Laborator 1

Fundamentele Limbajelor de Programare

Prolog

- Prolog este cel mai cunoscut limbaj de programare logică.
  - bazat pe logica clasică de ordinul I (cu predicate)
  - funcționează pe bază de unificare și căutare
- Multe implementări îl transformă în limbaj de programare "matur"
  - I/O, operații implementate deja în limbaj etc.

Prolog

- Vom folosi implementarea SWI-Prolog
  - gratuit
  - folosit des pentru predare
  - conține multe librării
  - `http://www.swi-prolog.org/`
- Varianta online SWISH a SWI-Prolog
  - `http://swish.swi-prolog.org/`

Prolog

TODO

- Cum arată un program în Prolog?
- Cum punem întrebări în Prolog?

Mai multe detalii

- `Capitolul 1` din *Learn Prolog Now!*.

Sintaxă: atomi

Atomi:

- secvențe de litere, numere și _, care încep cu o literă mică
- șiruri între apostrofuri `'Atom'`
- anumite simboluri speciale

Exemplu

- `elefant`
- `abcXYZ`
- `'Acesta este un atom'`
- `'(@ *+'`
- `  +`

```
?- atom('(@ *+ ').
true.
```

`atom/1` este un predicat predefinit

Sintaxă: constante și variabile

Constante:

- atomi: a, 'I am an atom'
- numere: 2, 2.5, -33

Variabile:

- secvențe de litere, numere și _, care încep cu o literă mare sau cu _
- Variabilă specială: _ este o variabilă anonimă
  - două apariții ale simbolului _ sunt variabile diferite
  - este folosită când nu vrem detalii despre variabila respectivă

Exemplu

- `X`
- `Animal`
- `_x`
- `X_1_2`

Sintaxă: termeni compuși

Termeni compuși:

- au forma `p(t1,...,tn)` unde
  - `p` este un atom,
  - `t1,...,tn` sunt termeni.

Exemplu

- `is_bigger(horse, X)`
- `is_bigger(horse,dog)`
- `f(g(X,_),7)`

- Un termen compus are
  - un nume (functor): `is_bigger` în `is_bigger(horse, X)`
  - o aritate (numărul de argumente): 2 în `is_bigger(horse, X)`

kb1: Un prim exemplu

Un program Prolog definește o bază de cunoștințe.

Exemplu

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).

is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

Predicate

O bază de cunoștințe este o mulțime de predicate prin care definim lumea(universul) programului respectiv.

Exemplu

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).

is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

Acest program conține două predicate:

`bigger/2`, `is_bigger/2`.

Definirea predicatelor

- Predicate cu același nume, dar cu arități diferite, sunt predicate diferite.
- Scriem `foo/n` pentru a indica că un predicat `foo` are aritatea `n`.
- Predicatele pot avea aritatea 0 (nu au argumente); sunt predefinite în limbaj (`true, false`).
- Predicate predefinite: `X=Y` (este adevărat dacă X poate fi unificat cu Y); `X\=Y` (este adevărat dacă X nu poate fi unificat cu Y);

Un exemplu cu fapte și reguli

- O **regulă** este o afirmație de forma `Head :- Body.` unde
  - `Head` este un predicat (termen complex)
  - `Body` este o secvență de predicate, separate prin virgulă.

Exemplu

```
is_smaller(X, Y) :- is_bigger(Y, X).
aunt(Aunt, Child) :- sister(Aunt, Parent),
                     parent(Parent, Child).
```

- Un **fapt** (*fact*) este o regulă fără `Body`.

Exemplu

```
bigger(whale, _).
life_is_beautiful.
```

Reguli

O **regulă** este o afirmație de forma `Head :- Body.`

Exemplu

```
is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

Interpretarea:

- Mai multe reguli care au același `Head` trebuie gândite că au sau între ele.
- `:-` se interpretează drept implicație ($\leftarrow$)
- `,` se interpretează drept conjuncție ($\wedge$)

Astfel, din punct de vedere al logicii, putem spune că `is_bigger(X, Y)` este adevarat dacă `bigger(X, Y)` $\vee$ `bigger(X, Z)` $\wedge$ `is_bigger(Z, Y)` este adevarat.

Definirea predicatelor

Mai multe reguli care au același `Head` trebuie gândite că au sau între ele.

Exemplu

```
is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

Ele se pot uni folosind `;`.

Exemplu

```
is_bigger(X, Y) :-
       bigger(X, Y);
       bigger(X, Z), is_bigger(Z, Y).
```

Sintaxă: program

Un program în Prolog este o colecție de fapte și reguli.

Faptele și regulile trebuie grupate după atomii folosiți în Head.

Exemplu

Corect:

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).
is_bigger(X, Y) :-
      bigger(X, Y).
is_bigger(X, Y) :-
      bigger(X, Z),
      is_bigger(Z, Y).
```

Incorect:

```
bigger(elephant, horse).
bigger(horse, donkey).
is_bigger(X, Y) :-
      bigger(X, Y).
bigger(donkey, dog).
bigger(donkey, monkey).
is_bigger(X, Y) :-
      bigger(X, Z),
      is_bigger(Z, Y).
```

Întrebări, răspunsuri, ținte

- O `întrebare` (*query*) este o secvență de forma

  ```
  ?- p1(t1,...,tn),...,pn(t1',...,tn').
  ```

- Fiind dată o întrebare (deci o țintă), Prolog caută răspunsuri.
  - `true`/ `false` dacă întrebarea nu conține variabile;
  - dacă întrebarea conține variabile, atunci sunt căutate valori care fac toate predicatele din întrebare să fie satisfăcute; dacă nu se găsesc astfel de valori, răspunsul este `false`.

- Predicatele care trebuie satisfăcute pentru a răspunde la o întrebare se numesc ținte (goals).

Exemple de întrebări și răspunsuri

Exemplu

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).

is_bigger(X, Y) :-
      bigger(X, Y).
is_bigger(X, Y) :-
      bigger(X, Z),
      is_bigger(Z, Y).
```

```
?- is_bigger(elephant, horse).
true

?- bigger(donkey, dog).
true

?- is_bigger(elephant, dog).
true

?- is_bigger(monkey, dog).
false

?- is_bigger(X, dog).
X = donkey ;
X = elephant ;
X = horse
```

În varianta online, puteți adăuga întrebări la finalul programului ca în exemplul de mai jos. Întrebările vor apărea în lista din *Examples* (partea dreaptă).

Exemplu

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).
is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).

