Paradigme de Programare

Ș.l. dr. ing. Mihnea Muraru
mmihnea@gmail.com

2013–2014, semestrul 2

Partea I

Introducere

Cuprins

Cuprins

Notare

- Teste la curs: 0,5
- Test grilă: 0,5
- Laborator: 1
- Teme: 4 (3 $\times$ 1.33)
- Examen: 4

Regulament

Vă rugăm să citiți regulamentul cu atenție!

`http://elf.cs.pub.ro/pp/regulament`

Desfășurarea cursului

- Recapitularea cursului anterior
- Predare
- Test din cursul anterior
- Feedback despre cursul curent (de acasă)

Cuprins

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?

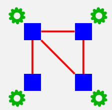
The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

Edsger Dijkstra,
How do we tell truths that might hurt

Descompunerea problemelor

Controlul complexității: descompunere și interfațare

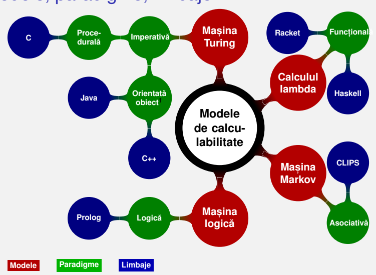
Descompunere	Accent pe	Rezultat
Procedurală	Acțiuni	Proceduri
Orientată obiect	Entități	Clase și obiecte
Funcțională	Relații	Funcții în sens matematic
Logică	Relații	Predicate și propoziții



De ce? (cont.)

- Lărgirea spectrului de abordare a problemelor
- Identificarea perspectivei ce permite modelarea simplă a unei probleme și 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



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

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?

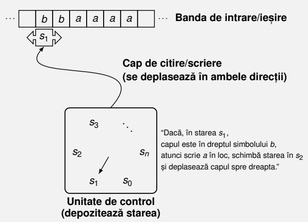
Cuprins

O primă problemă

Example 3.1.
Să se determine elementul minim dintr-un vector.

Abordare imperativă
Modelul

Mașina Turing



Prelucrare după: `http://www.texample.net/tikz/examples/turing-machine-2/`

Abordare imperativă (procedurală)
Limbajul

```
procedure MinList(L, n)
    min ← L[1]
    i ← 2
    while i ≤ n do
        if L[i] < min then
            min ← L[i]
        end if
        i ← i + 1
    end while
    return min
end procedure
```


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

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

"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, obținându-se rezultatul unui pas de evaluare."

Abordare funcțională
Limbajul

- Racket (2 variante):

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

(define (minList2 L)
  (foldl min (car L) (cdr L)))
```

- Haskell (aceleași 2 variante):

```
minList1 [h]     = h
minList1 (h : t) = min h (minList1 t)

minList2 (h : t) = foldl min h t
```


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

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

"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:
  1. $x \le y \Rightarrow min(x, y, x)$
  2. $y < x \Rightarrow min(x, y, y)$
  3. $minList([m], m)$
  4. $minList([y|t], n) \wedge min(x, n, m) \Rightarrow minList([x, y|t], m)$
- Prolog:

```
min(X, Y, X) :- X =< Y.
min(X, Y, Y) :- Y < X.

minList([M], M).
minList([X, Y | T], M) :-
    minList([Y | T], N), min(X, N, M).
```


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
- "Ce" trebuie obținut (vs. "cum" la imperativă)
- Se subsumează abordării declarative, opuse celei imperative

Cuprins

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)

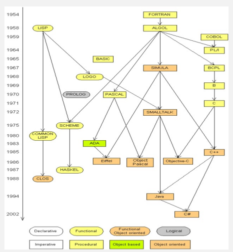
Accepții asupra limbajelor

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

Istoric



Câteva trăsături

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

Rezumat

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

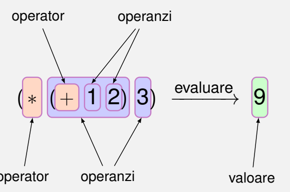
Partea II

Limbajul Racket

Cuprins

Cuprins

Expresii



Evaluarea expresiilor primitive

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

Recursiv pentru subexpresii

```
(* (+ 1 2) 3) → (* 3 3) → 9
```

Racket stepper

Construcția `define`

Scop

```
(define WIDTH 100)
```

- *Leagă* o variabilă globală la valoarea unei expresii
- Atenție! Principial, 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. La definire, se evaluează expresia, și se *leagă* variabila la valoarea ei
2. La utilizare, variabila se evaluează la valoarea ei

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


Funcții

Definire

nume

parametri

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

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

corp (o **unică** expresie)

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

Funcții

Evaluare

Definire:

- Înregistrarea definiției funcției

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

Aplicare:

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

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


Construcția `if`

Prezentare

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

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

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!

```
(if (< 1 2) (+ 3 4) (+ 5 6))
→ (if true (+ 3 4) (+ 5 6))
→ (+ 3 4) → 7
```


Cuprins

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
  - O listă se obține prin introducerea unui element (*head*) în vârful altei liste (*tail*)

    `(cons 0 '(1 2)) → '(0 1 2)`

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

  `(list 0 1 2)`

- Selectori

```
(car '(0 1 2)) → 0
(cdr '(0 1 2)) → '(1 2)
```


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*:

    $(minList\ (cons\ e\ '()))=e$

    $(minList\ (cons\ e\ L))\ =(min\ e\ (minList\ (cdr\ L)))$

  - Implementare

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

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

Perechi

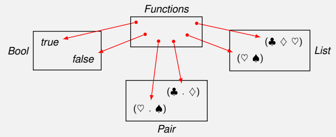
- Intern, listă ≡ pereche *head-tail*
- `cons`, aplicabil asupra oricăror doi operanzi, pentru generarea unei perechi cu punct (*dotted pair*)

```
(cons 0 1) → '(0 . 1)
'(0 1 2) ≡ '(0 . (1 . (2 . ())))
```

- 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)
- Majoritatea tipurilor uzuale, codificabile direct prin intermediul funcțiilor



Cuprins

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")
```

- 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"))
```

- 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

Omoiconicitate și metaprogramare

- Corepondență între sintaxa programului și strucura 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

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

(define minus (cons '- (cdr plus)))  ; '(- 3 2)
(eval minus)  ; 1
```

Forțarea evaluării de către `eval`

Rezumat

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

Partea III

Recursivitate

Cuprins

Cuprins

Recursivitate

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

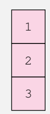
Cuprins

Funcția *factorial*

Recursivitate pe stivă, liniară

```
(define (fact-stack n)
  (if (= n 1)
      1
      (* n (fact-stack (- n 1)))))
```

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



Stiva procesului

Exemple preluate din: Abelson and Sussman (1996)

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** $i \le n$ **do**
5: $product \leftarrow product \cdot i$
6: $i \leftarrow i+1$
7: **end while**
8: **return** $product$
9: **end procedure**

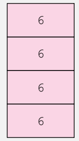
- Starea programului: variabilele i și $product$
- Spațiu constant pe stivă!
- Cum putem exploata această idee?

Funcția *factorial*

Recursivitate pe coadă

```
(define (fact-tail n)
  (fact-tail-helper 1 1 n))

(define (fact-tail-helper product i n)
  (if (> i n)
      product
      (fact-tail-helper (* product i)
                        (+ i 1)
                        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
```

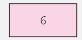


Stiva aparentă

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: $\Theta(n)$
- 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
- Proces: Funcție în execuție, aspectul ei dinamic
- Posibilitatea unei funcții textual recursive (e.g. pe coadă) de a genera un proces iterativ!

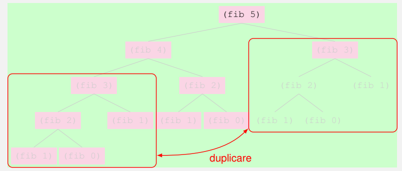
Funcția *Fibonacci*

Recursivitate pe stivă, arborescentă

```
(define (fib-stack n)
  (cond [(= n 0) 0]
        [(= n 1) 1]
        [else (+ (fib-stack (- n 1))
                 (fib-stack (- n 2)))]))
```


Funcția *Fibonacci* (cont.)

