirea** unei clase
- ▶ **Moștenirea** claselor, cu preluarea operațiilor din clasa moștenită
- ▶ **Necesitatea** aderenței la clasa `Eq` în momentul instanțierii clasei `Ord`
- ▶ **Suficiența** supradefinirii lui `(<=)` la instanțiere



Clase Haskell vs. POO

Haskell

- ▶ Mulțimi de **tipuri**

POO

- ▶ Mulțimi de **obiecte**: *tipuri*



Clase Haskell vs. POO

Haskell

- ▶ Mulțimi de **tipuri**
- ▶ **Instanțierea** claselor de către tipuri

POO

- ▶ Mulțimi de **obiecte**: *tipuri*
- ▶ **Implementarea** interfețelor de către clase



Clase Haskell vs. POO

Haskell

- ▶ Mulțimi de **tipuri**
- ▶ **Instanțierea** claselor de către tipuri
- ▶ Implementarea operațiilor **în afara** definiției tipului

POO

- ▶ Mulțimi de **obiecte**: *tipuri*
- ▶ **Implementarea** interfețelor de către clase
- ▶ Implementarea operațiilor **în cadrul** definiției tipului



Clase Haskell vs. POO

Haskell

- ▶ Mulțimi de **tipuri**
- ▶ **Instanțierea** claselor de către tipuri
- ▶ Implementarea operațiilor **în afara** definiției tipului

POO

- ▶ Mulțimi de **obiecte**: *tipuri*
- ▶ **Implementarea** interfețelor de către clase
- ▶ Implementarea operațiilor **în cadrul** definiției tipului

Clase Haskell ~ Interfețe Java



Cuprins

Clase

Aplicație pentru clase



invert |

Fie constructorii de tip:

```
3 data Pair a = P a a
4
5 data NestedList a
6     = Atom a
7     | List [NestedList a]
```

Să se definească operația `invert`, aplicabilă pe obiecte de tipuri diferite, inclusiv `Pair a` și `NestedList a`, comportamentul fiind **specific** fiecărui tip.



invert II

```
5  class Invert a where
6      invert :: a -> a
7      invert = id
8
9  instance Invert (Pair a) where
10     invert (P x y) = P y x
11
12 instance Invert a => Invert (NestedList a) where
13     invert (Atom x) = Atom (invert x)
14     invert (List x) = List $ reverse $ map invert x
15
16 instance Invert a => Invert [a] where
17     invert lst = reverse $ map invert lst
```

Necesitatea **contextului**, în cazul tipurilor [a]
și NestedList a, pentru inversarea elementelor **înselor**



contents |

Să se definească operația `contents`, aplicabilă pe obiecte **structurate**, inclusiv pe cele aparținând tipurilor `Pair a` și `NestedList a`, care întoarce elementele, sub forma unei **liste**.

```
1 class Container a where
2     contents :: a -> [??]
```

- ▶ `a` este tipul unui **container**, ca `NestedList b`
- ▶ Elementele listei întoarse sunt cele din **container**
- ▶ Cum **precizăm** tipul acestora, `b`?



contents II

```
1 class Container a where  
2     contents :: a -> [a]  
3  
4 instance Container [a] where  
5     contents = id
```

- Conform definiției clasei:

```
1 contents :: Container [a] => [a] -> [[a]]
```

- Conform supraîncărcării funcției (`id`):

```
1 contents :: Container [a] => [a] -> [a]
```

- Ecuația $[a] = [[a]]$ nu are soluție — eroare!



contents III

```
1 class Container a where  
2     contents :: a -> [b]  
3  
4 instance Container [a] where  
5     contents = id
```

- Conform definiției clasei:

```
1 contents :: Container [a] => [a] -> [b]
```

- Conform supraîncărcării funcției (`id`):

```
1 contents :: Container [a] => [a] -> [a]
```

- Ecuația $[a] = [b]$ **are** soluție pentru $a = b$
- Dar, $[a] \rightarrow [a]$ **insuficient** de general în raport cu $[a] \rightarrow [b]$ — **eroare!**



contents IV

Soluție: clasa primește **constructorul** de tip,
și nu tipul container propriu-zis

```
5 class Container t where
6     contents :: t a -> [a]
7
8 instance Container Pair where    -- nu (Pair a)!
9     contents (P x y) = [x, y]
10
11 instance Container NestedList where
12     contents (Atom x) = [x]
13     contents (List l) = concatMap contents l
14
15 instance Container [] where
16     contents = id
```



Contexte I

```
6 fun1 :: Eq a => a -> a -> a -> a
7 fun1 x y z = if x == y then x else z
8
9 fun2 :: (Container a, Invert (a b), Eq (a b))
10      => (a b) -> (a b) -> [b]
11 fun2 x y = if (invert x) == (invert y)
12           then contents x
13           else contents y
14
15 fun3 :: Invert a => [a] -> [a] -> [a]
16 fun3 x y = (invert x) ++ (invert y)
17
18 fun4 :: Ord a => a -> a -> a -> a
19 fun4 x y z = if x == y
20             then z
21             else if x > y
22                  then x
23                  else y
```



Contexte II

- ▶ **Simplificarea** contextului lui `fun3`, de la `Invert [a]` la `Invert a`
- ▶ **Simplificarea** contextului lui `fun4`, de la `(Eq a, Ord a)` la `Ord a`, din moment ce clasa `Ord` este **derivată** din clasa `Eq`



Rezumat

- ▶ **Clase** = mulțimi de tipuri care supraîncarcă anumite operații
- ▶ Formă de polimorfism **ad-hoc**: tipuri diferite, comportamente diferite
- ▶ **Instanțierea** unei clase = aderarea unui tip la o clasă
- ▶ **Derivarea** unei clase = impunerea condiției ca un tip să fie deja membru al clasei părinte, în momentul instanțierii clasei copil, și moștenirea operațiilor din clasa părinte
- ▶ **Context** = mulțimea constrângerilor asupra tipurilor din semnatura unei funcții, în termenii aderenței la diverse clase



Partea X

Paradigma funcțională vs. paradigma imperativă



Cuprins

Efecte laterale și transparență referențială

Aspecte comparative

Aplicații ale programării funcționale



Cuprins

Efecte laterale și transparență referențială

Aspecte comparative

Aplicații ale programării funcționale



Efecte laterale (*side effects*)

Definiție

- ▶ În expresia $2 + (i = 3)$, subexpresia $(i = 3)$:

Efecte laterale (*side effects*)

Definiție

- ▶ În expresia $2 + (i = 3)$, subexpresia $(i = 3)$:
 - ▶ produce **valoarea** 3, conducând la rezultatul 5 pentru întreaga expresie



Efecte laterale (*side effects*)

Definiție

- ▶ În expresia $2 + (i = 3)$, subexpresia $(i = 3)$:
 - ▶ produce **valoarea** 3, conducând la rezultatul 5 pentru întreaga expresie
 - ▶ are **efectul lateral** de inițializare a lui i cu 3



Efecte laterale (*side effects*)

Definiție

- ▶ În expresia $2 + (i = 3)$, subexpresia $(i = 3)$:
 - ▶ produce **valoarea** 3, conducând la rezultatul 5 pentru întreaga expresie
 - ▶ are **efectul lateral** de inițializare a lui i cu 3
- ▶ Inerente în situațiile în care programul interacționează cu exteriorul — **I/O!**



Efecte laterale (*side effects*)

Consecințe

- ▶ În expresia $x-- + ++x$, cu $x = 0$:

Efecte laterale (*side effects*)

Consecințe

- ▶ În expresia $x-- + ++x$, cu $x = 0$:
 - ▶ evaluarea stânga-dreapta produce $0 + 0 = 0$



Efecte laterale (*side effects*)

Consecințe

- ▶ În expresia $x-- + ++x$, cu $x = 0$:
 - ▶ evaluarea stânga-dreapta produce $0 + 0 = 0$
 - ▶ evaluarea dreapta-stânga produce $1 + 1 = 2$



Efecte laterale (*side effects*)

Consecințe

- ▶ În expresia $x-- + ++x$, cu $x = 0$:
 - ▶ evaluarea stânga-dreapta produce $0 + 0 = 0$
 - ▶ evaluarea dreapta-stânga produce $1 + 1 = 2$
 - ▶ dacă înlocuim cele două subexpresii cu valorile pe care le reprezintă, obținem
$$x + (x + 1) = 0 + 1 = 1$$



Efecte laterale (*side effects*)

Consecințe

- ▶ În expresia $x-- + ++x$, cu $x = 0$:
 - ▶ evaluarea stânga-dreapta produce $0 + 0 = 0$
 - ▶ evaluarea dreapta-stânga produce $1 + 1 = 2$
 - ▶ dacă înlocuim cele două subexpresii cu valorile pe care le reprezintă, obținem
$$x + (x + 1) = 0 + 1 = 1$$
- ▶ Adunare **necomutativă**?!



Efecte laterale (*side effects*)

Consecințe

- ▶ În expresia $x-- + ++x$, cu $x = 0$:
 - ▶ evaluarea stânga-dreapta produce $0 + 0 = 0$
 - ▶ evaluarea dreapta-stânga produce $1 + 1 = 2$
 - ▶ dacă înlocuim cele două subexpresii cu valorile pe care le reprezintă, obținem
$$x + (x + 1) = 0 + 1 = 1$$
- ▶ Adunare **necomutativă**?!
- ▶ Importanța **ordinii de evaluare**!



Efecte laterale (*side effects*)

Consecințe

- ▶ În expresia $x-- + ++x$, cu $x = 0$:
 - ▶ evaluarea stânga-dreapta produce $0 + 0 = 0$
 - ▶ evaluarea dreapta-stânga produce $1 + 1 = 2$
 - ▶ dacă înlocuim cele două subexpresii cu valorile pe care le reprezintă, obținem
$$x + (x + 1) = 0 + 1 = 1$$
- ▶ Adunare **necomutativă**?!
- ▶ Importanța **ordinii de evaluare**!
- ▶ Dependențe **implicite**, dificil de desprins și posibile generatoare de bug-uri



Transparență referențială

- ▶ Zeus la greci $\equiv$ Jupiter la romani (Wooldridge și Jennings, 1995)



Transparență referențială

- ▶ Zeus la greci $\equiv$ Jupiter la romani (Wooldridge și Jennings, 1995)
 1. Cazul 1:

Transparență referențială

- ▶ Zeus la greci $\equiv$ Jupiter la romani (Wooldridge și Jennings, 1995)
 - 1. Cazul 1:
 - ▶ “**Zeus** este fiul lui Cronos”



Transparență referențială

- ▶ Zeus la greci $\equiv$ Jupiter la romani (Wooldridge și Jennings, 1995)

1. Cazul 1:

- ▶ “**Zeus** este fiul lui Cronos”
- ▶ “**Jupiter** este fiul lui Cronos”



Transparență referențială

- ▶ Zeus la greci $\equiv$ Jupiter la romani (Wooldridge și Jennings, 1995)

1. Cazul 1:

- ▶ “**Zeus** este fiul lui Cronos”
- ▶ “**Jupiter** este fiul lui Cronos”



