Paradigme de Programare

Ș.l. dr. ing. Andrei Olaru

Departamentul Calculatoare
slides: Andrei Olaru & Mihnea Muraru

2013 – 2014, semestrul 2

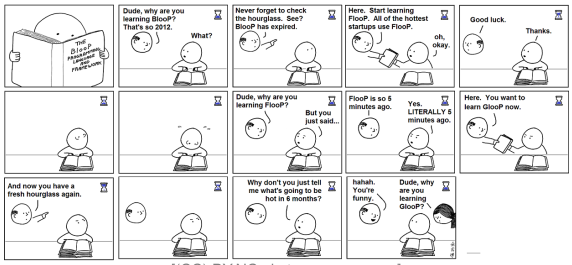
Cursul 1

Introducere

Cuprins

BlooP and FlooP and GlooP

[http://abstrusegoose.com/503]



Organizare

Unde găsesc informații?

Resurse de bază

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

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

Teme și forumuri: curs.cs → L-2-PP-CA-CC
`http://cs.curs.pub.ro/2013/course/view.php?id=201`

Elementele cursului sunt comune la seriile CA și CC.

Curs

- Teorie despre limbaje și mașinile de calcul din spatele lor.
- Detalii despre limbajele studiate (Scheme, Haskell, Prolog).
- Aplicații hands-on în limbajele studiate.

Laborator

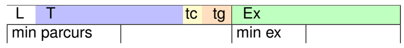
- Accent pe lucrul efectiv.
- Utile pentru pregătirea temelor.
- Parcurgerea documentaţiei înaintea laboratorului.

Teme

- Aplicații mai complexe în limbajele studiate, folosind tehnicile studiate atât la curs, cât și la laborator.
- Verificare automată folosind vmchecker.
- Verificare umană.
- Verificare la copiere.

Notare
mai multe la http://elf.cs.pub.ro/pp/regulament

- Laborator: 1p ← cu bonusuri, dar maxim 1p total
- Teme: 4p (3 × 1.33p) ← cu bonusuri, dar în limita a maxim 6p pe parcurs
- Teste la curs: 0.5p ← evaluează prezența, dar și atenția la curs
- Test din materia de laborator: 0.5p ← test grilă, de cunoaștere a limbajelor
- Examen: 4p ← limbaje + teorie



Obiective

De ce?

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.
- Adaptarea la transformarea limbajelor în limbaje multi-paradigmă.

De ce?
Câteva exemple

- prelucrare secvențială (exemplu în Java):

```
for(int i = 0; i < 10; i++)
    System.out.println("This is number" + i);
```

- lucrul cu mulțimi de numere (exemplu în Haskell):

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

- lucrul cu funcții ca valori de prim rang (exemplu în Scheme):

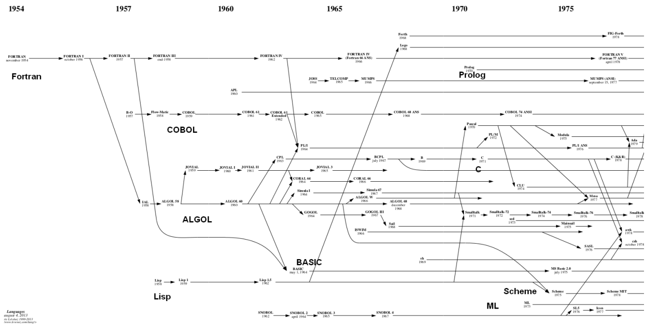
```
(do-testing Testing-matter
    (lambda (item) (> (car item) (cdr item))))
```


Ce vom studia?
Conținutul cursului

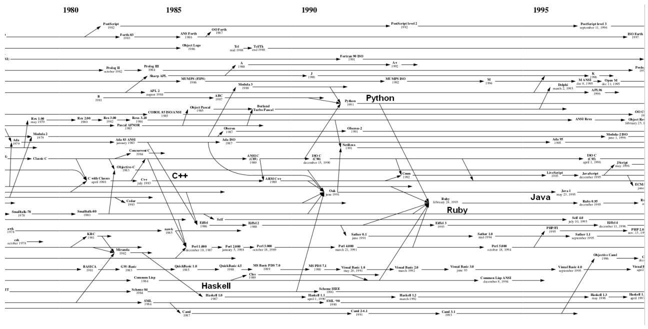
1. Diverse perspective conceptuale asupra noţiunii de calculabilitate efectivă → modele de calculabilitate.
2. Influenţa perspectivei alese asupra procesului de modelare şi rezolvare a problemelor → paradigme de programare.
3. Limbaje de programare aferente paradigmelor, cu accent pe aspectul comparativ.

Paradigme și limbaje de programare

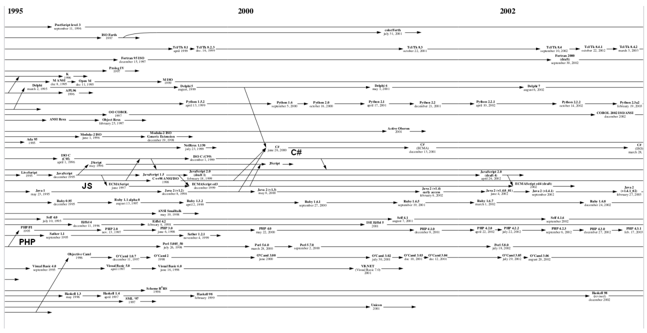
Istorie
1950-1975



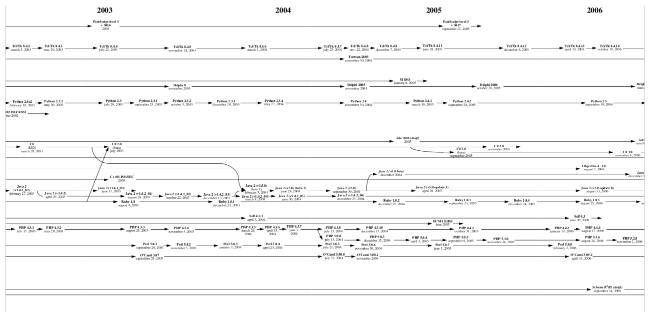
Istorie
1975-1995



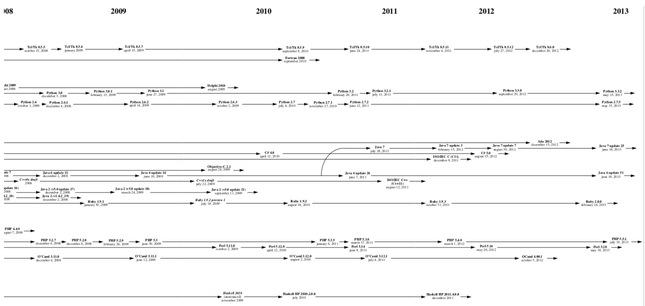
Istorie
1995-2002



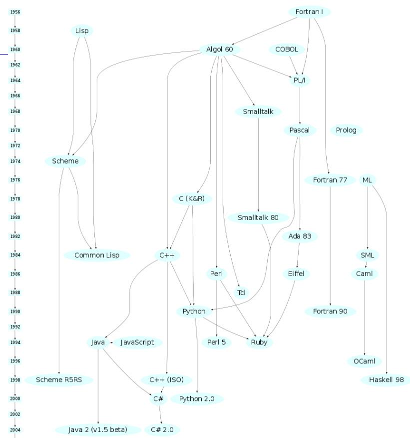
Istorie
2002-2006



Istorie
2006-2013



Istorie
'56-'04 pe scurt



Istorie
Resurse

- imagine navigabilă (slides precedente):
  [http://www.levenez.com/lang/]
- poster:
  [http://oreilly.com/pub/a/oreilly/news/languageposter_0504.html]
- arbore din slide precedent și arbore extins:
  [http://rigaux.org/language-study/diagram.html]
- Wikipedia:
  [http://en.wikipedia.org/wiki/Generational_list_of_programming_languages]

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

Modele → paradigme → limbaje
Modele de calculabilitate

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

Exemplu introductiv

O primă problemă

Exemplul 4.1.
Să se determine elementul minim dintr-un vector.

· $m \in V, \forall x \in V, m \leq x$

Modelare imperativă
Variantă procedurală

minList(L, n)
  $min \leftarrow L[0]$
  $i \leftarrow 1$
  while $i < n$ **do**
    if $L[i] < min$ **then**
      $min \leftarrow L[i]$
    end if
    $i \leftarrow i+1$
  end while
  return min

Modelare imperativă
Variantă modernă

minList(L)
  $min \leftarrow L[0]$
  for $x \in L$ **do**
    if $x < min$ **then**
      $min \leftarrow x$
    end if
  end for
  return min

