Curs 13

2019-2020

Programare Logică

Cuprins

Prolog impur

Negarea unui predicat: `\+ pred(X)`

Exemplu

```
animal(dog). animal(elephant). animal(sheep).

?- animal(cat).
false
?- \+ animal(cat).
true
```

Negarea unui predicat: `\+ pred(X)`

Exemplu

```
animal(dog). animal(elephant). animal(sheep).

?- animal(cat).
false
?- \+ animal(cat).
true
```

- Clauzele din Prolog dau doar condiții suficiente, dar nu și necesare pentru ca un predicat să fie adevărat.
- Pentru a da un răspuns pozitiv la o țintă, Prolog trebuie să construiască o "demonstrație" pentru a arată că mulțimea de fapte și reguli din program implică acea țintă.
- Astfel, un răspuns `false` nu înseamnă neapărat că ținta nu este adevărată, ci doar că Prolog nu a reușit să găsească o demonstrație.

Operatorul \+

- Negarea unei ținte se poate defini astfel:

```
neg(Goal) :- Goal, !, fail.
neg(Goal)
```

unde `fail/0` este un predicat care eșuează întotdeauna.

Operatorul \+

- Negarea unei ținte se poate defini astfel:

```
neg(Goal) :- Goal, !, fail.
neg(Goal)
```

  unde `fail/0` este un predicat care eșuează întotdeauna.
- În PROLOG acest predicat este predefinit sub numele \+.
- Operatorul \+ se foloseste pentru a nega un predicat.
- !(cut) este un predicat predefinit (de aritate 0) care restricționează mecanismul de backtracking: execuția subțintei ! se termină cu succes, deci alegerile (instanțierile) făcute înaite de a se ajunge la ! nu mai pot fi schimbate.
- O țintă \+ `Goal` reușește dacă Prolog nu găsește o demonstrație pentru `Goal`. Negația din Prolog este definită ca incapacitatea de a găsi o demonstrație.
- Semantica operatorului \+ se numește negation as failure.

Negația ca eșec ("negation as failure")

Exemplu

Să presupunem că avem o listă de fapte cu perechi de oameni căsătoriți între ei:

```
married(peter, lucy).
married(paul, mary).
married(bob, juliet).
married(harry, geraldine).
```

Negația ca eșec

Exemplu (cont.)

Putem să definim un predicat `single/1` care reușește dacă argumentul său nu este nici primul nici al doilea argument în faptele pentru `married`.

```
single(Person) :-
        \+ married(Person, _),
        \+ married(_, Person).
```

```
?- single(mary).   ?- single(anne).   ?- single(X).
false              true               false
```

Răspunsul la întrebarea `?- single(anne).` trebuie gândit astfel:

Presupunem că Anne este single,
deoarece nu am putut demonstra că este maritată.

Raționamente nemonotone

- În exemplul anterior extindem baza de cunoștințe astfel:

```
married(peter, lucy). married(paul, mary).
married(bob, juliet). married(harry, geraldine).
married(john, anne).

single(P) :- \+ married(P, _), \+ married(_, P).

?- single(anne).
false
```

Raționamente nemonotone

- În exemplul anterior extindem baza de cunoștințe astfel:

```
married(peter, lucy). married(paul, mary).
married(bob, juliet). married(harry, geraldine).
married(john, anne).

single(P) :- \+ married(P, _), \+ married(_, P).

?- single(anne).
false
```

Observăm că largind mulțimea de ipoteze (baza de cunoștințe) putem demonstra mai puțin! Acest tip de raționament se numește nemonoton.

Raționamente nemonotone

- În exemplul anterior extindem baza de cunoștințe astfel:

```
married(peter, lucy). married(paul, mary).
married(bob, juliet). married(harry, geraldine).
married(john, anne).

single(P) :- \+ married(P, _), \+ married(_, P).

?- single(anne).
false
```

  Observăm că largind mulțimea de ipoteze (baza de cunoștințe) putem demonstra mai puțin! Acest tip de raționament se numește nemonoton.

