Paradigme de Programare

As. dr. ing. Mihnea Muraru
mmihnea@gmail.com

2012–2013, semestrul 2

Cursul I

Introducere

Cuprins

Cuprins

Notare

- Teste la curs: 0,5
- Test grilă: 0,5
- Laborator: 1
- Teme: 4 (4×1)
- Examen: 4

Regulament

Vă rugăm să citiți regulamentul cu atenție!

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

Cuprins

Ce vom studia?

1. Diverse perspective conceptuale asupra noțiunii de calculabilitate efectivă: modele de calculabilitate
2. Influența perspectivei alese asupra procesului de modelare și rezolvare a problemelor: paradigme de programare
3. Mecanisme expresive, aferente paradigmelor, cu accent pe aspectul comparativ: limbaje de programare

De ce?

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

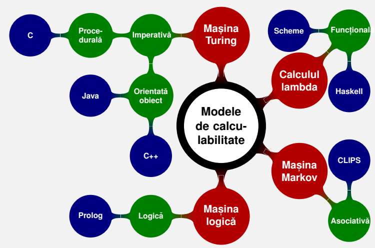
Edsger Dijkstra, *How do we tell truths that might hurt*

De ce?

Mai concret

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

Modele, paradigme, limbaje



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

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

Modelare imperativă

Varianta procedurală

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

Modelare funcțională

- Ideea:

$$\begin{aligned} minList(L) = if(&eq(length(L), 1),\\ &head(L),\\ &min(head(L), minList(tail(L)))) \end{aligned}$$

- Calculul: aplicarea funcțiilor, prin subtituție textuală
- Scheme:

```
(define minList
  (lambda (l)
    (if (= (length l) 1) (car l)
        (min (car l) (minList (cdr l))))))
```

- Haksell:

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

Modelare logică

- Axiome:
  1. $x \leq y \Rightarrow min(x,y,x)$
  2. $y < x \Rightarrow min(x,y,y)$
  3. $minList([m],m)$
  4. $minList([y|t],n) \wedge min(x,n,m) \Rightarrow minList([x,y|t],m)$
- Calculul: verificarea satisfiabilității predicatelor logice, prin legări de variabile
- 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).
```


Modelare asociativă

- Ideea:

$$minList(L) = m \in L \mid \nexists x \in L \bullet x < m$$

- Calculul: identificarea de șabloane și manipularea lor
- CLIPS:

```
(deffacts facts
    (elem 3)
    (elem 2)
    (elem 1))

(defrule minList
    (elem ?m)
    (not (elem ?x & :(< ?x ?m)))
  =>
    (assert (min ?m)))
```


Cuprins

Modelare imperativă

Varianta procedurală

```
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
```


Efecte laterale (*side effects*)

Definiție

Exemplul 4.1 (Efecte laterale).

În expresia `2 + (i = 3)`, subexpresia `(i = 3)`:

- produce valoarea `3`, conducând la rezultatul `5` al întregii expresii
- are efectul lateral de inițializare a lui `i` cu `3`

Definiția 4.2 (Efect lateral).

Modificarea adusă stării globale, de către o expresie.

Inerente în situațiile în care programul interacționează cu exteriorul — I/O!

Efecte laterale (*side effects*)

Consecințe

Exemplul 4.3 (Efecte laterale).

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

Definiție

Exemplul 4.4 (Transparență referențială).

Zeus de la greci ≡ Jupiter de 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

Definiția 4.5 (Transparență referențială).

Independența înțelesului unei propoziții în raport cu modul de desemnare a obiectelor — cazul 1.

Transparență referențială

Expresii

> *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*

Transparență referențială

Expresii

> *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*

Transparență referențială

Expresii

Exemplul 4.6 (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

Absentă în prezența efectelor laterale!

Transparență referențială

Funcții

- Funcție transparentă referențial: rezultatul întors depinde exclusiv de parametri

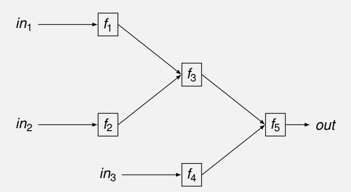
Exemplul 4.7 (Funcții (ne)transparente referențial).

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

```
int g = 0;

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

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

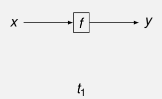
- Funcții transparente: `log`, `sin` etc.
- Funcții opace: `time`, `read` etc.

Înlănțuirea funcțiilor



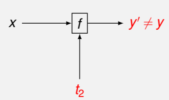
Calcul fără stare

Dependența ieșirii de intrare, nu și de timp

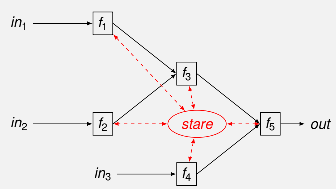


Calcul cu stare

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.

Transparență referențială

Avantaje

- Lizibilitatea codului
- Demonstrarea formală a corectitudinii programului
- Optimizare prin reordonarea instrucțiunilor de către compilator, și prin caching
- Paralelizare masivă, în urma eliminării modificărilor concurente

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

Paradigma imperativă

- Orientare spre acțiuni și efectele acestora
- „Cum" se obține soluția
- Atribuirea ca operație fundamentală
- Efecte laterale permise, compromițând transparența referențială
- Programe cu stare
- Secvențierea instrucțiunilor

Paradigma declarativă

- Accent pe formularea proprietăților soluției
- „Ce" trebuie obținut (vs. „cum" la imperativă)
- Include paradigmele:
  - funcțională
  - logică
  - asociativă

Paradigma funcțională

- Funcția văzută în sens matematic, exclusiv prin valoarea pe care o calculează
- Obținerea valorii finale prin compunerea celor intermediare
- Funcții ca valori de prim rang
- Interzicerea efectelor laterale, pentru eliminarea dependențelor implicite — modularitate sporită, la nivel de funcție!
- Promovarea transparenței referențiale, alături de avantajele acesteia
- Diminuarea importanței ordinii de evaluare
- Programe fără stare

Paradigma funcțională

It's really clear that the imperative style of programming has run its course. We're sort of done with that. However, in the declarative realm we can speculate a 10x improvement in productivity in certain domains.

Anders Hejlsberg
C# Architect

Funcții ca valori de prim rang

Definiție

Definiția 5.1 (Valoare de prim rang).
O valoare ce poate fi:
- creată dinamic
- stocată într-o variabilă
- trimisă ca parametru unei funcții
- întoarsă dintr-o funcție

Exemplul 5.2 (Compunerea a două funcții).
Funcția `compose`, ce primește, ca parametri, alte două funcții unare, `f` și `g`, și întoarce funcția obținută prin compunerea lor, `f ∘ g`.

`compose`

în C

```c
int compose(int (*f)(int), int (*g)(int), int x) {
    return (*f)((*g)(x));
}
```

În C, funcțiile nu sunt valori de prim rang.

`compose`

în Java

```java
abstract class Func<U, V> {

    public abstract V apply(U param);

    public <T> Func<T, V> compose(
            final Func<T, U> other) {
        return new Func<T, V>() {

            @Override
            public V apply(T param) {
                return Func.this.apply(
                        other.apply(param));
            }
        };
    }
}
```

În Java, funcțiile nu sunt valori de prim rang.

`compose`

în Scheme & Haskell

- Scheme:

```scheme
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))
```

- Haskell:

```haskell
compose = (.)
```

În Scheme și Haskell, funcțiile sunt valori de prim rang.

Funcții ca valori de prim rang

Aplicații parțiale

Exemplul 5.3 (Aplicații parțiale).

```scheme
(define sum-uncurried
  (lambda (x y)
    (+ x y)))

(sum-uncurried 1 2)
```

```scheme
(define sum-curried
  (lambda (x)
    (lambda (y)
      (+ x y))))

((sum-curried 1) 2)

(define sum-with-1
  (sum-curried 1))

(sum-with-1 2)
```


Funcții ca valori de prim rang

Funcții de ordin superior (funcționale)

Definiția 5.4 (Funcțională).
Funcție care ia funcții ca parametru și/sau întoarce o funcție.

Exemplul 5.5 (Funcționale).

```scheme
(define l '(1 2 3))

((compose car cdr) l) ; 2
(map list l)          ; ((1) (2) (3))
(filter odd? l)       ; (1 3)
(foldl + 0 l)         ; 6
```


Paradigmele logică și asociativă

- Accent pe formularea proprietăților soluției
- „Ce" trebuie obținut (vs. „cum" la imperativă)
- Fapte, reguli, înlănțuire înainte/înapoi
- Orientare spre date

Aplicații

- Manipulare simbolică în inteligența artificială
  - Sisteme expert
  - Demonstrarea de teoreme
- Calcul paralel
- Demonstrarea automată a corectitudinii programelor și testare, datorită modelului mai simplu de execuție
- Adoptare a paradigmei funcționale în limbajele noi: C#, F#, Python, JavaScript, Clojure (JVM), Scala
- Erlang (Ericsson): limbaj funcțional utilizat în telecomunicații, economie, comerț electronic

Cuprins

Accepții asupra limbajelor

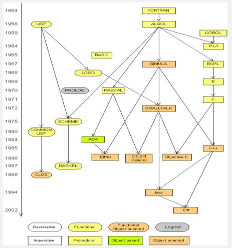
- Modalitate de exprimare a instrucțiunilor pe care calculatorul le execută
- Mai important, modalitate de exprimare a unui mod de gândire

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 potrivite pentru modelarea unei probleme particulare
- Importanța transparenței referențiale și dificultățile generate de absența acesteia, în prezența efectelor laterale

Bibliografie

Wooldridge, M. și Jennings, N. R. (1995).
Intelligent Agents: Theory and Practice.
Knowledge Engineering Review, 10:115–152.

Cursul II

Calculul Lambda

Cuprins

Cuprins

Calculul lambda

- Model de calculabilitate — Alonzo Church, 1932
- Echivalent cu Mașina Turing (v. Teza Church-Turing)
- Elementul fundamental: funcția
- Calculul: evaluarea aplicațiilor de funcții, prin substituție textuală
- Evaluare = obținerea unei valori, tot funcție!
- Absența efectelor laterale și a stării

Aplicații

- Baza teoretică a numeroase limbaje:
  - LISP
  - Scheme
  - Haskell
  - ML
  - F#
  - Clean
  - Clojure
  - Scala
  - Erlang
- Demonstrarea formală a corectitudinii programelor, datorită modelului simplu de execuție

Cuprins

λ-expresii
Definiție

Definiția 8.1 (λ-expresie).

- Variabilă: o variabilă x este o λ-expresie
- Funcție: dacă x este o variabilă și E este o λ-expresie, atunci $\lambda x.E$ este o λ-expresie, reprezentând funcția anonimă, unară, cu parametrul formal x și corpul E
- Aplicație: dacă F și A sunt λ-expresii, atunci $(F\ A)$ este o λ-expresie, reprezentând aplicația expresiei F asupra parametrului actual A

λ-expresii
Exemple

Exemplul 8.2 (λ-expresii).

- x : variabila x
- $\lambda x.x$: funcția identitate
- $\lambda x.\lambda y.x$: o funcție având altă funcție drept corp!
- $(\lambda x.x\ y)$: aplicația funcției identitate asupra parametrului actual y
- $(\lambda x.(x\ x)\ \lambda x.x)$

Intuiția din spatele evaluării aplicațiilor

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

Apariții ale variabilelor
Definiții

Definiția 8.3 (Apariție legată).
O apariție x_n a unei variabile x este legată într-o expresie E dacă:

- $E = \lambda x.F$ sau
- $E = \ldots \lambda x_n.F \ldots$ sau
- $E = \ldots \lambda x.F \ldots$ și x_n apare în F.

Definiția 8.4 (Apariție liberă).
O apariție a unei variabile este liberă într-o expresie dacă nu este legată în acea expresie.

Atenție! În raport cu o expresie dată!

Apariții ale variabilelor
Exemple

Exemplul 8.5 (Variabile legate și libere).
În expresia $E = (\lambda x.x\ x)$, evidențiem aparițiile lui x:

$$E = (\lambda x_1 . \underbrace{x_2}_{F}\ x_3).$$

- x_1, x_2 legate în E
- x_3 liberă în E
- x_2 liberă în F!
- x liberă în E și F

Apariții ale variabilelor
Exemple

Exemplul 8.6 (Variabile legate și libere).

În expresia $E = (\lambda x.\lambda z.(z\ x)\ (z\ y))$, evidențiem aparițiile lui x, y, z:

$$E = (\lambda x_1.\overbrace{\lambda z_1.(z_2\ x_2)}^{F}\ (z_3\ y_1)).$$

- x_1, x_2, z_1, z_2 legate în E
- y_1, z_3 libere în E
- z_1, z_2 legate în F
- x_2 liberă în F
- x legată în E, dar liberă în F
- y liberă în E
- z liberă în E, dar legată în F

Variabile
Definiții

Definiția 8.7 (Variabilă legată).
O variabilă este legată într-o expresie dacă toate aparițiile sale sunt legate în acea expresie.

Definiția 8.8 (Variabilă liberă).
O variabilă este liberă într-o expresie dacă nu este legată în acea expresie, i.e. dacă cel puțin o apariție a sa este liberă în acea expresie.

Definiția 8.9 (Variabilă de legare).
Parametrul formal, x, al funcției $\lambda x.E$.

Atenție! În raport cu o expresie dată!

Apariții ale variabilelor
Exemple

Exemplul 8.5 (Variabile legate și libere).

În expresia $E = (\lambda x.x\ x)$, evidențiem aparițiile lui x:

$$E = (\lambda x_1.\underbrace{x_2}_{F}\ x_3).$$

- x_1, x_2 legate în E
- x_3 liberă în E
- x_2 liberă în F!
- x liberă în E și F

Apariții ale variabilelor
Exemple

Exemplul 8.6 (Variabile legate și libere).

