Programare Logică

Adrian Crăciun

January 11, 2010

Cuprins

Programare logică: privire de ansamblu

Planul cursului

Partea I: O introducere în limbajul de programare logică Prolog. Se bazează în mare parte pe [Mellish, Clocksin 1994].

Partea II: Bazele teoretice ale programării logice. Se bazează pe capitolele corespunzătoare din [Ben-Ari, 2001] şi [Nilsson, Maluszynski, 2000].

Partea III: Tehnici avansate în programarea logică/PrologAdvanced topics in logic programming/Prolog. Se bazează pe capitolele corespunzătoare din [Ben-Ari, 2001] şi [Nilsson, Maluszynski, 2000].

Partea I
O introducere în Prolog

1 Preliminarii

1.1 Paradigme de progamare tradiţionale

Programarea maşinii Von Neumann

- Maşina (arhitectura) von Neumann se caracterizează prin:
  - un spaţiu de memorie uniform,
  - o unitate de procesare cu regiştri.
- Un program pentru maşina von Neumann este o secvenţă de instrucţiuni pentru
  - mişcarea datelor între memorie şi regiştri,
  - efectuarea de operaţii aritmetice şi logice între regiştri,
  - control, etc.
- Majoritatea limbajelor de programare (cum ar fi C, C++, Java, etc.) sunt influenţate şi au fost concepute pentru arhitectura von Neumann.
  - Asemenea limbaje de programare se folosesc de caracteristicile aceastei arhitecturi pentru a îmbunătăţi eficienţa programelor.
  - Observaţia de mai sus este una netrivială, şi a dus la o separare a activităţilor în programare ("criza software"):
    - găsirea de soluţii pentru probleme (folosind raţionamentul matefind-ingmatic),
    - implementarea soluţiilor într-un limbaj de programare (activitate ce poate deveni aproape trivială dar laborioasă).

O alternativă pentru maşina von Neumann

- Cum ar fi dacă programarea ar fi parte a rezolvării problemelor ...
- adică să se scrie programe concomitent cu rezolvarea problemelor ...
- obţinând astfel "prototipuri rapide"?
- Programarea logică este derivată dintr-un model abstract diferit de o reorganizare/abstractizare a maşinii von Neumann.
- În programarea logică
  - program = mulţime de axiome,
  - calculul = o demonstraţie constructivă a unei formule obiectiv.

1.2 Programarea logică

Programarea logică: istoric

- Programul lui David Hilbert (începutul secolului 20): propunea formalizarea matematicii folosind o mulţime de axiome finită, completă şi consitentă.
- Teoremele de incompletitudine ale lui Kurt Gödel (1931): orice teorie conţinând arianytmetica nu-şi poate demonstra propria consistenţă.
- Rezultate de indecidabilitate stabilite de Alonzo Church and Alan Turing (independent, 1936): în general nu există procedee mecanice pentru a decide adevărul.
- Alan Robinson (1965): metoda rezoluţiei pentru logica de ordinul întâi (raţionament mecanic în logica de ordinul întâi).
- Robert Kowalski (1971): interpretarea procedurală a clauzelor Horn, calcul mecanic în logică.
- Alan Colmerauer (1972): Prolog (PROgrammation en LOGique).
- David H.D. Warren (anii 1970): implementare eficientă a Prolog.
- 1981, a cincea generaţie de calculatoare, Japonia: un proiect pentru conceperea şi construţia urmatoarei generaţii de calculatoare, cu capabilităţi de inteligenţăartificială (folosind o versiune concurentă de Prolog ca limbaj de programare).

Aplicaţii ale programării logice

- Calcul simbolic:
  - baze de date relaţionale,
  - logică matematică,
  - rezolvare de probleme abstracte,
  - înţelegerea limbajului natural,
  - rezolvare de ecuaţii,
  - automatizarea designului de programe,
  - inteligenţă artificială,
  - analiza structurilor biochimice, etc.
- Programarea în Prolog presupune:
  - în loc de a descrie o secvenţă de paşi pentru rezolvarea unei probleme,
  - se descriu fapte şi relaţii între acestea şi se pun întrebări.

2 Prolog: introducere informală

2.1 Idei de bază

Rezolvarea de probleme cu Prolog

- Prolog se foloseşte la rezolvarea de probleme în care sunt prezente obiecte şi relaţii între acestea.
- Exemple:
  - obiecte: "Ion", "carte", "giuvaier", etc.
  - relaţii: "Cartea este a lui Ion", "Giuvaierul e preţios".
  - reguli: "Doi oameni sunt surori dacă sunt amandoi de sex feminin şi au aceeaşi părinţi.".
- Atenţie!!! Rezolvarea acestui tip de probleme necesită modelarea problemei (cu limitările corespunzătoare).
- Rezolvarea de probleme cu Prolog:
  - se declară fapte despre obiecte şi relaţiile dintre acestea,
  - se definesc reguli despre obiecte şi relaţiile lor.
  - se pun întrebări despre obiecte şi relaţiile dintre acestea.
- Programarea în Prolog este o conversaţie cu interpretorul de Prolog.

2.2 Fapte

- Enunţarea uni fapt în Prolog:

```
place(ion, maria).
```

  - Numele relaţiilor (predicatelor) şi obiectelor încep cu litera mică.
  - Prolog foloseşte mai ales notaţia prefix (dar sunt si excepţii).
  - Un fapt se încheie cu "." (punct).
- Programatorul este cel care construieşte modelul, iar faptele descriu acest model.
- Programatorul trebuie să fie conştient de interpretare:

```
place(ion, maria).
place(maria, ion).
```

  nu descriu acelaşi lucru (decât daca se precizează explicit).
- Un predicat poate avea un număr arbitrar de argumente.
- Notaţie: likes/2 indică un predicat binar.
- Faptele sun parte a bazei de cunoştinţe.

2.3 Interogări (ţinte)

- O interogare în Prolog:

```
?- detine(maria, carte).
```

  Prolog caută în baza de cunoştinţe fapte ce se potrivesc cu interogaţia:

- Răspunsul este YES dacă:
  - predicatul este acelaşi,
  - argumentele sunt aceleaşi.c
- Altfel răspunsul este NO:
  - numai ce se ştie este adevărat ("prezumpţia de lume închisă"),
  - Atenţie: NO nu este acelaşi lucru cu fals (mai degrabă e "nederivabil din baza de cunoştinţe").

Variable

- Similar cu variabilele din logica predicatelor.
- În loc de:

```
?- place(ion, maria).
?- place(ion, mere).
?- place(ion, acadele).
```

  se poate intreba ceva de genul "Ce-i place lui Ion?" (adică se cere tot ce-i place lui Ion).

- Variabilele stau în locul unor obiecte ce urmează să fie determinate de Prolog.
- Variabilele pot fi:
  - instanţiate - există un obiect de care variabila este legată,
  - neinstanţiate - nu se ştie încă la ce obiect este legată variabila.
- În Prolog variabilele încep cu LITERE MARI:

```
?- place(ion, X).
```

Calcul în Prolog: exemplu

- Se consideră următoarele fapte într-o baza de cunoştinţe:

```
...
place(ion, flori).
place(ion, maria).
place(paul, maria).
...
```

- La interogarea:

```
?-place(ion, X).
```

  Prolog va răspunde

```
X = flori.
```

  şi va aţepta instruţiuni.

Calculul răspunsului în Prolog

- Prolog caută în baza de cunoştinţe un fapt ce se potriveşte cu interogarea,
- când se găseşte o potrivire, aceasta este marcată,
- dacă utilizatorul apasă "`Enter`", căutarea se încheie,
- dacă utilizatorul apasă "`;`" şi apoi "`Enter`", Prolog caută o nouă potrivire, începând cu ultimul loc marcat, şi cu variabilele din interogare neinstanţiate.
- În exemplul de mai sus, apăsând încă de două ori "`; Enter`" va determina Prolog să răspundă:

```
X = maria.
no
```

- Când nu se mai găsesc în baza de cunoştinţe fapte ce să se potrivească, Prolog răspunde "no".

Conjuncţii: interogări mai complexe

- Se consideră următoarele fapte:

```
place(maria, mancare).
place(maria, vin).
place(ion, vin).
place(ion, maria).
```

- şi interogarea:

```
?-place(ion, maria), place(maria, ion).
```

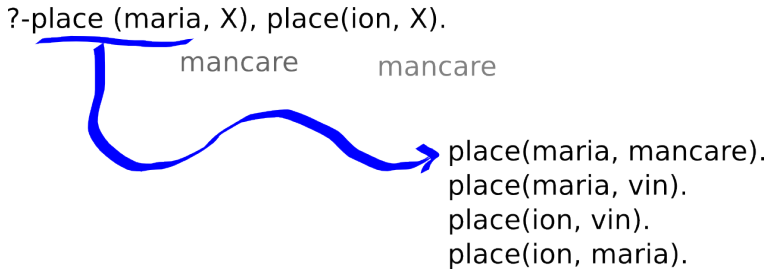


Figura 1: Success pentru prima parte a interogării.

  - interogarea se citeşte "îi place lui ion de maria şi îi place mariei de ion?"
  - Prolog va răspunde "no": caută fiecare fapt din interogare, unul după altul (toate trebuiesc satisfăcute, altfel interogarea va eşua, iar răspunsul va fi "no").

- Pentru interogarea:

```
?-place(maria, X), place(ion, X).
```

  - Prolog: încearcă să satisfacă primul obiectiv (prima parte a interogării), dacă reuşeste plasează un marcaj, apoi încearcă să satisfacă al doilea obiectiv (plasând un marcaj în caz de suces).

Exemplu: conjunctie, backtracking(revenire)

Modul în care Prolog calculează ră spunsul la interogarea de mai sus este reprezentat astfel:

- În Figura 1, prima parte a interogării este saitisfă cută, Prolog încearcă potrivire pentru restul interogării (cu variabila instanţiată).
- Eşecul cauzează revenire, vezi Figura 2.
- Noua alternativă este încercată şi de data aceasta are succes, vezi Figura 3.

2.4 Reguli

- "lui Ion îi plac toţi oamenii" poate fi reprezentată ca:

```
place(ion, alfred).
place(ion, bogdan).
place(ion, corina).
place(ion, david).
...
```

dar această abordare este nepractică!!!

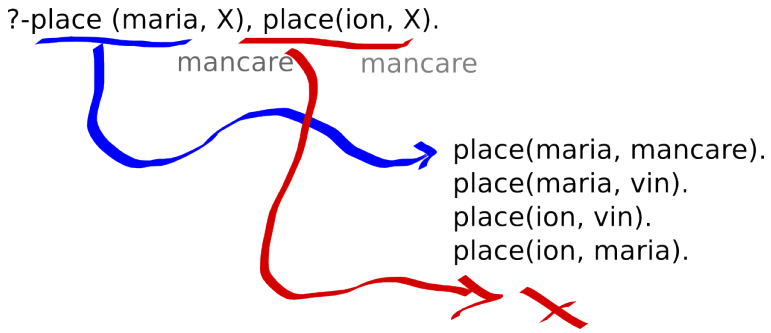


Figura 2: Eşecul pentru partea a doua a interogării cauzează revenire.

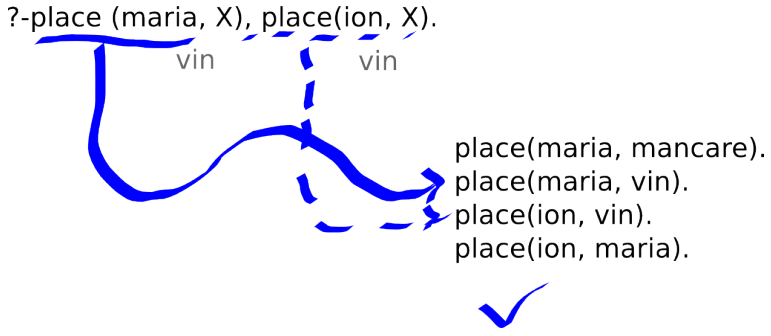


Figura 3: Succes cu instanţierea alternativă.

```
place(ion, X).
```

dar vorbim numai despre oameni!!!

- Aici intervin regulile: "lui ion îi place un obiect, dar numai acela care de fapr este o persoană" este o regulă despre ce îi place lui "ion".
- Regulile sunt un mod de a preciza cum unele fapte depind de alte fapte.

Reguli ca definiţii

- Regulile pot fi folosite pentru a specifica "definiţii".
- Exemplu:

  "X este o pasăre dacă X este un animal şi X are pene."

- Exemplu:

  "X este o soră a lui Y dacă X este femeie şi X şi Y au aceiaşi părinţi".

- Atenţie! Noţiunea de "definiţie" de mai sus nu este identică cu noţiunea de definiţie în logică:
  - "definiţiile" de mai sus permit detectarea predicatelor din capul regulii ,
  - dar mai pot fi alte feluri (de exemplu alte reguli ce "definesc" acelaşi lucru),
  - pentru a avea definiţii veritabile, avem nevoie de "dacă şi numai dacă" în loc de "dacă".
- Regulile sunt afirmaţii generale despre obiecte şi relaţiile dintre acestea şi în general în reguli vor apărea variabile – dar nu întotdeauna.

Reguli în Prolog

- Regulile în Prolog au un cap şi un corp.
- Corpul regulii descrie ţelurile ce trebuiesc satisfăcute pentru a satisface capul.
- Exemplu:

```
place(ion, X) :-
    place(X, vin).
place(ion, X) :-
    place(X,vin), place(X, mancare).
place(ion, X) :-
    femeie(X), place(X, vin).
```

- Atenţie! Scopul unei variabile este regula în care aceasta apare (reguli diferite nu au variabile în comun).

Exemplu (regal)

- Database:

```
male(albert).
male(edward).
female(alice).
female(victoria).
parents(alice, albert, victoria).
parents(edward, albert, victoria).
sister_of(X, Y):-
            female(X),
            parents(X, M, F).
            parents(Y, M, F).
```

- Interogări:

```
?-sister_of(alice, edward).
?-sister_of(alice, X).
```

Exerciţiu (hoţi)

- Se consideră:

```
/*1*/ thief(john).

/*2*/ likes(mary, food).
/*3*/ likes(mary, wine).
/*4*/ likes(john, X):- likes(X, wine).

/*5*/ may_steal(X, Y) :-
                thief(X), likes(X, Y).
```

- Să se explice cum se execută în Prolog interogarea:

```
?-may_steal(john, X).
```

2.5 Sumar

În această primă introducere în Prolog au fost discutate următoarele:

- declararea de fapte despre obiecte,
- punerea de întrebări despre fapte,
- folosirea de variabile, scopul unei variabile,
- conjunţii,
- backtracking – revenire(în exemple).

2.6 De citit, exerciţii

- Toate detaliile despre SWIProlog pot fi găsite la `http://www.swi-prolog.org` .
- Instalaţi SWI-Prolog şi încercaţi exemplele prezentate.
- De citit: Capitolul 1 (incluzând secţiunea de exerciţii) din [Mellish, Clocksin 1994].

3 Sintaxă şi structuri de date

3.1 Termeni

- Programele Prolog sunt construite din termeni (ce se scriu ca şiruri de caractere).
- Următorii sunt termeni:
  - constantele,
  - variabilele,
  - structurile.

3.1.1 Constante

- Constantele sunt termeni simpli.
- Constantele denumesc obiecte sau predicate (în Prolog nu există funcţii).
- Constantele sunt de două tipuri:
  - atomi,
  - numere: întregi, numere raţionale(suport special în biblioteci de sistem), numere reale - reprezentare în virgulă mobilă.

Exemple de atomi

- Sunt termeni:
  - place,
  - a (litere mici),
  - =,
  - −−>,
  - 'Void' (orice între ghilimele unice),
  - ion_popescu (constantele pot conţine _),
- Nu sunt termeni:
  - 314a5 (un termen nu poate începe cu un număr),
  - ion−popescu (un termen nu poate conţine liniuţă),
  - George (un termen nu poate începe cu literă mare),
  - _ceva (un termen nu poate începe cu _).

3.1.2 Variabile

- Variabilele sunt termeni simpli.
- Încep cu literă mare sau cu _,
- Exemple: X, Input, _something, _ (aceasta este o variabilă anonimă).
- Variabilele anonime nu este necesar să aibă interpretări consistente (nu este necesar să fie legate la aceeaşi valoare):

```
?-place(_,ion). % ii place cuiva de ion?
?-place(_,_).   % ii place cuiva de altcineva?
```

3.1.3 Structures

- Structurile sunt termeni compuşi, obiecte ce sunt formate din colecţii de alte obiecte (sau termeni).
- Structurile sunt folosite pentru organizarea datelor.
- O structură este specificată de functorul (numele) ei şi de componentele sale:

```
detine(ion,carte(orbitor, cartarescu)).
carte(orbitor,autor(mircea, cartarescu)).

?-detine(ion,carte(X,autor(Y, caragiale))).
% detine ion o carte (X) de Caragiale (Y,caragiale)?
```

Caractere în Prolog

- Characters:
  - A-Z
  - a-z
  - 0-9
  - + - * / \ ~ ^ < > : .
  - ? @ # $ &
- Caracterele permise sunt cele ASCII (ce pot fi tipărite), cu coduri mai mari de 32.
- Remark: ' ' permite folosirea oricărui caracter.