- Sistemele logice pe care le-am sudiat până acum (calculul cu propoziții clasic, logica de ordinul I, logica clauzelor Horn) sunt monotone:

  dacă din $\Gamma \vdash \varphi$ și $\Gamma \subseteq \Sigma$ atunci $\Sigma \vdash \varphi$.

Lista tuturor soluțiilor

Cum găsim lista tuturor soluțiilor unui predicat?

- În Prolog există meta-predicatul `findall/3`, care acceptă ca argument un predicat arbitrar.

  KB: `p(a). p(b). p(c). p(d). p(a).`

  ```
  ?- findall(X, p(X),S).
  S = [a, b, c, d, a].
  ```

- Definiția lui `findall/3`

```
?- listing(findall/3).
:- meta_predicate'$bags':findall(?,0,-).
'$bags':findall(Templ, Goal, List) :- findall(Templ, Goal, List, []).
?- listing(findall/4).
:- meta_predicate'$bags':findall(?,0,-,?).
'$bags':findall(Templ, Goal, List, Tail) :-
    setup_call_cleanup('$new_findall_bag',
                       findall_loop(Templ, Goal, List,Tail),
                       '$destroy_findall_bag').
```

Liste

Exercițiu

Fie `p/1` un predicat. Scrieți un predicat `all_p/1` astfel încât întrebarea `?- all_p(S)` să instanțieze `S` cu lista tuturor atomilor pentru care `p` este adevărat.

```
p(a). p(b). p(c). p(d). p(a).

?- all_p(S).
S = [d,c,b,a].
```

Liste

Exercițiu

Fie `p/1` un predicat. Scrieți un predicat `all_p/1` astfel încât întrebarea `?- all_p(S)` să instanțieze `S` cu lista tuturor atomilor pentru care `p` este adevărat.

```
p(a). p(b). p(c). p(d). p(a).

?- all_p(S).
S = [d,c,b,a].


find_all(X,L,S):- p(X), \+ member(X,L),
                  find_all(_,[X|L],S).
find_all(_,L,L).


all_p(S) :- find_all(_, [], S).
```

Lista tuturor soluțiilor fără repetiții

```
find_all(X,L,S):- p(X), \+ member(X,L), find_all(_,[X|L],S).
find_all(_,L,L).
all_p(S) :- find_all(_, [], S).


?- all_p(S).
S = [d, c, b, a].

?- all_p([a,b,c,d]).
true.

?- all_p([a,b,c]).
true.

?- all_p([a,b,c,a]).
false. % pentru că a apare de două ori
```

Predicate ca argumente

Putem scrie predicatul `all_p` astfel încât să-i transmitem predicatul ca argument?

Ar trebui ca numele predicatului să fie o variabilă care să fie instanțiată în momentul apelului, dar acest lucru nu este permis de sintaxa Prolog (un functor trebuie să fie un atom).

Există o soluție folosind predicatul predefinit `=../2` care convertește un predicat p(t1,...,tn) în lista [p,t1,..., tn].

```
?- p(a) =.. L.
L = [p, a].

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

Predicate ca argumente

Predicatul predefinit =../2 convertește un predicat p(t1,...,tn) în lista [p,t1,..., tn].

```
find_all(P,X,L,S):- Pr =..[P,X],Pr, \+ member(X,L),
                    find_all(P,_,[X|L],S).
find_all(_,_,L,L).
all(P,S) :- find_all(P,_, [], S).
```

Predicate ca argumente

Predicatul predefinit =../2 convertește un predicat p(t1,...,tn) în lista [p,t1,..., tn].

```
find_all(P,X,L,S):- Pr =..[P,X],Pr, \+ member(X,L),
                    find_all(P,_,[X|L],S).
find_all(_,_,L,L).
all(P,S) :- find_all(P,_, [], S).

p(a). p(b). p(c). p(d). p(a).
q(a). q(b). q(c).

?- all(q,S).
S = [c, b, a].