Modelare funcţională

- Ideea: minList(L) = if(eq(length(L), 1),
  head(L),
  min(head(L), minList(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)
```

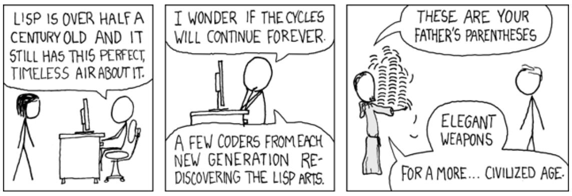

Modelare logică

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


Introducere în Scheme

Lips cycles
[http://xkcd.com/297/]



[(CC) BY-NC xkcd.com]

Scheme
din 1975

- funcțional
- dialect de Lisp
- totul este văzut ca o funcție
- constante – expresii neevaluate
- perechi / liste pentru structurarea datelor
- apeluri de funcții – liste de apelare, evaluate
- evaluare aplicativă, funcții stricte, cu anumite excepții

Ce am învățat
Sfârșitul primului curs

· Paradigme de programare, limbaje,

modele de calculabilitate, The law of instrument,

introducere Scheme.

Cursul 2

Calcul Lambda

Cuprins

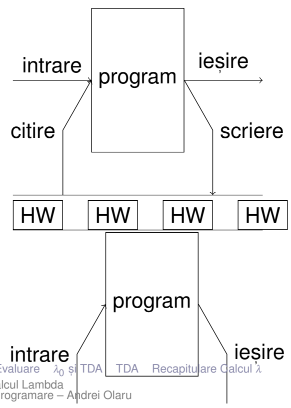
Introducere

Calculul Lambda
λ

- Model de calculabilitate (Alonzo Church, 1932) – introdus în cadrul cercetărilor asupra fundamentelor matematicii.
  [http://en.wikipedia.org/wiki/Lambda_calculus]
  - sistem formal pentru exprimarea calculului.
- Echivalent cu Maşina Turing (v. Teza Church-Turing)
- Axat pe conceptul matematic de funcţie – totul este o funcţie

Calculul Lambda
Perspectivă asupra efectelor laterale

- Mașină fizică
- Mașina Turing
- Calcul Lambda



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.

Lambda-expresii

λ-expresii
Exemple

Exemplul 7.1.

1. $x \to$ variabila (numele) x
2. $\lambda x.x \to$ funcţia identitate
3. $\lambda x.\lambda y.x \to$ funcţie selector
4. $(\lambda x.x\ y) \to$ aplicaţia funcţiei identitate asupra parametrului actual y
5. $(\lambda x.(x\ x)\ \lambda x.x)$

- Intuitiv, evaluarea aplicaţiei $(\lambda x.x\ y)$ presupune substituția textuală a lui x, în corp, prin $y \to$ rezultat y.

λ-expresii
Definiţie

Definiția 7.2 (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.

Evaluare
Intuitiv

$$((\lambda x.\lambda y.x\ z)\ t)$$
$$\| \text{ substituție } \Downarrow$$
$$(\lambda y.z\ t)$$
$$\| \text{ substituție } \Downarrow$$
$$z$$

nu mai este nicio funcție de aplicat

· cum știm cum reducem, în ce ordine, și ce apariții ale variabilelor înlocuim?

Reducere

β-redex
Cum arată (Formal, vezi definiția 8.10)

- β-redex: o λ-expresie de forma: $(\lambda x.E\ \ A)$
  - E – λ-expresie – este corpul funcției
  - A – λ-expresie – este parametrul actual
- β-redexul se reduce la $E_{[A/x]}$ – E cu toate aparițiile libere ale lui x din E înlocuite cu A prin substituție textuală (vezi definiția 8.9 mai târziu).

Apariţii ale variabilelor
Legate vs libere

Definiția 8.1 (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 8.2 (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!

Apariții ale variabilelor
Mod de gândire

· O apariție legată în expresie este o apariție a parametrului formal al unei funcții definite în expresie, și este în corpul funcției respective; o apariție liberă este apariție a parametrului formal al unei funcții definite în exteriorul expresiei, sau nu este parametru formal al niciunei funcții.

- x_1 – apariție liberă
- $(\lambda y.x_1\ z)$ – apariție încă liberă, nu o leagă nimeni
- $\lambda x_2.(\lambda y.x_1\ z)$ – λx_2 *leagă* apariția x_1
- $(\lambda x_2.\underbrace{(\lambda y.x_1\ z)}_{\text{corp}}\ x_3)$ – apariția x_3 este liberă – este în exteriorul corpului funcției cu parametrul formal x (x_2)
- $\lambda x_4.(\lambda x_2.(\lambda y.x_1\ z)\ x_3)$ – λx_4 *leagă* apariția x_3

Variabile
Legate vs libere

Definiția 8.3 (Variabilă legată).

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

Definiția 8.4 (Variabilă liberă).

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

- Atenție! În raport cu o expresie dată!

Variabile și Apariții ale lor
Exemplu 1

Exemplul 8.5.

În expresia $E = (\lambda x.x\ x)$, evidențiem aparițiile 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 și F

Variabile și Apariții ale lor
Exemplu 2

Exemplul 8.6.

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

- 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 și 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 8.7 (Expresie închisă).

Expresie ce nu conține variabile libere.

Exemplul 8.8.

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

β-reducere
Definiție

Definiția 8.9 (β-reducere).

Evaluarea expresiei $(\lambda x.E\ A)$, cu E și A λ-expresii, 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 8.10 (β-redex).

Expresia $(\lambda x.E\ A)$, cu E și A λ-expresii – o expresie pe care se poate aplica β-reducerea.

β-reducere
Exemple

Exemplul 8.11.

- $(\lambda x.x\ y) \rightarrow_\beta x_{[y/x]} \rightarrow y$
- $(\lambda x.\lambda x.x\ y) \rightarrow_\beta \lambda x.x_{[y/x]} \rightarrow \lambda x.x$
- $(\lambda x.\lambda y.x\ y) \rightarrow_\beta \lambda y.x_{[y/x]} \rightarrow \lambda y.y$ Greșit! Variabila liberă y devine legată, schimbându-și semnificația. $\rightarrow \lambda y^{(a)}.y^{(b)}$
  Care este problema?

β-reducere
Coliziuni

- Problemă: în expresia $(\lambda x.E\ A)$:
  - variabilele libere din A nu au nume comune cu variabilele legate din E: $FV(A) \cap BV(E) = \emptyset$
    $\rightarrow$ reducere întotdeauna corectă
  - există variabilele libere din A care au nume comune cu variabilele legate din E: $FV(A) \cap BV(E) \neq \emptyset$
    $\rightarrow$ reducere potențial greșită
- Soluție: redenumirea variabilelor legate din E, ce coincid cu cele libere din $A \rightarrow \alpha$-conversie.

Exemplul 8.12.
$(\lambda x.\lambda y.x\ y) \rightarrow_\alpha (\lambda x.\lambda z.x\ y) \rightarrow_\beta \lambda z.x_{[y/x]} \rightarrow \lambda z.y$

α-conversie
Definiție

Definiția 8.13 (α-conversie).
Redenumirea sistematică a variabilelor legate dintr-o funcție: $\lambda x.E \rightarrow_\alpha \lambda y.E_{[y/x]}$. Se impun două condiții.

Exemplul 8.14.
- $\lambda x.y \rightarrow_\alpha \lambda y.y_{[y/x]} \rightarrow \lambda y.y \rightarrow$ Greșit!
- $\lambda x.\lambda y.x \rightarrow_\alpha \lambda y.\lambda y.x_{[y/x]} \rightarrow \lambda y.\lambda y.y \rightarrow$ Greșit!

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

α-conversie
Exemple

Exemplul 8.15.
- $\lambda x.(x\ y) \rightarrow_\alpha \lambda z.(z\ y) \rightarrow$ Corect!
- $\lambda x.\lambda x.(x\ y) \rightarrow_\alpha \lambda y.\lambda x.(x\ y) \rightarrow$ Greșit! y este liberă în $\lambda x.(x\ y)$
- $\lambda x.\lambda y.(y\ x) \rightarrow_\alpha \lambda y.\lambda y.(y\ y) \rightarrow$ Greșit! Apariția liberă a lui x din $\lambda y.(y\ x)$ devine legată, după substituire, în $\lambda y.(y\ y)$
- $\lambda x.\lambda y.(y\ y) \rightarrow_\alpha \lambda y.\lambda y.(y\ y) \rightarrow$ Corect!

Reducere
Definiții

Definiția 8.16 (Pas de reducere).
O secvență formată dintr-o α-conversie și 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 8.17 (Secvență de reducere).
Succesiune de zero sau mai mulți pași de reducere:
$E_1 \rightarrow^* E_2$. Reprezintă un element din închiderea reflexiv-tranzitivă a relației $\rightarrow$.

Reducere
Proprietăți

- $E_1 \rightarrow E_2 \implies E_1 \rightarrow^* E_2$ – un pas este o secvență
- $E \rightarrow^* E$ – zero pași formează o secvență
- $E_1 \rightarrow^* E_2 \land E_2 \rightarrow^* E_3 \implies E_1 \rightarrow^* E_3$ – tranzitivitate

Exemplul 8.18.
- $((\lambda x.\lambda y.(y\ x)\ y)\ \lambda x.x) \rightarrow (\lambda z.(z\ y)\ \lambda x.x) \rightarrow (\lambda x.x\ y) \rightarrow y$
  $\Rightarrow$
- $((\lambda x.\lambda y.(y\ x)\ y)\ \lambda x.x) \rightarrow^* y$

Sfârșitul primei părți

Evaluare

Întrebări
Pentru construcția unei mașini de calcul

· Dacă am vrea să construim o mașină de calcul care să aibă ca program o λ-expresie și să aibă ca operație de bază pasul de reducere, ne punem câteva întrebări:

1. Când se termină calculul? Se termină întotdeauna?
2. Comportamentul depinde de secvența de reducere?
3. Dacă mai multe secvențe de reducere se termină, obținem întotdeauna același rezultat?
4. Dacă rezultatul este unic, cum îl obținem?

Forme normale
Cum știm că s-a terminat calculul?

· Calculul se termină atunci când expresia nu mai poate fi redusă → expresia nu mai conține β-redecși.

Definiția 9.1 (Formă normală).

Formă a unei expresii, ce nu mai conţine β-redecşi i.e. care nu mai poate fi redusă.

Definiția 9.2 (Formă normală funcţională – FNF).

$\lambda x.F$, chiar dacă F conţine β-redecşi.

Exemplul 9.3.

$(\lambda x.\lambda y.(x\ y)\ \lambda x.x) \rightarrow_{FNF} \lambda y.(\lambda x.x\ y) \rightarrow_{FN} \lambda y.y$

Terminarea reducerii (reductibilitate)
Exemplu și definiție

Exemplul 9.4.

$\Omega = (\lambda x.(x\ x)\ \lambda x.(x\ x)) \rightarrow (\lambda x.(x\ x)\ \lambda x.(x\ x)) \rightarrow^* \ldots$

Ω nu admite o secvenţă de reducere, care se termină.

Definiția 9.5 (Expresie reductibilă).

Expresie ce admite (vreo) secvenţă de reducere care se termină.

· expresia Ω nu este reductibilă.

Secvenţe de reducere
și terminare

Exemplul 9.6.

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

- E are o secvenţă de reducere care nu se termină;
- dar E are forma normală $y \rightarrow E$ este reductibilă;
- lungimea secvenţelor de reducere ale E este nemărginită.

Unicitatea formei normale
Rezultate

Teorema 9.7 (Church-Rosser / diamantului).

Dacă $E \rightarrow^* E_1$ și $E \rightarrow^* E_2$, atunci există E_3 astfel încât $E_1 \rightarrow^* E_3$ și $E_2 \rightarrow^* E_3$.

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

Corolarul 9.8.

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

Unicitatea formei normale
Exemplu

Exemplul 9.9.

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

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

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

⇒ Valorile sunt echivalente în raport cu redenumirea.

Modalităţi de reducere
Cum putem *organiza* reducerea?

Definiția 9.10 (Reducere stânga-dreapta).

Reducerea celui mai superficial şi mai din stânga β-redex.

Exemplul 9.11.

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

Definiția 9.12 (Reducere dreapta-stânga).

Reducerea celui mai adânc şi mai din dreapta β-redex.

Exemplul 9.13.

$((\lambda x.x\ \lambda x.y)\ \underline{(\lambda x.(x\ x)\ \lambda x.(x\ x))}) \rightarrow (\lambda x.y\ \underline{\Omega}) \rightarrow \ldots$

Ce modalitate alegem?

Teorema 9.14 (Normalizării).

Dacă o expresie este reductibilă, evaluarea stânga-dreapta a acesteia se termină.

- Teorema normalizării (normalizare = aducere la forma normală) nu garantează terminarea evaluării oricărei expresii, ci doar a celor reductibile!
- Dacă expresia este ireductibilă, nicio reducere nu se va termina.

Răspunsuri la întrebări

1. Când se termină calculul? Se termină întotdeauna?
→ NU.
2. Comportamentul depinde de secvența de reducere?
→ DA.
3. Dacă mai multe secvențe de reducere se termină, obţinem întotdeauna acelaşi rezultat?
→ DA.
4. Dacă rezultatul este unic, cum îl obţinem?
→ Reducere stânga-dreapta.
5. Care este valoarea expresiei?
→ Forma normală [funcțională] (FNF).

Ordine de evaluare
Tipuri

- **Evaluare aplicativă** (*eager*) – corespunde reducerii dreapta-stânga. Parametrii funcţiilor sunt evaluaţi înaintea aplicării funcţiei.
- **Evaluare normală** (*lazy*) – corespunde reducerii stânga-dreapta. Parametrii funcţiilor sunt evaluaţi la cerere.
- **Funcţie strictă** – funcţie cu evaluare aplicativă.
- **Funcţie nestrictă** – funcţie cu evaluare normală.

Ordine de evaluare
În practică

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

Exemplul 9.15.
`(+ (+ 2 3) (* 2 3))` → `(+ 5 6)` → `11`

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

Exemplul 9.16.
`(if (< 2 3) (+ 2 3) (* 2 3))` → `(< 2 3)` → `#t` → `(+ 2 3)` → `5`