Transparență referențială

- ▶ Zeus la greci $\equiv$ Jupiter la romani (Wooldridge și Jennings, 1995)

1. Cazul 1:

- ▶ “**Zeus** este fiul lui Cronos”
- ▶ “**Jupiter** este fiul lui Cronos”
- ▶ aceeași semnificație



Transparență referențială

- ▶ Zeus la greci $\equiv$ Jupiter la romani (Wooldridge și Jennings, 1995)

1. Cazul 1:

- ▶ “**Zeus** este fiul lui Cronos”
- ▶ “**Jupiter** este fiul lui Cronos”
- ▶ aceeași semnificație

2. Cazul 2:



Transparență referențială

- ▶ Zeus la greci $\equiv$ Jupiter la romani (Wooldridge și Jennings, 1995)

1. Cazul 1:

- ▶ “**Zeus** este fiul lui Cronos”
- ▶ “**Jupiter** este fiul lui Cronos”
- ▶ **aceeași** semnificație

2. Cazul 2:

- ▶ “Ionel știe că **Zeus** este fiul lui Cronos”



Transparență referențială

- ▶ Zeus la greci $\equiv$ Jupiter la romani (Wooldridge și Jennings, 1995)

1. Cazul 1:

- ▶ “**Zeus** este fiul lui Cronos”
- ▶ “**Jupiter** este fiul lui Cronos”
- ▶ aceeași semnificație

2. Cazul 2:

- ▶ “Ionel știe că **Zeus** este fiul lui Cronos”
- ▶ “Ionel știe că **Jupiter** este fiul lui Cronos”



Transparență referențială

- ▶ Zeus la greci $\equiv$ Jupiter la romani (Wooldridge și Jennings, 1995)

1. Cazul 1:

- ▶ “**Zeus** este fiul lui Cronos”
- ▶ “**Jupiter** este fiul lui Cronos”
- ▶ aceeași semnificație

2. Cazul 2:

- ▶ “Ionel știe că **Zeus** este fiul lui Cronos”
- ▶ “Ionel știe că **Jupiter** este fiul lui Cronos”



Transparență referențială

- ▶ Zeus la greci $\equiv$ Jupiter la romani (Wooldridge și Jennings, 1995)

1. Cazul 1:

- ▶ “**Zeus** este fiul lui Cronos”
- ▶ “**Jupiter** este fiul lui Cronos”
- ▶ aceeași semnificație

2. Cazul 2:

- ▶ “Ionel știe că **Zeus** este fiul lui Cronos”
- ▶ “Ionel știe că **Jupiter** este fiul lui Cronos”
- ▶ altă semnificație

Transparență referențială

- ▶ Zeus la greci $\equiv$ Jupiter la romani (Wooldridge și Jennings, 1995)

1. Cazul 1:

- ▶ “**Zeus** este fiul lui Cronos”
- ▶ “**Jupiter** este fiul lui Cronos”
- ▶ aceeași semnificație

2. Cazul 2:

- ▶ “Ionel știe că **Zeus** este fiul lui Cronos”
- ▶ “Ionel știe că **Jupiter** este fiul lui Cronos”
- ▶ altă semnificație

- ▶ *Transparență referențială* = **independența** înțelesului unei propoziții în raport cu modul de desemnare a obiectelor — cazul 1.



Expresii transparente referențial

*One of the most useful properties of expressions is [...] **referential transparency**. In essence this means that if we wish to find the value of an expression which contains a sub-expression, the only thing we need to know about the sub-expression is its **value**. Any other features of the sub-expression, such as its internal structure, the number and nature of its components, the order in which they are evaluated or the colour of the ink in which they are written, are **irrelevant** to the value of the main expression.*

Christopher Strachey,
Fundamental Concepts in Programming Languages



Expresii transparente referențial

*The only thing that matters about an expression is its value, and any subexpression can be replaced by **any other equal in value**. Moreover, the value of an expression is, within certain limits, the **same** whenever it occurs.*

Joseph Stoy,
*Denotational semantics: the Scott-Strachey
approach to programming language theory*



Expresii transparente referențial

- ▶ Expresii (ne)transparente referențial:



Expresii transparente referențial

- ▶ Expresii (ne)transparente referențial:

- ▶ `x-- + ++x`



Expresii transparente referențial

- ▶ Expresii (ne)transparente referențial:
 - ▶ $x-- + ++x$: **nu**, valoarea depinde de ordinea de evaluare



Expresii transparente referențial

- ▶ Expresii (ne)transparente referențial:
 - ▶ `x-- + ++x` : **nu**, valoarea depinde de ordinea de evaluare
 - ▶ `x = x + 1`



Expresii transparente referențial

- ▶ Expresii (ne)transparente referențial:
 - ▶ $x-- + ++x$: **nu**, valoarea depinde de ordinea de evaluare
 - ▶ $x = x + 1$: **nu**, două evaluări consecutive vor produce rezultate diferite



Expresii transparente referențial

- ▶ Expresii (ne)transparente referențial:
 - ▶ $x-- + ++x$: **nu**, valoarea depinde de ordinea de evaluare
 - ▶ $x = x + 1$: **nu**, două evaluări consecutive vor produce rezultate diferite
 - ▶ x



Expresii transparente referențial

- ▶ Expresii (ne)transparente referențial:
 - ▶ $x-- + ++x$: **nu**, valoarea depinde de ordinea de evaluare
 - ▶ $x = x + 1$: **nu**, două evaluări consecutive vor produce rezultate diferite
 - ▶ x : da, presupunând că x nu este modificată în altă parte



Expresii transparente referențial

- ▶ Expresii (ne)transparente referențial:
 - ▶ $x-- + ++x$: **nu**, valoarea depinde de ordinea de evaluare
 - ▶ $x = x + 1$: **nu**, două evaluări consecutive vor produce rezultate diferite
 - ▶ x : da, presupunând că x nu este modificată în altă parte
- ▶ **Efecte laterale** $\Rightarrow$ opacitate referențială!



Funcții transparente referențial

Funcție transparentă referențial:
rezultatul întors depinde **exclusiv** de parametri

```
1  int transparent(int x) {    5  int g = 0;
2      return x + 1;           6
3  }                          7  int opaque(int x) {
                               8      return x + ++g;
                               9  }
                               10
                               11  // opaque(3) != opaque(3)
```



Funcții transparente referențial

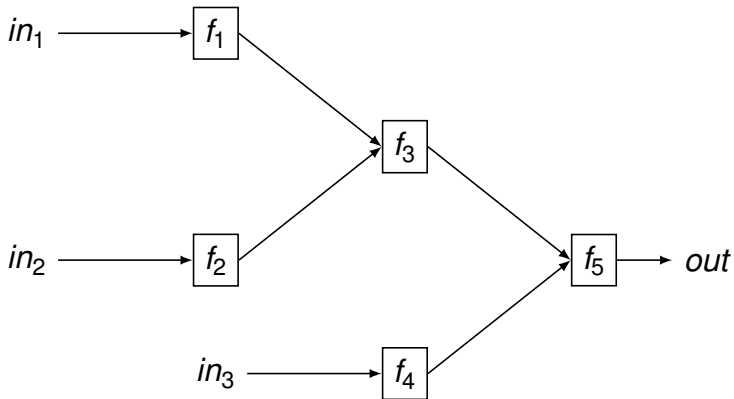
Funcție transparentă referențial:
rezultatul întors depinde **exclusiv** de parametri

```
1  int transparent(int x) {    5  int g = 0;
2      return x + 1;          6
3  }                          7  int opaque(int x) {
                              8      return x + ++g;
                              9  }
                              10
                              11  // opaque(3) != opaque(3)
```

- ▶ Funcții transparente: `log`, `sin` etc.
- ▶ Funcții opace: `time`, `read` etc.



Înlănțuirea funcțiilor



Calcul fără stare

Dependența ieșirii de **intrare**, nu și de timp



t_0

Calcul fără stare

Dependența ieșirii de **intrare**, nu și de timp



t_1

Calcul fără stare

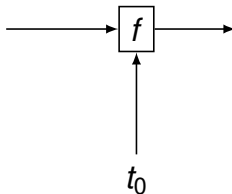
Dependența ieșirii de **intrare**, nu și de timp



t_2

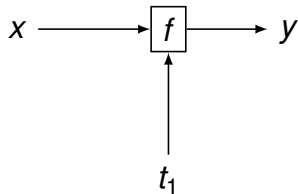
Calcul cu stare

Dependența ieșirii de **intrare**, și de **timp**



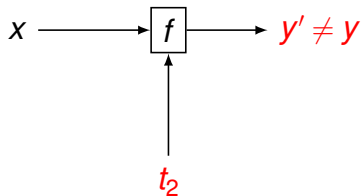
Calcul cu stare

Dependența ieșirii de **intrare**, și de **timp**

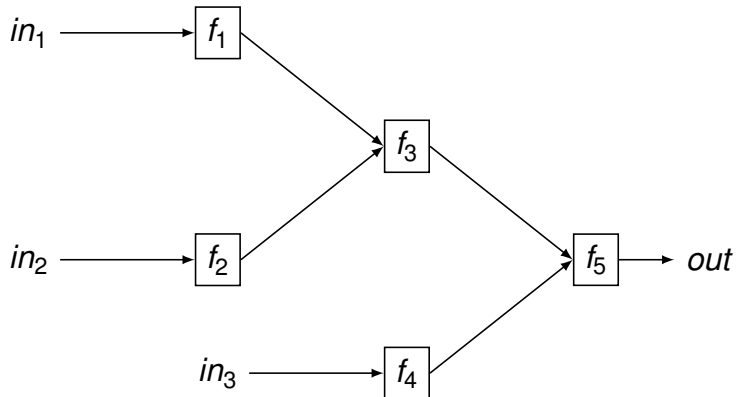


Calcul cu stare

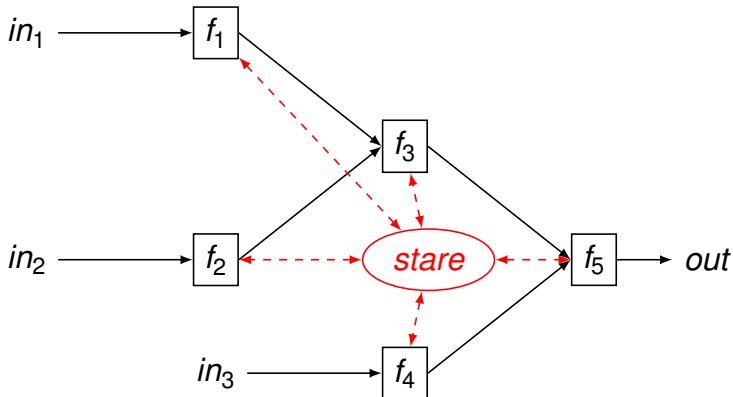
Dependența ieșirii de **intrare**, și de **timp**



Calcul cu stare



Calcul cu stare



Stare = mulțimea valorilor variabilelor, la un anumit moment, ce pot influența rezultatul evaluării aceleiași expresii.

