Paradigme de Programare

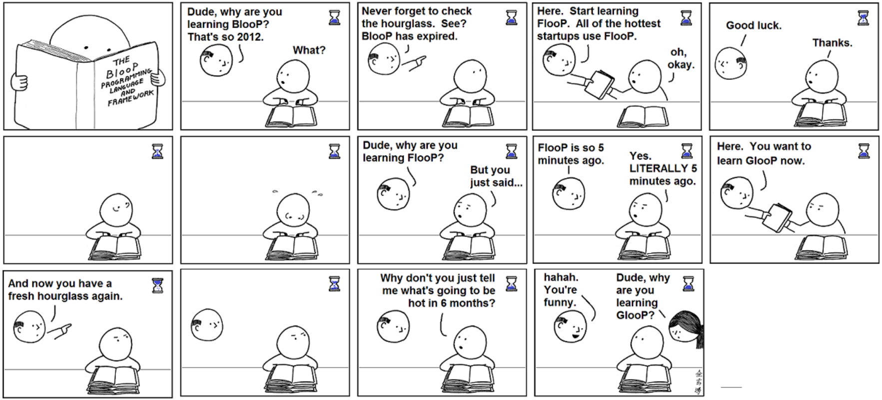
Conf. dr. ing. Andrei Olaru

andrei.olaru@upb.ro
Departamentul de Calculatoare

2026

Cursul 1: Introducere

BlooP and FlooP and GlooP



Exemplu

Exemplu

Ex Exemplu

Să se determine dacă un element e se regăsește într-o listă L ($e \in L$).

Să se sorteze o listă L.

Modelare funcțională (1)

Racket:

```
(define memList (lambda (e L)
    (if (null? L)
        #f
        (if (equal? (first L) e)
            #t
            (memList e (rest L))
            ))
    ))

(define ins (lambda (x L)
 (cond ((null? L) (list x))
       ((< x (first L)) (cons x L))
       (else (cons (first L) (ins x (rest L)))))))
```

Modelare funcțională (2)

λPP

Haskell

```
memList x [] = False
memList x (e:t) = x == e || memList x t

ins x [] = [x]
ins x l@(h:t) = if x < h then x:l else h : ins x t
```

Modelare logică

λPP

Prolog:

```
memberA(E, [E|_]) :- !.
memberA(E, [_|L]) :- memberA(E, L).

% elementul, lista, rezultatul
ins(E, [], [E]).
ins(E, [H | T], [E, H | T]) :- E < H, !.
ins(E, [H | T], [H | TE]) :- ins(E, T, TE).
```

Ce studiem la PP?

Elemente pe care le vom studia

- Paradigma funcțională și discuție despre paradigme de programare în general, în contrast cu paradigma imperativă și cea orientată-obiect.
- Tipuri de date abstracte
- Racket: introducere în programare funcțională
- Calculul λ ca bază teoretică a paradigmei funcționale
- Racket: întârzierea evaluării și fluxuri

- Haskell: programare funcțională cu o sintaxă avansată
- Haskell: evaluare leneșă și fluxuri
- Haskell: tipuri, sinteză de tip, și clase
- Equational Reasoning

De ce studiem această materie?

The first math class.

The first math class.

[(C) Zach Weinersmith, Saturday Morning Breakfast Cereal]

[https://www.smbc-comics.com/comic/a-new-method]

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

· până acum ați studiat paradigma imperativă (legată și cu paradigma orientată-obiect)

→ un anumit mod de a privi procesul de rezolvare al unei probleme și de a căuta soluții la probleme de programare.

· paradigmele declarative studiate oferă o gamă diferită (complementară!) de unelte → alte moduri de a rezolva anumite probleme.

⇒ o pregătire ce permite accesul la poziții de calificare mai înaltă (arhitect, designer, etc.)

De ce?

Sunt aceste paradigme relevante?

- evaluarea leneșă → prezentă în Python (de la v3), .NET (de la v4)
- funcții anonime → prezente în C++ (de la v11), C#/.NET (de la v3.0/v3.5), Dart, Go, Java (de la JDK8), JS/ES, Perl (de la v5), PHP (de la v5.0.1), Python, Ruby, Swift.
- Prolog și programarea logică sunt folosite în software-ul modern de A.I., e.g. Watson; automated theorem proving.
- În industrie sunt utilizate limbaje puternic funcționale precum Erlang, Scala, F#, Clojure.
- Limbaje multi-paradigmă → adaptarea paradigmei utilizate la necesități.

De ce?

O bună cunoaștere a paradigmelor alternative $\rightarrow$ \$\$\$

- Developer Survey 2025

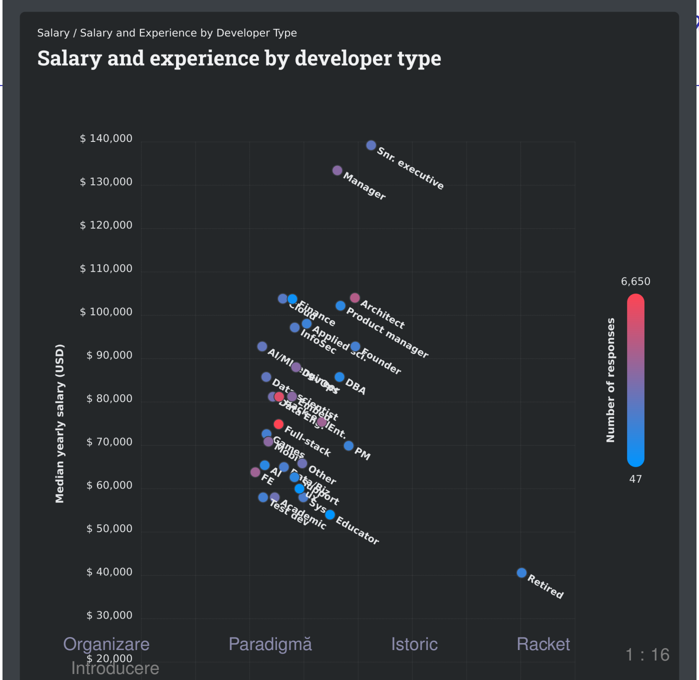
  [https://survey.stackoverflow.co/2025]

- Developer Survey 2024

  [https://survey.stackoverflow.co/2024]

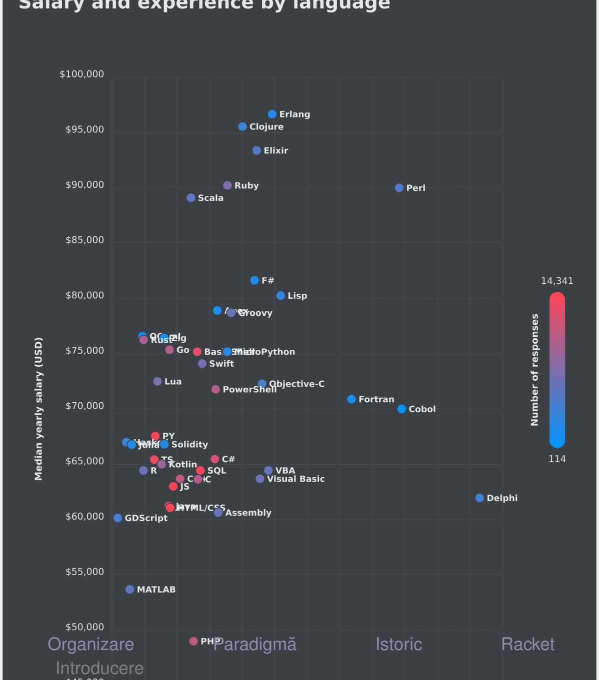
De ce?

Cine câștigă cel mai bine?



De ce?

Cine câștigă cel mai bine?



Organizare

Unde găsesc informații?
Resurse de bază

https://ocw.cs.pub.ro/courses/pp

Regulament: https://ocw.cs.pub.ro/courses/pp/--/regulament

Forumuri: Moodle → 03-ACS-L-CTI-Calculatoare-A2-S2-PP-CA-CB-CC
https://curs.upb.ro/2025/course/view.php?id=10195

Elementele cursului sunt comune la seriile CA, CB și CC.

Notare

mai multe la `https://ocw.cs.pub.ro/courses/pp/--/regulament`

- Laborator: 0.2p ← pentru strădiuință
- Teste grilă la laborator: 0.3p ← cu bonus până la 0.4p
- Teme (2): TBD ← cu bonusuri de până la 20%
- Activitate curs: 0.5p ← se punctează întrebările bune și foarte bune
- Test din materia de laborator: TBD

punctajele pe parcurs se trunchiază la 5p

- Examen: 5p ← limbaje + teorie

Paradigma de programare

Ce înseamnă paradigma de programare

Ce diferă între paradigme?

- diferă sintaxa ←
  - aceasta este o diferență între limbaje, dar este influențată și de natura paradigmei
  - mecanisme specifice unei paradigme aduc elemente noi de sintaxă

    e.g. funcțiile anonime

- diferă modul de construcție al expresiilor ← ce poate reprezenta o expresie, ce operatori putem aplica între expresii.

- diferă structura programului ←
  - ce anume reprezintă programul
  - cum se desfășoară execuția programului

Ce înseamnă paradigma de programare
Ce caracterizează o paradigmă?

- valorile de prim rang
- modul de construcție a programului
- modul de tipare al valorilor
- ordinea de evaluare (generare a valorilor)
- modul de legare al variabilelor (managementul valorilor)
- controlul execuției

· **Paradigma de programare** este dată de stilul fundamental de construcție al structurii și elementelor unui program.

Ce vom studia?
Conținutul cursului

λP.P

1. Diverse perspective conceptuale asupra noțiunii de calculabilitate efectivă → modele de calculabilitate.

2. Influența perspectivei alese asupra procesului de modelare și rezolvare a problemelor → paradigme de programare.

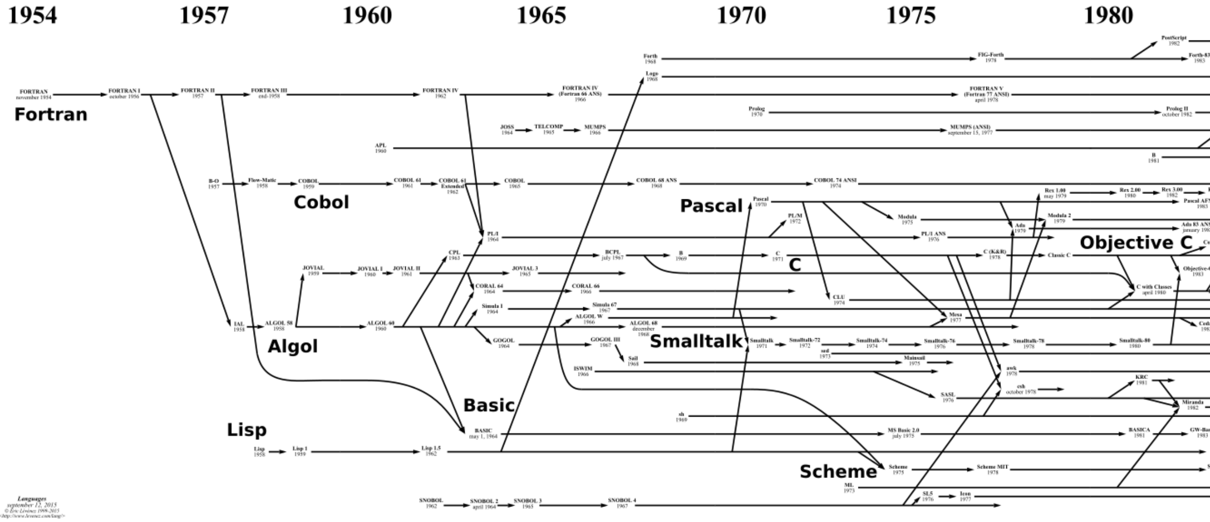
3. Limbaje de programare aferente paradigmelor, cu accent pe aspectul comparativ.