În expresia $E = (\lambda x.\lambda z.(z\ x)\ (z\ y))$, evidențiem aparițiile lui x, y, z:

$$E = (\lambda x_1.\overbrace{\lambda z_1.(z_2\ x_2)}^{F}\ (z_3\ y_1)).$$

- x_1, x_2, z_1, z_2 legate în E
- y_1, z_3 libere în E
- z_1, z_2 legate în F
- x_2 liberă în F
- x legată în E, dar liberă în F
- y liberă în E
- z liberă în E, dar legată în F

Determinarea variabilelor libere și legate

Variabile libere (*free variables*)
- $FV(x) = \{x\}$
- $FV(\lambda x.E) = FV(E) \setminus \{x\}$
- $FV((E_1\ E_2)) = FV(E_1) \cup FV(E_2)$

Variabile legate (*bound variables*)
- $BV(x) = \emptyset$
- $BV(\lambda x.E) = BV(E) \cup \{x\}$
- $BV((E_1\ E_2)) = BV(E_1) \setminus FV(E_2) \cup BV(E_2) \setminus FV(E_1)$

Expresii închise

Definiția 8.10 (Expresie închisă).
Expresie ce nu conține variabile libere.

Exemplul 8.11 (Expresii închise și deschise).
- $(\lambda x.x\ \lambda x.\lambda y.x)$: închisă
- $(\lambda x.x\ a)$: deschisă, deoarece a este liberă

- Variabilele libere dintr-o λ-expresie pot sta pentru alte λ-expresii, ca în $\lambda x.((+\ x)\ 1)$.
- Înaintea evaluării, o expresie trebuie adusă la forma închisă.
- Procesul de înlocuire trebuie să se termine.

Cuprins

β-reducere
Definiții

Definiția 9.1 (β-reducere).
Evaluarea expresiei $(\lambda x.E\ A)$, prin substituirea tuturor aparițiilor libere ale parametrului formal al funcției, x, din corpul acesteia, E, cu parametrul actual, A:
$(\lambda x.E\ A) \to_\beta E_{[A/x]}$.

Definiția 9.2 (β-redex).
Expresia $(\lambda x.E\ A)$.

β-reducere
Exemple

Exemplul 9.3 (β-reducere).
- $(\lambda x.x\ y) \to_\beta x_{[y/x]} \to y$
- $(\lambda x.\lambda x.x\ y) \to_\beta \lambda x.x_{[y/x]} \to \lambda x.x$
- $(\lambda x.\lambda y.x\ y) \to_\beta \lambda y.x_{[y/x]} \to \lambda y.y$

  Greșit! Variabila liberă y devine legată, schimbându-și semnificația!

β-reducere
Coliziuni

- Problemă: în expresia $(\lambda x.E\ A)$:
  - $FV(A) \cap BV(E) = \emptyset \Rightarrow$ reducere întotdeauna corectă
  - $FV(A) \cap BV(E) \neq \emptyset \Rightarrow$ reducere potențial greșită
- Soluție: redenumirea variabilelor legate din E, ce coincid cu cele libere din A.

Exemplul 9.4 (Redenumirea variabilelor legate).
$(\lambda x.\lambda y.x\ y) \to (\lambda x.\lambda z.x\ y) \to_\beta \lambda z.x_{[y/x]} \to \lambda z.y$

α-conversie
Definiție

Definiția 9.5 (α-conversie).
Redenumirea sistematică a variabilelor legate dintr-o funcție: $\lambda x.E \to_\alpha \lambda y.E_{[y/x]}$. Se impun două condiții.

Exemplul 9.6 (α-conversie).
- $\lambda x.y \to_\alpha \lambda y.y_{[y/x]} \to \lambda y.y$: Greșit!
- $\lambda x.\lambda y.x \to_\alpha \lambda y.\lambda y.x_{[y/x]} \to \lambda y.\lambda y.y$: Greșit!

Condiții:
- y nu este liberă în E
- o apariție liberă în E rămâne liberă în $E_{[y/x]}$

α-conversie
Exemple

Exemplul 9.7 (α-conversie).
- $\lambda x.(x\ y) \to_\alpha \lambda z.(z\ y)$: Corect!
- $\lambda x.\lambda x.(x\ y) \to_\alpha \lambda y.\lambda x.(x\ y)$: Greșit!
  y este liberă în $\lambda x.(x\ y)$.
- $\lambda x.\lambda y.(y\ x) \to_\alpha \lambda y.\lambda y.(y\ y)$: Greșit!
  Apariția liberă a lui x din $\lambda y.(y\ x)$ devine legată, după substituire, în $\lambda y.(y\ y)$.
- $\lambda x.\lambda y.(y\ y) \to_\alpha \lambda y.\lambda y.(y\ y)$: Corect!

Reducere
Definiții

Definiția 9.8 (Pas de reducere).
O secvență formată dintr-o posibilă α-conversie și o β-reducere, astfel încât a doua să se producă fără coliziuni: $E_1 \to E_2 \equiv E_1 \to_\alpha E_3 \to_\beta E_2$.

Definiția 9.9 (Secvență de reducere).
Succesiune de zero sau mai mulți pași de reducere: $E_1 \to^* E_2$. Reprezintă un element din închiderea reflexiv-tranzitivă a relației $\to$.

Reducere
Exemple

Exemplul 9.10 (Reduceri).
- $((\lambda x.\lambda y.(y\ x)\ y)\ \lambda x.x)$
  $\to (\lambda z.(z\ y)\ \lambda x.x)$
  $\to (\lambda x.x\ y)$
  $\to y$
- $((\lambda x.\lambda y.(y\ x)\ y)\ \lambda x.x) \to^* y$

Reducere
Proprietăți

- Pas de reducere = secvență de recucere:

$$E_1 \to E_2 \Rightarrow E_1 \to^* E_2$$

- Reflexivitate:

$$E \to^* E$$

- Tranzitivitate:

$$E_1 \to^* E_2 \wedge E_2 \to^* E_3 \Rightarrow E_1 \to^* E_3$$

Cuprins

Întrebări

1. Când se termină calculul? Se termină întotdeauna?
   - NU
2. Comportamentul depinde de secvența de reducere?
   - DA
3. Dacă se termină, obținem întotdeauna același rezultat?
   - DA
4. Dacă rezultatul este unic, cum îl obținem?
   - Reducere stânga-dreapta

Forme normale

Definiția 10.1 (Formă normală).
Formă a unei expresii care nu mai poate fi redusă,
de exemplu, care nu conține β-redecși.

Definiția 10.2 (Formă normală funcțională, FNF).
$\lambda x.F$, chiar dacă F conține β-redecși.

Exemplul 10.3 (Forme normale).
$(\lambda x.\lambda y.(x\ \ y)\ \ \lambda x.x) \rightarrow_{\text{FNF}} \lambda y.(\lambda x.x\ \ y) \rightarrow_{\text{FN}} \lambda y.y$

FNF este utilizată în programare, corpul unei funcții fiind evaluat de-abia în momentul aplicării.

Terminarea reducerii (reductibilitate)

Exemplul 10.4.
$\Omega \equiv (\lambda x.(x\ \ x)\ \ \lambda x.(x\ \ x)) \rightarrow (\lambda x.(x\ \ x)\ \ \lambda x.(x\ \ x)) \rightarrow^* \dots$
Ω nu admite o secvență de reducere, care să se termine.

Definiția 10.5 (Expresie reductibilă).
Expresie ce admite o secvență de reducere,
care se termină.

Întrebări

1. Când se termină calculul? Se termină întotdeauna?
   - NU
2. Comportamentul depinde de secvența de reducere?
   - DA
3. Dacă se termină, obținem întotdeauna același rezultat?
   - DA
4. Dacă rezultatul este unic, cum îl obținem?
   - Reducere stânga-dreapta

Secvențe de reducere

Exemplul 10.6.

$$E = (\lambda x.y\ \Omega)$$

- $\xrightarrow{1} y$
- $\xrightarrow{2} E \xrightarrow{1} y$
- $\xrightarrow{2} E \xrightarrow{2} E \xrightarrow{1} y$
- $\dots$
- $\xrightarrow{2^n 1}{}^* \ y,\ n \geq 0$
- $\xrightarrow{2^\infty}{}^* \ \dots$

- E are o secvență de reducere, care nu se termină, dar are forma normală y. E este reductibilă, Ω nu.
- Lungimea secvențelor de reducere, care se termină, este nemărginită.

Întrebări

1. Când se termină calculul? Se termină întotdeauna?
   - NU
2. Comportamentul depinde de secvența de reducere?
   - DA
3. Dacă se termină, obținem întotdeauna același rezultat?
   - DA
4. Dacă rezultatul este unic, cum îl obținem?
   - Reducere stânga-dreapta

Unicitatea formei normale

Rezultate

Teorema 10.7 (Church-Rosser / diamantului).
Dacă $E \rightarrow^* E_1$ și $E \rightarrow^* E_2$, atunci există E_3, astfel încât $E_1 \rightarrow^* E_3$ și $E_2 \rightarrow^* E_3$.

$$\begin{tikzcd} & E_1 \arrow[dr, "*"] & \\ E \arrow[ur, "*"] \arrow[dr, "*"'] & & E_3 \\ & E_2 \arrow[ur, "*"'] & \end{tikzcd}$$

Corolarul 10.8 (Unicitatea formei normale).
Dacă o expresie este reductibilă, forma ei normală este unică. Ea corespunde valorii expresiei.

Unicitatea formei normale

Exemple

Exemplul 10.9 (Unicitatea formei normale).

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

- $\rightarrow \lambda z.((\lambda x.x\ \ y)\ \ z) \rightarrow \lambda z.(y\ \ z) \rightarrow_\alpha \lambda a.(y\ \ a)$
- $\rightarrow (\lambda x.\lambda y.(x\ \ y)\ \ y) \rightarrow \lambda w.(y\ \ w) \rightarrow_\alpha \lambda a.(y\ \ a)$

- Forma normală: clasă de expresii, echivalente sub redenumiri sistematice
- Valoarea: un anumit membru al acestei clase

Întrebări

1. Când se termină calculul? Se termină întotdeauna?
   - NU
2. Comportamentul depinde de secvența de reducere?
   - DA
3. Dacă se termină, obținem întotdeauna același rezultat?
   - DA
4. Dacă rezultatul este unic, cum îl obținem?
   - Reducere stânga-dreapta

Modalități de reducere

Definiții și exemple

Definiția 10.10 (Pas de reducere stânga-dreapta).
Reducerea celui mai superficial și mai din stânga β-redex.

Exemplul 10.11 (Reducere stânga-dreapta).
$((\underline{\lambda x.x\ \ \lambda x.y})\ \ (\lambda x.(x\ \ x)\ \ \lambda x.(x\ \ x))) \rightarrow \underline{(\lambda x.y\ \ \Omega)} \rightarrow y$

Definiția 10.12 (Pas de reducere dreapta-stânga).
Reducerea celui mai adânc și mai din dreapta β-redex.

Exemplul 10.13 (Reducere dreapta-stânga).
$((\lambda x.x\ \ \lambda x.y)\ \ \underline{(\lambda x.(x\ \ x)\ \ \lambda x.(x\ \ x))}) \rightarrow (\lambda x.y\ \ \underline{\Omega}) \rightarrow \dots$

Modalități de reducere

Care este mai bună?

Teorema 10.14 (Normalizării).
Dacă o expresie este reductibilă, evaluarea stânga-dreapta a acesteia se termină.

Teorema normalizării nu garantează terminarea evaluării oricărei expresii, ci doar a celor reductibile!

Întrebări

1. Când se termină calculul? Se termină întotdeauna?
   - NU
2. Comportamentul depinde de secvența de reducere?
   - DA
3. Dacă se termină, obținem întotdeauna același rezultat?
   - DA
4. Dacă rezultatul este unic, cum îl obținem?
   - Reducere stânga-dreapta

Cuprins

Ordini de evaluare

Definiția 11.1 (Evaluare aplicativă).
Corespunde reducerii dreapta-stânga. Parametrii funcțiilor sunt evaluați înaintea aplicării funcției.

Definiția 11.2 (Funcție strictă).
Funcție cu evaluare aplicativă.

Definiția 11.3 (Evaluare normală).
Corespunde reducerii stânga-dreapta. Parametrii funcțiilor sunt evaluați la cerere.

Definiția 11.4 (Funcție nestrictă).
Funcție cu evaluare normală.

În practică I

Evaluarea aplicativă prezentă în majoritatea limbajelor, datorită eficienței — parametrii sunt evaluați o singură dată: C, Java, Scheme, PHP etc.

Exemplul 11.5 (Evaluare aplicativă în Scheme).

```
((λ (x) (+ x x)) (+ 2 3))
→ ((λ (x) (+ x x)) 5)
→ (+ 5 5)
→ 10
```


În practică II

Evaluare leneșă (o formă de evaluare normală) în Haskell: parametri evaluați la cerere, fapt ce permite construcții interesante

Exemplul 11.6 (Evaluare leneșă în Haskell).

```
((\x -> x + x) (2 + 3))
→ (2 + 3) + (2 + 3)
→ 5 + 5
→ 10
```

Nevoie de funcții nestricte, chiar în limbajele aplicative: `if`, `and`, `or` etc.

Transferul parametrilor

- Evaluare aplicativă
  - *Call by value*
  - *Call by sharing*
  - *Call by reference*
  - *Call by copying*
- Evaluare normală
  - *Call by name*
  - *Call by need*

Call by value

Exemplul 11.7 (*Call by value* în C).

```
1  void f(int x) {
2      x = 3;
3  }
```

```
9  void g(struct student s) {
10     s.age = 3;
11     s     = t;
12 }
```

Efectele liniilor 2, 10 și 11: invizibile la apelant.