/** <examples>

?- is_bigger(elephant, horse).
?- bigger(donkey, dog).
?- is_bigger(elephant, dog).
?- is_bigger(monkey, dog).
?- is_bigger(X, dog).
*/
```

Un exemplu cu date și reguli ce conțin variabile

Exemplu

```
?- is_bigger(X, Y), is_bigger(Y,Z).
X = elephant,
Y = horse,
Z = donkey
X = elephant,
Y = horse,
Z = dog
X = elephant,
Y = horse,
Z = monkey
X = horse,
Y = donkey,
Z = dog
.....
```

Diverse

- Un program în Prolog are extensia `.pl`
- Comentarii:

```
% comentează restul liniei

/* comentariu
pe mai multe linii */
```

- Nu uitați să puneți . la sfârșitul unui fapt sau al unei reguli.
- un program(o bază de cunoștințe) se încarcă folosind:

```
?- [nume].
?- ['...cale.../nume.pl'].
?- consult('...cale../nume.pl').
```

Practică

Exercițiul 1
Încercați să răspundeți la următoarele întrebări, verificând in interpretor.

1. Care dintre următoarele expresii sunt atomi?
   f, loves(john, mary), Mary, _c1, 'Hello'
2. Care dintre următoarele expresii sunt variabile?
   a, A, Paul, 'Hello', a_123, _,_abc

Exercițiul 2
Fișierul ex2.pl conține o bază de cunoștințe reprezentând un arbore genealogic.

- Definiți următoarele predicate, folosind male/1, female/1 și parent/2:
  - `father_of(Father, Child)`
  - `mother_of(Mother, Child)`
  - `grandfather_of(Grandfather, Child)`
  - `grandmother_of(Grandmother, Child)`
  - `sister_of(Sister,Person)`
  - `brother_of(Brother,Person)`
  - `aunt_of(Aunt,Person)`
  - `uncle_of(Uncle,Person)`
- Verificați predicate definite punând diverse întrebări.

Negația

În Prolog există predicatul predefinit `not` cu următoarea semnificație:

`not(goal)` este `true` dacă `goal` nu poate fi demonstrat în baza de date curentă.

Atenție: `not` nu este o negație logică, ci exprimă imposibilitatea de a face demonstrația (sau instanțierea) conform cunoștințelor din bază ("closed world assumption"). Pentru a marca această distincție, în variantele noi ale limbajului, în loc de `not` se poate folosi operatorul `\+`.

Exemplu

```
not_parent(X,Y) :- not(parent_of(X,Y)). % sau
not_parent(X,Y) :- \+ parent_of(X,Y).
```

Exercițiul 3: negația
Folosind baza anterioară (arbore genealogic) testați predicatul `not_parent`:

```
?- not_parent(bob,juliet).
?- not_parent(X,juliet).
?- not_parent(X,Y).
```

Ce observați? Încercați să analizați răspunsurile primite.

Exercițiul 3: negația
Folosind baza anterioară (arbore genealogic) testați predicatul `not_parent`:

```
?- not_parent(bob,juliet).
?- not_parent(X,juliet).
?- not_parent(X,Y).
```

Ce observați? Încercați să analizați răspunsurile primite.

- Corectați `not_parent` astfel încât să dea răspunsul corect la toate întrebările de mai sus.

Aritmetica în Prolog

Aritmetica în Prolog

Exemplu

```
?- 3+5 = +(3,5).
true
?- 3+5 = +(5,3).
false
?- 3+5 = 8.
false
```

Explicații:

- 3+5 este un termen.
- Prolog trebuie anunțat explicit pentru a îl evalua ca o expresie aritmetică, folosind predicate predefinite în Prolog, cum sunt `is/2`, `=:=/2`, `>/2` etc.

Aritmetica în Prolog

Exercițiu. *Analizați următoarele exemple:*

```
?- 3+5 is 8.
false

?= X is 3+5.
X = 8

?- 8 is 3+X.
is/2: Arguments are not sufficiently instantiated

?- X=4, 8 is 3+X.
false
```

Aritmetica în Prolog

Exercițiu. *Analizați următoarele exemple:*

```
?- X is 30-4.
X = 26

?- X is 3*5.
X = 15

?- X is 9/4.
X = 2.25
```

Aritmetica în Prolog

Operatorul `is`:

- Primește două argumente
- Al doilea argument trebuie să fie o expresie aritmetică validă, cu toate variabilele inițializate
- Primul argument este fie un număr, fie o variabilă
- Dacă primul argument este un număr, atunci rezultatul este `true` dacă este egal cu evaluarea expresiei aritmetice din al doilea argument.
- Dacă primul argument este o variabilă, răspunsul este pozitiv dacă variabila poate fi unificată cu evaluarea expresiei aritmetice din al doilea argument.

Totuși, nu este recomandat să folosiți `is` pentru a compara două expresii aritmetice, ci operatorul `=:=`.

Aritmetica în Prolog

Exercițiu. *Analizați următoarele exemple:*

```
?- 8 > 3.
true
```

```
?- 8+2 > 9-2.
true
```

```
?- 8 < 3.
false
```

```
?- 8 >= 3.
true
```

```
?- 8 =:= 3.
false
```

```
?- 8 =\= 3.
true
```

Operatori aritmetici

Operatorii aritmetici predefiniți în Prolog sunt de două tipuri:

- funcții
- relații

Funcții

- Adunarea și înmulțirea sunt exemple de funcții aritmetice.
- Aceste funcții sunt scrise în mod uzual și în Prolog.

Exemplu

```
2 + (-3.2 * X - max(17,X)) / 2 ** 5
```

- `2**5` înseamnă 2^5
- Exemple de alte funcții disponibile:
  `min/2`, `abs/1` (modul), `sqrt/1` (radical), `sin/1` (sinus)
- Operatorul `//` este folosit pentru împărțire întreagă.
- Operatorul `mod` este folosit pentru restul împărțirii întregi.

Relații

- Relatiile aritmetice sunt folosite pentru a compara evaluarea expresiilor aritmetice (e.g, `X > Y`)
- Exemple de relații disponibile:
  <, >, =<, >=, =\= (diferit), =:= (aritmetic egal)
- Atenție la diferența dintre =:= și =:
  - =:= compară două expresii aritmetice
  - = caută un unificator

Exemplu

```
?- 2 ** 3 =:= 3 + 5.
true
?- 2 ** 3 = 3 + 5.
false
```

Exercițiul 4: distanța dintre două puncte
Definiți un predicat `distance/3` pentru a calcula distanța dintre două puncte într-un plan 2-dimensional. Punctele sunt date ca perechi de coordonate.

Exemple:

```
?- distance((0,0), (3,4), X).
X = 5.0

?- distance((-2.5,1), (3.5,-4), X).
X = 7.810249675906654
```

Bibliografie

- `http://www.learnprolognow.org`
- `http://cs.union.edu/~striegnk/courses/esslli04prolog`
- U. Endriss, Lecture Notes. An Introduction to Prolog Programming, ILLC, Amsterdam, 2018.

Pe data viitoare!

Laborator 2

Fundamentele Limbajelor de Programare

Recursivitate

Bază de conștințe

În laboratorul trecut am folosit următoarea bază de cunoștințe:

```
parent(dean,bob).
parent(jane,bob).
parent(bob, lisa).
parent(bob, paul).
parent(bob, mary).
parent(juliet, lisa).
parent(juliet, paul).
parent(juliet, mary).
parent(peter, harry).
parent(lisa, harry).
parent(mary, dony).
parent(mary, sandra).
```

Recursivitate - strămoși

Putem defini un predicat `ancestor_of(X,Y)` care este adevărat dacă X este un strămoș al lui Y.

Definiția recursivă a predicatului `ancestor_of(X,Y)` :

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

Practică

Exercițiul 1: numerele Fibonacci
Scrieți un predicat `fib/2` pentru a calcula al n-ulea număr Fibonacci.

Secvența de numere Fibonacci este definită prin:

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-1} \text{ for } n \geq 2 \end{aligned}$$

Exemple:

```
?- fib(1,X).
X=1.
true
```

```
?- fib(5,X).
X=8.
true
```

```
?- fib(2,X).
X=2.
true
```

Exercițiul 1 (cont.)
Programul scris anterior vă găsește răspunsul la întrebarea de mai jos?

```
?- fib(50,X).
```

Dacă da, felicitări! Dacă nu, încercați să găsiți o soluție mai eficientă!

Hint: Încercați să construiți toate numerele Fibonacci până ajungeți la numărul căutat.

Afișări în Prolog

- Pentru a afișare se foloseste predicatul `write/1`.
- Predicatul `nl/0` conduce la afișarea unei linii goale.

Exemplu

```
?- write('Hello World!'), nl.
Hello World!
true

?- X = hello, write(X), nl.
hello
X = hello
```

Exercițiul 2: afișarea unui pătrat de caractere
Scrieți un program în Prolog pentru a afișa un pătrat de $n \times n$ caractere pe ecran.

Denumiți predicatul `square/2`. Primul argument este un număr natural diferit de 0, iar al doilea un caracter (i.e, orice termen în Prolog) care trebuie afișat.

Exemplu:

```
?- square(5, '* ').
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Liste

Liste

- Listele în Prolog sunt un tip special de date (termeni speciali).
- Listele se scriu între paranteze drepte, cu elementele despărțite prin virgulă.
- `[]` este lista vidă.

Exemplu

- `[elephant, horse, donkey, dog]`
- `[elephant, [], X, parent(X, tom), [a, b, c], f(22)]`

Head & Tail

- Primul element al unei liste se numește *head*, iar restul listei *tail*.
- Evident, o listă vidă nu are un prim element.
- În Prolog există o notație utilă pentru liste cu separatorul `|`, evidențiind primul element și restul listei.

Exemplu

```
?- [1, 2, 3, 4, 5] = [Head | Tail].
Head = 1
Tail = [2, 3, 4, 5]
```

Cu această notație putem să returnăm ușor, de exemplu, al doilea element dintr-o listă.

```
?- [quod, licet, jovi, non, licet, bovi] = [_, X | _].
X = licet
```

Lucrul cu liste

Exemplu (`elements_of/2`)

- un predicat care verifică dacă o listă conține un anumit termen
- `element_of(X,Y)` trebuie să fie adevărat dacă `X` este un element al lui `Y`.

```
/* Dacă primult element al listei este termenul
pe care îl căutăm, atunci am terminat.

Altfel, verificăm dacă termenul se află în restul listei.
*/


element_of(X,[X|_]). element_of(X,[_|Tail]) :-
element_of(X,Tail).

?- element_of(a,[a,b,c]).
?- element_of(X,[a,b,c]).
```

Exemplu (`concat_lists/3`)

- un predicat care este poate fi folosit pentru a concatena două liste
- al treilea argument este concatenarea listelor date ca prime două argumente

```
concat_lists([], List, List).
concat_lists([Elem | List1], List2, [Elem | List3]) :-
      concat_lists(List1, List2, List3).
```

```
?- concat_lists([1, 2, 3], [d, e, f, g], X).
?- concat_lists(X, Y, [a, b, c, d]).
```

Lucrul cu liste

În Prolog există niște predicate predefinite pentru lucrul cu liste. De exemplu:

- `length/2`: al doilea argument întoarce lungimea listei date ca prin argument
- `member/2`: este adevărat dacă primul argument se află în lista dată ca al doilea argument
- `append/3`: identic cu predicatul anterior `concat_lists/3`
- `last/2`: este adevărat dacă al doilea argument este identic cu ultimul element al listei date ca prim argument
- `reverse/2`: lista din al doilea argument este lista data ca prim elemenent în oglindă.

Exercițiul 3

A) Definiți un predicat `all_a/1` care primește ca argument o listă și care verifică dacă argumentul său este format doar din a-uri.

```
?- all_a([a,a,a,a]).
?- all_a([a,a,A,a]).
```

B) Scrieti un predicat `trans_a_b/2` care "traduce" o listă de a-uri într-o listă de b-uri. `trans_a_b(X,Y)` trebuie să fie adevărat dacă "intrarea" `X` este o listă de a-uri și "ieșirea" `Y` este o listă de b-uri, iar cele două liste au lungimi egale.

```
?- trans_a_b([a,a,a],L).
?- trans_a_b([a,a,a],[b]).
?- trans_a_b(L,[b,b]).
```

Exercițiul 4: Operații cu vectori
A) Scrieți un predicat `scalarMult/3` al cărui prim argument este un întreg, al doilea argument este o listă de întregi, iar al treilea argument este rezultatul înmulțirii cu scalari al celui de-al doilea argument cu primul.

De exemplu, la intrebarea

```
?-scalarMult(3,[2,7,4],Result).
```

ar trebui să obțineți `Result = [6,21,12]`.

Exercițiul 4 (cont.)
B) Scrieți un predicat `dot/3` al cărui prim argument este o listă de întregi, al doilea argument este o listă de întregi de lungimea primeia, iar al treilea argument este produsul scalar dintre primele două argumente.

De exemplu, la întrebarea

```
?-dot([2,5,6],[3,4,1],Result).
```

ar trebui să obțineți `Result = 32`.

Exercițiul 5
Scrieți un predicat `max/2` care caută elementul maxim într-o listă de numere naturale.

De exemplu, la întrebarea

```
?-max([4,2,6,8,1],Result).
```

ar trebui să obțineți `Result = 8.`

Exercițiul 6
Definiți un predicat `palindome/1` care este adevărat dacă lista primită ca argument este palindrom (lista citită de la stânga la dreapta este identică cu lista citită de la dreapta la stânga).

