Curs 12

2018-2019

Programare Logică

Cuprins

Bibliografie:
L.S. Sterling and E.Y. Shapiro, The Art of Prolog
`https://mitpress.mit.edu/books/art-prolog-second-edition`

Tipuri de date compuse

Termeni compuși `f(t1,..., tn)`

- Termenii sunt unitățile de bază prin care Prolog reprezintă datele.
- Sunt de 3 tipuri:
  - Constante: `23, sansa, 'Jon Snow'`
  - Variabile: `X, Stark, _house`
  - Termeni compuși:
    - predicate
    - termeni prin care reprezentăm datele

Exemplu

- `born(john, date(20,3,1977))`
  - `born/2` și `date/3` sunt *functori*
  - `born/2` este un predicat
  - `date/3` definește date compuse

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
  - `[]` este listă
  - `[X|L]` este listă, unde `X` este element și `L` este listă

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
  - `[]` este listă
  - `[X|L]` este listă, unde `X` este element și `L` este listă
- Cum definim arborii binari în Prolog?

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
  - `[]` este listă
  - `[X|L]` este listă, unde `X` este element și `L` este listă
- Cum definim arborii binari în Prolog? Soluție posibilă:

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
  - `[]` este listă
  - `[X|L]` este listă, unde `X` este element și `L` este listă
- Cum definim arborii binari în Prolog? Soluție posibilă:
  - `void` este arbore
  - `tree(X,A1,A2)` este arbore, unde `X` este un element, iar `A1` și `A2` sunt arbori

`tree(X,A1,A2)` este un termen compus, dar nu este un predicat!

Arbori binari în Prolog

- Cum arată un arbore?

Arbori binari în Prolog

- Cum arată un arbore?

  tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))

- În Prolog putem să definim:

```
def(arb, tree(a, tree(b,
                      tree(d,void,void),
                      void),
                tree(c, void,
                      tree(e,void,void)))).
```

Arbori binari în Prolog

Deoarece în Prolog nu avem declarații explicite de date, pentru a defini arborii vom scrie un predicat care este adevărat atunci când argumentul său este un arbore;

```
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :- binary_tree(Left),
                                         binary_tree(Right).
```

Eventual putem defini și un predicat pentru elemente:

```
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :- binary_tree(Left),
                                         binary_tree(Right),
                                         element_binary_tree(Element)

element_binary_tree(X):- integer(X). /* de exemplu */

test:- def(arb,T), binary_tree(T).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care verifică că un element aparține unui arbore.

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care verifică că un element aparține unui arbore.

```
tree_member(X,tree(X,Left,Right)).

tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Left).

tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Right).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care verifică că doi arbori binari sunt izomorfi.

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care verifică că doi arbori binari sunt izomorfi.

```
isotree(void,void).

isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
            isotree(Left1,Left2), isotree(Right1,Right2).

isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
             isotree(Left1,Right2), isotree(Right1,Left2).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care determină parcurgerea în preordine a unui arbore binar.

```
preorder(tree(X,L,R),Xs) :- preorder(L,Ls),
                            preorder(R,Rs),
                            append([X|Ls],Rs,Xs).
preorder(void,[]).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care determină parcurgerea în preordine a unui arbore binar.

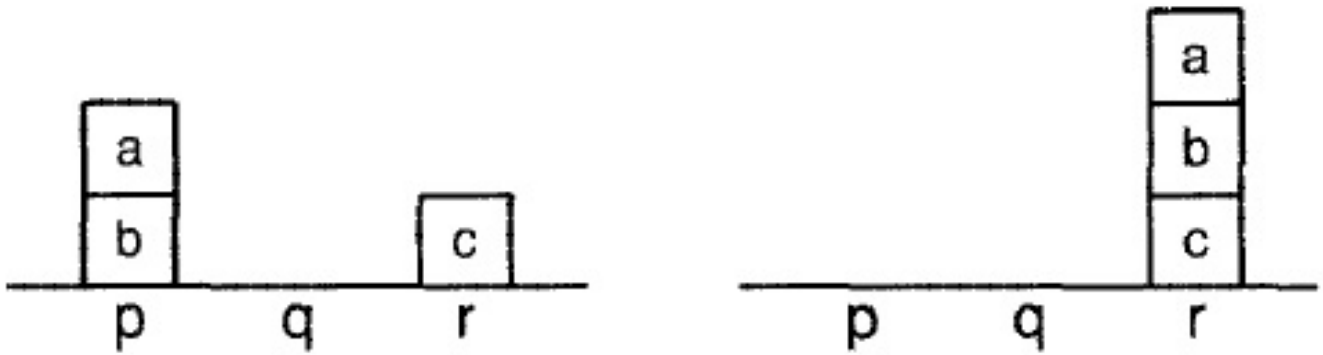
```
preorder(tree(X,L,R),Xs) :- preorder(L,Ls),
                            preorder(R,Rs),
                            append([X|Ls],Rs,Xs).
preorder(void,[]).

test(Tree,Pre):- def(arb, Tree), preorder(Tree,Pre).

?- test(T,P).
T = tree(a, tree(b, tree(d, void, void), void), tree(c,
void, tree(e, void, void))),
P = [a, b, d, c, e]
```