- Evaluarea parametrilor înaintea aplicării funcției și transferul unor copii ale valorilor acestora
- Modificări locale invizibile la apelant
- C, C++, tipurile primitive în Java

Call by sharing

Exemplu

Exemplul 11.8 (*Call by sharing* în Java).

```
4      void f(Student s) {
5          s.age = 3;
6          s     = new Student();
7      }
```

- Efectul liniei 5: vizibil la apelant
- Efectul liniei 6: invizibil la apelant

Call by sharing

Trăsături

- Variantă a *call by value*
- Trimiterea unei referințe la obiect
- Modificări locale asupra referinței: invizibile la apelant
- Modificări locale asupra obiectului referit: vizibile la apelant
- Scheme, tipurile referință în Java
- Diferență față de C, unde o structură trimisă ca parametru este complet copiată

Call by reference

Exemplul 11.9 (*Call by reference* în C++).

```
1  void f(int &x) {
2      x = 3;
3  }
```

Efectul liniei 2: vizibil la apelant

- Trimiterea unei referințe la obiect
- Modificări locale asupra referinței și obiectului referit: vizibile la apelant
- & în C++

Call by name

- Argumente neevaluate în momentul aplicării funcției, substituție directă în corp
- Evaluarea parametrilor la cerere, de fiecare dată când este nevoie de valoarea acestora, în contextul parametrilor formali

Exemplul 11.10 (*Call by name*).

```
int sum(by_name int term, int limit) {
    int x, s = 0;
    for (x = 1; x <= limit; x++)
        s += term;
    return s;
}
```

`sum(x * x, 10)` calculează $\sum_{x=1}^{10} x^2$

Call by need

- Variantă a *call by name*
- Evaluarea unui parametru doar la prima utilizare a acestuia
- Memorarea valorii unui parametru deja evaluat și returnarea acesteia în cazul utilizării repetate a aceluiași parametru
- Haskell, cu evaluare în contextul parametrilor actuali

Rezumat

- Calculul lambda: model de calculabilitate, bazat pe funcții și substituție textuală
- Variabile, respectiv apariții ale variabilelor, legate sau libere, în raport cu o anumită expresie
- β-reducere, α-conversie, pas/secvență/ordine de reducere, formă normală
- Reducere stânga-dreapta (evaluare în ordine normală): garanția terminării pentru expresii reductibile
- Reducere dreapta-stânga (evaluare în ordine aplicativă): mai eficientă, dar fără garanția terminării, nici măcar pentru expresii reductibile!

Cursul III

Calculul Lambda ca Limbaj de Programare

Cuprins

Cuprins

Scop

- Demonstrarea puterii expresive a calculului lambda
- Ipotetică mașină λ
- λ-expresii: cod mașină — limbajul λ_0
- Locul
  - biților
  - operațiilor pe biți,

  luat de
  - șiruri structurate de simboli
  - reducere — substituție textuală

Convenții

- Instrucțiuni:
  - λ-expresii
  - legări de variabile *top-level*: *variabila* $\equiv_{\text{def}}$ *expresie*, de exemplu: *true* $\equiv_{\text{def}} \lambda x.\lambda y.x$
- Valori reprezentate de funcții
- Expresii aduse la forma închisă, înaintea evaluării
- Evaluare normală
- Forma normală funcțională (v. Definiția 10.2)
- Absența tipurilor predefinite!

Scrieri prescurtate

- $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.E \rightsquigarrow \lambda x_1 x_2 \ldots x_n.E$
- $((\ldots((E\ A_1)\ A_2)\ \ldots)\ A_n) \rightsquigarrow (E\ A_1\ A_2\ \ldots\ A_n)$

Rolul tipurilor

- Modalitate de exprimare a intenției programatorului
- Documentare: ce operatori acționează asupra căror obiecte
- Reprezentarea particulară a valorilor de tipuri diferite: `1`, `“Hello”`, `#t` etc.
- Optimizarea operațiilor specifice
- Prevenirea erorilor
- Facilitarea verificării formale

Absența tipurilor

Cum sunt reprezentate entitățile?

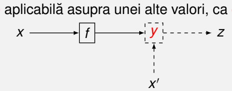
- Un număr, o listă sau un arbore, posibil desemnate de aceeași valoare! Exemplu:

  numărul 3 $\rightarrow \lambda x.\lambda y.x \leftarrow$ lista (() () ())

- Valori și operatori reprezentați de funcții, semnificația fiind dependentă de context

  numărul 3 $\rightarrow \lambda x.\lambda y.x \leftarrow$ operatorul *car*

- Valoare aplicabilă asupra unei alte valori, ca operator!



Absența tipurilor
Cum este afectată corectitudinea calculului?

- Incapacitatea mașinii λ de a
  - interpreta semnificația expresiilor
  - asigura corectitudinea acestora
- Orice operatori aplicabili asupra oricăror valori
- Delegarea aspectelor de mai sus programatorului
- Construcții eronate acceptate fără avertisment, dar calcule terminate cu
  - valori fără semnificație sau
  - expresii care nu sunt valori, dar nici nu mai pot fi reduse, de exemplu: $(x\ x)$

Absența tipurilor
Consecințe

- Flexibilitate sporită în reprezentare
- Potrivită în situațiile în care reprezentarea uniformă a obiectelor, ca liste de simboli, este convenabilă
- Predispoziție crescută la erori
- Instabilitatea programelor
- Dificultatea verificării și mentenanței

Deci...

- Cum utilizăm limbajul λ_0 în programarea cotidiană?
- Cum reprezentăm valorile uzuale — numere, booleeni, liste etc. — și operatorii aferenți?

Cuprins

Definiție

Definiția 13.1 (Tip de date abstract, TDA).
Model matematic al unei mulțimi de valori și al operațiilor valide pe acestea.

Exemplul 13.2 (TDA-uri).
Natural, *Bool*, *List*, *Set*, *Stack*, *Tree*, ... λ-expresie!

Componente:
- constructori de bază: cum se generează valorile
- operatori: ce se poate face cu acestea
- axiome: cum

TDA *Natural*
Constructori de bază și operatori

- Constructori de bază:
  - $zero : \to Natural$
  - $succ : Natural \to Natural$
- Operatori:
  - $zero? : Natural \to Bool$
  - $pred : Natural \setminus \{zero\} \to Natural$
  - $add : Natural^2 \to Natural$

TDA *Natural*
Axiome

- *zero?*
  - $(zero?\ zero) = T$
  - $(zero?\ (succ\ n)) = F$
- *pred*
  - $(pred\ (succ\ n)) = n$
- *add*
  - $(add\ zero\ n) = n$
  - $(add\ (succ\ m)\ n) = (succ\ (add\ m\ n))$

Scrierea axiomelor

- Câte o axiomă pentru fiecare pereche (operator, constructor de bază)
- Definiții suplimentare — inutile
- Definiții mai puține — insuficiente pentru specificarea completă a comportamentului operatorilor

De la TDA la programare funcțională
Exemplu

- Axiome:
  - $(add\ zero\ n) = n$
  - $(add\ (succ\ m)\ n) = (succ\ (add\ m\ n))$
- Scheme:

```
(define add
  (lambda (m n)
    (if (zero? m) n
        (+ 1 (add (- m 1) n)))))
```

- Haskell:

```
add 0 n       = n
add (m + 1) n = 1 + (add m n)
```


De la TDA la programare funcțională
Discuție

- Demonstrarea corectitudinii TDA — inducție structurală
- Demonstrarea proprietăților λ-expresiilor, aparținând unui TDA cu 3 constructori de bază!
- Programarea funcțională — reflectarea specificațiilor matematice
- Recursivitatea — instrument natural, moștenit din axiome
- Aplicarea procedeelor formale pe codul recursiv, exploatând absența efectelor laterale

Cuprins

TDA *Bool*
Constructori de bază și operatori

- Constructori de bază:
  - $T : \to Bool$
  - $F : \to Bool$
- Operatori:
  - $not : Bool \to Bool$
  - $and : Bool^2 \to Bool$
  - $or : Bool^2 \to Bool$
  - $if : Bool \times T \times T \to T$

TDA *Bool*
Axiome

- *not*
  - $(not\ T) = F$
  - $(not\ F) = T$
- *and*
  - $(and\ T\ a) = a$
  - $(and\ F\ a) = F$
- *or*
  - $(or\ T\ a) = T$
  - $(or\ F\ a) = a$
- *if*
  - $(if\ T\ a\ b) = a$
  - $(if\ F\ a\ b) = b$

TDA *Bool*
Implementarea constructorilor de bază

- Intuiție: selecția între cele două valori, *true* și *false*
- $T \equiv_{\text{def}} \lambda xy.x$
- $F \equiv_{\text{def}} \lambda xy.y$
- Comportament de selectori:
  - $(T\ a\ b) \to (\lambda xy.x\ a\ b) \to a$
  - $(F\ a\ b) \to (\lambda xy.y\ a\ b) \to b$

TDA *Bool*
Implementarea operatorilor

- $not \equiv_{\text{def}} \lambda x.(x\ F\ T)$
  - $(not\ T) \to (\lambda x.(x\ F\ T)\ T) \to (T\ F\ T) \to F$
  - $(not\ F) \to (\lambda x.(x\ F\ T)\ F) \to (F\ F\ T) \to T$
- $and \equiv_{\text{def}} \lambda xy.(x\ y\ F)$
  - $(and\ T\ a) \to (\lambda xy.(x\ y\ F)\ T\ a) \to (T\ a\ F) \to a$
  - $(and\ F\ a) \to (\lambda xy.(x\ y\ F)\ F\ a) \to (F\ a\ F) \to F$
- $or \equiv_{\text{def}} \lambda xy.(x\ T\ y)$
  - $(or\ T\ a) \to (\lambda xy.(x\ T\ y)\ T\ a) \to (T\ T\ a) \to T$
  - $(or\ F\ a) \to (\lambda xy.(x\ T\ y)\ F\ a) \to (F\ T\ a) \to a$
- $if \equiv_{\text{def}} \lambda cte.(c\ t\ e)$ nestrictă!
  - $(if\ T\ a\ b) \to (\lambda cte.(c\ t\ e)\ T\ a\ b) \to (T\ a\ b) \to a$
  - $(if\ F\ a\ b) \to (\lambda cte.(c\ t\ e)\ F\ a\ b) \to (F\ a\ b) \to b$

TDA *Pair*
Specificare

- Constructori de bază:
  - $pair : A \times B \to Pair$
- Operatori:
  - $fst : Pair \to A$
  - $snd : Pair \to B$
- Axiome:
  - $(fst\ (pair\ a\ b)) = a$
  - $(snd\ (pair\ a\ b)) = b$

TDA *Pair*
Implementare

- Intuiție: pereche = funcție ce așteaptă selectorul, pentru a-l aplica asupra membrilor
- $pair \equiv_{def} \lambda xys.(s\ x\ y)$
  - $(pair\ a\ b) \rightarrow (\lambda xys.(s\ x\ y)\ a\ b) \rightarrow \lambda s.(s\ a\ b)$
- $fst \equiv_{def} \lambda p.(p\ T)$
  - $(fst\ (pair\ a\ b)) \rightarrow (\lambda p.(p\ T)\ \lambda s.(s\ a\ b)) \rightarrow (\lambda s.(s\ a\ b)\ T) \rightarrow (T\ a\ b) \rightarrow a$
- $snd \equiv_{def} \lambda p.(p\ F)$
  - $(snd\ (pair\ a\ b)) \rightarrow (\lambda p.(p\ F)\ \lambda s.(s\ a\ b)) \rightarrow (\lambda s.(s\ a\ b)\ F) \rightarrow (F\ a\ b) \rightarrow b$

TDA *List*
Specificare

- Constructori de bază:
  - $null : \rightarrow List$
  - $cons : A \times List \rightarrow List$
- Operatori:
  - $car : List \setminus \{null\} \rightarrow A$
  - $cdr : List \setminus \{null\} \rightarrow List$
  - $null? : List \rightarrow Bool$
  - $append : List^2 \rightarrow List$

TDA *List*
Axiome

- *car*
  - $(car\ (cons\ e\ L)) = e$
- *cdr*
  - $(cdr\ (cons\ e\ L)) = L$
- *null?*
  - $(null?\ null) = T$
  - $(null?\ (cons\ e\ L)) = F$
- *append*
  - $(append\ null\ B) = B$
  - $(append\ (cons\ e\ A)\ B) = (cons\ e\ (append\ A\ B))$

TDA *List*
Implementare

- Intuiție: listă = pereche (*head*, *tail*)
- $null \equiv_{def} \lambda x.T$
- $cons \equiv_{def} pair$
- $car \equiv_{def} fst$
- $cdr \equiv_{def} snd$
- $null? \equiv_{def} \lambda L.(L\ \lambda xy.F)$
  - $(null?\ null) \rightarrow (\lambda L.(L\ \lambda xy.F)\ \lambda x.T) \rightarrow (\lambda x.T\ \ldots) \rightarrow T$
  - $(null?\ (cons\ e\ L)) \rightarrow (\lambda L.(L\ \lambda xy.F)\ \lambda s.(s\ e\ L)) \rightarrow (\lambda s.(s\ e\ L)\ \lambda xy.F) \rightarrow (\lambda xy.F\ e\ L) \rightarrow F$
- $append \equiv_{def}$ … nu are formă închisă
  $\lambda AB.(if\ (null?\ A)\ B\ (cons\ (car\ A)\ (append\ (cdr\ A)\ B)))$

TDA *Natural*
Axiome

- *zero?*
  - $(zero?\ zero) = T$
  - $(zero?\ (succ\ n)) = F$
- *pred*
  - $(pred\ (succ\ n)) = n$
- *add*
  - $(add\ zero\ n) = n$
  - $(add\ (succ\ m)\ n) = (succ\ (add\ m\ n))$

TDA *Natural*
Implementare

- Intuiție: număr = listă cu lungimea egală cu valoarea numărului
- $zero \equiv_{def} null$
- $succ \equiv_{def} \lambda n.(cons\ null\ n)$
- $zero? \equiv_{def} null?$
- $pred \equiv_{def} cdr$
- $add \equiv_{def} append$

Cuprins

Funcții

- Definiții ale funcției identitate:
  - $id(n) = n$
  - $id(n) = n + 1 - 1$
  - $id(n) = n + 2 - 2$
  - …
- O infinitate de reprezentări textuale ale aceleiași funcții
- Atunci… ce este o funcție? O relație între valori, independentă de reprezentările textuale: $id = \{(0,0),(1,1),(2,2),\ldots\}$

Perspective asupra recursivității

- Textuală: funcție care se autoapelează, folosindu-și numele
- Constructivistă: funcții recursive ca valori ale unui TDA, cu precizarea modalităților de generare
- Semantică: ce obiect matematic este desemnat de o funcție recursivă

Implementare *length*
Problemă

- Lungimea unei liste:

  $length \equiv_{def} \lambda L.(if\ (null?\ L)\ zero\ (succ\ (\underline{length}\ (cdr\ L))))$
- Cu ce înlocuim zona subliniată, pentru a evita recursivitatea textuală?
- Putem primi, ca parametru, o funcție echivalentă computațional cu *length*?