Limbajul lambda-0 și incursiune în TDA

Limbajul λ_0
Scop

- Am putea crea o mașină de calcul folosind calculul λ – mașină de calcul ipotetică;
- Mașina folosește limbajul $\lambda_0 \equiv$ calcul lambda;
- Programul → λ-expresie;
  + Legări top-level de expresii la nume.
- Datele → λ-expresii;
- Funcționarea mașinii → reducere – substituție textuală
  - evaluare normală;
  - terminarea evaluării cu forma normală funcțională;
  - se folosesc numai expresii închise.

Limbajul λ_0 – Convenţii
Evaluare

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

Exemplul 10.1.
$\lambda x.\lambda y.(x\ y) \to \lambda x\ y.(x\ y)$

Tipuri de date
Cum reprezentăm datele? Cum interpretăm valorile?

- Putem reprezenta toate datele prin funcții cărora, convențional, le dăm o semnificație abstractă.

Exemplul 10.2.
$T \equiv_{def} \lambda x.\lambda y.x \equiv \lambda x\ y.x$ $F \equiv_{def} \lambda x.\lambda y.y \equiv \lambda x\ y.y$

- Pentru aceste tipuri de date abstracte (TDA) creăm operatori care transformă datele în mod coerent cu interpretarea pe care o dăm valorilor.

Exemplul 10.3.
$not \equiv_{def} \lambda x.(x\ F\ T)$
$(not\ T) \to (\lambda x.(x\ F\ T)\ T) \to (T\ F\ T) \to F$

TDA
Definiţie

Definiția 10.4 (Tip de date abstract – TDA).
Model matematic al unei mulţimi de valori şi al operaţiilor valide pe acestea.

· Componente:
- constructori de bază: cum se generează valorile;
- operatori: ce se poate face cu acestea;
- axiome: cum lucrează operatorii / ce restricții există.

TDA *Bool*
Specificare

· Constructori:
$$\begin{array}{|ll} T: & \to Bool \\ F: & \to Bool \end{array}$$

· Operatori:
$$\begin{array}{|ll} not: & Bool \to Bool \\ and: & Bool^2 \to Bool \\ or: & Bool^2 \to Bool \end{array}$$

· Axiome:
$$\begin{array}{|ll} not: & not(T) = F \\ & not(F) = T \\ and: & and(T,a) = a \\ & and(F,a) = F \\ or: & or(T,a) = T \\ & or(F,a) = a \end{array}$$

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 *Pair*
Implementare

- Intuiţie: pereche → 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* și *Natural*
Implementare

- Intuiţie: listă → pereche (*head*, *tail*)
- $nil \equiv_{\text{def}} \lambda x.T$
- $cons \equiv_{\text{def}} pair$
  - $(cons\ e\ L) \rightarrow (\lambda x\, y\, z.(z\ x\ y)\ e\ L) \rightarrow \lambda z.(z\ e\ L)$
- $car \equiv_{\text{def}} fst$ $\quad cdr \equiv_{\text{def}} snd$

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

· vezi și [http://en.wikipedia.org/wiki/Lambda_calculus#Encoding_datatypes]

Absenţa tipurilor
Chiar avem nevoie de tipuri? – Rolul tipurilor

- Modalitate de exprimare a intenţiei programatorului;
- 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 → operator!

Absenţa tipurilor
Consecințe asupra corectitudinii calculului

- Incapacitatea Maşinii λde 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 (nu au asociată o semnificație), dar sunt ireductibile

  → instabilitate.

Absenţa tipurilor
Consecinţe pozitive

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

. . . vin cu prețul unei dificultăți sporite în depanare, verificare şi mentenanţă

Recursivitate
Perspective asupra recursivității

· Cum realizăm recursivitatea în λ_0, dacă nu avem nume de funcții?

- 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?

Combinator de punct fix
mai multe la
[http://en.wikipedia.org/wiki/Lambda_calculus#Recursion_and_fixed_points]

Exemplul 10.5.

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

- $(Fix\ F) \rightarrow (\lambda x.(F\ (x\ x))\ \lambda x.(F\ (x\ x))) \rightarrow$
  $(F\ \underline{(\lambda x.(F\ (x\ x))\ \lambda x.(F\ (x\ x)))}) \rightarrow (F\ (Fix\ F))$
- $(Fix\ F)$ este un punct fix al lui F.
- *Fix* se numește combinator de punct fix.
- $length \equiv_{def} (Fix\ Length) \rightarrow (Length\ (Fix\ Length)) \rightarrow$
  $\lambda L.(if\ (null\ L)\ zero\ (succ\ ((Fix\ Length)\ (cdr\ L))))$
- Funcţie recursivă, fără a fi textual recursivă!

Sfârșitul cursului 2
Ce am învățat

- Baza formală a calculului λ
- expresie λ, β-redex, variabile și apariții legate vs. libere, expresie închisă, α-conversie, β-reducere
- FN și FNF, reducere, reductibilitate, evaluare aplicativă și normală
- TDA și recursivitate pentru calcul lambda

Cursul 2

Anexă: TDA pentru calcul λ

Cuprins λ

TDA

TDA *Bool* λ
Specificare

· Constructori: $\begin{array}{|ll} T: & \rightarrow Bool \\ F: & \rightarrow Bool \end{array}$

· Operatori: $\begin{array}{|ll} not: & Bool \rightarrow Bool \\ and: & Bool^2 \rightarrow Bool \\ or: & Bool^2 \rightarrow Natural \end{array}$

· Axiome: $\begin{array}{|ll} not: & not(T) = F \\ & not(F) = T \\ and: & and(T,a) = a \\ & and(F,a) = F \\ or: & or(T,a) = T \\ & or(F,a) = a \end{array}$

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:
  - $(T\ a\ b) \to (\lambda x\ y.x\ a\ b) \to a$
  - $(F\ a\ b) \to (\lambda x\ y.y\ a\ b) \to b$

TDA *Bool*
Implementarea operatorilor

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

TDA *Bool*
Condiţionala *if*

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

TDA *Pair*
Specificare

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

TDA *Pair*
Implementare

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

TDA *List*
Specificare

- Constructori: $\begin{array}{|ll} nil: & \to List \\ cons: & A \times List \to List \end{array}$
- Operatori: $\begin{array}{|ll} car: & List \setminus \{nil\} \to A \\ cdr: & List \setminus \{nil\} \to List \\ null: & List \to Bool \\ append: & List^2 \to List \end{array}$
- Axiome:

$$\begin{array}{|ll} car: & car(cons(e, L)) = e \\ cdr: & cdr(cons(e, L)) = L \\ null: & null(nil) = T \\ & null(cons(e, L)) = F \\ append: & append(nil, B) = B \\ & append(cons(e, A), B) = cons(e, append(A, B)) \end{array}$$

TDA *List*
Implementare

- Intuiţie: listă →pereche (*head*, *tail*)
- $nil \equiv_{\text{def}} \lambda x.T$
- $cons \equiv_{\text{def}} pair$
  - $(cons\ e\ L) \to (\lambda x\ y\ z.(z\ x\ y)\ e\ L) \to \lambda z.(z\ e\ L)$
- $car \equiv_{\text{def}} fst$
- $cdr \equiv_{\text{def}} snd$
- $null \equiv_{\text{def}} \lambda L.(L\ \lambda x\ y.F)$
  - $(null\ nil) \to (\lambda L.(L\ \lambda x\ y.F)\ \lambda x.T) \to (\lambda x.T\ \ldots) \to T$
  - $(null\ (cons\ e\ L)) \to (\lambda L.(L\ \lambda x\ y.F)\ \lambda z.(z\ e\ L)) \to (\lambda z.(z\ e\ L)\ \lambda x\ y.F) \to (\lambda x\ y.F\ e\ L) \to F$

TDA *List*
Implementare *append*

- $append \equiv_{\text{def}}$
  $\lambda A\ B.(if\ (null\ A)\ B\ (cons\ (car\ A)\ (append\ (cdr\ A)\ B)))$

Problemă: expresia nu admite formă închisă! → vezi eliminarea recursivității textuale.

TDA *Natural*
Implementare

λ

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

Cursul 2

Anexă: Recapitulare Calcul λ

Cuprins

λ

Recapitulare Calcul λ

λ-expresie
Sintaxa pentru calcul λ

λ

· O λ-expresie poate fi:
- x
- $\lambda x.E$ — E λ-expresie
- $(F\ \ A)$ — F, A λ-expresii

Exemple:
- $\lambda x.x$
- $\lambda x.\lambda y.(x\ \ y)$
- $(\lambda x.x\ \ \lambda x.x)$

β Redex
Ce reducem?

λ

· Sursa pentru β-reducere și pasul de reducere.

· Este o funcție care se poate aplica.

$$(\lambda x.\underbrace{\quad\quad}_{corp}\ \underbrace{\quad\quad\quad\quad}_{parametru\ actual})$$

· x: numele parametrului formal.

β-reducere
Cum reducem?

λ

· substituție textuală

$$(\lambda x.\underbrace{\quad\quad}_{corp}\ \underbrace{\quad\quad\quad\quad}_{parametru\ actual}) \to_\beta \underbrace{\quad\quad}_{corp}{}_{[parametru\ actual/x]}$$

apariițiile libere ale lui x din corp sunt substituite textual cu parametrul actual

Substituție
Ce substituim?

λ

↓ apariție liberă în corp

$$\lambda x.\underbrace{\quad x \quad}_{corp}$$

$$\lambda x.\underbrace{\quad \lambda x.\overbrace{\quad x \quad}^{corp_2}}_{corp_1}$$

↑ leagă apariția în $corp_1$
(apariția este liberă în $corp_2$)

· O apariție x este legată de cea mai interioară definiție λx, care conține apariția în corpul său. Dacă λx care îl leagă este inclus în expresia E, apariția este legată în E, altfel este liberă în E.

· x are o apariție liberă în $E \Rightarrow x$ variabilă liberă în E (altfel legată).

· $\nexists$ variabile libere în $E \Rightarrow E$ închisă.

Condiții β-reducere pentru $(\lambda x.E\ A)$ λ
Când este corect să efectuăm substituția?

· Variabilele **libere** din A nu devin **legate** în $E_{[A/x]}$

· Mai precis, numele variabilelor libere din A nu sunt nume de variabile care sunt legate în contextele din E în care apare x.

· Exemplu: $(\lambda x.\lambda y.(y\ x)\ \lambda z.y) \rightarrow$ incorect să efectuăm β-reducere.

α-conversie în $(\lambda x.E\ A)$ λ
Cum rezolvăm problema anterioară?

· când? → când variabilele din A devin legate în $E_{[A/x]}$

· ce redenumim? → parametri formali ai tuturor funcțiilor din E care conțin apariții libere ale lui x în corp și au ca parametru formal numele unei variabile libere din A (redenumirea parametrilor formali implică folosirea noului nume în toate aparițiile libere ale parametrilor formali în corpurile funcțiilor respective).

· la ce redenumim? → la un nume care nu este nume de variabilă liberă în A sau în propriul corp, și care nu devine legat în corp.

Pas de reducere λ
Cum efectuăm o reducere corectă?

[α-conversie] + β-reducere fără coliziuni

1. avem β-redex
2. dacă este cazul, efectuăm α-conversie
3. efectuăm β-reducere

Secvență de reducere λ
Cum facem o reducere completă?

Secvență de reducere = $\rightarrow^*$

· Dacă expresia este reductibilă (are o secvență de reducere care se termină), reducerea în ordine stânga-dreapta se va termina cu valoarea expresiei.

Cursul 4

Programare funcţională în Racket

Cuprins λ

Introducere

Racket vs. Scheme
Cum se numește limbajul despre care discutăm?

- Racket este dialect de Lisp/Scheme (așa cum Scheme este dialect de Lisp);
- la nivelul studiat, Racket este identic cu Scheme;
- Racket este derivat din Scheme, oferind instrumente mai puternice;
- Racket (fost PLT Scheme) este compilat de mediul DrRacket (fost DrScheme);

[http://en.wikipedia.org/wiki/Racket_(programming_language)]
[http://racket-lang.org/new-name.html]

Racket vs. λ_0
Construcția expresiilor / sintaxă

	λ	Racket
Variabilă/nume	x	`x`
Funcție	$\lambda x.corp$	`(lambda (x) corp)`
uncurry	$\lambda x\ y.corp$	`(lambda (x y) corp)`
Aplicare	$(F\ A)$	`(f a)`
uncurry	$(F\ A1\ A2)$	`(f a1 a2)`
Legare top-level	-	`(define nume expr)`
Program	λ-expresie închisă	colecție de legări top-level (`define`)
Valori	λ-expresii / TDA	valori de diverse tipuri (numere, liste, etc.)

Racket vs. λ_0
Mai precis

- similar cu λ_0, folosește S-expresii (bază Lisp);
- tipat – dinamic/latent
  - variabilele nu au tip;
  - valorile au tip (`3`, `#f`);
  - verificarea se face la execuție, în momentul aplicării unei funcții;
- evaluare aplicativă;
- permite recursivitate textuală;
- avem legări top-level.

Discuție despre tipare

Modalități 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
Exemplu

Exemplul 14.1 (Tipare dinamică).

Javascript:

`var x = 5;`
`if(condition) x = "here";`
`print(x);` → ce tip are `x` aici?

Exemplul 14.2 (Tipare statică).

Java:

`int x = 5;`
`if(condition) x = "here";` → Eroare la compilare: `x` este `int`.
`print(x);`

Tipare statică vs. dinamică
Caracteristici

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
- Permite metaprogramare (v. `eval`)
- Python, Scheme/Racket, Prolog, JavaScript, PHP

Tipare tare vs. slabă
Exemple

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

Exemplul 14.3 (Tipare tare).

`1 + "23"` → Eroare (Haskell)

Exemplul 14.4 (Tipare slabă).

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

Tiparea în Racket

- este dinamică

Exemplul 14.5.

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

- este tare

Exemplul 14.6.

