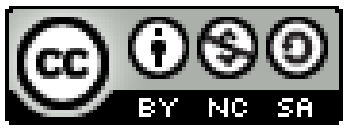
ΑΡΙΣΤΟΤΕΛΕΙΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΟΝΙΚΗΣ

ΑΝΟΙΚΤΑ
ΑΚΑΔΗΜΑΪΚΑ
ΜΑΘΗΜΑΤΑ

Υπολογιστική Λογική και Λογικός Προγραμματισμός

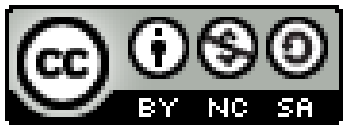
Ενότητα 7: Γλώσσα Prolog: Έλεγχος εκτέλεσης προγράμματος, Αποκοπή, Άρνηση

Νίκος Βασιλειάδης, Αναπλ. Καθηγητής
Τμήμα Πληροφορικής



Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται σε άδειες χρήσης Creative Commons.
- Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.



Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «Ανοικτά Ακαδημαϊκά Μαθήματα στο Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης» έχει χρηματοδοτήσει μόνο την αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

ΑΡΙΣΤΟΤΕΛΕΙΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΟΝΙΚΗΣ

ΑΝΟΙΚΤΑ
ΑΚΑΔΗΜΑΪΚΑ
ΜΑΘΗΜΑΤΑ

Γλώσσα Prolog: Έλεγχος εκτέλεσης προγράμματος, Αποκοπή, Άρνηση.

Αποκοπή

- 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.

Χρησιμότητα Αποκοπής

- Η κύρια δουλειά της αποκοπής είναι να μειώνει το χώρο αναζήτησης (search space) κλαδεύοντας δυναμικά (δηλ. κατά την ώρα της εκτέλεσης) το δένδρο υπολογισμού.
- Είναι ένας τρόπος ελέγχου του μηχανισμού οπισθοδρόμησης, «απαγορεύοντας» τη δημιουργία κλαδιών τα οποία γνωρίζουμε ότι δεν θα οδηγήσουν σε λύση.

Λειτουργία Αποκοπής (1/2)

- Η αποκοπή είναι ένα ενσωματωμένο κατηγόρημα, συμβολίζεται με το θαυμαστικό "!" και πετυχαίνει πάντα.
- Όταν ο μηχανισμός αναζήτησης συναντήσει αποκοπή τότε:
  - Αγνοούνται όλες οι προτάσεις του ίδιου κατηγορήματος με την πρόταση και που περιέχει την αποκοπή και έπονται αυτής.
    - Αγνοούνται όλοι οι εναλλακτικοί τρόποι ικανοποίησης της κλήσης που αντιστοιχεί στην διαδικασία που περιέχει την αποκοπή.
  - Αγνοούνται όλες οι εναλλακτικές λύσεις των ατομικών τύπων (κλήσεων) οι οποίοι βρίσκονται πριν από την αποκοπή μέσα στο σώμα του κανόνα στον οποίο εμφανίζεται.

Λειτουργία Αποκοπής (2/2)

?- p.

p_1 :- …. .

p_2 :- b_1, b_2, …, b_k, !, b_{k+1}, …, b_n.

p_3 :- … .

…

p_n :- … .

- Όταν εκτελεστεί η αποκοπή, «κόβονται» όσες εναλλακτικές λύσεις έχουν οι κλήσεις b_1, b_2, …, b_k.
- Επίσης, οι προτάσεις p_3, …, p_n, είναι σα να μην υπάρχουν.
- Η αποκοπή **δεν έχει καμία επίπτωση** στις κλήσεις b_{k+1}, …, b_n, καθώς και στην πρόταση p_1.

Παραδείγματα Αποκοπής (1/2)

- 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).
X=1; X=2; No	X=1, Y=1; X=1, Y=2; X=2, Y=1; X=2, Y=2; No	X=1, Y=1; X=1, Y=2; No

Παραδείγματα Αποκοπής (2/2)

- Τι θα γινόταν αν δεν υπήρχε η αποκοπή στο πρόγραμμα;

p(1). p(2). p(3).

?-p(X).	?-p(X), p(Y).	?-p(X), !, p(Y).
X=1; X=2; X=3; No	X=1, Y=1; X=1, Y=2; X=1, Y=3; X=2, Y=1; X=2, Y=2; X=2, Y=3; X=3, Y=1; X=3, Y=2; X=3, Y=3; No	X=1, Y=1; X=1, Y=2; X=1, Y=3; No

Επίπτωση Αποκοπής στο Δένδρο Εκτέλεσης

?- a.

