

Racket: Introducere

- Responsabil: Mihnea Muraru [mailto:mmihnea@gmail.com]
- Data publicării: 14.02.2020
- Data ultimei modificări: 14.02.2020

Obiective

Scopul acestui laborator este introducerea în **programarea funcțională** și prezentarea elementelor de bază ale limbajului **Racket**.

Aspectele urmărite sunt:

- particularitățile paradigmei **funcționale** în raport cu celelalte paradigme de programare studiate: calculul modelat ca o compunere de funcții, fără secvențe de instrucțiuni, fără cicluri, fără atribuiri (și fără efecte laterale în general)
- programarea în **Racket**: primitivele limbajului, definirea de noi expresii și funcții
- **modelul de evaluare** a expresiilor în Racket: modelul substituției + evaluare aplicativă
- **legătura** TDA (tipuri de date abstracte) - recursivitate - programare funcțională - demonstrații de corectitudine

Particularități ale paradigmelor de programare studiate

Există moduri diferite de a programa un calculator pentru a rezolva o anumită problemă. Vom înțelege prin paradigmă o școală de gândire referitor la organizarea procesului de calcul într-un limbaj de programare.

Vom relua, în ordinea în care au fost studiate, cele două paradigme de programare întâlnite până acum:

- paradigma **procedurală**
- paradigma **orientată obiect**

și vom trece în revistă caracteristicile acestora urmând să prezentăm apoi paradigma **funcțională**.

Programarea procedurală

În programarea procedurală, elementul de bază este **procedura**. Programul constă într-o succesiune de apeluri de proceduri (fie că sunt primitive ale limbajului, fie că sunt definite de programator, caz în care pot conține la rândul lor alte apeluri de proceduri, ș.a.m.d.).

Observăm:

- caracterul **imperativ** al apelurilor: *inițializează(!)*, *calculează(!)*, *dealocă(!)*
- calculele se realizează folosind **efecte laterale** (*side-effects*): Spunem că o procedură are efecte laterale dacă nu doar întoarce un rezultat, ci și modifică starea unor entități din exteriorul ei, de exemplu valorile unor variabile sau structuri reținute în memorie.

Programarea orientată obiect

Programarea orientată obiect mută centrul de interes de la procedură (secvență de prelucrare) la structura de date prelucrată. Elementul de bază este **obiectul**, și fiecare obiect încapsulează metode specifice prin care poate fi modificat/prelucrat.

Și în programarea orientată obiect prelucrarea se bazează pe calcule cu **efecte laterale**: o metodă modifică, de regulă, starea obiectului pe care a fost apelată.

Programarea funcțională

Una dintre principalele diferențe aduse de programarea funcțională este absența **efectelor laterale**, și se datorează faptului că programarea funcțională este atemporală. Nu există atribuiri, nu există secvență de comenzi, o anumită expresie are o singură valoare pe tot parcursul programului. Elementul central este **funcția** (văzută însă nu în sens procedural, ci mai degrabă în sens matematic). Programele constau în compuneri și aplicări de funcții. Exemplu (limbajul Haskell):

```
insertion_sort [] = []
insertion_sort (x:xs) = insert x (insertion_sort xs)

insert y [] = [y]
insert y (x:xs) = if y < x then (y:x:xs) else x:(insert y xs)
```

Observați asemănarea izbitoare între codul Haskell și **definirea axiomelor unui TDA**, studiată la cursul de *Analiza Algoritmilor* – funcțiile sunt definite pe cazurile de aplicare și folosesc recursivitatea pentru a referi un caz deja implementat.

Funcția `insertion_sort` primește o listă ca parametru, și o sortează prin inserție. Dacă parametrul lui `insertion_sort` este lista vidă, atunci funcția va întoarce lista vidă (care este sortată trivial). Altfel, `insertion_sort` sortează recursiv sublista `xs`, apoi inserează pe poziția corespunzătoare în această listă (sortată) elementul `x`.

Funcția `insert` primește doi parametri:

- un element
- o listă sortată.

Dacă lista primită ca parametru este vidă, `insert` întoarce o listă cu un singur element (primul parametru). Altfel, introduce elementul `y` în listă, astfel încât sortarea să se conserve.

Exemplul de mai sus este sugestiv pentru modul de construcție a programelor funcționale: rezultatul final se obține din rezultate intermediare, prin **compuneri și aplicări** de funcții. Definind funcția `insertion_sort` pe o listă nevidă, am observat că avem nevoie întâi să sortăm recursiv lista fără primul element, apoi să inserăm primul element „la locul lui” în lista sortată. Programarea funcțională **nu** ne permite secvențe de instrucțiuni de tipul „întâi fă asta, apoi fă cealaltă”, așa că am reformulat secvența de comenzi într-o secvență de aplicări de funcții: „inserează primul element în rezultatul obținut prin sortarea restului”. Nu este nicio problemă că încă nu avem o funcție care inserează un element într-o listă sortată; de fiecare dată când avem nevoie de un rezultat încă necalculat, ne putem **imagina** (*wishful thinking*) că avem deja o funcție care realizează calculul respectiv și putem apela acea funcție, urmând să o implementăm ulterior. Exact așa am procedat cu funcția `insert`. Această abordare ne face să spunem că programarea funcțională este de tip **wishful thinking** [https://en.wikipedia.org/wiki/Wishful_thinking].

Observați în exemplul de mai sus și **absența** efectelor laterale. Niciuna dintre funcții nu modifică zone de memorie din afara acesteia.

Racket

Racket este un limbaj:

- derivat din **Lisp** [[https://en.wikipedia.org/wiki/Lisp_\(programming_language\)](https://en.wikipedia.org/wiki/Lisp_(programming_language))] – sintaxă pe bază de paranteze.
- tipat dinamic – tipul valorilor nu este verificat la compilare.
- tipat tare (strong) – valorile nu se convertesc automat la tipul cerut de situație.
- cu evaluare aplicativă – argumentele funcțiilor sunt evaluate înainte de aplicarea funcției.
- multiparadigmă (suportă programarea funcțională dar are și constructe imperative, pe care noi nu le vom folosi)
- înrudit cu Scheme [<http://www.racket-lang.org/new-name.html>]

În centrul limbajului Racket se află **evaluarea expresiilor** constând în **aplicări de funcții**. Fie următoarea aplicare a funcției `max`, în **C**:

```
int result;
result = max (3,4);
```

Comparativ, iată aplicarea funcției `max` în Racket:

```
(max 3 4)
```

În **C**, apelul de funcție se realizează direct prin nume; acesta este urmat de paranteze, iar între paranteze sunt enumerați parametrii. În **Racket**, paranteza deschisă indică exclusiv faptul că urmează o **aplicare de funcție**. Între paranteze se află numele funcției, urmat de parametri. Următoarea construcție:

```
(max (+ 2 3) 4)
```

se interpretează astfel:

- evaluează aplicarea funcției `max`, care primește doi parametri
- primul parametru reprezintă o altă aplicare a funcției `+` cu parametrii 2 și 3
- aplicarea funcției `+` se evaluează la 5
- funcția `max` determină maximumul între două numere, iar aplicarea ei pe 5 și 4 se evaluează la 5.

Racket este un limbaj în care argumentele sunt transmise funcției prin valoare (**call-by-value**), astfel că prima expresie evaluată în exemplul de mai sus este `(+ 2 3)`.

Următoarea construcție:

```
(max (3) 4)
```

este **invalidă**. **Racket** va interpreta `(3)` ca pe o tentativă de a aplica funcția cu numele 3 pe zero parametri. Codul va genera eroare.

Exercițiu: încercați să priviți orice construcție din limbajul Racket ca pe o funcție. De exemplu, `if`:

```
(if (= 2 3) 2 (max 2 3))
```

Putem observa faptul că `if` se comportă ca o funcție cu trei parametri:

- primul parametru este **condiția**. Condiția reprezintă o aplicare de funcție, care întoarce (se evaluează la) `true` sau `false`
- al doilea parametru reprezintă expresia de evaluat când condiția este `true`
- al treilea parametru reprezintă expresia de evaluat când condiția este `false`

Cum 2 este diferit de 3, codul de mai sus va întoarce al treilea parametru (evaluarea expresiei `(max 2 3)`).

Tipuri de date

Următoarele tipuri de date sunt uzuale în **Racket** (cu **bold** tipurile pe care le vom folosi mai mult:

- Tipuri de date **simple**
 - **Boolean** – `#t` sau `#f`
 - **Număr**
 - **Simbol** (literal) – `'a`, `'b`, `'ceva`, `'un-simbol-din-mai-multe-cuvinte`, ...
 - Caracter
- Tipuri de date **compuse**
 - **Pereche**
 - **Listă**
 - Șir de caractere (String)
 - Vector

Simboluri

Simbolurile (numite și literal) sunt valori formate din unul sau mai multe caractere, fără spațiu. Diferențierea dintre un nume (care este legat la o valoare, similar unei variabile din limbajele imperative) și un simbol se face atașând în fața valorii simbolului un apostrof: `'simbol`.

Atenție! Apostroful în fața unui simbol (sau, vedem mai jos, a unei expresii în paranteză) este un operator, echivalent cu funcția `quote`, care determină ca simbolul sau expresia care îi urmează să nu fie evaluată. Astfel, `'simbol` se evaluează la un simbol și nu se încearcă evaluarea unei variabile cu numele `simbol` și găsirea unei valori asociate acestui nume.

Perechi

O pereche este un tuplu de două elemente, care pot avea tipuri diferite. Pentru manipularea perechilor, Racket ne pune la dispoziție un constructor (`cons`) și doi selectori (`car` și `cdr`). Utilizarea acestora este demonstrată în exemplele de mai jos:

```
(cons 1 2) ; construiește perechea (1 . 2)
```

```
(car (cons 1 2)) ; întoarce primul element din pereche, adică 1
(cdr (cons 1 2)) ; întoarce al doilea element din pereche, adică 2
```

```
(cons 3 (cons 1 2)) ; construiește PERECHEA (3 . (1 . 2)) (primul element al perechii este un număr, al doilea este o pereche)
```

```
(cdr (cons 3 (cons 1 2))) ; întoarce perechea (1 . 2)
```

Liste

Denumirea „Lisp” a limbajului părinte al Racket-ului provine de la „List Processing”, și într-adevăr lista este o structură de bază în cele două limbaje. Mulțumită faptului că se pot construi perechi eterogene (între elemente de tipuri diferite, de exemplu între un element și o listă), Racket implementează orice listă nevidă ca pe o pereche între primul element și restul listei. Astfel, tipul *listă împrumută* de la tipul *pereche* constructorul `cons` și selectorii `car` și `cdr`, la care se adaugă constructorul `null` pentru **lista vidă**. Exemple:

```
(cons 1 null) ; lista formată din elementul 1. Pentru ușurința citirii va fi afișată ca (1) și nu (1 . null), dar reprezintă același lucru
(cons 1 (cons 2 (cons 3 null))) ; lista (1 2 3)
(cons 'a (cons 'b (cons 'c '()))) ; lista (a b c); Atenție, lista vidă se poate reprezenta ca '() sau null;
```

Funcția `list` construiește o listă nouă care va conține elementele date ca argumente funcției:

```
(list 1 2 3 4)
```

Astfel, putem construi lista `(1 2 3 4)` fie folosind apostroful – `'(1 2 3 4)` – fie folosind funcția `list` – `(list 1 2 3 4)`, dar apostroful nu poate substitui oricând funcția `list`, după cum se observă în exemplele de mai jos:

```
(list 1 2 (+ 2 3)) ; se evaluează la lista (1 2 5)
'(1 2 (+ 2 3)) ; se evaluează la lista (1 2 (+ 2 3)), pentru că întreaga expresie de după apostrof nu se evaluează.
```

Operatori pe liste

```
(car '(1 2 3 4)) ; întoarce 1, adică primul element din perechea formată din elementul 1 și lista (2 3 4)
(cdr '(1 2 3 4)) ; întoarce (2 3 4), adică al doilea element din perechea formată din elementul 1 și lista (2 3 4)
```

Funcțiile `car` și `cdr` pot fi compuse pentru a obține diverse elemente ale listei. Exemple:

```
(car (cdr '(1 2 3 4 5))) ; întoarce 2
(cdr (car '(1 2 3 4 5))) ; cum (car list) nu întoarce o listă, ci un element, apelul produce eroare; funcția cdr așteaptă liste ca parametru
(cdr (cdr '(1 2 3 4 5))) ; întoarce (3 4 5)
(car (cdr (cdr '(1 2 3 4 5)))) ; întoarce 3
```

Racket permite forme prescurtate pentru compuneri de funcții de tip `car` și `cdr`. Rescriem exemplele de mai sus folosind aceste forme prescurtate:

```
(cadr '(1 2 3 4 5)) ; întoarce 2
(cdar '(1 2 3 4 5)) ; produce eroare
(cddr '(1 2 3 4 5)) ; întoarce (3 4 5)
(caddr '(1 2 3 4 5)) ; întoarce 3
```

Alte funcții utile pentru manipularea listelor:

```
(append '(1 2 3) '(4) '(5 6)) ; întoarce lista (1 2 3 4 5 6), din concatenarea tuturor listelor primite ca argumente
(null? '()) ; întoarce #t (adică true), întrucât lista primită ca argument este vidă
(null? '(1 2)) ; întoarce #f (adică false), întrucât lista primită ca argument este nevidă
(length '(1 2 3 4)) ; întoarce 4 (lungimea listei primită ca argument)
(length '(1 (2 3) 4)) ; întoarce 3 (lungimea listei primită ca argument)
(reverse '(1 (2 3) 4)) ; întoarce (4 (2 3) 1), adică lista primită ca argument - cu elementele în ordine inversă
(list? '()) ; întoarce #t, întrucât argumentul este o listă
(list? 2) ; întoarce #f, întrucât argumentul nu este o listă
```

Legarea variabilelor

Un identificator poate fi legat la o valoare folosind (printre altele) construcția (`define` `identificator` `valoare`). Efectul `define`-ului este de a permite referirea unei expresii (adesea complexă) cu ajutorul unui nume concis, nu acela de a atribui o valoare unei variabile. În urma `define`-urilor **nu** se suprascrisu valori la anumite locații din memorie. Într-un program Racket nu se poate face `define` de mai multe ori la același simbol).

```
(define x 2) ; x devine identificator pentru 2
(define y (+ x 2)) ; y devine identificator pentru 4, întrucât x este doar un alt nume pentru valoarea 2
(define my_list '(a 2 3)) ; my_list identifică lista (a 2 3)
```

```
(car my_list) ; întoarce a
(+ (cadr my_list) y) ; întoarce suma dintre al doilea element din lista my_list și y, deci 2 + 4 = 6
```

Funcții anonime (lambda)

O funcție anonimă se definește utilizând cuvântul cheie `lambda`. Sintaxa este: (`lambda` (`arg1 arg2 ...`) `ce_întoarce_funcția`).

```
(lambda (x) x) ; funcția identitate
; pentru a aplica această funcție procedăm în felul următor:
((lambda (x) x) 2) ; întoarce 2; se respectă sintaxa cunoscută (funcție arg1 arg2 ...)

(lambda (x y) (+ x y)) ; funcție care calculează suma a doi termeni x și y
(lambda (l1 l2) (append l2 l1)) ; funcție care concatenează listele l2 și l1, începând cu l2
```

Este destul de neplăcut să rescriem o funcție anonimă pentru a o utiliza în mai multe locuri. Folosim `define` când dorim să legăm un identificator la o funcție anonimă (în aparență funcția nu mai este anonimă).

```
(define identitate (lambda (x) x))
(identitate 3) ; întoarce 3
```

Limbajul ne permite să condensăm definirea unei funcții cu legarea ei la un nume, scriind ca (`define` (`nume-funcție` `arg1 arg2 ...`) `ce_întoarce_funcția`):

```
(define (identitate x) x)
(identitate 3) ; întoarce 3

(define append2 (lambda (l1 l2) (append l2 l1)))
(append2 '(1 2 3) '(4 5 6)) ; întoarce lista (4 5 6 1 2 3)
;fără 'lambda'
(define (append2 l1 l2) (append l2 l1))
(append2 '(1 2 3) '(4 5 6)) ; întoarce lista (4 5 6 1 2 3)
```

Putem oricând scrie `λ` în loc de `lambda` (folosind `Ctrl+\\`).

Funcții utile

Operatori:

- aritmetici: +, -, *, /, modulo, quotient
- relaționali: <, <=, >, >=, eq?, equal?
- logici: not, and, or

```
(modulo 5 2) ; 1, restul împărțirii lui 5 la 2
(quotient 5 2) ; 2, împărțire întreagă
```

```
(< 3 2) ; #f
(>= 3 2) ; #t
(= 1 1) ; #t, verifică egalitatea între numere
(= '(1 2) '(1 2)) ; eroare
(equal? '(1 2) '(1 2)) ; #t, verifică egalitatea între valori
(eq? '(1 2 3) '(1 2 3)) ; #f, asemănător cu "==" din Java, verifică dacă două obiecte referă aceeași zonă de memorie
```

```
(define x '(1 2 3))
(eq? x x) ; #t
```

Expresii condiționale:

- if
- cond

```
; (if testexp thenexp elseexp) ; sau fără bucata de else
(if (< a 0)
  a ; întoarce a dacă a este negativ
  (if (> a 10)
    (* a a) ; întoarce a * a dacă a este mai mare decât 10
    0)) ; întoarce 0 altfel

; (cond (test1 exp1) (...) ... (else exp...) ) ; sau fără bucata de else
(cond
  ((< a 0) a) ; întoarce a dacă a este negativ
  ((> a 10) (* a a)) ; întoarce a * a dacă a este mai mare decât 10
  (else 0)) ; întoarce 0 altfel
```

Cum trebuie gândit un program funcțional

Deși Racket este un limbaj multi-paradigmă, veți folosi Racket pentru a învăța să programați în spiritul programării funcționale. Sintetizăm mai jos acest spirit și modul în care puteți suplini lipsa „uneltelor” cu care v-ați obișnuit în celelalte paradigme:

- programarea este de tip **wishful thinking**
- secvența de instrucțiuni devine **compunere de funcții** (secvență de aplicări de funcții)
- lipsesc atribuiri (variabilă = valoare) și instrucțiunile de ciclare precum `for` sau `while`, dar puteți obține același efect folosind **funcții recursive**. În loc de ciclare și atribuiri (adică în loc să ținem starea curentă a problemei în variabile), vom folosi recursivitate și starea curentă a problemei se va pasa ca parametru în funcțiile recursive. Citiți și recitiți acest paragraf în tandem cu exemplele de mai jos.
- scrierea funcțiilor recursive derivă direct din **scrierea axiomelor** TDA-urilor implicate în problemă

Exemplul 1: o funcție care calculează factorialul unui număr natural n . Știm că TDA-ul Natural are doi constructori de bază, 0 și succ . Scriem axiomele operatorului factorial:

```
factorial(0) = 1
factorial(succ(n)) = succ(n) * factorial(n)
```

Traducem întocmai aceste axiome în cod Racket:

```
(define (factorial n) ; identificadorul factorial primește un parametru și anume numărul natural n
  (if (= n 0) ; cazul de bază, n=0, corespunzător primei axiome
    1 ; în acest caz, valoarea întoarsă este 1
    (* n (factorial (- n 1))))) ; altfel, rezultatul este n * factorial(n - 1), corespunzător celei de-a doua axiome

; observați că nu reținem în nicio variabilă n-ul la care am ajuns, dar el este trimis ca parametru dintr-un apel recursiv în altul

(factorial 0) ; întoarce 1
(factorial 1) ; întoarce 1
(factorial 2) ; întoarce 2
(factorial 3) ; întoarce 6
(factorial -1) ; intră în buclă infinită
```

Exemplul 2: o funcție care calculează suma elementelor dintr-o listă L . Știm că TDA-ul List are doi constructori de bază, null și cons . Scriem axiomele operatorului sum-list:

```
sum-list(null) = 0
sum-list(cons(a,L)) = a + sum-list(L)
```

Trecem axiomele în cod Racket:

```
(define (sum-list L); identificatorul sum-list care primește un parametru, lista L
  (if (null? L) ; dacă L este vidă
      0 ; întoarce 0, cazul corespunzător primei axiome
      (+ (car L) (sum-list (cdr L))))) ; altfel întoarce primul element + sum-list(restul listei), corespunzător celei de-a doua axiome

(sum-list '(1 2 3)) ; întoarce 6
(sum-list 1) ; eroare
```

Resurse

- [Exercitii](#)
- [Soluții](#)
- [Cheatsheet](#)

Referințe

- Documentație Racket [<http://docs.racket-lang.org/reference/index.html>], în special funcțiile pentru valori booleene [<http://docs.racket-lang.org/reference/booleans.html>], numere [<http://docs.racket-lang.org/reference/numbers.html>] și perechi/liste [<http://docs.racket-lang.org/reference/pairs.html>]
- Programare funcțională [https://en.wikipedia.org/wiki/Functional_programming]
- Recursivitate [<https://en.wikipedia.org/wiki/Recursive>]
- Lisp [https://en.wikipedia.org/wiki/Lisp_programming_language]
- Racket [[https://en.wikipedia.org/wiki/Racket_\(programming_language\)](https://en.wikipedia.org/wiki/Racket_(programming_language))]
- Tipare dinamică [https://en.wikipedia.org/wiki/Dynamic_typing#Dynamic_typing]
- Tipare strong [https://en.wikipedia.org/wiki/Strong_typing]
- Evaluare aplicativă [https://en.wikipedia.org/wiki/Eager_evaluation]
- Scheme coding style [<https://www.gnu.org/software/mit-scheme/documentation/mit-scheme-user/Coding-style.html>]

Racket: Recursivitate

- Responsabil: Bogdan Popa [mailto:popabogdan97@gmail.com]
- Data publicării: 24.02.2020
- Data ultimei modificări: 24.02.2020

Obiective

Scopul acestui laborator este studierea diverselor **tipuri** de recursivitate, într-o manieră comparativă, și a modalităților de **eficientizare** a proceselor recursive.

