Paradigme de Programare

S.l. dr. ing. Andrei Olaru
slides: Mihnea Muraru si Andrei Olaru

Catedra de Calculatoare

2013 – 2013, semestrul 2

Cursul 1

Introducere

Cuprins

Organizare

Resurse de bază
unde găsesc informații?

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

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

Teme și forumuri: `http://cs.curs.pub.ro` (în curând)

Notare

- Laborator: 1p (cu bonusuri, max 1p total)
- Teme: 4p (4 $\times$ 1p) (cu bonusuri, max 6p pe parcurs)
- Teste la curs: 0,5p
- Test din materia de laborator: 0,5p
- Examen: 4p

Laborator

- Accent pe lucrul efectiv
- Parcurgerea documentaţiei înaintea laboratorului
- Test la începutul laboratorului

Obiective

Conținutul cursului
Ce vom studia?

1. Diverse perspective conceptuale asupra noţiunii de calculabilitate efectivă → modele de calculabilitate
2. Influenţa perspectivei alese asupra procesului de modelare şi rezolvare a problemelor → paradigme de programare
3. Limbaje de programare aferente paradigmelor, cu accent pe aspectul comparativ

De ce?
Just a thought

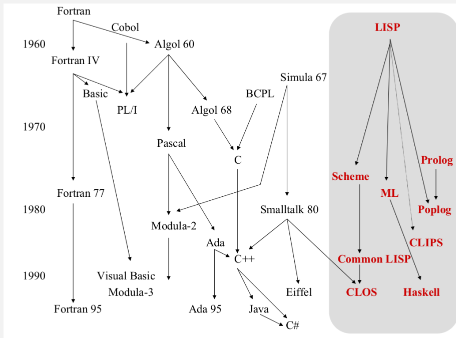
I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.

The law of instrument – Abraham Maslow

De ce?
Mai concret

- Lărgirea spectrului de abordare a problemelor
- Identificarea perspectivei naturale de modelare a unei probleme şi alegerea limbajului adecvat
- Sporirea capacităţii de învăţare a noi limbaje şi de adaptare la particularităţile şi diferenţele dintre acestea
- Exploatarea mecanismelor oferite de limbajele de programare

Limbaje de programare '50-'00
Un pic de istorie



Limitele calculabilităţii
Ce putem calcula și cum

- Teza Church-Turing: efectiv calculabil = Turing calculabil
- Echivalenţa celorlalte modele de calculabilitate – şi a multor altora – cu Maşina Turing
- Există vreun model mai expresiv?

Modele de calculabilitate
Modele → paradigme → limbaje

- Mașina Turing → Paradigma imperativă
  - → Procedurală → C
  - → Orientată-obiect → Java, C++
- Calcul Lambda → Paradigma funcțională → Scheme, Haskell
- Mașina Markov → Paradigma asociativă → CLIPS
- Mașina FOL → Paradigma logică → Prolog

Exemplu

O primă problemă

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

Modelare imperativă
Varianta procedurală

minList(L, n)

$min \leftarrow L[1]$
$i \leftarrow 2$
while $i \le n$ **do**
  if $L[i] < min$ **then**
    $min \leftarrow L[i]$
  end if
  $i \leftarrow i+1$
end while
return min

Modelare funcţională

- Ideea: $\text{minList}(L) = \text{if}(\text{eq}(\text{length}(L), 1), \text{head}(L), \text{min}(\text{head}(L), \text{minList}(\text{tail}(L))))$
- Scheme:

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

- Haskell:

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


Modelare asociativă

- Ideea: $\text{minList}(L) = m \in L \mid \nexists x \in L \bullet x < m$
- CLIPS:

```
(deffacts facts
    (elem 3)
    (elem 2)
    (elem 0)
    (elem 1))

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


Modelare logică

- Axiome:
  1. $x \le y \implies \text{min}(x, y, x)$
  2. $y < x \implies \text{min}(x, y, y)$
  3. $\text{minList}([m], m)$
  4. $\text{minList}([y|t], n) \wedge \text{min}(x, n, m) \implies \text{minList}([x, y|t], m)$
- Prolog:

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

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


Paradigme

Ce este o paradigmă de programare?

- Un set de convenţii ce dirijează maniera în care gândim programele
- Ea dictează modul în care:
  - reprezentăm datele
  - operaţiile prelucrează datele respective
- Atenție! Paradigma nu are legătură cu sintaxa limbajului!
- Există diferențe importante între paradigmele de programare. Vom discuta despre efecte laterale, transparență referențială și gestionarea funcțiilor în limbaj.

Efecte laterale (*side effects*)
Definiţie

Exemplul 4.1.

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

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

Definiția 4.2 (Efect lateral).

Pe lângă valoarea pe care o produce, o expresie sau o funcţie poate modifica starea globală.

- Inerente în situaţiile în care programul interacţionează cu exteriorul → I/O!

Efecte laterale (*side effects*)
Consecinţe

Exemplul 4.3.

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, obţinem `x + (x + 1) = 0 + 1 = 1`

- Adunare necomutativă?!
- Importanţa ordinii de evaluare!
- Dependenţe implicite, puţin lizibile şi posibile generatoare de bug-uri.

Transparenţă referenţială
Definiţie

Exemplul 4.4.

1. "**Zeus** este fiul lui Cronos"
   - Zeus este Jupiter în mitologia romană
   - "**Jupiter** este fiul lui Cronos" → aceeaşi semnificaţie
2. "Ionel ştie că **Zeus** este fiul lui Cronos"
   - "Ionel ştie că **Jupiter** este fiul lui Cronos" → altă semnificaţie

Definiția 4.5 (Transparenţă referenţială).
Confundarea unui obiect cu referinţa la acesta → cazul 1.

Transparenţă referenţială
Expresii

- Expresie transparentă referenţial: posedă o unică valoare, cu care poate fi substituită, păstrând semnificaţia programului.

Exemplul 4.6.

- `x-- + ++x` → nu, valoarea depinde de ordinea de evaluare
- `x = x + 1` → nu, două evaluări consecutive vor produce rezultate diferite
- `x` → ar putea fi, în funcţie de statutul lui `x` (globală, statică etc.)

- Absentă în prezenţa efectelor laterale!

Transparenţă referenţială
Funcţii

- Funcţie transparentă referenţial: rezultatul întors depinde exclusiv de parametri

Exemplul 4.7.

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

```
int g = 0;

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

- `opaque(3) - opaque(3) != 0`!
- Funcţii transparente: `log`, `sin` etc.
- Funcţii opace: `time`, `read` etc.

Transparenţă referenţială
Avantaje

- Lizibilitatea codului
- Demonstrarea formală a corectitudinii programului
- Optimizare prin reordonarea instrucţiunilor de către compilator şi prin caching
- Paralelizare masivă, prin eliminarea modificărilor concurente

Funcţii ca valori de prim rang
Definiţie

Definiția 4.8 (Valoare de prim rang).
O valoare ce poate fi:

- creată dinamic
- stocată într-o variabilă
- trimisă ca parametru unei funcţii
- întoarsă dintr-o funcţie

Exemplul 4.9.
Să se scrie funcţia `compose`, ce primeşte ca parametri alte 2 funcţii, `f` şi `g`, şi întoarce funcţia obţinută prin compunerea lor, $f \circ g$.

Funcţii ca valori de prim rang: `Compose`
C

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

- În C, funcţiile nu sunt valori de prim rang.

Funcţii ca valori de prim rang:
Java

```
abstract class Func<U, V> {
    public abstract V apply(U u);

    public <T> Func<T, V> compose(final Func<T, U> f) {
        final Func<U, V> outer = this;

        return new Func<T, V>() {
            public V apply(T t) {
                return outer.apply(f.apply(t));
            }
        };
    }
}
```

- În Java, funcţiile nu sunt valori de prim rang.

Funcţii ca valori de prim rang: `Compose`
Scheme & Haskell

- Scheme:

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

- Haskell:

```
compose = (.)
```

- În Scheme şi Haskell, funcţiile sunt valori de prim rang.

Funcţii ca valori de prim rang
Aplicaţii parţiale

Exemplul 4.10.

```
(define sum-uncurry
  (lambda (x y)
    (+ x y)))


(sum-uncurry 1 2)




;
```

```
(define sum-curry
  (lambda (x)
    (lambda (y)
      (+ x y))))

((sum-curry 1) 2)

(define sum-with-1
  (sum-curry 1))
(sum-with-1 2)
```


Funcţii ca valori de prim rang
Funcţii de ordin superior (funcţionale)

Definiţia 4.11 (Funcţională).
Funcţie care ia funcţii ca parametru şi/sau întoarce o funcţie.

Exemplul 4.12.

```
(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
```


Paradigma imperativă
Caracteristici

- Orientare spre acţiuni şi efectele acestora
- Cum se obţine soluţia
- Atribuirea ca operaţie fundamentală
- Efecte laterale permise, compromiţând transparenţa referenţială
- Secvenţierea instrucţiunilor
- Programe cu stare, văzută ca mulţimea valorilor variabilelor la un anumit moment, ce pot influenţa rezultatul evaluării aceleiaşi expresii

Paradigma funcţională
Caracteristici

- Funcţia văzută în sens matematic, exclusiv prin valoarea pe care o calculează
- Funcţii ca valori de prim rang
- Interzicerea efectelor laterale, pentru eliminarea dependenţelor implicite → modularitate sporită, la nivel de funcţie!
- Promovarea transparenţei referenţiale, alături de avantajele acesteia
- Diminuarea importanţei ordinii de evaluare
- Programe fără stare

Paradigma funcţională
Just a thought

> *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

Paradigmele asociativă şi logică
Caracteristici

- Accent pe formularea proprietăţilor soluţiei
- Ce trebuie obţinut (vs. “cum” la imperativă)
- Fapte, reguli, înlănţuire înainte/înapoi
- Orientare spre date

Aplicaţii ale diverselor paradigme

- Manipulare simbolică în inteligenţa artificială
  - Sisteme expert
  - Demonstrarea de teoreme
- Calcul paralel
- Demonstrarea automată a corectitudinii programelor şi testare, datorită modelului mai simplu de execuţie
- Adoptare a paradigmei funcţionale în limbajele noi: C#, F#, Python, JavaScript, Clojure (JVM), Scala
- Erlang (Ericsson) — limbaj funcţional utilizat în telecomunicaţii, economie, comerţ electronic

Limbaje

Câteva trăsături

- Paradigmă
  - Model de abordare a problemei
  - Model de execuție / calculabilitate
- Tipare
  - Statică/dinamică
  - Tare/slabă
- Ordinea de evaluare a parametrilor funcţiilor
  - Aplicativă
  - Normală
- Legarea variabilelor
  - Statică
  - Dinamică

Sfârșitul primului curs

Ce am învățat

· Paradigme de programare, limbaje, modele de calculabilitate, The law of instrument, efect lateral, transparență referențială, valori de prim rang, Scheme, Haskell, Prolog, CLIPS.

Cursul 2

Calcul Lambda

Cuprins

Introducere

Calculul Lambda

λ

- Model de calculabilitate (Alonzo Church, 1932) – introdus în cadrul cercetărilor asupra fundamentelor matematicii.
  [http://en.wikipedia.org/wiki/Lambda_calculus]
- Echivalent cu Maşina Turing (v. Teza Church-Turing)
  - Axat pe conceptul matematic de funcţie – totul este funcţie
  - Calculul $=$ evaluarea aplicaţiilor de funcţii, prin substituţie textuală
  - Evaluare = obţinerea unei valori $\rightarrow$ funcţie!
  - Absenţa efectelor laterale şi a stării
  - Model formal

Aplicaţii

ale calculului λ

- Aplicaţii importante în
  - programare
  - demonstrarea formală a corectitudinii programelor,

  datorită modelului simplu de execuţie
- Baza teoretică a numeroase limbaje:
  LISP, Scheme, Haskell, ML, F#, Clean, Clojure, Scala, Erlang etc.

λ-Expresii

λ-expresii
Definiţie

Definiția 7.1 (λ-expresie).

- Variabilă: o variabilă x este o λ-expresie
- Funcţie: dacă x este o variabilă şi E este o λ-expresie, atunci $\lambda x.E$ este o λ-expresie, reprezentând funcţia anonimă, unară, cu parametrul formal x şi corpul E
- Aplicaţie: dacă F şi A sunt λ-expresii, atunci $(F\ A)$ este o λ-expresie, reprezentând aplicaţia expresiei F asupra parametrului actual A

λ-expresii
Exemple

Exemplul 7.2.

1. $x \rightarrow$ variabila x
2. $\lambda x.x \rightarrow$ funcţia identitate
3. $\lambda x.\lambda y.x \rightarrow$ funcţie selector, cu altă funcţie drept corp!
4. $(\lambda x.x\ y) \rightarrow$ aplicaţia funcţiei identitate asupra parametrului actual y
5. $(\lambda x.(x\ x)\ \lambda x.x)$

- Intuitiv, evaluarea aplicaţiei $(\lambda x.x\ y)$ presupune substituirea lui x, în corp, prin $y \rightarrow$ rezultat y

Apariţii ale variabilelor
Definiţii

Definiția 7.3 (Apariţie legată).

O apariţie 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
- $E = \ldots\ \lambda x.F\ \ldots$ şi x_n apare în F.

Definiția 7.4 (Apariţie liberă).

O apariţie a unei variabile este liberă într-o expresie dacă nu este legată în acea expresie.

- Atenţie! În raport cu o expresie dată!

Variabile
Definiţii

Definiția 7.5 (Variabilă legată).

O variabilă este legată într-o expresie dacă toate apariţiile sale sunt legate în acea expresie.

Definiția 7.6 (Variabilă liberă).

O variabilă este liberă într-o expresie dacă nu este legată în acea expresie i.e. dacă cel puţin o apariţie a sa este liberă în acea expresie.

Definiția 7.7 (Variabilă de legare).

Parametrul formal, x, al funcţiei $\lambda x.E$.

- Atenţie! În raport cu o expresie dată!

[Apariţii ale]variabilelor
Exemplu 1

Exemplul 7.8.

În expresia $E = (\lambda x.x\ x)$, evidenţiem apariţiile lui x:
$(\lambda x_1.\underbrace{x_2}_{F}\ x_3)$.

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

[Apariţii ale]variabilelor
Exemplu 2

Exemplul 7.9.

În expresia $E = (\lambda x.\lambda z.(z\ x)\ (z\ y))$, evidenţiem apariţiile:
$(\lambda x_1.\underbrace{\lambda z_1.(z_2\ x_2)}_{F}\ (z_3\ y_1))$.

- 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 şi legate
O abordare formală

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 7.10 (Expresie închisă).

Expresie ce nu conţine variabile libere.

Exemplul 7.11.

- $(\lambda x.x\ \lambda x.\lambda y.x) \rightarrow$ închisă
- $(\lambda x.x\ a) \rightarrow$ deschisă, deoarece a este liberă

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

Reducere

β-reducere
Definiţii

Definiția 8.1 (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:
$(\lambda x.E\ A) \rightarrow_\beta E_{[A/x]}$.

Definiția 8.2 (β-redex).
Expresia $(\lambda x.E\ A)$ – o expresie pe care se poate aplica β-reducerea.

β-reducere
Exemple

Exemplul 8.3.
- $(\lambda x.x\ y) \rightarrow_\beta x_{[y/x]} \rightarrow y$
- $(\lambda x.\lambda x.x\ y) \rightarrow_\beta \lambda x.x_{[y/x]} \rightarrow \lambda x.x$
- $(\lambda x.\lambda y.x\ y) \rightarrow_\beta \lambda y.x_{[y/x]} \rightarrow \lambda y.y$ Greşit! Variabila liberă y devine legată, schimbându-şi semnificaţia. $\rightarrow \lambda y^{(a)}.y^{(b)}$
  Care este problema? (formal)

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 potenţial greşită
- Soluţie: redenumirea variabilelor legate din E, ce coincid cu cele libere din A.