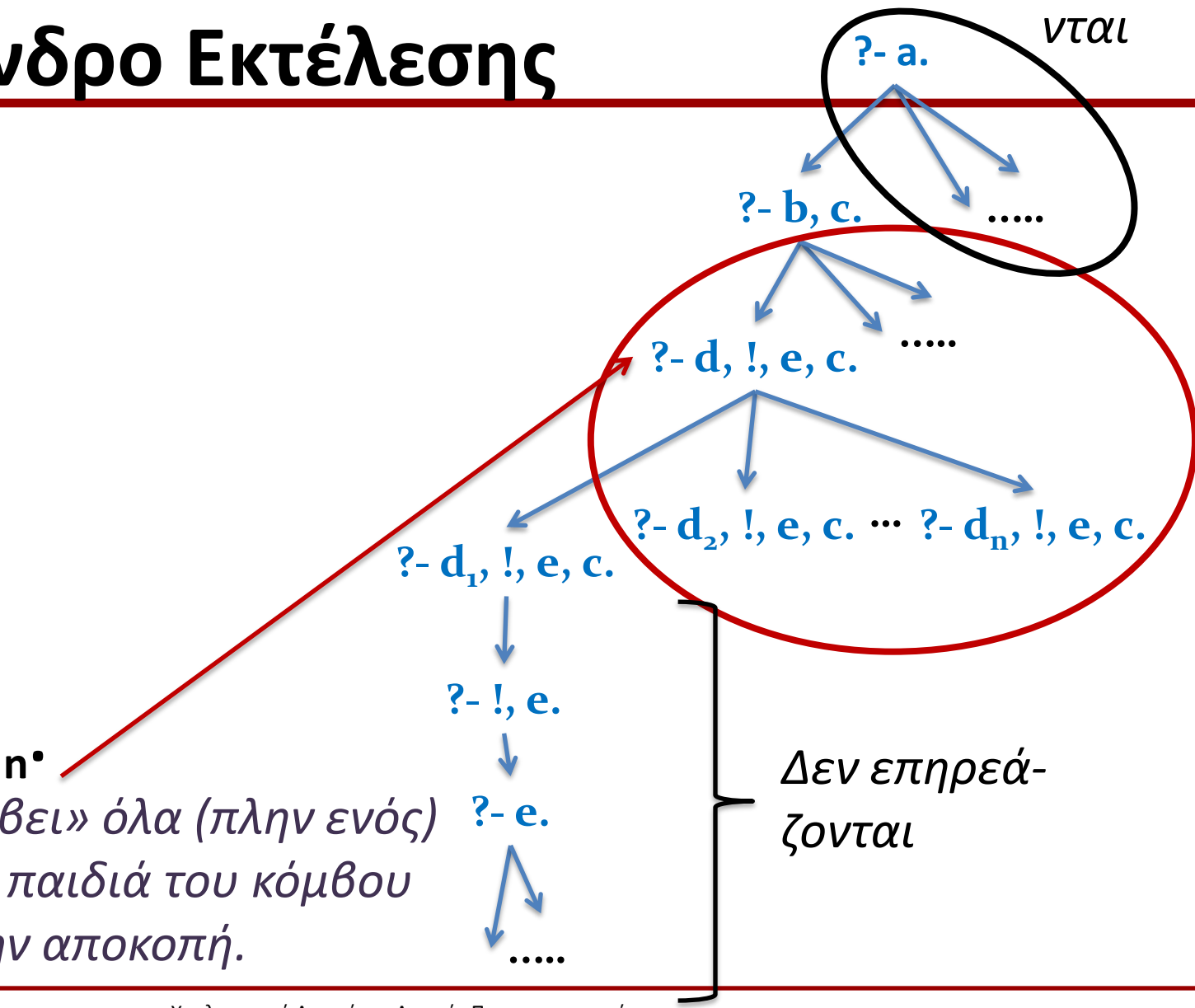
a :- b, c.

. . . . . .

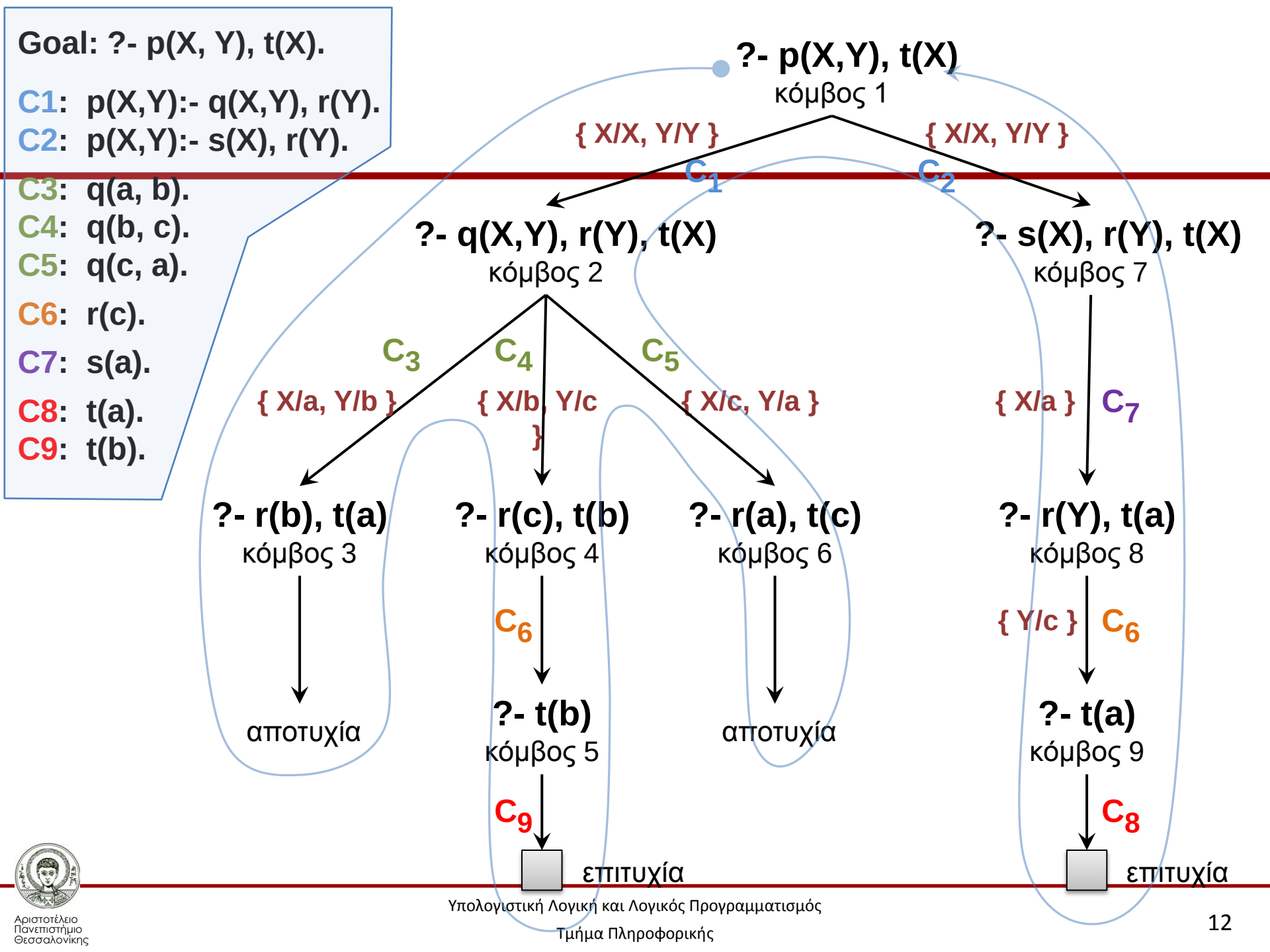
b :- d, !, e.