  $Length \equiv_{def} \lambda f L.(if\ (null?\ L)\ zero\ (succ\ (f\ (cdr\ L))))$
- $(Length\ length) \rightarrow length$ — un punct fix al lui *Length*!
- Cum obținem punctul fix?

Puncte fixe

Definiția 15.1 (Punct fix).
f este un punct fix al funcției F dacă $(F\ f) \rightarrow f$.

Exemplul 15.2 (Puncte fixe).

$$Fix = \lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x)))$$

- $(Fix\ F) \rightarrow (\lambda x.(F\ (x\ x))\ \lambda x.(F\ (x\ x))) \rightarrow (F\ \underline{(\lambda x.(F\ (x\ x))\ \lambda x.(F\ (x\ x)))}) = (F\ (Fix\ F))$
- $(Fix\ F)$ este un punct fix al lui F

Definiția 15.3 (Combinator de punct fix).
Funcție ce generează un punct fix al oricărei expresii.
Exemplu: *Fix*.

Implementare *length*
Soluție

- $length \equiv_{def} (Fix\ Length) \rightarrow (Length\ (Fix\ Length)) \rightarrow \lambda L.(if\ (null?\ L)\ zero\ (succ\ (\underbrace{(Fix\ Length)}_{length}\ (cdr\ L))))$
- Funcție recursivă, fără a fi textual recursivă!

Combinatori de punct fix

- Pentru funcții unare, de exemplu, *length*:
  $c_1 \equiv_{def} \lambda f.(\lambda gx.(f\ (g\ g)\ x)\ \lambda gx.(f\ (g\ g)\ x))$
- Pentru funcții binare, de exemplu, *append*:
  $c_2 \equiv_{def} \lambda f.(\lambda gxy.(f\ (g\ g)\ x\ y)\ \lambda gxy.(f\ (g\ g)\ x\ y))$

Rezumat

- Forța de expresie a calculului lambda: suficientă pentru reprezentarea valorilor uzuale și a operatorilor caracteristici
- Recursivitatea: trăsătură comportamentală, nu neapărat textuală

Cursul IV

Programare Funcțională în Scheme

Cuprins

Cuprins

Deosebiri față de λ_0

- Tipare: dinamică/latentă
  - Valorile au tip (`3`, `#f` etc.)
  - Variabilele nu au tip
  - Verificare la execuție, în momentul aplicării unei funcții
- Recursivitate textuală
- Diverse modalități de legare a variabilelor (eng. *scoping*)

Cuprins

Modalități de tipare

- Rolul tipurilor (v. slide-ul 110)
- După momentul verificării:
  - statică
  - dinamică
- După rigiditatea regulilor:
  - tare
  - slabă

Tipare statică vs. dinamică

Tipare statică
- La compilare
- Valori și variabile
- Rulare mai rapidă
- Rigidă: sancționează toate construcțiile
- Debugging mai facil
- Declarații explicite sau inferențe de tip
- Pascal, C, C++, Java, Haskell

Tipare dinamică
- La rulare
- Doar valori
- Rulare mai lentă
- Flexibilă: sancționează doar când este necesar
- Debugging mai dificil
- Metaprogramare (v. `eval`)
- Python, Scheme, Prolog, JavaScript, PHP

Tipare tare vs. slabă

Criteriu: libertatea de agregare a valorilor de tipuri diferite

Exemplul 17.1 (Tipare tare).
`1 + "23"` : Eroare (Haskell)

Exemplul 17.2 (Tipare slabă).
- Visual Basic: `1 + "23" = 24`
- JavaScript: `1 + "23" = "123"`

Tiparea în Scheme

- Dinamică
- Tare

Exemplul 17.3 (Tipare dinamică în Scheme).

```
1 (if #t 1 (+ 1 #t)) → 1
2 (if #f 1 (+ 1 #t)) → Eroare
```

Deși linia 1 conține o subexpresie eronată, aceasta nu împiedică desfășurarea calculului, din moment ce nu este evaluată.

Cuprins

Variabile
Proprietăți

- Tip: nu în Scheme!
- Identificator
- Valoarea legată (la un anumit moment)
- Domeniul de vizibilitate
- Durata de viață

Variabile
Stări

- Declarată: cunoaștem identificatorul
- Definită: cunoaștem și valoarea

Legarea variabilelor

Definiția 18.1 (Legarea variabilelor).
Modalitatea de asociere a apariției unei variabile cu definiția acesteia.

Definiția 18.2 (Domeniu de vizibilitate, *scope*).
Mulțimea punctelor din program unde o definiție este vizibilă, fiind determinată de modalitatea de legare a variabilelor.

Modalități de legare:
- statică
- dinamică

Legarea statică a variabilelor

Definiția 18.3 (Legare statică/lexicală).
Variabilele din corpul unei expresii sunt extrase din contextul în care aceasta a fost definită. Domeniul de vizibilitate este determinat prin construcțiile limbajului, putând fi desprins la compilare.

Exemplul 18.4 (Legare statică).

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

Ce va returna `g()`?
`0`

Legare statică în calculul lambda

Exemplul 18.5 (Legare statică).
Care sunt domeniile de vizibilitate a variabilelor de legare, în expresia $\lambda x.\lambda y.(\lambda x.x\ y)$?
- $\lambda x.\lambda y.(\lambda \underline{x}.x\ y)$
- $\lambda x.\lambda \underline{y}.(\lambda x.x\ y)$
- $\lambda \underline{x}.\lambda y.(\lambda x.x\ y)$

Legarea dinamică a variabilelor

Definiția 18.6 (Legare dinamică).
Valorile variabilelor depind de momentul în care o expresie este evaluată. Domeniul de vizibilitate este determinat la execuție.

Exemplul 18.7 (Legare dinamică).

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

Ce va returna `g()`?
`g()` → `x = 2` → `f()` → 2 (ultima valoare!)

Legare mixtă

Exemplul 18.8 (Legare mixtă).

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

Dacă variabilele locale sunt legate static, iar cele globale, dinamic, ce va returna `g()`?
`1`

Legarea variabilelor în Scheme

- Variabile declarate sau definite în exresii: 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. Exemplul 18.4)

Construcția `lambda`
Exemplu

Exemplul 18.9 (Construcția `lambda`).

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


Construcția `lambda`
Semantică

- Aplicație:

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

- Se evaluează argumentele `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. Exemplul 18.4)

Construcția `let`
Exemplu

Exemplul 18.10 (Construcția `let`).

```
(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. Exemplul 18.4)

Construcția `let*`
Exemplu

Exemplul 18.11 (Construcția `let*`).

```
(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. Exemplul 18.4)

Construcția `letrec`
Exemplu

Exemplul 18.12 (Construcția `letrec`).

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


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

Exemplul 18.13 (Construcția `define`).

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


Construcția `define`
Exemple

Exemplul 18.14 (Construcția `define`).

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

(factorial 5)

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

(g 5)
```

Output: `120 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: coruperea transparenței referențiale

Legarea variabilelor în Scheme
Exemplu mixt

Exemplul 18.15 (Codificarea Exemplului 18.8).

```
(define x 0)
(define f (lambda () x))
(define x 1)
(define g
  (lambda ()
    (let ([x 2])
      (f))))

(g)
```

Output: `1`

Cuprins

Construcția `set!`

Definiție

- Modifică valoarea unei variabile locale sau *top-level*
- Sintaxă:

```
(set! v expr)
```

- Diferență la nivel de intenție față de construcțiile anterioare
- `let` și `define`: definirea de variabile noi
- `set!`: modificarea celor existente!

Construcția `set!`

Exemplu

Exemplul 19.1 (Construcția `set!`).

```
(define x 0)

(define f
  (lambda (p)
    (set! x p)
    x))

(f 3) ; 3
x ; 3
```


Construcția `set!`

Semantică

- Se evaluează expresia, `expr`
- Noua valoare a lui `v` este valoarea lui `expr`

Atribuiri

- Avantaje:
  - Modelarea obiectelor cu stare variabilă în timp
  - Evitarea pasării explicite a fiecărei modificări de stare
- Dezavantaj: pierderea transparenței referențiale (v. Cursul 1, începând cu slide-ul 21)

Cuprins

Evaluarea în Scheme

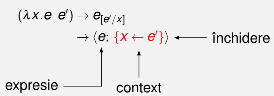
- Evaluare aplicativă: evaluarea parametrilor înaintea aplicării funcției asupra acestora
- Transferul parametrilor: *call by sharing*, variantă a *call by value* (v. slide-ul 99)
- Funcții stricte
- Excepții: `if`, `cond`, `and`, `or`, `quote` etc.

Substituție textuală

$$(\lambda x.e\ e') \rightarrow e_{[e'/x]}$$

- Ineficientă
- Constrânsă de restricția: $FV(e') \cap BV(e) = \emptyset$
- Imposibil de aplicat, în prezența efectelor laterale

Alternativă la substituția textuală



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

Contexte computaționale

Definiție

Definiția 20.1 (Context computațional).
Contextul computațional al unui punct P, dintr-un program, la momentul t, este mulțimea variabilelor și a valorilor acestora, pentru care domeniile de vizibilitate aferente îl conțin pe P, la momentul t.

- Legare statică — mulțimea variabilelor care îl conțin pe P în domeniul lexical de vizibilitate
- Legare dinamică — mulțimea variabilelor definite cel mai recent, la momentul t, și referite din P

Contexte computaționale

Exemplu

Exemplul 20.2 (Contexte computaționale).
Ce variabile locale conține contextul computațional al punctului P?

```
(lambda (x y)
  (lambda (z)
    (let ([x (car y)])
      ; ... P ...)))
```


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!

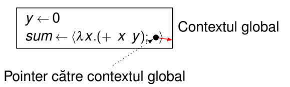
Închideri

Construcție

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

Exemplul 20.3 (Construcția închiderilor).

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



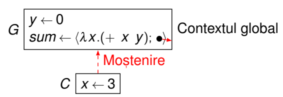
Închideri

Aplicare

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

Exemplul 20.4 (Aplicarea închiderilor).

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



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.

Exemplul 20.5 (Continuarea Exemplului 20.4).

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

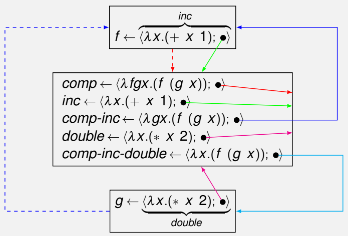
Închideri funcționale

Exemplu

Exemplul 20.6 (Închideri funcționale).

```
(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



Controlul evaluării

- `quote` sau `'`
  - funcție nestrictă
  - întoarce parametrul neevaluat
- `eval`
  - funcție strictă
  - forțează evaluarea parametrului și întoarce valoarea acestuia

Exemplul 20.7 (Controlul evaluării).

```
(define sum '(2 + 3))
sum ; (2 + 3)
(eval (list (cadr sum) (car sum) (caddr sum)))
; 5
```


Rezumat

- Tipare statică/dinamică[1], tare/slabă
- Legare statică/dinamică a variabilelor
- Evaluare aplicativă, contexte de evaluare, închideri
- Efecte laterale

[1] Scheme

Cursul V

Evaluare Leneșă în Scheme

Cuprins

Cuprins

Motivație

Exemplul 21.1 (Întârzierea evaluării).
Să se implementeze funcția nestrictă *prod*, astfel încât al doilea parametru să fie evaluat doar dacă primul este *true*:

- $prod(false, y) = 0$
- $prod(true, y) = y(y+1)$

Varianta 1
Implementare directă

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

(define test
  (lambda (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
  (lambda (x y)
    (if x (* (eval y) (+ (eval y) 1)) 0)))

(define test
  (lambda (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
  (lambda (x y)
    (if x (* (y) (+ (y) 1)) 0)))

(define test
  (lambda (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
  (lambda (x y)
    (if x (* (force y) (+ (force y) 1)) 0)))

(define test
  (lambda (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. Definiția 5.1)
- `delay`
  - construiește o promisiune
  - funcție nestrictă
- `force`
  - forțează respectarea unei promisiuni, evaluând expresia doar la prima aplicare, și salvându-i valoarea
  - începând cu a doua invocare, întoarce, direct, valoarea memorată

Promisiuni
Cerințe

- Salvarea contextului computațional al expresiei a cărei evaluare este întârziată și evaluarea ei, ulterioară, în acel context x — închideri funcționale?
- Salvarea rezultatului primei evaluări a expresiei
- Distingerea primei forțări de celelalte

Promisiuni I
Implementare

```
(define make-promise
  (lambda (closure)
    (let ([ready? #f]
          [result #f])
      ; promisiunea
      (lambda ()
        (if ready?
            result
            (let ([r (closure)])
              (if ready?
                  result
                  (begin (set! ready? #t)
                         (set! result r)
                         result)))))))))
```


Promisiuni II
Implementare

```
(define-macro my-delay
  (lambda (expr)
    `(make-promise (lambda () ,expr))))

(define my-force
  (lambda (p)
    (p)))

(define p1 (my-delay (begin (display "p1")
                            (+ 1 2))))
(my-force p1) ; p1 3
(my-force p1) ; 3
```


Promisiuni
Detalii de implementare

- Situații în care evaluarea expresiei împachetate declanșează, ea însăși, forțarea promisiunii — a doua verificare a lui `ready?`
- Promisiuni: obiecte cu stare
- Prima forțare — efecte laterale

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

Abstracții procedurale
Motivație

Exemplul 22.1 (Absența abstracțiilor).

```scheme
(lambda (x y)
  (/ 2
     (+ (/ 1
           (* x x))
        (/ 1
           (* y y)))))
```

Probleme ale secvenței de mai sus:

- Opacitate conceptuală — semnificația operațiilor este neclară
- Aglomerarea nivelelor de detaliu

Abstracții procedurale
Intuiție

Ce putem face?