Exemplul 8.4.
$(\lambda x.\lambda y.x\ y) \rightarrow_\alpha (\lambda x.\lambda z.x\ y) \rightarrow_\beta \lambda z.x_{[y/x]} \rightarrow \lambda z.y$

α-conversie
Definiţie

Definiția 8.5 (α-conversie).
Redenumirea sistematică a variabilelor legate dintr-o funcţie: $\lambda x.E \rightarrow_\alpha \lambda y.E_{[y/x]}$. Se impun două condiţii.

Exemplul 8.6.
- $\lambda x.y \rightarrow_\alpha \lambda y.y_{[y/x]} \rightarrow \lambda y.y \rightarrow$ Greşit!
- $\lambda x.\lambda y.x \rightarrow_\alpha \lambda y.\lambda y.x_{[y/x]} \rightarrow \lambda y.\lambda y.y \rightarrow$ Greşit!

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

α-conversie
Exemple

Exemplul 8.7.
- $\lambda x.(x\ y) \rightarrow_\alpha \lambda z.(z\ y) \rightarrow$ Corect!
- $\lambda x.\lambda x.(x\ y) \rightarrow_\alpha \lambda y.\lambda x.(x\ y) \rightarrow$ Greşit! y este liberă în $\lambda x.(x\ y)$
- $\lambda x.\lambda y.(y\ x) \rightarrow_\alpha \lambda y.\lambda y.(y\ y) \rightarrow$ Greşit! Apariţia liberă a lui x din $\lambda y.(y\ x)$ devine legată, după substituire, în $\lambda y.(y\ y)$
- $\lambda x.\lambda y.(y\ y) \rightarrow_\alpha \lambda y.\lambda y.(y\ y) \rightarrow$ Corect!

Reducere
Definiţii

Definiția 8.8 (Pas de reducere).
O secvenţă formată dintr-o α-conversie şi o β-reducere, astfel încât a doua se produce fără coliziuni:
$E_1 \rightarrow E_2 \equiv E_1 \rightarrow_\alpha E_3 \rightarrow_\beta E_2$.

Definiția 8.9 (Secvenţă de reducere).
Succesiune de zero sau mai mulţi paşi de reducere:
$E_1 \rightarrow^* E_2$. Reprezintă un element din închiderea reflexiv-tranzitivă a relaţiei $\rightarrow$.

Reducere
Proprietăţi

- $E_1 \rightarrow E_2 \implies E_1 \rightarrow^* E_2$
- $E \rightarrow^* E$
- $E_1 \rightarrow^* E_2 \wedge E_2 \rightarrow^* E_3 \implies E_1 \rightarrow^* E_3$

Exemplul 8.10.
- $((\lambda x.\lambda y.(y\ x)\ y)\ \lambda x.x) \rightarrow (\lambda z.(z\ y)\ \lambda x.x) \rightarrow (\lambda x.x\ y) \rightarrow y$
- $((\lambda x.\lambda y.(y\ x)\ y)\ \lambda x.x) \rightarrow^* y$

Forme normale

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
Definiții

Definiția 9.1 (Formă normală).
Formă a unei expresii, ce nu mai conține β-redecși i.e. care nu mai poate fi redusă.

Definiția 9.2 (Formă normală funcțională – FNF).
$\lambda x.F$, chiar dacă F conține β-redecși.

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

Exemplul 9.3.
$(\lambda x.\lambda y.(x\ y)\ \lambda x.x) \to_{FNF} \lambda y.(\lambda x.x\ y) \to_{FN} \lambda y.y$

Terminarea reducerii (reductibilitate)
Exemplu și definiție

Exemplul 9.4.
$\Omega = (\lambda x.(x\ x)\ \lambda x.(x\ x)) \to (\lambda x.(x\ x)\ \lambda x.(x\ x)) \to^* \ldots$
Ω nu admite o secvență de reducere, care se termină.

Definiția 9.5 (Expresie reductibilă).
Expresie ce admite o secvență de reducere, care se termină.

Secvențe de reducere

Exemplul 9.6.
$E = (\lambda x.y\ \Omega)$
$\to y$
$\to E \to y$
$\to E \to E \to y$
$\ldots$
$\xrightarrow{n}{}^* y,\ n \geq 0$
$\xrightarrow{\infty}{}^* \ldots$

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

Unicitatea formei normale
Rezultate

Teorema 9.7 (Church-Rosser / diamantului).
Dacă $E \to^* E_1$ și $E \to^* E_2$, atunci există E_3 astfel încât $E_1 \to^* E_3$ și $E_2 \to^* E_3$.

$$E \begin{array}{l} \to^* E_1 \to^* \\ \to^* E_2 \to^* \end{array} E_3$$

Corolarul 9.8.
Dacă o expresie este reductibilă, forma ei normală este unică. Ea corespunde valorii expresiei.

Unicitatea formei normale
Exemple

Exemplul 9.9.
$(\lambda x.\lambda y.(x\ y)\ (\lambda x.x\ y))$

- $\to \lambda z.((\lambda x.x\ y)\ z) \to \lambda z.(y\ z) \to_\alpha \lambda a.(y\ a)$
- $\to (\lambda x.\lambda y.(x\ y)\ y) \to \lambda w.(y\ w) \to_\alpha \lambda a.(y\ a)$

- Forma normală corespunde unei clase de expresii, echivalente sub redenumiri sistematice.
- Valoarea este un anumit membru al acestei clase.

Modalități de reducere

Definiția 9.10 (Reducere stânga-dreapta).
Reducerea celui mai superficial și mai din stânga β-redex.

Exemplul 9.11.
$(\underline{(\lambda x.x\ \lambda x.y)}\ (\lambda x.(x\ x)\ \lambda x.(x\ x))) \to \underline{(\lambda x.y\ \Omega)} \to y$

Definiția 9.12 (Reducere dreapta-stânga).
Reducerea celui mai adânc și mai din dreapta β-redex.

Exemplul 9.13.
$((\lambda x.x\ \lambda x.y)\ \underline{(\lambda x.(x\ x)\ \lambda x.(x\ x))}) \to (\lambda x.y\ \underline{\Omega}) \to \ldots$

Ce modalitate alegem?

Teorema 9.14 (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!
- Dacă expresia este ireductibilă, nicio reducere nu se va termina.

Evaluare

Ordini de evaluare
Diferă în diferite limbaje

Definiția 10.1 (Evaluare aplicativă – *eager*).
Corespunde reducerii dreapta-stânga. Parametrii funcţiilor sunt evaluaţi înaintea aplicării funcţiei.

Definiția 10.2 (Funcţie strictă).
Funcţie cu evaluare aplicativă.

Definiția 10.3 (Evaluare normală – *lazy*).
Corespunde reducerii stânga-dreapta. Parametrii funcţiilor sunt evaluaţi la cerere.

Definiția 10.4 (Funcţie nestrictă).
Funcţie cu evaluare normală.

În practică

- Evaluarea aplicativă prezentă în majoritatea limbajelor: C, Java, Scheme, PHP etc.

Exemplul 10.5.
`(+ (+ 2 3) (* 2 3))` $\rightarrow$ `(+ 5 6)` $\rightarrow$ `11`

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

Exemplul 10.6.
`(if (< 2 3) (+ 2 3) (* 2 3))` $\xrightarrow{\texttt{(< 2 3)} \rightarrow \texttt{\#t}}$ `(+ 2 3)` $\rightarrow$ `5`

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 10.7.

```
1 // C sau Java
2 void f(int x) {
3     x = 3;
4 }
```

```
1 // C
2 void g(struct str s) {
3     s.member = 3;
4 }
```

Efectul liniilor 3 este invizibil la apelant.

- Evaluarea parametrilor înaintea aplicaţiei funcţiei şi transferul unei copii a valorii acestuia
- Modificări locale invizibile la apelant
- C, C++, tipurile primitive în Java

Call by sharing

- Variantă a *call by value*
- Trimiterea unei referinţe la obiect
- Modificări locale asupra referinţei invizibile la apelant
- Modificări locale asupra obiectului referit vizibile la apelant
- Scheme, tipurile referinţă în Java
- Diferenţă faţă de C, unde o structură trimisă ca parametru este complet copiată

Call by reference

- Trimiterea unei referinţe la obiect
- Modificări locale asupra referinţei şi obiectului referit vizibile la apelant
- Folosirea & în C++

Call by name

- Argumente neevaluate în momentul aplicării funcţiei $\rightarrow$ substituţie directă (textuală) în corpul funcţiei
- Evaluare parametrilor la cerere, de fiecare dată când este nevoie de valoarea acestora

Call by need

- Variantă a *call by name*
- Evaluarea unui parametru doar la prima utilizare a acestuia
- Memorarea valorii unui parametru deja evaluat şi returnarea acesteia în cazul utilizării repetate a aceluiaşi parametru
- în Haskell

Sfârșitul cursului 2

Ce am învățat

· Baza formală a calculului λ, expresie λ, legat vs. liber, expresie închisă, α-conversie, β-reducere, FN și FNF, reductibilitate, evaluare aplicativă și normală, call by $*$.

Cursul 3

Calculul Lambda ca limbaj de programare

Cuprins

Limbajul λ_0

Limbajul λ_0

Scop

- Demonstrarea puterii expresive a Calculului Lambda;
- Considerăm o Maşină $\leftarrow$ ipotetică;
- λ-expresii = cod maşină $\rightarrow$ limbajul λ_0 este codul mașină pentru mașina de calcul $\leftarrow$;
- Locul
  - tipurilor de date simple (e.g. numere, etc.)
  - operaţiilor de bază (adunare, etc.)

  luat de
  - şiruri structurate de simboli – expresii $\leftarrow$
  - reducere — substituţie textuală

Limbajul λ_0 – Convenţii

Instrucțiuni

· Instrucţiuni:
- λ-expresii
- legări de variabile *top-level*: *variabila* $\equiv_{\text{def}}$ *expresie*.

Exemplul 11.1.
$true \equiv_{\text{def}} \lambda x.\lambda y.x$

· Valorile sunt reprezentate de funcţii

· Se folosesc numai expresii închise

Exemplul 11.2.
$length \equiv_{\text{def}} \lambda L.(if\ (null\ L)\ 0\ (succ\ (length\ (cdr\ L)))) \notin \lambda_0$!

Limbajul λ_0 – Convenţii

Evaluare

- Scrieri prescurtate:
  - $\lambda x_1.\lambda x_2.\ldots\lambda x_n.E \rightarrow \lambda x_1 x_2 \ldots x_n.E$
  - $((\ldots((E\ A_1)\ A_2)\ \ldots)\ A_n) \rightarrow (E\ A_1\ A_2\ \ldots\ A_n)$

Exemplul 11.3.

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

- Evaluare în ordine normală (stânga $\rightarrow$ dreapta)
- Forma normală funcţională (vezi Definiţia 9.2) – $\lambda x.F$
- Absenţa tipurilor explicite!

Limbajul λ_0 – Convenţii

Egalitate

- $E_1 = E_2$ dacă:
  - $E_1 \rightarrow^* E_2$ sau $E_2 \rightarrow^* E_1$ – structural egale *sau*
  - $\forall A_1, \ldots, A_n \bullet (E_1\ A_1\ \ldots\ A_n) = (E_2\ A_1\ \ldots\ A_n)$ – comportamental egale

Exemplul 11.4.

$E_1 = \lambda x.\lambda y.(x\ y)$
$E_2 = \lambda x.x$

- Structural: nu sunt egale
- Comportamental: egale
  $((E_1\ a)\ b) \rightarrow (\lambda y.(a\ y)\ b) \rightarrow (a\ b)$
  $((E_2\ a)\ b) \rightarrow (a\ b)$

Absenţa tipurilor

Chiar avem nevoie de tipuri? – Rolul tipurilor

- Modalitate de exprimare a intenţiei programatorului
- Susţinători ai abstractizării
- Documentare: ce operatori acţionează asupra căror obiecte
- Reprezentarea particulară a valorilor de tipuri diferite: `1`, `‘‘Hello’’`, `#t` etc.
- Optimizarea operaţiilor specifice
- Prevenirea erorilor
- Facilitarea verificării formale

Absenţa tipurilor

Consecințe asupra reprezentării obiectelor

- Un număr, o listă sau un arbore, posibil desemnate de aceeaşi valoare!
- Valori şi operatori reprezentaţi de funcţii, semnificaţia fiind dependentă de context
- Valoare aplicabilă asupra unei alte valori $\rightarrow$ operator!

Absenţa tipurilor

Consecințe asupra corectitudinii calculului

- Incapacitatea Maşinii λ de a
  - interpreta semnificaţia expresiilor
  - asigura corectitudinea acestora (dpdv al tipurilor)
- Delegarea celor două aspecte programatorului
- Orice operatori aplicabili asupra oricăror valori
- Construcţii eronate acceptate fără avertisment, dar calcule terminate cu
  - valori fără semnificaţie *sau*
  - expresii care nu sunt valori, dar sunt ireductibile

  $\rightarrow$ instabilitate

Absenţa tipurilor

Consecinţe pozitive

- Flexibilitate sporită în reprezentare
- Potrivită în situaţiile în care reprezentarea uniformă obiectelor, ca liste de simboli, este convenabilă

. . . vin cu preţul unei dificultăţi sporite în depanare, verificare şi mentenanţă

TDA – Definiție

TDA

Definiție

Definiția 12.1 (Tip de date abstract – TDA).

Model matematic al unei mulţimi de valori şi al operaţiilor valide pe acestea.

Exemplul 12.2 (TDA).

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 lucrează operatorii / ce restricții există

TDA *Natural*
Specificare – exemplu

- Constructori:
  $zero: \rightarrow Natural$
  $succ: Natural \rightarrow Natural$

- Operatori:
  $pred: Natural \setminus \{zero\} \rightarrow Natural$
  $zero?: Natural \rightarrow Bool$
  $add: Natural^2 \rightarrow Natural$

- Axiome:
  $pred: \quad pred(succ(n)) = n$
  $zero?: \quad zero?(zero) = T$
  $\qquad zero?(succ(n)) = F$
  $add: \quad add(zero, n) = n$
  $\qquad add(succ(m), n) = succ(add(m, n))$

Scrierea axiomelor
Câte axiome ne trebuie?