Avantajele transparenței referențiale

- ▶ Lizibilitatea codului



Avantajele transparenței referențiale

- ▶ **Lizibilitatea** codului
- ▶ Demonstrarea formală a **corectitudinii** programului



Avantajele transparenței referențiale

- ▶ **Lizibilitatea** codului
- ▶ Demonstrarea formală a **corectitudinii** programului
- ▶ **Optimizare** prin reordonarea instrucțiunilor de către compilator, și prin *caching*



Avantajele transparenței referențiale

- ▶ **Lizibilitatea** codului
- ▶ Demonstrarea formală a **corectitudinii** programului
- ▶ **Optimizare** prin reordonarea instrucțiunilor de către compilator, și prin *caching*
- ▶ **Paralelizare** masivă, în urma eliminării modificărilor concurente



Avantajele transparenței referențiale

- ▶ **Lizibilitatea** codului
- ▶ Demonstrarea formală a **corectitudinii** programului
- ▶ **Optimizare** prin reordonarea instrucțiunilor de către compilator, și prin *caching*
- ▶ **Paralelizare** masivă, în urma eliminării modificărilor concurente
- ▶ Evaluare **leneșă**, imposibilă în absența unei garanții despre menținerea valorii unei expresii, la momente diferite!



Cuprins

Efecte laterale și transparență referențială

Aspecte comparative

Aplicații ale programării funcționale



Explicitarea sensului programelor

```
1: procedure MINLIST( $L, n$ )
2:    $min \leftarrow L[1]$ 
3:    $i \leftarrow 2$ 
4:   while  $i \leq n$  do
5:     if  $L[i] < min$  then
6:        $min \leftarrow L[i]$ 
7:     end if
8:      $i \leftarrow i + 1$ 
9:   end while
10:  return  $min$ 
11: end procedure
```

```
1 minList [h]      = h
2 minList (h : t) = min h $ minList t
```



Verificarea programelor

Funcțional

- ▶ Definiția unei funcții = **proprietate** pe care o îndeplinește
- ▶ Aplicabilitatea **directă** a metodelor, e.g inducție structurală

Imperativ

- ▶ Necesitatea **adnotării** programelor cu descriptori de stare
- ▶ Necesitatea aplicării de metode **indirecte**, bazate pe adnotări



Funcții și variabile

Funcțional

- ▶ Funcții cu **aceleași** valori pentru aceiași parametri
- ▶ Variabile **nemodificabile**

Imperativ

- ▶ Funcții cu valori **diferite** pentru aceiași parametri
- ▶ Variabile **modificabile**



Evaluare leneșă

- ▶ Posibilă doar în **absența** efectelor laterale
- ▶ **Modularitate** eficientă,
separație producător-consumator
- ▶ **Fluxuri**



Problema expresivității

	Extinderea tipurilor	Extinderea operațiilor
Funcțional	Dificilă	Ușoară
OO	Ușoară	Dificilă

Alte aspecte

- ▶ Funcționale ca structuri de control
- ▶ Tipuri algebrice
- ▶ Polimorfism



Cuprins

Efecte laterale și transparență referențială

Aspecte comparative

Aplicații ale programării funcționale



Aplicații ale programării funcționale I

- ▶ *PureScript*, translator Haskell → JavaScript:
(<http://www.purescript.org/>)
- ▶ *Yesod Web Framework for Haskell*
(<http://www.yesodweb.com/>)
- ▶ Back-end Haskell pentru Android
(<https://wiki.haskell.org/Android>)
- ▶ *Yampa*, EDSL în Haskell
pentru *Functional Reactive Programming* (FRP)
(<https://wiki.haskell.org/Yampa>)



Aplicații ale programării funcționale II

- ▶ Programare paralelă

(<http://chimera.labs.oreilly.com/books/12300000000929>)

- ▶ Utilizare Haskell la Google și Facebook:

(<https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/>)

- ▶ Construcții lambda și funcționale, introduse în C++, Java 8, Swift

(<https://developer.apple.com/swift/>)



Bibliografie

Thompson, S. (2011). *Haskell: The Craft of Functional Programming*. Ediția a treia. Addison-Wesley.

Wooldridge, M. și Jennings, N. R. (1995). Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10:115–152.

Partea XI

Limbajul Prolog



Cuprins

Axiome și reguli

Procesul de demonstrare

Controlul execuției

Caracteristici



Cuprins

Axiome și reguli

Procesul de demonstrare

Controlul execuției

Caracteristici



Un prim exemplu

```
1  % constante -> litera mica
2  parent(andrei, bogdan).
3  parent(andrei, bianca).
4  parent(bogdan, cristi).
5
6  % variabile -> litera mare
7  grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

- ▶ $\text{parent}(\text{andrei}, \text{bogdan})$
- ▶ $\text{parent}(\text{andrei}, \text{bianca})$
- ▶ $\text{parent}(\text{bogdan}, \text{cristi})$
- ▶ $\forall x. \forall y. \forall z.$
 $(\text{parent}(x, z) \wedge \text{parent}(z, y) \Rightarrow \text{grandparent}(x, y))$



Un prim exemplu

```
1  % constante -> litera mica
2  parent (andrei, bogdan) .
3  parent (andrei, bianca) .
4  parent (bogdan, cristi) .
5
6  % variabile -> litera mare
7  grandparent (X, Y) :- parent (X, Z), parent (Z, Y) .
```

- ▶ $true \Rightarrow \text{parent}(\text{andrei}, \text{bogdan})$
- ▶ $true \Rightarrow \text{parent}(\text{andrei}, \text{bianca})$
- ▶ $true \Rightarrow \text{parent}(\text{bogdan}, \text{cristi})$
- ▶ $\forall x. \forall y. \forall z.$
 $(\text{parent}(x, z) \wedge \text{parent}(z, y) \Rightarrow \text{grandparent}(x, y))$



Interogări

```
1  ?- parent (andrei, bogdan) .
2  true .
3
4  ?- parent (andrei, cristi) .
5  false.
6
7  ?- parent (andrei, X) .
8  X = bogdan ;
9  X = bianca.
10
11 ?- grandparent (X, Y) .
12 X = andrei,
13 Y = cristi ;
14 false.
```

- ▶ “.” → oprire după **primul** răspuns
- ▶ “;” → solicitarea **următorului** răspuns



Concatenarea a două liste

```
1  % append(L1, L2, Res)
2  append([], L, L).
3  append([H|T], L, [H|Res]) :- append(T, L, Res).
```

Calcul

```
1  ?- append([1], [2], Res).
2  Res = [1, 2].
```

Generare

```
1  ?- append(L1, L2, [1, 2]).
2  L1 = [],
3  L2 = [1, 2] ;
4  L1 = [1],
5  L2 = [2] ;
6  L1 = [1, 2],
7  L2 = [] ;
8  false.
```



Concatenarea a două liste

```
1  % append(L1, L2, Res)
2  append([], L, L).
3  append([H|T], L, [H|Res]) :- append(T, L, Res).
```

Calcul

```
1  ?- append([1], [2], Res).
2  Res = [1, 2].
```

Generare

```
1  ?- append(L1, L2, [1, 2]).
2  L1 = [],
3  L2 = [1, 2] ;
4  L1 = [1],
5  L2 = [2] ;
6  L1 = [1, 2],
7  L2 = [] ;
8  false.
```

Estomparea granițelor dintre “intrare” și “ieșire”



Cuprins

Axiome și reguli

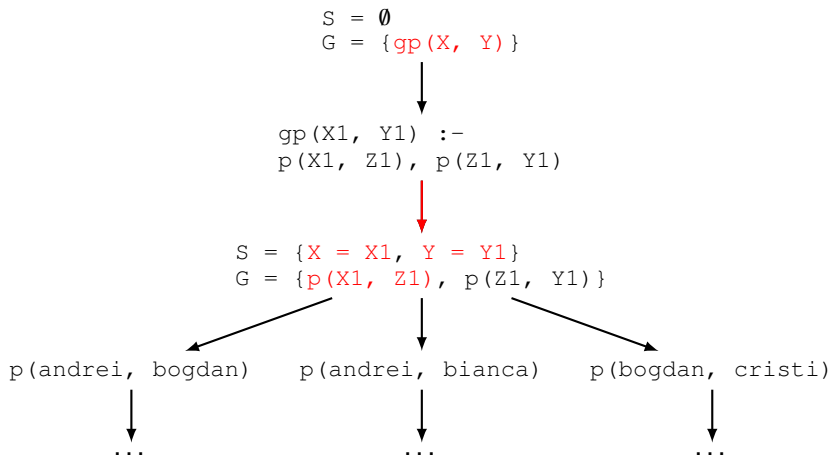
Procesul de demonstrare

Controlul execuției

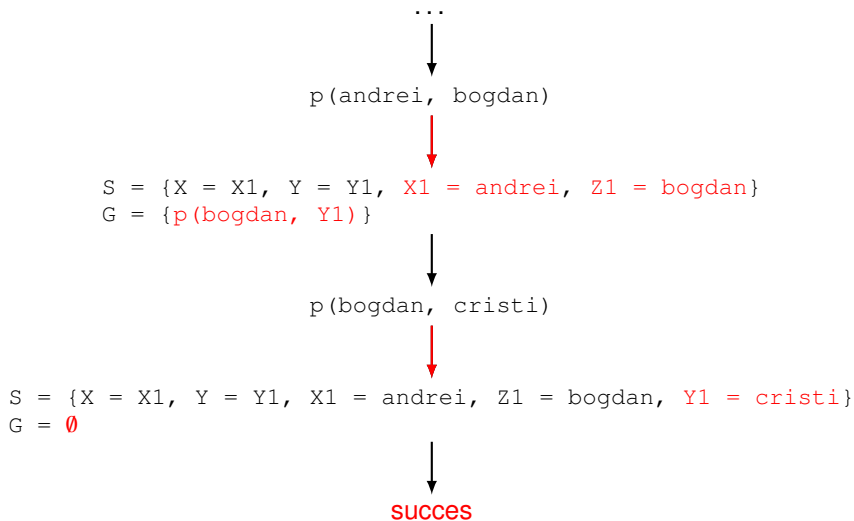
Caracteristici



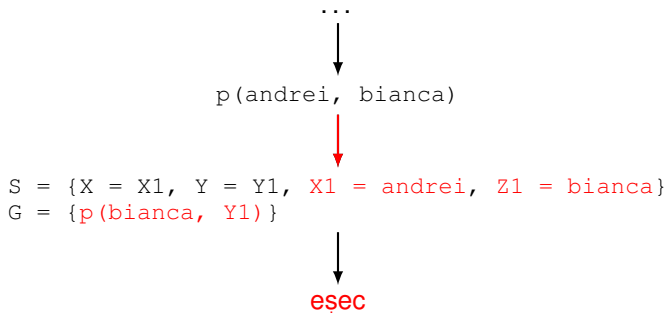
Exemplul genealogic I



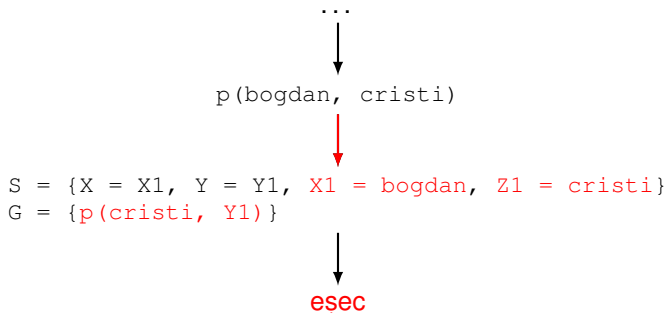
Exemplul genealogic II



Exemplul genealogic III



Exemplul genealogic IV



Pași în demonstrare I

1. Inițializarea **stivei de scopuri** cu scopul solicitat
2. Inițializarea **substituției** utilizate pe parcursul unificării cu mulțimea vidă
3. Extragerea scopului din **vârful** stivei și determinarea **primei** clauze din program cu a cărei concluzie **unifică**
4. Îmbogățirea corespunzătoare a **substituției** și adăugarea **premiselor** clauzei în stivă, în ordinea din program
5. Salt la pasul 3



Pași în demonstrare II

6. În cazul **imposibilității** satisfacerii scopului din vârful stivei, **revenirea** la scopul anterior (*backtracking*), și încercarea altei modalități de satisfacere
7. **Succes** la **golirea** stivei de scopuri
8. **Eșec** la imposibilitatea satisfacerii **ultimului** scop din stivă



Observații

- ▶ Ordinea **clauzelor** în program
- ▶ Ordinea **premiselor** în cadrul regulilor
- ▶ Recomandare: premisele **mai ușor** de satisfăcut, primele — exemplu: axiome



Strategii de control

Forward chaining (data-driven)

- ▶ Premise → scop
- ▶ Derivarea **tuturor** concluziilor posibile
- ▶ **Oprire** la obținerea scopului (scopurilor)

Backward chaining (goal-driven)

- ▶ Scop → premise
- ▶ Utilizarea **exclusivă** a regulilor care pot contribui efectiv la satisfacerea scopului
- ▶ Satisfacerea **premiselor** acestor reguli ș.a.m.d.



Cuprins

Axiome și reguli

Procesul de demonstrare

Controlul execuției

Caracteristici



Minimul a două numere I