Aspectele urmărite sunt:

- diferențele dintre mai multe **tipuri** de recursivitate, din perspectiva resurselor utilizate și, deci, a eficienței:
 - pe stivă
 - pe coadă
 - arborescentă
- modalități de **transformare** între diversele tipuri.

Arhiva de resurse conține un fișier cu **exerciții rezolvate**, la care textul face trimitere.

Recursivitate

Citiți secțiunea *1.2.1 Linear Recursion and Iteration* [<https://web.mit.edu/alexmv/6.037/sicp.pdf#subsection.1.2.1>] a cărții *Structure and Interpretation of Computer Programs* [<https://web.mit.edu/alexmv/6.037/sicp.pdf>], ediția a doua, până la *Example: Counting change*. Apoi, parcurgeți **rezumatul** următor, alături de **exercițiile rezolvate**.

În timpul rulării, un program utilizează o zonă de memorie, denumită **stivă**, cu informațiile de care (mai) are nevoie, în diferite momente. La fiecare **apel** de funcție, spațiul ocupat pe stivă **crește**. La fiecare **revenire** dintr-un apel, cu o anumită valoare, spațiul ocupat pe stivă **scade**. Când recursivitatea este foarte adâncă — există multe apeluri de funcție realizate, din care nu s-a revenit încă — spațiul ocupat pe stivă devine mult prea mare. Acest lucru **influențează**:

- **Memoria**, **principala** resursă afectată. În unele situații, trebuie reținute foarte multe informații. De exemplu, pentru o implementare particulară a funcției `factorial`, ar putea fi necesară reținerea tuturor înmulțirilor care trebuie efectuate la revenirea din recursivitate. Observați exercițiile rezolvate 1 și 2. De obicei, stiva are o dimensiune **maximă**, iar, atunci când aceasta este epuizată, programul se termină brusc, cu o **eroare** de tip „stack overflow”.
- **Timpul**, care, în locul **redimensionării** stivei, ar putea fi utilizat pentru alte calcule, programul rezultat fiind, astfel, mai rapid.

Recursivitate pe stivă

O funcție este recursivă **pe stivă** dacă apelul recursiv este parte a unei expresii mai complexe, fiind necesară **reținerea** de informații, pe stivă, pe avansul în recursivitate.

Exemplu

```
(define (factorial n)
  (if (= n 0)
```

```
1
(* n (factorial (- n 1))))
```

Recursivitate pe coadă

O funcție este recursivă **pe coadă** dacă valoarea întoarsă de apelul recursiv constituie valoarea de retur a apelului curent, i.e. apelul recursiv este un *tail call* [http://en.wikipedia.org/wiki/Tail_call], **nefiind** necesară reținerea de informație pe stivă.

Când o funcție este recursivă **pe coadă**, i se poate aplica *tail call optimization*. Din moment ce valoarea întoarsă de apelul curent este aceeași cu valoarea întoarsă de apelul recursiv, nu mai este necesară reținerea de informație pe stivă, pentru apelul curent, iar zona aferentă de stivă poate fi **înlocuită**, complet, de cea corespunzătoare apelului recursiv. Astfel, nu mai este necesară stocarea stării fiecărei funcții din apelul recursiv, **spațiul** utilizat fiind **O(1)**. Implementarea de Racket pe care o folosim conține această optimizare, ca parte a specificației.

Optimizarea menționată mai sus se poate aplica și în situații **mai relaxate**, precum *Tail recursion modulo cons* [http://en.wikipedia.org/wiki/Tail_call#Tail_recursion_modulo_cons]. Aici este permisă antrenarea valorii întoarse de apelul recursiv în anumite operații de **construcție**, cum este cons pentru liste. Pentru mai multe detalii, citiți informațiile de la adresa precedentă.

Pornind de la cele de mai sus, consumul de resurse poate fi **reduc** semnificativ, prin **transformarea** recursivității **pe stivă** în recursivitate **pe coadă** (numită și transformare a recursivității în iterație). Metoda de transformare prezentată în laborator constă în utilizarea unui **acumulator**, ca **parametru** al funcției, în care rezultatul final se construiește treptat, pe **avansul** în recursivitate, în loc de revenire.

Exemplu

```
(define (tail-recursion n acc)
  (if (= n 0)
      acc
      (tail-recursion (- n 1) (* n acc))))

(define (factorial n)
  (tail-recursion n 1))
```

Este important să aveți în vedere faptul că **funcțiile** recursive pe coadă **nu sunt definite** de prezența unui **acumulator**, precum nici funcțiile recursive pe stivă nu sunt definite de absența acestui acumulator. **Diferența** dintre cele două tipuri de funcții constă în **necesitatea** funcțiilor recursive **pe stivă** de a se **întoarce** din recursivitate pentru a **prelucra rezultatul**, respectiv **capacitatea** funcțiilor recursive **pe coadă** de a **genera** un rezultat **pe parcursul** apelului recursiv.

Exemplu

```
(define (member x L) ;; 0 posibilă implementare a funcției member
  (if (null? L) #f ;; Caz de bază, lista vidă nu conține elemente
      (if (equal? x (car L)) (cdr L) ;; Verificăm dacă elementul căutat este primul termen din listă
          (member x (cdr L))))) ;; Dacă nu, căutăm în restul listei
```

Recursivitate arborescentă

Recursivitatea arborescentă apare în cazul funcțiilor care conțin, în implementare, cel puțin două apeluri recursive care se execută necondiționat.

Exemplu

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


Se poate observa **ineficiența** implementării recursive pe stivă, atât din punct de vedere **temporal** (datorată **apelurilor duplicate**, precum apelul funcției pentru $n = 3$), cât și din punct de vedere **spațial** (datorată **stocării de „stack frames“** necesare pentru revenirea din recursivitate).

În cadrul exercițiilor rezolvate puteți observa mai în detaliu diferențele dintre recursivitatea pe stivă și cea pe coada (inclusiv în contextul funcției `fib`).

Resurse

Citiți exercițiile **rezolvate**; apoi, rezolvați exercițiile **propuse**.

- Exerciții rezolvate și propuse
- Soluții
- Cheatsheet (Nu uitați și de cheatsheet-ul de la primul laborator)

Referințe

- *Structure and Interpretation of Computer Programs* [<https://web.mit.edu/alexmv/6.037/sicp.pdf>], ediția a doua
- *Tail call* [http://en.wikipedia.org/wiki/Tail_call]

Racket: Funcții ca valori. Funcționale

- Responsabil: Ionuț Baciuc [mailto:ionut.b096@gmail.com]
- Data publicării: 03.03.2020

Obiective

Aspectele urmărite sunt:

- Funcții ca valori de ordinul 1
- Clasificarea funcțiilor după modul de primire a parametrilor
- Reutilizare de cod
- Funcționale

Funcții ca valori de prim rang

În programarea funcțională, funcțiile sunt valori de ordinul 1. Astfel, ele pot fi manipulate ca orice altă valoare, de exemplu:

- legate la un identificator: (define par? even?)
- stocate într-o structură de date: (list < > odd? even?)
- pasate ca argumente într-un apel de funcție: (list? even?)
- returnate ca rezultat al unui apel de funcție: funcții curry din paragraful următor

Funcții curry/uncurry

O funcție care își primește toți parametrii deodată se numește funcție **uncurry**. Până acum ați folosit doar funcții uncurry.

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

O funcție care returnează o nouă funcție atunci când este aplicată pe mai puține argumente decât îi sunt necesare se numește funcție **curry**.

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

La aplicarea unei funcții pe mai puține argumente, rezultatul este o funcție care așteaptă restul argumentelor.

Funcțiile curry facilitează reutilizarea de cod, permițând obținerea unor funcții particulare din funcții mai generale:

```
(define inc-curry (add-curry 1))
```

Reutilizare de cod

În secvența de cod de mai jos sunt implementate două funcții. Prima funcție obține pătratele elementelor unei liste, iar a doua funcție obține cuburile elementelor listei.

```
(define (sq x) (* x x))
(define (cub x) (* x x x))

(define (sq-every L)
  (if (null? L)
      L
      (cons (sq (car L)) (sq-every (cdr L)))))

(define (cub-every L)
  (if (null? L)
      L
      (cons (cub (car L)) (cub-every (cdr L)))))

(sq-every '(1 2 3 4))
(cub-every '(1 2 3 4))
```

După cum se poate observa, ambele funcții folosesc același pattern:

```
(define (?nume L)
  (if (null? L)
      L
      (cons (?functie (car L)) (?nume (cdr L)))))
```

Singurele diferențe sunt numele funcției (?nume) și funcția aplicată pe fiecare element al listei (?functie).

Prin urmare, pentru a nu scrie de două ori același cod putem defini o altă funcție mai generală:

```
(define (general-func f L)
  (if (null? L)
      L
      (cons (f (car L)) (general-func f (cdr L)))))
```

Această funcție poate fi apoi folosită pentru a implementa sq-every și cub-every:

```
(define (sq-every L) (general-func sq L))
(define (cub-every L) (general-func cub L))
```

Se observă că variabila ?functie din pattern-ul detectat anterior este acum parametru al funcției general-func. Această funcție mai generală poate fi particularizată prin pasarea diverselor funcții ca parametru (precum sq sau cub).

Deoarece funcțiile sq și cub sunt folosite o singură dată, acestea pot fi scrise in-place ca funcții anonime:

```
(define (sq-every-in-place L) (general-func (lambda (x) (* x x)) L))
(define (cub-every-in-place L) (general-func (lambda (x) (* x x x)) L))
```

Dacă dorim să scriem încă o funcție care să adune 2 la fiecare element al unei liste de numere, tot ce trebuie să facem este să folosim funcția general-func:

```
(define (+2-every L) (general-func (lambda (x) (+ 2 x)) L))
```

Însă codul de mai sus mai poate fi simplificat. Nu este nevoie să definim o nouă funcție pentru adunarea cu 2 a unui număr, ci ne putem folosi de funcția de adunare deja definită, pe care o aplicăm pe un singur parametru:

```
(define (+2-every L) (general-func (add-curry 2) L))
```

Este important de observat faptul că funcția de adunare trebuie să fie curry pentru a putea fi aplicată pe un singur parametru. Astfel, putem vedea utilitatea folosirii funcțiilor curry în scopul reutilizării de cod.

Funcționale

Funcționalele (în engleză higher order functions) sunt funcții care manipulează alte funcții, primindu-le ca argumente sau returnându-le ca rezultat, după cum e nevoie. Acestea sunt foarte utile în a aplica anumite modele de calcul des folosite.

Se poate observa că funcția `general-func` (definită mai sus) este, de asemenea, o funcțională deoarece primește ca parametru o altă funcție. Aplicarea unei funcții pe fiecare element al unei liste reprezintă un pattern des întâlnit în programarea funcțională și este deja implementat în Racket sub numele de `map`:

- `map`: returnează lista rezultatelor aplicării unei funcții `f` asupra fiecărui element dintr-o listă. [Se pot da `n` liste și atunci `f` are `n` parametri, fiecare din câte o listă. Listele trebuie să aibă aceeași lungime]

```
(map f L)
```

Mai jos se pot observa câteva exemple folosind `map`:

```
(map add1 '(1 4 7 10)) ; întoarce '(2 5 8 11)
(map sq '(1 2 3 4)) ; întoarce '(1 4 9 16)
(map + '(1 2 3) '(4 8 16) '(2 4 6)) ; întoarce '(7 14 25)
```

Alte exemple de funcționale des întâlnite:

- `filter`: returnează lista elementelor dintr-o listă care satisfac un predicat `p`.

```
(filter p L)
```

Mai jos se pot observa câteva exemple folosind `filter`:

```
(filter even? '(1 3 4 7 8)) ; întoarce '(4 8)
(filter positive? '(1 -2 3 4 -5)) ; întoarce '(1 3 4)
(filter (lambda (x) (>= x 5)) '(1 5 3 4 10 11)) ; întoarce '(5 10 11)
```

- `foldl` (*fold left*): returnează rezultatul aplicării funcției `f` pe rând asupra unui element din listă și a unui acumulator. Ordinea folosirii elementelor din listă este de la stânga la dreapta. [Se pot da `n` liste și atunci `f` are `(+ n 1)` parametri, dintre care ultimul este acumulatorul. Listele trebuie să aibă același număr de elemente]

```
(foldl f init L)
```

Mai jos se pot observa câteva exemple folosind `foldl`:

```
(foldl cons null '(1 2 3 4)) ; întoarce '(4 3 2 1)
(foldl (lambda (x y result) (* result (+ x y))) 1 '(4 7 2) '(-6 3 -1)) ; întoarce -20
```

- `foldr` (*fold right*): singurele diferențe între `foldl` și `foldr` este că `foldr` ia elementele de la dreapta spre stânga și că are nevoie de un spațiu proporțional cu lungimea listei.

```
(foldr f init L)
```

Mai jos se pot observa câteva exemple folosind `foldr`:

```
(foldr cons null '(1 2 3 4)) ; întoarce '(1 2 3 4)
(foldr (lambda (x acc) (cons (* x 2) acc)) null '(1 2 3 4)) ; întoarce '(2 4 6 8)
```

- `apply`: returnează rezultatul aplicării unei funcții `f` cu argumente elementele din lista `L`

```
(apply f L)
```

Mai jos se pot observa câteva exemple folosind `apply`:

```
(apply + '(1 2 3 4)) ; întoarce 10
(apply * 1 2 '(3 4)) ; întoarce 24
(apply cons '(1 2)) ; întoarce perechea '(1 . 2)
```

Fără funcționale ar trebui să scriem mereu un cod asemănător, ceea ce nu este în spiritul reutilizării codului. Ceea ce trebuie să facem când în spatele unor prelucrări observăm un mecanism mai general, mai abstract, este să scriem o funcțională care descrie acel mecanism.

Prin folosirea funcționalelor și funcțiilor `curry`, codul scris este mult mai restrâns dar și mult mai clar și ușor de urmărit.

Exemple

De multe ori, funcționalele `map` și `apply` sunt încurcate. Pentru a înțelege mai bine diferența dintre acestea, urmăriți rezultatele exemplurilor de mai jos.

```
(map list '(1 2 3)) ; întoarce '((1) (2) (3))
(apply list '(1 2 3)) ; întoarce '(1 2 3)
(map list '(1 2 3 4) '(5 6 7 8)) ; întoarce '((1 5) (2 6) (3 7) (4 8))
(apply list '(1 2 3 4) '(5 6 7 8)) ; întoarce '((1 2 3 4) 5 6 7 8)
```

O altă greșeală întâlnită frecvent apare atunci când funcțiile primite ca argument de către funcționale sunt plasate între paranteze.

```
(filter odd?) '(1 2 3 4 5)) ; odd?: arity mismatch; (GREȘIT)
(filter odd? '(1 2 3 4 5)) ; întoarce '(1 3 5) (CORECT)
(foldr (+) 0 '(8 9 10)) ; GREȘIT
(foldr + 0 '(8 9 10)) ; întoarce 27 (CORECT)
(foldr ((λ (x y) (* x y))) 1 '(1 2 3 4)) ; GREȘIT
(foldr (λ (x y) (* x y)) 1 '(1 2 3 4)) ; întoarce 24 (CORECT)
```

Resurse

Citiți exercițiile **rezolvate**; apoi, rezolvați exercițiile **propuse**.

- [Exerciții rezolvate și propuse](#)
- [Soluții](#)
- [Cheatsheet](#)

Referințe

- *Structure and Interpretation of Computer Programs* [<https://web.mit.edu/alexmv/6.037/sicp.pdf>], până înainte de „Using let to create local variables”
- *Mai multe functionale* [http://docs.racket-lang.org/reference/pairs.html#%28part_.List_.Iteration%29]

Racket: Legarea variabilelor. Închideri funcționale

- Responsabil: Teodor Szente [<mailto:teodor98sz@gmail.com>]
- Data publicării: 08.03.2020

Obiective

Scopul acestui laborator este prezentarea **conceptelor de legare (binding)** în limbajele din familia LISP, exemplificate în Racket.