- Câte o axiomă pentru fiecare pereche (operator, constructor de bază) (aproximativ)
- Definiţii suplimentare → inutile
- Definiţii mai puţine → insuficiente pentru specificarea completă a comportamentului operatorilor

De la TDA la programare funcţională
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 funcţională
Discuţie

- Demonstrarea corectitudinii TDA → inducţie structurală
- Demonstrarea proprietăţilor λ-expresiilor, văzute ca un TDA cu 3 constructori de bază!
- Programarea funcţională → reflectarea specificaţiilor matematice
- Recursivitatea → instrument natural, moştenit din axiome
- Aplicarea procedeelor formale pe codul recursiv, exploatând absenţa efectelor laterale

TDA – Implementare în λ_0

TDA *Bool*
Specificare

- Constructori:
  $T: \rightarrow Bool$
  $F: \rightarrow Bool$

- Operatori:
  $not: Bool \rightarrow Bool$
  $and: Bool^2 \rightarrow Bool$
  $or: Bool^2 \rightarrow Natural$

- Axiome:
  $not: \quad not(T) = F$
  $\qquad not(F) = T$
  $and: \quad and(T, a) = a$
  $\qquad and(F, a) = F$
  $or: \quad or(T, a) = T$
  $\qquad or(F, a) = a$

TDA *Bool*
Implementarea constructorilor de bază

- Intuiţie: selecţia între cele două valori, *true* şi *false*
- $T \equiv_{\text{def}} \lambda x\, y.x$
- $F \equiv_{\text{def}} \lambda x\, y.y$
- Comportament de selectori:
  - $(T\ a\ b) \rightarrow (\lambda x\, y.x\ a\ b) \rightarrow a$
  - $(F\ a\ b) \rightarrow (\lambda x\, y.y\ a\ b) \rightarrow b$

TDA *Bool*
Implementarea operatorilor

- $not \equiv_{\text{def}} \lambda x.(x\ F\ T)$
  - $(not\ T) \rightarrow (\lambda x.(x\ F\ T)\ T) \rightarrow (T\ F\ T) \rightarrow F$
  - $(not\ F) \rightarrow (\lambda x.(x\ F\ T)\ F) \rightarrow (F\ F\ T) \rightarrow T$
- $and \equiv_{\text{def}} \lambda x\, y.(x\ y\ F)$
  - $(and\ T\ a) \rightarrow (\lambda x\, y.(x\ y\ F)\ T\ a) \rightarrow (T\ a\ F) \rightarrow a$
  - $(and\ F\ a) \rightarrow (\lambda x\, y.(x\ y\ F)\ F\ a) \rightarrow (F\ a\ F) \rightarrow F$
- $or \equiv_{\text{def}} \lambda x\, y.(x\ T\ y)$
  - $(or\ T\ a) \rightarrow (\lambda x\, y.(x\ T\ y)\ T\ a) \rightarrow (T\ T\ a) \rightarrow T$
  - $(or\ F\ a) \rightarrow (\lambda x\, y.(x\ T\ y)\ F\ a) \rightarrow (F\ T\ a) \rightarrow a$

TDA *Bool*
Condiţionala *if*

- Axiome:
  - $(if\ T\ a\ b) \rightarrow a$
  - $(if\ F\ a\ b) \rightarrow b$
- Implementare: $if \equiv_{\text{def}} \lambda c\ t\ e.(c\ t\ e)$
  - $(if\ T\ a\ b) \rightarrow (\lambda c\ t\ e.(c\ t\ e)\ T\ a\ b) \rightarrow (T\ a\ b) \rightarrow a$
  - $(if\ F\ a\ b) \rightarrow (\lambda c\ t\ e.(c\ t\ e)\ F\ a\ b) \rightarrow (F\ a\ b) \rightarrow b$
- Funcţie nestrictă!

TDA *Pair*
Specificare

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

TDA *Pair*
Implementare

- Intuiţie: pereche $\rightarrow$ funcţie ce aşteaptă selectorul, pentru a-l aplica asupra membrilor
- $pair \equiv_{\text{def}} \lambda x\ y\ z.(z\ x\ y)$
  - $(pair\ a\ b) \rightarrow (\lambda x\ y\ z.(z\ x\ y)\ a\ b) \rightarrow \lambda z.(z\ a\ b)$
- $fst \equiv_{\text{def}} \lambda p.(p\ T)$
  - $(fst\ (pair\ a\ b)) \rightarrow (\lambda p.(p\ T)\ \lambda z.(z\ a\ b)) \rightarrow (\lambda z.(z\ a\ b)\ T) \rightarrow (T\ a\ b) \rightarrow a$
- $snd \equiv_{\text{def}} \lambda p.(p\ F)$
  - $(snd\ (pair\ a\ b)) \rightarrow (\lambda p.(p\ F)\ \lambda z.(z\ a\ b)) \rightarrow (\lambda z.(z\ a\ b)\ F) \rightarrow (F\ a\ b) \rightarrow b$

TDA *List*
Specificare

- Constructori:
  - $nil : \ \rightarrow List$
  - $cons : \ A \times List \rightarrow List$
- Operatori:
  - $car : \ List \setminus \{nil\} \rightarrow A$
  - $cdr : \ List \setminus \{nil\} \rightarrow List$
  - $null : \ List \rightarrow Bool$
  - $append : \ List^2 \rightarrow List$
- Axiome:
  - $car : \ car(cons(e,L)) = e$
  - $cdr : \ cdr(cons(e,L)) = L$
  - $null : \ null(nil) = T$
    - $null(cons(e,L)) = F$
  - $append : \ append(nil,B) = B$
    - $append(cons(e,A),B) = cons(e, append(A,B))$

TDA *List*
Implementare

- Intuiţie: listă $\rightarrow$ pereche (*head*, *tail*)
- $nil \equiv_{\text{def}} \lambda x.T$
- $cons \equiv_{\text{def}} pair$
  - $(cons\ e\ L) \rightarrow (\lambda x\ y\ z.(z\ x\ y)\ e\ L) \rightarrow \lambda z.(z\ e\ L)$
- $car \equiv_{\text{def}} fst$
- $cdr \equiv_{\text{def}} snd$
- $null \equiv_{\text{def}} \lambda L.(L\ \lambda x\ y.F)$
  - $(null\ nil) \rightarrow (\lambda L.(L\ \lambda x\ y.F)\ \lambda x.T) \rightarrow (\lambda x.T\ \ldots) \rightarrow T$
  - $(null\ (cons\ e\ L)) \rightarrow (\lambda L.(L\ \lambda x\ y.F)\ \lambda z.(z\ e\ L)) \rightarrow (\lambda z.(z\ e\ L)\ \lambda x\ y.F) \rightarrow (\lambda x\ y.F\ e\ L) \rightarrow F$
- $append \equiv_{\text{def}}$ $\lambda A\ B.(if\ (null\ A)\ B\ (cons\ (car\ A)\ (append\ (cdr\ A)\ B)))$
- Problemă: expresia nu admite formă închisă!

TDA *Natural*
Implementare

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

Recursivitate

Funcţii

- Definiţii ale funcţiei identitate:
  - $id(n) = n$
  - $id(n) = n + 1 - 1$
  - $id(n) = n + 2 - 2$
  - $\ldots$
- O infinitate de reprezentări textuale pentru aceeaşi funcţie
- Atunci... ce este o funcţie? $\rightarrow$ o relaţie între valori, independentă de reprezentările textuale
  - $id = \{(0,0),(1,1),(2,2),\ldots\}$

Perspective asupra recursivităţii

- Textuală: funcţie care se autoapelează, folosindu-şi numele
- Constructivistă: funcţii recursive ca valori ale unui TDA, cu precizarea modalităţilor de generare a acestora
- Semantică: ce obiect matematic este desemnat de o funcţie recursivă, cu posibilitatea construirii de funcţii recursive anonime

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 funcţie echivalentă computaţional cu *length*?
  $Length \equiv_{def} \lambda f\ L.(if\ (null\ L)\ zero\ (succ\ (f\ (cdr\ L))))$
- $(Length\ length) = length \rightarrow length$ este un punct fix al lui *Length*!
- Cum obţinem punctul fix?

Puncte fixe

Definiția 14.1 (Punct fix).
f este un punct fix al funcţiei F dacă $(F\ f) = f$.

Exemplul 14.2.
$Fix = \lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x)))$

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

Definiția 14.3 (Combinator de punct fix).
Funcţie ce generează un punct fix al oricărei expresii.
Exemplu: *Fix*.

Implementare *length*

Soluţie

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

Combinatori de punct fix

- Pentru funcţii unare – *length*
  $c_1 \equiv_{def} \lambda f.(\lambda g\ x.(f\ (g\ g)\ x)\ \lambda g\ x.(f\ (g\ g)\ x))$
- Pentru funcţii binare – *append*
  $c_2 \equiv_{def} \lambda f.(\lambda g\ x\ y.(f\ (g\ g)\ x\ y)\ \lambda g\ x\ y.(f\ (g\ g)\ x\ y))$

Sfârșitul cursului 3

Ce am învățat

· Limbajul λ_0 – instrucțiuni, discuție despre tipurile de date; construcția tipurilor de date abstracte folosind constructori de bază, operatori și axiomeș folosirea punctelor fixe pentru eliminarea recursivității textuale; combinatori de punct fix.

Cursul 4

Programare funcţională în Scheme

Cuprins

Introducere

Deosebiri faţă de λ_0

- Tipat – dinamic/latent
  - Variabilele nu au tip
  - Valorile au tip (`3`, `#f`)
  - Verificarea se face la execuţie, în momentul aplicării unei funcţii (evaluare aplicativă)
- Permite recursivitate textuală
- Avem domenii de vizibilitate a variabilelor

Tipare

Modalităţi de tipare

- Rolul tipurilor: exprimare a intenției programatorului, abstractizare, documentare, optimizare, verificare
- Clasificare după momentul verificării:
  - statică
  - dinamică
- Clasificare după rigiditatea regulilor:
  - tare
  - slabă

Tipare statică vs. dinamică

Tipare statică:

- La compilare
- Valori şi variabile
- Rulare mai rapidă
- Rigidă: sancţionează orice construcţie
- Debugging mai facil
- Declaraţii explicite sau inferenţe de tip
- Pascal, C, C++, Java, Haskell

Tipare dinamică:

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

Tipare tare vs. slabă

- Clasificare după libertatea de a agrega valori de tipuri diferite.

Exemplul 16.1 (Tipare tare).

`1 + "23"` → Eroare (Haskell)

Exemplul 16.2 (Tipare slabă).

`1 + "23" = 24` (Visual Basic)
`1 + "23" = "123"` (JavaScript)

Tiparea în Scheme

- este dinamică

Exemplul 16.3.

```
(if #t 'something (+ 1 #t)) → 'something
(if #f 'something (+ 1 #t)) → Eroare
```

- este tare

Exemplul 16.4.

```
(+ "1" 2) → Eroare
```

- Permite liste cu elemente de tipuri diferite.

Variabile

Variabile
Proprietăți

- Proprietăţi
  - tip – nu! (în Scheme)
  - identificator
  - valoarea legată (la un anumit moment)
  - domeniul de vizibilitate + durata de viaţă
- Stări
  - declarată: cunoaştem identificatorul
  - definită: cunoaştem şi valoarea

Exemplul 17.1.

```
int f(int x) {
    int y = 0; // definitie
    // domeniul de vizibilitate a lui y
}
```


Legarea variabilelor

Definiția 17.2 (Legarea variabilelor).
Modalitatea de asociere a apariţiei unei variabile cu definiţia acesteia.

Definiția 17.3 (Domeniu de vizibilitate – *scope*).
Mulţimea punctelor din program unde o definiţie este vizibilă. Este determinat de modalitatea de legare a variabilelor.

- Modalităţi de legare:
  - statică
  - dinamică

Legarea statică a variabilelor

Definiția 17.4 (Legare statică / lexicală).
Variabilele din corpul unei expresii sunt extrase din contextul în care aceasta a fost definită.

- Domeniu de vizibilitate determinat prin construcţiile limbajului, putând fi desprins la compilare

Exemplul 17.5.
Care sunt domeniile de vizibilitate a variabilelor de legare, în expresia $\lambda x.\lambda y.(\lambda x.x\ y)$?
$\lambda \underline{x}.\lambda y.(\lambda x.x\ y)$ | $\lambda x.\lambda \underline{y}.(\lambda x.x\ y)$ | $\lambda x.\lambda y.(\lambda \underline{x}.x\ y)$

Legarea dinamică a variabilelor

Definiția 17.6 (Legare dinamică).
Valorile variabilelor depind de momentul în care o expresie este evaluată.

- Domeniu de vizibilitate determinat la execuţie

Exemplul 17.7 (! Artificial).

```
int x = 0;
int f() { return x; }
int g(int x) { return f(); }
```

Ce va returna `g(2)`?
`x=0` → `g(2)` → `x=2` → `f()` → `2` (ultima valoare!)

Legarea variabilelor în Scheme

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

Construcţia `lambda`
Definiţie & Exemplu

- Leagă static parametrii formali ai unei funcţii
- Sintaxă:

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

- Domeniul de vizibilitate a parametrului `pk`: mulţimea punctelor din corpul funcţiei – `expr`, în care apariţiile lui `pk` sunt libere (v. Exemplul 17.5)

Exemplul 17.8.

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


Construcţia `lambda`
Semantică

- Aplicaţie:

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

- Evaluare aplicativă: se evaluează argumentele `ak`, în ordine aleatoare
- Se evaluează corpul funcţiei, `expr`, ţinând cont de legările `pk` ← *valoare*(`ak`)
- Valoarea aplicaţiei este valoarea lui `expr`

Construcţia `let`
Definiţie & Exemplu

- Leagă static variabile locale
- Sintaxă:

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

- Domeniul de vizibilitate a variabilei `vk` = mulţimea punctelor din corp – `expr`, în care apariţiile lui `vk` sunt libere (v. Exemplul 17.5)

Exemplul 17.9.

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

Construcţia `let`
Semantică

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

echivalent cu

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


Construcţia `let*`
Definiţie & Exemplu

- Leagă static variabile locale
- Sintaxă:

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

- Scope pentru variabila `vk` = mulţimea punctelor din
  - restul legărilor (legări anterioare) şi
  - corp – `expr`

  în care apariţiile lui `vk` sunt libere

Exemplul 17.10.

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


Construcţia `let*`
Semantică

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

echivalent cu

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

- Evaluarea expresiilor `ei` se face în ordine!

Construcţia `letrec`
Definiţie

- Leagă static variabile locale
- Sintaxă:

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

- Domeniul de vizibilitate a variabilei `vk` = mulţimea punctelor din întreaga construcţie, în care apariţiile lui `vk` sunt libere

Construcţia `letrec`
Exemplu

Exemplul 17.11.

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


Construcţia `define`
Definiţie & Exemplu

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

Exemplul 17.12.

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

Output: `0 1`

- Avantaje:
  - definirea variabilelor *top-level* în orice ordine
  - definirea de funcţii mutual recursive
- Dezavantaj: corupera transparenţei referenţiale

Construcţia `define`
Exemplu obscur

Exemplul 17.13.

```
(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`

Legarea variabilelor în Scheme
Exemplu mixt

Exemplul 17.14 (Variantă a Exemplului 17.7).

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

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

(g 2)
```

Output: `1`

Evaluare

Evaluarea în Scheme

- Evaluare aplicativă: evaluarea parametrilor înaintea aplicării funcţiei asupra acestora (în ordine aleatoare)
- Transferul parametrilor: *call by sharing* – variantă a *call by value*
- Funcţii stricte (i.e.cu evaluare aplicativă)
- Excepţii: `if`, `cond`, `and`, `or`, `quote`

Contexte computaţionale
Definiţie

Definiția 18.1 (Context computaţional).
Contextul computaţional al unui punct P, dintr-un program, la momentul t, este mulţimea variabilelor ale căror domenii de vizibilitate îl conţin pe P, la momentul t.

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

Contexte computaţionale
Exemplu

Exemplul 18.2.
Ce variabile locale conţine contextul computaţional al punctului P?

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


Închideri funcţionale
Motivaţie

- λ_0: evaluarea → substituţie textuală

Exemplul 18.3.
$((\lambda f.\lambda g.\lambda x.(f\ (g\ x))\ g_1)\ f_1) \rightarrow (\lambda g.\lambda x.(f_1\ (g\ x))\ g_1)$
$\rightarrow \lambda x.(f_1\ (g_1\ x))$

- Ineficienţa practică a procesului de substituţie
- Alternativă: salvarea contextului unei funcţii, în momentul creării acesteia
- Legarea variabilelor libere în contextul salvat → închidere funcţională

Închideri funcţionale
Definiţie

Definiția 18.4 (Închidere funcţională).
Funcţie care îşi salvează contextul, pe care îl va folosi, în momentul aplicării, pentru evaluarea corpului.

