

Introducere în Prolog

1 Obiective

Lucrarea de față vă familiarizează cu principalele concepte din limbajul Prolog: fapte, reguli și întrebări. Pe lângă acestea se vor prezenta: tipurile de date utilizate în Prolog și regulile de unificare.

1.1 Utilizări ale Prolog-ului

Prolog a fost creat în 1972 de către Alain Colmerauer și Philippe Roussel, bazat pe interpretarea procedurală a clauzelor de tip Horn. Popularitatea limbajului a crescut în anii 80 și 90 prin Europa, USA și Japonia pentru variate sisteme. A fost cel mai folosit pentru dezvoltarea unui set particular de aplicații pentru natural language processing, expert systems (animal care - Auspig, water distribution - WADNES and SERPES, sport advisor - Perfect Pitch) și environmental systems (weather predictions - MM4 Weather Modeling System). Această creștere a fost datorată vitezei ridicate a dezvoltării incrementale și capabilitățile de prototipizare în rezolvarea problemelor de tip AI. Urmând să primească extensii pentru programarea orientată obiect. Alte arii de utilizare includ: demonstrarea teoremelor, planificare automată și intenția originală de natural language processing.

În zilele noastre, câteva din utilizările Prolog pot fi găsite în (aplicații industriale, medicale și comerciale):

- expert systems care rezolvă probleme fără ajutor uman (e.g. automatically planning, monitoring, controlling and troubleshooting complex systems)
- decision support systems care ajută organizații în luarea deciziilor (e.g. decision systems for medical diagnoses)
- online support service pentru clienți

Prolog a fost utilizat într-un system capabil să răspundă la întrebări puse în limbaj natural, denumit Watson (dezvoltat de IBM). În mod specific, a fost utilizat pentru pattern matching pe natural language parse trees.

1.2 De ce ar trebui să înveți Prolog?

Învățarea Prolog-ului este extrem de benefică pentru studenți din mai multe motive. În primul rând, îi introduce în paradigma programării logice, oferindu-le o perspectivă unică asupra rezolvării problemelor, distinctă de abordările convenționale. Acest lucru stimulează o înțelegere mai profundă a logicii și a raționamentului deductiv, abilități esențiale aplicabile în diverse domenii. Suportul său pentru programarea logică cu constrângeri îmbogățește și mai mult abilitățile de rezolvare a problemelor, în special în problemele de satisfacere a constrângerilor. În plus, proeminența Prolog-ului în domeniul inteligenței artificiale (IA) și al sistemelor experte îi expune pe studenți la tehnologii de vârf și îi pregătește pentru roluri în cercetarea și dezvoltarea IA. Prolog facilitează, de asemenea, explicabilitatea ML oferind o abordare transparentă și bazată pe reguli pentru rezolvarea problemelor, contribuind la înțelegerea și interpretarea modelelor de învățare automată. Cu Prolog, studenții adoptă o paradigmă de gândire nouă, în care soluțiile sunt derivate din reguli logice și fapte în loc de instrucțiuni explicite. În plus, suportul Prolog-ului pentru recursivitate îl face ușor de înțeles și de implementat pentru algoritmi recursivi, consolidând conceptele de recursivitate învățate în cursurile de structuri de date și algoritmi. Natura sa bazată pe interpretor permite execuția și testarea instantanee a codului, oferind feedback imediat și facilitând dezvoltarea iterativă. În cele din urmă, învățarea Prolog-ului nu numai că îmbunătățește abilitățile de rezolvare a problemelor ale studenților, dar oferă și un rezumat al structurilor de date și algoritmilor într-o paradigmă nouă și captivantă.

2 Considerații teoretice

2.1 Paradigma logică

Paradigma de programare este un stil fundamental de programare care dictează:

- Modul de **reprezentare** a datelor (ex: fapte, variabile, clase etc.)
- Modul de **prelucrare** a datelor (ex: asignări, comparații, evaluări etc.)

Paradigma logică face parte din paradigma declarativă. Un limbaj de programare declarativ răspunde la întrebarea **CE** trebuie să facă un program. Să considerăm următorul exemplu în limbajul declarativ SQL:

```
SELECT nume, prenume FROM Studenți WHERE an = 3
```

Instrucțiunea de mai sus nu îi spune interpretorului SQL cum să realizeze căutarea, ci îi spune doar ce condiție trebuie să respecte rezultatul. Prolog este un limbaj de programare logic, deci implicit și declarativ.

În opoziție, un limbaj de programare imperativ (ex: C, Pascal, C++, C#, Java etc.) răspunde la întrebarea „**CUM** trebuie să rezolve programul o anumită problemă”.

2.2 Concepte Prolog

Instrucțiunile în Prolog sunt reprezentate de fapte și reguli. După ce programul sursă Prolog a fost consultat de interpretor se pot pune întrebări. Răspunsul la întrebări se determină pe baza faptelor și a regulilor folosind tehnica backtracking-ului.