C, Pascal → procedural Java, C++, Python → orientat-obiect	→ paradigma imperativă	→ Mașina Turing	echivalente !
Racket, Haskell	→ paradigma funcțională	→ Mașina λ	
Prolog	→ paradigma logică	→ FOL + Resolution	
CLIPS	→ paradigma asociativă	→ Mașina Markov	

T | Teza Church-Turing: efectiv calculabil = Turing calculabil

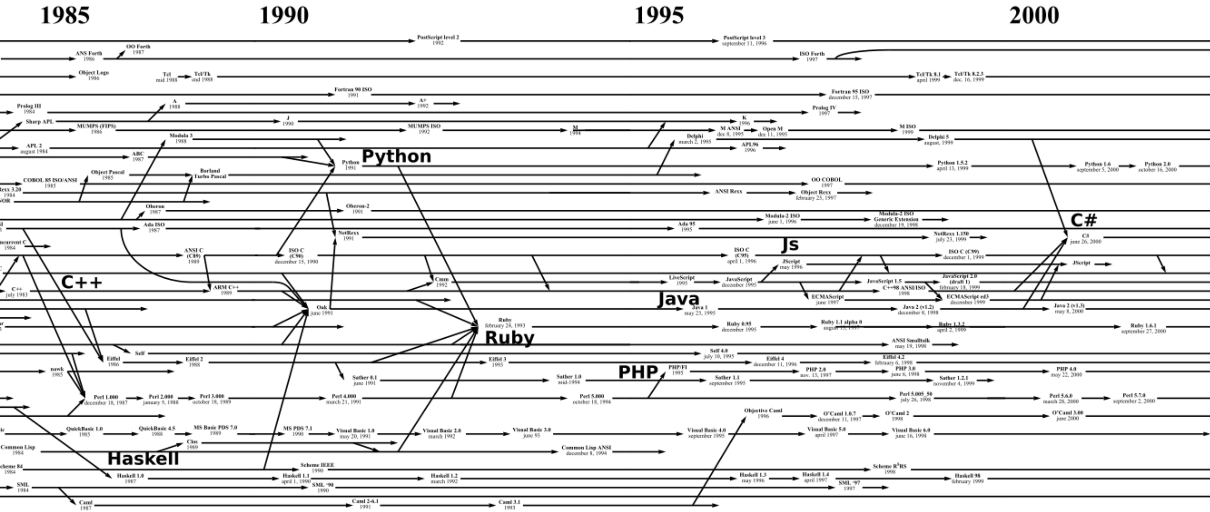
Istoric: Paradigme și limbaje de programare

Istorie
1950-1975

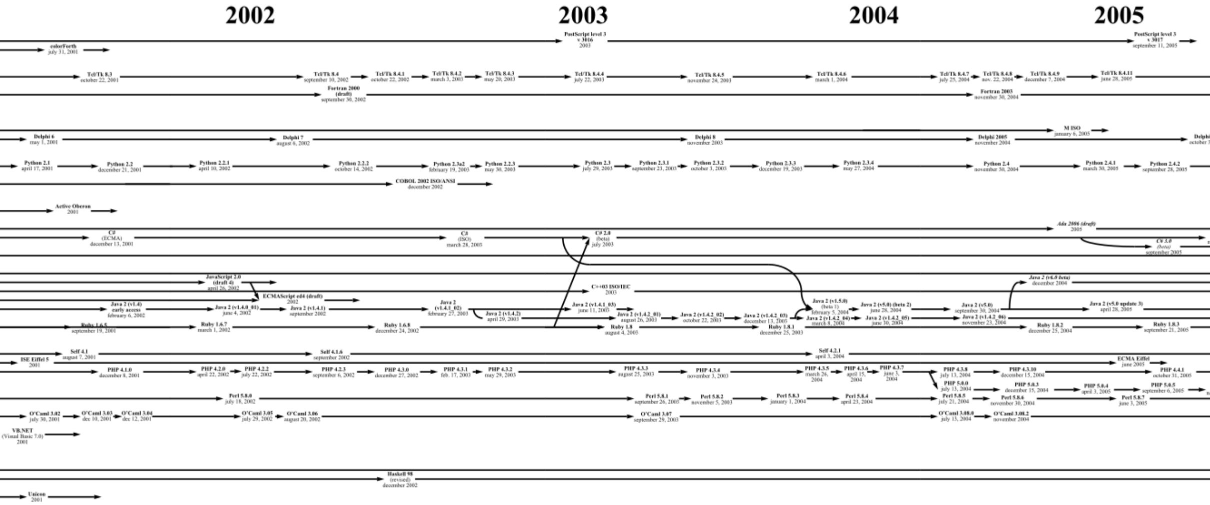


Istorie

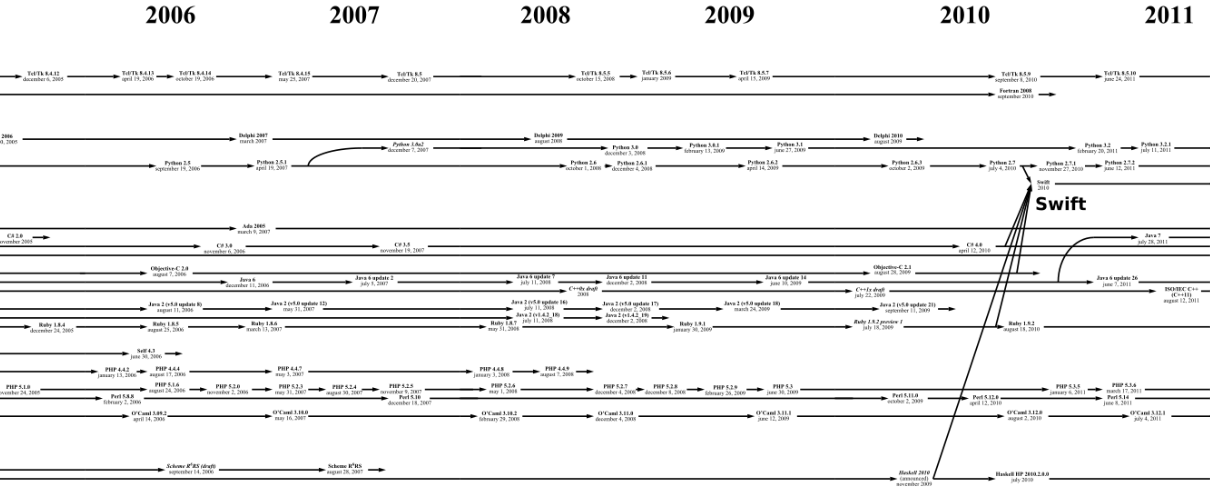
1975-1995



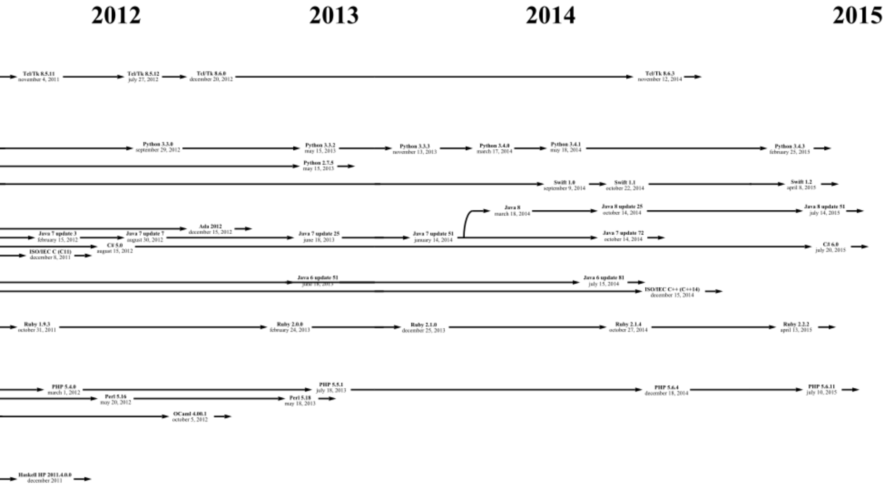
Istorie
1995-2002



Istorie
2002-2006



Istorie
2006-2013



λPP

- imagine navigabilă (slides precedente): [http://www.levenez.com/lang/]
- Wikipedia:

[http://en.wikipedia.org/wiki/Generational_list_of_programming_languages]

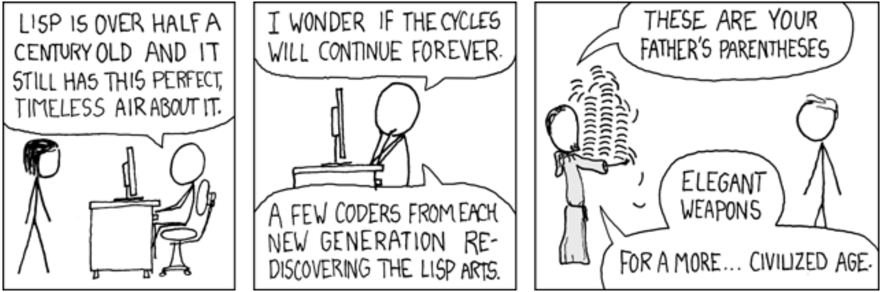
[https://en.wikipedia.org/wiki/Timeline_of_programming_languages]

[https://en.wikipedia.org/wiki/Programming_paradigm#/media/File:Programming_paradigms.svg]

Introducere în Racket

Lisp cycles

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



Racket

din 1975 (ca Scheme)

- 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

Cursul 2: Tipuri de date abstracte

8. TDA
9. Constructori & Axiome
10. Inducție structurală

TDA

Tipuri de date abstracte

Putem avea tipuri de date făcând abstracție de implementare?

- putem discuta despre tipuri de date fără a face apel la implementarea tipului și la modalitatea de stocare a datelor în memorie / pe disc
- definim tipurile de date într-un mod matematic
- avem nevoie să știm:
  1. cum construim valorile tipului de date?
  2. cum operăm cu valorile tipului de date?

Ex — Exemplu

- Ce este un număr natural?
- Ce este o listă?
- Ce este un arbore?
- Ce este o expresie aritmetică?

Structura unui TDA
Ce avem nevoie pentru a defini un TDA?

λPP

- Operații disponibile pe tipul de date → Operații

  Ex | Exemplu

  - numere naturale: adunare, scădere
  - liste: adăugare, calcul lungime
  - expresii aritmetice: evaluare, simplificare

- Moduri de a construi valorile tipului de date → Constructori

  Ex | Exemplu

  - numere naturale: adunare, scădere, incrementare, verificare pentru 0
  - liste: adăugare, eliminare, concatenare
  - expresii aritmetice: aplicare operatori

- Axiome care garanteaza funcționarea curentă a operațiilor

Tip de date abstract
Definiție

+ Tip de date abstract – TDA – Model matematic al unei mulțimi de valori și al operațiilor 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.