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

Exemplul 18.5.
$< \lambda x.z;\ \{z \leftarrow 2\} >$

· Utilizate în cazul legării statice!

Închideri funcţionale
Exemplu

Exemplul 18.6.

```
(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 funcţionale
Explicaţia exemplului

- *comp-inc* $\equiv < \lambda g.\lambda x.(f\ (g\ x));\ \{f \leftarrow \lambda x.(+\ x\ 1)\} >$
- *comp-inc-double* $\equiv$ $< \lambda x.(f\ (g\ x));\ \{f \leftarrow \lambda x.(+\ x\ 1),\ g \leftarrow \lambda x.(*\ x\ 2)\} >$
- Inutilitatea redefinirii lui `inc`: valoarea sa fusese deja salvată în context, în momentul aplicării

Controlul evaluării

- `quote` sau `'`
  - funcţie nestrictă
  - întoarce parametrul neevaluat
- `eval`
  - funcţie strictă
  - forţează evaluarea parametrului şi întoarce valoarea acestuia

Exemplul 18.7.

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


Efecte laterale

Construcţia `set!`

- Modifică valoarea unei variabile

Exemplul 19.1.

```
(define x 0)
(define f (lambda (p)
    (set! x p)
    x))
(f 3) ; 3
x ; 3
```

- Diferenţă la nivel de intenţie faţă de `let`-uri şi `define`, care urmăresc definirea de variabile noi şi nu modificarea celor existente!

Atribuiri

- Avantaje:
  - Modelarea obiectelor a căror stare variază în timp
  - Evitarea pasării explicite a fiecărei modificări de stare
- Dezavantaj: pierderea transparenţei referenţiale (v. Cursul 1)

Sfârșitul cursului 4

Ce am învățat

· Tipare dinamică vs. statică, tare vs. slabă, legare dinamică vs statică; Scheme: tipare dinamică, tare; domeniu al variabilelor, construcții speciale în Scheme: lambda, let, let*, letrec, define; controlul evaluării, set! și efecte laterale.

Cursul 5

Evaluare leneşă în Scheme

Cuprins

Întârziere

Motivaţie
Prin exemplu

Exemplul 20.1.
Să se implementeze funcţia nestrictă *prod*, astfel încât al doilea parametru să fie evaluat doar dacă primul este *true*:

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

Varianta 1
Încercare – implementare directă

```scheme
(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)
(test #t)
```

Output: `y 0 | y 30`

- Implementare eronată, deoarece ambii parametri sunt evaluaţi în momentul aplicării

Varianta 2
Încercare – `quote` & `eval`

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

(define test
  (lambda (x)
    (let ((y 5))
      (prod x '(begin (display "y ") y))))) ; quote

(test #f)
(test #t)
```

Output: `0 | reference to undefined identifier`

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

Varianta 3
Încercare – închideri funcţionale

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

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

(test #f)
(test #t)
```

Output: `0 | y y 30`

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

Varianta 4
Promisiuni: `delay` & `force`

```scheme
(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)
(test #t)
```

Output: `0 | y 30`

- Comportament corect: `y` evaluat la cerere, o singură dată → evaluare leneşă

Promisiuni
Descriere

- Rezultatul încă neevaluat al unei expresii
- Exemplu: `(delay (* 5 6))`
- Valori de prim rang în limbaj (v. Definiţia 4.8)
- `delay`
  - construieşte o promisiune
  - funcţie nestrictă
- `force`
  - forţează respectarea unei promisiuni, evaluând expresia doar la prima aplicare, şi salvându-i valoarea
  - începând cu a doua invocare, întoarce, direct, valoarea memorată

Promisiuni
Cerinţe

- Salvarea contextului computaţional al expresiei a cărei evaluare este întârziată şi evaluarea ei ulterioară în acel context → asemănător cu închiderile funcţionale.
- Salvarea rezultatului primei evaluări a expresiei
- Distingerea primei forţări de celelalte → efect lateral.

Promisiuni
Implementare în Scheme (1)

- `p` = o promisiune – expresie care se evaluează numai când este necesar prima oară, și reține valoarea la care s-a evaluat;
- `my-delay` construiește promisiunea;
- `my-force` evaluează promisiunea;

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

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

Promisiuni
Implementare în Scheme (2a)

```
(define make-promise (lambda (closure)
    (let ((ready? #f) (result #f))
      (lambda ()     ; promisiunea
        (if (not ready?)
          (begin (set! ready? #t)
                 (set! result (closure))))
        result
))))
```

Dar dacă:

```
(define x 1)
(define p (my-delay (if (= x 3) 0
         (begin (set! x (+ x 1)) (my-force p) 100)
)))
```

`(my-force p)` returnează 100, deşi prima valoare calculată de o promisiune terminată a fost 0 (când `x` a ajuns la 3).

Promisiuni
Implementare în Scheme (2b – corect)

```
(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
Implementare în Scheme – discuție

- Situaţii în care evaluarea expresiei împachetate declanşează, ea însăşi, forţarea promisiunii → a doua verificare a lui `ready?`.
- Promisiuni → obiecte cu stare.
- Prima forţare → efecte laterale.

Observaţii

- Dependenţă între mecanismul de întârziere şi cel de evaluare ulterioară a expresiilor — închideri/aplicaţii (varianta 3), `delay`/`force` (varianta 4) etc.
- Număr mare de modificări la înlocuirea unui mecanism existent, utilizat de un număr mare de funcţii
- Cum se pot diminua dependenţele?

Abstracţii

Abstracţii procedurale
Motivație

Context:

- Probleme cu complexitate ridicată.
- Descompunere în subprobleme, dar până unde?
- Nevoia restrângerii detaliilor luate în calcul la un anumit moment → abstractizare.
- Lucru la nivel conceptual, al gândirii programatorului, deasupra nivelului operaţiilor elementare din limbaj.

Abstracţii procedurale
Exemplu

Exemplul 21.1.

```
(define sum-of-squares
  (lambda (x y)
    (+ (square x) (square y))))

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

- `sum-of-squares`: conceptul de sumă a pătratelor, deasupra conceptului de ridicare la pătrat
- `square`: conceptul de ridicare la pătrat, deasupra conceptului de înmulţire

Abstracţii procedurale
Definiție

- Combinarea conceptelor pentru obţinerea de concepte mai complexe, cu propria identitate.
- `square`, din perspectiva `sum-of-squares`, substituibilă cu orice altă funcţie cu acelaşi comportament.

Definiția 21.2 (Abstracţie procedurală).
Funcţionalitate autonomă, independentă de implementare.

Abstracţii procedurale

Cerințe

- Izolarea implementării de utilizare → modularitate.
- Reutilizabilitate.
- `square`, `sum-of-squares` → generalizare la nivel de numere
- Funcţionale (e.g. `map`, `filter`, `foldl`) → generalizare la nivel de comportament !
- Gândirea în termenii diverselor abstracţii răspândite (*patterns*) → aplicarea lor în situaţii noi.

Abstracţii de date

Motivație

- Exemplu: cum reprezentăm expresiile cu evaluare întârziată?
- Abordarea din secţiunea precedentă: 1 singur nivel:

funcţii ce operează cu expresii cu evaluare întârziată: implementare şi utilizare, sub formă de închideri sau promisiuni

Abstracţii de date

Soluție

- Alternativ: 2 nivele, separate de o barieră de abstractizare

funcţii ce operează cu expresii cu evaluare întârziată: utilizare
interfaţă: `pack`, `unpack`
expresii cu evaluare întârziată, ca închideri funcţionale sau promisiuni: implementare

- Bariera:
  - limitează analiza detaliilor
  - elimină dependenţele dintre nivele

Abstracţii de date

Definiție

Definiția 21.3 (Abstracţie de date).
Tehnică de separare a utilizării unei structuri de date de implementarea acesteia.

- Permit *wishful thinking*: utilizarea structurii înaintea implementării acesteia.

Abstracţii de date

Implementări diferite, aceeași utilizare; v1: promisiuni

Exemplul 21.4 (Continuare a exemplului 20.1).

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

(define unpack force)

(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))) )))
```


Abstracţii de date

Implementări diferite, aceeași utilizare; v2: închideri

Exemplul 21.5 (Continuare a exemplului 20.1).

```scheme
(define-macro pack (lambda (expr)
    `(lambda () ,expr) ))

(define unpack (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))) )))
```


Fluxuri

Motivaţie

Exemplul 22.1.
Să se determine suma numerelor pare din intervalul $[a, b]$.

```scheme
(define even-sum-iter ; varianta 1
  (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 ; varianta 2
  (lambda (a b)
    (foldl + 0 (filter even? (interval a b)))))
```

Observaţii

- Varianta 1 – iterativă (d.p.d.v. proces): eficientă, datorită spaţiului suplimentar constant
- Varianta 2 – foloseşte liste:
  - elegantă şi concisă
  - ineficientă, datorită spaţiului posibil mare, ocupat la un moment dat – toate numerele din intervalul $[a, b]$
- Cum îmbinăm avantajele celor 2 abordări?

Fluxuri
Caracteristici

- Secvenţe construite parţial, extinse la cerere, ce creează iluzia completitudinii structurii
- Îmbinarea eleganţei manipulării listelor cu eficienţa calculului incremental
- Bariera de abstractizare:
  - componentele listelor evaluate la construcţie (`cons`)
  - componentele fluxurilor evaluate la selecţie (`cdr`)
- Construcţie şi utilizare:
  - separate la nivel conceptual → modularitate
  - întrepătrunse la nivel de proces

Fluxuri
Operatori: construcţie şi selecţie

- `cons`, `car`, `cdr`, `nil`, `null?`.

```
(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?)
```


Fluxuri
Operatori: `take` şi `drop`

- selecţie / eliminare dintr-un flux a n elemente.

```
(define stream-take (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)))
)))
```


Fluxuri
Operatori: `map` şi `filter`

- operatori de aplicare şi filtrare pe liste.

```
(define stream-map (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
Operatori: `zip`, `append` şi conversie

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

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

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


Fluxuri – Exemple
Implementarea unui flux de numere 1

- Definiţie cu închideri:

```
(define ones (lambda ()(cons 1 (lambda ()(ones)))))
```

- Definiţie cu fluxuri:

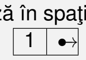
```
(define ones (stream-cons 1 ones))
(stream-take 5 ones) ; (1 1 1 1 1)
```

- Definiţie cu promisiuni:

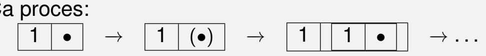
```
(define ones (delay (cons 1 ones)))
```


Fluxuri – Exemple
Flux de numere 1 – discuţie

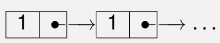
- Extinderea se realizează în spaţiu constant:



- Ca proces:



- Structural:



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 porţiunilor deja explorate.
- Promisiuni: multiple parcurgeri ale fluxului determină evaluarea dincolo de porţiunile deja explorate.

Fluxul numerelor naturale
Formulare implicită

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

- Porţiunea deja explorată din flux poate fi utilizată pentru explorarea porţiunii următoare

Fluxul numerelor pare
În două variante

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

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

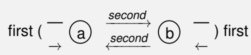

Fluxul numerelor prime
Metodă

- Ciurul lui Eratostene.
- Pornim de la fluxul numerelor naturale, începând cu 2.
- Elementul curent din fluxul iniţial aparţine fluxului numerelor prime.
- Restul fluxului generat se obţine
  - eliminând multiplii elementului curent din fluxul iniţial;
  - continuând procesul de filtrare, cu elementul următor.

Fluxul numerelor prime
Implementare

```
(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
Concept



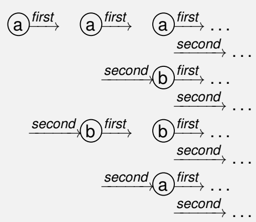
- Fiecare nod conţine:
  - cheia: `key`
  - legăturile către două noduri: `first`, `second`

Grafuri ciclice
Implementare

```
(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
Explorare

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



Căutare

Spaţiul stărilor unei probleme

Definiția 23.1 (Spaţiul stărilor unei probleme).
Mulţimea configuraţiilor valide din universul problemei.

Exemplul 23.2.
Fie problema Pal_n: *Să se determine palindroamele de lungime cel puţin n, ce se pot forma cu elementele unui alfabet fixat*.

Stările problemei →toate şirurile generabile cu elementele alfabetului respectiv.

Specificarea unei probleme prin spaţiul stărilor
Aplicaţie pe Pal_n

- Starea iniţială: şirul vid
- Operatorii de generare a stărilor succesor ale unei stări: inserarea unui caracter la începutul unui şir dat
- Operatorul de verificare a proprietăţii de *scop* a unei stări: palindrom

Căutare în spaţiul stărilor

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

Căutare în lăţime
Obișnuită

```
(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 soluţii
- Cum le obţinem pe celelalte, mai ales dacă spaţiul e infinit?

Căutare în lăţime
Leneșă (1) – fluxul stărilor *scop*

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


Căutare în lăţime
Leneșă (2)

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

- Nivel înalt, conceptual: separare între explorarea spaţiului şi identificarea stărilor *scop*.
- Nivel scăzut, al instrucţiunilor: întrepătrunderea celor două aspecte.
- Aplicaţii:
  - Palindroame
  - Problema reginelor

Sfârșitul cursului 5
Ce am învățat

· Aplicații ale evaluării întârziate, abstractizare procedurală, fluxuri, căutare în spațiul stărilor.

Cursul 6

Programare funcţională în Haskell

Cuprins

Introducere

Paralelă între limbaje

Criteriu	Scheme	Haskell
Funcţii	*Curry* sau *uncurry*	*Curry*
Tipare	Dinamică, tare	Statică, tare
Legarea variabilelor	Locale → statică, *top-level* → dinamică	Statică
Evaluare	Aplicativă	Normală
Transferul parametrilor	*Call by sharing*	*Call by need*
Efecte laterale	`set!` & co.	Interzise

Funcţii

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

Exemplul 24.1.

Definiţii echivalente ale funcţiei `add`:

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

result       =   add1 1 2    -- echivalent, ((add1 1) 2)
result2      =   add3 1 2    -- echivalent, ((add3 1) 2)
inc          =   add1 1
```


Funcţii şi operatori

- Aplicabilitatea parţială a operatorilor infixaţi
- Transformări operator → funcţie şi funcţie → operator

Exemplul 24.2.

Definiţii echivalente ale funcţiilor `add` şi `inc`:

```
add4        =   (+)
result1     =   (+) 1 2
result2     =   1 `add4` 2

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


Pattern matching

- Definirea comportamentului funcţiilor pornind de la structura parametrilor → traducerea axiomelor TDA.

Exemplul 24.3.

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

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

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

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


List comprehensions

- Definirea listelor prin proprietăţile elementelor, ca într-o specificare matematică

Exemplul 24.4.

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

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

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

Tipare

Tipuri

- Tipuri ca mulţimi de valori:
  - `Bool = {True, False}`
  - `Natural = {0, 1, 2, ...}`
  - `Char = {'a', 'b', 'c', ...}`
- Rolul tipurilor
- Tipare statică:
  - etapa de tipare anterioară etapei de evaluare
  - asocierea fiecărei expresii din program cu un tip
- Tipare tare: absenţa conversiilor implicite de tip
- Expresii de:
  - program: `5, 2 + 3, x && (not y)`
  - tip: `Integer, [Char], Char -> Bool, a`

Exemple de tipuri

Exemplul 25.1.