Tot ceea ce nu este cunoscut sau nu poate fi demonstrat este considerat fals.

Comentariile:

- pe o linie încep cu simbolul %
- pe mai multe linii se încadrează între simbolurile `/*` și `*/` (similar comentariilor din limbajul de programare C)

În cadrul acestui laborator, există două opțiuni de interpretoare Prolog:

2.2.1 (Opțiunea1) SWI Prolog (necesită instalare) & Notepad++

Programul sursă este scris în Notepad++ într-un fișier de format `*.pl` și va fi consultat în interpretorul SWI Prolog prin menu bar File -> Consult. Pentru consulturile următoare a aceluiași fișier se poate folosi File -> Reload modified files.

2.2.2 (Opțiunea2) SWISH Prolog – platformă online (fără instalare)

În contrast, platforma SWISH Prolog încorporează atât editorul cât și interpretatorul și permite folosirea ambelor prin intermediul unei interfețe simple.

2.3 Tipuri de date

Singurul tip de date în Prolog este termenul. Termenii pot să fie: atomi, numere, variabile sau termeni compuși (structuri). Figura de mai jos prezintă o clasificare a tipurilor de date în Prolog.

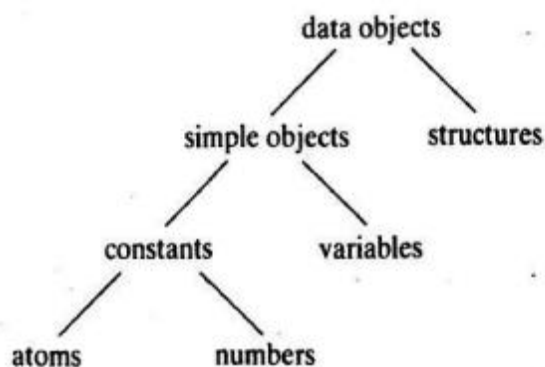


Figure 1 – Prolog data types¹

Simple:

- constante
 - numere (ex: 47, 6.3 etc.)
 - În contrast cu C – Prolog nu necesită declararea unui tip de date ca ,int' sau ,float'
 - simboluri/atom (ex: girafa, 'Romania', 'antilopa Gnu' etc.)
 - Dacă au caractere speciale trebuie înconjurate de ghilimele simple
- variabile
 - Încep cu **literă mare** sau cu **_** (ex: X, Aux, _12 etc.)
 - Un singur **_** reprezintă o variabilă anonimă/liberă

Compuse:

- structuri (ex: t(1, t(-2, nil, nil), t(8,nil,nil)) etc.)
 - liste (ex: [], [1, 2, 3], [1, 2 | _] etc.)
 - [] = lista vidă
 - Lista [1,2,3] este reprezentată intern prin '.(1, '.(2, '.(3, []))
 - Șablonul [**H**|**T**] împarte lista în H (= elementul din capul listei) și T (= **lista** ce conține restul elementelor)
 - string-uri (ex: "Hello World")

2.3.1 Exerciții pe Tipuri de Date

1. Ce tip de date sunt următorii termeni:

- | | | |
|---------|------------|-----------------------|
| a. X | d. hello | g. [a, b, c] |
| b. 'X' | e. Hello | h. [A, B, C] |
| c. _138 | f. 'Hello' | i. [Ana, are, 'mere'] |

2. Ce fac următoarele predicate predefinite în Prolog:

var(Term),
nonvar(Term),
number(Term),
atom(Term),
atomic(Term)

2.3.2 Informații adiționale despre liste in Prolog

Știind că putem folosi șablonul **[H|T]** poate fi folosit pentru a separa o listă în cap (primul element) și coadă (*Observație*: coada este restul listei și este o listă de sine stătătoare).

[1,2,3] poate fi scris ca [1 | [2,3]].

Întrebare: Traversarea unei liste se face prin folosirea șablonului, dar care este tehnica de programare folosită?

Răspuns: Recursivitate. Primul element este separat de coadă și un apel recursiv este aplicat pe coadă, astfel se poate și procesa primul element.

Vizualizând șablonul într-o manieră recursivă, poate fi scris ca:

[1 | [2 | [3 | []]]]

2.4 Fapte

Faptele sunt predicate care sunt **tot timpul** adevărate. Se mai numesc și axiome (din matematică). Pot avea zero, unu sau mai multe argumente. **Aritatea** unui predicat reprezintă numărul de argumente al acelui predicat.