Recursivitate pe stivă, arborescentă



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

```
(define (fib-tail n)
  (fib-tail-helper 1 0 n))

(define (fib-tail-helper a b count)
  (if (= count 0)
      b
      (fib-tail-helper (+ a b) a (- count 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.	Pe coadă
Elegantă, adesea apropiată de specificație	Obscură, necesitând prelucrări specifice
Ineficientă spațial și/sau temporal	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)
- În anumite situații, imposibilă direct:

```
(define (f x)
  (if (zero? x)
      0
      (g (f (- x 1)))))
      ; comportamentul lui g depinde
      ; de parametru
```


Cuprins

Construirea rezultatului

Recursivitate pe stivă

```
;; Inmulteste cu 10 toate elementele listei L
(define (mult-stack L)
  (if (null? L)
      L
      (cons (* (car L) 10)
            (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

```
;; Inmulteste cu 10 toate elementele listei L
(define (mult-tail-helper L Result)
  (if (null? L)
      Result
      (mult-tail-helper (cdr L)
                        (cons (* (car L) 10)
                              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:

- Inversarea listei finale

```
(if (null? L)
    (reverse Result)
    ...)
```

- Adăugarea elementului curent la sfârșitul acumul.

```
(if (null? L)
    ...
    (mult-all-iter
      (cdr L)
      (append Result
              (list (* (car L) 10)))))
```


Costul unei concatenări

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

Număr de operații proporțional cu lungimea primei liste!

Costul concatenărilor repetate

- Asociere la dreapta:

  `A ++ (B ++ (C ++ ...)...)`

  Număr de operații proporțional cu lungimea listei curente

- Asociere la stânga:

  `(...(... ++ A) ++ B) ++ C`

  Număr de operații proporțional cu lungimea tuturor listelor concatenate anterior

Consecințe asupra recursivității pe coadă

```
(define (mult-tail-helper L Result)
  (if (null? L)
      Result
      (mult-tail-helper
        (cdr L)
        (append Result
                (list (* (car L) 10))))))
```

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


Consecințe asupra recursivității pe coadă (cont.)

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

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

- 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

Cuprins

Abstractizare funcțională

double

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

```
(define (double n)
  (+ n n))
```

`(* 5 2)` `(* 10 2)`

- 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` = *abstracție funcțională*

Un nivel mai sus

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

;; Obtine paritatea fiecarui numar (true = par)
;; '(1 2 3) -> '(false true false)
(define (parities L)
  (if (null? L)
      L
      (cons (even? (car L))
            (parities (cdr L)))))
```

singura parte **variabilă**, dependentă de `(car L)`

Un nivel mai sus (cont.)

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

```
;; map = asociere

(define (mult-map x)
  (* x 10))

(define (parities-map x)
  (even? x))
```

rolul lui `(car L)`

Un nivel mai sus (cont.)

```
(define (map f L)
  (if (null? L)
      L
      (cons (f (car L))
            (map f (cdr L)))))

(define (mult L)
  (map mult-map L))

(define (parities L)
  (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

Funcții ca valori de prim rang

- În exemplele anterioare: funcții văzute ca date!
- Avantaj: sporire considerabilă a expresivității limbajului
- Statutul funcțiilor de valori de prim rang, 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!

```
> +
#<procedure:+>

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

> (list + - *)
(#<procedure:+> #<procedure:-> #<procedure:*>)
```


Funcții ca parametru

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

```
(define (mult L)
  (map mult-map L))

(define (parities L)
  (map parities-map L))
```

- Putem defini funcțiile local unei expresii?

Funcții anonime

constructor
parametru
corp

```
(define (mult L)
  (map (lambda (x) (* x 10)) L))

(define (parities L)
  (map (lambda (x) (even? x)) L))
```

De fapt,

```
(define (mult-map x)
  (* x 10))
```

≡

```
(define mult-map
  (lambda (x)
    (* 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?
- Posibilă utilizare, pentru înmulțirea cu 5:

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

funcție

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

```
(define (mult-map-by q x)
  (* x q))
```

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

simultan (***uncurried***)

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
- Avantaj: reutilizare, prin aplicare parțială!
- Funcție *curried*: preia parametrii pe rând (aparent)
- Funcție *uncurried*: preia parametrii simultan

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 transpare la utilizarea *stepper*-ului din Racket, dar este prezent pe slide pentru completitudine.

Aplicație: compunerea a două funcții

```
(define (comp f g)
  (lambda (x)
    (f (g x))))

((comp car cdr) '(1 2 3)) → 2
```


Cuprins

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):
  - `map`
  - `filter`
  - `foldl` (*fold left*)
  - `foldr` (*fold right*)

Funcționala `map`

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

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


Funcționala `filter`

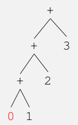
- Extragerea dintr-o listă a elementelor care satisfac un predicat logic
- Funcția primită ca parametru trebuie să întoarcă o valoare booleană

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


Funcționala `foldl`

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

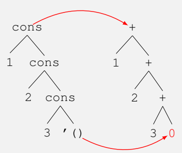
```
(foldl + 0 '(1 2 3))
→ 6
```



Funcționala `foldr`

- Similar cu `foldl`
- Pacurgere dreapta → stânga
- Operare pe structura listei inițiale
- Natural recursivă pe stivă

```
(foldr + 0 '(1 2 3))
→ 6
```



Universalitatea funcționalelor `fold*`

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

Cuprins

Trăsături

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

Evaluare

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

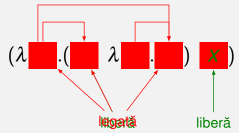
"Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, y, se indentifică parametrul formal, x, în corpul funcției, x, iar aparițiile primului, x (singura), se **substituie** cu parametrul actual, obținându-se rezultatul unui pas de evaluare."

Formalizarea substituției

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

- 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



- 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**!
- 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

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

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
- În exemplul anterior, $(\lambda x.\lambda z.x \;\; cons) \rightarrow \lambda z.cons$
- Rezultat corect, cu păstrarea statutului de apariție liberă

Universalitatea funcțiilor

- Posibilitatea reprezentării tuturor valorilor uzuale exclusiv prin funcții (v. slide-ul 50)
- Mai devreme, funcții ca date (parametri, valori de retur etc.)
- Acum, date ca funcții!!
- V. sursele ataște 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

Cuprins

Variabile

Proprietăți

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

Variabile

Stări

- Declarată: cunoaștem identificatorul
- Definită: cunoaștem și valoarea

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

```
def x = 0
f() { return x }
def x = 1
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
- Domeniu de vizibilitate determinat prin constrcuțiile limbajului (lexical), la compilare (static)

```
def x = 0
f() { return x }
def x = 1
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?

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

Legare dinamică

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

```
def x = 0
f() { return x }                      ◁ f() -> 0
def x = 1                             ◁ f() -> 1
g() { def x = 2 ; return f() }        ◁ 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

```
def x = 0
f() { return x }                      ◁ f() -> 0
def x = 1                             ◁ f() -> 1
g() { def x = 2 ; return f() }        ◁ 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`
- Variabile *top-level*: dinamic:
  - `define`

Construcția `lambda`
Definiție

- Leagă static parametrii formali ai unei funcții
- Sintaxă:

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

- Domeniul de vizibilitate a parametrului `pk` = mulțimea punctelor din corpul funcției, `expr`, în care aparițiile lui `pk` sunt libere (v. slide-ul 128)

Construcția `lambda`
Exemplu

```
(lambda (x)
  (x (lambda (y) y)))
```


Construcția `lambda`
Semantică

- Aplicație:

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

- Se evaluează operanzii `ak`, în ordine aleatoare (evaluare aplicativă)
- Se evaluează corpul funcției, `expr`, ținând cont de legările `pk` ← *valoare*(`ak`)
- Valoarea aplicației este valoarea lui `expr`

Construcția `let`
Definiție

- Leagă static variabile locale
- Sintaxă:

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

- Domeniul de vizibilitate a variabilei `vk` = mulțimea punctelor din corp, `expr`, în care aparițiile lui `vk` sunt libere (v. slide-ul 128)

Construcția `let`
Exemplu

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


Construcția `let`
Semantică

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

echivalent cu

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


Construcția `let*`
Definiție

- Leagă static variabile locale
- Sintaxă:

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

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

  în care aparițiile lui `vk` sunt libere (v. slide-ul 128)

Construcția `let*`
Exemplu

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


Construcția `let*`
Semantică

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

echivalent cu

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

Evaluarea expresiilor se face în ordine!