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

etc.
```

- Tipurile de bază sunt tipurile elementare din limbaj:
  `Bool, Char, Integer, Int, Float, ...`

Constructori de tip

- Funcţii de tip, ce îmbogăţesc tipurile din limbaj.

Exemplul 25.2 (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 funcţiilor

- Constructorul `->` este asociativ dreapta:
  `Integer -> Integer -> Integer`
  ≡ `Integer -> (Integer -> Integer)`

Exemplul 25.3.

```
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 25.4 (Polimorfism parametric).
Manifestarea aceluiaşi comportament pentru parametri de tipuri diferite. Exemplu: `idd`.

Definiția 25.5 (Polimorfism ad-hoc).
Manifestarea unor comportamente diferite pentru parametri de tipuri diferite. Exemplu: `==`.

Constructorul de tip `Natural`

Exemplu de definire TDA

Exemplul 25.6.

```
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)

-- try addNat (Succ (Succ doi)) (Succ (Succ (Succ Zero)))
```


Constructorul de tip `Natural`

Comentarii

- Constructor de tip: `Natural`
  - nular
  - se confundă cu tipul pe care-l construieşte
- Constructori de date:
  - `Zero`: nular
  - `Succ`: unar
- Constructorii de date ca funcţii, dar utilizabile în *pattern matching*

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

Constructorul de tip `Pair`
Exemplu de definire TDA

Exemplul 25.7.

```haskell
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`
Comentarii

- Constructor de tip: `Pair`
  - polimorfic, binar
  - generează un tip în momentul aplicării asupra 2 tipuri
- Constructor de date: `P`, binar:

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


Uniformitatea reprezentării tipurilor

Exemplul 25.8 (Tipurile de bază).

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

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

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

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

etc.

Proprietăţi induse de tipuri

Definiția 25.9 (Progres).
O expresie bine-tipată (căreia i se poate asocia un tip):
- este o valoare *sau*
- poate fi redusă.

Definiția 25.10 (Conservare).
Evaluarea unei expresii bine-tipate produce o expresie bine-tipată – de obicei, cu acelaşi tip.

Sinteza

Sinteza de tip
Definiție

Definiția 26.1 (Sinteză de tip — *type inference*).
Determinarea automată a tipului unei expresii, pe baza unor reguli precise.

- Adnotările explicite de tip, deşi posibile, nenecesare în majoritatea cazurilor
- Dependentă de:
  - componentele expresiei
  - contextul lexical al expresiei
- Reprezentarea tipurilor → expresii de tip:
  - constante de tip: tipuri de bază
  - variabile de tip: pot fi legate la orice expresii de tip
  - aplicaţii ale constructorilor de tip pe expresii de tip

Reguli simplificate de sinteză de tip
Exemple

- Formă: $\dfrac{\texttt{premisa-1 ... premisa-m}}{\texttt{concluzie-1 ... concluzie-n}}\ \texttt{(nume)}$
- Funcţie: $\dfrac{\texttt{Var :: a}\quad\texttt{Expr :: b}}{\texttt{\textbackslash Var -> Expr :: a -> b}}\ \texttt{(TLambda)}$
- Aplicaţie: $\dfrac{\texttt{Expr1 :: a -> b}\quad\texttt{Expr2 :: a}}{\texttt{(Expr1 Expr2) :: b}}\ \texttt{(TApp)}$
- Operatorul +: $\dfrac{\texttt{Expr1 :: Int}\quad\texttt{Expr2 :: Int}}{\texttt{Expr1 + Expr2 :: Int}}\ \texttt{(T+)}$
- Literali întregi: $\dfrac{}{\texttt{0, 1, 2, ... :: Int}}\ \texttt{(TInt)}$

Exemple de sinteză de tip
Transformare de funcție

Exemplul 26.2.

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

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

$$\dfrac{\texttt{(g 3) :: Int}\quad\texttt{1 :: Int}}{\texttt{(g 3) + 1 :: Int}}\ \texttt{(T+)}$$

$$\Rightarrow \texttt{b = Int}$$

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

$$\Rightarrow \texttt{a = c -> d, c = Int, d = Int}$$

$$\Rightarrow \texttt{f :: (Int -> Int) -> Int}$$

Exemple de sinteză de tip
Combinator de punct fix

Exemplul 26.3.

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

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

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

$$\Rightarrow \texttt{a = c -> d, b = d}$$

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

$$\Rightarrow \texttt{a -> b = e -> g, a = e, b = g, c = g}$$

$$\Rightarrow \texttt{f :: (c -> d) -> b = (g -> g) -> g}$$

Exemple de sinteză de tip
O funcție ne-tipabilă

Exemplul 26.4.

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

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

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

Ecuația `c -> d = c` nu are soluție ($\nexists$ tipuri recursive)
$\Rightarrow$ funcția nu poate fi tipată.

Unificare
Definiție

Definiția 26.5 (Unificare).
Procesul de identificare a valorilor variabilelor din 2 sau mai multe formule, astfel încât substituirea variabilelor prin valorile asociate să conducă la coincidența formulelor.

Definiția 26.6 (Substituție).
O substituție este o mulțime de legări variabilă - valoare.

Unificare
Exemplu

Exemplul 26.7.

- Pentru expresiile de tip:
  - `t1 = (a, [b])`
  - `t2 = (Int, c)`
- Putem avea substituțiile (variante):
  - `S1 = {a ← Int, b ← Int, c ← [Int]}`
  - `S1 = {a ← Int, c ← [b] }`
- Forme comune pentru `S1` respectiv `S2`:
  - `t1/S1 = t2/S1 = (Int, [Int])`
  - `t1/S2 = t2/S2 = (Int, [b])`

Definiția 26.8 (*Most general unifier* – MGU).
Cea mai generală substituție sub care formulele unifică. Exemplu: `S2`.

Unificare
Condiții

- O variabilă de tip `a` unifică cu o expresie de tip `E` doar dacă:
  - $E = a$ *sau*
  - $E \neq a$ și `E` nu conține `a` (*occurrence check*).
- 2 constante de tip unifică doar dacă sunt egale.
- 2 aplicații de tip unifică doar dacă implică același constructor de tip și argumente ce unifică recursiv.

Tip principal
Exemplu și definiție

Exemplul 26.9.

- Tipurile: `t = (a, [b])` , `t' = (Integer, c)`
  MGU: `S = {a ← Integer, c ← [b]}`
  Tipuri mai particulare (instanțe): `(Integer, [Integer])`, `(Integer, [Char])`, etc
- Funcția: `\x -> x`
  Tipuri corecte: `Int -> Int` , `Bool -> Bool` , `a -> a`

Definiția 26.10 (Tip principal al unei expresii).
Cel mai general tip care descrie complet natura expresiei. Se obține prin utilizarea MGU.

Evaluare

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 27.1.

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

Evaluare:

```
f (2 + 3, 3 + 5) (5 + 8)
→ (2 + 3) + (2 + 3)
→ 5 + 5        reutilizăm rezultatul primei evaluări!
→ 10            ceilalți parametri nu sunt evaluați
```

Paşi în aplicarea funcţiilor

Exemplu

Exemplul 27.2.

```
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]
```


Paşi în aplicarea funcţiilor

Ordine

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`)

Paşi în aplicarea funcţiilor

Exemplu – revisited

Exemplul 27.2 (execuție).

```
f 3 5                                  evaluare pattern
??  notNil  xs                          evaluare prima gardă
??      where                           necesar xs → evaluare where
??          xs = [3 .. 5]
??          → 3:[4 .. 5]
??  → notNil (3:[4 .. 5])
??  → True
→ front xs                                  evaluare valoare gardă
    where
        xs = 3:[4 .. 5]                 xs deja calculat
        → 3:4:[5]
→ front (3:4:[5])
→ 3 + 4 → 7
```


Consecinţe

- Evaluarea parţială a structurilor – liste, tupluri etc.
- Listele sunt, implicit, văzute ca fluxuri!

Exemplul 27.3.

```
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))
```


Sfârșitul cursului 6

Ce am învățat

· Haskell, diferențe față de Scheme, pattern matching și list comprehensions, tipuri în Haskell, construcție de tipuri, sinteză de tip, unificare, evaluare în Haskell.

Cursul 7

Evaluare Leneșă în Haskell

Cuprins

Evaluare leneșă

Programare orientată spre date

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

Exemplul 28.1 (Suma pătratelor).

Suma pătratelor numerelor naturale până la `n` ca sumă pe o listă:

```
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

Exemplul 28.2 (Minimul unei liste – definiție).

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

Exemplul 28.3 (Minimul unei liste – execuție).

```
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 28.4 (Accesibilitatea într-un graf).

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.

Backtracking eficient

Continuare exemplu

Exemplul 28.5 (Accesibilitatea într-un graf – căi).

```
neighbors node = map snd . filter ((== node) . fst)

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


Studiu de caz

Bibliotecă de parsare (din bibliografie).

Fișierul `ParserAO.hs`, de văzut împreună cu codul echivalent de la seria CA/CB.

Bibliografie

[Thompson, S. (1999), Haskell: The Craft of Functional Programming, Second Edition, Addison-Wesley.]

Cursul 8

Clase în Haskell

Cuprins

Motivație

Motivație

Exemplu

Exemplul 29.1.

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\""
```


Motivație

Varianta 1 – Funcții dedicate fiecărui tip

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

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

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


Motivație

Varianta 1 – Funcții dedicate – discuție

- 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.

Motivație

Varianta 2 – Supraîncărcarea funcției

- Definirea mulțimii `Show`, a tipurilor care expun `show`

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

- Precizarea apartenenței unui tip la această mulțime (instanța *aderă* la clasă)

```
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"
```


Motivație

Varianta 2 – Supraîncărcare – discuție (1)

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

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

Semnificație: *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`.

Motivație

Varianta 2 – Supraîncărcare – discuție (2)

- 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 Haskell

Clase şi instanţe

Definiţii

Definiția 30.1 (Clasă).
Mulţime de tipuri ce pot supraîncarca operaţiile specifice clasei. Reprezintă o modalitate structurată de control asupra polimorfismului ad-hoc. Exemplu: clasa `Show`, cu operaţia `show`.

Definiția 30.2 (Instanţă a unei clase).
Tip care supraîncarcă operaţiile clasei. Exemplu: tipul `Bool` în raport cu clasa `Show`.

Clase predefinite

`Show, Eq`

```
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 definiţii implicite (v. liniile 7–8).
- Necesitatea suprascrierii cel puţin unuia din cei 2 operatori ai clasei `Eq` pentru instanţierea corectă.

Clase predefinite

`Ord`

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

- Contexte utilzabile şi la definirea unei clase.
- Moştenirea claselor, cu preluarea operaţiilor din clasa moştenită.
- Necesitatea aderării la clasa `Eq` în momentul instanţierii clasei `Ord`.

Clase Haskell vs. Clase în POO

Haskell

- Clasele sunt mulţimi de tipuri (superclase)
- Instanţierea claselor de către tipuri

POO (e.g. Java)

- Clasele sunt mulţimi de obiecte (tipuri)
- Implementarea interfeţelor de către clase

Aplicaţii clase

`invert`

Problema

Exemplul 31.1 (`invert`).
Fie constructorii de tip:

```
data Pair a = P a a

data NestedList a
    =   Atom a
    |   Seq [NestedList a]
```

Să se definească operaţia `invert`, aplicabilă pe valori de tipuri diferite, inclusiv `Pair a` şi `NestedList a`, comportamentul fiind specific fiecărui tip.

`invert`

Implementare

```
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 (Seq x)  =   Seq $ reverse . map invert $ x

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

- Necesitatea contextului, în cazul tipurilor `[a]` şi `NestedList a`, pentru inversarea elementelor înselor.

contents
Problema

Exemplul 31.2 (`contents`).
Să se definească operaţia `contents`, aplicabilă pe obiecte structurate, inclusiv pe cele aparţinând tipurilor `Pair a` şi `NestedList a`, care întoarce elementele din componenţă, sub forma unei liste Haskell.

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

- `a` este tipul unui container, e.g. `NestedList b`
- Elementele listei întoarse sunt cele din container
- Cum precizăm tipul acestora (`b`)?

contents
Varianta 1a

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

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

- Conform definiţiei clasei:

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

- Conform supraîncărcării funcţiei (`id`):

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

- Ecuaţia `[a] = [[a]]` nu are soluţie ⇒ eroare.

contents
Varianta 1b

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

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

- Conform definiţiei clasei:

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

- Conform supraîncărcării funcţiei (`id`):

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

- Ecuaţia `[a] = [b]` are soluţie pentru `a = b`, dar tipul `[a] -> [a]` insuficient de general în raport cu `[a] -> [b]` ⇒ eroare!

contents
Varianta 2

- Soluţie: clasa primeşte constructorul de tip, şi nu tipul container propriu-zis

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

instance Container Pair where
    contents (P x y) = [x, y]

instance Container NestedList where
    contents (Atom x)    =    [x]
    contents (Seq x)     =    concatMap contents x
```