Exemple:

```
suntem_in_sala_108.  
animal(elefant).  
animal('antilopa Gnu').  
inaltime(girafa, 5.5). % înălțimea îi în metrii  
arbore( t(1, t(-2, nil, nil), t(8,nil,nil)) ).
```

2.5 Întrebări

Întrebările pot fi văzute ca și scopuri ale programului Prolog. Răspunsul la o întrebare poate fi afirmativ sau negativ. Dacă folosim variabile în întrebare putem obține și alte informații.

Exemple:

```
?- suntem_in_sala_108.  
true.  
  
?- animal('antilopa Gnu').  
true.  
?- animal(X).  
X = elefant ; % repetă întrebarea cu ; sau n sau spațiu  
X = 'antilopa Gnu' ;  
false. % nu mai există un alt animal definit în program
```

2.6 Unificare

Simbolul = realizează unificarea dintre cei 2 termeni.

Valoare este orice tip de dată care nu este/nu conține o variabilă neinițializată.

Termen 1	Termen 2	Operație
valoare(/variabila inițializată)	Valoare (/variabila inițializată)	Comparație
valoare(/variabila inițializată)	variabila neinițializată	Asignare
variabila neinițializată	valoare(/variabila inițializată)	Asignare
variabila neinițializată	variabila neinițializată	Variabilele devin sinonime

		(point-ează către același loc în memorie)
--	--	---

Observație: Unificarea din prolog nu funcționează ca atribuirea din C.

2.6.1 Exerciții folosind Unificarea

Testează următoarele unificări

- a. $\text{?- } a = a.$
- b. $\text{?- } a = b.$
- c. $\text{?- } 1 = 2.$
- d. $\text{?- 'ana' = 'Ana'.$
- e. $\text{?- } X = 1, Y = X.$
- f. $\text{?- } X = 3, Y = 2, X = Y.$
- g. $\text{?- } X = 3, X = Y, Y = 2.$
- h. $\text{?- } X = \text{ana}.$
- i. $\text{?- } X = \text{ana}, Y = \text{'ana'}, X = Y.$
- j. $\text{?- } a(b,c) = a(X,Y).$
- k. $\text{?- } a(X,c(d,X)) = a(2,c(d,Y)).$
- l. $\text{?- } a(X,Y) = a(b(c,Y),Z).$
- m. $\text{?- } \text{tree}(\text{left}, \text{root}, \text{Right}) = \text{tree}(\text{left}, \text{root}, \text{tree}(a, b, \text{tree}(c, d, e))).$
- n. $\text{?- } k(s(g),t(k)) = k(X,t(Y)).$
- o. $\text{?- } \text{father}(X) = X.$
- p. $\text{?- } \text{loves}(X,X) = \text{loves}(\text{marsellus}, \text{mia}).$
- q. $\text{?- } [1, 2, 3] = [a, b, c].$
- r. $\text{?- } [1, 2, 3] = [A, B, C].$
- s. $\text{?- } [abc, 1, f(x) \mid L2] = [abc \mid T].$
- t. $\text{?- } [abc, 1, f(x) \mid L2] = [abc, 1, f(x)].$

Observație1: O listă în formatul $[1,2,3]$ este echivalentă cu $[1,2,3 \mid []]$ dar și cu $[1 \mid [2 \mid [3 \mid []]]]$.

Observație2: Șablonul **[H|T]** poate fi extins la $[H1, H2 \mid T]$ sau $[H1, H2, H3 \mid T]$ sau mai multe. Este important să reținem că primul element nu poate fi null, cu toate că coada poate să fie listă vidă []. Se poate infera astfel că șablonul de tip $[H1, H2, \dots, Hn \mid T]$ poate să fie unificat doar cu o listă cu minim n elemente. Acest fapt înseamnă că este necesar să avem la fel de multe condiții de oprire ca numărul de capuri de listă (Hn) pentru a adresa toate cazurile posibile.

Observație:* (Rețineți) Unificarea din Prolog și atribuirea din C funcționează în moduri diferite cu toate că folosesc același simbol =. Pe când atribuirea din C funcționează unidirecțional, unificarea din Prolog este bidirecțională (ordinea

termenilor nu contează). În unele cazuri este mai ușor să fie interpretat ca o comparație (deoarece chiar returnează true/false), deși nici această perspectivă nu este una corectă în întregime.

2.7 Reguli

Regula reprezintă definiția unui predicat sub forma clauzei de tip Horn:

$p(\dots)$ dacă $p_{11}(\dots)$ și $p_{12}(\dots)$ și ... și $p_{1n}(\dots)$.

...

$p(\dots)$ dacă $p_{m1}(\dots)$ și $p_{m2}(\dots)$ și ... și $p_{mk}(\dots)$.

