

Η Αποκοπή και η Άρνηση

- Η στρατηγική αναζήτησης λύσεων της Prolog βασίζεται στον κανόνα "*Από αριστερά προς τα δεξιά και κατά βάθος*" (LRDF) αναζήτηση στο δένδρο υπολογισμού.
- Μπορούμε να επέμβουμε κάπως σε αυτή τη διαδικασία, αλλάζοντας τη σειρά των προτάσεων ή την σειρά των κλήσεων στις προτάσεις.
- Η Prolog προσφέρει μια ενσωματωμένη διαδικασία, την αποκοπή (**cut**) η οποία συμβολίζεται με **!** και επιδρά στη διαδικασία εύρεσης απαντήσεων της Prolog.
- Η κύρια δουλειά της είναι να μειώνει το χώρο αναζήτησης (search space) κλαδεύοντας δυναμικά (δηλ. κατά την ώρα της εκτέλεσης) το δένδρο υπολογισμού.
 - Αυτό γίνεται με την αφαίρεση εναλλακτικών κλάδων (ή κλάδων οπισθοδρόμησης) θεωρώντας ότι δεν έχουν να προσφέρουν τίποτε στην αναζήτηση της λύσης.
- Η αποκοπή προσφέρει:
 - Ταχύτερη εκτέλεση προγράμματος
 - Εξοικονόμηση μνήμης
 - Αποφυγή ατέρμονων αναζητήσεων
 - Περικοπή λύσεων που πλεονάζουν
 - Ανάπτυξη προγραμμάτων που θα ήταν αδύνατη χωρίς την αποκοπή
 - Υλοποίηση της άρνησης

62

Χρήσεις της αποκοπής

1. Τερματισμός παραγωγής εναλλακτικών λύσεων.
2. Εξαναγκασμός του συστήματος σε αποτυχία.
3. Παράλειψη ελέγχων.
4. Ορισμός δομών ελέγχου.

63

Τερματισμός παραγωγής εναλλακτικών λύσεων.

- Με την αποκοπή εμποδίζουμε να εξεταστούν για λύση κατά την οπισθοδρόμηση οι υπόλοιπες προτάσεις της διαδικασίας. (επιταχύνεται σημαντικά η εκτέλεση)
- Η αποκοπή χρησιμοποιείται με αυτή τη σημασία, όταν έχουμε μια διαδικασία μ' ένα σύνολο προτάσεων, και
 - είτε είναι γνωστόν εκ των πρότερων ότι μόνο μία πρόταση μπορεί να εφαρμοστεί κάθε φορά,
 - είτε δε χρειαζόμαστε άλλη εναλλακτική λύση.

Παράδειγμα (βιβλίου): Συγγώνευση δύο διατεταγμένων κατ' αύξουσα σειρά λιστών ακέραιων αριθμών.

```
merge ( [X|XR] , [Y|YR] , [X|ZR] ) :-  
    X < Y, !,  
    merge (XR, [Y|YR] , ZR) .  
merge ( [X|XR] , [Y|YR] , [Y|ZR] ) :-  
    X > Y, !,  
    merge ( [X|XR] , YR, ZR) .  
merge ( [X|XR] , [X|YR] , [X,X|ZR] ) :- !,  
    merge (XR, YR, ZR) .  
merge (XR, [], XR) :- !.  
merge ( [], YR, YR) :- !.
```

Η επιλογή της κατάλληλης πρότασης γίνεται με τη βοήθεια κάποιων ελέγχων πάνω στις τιμές των παραμέτρων.