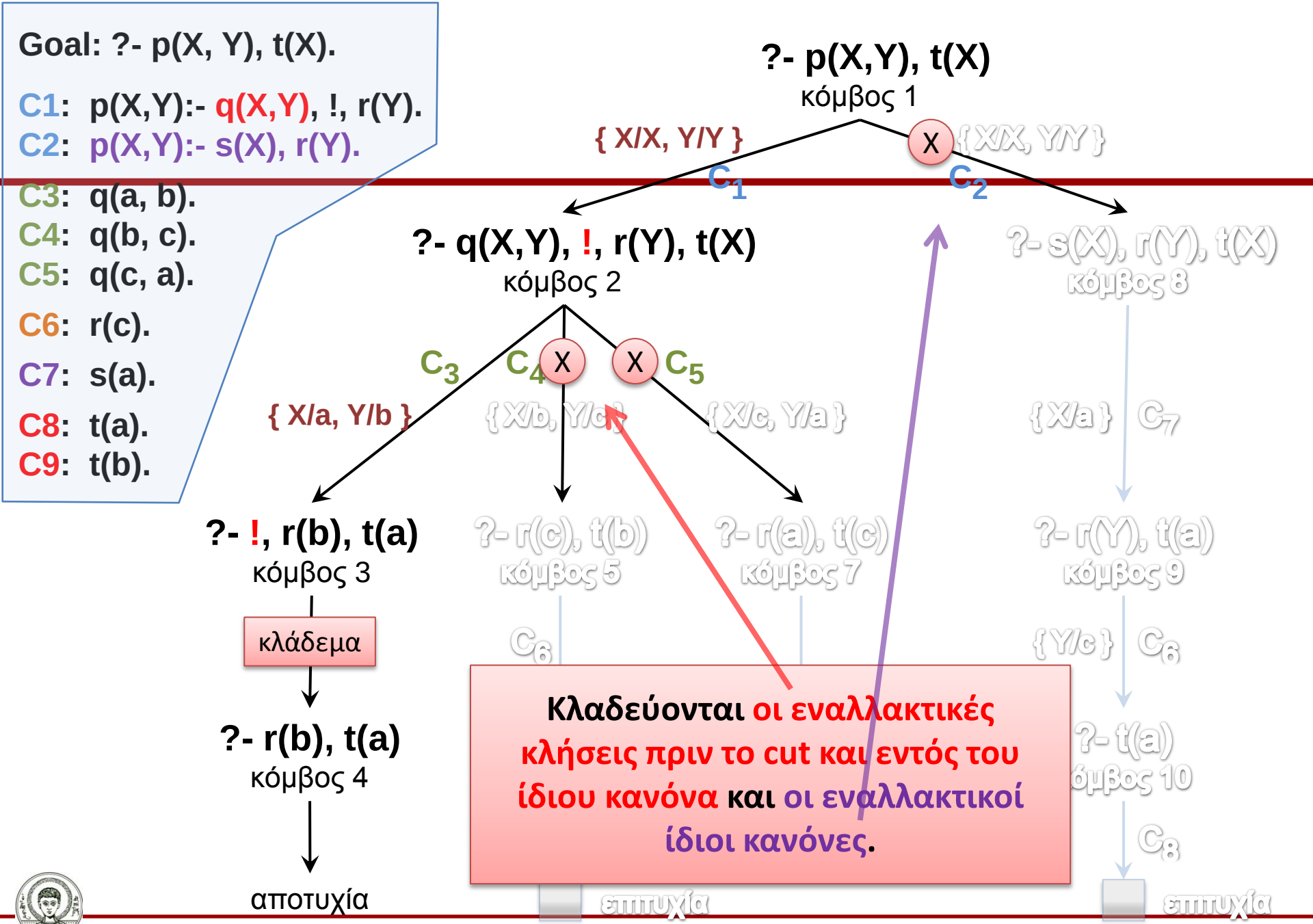
. . . . . .