$p(\dots)$ = capul clauzei (maxim un predicat)

$p_{11}(\dots)$ și ... = corpul clauzei (zero, unu sau mai multe predicate)

fapt = clauză fără corp

întrebare = clauză fără cap

Simboluri speciale (operatori) în Prolog:

`:-` = „dacă”

`,` = „și”

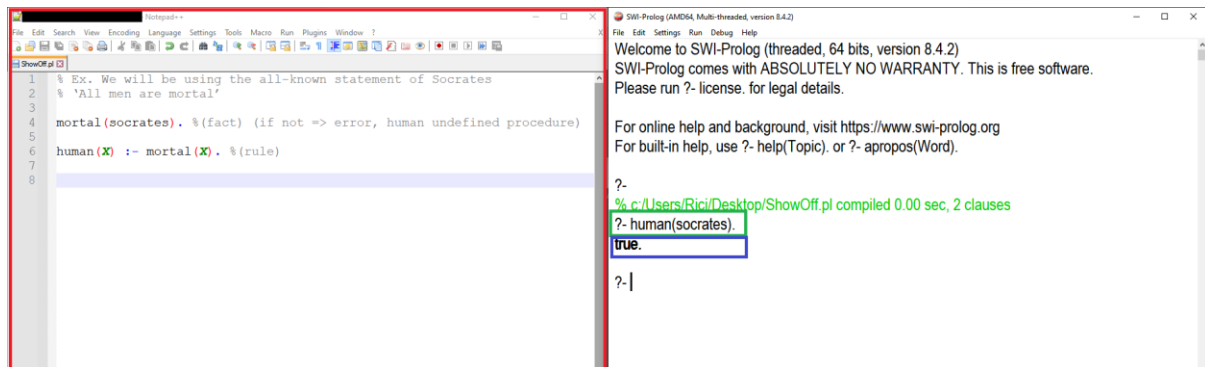
`;` = „sau” (poate fi realizat și prin scrierea mai multor clauze pentru același predicat)

Exemple:

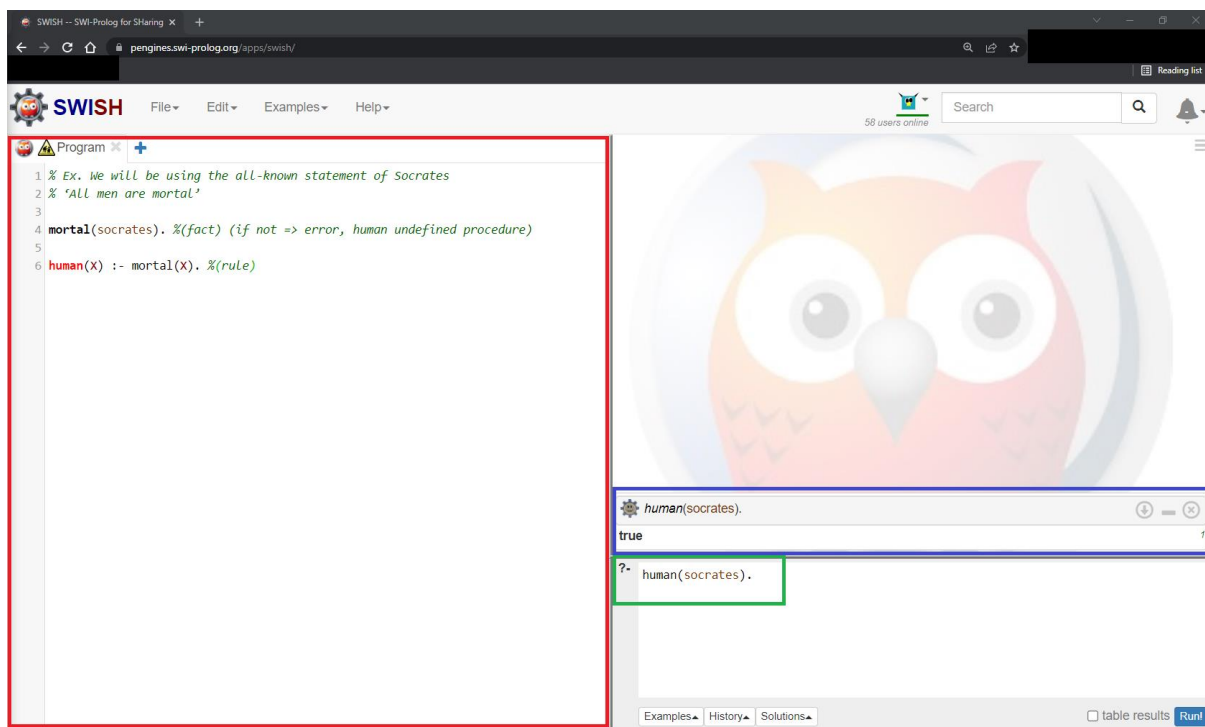
```
mai_inalt(X,Y) :- inaltime(X,Hx), inaltime(Y,Hy), Hx>Hy.  
% prima data luam înălțimea lui X, apoi luam înălțimea lui Y  
% și în final verificăm inegalitatea  
  
drum(X,Y) :- muchie(X,Y).  
drum(X,Y) :- muchie(X,Intermediar), drum(Intermediar,Y).  
% drumul între nodurile X și Y poate să fie conexiunea directă  
% între cele 2 noduri SAU dacă nu există conexiune directă ne  
% folosim de o conexiune indirectă printr-un nod intermediar
```


2.8 SWISH vs SWI

Această primă imagine prezintă setup-ul de SWI Prolog & Notepad++.