?- all(p,S).
S = [d, c, b, a].
```

Concluzii

- Programarea logică este bazată pe o teorie matematică clară: logica clauzelor Horn. Semantica denotațională a unui program poate fi defnită matematic, iar semantica operațională este bazată pe rezoluție.
- Limbajul Prolog este un limbaj complex, care îmbină cosntrucii teoretice (care au un corespondent în logică) cu trăsături nelogice, care cresc puterea de expresivitate.
- La acest curs ne interesează în special partea pură a limbajului Prolog, adică acele programe care se pot exprima în logica clauzelor Horn definite.

DCG (Definite Clause Grammars)

Structura frazelor

- Aristotel, On Interpretation, `http://classics.mit.edu/Aristotle/interpretation.1.1.html`:

  "Every affirmation, then, and every denial, will consist of a noun and a verb, either definite or indefinite."

Structura frazelor

- Aristotel, On Interpretation,
  `http://classics.mit.edu/Aristotle/interpretation.1.1.html`:

  "Every affirmation, then, and every denial, will consist of a noun and a verb, either definite or indefinite."

- N. Chomsy, Syntactic structure, Mouton Publishers, First printing 1957 - Fourteenth printing 1985 [Chapter 4 (Phrase Structure)]

(i) $Sentence \rightarrow NP + VP$
(ii) $NP \rightarrow T + N$
(iii) $VP \rightarrow Verb + NP$
(iv) $T \rightarrow the$
(q) $N \rightarrow fman, ball$, etc.
(vi) $V \rightarrow hit, took$, etc.

Gramatică independentă de context

- Definim structura propozițiilor folosind o gramatică independentă de context:

S	$\rightarrow$	NP VP
NP	$\rightarrow$	Det N
VP	$\rightarrow$	V
VP	$\rightarrow$	V NP
Det	$\rightarrow$	*the*
Det	$\rightarrow$	*a*
N	$\rightarrow$	*boy*
N	$\rightarrow$	*girl*
V	$\rightarrow$	*loves*
V	$\rightarrow$	*hates*

- Neterminalele definesc categorii gramaticale:

  S (propozițiile),
  NP (expresiile substantivale),
  VP (expresiile verbale),
  V (verbele),
  N (substantivele),
  Det (articolele).

- Terminalele definesc cuvintele.

Gramatică independentă de context

GIC

S	$\rightarrow$	NP VP	Det	$\rightarrow$	*the*
NP	$\rightarrow$	Det N	Det	$\rightarrow$	*a*
VP	$\rightarrow$	V	N	$\rightarrow$	*boy*
VP	$\rightarrow$	V NP	N	$\rightarrow$	*girl*
			V	$\rightarrow$	*loves*
			V	$\rightarrow$	*hates*

Ce vrem să facem?

- Vrem să scriem un program în Prolog care să recunoască propozițiile generate de această gramatică.
- Reprezentăm propozițiile prin liste.

```
?- atomic_list_concat(SL,' ' , 'a boy loves a girl').
SL = [a, boy, loves, a, girl]
```

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

```
SL = [a, boy, loves, a, girl]
```

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

  `SL = [a, boy, loves, a, girl]`

- Fiecărui neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective. `n([boy]).`

  `n([girl]).  det([the]).  v([loves]).`

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

  ```
  SL = [a, boy, loves, a, girl]
  ```

- Fiecărui neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective. `n([boy]).`

  ```
  n([girl]).  det([the]).  v([loves]).
  ```

- Lista asociată unei propoziții se obține prin concatenarea listelor asociate elementelor componente.

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

  `SL = [a, boy, loves, a, girl]`

- Fiecărui neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective. `n([boy]).`

  `n([girl]).  det([the]).  v([loves]).`

- Lista asociată unei propoziții se obține prin concatenarea listelor asociate elementelor componente.

  De exemplu, interpretăm regula S $\rightarrow$ NP VP astfel:

  o propozi tie este o listă `L` care se obține prin concatenarea a două liste, `X` și `Y`, unde `X` reprezintă o expresie substantivală și `Y` reprezintă o expresie verbală.