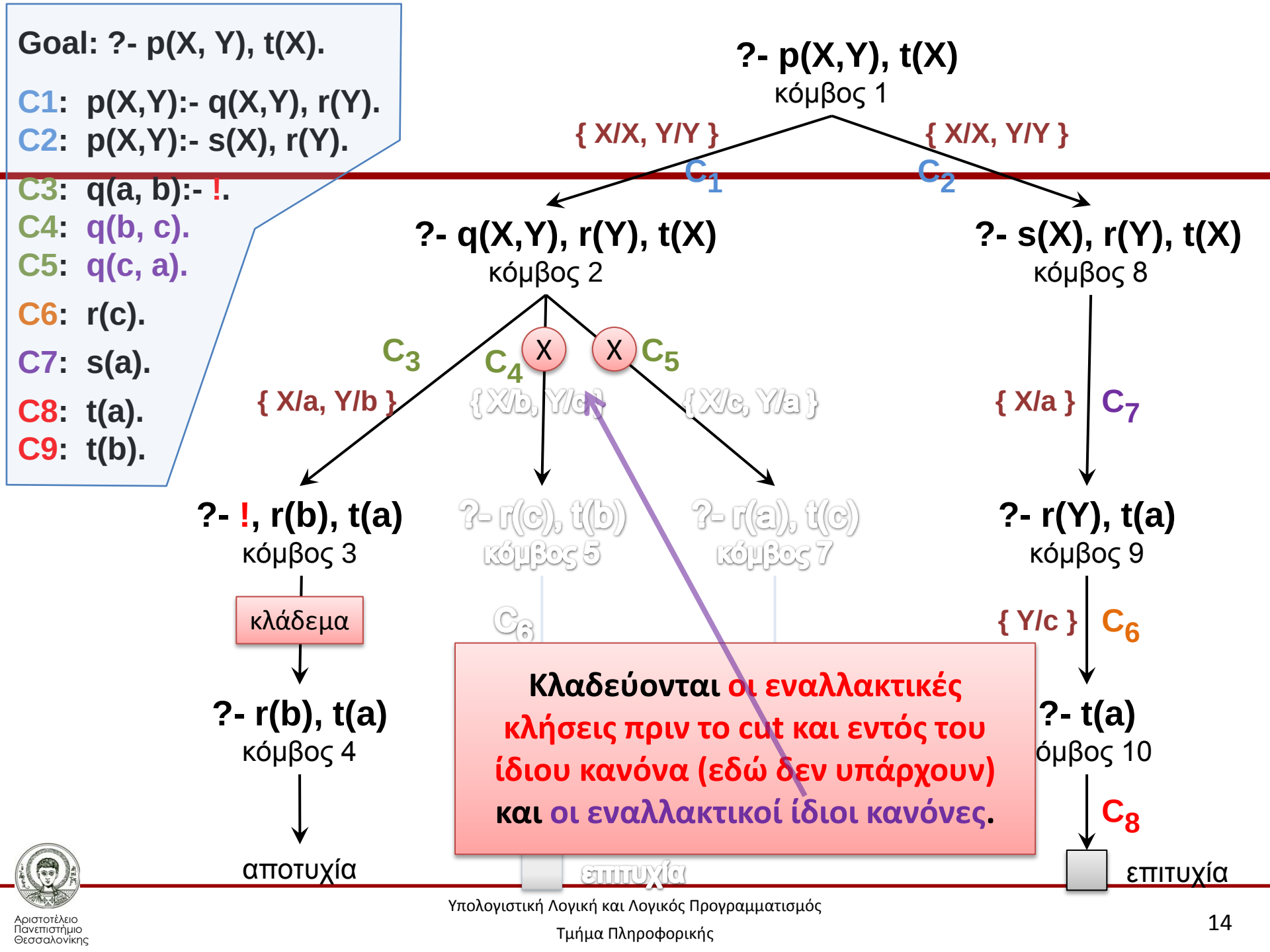
d_1. d_2. … d_n.



Η αποκοπή «κόβει» όλα (πλην ενός) τα αδέλφια και παιδιά του κόμβου που περιέχει την αποκοπή.


Goal: ?- p(X, Y), t(X).
C1: p(X,Y):- q(X,Y), r(Y).
C2: p(X,Y):- s(X), r(Y).
C3: q(a, b).
C4: q(b, c).
C5: q(c, a).
C6: r(c).
C7: s(a).
C8: t(a).
C9: t(b).
?- p(X,Y), t(X)
κόμβος 1
{ X/X, Y/Y }
C1
{ X/X, Y/Y }
C2
?- q(X,Y), r(Y), t(X)
κόμβος 2
?- s(X), r(Y), t(X)
κόμβος 7
C3
{ X/a, Y/b }
C4
{ X/b, Y/c }
C5
{ X/c, Y/a }
{ X/a }
C7
?- r(b), t(a)
κόμβος 3
?- r(c), t(b)
κόμβος 4
?- r(a), t(c)
κόμβος 6
?- r(Y), t(a)
κόμβος 8
αποτυχία
C6
αποτυχία
{ Y/c }
C6
?- t(b)
κόμβος 5
?- t(a)
κόμβος 9
C9
C8
επιτυχία
επιτυχία


Goal: ?- p(X, Y), t(X).
C1: p(X,Y):- q(X,Y), !, r(Y).
C2: p(X,Y):- s(X), r(Y).
C3: q(a, b).
C4: q(b, c).
C5: q(c, a).
C6: r(c).
C7: s(a).
C8: t(a).
C9: t(b).
?- p(X,Y), t(X)
κόμβος 1
{ X/X, Y/Y }
C1
C2
{ X/X, Y/Y }
?- q(X,Y), !, r(Y), t(X)
κόμβος 2
C3
C4
C5
{ X/a, Y/b }
{ X/b, Y/c }
{ X/c, Y/a }
?- !, r(b), t(a)
κόμβος 3
κλάδεμα
?- r(b), t(a)
κόμβος 4
αποτυχία
?- r(c), t(b)
κόμβος 5
C6
?- r(a), t(c)
κόμβος 7
?- s(X), r(Y), t(X)
κόμβος 8
{ X/a }
C7
?- r(Y), t(a)
κόμβος 9
{ Y/c }
C6
?- t(a)
κόμβος 10
C8
επιτυχία
επιτυχία
Κλαδεύονται οι εναλλακτικές κλήσεις πριν το cut και εντός του ίδιου κανόνα και οι εναλλακτικοί ίδιοι κανόνες.