64

Εξαναγκασμός του συστήματος σε αποτυχία

- Με την αποκοπή και το κατηγορημα **fail** μπορούμε να εξαναγκάσουμε το μηχανισμό της Prolog να αποτύχει, δηλαδή να επιστρέψει **no**, όταν το θελήσουμε εμείς.

```
not_member (X,L) :- member (X,L) , !, fail.  
not_member (X,L) .
```

```
?- not_member (a, [b,a,c,d]) .  
no  
?- not_member (a, [b,q,c,d]) .  
yes
```

- Γενικότερα, η άρνηση μέσω αποτυχίας για οποιοδήποτε κατηγορημα επιτυγχάνεται ως εξής:

```
not(Goal) :- Goal, !, fail.  
not(_).
```

65

Παράλειψη Ελέγχων

- Με την αποκοπή μπορούμε να παραλείψουμε κάποιους επιπλέον ελέγχους σε κατηγορήματα με πολλές περιπτώσεις-κανόνες.
- Εξασφαλίζεται έτσι η σωστή λύση σε περίπτωση οπισθοδρόμησης.
- Να ορισθεί η σχέση **sign** (πρόσημο) που να επιστρέφει την ένδειξη **positive**, **negative**, **zero** ανάλογα με την τιμή της παραμέτρου.

(Σημείωση: Να γίνει με και χωρίς αποκοπή).

(χωρίς αποκοπή)	<pre>?- sign(5,A). A = positive; no</pre>	
(με αποκοπή)	<pre>?- sign(5,A). A = positive; no</pre>	<pre>Αν σβήσουμε τα cut ?- sign(5,A). A = positive; A = negative; no</pre>

66

Ορισμός δομών ελέγχου

- Με το cut μπορούμε να ελέγξουμε τη ροή εκτέλεσης ενός προγράμματος.
- Να οριστεί το κατηγορήμα **if_then_else(Condition,Then,Else)**, το οποίο ελέγχει τη συνθήκη **Condition** και αν αληθεύει εκτελεί το στόχο **Then**, αλλιώς εκτελεί το στόχο **Else**.

```
if_then_else(Condition,Then,Else) :- Condition, !, Then.
```

```
if_then_else(Condition,Then,Else) :- Else.
```

```
?- if_then_else(5=5,write(ok),write(wrong)).
```

```
ok
```

```
?- if_then_else(5=6,write(ok),write(wrong)).
```

```
wrong
```

- Να οριστεί το κατηγορήμα **max(X,Y,Max)**, το οποίο επιστρέφει το μέγιστο από τα **X**, **Y** στο όρισμα **Max**.

```
max(X,Y,Max) :- if_then_else(X>Y,Max=X,Max=Y).
```

```
?- max(4,7,A).
```

```
A = 7
```

67

Ορισμός δομών ελέγχου

- Να οριστεί το κατηγορήμα **case(ListOfConditions,ListOfActions,Otherwise)**, το οποίο να υλοποιεί τη δομή ελέγχου **case** των συμβατικών γλωσσών προγραμματισμού.

```
case([],[],Otherwise) :- Otherwise.
```

```
case([Condition|_],[Action|_],_) :-
```

```
Condition, !, Action.
```

```
case([_|RestConditions],[_|RestActions],Otherwise) :-
```

```
case(RestConditions,RestActions,Otherwise).
```

```
menu :-
```

```
read(X),
```

```
case( [X=1,X=2,X=3],
```

```
[write('1st choice'),write('2nd choice'),write('3rd choice')],
```

```
write('Out of Range')), nl.
```

<pre> ?- menu. : 2. 2nd choice yes</pre>	<pre> ?- menu. : 5. Out of Range yes</pre>
--	--

68

Προβλήματα με την αποκοπή

- Το σοβαρότερο πρόβλημα της αποκοπής είναι ότι πολλές φορές καταστρέφει την ισοδυναμία μεταξύ δηλωτικής και διαδικαστικής ερμηνείας των Prolog προγραμμάτων.