```
(+ "1" 2) → Eroare
```

- permite liste cu elemente de tipuri diferite.

Legarea variabilelor

Variabile
Proprietăți

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

Exemplul 15.1 (În C).

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


Legarea variabilelor

Definiția 15.2 (Legarea variabilelor).
Modalitatea de asociere a apariţiei unei variabile cu definiţia acesteia (deci cu valoarea).

Definiția 15.3 (Domeniu de vizibilitate – *scope*).
Mulţimea punctelor din program unde o definiţie (legare) este vizibilă. Este determinat de modalitatea de legare a variabilelor.

- Modalităţi de legare:
  - statică
  - dinamică

Legarea statică a variabilelor

Definiția 15.4 (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 15.5 (Calcul $\lambda \rightarrow$ legare statică).
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 15.6 (Legare dinamică).
Valorile variabilelor depind de momentul în care o expresie este evaluată.

- Domeniu de vizibilitate (al unei legări) determinat la execuţie.

Exemplul 15.7.

```c
int f(boolean flag) {
int x = 2;
if(flag) x = 3;
return x;   → fie legarea =3 fie =2, depinde de flag
}
```


Legarea variabilelor în Racket

- Variabile declarate (! și definite) în construcții care leagă → legate static
  - `lambda`
  - `let`
  - `let*`
  - `letrec`
- Variabile *top-level* → legate dinamic → posibilitatea definirii multiple
  - `define`

Construcţia `lambda`
Definiţie & Exemplu

- Leagă static parametrii formali ai unei funcţii
- Sintaxă:

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

- Domeniul de vizibilitate a parametrului `pk`: mulţimea punctelor din `expr` (care este corpul funcţiei), puncte în care apariţia lui `pk` este libere. [v. Exemplul 15.5]

Exemplul 15.8.
$\lambda x.(x\ \lambda y.y)\ \equiv$

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


Construcţia `lambda`
Semantică

- Aplicaţie:

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

1. Evaluare aplicativă: se evaluează argumentele `ak`, în ordine aleatoare (nu se garantează o anumită ordine).
2. Se evaluează corpul funcţiei, `expr`, ţinând cont de legările `pk` ← *valoare*(`ak`).
3. Valoarea aplicaţiei este valoarea lui `expr`.

Construcţia `let`
Definiţie, Exemplu, Semantică

- Leagă static variabile locale
- Sintaxă:

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

- Domeniul de vizibilitate a variabilei `vk` (cu valoarea `ek`): mulţimea punctelor din `expr` (corp let), în care apariţiile lui `vk` sunt libere.

Exemplul 15.9.

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

- Atenţie! Construcţia `(let ((v1 e1) ...(vn en)) expr)` – echivalentă 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 ulterioare) şi
  - corp – `expr`

  în care apariţiile lui `vk` sunt libere

Exemplul 15.10.

```
(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 15.11.

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


Construcţia `define`
Definiţie & Exemplu

- Leagă dinamic variabile *top-level*.

Exemplul 15.12.

```
(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: corуperea transparenţei referenţiale

Construcţia `define`
Exemplu mai obscur

Exemplul 15.13.

```
(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
Exerciţiu pentru domenii de vizibilitate

Exemplul 15.14.

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

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

(g 2)
```

Output: `1`

Evaluare, contexte, închideri

Evaluarea în Racket

- Evaluare aplicativă: evaluarea parametrilor înaintea aplicării funcţiei asupra acestora (în ordine aleatoare).
- Funcţii stricte (i.e.cu evaluare aplicativă)
  - Excepţii: `if`, `cond`, `and`, `or`, `quote`.

Contexte computaţionale
Definiţie

Definiția 16.1 (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 16.2.
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 16.3.
$((\lambda f.\lambda g.\lambda x.(f\ (g\ x))\ g_1)\ f_1) \rightarrow (\lambda g.\lambda x.(f_1\ (g\ x))\ g_1)$
$\rightarrow \lambda x.(f_1\ (g_1\ x))$

- 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 16.4 (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 16.5.
$< \lambda x.z;\ \{z \leftarrow 2\} >$

Închideri funcţionale
Exemplu

Exemplul 16.6.

```
(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

- $comp\text{-}inc \equiv < \lambda g.\lambda x.(f\ (g\ x));\ \{f \leftarrow \lambda x.(+\ x\ 1)\} >$
- $comp\text{-}inc\text{-}double \equiv$
  $< \lambda x.(f\ (g\ x));\ \{f \leftarrow \lambda x.(+\ x\ 1),\ g \leftarrow \lambda x.(*\ x\ 2)\} >$
- 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 16.7.

```
(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 17.1.

```
(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.

Sfârșitul cursului 4
Ce am învățat

- Tipare dinamică vs. statică, tare vs. slabă, legare dinamică vs statică;
- Racket: tipare dinamică, tare; domeniu al variabilelor;
- construcţii speciale în Racket: lambda, let, let*, letrec, define; controlul evaluării
- efecte laterale

Cursul 5

Evaluare leneşă în Racket

Cuprins

Întârzierea evaluării

Motivaţie
De ce? → Luăm un exemplu

Exemplul 18.1.
Să se implementeze funcţia nestrictă *prod*, astfel încât al doilea parametru să fie evaluat doar dacă primul este *true*:

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

Dar, evaluarea parametrului *y* al funcției să se facă numai o singură dată.

· Problema de rezolvat: evaluarea la cerere.

Varianta 1
Încercare → implementare directă

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

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

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

Output: `y 0 | y 30`

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

Varianta 2
Încercare → `quote` & `eval`

```
(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

```
(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`

```
(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
- `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
Proprietăți

- 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.

Observaţii

- Dependenţă între mecanismul de întârziere şi cel de evaluare ulterioară a expresiilor – vezi 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?

→ Introducem ideea de abstracție procedurală – o interfață uniformă pentru o anumită funcționalitate, indiferent de implementarea reală.

Abstracţii procedurale
Cerinţe

- Izolarea implementării de utilizare → modularitate;
- Reutilizabilitate;
- exemplu: 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
Motivaţie

- 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
Soluţie

- 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.
- Atenţie! `pack` şi `unpack` sunt funcţii implementate de noi.

Abstracţii de date
Implementare cu promisiuni

Exemplul 18.2 (Continuare a exemplului 18.1).

```
(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))) )))
```

· utilizarea nu depinde de implementare.

Abstracţii de date
Implementare cu închideri

Exemplul 18.3 (Continuare a exemplului 18.1).

```
(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))) )))
```

· utilizarea nu depinde de implementare.

Fluxuri

Motivaţie

Exemplul 19.1.
Să se determine suma numerelor pare din intervalul $[a, b]$.

```
(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;
  - ne-elegantă → trebuie să implementăm generarea numerelor.
- 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

Fluxuri
Caracteristici

- Secvențe construite parțial, extinse la cerere, ce creează iluzia completitudinii structurii;
- Îmbinarea eleganței manipulării listelor cu eficiența calculului incremental;
- Bariera de abstractizare:
  - componentele listelor evaluate la construcție (cons)
  - componentele fluxurilor evaluate la selecție (cdr)
- Construcție și utilizare:
  - separate la nivel conceptual → modularitate;
  - întrepătrunse la nivel de proces (utilizare înseamnă construcție).

Fluxuri
Operatori: construcție și selecție

- 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-nil '())

(define stream-null? null?)
```


Fluxuri
Operatori: take și drop

- selecție / 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 și filter

- operatori de aplicare și 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 și 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

- Definiție cu închideri:

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

- Definiție cu fluxuri:

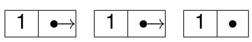
```
(define ones (stream-cons 1 ones))
(stream-take 5 ones) ; (1 1 1 1 1)
```

- Definiție cu promisiuni:

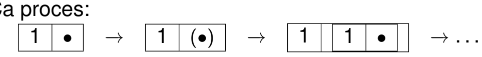
```
(define ones (delay (cons 1 ones)))
```


Fluxuri – Exemple
Flux de numere 1 – discuție

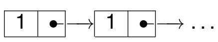
- Extinderea se realizează în spațiu constant:



- 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))
```

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

· Atenție:

- Închideri: multiple parcurgeri ale fluxului determină reevaluarea porțiunilor deja explorate.
- Promisiuni: parcurgerea fluxului determină evaluarea dincolo de porțiunile deja explorate.