Constructori & Axiome

+ Constructor de bază – parte a unui set de constructori care este necesar și suficient pentru a obține toate valorile din mulțimea de valori a tipului de date abstract.

Ex Exemplu

- tipul Natural: $zero : Natural$; $succesor : Natural \rightarrow Natural$
- tipul Listă: $null : List$; $cons : T \times List \rightarrow List$
- tipul Expresie: $valoare : Natural \rightarrow Expresie$;
  $operatie : Expresie \times Operand \times Expresie \rightarrow Expresie$

Constructori de bază

Setul de constructori nu este unic

- Pot avea și alte seturi de constructori de bază.
- E.g. pentru listă pot avea
  - *null* : $List$
  - *singleton* : $T \rightarrow List$
  - *concatenare* : $List \times List \rightarrow List$

+ Constructorii pentru un tip T pot fi de tip:

- Nular – 0 parametri
  Ex *zero*, *null*
- Extern – doar cu parametri care nu sunt de tip T
  Ex *singleton*, *valoare*
- Intern – cu cel puțin un parametru de tip T
  Ex *succesor*, *cons*, *operație*

+ | Dimensiunea unei valori este numărul de constructori de bază din formula care produce valoarea

Ex Exemplu

- "2" : $succ(succ(zero))$ — dimensiunea 3
- "[1, 7, 2]" : $cons(1, cons(7, cons(2, null)))$ — dimensiunea 4

- Constructorii nulari și externi produc întotdeauna valori de dimensiune 1
- Constructorii interni produc întotdeauna valori de dimensiune > 1

Ex

Exemplu

- $concatenare(cons(x, L1), L2) = cons(x, concatenare(L1, L2))$
- $concatenare(null, L) = L$

- $adunare(succesor(n), m) = succesor(adunare(n, m))$
- $adunare(zero, m) = m$

- $evaluare(operatie(E1, +, E2)) = adunare(evaluare(E1), evaluare(E2))$

Axiome
Definiție

+ | Axiomele definesc comportamentul operatorilor pe toate valorile tipului.

Pentru fiecare operator avem nevoie de câte o axiomă pentru fiecare constructor de bază.

Inducție structurală

· La inducția matematică, demonstrăm cazul de bază (e.g. $n = 0$) și presupunând ca adevărată afirmație pentru valorile până la $n - 1$, demonstrăm adevărul afirmației pentru n.

Putem folosi inducția matematică pentru tipul *Natural*, dar pentru celelalte?

Pentru celelate, folosim inducția structurală.

+ Inducția structurală permite demonstrarea că dacă o afirmație despre o operație este adevărată pentru cazurile de bază (constructori nulari și interni) și dacă, presupunând că afirmația este adevărată pentru valorile de dimensiune $n-1$, putem demonstra că afirmația este adevărată pentru valorile de dimensiune n, atunci afirmația este adevărată întotdeauna.

Exemplu

Pentru $add : Natural \times Natural \rightarrow Natural$

cu axiomele

- ADD1 $add(zero, n) = n$
- ADD2 $add(succesor(m), n) = succesor(add(m, n))$

demonstrăm că
$\forall a, b, c \in Natural \,.\, add(a, add(b, c)) = add(add(a, b), c)$ (asociativitate)

Caz de bază:

$add(zero, add(b, c)) = add(b, c)$ (conform ADD1)

$add(add(zero, b), c) = add(b, c)$ (conform ADD1) $\square$

Exemplu

cu axiomele

- ADD1 $add(zero, n) = n$
- ADD2 $add(succesor(m), n) = succesor(add(m, n))$

Ipoteză inductivă: $add(x, add(b, c)) = add(add(x, b), c)$

Pas de inducție:

stânga: $add(succ(x), add(b, c)) = succ(add(x, add(b, c)))$ (conform ADD2)

dreapta: $add(add(succ(x), b), c) = add(succ(add(x, b)), c)$ (conform ADD2)
$= succ(add(add(x, b), c))$ (conform ADD2)

dar: $add(x, add(b, c)) = add(add(x, b), c)$ (conform ipoteză) $\square$

Introducere

Modele de calculabilitate

De ce?

- ne punem problema dacă putem realiza un calcul sau nu $\rightarrow$ pentru a demonstra trebuie să avem un model simplu al calculului (cum realizăm calculul, în mod formal).
- un model de calculabilitate trebuie să fie cât mai simplu, atât ca număr de operații disponibile cât și ca mod de construcție a valorilor.
- corectitudinea unui program se demonstrează mai ușor dacă limbajul de programare este mai apropiat de mașina teoretică (modelul abstract de calculabilitate).

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 Mașina Turing (v. Teza Church-Turing)
- Axat pe conceptul matematic de funcție – totul este o funcție

Aplicații
ale calculului λ

- Aplicații importante în
  - programare
  - demonstrarea formală a corectitudinii programelor, datorită modelului simplu de execuție

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

Lambda-expresii

λ-expresii
Exemple

Exemplu

1. $x \rightarrow$ variabila (numele) x
2. $\lambda x.x \rightarrow$ funcția identitate
3. $\lambda x.\lambda y.x \rightarrow$ funcție selector
4. $(\lambda x.x\ \ y) \rightarrow$ aplicația funcției identitate asupra parametrului actual y
5. $(\lambda x.(x\ \ x)\ \ \lambda x.x) \rightarrow$?

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

λ

+ λ-expresie

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

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

$$(\ \boxed{(\ \lambda x.\lambda y.x\ \ z\ )}\ \ t)$$ ← parametru formal
parametru actual

$$\Vert$$

substituție

$$\Downarrow$$

$$(\lambda y.z\ \ t)$$

$$\boxed{(\ \lambda y.z\ \ t\ )}$$ ← parametru formal
parametru actual

$$\Vert$$

substituție

$$\Downarrow$$

$$z$$ ← nu mai este nicio funcție de aplicat

Reducere

β-redex
Cum arată (Formal, vedem mai târziu)

- 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.

+ | Apariție legată O apariție x_n a unei variabile x este legată într-o expresie E dacă:

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

+ | Apariție liberă O apariție a unei variabile este liberă într-o expresie dacă nu este legată în acea expresie.

- Atenție! În raport cu o expresie dată!

Apariții ale variabilelor

λ

Mod de gândire

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

- $\underset{<1>}{x}$ $\leftarrow$ apariție liberă

- $(\lambda y.\ \underset{<1>}{x}\ \ z)$ $\leftarrow$ apariție încă liberă, nu o leagă nimeni

---

- $\lambda \underset{<2>}{x}.(\lambda y.\ \underset{<1>}{x}\ \ z)$ $\leftarrow$ $\lambda \underset{<2>}{x}$ *leagă* apariția $\underset{<1>}{x}$

- $(\lambda \underset{<2>}{x}.\underbrace{(\lambda y.\ \underset{<1>}{x}\ \ z)}_{\text{corp } \lambda x_2}\ \underset{<3>}{x})$ $\leftarrow$ apariția x_3 este liberă – este în exteriorul corpului funcției cu parametrul formal x (λx_2)

- $\lambda \underset{<4>}{x}.(\lambda \underset{<2>}{x}.(\lambda y.\ \underset{<1>}{x}\ \ z)\ \underset{<3>}{x})$ $\leftarrow$ $\lambda \underset{<4>}{x}$ *leagă* apariția $\underset{<3>}{x}$

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

+ | 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

λ

Exemplu

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

$(\lambda \underset{<1>}{x} . \underbrace{\underset{<2>}{x}}_{F} \underset{<3>}{x}).$

- $\underset{<1>}{x}, \underset{<2>}{x}$ legate în E
- $\underset{<3>}{x}$ liberă în E
- $\underset{<2>}{x}$ liberă în F!
- x liberă în E și F

Variabile și apariții ale lor

Exemplu 2

Exemplu

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

$$(\lambda \underset{<1>}{x}.\underbrace{\lambda \underset{<1>}{z}.(\underset{<2>}{z}\ \underset{<2>}{x})}_{F}\ (\underset{<3>}{z}\ \underset{<1>}{y})).$$

- $\underset{<1>}{x}, \underset{<2>}{x}, \underset{<1>}{z}, \underset{<2>}{z}$ legate în E
- $\underset{<1>}{y}, \underset{<3>}{z}$ libere în E
- $\underset{<1>}{z}, \underset{<2>}{z}$ legate în F
- $\underset{<2>}{x}$ 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

λ

+ **O expresie închisă** este o expresie care nu conține variabile libere.

Ex Exemplu

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

- Variabilele libere dintr-o λ-expresie pot sta pentru alte λ-expresii
- Înaintea evaluării, o expresie trebuie adusă la forma închisă.
- Procesul de înlocuire trebuie să se termine.

β-reducere
Definiție

+ | β-reducere: Evaluarea expresiei $(\lambda x.E\ \ A)$, cu E și A λ-expresii, prin substituirea textuală a tuturor aparițiilor libere ale parametrului **formal** al funcției, x, din corpul acesteia, E, cu parametrul **actual**, A:

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

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

λ

Exemplu

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

  Care este problema?

- **Problemă**: în expresia $(\lambda x.E\ \ A)$:
  - dacă variabilele libere din A nu au nume comune cu variabilele legate din E: $FV(A) \cap BV(E) = \emptyset$
    $\rightarrow$ reducere întotdeauna corectă
  - dacă 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 $A \rightarrow$ α-conversie.

Ex | Exemplu

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

α-conversie
Definiție

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

Exemplu

- $\lambda x.y \rightarrow_\alpha \lambda y.y_{[y/x]} \rightarrow \lambda y.y \rightarrow$ 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 o variabilă liberă, existentă deja în E
- orice apariție liberă în E rămâne liberă în $E_{[y/x]}$

α-conversie
Exemple

λ

Ex Exemplu

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

+ Pas de reducere: O secvență formată dintr-o α-conversie și o β-reducere, astfel încât a doua se produce fără coliziuni:
$E_1 \rightarrow E_2 \equiv E_1 \rightarrow_\alpha E_3 \rightarrow_\beta E_2$.

+ 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 $\rightarrow$.

λ

: Reducere

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

Ex Exemplu

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

$$\Rightarrow$$

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

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

Terminarea reducerii (reductibilitate)

Exemplu și definiție

Ex Exemplu

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

Ω nu admite nicio secvență de reducere care se termină.

+ Expresie reductibilă este o expresie care admite (cel puțin o) secvență de reducere care se termină.

· expresia Ω nu este reductibilă.

Secvențe de reducere
și terminare

λ

Dar!

Exemplu

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

- E are o secvență de reducere care nu se termină;
- dar E are forma normală $y \Rightarrow E$ este reductibilă;
- lungimea secvențelor de reducere ale E este nemărginită.

Forme normale
Cum știm că s-a terminat calculul?

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

+ | Forma normală a unei expresii este o formă (la care se ajunge prin reducere, care nu mai conține β-redecși i.e. care nu mai poate fi redusă.

λ

+ | Forma normală funcțională – FNF este o formă $\lambda x.F$, în care F *poate* conține β-redecși.

Ex | Exemplu

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

- FN a unei expresii închise este în mod necesar FNF.
- într-o FNF nu există o necesitate imediată de a evalua eventualii β-redecși interiori (funcția nu a fost încă aplicată).