- Pornim de la operațiile primitive din limbaj
- Le antrenăm în funcționalități complexe
- Le asociem, celor din urmă, o identitate proprie
- Obținem abstracții procedurale, privite prin prisma funcționalității, și nu a implementării

Abstracții procedurale I
Mai concret

Exemplul 22.2 (Abstracții procedurale).

```scheme
(define square
  (lambda (x)
    (* x x)))

(define inverse
  (lambda (x)
    (/ 1 x)))

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


Abstracții procedurale II
Mai concret

Exemplul 22.2 (Abstracții procedurale).

```scheme
(define harmonic-mean
  (lambda (x y)
    (inverse (average (inverse x)
                      (inverse y)))))

(define f
  (lambda (x y)
    (harmonic-mean (square x)
                   (square y))))
```


Abstracții procedurale
Avantaje

Ce am obținut?

- Evidențierea conceptelor utilizate
- Izolarea nivelelor de detaliu
- Reutilizare
- Substituibilitatea funcțiilor: de exemplu, din perspectiva lui `f`, nu interesează implementarea lui `square`

Abstracții 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

Abstracții 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

Abstracții de date III

Definiția 22.3 (Abstracție de date).
Tehnică de separare a utilizării unei structuri de date de implementarea acesteia.

Permit *wishful thinking*: utilizarea structurii înaintea implementării acesteia

Abstracții de date IV

Exemplul 22.4 (Abstracții de date).

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

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

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

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


Cuprins

Motivație

Exemplul 23.1 (Acumulare vs. liste).
Să se determine suma numerelor pare din intervalul $[a, b]$.

```scheme
(define even-sum-iter
  (lambda (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
  (lambda (a b)
    (foldl + 0 (filter even? (interval a b)))))
```


Comparație

- Varianta iterativă (d.p.d.v. proces): eficientă, datorită spațiului suplimentar constant
- Varianta pe liste:
  - elegantă și concisă
  - ineficientă, datorită spațiului posibil mare, ocupat la un moment dat — toate numerele din intervalul $[a, b]$
- Cum îmbinăm avantajele celor 2 abordări?

Fluxuri
Caracteristici

- Secvențe construite parțial, extinse la cerere, ce creează iluzia completitudinii structurii
- Îmbinarea eleganței manipulării listelor cu eficiența calculului incremental
- Bariera de abstractizare:
  - componentele listelor evaluate la construcție (`cons`)
  - ale fluxurilor la selecție (`cdr`)
- Construcția și utilizarea:
  - separate la nivel conceptual — modularitate
  - întrepătrunse la nivel de proces

Fluxuri I
Operatori

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

(define stream-car car)

(define stream-cdr
  (lambda (s)
    (unpack (cdr s))))

(define stream-null '())

(define stream-null? null?)

(define stream-take
```

Fluxuri II
Operatori

```
  (lambda (n s)
    (cond [(zero? n) '()]
          [(stream-null? s) '()]
          [else (cons (stream-car s)
                      (stream-take
                       (- n 1)
                       (stream-cdr s)))])))

(define stream-drop
  (lambda (n s)
    (cond [(zero? n) s]
          [(stream-null? s) s]
          [else (stream-drop (- n 1)
                             (stream-cdr s))])))

(define stream-map
```


Fluxuri III
Operatori

```
  (lambda (f s)
    (if (stream-null? s) s
        (stream-cons
         (f (stream-car s))
         (stream-map f (stream-cdr s))))))

(define stream-filter
  (lambda (f? s)
    (cond [(stream-null? s) s]
          [(f? (stream-car s))
           (stream-cons
            (stream-car s)
            (stream-filter f? (stream-cdr s)))]
          [else (stream-filter
                 f?
                 (stream-cdr s))])))
```


Fluxuri IV
Operatori

```

(define stream-zip-with
  (lambda (f s1 s2)
    (if (stream-null? s1) s2
        (stream-cons
         (f (stream-car s1) (stream-car s2))
         (stream-zip-with
          f
          (stream-cdr s1)
          (stream-cdr s2))))))


(define stream-append
  (lambda (s1 s2)
    (if (stream-null? s1) s2
        (stream-cons (stream-car s1)
```


Fluxuri V
Operatori

```
                     (stream-append
                      (stream-cdr s1)
                      s2)))))

(define list->stream
  (lambda (L)
    (if (null? L) stream-null
        (stream-cons (car L)
                     (list->stream (cdr L))))))
```


Barierele de abstractizare

Fluxuri,
ca entități autonome:
utilizare

Interfață: `stream-*`

Expresii cu evaluare întârziată, ca entități autonome: utilizare

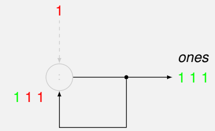
Fluxuri, ca perechi conținând expresii cu evaluare întârziată: implementare

Interfață: `pack`, `unpack`

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

Fluxul de numere 1
Implementare

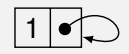
```
(define ones (stream-cons 1 ones))
; (stream-take 5 ones) ; (1 1 1 1 1)
```



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

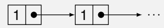
Fluxul de numere 1
Utilizarea memoriei

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

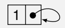


Alternativ: `(define ones (pack (cons 1 ones)))`

- închideri:



- promisiuni:



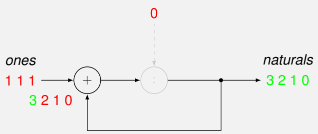
Fluxul numerelor naturale
Formulare explicită

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

(define naturals (naturals-from 0))
```

- Închideri: multiple parcurgeri ale fluxului determină reevaluarea porțiunilor deja explorate
  - Explorare 1, cu 3 elemente: 0 1 2
  - Explorare 2, cu 5 elemente: 0 1 2 3 4
- Promisiuni: multiple parcurgeri ale fluxului determină evaluarea dincolo de porțiunile deja explorate
  - Explorare 1, cu 3 elemente: 0 1 2
  - Explorare 2, cu 5 elemente: 0 1 2 3 4

Fluxul numerelor naturale
Formulare implicită

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

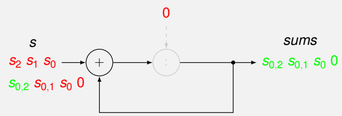


Fluxul numerelor pare

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

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


Fluxul sumelor parțiale

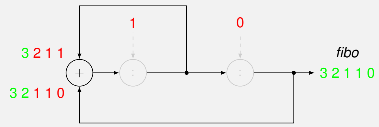
```
(define sums
  (lambda (s)
    (letrec ([out (stream-cons
                   0
                   (stream-zip-with + s out))])
      out)))
```



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

Fluxul numerelor Fibonacci
Formulare implicită

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



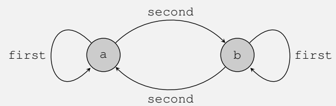
Fluxul numerelor prime I

- 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
  (lambda (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:

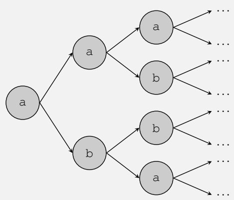
- cheia: `key`
- legăturile către două noduri: `first`, `second`

Grafuri ciclice II

```
(define-macro node
  (lambda (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

Definiția 24.1 (Spațiul stărilor unei probleme).
Mulțimea configurațiilor valide din universul problemei.

Problema palindroamelor
Definiție

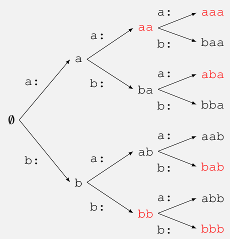
Definiția 24.2 (Problema palindroamelor, *Pal$_n$*).
Să se determine palindroamele de lungime cel puțin n, ce se pot forma cu elementele unui alfabet fixat.

Stările problemei: toate șirurile generabile cu elementele alfabetului respectiv.

Problema palindroamelor
Specificare *Pal$_n$*

- Starea inițială: șirul vid
- Operatorii de generare a stărilor succesor ale unei stări: inserarea unui caracter la începutul unui șir dat
- Operatorul de verificare a proprietății de *gol* a unei stări: palindrom, de lungime cel puțin *n*

Problema palindroamelor
Spațiul stărilor lui *Pal$_2$*



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

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

Căutare leneșă în lățime I
Fluxul stărilor *gol*

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


Căutare leneșă în lățime II
Fluxul stărilor *gol*

```
                             state))))))))])
      (search (stream-cons init stream-null)))))

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

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

Aplicații

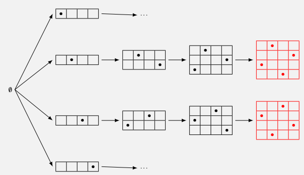
- Palindroame
- Problema reginelor

Problema reginelor
Definiție

Definiția 24.3 (Problema reginelor, *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.
- Abstracțiile procedurale și de date permit
  - evidențierea conceptelor în termenii cărora o implementare este gândită
  - dezvăluirea treptată, a nivelelor de detaliu, i.e. modularizarea
  - reutilizarea.

Bibliografie

Abelson, H. și Sussman, G. J. (1996).
Structure and Interpretation of Computer Programs.
Ediția a doua. MIT Press.

Cursul VI

Programare funcțională în Haskell

Cuprins

- 25 Introducere
- 26 Tipare
- 27 Sinteza de tip
- 28 Evaluare

Cuprins

- 25 Introducere
- 26 Tipare
- 27 Sinteza de tip
- 28 Evaluare

Paralelă între limbaje

Criteriu	Scheme	Haskell
Funcții	*Curried* / *uncurried*	*Curried*
Tipare	Dinamică, tare	Statică, tare
Legarea variabilelor	Locale → statică, *top-level* → dinamică	Statică
Evaluare	Aplicativă	Leneșă
Transferul parametrilor	*Call by sharing*	*Call by need*
Efecte laterale	`set!`	Interzise, direct

Funcții

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

Exemplul 25.1 (Definiții echivalente ale funcției `add`).

```
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
```


Funcții și operatori

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

Exemplul 25.2 (Definiții echivalente ale lui `add` și `inc`).

```
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

Exemplul 25.3 (*Pattern matching*).

```
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, ca într-o specificare matematică

Exemplul 25.4 (*List comprehensions*).

```
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

- 25 Introducere
- 26 Tipare
- 27 Sinteza de tip
- 28 Evaluare

Tipuri

- Tipuri ca mulțimi de valori:
  - `Bool = {True, False}`
  - `Natural = {0, 1, 2, ...}`
  - `Char = {'a', 'b', 'c', ...}`
- Rolul tipurilor (v. slide-ul 110)
- 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

Exemplul 26.1 (Valori și tipurile acestora).

```
5                :: Integer
'a'              :: Char
inc              :: Integer -> Integer
[1,2,3]          :: [Integer]
(True, "Hello")  :: (Bool, [Char])
```


Tipuri de bază

- Tipurile elementare din limbaj
- Exemple:
  - `Bool`
  - `Char`
  - `Integer`
  - `Int`
  - `Float`

Constructori de tip

Funcții de tip, ce îmbogățesc tipurile din limbaj

Exemplul 26.2 (Constructori de tip predefiniți).

```
-- 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 dreapta:
`Integer -> Integer -> Integer`
    ≡ `Integer -> (Integer -> Integer)`

Exemplul 26.3 (Tipurile funcțiilor).

```
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

Definiția 26.4 (Polimorfism parametric).
Manifestarea aceluiași comportament pentru parametri de tipuri diferite. Exemplu: `idd`.

Definiția 26.5 (Polimorfism ad-hoc).
Manifestarea unor comportamente diferite pentru parametri de tipuri diferite. Exemplu: `==`.

Constructorul de tip `Natural` I

Definit de utilizator

Exemplul 26.6 (Constructorul de tip `Natural`).

```
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

Exemplul 26.7 (Constructorul de tip `Pair`).

```
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

Exemplul 26.8 (Reprezentarea tipurilor).

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

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

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

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


Proprietăți induse de tipuri

Definiția 26.9 (Progres).
O expresie bine-tipată (căreia i se poate asocia un tip):
- este o valoare sau
- poate fi redusă.

Definiția 26.10 (Conservare).
Evaluarea unei expresii bine-tipate produce o expresie bine-tipată — de obicei, cu același tip.

Cuprins

Sinteza de tip

Definiția 27.1 (Sinteză de tip, *type inference*).
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
  - variabile de tip: pot fi legate la orice expresii de tip
  - aplicații ale constructorilor de tip pe expresii de tip

Reguli simplificate de sinteză de tip I

- Formă:

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

- Funcție:

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

- Aplicație:

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

Reguli simplificate de sinteză de tip II

- Operatorul `+`:

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

- Literali întregi:

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

Exemple de sinteză de tip I

Exemplul 27.2 (Sinteză de tip).

```
f g = (g 3) + 1
```

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

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

`b = Int`

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

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

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

Exemple de sinteză de tip II

Exemplul 27.3 (Sinteză de tip).

```
fix f = f (fix f)
```

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

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

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

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

`a -> b = e -> g, a = e, b = g, c = g`

`f :: (c -> d) -> b = (g -> g) -> g`

Exemple de sinteză de tip III

Exemplul 27.4 (Sinteză de tip).

```
f x = (x x)
```

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

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

Ecuația `c -> d = c` nu are soluție, deci funcția nu poate fi tipată.

Unificare I

Sinteza de tip presupune legarea variabilelor în scopul unificării diverselor expresii de tip, elaborate.

Definiția 27.5 (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.

Definiția 27.6 (Substituție).
Mulțime de legări variabilă-valoare.