3.2 Operatori

Operatori aritmetici

- Operatori aritmetici:
  - +,
  - −,
  - *,
  - /,
- $+(x, *(y, z))$ este echivalent cu $x + (y \cdot z)$
- Operatorii nu cauzează evaluare în Prolog.
- Exemplu: 3+4 (structura) nu înseamnă acelaşi lucru cu 7 (termen).
- X **is** 3+4 cauzează evaluare (**is** reprezintă evaluatorul din Prolog).
- Rezultarul evaluării este că X este legat de valoarea 7.

Parcurgerea expresiilor aritmetice

- Pentru parcurgerea expresiilor aritmetice trebuiesc ştiute:
  - Poziţia:
    - infix $x + y$, $x * y$
    - prefix $-x$
    - postfix $x!$
  - Precedenţa: $x + y * z$?
  - Asociativitatea: Ce este $x + y + z$? $x + (y + z)$ sau $(x + y) + z$?
- Fiecare operator are o clasă de precedentţă:
  - 1 - cea mai mare
  - 2 - mai mică
  - ...
  - cea mai mică.
- Exemple:

  $*$ / au precedenţă mai mare decât $+ -$,

  8/2/2 se evaluează la:
  - 8 (8/(2/2)) - asociativ la dreapta?
  - or 2 ((8/2)/2) - asociativ la stânga?
- În Prolog, operatorii aritmetici sunt asociativi la stânga.

3.3 Unificare/Potrivire

Predicatul de unificare '='

- = - predicat infix predefinit în Prolog.

  ?–X = Y.

  Prolog va încerca să potrivească (unifice) X şi Y, şi va răspunde yes dacă reuşeste.

- În general se încearcă unificarea a doi termeni (constante, variabile, structuri):

  ?–T1 = T2.

- Observaţie privind terminologia: unele surse Prolog (de exemplu [Mellish, Clocksin 1994]) folosesc termenul potrivire ("matching"), însă acest termen este rezervat în literatura de specialitate (logica) pentru situaţia în care unul din termeni conţine numai componente fără variabile. Termenul corect pentru ceea ce = face este unificare.

Procedura de unificare

Pentru a decide ?– T1 = T2:

- Dacă T1 şi T2 sunt constante identice, succes (Prolog răspunde yes);
- Dacă T1 şi T2 sunt variabile neinstanţiate, succes (redenumire de variabile);
- Dacă T1 este o variabilă neinstanţiată şi T2 este o constantă sau o structură, succes, şi T1 este instanţiată cu T2;
- Dacă T1 şi T2 sunt variabile instanţiate, atunci decizia se face în funcţie de valorile respective);
- Dacă T1 este o structură: $f(X_1, X_2, ..., X_n)$ şi T2 are aceeaşi nume de functor şi acelaşi număr de argumente: $f(Y_1, Y_2, ..., Y_n)$ atunci se încearcă recursiv unificarea argumentelor ($X_1 = Y_1$, $X_2 = Y_2$, etc.). Dacă toate argumentele se pot unifica, succes, Prolog răspunde yes, altfel răspunsul e no (unificarea eşuează);
- În orice altă situaţie unificarea eşuează.

Verificarea apariţiei

- Se consideră următoarea problemă de unificare:

  ?– X = f(X).

- Prolog răspunde:

```
X = f(**).
```

- Conform procedurii de unificare, rezultatul este $X = f(X) = f(f(X)) = \ldots = f(f(\ldots(f(X)\ldots))) \ldots$ - ar fi generată o buclă infinită.
- Unificarea trebuie să eşueze în asemenea situaţii.
- Pentru evitarea unei astfel de situaţii ar trebui efectuată o verificare a apariţiei: Dacă T1 este o variabilă şi T2 o structură într-o expresie T1 = T2, T1 nu trebuie să apară în T2.
- Verificarea apariţiei este deactivată implicit în cele mai multe implementări Prolog (este foarte scumpă din punct de vedere computaţional) - Prolog face un compromis: preferă viteza corectitudinii.

Un predicat complementar celui de unificare este:

- \= - reuşeşte numai când = eşuează,
- adică dacă T1 \= T2 nu pot fi unificate.

3.4 Aritmetica în Prolog

Predicate predefinite pentru aritmetică

- Prolog are numere (întregi) predefinite.
- Predicate predefinite pentru numere includ:

  X = Y,
  X \= Y,
  X < Y,
  X > Y,
  X =< Y,
  X >= Y,

  cu comportamentul aşteptat.
- Atenţie! Pentru majoritatea predicatelor de mai sus, variabilele care apar trebuie să fie instanţiate (excepţie fac primele două, unde în cazul variabilelor neinstanţiate se face unificare).

Evaluatorul aritmetic is

- Prolog oferă şi operatori aritmetici ("funcţii"), de exemplu:

  +, −, ∗, /, **mod**, rem, abs, max, min, random, round, floor, ceiling, ... , dar acestea nu pot fi folosite direct pentru calcul (2+3 înseamnă 2+3, not 5) - expresiile care conţin operatori nu sunt evaluate implicit.

- Evaluarea explicită se face cu evaluatorul Prolog **is** ce are forma:

  ```
  X is Expr.
  ```

  unde X este o variabilă neinstanţiată, şi Expr este o expresie aritmetică în care toate variabilele trebuie să fie instanţiată - Prolog nu are un rezolvitor de (in)ecuaţii).

Exemplu (aritmetică (1))

```
reigns(rhondri, 844, 878).
reigns(anarawd, 878, 916).
reigns(hywel_dda, 916, 950).
reigns(lago_ap_idwal, 950, 979).
reigns(hywel_ap_ieuaf, 979, 985).
reigns(cadwallon, 985, 986).
reigns(maredudd, 986, 999).

prince(X, Y):-
     reigns(X, A, B),
     Y >= A,
     Y =< B.

?- prince(cadwallon, 986).
yes
?- prince(X, 979).
X = lago_ap_idwal ;
X = hywel_ap_ieuaf
```

Exemplu (aritmetică (2))

```
pop(loc1, 203).
pop(loc2, 548).
pop(loc3, 800).
pop(loc4, 108).

aria(loc1, 3).
aria(loc2, 1).
aria(loc3, 4).
aria(loc4, 3).
```

```
densitate(X, Y):-
        pop(X, P),
        aria(X, A),
        Y is P/A.

?-densitate(loc3, X).
X = 200
yes
```

3.5 Sumar

- Noţiunile discutate în această secţiune:
  - Sintaxa Prolog: termeni (constante, variabile, structuri).
  - Aritmetica în Prolog.
  - Procedura de unificare.
  - Problemă subtilă: verificarea apariţiei.

3.6 De citit şi exerciţii

- Read: Chapter 2 of [Mellish, Clocksin 1994].
- Try out all the examples in these notes, and in the above mentioned Chapter 2 of [Mellish, Clocksin 1994].

4 Structuri de date

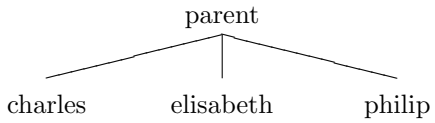
4.1 Structuri de date reprezentate prin arbori

Structuri reprezentate prin arbori

- Se consideră următoarea structură:

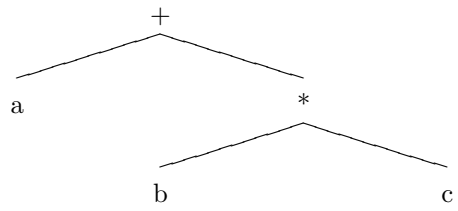
  ```
  parent(charles, elisabeth, philip).
  ```

  aceasta poate fi reprezentată prin:

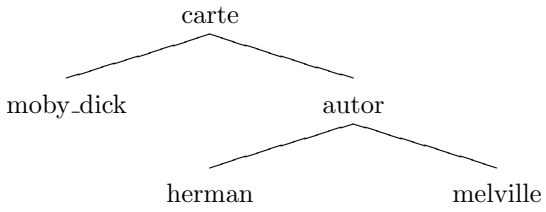


- Alte exemple:

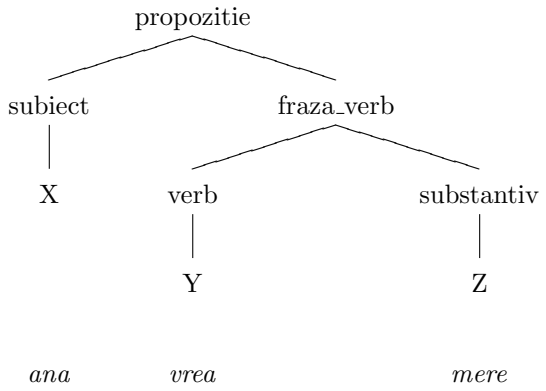
  ```
  a + b*c
  ```



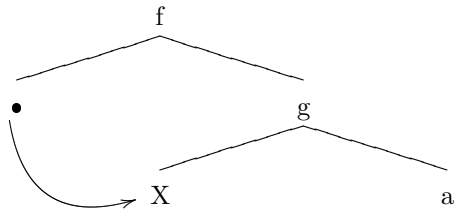

```
carte(moby_dick, autor(herman, melville)).
```




```
propozitie(subiect(X), fraza_verb(verb(Y), substantiv(Z))).
```




```
f(X, g(X, a)).
```



Mai sus, the variabila X este comună mai multor noduri din reprezentare.

4.2 Liste

Introducerea listelor

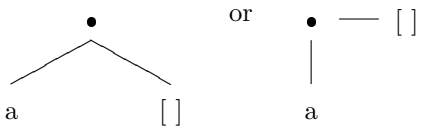
- Listele sunt o structură de date comună în calculul simbolic.
- Listele conţin elemente care sunt ordonate.
- Elementele unei liste sunt termeni (orice tip, incluând alte liste).
- Listele sunt singurul tip de date în LISP.
- Acestea sunt o structură de date în Prolog.
- Practic orice structură de date se poate reprezenta prin liste.

Liste (domeniu inductiv)

- "Caz de bază": [ ] – lista vidă.
- "Caz general" : .(h, t) – lista nevidă, unde:
  - h - capul, poate fi orice termen,
  - t - coada, trebuie să fie o listă.

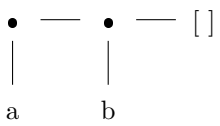
Reprezentări ale listelor

- .(a, [ ]) este reprezentată ca

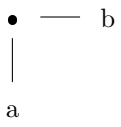


"reprezentare prin arbore" *"reprezentare prin vie"*

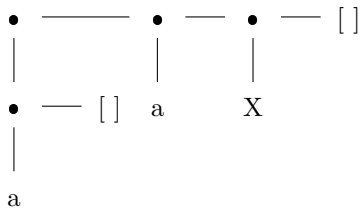
- .(a, .(b, [ ])) este



- .(a, b) *nu este* o listă, dar este o structură legală în Prolog, reprezentată ca



- .(.(a, []), .(a, .(X, [ ]))) este



"Syntactic sugar"

- Notaţia poate fi simplificată, folosing "['' ... ", '' ... "] '' .
- În noua notaţie, listele introduse mai sus sunt:

  [a],

  [a, b],

  [[a], a, X].

Manipularea listelor

- Listele se descompun natural în cap şi coadă.
- Prolog oferă o facilitate care ţine cont de acest fapt: [H | T].
- De exemplu:

```
p([1, 2, 3]).
p([the, cat, sat, [on, the, mat]]).
```

- în Prolog:

```
?-p([H | T]).
H = 1,
T = [2, 3];
H = the
T = [cat, sat, [on, the, mat]];
no
```

- Atenţie! [a | b] nu este o listă, dar este o construcţie legală în Prolog, .(a, b).

Unificarea listelor: exemple

```
[X, Y, Z] = [john, likes, fish]

    X = john
    Y = likes
    Z = fish


[cat] = [X | Y]

    X = cat
    Y = [ ]


[X, Y | Z] = [mary, likes, wine]

  X = mary
  Y = likes
  Z = [wine]


[[the, Y] | Z] = [[X, hare], [is, here]]

    X = the
    Y = hare
    Z = [[is, here]]


[golden | T] = [golden, norfolk]
   T = [norfolk]

[vale, horse] = [horse, X]
     no

[white|Q] = [P|horse]
     P = white
     Q = horse
```

Şiruri de caractere

- În Prolog, şirurile de caractere se scriu între ghilimele.
- Exemplu: `"sir caractere"`.
- Reprezentarea internă a unui şir de caractere este o listă de întregi formată din codul ASCII corespunzător fiecărui caracter.
- ```
  ?- X = "caractere".
  X = [99, 97, 114, 97, 99, 116, 101, 114, 101].
  ```

Sumar

- În această Secţiune am au fost discutate:
  - anatomia unei liste în Prolog `.(h, t)`,
  - reprezentări grafice ale listelor: "arbore", "vie",
  - "syntactic sugar" pentru liste `[...]` ,
  - manipularea listelor: notaţia `[H|T]`,
  - unifyicarea listelor,
  - şiruri de caractere = liste.

De citit

- Capitolul 3, Secţiunea 3.2, din [Mellish, Clocksin 1994].
  ```

5 Recursie

5.1 Introducere

Inducţie/Recursie

- Domeniu inductiv:
  - Un domeniu compus din obiecte compuse într-un mod "abordabil" în următorul sens:
  - există "cele mai simple" obiecte, ce nu pot fi descompuse,
  - obiectele complexe pot fi descompuse într-un număr **finit** de obiecte "mai simple",
  - şi acest proces de decompoziţie poate nu poate continua indefinit: dupa un număr finit de paşi se ating obiectele "cele mai simple".
  - În astfel de domenii se poate folosi inducţia ca metodă de raţionament.
- Recursia este dualul induţiei:
  - se foloseşte recursia pentru descrierea calculului în domenii inductive,
  - procedurile recursive (funcţii, predicate) se autoapelează,
  - însă apelul recursiv trebuie făcut cu argumente "mai simple".
  - În consecinţă, o procedură recursivă va trebui să descrie comportamentul pentru:

  (a) "cele mai simple" obiecte şi/sau situaţiile unde calculul se opreşte - condiţiile limită, şi

  (b) cazul generl - apelul recursiv.

Exemplu: liste ca domeniu inductiv

- cel mai simplu obiect: lista vidă [ ].
- orice altă listă este compusă dintr-un cap şi o coadă (iar coada trebuie să fie o listă): [H|T].

Exemplu: membru într-o listă

- Să se implementeze în Prolog predicatul member/2, astfel încât member(X, Y) este satisfăcut când X este membru al listei Y.