Unicitatea formei normale

Rezultate

T | Teorema 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 $E_2 \rightarrow^* E_3$.

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

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

λ

Ex Exemplu

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

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

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

$\Rightarrow$ Valorile sunt echivalente în raport cu redenumirea.

Modalități de reducere

Cum putem *organiza* reducerea?

λ

+ **Reducere stânga-dreapta:** Reducerea celui mai superficial și mai din stânga β-redex.

Ex Exemplu

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

+ **Reducere dreapta-stânga:** Reducerea celui mai adânc și mai din dreapta β-redex.

Ex Exemplu

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

Ce modalitate alegem?

T | Teorema 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

λ

1. Când se termină calculul? Se termină întotdeauna?
   → se termină cu forma normală [funcțională]. NU se termină decât dacă expresia este reductibilă.
2. Comportamentul depinde de secvența de reducere?
   → DA.
3. Dacă mai multe secvențe de reducere se termină, obținem întotdeauna același rezultat?
   → DA.
4. Dacă rezultatul este unic, cum îl obținem?
   → Reducere stânga-dreapta.
5. Care este valoarea expresiei?
   → Forma normală [funcțională] (FN[F]).

Ordine de evaluare
Tipuri

- + **Evaluare aplicativă** (*eager*) – corespunde unei reduceri *mai degrabă* dreapta-stânga. Parametrii funcțiilor sunt evaluați înaintea aplicării funcției.

- + **Evaluare normală** (*lazy*) – corespunde reducerii stânga-dreapta. Parametrii funcțiilor sunt evaluați la cerere.

- + **Funcție strictă** – funcție cu evaluare aplicativă.

- + **Funcție nestrictă** – funcție cu evaluare normală.

Ordine de evaluare
În practică

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

Exemplu

`(+ (+ 2 3) (* 2 3))` → `(+ 5 6)` → `11`

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

Exemplu

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

Limbajul lambda-0 și incursiune în TDA

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

Tipuri de date

Cum reprezentăm datele? Cum interpretăm valorile?

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

  Ex Exemplu

  $T \equiv_{\text{def}} \lambda x.\lambda y.x \qquad F \equiv_{\text{def}} \lambda x.\lambda y.y$

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

  Ex Exemplu

  $not \equiv_{\text{def}} \lambda x.((x\ F)\ T)$

  $(not\ T) \rightarrow (\lambda x.((x\ F)\ T)\ T) \rightarrow ((T\ F)\ T) \rightarrow F$

TDA *Bool*

Specificare

· Constructori:

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

· Operatori:

$$\begin{array}{|ll} not: & Bool \rightarrow Bool \\ and: & Bool^2 \rightarrow Bool \\ or: & Bool^2 \rightarrow Bool \\ if: & Bool \times A \times A \rightarrow A \end{array}$$

· Axiome:

$$\begin{array}{|lll} not: & not(T) = F & not(F) = T \\ and: & and(T, a) = a & and(F, a) = F \\ or: & or(T, a) = T & or(F, a) = a \\ if: & if(T, a, b) = a & if(F, a, b) = b \end{array}$$

TDA *Bool*

Implementarea constructorilor de bază

Intuiție bazat pe comportamentul necesar pentru if: selecția între cele două valori

- $T \equiv_{\text{def}} \lambda x.\lambda y.x$
- $F \equiv_{\text{def}} \lambda x.\lambda y.y$

λ

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

TDA *Pair*
Implementare

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

TDA *List* și *Natural*

Implementare

Intuiție: listă → pereche (*head*, *tail*)

- $nil \equiv_{\text{def}} \lambda x.T$
- $cons \equiv_{\text{def}} pair$
  - $((cons\ e)\ L) \rightarrow ((\lambda x.\lambda y.\lambda z.((z\ x)\ y)\ e)\ L) \rightarrow \lambda z.((z\ e)\ L)$
- $car \equiv_{\text{def}} fst$ $\qquad cdr \equiv_{\text{def}} snd$

Intuiție: număr → listă cu lungimea egală cu valoarea numărului

- $zero \equiv_{\text{def}} nil$
- $succ \equiv_{\text{def}} \lambda n.((cons\ nil)\ n)$
- $pred \equiv_{\text{def}} cdr$

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

Absența tipurilor

Chiar avem nevoie de tipuri? – Rolul tipurilor

λ

- Modalitate de exprimare a intenției programatorului;
- Documentare: ce operatori acționează asupra căror obiecte;
- Reprezentarea particulară a valorilor de tipuri diferite:
  `1`, `‘‘Hello’’`, `#t` etc.;
- Optimizarea operațiilor specifice;
- Prevenirea erorilor;
- Facilitarea verificării formale;

Absența tipurilor

Consecințe asupra reprezentării obiectelor

- Un număr, o listă sau un arbore, posibil desemnate de aceeași valoare!
- Valori și operatori reprezentați de funcții, semnificația fiind dependentă de context.
- Valoare aplicabilă asupra unei alte valori $\rightarrow$ operator!

Absența tipurilor

Consecințe asupra corectitudinii calculului

- Incapacitatea Mașinii λ de a
  - interpreta semnificația expresiilor;
  - asigura corectitudinea acestora (dpdv al tipurilor).
- Delegarea celor două aspecte programatorului;
- Orice operatori aplicabili asupra oricăror valori;
- Construcții eronate acceptate fără avertisment, dar calcule terminate cu
  - valori fără semnificație *sau*
  - expresii care nu sunt valori (nu au asociată o semnificație), dar sunt ireductibile

$\rightarrow$ instabilitate.

Absența tipurilor
Consecințe pozitive

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

…vin cu prețul unei dificultăți sporite în depanare, verificare și mentenanță

- Cum realizăm recursivitatea în λ_0, dacă nu avem nume de funcții?
  - Textuală: funcție care se autoapelează, folosindu-și numele;
  - Semantică: ce obiect matematic este desemnat de o funcție recursivă, cu posibilitatea construirii de funcții 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? (expresia pentru *length* nu este închisă!)
- Putem primi ca parametru o funcție echivalentă computațional cu *length*?
  $Length \equiv_{def} \lambda f\ L.(if\ (null?\ L)\ zero\ (succ\ (f\ (cdr\ L))))$
- $(Length\ length) = length \rightarrow$ *length* este un punct fix al lui *Length*!
- Cum obținem punctul fix?

Combinator de punct fix

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

λ

Exemplu

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

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

- $length \equiv_{\text{def}} (Fix\ Length) \sim (Length\ (Fix\ Length)) \sim$
  $\lambda L.(if\ (null?\ L)\ zero\ (succ\ ((Fix\ Length)\ (cdr\ L))))$

- Funcție recursivă, fără a fi textual recursivă!

Racket vs. lambda-0

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

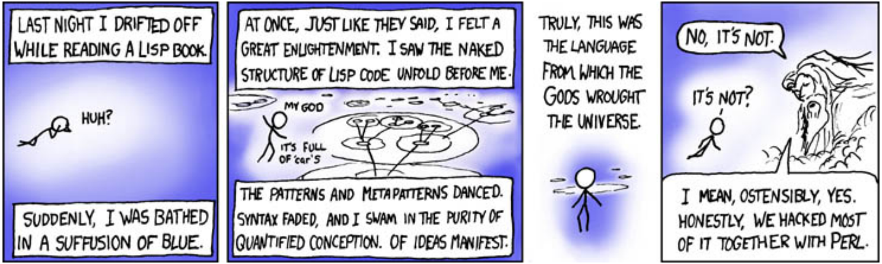
Mai precis

- 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.

Cursul 4: Programare funcțională în Racket

LISP

[(CC) BY-NC xkcd.com] [`https://xkcd.com/224/`]



Introducere

- Gestionarea valorilor
  - modul de tipare al valorilor
  - modul de legare al variabilelor
  - valorile de prim rang
- Gestionarea execuției
  - ordinea de evaluare (generare a valorilor)
  - controlul evaluării
  - modul de construcție al programelor

Legarea variabilelor

Variabile (Nume)

Proprietăți generale ale variabilelor

⁝ Proprietăți

- identificator
- valoarea legată (la un anumit moment)
- domeniul de vizibilitate (*scope*) + durata de viață
- tip

⁝ Stări

- declarată: cunoaștem identificatorul
- definită: cunoaștem și valoarea → variabila a fost *legată*

· în Racket, variabilele (numele) sunt legate *static* prin construcțiile `lambda`, `let`, `let*`, `letrec` și `define`, și sunt vizibile în domeniul construcției unde au fost definite (excepție face `define`).

Legarea variabilelor

Definiții (1)

+ **Legarea variabilelor** – modalitatea de asociere a apariției unei variabile cu definiția acesteia (deci cu valoarea).

+ **Domeniul de vizibilitate** – *scope* – mulțimea punctelor din program unde o definiție (legare) este vizibilă.

+ | Legare statică – Valoarea pentru un nume este legată o singură dată, la declarare, în contextul în care aceasta a fost definită. Valoarea depinde doar de contextul static al variabilei.

- Domeniu de vizibilitate al legării poate fi desprins la compilare.

+ | Legare dinamică – Valorile variabilelor depind de contextul de execuție în care o expresie este evaluată.

Legarea variabilelor în Racket

- Variabile definite în construcții interioare → legate static, local:
  - `lambda`
  - `let`
  - `let*`
  - `letrec`

- Variabile *top-level* → legate static, global:
  - `define`

Construcția `lambda`
Definiție & Exemplu

- Leagă static parametrii formali ai unei funcții
- Sintaxă:

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

- Domeniul de vizibilitate al parametrului `pk`: mulțimea punctelor din `expr` (care este corpul funcției), puncte în care apariția lui `pk` este liberă.

Construcția `lambda`
Semantică

- Aplicație:

```
((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 funcției, `expr`, ținând cont de legările `pk` $\leftarrow$ *valoare*(`ak`).
3. Valoarea aplicației este valoarea lui `expr`, evaluată mai sus.

Constructia `let`

Definiție, Exemplu, Semantică

- Leagă static variabile locale
- Sintaxă:

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

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

Exemplu

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

· Atenție! Construcția `(let ((v1 e1) ...(vn en)) expr)` – echivalentă cu `((lambda (v1 ...vn) expr) e1 ...en)`

Construcția `let*`
Definiție & Exemplu

- Leagă static variabile locale
- Sintaxă:

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

- Scope pentru variabila `vk` = mulțimea punctelor din
  - restul legărilor (legări ulterioare) și
  - corp – `expr`

  în care aparițiile lui `vk` sunt libere.

Ex Exemplu

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

Construcția `let*`
Semantică

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

echivalent cu

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

- Evaluarea expresiilor `ei` se face în ordine!

Construcția `letrec`
Definiție

- Leagă static variabile locale
- Sintaxă:

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

- Domeniul de vizibilitate a variabilei `vk` = mulțimea punctelor din întreaga construcție, în care aparițiile lui `vk` sunt libere.

Construcția `letrec`
Exemplu

Ex | Exemplu

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

Construcția `define`
Definiție & Exemplu

- Leagă static variabile top-level.
- Avantaje:
  - definirea variabilelor *top-level* în orice ordine
  - definirea de funcții mutual recursive

Ex Definiții echivalente:

```
(define f1
    (lambda (x)
        (add1 x)
))