Goal: ?- p(X, Y), t(X).
C1: p(X,Y):- q(X,Y), r(Y).
C2: p(X,Y):- s(X), r(Y).
C3: q(a, b):- !.
C4: q(b, c).
C5: q(c, a).
C6: r(c).
C7: s(a).
C8: t(a).
C9: t(b).
?- p(X,Y), t(X)
κόμβος 1
{ X/X, Y/Y }
C1
{ X/X, Y/Y }
C2
?- q(X,Y), r(Y), t(X)
κόμβος 2
?- s(X), r(Y), t(X)
κόμβος 8
C3
{ X/a, Y/b }
C4
{ X/b, Y/c }
X
X
C5
{ X/c, Y/a }
{ X/a }
C7
?- !, r(b), t(a)
κόμβος 3
?- r(c), t(b)
κόμβος 5
?- r(a), t(c)
κόμβος 7
?- r(Y), t(a)
κόμβος 9
κλάδεμα
C6
{ Y/c }
C6
?- r(b), t(a)
κόμβος 4
?- t(a)
κόμβος 10
C8
αποτυχία
επιτυχία
επιτυχία
Κλαδεύονται οι εναλλακτικές κλήσεις πριν το cut και εντός του ίδιου κανόνα (εδώ δεν υπάρχουν) και οι εναλλακτικοί ίδιοι κανόνες.

Πλεονεκτήματα – Χρήσεις Αποκοπής

- Πλεονεκτήματα.
  - Ταχύτερη εκτέλεση προγράμματος.
  - Εξοικονόμηση μνήμης.
  - Αποφυγή ατέρμονων αναζητήσεων.
  - Περικοπή λύσεων που πλεονάζουν.
  - Ανάπτυξη προγραμμάτων που θα ήταν αδύνατα χωρίς αυτήν.
  - Υλοποίηση της άρνησης.
- Χρήσεις.
  - Παράλειψη άσκοπων ελέγχων.
  - Τερματισμός παραγωγής εναλλακτικών λύσεων.
  - Εξαναγκασμός του συστήματος σε αποτυχία.
  - Ορισμός δομών ελέγχου.

Παράλειψη άσκοπων ελέγχων (1/3)

- Η αποκοπή μπορεί να χρησιμοποιηθεί για την αποφυγή άσκοπων ελέγχων κατά την οπισθοδρόμηση, όταν γνωρίζουμε ότι μόνο μια πρόταση από το κατηγόρημα μπορεί να επιτύχει σε κάθε δεδομένη χρονική στιγμή.
  - Στο παράδειγμα αποφεύγονται οι περιττοί έλεγχοι κατά την οπισθοδρόμηση αφού γνωρίζουμε εκ των προτέρων ότι μόνο μια από τις τρεις προτάσεις του προγράμματος μπορεί να είναι αληθής κάθε φορά.

```
sign(X, positive) :- X > 0, !.
sign(X, zero) :- X =:= 0, !.
sign(X, negative) :- X < 0.
```

```
?- sign(5,A).
A = positive;
no
?- sign(-5,A).
A = negative
```

- *Για να απαντηθεί το **no** δεν εκτελείται κανένας έλεγχος.*
- *Χωρίς cut θα εκτελούνταν άλλοι 2 έλεγχοι.*

Παράλειψη άσκοπων ελέγχων (2/3)

- Χρησιμοποιώντας την αποκοπή μπορούμε ακόμη και να παραλείψουμε συνθήκες.
- Έτσι επιταχύνεται η εκτέλεση των προγραμμάτων καθώς εκτελούνται λιγότερες εντολές.
- Το προηγούμενο παράδειγμα μπορεί να γραφεί (χωρίς να αλλάξουν τα αποτελέσματα που επιστρέφει):

```
sign(X, positive) :- X > 0, !.
sign(X, zero) :- X =:= 0, !.
sign(X, negative).
```

```
?- sign(5,A).
A = positive;
no
?- sign(-5,A).
A = negative
```

*Για να απαντηθεί το **negative** εκτελούνται 2 έλεγχοι και όχι 3 (όπως στο προηγούμενο παράδειγμα).*

Παράλειψη άσκοπων ελέγχων (3/3)

- Γιατί είναι απαραίτητη η αποκοπή για να παραλείψουμε συνθήκες;
- *Η σειριακότητα της Prolog δεν αρκεί?*

sign(X, positive) :- X > 0. **sign(X, zero) :- X =:= 0.** **sign(X, negative).**	**?- sign(-5,A).** **A = negative** **?- sign(5,A).** **A = positive;** **A = negative;** **no**

*Για να απαντηθεί το **negative** εκτελούνται 2 έλεγχοι.*

Το κατηγόρημα επιστρέφει επιπλέον λύσεις (ΛΑΘΟΣ) στην οπισθοδρόμηση.

Τερματισμός παραγωγής εναλλακτικών λύσεων (1/4)

- Το **member/2** επιστρέφει όλα τα στοιχεία μιας λίστας αν το πρώτο όρισμα είναι μεταβλητή.

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
?- member(X,[a,b,c]).
X=a;
X=b;
X=c;
no
```

Τερματισμός παραγωγής εναλλακτικών λύσεων (2/4)

- Έστω ότι θέλω το **member/2** να επιστρέφει ένα μόνο στοιχείο της λίστας (π.χ. το πρώτο) αν το πρώτο όρισμα είναι μεταβλητή.