  ```
  s(L) :- np(X), vp(Y), append(X,Y,L).
  ```

Definirea unei gramatici în Prolog

Gramatică independentă de context

S	$\rightarrow$	NP VP	Det	$\rightarrow$	*the*
NP	$\rightarrow$	Det N	Det	$\rightarrow$	*a*
VP	$\rightarrow$	V	N	$\rightarrow$	*boy*
VP	$\rightarrow$	V NP	N	$\rightarrow$	*girl*
			V	$\rightarrow$	*loves*
			V	$\rightarrow$	*hates*

Prolog

```
s(L) :- np(X), vp(Y),
        append(X,Y,L).
np(L) :- det(X), n(Y),
         append(X,Y,L) .
vp(L) :- v(L).
vp(L):- v(X), np(Y),
        append(X,Y,L) .
```

```
det([the]).
det([a]).
n([boy]).
n([girl]).
v([loves]).
v([hates]).
```

Definirea unei gramatici în Prolog

```
s(L) :- np(X), vp(Y),
        append(X,Y,L).

np(L) :- det(X), n(Y),
         append(X,Y,L) .

vp(L) :- v(L).
vp(L):- v(X), np(Y),
        append(X,Y,L) .

det([the]).
det([a]).
n([boy]).
n([girl]).
v([loves]).
v([hates]).
```

```
?- s([a,boy,loves, a,
girl]).
true .

?- s[a, girl|T].
T = [loves] ;
T = [hates] ;
T = [loves, the, boy] ;
⋮

?- s(S).
S = [the, boy, loves] ;
S = [the, boy, hates] ;
⋮
```

Definirea unei gramatici în Prolog

- Deși corectă, reprezentarea anterioară este ineficientă, arborele de căutare este foarte mare.

Definirea unei gramatici în Prolog

- Deși corectă, reprezentarea anterioară este ineficientă, arborele de căutare este foarte mare.
- Pentru a optimiza, folosim *reprezentarea listelor ca diferențe*, plecând de la observația că

  `append(X,Y,L)` este echivalent cu `X = L - Y`

  - lista `[t1,...,tn]` va fi reprezentată printr-o pereche

    `([t1,...,tn|T], T)`

  - definiția concatenării este:

    ```
    dlappend((R,P),(P,T),(R,T)).
    ```

  - `dlappend` este foarte rapid, dar nu poate fi folosit pentru generare, ci numai pentru verificare.

Definirea unei gramatici în Prolog

Regula `s(L) :- np(X), vp(Y), append(X,Y,L)` devine

```
s(L,Z) :- np(L,Y), vp(Y,Z)
```

Definirea unei gramatici în Prolog

Regula `s(L) :- np(X), vp(Y), append(X,Y,L)` devine

```
s(L,Z) :- np(L,Y), vp(Y,Z)
```

Acestă scriere are și următoarea semnificație:

- fiecare predicat care definește o categorie gramaticală (în exemplu: `s`, `np`, `vp`,`det`, `n`, `v`) are ca argumente o *listă de intrare* `In` și o *listă de ieșire* `Out`
- predicatul consumă din `In` categoria pe care o definește, iar lista `Out` este ceea ce a rămas neconsumat.

  De exemplu: `np(L,Y)` consumă expresia substantivală de la începutul lui `L`, `v(L,Y)` consumă verbul de la începutul lui `L`, etc.

Definirea unei gramatici în Prolog

```
?- s([a, boy, loves, a , girl], []).
true.
```

```
s(L,M) :- np(L,Y),
          vp(Y,M).
np(L,M) :- det(L,Y),
           n(Y,M).
vp(L,M) :- v(L,M).
vp(L,M):- v(L,Y),
          np(Y,M).
det([the|M],M).
det([a|M],M).
n([boy|M],M).
n([girl|M],M).
v([loves|M],M).
v([hates|M],M).
```

Definirea unei gramatici în Prolog

```
s(L,M) :- np(L,Y),
          vp(Y,M).
np(L,M) :- det(L,Y),
           n(Y,M).
vp(L,M) :- v(L,M).
vp(L,M):- v(L,Y),
          np(Y,M).
det([the|M],M).
det([a|M],M).
n([boy|M],M).
n([girl|M],M).
v([loves|M],M).
v([hates|M],M).
```

```
?- s([a, boy, loves, a , girl], []).
true.