(define (f2 x)
    (add1 x)
))
```

Evaluare

Evaluarea în Racket

- Evaluare aplicativă: evaluarea parametrilor înaintea aplicării funcției asupra acestora (în ordine aleatoare).

- Funcții stricte (i.e.cu evaluare aplicativă)
  - Excepții: `if`, `cond`, `and`, `or`, `quote`.

Controlul evaluării

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

Exemplu

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

Construcția programelor prin recursivitate

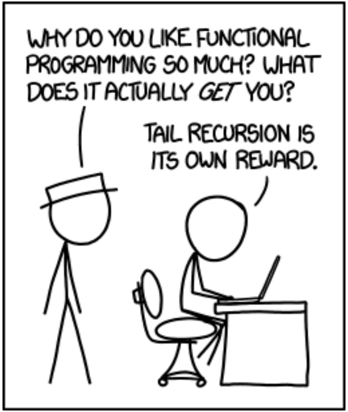
Recursivitate

- Recursivitatea – element fundamental al paradigmei funcționale
  - Numai prin recursivitate (sau iterare) se pot realiza prelucrări pe date de dimensiuni nedefinite.

- Dar, este eficient să folosim recursivitatea?
  - recursivitatea (pe stivă) poate încărca stiva.

Tail recursion

[(CC) BY-NC xkcd.com] [https://xkcd.com/1270/]



Alt text: Functional programming combines the flexibility and power of abstract mathematics with the intuitive clarity of abstract mathematics.

- pe stivă: $factorial(n) = n * factorial(n-1)$
  - timp: liniar
  - spațiu: liniar (ocupat pe stivă)
  - dar, în procedural putem implementa factorialul în spațiu constant.

- pe coadă:
  $factorial(n) = fH(n, 1)$
  $fH(n, p) = fH(n-1, p*n)\ ,\ n > 1\ ;\ p$ altfel
  - timp: liniar
  - spațiu: constant

- beneficiu *tail call optimization*

Discuție despre tipare

Tipuri în Racket

În Racket avem:

- numere: `1`, `2`, `1.5`
- simboli (literali): `'abcd`, `'andrei`
- valori booleene: `#t`, `#f`
- șiruri de caractere: `"șir de caractere"`

- perechi: `(cons 1 2)` → `'(1 . 2)`
- liste: `(cons 1 (cons 2 '()))` → `'(1 2)`

- funcții: `(λ (e f) (cons e f))` → `#<procedure>`

· Cum sunt gestionate tipurilor valorilor (variabilelor) la compilare (verificare) și la execuție?

Modalități de tipare

· Rolul tipurilor: exprimare a intenției programatorului, abstractizare, documentare, optimizare, verificare

+ Tipare – modul de gestionare a tipurilor.

⁝ Clasificare după momentul verificării:

- statică
- dinamică

⁝ Clasificare după rigiditatea regulilor:

- tare
- slabă

Ex Tipare dinamică

Exemplu

Javascript:

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

Ex Tipare statică

Exemplu

Java:

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

Tipare statică vs. dinamică
Caracteristici

: Tipare statică

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

: Tipare dinamică

- La rulare
- Doar valori
- Rulare mai lentă (necesită verificarea tipurilor)
- Flexibilă: sancționează 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.

Exemplu — Tipare tare

`1 + "23"` → Eroare (Haskell, Python)

Exemplu — Tipare slabă

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

`1 + "23" = "123"` (JavaScript)

Tiparea în Racket

- este dinamică

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

- este tare

```
(+ "1" 2) → Eroare
```

- dar, permite liste cu elemente de tipuri diferite.

Cursul 5: Evaluare leneșă în Racket

Întârzierea evaluării

Motivație

De ce? → Luăm un exemplu

Ex Exemplu

Să se implementeze funcția 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 (and (display "y ") y)))))
(test #f)
(test #t)
```

Output: `y 0 | y 30`

- Implementarea nu respectă specificația, deoarece ambii parametri sunt evaluați în momentul aplicării

Varianta 2

Încercare → `quote` & `eval`

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

(define test
  (lambda (x)
    (let ((y 5))
      (prod x (quote (and (display "y␣") y))))))
(test #f)
(test #t)
```

Output: `0 | y undefined`

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

Contexte computaționale
Definiție

+ | Context computațional Contextul computațional al unui punct P, dintr-un program, la momentul t, este mulțimea variabilelor ale căror domenii de vizibilitate îl conțin pe P, la momentul t.

- Legare statică → mulțimea variabilelor care îl conțin pe P în domeniul lexical de vizibilitate

- Legare dinamică → mulțimea variabilelor definite cel mai recent, la momentul t, și referite din P

Exemplu Ce variabile locale conține contextul computațional al punctului *P*?

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

Închideri funcționale
Definiție

+ Închidere funcțională: funcție care își salvează contextul, pe care îl va folosi, în momentul aplicării, pentru evaluarea corpului.

· Notație: închiderea funcției f în contextul $C \rightarrow < f;\ C >$

Ex Exemplu

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

Varianta 3

Încercare → închideri funcționale

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

(define test
  (lambda (x)
    (let ((y 5))
      (prod x
            (lambda () (and (display "y ") y))))))
(test #f)
(test #t)
```

Output: `0 | y y 30`

- Comportament corect: `y` evaluat la cerere (deci leneș)
- `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 (and (display "y ") y))))))
(test #f)
(test #t)
```

Output: `0 | y 30`

- Rezultat corect: `y` evaluat la cerere, o singură dată
  → evaluare leneșă *eficientă*

Promisiuni
Descriere

- Rezultatul încă neevaluat al unei expresii
- Valori de prim rang în limbaj
- `delay`
  - construiește o promisiune;
  - funcție nestrictă.
- `force`
  - forțează respectarea unei promisiuni, evaluând expresia doar la prima aplicare, și salvându-i valoarea;
  - începând cu a doua invocare, întoarce, direct, valoarea memorată.

Promisiuni
Proprietăți

- Salvarea contextului computațional al expresiei a cărei evaluare este întârziată și evaluarea ei ulterioară în acel context → asemănător cu închiderile funcționale.

- Salvarea rezultatului primei evaluări a expresiei.

- Distingerea primei forțări de celelalte → efect lateral, dar acceptabil din moment ce legările se fac static – nu pot exista valori care se schimbă *între timp*.

Evaluare întârziată

Abstractizare a implementării cu promisiuni

Ex Continuare a exemplului cu funcția prod

```
(define-syntax-rule (pack 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 (and (display "y ") y))) )))
```

· utilizarea nu depinde de implementare (am definit funcțiile `pack` și `unpack` care abstractizează implementarea concretă a evaluării întârziate.

Evaluare întârziată

Abstractizare a implementării cu închideri

Ex Continuare a exemplului cu funcția `prod`

```
(define-syntax-rule (pack 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 (and (display "y␣") y))) )))
```

· utilizarea nu depinde de implementare (același cod ca și anterior, altă implementare a funcționalității de evaluare întârziată, acum mai puțin eficientă).

Fluxuri

Motivație
Luăm un exemplu

Ex Determinați suma numerelor pare[1] 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)))))
```

[1] stă pentru o verificare potențial mai complexă, e.g. numere prime

Motivație
Observații

- Varianta 1 – iterativă (d.p.d.v. proces):
  - eficientă, datorită spațiului suplimentar constant;
  - ne-elegantă $\rightarrow$ trebuie să implementăm generarea numerelor.
- Varianta 2 – folosește liste:
  - ineficientă, datorită spațiului posibil mare, ocupat la un moment dat – toate numerele din intervalul $[a, b]$.
  - elegantă și concisă;
- Cum îmbinăm avantajele celor 2 abordări? Putem stoca procesul fără a stoca rezultatul procesului?

  $\rightarrow$ Fluxuri

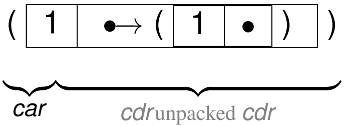
Fluxuri
Caracteristici

- 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 (utilizarea necesită construcția concretă).

Fluxuri

Intuitiv

- o listă este o pereche;
- explorarea listei se face prin operatorii `car` – primul element – și `cdr` – restul listei;
- am dori să generăm `cdr` algoritmic, dar la cerere.



Fluxuri
Operatori: construcție și selecție

- cons, car, cdr, nil, null?

```
(define-syntax-rule (stream-cons 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 – Exemple
Implementarea unui flux de numere 1

- Definiție cu închideri:

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

- Definiție cu fluxuri:

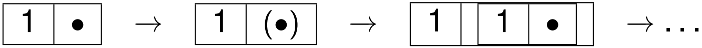
```
(define ones (stream-cons 1 ones))
(stream-take 5 ones) ; (1 1 1 1 1)
```

- Definiție cu promisiuni:

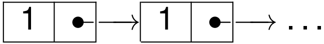
```
(define ones (delay (cons 1 ones)))
```

Fluxuri – Exemple
Flux de numere 1 – discuție

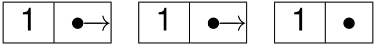
- Ca proces:



- Structural:



- Extinderea se realizează în spațiu constant:



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)))
```

Căutare leneșă în spațiul stărilor

Spațiul stărilor unei probleme

+ Spațiul stărilor unei probleme Mulțimea configurațiilor valide din universul problemei.

Ex Exemplu

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

Stările problemei $\rightarrow$ toate șirurile generabile cu elementele alfabetului respectiv.

Specificarea unei probleme

Aplicație pe Pal_n

- Starea inițială: șirul vid
- Operatorii de generare a stărilor succesor ale unei stări: inserarea unui caracter la începutul unui șir dat
- Operatorul de verificare a proprietății de *scop* a unei stări: palindrom

Căutare în spațiul stărilor

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

Căutare în lățime

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 soluții
- Cum le obținem pe celelalte, mai ales dacă spațiul e infinit?

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

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

Căutare în lățime
Leneșă (2)

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

- Nivel înalt, conceptual: separare între explorarea spațiului și identificarea stărilor *scop*.
- Nivel scăzut, al instrucțiunilor: întrepătrunderea celor două aspecte.
- Aplicații:
  - Palindroame
  - Problema reginelor

Cursul 6: Programare funcțională în Haskell

25 Introducere

26 Sintaxă

27 Evaluare

Haskell

[(CC) BY-NC xkcd.com] [https://xkcd.com/1312/]

Alt text: The problem with Haskell is that it's a language built on lazy evaluation and nobody's actually called for it.

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 Stack
- nume dat după logicianul Haskell Curry;
- aplicații: Pugs, Darcs, Linspire, Xmonad, Cryptol, seL4, Pandoc, web frameworks.