Se urmărește stăpânirea conceptelor teoretice:

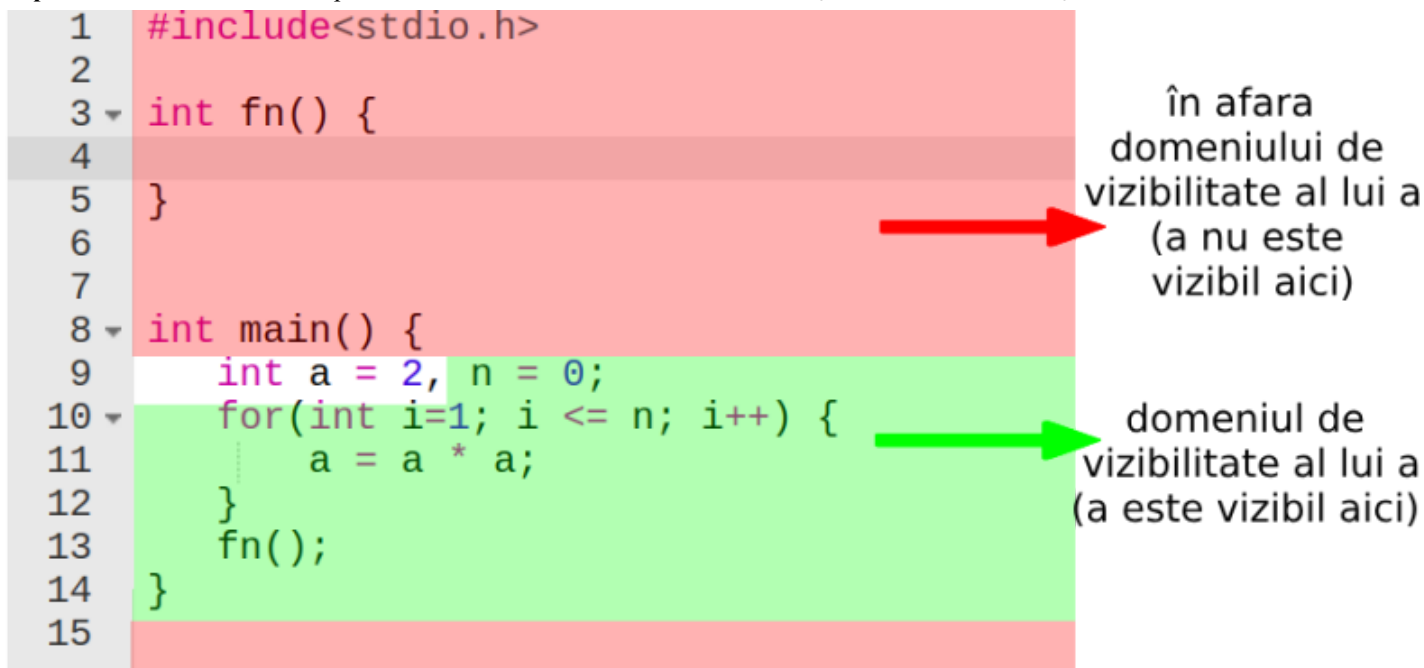
- **domeniu de vizibilitate**
- **context computațional**
- **legare**
- **legare statică și legare dinamică**
- **închidere funcțională**
- **întârzierea evaluării** (folosind închideri funcționale)

Domeniu de vizibilitate

în engleză: Scope [[https://en.wikipedia.org/wiki/Scope_\(computer_science\)](https://en.wikipedia.org/wiki/Scope_(computer_science))]

Domeniul de vizibilitate al unei variabile este mulțimea punctelor din program în care variabila este vizibilă. Cu alte cuvinte, domeniul de vizibilitate al variabilei x este reprezentat de porțiunile din program în care aceasta poate fi accesată (este vizibilă).

Exemplu: Domeniul de vizibilitate pentru variabila a este format din liniile de cod {9, 10, 11, 12, 13, 14}



```

1  #include<stdio.h>
2
3  int fn() {
4
5  }
6
7
8  int main() {
9      int a = 2, n = 0;
10     for(int i=1; i <= n; i++) {
11         a = a * a;
12     }
13     fn();
14 }
15

```

în afara domeniului de vizibilitate al lui a (a nu este vizibil aici)

domeniul de vizibilitate al lui a (a este vizibil aici)

Context computațional

în engleză: Context [[https://en.wikipedia.org/wiki/Context_\(computing\)](https://en.wikipedia.org/wiki/Context_(computing))]

Definim **contextul computațional** al unui punct P din program la un moment t ca fiind mulțimea variabilelor vizibile în punctul P , la momentul t . Cu alte cuvinte, contextul computațional al unui punct este dat de toate variabilele și valorile acestora, vizibile în acel punct.

Exemplu: Pe linia 6 contextul computațional este: {(a 2) (b 32) (s P)}

The diagram shows a Racket code snippet with line numbers 1 to 7. Line 1: `#include<stdio.h>`. Line 2: (empty). Line 3: `int main() {`. Line 4: `int a = 2, b = 32;`. Line 5: `char s = 'P';`. Line 6: (empty). Line 7: `}`. To the right of the code, the text "Contextul computațional pe linia 6 este:" is followed by the environment frame `((a 2) (b 32) (s 'P'))`. The variables `a`, `b`, and `s` in the code are highlighted in green, red, and blue respectively, matching the colors in the environment frame.

Legare

în engleză: [Name binding](https://en.wikipedia.org/wiki/Name_binding) [https://en.wikipedia.org/wiki/Name_binding]

O variabilă poate fi reprezentată printr-o pereche dintre un identificator și valoarea acesteia la un moment dat.

```
int random_number = 42;
//      |           |
//      |           |-----> valoarea variabilei la un moment dat.
//      v
// "random_number" este valoarea identicatorului.
```

Legarea este procedeul prin care se face asocierea dintre identificatori și locul în care se află valorile efective ale variabilei.

Nu confundați assignment-ul unei variabile cu legarea, citiți [această explicație](https://stackoverflow.com/a/48103225) [https://stackoverflow.com/a/48103225].

Legarea poate fi de două tipuri:

- dinamică - toți identificatorii și valorile variabilelor sunt puse într-un context global
- statică - pentru fiecare legare se creează un nou context de identificatori și valori.

În Racket legarea se face prin construcția `let` care permite crearea de variabile ce vor fi vizibile în corpul `let`-ului.

```
(let ((x 2))
  ;; x este vizibil aici si are valoarea 2;))
```

Legare statică

Este folosită în majoritatea limbajelor de programare din motive istorice (ALGOL 60/ALGOL 58/Fortran o folosesc) dar și din motive pragmatice: este ușor de interpretat de către oameni și calculatoare.

Legarea statică creează un nou domeniu de vizibilitate (scope) pentru o variabilă, în funcție de contextul lexical al programului (partea programului care este evaluată), așa că în literatura de specialitate se mai numește **lexical scoping** / **lexical binding**

În racket `let` face legare statică:

```
(define (lexical-binding-example x y)
  (let ((x 20)) (+ x y)))

(lexical-binding-example 30 40)
```

- 2) Construcția `let` va crea un nou context copiindu-l pe cel actual și va adăuga (suprascrie acolo unde este cazul) noile legături:
`((x 20) (y 40))`
- 1) Când se apelează funcția se creează un nou context și se adaugă parametrii:
`((x 30) (y 40))`

- 3) La ieșirea din construcția `let` se va restaura contextul anterior:
`((x 30) (y 40))`

Legare dinamică

A fost implementată prima dată în LISP. Este folosită în unele limbaje imperative pentru implementarea variabilelor globale.

În Racket `define` face legare dinamică:

```
(define a 1)
(define (dynamic-binding-example x)
  (+ x a))
(dynamic-binding-example 2)
(define a 2)
(dynamic-binding-example 2)
```

- 1) Define-ul va scrie în contextul global:
`((a 1))`
- 2) Când se apelează funcția se creează un nou context și se adaugă parametri:
`((x 2) (a 1))`
- 3) Define-ul va scrie în contextul global
`((a 2))`
- 4) Când se apelează funcția se creează un nou context și se adaugă parametri:
`((x 2) (a 2))`

Observați că același apel de funcție cu aceiași parametri întoarce rezultate diferite în funcție de contextul global ⇒ **introduce efecte laterale**, de aceea editarea contextului global cu `define` este interzisă în Racket.

Legare în racket. Construcții pentru legare

let

În realitate, un `let` este zăhărel sintactic pentru o λ -expresie. Codul anterior este echivalent cu:

```
((lambda (<id1> ...<idn>)
  <expr1>
  ...
  <exprn>)
<val1>
...
<valn>)
```

Corpul unui `let` conține una sau mai multe expresii (`<expr1> ... <exprn>` în exemplul de mai sus). Acestea sunt evaluate în ordine, iar rezultatul întors de `let` este rezultatul evaluării ultimei expresii.

Iată și câteva exemple:


```
(define a 10)
```

```
(let ((a 1) (b (+ a 1)))      ; aici suntem în zona de definiții, nu în corpul let-ului => a e legat la 10
  (cons a b))                ; în corpul let-ului este vizibilă legarea lui a la 1
```

Codul anterior produce perechea (1 . 11), întrucât legarea (a 1) este vizibilă doar în corpul let-ului. a-ul folosit în legarea lui b este 10 mulțumită define-ului de mai sus. Fără acel define, codul ar fi generat eroare.

```
(let ((a 1))                  ; prima legare
  (let ((f (lambda () (print a))))
    (let ((a 2))              ; a doua legare
      (f))))                  ; afișează 1
```

În punctul din program corespunzător definirii lui f, identificatorul a este legat la valoarea 1. Faptul că în contextul în care se apelează funcția f a este legat la valoarea 2 nu are importanță. Comparați acest comportament cu exemplul din secțiunea *Legare dinamică*, de mai jos.

let*

Este asemănător cu let, însă domeniul de vizibilitate al variabilelor începe imediat după definire. Asta înseamnă că o variabilă definită în let* poate fi folosită în următoarele definiții din cadrul let*.

```
(define a 10)
```

```
(let* ((a 1) (b (+ a 1)))    ; în momentul definirii lui b, este vizibilă legarea lui a la 1
  (cons a b))                ; desigur, aceeași legare e vizibilă și în corpul let-ului
```

Codul anterior întoarce perechea (1 . 2), spre deosebire de (1 . 11) pe care l-am obținut folosind let.

letrec

Domeniul de vizibilitate este întregul letrec - cu alte cuvinte inclusiv zona de definire a variabilelor care precede definirea variabilei curente. La momentul în care valoarea variabilei este folosită, ea trebuie să fi fost deja definită (poate fi folosită într-un punct dinaintea definiției sale - din punct de vedere textual - dar nu și din punct de vedere temporal). Acest aspect este ilustrat în exemplele de mai jos:

```
(letrec ((a b) (b 1))        ; în momentul definirii lui a este nevoie de valoarea lui b, necunoscută încă
  (cons a b))                ; de aceea codul produce eroare
```

```
(letrec
  ((even-length?
    (lambda (L)                ; even-length? este o închidere funcțională
      (if (null? L)            ; deci corpul funcției nu este evaluat la
        #t                    ; momentul definirii ei
        (odd-length? (cdr L)))) ; deci nu e o problemă că încă nu știm cine e odd-length?
    (odd-length?
      (lambda (L)
        (if (null? L)
          #f
          (even-length? (cdr L))))))
  (even-length? '(1 2 3 4 5 6))) ; în acest moment deja ambele funcții au fost definite
```

Codul de mai sus întoarce true. odd-length? este vizibilă în zona de program corespunzătoare definiției lui even-length? și va funcționa corect cu condiția ca momentul în care solicităm valoarea sa în acel punct din program să succeadă momentului definirii lui odd-length?.

Named let

Această construcție se folosește pentru a obține un mod de a itera în interiorul unei funcții. Numele dat let-ului (în cazul de mai jos, iter) referă o procedură care primește ca parametri variabilele din lista de definiții și care evaluează expresiile din corpul let-ului. Exemplul următor generează intervalul numeric [a, b] cu pasul step (se presupune că diferența dintre b și a este multiplu de step):

```
(define (interval a b step)
  (let iter ((b b) (result '()))      ; parametrul b este inițializat cu valoarea b, result cu '()
    (if (> a b)                        ; aici e vizibil b-ul parametru pentru iter, nu b-ul funcției interval
      result
      (iter (- b step) (cons b result)))) ; iter este apelată recursiv ca o funcție de 2 parametri

(interval 2 10 2)                    ; întoarce valoarea '(2 4 6 8 10)
```

Închideri funcționale

Conceptul de închidere funcțională a fost inventat în anii '60 și implementat pentru prima dată în Scheme (din care a fost derivat Racket). Pentru a înțelege acest concept, să ne gândim ce se întâmplă în Racket când definim o funcție, de exemplu funcția de mai sus: `(define f (lambda (x) (+ x a)))`. Ceea ce face orice `define` este să creeze o pereche identificator-valoare; în acest caz se leagă identificatorul `f` la valoarea produsă de evaluarea λ -expresiei `(lambda (x) (+ x a))`. Dar ce valoare produce evaluarea unei λ -expresii? **Evaluarea oricărei λ -expresii produce o închidere funcțională.**

O **închidere funcțională** este o pereche între:

- **textul** λ -expresiei `((lambda (x) (+ x a)))` pe exemplul nostru)
- **contextul** computațional în punctul de definire a λ -expresiei `((a 1))` pe exemplul nostru)

Ceea ce salvăm în context este de fapt mulțimea **variabilelor libere** în λ -expresia noastră, adică toate acele variabile referite în textul λ -expresiei dar definite în afara ei. Contextul unei închideri funcționale rămâne cel din momentul creării închiderii funcționale, cu excepția variabilelor definite cu `define`, care ar putea fi înlocuite în timp.

Când o închidere funcțională este aplicată pe argumente, contextul salvat este folosit pentru a da semnificație variabilelor libere din textul λ -expresiei. Este vorba de contextul din momentul aplicării, nu din momentul creării închiderii.

Un aspect remarcabil al închiderilor funcționale este că pot fi folosite pentru a **întârzia evaluarea**. Plecând de la ideea că o închidere funcțională este o pereche text-context, iar textul nu este câtuși de puțin evaluat înainte de aplicarea λ -expresiei pe argumente, consecința este că putem „închide” orice calcul pe care vrem să îl amânăm pe mai târziu într-o expresie `(λ () calcul)` și să provocăm evaluarea exact la momentul dorit, aplicând această expresie (aici pe 0 argumente).

Resurse

- Documentație racket [<https://docs.racket-lang.org/reference/let.html>]
- [Exerciții](#)
- [Soluții](#)
- [Cheatsheet](#)

Referințe

- [Lexical Binding](#) [<https://www.cs.oberlin.edu/~bob/cs275.spring14/Examples%20and%20Notes/February/February%2028/Lexical%20and%20Dynamic%20Binding.pdf>]
- [legare statica vs legare dinamica](#) [<https://www.emacswiki.org/emacs/DynamicBindingVsLexicalBinding>]
- [Name, scope, binding](#) [http://www.cs.iusb.edu/~danav/teach/c311/c311_3_scope.html]

Racket: Întârzierea evaluării

- Responsabil: Mihaela Balint [mailto:oomot@gmail.com]
- Data publicării: 16.03.2020

Obiective

Scopul acestui laborator este înțelegerea diverselor tipuri de evaluare, respectiv a controlului evaluării în Racket.

Conceptele introduse sunt:

- evaluare aplicativă
- evaluare leneșă
- promisiuni
- fluxuri

Evaluare aplicativă vs. evaluare leneșă

Fie următoarea aplicație, scrisă în **calcul Lambda**:

```
(λx.λy.(x + y) 1 2)
```

În urma aplicării funcției de mai sus, expresia se va evalua la 3. Ce se întâmplă însă dacă parametrii funcției noastre reprezintă alte aplicații de funcții?

```
(λx.λy.(x + y) 1 (λz.(z + 2) 3))
```

Intuitiv, expresia de mai sus va aduna 1 cu 5, unde 5 este rezultatul evaluării expresiei $(\lambda z.(z + 2) 3)$. În cadrul acestui raționament, am presupus că parametrii sunt evaluați **înainte** aplicării funcției asupra acestora. Vom vedea, în cele ce urmează, că evaluarea se poate realiza și în altă ordine.

Evaluare aplicativă

În **evaluarea aplicativă** (*eager evaluation*), o expresie este evaluată **imediat** ce o variabilă se leagă la aceasta. Pe exemplul de mai sus, **evaluarea aplicativă** decurge astfel:

```
(λx.λy.(x + y) 1 (λz.(z + 2) 3))
(λx.λy.(x + y) 1 5)
6
```

Observații:

- parametrii funcției sunt evaluați **înainte** aplicării funcției asupra acestora; afirmăm că transferul parametrilor se face **prin valoare**;
- **Racket** (ca și majoritatea limbajelor tradiționale) folosește **evaluare aplicativă**.

Evaluare leneșă

Spre deosebire de **evaluarea aplicativă**, **evaluarea leneșă** va întârzi evaluarea unei expresii până când valoarea ei este necesară. Exemplu:

```
(λx.λy.(x + y) 1 (λz.(z + 2) 3))
(1 + (λz.(z + 2) 3))
(1 + 5)
6
```

Observații:

- aplicația funcției anonime de parametru `z` este trimisă ca parametru și nu se evaluează înainte ca acest lucru să devină necesar; afirmăm că transferul parametrilor se face **prin nume**;
- **Haske11** folosește **evaluare leneșă**.

Există avantaje și dezavantaje ale **evaluării leneșe**. O situație în care **evaluarea leneșă** se poate dovedi utilă este următoarea:

Fie funcția `Fix`:

```
Fix = λf.(f (Fix f))
```

`Fix` primește ca parametru o funcție `f`, care este aplicată asupra parametrului `(Fix f)`. Să considerăm funcția constantă (care întoarce mereu valoarea `b`):

```
ct = λx.b
```

Să aplicăm `Fix` asupra funcției constante `ct`, folosind **evaluarea aplicativă**:

```
(Fix ct)
(λf.(f (Fix f)) ct)
(ct (Fix ct))
(ct (λf.(f (Fix f)) ct))
(ct (ct (Fix ct)))
(ct (ct (λf.(f (Fix f)) ct)))
(ct (ct (ct (Fix ct))))
...
```

Cum evaluarea este **aplicativă**, parametrul trimis lui `ct`, mai precis `(Fix ct)`, va fi evaluat, în mod recursiv, la infinit.

Să încercăm aceeași aplicare, folosind de data aceasta **evaluarea leneșă**:

```
(Fix ct)
(λf.(f (Fix f)) ct)
(ct (Fix ct))
(λx.b (Fix ct))
b
```

Observăm că evaluarea **leneșă** a lui `(Fix ct)` se termină și întoarce valoarea `b`. Terminarea este posibilă datorită faptului că evaluarea expresiei `(Fix ct)`, trimisă ca parametru lui `ct`, este întârziată până când este nevoie de ea (în acest caz, niciodată).

Construcții precum `Fix` pot părea ezoterice și nefolositoare. În realitate însă, ele au aplicabilitate. `Fix` este un *combinator de punct fix* („generator“ de funcții recursive).

Evaluarea în Racket

Fie următorul cod:

```
(+ 1 (+ 2 3))
```

Codul de mai sus este o rescriere din **calculul Lambda** în **Racket** a exemplului introductiv. Reamintim că evaluarea în Racket este **aplicativă**, așadar etapele parcurse sunt următoarele:

- `(+ 2 3)`: al doilea parametru al funcției `+` se evaluează la 5;
- `(+ 1 5)` se evaluează la 6.

Pentru a obține beneficiile evaluării **leneșe** în Racket, putem **întârzia** evaluarea unei expresii în două moduri:

- închideri funcționale (de obicei nulare (fără parametri)): `(lambda () (...))`;
- promisiuni: `delay/force`.

