Laboratorul 1 - Programare Logică și Funcțională

Seria 36

Februarie & Martie 2024

1 De ce programare logică?

- Programarea logică este utilă în strategii de căutare, prototipuri, rezolvare de puzzle-uri etc.
- Idei *declarative* apar în multe domenii din informatică:
  - conceptele din programarea logică sunt utile în AI și în bazele de date; demonstratoare automate, *model-checking*, *constrain programming*
  - sunt importante în analiza programelor, în semantica Web.

Prolog este cel mai cunoscut limbaj de programare logică. Este bazat pe logica clasică de ordinul I și funcționează pe bază de unificare și căutare.

În acest laborator, vom folosi implementarea **SWI-Prolog**:

- gratuit, folosit pentru predare, conține multe librării;
- `https://www.swi-prolog.org/`
- există și o variantă online: `https://swish.swi-prolog.org/`

În laboratorul de astăzi răspundem următoarelor două întrebări:

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

2 Sintaxă

Ca elemente de sintaxă întâlnim *constantele*, *variabilele* și *termenii*.

Constantele din limbajul Prolog sunt atomii și numerele.

- atomi: secvențe de litere, numere și _ care încep cu o literă mică, șiruri între apostrofuri și anumite simboluri speciale: `atom`, `my_atom`, `'Sir de Caractere'`, `'(@ *+'`, `+`;
- numere: `2`, `2.5`, `-33`.

În Prolog avem predefinit un predicat `atom` de aritate 1 (și notăm `atom/1`).

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

Variabilele sunt secvențe de litere, numere și _ care încep cu literă mare sau cu simbolul _. Variabila _ este o *variabilă anonimă*. Două apariții ale simbolului _ sunt variabile diferite și, în general, folosim _ atunci când nu vrem detalii despre variabila respectivă. Exemple: `X`, `_myVar`, `_`.

Termenii sunt constante sau variabile, iar *termenii compuși* au forma `p(t1, ..., tn)` unde `p` este un atom, iar `t1, ..., tn` sunt termeni. De exemplu,

```
father(oliver, ben).
father(oliver, X).
```

Un termen compus are, astfel, două atribute, și anume:

- un *nume* (*functor*): `father` în exemplul de mai sus;

- o *aritate* (numărul de argumente): 2 argumente în cazul de mai sus.

Observații:

- predicatele cu același nume, dar cu arități diferite, sunt predicate *diferite*;
- există predicate de aritate 0 (nu au argumente): de exemplu `true` și `false`;
- scriem `foo/n` pentru a indica predicatul `foo` de aritate `n`.

Pentru a defini un program în Prolog, vom avea o bază de cunoștințe (adevăruri în universul programului, *facts*) și reguli de inferență (*rules*). În general, o regulă este de forma

```
Head :- Body.
```

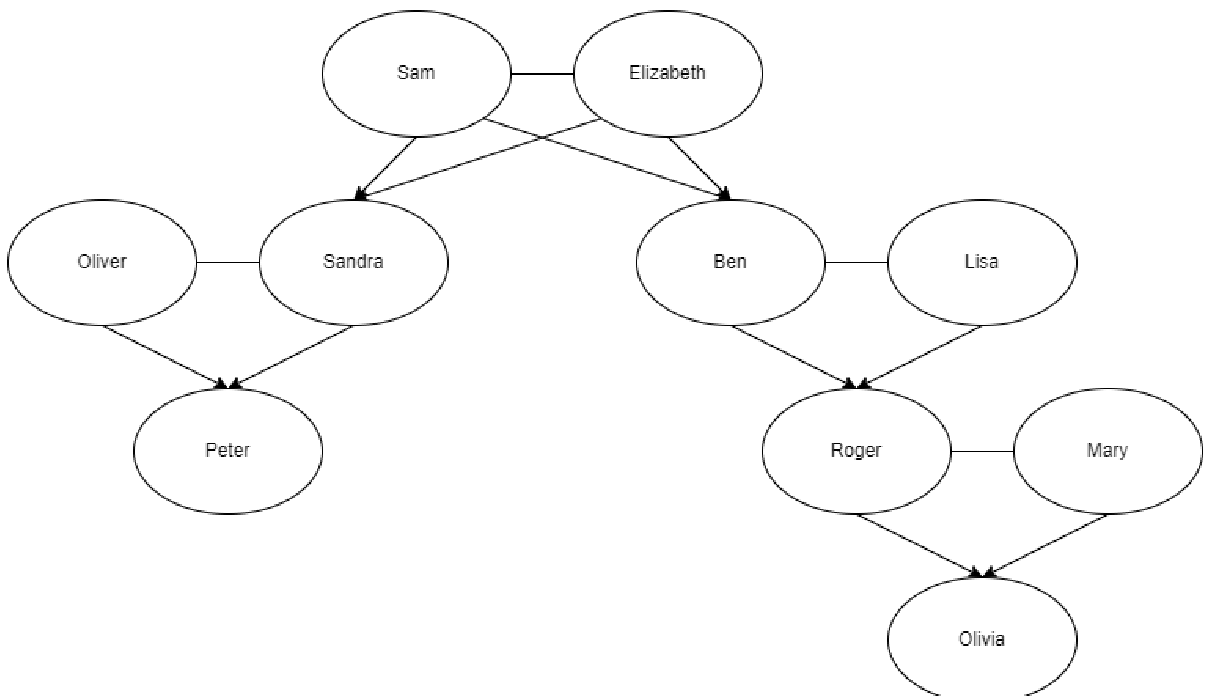
și citim "`Head` dacă `Body`" sau "dacă `Body` este adevărat, atunci `Head`" (implicația este de la dreapta - antecendent, la stânga - consecvent). Un adevăr (*fact*) este o regulă care nu are un `Body`.