```
1  min(X, Y, M) :- X =< Y, M is X.
2  min(X, Y, M) :- X > Y,  M is Y.
3
4  min2(X, Y, M) :- X =< Y, M = X.
5  min2(X, Y, M) :- X > Y,  M = Y.
6
7  % Echivalent cu min2.
8  min3(X, Y, X) :- X =< Y.
9  min3(X, Y, Y) :- X > Y.
```



Minimul a două numere II

```
1  ?- min(1+2, 3+4, M) .
2  M = 3 ;
3  false.
4
5  ?- min(3+4, 1+2, M) .
6  M = 3.
7
8  ?- min2(1+2, 3+4, M) .
9  M = 1+2 ;
10 false.
11
12 ?- min2(3+4, 1+2, M) .
13 M = 1+2.
```



Minimul a două numere III

Condiții mutual exclusive: $X \leq Y$ și $X > Y$ — cum putem **elimina** redundanța?

```
12 min4(X, Y, X) :- X <= Y.
```

```
13 min4(X, Y, Y) .
```

```
1  ?- min4(1+2, 3+4, M) .
```

```
2  M = 1+2 ;
```

```
3  M = 3+4 .
```

Greșit!



Minimul a două numere IV

Soluție: **oprirea** recursivității după prima satisfacere a scopului

```
15 min5(X, Y, X) :- X =< Y, !.
```

```
16 min5(X, Y, Y).
```

```
1  ?- min5(1+2, 3+4, M).
```

```
2  M = 1+2.
```



Operatorul *cut* I

- ▶ La **prima** întâlnire: **satisfacere**
- ▶ La **a doua** întâlnire, în momentul revenirii (*backtracking*): **eșec**, cu inhibarea **tuturor** căilor ulterioare de satisfacere a scopului care a unificat cu concluzia regulii curente
- ▶ Utilitate în **eficientizarea** programelor



Operatorul *cut* II

```
1  girl(mary).
2  girl(ann).
3
4  boy(john).
5  boy(bill).
6
7  pair(X, Y) :- girl(X), boy(Y).
8  pair(bella, harry).
9
10 pair2(X, Y) :- girl(X), !, boy(Y).
11 pair2(bella, harry).
```

Backtracking doar la **dreapta** operatorului



Operatorul *cut* III

```
1  ?- pair(X, Y) .  
2  X = mary,  
3  Y = john ;  
4  X = mary,  
5  Y = bill ;  
6  X = ann,  
7  Y = john ;  
8  X = ann,  
9  Y = bill ;  
10 X = bella,  
11 Y = harry.
```

```
1  ?- pair2(X, Y) .  
2  X = mary,  
3  Y = john ;  
4  X = mary,  
5  Y = bill.
```



Cuprins

Axiome și reguli

Procesul de demonstrare

Controlul execuției

Caracteristici



Programare logică

- ▶ Reprezentare **simbolică**
- ▶ Stil **declarativ**
- ▶ **Separarea** datelor de procesul de inferență, încorporat în limbaj
- ▶ **Uniformitatea** reprezentării axiomelor și a regulilor de derivare
- ▶ Reprezentarea **modularizată** a cunoștințelor
- ▶ Posibilitatea modificării **dinamice** a programelor, prin adăugarea și retragerea axiomelor și a regulilor



Prolog I

- ▶ Bazat pe logica cu predicate de ordin 1, **restricționată**
- ▶ “Calculul”: satisfacerea de scopuri, prin **reducere la absurd**
- ▶ Regula de inferență: **rezoluția**
- ▶ Strategia de control, în evoluția demonstrațiilor:
 - ▶ **backward chaining**: de la scop către axiome
 - ▶ parcurgere în **adâncime**, în arborele de derivare
- ▶ Parcurgerea în **adâncime**:
 - ▶ pericolul coborârii pe o cale infinită, ce nu conține soluția — strategie **incompletă**
 - ▶ **eficiență** sporită în utilizarea **spațiului**



Prolog II

- ▶ Exclusiv clauze **Horn**:

$$A_1 \wedge \dots \wedge A_n \Rightarrow A \quad (\text{Regulă})$$

$$true \Rightarrow B \quad (\text{Axiomă})$$

- ▶ Absența **negațiilor** explicite — desprinderea falsității pe baza imposibilității de a demonstra
- ▶ Ipoteza lumii **închise** (*closed world assumption*): ceea ce nu poate fi demonstrat este **fals**
- ▶ Prin opoziție, ipoteza lumii **deschise** (*open world assumption*): nu se poate afirma **nimic** despre ceea ce nu poate fi demonstrat



Negația ca eșec