Construcția `letrec`
Definiție

- Leagă static variabile locale
- Sintaxă:

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

- Domeniul de vizibilitate a variabilei `vk` = mulțimea punctelor din întreaga construcție, în care aparițiile lui `vk` sunt libere (v. slide-ul 128)

Construcția `letrec`
Exemplu

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


Construcția `define`
Definiție

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

```
(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

```
(define x 0)
(define f (lambda () x))
(f)  ; 0
(define x 1)
(f)  ; 1
```


Construcția `define`
Exemple

```
(define factorial
  (lambda (n)
    (if (zero? n) 1
        (* n (factorial (- n 1))))))

(factorial 5)  ; 120

(define g factorial)
(define factorial (lambda (x) x))

(g 5)          ; 20
```


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 131

```
(define x 0)
(define f (lambda () x))
(define x 1)

(define g
  (lambda (x)
    (f)))

(g 2)  ; 1
```


Aplicație pentru legarea variabilelor

```
(define (app A B)
  (if (null? A)
      B
      (cons (car A) (app (cdr A) B))))
```

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

```
(define (app2 A B)
  (letrec ([internal
            (lambda (L)
              (if (null? L) B
                  (cons (car L)
                        (internal (cdr L)))))])
    (internal A)))
```


Cuprins

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

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

$\to \langle x; \{x \leftarrow y\}\rangle$ ⟵ închidere

expresie, context

- 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
- Dependent de punctul din program și de momentul de timp
- Legare statică — mulțimea variabilelor care conțin punctul curent conform structurii lexicale a programului

```
(let ([x 1])
  (+ x (let ([y 2])
         (* 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
- 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. Construcție prin evaluarea unei expresii `lambda`, într-un context dat
2. Legarea variabilelor *top-level*, în contextul global, prin `define`

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

$y \leftarrow 0$ $sum \leftarrow \langle \lambda x.(+\ x\ y); \bullet\rangle$	Contextul global

Pointer către contextul global

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

```
(sum (+ 1 2))
```

G: $y \leftarrow 0$, $sum \leftarrow \langle \lambda x.(+\ x\ y); \bullet\rangle$ — Contextul global

Moștenire

C: $x \leftarrow 3$

Contextul în care se evaluează corpul $(+\ x\ y)$

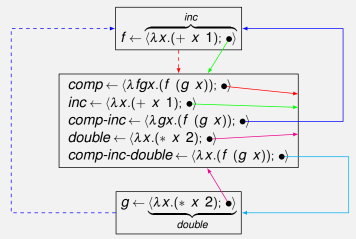
Ierarhia de contexte

- 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 156:
  - x: identificat în C
  - y: absent din C, dar identificat în G, părintele lui C

Închideri funcționale
Exemplu

```
(define comp
  (lambda (f)
    (lambda (g)
      (lambda (x)
        (f (g x))))))

(define inc (lambda (x) (+ x 1)))
(define comp-inc (comp inc))

(define double (lambda (x) (* x 2)))
(define comp-inc-double (comp-inc double))

(comp-inc-double 5)  ; 11

(define inc (lambda (x) x))
(comp-inc-double 5)  ; tot 11!
```


Închideri funcționale
Explicația exemplului



Rezumat

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

Partea VI

Întârzierea evaluării

Cuprins

Cuprins

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

```
(define (prod x y)
  (if x (* y (+ y 1)) 0))

(define (test x)
  (let ([y 5])
    (prod x (begin (display "y") y))))

(test #f)  ; y  0
(test #t)  ; y 30
```

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

Varianta 2

`quote` & `eval`

```
(define (prod x y)
  (if x (* (eval y) (+ (eval y) 1)) 0))

(define (test x)
  (let ([y 5])
    (prod x '(begin (display "y") y))))

(test #f)  ; 0
(test #t)  ; y y: undefined
```

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

Varianta 3

Închideri funcționale

```
(define (prod x y)
  (if x (* (y) (+ (y) 1)) 0))

(define (test x)
  (let ([y 5])
    (prod x (lambda ()
              (begin (display "y") y)))))

(test #f)  ; 0
(test #t)  ; yy 30
```

- Comportament corect: `y` evaluat la cerere
- `x = #t` — `y` evaluat de 2 ori, ineficient

Varianta 4

Promisiuni: `delay` & `force`

```
(define (prod x y)
  (if x (* (force y) (+ (force y) 1)) 0))

(define (test x)
  (let ([y 5])
    (prod x (delay (begin (display "y") y)))))

(test #f)  ; 0
(test #t)  ; y 30
```

Comportament corect: `y` evaluat la cerere, o singură dată — evaluare leneșă

Promisiuni

Descriere

- Rezultatul încă neevaluat al unei expresii
- Exemplu: `(delay (* 5 6))`
- Valori de prim rang în limbaj (v. slide-ul 96)
- `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

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

Expresii cu evaluare întârziată, ca entități autonome: utilizare
Interfață: `pack`, `unpack`
Expresii cu evaluare întârziată, ca închideri funcționale sau promisiuni: implementare

- 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

```
(define-macro (pack expr)
  `(delay ,expr))          ; sau `(lambda () ,expr)

(define unpack force)  ; sau  (lambda (p) (p))

(define (prod x y)
  (if x (* (unpack y) (+ (unpack y) 1)) 0))

(define (test x)
  (let ([y 5])
    (prod x (pack (begin (display "y") y)))))
```


Cuprins

Motivație

Să se determine suma numerelor pare din intervalul $[a, b]$.

```
(define (even-sum-iter a b)
  (let iter ([n a]
             [sum 0])
    (cond [(> n b) sum]
          [(even? n) (iter (+ n 1) (+ sum n))]
          [else (iter (+ n 1) sum)])))

(define (even-sum-lists a b)
    (foldl + 0 (filter even? (interval 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`)
- Cum îmbinăm avantajele celor două abordări?

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

```
(define-macro (stream-cons head tail)
  `(cons ,head (pack ,tail)))

(define stream-car car)

(define stream-cdr (compose unpack cdr))

(define stream-null '())

(define stream-null? 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

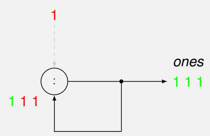
Interfață: `pack`, `unpack`

Expresii cu evaluare întârziată,
ca închideri funcționale sau promisiuni:
implementare

Fluxul de numere 1

Implementare

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

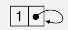


- Linii continue: fluxuri
- Linii întrerupte: intrări scalare, utilizate o singură dată
- Cifre: intrări / ieșiri

Fluxul de numere 1

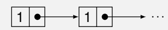
Utilizarea memoriei

Atât cu închideri, cât și cu promisiuni, extinderea se realizează în spațiu constant:

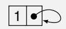


Alternativ: `(define ones (pack (cons 1 ones)))`

- închideri:



- promisiuni:



Fluxul numerelor naturale

Formulare explicită

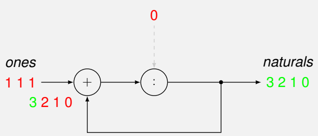
```
(define (naturals-from n)
  (stream-cons n (naturals-from (+ n 1))))

(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

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

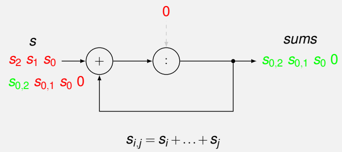


Fluxul numerelor pare

```
(define even-naturals-1
  (stream-filter even? naturals))

(define even-naturals-2
  (stream-zip-with + naturals naturals))
```


Fluxul sumelor parțiale ale altui flux

```
(define (sums s)
  (letrec ([out (stream-cons
                 0
                 (stream-zip-with + s out))])
    out))
```

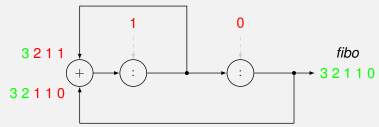


$$s_{i,j} = s_i + \ldots + s_j$$

Fluxul numerelor Fibonacci

Formulare implicită

```
(define fibo
  (stream-cons 0
   (stream-cons 1
    (stream-zip-with +
                     fibo
                     (stream-cdr fibo)))))
```



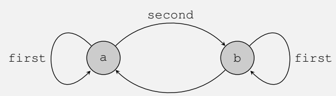
Fluxul numerelor prime I

- 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

```
(define (sieve s)
  (if (stream-null? s) s
      (stream-cons
       (stream-car s)
       (sieve
        (stream-filter
         (lambda (n)
           (not (zero? (remainder
                        n
                        (stream-car s)))))
         (stream-cdr s))))))

(define primes (sieve (naturals-from 2)))
```


Grafuri ciclice I



Fiecare nod conține:

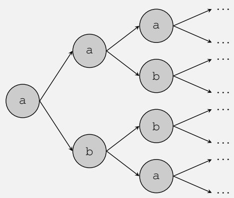
- cheia: `key`
- legăturile către două noduri: `first`, `second`

Grafuri ciclice II

```
(define-macro (node key fst snd)
  `(pack (list ,key ,fst ,snd)))

(define key car)
(define fst (compose unpack cadr))
(define snd (compose unpack caddr))

(define graph
  (letrec ([a (node 'a a b)]
           [b (node 'b b a)])
    (unpack a)))

(eq? graph (fst graph)) ; similar cu == din Java
; #f pentru inchideri, #t pentru promisiuni
```

Grafuri ciclice III

- Explorarea grafului în cazul închiderilor: nodurile sunt regenerate la fiecare vizitare



Cuprins

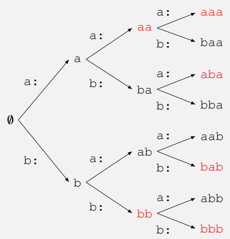
Spațiul stărilor unei probleme

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

Căutare în lățime