Evaluare leneșă folosind închideri

Fie următorul exemplu:

```
(define sum
  (lambda (x y)
    (lambda ()
      (+ x y))))
(sum 1 2)
```

Observați că `(sum 1 2)` nu evaluează suma parametrilor, ci întoarce o **funcție**. Mai precis, avem de-a face cu o **închidere funcțională** (funcție care își salvează contextul). Pentru a **forța** evaluarea este suficient să aplicăm rezultatul întors de `(sum 1 2)` asupra celor zero parametri pe care îi așteaptă, astfel:

```
((sum 1 2)) ; se va evalua la 3
```

Evaluare leneșă cu promisiuni

Fie definiția convențională a sumei între două numere:

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

Putem **întârzia** evaluarea unei sume astfel:

```
(delay (sum 1 2)) ; va afișa #<promise>
```

Pentru a scrie o funcție **cu evaluare leneșă**, asemănătoare celei scrise cu `(lambda () ...)`, procedăm astfel:

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

```
(sum 1 2) ; va afișa #<promise>
```

Pentru a forța evaluarea, folosim `force`:

```
(force (sum 1 2)) ; va afișa 3
```

Fluxuri - aplicații ale evaluării leneșe

Folosind evaluarea leneșă, putem construi **obiecte infinite** sau **fluxuri** (*streams*). Exemplu de *flux*: șirul numerelor naturale: $(0\ 1\ 2\ 3\ \dots\ n\ \dots)$. Un astfel de obiect se reprezintă ca o **pereche** între:

- un **element curent** (primul element din flux, asemănător unui `car` pe liste);
- un **generator** (o promisiune sau o închidere funcțională) care, pornit (prin `force` sau aplicație), va întoarce următoarea pereche din *flux* (restul fluxului, asemănător unui `cdr` pe liste).

Exemple:

Șirul constant de 1

Fie șirul infinit:

$a(n) = 1, n \geq 0$

Primele 5 elemente ale acestui șir sunt:

```
(1 1 1 1 1)
```

Șirul se poate genera astfel, folosind închideri funcționale:

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

Observăm că șirul este o pereche în care:

- primul element este **valoarea curentă**, 1;
- al doilea element este o **funcție** ((lambda () ones-stream)) capabilă să genereze restul fluxului.

Folosind promisiuni, același șir este generat ca mai jos:

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

Limbajul Racket pune la dispoziție o interfață pentru manipularea fluxurilor, asemănătoare cu aceea pentru manipularea listelor:

- stream-cons este omologul lui cons (adaugă un element la începutul unui flux);
- stream-first este omologul lui car (extrage primul element din flux);
- stream-rest este omologul lui cdr (reprezintă fluxul fără primul său element);
- empty-stream este omologul lui '() (fluxul vid);
- stream-empty? este omologul lui null? (testează dacă un flux este vid);
- stream-map, stream-filter corespund lui map, filter (însă stream-map acceptă doar funcții unare).

În spiritul abstractizării, putem folosi această interfață fără să ne preocupe dacă funcțiile sunt implementate folosind închideri funcționale sau promisiuni (pentru curioși - sunt implementate folosind promisiuni). Șirul infinit de 1 se va genera astfel:

```
(define ones-stream
  (stream-cons 1 ones-stream))
```

Să scriem o funcție care întoarce primele n elemente din șir:

```
; extragerea primelor n elemente din șirul s
(define (stream-take s n)
  (cond ((zero? n) '())
        ((stream-empty? s) '())
        (else (cons (stream-first s)
                      (stream-take (stream-rest s) (- n 1))))))
```

```
; testare
(stream-take ones-stream 5) ; va afișa (1 1 1 1 1)
```

Șirul numerelor naturale

Fie următorul cod, care implementează șirul numerelor naturale:

```
; generator pentru numere naturale
(define (make-naturals k)
  (stream-cons k (make-naturals (add1 k))))
```

```
; fluxul numerelor naturale
(define naturals-stream (make-naturals 0))
```

```
; testare
(stream-take naturals-stream 4) ; va afișa (0 1 2 3)
```

Observații:

- în plus față de cum am generat șirul infinit de 1, aici am folosit o funcție, `add1`, care calculează elementul următor din flux pe baza elementului curent - din acest motiv, am definit un generator care primește ca parametru elementul curent;
- în funcție de fluxul construit, generatorul poate primi alți parametri.

Șirul lui Fibonacci

Șirul lui Fibonacci este:

`Fibo = t0 t1 t2 t3 ... tn ...`

unde:

`t0 = 0`

`t1 = 1`

`tk = t(k-2) + t(k-1)` pentru `k >= 2`

În cazul acestui șir, un nou element este calculat pe baza unor **valori** calculate **anterior**. Observăm că:

`Fibo = t0 t1 t2 t3 ... t(k-2) ... +`
`(tail Fibo) = t1 t2 t3 t4 ... t(k-1) ...`

`Fibo = t0 t1 t2 t3 t4 t5 ... tk ...`

Adunând elemente din șirurile (fluxurile) `Fibo` și `(tail Fibo)`, obținem șirul `t2 t3 ... tk ...`. Dacă adăugăm la începutul acestui rezultat elementele `t0` și `t1`, obținem exact `Fibo`.

Pentru a implementa expresia de mai sus, avem un singur **obstacol** de depășit, și anume trebuie să scriem o funcție care **adună două fluxuri**. O implementare posibilă este:

```
(define add
  (lambda (s1 s2)
    (stream-cons (+ (stream-first s1) (stream-first s2))
                  (add (stream-rest s1) (stream-rest s2)))))
```

Definim fluxul `Fibo` pe baza adunării de mai sus:

```
(define fibo-stream
  (stream-cons 0
               (stream-cons 1
                             (add fibo-stream (stream-rest fibo-stream)))))
```

Fluxul numerelor prime

Eratostene a conceput un algoritm pentru determinarea fluxului numerelor prime, care funcționează astfel: fie șirul numerelor naturale, începând cu 2:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 ...

—

Păstrăm primul element din șir (2) și eliminăm elementele care se divid cu el:

2 3 5 7 9 11 13 15 17 19 21 ...

—

Apoi păstrăm următorul element din șirul rămas (3) și eliminăm toate elementele care se divid cu el:

2 3 5 7 11 13 17 19 ...

—

etc.

Algoritmul este implementat mai jos:

```
(define sieve
  (lambda (s)
    (let ((divisor (stream-first s)))
      (stream-cons
        ;; păstrează primul element
        divisor
        ;; și aplică recursiv algoritmul pe numerele care nu se divid cu el
        (sieve (stream-filter (lambda (x) (> (modulo x divisor) 0))
                              (stream-rest s)))))))

(define primes-stream
  (sieve
   ;; șirul numerelor naturale începând de la 2
   (stream-rest (stream-rest naturals-stream))))

; testare
(stream-take primes-stream 10) ; va afișa (2 3 5 7 11 13 17 19 23 29)
```

Resurse

- [Exerciții](#)
- [Soluții](#)
- [Cheatsheet](#)

Referințe

- *Structure and Interpretation of Computer Programs* [<https://web.mit.edu/alexmv/6.037/sicp.pdf>], ediția a doua
- *Evaluation strategy* [http://en.wikipedia.org/wiki/Evaluation_strategy]

Haskell: Introducere

- Responsabil: Vlad Nedelcu [mailto:nedelcu_vlad@yahoo.com]
- Data publicării: 25.03.2020
- Data ultimei modificări: 10.04.2020

Obiective

Scopul acestui laborator este acomodarea cu primele noțiuni din Haskell și observarea incipientă a diferențelor dintre acesta și Racket.

Aspectele urmărite sunt:

- diferențele principale dintre cele 2 limbaje
- definirea funcțiilor în Haskell
- pattern matching
- list comprehensions
- tipuri
- gărzi

Introducere

În jurul anilor 1990 un comitet de cercetători în limbaje de programare (Simon Marlow, Simon Peyton Jones, Philip Wadler etc) au creat un limbaj nou care a ajuns să fie standardul de-facto în cercetarea din domeniul programării funcționale. Inspirat dintr-o varietate de limbaje – Miranda, ML, Scheme, APL, FP – limbajul a influențat la rândul lui majoritatea limbajelor de programare cunoscute.

Haskell este un limbaj funcțional **pur**. Spre deosebire de limbajele imperative (sau Racket unde există `set!`), în Haskell aproape toate funcțiile sunt pure. Funcțiile impure sunt marcate diferit prin intermediul sistemului de tipuri.

În plus față de Racket, Haskell are **tipare statica** (și tipuri **polimorfice**). Fiecare expresie are un tip și este sarcina programatorului să efectueze conversiile necesare între tipuri dacă este necesar. De cele mai multe ori, informația despre ce face o funcție se găsește integral în tipul acesteia și numele ei. Astfel, se pune accentul pe *ce* face funcția, nu *cum* efectuează ea operațiile cerute.

Deși tipurile există, programatorul nu este obligat să depună efort în a le scrie în program. Haskell deține inferență de tipuri puternică. Exceptând cazurile în care se folosesc concepte avansate, programatorul poate lucra fără a scrie o singură semnătură de funcție (deși nu e recomandat pentru că se pierde o parte din documentația funcției).

În plus, tipurile ajută programatorul în procesul de scriere a codului transformându-l într-un exercițiu de rezolvare a unui puzzle: pur și simplu trebuie să găsești tipurile folosind funcții existente sau scriind alte funcții. Pentru a căuta funcțiile ce respectă o semnătură se poate folosi Hoogle [<http://www.haskell.org/hoogle/>], un motor de căutare similar Google dar doar pentru funcții Haskell.

Deosebirea fundamentală față de alte limbaje este **evaluarea leneșă**. Funcțiile nu vor fi evaluate și expresiile nu vor fi reduse până în momentul în care valoarea lor este necesară. Programatorul poate astfel lucra cu date infinite, extrăgând din ele strictul necesar pentru a obține soluția. Ca dezavantaj, analiza performanței unui cod Haskell este puțin mai dificilă, dar există instrumente auxiliare (dezvoltate în Haskell).

În cele ce urmează vom parcurge fiecare dintre aceste particularități, accentuând diferențele față de Racket. Pe scurt, o **paralela** între cele două limbaje arată în felul următor:

Limbaaj	Evaluare	Tipare	Efecte laterale
Racket	Aplicativă	Dinamică	Da
Haskell	Leneșă	Statică	Nu

Laboratoarele de Haskell din cadrul cursului se vor axa pe evidențierea acestor diferențe și a beneficiilor care rezultă din ele. Pentru noțiuni avansate despre limbaj, consultați primele 2 resurse din bibliografie (sau Reddit, Stack Overflow, Planet Haskell, etc.)

Un exemplu pentru diferența de evaluare dintre Racket și Haskell:

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

square x = x * x
```

Evaluarea aplicativă din Racket va forța evaluarea parametrului x și apoi va apela funcția square:

```
(square (+ 1 2))
(square 3)
(* 3 3)
9
```

Evaluarea leneșă din Haskell va evalua parametrul x la cerere în cadrul apelului funcției square:

```
square (1 + 2)
(1 + 2) * (1 + 2)
3 * (1 + 2)
3 * 3
9
```

GHC. GHCi

Există o varietate de compilatoare și interpretoare pentru Haskell. În momentul de față, limbajul și evoluția lui sunt strâns legate de eforturile dezvoltatorilor de la Glasgow. GHC, *The Glorious Glasgow Haskell Compilation System*, este și compilatorul pe care-l vom folosi pentru cursul de PP.

Pentru a avea o experiență bună cu acest limbaj, recomandarea este să vă instalați Haskell Stack (https://docs.haskellstack.org/en/stable/install_and_upgrade/) (cititi instructiunile de aici: Limbaje). Față de compilator și suita minimală de pachete, Haskell Stack aduce în plus o suită de biblioteci utile pentru dezvoltarea unor aplicații reale.

Codul Haskell poate fi atât compilat cât și interpretat. Pentru interpretare vom folosi **ghci**, iar pentru compilare vom folosi **ghc**. Fișierele de cod Haskell au în mod normal extensia `.hs`, dar se poate folosi și `.lhs` pentru variantele de **Literate Haskell** (programare ca o poveste - comentariile ocupă majoritatea textului în timp ce secvențele de cod sunt puține - este formatul preferat pentru publicarea de articole despre Haskell pe bloguri, oricine poate copia textul articolului într-un fișier și îl poate compila și rula apoi).

În cadrul laboratorului, recomandăm folosirea **ghci** în modul următor:

- se salvează definițiile de funcții într-un fișier cu extensia `.hs` (ex.: `lab6-ex.hs`)
- se deschide consola sistemului (ex.: `cmd/powershell` pe Win) în directorul cu fișierul `.hs`
- se rulează `stack exec ghci nume.hs` pentru a-l încărca în interpretor
- se rulează în interpretor apelurile de funcții de test necesare, verificarea de tipuri, etc
- dacă se dorește **editarea fișierului** se poate face într-un terminal separat (recomandat) sau folosind `:e (:edit)` din `ghci`
- ATENȚIE! după editarea fișierului, pentru a reîncărca definițiile se folosește `:re (:reload)`
- dacă se dorește încărcarea altor module în interpretor (pentru testare), se poate folosi `:m +Nume.Modul`

- pentru a verifica tipul unor expresii se folosește `:t expresie`
- pentru a ieși din interpretor se folosește `:q` (sau `EOF` - `^Z` pe Linux/OSX, `^D` pe Windows)

Modulul `Prelude` conține funcții des folosite și este inclus implicit în orice fișier (deși poate fi exclus la nevoie, consultați [bibliografia](#)).

În cadrul unor programe reale va trebui să compilați sursele. Pentru aceasta va trebui să aveți un modul `Main.hs` care să conțină o funcție `main`. Modulul poate importa alte module prin `import Nume.Modul`. Pentru temă, nu va trebui să efectuați acești pași, scheletul de cod ce va fi oferit va conține partea de interacțiune cu mediul exterior. 😊

Comentarii

În Haskell, avem comentarii pe un singur rând, folosind `--` (2 de - legați) sau comentarii pe mai multe rânduri, încadrate de `{ - și - }`. În plus, există comentarii pentru realizarea de documentație dar nu vom insista pe ele aici.

Funcții

O funcție anonimă în **Racket**

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

poate fi tradusă imediat în Haskell utilizând tot o *abstracție lambda*

```
\x y -> x + y
```

Sintaxa Haskell este ceva mai apropiată de cea a **calculului Lambda**: \ anunță parametrii (separați prin spații), iar `→` precizează corpul funcției.

În Racket, puteam da nume funcției folosind legarea dinamică prin intermediul lui `define` sau legarea statică prin intermediul lui `let`. Exemplele următoare ilustrează cele 2 cazuri, împreună cu apelul funcției.

```
(define f (lambda (x y) (+ x y)))
(f 2 3)

(let ((f (lambda (x y) (+ x y)))) (f 2 3))
```

În Haskell, funcția echivalentă ar fi

```
f = \x y -> x + y
```

sau

```
f x y = x + y
```

Și apelul

```
f 2 3
```

Observați că se renunță la paranteze. Unul dintre principiile dezvoltării limbajului a fost oferirea de construcții cât mai simple pentru lucrurile folosite cel mai des. Fiind vorba de programarea funcțională, aplicarea de funcții trebuia făcută cât mai simplu posibil.

Tot din același considerent, operatorii în Haskell sunt scriși în forma **infixată**. Restul apelurilor de funcții sunt în forma prefixată, exact ca în Racket. Totuși, se pot folosi ambele forme: pentru a folosi un operator prefixat se folosesc paranteze, în timp ce pentru a infixa o funcție se folosesc *back-quotes* (```).

```
2 + 3 == (+) 2 3
elem 2 [1,2,3] == 2 `elem` [1, 2, 3]
```

Tot pentru simplitate, Haskell permite folosirea unor secțiuni – elemente de zăhărel sintactic care se vor traduce în funcții anonime:

```
(2 +) == \x -> 2 + x
(+ 2) == \x -> x + 2
(- 2) == -2
(2 -) == \x -> 2 - x
```

Atenție: În construcția `(- x)` operatorul `-` este unar, nu binar (este echivalentul funcției negare).

Tipuri de bază

În această secțiune vom prezenta tipurile existente în limbajul Haskell. Veți observa că limbajul este mult mai bogat în tipuri decât Racket. Programatorul își va putea defini alte tipuri proprii dacă dorește.

Pentru a putea vedea tipul unei expresii în ghci folosiți `:t expresie`.

```
> :t 'a'
'a' :: Char
```

Operatorul `::` separă o expresie de tipul acesteia.

```
> :t (42::Int)
(42::Int) :: Int
```

Numere, caractere, siruri, booleeni

Urmatorii literalii sunt valizi in **Haskell** (dupa operatorul `::` sunt precizate tipurile acestora):

```
5      :: Int
'H'    :: Char
"Hello" :: String -- sau [Char] -- lista de Char
True   :: Bool
False  :: Bool
```

Observați că există tipul caracter și tipul șir de caractere. Tipul `String` este de fapt un sinonim pentru tipul `[Char]` - tipul listă de caractere. Astfel, operațiile pe liste vor funcționa și pe șiruri.

Tipurile numerice sunt puțin diferite față de alte limbaje de programare cunoscute:

```
> :t 42
42 :: Num a => a
```

În exemplul de mai sus, `a` este o variabilă de tip (stă pentru orice fel de tip) restricționată (prin folosirea `=>`) la toate tipurile numerice (`Num a`).

Important de reținut este faptul că există 2 tipuri întregi: `Int` și `Integer`. Primul este finit, determinat de arhitectură, în timp ce al doilea este infinit, putând ajunge oricât de mare.

Liste

O listă în Haskell se construiește similar ca în Racket, prin intermediul celor doi constructori:

```
[] -- lista vidă
(:) -- operatorul de adăugare la începutul listei - cons
```

Pentru simplitate, o construcție de forma

```
1:3:5:[]
```

se poate scrie și

```
[1, 3, 5]
```

sau

```
[1, 3 .. 6]
```

sau

```
[1, 3 .. 5]
```

Echivalente pentru `car` și `cdr` din Racket sunt funcțiile `head` și `tail`:

```
> head [1, 2, 3]
1
> tail [1, 2, 3]
[2, 3]
```

Operatorul de **concatenare** este `++`:

```
> [1, 2, 3] ++ [4, 5]
[1, 2, 3, 4, 5]
```

Haskell oferă un mod suplimentar de a genera liste: scriem proprietățile pe care ar trebui să le respecte elementele listei într-o sintaxă numită **list comprehension**. Este o sintaxă similară celei din matematică. De exemplu, vrem lista numerelor pare, divizibile cu 3. În matematică, am fi avut ceva de tipul $\{x \mid x \in \mathbb{N}_2, x \equiv 0 \pmod{3}\}$ (pentru \mathbb{N}_2 mulțimea numerelor pare). În Haskell, avem

```
> [x | x <- [0, 2 ..], x `mod` 3 == 0] -- lista numerelor naturale pare, divizibile cu 3
[0,6,12,18,24,30,36,42,48,54,60,66,72,78,84,90,96,102,108, Interrupted.]
```

Observați că se generează elemente la infinit. Pentru fiecare element din lista `[0, 2 ..]` (din expresia `x <- [0, 2 ..]`) se testează condițiile următoare. Dacă toate sunt îndeplinite, se generează elementul din fața `|`.

Pentru a putea vedea o porțiune a fluxului folosim funcțiile `take` și `drop`.

Perechi

În Haskell, elementele unei liste **sunt de același tip**. Dacă dorim construcții cu elemente de tipuri diferite vom folosi tuplurile

```
(3, "Ana") :: (Int, String)
(3, True, "Ana") :: (Int, Bool, String)
```

Haskell oferă funcții pentru extragerea componentelor din perechi

```
> fst (3, "Ana")
3
> snd (3, "Ana")
"Ana"
```

Se ofera de asemenea și funcțiile `zip` și `unzip`, **doar** pentru perechi, dar există și `zip3` și `unzip3` pentru triplete.

Pentru tuplurile cu mai mult de 2 elemente este sarcina programatorului să definească funcțiile folosite.

Definirea funcțiilor

Să considerăm că vrem să scriem o funcție pentru factorialul unui număr. Prima implementare ar folosi sintaxa `if`:

```
factorial_if x = if x < 1 then 1 else x * factorial_if (x - 1)
```

Este obligatoriu ca `if` să conțină **ambele** ramuri și ca acestea să fie **de același tip**.

Mai frumos, putem scrie funcția de mai sus folosind gărzi:

```
factorial_guards x
| x < 1 = 1
| otherwise = x * factorial_guards (x - 1)
```

Observați indentarea: orice linie care face parte din aceeași expresie ca cea de deasupra trebuie să înceapă la exact aceeași indentare. Orice linie care este o subexpresie a expresiei de mai sus (then sau else în cazul if, fiecare gardă în parte, etc.) trebuie să fie indentată mai spre dreapta.

Construcția otherwise este echivalentă expresiei True. O gardă poate conține orice fel de expresie care se va evalua la o valoare de tip Bool.

Fiecare gardă este testată în ordine, prima adevărată este executată.

Pentru a ne apropia de definiția matematică, putem scrie aceeași funcție folosind case (echivalentul switch-ului din C dar mult mai avansat):

```
factorial_case x = case x < 1 of
  True -> 1
  _ -> x * factorial_case (x - 1)
```

Observați regula indentării aplicată și aici. Expresia _ semnifică orice valoare, indiferent de valoarea ei. Expresia din case poate fi oricare.

Fiecare caz este tratat în ordine, prima potrivire este executată.

În final, putem scrie aceeași funcție folosind **pattern matching**:

```
factorial_pm 0 = 1
factorial_pm x = x * factorial_pm (x - 1)
```

Expresiile din interiorul fiecărei ramuri din case sau din interiorul pattern-match nu pot fi decât constructorii unui tip (similar cu ce ați făcut la AA). În continuare vom folosi cele 4 stiluri pentru a ilustra funcția care calculează lungimea unei liste

```
length_if l = if l /= [] then 1 + length_if (tail l) else 0
```

```
length_guard l
| l /= [] = 1 + length_guard (tail l)
| otherwise = 0
```

```
length_case l = case l of
  (_ : xs) -> 1 + length_case xs
  _ -> 0
```

```
length_pm [] = 0
length_pm (_:xs) = 1 + length_pm xs
```

Observați în exemplele de mai sus expresivitatea limbajului. Folosirea construcției potrivite duce la definiții scurte și ușor de înțeles.

Curry vs Uncurry

Fiind un limbaj funcțional, în Haskell funcțiile sunt valori de prim rang. Astfel, putem afla tipul unei funcții:

```
f x y = x + y

> :t f
f :: Num a => a -> a -> a
```

Fiecare argument este separat prin → de următorul sau de rezultat.

Amintindu-ne de discuția despre funcții curry și uncurry, rezultatul următor nu trebuie să ne surprindă

```
> :t f 3
f 3 :: Num a => a -> a
```

Toate funcțiile din Haskell sunt în formă **curry**.

Pentru a face o funcție uncurry putem folosi funcția `uncurry :: (a -> b -> c) -> (a, b) -> c` dacă vrem să transformăm o funcție echivalentă sau putem s-o definim în mod direct folosind perechi.

Funcționale uzuale

Haskell oferă un set de funcționale (similar celui din Racket) pentru cazuri de folosire des întâlnite (dacă nu mai știți ce fac, încercați să ghiciți citind semnătura și numele ca o documentație):

```
> :t map
map :: (a -> b) -> [a] -> [b]
> :t filter
filter :: (a -> Bool) -> [a] -> [a]
> :t foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

Folosirea lor duce la un cod mai ușor de citit și de întreținut.

Point-free programming

În Haskell, compunerea funcțiilor se realizează cu ajutorul operatorului `..`. De interes este și operatorul `$` definit ca:

```
f $ x = f x
```

El are avantajul de a grupa expresiile din dreapta și stânga lui înainte de aplicarea funcției, scăpând astfel de paranteze:

```
length $ 3 : [1, 2] -- length (3 : [1, 2])
```

Împreună cu `curry`, `uncurry` și `flip` aceste funcții duc la un stil de programare în care valoarea în care se evaluează funcția nu este prezentă. Urmăriți exemplul următor de transformare

```
square x = x*x
inc x = x+1
f1 x = inc (square x)
f2 x = inc $ square x
f3 x = inc . square $ x
f4 = inc . square
```

Stilul are câteva avantaje în domeniul expresivității și al verificării programului dar poate duce ușor la cod obfuscant.

Resurse

- [Exerciții](#)
- [Cheatsheet](#)
- [Soluții](#)

Referințe

- *Learn You a Haskell* [<http://learnyouahaskell.com/chapters/>], este utilă pentru toate laboratoarele de Haskell plus ceva extra
- *Hoogle* [<http://www.haskell.org/hoogle/>], search for Haskell functions starting from types
- *Hackage* [<http://hackage.haskell.org/packages/hackage.html>], repository pentru pachete Haskell

- *Pointfree on Haskell Wiki* [<http://www.haskell.org/haskellwiki/Pointfree>]
- *Pointfree Style - Whay is it good for* [<http://buffered.io/posts/point-free-style-what-is-it-good-for/>]
- *Advantages of point-free on Lambda the Ultimate* [<http://lambda-the-ultimate.org/node/3233>]

20/laboratoare/haskell/intro.txt · Ultima modificare: 2020/04/10 15:40 de către Bogdan Popa

Haskell: Legarea variabilelor. Structuri de date infinite. Funcționale

- Responsabil: Tiberiu Lepadatu [<mailto:tiberiulepadatu14@gmail.com>]
- Data publicării: 30.03.2020
- Data ultimei modificări: 10.04.2020

Obiective

Scopul acestui laborator îl reprezintă acomodarea cu noțiuni avansate de programare în Haskell, având în vedere scrierea de cod **clar și concis**.

Aspectele urmărite sunt:

- domenii de vizibilitate ale definițiilor: top-level și local
- definirea de structuri de date infinite
- funcționalele ca șabloane de proiectare
- programare „point-free“

Domenii de vizibilitate

Spre deosebire de Racket, unde legarea variabilelor la nivelul cel mai de sus (top-level) este dinamică, Haskell leagă definițiile **static**, acestea fiind vizibile implicit la nivel **global**. De exemplu, o definiție de forma:

```
theAnswer = 42
```

va putea fi utilizată implicit în toate fișierele încărcate de către compilator/interpretor la un moment dat. Domeniul de vizibilitate al definițiilor top-level poate fi însă redus cu ajutorul definirii de module: consultați capitolul 11 din A Gentle Introduction to Haskell [<http://www.haskell.org/tutorial/modules.html>] pentru mai multe detalii.

La fel ca Racket, Haskell permite definirea în cadrul domeniilor de vizibilitate locală (mai exact în cadrul funcțiilor), cu ajutorul clauzelor `let` și `where`.

let

Forma generală a clauzei `let` este următoarea:

```
let id1 = val1
    id2 = val2
    ...
    idn = valn
in expr
```

unde `expr` este o expresie Haskell care poate depinde de `id1`, `id2`, ..., `idn`. De asemenea, domeniul de vizibilitate ale definițiilor locale este întreaga clauză `let` (similar cu `letrec` în Racket). Astfel, definiția următoare:

```
p = let x = y + 1
      y = 2
      b n = if n == 0 then [] else n : b (n - 1)
in (x + y, b 2)
```

este corectă. x poate să depindă de y datorită **evaluării leneșe**: în fapt x va fi evaluat în corpul clauzei, în cadrul expresiei $(x + y, b^2)$, unde y e deja definit.

where

Clauza `where` este similară cu `let`, diferența principală constând în folosirea acestuia **după** corpul funcției. Forma generală a acesteia este:

```
def = expr
  where
    id1 = val1
    id2 = val2
    ...
    idn = valn
```

cu aceleași observații ca în cazul `let`.

Un exemplu de folosire este implementarea metodei de sortare QuickSort:

```
qsort [] = []
qsort (p : xs) = qsort left ++ [p] ++ qsort right
  where
    left = filter (< p) xs
    right = filter (>= p) xs
```

Clauzele de tip `let` și `where` facilitează **reutilizarea** codului. De exemplu, funcția:

```
inRange :: Double -> Double -> String
inRange x max
  | f < low           = "Too low!"
  | f >= low && f <= high = "In range"
  | otherwise         = "Too high!"
  where
    f = x / max
    (low, high) = (0.5, 1.0)
```

verifică dacă o valoare normată se află într-un interval fixat. Expresia dată de f este folosită de mai multe ori în corpul funcției, motiv pentru care este urmărită încapsularea ei într-o definiție. De asemenea, se observă că definițiile locale, ca și cele top-level, permit pattern matching-ul pe constructorii de tip, în cazul acesta constructorul tipului pereche.

Observăm că `where` și `let` sunt de asemenea utile pentru definirea de funcții auxiliare:

```
naturals = iter 0
  where iter x = x : iter (x + 1)
```

`iter` având în exemplul de mai sus rolul de generator auxiliar al listei numerelor naturale.

Structuri de date infinite, funcționale

După cum am observat în cadrul laboratorului introductiv, Haskell facilitează definirea structurilor de date infinite, dată fiind natura **leneșă** a evaluării expresiilor. De exemplu mulțimea numerelor naturale poate fi definită după cum urmează:

```
naturals = [0..]
```

Definiția de mai sus este însă **zahăr sintactic** pentru următoarea expresie (exemplificată anterior):

```
naturals = iter 0
  where iter x = x : iter (x + 1)
```

Putem de asemenea să generăm liste infinite folosind funcționala `iterate` [<http://hackage.haskell.org/package/base-4.6.0.1/docs/Prelude.html#v:iterate>], având prototipul:

```
> :t iterate
iterate :: (a -> a) -> a -> [a]
```

`iterate` primește o funcție `f` și o valoare inițială `x` și generează o listă infinită din aplicarea repetată a lui `f`. Implementarea listei numerelor naturale va arăta deci astfel:

```
naturals = iterate (\ x -> x + 1) 0
```

Observăm că `iterate` este nu numai o funcțională, ci și un **șablon de proiectare** (design pattern). Alte funcționale cu care putem genera liste infinite sunt:

- `repeat`: repetă o valoare la infinit
- `intersperse`: introduce o valoare între elementele unei liste
- `zipWith`: generalizare a lui `zip` care, aplică o funcție binară pe elementele a două liste; e completată de `zipWith3` (pentru funcții ternare), `zipWith4` etc.
- `foldl`, `foldr`, `map`, `filter`

Exemple de utilizare:

```
ones = repeat 1 -- [1, 1, 1, ..]
onesTwo = intersperse 2 ones -- [1, 2, 1, 2, ..]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs) -- sirul lui Fibonacci
powsOfTwo = iterate (* 2) 1 -- puterile lui 2
palindromes = filter isPalindrome [0..] -- palindroame
  where
    -- truc: reprezintă numărul ca String
    isPalindrome x = show x == reverse (show x)
```

Point-free programming

"Point-free style" [<https://wiki.haskell.org/Pointfree>] reprezintă o paradigmă de programare în care evităm menționarea explicită a parametrilor unei funcții în definiția acesteia. Cu alte cuvinte, se referă la scrierea unei funcții ca o succesiune de compuneri de funcții. Această abordare ne ajută, atunci când scriem sau citim cod, să ne concentrăm asupra obiectivului urmărit de algoritm deoarece expunem mai transparent ordinea în care sunt efectuate operațiile. În multe situații, codul realizat astfel este mai compact și mai ușor de urmărit.

De exemplu, putem să definim operația de însumare a elementelor unei liste în felul următor:

```
> let sum xs = foldl (+) 0 xs
```

Alternativ, în stilul „point-free“, vom evita descrierea explicită a argumentului funcției:

```
> let sum = foldl (+) 0
```

Practic, ne-am folosit de faptul că în Haskell toate funcțiile sunt în formă `curry` [<https://wiki.haskell.org/Currying>], aplicând parțial funcția `foldl` pe primele două argumente, pentru a obține o expresie care așteaptă o listă și produce rezultatul dorit.

Pentru a compune două funcții, vom folosi operatorul `.`:

```
> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Dacă analizăm tipul operatorului, observăm că acesta primește ca argumente o funcție care acceptă o intrare de tipul `b` și întoarce o valoare de tipul `c`, respectiv încă o funcție care primește o intrare de tipul `a` și produce o valoare de tipul `b`, compatibilă cu intrarea așteptată de prima funcție. Rezultatul întors este o funcție care acceptă valori de tipul `a` și produce rezultate de tipul `c`.

Cu alte cuvinte, expresia $(f \circ g)(x)$ este echivalentă cu $f(g(x))$.

Spre exemplu, pentru a calcula expresia $2x+1$ pentru orice element dintr-o listă, fără a folosi o funcție lambda, putem scrie direct:

```
> let sm = map ((+ 1) . (* 2))
> :t sm
sm :: [Integer] -> [Integer]
```

De observat că au fost necesare paranteze pentru ca întreaga expresie $(+ 1) . (* 2)$ să reprezinte un singur argument pentru funcția `map`. Am putea obține un cod mai concis folosind operatorul `$`:

```
> let sm = map $ (+ 1) . (* 2)
> :t ($)
($) :: (a -> b) -> a -> b
```

`$` este un operator care aplică o funcție unară pe parametrul său. Semantica expresiei $f \$ x$ este aceeași cu cea a $f x$, diferența fiind pur sintactică: precedența `$` impune ordinea de evaluare astfel încât expresia din stânga `$` este aplicată pe cea din dreapta. Avantajul practic este că putem să ometem parantezele în anumite situații, în general atunci când ar trebui să plasăm o paranteză de la punctul unde se află `$` până la sfârșitul expresiei curente.

De exemplu, expresiile următoare sunt echivalente:

```
> length (tail (zip [1,2,3,4] ("abc" ++ "d")))
> length $ tail $ zip [1,2,3,4] $ "abc" ++ "d"
```

Alternativ, folosind operatorul de compunere `.`:

```
> (length . tail . zip [1,2,3,4]) ("abc" ++ "d")
> length . tail . zip [1,2,3,4] $ "abc" ++ "d"
```

Atenție! Cei doi operatori, `.` și `$` nu sunt, în general, interschimbabili:

```
> let f = (+ 1)
> let g = (* 2)
> :t f . g
f . g :: Integer -> Integer
> :t f $ g
-- eroare, f așteaptă un Integer, g este o funcție
> f . g $ 2
5
> f . g 2
-- eroare, echivalent cu f . (g 2), f . g (2)
> f $ g $ 2
5
> f $ g 2
5
```

Uneori, ne dorim să schimbăm ordinea de aplicare a argumentelor pentru o funcție. Un exemplu de funcție care ne-ar putea ajuta în acest sens este funcția `flip`, care interschimbă parametrii unei funcții binare:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

De exemplu, dacă vrem să construim o funcție point-free care aplică o funcție primită ca argument pe fluxul numerelor naturale, am putea scrie:

```
> :t map
map :: (a -> b) -> [a] -> [b]
> :t flip map
flip map :: [a] -> (a -> b) -> [b]
> let f = flip map $ [0..]
> take 10 $ f (*2)
[0,2,4,6,8,10,12,14,16,18]
```

Un șablon de proiectare des folosit este `concatMap`, sau `map/reduce` [<https://en.wikipedia.org/wiki/MapReduce>], care implică compunerea funcționalelor `map` și `fold`. De exemplu, funcția `intersperse`, prezentată anterior, poate fi definită sub forma:

```
myIntersperse :: a -> [a] -> [a]
myIntersperse y = foldr (++) [] . map (: [y])
```

Observăm că al doilea argument al funcției `myIntersperse` nu este menționat explicit.

Atenție! Folosirea stilului „point-free“ poate să scadă în anumite cazuri lizibilitatea codului! Ideal este să folosiți construcții „point-free“ doar atunci când acestea fac programul **mai ușor** de înțeles. Proiectele Haskell de dimensiuni mari încurajează stilul „point-free“ **doar** împreună cu clauze `let` și `where`, și uneori cu funcții anonime.

Un exemplu de aplicație uzuală de „point-free style programming“, cu care probabil sunteți deja familiari, este folosirea operatorului pipe (`,`) în `unix shell scripting` [<http://www.vex.net/~trebla/weblog/pointfree.html>].

Resurse

- [Exerciții](#)
- [Cheatsheet](#)
- [Soluții](#)

Referințe

- *Let vs Where* [https://wiki.haskell.org/Let_vs._Where]
- *Haskell: function composition (.) vs function application (\$) (SO)* [<http://stackoverflow.com/questions/3030675/haskell-function-composition-and-function-application-idioms-correct-us>]
- *Pointfree (Haskell Wiki)* [<http://www.haskell.org/haskellwiki/Pointfree>]
- *Higher order function (Haskell Wiki)* [http://www.haskell.org/haskellwiki/Higher_order_function]
- *Learn You a Haskell* [<http://learnyouahaskell.com/chapters/>]
- *Real World Haskell* [<http://book.realworldhaskell.org/read/>]
- *Hoogle* [<http://www.haskell.org/hoogle/>]
- *Hayoo!* [<http://holumbus.fh-wedel.de/hayoo/hayoo.html>]
- *Hackage* [<http://hackage.haskell.org/packages/hackage.html>]
- *Local definitions* [http://en.wikibooks.org/wiki/Haskell/Variables_and_functions#Local_definitions]
- *Comparație* [<http://www.lambdadays.org/static/upload/media/14254629275479beameraghfuneval2.pdf>] între diferiți algoritmi de sortare implementați în limbaje funcționale

Haskell: Tipuri de date utilizator

- Responsabil: Mihai Nan [<mailto:mihai.nan.cti@gmail.com>]
- Data publicării: 08.04.2020
- Data ultimei modificări: 06.04.2018

Obiective

Scopul acestui laborator este introducerea **mecanismului de tipuri** al limbajului Haskell, precum și prezentarea unor modalități de a defini **noi tipuri** de date.

Aspectele urmărite sunt:

- particularități ale tipurilor de date în Haskell
- sinteza de tip
- definirea tipurilor de date utilizator:
 - sinonime de tip
 - tipuri de date enumerate
 - tipuri înregistrare
 - tipuri parametrizate
 - tipuri recursive
 - tipuri izomorfe

Introducere

După cum am discutat în cadrul laboratorului introductiv, limbajul Haskell implementează un **mecanism de tipuri** specific, având la bază tipurile de date algebrice [http://www.haskell.org/haskellwiki/Algebraic_data_type]. Scopul mecanismului este impunerea **corectitudinii** la nivelul programelor, mai exact, garantarea unor proprietăți de corectitudine în funcție de tipurile folosite. Aceasta implică o serie de caracteristici ale limbajului în raport cu sistemul de tipuri.

În primul rând, Haskell este un limbaj **puternic** tipat. Astfel, două tipuri A și B vor fi tratate distinct, conversia între acestea realizându-se **explicit**. De exemplu, în C următoarea secvență de cod:

```
int x = -1;
double y = x;
```

este considerată corectă de către compilator, deși variabilele x și y au tipuri diferite. În Haskell, secvența echivalentă de cod:

```
x :: Int
x = -1

y :: Double
y = x
```

va genera o eroare de tip, fiind necesară folosirea unor funcții de conversie (de exemplu `fromIntegral`) pentru realizarea „cast”-urilor de la un tip la altul.

Observăm că o consecință a tipării puternice o reprezintă imposibilitatea de a defini liste eterogene. De exemplu următoarea expresie rezultă într-un mesaj de eroare:

```
> :t [1, 'a', True]
<interactive>
Couldn't match expected type `Char' with actual type `Bool'
In the expression: True
In the expression: [1, 'a', True]
```

De asemenea, Haskell este tipat **static**, sau **la compilare**: după cum am observat și în exemplul anterior, programul nu va compila decât dacă programul este lipsit de erori la nivel de tip. Raționamentul este acela că tipurile de date reprezintă principala metodă de **abstractizare** în limbajele de programare, astfel că, dacă semantica programelor este corectă, atunci corectitudinea implementării va decurge din aceasta. Evident, afirmația nu e general valabilă, printre altele datorită faptului că Haskell acceptă implementarea funcțiilor parțiale. De exemplu, în expresia:

```
> head []
*** Exception: Prelude.head: empty list
```

funcția `head` poate fi aplicată în general pe liste, însă aplicarea ei pe lista vidă va genera o eroare **dinamică** (în timpul rulării programului), deoarece nu este posibilă definirea funcției pentru această valoare.

Stabilirea statică a tipurilor este făcută cu ajutorul unui mecanism de **sinteză de tip**: la compilare sunt verificate tipurile tuturor expresiilor, compilarea terminându-se cu succes doar când acestea corespund. Sinteza este efectuată pe tipuri de date oricât de complexe, astfel că, de exemplu, o expresie `expr` având tipul:

```
expr :: [(a,Int)]
```

va fi verificată în adâncime, de la „rădăcină” (tipul listă) către „frunze” (variabila de tip `a`, tipul `Int`).

În continuare, vom studia construcțiile sintactice Haskell care ne permit definirea tipurilor de date utilizator.

type

Construcția `type` ne permite definirea unui **sinonim** de tip, similar cu `typedef` din C. De exemplu:

```
type Point = (Int, Int)
```

Putem astfel să declarăm o definiție de forma:

```
p :: Point
p = (2, 3)
```

Observăm că Haskell nu face distincția între constructorul perechii `(2, 3)` și constructorul `Point`, cele două tipuri fiind identice. Singura restricție este aceea că valorile perechii trebuie să fie de tip `Int`, astfel că expresia:

```
p2 :: Point
p2 = (2.0, 3.0)
```

va genera o eroare de tip, deoarece `Point` este identic cu `(Int, Int)`, iar valorile `2.0`, respectiv `3.0`, au tipuri fracționare.

data

Construcția `data` permite definirea de noi tipuri de date algebrice, având următoarea formă:

```
data NumTip = Constructor1 | Constructor2 | .. | ConstructorN
```

Observăm distincția între *numele tipului* (denumit și *constructor de tip*), care poate fi folosit în expresii de tip (spre exemplu, `expr :: NumTip`), și *numele constructorilor* (denumiți și *constructori de date*), acestea fiind folosite în definiții, cum ar fi `expr = Constructor1`. De exemplu:

```
data PointT = PointC Double Double deriving Show
```

definește tipul `PointT` prin constructorul `PointC`, construit pe baza unei perechi de `Double`. Cele două nume sunt **distincte** din punctul de vedere al limbajului, însă pot fi suprapuse. De exemplu, un punct în trei dimensiuni poate fi definit astfel:

```
data Point3D = Point3D Double Double Double deriving Show
```

În Haskell, constructorii de date sunt reprezentați ca funcții. Dacă inspectăm tipul constructorilor definiți anterior, vom obține:

```
> :t PointC
PointC :: Double -> Double -> PointT
> :t Point3D
Point3D :: Double -> Double -> Double -> Point3D
```

De asemenea, putem consulta tipurile constructorilor definiți implicit de către limbaj:

```
> :t (,)
(,) :: a -> b -> (a, b)
> :t []
[] :: [a]
> :t (:)
(:) :: a -> [a] -> [a]
```

Tipuri enumerate

`data` permite declararea de tipuri enumerate, similare cu construcția `enum` din C. De exemplu:

```
data Colour = Red | Green | Blue | Black deriving Show
```

Observăm faptul că această construcție permite pattern matching-ul pe constructorii tipului:

```
nonColour :: Colour -> Bool
nonColour Black = True
nonColour _     = False
```

De asemenea, e util de menționat faptul că sintaxa `|` denotă o sumă algebrică la nivel de tipuri, fiind în acest sens asemănătoare cu construcția `union` din C.

Tipuri înregistrare

Putem redefini tipul anterior `PointT` pentru a arăta după cum urmează:

```
data PointT = PointC
  { px :: Double
  , py :: Double
  } deriving Show
```

Definiția este semantic identică cu cea anterioară, singura diferență fiind asocierea unor **nume** câmpurilor structurii de date. Aceasta duce la definiția implicită a două funcții, `px` și `py`, având următoarea semnătură:

```
> :t px
px :: PointT -> Double
> :t py
py :: PointT -> Double
```

Acestea au rolul de a selecta valorile asociate fiecărui câmp în parte, având deci implementarea implicită:

```
px (PointC x _) = x
py (PointC _ y) = y
```

Numele câmpurilor pot fi folosite și pentru „modificarea” selectivă a câmpurilor unui obiect. De exemplu pentru `p` de tipul `PointT`, următorul cod va crea un nou `PointT` al cărui câmp `px` va avea valoarea 5, restul câmpurilor având aceleași valori ca pentru `p`.

```
newP = p { px = 5 }
```


Tipuri parametrizate

Haskell ne permite crearea de tipuri care primesc ca parametru un alt tip. De exemplu tipul de date `Maybe` [<http://www.haskell.org/haskellwiki/Maybe>] are următoarea definiție:

```
data Maybe a = Just a | Nothing deriving (Show, Eq, Ord)
```

unde `a` este o variabilă de tip. Acesta are doi constructori, `Just` și `Nothing`, tipurile acestora fiind:

```
> :t Just
Just :: a -> Maybe a
> :t Nothing
Nothing :: Maybe a
```

Observăm că valorile de tip `Maybe a` pot fie să încapsuleze o valoare de tipul `a`, fie să nu conțină nimic, în mod similar cu tipul `void` din C. Această structură ne este utilă atunci când lucrăm cu funcții care pot eșua în a întoarce o valoare utilă. De exemplu, putem folosi `Maybe` pentru a reimplementa funcția `head` în așa fel încât să evităm excepțiile dinamice apărute de aplicarea funcției pe lista vidă:

```
maybeHead :: [a] -> Maybe a
maybeHead (x : _) = Just x
maybeHead _      = Nothing
```

Observație: Parametrizarea la nivel de tip poate fi efectuată și în cazul construcțiilor `type` și `newtype` (prezentată mai jos), în mod similar cu `data`.

Tipuri recursive

Haskell permite **recurența** la nivel de tip, mai exact referirea tipului declarat la un moment dat în cadrul propriilor constructori. Astfel, putem defini tipul listă în următorul fel:

```
data List a = Void | Cons a (List a) deriving Show
```

Această construcție este de fapt implicit prezentă în Haskell, ca **zahăr sintactic**:

```
data [a] = [] | a : [a] deriving Show
```

Un alt exemplu este definirea mulțimii numerelor naturale în aritmetica Peano:

```
data Natural = Zero | Succ Natural deriving Show
```

newtype

Construcția `newtype` este similară cu `data`, cu diferența că ne permite crearea unui tip de date cu **un singur** constructor, pe baza altor tipuri de date existente. De exemplu:

```
newtype Celsius = MakeCelsius Float deriving Show
```

sau

```
newtype Celsius = MakeCelsius { getDegrees :: Float } deriving Show
```

folosind sintaxa de tip înregistrare.

Observăm că `newtype`, spre deosebire de `type`, creează un **nou tip**, nu un tip identic. Acest lucru ne este util când dorim să forțăm folosirea unui anumit tip cu o semantică dată. De exemplu atât `Celsius` cât și `Fahrenheit` pot fi reprezentate ca `Float`, însă acestea sunt tipuri de date diferite:

```
newtype Fahrenheit = MakeFahrenheit Float deriving Show
```

```
celsiusToFahrenheit :: Celsius -> Fahrenheit
celsiusToFahrenheit (MakeCelsius c) = MakeFahrenheit $ c * 9/5 + 32
```

Diferența principală între `data` și `newtype` este că `newtype` permite crearea de tipuri **izomorfe**: atât `Celsius` cât și `Fahrenheit` sunt tipuri identice cu `Float` din punctul de vedere al structurii, însă folosirea lor în cadrul programului diferă, `Float` având o semantică mai generală (orice număr în virgulă mobilă).

Resurse

Citiți exercițiile rezolvate în fișierul `lab8-doc.hs`. Apoi, rezolvați exercițiile din fișierul `lab8-ex.hs`.

- [Exerciții propuse și rezolvate](#)
- [Cheat sheet](#)

Referințe

- [Algebraic data type](http://www.haskell.org/haskellwiki/Algebraic_data_type) [http://www.haskell.org/haskellwiki/Algebraic_data_type]
- [Haskell Wikibook](http://en.wikibooks.org/wiki/Haskell/Type_declarations) [http://en.wikibooks.org/wiki/Haskell/Type_declarations] - Declararea tipurilor
- [Constructor](https://wiki.haskell.org/Constructor) [<https://wiki.haskell.org/Constructor>] - Distincție între constructori de tip și constructori de date
- [Learn you a Haskell](http://learnyouahaskell.com/making-our-own-types-and-typeclasses) [<http://learnyouahaskell.com/making-our-own-types-and-typeclasses>] - Capitolul „Making your own types“
- [Real World Haskell](http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html) [<http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html>] - Capitolul „Defining types“

Haskell: Polimorfism și clase

- Responsabil: George Muraru [mailto:murarugeorgec@gmail.com]
- Data publicării: 12.04.2020
- Data ultimei modificări: 12.04.2020

Obiective

Scopul acestui laborator este de a prezenta implementarea polimorfismului în sistemul de tipuri al limbajului Haskell.

Aspectele urmărite sunt:

- polimorfism
- polimorfism parametric
- polimorfism ad-hoc
- clase

Polimorfism

Polimorfismul este un mecanism al limbajului Haskell (și nu doar al acestuia) prin care se poate defini un *set de operații* (interfață comună) pentru mai multe tipuri. Categoriile de polimorfism pe care le vom întâlni cel mai des în Haskell sunt: parametric și ad-hoc.

Polimorfism parametric

Așa cum ați observat în laboratoarele anterioare, Haskell permite definirea de funcții care operează pe structuri de date generice. Să luăm spre exemplu funcția `length`. Aceasta permite calcularea lungimii oricărei liste, indiferent de tipul elementelor din listă. Această proprietate a limbajului poartă denumirea de **polimorfism parametric**.

Fără acesta, am fi nevoiți să avem câte o versiune a funcției `length` pentru fiecare tip de listă în parte. Am avea, pentru liste de întregi `lengthInts :: [Int] → Int`, pentru liste de string-uri `lengthStrings :: [String] → Int`, pentru liste de numere reale `lengthDouble :: [Double] → Int`, și așa mai departe. Și pentru toate aceste funcții definiția este aceeași:

```
length_ [] = 0
length_ (_:xs) = 1 + length_ xs
```

Este evident că lungimea unei liste nu depinde în niciun fel de tipul elementelor din listă. Putem face abstracție de acesta și îl putem înlocui cu o **variabilă de tip**, obținând astfel următorul tip pentru funcția `length`

```
length :: [a] -> Int
```

Semnătura de tip spune: „Pentru orice tip `a`, funcția `length` ia o listă cu elemente din `a` și întoarce un întreg”. Spunem în acest moment că funcția `length` este polimorfică.

O funcție poate avea oricâte variabile de tip în semnătura ei. De exemplu funcția `map` are tipul:

```
map :: (a -> b) -> [a] -> [b]
```

Cu alte cuvinte definiția lui `map` nu depinde în niciun fel de tipul funcției sau de tipul listei de elemente pe care o va traversa și asupra căreia va aplica funcția argument, atâta vreme cât tipul funcției este compatibil cu tipul elementelor din listă.

Această proprietate a limbajului ne permite să implementăm algoritmi generici, aplicabili într-un număr mare de cazuri, încurajând reutilizarea de cod.

Polimorfism ad-hoc

Să analizăm funcția `elem`. `elem` caută un element într-o listă și întoarce `True` dacă lista conține elementul căutat sau `False` altfel:

```
elem _ []      = False
elem x (y:ys) = x==y || elem x ys
```

Tipul acestei funcții este:

```
elem :: Eq a => a -> [a] -> Bool
```

Elementul de noutate este `Eq a =>`. Acesta se numește **constrângere de tip** și apare ca urmare a folosirii funcției `(==)`. Spre deosebire de funcția `length`, care putea fi folosită indiferent de tipul listei, funcția `elem` este generică într-un sens mai restrâns. Ea funcționează doar pentru liste cu elemente care definesc egalitatea (operatorul `==`).

Cu alte cuvinte, `elem` poate fi aplicată pentru o listă de `Int` sau `String`, pentru că știm să definim egalitatea pentru aceste tipuri, dar nu și pentru o listă de funcții. Deși pare neintuitiv, următoarea expresie:

```
elem (+1) [(+1), (*2), (/3)]
```

va produce următoarea eroare la compilare:

```
No instance for (Eq (a0 -> a0))
  arising from a use of `elem'
Possible fix: add an instance declaration for (Eq (a0 -> a0))
In the expression: elem (+ 1) [(+ 1), (* 2), (/ 3)]
```

Cu alte cuvinte, tipul funcție nu aparține clasei `Eq` și deci nu definește `(==)`.

Revenind la definiția lui `elem`, nu numai că această funcție merge doar pe tipuri care definesc operația de egalitate, dar se va comporta diferit pentru fiecare mod în care egalitatea este implementată. Pentru o implementare diferită a egalității pentru numere întregi, de exemplu, vom obține un comportament diferit pentru funcția `elem`. Spunem că `elem` este **polimorfică ad-hoc**.

Ca să înțelegem mai bine acest concept, trebuie să înțelegem conceptul de clasă din Haskell.

Clase

Am întâlnit conceptele de clasă până acum sub forma constrângerilor de tip, așa cum este și cazul funcției `elem`. Clasele (eng. Type class) sunt un concept esențial în Haskell și unul dintre punctele forte ale limbajului.

Primul lucru de remarcat este că noțiunea de clasă din Haskell și noțiunea de clasă din Java sunt concepte foarte diferite. Nu încercați să înțelegeți una în funcție de cealaltă – asemănarea celor două denumiri este o coincidență.

Clasele din Haskell seamănă mai mult cu conceptul de interfață din Java. O clasă reprezintă un set de funcții care definesc o interfață sau un comportament unitar pentru un tip de date.

Să luăm exemplul clasei `Eq`, întâlnită mai sus. Ea este definită astfel:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

`Eq` definește 2 funcții, `(==)` și `(/=)`. Pentru a înrola un tip în clasa `Eq`, ambele funcții trebuie implementate.

Echivalentul din Java pentru `Eq` ar fi următorul:

```
interface Eq<A> implements Eq<A>> {
    boolean eq(another: A);
    boolean notEq(another: A);
}
```

În continuare vom analiza cum un tip poate fi înrolat în clasa Eq. În acest scop, vom defini tipul Person:

```
data Person = Person {name :: String, cnp :: Integer}
```

Includem Person în clasa Eq astfel:

```
instance Eq Person where
    Person name1 cnp1 == Person name2 cnp2 = name1 == name2 && cnp1 == cnp2
    p1 /= p2 = not (p1 == p2)
```

Observați că adăugarea Person la Eq se face independent de definiția lui Person – spre deosebire de Java, unde o interfață poate fi implementată de un tip doar la definirea acestuia din urmă.

Această abordare are câteva avantaje:

- decuplarea implementării clasei de definirea tipului – modularitate mai bună
- putem înrola tipuri create anterior în alte biblioteci în clase proaspăt create de noi
- putem defini multiple implementări ale unei clase pentru un același tip de date în module diferite și să importăm doar implementarea care ne interesează într-un anumit caz (însă acest lucru nu este recomandat)

Putem adăuga și tipuri de date generice într-o clasă. Să luăm exemplul tipului BST definit în laboratorul anterior:

```
data BST a = Empty | Node a (BST a) (BST a)
```

Adăugarea acestui tip la clasa Eq se face astfel:

```
instance Eq a => Eq (BST a) where
    Empty == Empty = True
    Node info1 l1 r1 == Node info2 l2 r2 = info1 == info2 && l1 == l2 && r1 == r2
    _ == _ = False

    t1 /= t2 = not (t1 == t2)
```

Observăm că sintaxa $Eq\ a \Rightarrow$ își face din nou apariția. Forma $Eq\ a \Rightarrow Eq\ (BST\ a)$ înseamnă: „dacă a aparține clasei Eq, atunci și tipul (BST a) aparține clasei Eq”. Această constrângere este necesară deoarece pentru a verifica că un nod este egal cu un altul ($Node\ info1\ l1\ r1 == Node\ info2\ l2\ r2$, trebuie să verificăm că informațiile corespunzătoare din noduri sunt egale ($info1 == info2$) – ceea ce înseamnă că elementele stocate în arbore trebuie să aparțină la rândul lor clasei Eq.

Extindere de clase

Haskell permite ca o clasă să extindă o altă clasă. Acest lucru este necesar când dorim ca un tip inclus într-o clasă să fie inclus doar dacă face deja parte dintr-o altă clasă.

De exemplu Ord, care conține tipuri cu elemente care pot fi ordonate (<, >, >=, etc.) este definită astfel:

```
class (Eq a) => Ord a where
    compare :: a -> a -> Ordering
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min :: a -> a -> a

    compare x y = if x == y then EQ
                  else if x <= y then LT
                  else GT

    x < y = case compare x y of { LT -> True; _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x > y = case compare x y of { GT -> True; _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }
```

```

max x y = if x <= y then y else x
min x y = if x <= y then x else y

```

Important de reținut din definiția de mai sus este linia `class (Eq a) => Ord a`. Semnificația aici este: „Dacă `a` este în `Eq`, atunci `a` poate fi în `Ord` dacă definește funcțiile de mai jos.”

Aceasta este o definiție naturală pentru clasa `Ord`. Nu are sens să discutăm despre ordonarea elementelor dintr-un tip dacă acesta nu definește ce înseamnă egalitatea dintre elemente.

Membri implicați

Observăm că de fiecare dată când am definit funcția `(/=)`, am definit-o de fapt ca fiind opusul funcției `(==)`. Acesta poate fi un caz foarte des întâlnit. Ar fi util dacă Haskell ne-ar permite să oferim implementări implicite pentru funcțiile din clase.

Vestea bună e că Haskell pune la dispoziție această facilitate. Iată cum putem defini `Eq`:

```

class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

    a /= b = not (a == b)

```

Acum, când înrolăm un tip în `Eq` e suficient să implementăm doar `(==)`.