```
member(X,[X|_]) :- !.
member(X,[_|T]) :- member(X,T).
?- member(X,[a,b,c]).
X=a;
no
```

Τερματισμός παραγωγής εναλλακτικών λύσεων (3/4)

- Αν βάλω το cut στον δεύτερο κανόνα του **member** τι θα συμβεί;

member(X,[X|_]).

Ο πρώτος κανόνας μπορεί να έχει εναλλακτικές λύσεις.

member(X,[_|T]) :- member(X,T), !.

?- member(X,[a,b,c]).

X=a;

X=b;

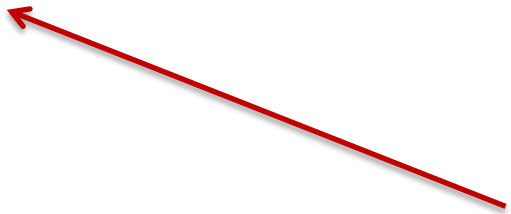
no

Αφού εκτελεστεί η πρώτη αναδρομική κλήση και δοθεί η δεύτερη απάντηση, το cut κόβει τις εναλλακτικές κλήσεις της.

Τερματισμός παραγωγής εναλλακτικών λύσεων (4/4)

- Αν βάλω το cut στον δεύτερο κανόνα του **member** (πιο αριστερά) τι θα συμβεί;

```
member(X,[X|_]).
member(X,[_|T]) :- !, member(X,T).
?- member(X,[a,b,c]).
X=a;
X=b;
X=c;
no
```



Το cut δεν κόβει τίποτε, αφού δεν υπάρχει αριστερότερη κλήση, ούτε πιο κάτω κανόνας.

Εξαναγκασμός σε αποτυχία

- Με την αποκοπή και το κατηγόρημα **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
```

Άρνηση στην Prolog ως αποτυχία (1/3)

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

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

not(_).

- Το κατηγόρημα **not** δέχεται ως όρισμα μια οποιαδήποτε κλήση της Prolog.
  - Αν η κλήση μπορεί να αποδειχθεί τότε το κατηγόρημα αποτυγχάνει.
  - Αν όχι τότε πετυχαίνει.

Άρνηση στην Prolog ως αποτυχία (2/3)

?- not(member(a,[b,a,c,d])). no	?- not(last(a,[c,b,a])). no
?- not(member(a,[b,q,c,d])). yes	?- not(last(d,[c,b,a])). yes

Άρνηση στην Prolog ως αποτυχία (3/4)

- Μία από τις σημαντικότερες διαφορές της Prolog από την κατηγορηματική λογική είναι η σημασία της *άρνησης*.
- Στην κατηγορηματική λογική μπορούν να υπάρξουν αρνητικά γεγονότα.
  - Ατομικοί τύποι για τους οποίους δηλώνεται ρητά ότι είναι ψευδείς.
- Στην Prolog υπάρχει η δυνατότητα αναπαράστασης μόνο θετικής γνώσης.
  - Η ύπαρξη κάποιου γεγονότος δηλώνει την αλήθεια του.
  - Θεωρούμε πως ό,τι δεν αναφέρεται ρητά στην μνήμη της Prolog, τότε δεν ισχύει.
- Αυτό ονομάζεται "*υπόθεση του κλειστού κόσμου*" (*closed-world assumption*).
  - Δίνει τη δυνατότητα ύπαρξης ενός είδους περιορισμένης άρνησης, της "*άρνησης ως αποτυχίας*" (*negation-as-failure*).

Άρνηση στην Prolog ως αποτυχία (4/4)

- Σύμφωνα με την *υπόθεση του κλειστού κόσμου*, οποιαδήποτε σχέση δεν μπορεί να αποδειχθεί από το σύστημα θεωρείται ως ψευδής.
- Η υλοποίηση αυτή της άρνησης διαφέρει σαφώς από την κλασσική έννοια της άρνησης στην λογική.
  - Χρησιμοποιήθηκε καθώς απαλλάσσει τον προγραμματιστή από την υποχρέωση του ορισμού όλης την αρνητικής πληροφορίας για κάποια εφαρμογή.
  - Χρησιμοποιείται ευρέως και σε άλλα συστήματα, όπως π.χ. οι Βάσεις Δεδομένων.

Προβλήματα με την άρνηση

```
programmer(mary).

male(nick).

female(X) :- not(male(X)).
```

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

X = mary

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

No

- *Υπάρχει πρόβλημα όταν οι κλήσεις που βρίσκονται μέσα στην άρνηση δεν έχουν μεταβλητές με τιμές όταν εκτελούνται*
- *Η άρνηση ΔΕΝ μπορεί να δώσει τιμές στις μεταβλητές*
- *Επιλύεται με αναδιάταξη των προτάσεων ή των κλήσεων.*

Ορισμός δομών ελέγχου

- Με το cut μπορούμε να ελέγξουμε τη ροή εκτέλεσης ενός προγράμματος.
- Κατηγόρημα **if_then_else(Condition,Then,Else)**
  - αν αληθεύει η συνθήκη **Condition** εκτελείται ο στόχος **Then.**
  - αλλιώς εκτελείται ο στόχος **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
```

Παράδειγμα Δομών Ελέγχου

- Να οριστεί το κατηγόρημα **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(X,Y,Max) :- (X>Y -> Max=X ; Max=Y).

?- max(4,7,A).

A = 7

?- max(8,2,A).

A = 8

?- max(6,6,A).

A = 6

Δομή Ελέγχου case (1/2)

- Να οριστεί το κατηγόρημα **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).
```

Δομή Ελέγχου case (2/2)

```
menu :-
    read(X),
    case( [X=1, X=2, X=3],
          [write('1st choice'), write('2nd choice'),
                write('3rd choice')],
         write('Out of Range')),
    nl.
```

Διαβάζει έναν όρο από το πληκτρολόγιο.

```
| ?- menu.
|: 2.
2nd choice
yes
```

```
| ?- menu.
|: 5.
Out of Range
yes
```

Είδη Αποκοπής

- Η αποκοπή μπορεί να μεταβάλει την δηλωτική ερμηνεία ενός προγράμματος.
  - **ΠΡΑΚΤΙΚΑ**: Δηλωτική ερμηνεία ενός προγράμματος είναι το σύνολο όλων των συμπερασμάτων στα οποία καταλήγει.
- Αν η αποκοπή μεταβάλλει την δηλωτική ερμηνεία ενός προγράμματος ονομάζεται κόκκινη αποκοπή (red cut).
- Αν η αποκοπή δεν μεταβάλλει την δηλωτική ερμηνεία ενός προγράμματος, αλλά απλά κάνει το πρόγραμμα πιο γρήγορο/αποδοτικό, ονομάζεται πράσινη αποκοπή (green cut).