```
(define (breadth-search-goal init expand goal?)
  (let search ([states (list init)])
    (if (null? states) '()
        (let ([state  (car states)]
              [states (cdr states)])
          (if (goal? state) state
              (search (append states
                              (expand
                               state))))))))
```

- Generarea unei singure soluții
- Cum le obținem pe celelalte, mai ales dacă spațiul este infinit?

Căutare leneșă în lățime I
Fluxul stărilor soluție

```
(define (lazy-breadth-search init expand)
  (let search
    ([states (stream-cons init stream-null)])
    (if (stream-null? states) states
        (let ([state  (stream-car states)]
              [states (stream-cdr states)])
          (stream-cons
           state
           (search (stream-append
                    states
                    (expand state))))))))

(define (lazy-breadth-search-goal
         init expand goal?)
  (stream-filter goal?
```


Căutare leneșă în lățime II
Fluxul stărilor soluție

```
                 (lazy-breadth-search init
                                      expand)))
```

- La nivel înalt, conceptual: separare între explorarea spațiului și identificarea stărilor soluție
- La nivelul scăzut, al instrucțiunilor: întrepătrunderea celor două aspecte

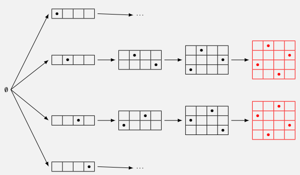
Aplicații

- 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.*
- 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

Cuprins

Paralelă între limbaje

Criteriu	Scheme	Haskell
Funcții	*Curried* / *uncurried*	*Curried*
Evaluare	Aplicativă	Leneșă
Tipare	Dinamică, tare	Statică, tare
Legarea variabilelor	Locale → statică, *top-level* → dinamică	Statică

Funcții

- *Curried*
- Aplicabile asupra oricâtor parametri la un moment dat

```
add1 x y = x + y
add2     = \x y -> x + y
add3     = \x -> \y -> x + y

result   = add1 1 2  -- sau ((add1 1) 2)
inc      = add1 1    -- functie
```


Funcții și operatori

- Aplicabilitatea parțială a operatorilor infixați (secțiuni)
- Transformări operator→funcție și funcție→operator

```
add4    = (+)

result1 = (+) 1 2      -- operator ca functie
result2 = 1 `add4` 2   -- functie ca operator

inc1    = (1 +)        -- sectiuni
inc2    = (+ 1)
inc3    = (1 `add4`)
inc4    = (`add4` 1)
```


Pattern matching

Definirea comportamentului funcțiilor pornind de la structura parametrilor — traducerea axiomelor TDA

```
add5 0 y           = y        -- add5 1 2
add5 (x + 1) y     = 1 + add5 x y

listSum []         = 0        -- sumList [1, 2, 3]
listSum (hd : tl)  = hd + listSum tl

pairSum (x, y)     = x + y   -- sumPair (1, 2)

wackySum (x, y, z@(hd : _)) =    -- wackySum
    x + y + hd + listSum z       --  (1, 2, [3, 4, 5])
```


List comprehensions

Definirea listelor prin proprietățile elementelor, similar unei specificații matematice

```
squares lst   = [ x * x | x <- lst ]

qSort []      = []
qSort (h : t) =   qSort [ x | x <- t, x <= h ]
              ++  [h]
              ++  qSort [ x | x <- t, x > h ]

interval      = [ 0 .. 10 ]
evenInterval  = [ 0, 2 .. 10 ]
naturals      = [ 0 .. ]
```


Cuprins

Evaluare

- Evaluare leneșă: parametri evaluați la cerere, cel mult o dată, eventual parțial, în cazul obiectelor structurate
- Funcții nestricte!

```
f (x, y) z = x + x

f (2 + 3, 3 + 5) (5 + 8)
→  (2 + 3) + (2 + 3)
→  5 + 5    -- reutilizam rezultatul primei evauari
→  10
```


Pași în aplicarea funcțiilor I

```
front (x : y : zs)  =   x + y
front [x]           =   x

notNil []           =   False
notNil (_ : _)      =   True

f m n
    | notNil xs     =   front xs
    | otherwise     =   n
  where
    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

```
f 3 5
??  notNil xs
??    where
??      xs = [3 .. 5]
??      → 3 : [4 .. 5]
??  → notNil (3 : [4 .. 5])
??  → True
→ front xs
      where
        xs = 3 : [4 .. 5]
        → 3 : 4 : [5]
→ front (3 : 4 : [5])
→ 3 + 4
→ 7
```


Consecințe

- Evaluarea parțială a obiectelor structurate (liste etc.)
- Liste, implicit, ca fluxuri!

```
ones           = 1 : ones

naturalsFrom n = n : (naturalsFrom (n + 1))
naturals1      = naturalsFrom 0
naturals2      = 0 : (zipWith (+) ones naturals2)

evenNaturals1  = filter even naturals1
evenNaturals2  = zipWith (+) naturals1 naturals2

fibo           = 0 : 1 :
                 (zipWith (+) fibo (tail fibo))
```


Cuprins

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

```
5               :: Integer
'a'             :: Char
inc             :: Integer -> Integer
[1,2,3]         :: [Integer]
(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

```
-- Constructorul de tip functie: ->
(-> Bool Bool) ⇒ Bool -> Bool
(-> Bool (Bool -> Bool)) ⇒ Bool -> (Bool -> Bool)

-- Constructorul de tip lista: []
([] Bool) ⇒ [Bool]
([] [Bool]) ⇒ [[Bool]]

-- Constructorul de tip tuplu: (,...,)
((,) Bool Char) ⇒ (Bool, Char)
((,,) Bool ((,) Char [Bool]) Bool)
    ⇒ (Bool, (Char, [Bool]), Bool)
```


Tipurile funcțiilor

Constructorul "`->`" asociativ la dreapta:

```
Integer -> Integer -> Integer
    ≡ Integer -> (Integer -> Integer)
```

```
add6      :: Integer -> Integer -> Integer
add6 x y  =  x + y

f         :: (Integer -> Integer) -> Integer
f g       =  (g 3) + 1

idd       :: a -> a   -- functie polimorfica
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

```
data Natural
    = Zero
    | Succ Natural
    deriving (Show, Eq)

unu                 =   Succ Zero
doi                 =   Succ unu

addNat Zero n       =   n
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*

```
Zero :: Natural
Succ :: Natural -> Natural
```


Constructorul de tip `Pair` I

Definit de utilizator

```
data Pair a b
    = P a b
    deriving (Show, Eq)

pair1           =   P 2 True
pair2           =   P 1 pair1

myFst (P x y)   =   x
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

```
P :: a -> b -> Pair a b
```


Uniformitatea reprezentării tipurilor

```
data Integer = ... | -2 | -1 | 0 | 1 | 2 | ...

data Char = 'a' | 'b' | 'c' | ...

data [a] = [] | a : [a]

data (a, b) = (a, b)
```


Cuprins

Sinteza de tip

- Definiție: determinarea automată a tipului unei expresii, pe baza unor reguli precise
- Adnotările explicite de tip, deși posibile, nenecesare 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

- Formă generală:

$$\frac{\texttt{premisa-1 ... premisa-m}}{\texttt{concluzie-1 ... concluzie-n}}\quad\texttt{(nume)}$$

- Funcție:

$$\frac{\texttt{Var :: a}\qquad\texttt{Expr :: b}}{\backslash\texttt{Var -> Expr :: a -> b}}\quad\texttt{(TLambda)}$$

- Aplicație:

$$\frac{\texttt{Expr1 :: a -> b}\qquad\texttt{Expr2 :: a}}{\texttt{(Expr1 Expr2) :: b}}\quad\texttt{(TApp)}$$

Reguli simplificate de sinteză de tip II

- Operatorul `+`:

$$\frac{\texttt{Expr1 :: Int}\qquad\texttt{Expr2 :: Int}}{\texttt{Expr1 + Expr2 :: Int}}\quad\texttt{(T+)}$$

- Literali întregi:

$$\frac{}{\texttt{0, 1, 2, ... :: Int}}\quad\texttt{(TInt)}$$

Exemple de sinteză de tip I

`f g = (g 3) + 1`

$$\frac{\texttt{g :: a}\qquad\texttt{(g 3) + 1 :: b}}{\texttt{f :: a -> b}}\quad\texttt{(TLambda)}$$

$$\frac{\texttt{(g 3) :: Int}\qquad\texttt{1 :: Int}}{\texttt{(g 3) + 1 :: Int}}\quad\texttt{(T+, TInt)}$$