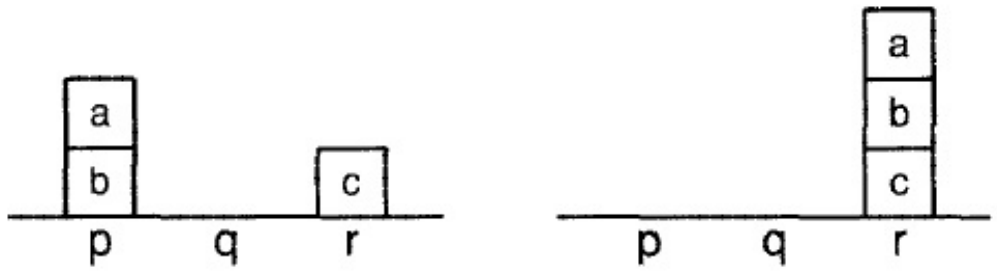
Planning în Prolog

Lumea blocurilor



- Lumea blocurilor este formată din:
  - trei blocuri: `a`,`b`, `c`
  - trei poziții: `p`,`q`, `r`
  - un bloc poate sta peste un alt bloc sau pe o poziție
- Un bloc poate fi mutat pe o poziție liberă sau pe un alt bloc.
- Problema este de a găsi un șir de mutări astfel încât dintr-o stare inițială să se ajungă într-o stare finală.

Lumea blocurilor



- Reprezentarea blocurilor, pozițiilor și a stărilor:

```
block(a). block(b). block(c).
place(p). place(q). place(r).
```

```
initial_state([on(a,b), on(b,p),on(c,r)]).
final_state([on(a,b),on(b,c),on(c,r)]).
```

Observați că `on(a,b)`, `on(b,c)`, etc. sunt date compuse.

- O stare este o listă de termenii de tipul `on(X,Y)`.
Într-o listă care reprezintă o stare, termenii `on(X,Y)` sunt ordonați după prima componentă.

Lumea blocurilor

- Predicatul `transform(State1,State2,Plan)` va genera în variabila `Plan` un șir de mutări permise care transformă starea `State1` în starea `State2`.

Lumea blocurilor

- Predicatul `transform(State1,State2,Plan)` va genera în variabila `Plan` un șir de mutări permise care transformă starea `State1` în starea `State2`.

```
transform(State1,State2,Plan) :-
                    transform(State1,State2,[State1],Plan).

transform(State,State,Visited,[],_).
transform(State1,State2,Visited,[Action|Actions]) :-
                                legal_action(Action,State1),
                                update(Action,State1,State),
                                   \+ member(State,Visited),
          transform(State,State2,[State|Visited],Actions).
```

Lumea blocurilor

- Predicatul `transform(State1,State2,Plan)` va genera în variabila `Plan` un șir de mutări permise care transformă starea `State1` în starea `State2`.

```
transform(State1,State2,Plan) :-
                    transform(State1,State2,[State1],Plan).

transform(State,State,Visited,[],_).
transform(State1,State2,Visited,[Action|Actions]) :-
                                legal_action(Action,State1),
                                update(Action,State1,State),
                                   \+ member(State,Visited),
          transform(State,State2,[State|Visited],Actions).
```

- Căutare de tip depth-first.

Lumea blocurilor

- Predicatul `legal_action(Action,State)` va instanția `Action` cu o mutare care poate fi efectuată în starea `State`. Există două mutări posibile: *mutarea pe un bloc* și *mutarea pe o poziție*.