?- s([a, boy |M], M).
M = [loves|M] ;
M = [hates|M] ;
M = [loves, the, boy|M] ;
...
```

Definirea unei gramatici în Prolog

```
s(L,M) :- np(L,Y),
          vp(Y,M).
np(L,M) :- det(L,Y),
           n(Y,M).
vp(L,M) :- v(L,M).
vp(L,M):- v(L,Y),
          np(Y,M).
det([the|M],M).
det([a|M],M).
n([boy|M],M).
n([girl|M],M).
v([loves|M],M).
v([hates|M],M).
```

```
?- s([a, boy, loves, a , girl], []).
true.

?- s([a, boy |M], M).
M = [loves|M] ;
M = [hates|M] ;
M = [loves, the, boy|M] ;
...

?- s(L, []).
L = [the, boy, loves] ;
L = [the, boy, hates] ;
...
```

Definirea unei gramatici în Prolog

```
s(L,M) :- np(L,Y),
          vp(Y,M).
np(L,M) :- det(L,Y),
           n(Y,M).
vp(L,M) :- v(L,M).
vp(L,M):- v(L,Y),
          np(Y,M).
det([the|M],M).
det([a|M],M).
n([boy|M],M).
n([girl|M],M).
v([loves|M],M).
v([hates|M],M).
```

```
?- s([a, boy, loves, a , girl], []).
true.

?- s([a, boy |M], M).
M = [loves|M] ;
M = [hates|M] ;
M = [loves, the, boy|M] ;
...

?- s(L, []).
L = [the, boy, loves] ;
L = [the, boy, hates] ;
...

?- s([X|M], M).
X = the,
M = [boy, loves|M] ;
X = the,
M = [boy, hates|M] ;
...
```

DCG în Prolog

- DCG(Definite Clause Grammar) este o notație introdusă pentru a facilita definirea gramaticilor.
- În loc de s(L,M) :- np(L,Y), vp(Y,M). vom scrie

```
s --> np, vp.
```

iar codul scris anterior va fi generat automat.

Definite Clause Grammar

```
s    -->  np, vp.
np   -->  det, n.
vp   -->  v.
vp   -->  v, np.
```

```
det  -->  [the].
det  -->  [a].
n    -->  [boy].
n    -->  [girl].
v    -->  [loves].
v    -->  [hates].
```

```
?- listing(s).
s(A, B) :- np(A, C), vp(C, B).
```

DCG în Prolog

Definite Clause Grammar

```
s    -->  np, vp.
np   -->  det, n.
vp   -->  v.
vp   -->  v, np.

det  -->  [the].
det  -->  [a].
n    -->  [boy].
n    -->  [girl].
v    -->  [loves].
v    -->  [hates].
```

- Putem pune întrebările ca înainte:

```
?- s([the, girl, hates, the, boy], []).
true.
```

- Putem folosi predicatul phrase/2:

```
?- phrase(s, [the, girl, hates, the, boy]).
true.
```

DCG în Prolog

```
?- phrase(s, [the, girl, hates, the, boy]).
true.

?- phrase(s, X).
X = [the, boy, loves] .
X = [the, boy, loves] ;
X = [the, boy, hates] ;
...


?- phrase(np,X). %toate expresiile substantivale
X = [the, boy] ;
X = [the, girl] ;
X = [a, boy] ;
X = [a, girl].


?- phrase(v,X). % toate verbele
X = [loves] ;
X = [hates].
```

DCG în Prolog

Exemplu

Definiți numerele naturale folosind DCG.

DCG în Prolog

Exemplu

Definiți numerele naturale folosind DCG.

```
nat --> [o].
nat --> [s], nat.
```

Definiția generată automat este:

```
?- listing(nat).
nat([o|A], A).
nat([s|A], B) :- nat(A, B).
```

Putem transforma listele în atomi:

```
is_nat(X) :- phrase(nat,Y), atomic_list_concat(Y,'',X).
```

```
?- is_nat(X).
X = o ; X = so ; X = sso ; X = ssso ; X = sssso;
...
```

Chatbot: Eliza

Eliza

- Am văzut ca gramaticile DCG pot fi folosite pentru prelucrarea limbajului natural.
- În 1966, J. Weizenbaum (MIT) a scris un articol în care prelucarea limbajului natural se face are la bază recunoaștere șabloanelor (*pattern matching*).
  "ELIZA is a program operating within the MAC time-sharing system at MIT which makes certain kinds of natural language conversation between man and computer possible. Input sentences are analyzed on the basis of decomposition rules which are triggered by key words appearing in the input text. Responses are generated by reassembly rules associated with selected decomposition rules. The fundamental technical problems with which ELIZA is concerned are: 1.the identification of key words, 2.the discovery of minimal context, 3.the choice of appropriate transformations, 4.generation of responses in the absence of keywords, and 5.the provision of an ending capacity for ELIZA "scripts"."

Eliza: o variantă Prolog (cu menționarea autorilor)

```
Hello. I am ELIZA. How can I help you?
> I am upset.
Is it because you are upset that you came to me?
> yes.
You seem quite positive.
> I am not positive.
How long have you been not positive?
> Two days. How can you help me?
Why do you ask?
> I want your help.
What would it mean to you if you got my help?
> Maybe I will feel better.
You do not seem quite certain.
> I am not.
Do you believe it normal to be not?
> I think it is normal to be happy.
You say you think it is normal to be happy
> yes.
You are sure?
> Do you have any advise?
We were discussing you, not me.
```

Eliza: varianta simplă

- Vom implementa o varianta mai simplă:

```
?- eliza.
tell me
|: 'hello'.
Hi there !
|: 'i am very upset'.
How long have you been very upset ?
|: 'two days'.
Please go on .
|: 'can you help me'.
What makes you think I help you ?
|: 'my sister told me'.
Please you tell me more about sister
```

Eliza: varianta simplă

- Programul nostru va trebui să:
  - definească un set de perechi de șabloane, unul pentru intrare și unul pentru ieșire
  - să identifice ce șablon se aplică șirului de intrare
  - să construiască răspunsul pe baza șablonului pereche

```
pattern([i,am,1],['How',long,have,you,been,1,?]).
```

Intrarea:	`'i am very unhappy'`
Șablonul de intrare:	`[i, am, 1]`
Șablonul de ieșire	`['How',long,have,you,been,1,?]`
Ieșirea:	`How long have you been very unhappy?`

Eliza: varianta simplă

- Programul nostru va trebui să:
  - definească un set de perechi de șabloane, unul pentru intrare și unul pentru ieșire
  - să identifice ce șablon se aplică șirului de intrare
  - să construiască răspunsul pe baza șablonului pereche

```
pattern([i,am,1],['How',long,have,you,been,1,?]).
```

Intrarea:	`'i am very unhappy'`
Șablonul de intrare:	`[i, am, 1]`
Șablonul de ieșire	`['How',long,have,you,been,1,?]`
Ieșirea:	`How long have you been very unhappy?`

- procedeul este mecanic: nu se analizează din punct de vedere sintactic sau semantic frazele!

Eliza: varianta simplă

- Se definesc diferite șabloane:

```
pattern([i,am,1],['How',long,have,you,been,1,?]).

pattern([i,like,1],['Does,anyone,else,in,your,
                                        family,like,1,?]).

pattern([i,feel|_],['Do',you,often,feel,that,way,?]).

'
```

- Trebuie să existe un șablon pentru toate celelalte cazuri:

```
pattern(_,['Please',go,on,'.']).
```

Eliza: varianta simplă

- Se definesc diferite șabloane:

```
pattern([i,am,1],['How',long,have,you,been,1,?]).

pattern([i,like,1],['Does,anyone,else,in,your,
                                        family,like,1,?]).

pattern([i,feel|_],['Do',you,often,feel,that,way,?]).