```
% conditia limita
    member(X, [X|_]).
% apelul recursiv
    member(X, [_|Y]):-
                member(X, Y).
```

- În acest caz conditia de limită descrie situaţia când calculul se opreşte (aceasta nu coincide cu momentul atingerii celui mai simplu obiect, [ ]).
- Pentru [ ] predicatul este fals, de aceea nu apare în program.
- De remarcat faptul că apelul recursiv se face pe o listă mai simplă (coada).

Despre folosirea recursivităţii

- Evitaţi apelurile circulare:

```
parinte(X, Y):- copil(Y, X).
copil(X, Y):- parinte(Y, X).
```

- Atenţie la apeluri recursive în partea stângă:

```
persoana(X):-persoana(Y), mama(X, Y).
persoana(adam).
```

  În acest caz, interogarea

```
?-persoana(X).
```

  va intra într-o buclă infinită (fără şanse de backtracking). Prolog încearcă întâi să satisfacă regulaceea ce duce la bucla infinită.

- Ordinea clauzelor:

```
is_list([A|B]):- is_list(B).
is_list([]).
```

  Următoarea interogare va intra în buclă infinită:

```
?-is_list(X)
```

- Ordinea în care se introduc clauzele contează. În general este bine să se plaseze **fapte înaintea regulilor**.

5.2 Mapare recursivă

- Mapare: fiind date 2 structuri similare, să se schimbe prima în a doua, după nişte reguli date
- Exemplu:

  "you are a computer" se mapează în "i am not a computer",

  "do you speak french" se mapează în "i do not speak german".

- Procedura de mapare:

  1. se acceptă o propoziţie,

2. se schimbă “you” în “i”,
3. se schimbă “are” în “am not”,
4. se schimbă “french” în “german”,
5. se schimbă “do” în “no”,
6. restul se lasă neschimbat.

- Programul:

```
change(you, i).
change(are, [am, not]).
change(french, german).
change(do, no).
change(X, X).

alter([],[]).
alter([H|T], [X|Y]):-
        change(H, X),
        alter(T, Y).
```

- Acest program este limitat:
  - va schimba “i do like you” în “i no like i”,
  - noi reguli vor trebui adăugate pentru a adresa astfel de situaţii.

5.3 Comparaţie recursivă

- Comparare după ordonarea din dicţionar (lexicografică) a atomilor: aless/2
  1. aless(book, bookbinder) este adevărat.
  2. aless(elephant, elevator) este adevărat.
  3. aless(lazy, leather) se decide de aless(azy, eather).
  4. aless(same, same) eşuează.
  5. aless(alphabetic, alp) eşuează.
- Se foloseşte predicatul **name**/2 ce dă numele unui atom (ca şir de caractere):

```
?-name(X, [97,108, 112]).
X=alp.
```

- Programul:

```
aless(X, Y):-
        name(X, L), name(Y, M), alessx(L,M).

alessx([], [_|_]).
alessx([X|_], [Y|_]):- X < Y.
alessx([H|X], [H|Y]):- aless(X, Y).
```

5.4 Îmbinarea structurilor

- Vrem să concatenam două liste

```
?-concatenareListe([a,b,c], [3,2,1], [a,b,c,3,2,1]).
true
```

Exemplul de mai sus ilustrează folosirea lui concatenareListe/3 pentru testarea faptului că o listă este rezultatul concatenării altor două.

- Alte moduri de folosireconcatenareListe/3:

- Calculul concatenării:

```
?-concatenareListe([a, b, c], [3, 2, 1], X).
```

- Izolare:

```
?-concatenareListe(X, [2, 1], [a, b, c, 2, 1]).
```

- Descompunere:

```
?-concatenareListe(X, Y, [a, b, c, 3, 2, 1]).
```

```
% conditia limita
concatenareListe([ ], L, L).
% apel recursiv
concatenareListe([X|L1], L2, [X| L3]):-
        concatenareListe(L1, L2, L3).
```

5.5 Acumulatori

Sumar

- Natura recursivă a structurilor (în particular a listelor) se pretează la traversarea lor prin descompunere recursivă.
- Când se atinge condiţia limită, decompoziţia se opreşte şi rezultatul este compus într-o inversare a procesului de descompunere.
- Acest poate face mai eficient prin introducerea unei extra variabile în care se acumulează "rezultaul de pâna acum"
- La momentul atingerii condiţiei de limită această variabilă adiţională va conţine rezultatul final.
- Această variabilă se cheamă acumulator.

Example: lungimea listelor

- Fără acumulator:

```
% lungimea unei liste
 % conditia limita
        lungimeLista([], 0).
 % apel recursiv
        lungimeLista([H|T], N):-
              lungimeLista(T, N1),
              N is N1+1.
```

- Cu acumulator:

```
% lungimea listei cu  acumulator
 % invocarea variantei cu acumulator:
        lungimeLista1(L, N):-
               lungimeListaAc(L, 0, N).
 % conditia de limita pentru acumulator
        lungimeListaAc([], A, A).
 % apel recursiv acumulator
        lungimeListaAc([H|T], A, N):-
              A1 is A + 1,
              lungimeListaAc(T, A1, N).
```

- Intern în Prolog, pentru interogarea ?−lungimeLista1([a, b, c], N):

```
lungimeListaAc([a, b, c], 0, N).
lungimeListaAc([b, c], 1, N).
lungimeListaAc([c], 2, N).
lungimeListaAc([], 3, N)
```

  Variabila care conţine răspunsul este comună de-a lungul apelurilor recursive (neinstanţiată) şi este instanţiată în momentul atingerii condiţiei de limită

Example: inversarea unei liste

- Fără acumulatori:

```
%% reverse
% conditie limita
        reverse1([],[]).
% apel recursiv
        reverse1([X|TX], L):-
             reverse1(TX, NL),
             concatenareListe(NL, [X], L).
```

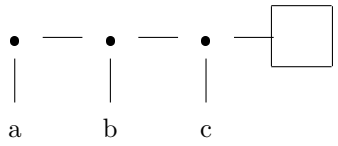
- Cu acumulatori:

```
%% reverse cu acumulatori
% invocarea variantei cu acumulatori
        reverse2(L, R):-
              reverseAcc(L, [], R).
% conditie de limita acumulator
        reverseAcc([], R, R).
% apel recursiv acumulator
        reverseAcc([H|T], A, R):-
              reverseAcc(T, [H|A], R).
```

5.6 Structuri cu diferenţă

Sumar

- Acumulatorii oferă o tehnică pentru a ţine informaţia despre "rezultatul de până acum" la fiecare pas al calculului recursiv.
- Astfel, când calculul se termină (structura a fost consumată), acumulatorul conţine "rezultatul final", care apoi este pasat "variabilei de ieşire".
- Acum vom considera o tehnică unde se foloseşte o variabilă pentru a ţine "rezultatul final" şi o a doua variabilă pentru o "gaură în rezultatul final" în care încă se pot pune obiecte.
- Se consideră [a, b, c | X] - ştim că această structură este o listă până la un punct (până la X). Vom denumi un astfel de termen listă deschisă (listă cu o gaură).



Utilizarea listelor deschise

- Se consideră:

```
?- X = [a, b, c | L], L = [d, e, f, g].
   X = [a, b, c, d, e, f, g],
   L = [d, e, f, g].
```

  - rezultatul este concatenarea începutului lui X (lista dinainte de "gaură") cu L:
  - am umplut gaura din X cu L,
  - de notat că operaţia a fost făcută într-un singur pas!

- Gaura se poate umple şi cu o listă deschisă:

```
?- X = [a, b, c | L], L = [d, e | L1].
   X = [a, b, c, d, e|L1],
   L = [d, e|L1].
```

– gaura a fost umplută parţial.

- Exprimăm cele de mai sus ca predicat Prolog:

```
concat_diff1(LstDesch, Gaura, L):-
        Gaura=L.
```

adică avem o listă deschisă (LstDesch), care are o gaură (Gaura) ce se umple cu o listă (L):

```
?- X = [a, b, c, d | Gaura], concat_diff1(X, Gaura, [d, e]).
   X = [a, b, c, d, d, e],
   Gaura = [d, e].
```

- Atunci când se lucrează cu liste deschise trebuie să avem informaţii (deci să folosim variabile în care să stocăm aceste informaţii) atât despre lista deschisă cât şi despre gaură.
- O listă se poate reprezenta ca diferenţa dintre o listă deschisă şi gaura sa.
- Notaţie: LstDesch−Gaura
  - aici operatorul de diferenţă − nu are nici o interpretare,
  - de fapt se pot folosi şi alţi operatori.
- Să modificăm predicatul pentru a folosi notaţia cu liste cu diferenţă:

```
concat_diff2(LstDesch-Gaura, L):-
        Gaura = L.
```

apelul predicatului:

```
?- X = [a, b, c, d | Gaura]-Gaura, concat_diff2(X,[d, e]).
   X = [a, b, c, d, d, e]-[d, e],
   Gaura = [d, e].
```

- Răspunsul este o listă cu diferenţă, ceea ce poate fi vazut ca un inconvenient – vrem liste.
- O nouă versiune returnează răspunsul într-o listă (răspunsul este obţinut prin umplerea unei liste deschise).

```
concat_diff3(LstDesch-Gaura, L, LstDesch):-
                                    Gaura = L.
```

apelul predicatului:

```
?- X = [a, b, c, d | Gaura]-Gaura, concat_diff3(X,[d, e], Rasp).
   X = [a, b, c, d, d, e]-[d, e],
   Gaura = [d, e],
   Rasp= [a, b, c, d, d, e].
```

- concat_diff3 are
  - o listă cu diferenţe ca prim argument,
  - o listă normală ca al doilea argument,
  - returnează o listă normală.
- O nouă modificare – pentru a fi sitematici – o versiune cu toate argumentele de tip listă cu diferenţe:

```
concat_diff4(OL1-Gaura1, OL2-Gaura2, OL1-Gaura2):-
      Gaura1 = OL2.
```

apelul predicatului:

```
?- X=[a,b,c|Ho]-Ho, concat_diff4(X, [d,e,f|Gaura2]-Gaura2, Rasp).
   X = [a, b, c, d, e, f|Gaura2]-[d, e, f|Gaura2],
   Ho = [d, e, f|Gaura2],
   Rasp= [a, b, c, d, e, f|Gaura2]-Gaura2.
```

sau, dacă insistăm pentru un rezultat ca listă normală, umplem gaura cu lista vidă:

```
?- X=[a,b,c|G]-G, concat_diff6(X, [d,e,f|Gaura2]-Gaura2, Rasp-[]).
   X = [a, b, c, d, e, f]-[d, e, f],
   G = [d, e, f],
   Gaura2 = [],
   Rasp= [a, b, c, d, e, f].
```

- Încă o modificare, pentru a scrie un predicat compact:

```
concatenare_diff(OL1-Gaura1, Gaura1-Gaura2, OL1-Gaura2).
```

apelul predicatului:

```
?- X=[a,b,c|H]-H, concatenare_diff(X, [d,e,f|Gaura2]-Gaura2, Rasp-[]).
   X = [a, b, c, d, e, f]-[d, e, f],
   H = [d, e, f],
   Gaura2 = [],
   Rasp= [a, b, c, d, e, f].
```

Examplu: adăugarea în spatele unei liste:

- Se consideră programul recursiv pentru adăugarea unui element în spatele unei liste:

```
% conditia limita
adauga_sfarsit(El,[],[El]).
% recursie
adauga_sfarsit(El,[Cap|Coada],[Cap|CoadaNoua]):-
                    adauga_sfarsit(El,Coada,CoadaNoua).
```

- Programul este destul de ineficient, mai ales comparat cu operaţia duală, de adăugare a unui element la începutul unei liste (respectiv timp liniar în funţie de lungimea listei versus timp constant – un singur pas).

- Listele cu diferenţă pot ajuta - gaura este la sfârsitul listei:

```
adauga_sfarsit_d(El,LstDesch-Gaura, Rasp):-
      concatenare_dif(LstDesch-Gaura, [El|ElGaura]-ElGaura, Rasp-[]).
```

Probleme cu listele de diferenţă

- Se consideră:

```
?- concatenare_dif([a, b] - [b], [c, d]-[d], L).
   false.
```

  Interogarea de deasupra nu funţionează! (nu sunt găuri de umplut).

- Apar probleme şi la verificarea apariţiei variabilei in termenul de unificare:

```
vida(L-L).

?- vida([a|Y]-Y).
   Y = [a|**].
```

- – în listele cu diferenţe este o funţie parţială. Nu este definită pentru [a, b, c]–[d] :

```
?- concatenare_dif([a, b]-[c], [c]-[d], L).
   L = [a, b]-[d].
```

  Interogarea este satisfăcută, dar rezultatul nu este cel asteptat.

- Se poate adresa această problemă:

```
concatenare_dif_fix(X-Y, Y-Z, X-Z):-
      sufix(Y, X),
      sufix(Z, Y).
```

  problema e că acum timpul de execuţie devine liniar.

5.7 De citit

- Capitolul 3, Capitolul 7, Secţiunile 7.5, 7.6, 7.7 din [Mellish, Clocksin 1994].
- Capitolul 7 din [Nilsson, Maluszynski, 2000].
- Secţiunea 12.2 din [Brna, 1988].
- Să se scrie şi să se ruleze exemplele în Prolog.

6 Revenirea (backtracking) şi predicatul de tăiere (!)

6.1 Comportamentul la revenire (backtracking)

Comportament (aparent) surprinzător

- Sunt situaţii în care Prolog nu se comportă conform aşteptărilor.
- Exemplu:

```
tata(maria, george).
tata(ion, george).
tata(elena, eric).
tata(george, marcel).
```

- Interogarea următoare funţionează conform aşteptărilor:

```
?- tata(X, Y).
   X = maria,
   Y = george ;
   X = ion,
   Y = george ;
   X = elena,
   Y = eric ;
   X = george,
   Y = marcel.
```

- Pentru interogarea:

```
?- tata(_, X).
   X = george ;
   X = george ;
   X = eric ;
   X = marcel.
```

- Odată ce ştim că george este tată, nu e interesant să mai aflăm asta încă odată.
- Se consireră următoarea definiţie recursivă:

```
nat(0).
nat(X):-
    nat(Y),
    X is Y+1.
```

  Folosind explicit comanda pentru revenire, se pot genera toate numerele naturale:

```
?- nat(X).
   X = 0 ;
   X = 1 ;
   X = 2 ;
   X = 3 ;
   X = 4 ;
   X = 5 ;
   X = 6 ;
   ...
```

- Acest comportament e de aşteptat!
- Se consideră:

```
membru(X, [X|_]).
membru(X, [_|Y]):-
          membru(X, Y).
```

- şi interogarea:

```
?- membru(a,[a, b, a, a, a, v, d, e, e, g ,a]).
   true ;
   true ;
   true ;
   true ;
   true ;
   false.
```

- Procesul de revenire (backtracking-ul) confirmă răspunsul de câteva ori. Dar avem nevoie de el o singură dată.

6.2 Predicatul de tăiere(!)

Predicatul de tăiere (!)

- Predicatul de tăiere !/0 indică sistemului Prolog să ignore anumite puncte de revenire.
- Efectul este "tăierea" unor ramuri din spaţiul de căutare.
- În consecinţă,
  - programele vor rula mai repede,
  - programele vor ocupa mai puţină memorie (sunt mai puţine puncte de revenire care trebuiesc memorate).

Exemplu: bibliotecă

- Într-o bibliotecă:
  - un abonat poate avea acces la facilităţi: *de bază*, *generale*.
  - dacă un abonat are o carte împrumutată peste termen, atunci are acces numai le facilităţi de bază.

```
facilitate(Pers, Fac):-
        carte_depasita(Pers, Carte),
        !,
        facilitate_baza(Fac).

facilitate_baza(referinte).
facilitate_baza(informatii).

facilitate_aditionala(imprumut).
facilitate_aditionala(schimb_interbiblioteci).

facilitate_generala(X):-
         facilitate_baza(X).
facilitate_generala(X):-
         facilitate_aditionala(X).

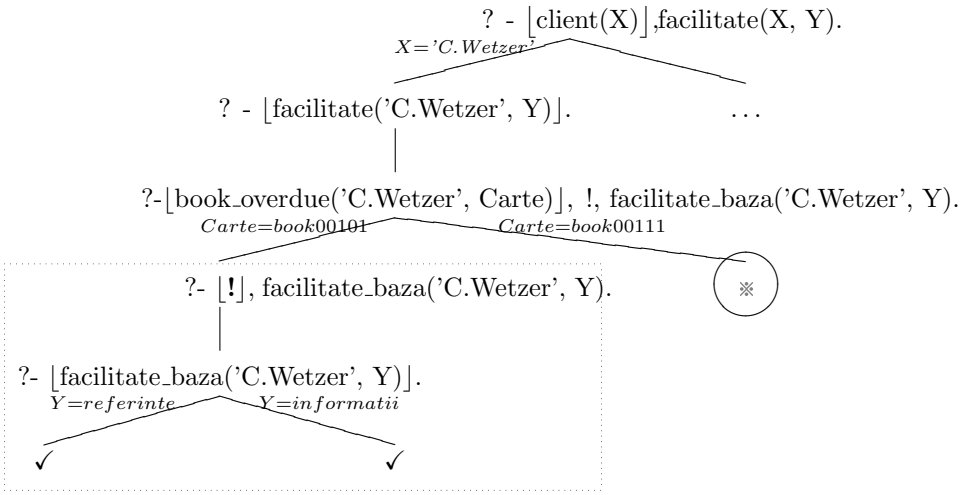


client('C.Wetzer').
client('A.Jones').

carte_depasita('C.Wetzer', book00101).
carte_depasita('C.Wetzer', book00111).
carte_depasita('A.Jones', book010011).
```

```
?- client(X), facilitate(X, Y).
   X = 'C.Wetzer',
   Y = referinte ;
   X = 'C.Wetzer',
   Y = informatii ;
   X = 'A.Jones',
   Y = referinte ;
   X = 'A.Jones',
   Y = informatii.
```

- Interogatia ?−client(X), facilitate (X, Y) este executată de Prolog în modul următor:



- Metafora: poartă cu străjer!
- Efectul predicatului de tăiere:
  - dacă un client are o carte peste termen, accesul în bibliotecă este restricţionat la facilităţi de bază,
  - nu este nevoie să se caute alte cărţi peste termen,
  - nu este nevoie să se considere alte reguli pentru facilităţi.
- ! este satisfăcut întotdeauna.
- Când se întâlneşte operatorul de tăiere:
  - sistemul devine dedicat tuturor alegerilor făcute de când s-a apelat predicatul părinte (în exemplul de mai sus facilitate),
  - orice alternative sunt abandonate,
  - orice încercare de a satisface o interogare între părinte şi predicatul de tăiere va eşua.
  - dacă se cere o altă soluţie, Prolog alege un punct de revenire (backtracking) deasupra părintelui. În exemplul de mai sus punctul de revenire la care se poate reveni este client (x).

6.3 Situaţii comune pentru folosirea predicatului de tăiere (!)

1. **Confirmarea alegerii unei reguli** (se indică sistemului faptul că regula potrivită a fost găsită).
2. **Combinaţia tăiere-eşec** (se indică sistemului că trebuie să eşueze fără să încerce să caute soluţii alternative.

3. **Terminarea unui proces de "generare-şi -testare"** (se indică sistemului să termine generarea de soluţii alternative prin revenire - backtracking).

Confirmarea alegerii unei reguli

- Situaţie:
  - Există mai multe clauze asociate cu acelaşi predicat.
  - Unele dintre clauze sunt portrivite pentru argumente de o anumită formă.
  - În multe cazuri pot si date şabloane pentru argumente (de exemplu pentru liste), dar nu totdeauna.
  - Dacă nu se poate prezenta o listă de şabloane care să acopere toate posibilităţile, trebuie date reguli pentru cazurile particulare şi o regulă ce să "prindă" toate celelalte obiecte.