`b = Int`

$$\frac{\texttt{g :: c -> d}\qquad\texttt{3 :: c}}{\texttt{(g 3) :: d}}\quad\texttt{(TApp)}$$

`a = c -> d, c = Int, d = Int`

`f :: (Int -> Int) -> Int`

Exemple de sinteză de tip II

`fix f = f (fix f)`

$$\frac{\texttt{f :: a}\qquad\texttt{f (fix f) :: b}}{\texttt{fix :: a -> b}}\quad\texttt{(TLambda)}$$

$$\frac{\texttt{f :: c -> d}\qquad\texttt{(fix f) :: c}}{\texttt{f (fix f) :: d}}\quad\texttt{(TApp)}$$

`a = c -> d, b = d`

$$\frac{\texttt{fix :: e -> g}\qquad\texttt{f :: e}}{\texttt{(fix f) :: g}}\quad\texttt{(TApp)}$$

`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)`

$$\frac{\texttt{x :: a}\qquad\texttt{(x x) :: b}}{\texttt{f :: a -> b}}\quad\texttt{(TLambda)}$$

$$\frac{\texttt{x :: c -> d}\qquad\texttt{x :: c}}{\texttt{(x x) :: d}}\quad\texttt{(TApp)}$$

Ecuația `c -> 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 ← Int, b ← Int, c ← [Int]}`
  - `S2 = {a ← Int, c ← [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: `\x -> x`
- Tipuri corecte:
  - `Int -> Int`
  - `Bool -> Bool`
  - `a -> 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:

```
sum (map (^2) [1 .. n])
→ sum (map (^2) 1 : [2 .. n])
→ sum (1^2 : (map (^2) [2 .. n]))
→ 1^2 + sum (map (^2) [2 .. n])
→ 1 + sum (map (^2) [2 .. n])
...
→ 1 + (4 + sum (map (^2) [3 .. n]))
...
→ 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):

```
ins x []          = [x]
ins x (h : t)
    | x <= h      = x : h : t
    | otherwise   = h : (ins x t)

isort []          = []
isort (h : t)     = ins h (isort t)

minList l = head (isort l)
```


Elementul minim al unei liste II

```
    minList [3, 2, 1]
    = head (isort [3, 2, 1])
    = head (isort (3 : [2, 1]))
    = head (ins 3 (isort [2, 1]))
    = head (ins 3 (isort (2 : [1])))
    = head (ins 3 (ins 2 (isort [1])))
    = head (ins 3 (ins 2 (isort (1 : []))))
    = head (ins 3 (ins 2 (ins 1 (isort []))))
    = head (ins 3 (ins 2 (ins 1 [])))
    = head (ins 3 (ins 2 (1 : [])))
    = head (ins 3 (1 : ins 2 []))
    = head (1 : (ins 3 (ins 2 [])))
    = 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 ⇔ existența elementelor în mulțimea tuturor căilor dintre cele două noduri (Thompson, 1999):

```
routes source dest graph explored
    | source == dest = [[source]]
    | otherwise      = [ source : path
                       | neighbor <- neighbors source
                         graph \\ explored
                       , path <- routes neighbor dest
                         graph (source : explored)
                       ]

accessible source dest graph =
    (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

Bibliotecă de parsare (Thompson, 1999)

Bibliografie

Thompson, S. (1999). *Haskell: The Craft of Functional Programming*. Ediția a doua. Addison-Wesley.

Partea IX

Clase în Haskell

Cuprins

Cuprins

Motivație

Să se definească operația `show`, capabilă să producă reprezentarea oricărui obiect ca șir de caractere. Comportamentul este specific fiecărui tip.

```
show 3 → "3"
show True → "True"
show 'a' → "'a'"
show "a" → "\"a\""
```


Varianta 1 I

Funcții dedicate fiecărui tip

```
show4Bool True  = "True"
show4Bool False = "False"

show4Char c     = "'" ++ [c] ++ "'"

show4String s   = "\"" ++ s ++ "\""
```


Varianta 1 II

Funcții dedicate fiecărui tip

- Funcția `showNewLine`, care adaugă caracterul "linie nouă" la reprezentarea ca șir:

```
showNewLine x = (show... x) ++ "\n"
```

- `showNewLine` nu poate fi polimorfică → `showNewLine4Bool`, `showNewLine4Char` etc.
- Alternativ, trimiterea ca parametru a funcției `show*`, corespunzătoare:

```
showNewLine sh x = (sh x) ++ "\n"
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`:

```
class Show a where
    show :: a -> String
    ...
```

- Precizarea aderenței unui tip la această mulțime:

```
instance Show Bool where
    show True  = "True"
    show False = "False"

instance Show Char where
    show c = "'" ++ [c] ++ "'"
```

- Funcția `showNewLine` polimorfică!

```
showNewLine x = (show x) ++ "\n"
```


Varianta 2 II

Supraîncărcarea funcției

- Ce tip au funcțiile `show`, respectiv `showNewLine`?

```
show        :: Show a => a -> String
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:

```
instance (Show a, Show b) => Show (a, b) where
    show (x, y) =  "("  ++ (show x)
                ++ ",␣" ++ (show y)
                ++ ")"
```

- "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

```
class Eq a => Ord a where
    (<), (<=), (>=), (>) :: a -> a -> Bool
    ...
```

- Contexte utilzabile 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	POO
Mulțimi de tipuri	Mulțimi de obiecte: *tipuri*
Instanțierea claselor de către tipuri	Implementarea interfețelor de către clase
Implementarea operațiilor în afara definiției tipului	Implementarea operațiilor în cadrul definiției tipului

Clase Haskell ~ Interfețe Java

Cuprins

`invert` I

Fie constructorii de tip:

```
data Pair a = P a a

data NestedList a
    = Atom a
    | 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

```
class Invert a where
    invert :: a -> a
    invert = id

instance Invert (Pair a) where
    invert (P x y) = P y x

instance Invert a => Invert (NestedList a) where
    invert (Atom x)  = Atom (invert x)
    invert (List x)  = List $ reverse $ map invert x

instance Invert a => Invert [a] where
    invert lst = reverse $ map invert lst
```

Necesitatea contextului, în cazul tipurilor `[a]` și `NestedList a`, pentru inversarea elementelor înselor

`contents` I

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.

```
class Container a where
    contents :: a -> [??]
```

- `a` este tipul unui container, ca `NestedList b`
- Elementele listei întoarse sunt cele din container
- Cum precizăm tipul acestora, `b`?

`contents` II

```
class Container a where
    contents :: a -> [a]

instance Container [a] where
    contents = id
```

- Conform definiției clasei:

```
contents :: Container [a] => [a] -> [[a]]
```

- Conform supraîncărcării funcției (`id`):

```
contents :: Container [a] => [a] -> [a]
```

- Ecuația `[a] = [[a]]` nu are soluție — eroare!

`contents` III

```
class Container a where
    contents :: a -> [b]

instance Container [a] where
    contents = id
```

- Conform definiției clasei:

```
contents :: Container [a] => [a] -> [b]
```

- Conform supraîncărcării funcției (`id`):

```
contents :: Container [a] => [a] -> [a]
```

- Ecuația `[a] = [b]` are soluție pentru `a = b`
- Dar, `[a] -> [a]` insuficient de general în raport cu `[a] -> [b]` — eroare!

`contents` IV

Soluție: clasa primește constructorul de tip, și nu tipul container propriu-zis

```
class Container t where
    contents :: t a -> [a]

instance Container Pair where  -- nu (Pair a)!
    contents (P x y) = [x, y]

instance Container NestedList where
    contents (Atom x) = [x]
    contents (List l) = concatMap contents l

instance Container [] where
    contents = id
```


Contexte I

```
fun1 :: Eq a => a -> a -> a -> a
fun1 x y z = if x == y then x else z

fun2 :: (Container a, Invert (a b), Eq (a b))
     => (a b) -> (a b) -> [b]
fun2 x y = if (invert x) == (invert y)
           then contents x
           else contents y

fun3 :: Invert a => [a] -> [a] -> [a]
fun3 x y = (invert x) ++ (invert y)

fun4 :: Ord a => a -> a -> a -> a
fun4 x y z = if x == y
             then z
             else if x > y
                  then x
                  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 signatura unei funcții, în termenii aderenței la diverse clase

Partea X

Paradigma funcțională vs. paradigma imperativă

Cuprins

Cuprins

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`:
  - 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 ≡ 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:
  - `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 ⇒ opacitate referențială!

Funcții transparente referențial

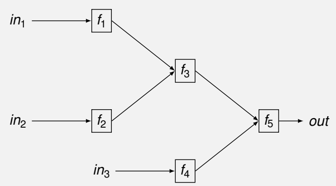
Funcție transparentă referențial:
rezultatul întors depinde exclusiv de parametri