Contexte
Câteva exemple

```
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
Observaţii

- 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`.

Sfârșitul cursului 8
Ce am învățat

· Clase Haskell, polimorfism ad-hoc, instanțiere de clase, derivare a unei clase, context.

Cursul 9

Logica cu predicate de ordinul I

Cuprins

Introducere

Logică

- formalism simbolic pentru reprezentarea faptelor și raționament.
- se bazează pe ideea de valoare de adevăr – e.g. *Adevărat* sau *Fals*.
- permite realizarea de argumente (argumentare) și demonstrații – deducție, inducție, rezoluție, etc.

Programare logică

- program scris folosind propoziții logice (clauze Horn pentru Prolog);
- mediul de execuție poate folosi propozițiile pentru a demonstra teoreme sau pentru a deduce fapte.
- pentru a înțelege cum funcționează programele scrise într-un limbaj de programare logică trebuie să înțelegem
  - ce sunt propozițiile, ce înseamnă și cum pot fi ele reprezentate;
  - cum funcționează procesele teoretice pe care se bazează mediul de execuție.

Logica propoziţională

Logica propoziţională

Context și elemente principale

- Cadru pentru:
  - descrierea proprietăţilor obiectelor, prin intermediul unui limbaj, cu o semantică asociată;
  - deducerea de noi proprietăţi, pe baza celor existente.
- Expresia din limbaj: propoziţia, corespunzătoare unei afirmaţii, ce poate fi adevărată sau falsă.
- Exemplu: "Profesorul vorbește și studenții ascultă."
- Accepţii asupra unei propoziţii:
  - secvenţa de simboluri utilizate (abordarea aleasă) sau
  - înţelesul propriu-zis al acesteia, într-o interpretare.
- Valoarea de adevăr a unei propoziţii determinată de valorile de adevăr ale propoziţiilor constituente.

Sintaxă

- 2 categorii de propoziţii
  - simple → fapte atomice: "Profesorul vorbește.", "Studenții ascultă."
  - compuse → relaţii între propoziţii mai simple: "Telefonul sună şi câinele latră."
- Propoziţii simple: $p, q, r, \ldots$
- Negaţii: $\neg\alpha$
- Conjuncţii: $(\alpha \wedge \beta)$
- Disjuncţii: $(\alpha \vee \beta)$
- Implicaţii: $(\alpha \Rightarrow \beta)$
- Echivalenţe: $(\alpha \Leftrightarrow \beta)$

Semantică

- Scop: dezvoltarea unor mecanisme de prelucrare, aplicabile independent de valoarea de adevăr a propoziţiilor, într-o situaţie particulară.
- Accent pe relaţiile între propoziţiile compuse şi cele constituente.
- Pentru explicitarea legăturilor → utilizarea conceptului de interpretare.

Semantică
Interpretare

Definiția 33.1 (Interpretare).
Mulţime de asocieri între fiecare propoziţie simplă din limbaj şi o valoare de adevăr.

Exemplul 33.2.

Interpretarea I:
- $p^I = false$
- $q^I = true$
- $r^I = false$

Interpretarea J:
- $p^J = true$
- $q^J = true$
- $r^J = true$

Semantică
Propoziţii compuse (1)

- Sub o interpretare *fixată* → dependenţa valorii de adevăr a unei propoziţii compuse de valorile de adevăr ale celor constituente
- Negaţie: $(\neg\alpha)^I = \begin{cases} true & \text{dacă } \alpha^I = false \\ false & \text{altfel} \end{cases}$
- Conjuncţie:
  $(\alpha \wedge \beta)^I = \begin{cases} true & \text{dacă } \alpha^I = true \text{ şi } \beta^I = true \\ false & \text{altfel} \end{cases}$
- Disjuncţie:
  $(\alpha \vee \beta)^I = \begin{cases} false & \text{dacă } \alpha^I = false \text{ şi } \beta^I = false \\ true & \text{altfel} \end{cases}$

Semantică
Propoziţii compuse (2)

- Implicaţie:
  $(\alpha \vee \beta)^I = \begin{cases} false & \text{dacă } \alpha^I = true \text{ şi } \beta^I = false \\ true & \text{altfel} \end{cases}$
- Echivalenţă: $(\alpha \vee \beta)^I = \begin{cases} true & \text{dacă } \alpha^I = \beta^I \\ false & \text{altfel} \end{cases}$

Evaluare
Cum determinăm valoarea de adevăr

Definiția 33.3 (Evaluare).
Determinarea valorii de adevăr a unei propoziţii, sub o interpretare, prin aplicarea regulilor semantice anterioare.

Exemplul 33.4.
- Interpretarea I:
  - $p^I = false$
  - $q^I = true$
  - $r^I = false$
- Propoziţia: $\phi = (p \wedge q) \vee (q \Rightarrow r)$
  $\phi^I = (false \wedge true) \vee (true \Rightarrow false) = false \vee false = false$

Satisfiabilitate
Valoare de adevăr peste mai multe interpretări

Definiția 33.5 (Satisfiabilitate).
Proprietatea unei propoziţii care este adevărată sub cel puţin o interpretare. Acea interpretare satisface propoziţia.

Exemplul 33.6 (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 33.7 (Validitate).
Proprietatea unei propoziţii care este adevărată în toate interpretările. Propoziţia se mai numeşte tautologie.

Exemplul 33.8 (Validitate).
Propoziţia $p \vee \neg p$ este adevărată, indiferent de valoarea de adevăr a lui p, deci este validă.

- Verificabilă prin metoda tabelei de adevăr.

Nesatisfiabilitate

Definiția 33.9 (Nesatisfiabilitate).
Proprietatea unei propoziţii care este falsă în toate interpretările. Propoziţia se mai numeşte contradicţie.

Exemplul 33.10 (Nesatisfiabilitate).
Propoziţia $p \Leftrightarrow \neg p$ este falsă, indiferent de valoarea de adevăr a lui p, deci este nesatisfiabilă.

- Verificabilă prin metoda tabelei de adevăr.

Derivabilitate
Definiţie

Definiția 33.11 (Derivabilitate logică).
Proprietatea unei propoziţii de a reprezenta consecinţa logică a unei mulţimi de alte propoziţii, numite premise. Mulţimea de propoziţii Δ derivă propoziţia ϕ, fapt notat prin $\Delta \models \phi$, dacă şi numai dacă orice interpretare care satisface toate propoziţiile din Δ satisface şi ϕ.

Exemplul 33.12.
- $\{p\} \models p \vee q$
- $\{p, q\} \models p \wedge q$
- $\{p\} \not\models p \wedge q$
- $\{p, p \Rightarrow q\} \models q$

Derivabilitate
Verificare

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

Exemplul 33.13.

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 şi $p \Rightarrow q$, sunt adevărate, precizează şi adevărul concluziei, q.

Formulări echivalente ale derivabilităţii

- $\{\phi_1, \ldots, \phi_n\} \models \phi$

sau

- Propoziţia $\phi_1 \wedge \ldots \wedge \phi_n \Rightarrow \phi$ este validă

sau

- Propoziţia $\phi_1 \wedge \ldots \wedge \phi_n \wedge \neg\phi$ este nesatisfiabilă

Inferență
Motivaţie

- Derivabilitate logică → proprietate a propoziţiilor.
- Derivare mecanică (inferenţă) → demers de calcul, în scopul verificării derivabilităţii logice.
- Creşterea exponenţială a numărului de interpretări în raport cu numărul de propoziţii simple.
- De aici, diminuarea valorii practice a metodelor semantice, precum cea a tabelei de adevăr.
- Alternativ, metode sintactice, care manipulează doar reprezentarea simbolică.

Inferenţă
Definiție

Definiția 33.14 (Inferenţă).

Derivarea mecanică a concluziilor unui set de premise.

Definiția 33.15 (Regulă de inferenţă).

Procedură de calcul capabilă să deriveze concluziile unui set de premise. Derivabilitatea mecanică a concluziei ϕ din mulţimea de premise Δ, utilizând regula de inferenţă *inf*, se notează $\Delta \vdash_{inf} \phi$.

Inferență
Reguli de inferenţă

- Şabloane parametrizate de raţionament, formate dintr-o mulţime de premise şi o mulţime de concluzii.
- exemplu: *Modus Ponens* (MP):

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

- exemplu: *Modus Tollens*:

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

Inferență
Proprietăţi ale regulilor

Definiția 33.16 (Consistenţă (*soundness*)).

Regula de inferenţă determină doar propoziţii care sunt, într-adevăr, consecinţe logice ale premiselor. Echivalent, $\Delta \vdash_{inf} \phi \Rightarrow \Delta \models \phi$.

Definiția 33.17 (Completitudine (*completeness*)).

Regula de inferenţă determină toate consecinţele logice ale premiselor. Echivalent, $\Delta \models \phi \Rightarrow \Delta \vdash_{inf} \phi$.

- Ideal, ambele proprietăţi – "nici în plus, nici în minus".
- Incompletitudinea regulii *Modus Ponens*, din imposibilitatea scrierii oricărei propoziţii ca implicaţie.

Axiome

- Exemplu: verificarea că $\{p \Rightarrow q, q \Rightarrow r\} \vdash p \Rightarrow r$
- Caz în care premisele sunt insuficiente pentru aplicarea regulilor de inferenţă.
- Soluţia: axiome – reguli de inferenţă fără premise.
- Exemplu de set de axiome:

$\alpha \Rightarrow (\beta \Rightarrow \alpha)$	Introducerea "$\Rightarrow$"
$(\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma))$	Distribuirea "$\Rightarrow$"
$(\neg\alpha \Rightarrow \neg\beta) \Rightarrow (\beta \Rightarrow \alpha)$	Inversarea "$\Rightarrow$"

Demonstraţii
Definiții

Definiția 33.18 (Demonstraţie).

Secvenţă de propoziţii, finalizată cu o concluzie, conţinând:

- premise
- instanţe ale axiomelor
- rezultate ale aplicării regulilor de inferenţă asupra elementelor precedente din secvenţă.

Definiția 33.19 (Teoremă).

Concluzia cu care se termină o demonstraţie.

Definiția 33.20 (Procedură de demonstrare).

Mecanism constând din (1) o mulţime de reguli de inferenţă şi (2) o strategie de control, ce dă ordinea aplicării regulilor.

Demonstrații
Exemplu

Exemplul 33.21.

Demonstrăm că $\{p \Rightarrow q, q \Rightarrow r\} \vdash p \Rightarrow r$.

1	$p \Rightarrow q$	Premisă
2	$q \Rightarrow r$	Premisă
3	$(q \Rightarrow r) \Rightarrow (p \Rightarrow (q \Rightarrow r))$	Introd. "$\Rightarrow$"
4	$p \Rightarrow (q \Rightarrow r)$	MP (3), (2)
5	$(p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r))$	Distrib "$\Rightarrow$"
6	$(p \Rightarrow q) \Rightarrow (p \Rightarrow r)$	MP (5), (4)
7	$p \Rightarrow r$	MP 6, 1

Demonstrații
Necesități

- Existența unui sistem de inferență consistent și complet, bazat pe:
  - axiomele de mai devreme.
  - regula de inferență *Modus Ponens*.

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

Rezoluție
O regulă de inferență mai bună

- Regulă de inferență foarte puternică.
- Baza unui demonstrator de teoreme consistent și complet.
- Spațiul de căutare mult mai mic ca în abordarea standard (vezi mai sus).
- Se bazează pe lucrul cu propoziții în forma clauzală.

Forma clauzală
Definiții

Definiția 33.22 (Literal).

Propoziție simplă sau negația ei. E.g. p și $\neg p$.

Definiția 33.23 (Expresie clauzală).

Literal sau disjuncție de literali. E.g. $p \vee \neg q \vee r$.

Definiția 33.24 (Clauză).

Mulțime de literali dintr-o expresie clauzală. E.g. $\{p, \neg q, r\}$.

Definiția 33.25 (Forma clauzală – CNF).

Reprezentarea unei propoziții sub forma unei mulțimi de clauze, implicit legate prin conjuncții.

Forma clauzală
Exemplu

Exemplul 33.26 (FNC).

Forma clauzală a propoziției

$p \wedge (\neg q \vee r) \wedge (\neg p \vee \neg r)$

este

$\{p\}, \{\neg q, r\}, \{\neg p, \neg r\}$.

Forma clauzală
Obținere

- Orice propoziție convertibilă în această formă astfel:

1. Eliminarea implicațiilor:
   $\alpha \Rightarrow \beta \;\rightarrow\; \neg\alpha \vee \beta$
2. Avansarea negațiilor până la literali:
   $\neg(\alpha \wedge \beta) \;\rightarrow\; \neg\alpha \vee \neg\beta$, $\neg(\alpha \vee \beta) \;\rightarrow\; \neg\alpha \wedge \neg\beta$,
   $\neg(\neg\alpha) \;\rightarrow\; \alpha$
3. Distribuirea lui $\vee$ față de $\wedge$:
   $\alpha \vee (\beta \wedge \gamma) \;\rightarrow\; (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$
4. Transformarea expresiilor în clauze:
   $\phi_1 \vee \ldots \vee \phi_n \;\rightarrow\; \{\phi_1, \ldots, \phi_n\}$
   $\phi_1 \wedge \ldots \wedge \phi_n \;\rightarrow\; \{\phi_1\}, \ldots, \{\phi_n\}$

Forma clauzală
Obținere – Exemplu

Exemplul 33.27.

Transformăm propoziția $p \wedge (q \Rightarrow r)$ în formă clauzală.

1.	$p \wedge (\neg q \vee r)$	Eliminare implicații
2.	$\{p\}, \{\neg q, r\}$	Formare clauze

Exemplul 33.28.

Transformăm propoziția $\neg(p \wedge (q \Rightarrow r))$ în formă clauzală.

1.	$\neg(p \wedge (\neg q \vee r))$	Eliminare implicații
2.	$\neg p \vee \neg(\neg q \vee r)$	Împinge negații (1)
3.	$\neg p \vee (q \wedge \neg r)$	Împinge negații (2,3)
4.	$(\neg p \vee q) \wedge (\neg p \vee \neg r)$	Distribuire $\vee$
5.	$\{\neg p, q\}, \{\neg p, \neg r\}$	Formare clauze

Rezoluție
Principiu de bază

- Ideea:

$$\frac{\begin{matrix}\{p \Rightarrow q\} \\ \{\neg p \Rightarrow r\}\end{matrix}}{\{q, r\}}$$

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

$$\frac{\begin{matrix}\{p_1, \ldots, r, \ldots, p_m\} \\ \{q_1, \ldots, \neg r, \ldots, q_n\}\end{matrix}}{\{p_1, \ldots, p_m, q_1, \ldots, q_n\}}$$

Rezoluţie
Cazuri speciale

- Clauza vidă → indicator de contradicţie între premise

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

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

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

Rezoluţie
Alte reguli de inferență – cazuri particulare ale rezoluției

- *Modus Ponens*:

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

- *Modus Tollens*

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

- Tranzitivitatea implicaţiei:

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

Rezoluţie
Observații

- Demonstrarea nesatisfiabilităţii → derivarea clauzei vide.
- Demonstrarea derivabilităţii concluziei ϕ din premisele $\phi_1,\ldots,\phi_n$ → demonstrarea nesatisfiabilităţii propoziţiei $\phi_1 \wedge \ldots \wedge \phi_n \wedge \neg\phi$.
- Demonstrarea validităţii propoziţiei ϕ → demonstrarea nesatisfiabilităţii propoziţiei $\neg\phi$.
- Rezoluţia → incompletă generativ, i.e. concluziile nu pot fi derivate direct, răspunsul fiind dat în raport cu o "întrebare" fixată.

Rezoluţie
Exemplu

Exemplul 33.29.

Demonstrăm că $\{p \Rightarrow q, q \Rightarrow r\} \vdash p \Rightarrow r$, i.e. mulţimea $\{p \Rightarrow q, q \Rightarrow r, \neg(p \Rightarrow r)\}$ conţine o contradicţie.

1. $\{\neg p,q\}$ Premisă
2. $\{\neg q,r\}$ Premisă
3. $\{p\}$ Concluzie negată
4. $\{\neg r\}$ Concluzie negată
5. $\{q\}$ Rezoluție 1, 3
6. $\{r\}$ Rezoluție 2, 5
7. $\{\}$ Rezoluție 4, 6 → clauza vidă

Rezoluţie
Consistență și completitudine

Teorema 33.30 (Rezoluţiei).

Rezoluţia propoziţională este consistentă şi completă, i.e. $\Delta \models \phi \Leftrightarrow \Delta \vdash_{rez} \phi$.

- Terminarea garantată a procedurii de aplicare a rezoluţiei: număr finit de clauze → număr finit de concluzii.

Logica cu predicate de ordinul I

Logica cu predicate de ordinul I
First Order Logic (FOL) – Context

- Extensie a logicii propoziţionale, cu explicitarea:
  - obiectelor din universul problemei;
  - relaţiilor dintre acestea.
- Logica propoziţională:
  - p: "Andrei este prieten cu Bogdan."
  - q: "Bogdan este prieten cu Andrei."
  - $p \Leftrightarrow q$
  - → Opacitate în raport cu obiectele şi relaţiile 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.
  - → Transparenţă în raport cu obiectele şi relaţiile referite.

Sintaxă
Simboluri utilizate

- Constante: obiecte particulare din universul discursului: c, d, $andrei$, $bogdan$, ...
- Variabile: obiecte generice: x, y, ...
- Simboluri funcţionale: $succesor$, $+$, ...
- Simboluri relaţionale (predicate): relaţii 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
Termeni

- Termeni (obiecte):
  - Constante;
  - Variabile;
  - Aplicaţii de funcţii: $f(t_1,\ldots,t_n)$, unde f este un simbol funcţional n-ar şi $t_1,\ldots,t_n$ sunt termeni.
    Exemple:
    - $succesor(4)$: succesorul lui 4, şi anume 5.
    - $+(2,x)$: aplicaţia funcţiei de adunare asupra numerelor 2 şi x, şi, totodată, suma lor.

Sintaxă
Atomi

- Atomi (relaţii): atomul $p(t_1,\ldots,t_n)$, unde p este un predicat n-ar şi $t_1,\ldots,t_n$ sunt termeni.
  Exemple:
  - $impar(3)$
  - $varsta(ion,20)$
  - $=(+(2,3),5)$

Sintaxă
Propoziţii

- Propoziţii (fapte) – dacă x variabilă, A atom, și α și β propoziţii, atunci o propoziţie are forma:
  - Fals, adevărat: $\perp$, $\top$
  - Atomi: A
  - Negaţii: $\neg\alpha$
  - Conectori: $\alpha\wedge\beta$, $\alpha\Rightarrow\beta$, ...
  - Cuantificări: $\forall x.\alpha$, $\exists x.\alpha$

Sintaxă
Exemplu și observații

Exemplul 34.1.

"Dan este prieten cu sora Ioanei"

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

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

Semantică
Interpretare

Definiția 34.2 (Interpretare).

O interpretare constă din:

- Un domeniu nevid, D
- Pentru fiecare constantă c, un element $c^I \in D$
- Pentru fiecare simbol funcţional, n-ar f, o funcţie $f^I : D^n \rightarrow D$
- Pentru fiecare predicat n-ar p, o funcţie $p^I : D^n \rightarrow \{false, true\}$.

Semantică
Elemente

- Atom:
  $(p(t_1,\ldots,t_n))^I = p^I(t_1^I,\ldots,t_n^I)$
- Negaţie, conectori, implicații: v. logica propoziţională
- Cuantificare universală:
  $(\forall x.\alpha)^I = \begin{cases} false & \text{dacă } \exists d \in D \,.\, \alpha^I_{[d/x]} = false \\ true & \text{altfel} \end{cases}$
- Cuantificare existenţială:
  $(\exists x.\alpha)^I = \begin{cases} true & \text{dacă } \exists d \in D \,.\, \alpha^I_{[d/x]} = true \\ false & \text{altfel} \end{cases}$

Exemple
Cu cuantificatori

Exemplul 34.3.

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 şi numai vrăbiile visează mălai."
   $\forall x.(viseaza(x,malai) \Leftrightarrow vrabie(x))$

Cuantificatori
Greşeli frecvente

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

Cuantificatori
Proprietăţi

- Necomutativitate:
  - $\forall x.\exists y.viseaza(x,y) \rightarrow$ "Toţi visează la ceva anume."
  - $\exists x.\forall y.viseaza(x,y) \rightarrow$ "Există cineva care visează la orice."
- Dualitate:
  - $\neg(\forall x.\alpha) \equiv \exists x.\neg\alpha$
  - $\neg(\exists x.\alpha) \equiv \forall x.\neg\alpha$

Aspecte legate de propoziţii
Analoage logicii propoziţionale

- Satisfiabilitate
- Validitate
- Derivabilitate
- Inferenţă
- Demonstraţie

Forme normale
Definiţii (1)

Definiţia 34.4 (Literal).
Atom sau negaţia lui. Exemplu: $prieten(x,y)$, $\neg prieten(x,y)$.

Definiţia 34.5 (Expresie clauzală).
Literal sau disjuncţie de literali.
Exemplu: $prieten(x,y) \vee \neg doctor(x)$.

Definiţia 34.6 (Clauză).
Mulţime de literali dintr-o expresie clauzală. Exemplu: $\{prieten(x,y), \neg doctor(x)\}$.

Forme normale
Definiţii (2)

Definiţia 34.7 (Forma clauzală / Forma normală conjunctivă – FNC).
Reprezentarea unei propoziţii sub forma unei mulţimi de clauze, implicit legate prin conjuncţii.

Definiţia 34.8 (Forma normală implicativă – FNI).
Reprezentarea unei propoziţii sub forma unei mulţimi de clauze, implicit legate prin conjuncţii, în care fiecare clauză are forma grupată
$\{\neg A_1, \ldots, \neg A_m, B_1, \ldots, B_n\}$,
corespunzătoare implicaţiei
$(A_1 \wedge \ldots \wedge A_m) \Rightarrow (B_1 \vee \ldots \vee B_n)$, unde A_i şi B_j sunt atomi.

Forme normale
Definiţii (3) – Clauze Horn

Definiţia 34.9 (Clauză Horn).
Clauză în care un singur literal este în formă pozitivă:
$\{\neg A_1, \ldots, \neg A_n, A\}$,
corespunzătoare implicaţiei
$A_1 \wedge \ldots \wedge A_n \Rightarrow A$.

Exemplul 34.10 (Clauze Horn).
Transformarea propoziţiei
$vrabie(x) \vee ciocarlie(x) \Rightarrow pasare(x)$ în forme normale, utilizând 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 propoziţiilor în FNC (1)
Eliminare implicaţii, împingere negaţii, redenumiri

1. Eliminarea implicaţiilor ($\not\Rightarrow$)
2. Împingerea negaţiilor până în faţa literalilor ($\neg\rightarrow$)
3. Redenumirea variabilelor cuantificate pentru obţinerea unicităţii 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 propoziţiilor în FNC (2)
Skolemizare

5. Eliminarea cuantificatorilor existenţiali (skolemizare) (S):
   - Dacă nu este precedat de cuantificatori universali: înlocuirea apariţiilor variabilei cuantificate printr-o constantă:
   $$\exists x.p(x) \rightarrow p(c_x)$$
   - Dacă este precedat de cuantificatori universali: înlocuirea apariţiilor variabilei cuantificate prin aplicaţia unei funcţii 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 propoziţiilor în FNC (3)
Cuantificatori universali, Distribuire $\vee$, Clauze

6. Eliminarea cuantificatorilor universali, consideraţi, acum, impliciţi ($\not\forall$):
$$\forall x.\forall y.(p(x) \wedge q(y) \vee r(f_z(x,y))) \rightarrow p(x) \wedge q(y) \vee r(f_z(x,y))$$
7. Distribuirea lui $\vee$ faţă de $\wedge$ ($\vee/\wedge$):
$$\alpha \vee (\beta \wedge \gamma) \rightarrow (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$$
8. Transformarea expresiilor în clauze (C).

Conversia propoziţiilor în FNC – Exemplu

Exemplul 34.11.

“Cine rezolvă toate laboratoarele este apreciat de cineva.”

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

$\not\Rightarrow$	$\forall x.(\neg\forall y.(\neg lab(y) \vee rezolva(x,y)) \vee \exists y.apreciaza(y,x))$
$\overset{\rightarrow}{\neg}$	$\forall x.(\exists y.\neg(\neg lab(y) \vee rezolva(x,y)) \vee \exists y.apreciaza(y,x))$
$\overset{\rightarrow}{\neg}$	$\forall x.(\exists y.(lab(y) \wedge \neg rezolva(x,y)) \vee \exists y.apreciaza(y,x))$
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	$\forall x.((lab(f_y(x)) \wedge \neg rezolva(x,f_y(x))) \vee apreciaza(f_z(x),x))$
$\not\forall$	$(lab(f_y(x)) \wedge \neg rezolva(x,f_y(x))) \vee apreciaza(f_z(x),x)$
$\vee/\wedge$	$(lab(f_y(x)) \vee apr(f_z(x),x)) \wedge (\neg rez(x,f_y(x)) \vee apr(f_z(x),x))$
C	$\{lab(f_y(x)), apr(f_z(x),x)\}, \{\neg rez(x,f_y(x)), aprc(f_z(x),x)\}$

Unificare
Motivaţie

- Rezoluţie:

$$\frac{\{prieten(x, mama(y)), doctor(x)\} \quad \{\neg prieten(mama(z), z)\}}{?}$$

- Cum aplicăm rezoluţia?
- Soluţia: unificare (vezi sinteza de tip – Def. 26.5, 26.6, 26.8)
- MGU (Most General Unifier): $S = \{x \leftarrow mama(z), z \leftarrow mama(y)\}$
- Forma comună a celor doi atomi: $prieten(mama(mama(y)), mama(y))$
- Rezolvent: $doctor(mama(mama(y)))$

Unificare
Observaţii

- Problemă NP-completă;
- Posibile legări ciclice;
- Exemplu:
  $prieten(x, mama(x))$ şi $prieten(mama(y), y)$
  MGU: $S = \{x \leftarrow mama(y), y \leftarrow mama(x)\}$
  $\Rightarrow x \leftarrow mama(mama(x)) \rightarrow$ imposibil!
- Soluţie: verificarea apariţiei unei variabile în valoarea la care a fost legată (*occurrence check*);

Unificare
Rolul în rezoluţie

- Rezoluţia pentru clauze Horn:

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

- $unificare(\alpha, \beta) \rightarrow$ substituţia sub care unifică propoziţiile α şi β;
- $subst(S, \alpha) \rightarrow$ propoziţia rezultată în urma aplicării substituţiei S asupra propoziţiei α.

Sfârșitul cursului 9
Ce am învățat

· Bazele logicii propoziționale și cu predicate de ordinul I, Sintaxă și Semantică, Forme normale, Demonstrații, Axiome, Rezoluția ca regulă de inferență, Unificare.

Cursul 10

Programare logică în Prolog

Cuprins

Introducere

Programare logică

- Reprezentare simbolică;
- Stil declarativ;
- Separarea datelor de procesul de inferenţă, incorporat în mediul de execuție;
- Uniformitatea reprezentării axiomelor şi a regulilor de derivare;
- Reprezentarea modularizată a cunoştinţelor;
- Posibilitatea modificării dinamice a programelor, prin adăugarea şi retragerea axiomelor şi a regulilor.

Prolog
Execuție

- Bazat pe FOL restricționat;
- "Calculul" → satisfacerea de scopuri, prin reducere la absurd;
- Regula de inferență → rezoluţia, cu unificare;
- Strategia de control, din evoluţia demonstraţiilor:
  - *backward chaining*: de la scop către axiome;
  - parcurgere în adâncime, în arborele de derivare;
    - pericolul coborârii pe o cale infinită, ce nu conţine soluţia → strategie incompletă;
    - eficienţă sporită în utilizarea spaţiului.

Prolog
Program

- Exclusiv clauze Horn:

$$\begin{array}{rcll} A_1 \wedge \ldots \wedge A_n & \Rightarrow & A & \text{(Regulă)} \\ true & \Rightarrow & B & \text{(Axiomă)} \end{array}$$

- Absenţa negaţiilor explicite → desprinderea falsităţii pe baza imposibilităţii de a demonstra;
- Ipoteza lumii închise (*closed world assumption*) → ceea ce nu poate fi demonstrat este fals;
- Prin opoziţie, ipoteza lumii deschise (*open world assumption*) → nu se poate afirma nimic despre ceea ce nu poate fi demonstrat.

Axiome şi reguli

Un prim exemplu de program Prolog

Exemplul 36.1.