Unificare II

Exemplul 27.7 (Unificare).
- 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])`

Definiția 27.8 (*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

Exemplul 27.9 (Cel mai general tip al unei expresii).
- Funcție: `\x -> x`
- Tipuri corecte:
  - `Int -> Int`
  - `Bool -> Bool`
  - `a -> a`
- Unele tipuri se obțin prin instanțierea altora.

Definiția 27.10 (Tip principal al unei expresii).
Cel mai general tip care descrie complet natura expresiei. Se obține prin utilizarea MGU.

Cuprins

Evaluare

- Evaluare leneșă: parametri evaluați la cerere, cel mult o dată, eventual parțial, în cazul obiectelor structurate
- Transferul parametrilor: *call by need*
- Funcții nestricte!

Exemplul 28.1 (Evaluare).

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

f (2 + 3, 3 + 5) (5 + 8)
→ (2 + 3) + (2 + 3)
→ 5 + 5     reutilizăm rezultatul primei evaluări!
→ 10
```


Pași în aplicarea funcțiilor I

Exemplul 28.2 (Evaluare [Thompson, 1999]).

```
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]
```


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

Exemplul 28.2 (continuare).

```
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!

Exemplul 28.3 (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))
```


Rezumat

- Tipare statică și tare, anterioară evaluării
- Evaluare leneșă

Bibliografie

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

Cursul VII

Evaluare Leneșă în Haskell

Cuprins

Programare orientată spre date

Prelucrări traduse în termenii unor operații pe structuri de date, posibil niciodată generate complet!

Exemplul 28.4 (Suma pătratelor [Thompson, 1999]).

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.

Programare orientată spre date I

Exemplul 28.5 (Minimul unei liste [Thompson, 1999]).

Minimul unei liste, drept prim element al acesteia, după sortarea prin inserție.

```
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)
```


Programare orientată spre date II

Exemplul 28.5 (Minimul unei liste [Thompson, 1999]).

```
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, tras în fața acesteia și întors.

Backtracking eficient

Găsirea eficientă a unui obiect, prin generarea aparentă, a tuturor acestora.

Exemplul 28.6 (Accesibilitatea într-un graf [Thompson, 1999]).

Accesibilitatea între două noduri, ca existență a elementelor în mulțimea tuturor căilor dintre cele două noduri:

```
theGraph = [(1, 2), (1, 4), (2, 1), (2, 3),
            (3, 5), (3, 6), (5, 6), (6, 1)]

accessible source dest graph =
    (routes source dest graph []) /= []
```

Backtracking desfășurat doar până la determinarea primului element al listei.

Studiu de caz

Bibliotecă de parsare [Thompson, 1999]

Bibliografie

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

Cursul VIII

Clase în Haskell

Cuprins

Cuprins

Motivație

Exemplul 29.1 (`show`).
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)
                ++ ")"
```

- Tipul pereche reprezentabil ca șir doar dacă tipurile celor doi membri respectă aceeași proprietate

Clase și instanțe

Definiția 29.2 (Clasă).
Mulțime de tipuri ce supraîncarcă operațiile specifice clasei. Reprezintă o modalitate structurată de control al polimorfismului ad-hoc. Exemplu: clasa `Show`, cu operația `show`.

Definiția 29.3 (Instanță a unei clase).
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 din cei 2 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

- Mulțimi de tipuri
- Instanțierea claselor de către tipuri
- Implementarea operațiilor în afara definiției tipului

POO

- Mulțimi de obiecte: *tipuri*
- Implementarea interfețelor de clase
- Implementarea operațiilor în cadrul definiției tipului

Cuprins

`invert` I

Exemplul 30.1 (`invert`).
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

Exemplul 30.2 (`contents`).
Să se definească operația `contents`, aplicabilă pe obiecte structurate, inclusiv pe cele aparținând tipurilor `Pair a` și `NestedList a`, care întoarce elementele, sub forma unei liste.

```
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

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

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

- Conform definiției clasei:

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

- Conform supraîncărcării funcției (`id`):

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

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

`contents` III

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

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

- Conform definiției clasei:

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

- Conform supraîncărcării funcției (`id`):

```haskell
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

```haskell
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

```haskell
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

Cursul IX

Logica Propozițională și cu Predicate de Ordinul I

Cuprins

Cuprins

Logică [Harrison, 2009]

- Scop: reducerea efectuării de raționamente, la calcul
- Problemele de decidabilitate din logică: stimulent pentru dezvoltarea modelelor de calculabilitate
- Între domeniul logicii și al calculatoarelor, împrumuturi în ambele direcții:
  - proiectarea și verificarea programelor → logică
  - principii logice → proiectarea limbajelor de programare

Rolurile logicii

- Descrierea proprietăților obiectelor, într-o manieră neambiguă, prin intermediul unui limbaj, cu următoarele componente:
  - sintaxă: modalitatea de construcție a expresiilor din limbaj
  - 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 (abordarea aleasă) sau
  - înțelesul propriu-zis al acesteia, într-o interpretare
- Valoarea de adevăr a unei propoziții determinată de valorile de adevăr ale propozițiilor constituente

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

- Scop: dezvoltarea unor mecanisme de prelucrare, aplicabile independent de valoarea de adevăr al propozițiilor, într-o situație particulară
- Accent pe relațiile între propozițiile compuse și cele constituente
- Pentru explicitarea legăturilor, utilizarea conceptului de interpretare

Semantică II

Definiția 32.1 (Interpretare).
Mulțime de asocieri între fiecare propoziție simplă din limbaj și o valoare de adevăr.

Exemplul 32.2 (Interpretări).

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 al 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

Definiția 32.3 (Evaluare).
Determinarea valorii de adevăr a unei propoziții, sub o interpretare, prin aplicarea regulilor semantice anterioare.

Exemplul 32.4 (Evaluare).
- 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

Definiția 32.5 (Satisfiabilitate).
Proprietatea unei propoziții adevărate sub cel puțin o interpretare. Acea interpretare satisface propoziția.

Exemplul 32.6 (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

Definiția 32.7 (Validitate).
Proprietatea unei propoziții adevărate în toate interpretările. Propoziția se mai numește tautologie.

Exemplul 32.8 (Validitate).
Propoziția $p \vee \neg p$ este adevărată, indiferent de valoarea de adevăr al lui p, deci este validă.

Verificabilă prin metoda tabelei de adevăr

Nesatisfiabilitate

Definiția 32.9 (Nesatisfiabilitate).
Proprietatea unei propoziții false în toate interpretările. Propoziția se mai numește contradicție.

Exemplul 32.10 (Nesatisfiabilitate).
Propoziția $p \Leftrightarrow \neg p$ este falsă, indiferent de valoarea de adevăr al lui p, deci este nesatisfiabilă.

Verificabilă prin metoda tabelei de adevăr

Cuprins

Derivabilitate I

Definiția 32.11 (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, fapt notat prin $\Delta \models \phi$, dacă și numai dacă orice interpretare care satisface toate propozițiile din Δ satisface și ϕ.

Exemplul 32.12 (Derivabilitate logică).
- $\{p\} \models p \vee q$
- $\{p, q\} \models p \wedge q$
- $\{p\} \not\models p \wedge q$
- $\{p, p \Rightarrow q\} \models q$

Derivabilitate II

Verificabilă prin metoda tabelei de adevăr: toate intrările pentru care premisele sunt adevărate trebuie să inducă adevărul concluziei

Exemplul 32.13 (Derivabilitate logică).
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

Definiția 32.14 (Inferență).
Derivarea mecanică a concluziilor unui set de premise.

Definiția 32.15 (Regulă de inferență).
Procedură de calcul capabilă să deriveze concluziile unui set de premise. Derivabilitatea mecanică, a concluziei ϕ, din mulțimea de premise Δ, utilizând regula de inferență *inf*, se notează $\Delta \vdash_{inf} \phi$.

Reguli de inferență

- Șabloane parametrizate de raționament, formate dintr-o mulțime de premise și o mulțime de concluzii
- *Modus Ponens* (MP):

$$\frac{\begin{array}{c}\alpha \Rightarrow \beta \\ \alpha\end{array}}{\beta}$$

- *Modus Tollens*:

$$\frac{\begin{array}{c}\alpha \Rightarrow \beta \\ \neg\beta\end{array}}{\neg\alpha}$$

Proprietăți ale regulilor de inferență

Definiția 32.16 (Consistență, *soundness*).
Regula de inferență determină doar propoziții care sunt, într-adevăr, consecințe logice ale premiselor. Echivalent, $\Delta \vdash_{inf} \phi \Rightarrow \Delta \models \phi$.

Definiția 32.17 (Completitudine, *completeness*).
Regula de inferență determină toate consecințele logice ale premiselor. Echivalent, $\Delta \models \phi \Rightarrow \Delta \vdash_{inf} \phi$.

- Ideal, ambele proprietăți: "nici în plus, nici în minus"
- Incompletitudinea regulii *Modus Ponens*, din imposibilitatea scrierii oricărei propoziții, ca implicație

Axiome

- Exemplu: verificarea că $\{p \Rightarrow q, q \Rightarrow r\} \vdash p \Rightarrow r$
- Caz în care premisele sunt insuficiente pentru aplicarea regulilor de inferență
- Soluția: 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

Definiția 32.18 (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.

Definiția 32.19 (Teoremă).
Concluzia cu care se termină o demonstrație.

Demonstrații II

Definiția 32.20 (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

Exemplul 32.21 (Demonstrație).
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

- 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

Definiția 32.22 (Literal).
Propoziție simplă sau negația ei. Exemplu: p și $\neg p$.

Definiția 32.23 (Expresie clauzală).
Literal sau disjuncție de literali. Exemplu: $p \lor \neg q \lor r$.

Definiția 32.24 (Clauză).
Mulțime de literali dintr-o expresie clauzală. Exemplu: $\{p, \neg q, r\}$.

Forma clauzală II

Definiția 32.25 (Forma clauzală / Forma normală conjunctivă — FNC).
Reprezentarea unei propoziții sub forma unei mulțimi de clauze, implicit legate prin conjuncții.

Exemplul 32.26 (FNC).
Forma clauzală a propoziției $p \land (\neg q \lor r) \land (\neg p \lor \neg r)$ este $\{p\}, \{\neg q, r\}, \{\neg p, \neg r\}$.

Orice propoziție convertibilă în această formă, conform algoritmului următor

Forma clauzală III

1. Eliminarea implicațiilor (I):

$$\alpha \Rightarrow \beta \ \rightarrow \ \neg\alpha \lor \beta$$

2. Introducerea negațiilor în paranteze (N):

$$\neg(\alpha \land \beta) \ \rightarrow \ \neg\alpha \lor \neg\beta \text{ etc.}$$

3. Distribuirea lui $\lor$ față de $\land$ (D):

$$\alpha \lor (\beta \land \gamma) \ \rightarrow \ (\alpha \lor \beta) \land (\alpha \lor \gamma)$$

4. Transformarea expresiilor în clauze (C):

$$\phi_1 \lor \ldots \lor \phi_n \ \rightarrow \ \{\phi_1, \ldots, \phi_n\}$$
$$\phi_1 \land \ldots \land \phi_n \ \rightarrow \ \{\phi_1\}, \ldots, \{\phi_n\}$$

Forma clauzală IV

Exemplul 32.27 (Transformare în forma clauzală).
Transformăm propoziția $p \land (q \Rightarrow r)$ în formă clauzală.

I	$p \land (\neg q \lor r)$
C	$\{p\}, \{\neg q, r\}$

Exemplul 32.28 (Transformare în forma clauzală).
Transformăm propoziția $\neg(p \land (q \Rightarrow r))$ în formă clauzală.

I	$\neg(p \land (\neg q \lor r))$
N	$\neg p \lor \neg(\neg q \lor r)$
N	$\neg p \lor (q \land \neg r)$
D	$(\neg p \lor q) \land (\neg p \lor \neg r)$
C	$\{\neg p, q\}, \{\neg p, \neg r\}$

Rezoluție I

- Ideea:

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

- "Anularea" lui p
- p adevărată, $\neg p$ falsă, r adevărată
- p falsă, q adevărată
- Cel puțin una dintre q și r adevărată
- Forma generală:

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

Rezoluție II

- Rezolvent vid — contradicție între premise:

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

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

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

Rezoluție III

- *Modus Ponens* — caz particular al rezoluției:

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

- *Modus Tollens* — caz particular al rezoluției:

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

- Tranzitivitatea implicației:

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

Rezoluție IV

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

Exemplul 32.29 (Reducere la absurd).

Demonstrăm că $\{p \Rightarrow q, q \Rightarrow r\} \vdash p \Rightarrow r$, i.e. 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 32.30 (Rezoluției).

Rezoluția propozițională este consistentă și completă, i.e. $\Delta \models \phi \Leftrightarrow \Delta \vdash \phi$.

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

Cuprins

Logica cu predicate de ordinul I

- *First Order Logic* (FOL)
- Extensie a logicii propoziționale, cu explicitarea:
  - obiectelor din universul problemei
  - relațiilor dintre acestea
- 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
- 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

Cuprins

Sintaxă

Simboluri utilizate

- Constante: obiecte particulare din universul discursului: c, d, $andrei$, $bogdan$, ...
- Variabile: obiecte generice: x, y, ...
- Simboluri funcționale: $succesor$, $+$, ...
- Simboluri relaționale (predicate): relații n-are peste obiectele din universul discursului: $prieten = \{(andrei, bogdan), (bogdan, andrei), \ldots\}$, $impar = \{1, 3, \ldots\}$, ...
- 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, și anume 5
    - $+(2, x)$: aplicația funcției de adunare asupra numerelor 2 și x, și, totodată, suma lor

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: $\perp$, $\top$
  - Atomi: A
  - Negații: $\neg F$
  - ...
  - Cuantificări: $\forall x.\alpha$, $\exists x.\alpha$

Sintaxă III

Termeni, atomi, propoziții

Exemplul 33.1.

"Dan este prieten cu sora Ioanei":

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

- Simplificare: legarea tuturor variabilelor, prin cuantificatori universali sau existențiali
- Domeniul de vizibilitate al unui cuantificator $\rightarrow$ restul propoziției (v. simbolul λ în Calculul Lambda)

Semantică I

Definiția 33.2 (Interpretare).

O interpretare constă din:
- Un domeniu nevid, D
- Pentru fiecare constantă c, un element $c^I \in D$
- Pentru fiecare simbol funcțional, n-ar, f, o funcție $f^I : D^n \rightarrow D$
- Pentru fiecare predicat n-ar, p, o funcție $p^I : D^n \rightarrow \{false, true\}$.

Semantică II

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

Exemplul 33.3.

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

- $\forall x.(vrabie(x) \Rightarrow viseaza(x, malai))$
  $\rightarrow$ corect: "Toate vrăbiile visează mălai."
- $\forall x.(vrabie(x) \wedge viseaza(x, malai))$
  $\rightarrow$ greșit: "Toți sunt vrăbii care visează mălai."
- $\exists x.(vrabie(x) \wedge viseaza(x, malai))$
  $\rightarrow$ corect: "Unele vrăbii visează mălai."
- $\exists x.(vrabie(x) \Rightarrow viseaza(x, malai))$
  $\rightarrow$ greșit: adevărată și dacă există cineva care nu este vrabie

Cuantificatori

Proprietăți

- Necomutativitate:
  - $\forall x.\exists y.viseaza(x, y) \rightarrow$ "Toți visează la ceva anume."
  - $\exists y.\forall x.viseaza(x, y) \rightarrow$ "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

Forme normale I

Definiția 33.4 (Literal).
Atom sau negația lui. Exemplu: $prieten(x, y)$ și $\neg prieten(x, y)$.

Definiția 33.5 (Expresie clauzală).
Literal sau disjuncție de literali. Exemplu: $prieten(x, y) \vee \neg doctor(x)$.

Definiția 33.6 (Clauză).
Mulțime de literali dintr-o expresie clauzală. Exemplu: $\{prieten(x, y), \neg doctor(x)\}$.

Forme normale II

Definiția 33.7 (Forma clauzală / Forma normală conjunctivă — FNC).
Reprezentarea unei propoziții sub forma unei mulțimi de clauze, implicit legate prin conjuncții.

Definiția 33.8 (Forma normală implicativă — FNI).
Reprezentarea unei propoziții sub forma unei mulțimi de clauze, implicit legate prin conjuncții, în care fiecare clauză are forma grupată $\{\neg A_1, \ldots, \neg A_m, B_1, \ldots, B_n\}$, corespunzătoare implicației $(A_1 \wedge \ldots \wedge A_m) \Rightarrow (B_1 \vee \ldots \vee B_n)$, unde A_i și B_j sunt atomi.

Forme normale III

Definiția 33.9 (Clauză Horn).
Clauză în care un singur literal este în formă pozitivă: $\{\neg A_1, \ldots, \neg A_n, A\}$, corespunzătoare implicației $A_1 \wedge \ldots \wedge A_n \Rightarrow A$.

Exemplul 33.10 (Clauze Horn).
Transformarea propoziției $vrabie(x) \vee ciocarlie(x) \Rightarrow pasare(x)$ în forme normale, utulizând clauze Horn:

- FNC: $\{\neg vrabie(x), pasare(x)\}, \{\neg ciocarlie(x), pasare(x)\}$
- FNI: $vrabie(x) \Rightarrow pasare(x), ciocarlie(x) \Rightarrow pasare(x)$

Conversia propozițiilor în FNC 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):

   $\forall x.p(x) \wedge \forall y.q(y) \vee \exists z.r(z) \rightarrow \forall x.\forall y.\exists z.(p(x) \wedge q(y) \vee r(z))$

Conversia propozițiilor în FNC 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:

     $\forall x.\forall y.\exists z.(p(x) \wedge q(y) \vee r(z)) \rightarrow \forall x.\forall y.(p(x) \wedge q(y) \vee r(f_z(x, y)))$

Conversia propozițiilor în FNC 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)

Conversia propozițiilor în FNC IV

Exemplul 33.11.
"Cine rezolvă toate laboratoarele este apreciat de cineva."
$\forall x.(\forall y.(lab(y) \Rightarrow rezolva(x, y)) \Rightarrow \exists y.apreciaza(y, x))$

I	$\forall x.(\neg \forall y.(\neg lab(y) \vee rezolva(x, y)) \vee \exists y.apreciaza(y, x))$
N	$\forall x.(\exists y.\neg(\neg lab(y) \vee rezolva(x, y)) \vee \exists y.apreciaza(y, x))$
N	$\forall x.(\exists y.(lab(y) \wedge \neg rezolva(x, y)) \vee \exists y.apreciaza(y, x))$
R	$\forall x.(\exists y.(lab(y) \wedge \neg rezolva(x, y)) \vee \exists z.apreciaza(z, x))$
P	$\forall x.\exists y.\exists z.((lab(y) \wedge \neg rezolva(x, y)) \vee apreciaza(z, x))$
S	$\forall x.((lab(f_y(x)) \wedge \neg rezolva(x, f_y(x))) \vee apreciaza(f_z(x), x))$
U	$(lab(f_y(x)) \wedge \neg rezolva(x, f_y(x))) \vee apreciaza(f_z(x), x)$
D	$(lab(f_y(x)) \vee apreciaza(f_z(x), x))$ $\wedge (\neg rezolva(x, f_y(x)) \vee apreciaza(f_z(x), x))$
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 — Definițiile 27.5, 27.6, 27.8)
- 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:

  $$\frac{\begin{array}{l}A_1 \wedge \ldots \wedge A_m \Rightarrow A\\ B_1 \wedge \ldots \wedge A' \wedge \ldots \wedge B_n \Rightarrow B\\ unificare(A, A') = S\end{array}}{subst(S, \quad A_1 \wedge \ldots \wedge A_m \wedge B_1 \wedge \ldots \wedge B_n \Rightarrow B)}$$

- $unificare(\alpha, \beta) \rightarrow$ substituția sub care unifică propozițiile α și β
- $subst(S, \alpha) \rightarrow$ 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

Cursul X

Programare logică în Prolog

Cuprins

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 FOL restricționat
- "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

Cuprins

Un prim exemplu

Exemplul 35.1.