```
1 nott(P) :- P, !, fail.  
2 nott(P).
```

- ▶ $P \rightarrow \text{atom}$ — exemplu: `boy(john)`
- ▶ P **satisfiabil**:
 - ▶ eșecul **primei** reguli, din cauza lui `fail`
 - ▶ abandonarea celei **de-a doua** reguli, din cauza lui `!`
 - ▶ rezultat: `nott(P)` **nesatisfiabil**
- ▶ P **nesatisfiabil**:
 - ▶ eșecul **primei** reguli
 - ▶ succesul celei **de-a doua** reguli
 - ▶ rezultat: `nott(P)` **satisfiabil**



Rezumat

- ▶ Date: clauze **Horn**
- ▶ Regula de inferență: **rezoluție**
- ▶ Strategia de căutare: *backward chaining*,
dinspre concluzie spre ipoteze
- ▶ Posibilități **generative**, pe baza unui anumit stil
de scriere a regulilor



Partea XII

Logica propozițională și logica cu predicate de ordinul I



Cuprins

Introducere

Logica propozițională

- Sintaxă și semantică

- Satisfiabilitate și validitate

- Derivabilitate

- Inferență și demonstrație

- Rezoluție

Logica cu predicate de ordinul I

- Sintaxă și semantică

- Forma clauzală

- Unificare



Cuprins

Introducere

Logica propozițională

Sintaxă și semantică

Satisfiabilitate și validitate

Derivabilitate

Inferență și demonstrație

Rezoluție

Logica cu predicate de ordinul I

Sintaxă și semantică

Forma clauzală

Unificare



Logică

- ▶ Scop: reducerea efectuării de raționamente la **calcul**

(Harrison, 2009)



Logică

- ▶ Scop: reducerea efectuării de raționamente la **calcul**
- ▶ Problemele de **decidabilitate** din logică: stimulent pentru dezvoltarea modelelor de calculabilitate

(Harrison, 2009)



Logică

- ▶ Scop: reducerea efectuării de raționamente la **calcul**
- ▶ Problemele de **decidabilitate** din logică: stimulent pentru dezvoltarea modelelor de calculabilitate
- ▶ Împrumuturi **reciproce** între domeniile logicii și calculabilității:

(Harrison, 2009)



Logică

- ▶ Scop: reducerea efectuării de raționamente la **calcul**
- ▶ Problemele de **decidabilitate** din logică: stimulent pentru dezvoltarea modelelor de calculabilitate
- ▶ Împrumuturi **reciproce** între domeniile logicii și calculabilității:
 - ▶ proiectarea și verificarea programelor → logică

(Harrison, 2009)



Logică

- ▶ Scop: reducerea efectuării de raționamente la **calcul**
- ▶ Problemele de **decidabilitate** din logică: stimulent pentru dezvoltarea modelelor de calculabilitate
- ▶ Împrumuturi **reciproce** între domeniile logicii și calculabilității:
 - ▶ proiectarea și verificarea programelor → logică
 - ▶ principiile logice → proiectarea limbajelor de programare

(Harrison, 2009)



Rolurile logicii

- ▶ **Descrierea** proprietăților obiectelor, într-o manieră neambiguă, prin intermediul unui **limbaj**, cu următoarele componente:



Rolurile logicii

- ▶ **Descrierea** proprietăților obiectelor, într-o manieră neambiguă, prin intermediul unui **limbaj**, cu următoarele componente:
 - ▶ **sintaxă**: modalitatea de construcție a expresiilor



Rolurile logicii

- ▶ **Descrierea** proprietăților obiectelor, într-o manieră neambiguă, prin intermediul unui **limbaj**, cu următoarele componente:
 - ▶ **sintaxă**: modalitatea de construcție a expresiilor
 - ▶ **semantică**: semnificația expresiilor construite



Rolurile logicii

- ▶ **Descrierea** proprietăților obiectelor, într-o manieră neambiguă, prin intermediul unui **limbaj**, cu următoarele componente:
 - ▶ **sintaxă**: modalitatea de construcție a expresiilor
 - ▶ **semantică**: semnificația expresiilor construite
- ▶ **Deducerea** de noi proprietăți, pe baza celor existente



Cuprins

Introducere

Logica propozițională

- Sintaxă și semantică

- Satisfiabilitate și validitate

- Derivabilitate

- Inferență și demonstrație

- Rezoluție

Logica cu predicate de ordinul I

- Sintaxă și semantică

- Forma clauzală

- Unificare



Logica propozițională

- Expresia din limbaj: **propoziția**, corespunzătoare unei afirmații, ce poate fi adevărată sau falsă

(Genesereth, 2010)



Logica propozițională

- ▶ Expresia din limbaj: **propoziția**, corespunzătoare unei afirmații, ce poate fi adevărată sau falsă
- ▶ Exemplu: “Telefonul sună și câinele latră.”

(Genesereth, 2010)



Logica propozițională

- ▶ Expresia din limbaj: **propoziția**, corespunzătoare unei afirmații, ce poate fi adevărată sau falsă
- ▶ Exemplu: “Telefonul sună și câinele latră.”
- ▶ **Accepții** asupra unei propoziții:

(Genesereth, 2010)



Logica propozițională

- ▶ Expresia din limbaj: **propoziția**, corespunzătoare unei afirmații, ce poate fi adevărată sau falsă
- ▶ Exemplu: “Telefonul sună și câinele latră.”
- ▶ **Acceptii** asupra unei propoziții:
 - ▶ secvența de **simboluri** utilizate sau

(Genesereth, 2010)



Logica propozițională

- ▶ Expresia din limbaj: **propoziția**, corespunzătoare unei afirmații, ce poate fi adevărată sau falsă
- ▶ Exemplu: “Telefonul sună și câinele latră.”
- ▶ **Acceptii** asupra unei propoziții:
 - ▶ secvența de **simboluri** utilizate sau
 - ▶ **înțelesul** propriu-zis al acesteia, într-o **interpretare**

(Genesereth, 2010)



Logica propozițională

- ▶ Expresia din limbaj: **propoziția**, corespunzătoare unei afirmații, ce poate fi adevărată sau falsă
- ▶ Exemplu: “Telefonul sună și câinele latră.”
- ▶ **Accepții** asupra unei propoziții:
 - ▶ secvența de **simboluri** utilizate sau
 - ▶ **înțelesul** propriu-zis al acesteia, într-o **interpretare**
- ▶ **Valoarea de adevăr** a unei propoziții — determinată de valorile de adevăr ale propozițiilor **constituente**

(Genesereth, 2010)



Cuprins

Introducere

Logica propozițională

Sintaxă și semantică

Satisfiabilitate și validitate

Derivabilitate

Inferență și demonstrație

Rezoluție

Logica cu predicate de ordinul I

Sintaxă și semantică

Forma clauzală

Unificare



Sintaxă

- ▶ 2 categorii de propoziții
 - ▶ simple: fapte **atomice**:
“Telefonul sună.”, “Câinele latră.”
 - ▶ compuse: **relații** între propoziții mai simple:
“Telefonul sună și câinele latră.”



Sintaxă

- ▶ 2 categorii de propoziții
 - ▶ simple: fapte **atomice**:
“Telefonul sună.”, “Câinele latră.”
 - ▶ compuse: **relații** între propoziții mai simple:
“Telefonul sună și câinele latră.”
- ▶ Propoziții simple: $p, q, r, \dots$
- ▶ Negatii: $\neg \alpha$
- ▶ Conjuncții: $(\alpha \wedge \beta)$
- ▶ Disjuncții: $(\alpha \vee \beta)$
- ▶ Implicații: $(\alpha \Rightarrow \beta)$
- ▶ Echivalențe: $(\alpha \Leftrightarrow \beta)$



Semantică I

- ▶ Atribuirea de **valori de adevăr** propozițiilor
- ▶ Accent pe **relațiile** dintre propozițiile compuse și cele constituente
- ▶ Pentru explicitarea legăturilor, utilizarea conceptului de *interpretare*



Semantică II

- *Interpretare* = mulțime de **asocieri** între fiecare propoziție **simplă** din limbaj și o valoare de adevăr

- Exemplu:

Interpretarea I :

- $p^I = false$
- $q^I = true$
- $r^I = false$

Interpretarea J :

- $p^J = true$
- $q^J = true$
- $r^J = true$

- Sub o interpretare fixată, **dependența** valorii de adevăr a unei propoziții compuse de valorile de adevăr ale celor constituente



Semantică III

- Negatie:

$$(\neg \alpha)' = \begin{cases} true & \text{dacă } \alpha' = false \\ false & \text{altfel} \end{cases}$$

- Conjuncție:

$$(\alpha \wedge \beta)' = \begin{cases} true & \text{dacă } \alpha' = true \text{ și } \beta' = true \\ false & \text{altfel} \end{cases}$$

- Disjuncție:

$$(\alpha \vee \beta)' = \begin{cases} false & \text{dacă } \alpha' = false \text{ și } \beta' = false \\ true & \text{altfel} \end{cases}$$



Semantică IV

► Implicație:

$$(\alpha \Rightarrow \beta)' = \begin{cases} false & \text{dacă } \alpha' = true \text{ și } \beta' = false \\ true & \text{altfel} \end{cases}$$

► Echivalență:

$$(\alpha \Leftrightarrow \beta)' = \begin{cases} true & \text{dacă } \alpha' = \beta' \\ false & \text{altfel} \end{cases}$$



Evaluare

- ▶ *Evaluare* = determinarea **valorii de adevăr** a unei propoziții, sub o interpretare, prin aplicarea regulilor semantice anterioare
- ▶ Exemplu:

Interpretarea I :

- ▶ $p^I = false$
- ▶ $q^I = true$
- ▶ $r^I = false$

Propoziția: $\phi = (p \wedge q) \vee (q \Rightarrow r)$

$$\begin{aligned}\phi^I &= (false \wedge true) \vee (true \Rightarrow false) \\ &= false \vee false \\ &= false\end{aligned}$$



Cuprins

Introducere

Logica propozițională

Sintaxă și semantică

Satisfiabilitate și validitate

Derivabilitate

Inferență și demonstrație

Rezoluție

Logica cu predicate de ordinul I

Sintaxă și semantică

Forma clauzală

Unificare



Satisfiabilitate

- *Satisfiabilitate* = proprietatea unei propoziții **adevărate** în **cel puțin o** interpretare
- Metoda tabeli de adevăr:

p	q	r	$(p \wedge q) \vee (q \Rightarrow r)$
<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>
<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>



Validitate

- ▶ *Validitate* = proprietatea unei propoziții adevărate în **toate** interpretările (*tautologie*)
- ▶ Exemplu: $p \vee \neg p$
- ▶ Verificabilă prin metoda tabeli de adevăr



Nesatisfiabilitate

- ▶ *Nesatisfiabilitate* = proprietatea unei propoziții **false** în **toate** interpretările (*contradicție*)
- ▶ Exemplu: $p \Leftrightarrow \neg p$
- ▶ Verificabilă prin metoda tabelii de adevăr



Cuprins

Introducere

Logica propozițională

Sintaxă și semantică

Satisfiabilitate și validitate

Derivabilitate

Inferență și demonstrație

Rezoluție

Logica cu predicate de ordinul I

Sintaxă și semantică

Forma clauzală

Unificare



Derivabilitate I

- ▶ *Derivabilitate logică* = proprietatea unei propoziții de a reprezenta **consecința logică** a unei mulțimi de alte propoziții, numite *premise*