```
suma_la(1, 1).
        suma_la(N, Res):-
        N1 is N-1,
        suma_la(N1, Res1),
        Res is Res1 + N.
```

  Când se cere revenirea apare o eroare (buclă infinită - de ce?):

```
?- suma_la(5, X).
  X = 15 ;
 ERROR: Out of local stack
```

- Cu predicatul de tăiere (!):

```
csuma_la(1, 1):- !.
csuma_la(N, Res):-
      N1 is N-1,
      csuma_la(N1, Res1),
      Res is Res1 + N.
```

  Sistemul este dedicat condiţiei de limită, nu va mai căuta alternative:

```
?- csuma_la(5, X).
  X = 15.
```

  însă:

```
?- csuma_la(-3, Res).
ERROR: Out of local stack
```

  Adăugând condiţia `N =< 1` la clauza la limită, problema e rezolvată:

```
ssuma_la(N, 1):-
        N =< 1, !.

ssuma_la(N, Res):-
        N1 is N-1,
        ssuma_la(N1, Res1),
        Res is Res1 + N.
```

Predicatul de tăiere ! şi not

- Când ! este folosit pentru confirmarea alegerii unei reguli, poate fi înlocuit cu **not**/1.
- **not**(X) este satisfăcut când X eşuează.
- Folosirea lui **not** este considerată un bun stil de programare:
  - dar programele pot pierde din eficienţă,
  - se face un compromis între lizibilitate şi eficienţă!
- Varianta cu **not**:

```
nsuma_la(1, 1).
nsuma_la(N, Res):-
    not(N =< 1),
    N1 is N-1,
    nsuma_la(N1, Res1),
    Res is Res1 + N.
```

- La folosirea lui **not**, există posibilitatea de dublare a efortului:

```
A:- B, C.
A:- not(B), D.
```

  mai sus, B trebuie satisfăcut de două ori (la revenire).

Combinaţia tăiere-eşec

- **fail**/0 este un predicat predefinit.
- Cînd este întâlnit ca obiectiv eşueazăşi cauzează revenire (backtracking).
- Folosirea lui **fail** după tăiere schimbă comportamentul revenirii.
- Exemplu:
  - ne interesează contribuabilii obişnuiti,
  - străinii nu sunt obişnuiti,
  - dacă un contribuabil nu este străin, se aplică un criteriu general.

```
contr_obisnuit(X):-
    strain(X), !, fail.
contr_obisnuit(X):-
    satisface_criteriu_general(X).
```

Ce-ar fi dacă nu s-ar folosi ! ?

- atunci un străin s-ar putea să satisfacă criteriul general şi să fie considerat contribuabil obişnuit.

- Criteriul general:
  - o persoană căsătorită cu cineva care câstigă mai mult de 3000 nu e contribuabil obişnuit,
  - altfel, este dacă câştigă între 1000 şi 3000.

```
satisface_criteriu_general(X):-
    sot(X, Y),
    venit_total(Y, Inc),
    Inc > 3000,
    !, fail.

satisface_criteriu_general(X):-
    venit_total(X, Inc),
    Inc < 3000,
    Inc > 1000.
```

- Venit total:
  - pensionarii ce câstigă mai putin de 500 nu au venit,
  - altfel, câstigul total este suma salariului plus câştiguri din investiţii.

```
venit_total(X, Y):-
  primeste_pensie(X, P),
  P < 500,
  !, fail.

venit_total(X, Y):-
    salariu(X, Z),
    castig_investitii(X, W),
    Y is Z + W.
```

- **not** poate fi implementat prin combinaţia tăiere-eşec.

```
not(P):- call(P), !, fail.
not(P).
```

- Prolog se va plânge că i se modifică predicatele predefinite.

Înlocuirea tăierii în situaţia tăiere-eşec

- Tăierea se poate înlocui cu **not**.
- De data asta eficienţa nu este afectată.
- Programele trebuiesc rearanjate:

```
contr_obisnuit(X):-
      not(strain(X)),
      not((sot(X, Y),
           venit_total(Y, Inc),
           Inc > 3000)
      ), ...
```

Terminarea unui "generare şi testare"

- X-O.
- Diviziune întreagă:

```
divide(N1, N2, Rezultat):-
    nat(Rezultat),
    Produs1 is Rezultat * N2,
    Produs2 is (Rezultat+1)*N2,
    Produs1 =< N1, Produs2 > N1,
    !.
```

Probleme cu predicatul de tăiere

- Se consideră o variantă cu tăiere a predicatului de concatenare:

```
concattenare([], X, X):-!.
concattenare([A|B], C, [A|D]):-
              concattenare(B, C, D).

?- concattenare([1, 2, 3], [a, b, c], X).
X = [1, 2, 3, a, b, c].

?- concattenare([1, 2, 3], X, [1, 2, 3, a, b, c]).
X = [a, b, c].

?- concattenare(X, Y, [1, 2, 3, a, b, c]).
X = [],
Y = [1, 2, 3, a, b, c].
```

- Comportamentul este cel aşteptat pentru primele două interogări. Însă pentru al treilea, Prolog oferă numai o soluţie (cea care se potriveşte cu situaţia de limită, unde se taie). Restul soluţiilor sunt tăiate.

- Se consider a:

```
numar_parinti(adam, 0):-!.
numar_parinti(eva, 0):-!.
numar_parinti(X, 2).

?- numar_parinti(eva, X).
X = 0.

?- numar_parinti(ion, X).
X = 2.

?- numar_parinti(eva, 2).
true.
```

- Primele două interogări funcţionează conform aşteptărilor.
- A treia interogare dă un răspuns neaşteptat. Acesta se datorează faptului că instanţierea particulară argumentelor nu se potriveşte cu condiţia specială unde s-a folodit tăierea.
- De fapt, în acest caz, şablonul ce distinge între condiţiile speciale şi cazul general este format din ambele argumente (împreună).
- Predicatul de mai sus se poate corecta în două feluri:

```
numar_parinti_1(adam, N):-!, N = 0.
numar_parinti_1(eva, N):-!, N = 0.
numar_parinti_1(X, 2).

numar_parinti_2(adam, 0):-!.
numar_parinti_2(eva, 0):-!.
numar_parinti_2(X, 2):-
        X \= adam, X, \= eva.
```

- Predicatul de tăiere este foarte puterninc şi trebuie folosit cu grijă.
- Avantajele folosirii tăierii pot fi majore, dar la fel sunt şi pericolele.
- Se pot distinge două tipuri de tăieturi:
  - tăieturi verzi: nu se pierd soluţii potenţiale,
  - tăieturi roşii: partea din spaţiul de căutare ce se taie conţine soluţii.
- Tăieturile verzi nu sunt periculoase, în timp ce tăieturile roşii trebuie folosite cu mare atenţie.

6.4 De citit

- Capitolul 4 din [Mellish, Clocksin 1994].
- De asemenea: Capitolul 5, Secţiunea 5.1 din [Nilsson, Maluszynski, 2000].
- Să se încerce exemplele în Prolog.
- Noţiuni de interes:
  - efectul predicatului de tăiere (!), metafora "poartă cu străjer",
  - situaţii comune pentru folosirea predicatului de tăiere:
    1. confirmarea unei reguli,
    2. combinaţia tăiere-eşec,
    3. terminarea unui "generare-şi-testare".
  - eliminarea predicatului de tăiere (se poate face, influenţează eficienţa?)
  - probleme asociate cu tăierea (tăieturi roşii/verzi).

7 Prolog Eficient

7.1 Gândire declarativă şi gândire procedurală

Aspectul procedural al Prolog

- Deşi Prolog este descris ca un limbaj de programare declarativ, el poate fi văzut din punct de vedere procedural:

```
in(X, romania):-
        in(X, timis).
```

  Predicatul de mai sus poate fi considerat:

  - din punct de vedere declarativ: "Z este în România dacă X este în Timiş",
  - din punct de vedere procedural: "Pentru a-l găsi pe X în România, este suficient sa-l găsim în Romania".

- Şi limbajele de programare procedurale pot conţine aspecte declarative:

```
x = y + z;
```

  se poate citi

  - declarativ, ca fiind ecuaţia $x = y + z$,
  - procedural: încarcă y, încarcă z, stochează în z.

Necesitatea înţelegerii aspectelor procedurale/declarative

- Aspectele declarative/procedurale nu sunt "simetrice": există situaţii în care neînţelegerea unuia dintre aspecte poate duce la probleme.
- Pentru programele procedurale: A = (B + C) + D şi A = B + (C +D) par să aibă înţelesuri declarative echivalente, dar:
  - dacă cel mai mare număr ce se poate reprezenta pe o maşină este 1000,
  - atunci pentru B = 501, C = 501, D = -3, cele două reprezentări se evaluează la rezultate diferite!
- Acelaşi lucru se poate întâmpla şi în Prolog. Declarativ, următoarele fapte sunt corecte:

```
strabun(A, C):-
        strabun(A, B),
        strabun(B, C).
```

  Totuşi, ignorarea înţelesului procedural poate duce la bucle infinite (când B şi C sunt ambele necunoscute/neinstanţiate).

7.2 Restrângerea spaţiului de căutare

- Programatorul Prolog este cel care trebuie să construiască un model al problemei şi să-l reprezinte în Prolog.
- Buna cunoaştere a acestui model poate îmbunătăţi semnificativ performanţa programului.

  ?−cal(X), gei(X). va găsi răspunsul mult mai rapid decât?−gray(X), horse(X). într-un model cu 1000 de obiecte gri şi 10 cai.
- Restrângerea spaţiului de căutare poate fi şi mai subtilă:

```
echivalent(L1, L2):-
        permutare(L1, L2).
```

  adică pentru a decide dacă două liste sunt echivalente ca mulţimi, este suficient să se decidă dacă cele două liste sunt permutări una alteia. Dar pentru liste cu N elemente, există $N!$ permutări posibile (de exemplu, pentru 20 de elemente, 2.4×10^{18} permutări posibile).
- În schimb programul:

```
echivalent(L1, L2):-
      sort(L1, L3),
      sort(L2, L3).
```

  adică două liste sunt echivalente ca mulţimi dacă versiunile lor sortate sunt aceleaşi. Sortarea se poate face în $NlogN$ paşi (adică aproximativ 86 de paşi pentru liste de 20 de elemente).

7.3 Lăsaţi unificarea să lucreze

- Când se pot exprima anumite lucruri cu ajutorul unor şabloane, unificarea le poate folosi, programatorul putând evita descrierea unor operaţii.
- De exemplu, consideraţi următoarele variante ale unui predicat ce detectează liste cu 3 elemente:

```
−   are_3_elemente(X):-
                    lungime(X, N),
                    N = 3.

    are_3_elemente([_, _, _]).
```

- De asemenea, consideraţi următorul predicat pentru interschimbarea primelor două elemente ale unei liste.

```
interschimbare_primele_2([A, B| Rest], [B, A| Rest]).
```

  Folosirea unificării salvează utilizatorului parcurgerea listelor.

7.4 Înţelegerea mecanismului de jetoane

- În Prolofg, atomii sunt reprezentaţi într-o tabelă de simboluri (jetoane).
- Atomii dintr-un program sunt înlocuiţi de adresa lor din tabela de simboluri.
- Din această cauză:

```
f('Ce atom lung pare sa fie acesta',
  'Ce atom lung pare sa fie acesta',
  'Ce atom lung pare sa fie acesta').
```

  va ocupa de fapt mai puţină memorie decât `g(a, b, c)`
- Compararea atomilor se face, de aceea, foarte repede.
- De exemplu `a \= b` şi `aaaaaaaaa \= aaaaaaaab` pot fi evaluate în acelaşi timp, fără a fi nevoie de parcurgerea numelui atomilor.

7.5 Recursie finală

Continuări, puncte de revenire

- Se consideră:

```
a:- b, c.
a:- d.
```

- Pentru evaluarea `?- a.`, când se cheamă `b`, Prolog trebuie să salveze în memorie:
  - *continuarea*, adică ce trebuie făcut (adică `c`),după întoarcerea cu succes din `b`
  - *punctul de revenire*, adică unde se poate găsi o alternativă (adică `d`) la întoarcerea cu eşec din `b`.
- Pentru procedurile recursive, continuarea şi punctul de revenire trebuiesc memorate pentru fiecare apel recursiv.
- Aceasta poate duce la consum mare de memorie.

Recursie finală

- Dacă un predicat recursiv nu are continuare şi nici punct de revenire, Prolog poate detecta automat acest fapt, şi nu va aloca memorie pentru acestea.
- Astfel de predicate recursive se numesc *final recursive* (apelul recursiv este ultimul din clauză, şi nu sunt alternative).

- Aceste predicate sunt mult mai eficiente decât variantele non-final recursive.
- Următorul predicat este final recursiv:

```
test1(N):- write(N), nl, NewN is N+1, test1(NewN).
```

  Mai sus **write** scrie (tipăreşte) argumentul la consolă şi este satisfăcut , **nl** mută cursorul la linie nouă (în consolă) şi este satisfăcut. Predicatul va tipăti numerele naturale la consolă până epuizează resursele sistemului (memorie sau atinge limita reprezentabilităţii numerelor)
- Următorul predicat nu este final recursiv (are o continuare):

```
test2(N):- write(N), nl, NewN is N+1, test2(NewN),nl.
```

  Când este rulat, acesta va epuiza memoria relativ rapid.
- Următorul predicat nu este final recursiv (are un punct de revenire):

```
test3(N):- write(N), nl, NewN is N+1, test3(NewN).
test3(N):- N<0.
```

- Următorul predicat este final recursiv (clauza alternativă apare înaintea apelului recursiv, deci nu există un punct de revenire din apelul recursiv):

```
test3a(N):- N<0.
test3a(N):- write(N), nl, NewN is N+1, test3a(NewN).
```

- Următorul predicat nu este final recursiv (are alternative pentru predicatele din clauza recursivă, ce preced apelul recursiv, deci revenirea la o alegere făcută ar putea fi necesară):

```
test4(N):- write(N), nl, m(N, NewN), test4(NewN).

m(N, NewN):- N >= 0, NewN is N + 1.
m(N, NewN):- N < 0, NewN is (-1)*N.
```

transformarea predicatelor recursive în predicate final recursive

- Dacă un predicat nu este final recursiv pentru că are puncte de revenire, atunci poate fi făcut final recursiv prin plasarea unui operator de tăiere înaintea apelului recursiv.
- Următoarele sunt acum final recursive:

```
test5(N):- write(N), nl, NewN is N+1,! ,test5(NewN).
test3(N):- N<0.
```

```
test6(N):- write(N), nl, m(N, NewN), !, test6(NewN).

m(N, NewN):- N >= 0, NewN is N + 1.
m(N, NewN):- N < 0, NewN is (-1)*N.
```

- De notat că recursivitatea finală poate fi indirectă. Următoarele sunt final recursive:

```
test7(N):- write(N), nl,   test7a(N).
test7a(N):- NewN is N+1, test7(NewN).
```

  Avem aici recursie mutuală, dar remarcaţi că test7a se foloseşte doae pentru a redenumi parte a predicatului test7.

Sumar: recursia finală

- În Prolog, recursia finală există când:
  - apelul recursiv este ultimul din clauză,
  - nu există clauze alternative neîncercate,
  - nu există alternative neîncercate pentru nici unul din predicatele ce preced apelul recursiv.

7.6 Folosirea indexării

- Se consideră programul:

```
a(b).
a(c).

d(e).
d(f).
```

  şi interogarea ?-d(f).

- Contrar aşteptărilor, majoritatea implementărilor Prolog nu vor trebui să parcurgă toată baza de cunoştiinţe.
- Prolog foloseşte *indexarea* peste numele functorului şi primul argument.
- Aceşti indici vor fi stocaţi ca o tabelă de asociere (sau ceva similar), pentru a permite accesul rapid.
- În consecinţă, Prolog va găsi direct d(f).
- Folosirea indexării poate face predicatele final recursive, chiar daca acestea nu ar fi in mod normal:

```
test8(0):-write('Still going'), nl, test8(0).
test8(-1).
```

A doua clauză nu este o alternativă din cauza indexării.

- De notat că indexarea funcţionează numai când primul argument al predicatului este instanţiat.

7.7 Cum se documentează programele Prolog

- În documentaţia predicatelor predefinite din Prolog găsim:

  append(?List1, ?List2, ?List3)

  Succeeds when List3 unifies with the concatenation of List1 and List2. The predicate can be used with any instantiation pattern (even three variables).

  –Number **is** +Expr [ISO]

  True if Number has successfully been unified with the number Expr evaluates to. If Expr evaluates to a float that can be represented using an integer (i.e, the value is integer and within the range that can be described by Prolog's integer representation), Expr is unified with the integer value.

- Exemplele de mai sus folosesc o convenţie de notaţie pentru descrierea comportamentului argumentelor - *declaraţii de mod*:

  \+ indică un argument ce trebuie să fie deja instanţiat când predicatul este apelat,

  \- indică un argument care în mod normal nu este instanţiat până când acest predicat nu-l instanţiază,

  ? indică un argument ce poate sau nu să fie instanţiat,

  @ este folosit de unii programatori pentru a indica un argument ce conţine variabile ce nu trebuiesc instanţiate.

  De notat că prezenţa descrierilor de mai sus nu garantează comportamentul indicat, şi nici nu spune nimic despre folosirea argumentelor în alte moduri. Aceste descrieri sunt convenţii, şi folosirea lor corctă este sarcina programatorului.

7.8 De citit şi alte exerciţii

- De citit: [Covington, 1989, Covington et al, 1997].
- Încercaţi exemplele în Prolog.
- Puncte de interes:
  - Înţelesul procedural şi declarativ al programelor Prolog.
  - Restrângerea spaţiului de căutare.
  - Folosirea unificării.

– Înţelegerea jetoanelor.
– Recursie finală: recunoaşterea, eficienţa programelor.
– Folosirea indexării.
– Documentarea programelor.

8 Operaţii de Intrare/Ieşire cu Prolog

- Există două stiluri de operare cu date de intrare/ieşire în Prolog:
  - stilul Edinburg este depăşit, dar încă suportat de unele implementări Prolog. Este relativ simplu de folosit, dar are limitări.
  - intrare/ieşire ISO este standardul suportat de toate implementările Prolog.
- Între cele două stiluri există anumite părţi în comun.

8.1 Stilul Edinburgh pentru intrare/ieşire

Scrierea termenilor

- Predicatul predefinit **write** ia orice termen Prolog şi îl afişează pe ecran.
- Predicatul **nl** mută "cursorul" la o linie nouă.