```
int transparent(int x) {
    return x + 1;
}
```

```
int g = 0;

int opaque(int x) {
    return x + ++g;
}

// opaque(3) != opaque(3)
```

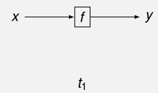
- Funcții transparente: `log`, `sin` etc.
- Funcții opace: `time`, `read` etc.

Înlănțuirea funcțiilor



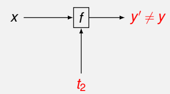
Calcul fără stare

Dependența ieșirii de intrare, nu și de timp

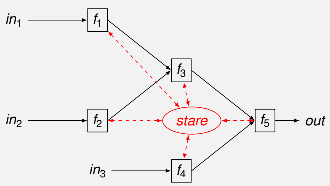


Calcul cu stare

Dependența ieșirii de intrare, și de timp



Calcul cu stare



Stare = mulțimea valorilor variabilelor, la un anumit moment, ce pot influența rezultatul evaluării aceleiași expresii.

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

Explicitarea sensului programelor

procedure MinList(L, n)
    $min \leftarrow L[1]$
    $i \leftarrow 2$
    while $i \leq n$ **do**
        if $L[i] < min$ **then**
            $min \leftarrow L[i]$
        end if
        $i \leftarrow i + 1$
    end while
    return *min*
end procedure

```
minList [h]     = h
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

Alte aspecte

- Funcționale ca structuri de control
- Tipuri algebrice
- Polimorfism

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

Cuprins

Un prim exemplu

```
% constante -> litera mica
parent(andrei, bogdan).
parent(andrei, bianca).
parent(bogdan, cristi).

% variabile -> litera mare
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

- $true \Rightarrow parent(andrei, bogdan)$
- $true \Rightarrow parent(andrei, bianca)$
- $true \Rightarrow parent(bogdan, cristi)$
- $\forall x.\forall y.\forall z.$
  $(parent(x,z) \wedge parent(z,y) \Rightarrow grandparent(x,y))$

Interogări

```
?- parent(andrei, bogdan).
true .

?- parent(andrei, cristi).
false.

?- parent(andrei, X).
X = bogdan ;
X = bianca.

?- grandparent(X, Y).
X = andrei,
Y = cristi ;
false.
```

- "." → oprire după primul răspuns
- ";" → solicitarea următorului răspuns

Concatenarea a două liste

```
% append(L1, L2, Res)
append([], L, L).
append([H|T], L, [H|Res]) :- append(T, L, Res).
```

Calcul

```
?- append([1], [2], Res).
Res = [1, 2].
```

Generare

```
?- append(L1, L2, [1, 2]).
L1 = [],
L2 = [1, 2] ;
L1 = [1],
L2 = [2] ;
L1 = [1, 2],
L2 = [] ;
false.
```

Estomparea granițelor dintre "intrare" și "ieșire"

Cuprins

Exemplul genealogic I

```
                  S = ∅
                  G = {gp(X, Y)}
                        ↓
                gp(X1, Y1) :-
                p(X1, Z1), p(Z1, Y1)
                        ↓
             S = {X = X1, Y = Y1}
             G = {p(X1, Z1), p(Z1, Y1)}
          ↙             ↓              ↘
p(andrei, bogdan)  p(andrei, bianca)  p(bogdan, cristi)
        ↓                 ↓                  ↓
       ...               ...                ...
```


Exemplul genealogic II

```
                      ...
                       ↓
               p(andrei, bogdan)
                       ↓
   S = {X = X1, Y = Y1, X1 = andrei, Z1 = bogdan}
   G = {p(bogdan, Y1)}
                       ↓
               p(bogdan, cristi)
                       ↓
S = {X = X1, Y = Y1, X1 = andrei, Z1 = bogdan, Y1 = cristi}
G = ∅
                       ↓
                    succes
```


Exemplul genealogic III

```
                      ...
                       ↓
               p(andrei, bianca)
                       ↓
   S = {X = X1, Y = Y1, X1 = andrei, Z1 = bianca}
   G = {p(bianca, Y1)}
                       ↓
                     eșec
```


Exemplul genealogic IV

```
                      ...
                       ↓
               p(bogdan, cristi)
                       ↓
   S = {X = X1, Y = Y1, X1 = bogdan, Z1 = cristi}
   G = {p(cristi, Y1)}
                       ↓
                     eșec
```


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

Minimul a două numere I

```
min(X, Y, M) :- X =< Y, M is X.
min(X, Y, M) :- X > Y,  M is Y.

min2(X, Y, M) :- X =< Y, M = X.
min2(X, Y, M) :- X > Y,  M = Y.

% Echivalent cu min2.
min3(X, Y, X) :- X =< Y.
min3(X, Y, Y) :- X > Y.
```


Minimul a două numere II

```
?- min(1+2, 3+4, M).
M = 3 ;
false.

?- min(3+4, 1+2, M).
M = 3.

?- min2(1+2, 3+4, M).
M = 1+2 ;
false.

?- min2(3+4, 1+2, M).
M = 1+2.
```


Minimul a două numere III

Condiții mutual exclusive: `X =< Y` și `X > Y` — cum putem elimina redundanța?

```
min4(X, Y, X) :- X =< Y.
min4(X, Y, Y).
```

```
?- min4(1+2, 3+4, M).
M = 1+2 ;
M = 3+4.
```

Greșit!

Minimul a două numere IV

Soluție: oprirea recursivității după prima satisfacere a scopului

```
min5(X, Y, X) :- X =< Y, !.
min5(X, Y, Y).
```

```
?- min5(1+2, 3+4, M).
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

```
girl(mary).
girl(ann).

boy(john).
boy(bill).

pair(X, Y) :- girl(X), boy(Y).
pair(bella, harry).

pair2(X, Y) :- girl(X), !, boy(Y).
pair2(bella, harry).
```

Backtracking doar la dreapta operatorului

Operatorul *cut* III

```
?- pair(X, Y).
X = mary,
Y = john ;
X = mary,
Y = bill ;
X = ann,
Y = john ;
X = ann,
Y = bill ;
X = bella,
Y = harry.
```

```
?- pair2(X, Y).
X = mary,
Y = john ;
X = mary,
Y = bill.
```


Cuprins

Programare logică

- Reprezentare simbolică
- Stil declarativ
- Separarea datelor de procesul de inferență, incorporat 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:

$$A_1 \wedge \ldots \wedge A_n \Rightarrow A \qquad \text{(Regulă)}$$
$$true \Rightarrow B \qquad \text{(Axiomă)}$$

- 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

```
nott(P) :- P, !, fail.
nott(P).
```

- `P` → 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

Cuprins

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:
  - sintaxă: modalitatea de construcție a expresiilor
  - semantică: semnificația expresiilor construite
- Deducerea de noi proprietăți, pe baza celor existente

Cuprins

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

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: $p, q, r, \ldots$
- Negații: $\neg\alpha$
- 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

- Negație:

$$(\neg\alpha)^I = \begin{cases} true & \text{dacă } \alpha^I = false \\ false & \text{altfel} \end{cases}$$

- Conjuncție:

$$(\alpha \wedge \beta)^I = \begin{cases} true & \text{dacă } \alpha^I = true \text{ și } \beta^I = true \\ false & \text{altfel} \end{cases}$$

- Disjuncție:

$$(\alpha \vee \beta)^I = \begin{cases} false & \text{dacă } \alpha^I = false \text{ și } \beta^I = false \\ true & \text{altfel} \end{cases}$$

Semantică IV

- Implicație:

$$(\alpha \Rightarrow \beta)^I = \begin{cases} false & \text{dacă } \alpha^I = true \text{ și } \beta^I = false \\ true & \text{altfel} \end{cases}$$

- Echivalență:

$$(\alpha \Leftrightarrow \beta)^I = \begin{cases} true & \text{dacă } \alpha^I = \beta^I \\ false & \text{altfel} \end{cases}$$

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

Satisfiabilitate

- *Satisfiabilitate* = proprietatea unei propoziții adevărate în cel puțin o interpretare
- Metoda tabelei de adevăr:

p	q	r	$(p \wedge q) \vee (q \Rightarrow r)$
true	*true*	*true*	*true*
true	*true*	*false*	*true*
true	*false*	*true*	*true*
true	*false*	*false*	*true*
false	*true*	*true*	*true*
false	*true*	*false*	*false*
false	*false*	*true*	*false*
false	*false*	*false*	*false*

Validitate

- *Validitate* = proprietatea unei propoziții adevărate în toate interpretările (*tautologie*)
- Exemplu: $p \vee \neg p$
- Verificabilă prin metoda tabelei de adevăr

Nesatisfiabilitate

- *Nesatisfiabilitate* = proprietatea unei propoziții false în toate interpretările (*contradicție*)
- Exemplu: $p \Leftrightarrow \neg p$
- Verificabilă prin metoda tabelei de adevăr

Cuprins

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 tabelei 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	$p \Rightarrow q$
true	*true*	*true*
true	*false*	*false*
false	*true*	*true*
false	*false*	*true*

Singura intrare în care ambele premise, p și $p \Rightarrow q$, sunt adevărate, precizează și adevărul concluziei, q.