- Για παράδειγμα, ο ορισμός του **min**:

```
α) min(A1,A2,A1) :- A1 < A2, !.
```

(Πράσινο cut)

```
min(A1,A2,A2) :- A1 >= A2.
```

εκτελείται σωστά και αν αλλάζουμε τη σειρά των προτάσεων, ενώ δεν ισχύει αυτό για τον επόμενο ορισμό:

```
β) min(A1,A2 A1) :- A1 < A2, !.
```

(Κόκκινο cut)

```
min(A1,A2,A2).
```

- Πράσινη αποκοπή (green cut) όταν δεν αλλάζει η δηλωτική ερμηνεία του προγράμματος
 - π.χ. ο (α) ορισμός του **min**, κατηγορήμα **merge**
- Κόκκινη αποκοπή (red cut) όταν αλλάζει τη δηλωτική ερμηνεία
 - π.χ. ο (β) ορισμός του **min**.

69

Προβλήματα με την άρνηση

- Ο ορισμός της άρνησης δεν αντιστοιχεί επακριβώς στην άρνηση της μαθηματικής λογικής.
- Η άρνηση μιας πρότασης δε σημαίνει πράγματι ότι δεν ισχύει αλλά ότι το σύστημα με βάση τις πληροφορίες που περιέχει δεν μπορεί να αποδείξει ότι ισχύει.

Παράδειγμα

```
programmer(mary).
male(nick).
female(X) :- not(male(X)).

?- programmer(X), female(X).
X = mary

?- female(X), programmer(X).
No
```

- Αυτό συμβαίνει όταν τα negated goals δεν είναι ground όταν εκτελούνται
- Δηλαδή υπάρχει πρόβλημα
- Επιλύεται με αναδιάταξη των προτάσεων ή των goals.

70

Παραδείγματα χρήσης αποκοπής

- `member(E, [E|_]) :- !.` (προσθέτω cut)
- `member(E, [_|T]) :- member(E, T).`

<code>member</code> χωρίς αποκοπή	<code>member</code> με αποκοπή
<pre>?- member(X, [a,b,c]). X = a; X = b; X = c; no</pre>	<pre>?- member(X, [a,b,c]). X = a; no</pre>

- `append([], L, L) :- !.` (προσθέτω cut)
- `append([H|L1], L2, [H|L]) :- append(L1, L2, L).`

<code>append</code> χωρίς αποκοπή	<code>append</code> με αποκοπή
<pre>? - append(X,Y, [1,2]). X = [], Y = [1,2]; X = [1], Y = [2]; X = [1,2], Y = [] ; no</pre>	<pre>?- append(X,Y, [1,2]). X = [], Y = [1,2]; no</pre>

71

Παραδείγματα Αποκοπής

- Δίνεται το ακόλουθο Prolog πρόγραμμα:
- ```
p(1).
p(2) :- !.
p(3).
```
- Ποιες είναι οι απαντήσεις στις ακόλουθες ερωτήσεις;
- `?-p(X).`
  - `?-p(X), p(Y).`
  - `?-p(X), !, p(Y).`

#### Απάντηση:

|                                                              |                                                                                                                            |                                                                        |
|--------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|
| α) <code>X=1;</code><br><code>X=2;</code><br><code>No</code> | β) <code>X=1, Y=1;</code><br><code>X=1, Y=2;</code><br><code>X=2, Y=1;</code><br><code>X=2, Y=2;</code><br><code>No</code> | γ) <code>X=1, Y=1;</code><br><code>X=1, Y=2;</code><br><code>No</code> |
|--------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|

72

### Ασκήσεις με αποκοπή

- Να ορισθεί η σχέση `split(Numbers,Positives,Negatives)` που χωρίζει μια λίστα αριθμών σε 2 κατηγορίες: Θετικούς (και το 0) και αρνητικούς. Να δοθούν 2 λύσεις : με και χωρίς αποκοπή.
- π.χ. `?- split([3,-1,0,5,-2],[3,0,5],[-1,-2]).`  
`yes`

```
split([],[],[]).
split([X|L],[X|L1],L2) :- X >= 0, !, split(L,L1,L2).
split([X|L],L1,[X|L2]) :- split(L,L1,L2)
```

χωρίς cut :