```
% 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

Exemplul 35.2.

```
% 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.
```


Cuprins

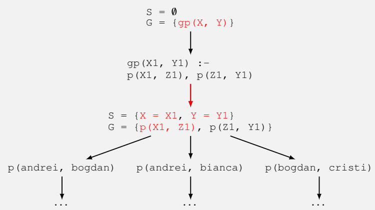
Pași în demonstrare I

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

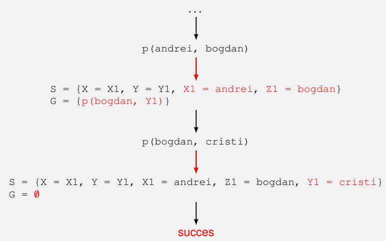
Pași în demonstrare II

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

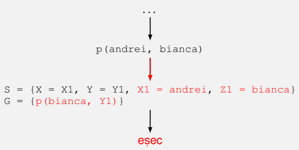
Exemplul genealogic I



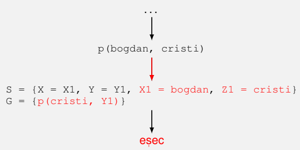
Exemplul genealogic II



Exemplul genealogic III



Exemplul genealogic IV



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*)

- Derivarea tuturor concluziilor, pornind de la datele inițiale
- Oprire la obținerea scopului (scopurilor)
- Principiul de funcționare a agendei CLIPS

Backward chaining (*goal-driven*)

- Utilizarea exclusivă a regulilor care pot contribui efectiv la satisfacerea scopului
- Determinarea regulilor a căror concluzie unifică cu scopul
- Încercarea de satisfacere a premiselor acestor reguli ș.a.m.d.

Backward chaining I

Intrare: *rules* — lista regulilor din program
Intrare: *goals* — stiva de scopuri
Intrare: *subst* — substituția curentă, inițial vidă
Ieșire: satisfiabilitatea scopurilor

```
1:  procedure BackwardChaining(rules, goals, subst)
2:      if goals = ∅ then
3:          return SUCCESS
4:      end if
5:      goal ← head(goals)
6:      goals ← tail(goals)
```


Backward chaining II

```
7:      for-each rule ∈ rules, în ordinea din program do
8:          if unify(goal, conclusion(rule), subst, bindings)
    then
9:              newGoals ← premises(rule) ∪ goals
10:             newSubst ← subst ∪ bindings
11:             if
    BackwardChaining(rules, newGoals, newSubst) then
12:                 return SUCCESS
13:             end if
14:         end if
15:     end for
16:     return FAILURE
17: end procedure
```

Linia 9: căutare în adâncime

Cuprins

Minimul a două numere I

Exemplul 37.1 (Minimul a două numere).

```
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?

Exemplul 37.2 (Eliminarea eronată, a unei condiții).

```
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

Exemplul 37.3.

```
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

Exemplul 37.4 (`cut`).

```
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.
```


Negația ca eșec

Exemplul 37.5 (Implementare `nott`).

```
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

Cursul XI

Mașina algoritmică Markov

Cuprins

Cuprins

Mașina algoritmică Markov

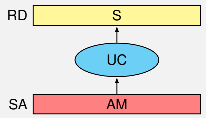
- Model de calculabilitate efectivă, echivalent cu mașina Turing și cu calculul lambda
- Principiul de funcționare: identificare de șabloane (eng. *pattern matching*) și substituție
- Fundamentul teoretic al paradigmei asociative și al limbajelor bazate pe reguli

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

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: $\mathtt{ag_1c}$ `-> 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

Mulțimi ordonate de reguli, îmbogățite cu declarații de:

- partiționare a mulțimii A_b
- variabile generice

Exemplul 39.1 (Algoritm Markov).

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
```

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

Aplicabilitatea regulilor

Definiția 39.2 (Aplicabilitatea unei reguli).

Regula $\mathtt{r\ :\ a_1 \ldots a_n\ \text{->}\ b_1 \ldots b_m}$ este aplicabilă dacă și numai dacă există un subșir $\mathtt{c_1 \ldots c_n}$, în RD, astfel încât, pentru orice $\mathtt{i} = \overline{\mathtt{1,n}}$, exact o condiție de mai jos este îndeplinită:

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

Aplicarea regulilor

Definiția 39.3 (Aplicarea unei reguli).

Aplicarea regulii $\mathtt{r\ :\ a_1 \ldots a_n\ \text{->}\ b_1 \ldots b_m}$ asupra unui subșir $\mathtt{s = c_1 \ldots c_n}$, în raport cu care este aplicabilă, constă în substituirea lui `s` prin subșirul $\mathtt{q_1 \ldots q_m}$, calculat astfel:

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

Exemplu de aplicare

Exemplul 39.4 (Aplicarea unei reguli).

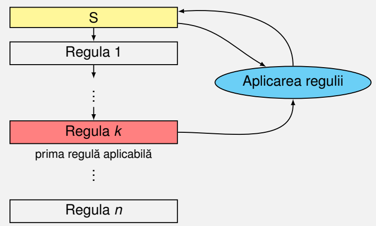
- A_b = `{1, 2, 3}`
- A_l = `{x, y}`
- $\mathtt{Dom(g_1)} = \mathtt{\{2\}}$
- $\mathtt{Dom(g_2)} = A_b$
- `s` = `1111112x2y31111`
- $\mathtt{r} : \mathtt{1g_1xg_1yg_2\ \text{->}\ 1g_2x}$

s	=	11111	1	2	x	2	y	3	1111	
r	:		1	$\mathtt{g_1}$	x	$\mathtt{g_1}$	y	$\mathtt{g_2}$	->	$\mathtt{1g_2x}$
s'	=	1111113x1111								

Aplicabilitate vs. aplicare

- Aplicabilitatea
  - 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

Unitatea de control III

procedure CONTROL(*s*, *rules*)
  $i \leftarrow 1$
  $n \leftarrow |rules|$
  $status \leftarrow$ RUNNING
  while $i \leq n$ **and** $status =$ RUNNING **do**
    $r \leftarrow rules[i]$
    if *isApplicable*(s, r) **then**
      $s \leftarrow fire(s, r)$
      if *isTerminal*(r) **then**
        $status \leftarrow$ TERMINATED
      else
        $i \leftarrow 1$
      end if

Unitatea de control IV

    else
      $i \leftarrow i + 1$
    end if
  end while
  if $status =$ TERMINATED **then**
    return s
  else
    error("Execution blocked")
  end if
end procedure

Inversarea intrării

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

```
Reverse(A); A g1, g2;
    ag1g2 -> g2ag1;
    ag1 -> bg1;
    abg1 -> g1a;
    a -> .;
    -> a;
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 substituție

Cursul XII

Programare asociativă în CLIPS

Cuprins

Cuprins

CLIPS

- “C Language Integrated Production System”
- Sistem bazat pe reguli — “producție” = regulă
- Principiu de funcționare similar cu al mașinii Markov
- Dezvoltat la NASA
- Posibilitatea codificării de implicații logice în reguli — sisteme expert

Sisteme expert
Trăsături

- Edward Feigenbaum: “un program inteligent care folosește cunoștințe și reguli de inferență pentru a rezolva probleme suficient de dificile încât să necesite expertiză umană semnifcativă”
- Mimarea procesului de decizie, al unui expert uman
- Limitarea la un domeniu specific — dificultatea construirii unui rezolvitor general de probleme

Sisteme expert
Aplicații

- Configurare de sisteme
- Diagnoză (medicală etc.)
- Educație
- Planificare
- Prognoză
- ...