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

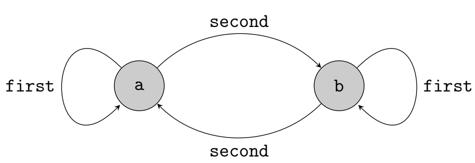
- Ciurul lui Eratostene.
- Pornim de la fluxul numerelor naturale, începând cu 2.
- Elementul curent din fluxul inițial aparține fluxului numerelor prime.
- Restul fluxului generat se obține
  - eliminând multiplii elementului curent din fluxul inițial;
  - continuând procesul de filtrare, cu elementul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

Grafuri ciclice
Concept



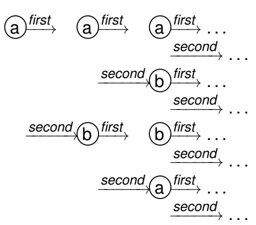
- Fiecare nod conține:
  - cheia: `key`
  - legăturile către două noduri: `first`, `second`

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 leneșă în spațiul stărilor

Spaţiul stărilor unei probleme

Definiția 21.1 (Spaţiul stărilor unei probleme).
Mulţimea configuraţiilor valide din universul problemei.

Exemplul 21.2.
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
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
Leneşă (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-nil))
)))
```


Căutare în lăţime
Leneşă (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.
- Aplicații:
  - 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

Haskell

[https://en.wikipedia.org/wiki/Haskell_(programming_language)]

- din 1990;
- GHC – Glasgow Haskell Compiler (The Glorious Glasgow Haskell Compilation System)
  - dialect Haskell standard *de facto*;
  - compilează în/folosind C;
- Haskell Platform
  - include manager de pachete, debugger, generator de parsere, unelte de documentație, legare la cod C, etc.
- nume luat după logicianul Haskell Curry;
- aplicații: Pugs, Darcs, Linspire, Xmonad, Cryptol, seL4, Pandoc, web frameworks.

Paralelă între limbaje

Criteriu	Racket	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

Sintaxă

Funcţii

- toate funcțiile sunt *Curry*;
- aplicabile asupra oricâtor parametri la un moment dat.

Exemplul 23.1.

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 vs operatori

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

Exemplul 23.2.

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 23.3.

```
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 23.4.

```
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
Pentru toate valorile (inclusiv funcții)

- Tipuri ca mulţimi de valori:
  - `Bool = {True, False}`
  - `Natural = {0, 1, 2, ...}`
  - `Char = {'a', 'b', 'c', ...}`
- Rolul tipurilor (vezi curs 2-3);
- 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
Pentru valori care nu sunt funcții

Exemplul 24.1.

```
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
⇒ tipuri noi pentru valori sau funcții

- Funcţii de tip, ce îmbogăţesc tipurile din limbaj.

Exemplul 24.2 (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 24.3.

```
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 24.4 (Polimorfism parametric).

Manifestarea aceluiaşi comportament pentru parametri de tipuri diferite. Exemplu: `idd`.

Definiția 24.5 (Polimorfism ad-hoc).

Manifestarea unor comportamente diferite pentru parametri de tipuri diferite. Exemplu: `==`.

Constructorul de tip `Natural`
Exemplu de definire TDA 1

Exemplul 24.6.

```
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 2

Exemplul 24.7.

```
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:

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


Uniformitatea reprezentării tipurilor

Exemplul 24.8 (Tipurile de bază).

```
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 24.9 (Progres).
O expresie bine-tipată (căreia i se poate asocia un tip):

- este o valoare (nu este o aplicare de funcție) *sau*
- (este aplicarea unei funcții și) poate fi redusă (vezi β-redex).

Definiția 24.10 (Conservare).
Evaluarea unei expresii bine-tipate produce o expresie bine-tipată – de obicei, cu acelaşi tip.

- dacă *sinteza de tip* pentru expresia *E* dă tipul *t*, atunci după reducere, valoarea expresiei *E* va fi de tipul *t*.

Sinteză de tip

Sinteza de tip
Definiție

Definiția 25.1 (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.

Exemple de sinteză de tip
Câteva reguli simplificate de sinteză de tip

- Formă: $\dfrac{\texttt{premisa-1 ...} \qquad \texttt{premisa-m}}{\texttt{concluzie-1 ... concluzie-n}}\ (\texttt{nume})$

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

Exemple de sinteză de tip
Transformare de funcție

Exemplul 25.2.

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

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

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

$\Rightarrow$ `b = Int`

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

$\Rightarrow$ `a = c -> d, c = Int, d = Int`

$\Rightarrow$ `f :: (Int -> Int) -> Int`

Exemple de sinteză de tip
Combinator de punct fix

Exemplul 25.3.

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

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

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

$\Rightarrow$ `a = c -> d, b = d`

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

$\Rightarrow$ `a -> b = e -> g, a = e, b = g, c = g`

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

Exemple de sinteză de tip
O funcție ne-tipabilă

Exemplul 25.4.

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

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

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

Ecuaţia `c -> d = c` nu are soluţie ($\nexists$ tipuri recursive)
$\Rightarrow$ funcţia nu poate fi tipată.

Unificare
Definiție

- la baza sintezei de tip: unificarea $\rightarrow$ legarea variabilelor în timpul procesului de sinteză, în scopul unificării diverselor formule de tip elaborate.

Definiția 25.5 (Unificare).

Procesul de identificare a valorilor variabilelor din 2 sau mai multe formule, astfel încât substituirea variabilelor prin valorile asociate să conducă la coincidenţa formulelor.

Definiția 25.6 (Substituţie).

O substituție este o mulţime de legări variabilă - valoare.

Unificare
Exemplu

Exemplul 25.7.

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

Definiția 25.8 (*Most general unifier* – MGU).

Cea mai generală substituţie 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` şi `E` nu conţine `a` (*occurrence check*).
    Exemplu: `a` unifică cu `b -> c` dar nu cu `a -> b`.
- 2 constante de tip unifică doar dacă sunt egale;
- 2 aplicaţii de tip unifică doar dacă implică acelaşi constructor de tip şi argumente ce unifică recursiv.

Tip principal
Exemplu și definiție

Exemplul 25.9.

- Tipurile: `t1 = (a, [b])` , `t2 = (Int, c)`
  MGU: `S = {a ← Int, c ← [b]}`
  Tipuri mai particulare (instanţe): `(Integer, [Integer])`, `(Integer, [Char])`, etc
- Funcția: `\x -> x`
  Tipuri corecte: `Int -> Int` , `Bool -> Bool` , `a -> a`

Definiția 25.10 (Tip principal al unei expresii).

Cel mai general tip care descrie complet natura expresiei.
Se obţine prin utilizarea MGU.

Evaluare

Evaluare

- Evaluare leneşă: parametri evaluaţi la cerere, cel mult o dată, eventual parţial, în cazul obiectelor structurate
- Transferul parametrilor: *call by need*
- Funcţii nestricte!

Exemplul 26.1.

```
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 26.2.

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

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

frontInterval m n
    | notNil xs  =   frontSum 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 26.2 (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 26.3.

```
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 Racket · 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 în Haskell

Programare orientată spre date

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

Exemplul 27.1 (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 27.2 (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 27.3 (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 27.4 (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 27.5 (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)
  ]
```


Bibliografie

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

Cursul 8

Clase în Haskell

Cuprins

Motivație

Motivație
Exemplu

Exemplul 28.1.
Să se definească operația `show`, capabilă să producă reprezentarea oricărui obiect ca șir de caractere. Comportamentul este specific fiecărui tip (polimorfism ad-hoc).

```
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 ⇒ avem nevoie de `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: $\underbrace{\texttt{Show a =>}}_{\textit{context}}$
- 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 (dată de contextul `Show`).

Clase Haskell

Clase şi instanţe
Definiții

Definiția 29.1 (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 29.2 (Instanţă a unei clase).

Tip care supraîncarcă operaţiile clasei. Exemplu: tipul `Bool` în raport cu clasa `Show`.

- *clasa* definește funcțiile suportate;
- *instanța* definește implementarea funcțiilor.

Clase predefinite
`Show, Eq`

```
1  class Show a where
2      show :: a -> String
3
4  class Eq a where
5      (==), (/=) :: a -> a -> Bool
6      x /= y      =  not (x == y)
7      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 utilizabile şi la definirea unei clase.
- Moştenirea claselor, cu preluarea operaţiilor din clasa moştenită.
- Necesitatea aderării la clasa `Eq` în momentul instanţierii clasei `Ord`.
- Suficienţa supradefinirii lui `(<=)` la instanţiere.

Utilizarea claselor predefinite
Pentru tipuri de date noi

- Anumite tipuri de date (definite folosind `Data` pot beneficia de implementarea automată a anumitor funcționalități, oferite de tipurile predefinite în `Prelude`:
  - `Eq, Read, Show, Ord, Enum, Ix, Bounded`.

```
data Alarm = Soft | Loud | Deafening
    deriving (Eq, Ord, Show)
```

- variabilele de tipul `Alarm` pot fi comparate, testate la egalitate, și afișate.

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); interfeţele sunt mulțimi de tipuri;
- Implementarea interfeţelor de către clase.

Aplicaţii ale claselor

`invert`
Problema

Exemplul 30.1 (`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 înseler.

contents
Problema

Exemplul 30.2 (`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
```

Testăm pentru `contents [1,2,3]`:

- 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
```

Testăm pentru `contents [1,2,3]`:

- 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
Observații

- 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

Concluzie – Paradigma Funcțională

Cuprins

Caracteristici ale paradigmelor de programare

Paradigma de programare λ

Impact în scrierea unui program

- Paradigma de programare – un mod de a:
  - aborda rezolvarea unei probleme printr-un program;
  - structura un program;
  - reprezenta datele dintr-un program;
  - implementa diversele aspecte dintr-un program (cum prelucrăm datele);
- Un limbaj poate include caracteristici dintr-una sau mai multe paradigme;
  - în general există o paradigmă dominantă;
- Atenție! Paradigma nu are legătură cu sintaxa limbajului!

Paradigma de programare λ

Legătura cu mașina de calcul

- paradigmele sunt legate teoretic de o mașină de calcul (mai mult sau mai puțin teoretică) în care prelucrările caracteristice paradigmei se fac la nivelul mașinii;
- dar putem executa orice program, scris în orice paradigmă, pe orice mașină.

Paradigma de programare λ

Ce o definește

· În principal, paradigma este definită de

- elementele principale din sintaxa limbajului – e.g. existența și semnificația variabilelor, semnificația operatorilor asupra datelor, modul de construire aprogramului;
- modul de construire a tipurilor variabilelor;
- modul de definire și statutul elementelor principale de prelucrare a datelor din program (e.g. obiecte, funcții, predicate);
- legarea variabilelor, efecte laterale, transparență referențială, modul de transfer al parametrilor pentru elementele de prelucrare a datelor.

Variabile λ

Nume date unor valori

- în majoritatea limbajelor există variabile, ca NUME date unor valori – rezultatul anumitor procesări (calcule, inferențe, substituții);
- variabilele pot fi o referință pentru un spațiu de memorie sau pentru un rezultat abstract;
- elementele de procesare a datelor pot sau nu să fie valori de prim rang (să poată fi asociate cu variabile).

Funcţii ca valori de prim rang λ

Definiție

Definiția 31.1 (Valoare de prim rang).

O valoare care poate fi:
- creată dinamic
- stocată într-o variabilă
- trimisă ca parametru unei funcţii
- întoarsă dintr-o funcţie

Funcţii ca valori de prim rang λ
Exemplu

Exemplul 31.2.

Să se scrie funcţia `compose`, ce primeşte ca parametri alte 2 funcţii, `f` şi `g`, şi întoarce funcţia obţinută prin compunerea lor, `f ∘ g`.

Funcţii ca valori de prim rang: Compose λ
C

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

- în C, funcţiile nu sunt valori de prim rang;
- pot scrie o funcție care compune două funcții, dar nu pot crea o referință (pointer) la rezultatul dorit, care să fie folosit ca o funcție obișnuită de un singur argument.

Funcţii ca valori de prim rang: λ
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 – pot crea rezultatul dar este foarte complicat, și rezultatul nu este o funcție obișnuită, ci un obiect.

Funcţii ca valori de prim rang: Compose λ
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.
- mai mult, ele pot fi aplicate parțial, și putem avea funcționale – funcții care iau alte funcții ca parametru.

Legarea variabilelor

Legarea variabilelor λ
Impactul asupra programului

· două posibilități esențiale:

- un nume este întotdeauna legat la aceeași valoare / la același calcul ⇒ numele stă pentru un calcul;
  - legare statică.
- un nume poate fi legat la mai multe valori pe parcursul execuției ⇒ numele stă pentru un spațiu de stocare – fiecare element de stocare fiind identificat printr-un nume;
  - legare dinamică.

Efecte laterale (*side effects*) λ
Definiție

Exemplul 32.1.

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 32.2 (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*) λ
Consecinţe

Exemplul 32.3.

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

Efecte laterale (*side effects*)
Consecințe asupra programării leneșe

- În prezența efectelor laterale, programarea leneșă devine foarte dificilă;
- Efectele laterale pot fi gestionate corect numai atunci când secvența evaluării este garantată → garanție inexistentă în programarea leneșă.
  - nu știm când anume va fi nevoie de valoarea unei expresii.

Transparență referențială
Definiție

Exemplul 32.4.

"care este numărul cu 2 mai mare decât 5?"

1.
   - afișează "întrebarea este" x
   - y este succesorul lui x
   - z este succesorul lui y
   - răspunsul este z → orice variabilă poate fi înlocuită cu semnificația ei
2.
   - afișează "întrebarea este" x
   - incrementează x cu 2
   - răspunsul este x → x își schimbă semnificația pe parcursul evaluării

Definiția 32.5 (Transparență referențială).

Confundarea unui obiect cu referința la acesta → cazul 1.

Transparență referențială
Pentru expresii

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

Exemplul 32.6.

- `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 funcție de statutul lui `x` (globală, statică etc.)

- Absentă în prezența efectelor laterale!

Transparență referențială
Pentru funcții

- Funcție transparentă referențial: rezultatul întors depinde exclusiv de parametri

Exemplul 32.7.

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

```
int g = 0;

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

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

Transparență referențială
Avantaje

- Lizibilitatea codului;
- Demonstrarea formală a corectitudinii programului – mai ușoară datorită lipsei stării;
- Optimizare prin reordonarea instrucțiunilor de către compilator și prin caching;
- Paralelizare masivă, prin eliminarea modificărilor concurente.

Modul de evaluare

Evaluare
Mod de evaluare și execuția programelor

- modul de evaluare al expresiilor dictează modul în care este executat programul;
- este legat de funcționarea mașinii teoretice corespunzătoare paradigmei;
- ne interesează în special ordinea în care expresiile se evaluează;
- în final, întregul program se evaluează la o valoare;
- important în modul de evaluare este modul de evaluare / transfer a parametrilor.

Transferul parametrilor

- Evaluare aplicativă – parametrii sunt evaluați înainte de evaluarea corpului funcției.
  - *Call by value*
  - *Call by sharing*
  - *Call by reference*
- Evaluare normală – funcția este evaluată fără ca parametrii să fie evaluați înainte.
  - *Call by name*
  - *Call by need*