Lumea blocurilor

- Predicatul `legal_action(Action,State)` va instanția `Action` cu o mutare care poate fi efectuată în starea `State`. Există două mutări posibile: *mutarea pe un bloc* și *mutarea pe o poziție*.

```
legal_action(to_place(Block,Y,Place),State) :-
                        block(Block), clear(Block,State),
                         place(Place), clear(Place,State).

legal_action(to_block(Block1,Y,Block2),State) :-
                      block(Block1), clear(Block1,State),
                        block(Block2), Block1 \== Block2,
                                      clear(Block2,State).


clear(X,State) :- \+ member(on(A,X),State).

on(X,Y,State) :- member(on(X,Y),State).
```

Lumea blocurilor

- Predicatul `update(Action,State,State1)` are următoarea semnificație: făcând mutarea `Action` în starea `State` se ajunge în starea `State1`.

Lumea blocurilor

- Predicatul `update(Action,State,State1)` are următoarea semnificație: făcând mutarea `Action` în starea `State` se ajunge în starea `State1`.

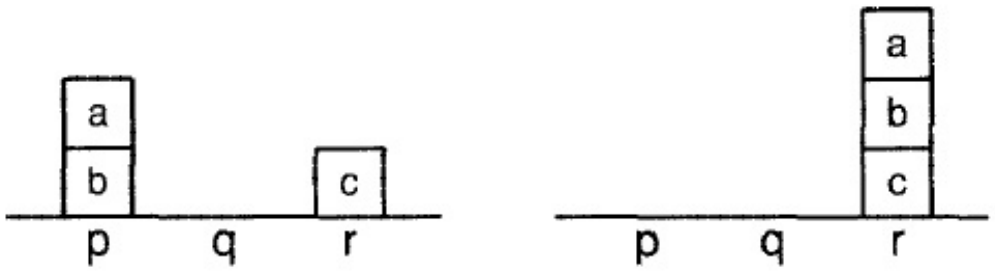
```
update(to_block(X,Y,Z),State,State1) :-
                substitute(on(X,Y),on(X,Z),State,State1).

update(to_place(X,Y,Z),State,State1) :-
                substitute(on(X,Y),on(X,Z),State,State1).

substitute(X,Y,[X|Xs],[Y|Xs]).

substitute(X,Y,[X1|Xs],[X1|Ys]) :- X \= X1,
                                   substitute(X,Y,Xs,Ys).
```

Lumea blocurilor




```
test_plan(Plan) :- initial_state(I), final_state(F),
                                    transform(I,F,Plan).
```

```
?- test(Plan).
Plan = [to_place(a, b, q), to_block(a, q, c),
to_place(b, p, q), to_place(a, c, p), to_block(a, p, b),
to_place(c, r, p), to_place(a, b, r), to_block(a, r, c),
to_place(b, q, r), to_place(a, c, q), to_block(a, q, b),
to_place(c, p, q), to_place(a, b, p), to_block(b, r, a),
to_place(c, q, r), to_block(b, a, c), to_place(a, p, q),
to_block(a, q, b)]
```

Lumea blocurilor

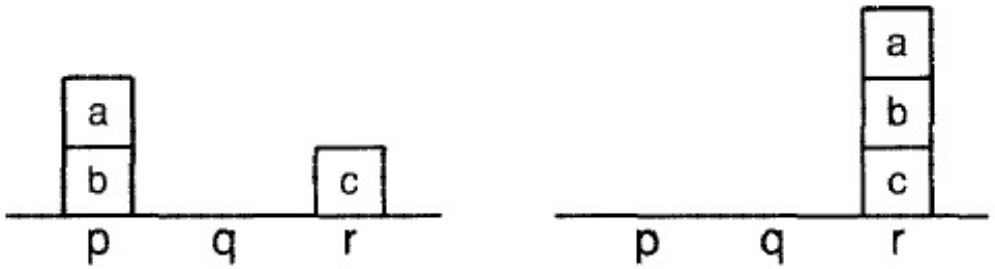
Pentru a obține o soluție mai simplă, putem limita numărul de mutări!

```
transform(State1,State2,Plan, N) :-
                    transform(State1,State2,[State1],Plan,N).


transform(State,State,Visited,[],_).

transform(State1,State2,Visited,[Action|Actions],N) :-
                                  legal_action(Action,State1),
                                  update(Action,State1,State),
        \+ member(State,Visited), length(Visited,M), M < N,
          transform(State,State2,[State|Visited],Actions,N).
```

Lumea blocurilor




```
test_plan(Plan,N) :- initial_state(I), final_state(F),
                                        transform(I,F,Plan,N).
```

```
?- test(Plan,3).
false
?- test(Plan,4).
Plan = [to_place(a, b, q), to_block(b, p, c), to_block(a, q, b)]
```

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).
```

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
  `np(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 `np(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 `np(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 .
```

Pe săptămâna viitoare!