Paralelă între limbaje

Criteriu	Racket	Haskell
Funcții	*Curry* sau *uncurry*	*Curry*
Tipare	Dinamică, tare (-liste)	Statică, tare
Legarea variabilelor	Statică	Statică
Evaluare	Aplicativă	Normală (Leneșă)
Transferul parametrilor	*Call by sharing*	*Call by need*
Efecte laterale	`set!`*	Interzise

Sintaxă

Funcții

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

Ex Exemplu : Definiții echivalente ale funcției `add`:

```haskell
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
```

Funcții vs operatori

- Aplicabilitatea parțială a operatorilor infixați
- Transformări operator → funcție și funcție → operator

Ex Exemplu Definiții echivalente ale funcțiilor `add` și `inc`:

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

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

Pattern matching

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

Exemplu

```
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 proprietățile elementelor, ca într-o specificare matematică

Exemplu

```
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 ..]
```

Evaluare

Evaluare

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

Exemplu

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

Evaluare:

```
f (2 + 3, 3 + 5) (5 + 8)
→ (2 + 3) + (2 + 3)
→ 5 + 5
→ 10
```

reutilizăm rezultatul primei evaluări!

ceilalți parametri nu sunt evaluați

Pași în aplicarea funcțiilor
Exemplu

Exemplu

```
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]
```

Pași în aplicarea funcțiilor
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`).

Pași în aplicarea funcțiilor

Exemplu – revisited

Ex execuția exemplului anterior

`frontInterval 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`	
`→ frontSum xs`	evaluare valoare gardă
`    where`	
`        xs = 3:[4 .. 5]`	`xs` deja calculat
`        → 3:4:[5]`	
`→ frontSum (3:4:[5])`	
`→ 3 + 4 → 7`	

Consecințe

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

Exemplu

```
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))
```

Cursul 7: Tipuri în Haskell

Tipare

Tipuri

Pentru toate valorile (inclusiv funcții)

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

Tipuri
Exemple de valori

Exemplu

```
5              :: Integer
'a'            :: Char
(+1)           :: 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, ...`
- Reprezentare uniformă:

```
    data Integer   =   ... | -2 | -1 | 0 | 1 | 2 | ...
    data Char      =   'a' | 'b' | 'c' | ...
```

Constructori de tip

⇒ tipuri noi pentru valori sau funcții

- Funcții de tip, ce îmbogățesc tipurile din limbaj.

Ex 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)
```

Constructori de tip
Tipurile funcțiilor

- Constructorul -> este asociativ dreapta:

  `Integer -> Integer -> Integer`

  ≡ `Integer -> (Integer -> Integer)`

Exemplu

```haskell
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!
```

Sinteză de tip

Sinteza de tip
Definiție

+ Sinteză de tip – *type inference* – Determinarea automată a tipului unei expresii, pe baza unor reguli precise.

- Adnotările explicite de tip, deși posibile, nenecesare în majoritatea cazurilor
- Dependentă de:
  - componentele expresiei
  - contextul lexical al expresiei
- Reprezentarea tipurilor $\rightarrow$ expresii de tip:
  - constante de tip: tipuri de bază;
  - variabile de tip: pot fi legate la orice expresii de tip;
  - aplicații ale constructorilor de tip pe expresii de tip.

Proprietăți induse de tipuri

+ 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).

+ Conservare Evaluarea unei expresii bine-tipate produce o expresie bine-tipată – de obicei, cu același tip.

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

Exemple de sinteză de tip

Câteva reguli simplificate de sinteză de tip

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

- Funcție: $\dfrac{\texttt{Var :: a} \qquad \texttt{Expr :: b}}{\texttt{\textbackslash Var -> Expr :: a -> b}}$ $\texttt{(TLambda)}$

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

- 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 1

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

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

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

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

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

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

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

Exemple de sinteză de tip

Combinator de punct fix

Exemplul 2

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

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

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

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

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

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

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

Exemple de sinteză de tip

O funcție ne-tipabilă

Exemplul 3

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

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

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

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

Unificare
Definiție

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

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

+ Substituție O substituție este o mulțime de legări variabilă - valoare.

Unificare
Condiții

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

Exemplu

- Pentru a unifica expresiile de tip:
  - `t1 = (a, [b])`
  - `t2 = (Int, c)`
- putem avea substituțiile (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])`

+ *Most general unifier* – MGU Cea mai generală substituție sub care formulele unifică. Exemplu: `S2`.

Tip principal
Exemplu și definiție

Exemplu

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

+ **Tip principal al unei expresii** – Cel mai general tip care descrie complet natura expresiei. Se obține prin utilizarea MGU.

TDA

Constructorul de tip `Natural`
Exemplu de definire TDA 1

Exemplu

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

unu              =   Succ Zero
doi              =   Succ unu

addNat Zero n    =   n
addNat (Succ m) n =  Succ (addNat m n)
```

Constructorul de tip `Natural`

Comentarii

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

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

Constructorul de tip `Pair`

Exemplu de definire TDA 2

Exemplu

```
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
```

Motivație

Polimorfism

+ | Polimorfism parametric Manifestarea aceluiași comportament pentru parametri de tipuri diferite. Exemplu: `id`, `Pair`.

+ | Polimorfism ad-hoc Manifestarea unor comportamente diferite pentru parametri de tipuri diferite. Exemplu: `==`.

Motivație
Exemplu

Ex | Exemplu

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

```haskell
showBool True    =    "True"
showBool False   =    "False"

showChar c       =    "'" ++ [c] ++ "'"

showString s     =    "\"" ++ s ++ "\""
```

- Dorim să implementăm funcția `showNewLine`, care adaugă caracterul “linie nouă” la reprezentarea ca șir:

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

- `showNewLine` nu poate fi polimorfică ⇒ avem nevoie de `showNewLineBool`, `showNewLineChar` etc.
- Alternativ, trimiterea ca parametru a funcției `show*` corespunzătoare:

```
showNewLine sh x = (sh x) ++ "\n"
showNewLineBool = showNewLine showBool
```

- Prea general, fiind posibilă trimiterea unei funcții cu alt comportament, în măsura în care respectă tipul.

Motivație
Cum putem obține un comportament coerent?

- într-un limbaj care suportă supraîncărcarea operatorilor / funcțiilor, aș defini câte o funcție `show` pentru fiecare tip care suportă afișare (cum este `toString` în Java)

- dar cum pot defini în mod coerent tipul lui `showNewLine`?

"`showNewLine` poate primi ca argument orice tip a supraîncărcat funcția `show`."

$\Rightarrow$ Clasa (*mulțimea de tipuri*) `Show`, care necesită implementarea funcției `show`.

Motivație

Varianta 2 – *Supraîncărcarea* funcției → funcție polimorfică ad-hoc

- 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

- 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 Haskell vs. Clase în POO

Haskell

- Tipurile sunt mulțimi de valori;
- Clasele sunt mulțimi de tipuri; tipurile *aderă* la clase;
- Instanțierea claselor de către tipuri pentru ca funcțiile definite în clasă să fie disponibile pentru valorile tipului;
- Operațiile specifice clasei sunt implementate în cadrul declarației de instanțiere.

POO (e.g. Java)

- Clasele sunt mulțimi de obiecte (instanțe);
- Interfețele sunt mulțimi de clase; clasele *implementează* interfețe;
- Implementarea interfețelor de către clase pentru ca funcțiile definite în interfață să fie disponibile pentru instanțele clasei;
- Operațiile specifice interfeței sunt implementate în cadrul definiției clasei.

Clase și instanțe
Definiții

+ Clasa – Mulțime de tipuri ce pot supraîncarca operațiile specifice clasei. Reprezintă o modalitate structurată de control asupra polimorfismului ad-hoc. Exemplu: clasa `Show`, cu operația `show`.

+ Instanță a unei clase – Tip care supraîncarcă operațiile clasei. Exemplu: tipul `Bool` în raport cu clasa `Show`.

- *clasa* definește funcțiile suportate;
- clasa se definește peste o variabilă care stă pentru constructorul unui tip;
- *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 definiții implicite (v. liniile 6–7).
- Necesitatea suprascrierii cel puțin unuia din cei 2 operatori ai clasei `Eq` pentru instanțierea corectă.

Clase predefinite

Ord

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

- contextele – utilzabile și la definirea unei clase.
- clasa `Ord` moștenește clasa `Eq`, cu preluarea operațiilor din clasa moștenită.
- este necesară aderarea la clasa `Eq` în momentul instanțierii clasei `Ord`.
- este suficientă supradefinirea lui `(<=)` la instanțiere.

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.

Aplicații ale claselor

invert
Problemă

Ex invert

Fie constructorii de tip:

```
data Pair a = P a a

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

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

invert
Implementare

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

instance Invertible (Pair a) where
    invert (P x y) = P y x
instance Invertible a => Invertible (NestedList a) where
    invert (Atom x) =   Atom (invert x)
    invert (List x) =   List $ reverse $ map invert x
instance Invertible a => Invertible [a] where
    invert lst = reverse $ map invert lst
instance Invertible Int ...
```

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

Ex `contents`

Să se definească operația `contents`, aplicabilă pe obiecte structurate, inclusiv pe cele aparținând tipurilor `Pair a` și `NestedList a`, care întoarce elementele din componență, 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 [x] where
    contents = id
```

Testăm pentru `contents [1,2,3]`:

- Conform definiției clasei:

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

- Conform supraîncărcării funcției (`id`):

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

- Ecuația `[a] = [[a]]` nu are soluție ⇒ eroare.

`contents`
Varianta 1b

```
class Container a where
    contents :: a -> [b]
instance Container [x] where
    contents = id
```

Testăm pentru `contents [1,2,3]`:

- Conform definiției clasei:

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

- Conform supraîncărcării funcției (`id`):

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

- Ecuația `[a] = [b]` are soluție pentru `a = b`, dar tipul `[a] -> [a]` este insuficient de general (prea specific) în raport cu `[a] -> [b]` ⇒ eroare!

`contents`
Varianta 2

Soluție clasa primește constructorul de tip, și nu tipul container propriu-zis (rezultat după aplicarea constructorului) $\Rightarrow$ includem tipul conținut de container în expresia de tip a funcției `contents`:

```
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

instance Container [] where contents = id
```

Contexte
Câteva exemple

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

 fun2 ::  (Container a, Invertible (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 ::  Invertible 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 `Invertible [a]` la `Invertible a`.
- Simplificarea contextului lui `fun4`, de la `(Eq a, Ord a)` la `Ord a`, din moment ce clasa `Ord` este derivată din clasa `Eq`.

Cursul 9: Concluzie – Paradigma Funcțională

λP.P

Caracteristici ale paradigmelor de programare

- 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 din una sau mai multe paradigme;
  - în general există o paradigmă dominantă;
- Atenție! Paradigma nu are legătură cu sintaxa limbajului!

Paradigma de programare

Legătura cu mașina de calcul

- paradigmele sunt legate teoretic de o mașină de calcul 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

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 a programului;
- modul de construire al tipurilor variabilelor;
- modul de definire și statutul operatorilor – elementele 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 și valori de prim rang

Variable

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).

Funcții ca valori de prim rang

Definiție

λPP

+ Valoare de prim rang – O valoare care poate fi:

- creată dinamic
- stocată într-o variabilă
- trimisă ca parametru unei funcții
- întoarsă dintr-o funcție

Ex Să se scrie funcția `compose`, ce primește ca parametri alte 2 funcții, `f` și `g`, și întoarce funcția obținută prin compunerea lor, $f \circ g$.

Funcții ca valori de prim rang: Compose C

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

- în C, funcțiile nu sunt valori de prim rang;
- pot scrie o funcție care compune două funcții pe o anumită valoare (ca mai sus)
- pot întoarce pointer la o funcție existentă
- dar nu pot crea o referință (pointer) la o funcție nouă, care să fie folosit apoi ca o funcție obișnuită

Funcții ca valori de prim rang:
Java

```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, funcțiile nu sunt valori de prim rang – pot crea rezultatul dar este complicat, și rezultatul nu este o funcție obișnuită, ci un obiect.

Funcții ca valori de prim rang: `Compose`

Racket & Haskell

- Racket:

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

- Haskell:

```
compose = (.)
```

- În Racket și Haskell, funcțiile sunt valori de prim rang.
- mai mult, ele pot fi aplicate parțial, și putem avea funcționale – funcții care iau alte funcții ca parametru.