De exemplu, la întrebarea

```
?-palindrome([r,e,d,i,v,i,d,e,r]).
```

ar trebui să obțineți `true`.

Nu folosiți predicatul predefinit `reverse`, ci propria implementare a acestui predicat.

Practică

Exercițiul 7
Definiți un predicat `remove_duplicates/2` care șterge toate duplicatele din lista dată ca prim argument și întoarce rezultatul în al doilea argument.

De exemplu, la întrebarea

```
?- remove_duplicates([a, b, a, c, d, d], List).
```

ar trebui să obțineți `List = [b, a, c, d]`.

Practică

Exercițiul 8
Definiți un predicat `replace/4` care înlocuiește toate aparițiile elementul dat în al doilea argument cu un alt element dat în al treilea argument într-o listă data ca prim argument.

De exemplu, la întrebarea

```
?- replace([1, 2, 3, 4, 3, 5, 6, 3], 3, x, List).
```

ar trebui să obțineți `List = [1, 2, x, 4, x, 5, 6, x]`

Pe data viitoare!

Laborator 3

Fundamentele Limbajelor de Programare

Cum răspunde Prolog întrebărilor

Unificare

- În acest laborator prezentăm doar intuitiv ce înseamnă un unificator.
- Mai multe detalii și algoritmul de unificare care găsește un unificator pentru o mulțime de termeni, sunt prezentate în cadrul cursului.

Unificare

- Prolog are un operator (infixat) pentru egalitate:
  `t = u` (sau echivalent `=(t,u)`)
- Ecuația $t = u$ este o țintă de bază, cu o semnificație specială.
- Ce se întâmplă dacă punem următoarele întrebări:

```
?- X = c.
?- f(X,g(Y,Z)) = f(c,g(X,Y)).
?- f(X,g(Y,f(X))) = f(c,g(X,Y)).
```

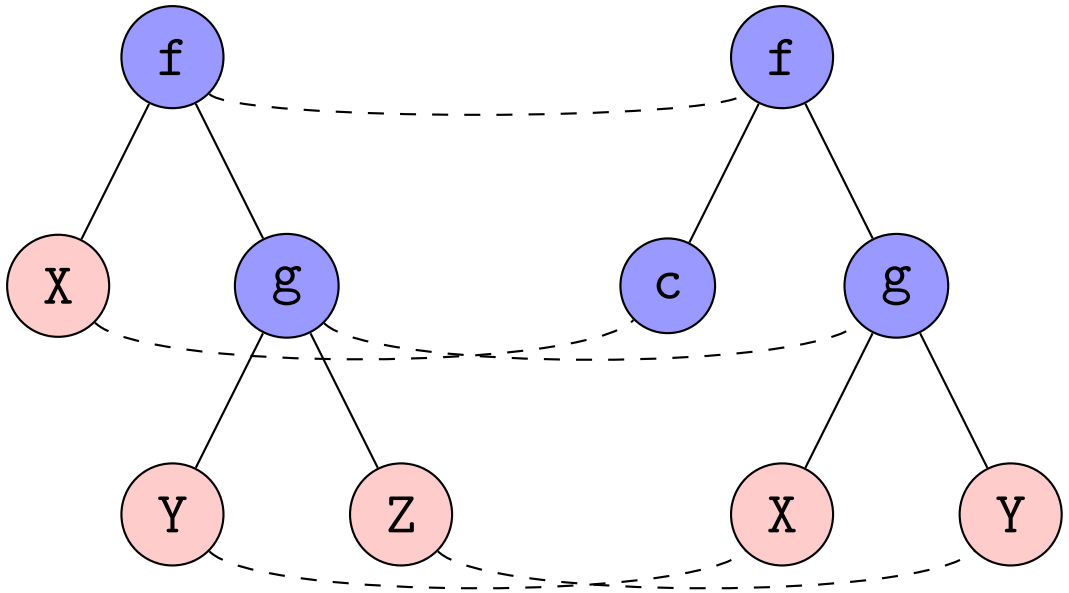
- Cum găsește aceste răspunsuri?

Unificare

- O substituție este o funcție (parțială) de la variabile la termeni.
  - $X_1 = t_1, \ldots, X_n = t_n$
- Pentru doi termeni t și u, cu variabilele $X_1, \ldots, X_n$, un unificator este o substituție care aplicată termenilor t și u îi face identici.

Exemplu

f(X,g(Y,Z)) = f(c,g(X,Y))



X=c

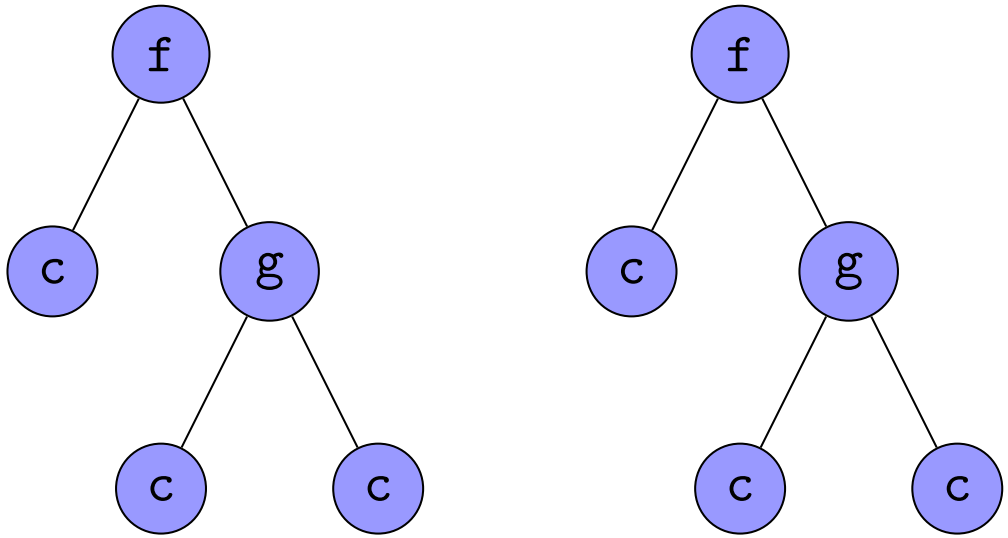
Y=X

Z=Y

Exemplu: aplicând substituția

f(X,g(Y,Z)) = f(c,g(X,Y))

X=c
Y=c
Z=c



Unificare

- Ce se întâmplă dacă încercăm să unificăm X cu ceva care conține X? Exemplu: `?- X = f(X).`
- Conform teoriei, acești termeni nu se pot unifica.
- Totuși, multe implementări ale Prolog-ului sar peste această verificare din motive de eficiență.
  - putem folosi `unify_with_occurs_check/2`

Ce se întamplă în Prolog când punem o întrebare?