- Η 3<sup>η</sup> πρόταση:
- ```
split([X|L],L1,[X|L2]) :- X < 0, split(L,L1,L2).
```

73

Ασκήσεις με αποκοπή

```
del_all(_, [], []).
del_all(X, [X|T], Res) :-
    del_all(X, T, Res).
del_all(X, [H|T], [H|Res]) :-
    X \= H,
    del_all(X, T, Res).
```

```
del_all(_, [], []).
del_all(X, [X|T], Res) :- !,
    del_all(X, T, Res).
del_all(X, [H|T], [H|Res]) :-
    del_all(X, T, Res).
```

Red cut

```
del_all(_, [], []).
del_all(X, [X|T], Res) :- !,
    del_all(X, T, Res).
del_all(X, [H|T], [H|Res]) :-
    X \= H,
    del_all(X, T, Res).
```

Green cut

74

Ασκήσεις με αποκοπή

- Να ορισθεί η σχέση 'διαφορά συνόλων' (τα στοιχεία του 1^{ου} συνόλου που δεν ανήκουν στο 2^ο).

π.χ. `?- dif([a,b,c,d],[b,d,e,f],[a,c]).`

yes

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

```
dif([X|L1], L2, L) :- member(X, L2), !, dif(L1, L2, L). (αγνοεί το X αν ανήκει και στο L2)
```

```
dif([X|L1], L2, [X|L]) :- dif(L1, L2, L).
```

- Η 3^η κλήση μπορεί να γραφεί ως εξής :

```
dif([X|L1], L2, [X|L]) :- not(member(X, L2)), dif(L1, L2, L).
```

- Έτσι δε χρειάζεται αποκοπή (!) στη 2^η κλήση.

- Παρ' όλα αυτά η **dif** ΔΕΝ μπορεί να 'δουλέψει' σωστά χωρίς cut (!). ΓΙΑΤΙ ??

75

Ασκήσεις με αποκοπή

```
union([], L2, L2).
union([H|T], L2, Res) :-
    member(H, L2),
    union(T, L2, Res).
union([H|T], L2, [H|Res]) :-
    not_member(H, List2),
    union(T, L2, Res).
```

```
union([], L2, L2).
union([H|T], L2, Res) :-
    member(H, L2), !,
    union(T, L2, Res).
union([H|T], L2, [H|Res]) :-
    not(member(H, List2)),
    union(T, L2, Res).
```

Green cut

```
union([], L2, L2) :- !.
union([H|T], L2, Res) :-
    member(H, L2), !,
    union(T, L2, Res).
union([H|T], L2, [H|Res]) :-
    union(T, L2, Res).
```

Red cut

76

Παράδειγμα (βιβλίου)

- Να ορισθεί μια διαδικασία **count** που θα μετρά πόσες φορές εμφανίζεται ένα στοιχείο (άτομο) σε μια λίστα.

count(A, L, N) **A**: άτομο, **L**: λίστα, **N**: αριθμός εμφανίσεων

```
count(_, [], 0).
```

```
count(A, [A|L], N) :- !, count(A, L, N1), N is N1 + 1.
```

```
count(A, [_|L], N) :- count(A, L, N).
```

```
?- count(a, [a,b,c,a,d], N).
N = 2
```

```
?- count(a, [a,X,b,a,Y,d], N).
```

```
X = a
```

```
Y = a
```

```
N = 4
```

Τα **X** και **Y** της ερώτησης κατά την εκτέλεση με την 2^η πρόταση ταυτοποιούνται με **a**.

Λύση:

- Η 2^η πρόταση γίνεται:

```
count(A, [B|L], N) :- nonvar(B), A = B, !, count(A, L, N1), N is N1 + 1.
```

```
?- count(a, [a,b,c,a,d], N).
N = 2
```

```
?- count(a, [a,X,b,a,Y,d], N).
```

```
X = _
```

```
Y = _
```

```
N = 2
```

Χωρίς τιμή

77