```
? - write('Hello there'), nl, write('Goodbye').
Hello there
Goodbye
true.
```

- De notat că atomii care sunt citaţi sunt afişati fără ghilimele (simple). Varianta writeq a lui **write** va afişa şi ghilimelele.
- ```
  ?- write(X).
  _G243
  true.

  ?- write("some string").
  [115, 111, 109, 101, 32, 115, 116, 114, 105, 110, 103]
  true.
  ```
- De notat că Prolog afişează reprezentarea internă a termenilor. În particular, reprezentarea internă a variabilelor.

Citirea termenilor

- Predicatul **read** acceptă orice termen Prolog scris la tastatură (în sintaxă Prolog, urmat de punct).

```
?- read(X).
|: hello.
X = hello.

?- read(X).
|: 'hello there'.
```
```

```
X = 'hello there'.

?- read(X).
|: hello there.
ERROR: Stream user_input:0:37 Syntax error: Operator expected

?- read(hello).
|: hello.
true.

?- read(hello).
|: bye.
false.


?- read(X).
|: mother(Y, ada).
X = mother(_G288, ada).
```

- Predicatul **read** este satisfăcut dacă argumentul său poate fi unificat cu un termen dat de utilizator . Exemplele de mai sus ilustrează mai multe situaţii de folosire.

Lucrul cu fişiere

- Predicatul **see** ia ca argument un fişier, efectul său este să deschidă fişierul pentru citire, astfel ca Prolog să ia date de intrare din fişier, nu de la consolă.
- Predicatul **seen** închide toate fişierele deschise, datele de intrare provin din nou de la consolă.

```
? - see('myfile.txt'),
read(X),
read(Y),
read(Z),
seen.
```

- Când un fişier este deschis, Prolog va reţine poziţia "cursorului" în acel fişier.
- Se poate comuta între mai multe fişiere deschise:

```
?- see('aaaa'),
   read(X1),
   see('bbbb'),
   read(X2),
   see('cccc'),
   read(X3),
   seen.
```

- Predicatul **tell** deschide un fişier pentru scriere şi comută ieşirea la acesta.
- Predicatul **told** închide toate fişierele deschise pentru scriere şi comută ieşirea la consolă (ecran).

```
? - tell('myfile.txt'),
    write('Hello there'),
    nl,
    told.
```

- Mai multe fişiere pot fi deschise pentru scriere, şi se poate comuta între ele:

```
?- tell('aaaa'),
   write('first line of aaaa'), nl,
   tell('bbbb'),
   write('first line of bbbb'), nl,
   tell('cccc'),
   write('first line of cccc'), nl,
   told.
```

Intrare/ieşire la nivel de caractere

- Predicatul put scrie un caracter (întreg reprezentând codul ASCII al caracterului).

```
?- put(42).
*
true.
```

- Predicatul get citeşte un caracted de la intrarea implicită (consola).

```
?- get(X).
|    %

X = 37.
```

- În SWI Prolog, put poate funcţiona şi cu caractere neprintabile:

```
?- write(hello), put(8), write(bye).
hellbye
true.
```

Informaţii complete: manualul SWI-Prolog

- Pentru detaliile complete ale sistemului de intrare/ieşire, consultaţi [**?**] (prezent în SWI Prolog prin invocarea ? - help.).

8.2 Intrare/ieşire standard ISO

Streamuri

- Standardul ISO pentru intrare/ieşire din Prolog consideră noţiunea de stream (fişiere sau obiecte similare cu fişierele).
- Sunt disponibile predicate pentru:
  - Deschiderea şi închiderea streamurilor în diferite moduri.
  - Inspectarea stării unui stream, precum şi a altor informaţii.
  - Scrierea/citirea se face în streamuri.
- Există două streamuri permanent deschise: user_input and user_output, corespunzând intrării şi ieşirii la consolă.

Deschiderea streamurilor

- Predicatul open(Filename, Mode, Stream, Options) deschide un stream, unde:

  Filename indică un nume de fişier (depinde de implementare, sistem de operare),

  Mode este unul dintre **read**, **write**, append,

  Stream este un "handle" pentru fişier,

  Options este o listă (posibil vidă) de opţiuni. Acestea includ:

  * type(text) (implicit) sau type(binary),
  * reposition(**true**) sau reposition(false) (implicit) – indică dacă este posibil să se sară înainte şi înapoi la poziţii specificate,
  * alias(**Atom**) – un nume (atom) pentru stram,
  * aţiune pentru încercarea de a citi dincolo de sfârşitul fişierului: eof_action(error) - ridicarea unei condiţii de eroare, oef_action(eof_code) - returnarea unui cod de eroare, eof_action(reset) - cere reexaminarea fişierului (în caz că a fost actualizat, de exemplu de un alt proces concurent).
- Exemplu:

```
test:-
   open('file.txt', read, MyStream, [type(text)]),
   read_term(MyStream, Term, [quoted(true)]),
   close(MyStream), write(Term).
```

Închiderea streamurilor

- Predicatul **close**(Stream, Options) închide Stream cu opţiunile Options.
- **close**(Stream) este versiunea fără opţiuni.
- Optiunile includ force(false) (implicit) şi force(**true**) - chiar dacă există o eroare (de exemplu fişierul se afla pe un disc extern ce a fost deconectat), fişierul se consideră închis, fără să ridice o eroare.

Proprietăţi ale streamurilor

- Predicatul stream_property(Stream, Property) poate fi folosit pentru detectarea proprietăţilor:
  - file_name (...) ,
  - mode(M),
  - alias (A),
  - etc, consultaţi documentaţia [**?**] pentru detalii pentru restul proprietăţilor.
- Exemplu:

```
?- stream_property(user_input, mode(What)).
What = read.
```

Citirea termenilor

- Predicate pentru citirea termenilor:

  read_term(Stream, Term, Options),

  read_term(Term, Options), folosind streamul de intrare curent,

  read(Stream, Term) ca mai sus, fără opţiuni,

  read(Term) ca mai sus, de la streamul de intrare curent.
- Mai multe informaţii despre optiuni se găsesc în documentaţie [**?**].
- Următorul exemplu ilustrează folosirea variable_names, variables, singletons:

```
?- read_term(Term, [variable_names(Vars), singletons(S), variables(List)]).
|    f(X, X, Y, Z).
Term = f(_G359, _G359, _G361, _G362),
Vars = ['X'=_G359, 'Y'=_G361, 'Z'=_G362],
S = ['Y'=_G361, 'Z'=_G362],
List = [_G359, _G361, _G362].
```

Scrierea termenilor

- Predicate pentru scrierea termenilor:

  write_term(Stream, Term, Options),

  write_term(Term, Options),

  write(Stream, Term),

  write(Term),

  comportamentul diferitelor variante fiind analog cu predicatelor de citire.

- Pentru detalii, consultaţi documentaţia.

Alte predicate ISO de intrare/ieşire

- Alte predicate includ cele pentru scriere/citire de caractere/bytes:
  - get_char, peek_char, put_char, put_code, get_code, peek_code, get_byte, peek_byte, put_byte.
- sau:
  - current_input, current_output, set_input, set_output, flush_output, at_the_end_of_stream, **nl**, etc.
- Consultaţi documentaţia pentru detalii.

8.3 De citit şi exerciţii

- De citit: Secţiunile 2.2, 2.3, 2.6, 2.10, 2.12, A.7 din [Covington et al, 1997].
- Încercaţi exemplele din Prolog.
- Subiecte de interes:
  - Intrare/ieşire în stil Edinburgh, limitaţii.
  - Standard ISO pentru intrare/ieşire: streamuri, termeni, altele.

9 Definirea de operatori noi

Operatori în Prolog

- Modul uzual de a scrie predicate în Prolog este de a pune functorul în faţa argumentelor:

```
functor(arg1, arg2, ...)
```

- Totuşi, în unele situaţii, folosirea altor poziţii ar face programele mai usor de înţeles:

```
X este_tatal_lui Y
```

- În Prolog se pot defini operatori noi în acest mod.
- Pentru definirea unui operator trebuie specificate:
  - **poziţia**: dacă operatorul este prefix (implicit), infix sau postfix,
  - **precedenţa**: pentru a decide care operator se aplică prima dată (de exemplu, este 2+3*4 (2+3)*4 sau 2+(3*4)?), precendenţa mai mică leagă mai puternic, precedenţele sunt între 1 şi 1200,
  - **associativitatea**: de exemplu este 8/2/2 (8/2)/2 sau este 8/(2/2)?
  - De notat că predicatele (în sensul logic, adică acele expresii care se evaluează la "adevărat" sau "fals") nu sunt asociative. Consideraţi:

    $3 = 4 = 3$, şi să presupunem ca "=" ar fi asociativ la stânga, atunci $(3 = 4) = 3$ se evalueaă la "fals" $= 3$, ceea ce schimbă tipul argumentelor (din numere în valori de adevăr).

Specificatori pentru sintaxa operatorilor

Specificator	**Înţeles**
fx	Prefix, neasociativ.
fy	Prefix, asociativ la dreapta.
xf	Postfix, neasociativ.
yf	Postfix, asociativ la stânga.
xfx	Infix, neasociativ (ca şi =).
xfy	Infix, asociativ la dreapta (ca şi operatorul "," din interogări complexe în Prolog).
yfx	Infix, asociativ la dreapta (ca şi +).

Operatori Prolog predefiniţi

Priority	Specificator	Operatori
1200	xfx	:–
1200	fx	:– ?–
1100	xfx	;
1050	xfy	–>
1000	xfy	,
900	fy	**not**
700	xfx	= \= == \== @< **is** =<
500	yfx	+ –
400	yfx	* / // **mod**
200	xfy	^
200	fy	–

Exemplu

```
% observati sintaza pentru declararea noului operator:

:- op(100, xfx, este_tatal_lui).

mihai este_tatal_lui carmen.
X este_tatal_lui Y :- barbat(X), parinte(X, Y).

?- X este_tatal_lui carmen.

X = mihai.
```

9.1 De citit şi exerciţii

- De citit: Secţiunea 6.6 din [Covington et al, 1997].
- Încercaţi exemplele din Prolog.
- Subiecte de interes:
  - Definirea operatorilor.

Partea II
Bazele teoretice ale programării logice

10 Logică

10.1 Logica predicatelor (de ordinul întâi)

- În această secţiune vom considera logica predicatelor (de ordinul întâi):
  - sintaxa,
  - semantica,
  - vom ilustra unele dificultăţi legate de evaluarea înţelesului expresiilor din logica de ordinul întâi,
  - vom menţiona unele rezultate ce adresează aceste dificultăţi.

Sintaxa logicii predicatelor de ordinul întâi

- **Vocabularul** limbajului conţine simbolurile din care sunt construite expresiile limbajului:
  - Simboluri rezervate:
    - $($ $)$,
    - $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$,
    - $\forall, \exists$.
  - Mulţimea variabilelor $\mathcal{V}$ (mulţime numărabilă).
  - Mulţimea simbolurilor limbajului $\mathcal{L}$:
    - $\mathcal{F}$ - simboluri funcţionale (cu aritate – număr de argumente – ataşata),
    - $\mathcal{P}$ - simboluri predicative (cu aritate),
    - $\mathcal{C}$ - simboluri pentru constante.
- Exemplu: $\mathcal{L} = \{\{+_{/2}, -_{/1}\}, \{<_{/2}, \geq_{/2}\}, \{0, 1\}\}$. Folosim o notaţie similară cu cea a Prologului pentru a indica numărul de argumente (aritatea).
- Expresiile logicii de ordinul întâi:
  - Termeni:
    - variabilele $v \in \mathcal{V}$ sunt termeni,
    - constantele $c \in \mathcal{C}$ sunt termeni,
    - dacă $f_{/n} \in \mathcal{F}$ şi $t_1, \ldots, t_n$ sunt termeni, atunci şi $f(t_1, \ldots, t_n)$ este un termen.

- Formule:
  - dacă $p_{/n} \in \mathcal{P}$ şi $t_1, \ldots, t_n$ sunt termeni, atunci $p(t_1, \ldots, t_n)$ este o formulă atomică,
  - dacă F, G sunt formule, atunci $\neg F$, $F \wedge G$, $F \vee G$, $F \Rightarrow G$, $F \Leftrightarrow G$ sunt formule compuse,
  - dacă $x \in \mathcal{V}$ şi F este o formulă, atunci $\forall x F$, $\exists x F$ sunt formule cuantificate (formulele cuantificate universal şi, respectiv, existenţial).

Semantica logicii predicatelor de ordinul întâi

- Semantica logicii predicatelor de ordinul întâi descrie înţelesul expresiilor limbajului.
- Un astfel de limbaj este folosit pentru a descrie:
  - un **domeniu** de obiecte,
  - **relaţii** între obiecte (sau proprietăţi ale obiectelor),
  - **procese** sau **funcţii** ce produc obiecte noi din alte obiecte.
- Pentru a determina (calcula) înţelesul unei expresii, trebuie definită în primul rând **interpretarea** simbolurilor:
  - constantele sunt interpretate ca obiecte în domeniul descris de limbaj,
  - simbolurile funţionale sunt interpretate ca procese (funţii) în domeniul descris de limbaj,
  - simbolurile predicative sunt interpretate ca relaţii/proprietăţi între obiecte/ale obiectelor din domeniul descris de limbaj.
- Consideraţi limbajul descris mai sus $\mathcal{L} = \{\{+_{/2}, -_{/1}\}, \{<_{/2}, \geq_{/2}\}, \{0, 1\}\}$. Considerăm două interpretări ale acestui limbaj:
  - $\mathcal{I}_1$ este o interpretare în domeniul numerelor naturale:
    - $\mathcal{I}_1(0) =$ şapte,
    - $\mathcal{I}_1(1) =$ zero,
    - $\mathcal{I}_1(+) =$ înmulţire,
    - $\mathcal{I}_1(-) =$ factorial,
    - $\mathcal{I}_1(<) =$ mai mic decât,
    - $\mathcal{I}_1(\geq) =$ divide.
  - $\mathcal{I}_2$ este o interpretare în domeniul şirurilor de caractere:
    - $\mathcal{I}_2(0) =$ " ",
    - $\mathcal{I}_2(1) =$ "unu",
    - $\mathcal{I}_2(+) =$ concatenare,
    - $\mathcal{I}_2(-) =$ revers,

* $\mathcal{I}_2(<) =$ subşir,
* $\mathcal{I}_1(\geq) =$ versiune sortată.

- Observaţi că o interpretare descrie corespondenţa dintre numele unui concept (constantă, simbol funcţional sau predicativ şi conceptul descris de acest nume.