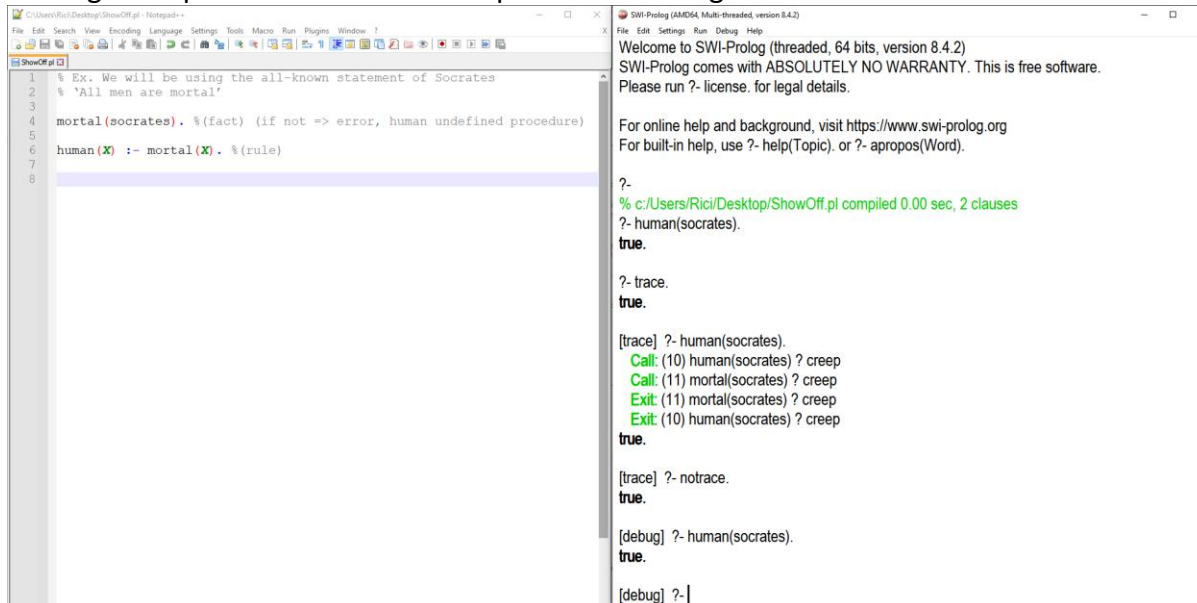
Următoarea imagine prezintă site-ul SWISH Prolog. Încadrarea prin culori a fost ajustată părților corespundente a celor două opțiuni.



2.9 Debugging în Prolog

Debugging-ul în Prolog se numește tracing iar *,trace'* este comanda Prolog care permite debugging-ul codului Prolog.

Tracing făcut prin versiunea SWI este prezentat în imaginea următoare:



```
1 % Ex. We will be using the all-known statement of Socrates
2 % 'All men are mortal'
3
4 mortal(socrates). %(fact) (if not => error, human undefined procedure)
5
6 human(X) :- mortal(X). %(rule)
7
8
```

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.2)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
% c:/Users/Rici/Desktop/ShowOff.pl compiled 0.00 sec, 2 clauses
?- human(socrates).
true.

?- trace.
true.

[trace] ?- human(socrates).
Call: (10) human(socrates) ? creep
Call: (11) mortal(socrates) ? creep
Exit: (11) mortal(socrates) ? creep
Exit: (10) human(socrates) ? creep
true.