,
```

- Trebuie să existe un șablon pentru toate celelalte cazuri:

```
pattern(_,['Please',go,on,'.']).
```

Atenție! folosim numere și nu variable pentru a identifica elementele care lipsesc deoarece acestea pot fi formate din mai mulți atomi: `very unhappy`.

Eliza: varianta simplă

- Alte tipuri de șabloane:

```
pattern(R,['What',makes,you,think,'I',2,you,?]) :-
      member(R,[[i,know,you,2,me], [i,think,you,2,me],
                [i,belive,you,2,me],[can, you, 2, me]]).

pattern(G,AG) :- member(G,[[hi],[hello]]),
                        random_select(AG,[['Hi',there,!],
['Hello',!,'How',are,you,today,?]],_).

pattern([X],['Please',you,tell,me,more,about,X]) :-
                                          important(X).

important(father). important(mother).
important(sister). important(brother).
important(son). important(daughter).
```

Eliza: varianta simplă

- apelul, transformarea atomilor de intrare în liste

```
eliza :- write('tell me'),nl,read(AInput),
         atomic_list_concat(Input,' ',AInput),
         eliza(Input).
```

- programul principal

```
eliza([bye]) :- write(['Goodbye. I hope I have helped you'])

eliza(Input) :- ... /* slide-ul urmator */
```

Eliza: varianta simplă

- programul principal

```
eliza([bye]) :- write(['Goodbye. I hope I have helped you'])

eliza(Input) :- pattern(Stimulus,Response),
                match(Stimulus,Table,Input),
                make(Response,Table,Output),
                reply(Output),
                read(AInput1),
                atomic_list_concat(Input1,' ',AInput1),
                eliza(Input1).


reply([Head|Tail]) :- write(Head), write(' '), reply(Tail).
reply([]) :- nl.
```

Eliza: varianta simplă

Observați perechea

```
match(Stimulus,Table,Input),
make(Response,Table,Output)
```

- predicatul `match` va identifica un șablon în șirul de intrare și va construi o listă de corespondențe. De exemplu, pentru

  Input: `'i am very unhappy'`
  Stimulus: `[i, am, 1]`

  se va introduce în lista de corespondențe perechea
  `nw(1,[very,unhappy]).`

Eliza: varianta simplă

Observați perechea

```
match(Stimulus,Table,Input),
make(Response,Table,Output)
```

- predicatul `match` va identifica un șablon în șirul de intrare și va construi o listă de corespondențe. De exemplu, pentru

  Input: `'i am very unhappy'`
  Stimulus: `[i, am, 1]`

  se va introduce în lista de corespondențe perechea `nw(1,[very,unhappy])`.

- perdicatul `make` va construi răspunsul corespunzător pe baza listei de corespondențe. În cazul exemplului

  Response: `['How',long,have,you,been,1,?]`
  Output: `How long have you been very unhappy?`

Eliza: varianta simplă

```
match([N|Pattern],Table,Target) :- integer(N),
            lookup(N,Table,LeftTarget), LeftTarget \== [],
                    append(LeftTarget,RightTarget,Target),
                        match(Pattern,Table,RightTarget).

match([N|Pattern],Table,Target) :- integer(N),
  append(LeftTarget,RightTarget,Target), LeftTarget \== [],
       match(Pattern,[nw(N,LeftTarget)|Table],RightTarget).

match([X],_,Target):- member(X,Target),important(X).

match([Word|Pattern],Table,[Word|Target]) :- atom(Word),
                                  match(Pattern,Table,Target).

match([],Table,[]).
```

Eliza: varianta simplă

```
make([N|Pattern],Table, Target) :- integer(N),
                                   lookup(N,Table,LeftTarget),
                                   make(Pattern,Table,RightTarget),
                                   append(LeftTarget,RightTarget,Target).

make([Word|Pattern],Table,[Word|Target]) :- atom(Word),
                                   make(Pattern,Table,Target).

make([],Table,[]).
```

- pentru `N` dat, `lookup` caută perechea `nw(N,L)` în lista de corespondențe și întoarce `L`.

Eliza: varianta simplă

```
?- eliza.
tell me
|: 'i am very upset'.
How long have you been very upset ?
|: 'two days'.
Please go on .
|: 'can you help me'.
What makes you think I help you ?
|: 'my sister told me'.
Please you tell me more about sister
|: 'i like her very much'.
Does anyone else in your family like her very much ?
|: 'yes my brother'.
Please you tell me more about brother
|: 'i like teasing him'.
Does anyone else in your family like teasing him ?
|: 'bye'.
Goodbye. I hope I have helped you
true .
```

Succes la examen!