Un exemplu de bază de cunoștințe:

```
male(sam).
male(oliver).
male(ben).
male(peter).
male(roger).

female(elizabeth).
female(sandra).
female(mary).
female(lisa).
female(olivia).

parent_of(sandra, sam).
parent_of(sandra, elizabeth).
parent_of(ben, sam).
parent_of(ben, elizabeth).
parent_of(peter, oliver).
parent_of(peter, sandra).
parent_of(roger, ben).
parent_of(roger, lisa).
parent_of(olivia, roger).
parent_of(olivia, mary).
```



Pentru a scrie o regulă, avem în vedere următoarele:

- `Head` este un predicat (termen compus);
- `Body` este o secvență de predicate, separate prin virgulă.

În Prolog, virgula se interpretează drept *conjuncție*, astfel că citim

```
P :- Q1, Q2, ..., Qn.
```

ca fiind formula $Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n \rightarrow P$.

De exemplu, putem adăuga în baza de cunoșințe un predicat `happy/1`. Vom pune acest predicat pentru Roger si Olivia:

```
happy(roger).
happy(olivia).
```

Vom defini ca regulă că Ben este *happy* dacă și fiul lui, și nepoata lui sunt *happy*.

```
happy(ben) :- happy(roger), happy(olivia).
```

Observație: în Prolog toate demonstrațiile sunt făcute într-un univers închis! În acest moment, putem demonstra că Roger, Olivia și Ben sunt *happy* pentru că avem această informație. Acest lucru nu înseamnă automat că toți ceilalți sunt *sad*, ci înseamnă doar că nu știm nimic despre ei. În Prolog dacă nu putem demonstra o proprietate P, înseamnă doar că nu am reușit să îi găsim demonstrația.

Vom defini o regulă ca Elizabeth să fie *happy* dacă cel puțin unul dintre copiii ei este *happy* (Sandra **sau** Ben). Pentru a introduce disjuncția în Prolog, avem două variante: sau cu operatorul `;`, sau scriind regula pe mai multe linii, câte una pentru fiecare disjunct. Astfel, putem avea una dintre următoarele două:

```
happy(elizabeth) :- happy(sandra) ; happy(ben).
```

SAU

```
happy(elizabeth) :- happy(sandra).
happy(elizabeth) :- happy(ben);
```

Important! Predicatele scrise (atât adevărurile cât și regulile de inferență) trebuie să fie grupate după atomii din `Head`. În caz contrar, există riscul erorilor de parsare, iar răspunsul dat de Prolog poate fi greșit. *Contează ordinea de scriere a clauzelor*

O *întrebare* în Prolog este o secvență de forma

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

Fiind dată o întrebare (țintă), Prolog caută răspunsuri. Va returna `true` sau `false` dacă întrebarea nu conține variabile, respectiv daca întrebarea conține variabile, atunci sunt căutate valori care fac toate predicatele din întrebare să fie satisfăcute. Daca nu se găsesc astfel de valori, răspunsul este `false`. Predicatele care trebuie satisfăcute se numesc *goals*.

Exemplu de întrebare:

```
?- happy(elizabeth).
true.
```

Putem pune întrebări care conțin și variabile, de exemplu:

```
?- parent_of(ben, X).
```

Observăm că Prolog răspunde cu

```
X = sam
```

dar nu este unicul răspuns. Pentru a-l obține pe următorul, folosim `;`, și obținem și

```
X = elizabeth
```

Putem defini reguli complexe folosind variabile. Vom defini predicatul `father_of/2`, astfel:

```
father_of(X, Y) :-
    parent_of(X, Y),
    male(Y).
```

Utilizăm în definiție ce înseamnă `father`: să fie părintele direct și să fie bărbat.