```
% 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
La consola Prolog

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

?- parent(andrei, cristi).
false.

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

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

- ";" → solicitarea următorului răspuns;
- "." → oprire după *acest* răspuns.

Exemplu de utilizare
Concatenarea a două liste

Exemplul 36.2.

```
% 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.
```


Demonstrare

Paşi în demonstrare (1)

1. Iniţializarea stivei de scopuri cu scopul solicitat;
2. Iniţializarea substituţiei utilizate pe parcursul unificării cu mulţimea vidă;
3. Extragerea scopului din vârful stivei şi determinarea primei clauze din program cu a cărei concluzie unifică;
4. Îmbogăţirea corespunzătoare a substituţiei şi adăugarea premiselor clauzei în stivă, în ordinea din program;
5. Salt la pasul 3.

Paşi în demonstrare (2)

6. În cazul imposibilităţii satisfacerii scopului din vârful stivei, revenirea la scopul anterior (*backtracking*), şi încercarea altei modalităţi de satisfacere;
7. Succes la golirea stivei de scopuri;
8. Eşec la imposibilitatea satisfacerii ultimului scop din stivă.

Exemplul genealogic (1)

Bazat pe Exemplul 36.1

$S = \emptyset$
$G = \{gp(X, Y)\}$

↓

`gp(X1, Y1) :- p(X1, Z1), p(Z1, Y1)`

↓

$S = \{X = X1, Y = Y1\}$
$G = \{p(X1, Z1), p(Z1, Y1)\};$

↓ `p(andrei, bogdan)` ↓ $\cdots 1$

↓ `p(andrei, bianca)` ↓ $\cdots 2$

↓ `p(bogdan, cristi)` ↓ $\cdots 3$

Exemplul genealogic (2)

Ramura 1

$\cdots 1$

↓

`p(andrei, bogdan)`

↓

$S = \{X = X1, Y = Y1, X1 = andrei, Z1 = bogdan\}$
$G = \{p(bogdan, Y1)\};$

↓

`p(bogdan, cristi)`

↓

$S = \{X = X1, Y = Y1, X1 = andrei, Z1 = bogdan, Y1 = cristi\}$
$G = \emptyset;$

↓

success

`gp(andrei, cristi)`

Exemplul genealogic (3)

Ramura 2

$\cdots 2$

↓

`p(andrei, bianca)`

↓

$S = \{X = X1, Y = Y1, X1 = andrei, Z1 = bianca\}$
$G = \{p(bianca, Y1)\};$

↓

eșec

Exemplul genealogic (4)

Ramura 3

$\cdots 3$

↓

`p(bogdan, cristi)`

↓

$S = \{X = X1, Y = Y1, X1 = bogdan, Z1 = cristi\}$
$G = \{p(cristi, Y1)\};$

↓

eșec

Observaţii

- Ordinea evaluării / încercării demonstrării scopurilor
  - Ordinea clauzelor în program;
  - Ordinea premiselor în cadrul regulilor.
- Recomandare: premisele mai uşor de satisfăcut și mai specifice primele – exemplu: axiome.

Strategii de control

Ale demonstratiilor

Forward chaining (*data-driven*)

- Derivarea tuturor concluziilor, pornind de la datele iniţiale;
- Oprire la obţinerea scopului (scopurilor);

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 ş.a.m.d.

Strategii de control I

Algoritm Backward chaining

```
BackwardChaining(rules, goals, subst)
 lista regulilor din program, stiva de scopuri, substituţia
 curentă, iniţial vidă.
 returns satisfiabilitatea scopurilor
if goals = ∅ then
   return SUCCESS
goal ← head(goals)
goals ← tail(goals)
for-each rule ∈ rules do      // în ordinea din program
   if unify(goal, conclusion(rule), subst) → bindings
      newGoals ← premises(rule) ∪ goals      // adâncime
      newSubst ← subst ∪ bindings
      if BackwardChaining(rules, newGoals, newSubst)
         then return SUCCESS
return FAILURE
```


Controlul execuţiei

Exemplu – Minimul a două numere

Cod Prolog

Exemplul 38.1 (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.
```


Exemplu – Minimul a două numere

Utilizare

```
?- 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.
```


Exemplu – Minimul a două numere

Observaţii

- Condiţii mutual exclusive: `X =< Y` şi `X > Y` → cum putem elimina redundanţa?

Exemplul 38.2.

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

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

- Greşit!

Exemplu – Minimul a două numere

Îmbunătăţire

- Soluţie: oprirea recursivităţii după prima satisfacere a scopului.

Exemplul 38.3.

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

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


Operatorul *cut*

Definiţie

- La prima întâlnire → satisfacere;
- La a doua întâlnire în momentul revenirii (*backtracking*) → eşec, cu inhibarea tuturor căilor ulterioare de satisfacere a scopului care a unificat cu concluzia regulii curente;
- Utilitate în eficientizarea programelor.

Operatorul *cut*

Exemplu

Exemplul 38.4.