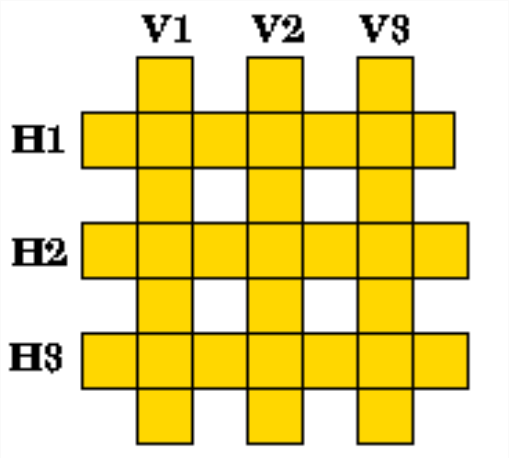
- Pentru a găsi un răspuns, Prolog încearcă regulile în ordinea în care sunt scrise.
- Folosește unificarea pentru a potrivi țintele și clauzele (reguli și fapte).
- Prolog poate da 2 tipuri de răspunsuri:
  - `false` – în cazul în care întrebarea nu este o consecință a programului.
  - `true` sau valori pentru variabilele din întrebare în cazul în care întrebarea este o consecință a programului.
- Poate găsi zero, una sau mai multe soluții.
- Execuția se poate întoarce (*backtracking*).

Exerciții

Exercițiul 1: cunvinte încrucișate
Șase cuvinte din engleză

abalone, abandon, anagram, connect, elegant, enhance

trebuie aranjate într-un puzzle de cuvinte încrucișate ca în figură.



Fișierul `words.pl` este o bază de cunoștințe ce conține aceste cuvinte.

Exercițiul 1 (cont.)
Definiți un predicat `crosswd/6` care calculează toate variantele în care puteți completa grila. Primele trei argumente trebuie să fie cuvintele pe verticală, de la stânga la dreapta, (V1,V2,V3), iar următoarele trei argumente trebuie să fie cuvintele pe orizontală, de sus în jos (H1,H2,H3).

Hint: Specificați că V1, V2, V3, H1, H2, H3 sunt cuvinte care au anumite litere comune. Unde este cazul, folosiți variabile anonime.

Exercițiul 2: baza de date

În acest exercițiu vom demonstra cum se poate implementa o bază de date simplă în Prolog.

Folosiți în programul vostru următoarea bază de cunoștințe:

```
born(jan, date(20,3,1977)).
born(jeroen, date(2,2,1992)).
born(joris, date(17,3,1995)).
born(jelle, date(1,1,2004)).
born(joan, date(24,12,0)).
born(joop, date(30,4,1989)).
born(jannecke, date(17,3,1993)).
born(jaap, date(16,11,1995)).
```

Reprezentăm datele calendaristice ca termeni de forma `date(Day,Month,Year)`.

U. Endriss, Lecture Notes. An Introduction to Prolog Programming,

Exercițiul 2 (cont.)
a) Scrieți un predicat `year/2` care găsește toate persoanele născute într-un anumit an.

Exemplu:

```
?- year(1995, Person).
Person = joris
Person = jaap
```

Hint: Folosiți variabile anonime.

Exercițiul 2 (cont.)
b) Scrieți un predicat `before/2` care primește două date calendaristice și care este adevărat dacă prima expresie reprezintă o dată calendaristică înaintea datei reprezentate de a doua expresie (puteți presupune că datele sunt corecte, e.g., nu puteți primi 31 Aprilie).

Exemplu:

```
?- before(date(31,1,1990), date(7,7,1990)).
true
```

Exercițiul 2 (cont.)
c) Scrieți un predicat `older/2` care este adevărat dacă persoana dată ca prim argument este mai în vârstă (strict) decât persoana dată ca al doilea argument.

Exemplu:

```
?- older(jannecke,X).
X = joris
X = jelle
X = jaap
```

Exercițiul 3: drumurile într-un labirint
Baza de cunoștințe din `maze.pl` descrie un labirint.

Faptele indică ce puncte sunt conectate (din ce punct se poate ajunge într-un alt punct într-un pas).

Drumurile sunt cu sens unic (se poate merge pe ele doar într-o direcție).

De exemplu, se poate ajunge într-un pas de la 1 la 2, dar nu și invers.

Adăugați un predicat `path/2` care indică dacă dintr-un punct puteți să ajungeți într-un alt punct (în mai mulți pași), legând conexiunile din baza de cunoștințe.

Exercițiul 3 (cont.)
Puneți următoarele întrebări:

- Puteți ajunge din punctul 5 în punctul 10?
- În ce puncte puteți să ajungeți plecând din 1?
- Din ce puncte puteți să ajungeți în punctul 13?

Exercițiul 3 (cont.)
Testați programul pentru următoarea bază de cunoștinte:

```
connected(1,2).
connected(2,1).
connected(1,3).
connected(3,4).
```

```
?- path(1,4).
```

Atenție!
Dacă graful conține cicluri, este posibil ca programul să nu se termine.

Scrieți un predicat care determină existența drumurilor, evitând ciclările.

Indicație: folosiți un predicat auxiliar care reține într-o listă punctele vizitate până la momentul curent.

Exercițiul 4: numere naturale

În acest exercițiu vom reprezenta numerele sub următoarea formă:

```
0   :   []
1   :   [x]
2   :   [x,x]
3   :   [x,x,x]
4   :   [x,x,x,x]
...
```

Exercițiul 4 (cont.)
a) Definiți un predicat `successor/2` care întoarce în al doilea argument succesorul numărului dat ca prim argument.

Exemplu:

```
?- successor([x,x,x],Result).
Result = [x,x,x,x]


?- successor([],Result).
Result = [x]
```

Exercițiul 4 (cont.)
b) Definiți un predicat `plus/3` care adună două numere.

Exemplu:

```
?- plus([x, x], [x, x, x, x], Result).
Result = [x, x, x, x, x, x]
```

Exercițiul 4 (cont.)
c) Definiți un predicat `times/3` care înmulțește două numere.

Exemplu:

```
?- times([x, x], [x, x, x, x], Result).
Result = [x, x, x, x, x, x, x, x]
```

Exercițiul 5
Definiți un predicat `element_at/3` care, primind o listă și un număr natural *n*, întoarce al n-ulea element din listă.

Exemplu:

```
?- element_at([tiger, dog, teddy_bear, horse, cow], 3, X).
X = teddy_bear


?- element_at([a, b, c, d], 27, X).
false
```

Exercițiul 6

În fișierul `animale.pl` găsiți o bază de cunoștinte care definește animalele cunoscute. Un "mutant" se obține prin combinarea a două animale cunoscute, cu proprietatea ca numele unuia are un sufix care este prefix pentru numele celuilalt. Numele animalului mutant se obține prin concatenarea celor două nume, în care partea comună apare o singură dată. De exemplu, din `alligator` și `tortue` se obține `aligatortue`.