3 Aritmetică în Prolog

În Prolog avem operațiile aritmetice uzuale:

1. `+` pentru adunare;
2. `-` pentru diferență;
3. `*` pentru înmulțire;
4. `/` pentru împărțire pe floats;
5. `//` pentru câtul împărțirii;
6. `div` tot pentru câtul împărțirii;
7. `mod` pentru restul împărțirii;
8. `**` pentru ridicarea la o putere.

Toate operațiile de mai sus sunt simboluri binare, astfel că avem egalitate, de exemplu, între `+(3,5) = 3 + 5`.

ATENȚIE! În Prolog există mai multe simboluri pentru egalitate, fiecare cu alt scop!

1. `=` caută un unificator între termeni (vom vedea);
2. `==` verifică egalitatea structurală a termenilor;
3. `=:=` **compară egalitatea aritmetică**.

Astfel, `8 == 3 + 5`, respectiv `8 = 3 + 5` vor fi evaluate la `false`, dar `8 =:= 3 + 5` va fi evaluat la `true`.

Pentru a efectua calcule, nu putem folosi niciunul dintre operatorii de mai sus. Dacă vrem să evaluăm expresia aritmetica `3 + 5` și să stocăm rezultatul în variabila `X`, sintaxa care ne permite este cea de a utiliza operatorul `is/2`, sub forma `X is expression`.

Despre operatorul `is` avem următoarele:

- primește două argumente;
- al doilea argument trebuie să fie o expresie aritmetică validă, cu **toate** variabilele instanțiate;
- primul argument este fie un număr, fie o variabilă;
- dacă primul argument este un număr, atunci operatorul se comportă ca egalitatea aritmetică;
- dacă primul argument este o variabilă, atunci răspunsul este pozitiv dacă variabila poate fi unificată cu evaluarea expresiei aritmetice din al doilea argument.

Pentru condițiile *booleene* avem operatorii de aritate 2: `>`, `>=`, `<`, `=<` (atenție la ultimul din această listă).

Operatorii aritmetici se împart, astfel, în funcții și relații: adunarea, înmulțirea etc. sunt exemple de funcții aritmetice, și le putem scrie în Prolog în mod uzual,

```
?- X is 2 + (-3.2 * 7 - max(17, 3)) / 2 ** 5.
X = 0.7687499999999998
```

dar avem acces și la alte funcții precum `min/1`, `abs/1`, `sqrt/1`, `sin/1` etc. Relațiile sunt relațiile de ordine.

Un exemplu de prediat pentru suma a trei numere este `add/4`:

```
add(X, Y, Z, Result) :- Result is X + Y + Z.

?- add(1, 2, 3, Res).
Res = 6
```

4 Exerciții

Rezolvați următoarele exerciții

4.1 Exercițiul 1

Definiți, pentru baza de cunoștințe de mai sus, predicatele următoare: `mother_of/2`, `brother_of/2`, `sister_of/2`. `uncle_of/2`, `aunt_of/2`, `grandfather_of/2`, `grandmother_of/2`. Verificați aceste predicate punând întrebări și urmărind pe arborele genealogic de mai sus.

4.2 Exercițiul 2

Utilizând aceeași bază de cunoștințe de mai sus, definiți un predicat `ancestor_of/2`, care să verifice dacă al doilea argument este strămoș al primului. Predicatul trebuie să fie `true` pentru toate întrebările pe linia directă, de exemplu `?- ancestor_of(olivia, sam)`.

4.3 Exercițiul 3

Definiți un predicat `distance/3` pentru a calcula distanța dintre două puncte într-un plan 2-dimensional.

4.4 Exercițiul 4

Fie `write/1` un predicat care scrie argumentul primit (un atom) la STDOUT. Folosiți acest predicat pentru a defini un predicat nou, `write_n/2` care primește ca argumente un număr natural nenul și un atom și afișează la STDOUT un triunghi format din atomul primit și din dimensiunea primită. De exemplu, pentru `write_n(5, *)` se va afișa:

```
*****
****
***
**
*
```

Modificați afișarea pentru a obține reprezentarea:

```
    *
   **
  ***
 ****
*****
```

4.5 Exercițiul 5

Scrieți un predicat `min/3` care să calculeze minimul a două elemente. Ce obervați în implementare?

4.6 Exercițiul 6

Scrieți un program Prolog care, primind trei puncte în plan, verifică dacă ele determină un triunghi dreptunghic. Punctele sunt reprezentate ca perechi `(X, Y)`.