- Odată definită o interpretare, se poate calcula **valoarea unei expresii** E **sub interpretarea** $\mathcal{I}$**,** $\upsilon_{\mathcal{I}}(E)$ (adică înţelesul expresiei sub interpretare) după cum urmează:

- **Valoarea termenilor sub interpretare**:
  - În general, termenii se vor evalua la obiecte în universul de discurs.
  - Dacă $c \in \mathcal{C}$, $\upsilon_{\mathcal{I}}(c) = \mathcal{I}(c)$.
  - Dacă $x \in \mathcal{V}$, $\upsilon_{\mathcal{I}}(v)$ nu este definit decât dacă variabilei v îi este asignată o valoare. Altfel spus, valoarea unei expresii ce conţine variabile libere nu poate fi determinată decât după ce variabilelor li se asignează valori.
  - Dacă $f(t_1, \ldots, t_n)$ este un termen, atunci
$$\upsilon_{\mathcal{I}}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(\upsilon_{\mathcal{I}}(t_1), \ldots, \upsilon_{\mathcal{I}}(t_n)).$$

- **Valoarea formulelor sub interpretare**:
  - Formulele se vor evalua la **adevărat** sau **fals** (dar nu ambele).
  - Pentru formule atomice,
$$\upsilon_{\mathcal{I}}(p(t_1, \ldots, t_n)) = \mathcal{I}(p)(\upsilon_{\mathcal{I}}(t_1), \ldots, \upsilon_{\mathcal{I}}(t_n)).$$
  - Pentru formule compuse:
    * $\upsilon_{\mathcal{I}}(\neg F) =$ **adevărat** dacă şi numai dacă $\upsilon_{\mathcal{I}}(F) =$ **fals**.
    * $\upsilon_{\mathcal{I}}(F \wedge G) =$ **adevărat** dacă şi numai dacă $\upsilon_{\mathcal{I}}(F) =$ **adevărat** and $\upsilon_{\mathcal{I}}(G) =$ **adevărat**.
    *
    * $\upsilon_{\mathcal{I}}(F \vee G) =$ **adevărat** dacă şi numai dacă $\upsilon_{\mathcal{I}}(F) =$ **adevărat** sau $\upsilon_{\mathcal{I}}(G) =$ **adevărat** (cel puţin una este adevărată).
    * $\upsilon_{\mathcal{I}}(F \Rightarrow G) =$ **fals** dacă şi numai dacă $\upsilon_{\mathcal{I}}(F) =$ **adevărat** and $\upsilon_{\mathcal{I}}(G) =$ **fals**.
    * $\upsilon_{\mathcal{I}}(F \Leftrightarrow G) =$ **adevărat** dacă şi numai dacă $\upsilon_{\mathcal{I}}(F) = \upsilon_{\mathcal{I}}(G)$.
  - Pentru formule cuantificate:
    * $\upsilon_{\mathcal{I}}(\forall x F) =$ **adevărat** dacă şi numai dacă pentru toate valorile lui x din domeniu, $\upsilon_{\mathcal{I}}(F) =$ **adevărat**.
    * $\upsilon_{\mathcal{I}}(\exists x F) =$ **adevărat** dacă şi numai dacă pentru unele valori din x domaeniu, $\upsilon_{\mathcal{I}}(F) =$ **adevărat**.

- De exemplu, consideraţi $\mathcal{I}_1$ definită mai sus:

$$\begin{aligned}&\upsilon_{\mathcal{I}_1}(-(0+1))) = \\ &\qquad \mathcal{I}_1(-)(\upsilon_{\mathcal{I}_1}(0+1)) = \\ &\qquad \text{factorial}(\mathcal{I}_1(+)(\upsilon_{\mathcal{I}_1}(0), \upsilon_{\mathcal{I}_1}(1))) = \\ &\qquad \text{factorial}(\text{înmulţire}(\text{şapte}, \text{zero})) = \\ &\qquad \text{factorial}(\text{zero}) = \\ &\qquad \text{unu}.\end{aligned}$$

Validitate, satisfiabilitate, nesatisfiabilitate

- Înţelesul formulelor este de interes, în particular:
  - Dacă o formulă este **validă**, adică adevărată sub orice interpretare.
  - Dacă o formulă este **satisfiabilă**, adică există interpretări sub care formula este adevarată.
  - Dacă formula este **nesatisfiabilă**, adică i.e. the formula is false under all possible interpretations.
  - Dacă două formule sunt **logic ehivalente**, adică formulele au acelaşi înţeles sub aceleaşi interpretari (notăm $F_1 \equiv F_2$).
  - Dacă o formulă este **consecinţă logică** a unei mulţimi de formule , adică dacă formula este adevărată sub orice interpretare pentru care toate celelalte formule sunt adevărate în acelaşi timp (notăm $F_1, \ldots, F_n \vDash G$).
- Folosirea acestor noţiuni este foarte dificilă în practică: numărul posibil de interpretări pentru un limbaj este potenţial infinit. A verifica valoarea unei expresii sub toate interpretările posibile devine astfel nepractic.
- Dacă o formulă (sau o mulţime de formule) este adevarată (sunt adevărate) sun o interpretare într-un domeniu, atunci domeniul respectiv se cheamă un **model** al formulei (formulelor) respective.

10.2 Teorema lui Herbrand

Universul Herbrand

- Din fericire, dificultatea reprezentată de numărul imens (infinit) de interpretări posibile poate fi depăşită.
- Vom defini un domeniu şi o interpretare care "capturează" proprietăţile tuturot domeniilor şi interpretărilor posibile.
- Verificarea satisfiabilităţii (şi validităţii) unei formule (sau mulţimi de formule) poate fi făcută verificând doar valoarea sub această interpretare.
- Fie $\mathcal{L}$ un limbaj conţinând simbolurile constante $\mathcal{C}$, simbolurile funcţionale $\mathcal{F}$ şi simbolurile predicative $\mathcal{P}$. Fie F o formulă peste $\mathcal{L}$.

- **Universul Herbrand** $\mathcal{H}$ ce corespunde limbajului $\mathcal{L}$ (sau formulei F) se defineşte în modul următor:
  - Dacă $c \in \mathcal{C}$ atunci $c \in \mathcal{H}$.
  - Dacă $t_1, \ldots, t_n$ şi $f \in \mathcal{F}$ atunci $f(t_1, \ldots, t_n) \in \mathcal{H}$.
  - Dacă $\mathcal{C} = \varnothing$ atunci se adaugă o constantă arbitrară la universul Herbrand $\mathcal{H}$.
- Universul Herbrand este mulţimea de termeni fără variabile care poate fi formată din constantele şi simbolurile funcţionale ale limbajului.
- **Baza Herbrand** $\mathcal{B}$ a limbajului $\mathcal{L}$ (sau formulei F) este mulţimea de atomi fără variabile ce se poate forma din simbolurile predicative din $\mathcal{P}$ aplicate termenilor din $\mathcal{H}$.
- O **interpretare Herbrand** $\mathcal{I}_\mathcal{H}$ pentru limbajul $\mathcal{L}$ este o interpretare a cărei domeniu este universul Herbrand $\mathcal{H}$ ale cărui simboluri se intepretează la ele însele:
  - Dacă $c \in \mathcal{C}$, $\mathcal{I}_\mathcal{H}(c) = c$.
  - Dacă $f \in \mathcal{F}$, $\mathcal{I}_\mathcal{H}(f) = f$.
  - Dacă $f \in \mathcal{P}$, $\mathcal{I}_\mathcal{H}(p) = p$.
- Un **model Herbrand** pentru o formulă (mulţile de formule) F este o interpretare Herbrand ce satisface F. Un model Herbrand poate fi identificat cu o submulţime a bazei Herbrand, anume aceea pentru care

$$v_{\mathcal{I}_\mathcal{H}}(p(t_1, \ldots, t_n)) = \textbf{adevărat}.$$

Teorema Herbrand

Au fost demonstrate câteva rezultate remarcabile.

Teorema 1. *Fie F o formulă. F are un model dacă şi numai dacă are un model Herbrand.*

Teorema 2 (Teorema Herbrand, formă semantică)**.** *Fie F o formulă (sau mulţime de formule). F este nesarisfiabilă dacă şi numai dacă o formulă construită dintr-o mulţime **finită** de instanţe fără variabile ale unor subformule din F este nesatisfiabilă.*

Teorema 3 (Teorema Herbrand, formă sintactică)**.** *O formulă F poate fi demonstrată dacă şi numai dacă o formulă construită dintr-o mulţime finită de instanţe fără variabile ale lui F poate fi demonstrată în logica propoziţională.*

- Teorema Herbrand (forma semantică) ne spune cărobleme nesatisfiabilităţii în logica predicatelor se poate reduce la problema nesatisfiabilităţii în logica propoziţiilor.

- Pentru logica propoziţiilor, **metoda rezoluţiei** decide problema satisfiabilităţii. Vezi [Craciun, 2009] pentru detalii.
- Pentru a folosi metoda rezoluţiei în logica propoziţiilor, formulele propoziţionale sunt transformate în **formă normal conjunctivă** (FNC).
- Pentru a folosi teorema Herbrand în combinaţie cu rezoluţia în logica propoziţiilor, vom avea nevoie de o transformare similară pentru logica predicatelor.

10.3 Formă clauzală a formulelor

- Un **literal** în logica predicatelor este o formulă atomic a sau negaţia unei formule atomice.
- O formulă din logica predicatelor este în **formă normal conjunctivă (FNC)** dacăşi numai dacă este o conjuncţie de disjuncţii de literali.
- O formulă din logica predicatelor este în **formă prenex normal conjunctivă (FPNC)** dacă şi numai dacă este de forma

  $$Q_1 x_1 \dots Q_n x_n M,$$

  unde Q_i este un cuantificator (unul din $\forall$, $\exists$), pentru $i = 1 \dots n$, şi M este o formulă fără cuantificatoriîn FNC. $Q_1 x_1 \dots Q_n x_n$ se cheamă **prefixul** şi M se cheamă **matricea**.
- O formulă se cheamă **închisă** dacă şi numai dacă nu are variabile libere (toate variabilele sunt legate de un cunatificator).
- O formulă închisă este în **formă clauzală** dacă şi numai dacă este în FPNC şi prefixul său conţine numai cuantificatori universali.
- O **clauză** este o disjunţie de literali.
- Examplu: Formula următoare este în formă clauzală:

  $$\forall x \forall y \forall z \Big(\big(p(f(x)) \vee \neg q(y,z)\big) \wedge \big(\neg p(x) \vee q(y, f(z)) \vee r(x,y)\big) \wedge \big(q(x, f(z)) \vee \neg r(f(y), f(z))\big) \Big).$$

- Notaţie: Din moment ce matricea unei formule în formă clauzală conţine doar cuantificatori universali, aceştia pot fi omişi. Forma clauzală poate fi reprezentată în modul următor (cauze ca **mulţimi de literali**, formule ca **mulţimi de clauze**):

  $$\Big\{ \big\{p(f(x)), \neg q(y,z)\big\}, \big\{\neg p(x), q(y, f(z)), r(x,y)\big\}, \big\{q(x, f(z)), \neg r(f(y), f(z))\big\} \Big\}.$$

- Notaţie: Fie F, G formule. Notăm $F \approx G$ dacă F şi G sunt echisatisfiabile (adică F este satisfiabilă dacă şi numai dacă G este satisfiabilă).

Teorema 4 (Skolem)**.** *Fie F o formulă închisă. Atunci există o formulă F' în formă clauzală astfel încât $F \approx F'$.*

- Teorema lui Skolem poate fi folosită pentru a decide dacă o formulă este nesatisfiabilă, în cazul că există o metodă pentru a determina nesatisfiabilitatea unei formule în formă clauzală. Acesta este subiectul Capitolului următor.

Algoritmul de Skolemizare

INTRARE: formula închisă F.

IEŞIRE: formula F' în formă clauzală astfel încât $F \approx F'$.

Exemplu:

$$\forall x(p(x) \Rightarrow q(x)) \Rightarrow (\forall x p(x) \Rightarrow \forall x q(x))$$

1: Se redenumesc variabilele astfel încât nici o variabilă nu apare legată de doi cuantificatori diferiţi.

$$\forall x(p(x) \Rightarrow q(x)) \Rightarrow (\forall y p(y) \Rightarrow \forall z q(z))$$

2: Se elimină toate echivalenţele şi implicaţiile $(\Leftrightarrow, \Rightarrow)$, folosind echivalenţele $(F \Leftrightarrow G) \equiv ((F \Rightarrow G) \wedge (G \Rightarrow F))$ şi $(F \Rightarrow G) \equiv ((\neg F) \vee G)$.

$$\neg\forall x(\neg p(x) \vee q(x)) \vee (\neg\forall y p(y) \vee \forall z q(z))$$

3: Se împing negaţiile în interiorul parantezelor, până negaţiile se aplică numai formulelor atomice. Se folosesc echivalenţele $\neg(\neg F) \equiv F$, $\neg(F \wedge G) \equiv (\neg F \vee \neg G)$, $\neg(F \vee G) \equiv (\neg F \wedge \neg G)$, $\neg(\forall x F[x]) \equiv \exists x \neg F[x]$, $\neg(\exists x F[x]) \equiv \forall x \neg F[x]$.

$$\exists x(p(x) \wedge \neg q(x)) \vee \exists y \neg p(y) \vee \forall z q(z)$$

4: Se extrag cuantificatorii din matrice. Din moment ce variabilele au fost redenumite, se pot folosi următoarele echivalenţe: $AopQxB[x] \equiv Qx(AopB[X])$ and $QxB[x]opA \equiv Qx(B[X]opA)$ unde Q este unul din $\forall, \exists$ şi op este unul din $\wedge, \vee$.

$$\exists x \exists y \forall z((p(x) \wedge \neg q(x)) \vee \neg p(y) \vee q(z))$$

5: Se folosesc $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$, $(P \wedge Q) \vee R) \equiv (P \vee R) \wedge (Q \vee R)$ pentru a transforma matricea în FNC.

$$\exists x \exists y \forall z((p(x) \vee \neg p(y) \vee q(z)) \wedge (\neg q(x) \vee \neg p(y) \vee q(z)))$$

6: **Skolemizare**

- Dacă prefixul este de forma $\forall y_1 \ldots \forall y_n \exists x$, se ia f să fir un noi dimbol funţional cu aritate n. Se elimină $\exists x$ din prefix şi se înlocuiesc toate apariţiile lui x în matrice cu $f(y_1, \ldots, y_n)$. Funţia f se cheamă **funţie Skolem**.
- Dacă nu există cuantificatori universali înainte de $\exists x$ în prefix, se ia a o constantă nouă. Se elimină $\exists x$ din prefix şi se înlocuieşte fiecare apariţie a lui x în matrice cu$a$. Constanta a se cheamă **constantă Skolem**.

$$\forall z((p(a) \vee \neg p(b) \vee q(z)) \wedge (\neg q(a) \vee \neg p(b) \vee q(z)))$$

Observaţi că paşii 1-5 păstrează echivalenţa logică. Se poate ară ta relativ uşor că pasul 6 păstrează satisfiabilitatea. Pentru detalii, consultaţi [Ben-Ari, 2001].

10.4 De citit şi exerciţii

- Citiţi: Capitolul 7, secţiunile 7.1-7.4 din [Ben-Ari, 2001].
- De interes (nu se cer demonstraţii):
  - Limbajul logicii predicatelor: sintaxă, semantică (interpretare, model).
  - Univers Herbrand, bază Herbrand, interpretare Herbrand.
  - Teorema Herbrand, semnificaţia teoremei Herobrand.
  - Forma clauzală a formulelor de ordinul întâi, Skolemizare (funcţii, constante Skolem), algoritmul de transformare în formă clauzală.

11 Rezoluţie

11.1 Rezoluţie fără variabile

- Teorema Herbrand reduce problema nesatisfiabilităţii unei formule (mulţimi de formule) la problema nesatisfiabilităţii unei mulţimi finite de formule fără variabile.
- Din punct de vedere practic, fiind dată o mulţime finită de formule fără variabile, se pot redenumi atomii distincţi cu variabile propoziţionale distincte, reducând astfel problema la nesatisfiabilitatea în **logica propoziţiilor**, astfel putând folosi **rezoluţia pentru propoziţii** pentru rezolvarea problemei. Vezi [Craciun, 2009] pentru detalii privind rezoluţia pentru propoziţii.
- Din păcate această abordare nu este practică: nu există nici o metodă prin care să se poată găsi mulţimea finită de formule fără variabile – mulţimea posibilelor instanţieri este atât nemărginită cât şi nestructurată.

11.2 Substituţii

- O **substituţie** de termeni pentru variabile este o mulţime:

  $$\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$$

  unde, pentru $i = 1 \ldots n$, x_i sunt variabile distincte, t_i sunt termeni astfel încât x_i şi t_i sunt distincte. Substituţiile se vor nota cu litere mici greceţi (λ, θ, δ,σ). Substituţia **vidă** se notează cu ϵ.
- O **expresie** este un termen, formulă (in particular literal, clauză, mulţime de clauze).
- Fie E o expresie, $\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$. O **instanţă a lui** E (sau rezultatul aplicării substituţiei θ la E), $E\theta$ este expresia obţinută prin înlocuirea **simultană** a tuturor apariţiilor variabilelor x_i în E cu t_i.
- Exemplu: Fie $E = p(x) \vee q(f(y))$, $\theta = \{x \leftarrow y, y \leftarrow f(a)\}$. Atunci

  $$E\theta = p(y) \vee q(f(f(a))).$$

- Fie $\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ şi $\sigma = \{y_1 \leftarrow s_1, \ldots, y_k \leftarrow s_k\}$ substituţii. Fie X, Y mulţimile de variabile din θ, respectiv σ. **Compoziţia lui θ şi σ, $\theta\sigma$** este substituţia:

  $$\theta\sigma = \{x_i \leftarrow t_i\sigma | x_i \in X, x_i \neq t_i\sigma\} \cup \{y_j \leftarrow s_j | y_j \in Y, y_j \notin X\},$$

  în alte cuvinte, se aplică substituţia σ termenilor t_i (cu condiţia ca rezultatul să nu devină$x_i \leftarrow x_i$), apoi se concatenează substituţiile din σ ale căror variabile nu apar deja în θ.