Scrieți un predicat `mutant/1` care generează pe rând toate animalele mutante.

H. Coelho, J.C.Cotta, Prolog by example, 1998

- Folosiți predicatul `name/2` care face conversia între un atom și lista caracterelor sale, reprezentate prin codurile ASCII. Verificați:

```
?- name(alligator,L).
?- name(A, [97, 108, 108, 105, 103, 97, 116, 111, 114]).
```

Pe data viitoare!

Laborator 4

Fundamentele Limbajelor de Programare

Practică

Exercițiul 1: Zebra puzzle
Zebra puzzle este unul dintre cele mai cunoscute puzzle-uri logice.

Citiți mai jos mai multe detalii despre acest puzzle:

`https://en.wikipedia.org/wiki/Zebra_Puzzle`

Vom rezolva următoarea versiune publicată în 1962 în care fiecare personaj

- locuiește într-o casă care are o anumită culoare,
- are o naționalitate,
- are un anumit animal de companie,
- are o băutură preferată,
- fumează un anumit tip de țigări.

Exercițiul 1 (cont.)
Vrem să aflăm cine are o zebră având următoarele informații:

1. Sunt cinci case.
2. Englezul locuiește în casa roșie.
3. Spaniolul are un câine.
4. În casa verde se bea cafea.
5. Ucraineanul bea ceai.
6. Casa verde este imediat în dreapta casei bej.
7. Fumătorul de "Old Gold" are melci.
8. În casa galbenă se fumează "Kools".

Exercițiul 1 (cont.)

9. În casa din mjloc se bea lapte.
10. Norvegianul locuiește în prima casă.
11. Fumătorul de "Chesterfields" locuiește lângă cel care are o vulpe.
12. "Kools" sunt fumate în casa de lângă cea în care se ține calul.
13. Fumătorul de "Lucky Strike" bea suc de portocale.
14. Japonezul fumează "Parliaments".
15. Norvegianul locuiește lângă casa albastră.

Exercițiul 1 (cont.)
Vom rezolva acest puzzle în Prolog.

1) Definiți un predicat `la_dreapta(X,Y)` care este adevărat când numerele X și Y sunt consecutive, X fiind cel mai mare dintre ele.

2) Definiți un predicat `la_stanga(X,Y)` care este adevărat când numerele X și Y sunt consecutive, Y fiind cel mai mare dintre ele.

3) Definiți un predicat `langa(X,Y)` care este adevărat când numerele X și Y sunt consecutive.

Exercițiul 1 (cont.)

În continuare definim un predicat `solutie(Strada,PosesorZebra)` în care includem toate informațiile pe care le deținem:

```
solutie(Strada,PosesorZebra) :-
  Strada = [
      casa(1,_,_,_,_,_,_),
      casa(2,_,_,_,_,_,_),
      casa(3,_,_,_,_,_,_),
      casa(4,_,_,_,_,_,_),
      casa(5,_,_,_,_,_,_)],
    member(casa(_,englez,rosie,_,_,_), Strada),
    member(casa(_,spaniol,_,caine,_,_), Strada),
    member(casa(_,_,verde,_,cafea,_), Strada),
    ...,
    member(casa(_,PosesorZebra,_,zebra,_,_), Strada),
```

Completați toate informațiile pentru a afla soluția ținând cont de codarea:

```
casa(Numar,Nationalitate,Culoare,AnimalCompanie,Bautura,Tigari)
```

Exercițiul 2: Cel mai lung cuvânt
Acest exemplu este din

Ulle Endriss, *Lecture Notes – An Introduction to Prolog Programming.*

Countdown este un joc de televiziune popular în Marea Britanie în care jucătorii trebuie să găsească un cuvânt cât mai lung cu literele dintr-o mulțime dată de nouă litere.

Să încercăm să rezolvăm acest joc cu Prolog!

Exercițiul 2: Cel mai lung cuvânt
Concret, vom încerca să găsim o soluție optimă pentru următorul joc:

Primind o listă cu litere din alfabet (nu neapărat unice),
trebuie să construim cel mai lung cuvânt format din literele date
(pot rămâne litere nefolosite).

Vom rezolva jocul pentru cuvinte din limba engleză.

Scorul obținut este lungimea cuvântului găsit.

Exercițiul 2 (cont.)
Scopul final este de a construi un predicat în Prolog `topsolution/2`:

dându-se o listă de litere în primul său argument, trebuie să returneze în al doilea argument o soluție cât mai bună, adică un cuvânt din limba engleză de lungime maximă care poate fi format cu literele din primul argument.

```
?- topsolution([r,d,i,o,m,t,a,p,v],Word).
Word = dioptra
```

Exercițiul 2 (cont.)
Începeți prin a descărca fișierul `words.pl` în același director cu fișierul programului vostru.

Acest fișier conține o listă cu peste 350.000 de cuvinte din limba engleză, de la `a` la `zyzzyva`, sub formă de fapte.

Includeți linia `:- include('words.pl').` în programul vostru pentru a putea folosi aceste fapte.

Exercițiul 2 (cont.)
Predicatul predefinit în Prolog `atom_chars(Atom,CharList)` descompune un atom într-o listă de caractere.

Folosiți acest predicat pentru a defini un predicat `word_letters/2` care transformă un cuvânt (i.e, un atom în Prolog) într-o listă de litere.
De exemplu:

```
?- word_letters(hello,X).
X = [h,e,l,l,o]
```

Ca o paranteza, observați că puteți folosi acest predicat pentru a găsi cuvinte în engleză de 45 de litere:

```
?- word(Word), word_letters(Word,Letters),
   length(Letters,45).
```

Exercițiul 2 (cont.)
Mai departe, scrieți un predicat `cover/2` care, primind două liste, verifică dacă a doua listă "acoperă" prima listă (i.e., verifică dacă fiecare element care apare de k ori în prima listă apare de cel puțin k ori în a doua listă).

De exemplu

```
?- cover([a,e,i,o], [m,o,n,k,e,y,b,r,a,i,n]).
true

?- cover([e,e,l], [h,e,l,l,o]).
false
```

Exercițiul 2 (cont.)
Scrieți un predicat `solution/3` care primind o listă de litere ca prim argument și un scor dorit ca al treilea argument, returnează prin al doilea argument un cuvânt cu lungimea egală cu scorul dorit, "acoperit" de lista respectivă de litere.

De exemplu

```
?- solution([g,i,g,c,n,o,a,s,t], Word, 3).
Word = act
```

Exercițiul 2 (cont.)
Acum implementați predicatul `topsolution/3`.