Formulări echivalente ale derivabilității

- $\{\phi_1, \ldots, \phi_n\} \models \phi$
- Propoziția $\phi_1 \wedge \ldots \wedge \phi_n \Rightarrow \phi$ este validă
- Propoziția $\phi_1 \wedge \ldots \wedge \phi_n \wedge \neg\phi$ este nesatisfiabilă

Cuprins

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 tabelei de adevăr
- Alternativ, metode sintactice, care manipulează doar reprezentarea simbolică

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
- *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:

$$\Delta \vdash_{inf} \phi \Rightarrow \Delta \models \phi$$

- *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 $\{p \Rightarrow q, q \Rightarrow r\} \models p \Rightarrow r$
- 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

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):

$$\phi_1 \vee \ldots \vee \phi_n \ \rightarrow \ \{\phi_1, \ldots, \phi_n\}$$
$$\phi_1 \wedge \ldots \wedge \phi_n \ \rightarrow \ \{\phi_1\}, \ldots, \{\phi_n\}$$

Transformarea în formă clauzală II

- Exemplu: $p \wedge (q \Rightarrow r)$

I	$p \wedge (\neg q \vee r)$
C	$\{p\}, \{\neg q, r\}$

- Exemplu: $\neg(p \wedge (q \Rightarrow r))$

I	$\neg(p \wedge (\neg q \vee r))$
N	$\neg p \vee \neg(\neg q \vee r)$
N	$\neg p \vee (q \wedge \neg r)$
D	$(\neg p \vee q) \wedge (\neg p \vee \neg r)$
C	$\{\neg p, q\}, \{\neg p, \neg r\}$

Rezoluție I

- Ideea:

$$\begin{array}{c}\{p,q\}\\ \{\neg p,r\}\\ \hline \{q,r\}\end{array}$$

- "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:

$$\begin{array}{c}\{p_1,\ldots,r,\ldots,p_m\}\\ \{q_1,\ldots,\neg r,\ldots,q_n\}\\ \hline \{p_1,\ldots,p_m,q_1,\ldots,q_n\}\end{array}$$

Rezoluție II

- Rezolvent vid — contradicție între premise:

$$\begin{array}{c}\{\neg p\}\\ \{p\}\\ \hline \{\}\end{array}$$

- Mai mult de 2 rezolvenți posibili — se alege doar unul:

$$\begin{array}{c}\{p,q\}\\ \{\neg p,\neg q\}\\ \hline \{p,\neg p\}\\ \{q,\neg q\}\end{array}$$

Rezoluție III

- *Modus Ponens* — caz particular al rezoluției:

$$\begin{array}{c}p\Rightarrow q\\ p\\ \hline q\end{array}\qquad\begin{array}{c}\{\neg p,q\}\\ \{p\}\\ \hline \{q\}\end{array}$$

- *Modus Tollens* — caz particular al rezoluției:

$$\begin{array}{c}p\Rightarrow q\\ \neg q\\ \hline \neg p\end{array}\qquad\begin{array}{c}\{\neg p,q\}\\ \{\neg q\}\\ \hline \{\neg p\}\end{array}$$

- Tranzitivitatea implicației:

$$\begin{array}{c}p\Rightarrow q\\ q\Rightarrow r\\ \hline p\Rightarrow r\end{array}\qquad\begin{array}{c}\{\neg p,q\}\\ \{\neg q,r\}\\ \hline \{\neg p,r\}\end{array}$$

Rezoluție IV

- Demonstrarea nesatisfiabilității — derivarea clauzei vide
- Demonstrarea derivabilității concluziei ϕ din premisele $\phi_1,\ldots,\phi_n$ — demonstrarea nesatisfiabilității propoziției $\phi_1\wedge\ldots\wedge\phi_n\wedge\neg\phi$ (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 364):

$$\Delta\models\phi\Leftrightarrow\Delta\vdash\phi$$

- Terminarea garantată a procedurii de aplicare a rezoluției: număr finit de clauze, număr finit de concluzii

Cuprins

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
  - Transparență în raport cu obiectele și relațiile referite

(Genesereth, 2010)

Cuprins

Sintaxă

Simboluri utilizate

- Constante: obiecte particulare din universul discursului: c, d, $andrei$, $bogdan$, ...
- Variabile: 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: $f(t_1,\ldots,t_n)$, unde f este un simbol funcțional n-ar și $t_1,\ldots,t_n$ sunt termeni. Exemple:
    - $succesor(4)$: succesorul lui 4
    - $+(2,x)$: suma simbolurilor 2 și x

Sintaxă II

Termeni, atomi, propoziții

- Atomi (relații): $p(t_1,\ldots,t_n)$, unde p este un predicat n-ar și $t_1,\ldots,t_n$ sunt termeni. Exemple:
  - $impar(3)$
  - $varsta(ion,20)$
  - $=(+(2,3),5)$
- Propoziții (fapte) — x variabilă, A atom, α propoziție:
  - Fals, adevărat: $\bot$, $\top$
  - Atomi: A
  - Negații: $\neg\alpha$
  - ...
  - Cuantificări: $\forall x.\alpha$, $\exists x.\alpha$

Sintaxă III

Termeni, atomi, propoziții

Exemplu: "Dan este prieten cu sora Ioanei":

$$\underbrace{prieten(\underbrace{dan}_{\text{termen}},\underbrace{sora(\overbrace{ioana}^{\text{termen}})}_{\text{termen}})}_{\text{atom/propoziție}}$$

- 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 $f^I: D^n\to D$
- Pentru fiecare predicat n-ar, p, o funcție $p^I: D^n\to\{false, true\}$.

Semantică II

- Atom:

$$(p(t_1,\ldots,t_n))^I = p^I(t_1^I,\ldots,t_n^I)$$

- Negație 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."
   $\forall x.(vrabie(x)\Rightarrow viseaza(x,malai))$
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*:
  - $\forall x.\exists y.viseaza(x, y)$: “Toți visează la ceva particular.”
  - $\exists y.\forall x.viseaza(x, y)$: “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

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. $\{prieten(x, y), \neg doctor(x)\}$
- *Clauză Horn* = clauză în care un singur literal este în formă pozitivă, e.g. $\{\neg A_1, \ldots, \neg A_n, A\}$, corespunzătoare implicației $A_1 \wedge \ldots \wedge A_n \Rightarrow A$

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 universali: înlocuirea aparițiilor variabilei cuantificate printr-o constantă:

     $$\exists x.p(x) \rightarrow p(c_x)$$
   - Dacă este precedat de cuantificatori universali: 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 impliciți (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.”

$\forall x.(\underline{\forall y.(lab(y) \Rightarrow rezolva(x, y))} \Rightarrow \underline{\exists y.apreciaza(y, x)})$

I	$\forall x.(\underline{\neg\forall y.(\neg lab(y) \vee rezolva(x, y))} \vee \underline{\exists y.apreciaza(y, x)})$
N	$\forall x.(\underline{\exists y.\neg(\neg lab(y) \vee rezolva(x, y))} \vee \underline{\exists y.apreciaza(y, x)})$
N	$\forall x.(\underline{\exists y.(lab(y) \wedge \neg rezolva(x, y))} \vee \underline{\exists y.apreciaza(y, x)})$
R	$\forall x.(\underline{\exists y.(lab(y) \wedge \neg rezolva(x, y))} \vee \underline{\exists z.apreciaza(z, x)})$
P	$\forall x.\exists y.\exists z.(\underline{(lab(y) \wedge \neg rezolva(x, y))} \vee \underline{apreciaza(z, x)})$
S	$\forall x.(\underline{(lab(f_y(x)) \wedge \neg rezolva(x, f_y(x)))} \vee \underline{apreciaza(f_z(x), x)})$
U	$\underline{(lab(f_y(x)) \wedge \neg rezolva(x, f_y(x)))} \vee \underline{apreciaza(f_z(x), x)}$
D	$\underline{(lab(f_y(x)) \vee apreciaza(f_z(x), x))}$ $\wedge \underline{(\neg rezolva(x, f_y(x)) \vee apreciaza(f_z(x), x))}$
C	$\{lab(f_y(x)), apreciaza(f_z(x), x)\}$, $\{\neg rezolva(x, f_y(x)), apreciaza(f_z(x), x)\}$