- Exemplu: fie

$$\theta = \{x \leftarrow f(y), y \leftarrow f(a), z \leftarrow u\},$$
$$\sigma = \{y \leftarrow g(a), u \leftarrow z, v \leftarrow f(f(a))\},$$

  atunci:

$$\theta\sigma = \{x \leftarrow f(g(a)), y \leftarrow f(a), u \leftarrow z, v \leftarrow f(f(a))\}.$$

- Fie E o expresie şi θ, σ substituţii. Atunci $E(\theta\sigma) = (E\theta)\sigma$.
- Fie θ, σ, λ substituţii. Atunci $\theta(\sigma\lambda) = (\theta\sigma)\lambda$.

11.3 Unificare

Unificatori

- Consideraţi doi literali cu variabile: $p(f(x), g(y))$ şi $p(f(f(a)), g(z))$:
  - substituţia

$$\{x \leftarrow f(a), y \leftarrow f(g(a)), z \leftarrow f(g(a))\}$$

    aplicată celor doi literali îi va face identici (îi va "unifica"),
  - acelaşi efect se obţine aplicând substituţiile

$$\{x \leftarrow f(a), y \leftarrow a, z \leftarrow a\},$$

$$\{x \leftarrow f(a), z \leftarrow y\}.$$

- Fiind dată o mulţime de literali, un **unificator** este o substituţie care face literalii din mulţime identici. Un **cel mai general unificator (cmgu)** este un unificator μ astfel încât orice alt unificator θ poate fi obţinut din μ prin aplicarea unei noi substituţii λ astfel încât $\theta = \mu\lambda$.
- Observaţi că nu toţi literalii sunt unificabili: dacă simbolurile lor predicative sunt diferite, literalii nu pot fi unificaţi. De asemenea, consideraţi cazul $p(x)$ şi $p(f(x))$. Din moment ce substituţia variabilei x trebuie făcută simultan, termenii x şi $f(x)$ nu pot fi făcuţi identici, şi unificarea va eşua.

Ecuaţii de termeni

- Observaţi că problema unificării unor literali, de exemplu $p(f(x), g(y))$ şi $p(f(f(a)), g(z))$, poate fi exprimată ca o mulţime de ecuaţii de termeni:

$$f(x) = f(f(a))$$
$$g(y) = g(z).$$

- O mulţime de ecuaţii de termeni este în **formă rezolvată** dacă şi numai dacă:

– toate ecuaţiile sunt de forma $x_i = t_i$, unde x_i sunt variabile,

– fiecare variabilă x_i ce apare în partea stângă a unei ecuaţii nu apare în alt loc în mulţimea de ecuaţii.

O mulţime de ecuaţii în formă rezolvată defineşte o substituţie în mod natural: se transformă fiecare ecuaţie $x_i = t_i$ într-un element al subastituţiei, $x_i \leftarrow t_i$.

Algoritm de unificare

INTRARE: O mulţime de ecuaţii de termeni.

OUTPUT: O mulţime de ecuaţii de termeni în formă rezolvare, sau "neunificabil".

Se efectuează următoarele transformări peste mulţimea de ecuaţii, atât timp cât acestea sunt posibile:

1. Se transformă $t = x$ în $x = t$, unde x este o variabilă iar t nu este.
2. Se şterge ecuaţia $x = x$, pentru toate variabilele x.
3. Fie $t' = t''$ o ecuaţie, unde t', t'' nu sunt variabile. Dacă cele mai semnificative simboluri (funcţionale) ale lui t' şi t'' nu sunt identice algoritmul se termină cu răspunsul "neunificabil". Altfel, dacă t' este de forma $f(t'_1, \ldots, t'_k)$ şi t'' este de forma $f(t''_1, \ldots, t''_k)$, se înlocuieşte ecuaţia $f(t'_1, \ldots, t'_k) = f(t''_1, \ldots, t''_k)$ cu k equaţii
$$t'_1 = t''_1, \ldots, t'_k = t''_k.$$
4. Fie $x = t$ o ecuaţie de termeniastfel încât x apare în alt loc în mulţimea de ecuaţii. Dacă x apare în t (verificarea apariţiei!), algoritmul se termină cu răspunsul "neunificabil". Altfel, se transformă molţimea de ecuaţii prin înlocuirea fiecarei apariţii a lui x în alte ecuaţii cu t.

Exemplu 1 (Unificare, din [Ben-Ari, 2001])**.**

Se consideră ecuaţiile:

$$g(y) = x$$
$$f(x, h(x), y) = f(g(z), w, z).$$

- *Se aplică regula 1 primei ecuaţii şi regula 3 celei de-a doua:*

$$x = g(y)$$
$$x = g(z)$$
$$h(x) = w$$
$$y = z.$$

- *Se aplică regula 4 celei de-a doua ecuaţii pentru a înlocui apariţiile lui x:*

$$\begin{gathered} g(z) = g(y) \\ x = g(z) \\ h(g(z)) = w \\ y = z. \end{gathered}$$

- *Se aplică regula 3 primei ecuaţii*

$$\begin{gathered} z = y \\ x = g(z) \\ h(g(z)) = w \\ y = z. \end{gathered}$$

- *Se aplică regula 4 ultimei ecuaţii pentru a înlocui y cu z în prima ecuaţie, apoi se şterge rezultatul $z = z$ folosind regula 2:*

$$\begin{gathered} x = g(z) \\ h(g(z)) = w \\ y = z. \end{gathered}$$

- *Se transformă a doua ecuaţie cu regula 1:*

$$\begin{gathered} x = g(z) \\ w = h(g(z)) \\ y = z. \end{gathered}$$

- *Algorimul se termină cu succes. Substituţia ce rezultă*

$$\{x \leftarrow g(z), w \leftarrow h(g(z)), y \leftarrow z\}$$

este cmgu pentru mulţimea iniţială de ecuaţii.

Teorema 5 (Corectitudinea algoritmului de unificare)**.** *Algoritmul de unificare se termină. Dacă se temrină cu răspunsul "neunificabil", nu există unificator pentru mulţimea de ecuaţii.Dacă algoritmul se termină cu succes, mulţimea de ecuazţii în formă rezolvată rezultată defineşte un cmgu*

$$\mu = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$$

pentru mulţimea de ecuaţii.

Proof. Vezi[Ben-Ari, 2001], pp. 158. □

11.4 Rezoluţia

- Rezoluţia fără variabile nu poate fi aplicată în practică.

- Din fericire, o versiune practică a rezoluţiei este posibilă, cu ajutorul unificării.
- Noţiunile de literal, clauză, mulţime de clauze au fost introduse în Subsecţiunea 10.3.
- **Notaţie.** Fie L un literal. Notăm L^c literalul complementar (L şi L^c sunt unul negaţia celuilalt).

Definition 1 (Pas general de rezoluţie)**.** Fie C_1, C_2 clauze *fără variabile în comun*. Fie $L_1 \in C_1$ şi $L_2 \in C_2$ literali în aceste clauze astfel încât L_1 şi ${L_2}^c$ pot fi unificaţi de un cmgu σ. Atunci C_1 şi C_2 se cheamă că sunt **clauze contradictorii**, care **se contrazic pe literalii** L_1 şi L_2, şi **resolventul lui** C_1 **şi** C_2 este clauza:

$$Res(C_1, C_2) = (C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma).$$

Exemplu 2 (Rezolvent a două clauze)**.**

Consideraţi clauzele:

$$p(f(x), g(y)) \vee q(x, y) \qquad \neg p(f(f(a)), g(z)) \vee q(f(a), g(z))$$

$L_1 = p(f(x), g(y))$ *şi* ${L_2}^c = p(f(f(a)), g(z))$ *pot fi unificaţi de cmgu* $\{x \leftarrow f(a), y \leftarrow z\}$ *şi rezolventul clauzelor este:*

$$q(f(a), z) \vee q(f(a), g(z)).$$

- Observaţi că cerinţa ca două clauze să nu aibă variabile în comun nu impune nici o restricţie pe mulţimea de clauze. Clauzele sunt implicit cuantificate universal, deci schimbarea numelui variabilelor nu schimbă înţelesul clauzelor.

Procedura generală de rezoluţie.

INTRARE: O mulţime S de clauze.

IEŞIRE: S este satisfiabilă sau S este nesatisfiabilă. Procedura poate să nu se termine.

- $S_0 = S$.
- Repeat
  - **Aleg** $C_1, C_2 \in S_i$ clauze contradictoriisi fie $C = Res(C_1, C_2)$.
  - Dacă $C = \emptyset$ termină cu "nesatisfiabil".
  - Altfel, $S_{i+1} = S_i \cup \{C\}$.
- until $S_{i+1} = S_i$
- Returnează "satisfiabil".

Procedura de mai sus poate să nu se termine, nu este o procedură de decizie (într-adevăr acest lucru e de aşteptat, din moment ce logica de ordinul întâi este nedecidabilă). Motivul pentru neterminare este existenţa modelelor infinite.

Exemplu 3 (Rezoluţie, din [Ben-Ari, 2001])**.**

Liniile 1-7 conţin mulţimea iniţială de clauze. Restul liniilor reprezintă execuţia procedurii. Pe fiecare linie avem rezolventul, cmgu şi numărul clauzelor contradictorii care au fost rezolvate.

1.	$\neg p(x) \vee q(x) \vee r(x, f(x))$		
2.	$\neg p(x) \vee q(x) \vee s(f(x))$		
3.	$t(a)$		
4.	$p(a)$		
5.	$\neg r(a, y) \vee t(y)$		
6.	$\neg t(x) \vee \neg q(x)$		
7.	$\neg t(x) \vee \neg s(x)$		
8.	$\neg q(a)$	$x \leftarrow a$	$3, 6$
9.	$q(a) \vee s(f(a))$	$x \leftarrow a$	2.4
10.	$s(f(a))$		$8, 9$
11.	$q(a) \vee r(a, f(a))$	$x \leftarrow a$	$1, 4$
12.	$r(a, f(a))$		8.11
13.	$t(f(a))$	$y \leftarrow f(a)$	$5, 12$
14.	$\neg s(f(a))$	$x \leftarrow f(a)$	$7, 13$
15.	$\emptyset$		$10, 14$

Exemplu 4 (Resoluţie cu redenumire de variabile, din [Ben-Ari, 2001])**.**

Primele patru clauze reprezintă mulţimea iniţială.

1.	$\neg p(x, y) \vee p(y, x)$		
2.	$\neg p(x, y) \vee \neg p(y, z) \vee p(x, z)$		
3.	$p(x, f(x))$		
4.	$p(x, x)$		
$3'$.	$p(x', f(x'))$		*Redenumeşte* 3.
5.	$p(f(x), x)$	$\sigma_1 = \{y \leftarrow f(x), x' \leftarrow x\}$	$1, 3'$
$3''$.	$p(x'', f(x''))$		*Redenumeşte* 3
6.	$\neg p(f(x), z) \vee p(x, z)$	$\sigma_2 = \{y \leftarrow f(x), x'' \leftarrow x\}$	$2, 3''$
$5'''$.	$p(f(x'''), x''')$		*Redenumeşte* 5
7.	$p(x, x)$	$\sigma_3 = \{z \leftarrow x, x''' \leftarrow x\}$	$6, 5'''$
$4''''$.	$\neg p(x'''', x'''')$		*Redenumeşte* 4
8.	$\emptyset$	$\sigma_4 = \{x'''' \leftarrow x\}$	$7, 4''''$

- *Substituţia rezultată din compunerea tuturor substituţiilor intermediare este:*

$$\sigma = \sigma_1\sigma_2\sigma_3\sigma_4 = \{y \leftarrow f(x), z \leftarrow x, x' \leftarrow x, x'' \leftarrow x, x''' \leftarrow x, x'''' \leftarrow x\}$$

- *Restricţionată la variabilele din mulţimea iniţială:*

$$\sigma = \{y \leftarrow f(x), z \leftarrow x\}$$

Teorema 6 (Corectitudinea rezoluţiei)**.**

Dacă clauza nesatisfiabilă $\emptyset$ este derivată de execuţia procedurii de substituţie, atunci mulţimea iniţială de clauze este nesatisfiabilă.

Teorema 7 (Completitudinea rezoluţiei)**.**

Dacă o mulţime de clauze este nesatisfiabilă, atunci clauza $\emptyset$ poate fi derivată prin aplicarea procedurii de rezoluţie.

- Pentru demonstraţii, vezi [Ben-Ari, 2001].

Câteva observaţii cu privire la procedura de rezoluţie

- Procedura de rezoluţie este nondeterministă: nu se specifică cum se aleg clauzele contradictorii şi nici care literali contradictorii se aleg.
- Alegeri bune vor conduce la derivări scurte, alegeri mai puţin bune vor duce la derivări lungi şi chiar la situaţii în care procedura nu se termină.
- Teorema de completitudine spune că dacă mulţimea de clauze este nesatisfiabilă, există o rezoluţie ce derivă clauza vidă $\emptyset$, dar asta nu înseamnă că alegerea bună care ar duce la aceasta este uşor de găsit.

11.5 De citit şi exerciţii

- De citit: Capitolul 7, secţiunile 7.5-7.8 din [Ben-Ari, 2001].
- Subiecte de interes:
  - Rezoluţia fără variabile, motive pentru impracticalitatea acesteia.
  - Substituţii: compoziţie, unificatori, cmgu.
  - Algoritm de unificare.
  - Rezoluţie generală, corectitudine, completitudine (fără demonstraţii).

12 Programare Logică

12.1 Formule ca programe

- Consideraţi un fragment din teoria şirurilor, cu simbolul funţional binar $\cdot$ (concatenare) şi predicatele binare $substr, prefix, suffix$, descrisede următoarele axiome:

  1. $\forall x\ substr(x, x)$
  2. $\forall x \forall y \forall z\ ((substr(x, y) \wedge suffix(y, z)) \Rightarrow substr(x, z))$
  3. $\forall x \forall y\ suffix(x, y \cdot x)$
  4. $\forall x \forall y \forall z\ ((substr(x, y) \wedge prefix(y, z)) \Rightarrow substr(x, z))$
  5. $\forall x \forall y\ prefix(x, x \cdot y)$

- Interpretarea procedurală a acestor formule este:

  1. x este un subşir al lui x.
  2. Pentru a verifica faptul că x este un subşir al lui z, e suficient să se găsească un sufix y al lui z şi să se verifice că x este un subşir al lui y.
  3. x este un sufix al lui $y \cdot x$,
  4. Pentru a verifica faptul că x este un subşir al lui z, este suficient să se găsească un prefix y al lui z şi să se verifice că x este un subşir al lui y.
  5. x este un prefix al lui $x \cdot y$.

- Forma clauzală a acestor axiome este:

  1. $substr(x, x)$
  2. $\neg substr(x, y) \vee \neg suffix(y, z) \vee substr(x, z)$
  3. $suffix(x, y \cdot x)$
  4. $\neg substr(x, y) \vee \neg prefix(y, z) \vee substr(x, z)$
  5. $prefix(x, x \cdot y)$

- Acum considerăm rezoluţia aplicată lui $\neg substr(a \cdot b \cdot c, a \cdot a \cdot b \cdot c \cdot c)$:

6.	$\neg substr(a \cdot b \cdot c, a \cdot a \cdot b \cdot c \cdot c)$	
7.	$\neg substr(a \cdot b \cdot c, y1) \vee \neg suffix(y1, a \cdot a \cdot b \cdot c \cdot c)$	$6, 2$
8.	$\neg substr(a \cdot b \cdot c, a \cdot b \cdot c \cdot c)$	$7, 3$
9.	$\neg substr(a \cdot b \cdot c, y2) \vee \neg prefix(y2, a \cdot b \cdot c \cdot c)$	$8, 4$
10.	$\neg substr(a \cdot b \cdot c, a \cdot b \cdot c)$	$9, 5$
11.	$\emptyset$	$10, 1$

  adică am arătat prin rezoluţie că $substr(a \cdot b \cdot c, a \cdot a \cdot b \cdot c \cdot c)$.

- Mai putem folosi rezoluţia pentru a verifica dacă $\exists w substring(w, a \cdot a \cdot b \cdot c \cdot c)$. Dacuă notăm axiomele cu $Axiome$, trebuie să arătam că următoarea formulă este nesatisfiabilă:

$$Axiome \wedge \neg(\exists w substring(w, a \cdot a \cdot b \cdot c \cdot c)).$$

  Dar formula este echivalentă cu

$$Axiome \wedge \forall w(\neg substring(w, a \cdot a \cdot b \cdot c \cdot c))$$

  care se poate scrie imediat în formă clauzală şi se poate aplica rezoluţia imediat.

- O execuţie a procedurii de rezoluţie reuşeste (exerciţiu!) şi substituţia $\{w \leftarrow a \cdot b \cdot c\}$ este generată: nu numai că am arătat consecinţa logică, dar am **calculat** o valoare pentru care $substring(w, a \cdot a \cdot b \cdot c \cdot c)$ este adevărată. În acest context, axiomele şirurilor constituie un **program** prin care se calculează **răspunsuri** la întrebări. De notat însă că acest program este nondeterminist.Alegerile făcute în timpul execuţiei programului (rezoluţie) influenţează rezultatul şi chiar răspunsul.