Tipare a variabilelor

Modalități de tipare

· Rolul tipurilor: exprimare a intenției programatorului, abstractizare, documentare, optimizare, verificare

+ | Tipare – modul de gestionare a tipurilor.

⁝ Clasificare după momentul verificării:

- statică
- dinamică

⁝ Clasificare după rigiditatea regulilor:

- tare
- slabă

Tipare statică vs. dinamică
Exemplu

Ex Tipare dinamică

Exemplu

Javascript:

```
var x = 5;
if(condition) x = "here";
print(x);
```

$\rightarrow$ ce tip are `x` aici?

Ex Tipare statică

Exemplu

Java:

```
int x = 5;
if(condition)
    x = "here";
print(x);
```

`x = "here";` $\rightarrow$ Eroare la *compilare*: `x` este `int`.

Tipare statică vs. dinamică

Caracteristici

: Tipare statică

- La compilare
- Valori și variabile
- Rulare mai rapidă

- Rigidă: sancționează orice construcție
- Debugging mai facil
- Declarații explicite sau inferențe de tip
- Pascal, C, C++, Java, Haskell

: Tipare dinamică

- La rulare
- Doar valori
- Rulare mai lentă (necesită verificarea tipurilor)
- Flexibilă: sancționează 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.

Ex Tipare tare

Exemplu

`1 + "23"` → Eroare (Haskell, Python)

Ex Tipare slabă

Exemplu

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

`1 + "23" = "123"` (JavaScript)

Legarea variabilelor

Legarea variabilelor

Impactul asupra programului

· două posibilități esențiale:

- un nume este întotdeauna legat (într-un anumit context) la aceeași valoare / la același calcul ⇒ numele stă pentru un calcul;
  - legare statică.

- un nume (aceeași variabilă) 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*)

Definiție

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

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

\+ | **Efect lateral** Pe lângă valoarea pe care o produce, o expresie sau o funcție poate modifica starea globală.

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

Efecte laterale (*side effects*)

Consecințe

Ex În expresia `x-- + ++x`, cu `x = 0`:

- evaluarea stânga $\rightarrow$ dreapta produce `0 + 0 = 0`
- evaluarea dreapta $\rightarrow$ stânga produce `1 + 1 = 2`
- dacă înlocuim cele două subexpresii cu valorile pe care le reprezintă, obținem

  `x + (x + 1) = 0 + 1 = 1`

- Importanța ordinii de evaluare!
- Dependențe implicite, puțin lizibile și 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
Pentru expresii

+ Transparență referențială Confundarea unui obiect ("valoare") cu referința la acesta.

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

Exemplu

- `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.)

Transparență referențială

Pentru funcții

+ Funcție transparentă referențial: rezultatul întors depinde exclusiv de parametri.

Exemplu

```
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

În evaluarea aplicativă

Exemplu

```
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 aplicației funcției și transferul unei copii a valorii acestuia
- Modificări locale invizibile la apelant
- C, C++, tipurile primitive Java

Call by sharing

În evaluarea aplicativă

- Variantă a *call by value*;
- Trimiterea unei referințe la obiect;
- Modificări locale asupra referinței invizibile la apelant;
- Modificări locale asupra obiectului referit vizibile la apelant;
- Racket, Java;

Call by reference

În evaluarea aplicativă

- Trimiterea unei referințe la obiect;
- Modificări locale asupra referinței și obiectului referit vizibile la apelant;
- Folosirea “&” în C++.

Call by name

În evaluarea normală

- Argumente neevaluate în momentul aplicării funcției $\rightarrow$ substituție directă (textuală) în corpul funcției;
- Evaluare parametrilor la cerere, de fiecare dată când este nevoie de valoarea acestora;
- în calculul λ.

Call by need
În evaluarea normală

- Variantă a *call by name*;
- Evaluarea unui parametru doar la prima utilizare a acestuia;
- Memorarea valorii unui parametru deja evaluat și returnarea acesteia în cazul utilizării repetate a aceluiași parametru (datorită transparenței referențiale, o aceeași expresie are întotdeauna aceeași valoare) – memoizare;
- în Haskell.

Cursul 10: Logica cu predicate de ordinul I

$P \vee \overline{P}$

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.

Logica propozițională

- 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.

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

$P \vee \overline{P}$

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

+ Interpretare Mulțime de asocieri între fiecare propoziție simplă din limbaj și o valoare de adevăr.

Ex Exemplu

Interpretarea I:

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

Interpretarea J:

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

- cum știu dacă p este adevărat sau fals? Pot ști dacă știu interpretarea – p este doar un *nume* pe care îl dau unei propoziții concrete.

- Sub o interpretare fixată $\rightarrow$ dependența valorii de adevăr a unei propoziții compuse de valorile de adevăr ale celor constituente
- Negație: $(\neg\alpha)^I = \begin{cases} \textit{true} & \text{dacă } \alpha^I = \textit{false} \\ \textit{false} & \text{altfel} \end{cases}$
- Conjuncție: $(\alpha \wedge \beta)^I = \begin{cases} \textit{true} & \text{dacă } \alpha^I = \textit{true} \text{ și } \beta^I = \textit{true} \\ \textit{false} & \text{altfel} \end{cases}$
- Disjuncție: $(\alpha \vee \beta)^I = \begin{cases} \textit{false} & \text{dacă } \alpha^I = \textit{false} \text{ și } \beta^I = \textit{false} \\ \textit{true} & \text{altfel} \end{cases}$

- **Implicație**: $(\alpha \Rightarrow \beta)^I = \begin{cases} \textit{false} & \text{dacă } \alpha^I = \textit{true} \text{ și } \beta^I = \textit{false} \\ \textit{true} & \text{altfel} \end{cases}$
- **Echivalență**:
  $(\alpha \Leftrightarrow \beta)^I = \begin{cases} \textit{true} & \text{dacă } \alpha \Rightarrow \beta \ \wedge \ \beta \Rightarrow \alpha \\ \textit{false} & \text{altfel} \end{cases}$

Evaluarea valorii de adevăr

+ | Evaluare Determinarea valorii de adevăr a unei propoziții, sub o interpretare, prin aplicarea regulilor semantice anterioare.

Exemplu

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

  $\phi^I = (false \wedge true) \vee (true \Rightarrow false) = false \vee false = false$

Valoarea de adevăr în afara interpretării

Satisfiabilitate, Valididate, Nesatisfiabilitate

+ | Satisfiabilitate Proprietatea unei propoziții care este adevărată sub cel puțin o interpretare. Acea interpretare satisface propoziția.

+ | Validitate Proprietatea unei propoziții care este adevărată în toate interpretările. Propoziția se mai numește tautologie.

Ex | Exemplu Propoziția $p \vee \neg p$ este validă.

+ | Nesatisfiabilitate Proprietatea unei propoziții care este falsă în toate interpretările. Propoziția se mai numește contradicție.

Ex | Exemplu Propoziția $p \wedge \neg p$ este nesatisfiabilă.

Valoarea de adevăr în afara interpretării

Metoda tabelei de adevăr

Ex 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*

$\Rightarrow$ Propoziția $(p \wedge q) \vee (q \Rightarrow r)$ este satisfiabilă.

+ | 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 ($\Delta \models \phi$) dacă și numai dacă orice interpretare care satisface toate propozițiile din Δ satisface și ϕ.

Ex Exemplu

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

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

Exemplu

Demonstrăm că $\{p, p \Rightarrow q\} \models q$.

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

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

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

  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

- Creșterea exponențială a numărului de interpretări în raport cu numărul de propoziții simple.
- De aici, diminuarea valorii practice a metodelor semantice, precum cea a tabelei de adevăr.
- Alternativ, metode sintactice, care manipulează doar reprezentarea simbolică.
  - Inferență $\rightarrow$ Derivare mecanică $\rightarrow$ demers de calcul, în scopul verificării derivabilității logice.
  - folosind metodele de inferență, putem construi o mașină de calcul.

+ **Inferența** – Derivarea mecanică a concluziilor unui set de premise.

+ **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 $\Delta \vdash_{inf} \phi$.

Ex **Modus Ponens (MP)** :

$$\frac{\alpha \Rightarrow \beta \quad \alpha}{\beta}$$

Ex **Modus Tollens** :

$$\frac{\alpha \Rightarrow \beta \quad \neg\beta}{\neg\alpha}$$

+ | Consistență (*soundness*) – Regula de inferență determină numai propoziții care sunt, într-adevăr, consecințe logice ale premiselor. $\Delta \vdash_{inf} \phi \Rightarrow \Delta \models \phi$.

+ | Completitudine (*completeness*) – Regula de inferență determină toate consecințele logice ale premiselor. $\Delta \models \phi \Rightarrow \Delta \vdash_{inf} \phi$.

- Ideal, ambele proprietăți – "nici în plus, nici în minus" – $\Delta \models \phi \Leftrightarrow \Delta \vdash_{inf} \phi$
- Incompletitudinea regulii *Modus Ponens*, din imposibilitatea scrierii oricărei propoziții ca implicație.

Logica cu predicate de ordinul întâi

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.”
  - $p \Leftrightarrow q$ – pot ști doar din interpretare.

  $\rightarrow$ Opacitate în raport cu obiectele și relațiile referite.
- FOPL:
  - Generalizare: $prieten(x, y)$: “x este prieten cu y.”
  - $\forall x.\forall y.(prieten(x, y) \Leftrightarrow prieten(y, x))$

  $\rightarrow$ Aplicare pe cazuri particulare.

  $\rightarrow$ Transparență în raport cu obiectele și relațiile referite.

- $+$ | **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** $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$
- $+$ | **Cuantificatori** $\forall$, $\exists$

+ 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.

Ex 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.

Ex | Exemple

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

+ | 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

Exemplu

“Tatăl Ioanei are un prieten deștept”

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

LPOI – Semantică

Interpretarea constă din:

- Un domeniu nevid, D, de concepte (obiecte)
- 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 $p^I : D^n \to \{false, true\}$.

- 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:
  $$(\forall x.\alpha)^I = \begin{cases} \mathit{false} & \text{dacă } \exists d \in D \,.\, \alpha^I_{[d/x]} = \mathit{false} \\ \mathit{true} & \text{altfel} \end{cases}$$

- Cuantificare existențială:
  $$(\exists x.\alpha)^I = \begin{cases} \mathit{true} & \text{dacă } \exists d \in D \,.\, \alpha^I_{[d/x]} = \mathit{true} \\ \mathit{false} & \text{altfel} \end{cases}$$

Exemple cu cuantificatori

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." $\forall x.(viseaza(x, malai) \Rightarrow vrabie(x))$

Cuantificatori
Greșeli frecvente

- $\forall x.(vrabie(x) \Rightarrow viseaza(x, malai))$
  $\rightarrow$ corect: "Toate vrăbiile visează mălai."
- $\forall x.(vrabie(x) \wedge viseaza(x, malai))$
  $\rightarrow$ greșit: "Toți sunt vrăbii și toți visează mălai."
- $\exists x.(vrabie(x) \wedge viseaza(x, malai))$
  $\rightarrow$ corect: "Unele vrăbii visează mălai."
- $\exists x.(vrabie(x) \Rightarrow viseaza(x, malai))$
  $\rightarrow$ greșit: probabil nu are semnificația pe care o intenționăm. Este adevărată și dacă luăm un x care nu este vrabie (fals implică orice).