```
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).
```

Operatorul *cut*

Utilizare

```
?- 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.
```


Negaţia ca eşec

Exemplul 38.5.

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

- `P`: atom – exemplu: `boy(john)`
- dacă `P` este satisfiabil:
  - eşecul primei reguli, din cauza lui `fail`;
  - abandonarea celei de-a doua reguli, din cauza lui `!`;
  - rezultat: `nott(P)` nesatisfiabil.
- dacă `P` este nesatisfiabil:
  - eşecul primei reguli;
  - succesul celei de-a doua reguli;
  - rezultat: `nott(P)` satisfiabil.

Sfârșitul cursului 10

Ce am învățat

· Prolog: structura unui program, funcționarea unei demonstrații, ordinea evaluării, algoritmul de control al demonstrației, tehnici de control al execuției.

Cursul 11

Maşina algoritmică Markov

Cuprins

Introducere

Maşina algoritmică Markov

- Model de calculabilitate efectivă, echivalent cu Maşina Turing şi Calculul Lambda;
- Principiul de funcţionare: *pattern matching* şi substituţie;
- Fundamentul teoretic al paradigmei asociative şi al limbajelor bazate pe reguli (de forma *dacă-atunci*).

Paradigma asociativă

Caracteristici

- Potrivită mai ales în cazul problemelor ce nu admit o soluţie precisă algoritmică (ieftină);
- Codificarea cunoştinţelor specifice unui domeniu şi aplicarea lor într-o manieră euristică;
- Descrierea proprietăţilor soluţiei, prin contrast cu paşii care trebuie realizaţi pentru obţinerea acesteia (*ce* trebuie obţinut vs. *cum*);
- Absenţa unui flux explicit de control, deciziile fiind determinate, implicit, de cunoştinţele valabile la un anumit moment → *data-driven control*.

Maşina algoritmică Markov

Structură

RD	S
	↑
	UC
	↑
SA	AM

- Registrul de date, RD, cu secvenţa de simboluri, S
- Unitatea de control, UC
- Spaţiul de stocare a algoritmului, SA, ce conţine algoritmul Markov, AM

Registrul de date

Spaţiul de lucru al maşinii

- Nemărginit la dreapta
- Simboluri din alfabetul $A_b \cup A_l$:
  - A_b: alfabetul de bază
  - A_l: alfabetul local / de lucru
  - $A_b \cap A_l = \emptyset$
- Şirurile iniţial şi final formate doar cu simboluri din A_b;
- Simbolurile din A_l utilizabili exclusiv în timpul execuţiei;
- Şirul de simboluri posibil vid.

Reguli

Algoritmul după care lucrează maşina

- Unitatea de bază a unui algoritm Markov → regula asociativă de substituţie:

  `şablon identificare (LHS) → şablon substituţie (RHS)`
- Exemplu: `ag1c -> ac`
- Şabloanele → secvenţe de simboluri:
  - constante: simboluri din A_b
  - variabile locale: simboluri din A_l
  - variabile generice: simboluri speciale, din mulţimea G, legaţi la simboluri din A_b
- Dacă `RHS` este "." → regulă terminală, ce încheie execuţia maşinii.

Variabile generice

- De obicei, notate cu `g`, urmat de un indice;
- Mulţimea valorilor pe care le poate lua o variabilă → domeniul variabilei – `Dom(g)`;
- Legate la exact un simbol la un moment dat;
- Durata de viaţă → timpul aplicării regulii;
- Utilizabile în `RHS` doar în cazul apariţiei în `LHS`.

Algoritmi

Conţin programele

- Mulţimi ordonate de reguli, îmbogăţite cu declaraţii:
  - de partiţionare a mulţimii A_b
  - de variabile generice

Exemplul 40.1.

Eliminarea din mulţimea A simbolurilor ce aparţin mulţimii M:

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

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

- $A, B \subseteq A_b$
- g_1, g_2 → variabile generice
- a nedeclarată → variabilă locală ($a \in A_l$)

Reguli

Aplicabilitate

Definiţia 40.2 (Aplicabilitatea unei reguli).

Regula $r : a_1 \ldots a_n \to b_1 \ldots b_m$ este aplicabilă dacă şi numai dacă există un subşir $c_1 \ldots c_n$, în RD, astfel încât $\forall i = \overline{1,n}$ exact 1 condiţie din cele de mai jos este îndeplinită:

- $a_i \in A_b \wedge a_i = c_i$
- $a_i \in A_l \wedge a_i = c_i$
- $a_i \in G \wedge$
  $(\forall j = \overline{1,n} . a_j = a_i \Rightarrow c_j \in Dom(a_i) \wedge c_j = c_i)$,

  i.e. variabila a_i este legată la o valoare unică, obţinută prin potrivirea dintre şablon şi subşir.

Reguli

Aplicare

Definiţia 40.3 (Aplicarea unei reguli).

Aplicarea regulii $r : a_1 \ldots a_n \to b_1 \ldots b_m$ asupra unui subşir $s : c_1 \ldots c_n$, în raport cu care este aplicabilă, constă în substituirea lui s prin subşirul $q_1 \ldots q_m$, calculat astfel:

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

Exemplu de aplicare

Exemplul 40.4.

- A_b = {1, 2, 3}
- A_1 = {x, y}
- Dom(g_1) = {2}
- Dom(g_2) = A_b
- S = 1111112x2y31111
- r : $1g_1xg_1yg_2 \rightarrow 1g_2x$

```
S   =  11111  1  2   x  2   y  3    1111
r   :         1  g1  x  g1  y  g2   → 1g2x
S'  =  1111113x1111
```


Aplicabilitate vs. aplicare

Comentarii

- Aplicabilitatea
  - unei reguli pentru mai multe subşiruri;
  - mai multor reguli pentru acelaşi subşir.
- La un anumit moment, aplicarea propriu-zisă a unei singure reguli asupra unui singur subşir;
- Nedeterminism inerent, ce trebuie exploatat, sau rezolvat
- Convenţie care poate fi făcută:
  - aplicarea primei reguli aplicabile, asupra
  - celui mai din stânga subşir asupra căreia este aplicabilă

Unitatea de control

Funcționare

Regula 1
↓
. . .
↓
Regula *k* : S $\xrightarrow{aplicare}$ S' — înapoi la Regula 1 ↑
prima regulă aplicabilă
. . .
Regula *n*

Unitatea de control

Etape

- Analogia cu o sită pe mai multe nivele, ce corespund regulilor;
- Secvenţialitatea testării aplicabilităţii, nu a aplicării propriu-zise!
- Etape:
  1. determinarea primei reguli *aplicabile*
  2. aplicarea acesteia
  3. actualizarea RD
  4. salt la pasul 1

Unitatea de control

Algoritm

control(*S*, *Rules*)
1. $i \leftarrow 1$; $n \leftarrow |Rules|$; *status* ← RUNNING
2. **while** $i \leq n$ **and** *status* = RUNNING
3. $r \leftarrow Rules[i]$
4. **if** *isApplicable*(*S*, *r*) **then**
5. $S \leftarrow$ *fire*(*S*, *r*)
6. **if** *isTerminal*(*r*) **then**
7. *status* ← TERMINATED
8. **else**
9. $i \leftarrow 1$
10. **else**
11. $i \leftarrow i+1$
12. **if** *status* = TERMINATED **then**
13. **return** *S*
14. **else** *error*("Execution blocked")

Un exemplu

Inversarea intrării

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

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

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

Sfârșitul cursului 11

Ce am învățat

· Ce este și cum funcționează mașina algoritmică Markov: structură, variabile, reguli, algoritmul unității de control.

Cursul 12

Programare asociativă în CLIPS

Cuprins

Introducere

CLIPS

- "C Language Integrated Production System";
- Sistem bazat pe reguli → "producţie" = regulă;
- Principiu de funcţionare similar cu al maşinii Markov;
- Dezvoltat la NASA în anii 1980;
- Posibilitatea codificării de implicaţii logice în reguli → sisteme expert.

Sisteme expert

Trăsături

- Edward Feigenbaum: "un program inteligent care foloseşte cunoştinţe şi reguli de inferenţă pentru a rezolva probleme suficient de dificile încât să necesite expertiză umană semnificativă";
- Mimarea procesului de decizie al unui expert uman;
- Limitarea la un domeniu specific → dificultatea scrierii unui rezolvitor general de probleme.

Sisteme expert

Aplicaţii

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

. . .

Fapte şi reguli

Exemplu

Minimul a două numere – reprezentare individuală

Exemplul 42.1.

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

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


Fapte

- Reprezentarea datelor prin fapte → similare simbolurilor maşinii Markov;
- Afirmaţii despre atributele obiectelor;
- Date simbolice, construite conform unor şabloane;
- Mulţimea de fapte → baza de cunoştinţe (*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 maşinii Markov;
- Şablon de identificare → secvenţă de fapte parametrizate (vezi variabilele generice ale algoritmilor Markov) şi restricţii;
- Şablon de acţiune → secvenţă de acţiuni;
- *Pattern matching* secvenţial pe faptele din şablonul de identificare;
- Domeniul de vizibilitate a unei variabile → restul regulii, după prima apariţie a variabilei, în şablonul de identificare.

Înregistrări de activare

- Tuplul (regulă, fapte asupra cărora este aplicabilă) → înregistrare de activare (*activation record*);
- Reguli posibil aplicabile asupra diferitelor porţiuni ale aceloraşi fapte;
- Muţimea înregistrărilor de activare → agenda.

Înregistrări de activare

Exemplu – reluat de mai devreme: minimul a 2 numere

```
> (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)
```


Înregistrări de activare

Principiul refracţiei

- Aplicarea unei reguli o singură dată asupra aceloraşi fapte şi aceloraşi porţiuni ale acestora;
- Altfel, programe care nu s-ar termina.

Terminarea programelor

- Aplicarea unui număr maxim de reguli → `(run n)`;
- Întâlnirea acţiunii `(halt)`;
- Golirea agendei.

Exemple

Minimul a două numere

Reprezentare agregată a numerelor – Exemplu

Exemplul 43.1.

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

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


Minimul a două numere

Reprezentare agregată a numerelor – Observaţii

- `$?` este o variabilă anonimă, ce se potriveşte cu orice secvenţă, 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
Reprezentare agregată a numerelor – Exemplu (2)

Exemplul 43.2.

```
(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
Reprezentare agregată a numerelor – Observații (2)

- Selectarea explicită a celor 2 numere împiedică alegerea automată, convenabilă, a acestora, ca în Exemplul 43.1 → necesitatea celor 2 reguli.

```
> (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
Exemplu

Exemplul 43.3.

```
(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
Interogare (1)

```
> (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
Interogare (2)

```
> (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!
```


Suma oricâtor numere
Observații

- 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
Exemplu corect

Exemplul 43.4.

```
(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)))
```

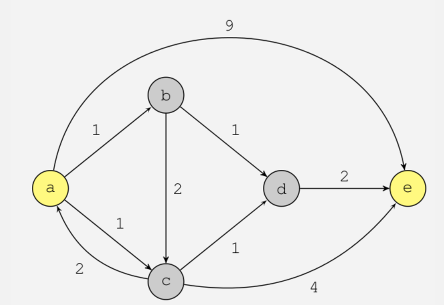

Suma oricâtor numere
Interogare pe exemplul corect (1)

```
> (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
Interogare pe exemplul corect (2)

```
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
Exemplu de graf



Accesibilitatea într-un graf
Definiția problemei

- Graful: $G = (V, E)$
- Relaţia 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
Cod (1)

Exemplul 43.5.

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


Accesibilitatea într-un graf
Cod (2)

Exemplul 43.5 (continuare).

```
(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
Cod (3)

Exemplul 43.5 (continuare).

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


Controlul execuţiei

Accesibilitatea într-un graf
Optimizare

- Exemplul 43.5: posibilitatea continuării explorării grafului după obţinerea răspunsului căutat;
- Optimizare: forţarea aplicării regulii `found`, imediat după identificarea răspunsului;
- Problemă: aplicabilitatea concomitentă a regulilor `expand` şi `found`;
- Soluţie: prioritizarea regulii `found`.

Accesibilitatea într-un graf

Optimizare pe exemplu

Exemplul 44.1 (optimizare a exemplului 43.5).

```
(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 în aplicare a unei reguli;
- Implicit 0, posibil negativă;
- Valoare mai mare → prioritate mai mare.

Minimul oricâtor numere

Determinare iterativă

Exemplul 44.2.

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

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


Minimul oricâtor numere

Determinare iterativă

Exemplul 44.2.

```
(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

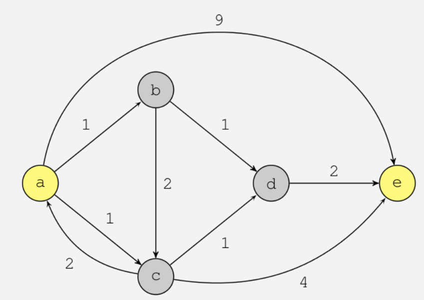
Exemplul 44.3.

```
(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 condiţii *inline* asupra variabilelor, prin `&`;
- Citirea regulii: “Minimul este acel element pentru care nu găsim altul mai mic”.

Drumurile optime în graf

Pe exemplul anterior de graf



Drumurile optime într-un graf cu costuri

Principiu

- Drumurile optime $source \rightsquigarrow dest$ sunt drumurile *utile* $source \rightsquigarrow dest$, în momentul în care nu se mai pot obţine alte drumuri *utile* prin extinderea drumurilor *utile* existente;
- Iniţial, există un singur drum, ce conţine doar nodul *source* şi are costul 0;
- Un drum $x \rightsquigarrow y, z$ extinde un drum $x \rightsquigarrow y$ dacă $(y,z) \in E$ şi $z \notin x \rightsquigarrow y$, unde $cost(x \rightsquigarrow y, z) = cost(x \rightsquigarrow y) + cost(y,z)$;
- Un drum $x \rightsquigarrow y$ se numeşte *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

Implementare (1)

Exemplul 44.4.

```
(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
Implementare (2)

Exemplul 44.4 (continuare).

```
(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
Implementare (3)

Exemplul 44.4 (continuare).

```
(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
Problema

- Criteriul optimizat: numărul de muchii;
- Soluţia 1: abordarea precedentă, presupunând că toate muchiile au costul 1;
- Soluţia 2: parcurgere în lăţime.

Drumurile optime într-un graf fără costuri
Strategia CLIPS pe lăţime

- Parcurgere în lăţime → necesitatea extinderii, într-un pas, a unei cele mai scurte căi;
- Observaţie: vârsta superioară a faptelor reprezentând căi mai scurte;
- Soluţie: alterarea ordinii în care faptele sunt evaluate în raport cu şabloanele de identificare ale regulilor.

Drumurile optime într-un graf fără costuri
Implementare (1)

Exemplul 44.5.

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

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


Drumurile optime într-un graf fără costuri
Implementare (2)

Exemplul 44.5 (continuare).

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

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


Strategii
Pentru ordinea potrivirii faptelor în CLIPS

- Ordinea în care faptele sunt evaluate în raport cu şabloanele de identificare ale regulilor;
- *Depth* (implicită): cel mai recent fapt potrivit primul;
- *Breadth*: cel mai vechi fapt potrivit primul;
- *Random*: alegere aleatorie.

Sfârșitul cursului 12
Ce am învățat

· CLIPS: fapte și reguli, înregistrări de activare, exemple de utilizare pe probleme simple, elemente de controlul execuției – salience și strategii.