Call by value λ
În evaluarea aplicativă

Exemplul 33.1.

```
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 Java

Call by sharing λ
În evaluarea aplicativă

- 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 λ
În evaluarea aplicativă

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

Call by name λ
În evaluarea normală

- Argumente neevaluate în momentul aplicării funcţiei → substituţie directă (textuală) în corpul funcţiei;
- Evaluare parametrilor la cerere, de fiecare dată când este nevoie de valoarea acestora;
- în calculul λ.

Call by need λ
În evaluarea normală

- 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 (datorită transparenței referențiale);
- în Haskell.

Sfârșitul cursului 9 λ
Ce am învățat

- caracteristicile unei paradigme;
- variabile, funcții ca valori de prim rang;
- legare, efecte laterale, transparență referențială;
- evaluare și moduri de transfer al parametrilor.

Cursul 10

Prolog și logica cu predicate de ordinul I

Cuprins λ

Introducere în Prolog

Prolog
Limbaj de programare logică

- introdus în anii 1970 ;
- programul → mulțime de propoziții logice în LPOI;
- mediul de execuție = demonstrator de teoreme care spune:
  - dacă un fapt este adevărat sau fals;
  - în ce condiții este un fapt adevărat.
- Resursă Prolog pe Wikibooks:
  [https://en.wikibooks.org/wiki/Prolog]

Prolog
Caracteristici

- fundamentare teoretică a procesului de raționament;
- motor de raționament ca unic mod de execuție;
  → modalități limitate de control al execuției.
- căutare automată a valorilor pentru variabilele nelegate (dacă este necesar);
- posibilitatea demonstrațiilor și deducțiilor simbolice.

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 propozițională

Logica propozițională
Context și elemente principale

- Cadru pentru:
  - descrierea proprietăților obiectelor, prin intermediul unui limbaj, cu o semantică asociată;
  - deducerea de noi proprietăți, pe baza celor existente.
- Expresia din limbaj: propoziția, corespunzătoare unei afirmații, ce poate fi adevărată sau falsă.
- Exemplu: “Afară este frumos.”
- Accepții asupra unei propoziții:
  - secvența de simboluri utilizate sau
  - înțelesul propriu-zis al acesteia, într-o interpretare.

Logica propozițională
Sintaxă

- 2 categorii de propoziții
  - simple → fapte atomice: “Afară este frumos.”
  - compuse → relații între propoziții mai simple: “Telefonul sună și câinele latră.”
- Propoziții simple: $p, q, r, \dots$
- Negații: $\neg\alpha$
- Conjuncții: $(\alpha \wedge \beta)$
- Disjuncții: $(\alpha \vee \beta)$
- Implicații: $(\alpha \Rightarrow \beta)$
- Echivalențe: $(\alpha \Leftrightarrow \beta)$

Logica propoziţională
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 propoziţiilor → utilizarea conceptului de interpretare.

Semantică
Interpretare

Definiţia 35.1 (Interpretare).

Mulţime de asocieri între fiecare propoziţie simplă din limbaj şi o valoare de adevăr.

Exemplul 35.2.

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

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

- cum ştiu dacă p este adevărat sau fals? Pot şti dacă ştiu interpretarea – p este doar un *nume* pe care îl dau unei propoziţii concrete.

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

Evaluare
Cum determinăm valoarea de adevăr

Definiţia 35.3 (Evaluare).

Determinarea valorii de adevăr a unei propoziţii, sub o interpretare, prin aplicarea regulilor semantice anterioare.

Exemplul 35.4.

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

Evaluarea valorii de adevăr

Valoarea de adevăr în afara interpretării
Satisfiabilitate

Definiţia 36.1 (Satisfiabilitate).

Proprietatea unei propoziţii care este adevărată sub cel puţin o interpretare. Acea interpretare satisface propoziţia.

Exemplul 36.2 (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

Valoarea de adevăr în afara interpretării
Validitate

Definiţia 36.3 (Validitate).

Proprietatea unei propoziţii care este adevărată în toate interpretările. Propoziţia se mai numeşte tautologie.

Exemplul 36.4 (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.

Valoarea de adevăr în afara interpretării
Nesatisfiabilitate

Definiția 36.5 (Nesatisfiabilitate).

Proprietatea unei propoziții care este falsă în toate interpretările. Propoziția se mai numeşte contradicție.

Exemplul 36.6 (Nesatisfiabilitate).

Propoziția $p \Leftrightarrow \neg p$ este falsă, indiferent de valoarea de adevăr a lui p, deci este nesatisfiabilă.

- Verificabilă prin metoda tabelei de adevăr.

Derivabilitate
Definiție

Definiția 36.7 (Derivabilitate logică).

Proprietatea unei propoziții de a reprezenta consecința logică a unei mulțimi de alte propoziții, numite premise. Mulțimea de propoziții Δ derivă propoziția ϕ, fapt notat prin $\Delta \models \phi$, dacă şi numai dacă orice interpretare care satisface toate propozițiile din Δ satisface şi ϕ.

Exemplul 36.8.

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

Derivabilitate
Verificare

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

Exemplul 36.9.

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.

Derivabilitate
Formulări echivalente

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

sau

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

sau

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

Inferența
Motivație

- Creşterea exponențială a numărului de interpretări în raport cu numărul de propoziții simple.
- De aici, diminuarea valorii practice a metodelor semantice, precum cea a tabelei de adevăr.
  - Derivabilitate logică → proprietate a propozițiilor, verificabilă prin tabela de adevăr.
- Alternativ, metode sintactice, care manipulează doar reprezentarea simbolică.
  - Inferență → Derivare mecanică → demers de calcul, în scopul verificării derivabilității logice.
  - folosind metodele de inferență, putem construi o maşină de calcul.

Inferența
Definiție

Definiția 36.10 (Inferență).

Derivarea mecanică a concluziilor unui set de premise.

Definiția 36.11 (Regulă de inferență).

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

Inferența
Reguli de inferență

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

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

- exemplu: *Modus Tollens*:

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

Inferența
Proprietăți ale regulilor

Definiția 36.12 (Consistență (*soundness*)).

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

Definiția 36.13 (Completitudine (*completeness*)).

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

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

Rezoluția

Rezoluție
O regulă de inferență completă și consistentă

- Regulă de inferență foarte puternică.
- Baza unui demonstrator de teoreme consistent și complet.
- Spațiul de căutare mai mic decât în alte sisteme.
- Se bazează pe lucrul cu propoziții în forma clauzală:
  - propoziție = mulțime de clauze (semnificație conjunctivă)
  - clauză = mulțime de literali (semnificație disjunctivă)
  - literal = atom sau atom negat
  - atom = propoziție simplă

Forma clauzală
Definiții

Definiția 37.1 (Literal).
Propoziție simplă sau negația ei. E.g. p și $\neg p$.

Definiția 37.2 (Expresie clauzală).
Literal sau disjuncție de literali. E.g. $p \vee \neg q \vee r$.

Definiția 37.3 (Clauză).
Mulțime de literali dintr-o expresie clauzală. E.g. $\{p, \neg q, r\}$.

Definiția 37.4 (Forma clauzală – CNF).
Reprezentarea unei propoziții sub forma unei mulțimi de clauze, implicit legate prin conjuncții.

Forma clauzală
Exemplu

Exemplul 37.5 (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 37.6.

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 37.7.

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 → pasul de rezoluție

- Ideea:

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

- "Anularea" lui p
- p falsă $\rightarrow$ $\neg p$ adevărată $\rightarrow$ r adevărată
- p adevărată $\rightarrow$ q adevărată
- $p \vee \neg p \Rightarrow$ Cel puțin una dintre q și r adevărată ($q \vee r$)
- Forma generală a pasului de rezoluție:

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

Rezoluție
Cazuri speciale

- Clauza vidă → indicator de contradicție între premise

$$\frac{\begin{array}{c}\{\neg p\} \\ \{p\}\end{array}}{\{\} = \emptyset}$$

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

$$\frac{\begin{array}{c}\{p, q\} \\ \{\neg p, \neg q\}\end{array}}{\begin{array}{c}\{p, \neg p\} \text{ sau} \\ \{q, \neg q\}\end{array}}$$

Rezoluție
Alte reguli de inferență → cazuri particulare ale rezoluției

- *Modus Ponens*:

$$\frac{p \Rightarrow q \quad p}{q} \sim \frac{\{\neg p, q\} \quad \{p\}}{\{q\}}$$

- *Modus Tollens*

$$\frac{p \Rightarrow q \quad \neg q}{\neg p} \sim \frac{\{\neg p, q\} \quad \{\neg q\}}{\{\neg p\}}$$

- Tranzitivitatea implicației:

$$\frac{p \Rightarrow q \quad q \Rightarrow r}{p \Rightarrow r} \sim \frac{\{\neg p, q\} \quad \{\neg q, r\}}{\{\neg p, r\}}$$

Rezoluție
Demonstrare

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

Rezoluție
Demonstrare – algoritm

0. Am premisele $\phi_1, \ldots, \phi_n$ și concluzia dorită ϕ
1. Transform $\phi_1, \ldots, \phi_n$ și $\neg\phi$ în FNC
   ⇒ mulțime de clauze $\phi_1, \ldots, \phi_n, \neg\phi$
2. Aleg două clauze și aplic pasul de rezoluție
3. **Dacă** rezultatul pasului de rezoluție este clauza vidă ($\emptyset$)
4. **atunci** am terminat demonstrația cu succes
5. **altfel** merg la pasul 2

Rezoluție
Exemplu

Exemplul 37.8.

Demonstrăm că $\{p \Rightarrow q, q \Rightarrow r\} \vdash p \Rightarrow r$, i.e. mulțimea $\{p \Rightarrow q, q \Rightarrow r, \neg(p \Rightarrow r)\}$ conține o contradicție.

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

Rezoluție
Consistență și completitudine

Teorema 37.9 (Rezoluției).

Rezoluția propozițională este consistentă și completă, i.e. $\Delta \models \phi \Leftrightarrow \Delta \vdash_{rez} \phi$.

- Terminare garantată a procedurii de aplicare a rezoluției: număr finit de clauze → număr finit de concluzii.

Sfârșitul cursului 10
Ce am învățat

- Bazele logicii propoziționale, Sintaxă și Semantică
- Inferență, Rezoluție, Forme normale

Cursul 11

Logica cu predicate de ordinul I

Cuprins

Introducere

Logica cu predicate de ordinul I

First Order Predicate Logic (FOL sau FOPL) – Context

- Extensie a logicii propoziționale, cu explicitarea:
  - obiectelor din universul problemei;
  - relațiilor dintre acestea.
- Logica propozițională:
  - p: "Andrei este prieten cu Bogdan."
  - q: "Bogdan este prieten cu Andrei."
  - $p \Leftrightarrow q$
  - → Opacitate în raport cu obiectele și relațiile referite.
- FOL:
  - Generalizare: $prieten(x,y)$: "x este prieten cu y."
  - $\forall x.\forall y.(prieten(x,y) \Leftrightarrow prieten(y,x))$
  - → Aplicare pe cazuri particulare.
  - → Transparență în raport cu obiectele și relațiile referite.

Sintaxă

Sintaxă

Simboluri utilizate

- Constante: obiecte particulare din universul discursului: c, d, $andrei$, $bogdan$, ...
- Variabile: obiecte generice: x, y, ...
- Simboluri funcționale: $succesor$, $+$, abs ...
- Simboluri relaționale (predicate): relații n-are peste obiectele din universul discursului: $prieten = \{(andrei, bogdan), (bogdan, andrei), \ldots\}$, $impar = \{1, 3, \ldots\}$, ...
- Conectori logici: $\neg$, $\wedge$, ...
- Cuantificatori: $\forall$, $\exists$

Sintaxă

Termeni

· Termeni (obiecte):

- Constante;
- Variabile;
- Aplicații de funcții: $f(t_1, \ldots, t_n)$, unde f este un simbol funcțional n-ar și $t_1, \ldots, t_n$ sunt termeni.

  Exemple:
  - $succesor(4)$: succesorul lui 4, și anume 5.
  - $+(2, x)$: aplicația funcției de adunare asupra numerelor 2 și x, și, totodată, suma lor.

Sintaxă

Atomi

· Atomi (relații): atomul $p(t_1, \ldots, t_n)$, unde p este un predicat n-ar și $t_1, \ldots, t_n$ sunt termeni.

Exemple:

- $impar(3)$
- $varsta(ion, 20)$
- $= (+(2,3), 5)$

Sintaxă

Propoziții

· Propoziții (fapte) – dacă x variabilă, A atom, și α și β propoziții, atunci o propoziție are forma:

- Fals, Adevărat: $\perp$, $\top$
- Atomi: A
- Negații: $\neg\alpha$
- Conectori: $\alpha \wedge \beta$, $\alpha \Rightarrow \beta$, ...
- Cuantificări: $\forall x.\alpha$, $\exists x.\alpha$

Sintaxă

Exemplu

Exemplul 39.1.

"Dan este prieten cu sora Ioanei"

$$\underbrace{\exists X.\underbrace{prieten(\underbrace{X}_{\text{termen}}, \underbrace{sora(\overbrace{ioana}^{\text{termen}})}_{\text{termen}})}_{\text{atom/propoziție}} \wedge \underbrace{destept(X)}_{\text{atom/propoziție}}}_{\text{propoziție}}$$

Semantică

Semantică
Interpretare

Definiția 40.1 (Interpretare).

O interpretare constă din:

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

Semantică
Elemente

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

Cuantificatori
Exemple

Exemplul 40.2.

1. "Vrabia mălai visează." $\forall x.(vrabie(x) \Rightarrow viseaza(x, malai))$
2. "Unele vrăbii visează mălai."
   $\exists x.(vrabie(x) \wedge viseaza(x, malai))$
3. "Nu toate vrăbiile visează mălai."
   $\exists x.(vrabie(x) \wedge \neg viseaza(x, malai))$
4. "Nicio vrabie nu visează mălai."
   $\forall x.(vrabie(x) \Rightarrow \neg viseaza(x, malai))$
5. "Numai vrăbiile visează mălai."
   $\forall x.(viseaza(x, malai) \Rightarrow vrabie(x))$
6. "Toate și numai vrăbiile visează mălai."
   $\forall x.(viseaza(x, malai) \Leftrightarrow vrabie(x))$

Cuantificatori
Greșeli frecvente

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

Cuantificatori
Proprietăți

- Necomutativitate:
  - $\forall x.\exists y.viseaza(x, y) \to$ "Toți visează la ceva anume."
  - $\exists x.\forall y.viseaza(x, y) \to$ "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 propoziții
Analoage logicii propoziționale

- Satisfiabilitate.
- Validitate.
- Derivabilitate.
- Inferență.

Forme normale

Forme normale
Definiții (1)

Definiția 41.1 (Literal).

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

Definiția 41.2 (Expresie clauzală).

Literal sau disjuncție de literali.
Exemplu: $prieten(x, y) \lor \neg doctor(x)$.

Definiția 41.3 (Clauză).

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

Forme normale
Definiții (2)

Definiția 41.4 (Forma clauzală / Forma normală conjunctivă – FNC).

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

Definiția 41.5 (Forma normală implicativă – FNI).

Reprezentarea unei propoziții sub forma unei mulțimi de clauze, implicit legate prin conjuncții, în care fiecare clauză are forma grupată
$\{\neg A_1, \ldots, \neg A_m, B_1, \ldots, B_n\}$,
corespunzătoare implicației
$(A_1 \land \ldots \land A_m) \Rightarrow (B_1 \lor \ldots \lor B_n)$, unde A_i și B_j sunt atomi.

Forme normale
Definiții (3) – Clauze Horn

Definiția 41.6 (Clauză Horn).

Clauză în care un singur literal este în formă pozitivă:
$\{\neg A_1, \ldots, \neg A_n, A\}$,
corespunzătoare implicației
$A_1 \land \ldots \land A_n \Rightarrow A$.

Exemplul 41.7 (Clauze Horn).

Transformarea propoziției
$vrabie(x) \lor ciocarlie(x) \Rightarrow pasare(x)$ 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 propozițiilor în FNC (1)
Eliminare implicații, împingere negații, redenumiri

1. Eliminarea implicațiilor ($\not\Rightarrow$)
2. Împingerea negațiilor până în fața literalilor ($\vec{\neg}$)
3. Redenumirea variabilelor cuantificate pentru obținerea unicității de nume (R):

$$\forall x.p(x) \land \forall x.q(x) \lor \exists x.r(x) \;\rightarrow\; \forall x.p(x) \land \forall y.q(y) \lor \exists z.r(z)$$

4. Deplasarea cuantificatorilor la începutul expresiei, conservându-le ordinea (forma normală *prenex*) (P):

$$\forall x.p(x) \land \forall y.q(y) \lor \exists z.r(z) \;\rightarrow\; \forall x.\forall y.\exists z.(p(x) \land q(y) \lor r(z))$$

Conversia propozițiilor în FNC (2)
Skolemizare

5. Eliminarea cuantificatorilor existențiali (skolemizare) (S):
   - Dacă nu este precedat de cuantificatori universali: înlocuirea aparițiilor variabilei cuantificate printr-o constantă:
   $$\exists x.p(x) \;\rightarrow\; p(c_x)$$
   - Dacă este precedat de cuantificatori universali: înlocuirea aparițiilor variabilei cuantificate prin aplicația unei funcții unice asupra variabilelor anterior cuantificate universal:
   $$\forall x.\forall y.\exists z.(p(x) \land q(y) \lor r(z))$$
   $$\rightarrow\; \forall x.\forall y.(p(x) \land q(y) \lor r(f_z(x, y)))$$

Conversia propozițiilor în FNC (3)
Cuantificatori universali, Distribuire $\lor$, Clauze

6. Eliminarea cuantificatorilor universali, considerați, acum, impliciți ($\not\forall$):

$$\forall x.\forall y.(p(x) \land q(y) \lor r(f_z(x, y))) \;\rightarrow\; p(x) \land q(y) \lor r(f_z(x, y))$$

7. Distribuirea lui $\lor$ față de $\land$ ($\lor/\land$):

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

8. Transformarea expresiilor în clauze (C).

Conversia propozițiilor în FNC – Exemplu

Exemplul 41.8.

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

$\not\Rightarrow$	$\forall x.(\neg\forall y.(\neg lab(y) \lor rezolva(x, y)) \lor \exists y.apreciaza(y, x))$
$\vec{\neg}$	$\forall x.(\exists y.\neg(\neg lab(y) \lor rezolva(x, y)) \lor \exists y.apreciaza(y, x))$
$\vec{\neg}$	$\forall x.(\exists y.(lab(y) \land \neg rezolva(x, y)) \lor \exists y.apreciaza(y, x))$
R	$\forall x.(\exists y.(lab(y) \land \neg rezolva(x, y)) \lor \exists z.apreciaza(z, x))$
P	$\forall x.\exists y.\exists z.((lab(y) \land \neg rezolva(x, y)) \lor apreciaza(z, x))$
S	$\forall x.((lab(f_y(x)) \land \neg rezolva(x, f_y(x))) \lor apreciaza(f_z(x), x))$
$\not\forall$	$(lab(f_y(x)) \land \neg rezolva(x, f_y(x))) \lor apreciaza(f_z(x), x)$
$\lor/\land$	$(lab(f_y(x)) \lor apr(f_z(x), x)) \land (\neg rez(x, f_y(x)) \lor apr(f_z(x), x))$
C	$\{lab(f_y(x)), apr(f_z(x), x)\}, \{\neg rez(x, f_y(x)), apr(f_z(x), x)\}$

Unificare și rezoluție

Unificare

- Utilizată pentru rezoluție
- vezi și sinteza de tip – Def. 25.5, 25.6, 25.8
- reguli:
  - o propoziție unifică cu o propoziție de aceeași formă
  - două predicate unifică dacă au același nume și parametri care unifică
  - o variabilă unifică cu un termen care nu conține variabila

Unificare
Observații

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

Unificare
Rolul în rezoluție

- Rezoluția pentru clauze Horn:

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

- $unificare(\alpha, \beta) \rightarrow$ substituția sub care unifică propozițiile α și β;
- $subst(S, \alpha) \rightarrow$ propoziția rezultată în urma aplicării substituției S asupra propoziției α.

Rezoluție
Exemplu

Exemplul 42.1.
Horses and hounds

- Horses are faster than dogs.
- There is a greyhound that is faster than any rabbit.
- Harry is a horse and Ralph is a rabbit.
- Is Harry faster than Ralph?

Sfârșitul cursului 11(a)
Ce am învățat

- sintaxa și semantica în LPOI
- Forme normale, Unificare, Rezoluție în LPOI

Cursul 12

Programare logică în Prolog

Cuprins

Introducere

Programare logică

- Reprezentare simbolică;
- Stil declarativ;
- Separarea datelor de procesul de inferență, 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*) este că nu se poate afirma nimic despre ceea ce nu poate fi demonstrat.

Procesul de 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.

Un exemplu de program Prolog

Exemplul 44.1.