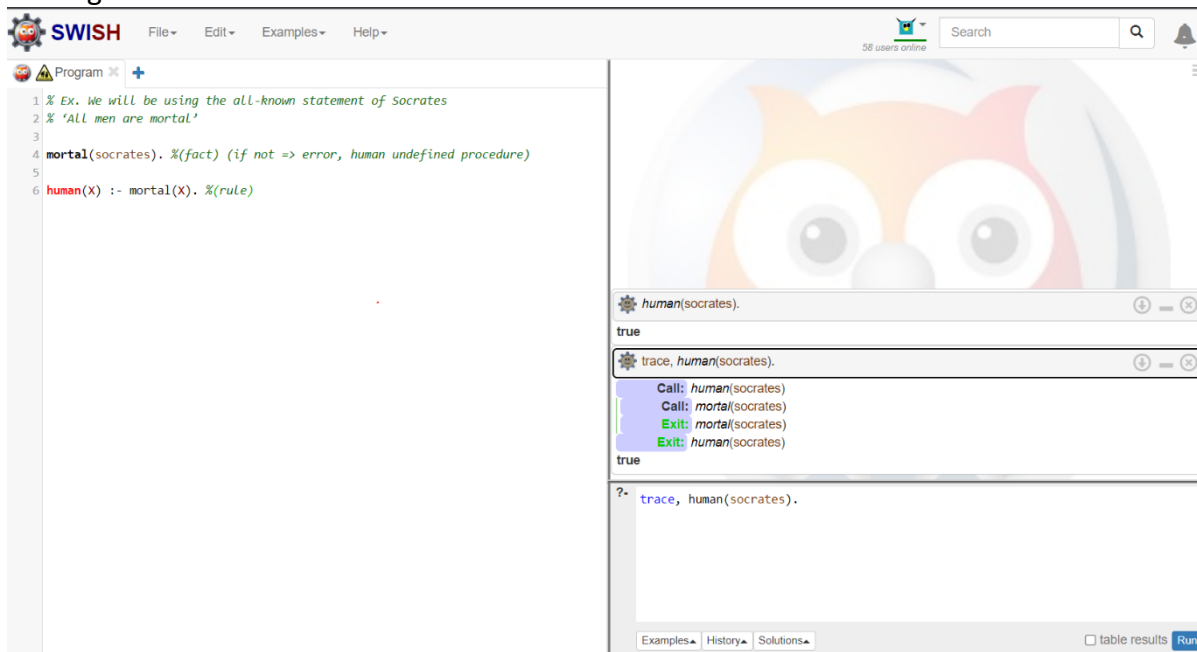
[trace] ?- notrace.
true.

[debug] ?- human(socrates).
true.

[debug] ?-
```

Este necesar să activăm opțiunea de trace prin rularea comenzii *,trace'* înainte de întrebare. Poate fi deactivat prin folosirea comenzii *,notrace'*.

Tracing în SWISH:



```
1 % Ex. We will be using the all-known statement of Socrates
2 % 'All men are mortal'
3
4 mortal(socrates). %(fact) (if not => error, human undefined procedure)
5
6 human(X) :- mortal(X). %(rule)

```

```
human(socrates).
true

trace, human(socrates).
Call: human(socrates)
Call: mortal(socrates)
Exit: mortal(socrates)
Exit: human(socrates)
true

?- trace, human(socrates).
```

Această versiunea necesită scrierea comenzii *,trace'* înaintea întrebării care urmează să fie apelată.

2.10 Backtracking

O funcționalitate intrinsecă Prolog-ului este backtracking-ul. Răspunsul la întrebările puse în Prolog poate fi vizualizat (printr-o simplificare) ca un proces Depth-First Search (DFS). Când interpretorul caută un răspuns, o face printr-o căutarea-în-adâncime, dacă întrebarea este repetată (prin ; sau Next) următorul răspuns este căutat prin procesul de backtracking. Rețineți că acest răspuns nou trebuie să fie unul distinct, altfel ar rezulta într-o buclă infinită a aceluiași răspuns.

Vom implementa următoarele linii de cod care decid cine poate să țină o petrecere prin trecerea printr-o listă de cerințe:

```
% hold_party/1 este o regulă care depinde de execuția reușită  
% a faptelor birthday/1 și happy/1  
hold_party(X):-  
    birthday(X),  
    happy(X).  
  
% o serie de fapte de tip birthday/1 și happy/1  
birthday(alex).  
birthday(maria).  
birthday(adriana).  
happy(ana).  
happy(george).  
happy(adriana).
```

Observație1. Rețineți că `"/x"` reprezintă aritatea unui predicat = numărul de argumente.

Observație2. În general, vrem ca faptele să fie specifice -> după cum puteți vedea, conțin constante. Simultan, vrem ca regulile să fie generale -> folosesc variabile. De ce? Când punem întrebări în Prolog, vrem să îi oferim abilitatea de a găsi el un răspuns, oricât de vagă este întrebarea. Dacă regula ar fi specifică, ar putea găsi un singur răspuns.

Vom urmări execuția predicatului *hold_party/1* prin SWISH Prolog:

```

1 hold_party(X):-
2   birthday(X),
3   happy(X).
4
5 birthday(alex).
6 birthday(maria).
7 birthday(adriana).
8 happy(ana).
9 happy(george).
10 happy(adriana).
11

```