```

instance Eq Person where
    Person name1 cnp1 == Person name2 cnp2 =
        name1 == name2 && cnp1 == cnp2

```

Desigur, dacă dorim, putem suprascrie implementarea implicită pentru `(/=)` cu o implementare proprie.

La drept vorbind, în biblioteca standard Haskell (modulul `Prelude`) `Eq` este definit astfel:

```

class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y      = not (x == y)
    x == y      = not (x /= y)

```

Observăm că ambele funcții au implementări implicite. Astfel, atunci când instanțiem `Eq` putem să implementăm fie `(/=)`, fie `(==)`, fie pe ambele.

Clase predefinite

Biblioteca standard `Prelude` oferă un set de clase predefinite, care sunt introduse implicit în programele Haskell. Enumerăm câteva:

- `Ord` – pentru tipuri care pot fi ordonate - definește funcții precum `<`, `>`, `<=`, etc.
- `Show` – pentru tipuri care pot fi reprezentate ca `String`-uri - principala funcție este `show`. Această funcție este folosită și de consola `GHCi` atunci când afișează rezultatele.
- `Read` – inversa lui `Show` - pentru tipuri care pot fi citite din `String`
- `Enum` – pentru tipuri care pot fi enumerate - folosită implicit de construcții de forma `[a..b]` care generează toate elementele între două limite (sau plecând de la un punct de start).
- `Num` – clasă pentru toate tipurile numerice - definește operațiile aritmetice de bază: `(+)`, `(-)`, `(*)`, etc.
- `Integral` – clasă pentru tipurile întregi. (`Int` și `Integer` sunt incluse aici). Definește funcții ca `mod` sau `div`
- `Fractional` – clasă pentru numere reprezentabile ca fracții - definește funcția `(/)`
- `Floating` – clasă pentru numere reale - definește funcții ca `sqrt`, `exp`, `sin`, `cos`, etc.
- `Monad` – definește tipuri care pot reprezenta acțiuni monadice. Mai multe despre monade aici: [Monade](http://book.realworldhaskell.org/read/monads.html) [<http://book.realworldhaskell.org/read/monads.html>]

Deriving

Am observat că implementările noastre pentru clasa `Eq` de mai sus au fost relativ simple. În general implementarea lui `Eq` pentru un tip nou de date presupune verificarea câmp cu câmp a fiecărei componente a noului tip de date. Pentru că este simplu, compilatorul de Haskell poate face asta automat pentru noi, folosind cuvântul cheie `deriving`.

```
data BST a = Empty | Node a (BST a) (BST a) deriving (Eq)
```

va genera o implementare a clasei `Eq` similare cu cea făcută de noi mai sus.

Există mai multe clase care pot fi instanțiate în acest fel: `Ord`, `Enum`, `Bounded`, `Show`, `Read`.

Num

Poate v-ați pus la un moment dat întrebarea ce tip au expresiile numerice simple, cum ar fi expresia `5`. Cu siguranță trebuie să aibă un tip numeric – însă despre ce tip e vorba? `5` este în același timp reprezentarea pentru numărul întreg `5`, pentru numărul real `5.0` sau pentru numărul complex `5+0i`. Deci, despre care `5` discutăm ?

GHCi ne poate raspunde la aceasta intrebare:

```
:t 5
=> 5 :: Num a => a
```

`5` este deci o reprezentare pentru toate tipurile numerice (adică tipurile incluse în clasa `Num`). `5` este o constantă polimorfică. La fel este de exemplu și expresia `1+5` sau `[]` (lista vidă):

```
:t 1+5
=> 1+5 :: Num a => a
```

```
:t []
=> [] :: [t]
```

Haskell nu constrânge constanta `5` la niciun tip numeric concret până când acest lucru nu este evident din context.

Tipuri de ordin superior

Să considerăm următoarea problemă: Dat fiind un arbore binar de tipul `BST a`, să definim o funcție `f` cu tipul `a → b`, care aplicată pe arbore va genera un arbore de tipul `BST b`. Va menține structura arborelui, dar pentru fiecare element dintr-un nod, va aplica funcția `f` asupra celui element.

Pentru că semantic seamănă foarte mult cu `map`, o să-i spunem `treeMap`.

```
treeMap :: (a -> b) -> BST a -> BST b
```

```
treeMap f Empty = Empty
treeMap f (BST info l r) = BST (f info) (treeMap f l) (treeMap f r)
```

Să presupunem că dorim să facem ceva similar și pentru tipul `Maybe`.

```
data Maybe a = Nothing | Just a

maybeMap :: (a -> b) -> Maybe a -> Maybe b
maybeMap f Nothing = Nothing
maybeMap f (Just x) = Just (f x)
```

Observăm un șablon comun între definițiile lui `map`, `treeMap`, `maybeMap`. Semnăturile lor sunt foarte similare. De fapt ele expun același principiu – definesc structuri de date, sau așa-zise „containere”, care pot fi „mapate” – adică putem aplica o funcție pe fiecare din elementele conținute și obținem un nou container cu aceeași formă, dar cu alt conținut.

Întrebarea firească este: putem captura acest concept de containere „mapabile“ într-o clasă? Răspunsul este da. Să-i spunem acelei clase Functor – această clasă va expune o singură metodă, numită `fmap`, care reprezintă generalizarea funcțiilor `map` de mai sus.

Functor poate fi definită în felul următor:

```
class Functor container where
  fmap :: (a -> b) -> container a -> container b
```

Observăm un lucru foarte interesant și anume că putem să definim clase pe tipuri care nu sunt încă complete. Observați că variabila de tip `container` trebuie să fie aplicată pe un tip de date ca să producă un tip de date valid – este deci un constructor de tip oarecare.

Să instanțiem clasa Functor pentru arbori binari:

```
instance Functor BST where
  fmap f Empty = Empty
  fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)
```

Similar putem defini `fmap` pentru liste, `Maybe` și alte structuri de date.

Cele spuse mai sus dezvăluie o caracteristică interesantă a sistemului de tipuri Haskell. De exemplu, o funcție în Haskell este ceva de tipul $a \rightarrow b$, i.e. ia o valoare de tip a și produce o valoare de tip b .

BST se comportă ca o funcție, dar la nivel de tip. Constructorii de tip BST iau ca argument un tip de date și produce alt tip de date. De exemplu, `BST Int` ia tipul `Int` ca argument și produce un tip de date care reprezintă un arbore de întregi. Spunem că BST este un **tip de ordin superior** (higher-order type) - în cazul de față este un **constructor de tip unar**.

Pentru a afla informații despre clase în ghci se poate utiliza comanda `:info <typeclass` unde `typeclass` este clasa despre care dorim să aflăm informații.

Mai multe detalii aici [<http://www.haskell.org/tutorial/classes.html>]

Exerciții

Exercițiile se găsesc în fișerul `PQueue.hs`.

Pentru primul exercițiu, asistentul trebuie să verifice dacă s-au definit cele 2 funcții `toList` și `fromList` utilizând operațiile `isEmpty`, `insert`, `empty`, `top`, `pop`.

- [Cheatsheet](#)
- [Exerciții](#)

Resurse utile

Polimorfism în Haskell [<https://wiki.haskell.org/Polymorphism>]

Constructorii de tip și date [<https://wiki.haskell.org/Constructor>]

Prolog: Introducere

- Responsabil: Vlad Talmaciu [mailto:vlad.talmaciu@gmail.com]
- Data publicării: 22.04.2019
- Data ultimei modificări: 22.04.2019

Obiective

Scopul acestui laborator este introducerea în programarea logică și învățarea primelor noțiuni despre Prolog.

Aspectele urmărite sunt:

- diferențierea dintre aceasta paradigmă și cele studiate anterior
- familiarizarea cu entitățile limbajului: **fapte**, **reguli**, **scopuri**
- sintaxa **Prolog**
- structuri de date
- backtracking
- unificare

SWI-Prolog

Folosim SWI-Prolog, așa cum este detaliat [aici](#).

În cadrul laboratorului, recomandăm folosirea **swipl** în modul următor:

- se rulează comanda `swipl` în terminal, sau direct `swipl fisier.pl` pentru a încărca un fișier existent
 - alternativ se poate folosi comanda `prolog`
- se salvează faptele și regulile într-un fișier cu extensia `.pl`
- pentru a invoca editorul prologului, se folosește comanda:
 - `edit.`, dacă a fost deja încărcat un fișier;
 - `edit(file('new.pl')).`, pentru a crea un fișier nou;
 - `edit('new.pl').` dacă fișierul a fost creat anterior.
- dacă fișierul nu a fost încărcat prin pasarea ca argument în linia de comandă, se încarcă folosind comanda `consult('file.pl').`
- pentru a reîncărca toate fișierele ce au fost modificate de la ultima încărcare se folosește `make.`

Comentarii

Simbolul `%` transformă restul rândului într-un comentariu.

Entitățile limbajului

Limbajul Prolog (al cărui nume provine de la Programmable Logic) este un limbaj **logic**, **descriptiv**, care permite specificarea problemei de rezolvat în termenii unor **fapte** cunoscute despre obiectele universului problemei și ai relațiilor existente între aceste obiecte.

Tot ceea ce nu este cunoscut sau nu poate fi demonstrat este considerat a fi fals (**ipoteza lumii închise**).

Execuția unui program Prolog constă în deducerea implicațiilor acestor **fapte** și **relații**, programul definind astfel o mulțime de consecințe ce reprezintă înțelesul sau semnificația declarativă a programului.

Un program Prolog conține următoarele **entități**:

- **fapte** despre obiecte și relațiile existente între aceste obiecte
- **reguli** despre obiecte și relațiile dintre ele, care permit deducerea (inferarea) de noi fapte pe baza celor cunoscute
- întrebări, numite și **scopuri**, despre obiecte și relațiile dintre ele, la care programul răspunde pe baza faptelor și regulilor existente

Fapte

Faptele sunt **predicate de ordinul întâi** de aritate n , considerate **adevărate**. Ele stabilesc relații între obiectele universului problemei. Numărul de argumente ale faptelor este dat de **aritatea** (numărul de argumente) corespunzătoare a predicatelor.

Exemple de fapte:

```
papagal(coco).  
iubește(mihai, maria).  
iubește(mihai, ana).  
frumoasă(ana).  
bun(gelu).  
deplasează(cub, camera1, camera2).
```

Structuri

Structurile au aceeași sintaxă cu faptele, dar apar ca argumente ale predicatelor.

Exemplu de structură:

```
are(ion, carte(aventuri, 2002)).
```

Scopuri

Obținerea consecințelor sau a rezultatului unui program Prolog se face prin fixarea unor **scopuri** care pot fi **adevărate** sau **false**, în funcție de conținutul **bazei de cunoștințe** Prolog. Scopurile sunt predicate pentru care se dorește aflarea valorii de adevăr în contextul faptelor existente în baza de cunoștințe.

Cum scopurile pot fi văzute ca **întrebări**, rezultatul unui program Prolog este răspunsul la o întrebare (sau la o conjuncție de întrebări). Acest răspuns poate fi afirmativ, **true**, sau negativ, **false** (în alte versiuni de Prolog răspunsul poate fi **yes** sau **no**; sau **true** sau **fail**).

Se va vedea mai târziu că programul Prolog, în cazul unui răspuns afirmativ la o întrebare, **poate furniza și alte informații** din baza de cunoștințe.

Considerând baza de cunoștințe specificată anterior, se pot pune diverse întrebări, cum ar fi:

```
?- iubește(mihai, maria).  
true.  
?- papagal(coco).  
true.  
?- papagal(mihai).  
false.  
?- inalt(gelu).  
false.
```

Variabile

În exemplele prezentate până acum, argumentele **faptelor** și **întrebărilor** au fost obiecte particulare, numite și **constante** sau **atomi simbolici**. Predicatele Prolog, ca orice predicate în logica cu predicate de ordinul I, admit ca argumente și obiecte generice numite **variabile**.

În Prolog, prin convenție, numele argumentelor variabile începe cu **literă mare** iar numele constantelor simbolice începe cu **literă mică**.

O variabilă poate fi **instanțiată** (legată) dacă există un obiect asociat acestei variabile, sau **neinstanțiată** (liberă) dacă nu se știe încă ce obiect va desemna variabila.

Semnul `_` (underscore) desemnează o variabilă a cărei valoare nu interesează.

```
?- papagal(coco).
true.
?- papagal(CineEste).
CineEste = coco
?- deplaseaza(_, DeUnde, Unde).
DeUnde = camera1, Unde = camera2
```

La fixarea unui **scop** Prolog care conține **variabile**, acestea sunt neinstanțiate iar sistemul încearcă satisfacerea acestui scop căutând printre faptele din baza de cunoștințe un fapt care poate identifica cu scopul, printr-o **instanțiere adecvată a variabilelor** din scopul dat. Este vorba de fapt de un proces de **unificare** a predicatului scop cu unul din predicatele fapte existente în baza de cunoștințe.

În exemplul de mai jos există mai multe răspunsuri posibile. Prima soluție este dată de prima unificare și există atâtea soluții câte unificări diferite există.

```
?- iubeste(mihai, X).
```

La realizarea primei unificări se **marchează** faptul care a unificat și care reprezintă prima soluție. La obținerea următoarei soluții, căutarea este reluată de la marcaj în jos în baza de cunoștințe.

Obținerea primei soluții este de obicei numită **satisfacerea scopului** iar obținerea altor soluții, **resatisfacerea scopului**.

La satisfacerea unui scop căutarea se face întotdeauna de la începutul bazei de cunoștințe. La resatisfacerea unui scop, căutarea se face începând de la marcajul stabilit de satisfacerea anterioară a acelui scop.

Sistemul Prolog, fiind un sistem **interactiv**, permite utilizatorului obținerea fie a primului răspuns, fie a tuturor răspunsurilor. În cazul în care, după afișarea tuturor răspunsurilor, un scop nu mai poate fi resatisfăcut, sistemul răspunde **false**.

În exemplul de mai jos, tastând caracterul `“;”` și Enter, cerem o nouă soluție.

```
?- iubeste(mihai, X).
X = maria;
X = ana;
false.
?- iubeste(Cine, PeCine).
Cine = mihai, PeCine = maria;
Cine = mihai, PeCine = ana;
false.
```

Reguli

O regulă Prolog exprimă un fapt care depinde de alte fapte și este de forma:

$S :- S_1, S_2, \dots, S_n.$

Fiecare S_i , $i = 1, n$ și S au forma faptelor Prolog, deci sunt predicate, cu argumente constante, variabile sau structuri. Faptul S care definește regula, se numește **antet de regulă**, iar S_1, S_2, \dots, S_n formează corpul regulii și reprezintă conjuncția de scopuri care trebuie satisfăcute pentru ca antetul regulii să fie satisfăcut.

Fie următoarea bază de cunoștințe:

```
frumoasa(ana).           %1
bun(vlad).               %2
cunoaste(vlad, maria).   %3
cunoaste(vlad, ana).     %4
iubeste(mihai, maria).   %5
iubeste(X, Y):- bun(X), cunoaste(X, Y), frumoasa(Y). %6
```

Se observă definirea atât printr-un fapt(linia 5), cât și printr-o regulă (linia 6) a predicatului iubeste(?Cine, ?PeCine).

Backtracking

Pentru a prezenta mecanismul de backtracking în Prolog, vom folosi baza de fapte dată ca exemplu mai sus.

Vom urmări în continuare pașii realizați pentru satisfacerea scopului:

```
?- iubeste(X, Y).
```

Așa cum am precizat anterior, căutarea pornește de la începutul bazei de cunoștințe. Prima potrivire va fi la linia 5, cu faptul:

```
iubeste(mihai, maria). %5
```

În urma unificării, variabilele *x* și *y* se vor instanția la constantele *mihai* și *maria* și se va realiza un marcaj la linia respectivă. Pentru a încerca resatisfacerea scopului, căutarea va începe acum după marcajul făcut anterior. Următoarea potrivire va fi la linia 6, cu regula:

```
iubeste(X, Y):- bun(X), cunoaste(X, Y), frumoasa(Y). %6
```

La unificarea scopului cu antetul unei reguli, pentru a putea satisface acest scop trebuie satisfăcută regula. Aceasta revine la a satisface toate faptele din corpul regulii, deci conjuncția de scopuri. Scopurile din corpul regulii devin subscopuri a căror satisfacere se va încerca printr-un mecanism identic cu cel al satisfacerii scopului inițial.

În continuare se va încerca satisfacerea scopului *bun(X)*.

Fiind vorba de un nou scop, căutarea va avea loc de la începutul bazei de cunoștințe. Acest scop va unifica cu predicatul de la linia 2:

```
bun(vlad). %2
```

În urma unificării, variabila *x* se va instanția la valoarea *vlad*.

În continuare se va încerca satisfacerea scopului *cunoaste(vlad, Y)*. care va unifica cu faptul de la linia 3 și va duce la legarea variabilei *Y* la *maria*:

```
cunoaste(vlad, maria). %3
```

Ultimul scop care mai trebuie satisfăcut este *frumoasa(maria)*, dar acesta eșuează deoarece nu unifica cu niciunul dintre faptele sau regulile din baza de cunoștințe.

Aici intervine mecanismul de **backtracking**. Se va reveni la scopul satisfăcut anterior, *cunoaste(vlad, Y)*. și se va încerca resatisfacerea acestuia. Dacă aceasta ar eșua, ne-am întoarce la scopul dinaintea sa ș.a.m.d. Acest lucru se va repeta până când se epuizează toate scopurile din corpul regulii, caz în care unificarea scopului inițial cu antetul regulii ar eșua, iar răspunsul ar fi *false*.

În cazul nostru, însă, este necesar un singur pas de întoarcere deoarece *cunoaste(vlad, Y)*. poate fi resatisfăcut prin unificarea cu faptul de la linia 4:

```
cunoaste(vlad, ana). %4
```

În urma unificării, variabila Y va fi legată la ana , iar următorul scop ce va trebui satisfăcut este $frumoasa(ana)$. Acesta va reuși deoarece unifică cu faptul de la linia 1:

```
frumoasa(ana).      %1
```

Astfel am obținut o nouă soluție pentru scopul $iubeste(X, Y)$, având $X = vlad$ și $Y = ana$.

Procesul nu se oprește însă până când nu s-au încercat toate unificările posibile pentru satisfacerea scopului. Tot prin intermediul mecanismului de backtracking, se va reveni la scopurile anterioare și se va observa dacă acestea pot fi resatisfăcute. În exemplul nostru acest lucru nu este posibil, deci răspunsul va fi `false`.

Prin urmare, rezultatul final va fi:

```
?- iubeste(X, Y).
X = mihai, Y = maria;
X = vlad, Y = ana;
false.
```

Operatori

- Aritmetici: $+$ $-$ $*$ $/$
- Relaționali: $=$ $\backslash=$ $<$ $>$ $=<$ $>=$ $:=$

La scrierea expresiei $1+2*(X/Y)$, valoarea acesteia nu este calculată, ci expresia este reținută ca atare. Se poate observa că operatorii $:=$ și is forțază evaluarea unei expresii, pe când $=$ verifică doar egalitatea structurală.

De asemenea, is și $=$ pot primi variabile neinstantiate pe care le instanțiază.

```
?- 1 + 2 := 2 + 1.
true.
```

```
?- 1 + 2 = 2 + 1.
false.
```

```
?- X = 2 + 1.
X = 2+1.
```

```
?- X is 2 + 1.
X = 3.
```

```
?- X := 2 + 1.
ERROR: :=/2: Arguments are not sufficiently instantiated
```

Liste

- Lista vidă: `[]`
- Lista cu elementele a, b, c : `[a,b,c]`
- Lista nevidă: `[Prim|Rest]` – unde variabila `Prim` unifică cu primul element al listei, iar variabila `Rest` cu lista fără acest prim element
- Lista care începe cu n elemente x_1, x_2, \dots, x_n și continuă cu o altă listă `Rest`: `[x1,x2,...,xN|Rest]`

Documentarea predicatelor și a argumentelor

Pentru claritate, antetele predicatelor se scriu sub forma:

- **predicat/nrArgumente**
- **predicat(+Arg1, -Arg2, ?Arg3, ..., +ArgN)**

Pentru a diferenția intrările (+) de ieșiri(-), se prefixează argumentele cu indicatori. Acele argumente care pot fi fie intrări, fie ieșiri se prefixează cu '?'. Instanțierea parametrilor ține de specificarea acestora:

- Arg1 va fi instanțiat atunci când se va încerca satisfacerea p/3
- Arg2 se va instanția odată cu satisfacerea p/3
- Arg3 va putea fi instanțiat sau nu atunci când se va satisface p/3

Puterea generativa a limbajului

Așa cum am văzut anterior scopurile pot fi privite ca întrebări ale căror răspunsuri sunt true sau false. În plus, acest răspuns poate fi însoțit de instanțierile variabilelor din cadrul scopului. Acest mecanism ne ajută să folosim scopurile pentru a obține rezultate de orice tip.

De exemplu, pentru a obține lungimea unei liste putem folosi:

```
% lungime(+Lista,-Lungime)
lungime([],0).
lungime(_|R, N):- lungime(R,N1), N is N1 + 1.