Testați, de exemplu, predicatul definit pe mulțimea de litere:

```
[y,c,a,l,b,e,o,s,x]
```

Aceasta este una listele de litere folosite în ediția de *Countdown* din 18 Decembrie 2002 din Marea Britanie în care Julian Fell a obținut cel mai mare scor din istoria concursului. Pentru lista de mai sus, el a găsit cuvântul *cables*, câștigând astfel 6 puncte.

Poate programul vostru să bată acest scor?

Pe data viitoare!

Laborator 5

Fundamentele Limbajelor de Programare

Liste în Prolog

Exercițiul 1
Scrieți un predicat `num_aparitii/3` care determină numărul de apariții al unui element într-o listă.

De exemplu, la întrebarea

```
?-num_aparitii([2,5,2,6,3,4,2,1],2,Result).
```

ar trebui să obțineți `Result = 3`.

Liste în Prolog

Exercițiul 2
Scrieți un predicat `lista_cifre/2` care determină lista de cifre pentru un număr dat.

```
?-lista_cifre(23423, [2,3,4,2,3]).
```

true

Exercițiul 3
Scrieți un predicat `listpermcirc/2` care determină lista de permutări circulare pentru o listă dată ca parametru.

```
?- listpermcirc([1,2,3],L).
```

L = [[2, 3, 1], [3, 1, 2], [1, 2, 3]] .

Mulțimi în Prolog

Exercițiul 4

a. Scrieți un predicat `elimina/3` care șterge toate aparițiile unui element dintr-o listă dată ca parametru

```
?- elimina([1,2,4,2,3], 2,L).
```

L = [1, 4, 3] .

b. Scrieți un predicat `multime/2` care transformă o listă în mulțime eliminând toate duplicatele.

```
?- multime([1,2,4,2,3,2,1,2,5,3], L).
```

L = [1, 2, 4, 3, 5] .

c. Scrieți un predicat `emult/1` care verifică dacă o listă este mulțime.

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

false

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

true

Mulțimi în Prolog

Exercițiul 5
Scriți predicate care determină următoarele operații pe mulțimi:

1. `inters/3` - intersecția;
2. `diff/3` - diferența;
3. `prod_cartezian/3` - produsul cartezian.

```
?- inters([1,2,3,4,5,6,7,8,9], [2,4,6,10,3,11,14],L).
```

L = [2, 3, 4, 6] .

```
?- diff([1,2,3,4,5,6,7,8,9], [2,4,6,10,3,11,14],L).
```

L = [1, 5, 7, 8, 9] .

```
?- prod_cartezian([1,2,3],[4,5,6],L).
```

L = [(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)] .

Arbori Binari în Prolog

Exercițiul 6
Vom reprezenta astfel arborii binari:

- `nil` va fi arborele vid;
- `arb(Radacina,SubarboreStang,SubarboreDrept)` va fi un arbore nevid.

a) Scriți predicate care determină parcurgerile arborelui: `srd/2` - inordine; `rsd/2` - preordine și `sdr/2` - postordine.

?- srd(arb(1,arb(2,nil,arb(3,nil,nil)),arb(4,arb(5,nil,nil),arb(6,nil,nil))),L).

L = [2, 3, 1, 5, 4, 6].

?- rsd(arb(1,arb(2,nil,arb(3,nil,nil)),arb(4,arb(5,nil,nil),arb(6,nil,nil))),L).

L = [1, 2, 3, 4, 5, 6].

?- sdr(arb(1,arb(2,nil,arb(3,nil,nil)),arb(4,arb(5,nil,nil),arb(6,nil,nil))),L).

L = [3, 2, 5, 6, 4, 1].

Exercițiul 6
Vom reprezenta astfel arborii binari:

- `nil` va fi arborele vid;
- `arb(Radacina,SubarboreStang,SubarboreDrept)` va fi un arbore nevid.

b) Scrieți un predicat `frunze/2` care determina lista de frunze pentru un arbore binar dat ca parametru.

?-
frunze(arb(1,arb(2,nil,arb(3,nil,nil)),arb(4,arb(5,nil,nil),arb(6,nil,nil))),L).

L = [3, 5, 6] .

Pe săptămâna viitoare!

Laboratorul 7: MicroHaskell

MicroHaskell

Fișierul microHaskell.hs conține un mini-limbaj funcțional, împreună cu semantica lui denotațională.

```
type Name = String

data  Value  =  VBool Bool
        |VInt Int
        |VFun (Value -> Value)
        |VError

data  Hask  = HTrue | HFalse
        |HIf Hask Hask Hask
        |HLit Int
        |Hask :==: Hask
        |Hask :+:  Hask
        |HVar Name
          |HLam Name Hask
          |Hask :$: Hask
        deriving (Read, Show)
```

1) Completați funcția de evaluare a unei expresii de tip `Hask`.

Definim comanda:

```
run :: Hask -> String
run pg = showV (hEval pg [])
```

Astfel, `run pgm` va întoarce rezultatul evaluării (rulării) programului `pgm`.

2.1) Scrieți mai multe programe si rulați-le pentru a vă familiariza cu sintaxa.

2.2) Adăugați operația de înmulțire pentru expresii, cu precedență mai mare decât a operației de adunare. Definiți semantica operației de înmulțire.

2.3) Folosind funcția `error`, înlocuiți acolo unde este posibil valoarea `VError` cu o eroare care să precizeze motivul apariției erorii.

2.4) Adăugați expresia `HLet Name Hask Hask` ca alternativă în definirea tipului

`Hask`. Semantica acestei expresii este cea uzuală: `HLet x ex e` va evalua `e` într-un mediu în care `x` are valoarea lui `ex` în mediul curent. De exemplu, dacă definim

h1 = HLet “x” (HLit 3) ((HLit 4) :+: HVar “x”)

atunci `run h1` va întoarce “7”.

3) Făcând o copie a programului, înlocuiți tipul funcției `hEval :: Hask -> HEnv -> Value` cu `hEval :: Hask -> HEnv -> Maybe Value` inlocuind aruncarea erorilor cu utilizarea elementului `Nothing`.

4) La fel ca exercitiul 3, doar ca rezultatul functiei va fi de tip Either String Value, unde varianta de eroare va fi de tip Left cu un mesaj string corespunzator.