- Formalismul nondeterministic poate fi transformat într-un limbaj de programare practic prin specificarea regulilir pentru alegeri.

Definition 2.

O **regulă de calcul** este o regulă pentru alegerea unui literal dintr-o interogare pentru a se rezolva cu acesta (pas de rezoluţie). O **regulă de căutare** este o regulă pentru alegerea unei clauze pentru a rezolva cu literalul ales de regula de calcul.

- Diferenţa între programarea logică şi programarea imperativă este controlul programului:
  - în programarea imperativă controlul programului este furnizat explicit ca parte a codului de către programator,
  - în programarea logică programatorul scrie formule (declarative) ce descriu relaţia între datele de intrare şi datele de ieşire, iar rezoluţia cu regulile de calcul şi de căutare oferă o structură de control **uniformă**.

- În consecinţă, există cazuri în care un program logic nu va fi la fel de eficient ca un program imperativ ce beneficiază de structuri de control dedicate anumitor calcule şi situaţii.

12.2 Clauze Horn

Definition 3 (Clauze Horn)**.**

O **clauză Horn** este o clauză $A \Leftarrow B_1, \ldots, B_n$, cu cel mult un literal pozitiv. Literalul pozitiv A se cheamă **capul clauzei** şi literalii negativi B_i se cheamă **corpul clauzei**. O clauză Horn unitate pozitivă $A \Leftarrow$ se cheamă **fapt**, iar o clauză Horn fără literali pozitivi $\Leftarrow B_1, \ldots, B_n$ se cheamă **clauză obiectiv, sau interogare**. O clauză Horn cu un literal pozitiv şi unul sau mai mulţi literali negativi se cheamă **clauză program**.

Observaţi că notaţia $A \Leftarrow B_1, \ldots, B_n$ este echivalentă cu $(B_1 \wedge \ldots \wedge B_n) \Rightarrow A$ care la rândul ei este echivalentă cu $\neg B_1 \vee \ldots \vee \neg B_n \vee A$.

- Notaţia folosită în prima parte a cursului (în particular pentru sintaxa Prolog) pentru clauze program este A*:-*$B_1, \ldots, B_n$. Aceasta va fi notaţia folosită de acum încolo.

Definition 4 (Programe logice)**.**

O mulţime de clauze Horn ce nu sunt interogări şi a căror cap are acelaşi simbol predicativ se numeşte **procedură**. O mulţime de proceduri formează un **(program logic)**. O procedură compusă numai din fapte fără variabile se numeşte **bază de date**.

Exemplu 5 (Program logic, din [Ben-Ari, 2001])**.**

Următorul program are două proceduri:

1. $q(x, y)$*:-*$p(x, y)$
2. $q(x, y)$*:-*$p(x, z), q(z, y)$

3. $p(b, a)$
4. $p(c, a)$
5. $p(d, b)$
6. $p(e, b)$
7. $p(f, b)$
8. $p(h, g)$
9. $p(i, h)$
10. $p(j, h)$

Definition 5 (Substituţie răspuns corectă)**.**

Fie P un program şi G o clauză obiectiv (interogare). O substituţie θ pentru variabile în G se cheamă o **substituţie răspuns corectă** dacă şi numai dacă $P \models \forall(\neg G\theta)$, unde prin $\forall$ se indică închiderea universală a variabilelor care sunt libere în G.

În alte cuvinte, substituţia răspuns corectă face negaţia clauzei obiectiv să fie o consecinţă logică a programului.

Exemplu 6.

Consideraţi o rezoluţie pentru clauza obiectiv :-$q(y,b), q(b,z)$ din programul introdus în Examplul 5. La fiecare pas, se alege un literal din clauza obiectiv şi o clauză program a cărui cap este contradictoriu cu literalul selectat:

1. *Se alege $q(y,b)$ şi se rezolvă cu clauza 1, obţinându-se :-$p(y,b), q(b,z)$.*
2. *Se alege $p(y,b)$ şi se rezolvă cu clauza 5, obţinându-se :-$q(b,z)$. Substitutţia necesară este $\{y \leftarrow d\}$.*
3. *Se alege literalul rămas :-$q(b,z)$ şi se rezolvă cu clauza 1, obţinându-se :-$p(b,z)$.*
4. *Se alege literalul rămas :-$p(b,z)$ şi se rezolvă cu clauza 1, obţinându-se $\emptyset$. Substitutţia necesară este $\{z \leftarrow a\}$.*

Am obţinut clauza vidă $\emptyset$. Cu stubstituţia răspuns corectă $\{y \leftarrow d, z \leftarrow a\}$ aplicată clauzei obiectiv, obţinem

$$P \vDash q(d,b) \wedge q(b,a).$$

12.3 SLD rezoluţie

Definition 6 (SLD Rezoluţie)**.**

Fie P o mulţime de clauze program, R o regulă de calcul şi G o clauză obiectiv. O **derivare prin SLD-rezoluţie** se defineşte ca o secvenţă de paşi de rezoluţie între clauza obiectiv şi clauzele program. Prima clauză obiectiv G_0 este G. Presupunem că G_i a fost derivată. G_{i+1} este definită prin **selectarea** unui literal $A_i \in G_i$ conform regulii de calcul R, **alegerea** unei clauze$C_i \in P$ astfel încât C_i se unifică cu A_i prin cmgu θ_i şi rezolvarea:

$$\begin{array}{rcl} G_i & = & \text{:-}A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n \\ C_i & = & A\text{:-}B_1, \ldots, B_k \\ A_i\theta_i & = & A\theta_i \\ G_{i+1} & = & \text{:-}(A_1, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1}, \ldots, A_n)\theta_i \end{array}$$

O **SLD respingere** este o derivare prin SLD-rezoluţie a clauzei vide $\emptyset$.

Corectitudine şi completitudine a SLD-rezoluţiei

Teorema 8 (Corectitudine a SLD-rezoluţiei)**.**

Fie P o mulţime de clauze program, R o regulă de calcul şi G o clauză obiectiv. Dacă există o SLD respingere a lui G, $\theta = \theta_1 \ldots \theta_n$ este sevenţa de unificatori folosită în respingere şi σ este restricţia lui θ la variabilele din G, atunci σ este o substituţie răspuns corectă pentru G.

Proof. Vezi [Ben-Ari, 2001], pp. 178. □

Teorema 9 (Completitudinea SLD-rezoluţiei)**.**

Fie P o mulţime de clauze program, R o regulă de calcul şi G o clauză obiectiv. Fie σ o substituţie răspuns corectă. Atunci există o SLD-respingere a lui G din P astfel încât σ este restricţia secvenţei de unificatori $\theta = \theta_1, \ldots, \theta_n$ la variabilele din G.

Proof. Vezi [Ben-Ari, 2001], pp. 179. □

- De remarcat că rezultatele de mai sus sunt adevărate numai pentru clauze Horn (programe logice). SLD-rezoluţia nu este completă pentru clauze arbitrare:

  $$p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q$$

  este nesatisfiabilă, dar nu există o SLD-respingere a lui $\emptyset$ din ea (exerciţiu!).

Exemplu 7 (Examplele 5, 6, revizitate)**.**

- *Dacă, în pasul 2 din Examplul 6 clauza 6 ar fi fost aleasă pentru a se rezolva, rezolventul ar fi fost :-$q(b, z)$. Rezoluţia va avea succes (exerciţiu!) dar substituţia răspuns corectă va fi diferită: $\{y \leftarrow e, z \leftarrow a\}$. Pentru o clauză obiectiv dată pot exista mai multe substituţii răspuns corecte.*

- *Presupunând că regula de calcul este să se aleagă tot timpul ultimul literal din clauza obiectiv. Rezolvând întotdeauna cu clauza 2 se obţine:*

  :-$q(y, b), q(b, z)$
  :-$q(y, b), p(b, z'), q(z', z)$
  :-$q(y, b), p(b, z'), p(z', z''), q(z'', z)$
  :-$q(y, b), p(b, z'), p(z', z''), p(z'', z'''), q(z''', z)$
  $\ldots$

  O substituţie răspuns corectă există dar procedura de rezoluţie nu se termină.

- *Consideraţi regula de calcul ce alege tot timpul primul literal din clauza obiectiv. SLD-rezoluţia se desfăsoară după cum urmează:*

  1. *$q(y, b)$ este ales şi rezolvat cu clauza 2, obţinându-se :-$p(y, z'), q(z', b), q(b, z)$*
  2. *Se alege primul literal $p(y, z')$ şi se rezolvă cu clauza 6 $p(e, b)$, apoi cu clauza 1 şi se obţine :-$q(b, b), q(b, z)$, apoi :-$p(b, b), q(b, z)$.*
  3. *Nici o clauză program nu se unifică cu $p(b, b)$ şi această rezoluţie eşuează.*

  Deşi există o substituţie răspuns corectă, rezoluţia se termină cu eşec.

Definition 7 (SLD-arbori)**.**

Fie P o mulţime de clauze program, R o regulă de calcul şi G o clauză obiectiv. Toate SLD-derivările posibile pot fi reprezentate într-un **SLD-arbore**, construit în modul următor:

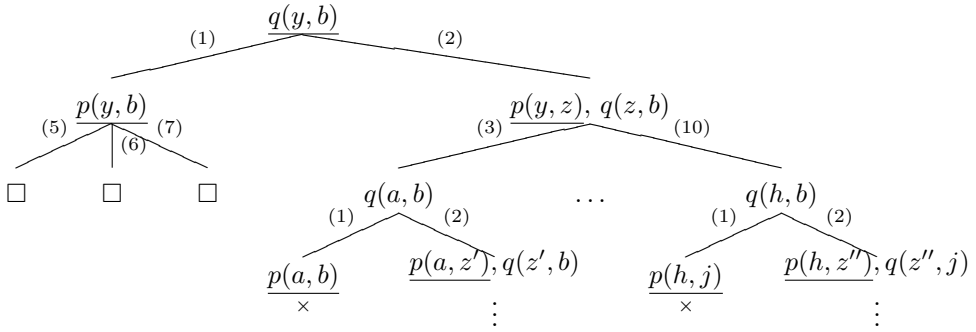
- Rădăcina este clauza obiectiv G.
- Dat fiind un nod n ce conţine clauza obiectiv G_n, se crează nodul fiu n_i pentru fiecare nouă clauză obiectiv G_{n_i} care poate fi obţinută prin rezolvarea literalului ales de R cu capul unei clauze din P.

Definition 8.

În an SLD-arbore, o ramură ce duce la o respingere se cheamă **ramură succes**. O ramură ce duce la o clauză obiectiv ce nu mai poate fi rezolvată se cheamă **ramură eşec**. O ramură ce corespunde unei derivări ce nu se termină se cheamă **ramură infinită**.

Exemplu 8 (SLD-arbore)**.**

Mai jos este SLD arborele pentru derivarea din Exemplele 5, 6 unde regula de calcul este rezolvarea celui mai din stânga literal din clauza obiectiv (vezi, de asemenea, Exemplul 7). Literalul ales este subliniat, clauza cu care se rezolvă este indicată pe fiecare muchie. Ramurile succes sunt indicate cu $\square$*, ramurile eşec cu* $\times$*, ramurile infinite cu* $\vdots$*.*



Teorema 10.

Fie P un program şi G o clauză obiectiv. Atunci fiecare SLD-arbore pentru P şi G are un număr infinit de ramuri infinite, sau toţi SLD-arborii au acelaşi număr finit de ramuri succes.

Proof. Nu este dată. $\square$

Definition 9.

O **regulă de căutare** este o procedură de căutare a unei respingeri (ramuri succes) în SLD-arbore. O **procedură de SLD-respingere** este algoritmul de SLD-rezoluţie algorithm cu regula de calcul şi regula de căutare specificate.

Câteva comentarii adupra completitudinii SLD rezoluţiei

- Observaţi că SLD-rezoluţia este completă pentru orice regulă de calcul (o respingere există). Totuşi, alegerea regulii de căutare este esenţială pentru găsirea respingerii.
- Dacă se consideră o noţiune mai restrictivă de completitudine (o respingere există **şi este găsită**, anumite reguli de căutare fac SLD rezoluţia incompletă: de exemplu căutarea pe adâncime (depth-first) (folosită de Prolog: Prolog = SLD rezoluţie cu selectarea literalului cel mai din stânga ca regulă de calcul şi căutare în adâncime).
- Există reguli de căutare pentru care SLD rezoluţia este completă (în sensul mai restrictiv):
  - căutarea pe lăţime (breath-first) – se caută prima dată soluţia în fiecare nivel al SLD arborelui,
  - căutarea în adâncime cu limite (bounded depth first) – se coboară în adâncime pâna la un anumit nivel, dacă nu se găseşte soluţia se revine, dacă spaţiul de căutare este epuizat fără a găsi soluţia, se măreşte adâncimea.

  Totuşi, aceste reguli complete sunt scumpe din punct de vedere computaţional. În practică se sacrifică completitudinea pentru viteză.

12.4 De citit şi exerciţii

- De citit: Capitolul 8, secţiunile 8.1-8.3 din [Ben-Ari, 2001].
- Alte surse: Capitolul 2, 3 din [Nilsson, Maluszynski, 2000].
- Subiecte de interes:
  - Interpretare declatativă şi procedurală a formulelor.
  - Rezoluţia ca metodă de calcul. Nondeterminismul rezoluţiei.
  - Reguli de calcul, reguli de căutare (în contextul rezoluţiei).
  - Clauze Horn.
  - SLD-rezoluţie. Completitudine, corectitudine (fără demonstrazţii).
  - Clauze Prolog exprimante sub formă de clauze Horn.
  - Traducerea clauzelor Prolog (Horn) în/din formule din logica predicatelor.
  - Prolog: probleme cu completitudinea.

Partea III
Tehnici Avansate de Programare Logică

13 Limbaj obiect, metalimbaj (inc.)

13.1 Ce este metalimbajul

13.2 Un autointerpretor Prolog în reprezentare fără variabile

13.3 Un autointerpretor Prolog în reprezentare cu variabile

13.4 `clause/2`

13.5 De citit

- De citit: Chapitolul 8 din [Nilsson, Maluszynski, 2000].
- Subiecte de interes:
  - Limbaj obiect / metalimbaj - diferenţe, exemple.
  - Interpretoare metacirculare. Diferenţierea între limbaj obiect şi metalimbaj.
  - Interpretoare metacirculare în Prolog: posibilităţi de implementare. Avantaje, dezavantaje.
  - Predicatul Prolog `clause/2`.

14 Gramatici de clauze definite - GCD (inc.)

14.1 Problema parcurgerii

14.2 Folosirea listelor de diferenţe pentru parcurgere

14.3 Contexte

14.4 GCD în Prolog

14.5 De citit

- De citit: Capitolul 10 din [Nilsson, Maluszynski, 2000].
- Subiecte de interes:
  - Gramatici libere de context, specificarea limbajelor, parcurgerea expresiilor din limbaje.
  - Parcurgerea expresiilor în Prolog folosind liste de diferenţe.
  - Contexte.
  - Mecanisme Prolog pentru parcurgere GCD, simboluri terminale, simboluri nonterminale, –¿.
  - Compilarea GCD-urilor în Prolog.
  - Exemple.

15 Programare Logică cu Constrângeri (inc.)

15.1 De citit

- De citit: Secţiunea 8.5 din [Ben-Ari, 2001], Capitolul 14 din [Nilsson, Maluszynski, 2000].
- Items of interest:
  - Constrângeri, derivări, rezervor de constrângeri, constrângere răspuns, răspuns condiţionat, evaluare lazy/eager, domenii de constrângere: boolean, numere reale, numere raţionale, domenii finite.

Bibliografie

[Ben-Ari, 2001] Mordechai Ben-Ari, *Mathematical Logic for Computer Science*, Springer-Verlag London, Second Edition, 2001.

[Brna, 1988] Paul Brna, *Prolog Programming A First Course*, Copyright Paul Brna, 1988.

[Covington, 1989] M. A. Covington, *Efficient Prolog: A Practical Guide*, Research Report AI-1989-08, The University of Georgia, Athens, Georgia, 1989.

[Covington et al, 1997] M.A. Covington, D. Nute, A. Vellino, *Prolog Programming in Depth* Prentice Hall, New Jersey, 1997.

[Craciun, 2009] Adrian Crăciun, *Logic for Computer Science*, Lecture Notes, `http://web.info.uvt.ro/~acraciun/lectures/logic/pdf/logicForCSNotes.pdf`.

[Kowalski, 1979] Robert Kowalski, *Logic for Problem Solving*, North Holland New York, Amsterdam, Oxford, 1979.

[Mellish, Clocksin 1994] C.S. Mellish, W. F. Clocksin, *Programming in Prolog*, Springer Verlag Telos, 4th edition, 1994.

[Nilsson, Maluszynski, 2000] Ulf Nilsson, Jan Maluszynski, *Logic, Programming and Prolog*, 2nd Edition, copyright Ulf Nilsson and Jan Maluszynski, 2000.