- ▶ Mulțimea de propoziții Δ derivă propoziția ϕ , dacă și numai dacă **orice** interpretare care satisface toate propozițiile din Δ satisface și ϕ :

$$\Delta \models \phi$$

- ▶ Exemple:
 - ▶ $\{p\} \models p \vee q$
 - ▶ $\{p, q\} \models p \wedge q$
 - ▶ $\{p\} \not\models p \wedge q$
 - ▶ $\{p, p \Rightarrow q\} \models q$



Derivabilitate II

- ▶ Verificabilă prin metoda tabelii de adevăr: **toate** intrările pentru care **premisele** sunt adevărate trebuie să inducă adevărul **concluziei**
- ▶ Exemplu: demonstrăm că $\{p, p \Rightarrow q\} \models q$.

p	q	$p \Rightarrow q$
<i>true</i>	<u><i>true</i></u>	<i>true</i>
<i>true</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>true</i>

Singura intrare în care ambele premise, p și $p \Rightarrow q$, sunt adevărate, precizează și adevărul concluziei, q .



Formulări echivalente ale derivabilității

- ▶ $\{\phi_1, \dots, \phi_n\} \models \phi$
- ▶ Propoziția $\phi_1 \wedge \dots \wedge \phi_n \Rightarrow \phi$ este



Formulări echivalente ale derivabilității

- ▶ $\{\phi_1, \dots, \phi_n\} \models \phi$
- ▶ Propoziția $\phi_1 \wedge \dots \wedge \phi_n \Rightarrow \phi$ este **validă**
- ▶ Propoziția $\phi_1 \wedge \dots \wedge \phi_n \wedge \neg \phi$ este



Formulări echivalente ale derivabilității

- ▶ $\{\phi_1, \dots, \phi_n\} \models \phi$
- ▶ Propoziția $\phi_1 \wedge \dots \wedge \phi_n \Rightarrow \phi$ este **validă**
- ▶ Propoziția $\phi_1 \wedge \dots \wedge \phi_n \wedge \neg \phi$ este **nesatisfiabilă**



Cuprins

Introducere

Logica propozițională

Sintaxă și semantică

Satisfiabilitate și validitate

Derivabilitate

Inferență și demonstrație

Rezoluție

Logica cu predicate de ordinul I

Sintaxă și semantică

Forma clauzală

Unificare



Motivație

- ▶ Derivabilitate **logică**: proprietate a propozițiilor



Motivație

- ▶ Derivabilitate **logică**: proprietate a propozițiilor
- ▶ Derivare **mecanică** (inferență): demers de **calcul**, în scopul verificării derivabilității logice



Motivație

- ▶ Derivabilitate **logică**: proprietate a propozițiilor
- ▶ Derivare **mecanică** (inferență): demers de **calcul**, în scopul verificării derivabilității logice
- ▶ Creșterea **exponențială** a numărului de interpretări în raport cu numărul de propoziții simple



Motivație

- ▶ Derivabilitate **logică**: proprietate a propozițiilor
- ▶ Derivare **mecanică** (inferență): demers de **calcul**, în scopul verificării derivabilității logice
- ▶ Creșterea **exponențială** a numărului de interpretări în raport cu numărul de propoziții simple
- ▶ De aici, **diminuarea** valorii practice a metodelor **semantice**, precum cea a tabelii de adevăr



Motivație

- ▶ Derivabilitate **logică**: proprietate a propozițiilor
- ▶ Derivare **mecanică** (inferență): demers de **calcul**, în scopul verificării derivabilității logice
- ▶ Creșterea **exponențială** a numărului de interpretări în raport cu numărul de propoziții simple
- ▶ De aici, **diminuarea** valorii practice a metodelor **semantice**, precum cea a tabelii de adevăr
- ▶ Alternativ, metode **sintactice**, care manipulează doar reprezentarea simbolică



Inferență

- ▶ *Inferență* = derivarea **mecanică** a concluziilor unei mulțimi de premise



Inferență

- ▶ *Inferență* = derivarea **mecanică** a concluziilor unei mulțimi de premise
- ▶ *Regulă de inferență* = **procedură** de calcul capabilă să deriveze concluziile unei mulțimi de premise



Inferență

- ▶ *Inferență* = derivarea **mecanică** a concluziilor unei mulțimi de premise
- ▶ *Regulă de inferență* = **procedură** de calcul capabilă să deriveze concluziile unei mulțimi de premise
- ▶ Derivabilitatea mecanică a concluziei ϕ din mulțimea de premise Δ , utilizând regula de inferență *inf*:

$$\Delta \vdash_{inf} \phi$$



Reguli de inferență

- ▶ Șabloane parametrizate de raționament, formate dintr-o mulțime de premise și o mulțime de concluzii

Reguli de inferență

- ▶ Șabloane **parametrizate** de raționament, formate dintr-o mulțime de **premise** și o mulțime de **concluzii**
- ▶ *Modus Ponens* (MP):

$$\frac{\alpha \Rightarrow \beta \quad \alpha}{\beta}$$



Reguli de inferență

- ▶ Șabloane **parametrizate** de raționament, formate dintr-o mulțime de **premise** și o mulțime de **concluzii**
- ▶ *Modus Ponens* (MP):

$$\frac{\alpha \Rightarrow \beta \quad \alpha}{\beta}$$

- ▶ *Modus Tollens*:

$$\frac{\alpha \Rightarrow \beta \quad \neg \beta}{\neg \alpha}$$

Proprietăți ale regulilor de inferență

- *Consistență (soundness)*: regula de inferență determină **doar** propoziții care sunt, într-adevăr, consecințe logice ale premiselor:

$$\Delta \vdash_{inf} \phi \Rightarrow \Delta \models \phi$$



Proprietăți ale regulilor de inferență

- ▶ *Consistență (soundness)*: regula de inferență determină **doar** propoziții care sunt, într-adevăr, consecințe logice ale premiselor:

$$\Delta \vdash_{inf} \phi \Rightarrow \Delta \models \phi$$

- ▶ *Completitudine (completeness)*: regula de inferență determină **toate** consecințele logice ale premiselor:

$$\Delta \models \phi \Rightarrow \Delta \vdash_{inf} \phi$$

Proprietăți ale regulilor de inferență

- ▶ *Consistență (soundness)*: regula de inferență determină **doar** propoziții care sunt, într-adevăr, consecințe logice ale premiselor:

$$\Delta \vdash_{inf} \phi \Rightarrow \Delta \models \phi$$

- ▶ *Completitudine (completeness)*: regula de inferență determină **toate** consecințele logice ale premiselor:

$$\Delta \models \phi \Rightarrow \Delta \vdash_{inf} \phi$$

- ▶ Ideal, **ambele** proprietăți: “nici în plus, nici în minus”

Proprietăți ale regulilor de inferență

- ▶ *Consistență (soundness)*: regula de inferență determină **doar** propoziții care sunt, într-adevăr, consecințe logice ale premiselor:

$$\Delta \vdash_{inf} \phi \Rightarrow \Delta \models \phi$$

- ▶ *Completitudine (completeness)*: regula de inferență determină **toate** consecințele logice ale premiselor:

$$\Delta \models \phi \Rightarrow \Delta \vdash_{inf} \phi$$

- ▶ Ideal, **ambele** proprietăți: “nici în plus, nici în minus”
- ▶ **Incompletitudinea** regulii *Modus Ponens*, din imposibilitatea scrierii oricărei propoziții ca implicație



Axiome

- ▶ Exemplu: verificarea că $\{p \Rightarrow q, q \Rightarrow r\} \models p \Rightarrow r$



Axiome

- ▶ Exemplu: verificarea că $\{p \Rightarrow q, q \Rightarrow r\} \models p \Rightarrow r$
- ▶ Caz în care premisele sunt **insuficiente** pentru aplicarea regulilor de inferență



Axiome

- ▶ Exemplu: verificarea că $\{p \Rightarrow q, q \Rightarrow r\} \models p \Rightarrow r$
- ▶ Caz în care premisele sunt **insuficiente** pentru aplicarea regulilor de inferență
- ▶ Soluția: adăugarea de **axiome**, reguli de inferență fără premise



Axiome

- ▶ Exemplu: verificarea că $\{p \Rightarrow q, q \Rightarrow r\} \models p \Rightarrow r$
- ▶ Caz în care premisele sunt **insuficiente** pentru aplicarea regulilor de inferență
- ▶ Soluția: adăugarea de **axiome**, reguli de inferență fără premise
- ▶ **Introducerea** implicației (II):

$$\alpha \Rightarrow (\beta \Rightarrow \alpha)$$



Axiome

- ▶ Exemplu: verificarea că $\{p \Rightarrow q, q \Rightarrow r\} \models p \Rightarrow r$
- ▶ Caz în care premisele sunt **insuficiente** pentru aplicarea regulilor de inferență
- ▶ Soluția: adăugarea de **axiome**, reguli de inferență fără premise
- ▶ **Introducerea** implicației (II):

$$\alpha \Rightarrow (\beta \Rightarrow \alpha)$$

- ▶ **Distribuirea** implicației (DI):

$$(\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma))$$



Demonstrații I

- ▶ *Demonstrație* = **secvență** de propoziții, finalizată cu o concluzie, și conținând:
 - ▶ **premise**
 - ▶ instanțe ale **axiomelor**
 - ▶ rezultate ale aplicării **regulilor de inferență** asupra elementelor precedente din secvență
- ▶ *Teoremă* = **concluzia** cu care se încheie o demonstrație



Demonstrații II

- ▶ *Procedură de demonstrare* = mecanism de demonstrare, constând din:
 - ▶ o mulțime de **reguli de inferență**
 - ▶ o **strategie de control**, ce dictează ordinea aplicării regulilor



Demonstrații III

Exemplu: demonstrăm că $\{p \Rightarrow q, q \Rightarrow r\} \vdash p \Rightarrow r$.

1	$p \Rightarrow q$	Premisă
2	$q \Rightarrow r$	Premisă
3	$(q \Rightarrow r) \Rightarrow (p \Rightarrow (q \Rightarrow r))$	II
4	$p \Rightarrow (q \Rightarrow r)$	MP 3, 2
5	$(p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r))$	DI
6	$(p \Rightarrow q) \Rightarrow (p \Rightarrow r)$	MP 5, 4
7	$p \Rightarrow r$	MP 6, 1



Demonstrații IV

Rezultat: existența unui sistem de inferență
consistent și complet, bazat pe:

- ▶ **axiomele** de mai devreme, îmbogățite cu altele
- ▶ regula de inferență ***Modus Ponens***

$$\Delta \models \phi \Leftrightarrow \Delta \vdash \phi$$



Cuprins

Introducere

Logica propozițională

Sintaxă și semantică

Satisfiabilitate și validitate

Derivabilitate

Inferență și demonstrație

Rezoluție

Logica cu predicate de ordinul I

Sintaxă și semantică

Forma clauzală

Unificare



Rezoluție

- ▶ Regulă de inferență foarte puternică



Rezoluție

- ▶ Regulă de inferență foarte puternică
- ▶ Baza unui demonstrator de teoreme
consistent și complet



Rezoluție

- ▶ Regulă de inferență foarte puternică
- ▶ Baza unui demonstrator de teoreme
consistent și complet
- ▶ Spațiul de căutare mult mai mic ca în abordarea
standard (v. subsecțiunea anterioară)



Rezoluție

- ▶ Regulă de inferență foarte puternică
- ▶ Baza unui demonstrator de teoreme
consistent și complet