Cuprins

Minimul a două numere
Reprezentare individuală a numerelor

Exemplul 41.1.

```
(deffacts numbers
    (number 1)
    (number 2))

(defrule min
    (number ?m)
    (number ?x)
    (test (< ?m ?x))
  =>
    (assert (min ?m)))
```


Fapte

- Reprezentarea datelor prin fapte, similare simbolilor mașinii Markov
- Afirmații despre atributele obiectelor
- Date simbolice, construite conform unor șabloane
- Mulțimea de fapte: baza de cunoștințe factuale (*factual knowledge base*)

```
> (facts)
f-0     (initial-fact)
f-1     (number 1)
f-2     (number 2)
For a total of 3 facts.
```


Reguli

- Similare regulilor mașinii Markov
- Șablon de identificare: secvență de fapte parametrizate (v. variabilele generice ale algoritmilor Markov) și restricții
- Șablon de acțiune: secvență de acțiuni
- *Pattern matching* secvențial pe faptele din șablonul de identificare
- Domeniul de vizibilitate a unei variabile: restul regulii, după prima apariție a variabilei, în șablonul de identificare

Înregistrări de activare I

- Tuplul (regulă, fapte asupra cărora este aplicabilă): înregistrare de activare (*activation record*)
- Reguli posibil aplicabile asupra diferitelor porțiuni ale acelorași fapte
- Mulțimea înregistrărilor de activare: agenda

Înregistrări de activare II

```
> (facts)
f-0     (initial-fact)
f-1     (number 1)
f-2     (number 2)
For a total of 3 facts.

> (agenda)
0      min: f-1,f-2
For a total of 1 activation.

> (run)
FIRE    1 min: f-1,f-2
==> f-3     (min 1)
```


Principiul refracției

- Aplicarea unei reguli, o singură dată, asupra acelorași (porțiuni ale unor) fapte
- Altfel, neterminarea programelor

Terminarea programelor

- Golirea agendei
- Execuția acțiunii `(halt)`
- Aplicarea unui număr maxim de reguli: `(run n)`

Cuprins

Minimul a două numere I
Reprezentare agregată a numerelor

Exemplul 42.1.

```
(deffacts numbers
    (numbers 1 2))

(defrule min
    (numbers $? ?m $?)
    (numbers $? ?x $?)
    (test (< ?m ?x))
  =>
    (assert (min ?m)))
```


Minimul a două numere II
Reprezentare agregată a numerelor

`$?` este o variabilă anonimă, ce se potrivește cu orice secvență, eventual vidă

```
> (facts)
f-0      (initial-fact)
f-1      (numbers 1 2)
For a total of 2 facts.

> (agenda)
0       min: f-1,f-1
For a total of 1 activation.
```


Minimul a două numere III
Reprezentare agregată a numerelor

Exemplul 42.2.

```
(deffacts numbers (numbers 1 2))

(defrule min1
    (numbers ?m ?x)
    (test (< ?m ?x))
  =>
    (assert (min ?m)))

(defrule min2
    (numbers ?x ?m)
    (test (< ?m ?x))
  =>
    (assert (min ?m)))
```


Minimul a două numere IV
Reprezentare agregată a numerelor

Selectarea explicită a celor 2 numere împiedică alegerea automată, convenabilă, a acestora, ca în Exemplul 42.1

```
> (facts)
f-0      (initial-fact)
f-1      (numbers 1 2)
For a total of 2 facts.

> (agenda)
0       min1: f-1
For a total of 1 activation.
```


Suma oricâtor numere I

Exemplul 42.3 (Abordare care nu se termină).

```
(deffacts numbers (numbers 1 2 3 4 5))

(defrule init
    ; implicit, (initial-fact)
  =>
    (assert (sum 0)))

(defrule sum
    ?f <- (sum ?s)
    (numbers $? ?x $?)
  =>
    (retract ?f)
    (assert (sum (+ ?s ?x))))
```


Suma oricâtor numere II

```
> (facts)
f-0      (initial-fact)
f-1      (numbers 1 2 3 4 5)
For a total of 2 facts.

> (agenda)
0       init: *
For a total of 1 activation.

> (run 1)
FIRE    1 init: *
==> f-2      (sum 0)

```


Suma oricâtor numere III

```
> (agenda)
0       sum: f-2,f-1
0       sum: f-2,f-1
0       sum: f-2,f-1
0       sum: f-2,f-1
0       sum: f-2,f-1
For a total of 5 activations.

> (run)
ciclează!
```

- Eroarea: adăugarea unui nou fapt `sum` induce aplicabilitatea repetată a regulii, asupra elementelor deja însumate
- Corect: consultarea primului număr din listă și eliminarea acestuia

Suma oricâtor numere IV

Exemplul 42.4 (Abordare corectă).

```
(deffacts numbers (numbers 1 2 3 4 5))

(defrule init
  =>
    (assert (sum 0)))

(defrule sum
    ?f <- (sum ?s)
    ?g <- (numbers ?x $?rest)
  =>
    (retract ?f)
    (assert (sum (+ ?s ?x)))
    (retract ?g)
    (assert (numbers $?rest)))
```

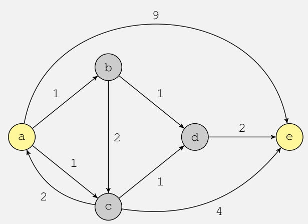

Suma oricâtor numere V

```
> (run)
FIRE    1 init: *
==> f-2      (sum 0)
FIRE    2 sum: f-2,f-1
<== f-2      (sum 0)
==> f-3      (sum 1)
<== f-1      (numbers 1 2 3 4 5)
==> f-4      (numbers 2 3 4 5)
FIRE    3 sum: f-3,f-4
<== f-3      (sum 1)
==> f-5      (sum 3)
<== f-4      (numbers 2 3 4 5)
==> f-6      (numbers 3 4 5)
```


Suma oricâtor numere VI

```
FIRE    4 sum: f-5,f-6
<== f-5      (sum 3)
==> f-7      (sum 6)
<== f-6      (numbers 3 4 5)
==> f-8      (numbers 4 5)
FIRE    5 sum: f-7,f-8
<== f-7      (sum 6)
==> f-9      (sum 10)
<== f-8      (numbers 4 5)
==> f-10     (numbers 5)
FIRE    6 sum: f-9,f-10
<== f-9      (sum 10)
==> f-11     (sum 15)
<== f-10     (numbers 5)
==> f-12     (numbers)
```


Accesibilitatea într-un graf I



Accesibilitatea într-un graf II

- Graful: $G = (V, E)$
- Relația de accesibilitate: $Acc \subseteq V^2$
- $(u, v) \in E \Rightarrow (u, v) \in Acc$
- $(x, y) \in Acc \wedge (y, z) \in E \Rightarrow (x, z) \in Acc$

Accesibilitatea într-un graf III

Exemplul 42.5.

```
(deftemplate edge (slot from) (slot to) (slot cost))
(deftemplate acc (slot source) (slot dest))
(deftemplate find (slot source) (slot dest))

(deffacts graph
    (edge (from a) (to b) (cost 1))
    (edge (from a) (to c) (cost 1))
    (edge (from a) (to e) (cost 9))
    (edge (from b) (to c) (cost 2))
    (edge (from b) (to d) (cost 1))
    (edge (from c) (to a) (cost 2))
    (edge (from c) (to d) (cost 1))
    (edge (from c) (to e) (cost 4))
```


Accesibilitatea într-un graf IV

Exemplul 42.5.

```
    (edge (from d) (to e) (cost 2))
    (find (source a) (dest e)))

(defrule base
    (edge (from ?x) (to ?y))
  =>
    (assert (acc (source ?x) (dest ?y))))

(defrule expand
    (acc (source ?x) (dest ?y))
    (edge (from ?y) (to ?z))
  =>
    (assert (acc (source ?x) (dest ?z))))
```


Accesibilitatea într-un graf V

Exemplul 42.5.

```
(defrule found
    (find (source ?x) (dest ?y))
    (acc (source ?x) (dest ?y))
  =>
    (printout t "Found" crlf)
    (halt)) ; Ne oprim cand raspundem afirmativ.
```


Cuprins

Accesibilitatea într-un graf I
Optimizare

- Exemplul 42.5: posibilitatea continuării explorării grafului, după obținerea răspunsului căutat
- Optimizare: forțarea aplicării regulii `found`, imediat după identificarea răspunsului
- Problemă: aplicabilitatea concomitentă, a regulilor `expand` și `found`
- Soluție: prioritizarea regulii `found`

Accesibilitatea într-un graf II
Optimizare

Exemplul 43.1.

```
(defrule found
    (declare (salience 10))
    (find (source ?x) (dest ?y))
    (acc (source ?x) (dest ?y))
  =>
    (printout t "Found" crlf)
    (halt)) ; Ne oprim cand raspundem afirmativ.
```


Salience

- *Salience* = prioritatea de aplicare, a unei reguli
- Implicit 0, posibil negativă
- Valoare mai mare: prioritate mai mare

Minimul oricâtor numere I
Determinare iterativă

Exemplul 43.2.

```
(deffacts numbers (numbers 5 7 1 3))

(defrule init
    (not (min ?m))
    (numbers ?x $?rest)
  =>
    (assert (min ?x)))
```


Minimul oricâtor numere II
Determinare iterativă

Exemplul 43.2.

```
(defrule compute
    ?f <- (min ?m)
    (numbers $? ?x $?)
    (test (< ?x ?m))
  =>
    (retract ?f)
    (assert (min ?x)))

(defrule print
    (declare (salience -10)) ; compute neaplicabila
    (min ?m)
  =>
    (printout t ?m crlf))
```

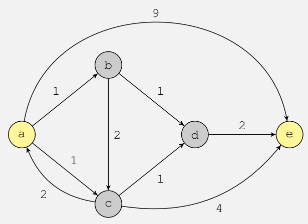

Minimul oricâtor numere
Determinare directă

Exemplul 43.3.

```
(deffacts numbers (numbers 1 5 7 3))

(defrule min
    (numbers $? ?m $?)
    (not (numbers $? ?x & :(< ?x ?m) $?))
  =>
    (assert (min ?m))
    (printout t ?m crlf))
```

- Definirea de condiții *inline* asupra variabilelor, prin `&`
- Citirea regulii: “Minimul este acel element pentru care nu găsim altul mai mic”

Drumurile optime într-un graf cu costuri I



Drumurile optime într-un graf cu costuri II

1. Drumurile optime *source* $\rightsquigarrow$ *dest* sunt drumurile *utile* *source* $\rightsquigarrow$ *dest*, în momentul în care nu se mai pot obține alte drumuri *utile* prin extinderea drumurilor *utile* existente.
2. Inițial, există un singur drum, ce conține doar nodul *source* și are costul 0.
3. Un drum $x \rightsquigarrow y, z$ extinde un drum $x \rightsquigarrow y$ dacă $(y, z) \in E$ și $z \notin x \rightsquigarrow y$, unde $cost(x \rightsquigarrow y, z) = cost(x \rightsquigarrow y) + cost(y, z)$.
4. Un drum $x \rightsquigarrow y$ se numește *util* dacă nu există un alt drum $x \rightsquigarrow y$, mai ieftin. Drumurile *neutile* sunt imediat eliminate, în timpul explorării.

Drumurile optime într-un graf cu costuri III

Exemplul 43.4.

```
(deftemplate edge (slot from) (slot to) (slot cost))
(deftemplate find (slot source) (slot dest))
(deftemplate path (multislot nodes) (slot cost))

(defrule init
    (find (source ?s))
  =>
    (assert (path (nodes ?s) (cost 0))))
```


Drumurile optime într-un graf cu costuri IV

Exemplul 43.4.

```
(defrule expand
    (path (nodes $?prefix ?last) (cost ?pc))
    (edge (from ?last) (to ?neighbor) (cost ?ec))
    (test (and (neq ?neighbor ?last)
               (not (member ?neighbor $?prefix))))
  =>
    (assert (path (nodes $?prefix ?last ?neighbor)
                  (cost (+ ?pc ?ec)))))
```


Drumurile optime într-un graf cu costuri V

Exemplul 43.4.

```
(defrule prune
    (declare (salience 10))
    (path (nodes $? ?dest) (cost ?gc))
    ?f <- (path (nodes $? ?dest) (cost ?bc))
    (test (> ?bc ?gc))
  =>
    (retract ?f))

(defrule announce
    (declare (salience -10))
    (find (dest ?d))
    (path (nodes $?prefix ?d))
  =>
    (printout t $?prefix " " ?d crlf))
```


Drumurile optime într-un graf fără costuri I

- Criteriul optimizat: numărul de muchii
- Soluția 1: abordarea precedentă, presupunând că toate muchiile au costul 1
- Soluția 2: parcurgere în lățime

Drumurile optime într-un graf fără costuri II

Exemplul 43.5.

```
1  (deftemplate edge (slot from) (slot to) (slot cost))
2  (deftemplate find (slot source) (slot dest))
3
4  (defrule init
5      (find (source ?s))
6    =>
7      (assert (path ?s))
8      (set-strategy breadth))
```


Drumurile optime într-un graf fără costuri III

Exemplul 43.5.

```
10 (defrule expand
11     (path $?prefix ?last)
12     (edge (from ?last) (to ?neighbor))
13     (test (and  (neq ?neighbor ?last)
14                 (not (member ?neighbor $?prefix))))
15   =>
16     (assert (path $?prefix ?last ?neighbor)))
17
18 (defrule announce
19     (declare (salience 10))
20     (find (dest ?d))
21     (path $?prefix ?d)
22   =>
23     (printout t $?prefix ?d crlf) (halt))
```


Drumurile optime într-un graf fără costuri IV

- Parcurgere în lățime — extinderea implicită, într-un pas, a unei cele mai scurte căi (linia 8)
- Observație: vârsta superioară a faptelor reprezentând căi mai scurte
- Soluție: alterarea ordinii în care faptele sunt evaluate în raport cu șabloanele de identificare, ale regulilor

Drumurile optime într-un graf fără costuri V

Exemplul 43.6.

```
(defrule init
    (find (source ?s))
  =>
    (assert (path ?s))
    (set-strategy breadth))
```

Rezumat

- Stil declarativ, prin specificarea proprietăților soluției, și nu a modului în care aceasta este construită
- Explorare euristică, dinspre ipoteze către concluzie (*forward chaining*), prin opoziție cu Prolog, unde căutarea este orientată dinspre concluzie spre ipoteze, (*backward chaining*)
- Fapte, reguli
- Posibilități de control al execuției: *salience*, strategii, module (lectură suplimentară)