```

trace, hold_party(X).
Call: hold_party(_4082)
Call: birthday(_4082)
Exit: birthday(alex) } - unifies with the first birthday/1 fact it finds -> X will unify with 'alex'
Call: happy(alex)
Fail: happy(alex) } - searches to see if happy(alex) exists, as it doesn't -> it fails
Redo: birthday(_4082) } - 'Redo' specifies the backtracking, it searches for a distinct birthday/1 fact and finds maria
Exit: birthday(maria)
Call: happy(maria)
Fail: happy(maria) } - Again it fails, as there is no happy(maria) fact
Redo: birthday(_4082) } - The final backtracking, it unifies with birthday(adriana) -> X=adriana
Exit: birthday(adriana)
Call: happy(adriana)
Exit: happy(adriana) } - Verifies if happy(adriana) exists - and it does, exits with success
Exit: hold_party(adriana) -> exits the hold_party call and returns answer
X = adriana
?- trace, hold_party(X).

```

2.11 Recursivitate

Vom separa recursivitatea empiric într-un subset de părți fundamentale, vom folosi implementarea din C a factorialului:

1. Recursivitatea necesită, prin definiție, apelul funcției în cadrul funcției
2. De asemenea conține un pas, în cazul factorialului, acest pas este o decrementare (sau incrementare)
3. În cadrul recursivității există și o procesare, pentru factorial este înmulțirea
4. Vom urmări ce se întâmplă când încercăm să calculăm un factorial de 3.
 - a. Înmulțim cu 3 & decrementăm -> recursivitate
 - b. Înmulțim cu 2 & decrementăm -> recursivitate
 - c. Înmulțim cu 1 & decrementăm -> recursivitate
 - d. Înmulțim cu 0 & decrementăm -> recursivitate
 - e. Înmulțim cu -1 & decrementăm -> recursivitate
 - f. ...
 - g. Overflow – Înmulțim cu 2147483647 & decrementăm -> recursivitate
 - h. *Puteți să identificați problema?* Ignorând faptul că se înmulțește cu 0, este o buclă infinită. O parte importantă a recursivității este condiția de oprire.

Vom considera de exemplu, un student care a fost acasă după sesiunea de examene și încearcă să ajungă înapoi pentru începerea semestrului.

```
% fapta on_route/1
on_route(camin).

% regula on_route/1 - o regulă recursivă
on_route(Place):-
    move(Place, Method, NewPlace),
    on_route(NewPlace).

% fapta move/3
move(acasa, taxi, gara).
move(gara, tren, cluj).
move(cluj, bus, camin).
```

Vom urmări execuția predicatului *on_route/1* prin SWI-Prolog:

```
Warning: c:/users/ardel/desktop/on_route.pl:4:
Warning: Singleton variables: [Method]
% c:/Users/ardel/Desktop/on_route.pl compiled 0.00 sec, 5 clauses
?- trace.
true.

[trace] ?- on_route(acasa).
Call: (10) on_route(acasa) ? creep
Call: (11) move(acasa, _57006, _56946) ? creep % Place unified with acasa (line 4) -> calls line 5
Exit: (11) move(acasa, taxi, gara) ? creep % it searches for a move with the
Call: (11) on_route(gara) ? creep % NewPlace unified with gara and recursion (line 6) is called
Call: (12) move(gara, _59274, _59214) ? creep % Through recursion we are back at line 4
Exit: (12) move(gara, tren, cluj) ? creep % and calls line 5 with Place=gara
Call: (12) on_route(cluj) ? creep
Call: (13) move(cluj, _61542, _61482) ? creep
Exit: (13) move(cluj, bus, camin) ? creep
Call: (13) on_route(camin) ? creep
Exit: (13) on_route(camin) ? creep
Exit: (12) on_route(cluj) ? creep
Exit: (11) on_route(gara) ? creep
Exit: (10) on_route(acasa) ? creep
true.