- ▶ Spațiul de căutare mult mai mic ca în abordarea
standard (v. subsecțiunea anterioară)
- ▶ Lucrul cu propoziții în forma clauzală



Forma clauzală I

- ▶ *Literal* = propoziție **simplă** (p) sau **negația** ei ($\neg p$)
- ▶ *Expresie clauzală* = **literal** sau **disjuncție** de literali,
e.g. $p \vee \neg q \vee r \vee p$
- ▶ *Clauză* = **mulțime** de literali dintr-o expresie clauzală,
e.g. $\{p, \neg q, r\}$



Forma clauzală II

- ▶ *Forma clauzală (forma normală conjunctivă, FNC) = reprezentarea unei propoziții sub forma unei mulțimi de clauze*, implicit legate prin conjuncții
- ▶ Exemplu: forma clauzală a propoziției $p \wedge (\neg q \vee r) \wedge (\neg p \vee \neg r)$ este $\{\{p\}, \{\neg q, r\}, \{\neg p, \neg r\}\}$.
- ▶ Posibilitatea *convertirii* oricărei propoziții în această formă, prin algoritmul următor



Transformarea în formă clauzală I

1. Eliminarea **implicațiilor** (I):

$$\alpha \Rightarrow \beta \rightarrow \neg \alpha \vee \beta$$

2. Introducerea **negațiilor** în paranteze (N):

$$\neg(\alpha \wedge \beta) \rightarrow \neg \alpha \vee \neg \beta \text{ etc.}$$

3. **Distribuirea** lui $\vee$ față de $\wedge$ (D):

$$\alpha \vee (\beta \wedge \gamma) \rightarrow (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$$

4. Transformarea expresiilor în **clauze** (C):

$$\phi_1 \vee \dots \vee \phi_n \rightarrow \{\phi_1, \dots, \phi_n\}$$

$$\phi_1 \wedge \dots \wedge \phi_n \rightarrow \{\phi_1\}, \dots, \{\phi_n\}$$



Transformarea în formă clauzală II

- ▶ Exemplu: $p \wedge (q \Rightarrow r)$

$$I \quad p \wedge (\neg q \vee r)$$

$$C \quad \{p\}, \{\neg q, r\}$$

- ▶ Exemplu: $\neg(p \wedge (q \Rightarrow r))$

$$I \quad \neg(p \wedge (\neg q \vee r))$$

$$N \quad \neg p \vee \neg(\neg q \vee r)$$

$$N \quad \neg p \vee (q \wedge \neg r)$$

$$D \quad (\neg p \vee q) \wedge (\neg p \vee \neg r)$$

$$C \quad \{\neg p, q\}, \{\neg p, \neg r\}$$



Rezoluție I

- ▶ Ideea:

$$\frac{\begin{array}{c} \{p, q\} \\ \{\neg p, r\} \end{array}}{\{q, r\}}$$

- ▶ “Anularea” lui p cu $\neg p$
- ▶ p adevărată, $\neg p$ falsă, deci r adevărată
- ▶ p falsă, deci q adevărată
- ▶ Cel puțin una dintre q și r adevărată
- ▶ Forma generală:

$$\frac{\begin{array}{c} \{p_1, \dots, r, \dots, p_m\} \\ \{q_1, \dots, \neg r, \dots, q_n\} \end{array}}{\{p_1, \dots, p_m, q_1, \dots, q_n\}}$$



Rezoluție II

- ▶ Rezolvent **vid** — **contradicție** între premise:

$$\frac{\begin{array}{c} \{\neg p\} \\ \{p\} \end{array}}{\{\}}$$

- ▶ **Mai mult de 2** rezolvenți posibili — se alege doar unul:

$$\frac{\begin{array}{c} \{p, q\} \\ \{\neg p, \neg q\} \end{array}}{\begin{array}{c} \{p, \neg p\} \\ \{q, \neg q\} \end{array}}$$

Rezoluție III

- *Modus Ponens* — caz particular al rezoluției:

$$\frac{p \Rightarrow q \quad p}{q} \qquad \frac{\{\neg p, q\} \quad \{p\}}{\{q\}}$$

- *Modus Tollens* — caz particular al rezoluției:

$$\frac{p \Rightarrow q \quad \neg q}{\neg p} \qquad \frac{\{\neg p, q\} \quad \{\neg q\}}{\{\neg p\}}$$

- *Tranzitivitatea* implicației:

$$\frac{p \Rightarrow q \quad q \Rightarrow r}{p \Rightarrow r} \qquad \frac{\{\neg p, q\} \quad \{\neg q, r\}}{\{\neg p, r\}}$$



Rezoluție IV

- ▶ Demonstrarea **nesatisfiabilității** — derivarea clauzei **vide**
- ▶ Demonstrarea **derivabilității** concluziei ϕ din premisele $\phi_1, \dots, \phi_n$ — demonstrarea **nesatisfiabilității** propoziției $\phi_1 \wedge \dots \wedge \phi_n \wedge \neg \phi$ (reducere la absurd)
- ▶ Demonstrarea **validității** propoziției ϕ — demonstrarea **nesatisfiabilității** propoziției $\neg \phi$
- ▶ Rezoluția incompletă **generativ**, i.e. concluziile **nu** pot fi derivate direct, răspunsul fiind dat în raport cu o “întrebare” fixată



Rezoluție V

Demonstrăm prin reducere la absurd că

$\{p \Rightarrow q, q \Rightarrow r\} \vdash p \Rightarrow r$, i.e. că mulțimea

$\{p \Rightarrow q, q \Rightarrow r, \neg(p \Rightarrow r)\}$ conține o **contradicție**.

1	$\{\neg p, q\}$	Premisă
2	$\{\neg q, r\}$	Premisă
3	$\{p\}$	Concluzie negată
4	$\{\neg r\}$	Concluzie negată
5	$\{q\}$	1, 3
6	$\{r\}$	2, 5
7	$\{\}$	4, 6



Rezoluție VI

- *Teorema rezoluției*: rezoluția propozițională este **consistentă și completă** (nu generativ, v. slide-ul 367):

$$\Delta \models \phi \Leftrightarrow \Delta \vdash \phi$$

- **Terminarea** garantată a procedurii de aplicare a rezoluției: număr **finit** de clauze, număr **finit** de concluzii



Cuprins

Introducere

Logica propozițională

Sintaxă și semantică

Satisfiabilitate și validitate

Derivabilitate

Inferență și demonstrație

Rezoluție

Logica cu predicate de ordinul I

Sintaxă și semantică

Forma clauzală

Unificare



Logica cu predicate de ordinul I

- ▶ Logica propozițională:
 - ▶ p : “Andrei este prieten cu Bogdan.”
 - ▶ q : “Bogdan este prieten cu Andrei.”
 - ▶ $p \Leftrightarrow q$
 - ▶ **Opacitate** în raport cu obiectele și relațiile referite



Logica cu predicate de ordinul I

- ▶ Logica propozițională:
 - ▶ p : “Andrei este prieten cu Bogdan.”
 - ▶ q : “Bogdan este prieten cu Andrei.”
 - ▶ $p \Leftrightarrow q$
 - ▶ **Opacitate** în raport cu obiectele și relațiile referite
- ▶ *First-order logic* (FOL) = **extensie** a logicii propoziționale, cu explicitarea:
 - ▶ **obiectelor** din universul problemei
 - ▶ **relațiilor** dintre acestea

Logica cu predicate de ordinul I

- ▶ Logica propozițională:
 - ▶ p : “Andrei este prieten cu Bogdan.”
 - ▶ q : “Bogdan este prieten cu Andrei.”
 - ▶ $p \Leftrightarrow q$
 - ▶ **Opacitate** în raport cu obiectele și relațiile referite
- ▶ *First-order logic* (FOL) = **extensie** a logicii propoziționale, cu explicitarea:
 - ▶ **obiectelor** din universul problemei
 - ▶ **relațiilor** dintre acestea
- ▶ FOL:
 - ▶ Generalizare: $prieten(x, y)$: “**x** este prieten cu **y**.”
 - ▶ $\forall x. \forall y. (prieten(x, y) \Leftrightarrow prieten(y, x))$
 - ▶ Aplicare pe cazuri **particulare**
 - ▶ **Transparentă** în raport cu obiectele și relațiile referite

(Genesereth, 2010)



Cuprins

Introducere

Logica propozițională

Sintaxă și semantică

Satisfiabilitate și validitate

Derivabilitate

Inferență și demonstrație

Rezoluție

Logica cu predicate de ordinul I

Sintaxă și semantică

Forma clauzală

Unificare



Sintaxă

Simboluri utilizate

- ▶ **Constante**: obiecte particulare din universul discursului: c , d , $andrei$, $bogdan$, ...
- ▶ **Variable**: obiecte generice: x , y , ...
- ▶ Simboluri **funcționale**: $succesor(x)$, $+(x, y)$, ...
- ▶ Simboluri **relaționale** (**predicate**): relații n -are peste obiectele din universul discursului: $divide(x, y)$, $impar(x)$, ...
- ▶ **Conectori** logici: $\neg$, $\wedge$, ...
- ▶ **Cuantificatori**: $\forall$, $\exists$



Sintaxă I

Termeni, atomi, propoziții

- ▶ **Termeni** (obiecte):
 - ▶ Constante
 - ▶ Variabile
 - ▶ Aplicații de funcții: $f(t_1, \dots, t_n)$, unde f este un simbol **funcțional** n -ar și $t_1, \dots, t_n$ sunt termeni. Exemple:
 - ▶ $\text{succesor}(4)$: succesorul lui 4
 - ▶ $+(2, x)$: suma simbolurilor 2 și x



Sintaxă II

Termeni, atomi, propoziții

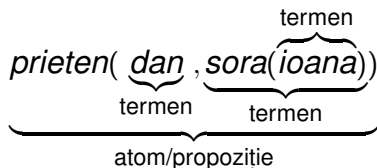
- ▶ **Atomi** (relații): $p(t_1, \dots, t_n)$, unde p este un **predicat** n -ar și $t_1, \dots, t_n$ sunt termeni. Exemple:
 - ▶ $impar(3)$
 - ▶ $varsta(ion, 20)$
 - ▶ $= (+ (2, 3), 5)$
- ▶ **Propoziții** (fapte) — x variabilă, A atom, α propoziție:
 - ▶ Fals, adevărat: $\perp, \top$
 - ▶ Atomi: A
 - ▶ Negații: $\neg \alpha$
 - ▶ ...
 - ▶ Cuantificări: $\forall x. \alpha, \exists x. \alpha$



Sintaxă III

Termeni, atomi, propoziții

Exemplu: “Dan este prieten cu sora Ioanei”:



- ▶ Simplificare: **legarea** tuturor variabilelor, prin cuantificatori universali sau existențiali
- ▶ **Zona de acțiune** a unui cuantificator: restul propoziției (v. simbolul λ în calculul lambda)



Semantică I

O *interpretare* constă din:

- ▶ Un **domeniu** nevid, D
- ▶ Pentru fiecare **constantă** c , un element $c^I \in D$
- ▶ Pentru fiecare simbol **funcțional** n -ar, f , o funcție $f^I : D^n \rightarrow D$
- ▶ Pentru fiecare **predicat** n -ar, p , o funcție $p^I : D^n \rightarrow \{false, true\}$.



Semantică II

- ▶ Atom:

$$(p(t_1, \dots, t_n))^I = p^I(t_1^I, \dots, t_n^I)$$

- ▶ Negatie etc. (v. logica propozițională)

- ▶ Cuantificare **universală**:

$$(\forall x. \alpha)^I = \begin{cases} false & \text{dacă există } d \in D \text{ cu } \alpha^I_{[d/x]} = false \\ true & \text{altfel} \end{cases}$$

- ▶ Cuantificare **existențială**:

$$(\exists x. \alpha)^I = \begin{cases} true & \text{dacă există } d \in D \text{ cu } \alpha^I_{[d/x]} = true \\ false & \text{altfel} \end{cases}$$



Exemple

1. “Vrabia mălai visează.”



Exemple

1. “Vrabia mălai visează.”

$$\forall x. (vrabie(x) \Rightarrow viseaza(x, malai))$$

Exemple

1. “Vrăbia mălai visează.”

$\forall x. (vrăbie(x) \Rightarrow viseaza(x, malai))$

2. “Unele vrăbii visează mălai.”



Example

1. “Vrăbia mălai visează.”

$$\forall x. (vrăbie(x) \Rightarrow viseaza(x, malai))$$

2. “Unele vrăbii visează mălai.”

$$\exists x. (vrăbie(x) \wedge viseaza(x, malai))$$



Example

1. “Vrabia mălai visează.”

$$\forall x. (vrabie(x) \Rightarrow viseaza(x, malai))$$

2. “Unele vrăbii visează mălai.”

$$\exists x. (vrabie(x) \wedge viseaza(x, malai))$$

3. “Nu toate vrăbiile visează mălai.”



Example

1. “Vrabia mălai visează.”

$$\forall x. (vrabie(x) \Rightarrow viseaza(x, malai))$$

2. “Unele vrăbii visează mălai.”

$$\exists x. (vrabie(x) \wedge viseaza(x, malai))$$

3. “Nu toate vrăbiile visează mălai.”