```
parent(andrei, bogdan).
parent(andrei, bianca).
parent(bogdan, cristi).

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

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

Exemplul genealogic (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)`	`p(andrei, bianca)`	`p(bogdan, cristi)`
↓	↓	↓
$\cdots 1$	$\cdots 2$	$\cdots 3$

Exemplul genealogic (2)
Ramura 1

$\cdots 1$

$\downarrow$

`p(andrei, bogdan)`

$\downarrow$

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

$\downarrow$

`p(bogdan, cristi)`

$\downarrow$

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

$\downarrow$

success

`gp(andrei, cristi)`

Exemplul genealogic (3)
Ramura 2

$\cdots 2$

$\downarrow$

`p(andrei, bianca)`

$\downarrow$

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

$\downarrow$

eșec

Exemplul genealogic (4)
Ramura 3

$\cdots 3$

$\downarrow$

`p(bogdan, cristi)`

$\downarrow$

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

$\downarrow$

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 $goals = \emptyset$ **then**
  return SUCCESS
$goal \leftarrow head(goals)$
$goals \leftarrow tail(goals)$
for-each $rule \in rules$ **do** // în ordinea din program
  if $unify(goal, conclusion(rule), subst) \rightarrow bindings$
    $newGoals \leftarrow premises(rule) \cup goals$ // adâncime
    $newSubst \leftarrow subst \cup bindings$
    if *BackwardChaining*(*rules*, *newGoals*, *newSubst*)
      then return SUCCESS
return FAILURE

Controlul execuţiei

Exemplu – Minimul a două numere
Cod Prolog

Exemplul 45.1 (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
Observații

- Condiții mutual exclusive: `X =< Y` și `X > Y` → cum putem eliminа redundanța?

Exemplul 45.2.

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

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

- Greșit!

Exemplu – Minimul a două numere
Îmbunătățire

- Soluție: oprirea recursivității după prima satisfacere a scopului.

Exemplul 45.3.

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

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


Operatorul *cut*
Definiție

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

Operatorul *cut*
Exemplu

Exemplul 45.4.

```
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.
```


Negația ca eșec

Exemplul 45.5.

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

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

Sfârșitul cursului 12
Ce am învățat

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

Cursul 13

Mașina algoritmică Markov și programare asociativă în CLIPS

Cuprins

Introducere

Mașina algoritmică Markov

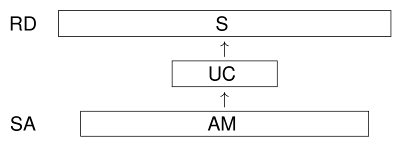
- Model de calculabilitate efectivă, echivalent cu Mașina Turing și Calculul Lambda;
- Principiul de funcționare: *pattern matching* și substituție;
- Fundamentul teoretic al paradigmei asociative și al limbajelor bazate pe reguli (de forma *dacă-atunci*).

Paradigma asociativă
Caracteristici

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

Mașina algoritmică Markov

Structura Mașinii Markov
Perspectivă generală



- Registrul de date, RD, cu secvența de simboluri, S
  - RD nemărginit la dreapta
  - $S \subset (A_b \cup A_l)^*, A_b \cap A_l = \emptyset$ – alfabet de bază și de lucru
- Unitatea de control, UC
- Spațiul de stocare a algoritmului, SA, ce conține algoritmul Markov, AM
  - format din reguli.

Structura Mașinii Markov
Reguli

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

  `șablon identificare (LHS)` → `șablon substituție (RHS)`

- Exemplu: `ag`$_1$`c -> ac`
- șabloanele → secvențe de simboluri:
  - constante: simboluri din A_b
  - variabile locale: simboluri din A_l
  - variabile generice: simboluri speciale, din mulțimea G, legați la simboluri din A_b
- Dacă RHS este "." → regulă terminală, ce încheie execuția mașinii (halt).

Structura Mașinii Markov
Variabile generice

- De obicei, notate cu `g`, urmat de un indice;
- Mulțimea valorilor pe care le poate lua o variabilă → domeniul variabilei – `Dom(g)` $\subseteq A_b \cup A_l$;
- Legate la exact un simbol la un moment dat;
- Durata de viață → timpul aplicării regulii – sunt legate la identificarea șablonului și legarea se pierde după înlocuirea șablonului de identificare cu cel de substituție;
- Utilizabile în `RHS` doar în cazul apariției în `LHS`.

Structura Mașinii Markov
Algoritm Markov

- Mulțimi ordonate de reguli, îmbogățite cu declarații:
  - de partiționare a mulțimii A_b
  - de variabile generice

Exemplul 47.1.

Eliminarea din mulțimea A simbolurilor ce aparțin mulțimii 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

Definiția 47.2 (Aplicabilitatea unei reguli).

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

- $a_i \in A_b \cup A_l \wedge a_i = c_i$
- $a_i \in G \wedge c_i \in Dom(a_i) \wedge$
  $(\forall j = \overline{1,n} \,.\; a_j = a_i \Rightarrow c_j = c_i)$,
- oriunde mai apare aceeași variabilă generică în șablonul de identificare, în poziția corespunzătoare din subșir avem același simbol.

Reguli
Aplicare

Definiția 47.3 (Aplicarea unei reguli).

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

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

Reguli
Exemplu de aplicare

Exemplul 47.4.

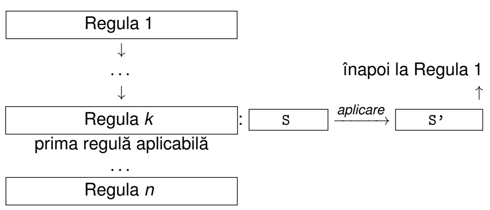
- $A_b = \{1, 2, 3\}$
- $A_l = \{x, y\}$
- $Dom(g_1) = \{2\}$
- $Dom(g_2) = A_b$
- $S = 1111112x2y31111$
- $r : 1g_1xg_1yg_2 \rightarrow 1g_2x$

S	=	11111	1	2	x	2	y	3	1111
r	:		1	g_1	x	g_1	y	g_2	→ $1g_2x$
S'	=	11111**13x**1111							

Unitatea de control
Aplicabilitate vs. aplicare

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

Unitatea de control
Funcționare



Exemplu
Inversarea intrării

- Ideea: mutarea, pe rând, a fiecărui element în poziția corespunzătoare. Mutarea se face prin pași incrementali de interschimbare a elementelor învecinate.

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

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

Introducere în CLIPS

CLIPS

- "C Language Integrated Production System";
- Sistem bazat pe reguli → "producție" = regulă;
- Principiu de funcționare similar cu al mașinii Markov;
- Dezvoltat la NASA în anii 1980;
- Posibilitatea codificării de implicații logice în reguli → sisteme expert (mimează procesul de decizie al unui expert uman).
- Sistemele expert pot fi folosite în configurarea de sisteme, diagnoză (medicală etc.), educație, planificare, prognoză, etc.

Exemplu

Minimul a două numere – reprezentare individuală

Exemplul 48.1.

```
(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 mașinii Markov;
- Afirmații despre atributele obiectelor;
- Date simbolice, construite conform unor șabloane;
- Mulțimea de fapte → baza de cunoștințe (*factual knowledge base*)

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


Reguli

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

Înregistrări de activare

Definiție

- Tuplul (regulă, fapte asupra cărora este aplicabilă) → înregistrare de activare (*activation record*);
- Reguli posibil aplicabile asupra diferitelor porțiuni ale acelorași fapte;
- Muțimea 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 refracției

- Aplicarea unei reguli o singură dată asupra acelorași fapte și acelorași porțiuni ale acestora;
- Altfel, programe care nu s-ar termina.

Terminarea programelor

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

Exemple

Minimul a două numere
Reprezentare agregată a numerelor – Exemplu (1)

Exemplul 49.1.

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

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

- Observați utilizarea `$?` pentru potrivirea unei secvențe, potențial vidă.

Minimul a două numere
Reprezentare agregată a numerelor – Exemplu (2)

```
> (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 – Alt exemplu (1)

Exemplul 49.2.

```
(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 – Alt exemplu (2)

- Selectarea explicită a celor 2 numere împiedică alegerea automată, convenabilă, a acestora, ca în exempl → 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 49.3.

```
(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 49.4.

```
(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)
```


Sfârșitul cursului 13
Ce am învățat

- Ce este și cum funcționează mașina algoritmică Markov: structură, variabile, reguli, algoritmul unității de control.
- Introducere în CLIPS – fapte, reguli, execuție.