- Necomutativitate:
  - $\forall x.\exists y.viseaza(x, y) \rightarrow$ "Toți visează la câte ceva."
  - $\exists y.\forall x.viseaza(x, y) \rightarrow$ "Există ceva la care visează toată lumea."

- 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

+ | Literal – Atom sau negația unui atom.

Ⓔx | Exemplu $prieten(x, y)$, $\neg prieten(x, y)$.

+ | Clauză – Mulțime de literali dintr-o expresie clauzală.

Ⓔx | Exemplu $\{prieten(x, y), \neg doctor(x)\}$.

+ | Forma normală conjunctivă – FNC – Reprezentare ca mulțime de clauze, cu semnificație conjunctivă.

+ | Forma normală implicativă – FNI – Reprezentare ca mulțime de clauze cu clauzele în forma grupată
$\{\neg A_1, \ldots, \neg A_m, B_1, \ldots, B_n\}, \Leftrightarrow \ (A_1 \wedge \ldots \wedge A_m) \Rightarrow (B_1 \vee \ldots \vee B_n)$

+ Clauză Horn – Clauză în care cel mult un literal este în formă pozitivă:
$\{\neg A_1, \ldots, \neg A_n, A\}$,
corespunzătoare implicației
$A_1 \wedge \ldots \wedge A_n \Rightarrow A$.

Ex Exemplu Transformarea propoziției
$\forall x.vrabie(x) \vee ciocarlie(x) \Rightarrow pasare(x)$ în formă normală, utilizând clauze Horn:
FNC: $\{\neg vrabie(x), pasare(x)\}, \{\neg ciocarlie(x), pasare(x)\}$

Conversia propozițiilor în FNC (1)

Eliminare implicații, împingere negații, redenumiri

$P \vee \overline{P}$

1. Eliminarea implicațiilor ($\not\Rightarrow$)

2. Împingerea negațiilor până în fața atomilor ($\overrightarrow{\neg}$)

3. Redenumirea variabilelor cuantificate pentru obținerea unicității de nume (R):

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

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

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

Conversia propozițiilor în FNC (2)

Skolemizare

5. Eliminarea cuantificatorilor existențiali (skolemizare) (S):
   - Dacă nu este precedat de cuantificatori universali: înlocuirea aparițiilor variabilei cuantificate printr-o constantă (bine aleasă):

$$\exists x.p(x) \rightarrow p(c_x)$$

   - Dacă este precedat de cuantificatori universali: înlocuirea aparițiilor variabilei cuantificate prin aplicația unei funcții unice asupra variabilelor anterior cuantificate universal:

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

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

6. Eliminarea cuantificatorilor universali, considerați, acum, impliciți ($\cancel{\forall}$):

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

7. Distribuirea lui $\vee$ față de $\wedge$ ($\vee/\wedge$):

$$\alpha \vee (\beta \wedge \gamma) \rightarrow (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$$

8. Transformarea expresiilor în clauze (C).

Conversia propozițiilor în FNC – Exemplu

Exemplu "Cine rezolvă toate laboratoarele este apreciat de cineva."

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

$$\not\Rightarrow \quad \forall x.(\neg\forall y.(\neg lab(y) \vee rezolva(x,y)) \vee \exists y.apreciaza(y,x))$$

$$\overrightarrow{\neg} \quad \forall x.(\exists y.\neg(\neg lab(y) \vee rezolva(x,y)) \vee \exists y.apreciaza(y,x))$$

$$\overrightarrow{\neg} \quad \forall x.(\exists y.(lab(y) \wedge \neg rezolva(x,y)) \vee \exists y.apreciaza(y,x))$$

$$\text{R} \quad \forall x.(\exists y.(lab(y) \wedge \neg rezolva(x,y)) \vee \exists z.apreciaza(z,x))$$

$$\text{P} \quad \forall x.\exists y.\exists z.((lab(y) \wedge \neg rezolva(x,y)) \vee apreciaza(z,x))$$

$$\text{S} \quad \forall x.((lab(f_y(x)) \wedge \neg rezolva(x,f_y(x))) \vee apreciaza(f_z(x),x))$$

$$\not\forall \quad (lab(f_y(x)) \wedge \neg rezolva(x,f_y(x))) \vee apreciaza(f_z(x),x)$$

$$\vee/\wedge \quad (lab(f_y(x)) \vee apr(f_z(x),x)) \wedge (\neg rez(x,f_y(x)) \vee apr(f_z(x),x))$$

$$\text{C} \quad \{lab(f_y(x)), apr(f_z(x),x)\}, \{\neg rez(x,f_y(x)), apr(f_z(x),x)\}$$

Unificare și rezoluție

- Pasul de rezoluție: 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 (clauze):
  - propoziție = mulțime de clauze (semnificație conjunctivă)
  - clauză = mulțime de literali (semnificație disjunctivă)
  - literal = atom sau atom negat
  - atom = propoziție simplă

Ideea (în LP):

$$\frac{\begin{array}{l}\{p \Rightarrow q\}\\ \{\neg p \Rightarrow r\}\end{array}}{\{q, r\}} \qquad \rightarrow \text{“Anularea” lui } p$$

- 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:

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

Rezoluție
Cazuri speciale

- Clauza vidă $\rightarrow$ indicator de contradicție între premise

$$\frac{\{\neg p\} \quad \{p\}}{\{\} = \emptyset}$$

- Mai mult de 2 rezolvenți posibili $\rightarrow$ se alege doar unul:

$$\frac{\{p, q\} \quad \{\neg p, \neg q\}}{\{p, \neg p\} \text{ sau } \{q, \neg q\}}$$

- Demonstrarea nesatisfiabilității $\rightarrow$ derivarea clauzei vide.

- Demonstrarea derivabilității concluziei ϕ din premisele $\phi_1, \ldots, \phi_n \rightarrow$ demonstrarea nesatisfiabilității propoziției $\phi_1 \wedge \ldots \wedge \phi_n \wedge \neg\phi$.

- Demonstrarea validității propoziției $\phi \rightarrow$ demonstrarea nesatisfiabilității propoziției $\neg\phi$.

Exemplu

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 $\rightarrow$ clauza vidă

$P \vee \overline{P}$

T | Teorema 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 $\rightarrow$ număr finit de concluzii.

Unificare

- Utilizată pentru rezoluția în LPOI
- vezi și sinteza de tip în Haskell

cum știm dacă folosind ipoteza $om(Marcel)$ și propoziția $\forall x.om(x) \Rightarrow are\_inima(x)$ putem demonstra că $are\_inima(Marcel)$ $\rightarrow$ unificând $om(Marcel)$ și $\forall om(x)$.

- reguli:
  - o propoziție unifică cu o propoziție de aceeași formă
  - două predicate unifică dacă au același nume și parametri care unifică (om cu om, x cu $Marcel$)
  - o constantă unifică cu o constantă cu același nume
  - o variabilă unifică cu un termen ce nu conține variabila (x cu $Marcel$)

Unificare
Observații

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

- Rezoluția pentru clauze Horn:

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

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

Exemplu

Horses and hounds

1. Horses are faster than dogs.
2. There is a greyhound that is faster than any rabbit.
3. Harry is a horse and Ralph is a rabbit.
4. Is Harry faster than Ralph?

Rezoluție

Exemplu Horses and Hounds

1. $\forall x.\forall y.horse(x) \wedge dog(y) \Rightarrow faster(x, y) \rightarrow \neg horse(x) \vee \neg dog(y) \vee faster(x, y)$
2. $\exists x.greyhound(x) \wedge (\forall y.rabbit(y) \Rightarrow faster(x, y))$
   $\rightarrow greyhound(Greg) \quad ; \quad \neg rabbit(y) \vee faster(Greg, y)$
3. $horse(Harry) \quad ; \quad rabbit(Ralph)$
4. $\neg faster(Harry, Ralph)$ (concluzia negată)
5. $\neg greyhound(x) \vee dog(x)$ (common knowledge)
6. $\neg faster(x, y) \vee \neg faster(y, z) \vee faster(x, z)$ (tranzitivitate)

7. 1 + 3a $\rightarrow \neg dog(y) \vee faster(Harry, y)$ (cu $\{Harry/x\}$)
8. 2a + 5 $\rightarrow dog(Greg)$ (cu $\{Greg/x\}$)
9. 7 + 8 $\rightarrow faster(Harry, Greg)$ (cu $\{Greg/y\}$)
10. 2b + 3b $\rightarrow faster(Greg, Ralph)$ (cu $\{Ralph/y\}$)
11. 6 + 9 + 10 $\rightarrow faster(Harry, Ralph)$ $\{Harry/x, Greg/y, Ralph/z\}$
12. 11 + 4 $\rightarrow \square$ q.e.d.

Cursul 11: Introducere în Prolog

Introducere în Prolog

Prolog
Limbaj de programare logică

- introdus în anii 1970 ;
- programul $\rightarrow$ 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]

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

Procesul de demonstrare

Pași în demonstrare (1)

1. Inițializarea stivei de scopuri cu scopul solicitat;
2. Inițializarea substituției (utilizate pe parcursul unificării) cu mulțimea vidă;
3. Extragerea scopului din vârful stivei și determinarea primei clauze din program cu a cărei concluzie unifică;
4. Îmbogățirea corespunzătoare a substituției și adăugarea premiselor clauzei în stivă, în ordinea din program;
5. Salt la pasul 3.

Pași în demonstrare (2)

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

Un exemplu de program Prolog

Exemplu

```
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)

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

$\downarrow$

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

$\downarrow$

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

$\downarrow$	$\downarrow$	$\downarrow$
`p(andrei, bogdan)`	`p(andrei, bianca)`	`p(bogdan, cristi)`
$\downarrow$	$\downarrow$	$\downarrow$
$\cdots 1$	$\cdots 2$	$\cdots 3$

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

`...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

Observații

- Ordinea evaluării / încercării demonstrării scopurilor
  - Ordinea clauzelor în program;
  - Ordinea premiselor în cadrul regulilor.

- Recomandare: premisele mai ușor de satisfăcut și mai specifice primele – exemplu: axiome.

Strategii de control
Ale demonstrațiilor

Forward chaining (data-driven)

- Derivarea tuturor concluziilor, pornind de la datele inițiale;
- Oprire la obținerea 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 ș.a.m.d.

Strategii de control

Algoritm Backward chaining

```
1.  BackwardChaining(rules, goals, subst)
     lista regulilor din program, stiva de scopuri, substituția
     curentă, inițial vidă.
     returns satisfiabilitatea scopurilor
2.  if goals = ∅ then
3.     return SUCCESS
4.  goal ← head(goals)
5.  goals ← tail(goals)
6.  for-each rule ∈ rules do        // în ordinea din program
7.     if unify(goal, conclusion(rule), subst) → bindings
8.        newGoals ← premises(rule) ∪ goals       // adâncime
9.        newSubst ← subst ∪ bindings
10.       if BackwardChaining(rules, newGoals, newSubst)
11.          then return SUCCESS
12. return FAILURE
```

Controlul execuției

Exemplu – Minimul a două numere
Cod Prolog

Ex 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` $\rightarrow$ cum putem elimina redundanța?

Exemplu

```
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.

Ex | Exemplu

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

Exemplu

```
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

Exemplu

```
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.