$$\exists x. (vrabie(x) \wedge \neg viseaza(x, malai))$$



Example

1. “Vrabia mălai visează.”

$$\forall x. (vrabie(x) \Rightarrow viseaza(x, malai))$$

2. “Unele vrăbii visează mălai.”

$$\exists x. (vrabie(x) \wedge viseaza(x, malai))$$

3. “Nu toate vrăbiile visează mălai.”

$$\exists x. (vrabie(x) \wedge \neg viseaza(x, malai))$$

4. “Nicio vrabie nu visează mălai.”



Example

1. “Vrabia mălai visează.”

$$\forall x. (vrabie(x) \Rightarrow viseaza(x, malai))$$

2. “Unele vrăbii visează mălai.”

$$\exists x. (vrabie(x) \wedge viseaza(x, malai))$$

3. “Nu toate vrăbiile visează mălai.”

$$\exists x. (vrabie(x) \wedge \neg viseaza(x, malai))$$

4. “Nicio vrabie nu visează mălai.”

$$\forall x. (vrabie(x) \Rightarrow \neg viseaza(x, malai))$$



Example

1. “Vrabia mălai visează.”
 $\forall x. (vrabie(x) \Rightarrow viseaza(x, malai))$
2. “Unele vrăbii visează mălai.”
 $\exists x. (vrabie(x) \wedge viseaza(x, malai))$
3. “Nu toate vrăbiile visează mălai.”
 $\exists x. (vrabie(x) \wedge \neg viseaza(x, malai))$
4. “Nicio vrabie nu visează mălai.”
 $\forall x. (vrabie(x) \Rightarrow \neg viseaza(x, malai))$
5. “Numai vrăbiile visează mălai.”



Example

1. “Vrabia mălai visează.”
 $\forall x. (vrabie(x) \Rightarrow viseaza(x, malai))$
2. “Unele vrăbii visează mălai.”
 $\exists x. (vrabie(x) \wedge viseaza(x, malai))$
3. “Nu toate vrăbiile visează mălai.”
 $\exists x. (vrabie(x) \wedge \neg viseaza(x, malai))$
4. “Nicio vrabie nu visează mălai.”
 $\forall x. (vrabie(x) \Rightarrow \neg viseaza(x, malai))$
5. “Numai vrăbiile visează mălai.”
 $\forall x. (viseaza(x, malai) \Rightarrow vrabie(x))$



Example

1. “Vrabia mălai visează.”
 $\forall x. (vrabie(x) \Rightarrow viseaza(x, malai))$
2. “Unele vrăbii visează mălai.”
 $\exists x. (vrabie(x) \wedge viseaza(x, malai))$
3. “Nu toate vrăbiile visează mălai.”
 $\exists x. (vrabie(x) \wedge \neg viseaza(x, malai))$
4. “Nicio vrabie nu visează mălai.”
 $\forall x. (vrabie(x) \Rightarrow \neg viseaza(x, malai))$
5. “Numai vrăbiile visează mălai.”
 $\forall x. (viseaza(x, malai) \Rightarrow vrabie(x))$
6. “Toate și numai vrăbiile visează mălai.”



Example

1. “Vrabia mălai visează.”

$$\forall x. (vrabie(x) \Rightarrow viseaza(x, malai))$$

2. “Unele vrăbii visează mălai.”

$$\exists x. (vrabie(x) \wedge viseaza(x, malai))$$

3. “Nu toate vrăbiile visează mălai.”

$$\exists x. (vrabie(x) \wedge \neg viseaza(x, malai))$$

4. “Nicio vrabie nu visează mălai.”

$$\forall x. (vrabie(x) \Rightarrow \neg viseaza(x, malai))$$

5. “Numai vrăbiile visează mălai.”

$$\forall x. (viseaza(x, malai) \Rightarrow vrabie(x))$$

6. “Toate și numai vrăbiile visează mălai.”

$$\forall x. (viseaza(x, malai) \Leftrightarrow vrabie(x))$$



Cuantificatori

Greșeli frecvente

- ▶ $\forall x.(vrabie(x) \Rightarrow viseaza(x, malai))$
→ corect: “Toate vrăbiile visează mălai.”
- ▶ $\forall x.(vrabie(x) \wedge viseaza(x, malai))$
→ **greșit**: “Toți sunt vrăbii care visează mălai.”
- ▶ $\exists x.(vrabie(x) \wedge viseaza(x, malai))$
→ corect: “Unele vrăbii visează mălai.”
- ▶ $\exists x.(vrabie(x) \Rightarrow viseaza(x, malai))$
→ **greșit**: adevărată și dacă există cineva care nu este vrabie



Cuantificatori

Proprietăți

► *Necomutativitate:*

- $\forall x. \exists y. \text{viseaza}(x, y)$: “Toți visează la ceva **particular**.”
- $\exists y. \forall x. \text{viseaza}(x, y)$: “Toți visează la **același** lucru.”

► *Dualitate:*

- $\neg(\forall x. \alpha) \equiv \exists x. \neg \alpha$
- $\neg(\exists x. \alpha) \equiv \forall x. \neg \alpha$



Aspecte legate de propoziții

Analoage logicii propoziționale:

- ▶ Satisfiabilitate
- ▶ Validitate
- ▶ Derivabilitate
- ▶ Inferență
- ▶ Demonstrație



Cuprins

Introducere

Logica propozițională

Sintaxă și semantică

Satisfiabilitate și validitate

Derivabilitate

Inferență și demonstrație

Rezoluție

Logica cu predicate de ordinul I

Sintaxă și semantică

Forma clauzală

Unificare



Forma clauzală

- ▶ *Literal*: **atom** ($prieten(x, y)$) sau **negația** lui ($\neg prieten(x, y)$)
- ▶ *Expresie clauzală* = **literal** sau **disjuncție** de literali, e.g. $prieten(x, y) \vee \neg doctor(x)$
- ▶ *Clauză* = **mulțime** de literali dintr-o expresie clauzală, e.g. $\{prien(x, y), \neg doctor(x)\}$
- ▶ *Clauză Horn* = clauză în care un **singur** literal este în formă pozitivă, e.g. $\{\neg A_1, \dots, \neg A_n, A\}$, corespunzătoare **implicației** $A_1 \wedge \dots \wedge A_n \Rightarrow A$



Transformarea în formă clauzală I

1. Eliminarea **implicațiilor** (I)
2. Introducerea **negațiilor** în interiorul expresiilor (N)
3. **Redenumirea** variabilelor cuantificate pentru obținerea **unicității** de nume (R):

$$\forall x.p(x) \wedge \forall x.q(x) \vee \exists x.r(x) \rightarrow \forall x.p(x) \wedge \forall y.q(y) \vee \exists z.r(z)$$

4. Deplasarea cuantificatorilor la **începutul** expresiei, conservându-le **ordinea** (forma normală *prenex*) (P):

$$\forall x.p(x) \wedge \forall y.q(y) \vee \exists z.r(z) \rightarrow \forall x.\forall y.\exists z.(p(x) \wedge q(y) \vee r(z))$$



Transformarea în formă clauzală II

5. Eliminarea cuantificatorilor **existențiali** (skolemizare) (S):

- ▶ Dacă **nu** este precedat de cuantificatori universal: înlocuirea aparițiilor variabilei cuantificate printr-o **constantă**:

$$\exists x.p(x) \rightarrow p(c_x)$$

- ▶ Dacă este **precedat** de cuantificatori universal: înlocuirea aparițiilor variabilei cuantificate prin aplicația unei **funcții** unice asupra variabilelor anterior cuantificate universal:

$$\forall x.\forall y.\exists z.(p(x) \wedge q(y) \vee r(z)) \rightarrow \forall x.\forall y.(p(x) \wedge q(y) \vee r(f_z(x, y)))$$



Transformarea în formă clauzală III

6. Eliminarea cuantificatorilor **universali**, considerați acum implicați (U):

$$\forall x. \forall y. (p(x) \wedge q(y) \vee r(f_z(x, y))) \rightarrow p(x) \wedge q(y) \vee r(f_z(x, y))$$

7. **Distribuirea** lui $\vee$ față de $\wedge$ (D):

$$\alpha \vee (\beta \wedge \gamma) \rightarrow (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$$

8. Transformarea expresiilor în **clauze** (C)

Transformarea în formă clauzală IV

Exemplu: "Cine rezolvă toate laboratoarele este apreciat de cineva."

$$\forall x. (\forall y. (lab(y) \Rightarrow rezolva(x, y)) \Rightarrow \exists y. apreciaza(y, x))$$

$$I \quad \forall x. (\neg \forall y. (\neg lab(y) \vee rezolva(x, y)) \vee \exists y. apreciaza(y, x))$$

$$N \quad \forall x. (\exists y. \neg (\neg lab(y) \vee rezolva(x, y)) \vee \exists y. apreciaza(y, x))$$

$$N \quad \forall x. (\exists y. (lab(y) \wedge \neg rezolva(x, y)) \vee \exists y. apreciaza(y, x))$$

$$R \quad \forall x. (\exists y. (lab(y) \wedge \neg rezolva(x, y)) \vee \exists z. apreciaza(z, x))$$

$$P \quad \forall x. \exists y. \exists z. ((lab(y) \wedge \neg rezolva(x, y)) \vee apreciaza(z, x))$$

$$S \quad \forall x. ((lab(f_y(x)) \wedge \neg rezolva(x, f_y(x))) \vee apreciaza(f_z(x), x))$$

$$U \quad (lab(f_y(x)) \wedge \neg rezolva(x, f_y(x))) \vee apreciaza(f_z(x), x)$$

$$D \quad (lab(f_y(x)) \vee apreciaza(f_z(x), x)) \\ \wedge (\neg rezolva(x, f_y(x)) \vee apreciaza(f_z(x), x))$$

$$C \quad \{lab(f_y(x)), apreciaza(f_z(x), x)\}, \\ \{\neg rezolva(x, f_y(x)), apreciaza(f_z(x), x)\}$$



Cuprins

Introducere

Logica propozițională

Sintaxă și semantică

Satisfiabilitate și validitate

Derivabilitate

Inferență și demonstrație

Rezoluție

Logica cu predicate de ordinul I

Sintaxă și semantică

Forma clauzală

Unificare



Motivație

- ▶ Rezoluție:

$$\frac{\{prieten(x, mama(y)), doctor(x)\} \\ \{\neg prieten(mama(z), z)\}}{?}$$

- ▶ Cum aplicăm rezoluția?
- ▶ Soluția: **unificare** (v. sinteza de tip, slide-ul 240)
- ▶ MGU: $S = \{x \leftarrow mama(z), z \leftarrow mama(y)\}$
- ▶ Forma **comună** a celor doi atomi:
 $prieten(mama(mama(y)), mama(y))$
- ▶ **Rezolvent**: $doctor(mama(mama(y)))$



Unificare I

- ▶ Problemă **NP-completă**
- ▶ Posibile legări **ciclice**
- ▶ Exemplu: $prieten(x, mama(x))$ și $prieten(mama(y), y)$
- ▶ MGU: $S = \{x \leftarrow mama(y), y \leftarrow mama(x)\}$
- ▶ $x \leftarrow mama(mama(x)) \rightarrow$ **imposibil!**
- ▶ Soluție: verificarea apariției unei variabile în expresia la care a fost **legată** (*occurrence check*)



Unificare II

- ▶ Rezoluția pentru clauze **Horn**:

$$\frac{\begin{array}{l} A_1 \wedge \dots \wedge A_m \Rightarrow A \\ B_1 \wedge \dots \wedge A' \wedge \dots \wedge B_n \Rightarrow B \\ \text{unificare}(A, A') = S \end{array}}{\text{subst}(S, A_1 \wedge \dots \wedge A_m \wedge B_1 \wedge \dots \wedge B_n \Rightarrow B)}$$

- ▶ $\text{unificare}(\alpha, \beta)$: **substituția** sub care unifică propozițiile α și β
- ▶ $\text{subst}(S, \alpha)$: propoziția rezultată în urma **aplicării** substituției S asupra propoziției α



Rezumat

- ▶ Expresivitatea superioară a logicii cu predicate de ordinul I, față de cea propozițională
- ▶ Propoziții satisfiabile, valide, nesatisfiabile
- ▶ Derivabilitate logică: proprietatea unei propoziții de a reprezenta consecința logică a altora
- ▶ Derivabilitate mecanică (inferență): posibilitatea unei propoziții de a fi determinată drept consecință a altora, în baza unei proceduri de calcul (de inferență)
- ▶ Rezoluție: procedură de inferență consistentă și completă (nu generativ)



Bibliografie

Harrison, J. (2009). *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press.

Genesereth, M. (2010). *CS157: Computational Logic*, curs Stanford.

<http://logic.stanford.edu/classes/cs157/2010/cs157.html>