Cuprins

Motivație

- Rezoluție:

  $$\frac{\begin{array}{l}\{prieten(x, mama(y)), doctor(x)\} \\ \{\neg prieten(mama(z), z)\}\end{array}}{?}$$
- Cum aplicăm rezoluția?
- Soluția: unificare (v. sinteza de tip, slide-ul 241)
- 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 valoarea la care a fost legată (*occurrence check*)

Unificare II

- Rezoluția pentru clauze Horn:

  $$\frac{\begin{array}{l}A_1 \wedge \ldots \wedge A_m \Rightarrow A \\ B_1 \wedge \ldots \wedge A' \wedge \ldots \wedge B_n \Rightarrow B \\ unificare(A, A') = S\end{array}}{subst(S, \quad A_1 \wedge \ldots \wedge A_m \wedge B_1 \wedge \ldots \wedge B_n \Rightarrow B)}$$
- $unificare(\alpha, \beta)$: substituția sub care unifică propozițiile α și β
- $subst(S, \alpha)$: 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

Partea XIII

Mașina algoritmică Markov

Cuprins

Cuprins

Mașina algoritmică Markov

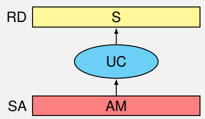
- Model de calculabilitate efectivă, echivalent cu mașina Turing și cu calculul lambda
- Principiul de funcționare: identificare de șabloane (eng. *pattern matching*) și substituție
- Fundamentul teoretic al paradigmei asociative și al limbajelor bazate pe reguli

Paradigma asociativă

- Potrivită mai ales în cazul problemelor ce nu admit o soluție precisă, algoritmică
- Codificarea cunoștințelor specifice unui domeniu și aplicarea lor într-o manieră euristică
- Descrierea proprietăților soluției, prin contrast cu pașii care trebuie realizați pentru obținerea acesteia (*ce* trebuie obținut vs. *cum*)
- Absența unui flux explicit de control, deciziile fiind determinate implicit, de cunoștințele valabile la un anumit moment — *data-driven control*

Cuprins

Structură



- Registrul de date, RD, cu secvența de simboli, S
- Unitatea de control, UC
- Spațiul de stocare a algoritmului, SA, ce conține algoritmul Markov, AM

Algoritmi

```
setDiff1(A, B); A g1; B g2;
    ag2 -> a;
    ag1 -> g1a;
    a -> .;
    -> a;
end
```

```
setDiff2(A, B); B g2;
    g2 -> ;
    -> .;
end
```


Registrul de date

- Nemărginit la dreapta
- Simboli din alfabetul $A_b \cup A_l$:
  - A_b: alfabetul de bază
  - A_l: alfabetul local / de lucru
  - $A_b \cap A_l = \emptyset$
- Șirurile inițial și final, formate doar cu simboli din A_b
- Simbolii din A_l, utilizabili exclusiv în timpul execuției
- Șirul de simboli, posibil vid

Reguli

- Unitatea de bază a unui algoritm Markov: regula asociativă, de substituție:
  ```
  șablon de identificare (LHS) ->
  șablon de substituție (RHS)
  ```
- Exemplu: `ag1c -> ac`
- Șabloanele: secvențe de simboli:
  - constante: simboli din A_b
  - variabile locale: simboli din A_l
  - variabile generice: simboli speciali, din mulțimea G, legați la simboli din A_b
- Pentru `RHS` = "." — regulă terminală, ce încheie execuția mașinii

Variabile generice

- Legate la exact un simbol
- De obicei, notate cu `g`, urmat de un indice
- Mulțimea valorilor pe care le poate lua o variabilă: domeniul variabilei, `Dom(g)`
- Utilizabile în `RHS` doar în cazul apariției în `LHS`

Algoritmi (detaliu)

- Mulțimi ordonate de reguli, îmbogățite cu declarații de
  - partiționare a mulțimii A_b
  - variabile generice
- Exemplu: eliminarea simbolilor ce aparțin mulțimii `B`:

```
setDiff1(A, B); A g1; B g2;
    ag2 -> a;
    ag1 -> g1a;
    a -> .;
    -> a;
end
```

```
setDiff2(A, B); B g2;
    g2 -> ;
    -> .;
end
```

  - $\mathtt{A}, \mathtt{B} \subseteq A_b$
  - g_1, g_2: variabile generice
  - `a`: nedeclarată, variabilă locală ($a \in A_l$)

Aplicabilitatea regulilor

Definition 44.1 (Aplicabilitatea unei reguli).
Regula $r : a_1 \ldots a_n \texttt{ -> } b_1 \ldots b_m$ este aplicabilă dacă și numai dacă există un subșir $c_1 \ldots c_n$, în RD, astfel încât, pentru orice $i = \overline{1,n}$, exact o condiție de mai jos este îndeplinită:

- $a_i \in A_b \wedge a_i = c_i$
- $a_i \in A_l \wedge a_i = c_i$
- $a_i \in G \wedge (\forall j = \overline{1,n} \bullet a_j = a_i \Rightarrow c_j \in \mathrm{Dom}(a_i) \wedge c_j = c_i)$, i.e. variabila a_i este legată la o valoare unică, obținută prin potrivirea dintre șablon și subșir.

Aplicarea regulilor

Definition 44.2 (Aplicarea unei reguli).
Aplicarea regulii $r : a_1 \ldots a_n \texttt{ -> } b_1 \ldots b_m$ asupra unui subșir $s = c_1 \ldots c_n$, în raport cu care este aplicabilă, constă în substituirea lui s prin subșirul $q_1 \ldots q_m$, calculat astfel:

- $b_i \in A_b \Rightarrow q_i = b_i$
- $b_i \in A_l \Rightarrow q_i = b_i$
- $b_i \in G \wedge (\exists j = \overline{1,n} \bullet b_i = a_j) \Rightarrow q_i = c_j$

Exemplu de aplicare

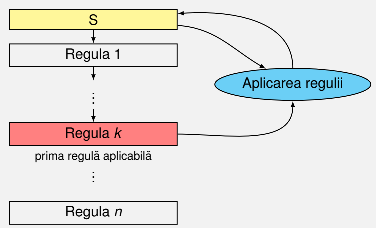
- A_b = {1, 2, 3}
- A_l = {x, y}
- $\mathrm{Dom}(g_1)$ = {2}
- $\mathrm{Dom}(g_2)$ = A_b
- s = 1111112x2y31111
- $r : 1g_1xg_1yg_2$ -> $1g_2x$

```
s   =  11111  1  2   x  2   y  3   1111
r   :         1  g1  x  g1  y  g2  -> 1g2x
s'  =  1111113x1111
```


Aplicabilitate vs. aplicare

- Aplicabilitatea potențială a
  - unei reguli pe mai multe subșiruri
  - mai multor reguli pe același subșir
- La un anumit moment, aplicarea propriu-zisă a unei singure reguli asupra unui singur subșir
- Nedeterminism inerent, ce trebuie rezolvat
- Convenție:
  - aplicarea primei reguli aplicabile, în ordinea definirii,
  - asupra celui mai din stânga subșir asupra căreia este aplicabilă

Unitatea de control I



Unitatea de control II

- Analogie cu o sită pe mai multe nivele, ce corespund regulilor
- Aplicabilitatea testată secvențial
- Etape:
  1. determinarea primei reguli aplicabile
  2. aplicarea acesteia
  3. actualizarea RD
  4. salt la pasul 1

Inversarea intrării

Ideea: mutarea pe rând, a fiecărui element în poziția corespunzătoare, prin interschimbarea elementelor adiacente

```
1 Reverse(A); A g1, g2;
2     ag1g2 -> g2ag1;
3     ag1 -> bg1;
4     abg1 -> g1a;
5     a -> .;
6     -> a;
7 end
```

$\mathtt{DOP} \xrightarrow{6} \mathtt{aDOP} \xrightarrow{2} \mathtt{OaDP} \xrightarrow{2} \mathtt{OPaD} \xrightarrow{3} \mathtt{OPbD} \xrightarrow{6} \mathtt{aOPbD}$
$\xrightarrow{2} \mathtt{PaObD} \xrightarrow{3} \mathtt{PbObD} \xrightarrow{6} \mathtt{aPbObD} \xrightarrow{3} \mathtt{bPbObD} \xrightarrow{6} \mathtt{abPbObD}$
$\xrightarrow{4} \mathtt{PabObD} \xrightarrow{4} \mathtt{POabD} \xrightarrow{4} \mathtt{PODa} \xrightarrow{5} \mathtt{.}$

Rezumat

Mașina Markov: model de calculabilitate, bazat pe identificări spontane de șabloane și pe substituție