?- lungime([1,2,3],N).
N = 3.
```

În exemplul de mai sus se va încerca satisfacerea scopului `lungime([1,2,3],N)`. prin instanțierea convenabilă a variabilei `N`. În acest caz soluția este unică, dar așa cum am văzut anterior, putem avea situații în care există mai multe unificări posibile. Putem folosi faptul că se încearcă resatisfacerea unui scop în mod exhaustiv pentru a genera multiple rezultate.

În exemplul de mai jos vom defini predicatul **membru(?Elem,+List)** care verifică apartenența unui element la o listă:

```
% membru(?Elem,+Lista)
membru(Elem,[Elem|_]).
membru(Elem,[_|Rest]) :- membru(Elem,Rest).
```

Putem folosi acest predicat pentru a obține un răspuns:

```
?- membru(3,[1,2,3,4,5]).
true .
```

Sau putem să îl folosim pentru a genera pe rând toate elementele unei liste:

```
?- membru(N,[1,2,3]).
N = 1 ;
N = 2 ;
N = 3 ;
false.
```

Inițial, scopul `membru(N,[1,2,3])`. va unifica cu faptul `membru(Elem,[Elem|_])`., în care `Elem = N` și `Elem = 1`, din care rezultă instanțierea `N = 1`. Apoi se va încerca unificarea cu antetul de regulă `membru(Elem,[_|Rest])`, în care `Elem = N`, iar `Rest = [2,3]`. Acest lucru implică satisfacerea unui nou scop, `membru(N,[2,3])`.. Noul scop va unifica, de asemenea, cu faptul de la linia 1, `membru(Elem,[Elem|_])`., din care va rezulta `N = 2`. Asemănător se va găsi și soluția `N = 3`, după care nu va mai reuși nicio altă unificare.

Pentru a exemplifica utilizarea acestui mecanism, vom considera următorul exemplu în care dorim generarea, pe rând, a tuturor permutărilor unei liste:

```
% remove(+Elem,+Lista,-ListaNoua)
remove(E,[E|R],R).
remove(E,[F|R],[F|L]):- remove(E,R,L).

% perm(+Lista,-Permutare)
perm([],[]).
perm([F|R],P):- perm(R,P1), remove(F,P,P1).
```

Observați ca am definit predicatul **remove(+Elem,+Lista,-ListaNoua)**, care șterge un element dintr-o listă. Rolul acestuia în cadrul regulii `perm([F|R],P):- perm(R,P1), remove(F,P,P1)` este, de fapt, de a insera elementul F în permutarea P1. Poziția pe care va fi inserat elementul va fi diferită la fiecare resatisfacere a scopului `remove(F,P,P1)`, ceea ce ne ajută să obținem permutările.

```
?- remove(3,L,[1,1,1]).
```

```
L = [3, 1, 1, 1] ;
```

```
L = [1, 3, 1, 1] ;
```

```
L = [1, 1, 3, 1] ;
```

```
L = [1, 1, 1, 3] ;
```

```
false.
```

```
?- perm([1,2,3],Perm).
```

```
Perm = [1, 2, 3] ;
```

```
Perm = [2, 1, 3] ;
```

```
Perm = [2, 3, 1] ;
```

```
Perm = [1, 3, 2] ;
```

```
Perm = [3, 1, 2] ;
```

```
Perm = [3, 2, 1] ;
```

```
false.
```

Resurse

- [Exemple](#)
- [Cheatsheet](#)
- [Exerciții](#)

Referințe

- [Learn prolog now! \[http://www.learnprolognow.org/\]](http://www.learnprolognow.org/)
- [Logic, Programming, and Prolog \[http://www.ida.liu.se/~ulfni53/lpp/bok/bok.pdf\]](http://www.ida.liu.se/~ulfni53/lpp/bok/bok.pdf)
- [Built-in Predicates \[http://www.swi-prolog.org/pldoc/doc_for?object=section%281,%274%27,swi%28%27/doc/Manual/builtin.html%27%29%29\]](http://www.swi-prolog.org/pldoc/doc_for?object=section%281,%274%27,swi%28%27/doc/Manual/builtin.html%27%29%29)
- [Red Cut vs Green Cut \[http://en.wikipedia.org/wiki/Cut_%28logic_programming%29#Red_Cut\]](http://en.wikipedia.org/wiki/Cut_%28logic_programming%29#Red_Cut)

Prolog: Probleme de căutare în spațiul stărilor

- Responsabil: Andrei Olaru [mailto:cs@andreiolaru.ro]
- Data publicării: 06.05.2019
- Data ultimei modificări: 06.05.2019

Obiective

Scopul acestui laborator este învățarea unor tehnici de rezolvare a problemelor de căutare în spațiul stărilor.

Aspectele urmărite sunt:

- metodele de reprezentare a datelor
- particularitățile diverselor tehnici de rezolvare
- eficiența
- abstractizarea procesului de rezolvare

Căutare în spațiul stărilor

Fie un sistem dinamic care se poate afla într-un număr finit de stări. Definim un graf orientat în care nodurile sunt stări, iar arcele reprezintă posibilitatea de a evolua direct dintr-o stare în alta. Graful conține un set de stări inițiale și un set de stări finale (scopuri). Problema descrisă de sistem va fi rezolvată prin parcurgerea spațiului stărilor până când este găsită o cale de la o stare inițială la o stare finală, în cazul în care problema admite soluție. Căutarea este un mecanism general, recomandat atunci când o metodă directă de rezolvare a problemei nu este cunoscută.

Exemple de strategii de căutare în spațiul stărilor:

- generare și testare
- backtracking
- căutare în adâncime/lățime
- algoritmul A*

Generare și testare

Fie problema colorării a 7 țări de pe o hartă folosind 3 culori. Scopul este acela de a atribui câte o culoare fiecărei țări, astfel încât nicio țară să nu aibă niciun vecin de aceeași culoare cu aceasta. Soluția problemei va fi o listă de atribuiri din domeniul [„r”, „g”, „b”], care desemnează culorile atribuite fiecărei țări (1, 2, 3, 4, 5, 6, 7).

Această strategie se traduce în următorul cod Prolog:

```
all_members([], _). % predicat care verifică că toate elementele din prima listă sunt prezente în a doua
all_members([X|Rest], In) :- member(X, In), all_members(Rest, In).
solve(S) :-
    L = [_|_], length(L, 7), all_members(L, ["r", "g", "b"]),
    safe(S). % predicat care verifică faptul că țările nu au culori identice cu niciun vecin
```

Programul anterior este foarte ineficient. El construiește extrem de multe atribuiri, fără a le respinge pe cele „ilegale” într-un stadiu incipient al construcției.

Backtracking atunci când cunoaștem lungimea căii către soluție

Mecanismul de backtracking ne oferă o rezolvare mai eficientă. Știm că orice soluție pentru problema colorării hărților constă într-o listă de atribuiri a 3 culori, de genul $[X1/C1, X2/C2, \dots, X7/C7]$, scopul programului de rezolvare fiind să instanțieze adecvat variabilele $X1, C1, X2, C2$ etc. Vom considera că orice soluție este de forma $[1/C1, 2/C2, \dots, 7/C7]$, deoarece ordinea țărilor nu este importantă. Fie problema mai generală a colorării a N țări folosind M culori. Definim soluția pentru $N=7$ ca o soluție pentru problema generală a colorării hărților, care în plus respectă template-ul $[1/Y1, 2/Y2, \dots, 7/Y7]$. Semnul „/” este în acest caz un separator, fără nicio legătură cu operația de împărțire.

În Prolog vom scrie:

```
%% Lungimea soluției este cunoscută și fixă.
template([1/_ , 2/_ , 3/_ , 4/_ , 5/_ , 6/_ , 7/_]).
```

```
correct([]):-!.
correct([X/Y|Others]):-
    correct(Others),
    member(Y, ["r", "g", "b"]),
    safe(X/Y, Others).
```

```
solve_maps(S):-template(S), correct(S).
```

Regulă: Atunci când calea către soluție respectă un anumit template (avem de instanțiat un număr finit, predeterminat, de variabile), este eficient să definim un astfel de template în program.

Observație: În exemplul de mai sus am reținut explicit ordinea celor 7 țări. Redundanța în reprezentarea datelor ne asigură un câștig în viteza de calcul (câștigul se observă la scrierea predicatului safe).

Backtracking atunci când calea către soluție admite un număr nedeterminat de stări intermediare

În această situație nu este posibil să definim un template care descrie forma soluției problemei. Vom defini o căutare mai generală, după modelul următor:

```
solve(Solution):-
    initial_state(State),
    search([State], Solution).
```

`search(+StăriVizitate, -Soluție)` definește mecanismul general de căutare astfel:

- căutarea începe de la o stare inițială dată (predicatul `initial_state/1`)
- dintr-o stare curentă se generează stările următoare posibile (predicatul `next_state/2`)
- se testează că starea în care s-a trecut este nevizitată anterior (evitând astfel traseele ciclice)
- căutarea continuă din noua stare, până se întâlnește o stare finală (predicatul `final_state/1`)

```
search([CurrentState|Other], Solution):-
    final_state(CurrentState), !,
    reverse([CurrentState|Other], Solution).
```

```
search([CurrentState|Other], Solution):-
    next_state(CurrentState, NextState),
    \+ member(NextState, Other),
    search([NextState,CurrentState|Other], Solution).
```

Căutare în lățime

Căutarea în lățime este adecvată situațiilor în care se dorește drumul minim între o stare inițială și o stare finală. La o căutare în lățime, expandarea stărilor „vechi” are prioritate în fața expandării stărilor „noi”.

```
do_bfs(Solution):-
    initial_node(StartNode),
    bfs([(StartNode,nil)], [], Discovered),
    extract_path(Discovered, Solution).
```

bfs(+CoadăStărilorNevizitate,+StăriVizitate,-Soluție) va defini mecanismul general de căutare în lățime astfel:

- căutarea începe de la o stare inițială dată care n-are predecesor în spațiul stărilor (StartNode cu părintele nil)
- se generează toate stările următoare posibile
- se adaugă toate aceste stări la coada de stări încă nevizitate
- căutarea continuă din starea aflată la începutul cozii, până se întâlnește o stare finală

Căutare A*

A* este un algoritm de căutare informată de tipul best-first search, care caută calea de cost minim (distanță, cost, etc.) către scop. Dintre toate stările căutate, o alege pe cea care pare să conducă cel mai repede la soluție. A* selectează o cale care minimizează $f(n) = g(n) + h(n)$, unde n este nodul curent din cale, $g(n)$ este costul de la nodul de start până la nodul n și $h(n)$ este o euristică ce estimează cel mai mic cost de la nodul n la nodul final.

```
astar_search(Start, End, Grid, Path) :-
    manhattan(Start, End, H),
    astar(End, [H:Start], [Start:(\"None\", 0)], Grid, Discovered),
    get_path(Start, End, Discovered, [End], Path).
```

astar(+End, +Frontier, +Discovered, +Grid, -Result) va defini mecanismul general de căutare A*, astfel:

- căutarea începe de la o stare inițială dată care n-are predecesor în spațiul stărilor (StartNode cu părintele „None“) și distanța estimată de la acesta până la nodul de final printr-o euristică (exemplu: distanța Manhattan)
- se generează toate stările următoare posibile și se calculează costul acestora adăugând costul acțiunii din părinte până în starea aleasă cu costul real calculat pentru a ajunge în părinte (costul părintelui în Discovered)
- dacă starea aleasă nu este în Discovered sau dacă noul cost calculat al acesteia este mai mic decât cel din Discovered, se adaugă în acesta, apoi va fi introdusă în coada de priorități (Frontier) cu prioritatea fiind costul cu care a fost adăugată în Discovered + valoarea dată de euristică din starea curentă până în cea finală
- căutarea continuă din starea aflată la începutul cozii, până se întâlnește o stare finală

Aflarea tuturor soluțiilor pentru satisfacerea unui scop

Prolog oferă un set special de predicate care pot construi liste din toate soluțiile de satisfacere a unui scop. Acestea sunt extrem de utile deoarece altfel este complicată culegerea informațiilor la revenirea din backtracking (o alternativă este prezentată în secțiunea următoare, mai mult în laboratorul viitor).

findall(+Template, +Goal, -Bag)

Predicatul `findall` pune în Bag câte un element Template pentru fiecare soluție a expresiei Goal. Desigur, predicatul este util atunci când Goal și Template au variabile comune. De exemplu:

```
even(Numbers, Even):-
    forall(X,(member(X, Numbers), X mod 2 == 0), Even).
```

```
?- even([1,2,3,4,5,6,7,8,9], Even).
Even = [2, 4, 6, 8].
```

Resurse

- [Exerciții](#)
- [Cheatsheet](#)

Referințe

- „Prolog Programming for Artificial Intelligence“, Ivan Bratko

20/laboratoare/prolog/cautare.txt · Ultima modificare: 2020/05/05 20:57 de către Mihnea Muraru

Prolog: Probleme

- Responsabil: Vlad Neculae [mailto:neculae.vlad@gmail.com]
- Data publicării: 12.05.2020
- Data ultimei modificări: 12.05.2020

Obiective

Scopul acestui laborator îl reprezintă învățarea unor concepte avansate de programare în Prolog, legate în special de controlul execuției, precum și de obținerea tuturor soluțiilor ce satisfac un scop.

Controlul execuției: operatorul cut (!), negația (\+) și fail

Negația ca eșec (\+)

\+ este operatorul folosit pentru negație în Prolog (meta-predicatul not nu mai este recomandat). Așa cum ați observat nu se pot adăuga în baza de date fapte în forma negată și nici nu se pot scrie reguli pentru acestea. Dacă nu se poate demonstra $\neg p$, atunci ce semnificație are în Prolog $\text{not}(\text{Goal})$ sau $\neg \text{Goal}$?

Prolog utilizează *presupunerea lumii închise*: ceea ce nu poate fi demonstrat, este fals. De aceea, în Prolog $\neg p$ trebuie citit ca „scopul p nu poate fi satisfăcut” sau „ p nu poate fi demonstrat”. Faptul că Prolog utilizează negația ca eșec (eng. *negation as failure*) are implicații asupra execuției programelor.

În logica de ordinul întâi, următoarele două expresii sunt echivalente: $\neg a(X) \ \& \ b(X)$ și $b(X) \ \& \ \neg a(X)$. În Prolog, următoarele 2 clauze ($p1$ și $p2$) vor produce rezultate diferite:

```
student(andrei).
student(marius).
lazy(marius).
```

```
p1(X):- student(X), \+ lazy(X).
p2(X):- \+ lazy(X), student(X).
```

Predicatul fail

fail este un predicat care eșuează întotdeauna. De ce am vrea să scriem o regulă care să eșueze? Un posibil răspuns este: datorită efectelor laterale pe care le pot avea scopurile satisfăcute până la întâlnirea predicatului fail. Un exemplu este următorul:

```
my_reverse(List, Acc, _):-format('List:~w, Acc:~w~n', [List, Acc]), fail.
my_reverse([], Sol, Sol).
my_reverse([Head|Tail], Acc, Sol):-my_reverse(Tail, [Head|Acc], Sol).
```

În codul de mai sus, am scris o regulă pentru funcția my_reverse care are un singur rol: acela de a afișa argumentele List și Acc în orice moment se dorește satisfacerea unui scop cu predicatul my_reverse/3.

```
?- my_reverse([1,2,3,4],[],Rev).
List:[1,2,3,4], Acc:[]
List:[2,3,4], Acc:[1]
List:[3,4], Acc:[2,1]
List:[4], Acc:[3,2,1]
```

```
List:[], Acc:[4,3,2,1]
Rev = [4, 3, 2, 1].
```

Operatorul cut

În Prolog, operatorul cut (!) are rolul de a elimina toate punctele de întoarcere create în predicatul curent.

Dincolo de utilizarea operatorului cut (!) pentru a influența execuția (rezultatele) programelor (revedeți exemplul cu max), acesta poate avea rol și în optimizarea programelor. Eliminarea unor puncte de întoarcere acolo unde acestea oricum ar fi inutile, poate salva memorie. Uneori, erorile „Out of global stack“ pot fi rezolvate prin introducerea de operatori cut ce nu modifică rezultatele programelor.

Așa cum veți observa din problemele din laborator, cut, fail și negația devin și mai utile atunci când sunt folosite împreună.

Aflarea tuturor soluțiilor pentru satisfacerea unui scop

Prolog oferă un set special de predicate care pot construi liste din toate soluțiile de satisfacere a unui scop. Acestea sunt extrem de utile deoarece altfel este complicată culegerea informațiilor la revenirea din backtracking (o alternativă este prezentată în secțiunea următoare).

findall(+Template, +Goal, -Bag)

Predicatul findall pune în Bag câte un element Template pentru fiecare soluție a expresiei Goal. Desigur, predicatul este util atunci când Goal și Template au variabile comune. De exemplu:

```
even(Numbers, Even):-
    forall(X,(member(X, Numbers), X mod 2 =:= 0), Even).

?- even([1,2,3,4,5,6,7,8,9], Even).
Even = [2, 4, 6, 8].
```

bagof(+Template, +Goal, -Bag)

Predicatul bagof seamănă cu findall, diferența fiind că bagof construiește câte o listă Bag pentru fiecare instanțiere diferită a variabilelor libere din Goal.

```
digits([1,2,3,4,5,6,7,8,9]).
numbers([4,7,9,14,15,18,24,28,33,35]).
```

```
multiples(D,L):-
    digits(Digits),
    numbers(Numbers),
    bagof(N,(member(D, Digits), member(N, Numbers), N mod D =:= 0), L).
```

```
?- multiples(D,L).
D = 1,
L = [4, 7, 9, 14, 15, 18, 24, 28, 33|...] ;
D = 2,
L = [4, 14, 18, 24, 28] ;
D = 3,
L = [9, 15, 18, 24, 33] ;
D = 4,
L = [4, 24, 28] ;
D = 5,
L = [15, 35] ;
D = 6,
L = [18, 24] ;
D = 7,
L = [7, 14, 28, 35] ;
D = 8,
L = [24] ;
```

```
D = 9,  
L = [9, 18].
```

Pentru a evita *gruparea* soluțiilor pentru fiecare valoare separată a variabilelor ce apar în scopul lui `bagof/3` se poate folosi construcția `Var^Goal`

setof(+Template, +Goal, -Bag)

Predicatul `setof/3` are aceeași comportare cu `bagof/3`, dar cu diferența că soluțiile găsite sunt sortate și se elimină duplicatele.

Câteva observații asupra purității

În logica de ordinul întâi clauzele $p(A,B) \wedge q(A,B)$ și $q(A,B) \wedge p(A,B)$ sunt echivalente. Ordinea termenilor dintr-o conjuncție (sau disjuncție) nu influențează valoarea de adevăr a clauzei.

În Prolog acest lucru nu este întotdeauna adevărat:

```
?- X = Y, X == Y.  
X = Y.
```

```
?- X == Y, X = Y.  
false.
```

Resurse

- [probleme](#)