Laboratorul Monade

I. Monade. Monada `Maybe`

Lucrați în fișierul `mMaybe.hs`, care conține definiția monadei `Maybe`. Definiția este comentată deoarece monada `Maybe` este definită în GHC.Base

0. Înțelegeți funcționarea operațiilor monadice (»=) și `return`

```
 return 3 :: Maybe Int
Just 3
(Just 3)  >>= (\ x -> if (x>0) then Just (x*x) else Nothing)
Just 9
```

Uneori vom folosi operația derivată (»)

```
ma >> mb = ma >>= \_ -> mb
```

```
(Just 3)  >> Nothing
 Nothing
(Just 3)  >> (Just 6)
 Just 6
```

1. Definiți operatorul de compunere a funcțiilor îmbogățite

```
(<=<) :: (a -> Maybe b) -> (c -> Maybe a) -> c -> Maybe b
f <=< g = (\ x -> g x >>= f)
```

1.1 Creați singuri exemple prin care să înțelegeți funcționarea acestui operator.

1.2 Definiți proprietatea

```
 asoc :: (Int -> Maybe Int) -> (Int -> Maybe Int) -> (Int -> Maybe Int) -> Int -> Bool
```

care pentru trei funcții date verifică asociativitatea operației (<=<):

```
  h <=< (g <=< f) $ x = (h <=< g) <=< f $ x
```

Verificați proprietatea pentru funcții particulare folosind `QuickCheck`.

2. Definim

```
pos :: Int -> Bool
pos  x = if (x>=0) then True else False
```

```
foo :: Maybe Int ->  Maybe Bool
foo  mx =  mx  >>= (\x -> Just (pos x))
```

2.1 Înțelegeți ce face funcția `foo`.

2.2 Definiți funcția `foo` folosind notația `do`.

3. Vrem să definim o funcție care adună două valori de tip `Maybe Int`

```
addM :: Maybe Int -> Maybe Int -> Maybe Int
addM mx my = undefined
```

Exemplu de funcționare:

```
addM (Just 4)  (Just 3)
Just 7
addM (Just 4)  Nothing
Nothing
addM Nothing  Nothing
Nothing
```

3.1 Definiți `addM` prin orice metodă (de exemplu, folosind șabloane).

3.2 Definiți `addM` folosind operații monadice și notația `do`.

Notația `do` și secvențiere

4. Să se treacă în notația `do` urmatoarele funcții:

```
cartesian_product xs ys = xs >>= ( \x -> (ys >>= \y-> return (x,y)))
```

```
prod f xs ys = [f x y | x <- xs, y<-ys]
```

```
myGetLine :: IO String
myGetLine = getChar >>= \x ->
      if x == '\n' then
          return []
      else
          myGetLine >>= \xs -> return (x:xs)
```

5. Să se treacă în notația cu secvențiere urmatoarea funcție:

```
prelNo noin =  sqrt noin
ioNumber = do
     noin  <- readLn :: IO Float
     putStrLn $ "Intrare\n" ++ (show noin)
     let  noout = prelNo noin
     putStrLn $ "Iesire"
     print noout
```

II. Monada `Writer log`

Pentru următoarele exerciții lucrați cu fișierul `mWriter.hs`.

1. Fișierul `mWriter.hs` conține o definiție a monadei `Writer String` (puțin modificată pentru a compila fără opțiuni suplimentare):

```
newtype WriterS a = Writer { runWriter :: (a, String) }
```

1.1 Definiți funcțiile `logIncrement` și `logIncrement2` din curs și testați funcționarea lor.

1.2 Definiți funcția `logIncrementN`, care generalizează `logIncrement2`, astfel:

```
logIncrementN :: Int -> Int -> WriterS Int
logIncrement x n = undefined
```

Exemplu de funcționare:

```
runWriter $ logIncrementN 2 4
(6,"increment:2\nincrement:3\nincrement:4\nincrement:5\n")
```

2 Modificați definiția monadei `WriterS` astfel încât să producă lista mesajelor logate și nu concatenarea lor. Pentru a evita posibile confuzii, lucrați în fișierul mWriterL.hs . Definiți funcția `logIncrementN` în acest context.

```
newtype WriterLS a = Writer {runWriter :: (a, [String])}
```

Exemplu de funcționare:

```
runWriter $ logIncrementN 2 4
(6,["increment:2","increment:3","increment:4","increment:5"])
```

Funcția `map` în context monadic

Vom lucra tot în fișierul mWriterL.hs

3. În mod uzual, primul argument al funcției `map` este o funcție `f :: a -> b`, de exemplu:

```
map (\x -> if (x>=0) then True else False) [1,-2,3]
[True,False,True]
```

În context monadic, funcția `f` este îmbogățită, adică întoarce o valoare monadică:

```
isPos :: Int -> WriterLS Bool
isPos x = if (x>= 0) then (Writer (True, ["poz"])) else (Writer (False, ["neg"]))
```

Ce se întâmplă când aplicăm `map`?

```
map isPos [1,2,3]
```

Obțineți un mesaj de eroare! Funcția `map` a întors o listă de rezulate monadice, care pot fi vizualizate astfel:

```
 map runWriter $ map isPos [1,-2,3]
[(True,["poz"]),(False,["neg"]),(True,["poz"])]
```

Problemă: cum procedăm dacă dorim ca efectele să fie înlănțuite, iar rezultatul final să fie lista rezultatelor?

4. Definiți o funcție care se comportă similar cu `map`, dar efectul final este înlănțuirea efectelor. Signatura acestei funcții este:

```
mapWriterLS :: (a -> WriterLS b) -> [a] -> WriterLS [b]
```

Exemplu de funcționare:

```
 runWriter $ mapWriterLS isPos [1,-2,3]
([True,False,True],["poz","neg","poz"])
```

5. Definiți funcții asemănătoare cu `mapWriterLS` pentru monadele `WriterS` și `Maybe` din exercițiile anterioare.

III. Monada `Reader`

În continuare, vom exersa monada `Reader`.

Monada `Reader` este definită în fișierul mReader.hs

Citiți și înțelegeți exemplul de la curs.

1. Definim tipul de date

data Person = Person { name :: String, age :: Int }

1.1 Definiți funcțiile

```
showPersonN :: Person -> String
showPersonA :: Person -> String
```

care afișează "frumos" numele și vârsta unei persoane, după modelul

```
showPersonN $ Person "ada" 20
"NAME:ada"

showPersonA $ Person "ada" 20
"AGE:20"
```

1.2 Combinând funcțiile definite la punctul 1.1, definiți funcția

```
showPerson :: Person -> String
```

care afișează "frumos" toate datele unei persoane, după modelul

```
showPerson $ Person "ada" 20
"(NAME:ada,AGE:20)"
```

1.3 Folosind monada `Reader`, definiți variante monadice pentru cele trei funcții definite anterior. Variantele monadice vor avea tipul

```
mshowPersonN ::  Reader Person String
mshowPersonA ::  Reader Person String
mshowPerson ::  Reader Person String
```

Exemplu de funcționare:

```
runReader mshowPersonN  $ Person "ada" 20
"NAME:ada"

runReader mshowPersonA  $ Person "ada" 20
"AGE:20"

runReader mshowPerson  $ Person "ada" 20
"(NAME:ada,AGE:20)"
```