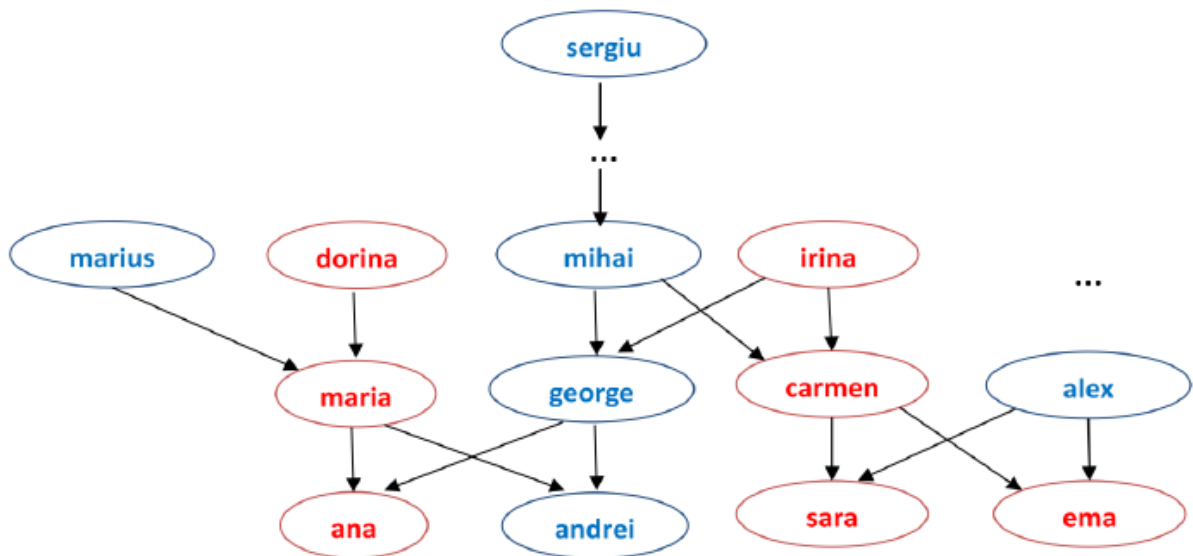
[trace] ?- |
```

Observație1. Nu uitați să consultați înainte de a pune întrebări în Prolog prin File->Consult care va rezulta într-un comentariu în verde.

Observație2. Nu uitați să activați ,trace'-ul.

3 Exercițiul final

1. Scrieți noi predicate pentru relațiile de rudenie.



Figură 1. Arbore genealogic

% Predicatul woman/1

woman(ana).

woman(sara).

woman(ema).

woman(maria). % ... adăugați restul faptelor acestui predicat

% Predicatul man/1

man(andrei).

man(george).

man(alex). % ... adăugați restul faptelor acestui predicat

% Predicatul parent/2

parent(maria, ana). % maria este părintele anei

parent(george, ana). % george este părintele anei

parent(maria, andrei).

parent(george, andrei). % ... adăugați restul faptelor acestui predicat

% Predicatul mother/2

% X este mama lui Y, daca X este femeie și X este părintele lui Y

mother(X,Y) :- woman(X), parent(X,Y).

Pentru SWI Prolog (varianta instalată):

Programul sursă (ex: genealogy.pl) poate fi consultat folosind meniul din interpretor: *File -> Consult* sau se poate scrie în interpretor următorul predicat predefinit:

```
?- consult('C:/Users/student/Desktop/genealogy.pl').  
true. % Dacă nu există nici o eroare va returna true
```

1.1. Testați următoarele întrebări:

**Rețineți că: întrebările sunt precedate de operatorul '?-', care este deja scris, restul liniilor reprezintă răspunsurile oferite de Prolog întrebărilor precizate):*

```
?- man(george).    % este george bărbat?  
true.  
?- man(X).         % cine este bărbat?  
X = andrei ? ;    % repetăm întrebarea cu ; sau n sau spațiu  
X = george ? ;  
X = alex ? ;  
false.
```

```
?- parent(X, andrei). % Cine sunt părinții lui andrei?  
X = maria ? ;  
X = george ? ;  
false.  
?- parent(maria, X). % Cine sunt copii mamei?  
X = ana ? ;  
X = andrei ? ;  
false.
```

```
?- mother(ana, X). % Cine sunt copii anei?  
false.  
?- mother(X, ana). % Care este mama anei?  
X = maria ? ; % repetăm întrebarea = mai are ana o alta mamă?  
false.
```

- 1.2. Scrieți predicatul father/2.
- 1.3. Completați predicatele man/1, woman/1 și parent/2 pentru a acoperi arborele genealogic de mai sus.
- 1.4. Testați următoarele întrebări:

?- father(alex, X).
 ?- father(X, Y).
 ?- mother(dorina, maria).

- 1.5. Testați următoarele predicate:

% Predicatul sibling/2

% X și Y sunt frați/surori dacă au cel puțin un părinte în comun

% și X diferit de Y

sibling(X,Y) :- parent(Z,X), parent(Z,Y), X\=Y.

% Predicatul sister/2

% X este sora lui Y dacă X este femeie și X și Y sunt frați/surori

sister(X,Y) :- sibling(X,Y), woman(X).

% Predicatul aunt/2

% X este mătușa lui Y dacă este sora lui Z și Z este părintele lui Y

aunt(X,Y) :- sister(X,Z), parent(Z,Y).

- 1.6. Scrieți predicatele brother/2, uncle/2, grandmother/2 și grandfather/2.
- 1.7. Urmăriți pașii pentru găsirea răspunsului la următoarele întrebări (prin utilizarea *trace*-ului):

?- aunt(carmen, X).
 ?- grandmother(dorina, Y).
 ?- grandfather(X, ana).

- 1.8. Scrieți predicatul ancestor/2, unde X este strămoșul lui Y dacă X este legat de Y printr-o serie (indiferent de număr) de relații de tip părinte.