Παράδειγμα Κόκκινης Αποκοπής

sign(X, positive) :- X > 0, !. sign(X, zero) :- X =:= 0, !. sign(X, negative).	?- sign(5,A). A = positive; no
sign(X, positive) :- X > 0. sign(X, zero) :- X =:= 0. sign(X, negative).	?- sign(5,A). A = positive; A = negative; no

- Η αποκοπή είναι κόκκινη γιατί άλλα αποτελέσματα επιστρέφονται με και άλλα χωρίς αυτήν.

Παράδειγμα Πράσινης Αποκοπής

sign(X, positive) :- X > 0, !. sign(X, zero) :- X =:= 0, !. sign(X, negative) :- X < 0.	?- sign(5,A). A = positive; no
sign(X, positive) :- X > 0. sign(X, zero) :- X =:= 0. sign(X, negative) :- X < 0.	?- sign(5,A). A = positive; no

- Η αποκοπή είναι πράσινη γιατί επιστρέφονται τα ίδια αποτελέσματα με και χωρίς αυτήν.
  - Η διαφορά με το προηγούμενο παράδειγμα οφείλεται στην έξτρα συνθήκη **X < 0.**
  - Το πρόγραμμα είναι αποδοτικότερο από αυτό χωρίς αποκοπή, αλλά όχι διαφορετικό.

Αποκοπή που δεν επηρεάζει (1/2)

- Η αποκοπή αν τοποθετηθεί στην αρχή της τελευταίας πρότασης ενός κατηγορήματος δεν επηρεάζει καθόλου.
  - Δεν υπάρχουν εναλλακτικές προτάσεις sign παρακάτω για να τις «κόψει».
  - Δεν υπάρχουν κλήσεις αριστερότερα της αποκοπής των οποίων τις εναλλακτικές λύσεις να «κόψει».

```
sign(X, positive) :- X > 0, !.
sign(X, zero) :- X =:= 0, !.
sign(X, negative) :- !, X < 0.
```

Αποκοπή που δεν επηρεάζει (2/2)

- Αν η αποκοπή τοποθετούνταν δεξιότερα στην τελευταία πρόταση, πάλι δεν θα επηρέαζε καθόλου, καθώς η κλήση X<0 δεν έχει εναλλακτικές λύσεις.

```
sign(X, positive) :- X > 0, !.
sign(X, zero) :- X =:= 0, !.
sign(X, negative) :- X < 0, ! .
```

Ασκήσεις με αποκοπή

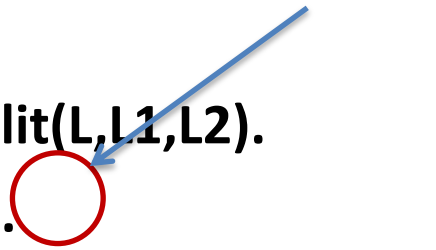
- Να ορισθεί σχέση **split(Nums,Pos,Neg)** που χωρίζει λίστα αριθμών **Nums** σε θετικούς **Pos** (και 0) και αρνητικούς **Neg.**

?- split([3,-1,0,5,-2],A,B).

A = [3,0,5], B = [-1,-2]

- Λύση με αποκοπή.

Κόκκινο cut



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

- Λύση χωρίς αποκοπή.

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

Συνδυαστική λύση

- Λύση με πράσινη αποκοπή.

```
split([],[],[]).

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

split([X|L],L1,[X|L2]) :- X < 0, split(L,L1,L2)
```

Ασκήσεις αποκοπής (count) (1/4)

- 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
```

Ασκήσεις αποκοπής (count) (2/4)

- Πώς συμπεριφέρεται το **count** όταν η λίστα περιέχει μεταβλητές?

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

X = a

Y = a

N = 4

- *Τα **X** και **Y** της ερώτησης κατά την εκτέλεση ταυτοποιούνται με* **a** *(λόγω της 2η πρότασης).*

Ασκήσεις αποκοπής (count) (3/4)

- **Λύση**: Η 2η πρόταση γίνεται:

```
count(_,[],0).
count(A,[B|L],N) :- A == B, !, 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 = _
Y = _
N = 2
```

Ασκήσεις αποκοπής (count) (4/4)

- **Εναλλακτική Λύση**: Η 2η πρόταση γίνεται:

```
count(_,[],0).
count(A,[B|L],N) :-
    nonvar(A), nonvar(B), A=B, !,
    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 = _, Y = _, N = 2
```

Επανάληψη στην Prolog

- Ως τώρα έχουμε δει πώς θα «εκτελέσουμε» επαναληπτικά στην Prolog κάποιες ενέργειες μέσω αναδρομής.
- Η αναδρομή καταναλώνει μνήμη σε κάθε επανάληψη.
- Αν χρειαστούν χιλιάδες ή δεκάδες χιλιάδες επαναλήψεις (π.χ. ανάγνωση δεδομένων από αρχείο) μπορεί να εξαντληθεί η μνήμη.
- Μερική λύση δίνει η χρήση της ουραίας αναδρομής (tail-recursion) με βοηθητικό κατηγόρημα.
  - Π.χ. ορισμός του «γρήγορου» reverse.

Επανάληψη μέσω οπισθοδρόμησης

- Υπάρχει άλλη τεχνική επανάληψης μέσω οπισθοδρόμησης.
- Η επανάληψη δεν καταναλώνει καθόλου μνήμη.
  - Η οπισθοδρόμηση ακυρώνει τις αναθέσεις τιμών στις μεταβλητές.
- Η οπισθοδρόμηση είναι πιο γρήγορη διαδικασία από την αναδρομή.
- Σε κάθε επανάληψη πρέπει να εκτελείται μία έξτρα-λογική (μη-αναστρέψιμη) ενέργεια η οποία διαφοροποιεί κάθε βήμα από το προηγούμενο.

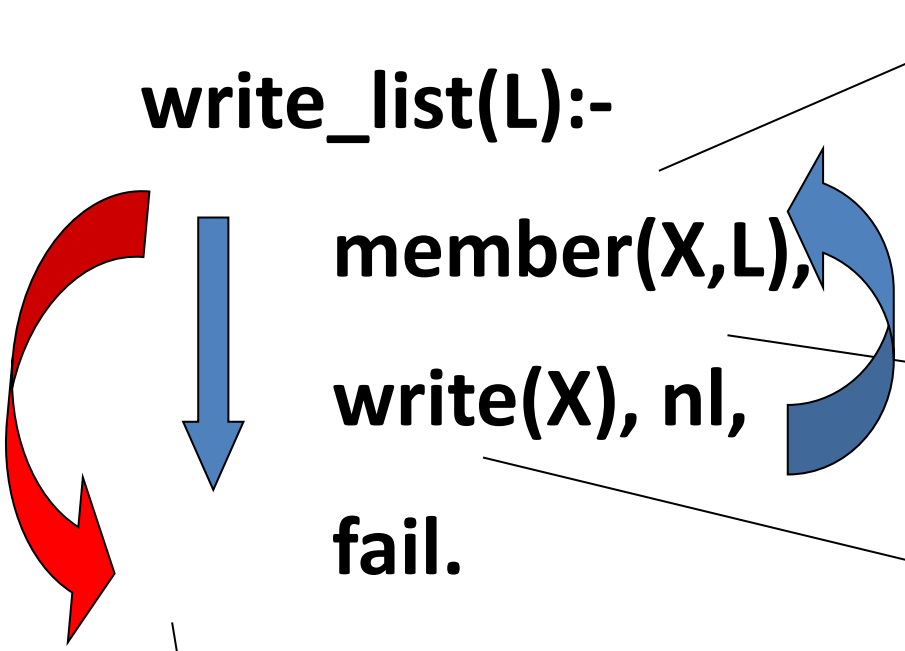
Εκτύπωση στοιχείων λίστας (1/2)

- Αναδρομικός ορισμός.

```
write_list([]).
write_list([X|T]) :-
    write(X), nl,
    writelist(T).
```

Εκτύπωση στοιχείων λίστας (2/2)

- Επανάληψη μέσω οπισθοδρόμησης.



Δημιουργία εναλλακτικών λύσεων.
Σημείο οπισθοδρόμησης.

Μη-αναστρέψιμες ενέργειες.

Αποτυχία.
Εξαναγκασμός σε οπισθοδρόμηση.

Θα «εκτελεστεί» μετά την έξοδο από το βρόχο.

Βρόχοι τύπου *while*

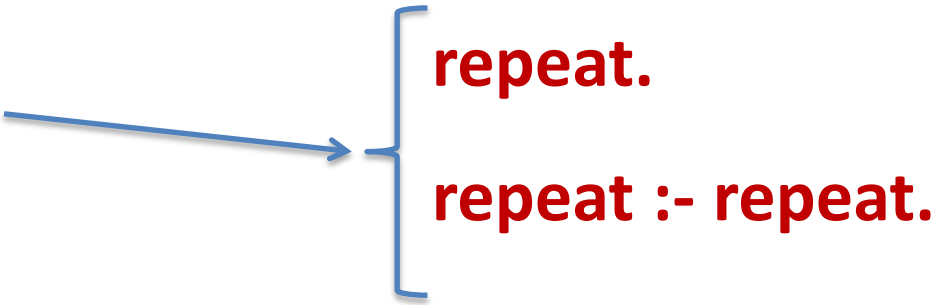
- Χρησιμοποιούμε κάποιο κατηγόρημα που παράγει (μέσω οπισθοδρόμησης) εναλλακτικές λύσεις (π.χ. **member**, **append**, κλπ.).
- Όταν εξαντλούνται οι εναλλακτικές λύσεις η οπισθοδρόμηση οδηγεί σε εναλλακτικό κανόνα.
- Αυτό το είδος της επανάληψης μοιάζει με *while.*
- Η είσοδος και η έξοδος στο βρόχο γίνεται από την κορυφή.
- Στη βάση της επανάληψης χρησιμοποιείται πάντα **fail.**

Βρόχοι τύπου *repeat/until*

- Χρησιμοποιούμε το κατηγόρημα **repeat.**
  - Το **repeat** ΠΑΝΤΑ πετυχαίνει και έχει ΠΑΝΤΑ εναλλακτικές λύσεις.
- Η έξοδος από το βρόχο ΔΕΝ γίνεται από την κορυφή αλλά ΜΟΝΟ από την βάση.
- ΔΕΝ χρησιμοποιούμε **fail** γιατί τότε θα έχουμε ατέρμονα βρόχο.
- Χρησιμοποιούμε κάποιο κατηγόρημα που συνεχώς αποτυγχάνει (για να προκαλέσει οπισθοδρόμηση).
  - ΕΚΤΟΣ από 1 φορά – την τελευταία.
  - Έξοδος από το βρόχο.

Ατέρμονας βρόχος

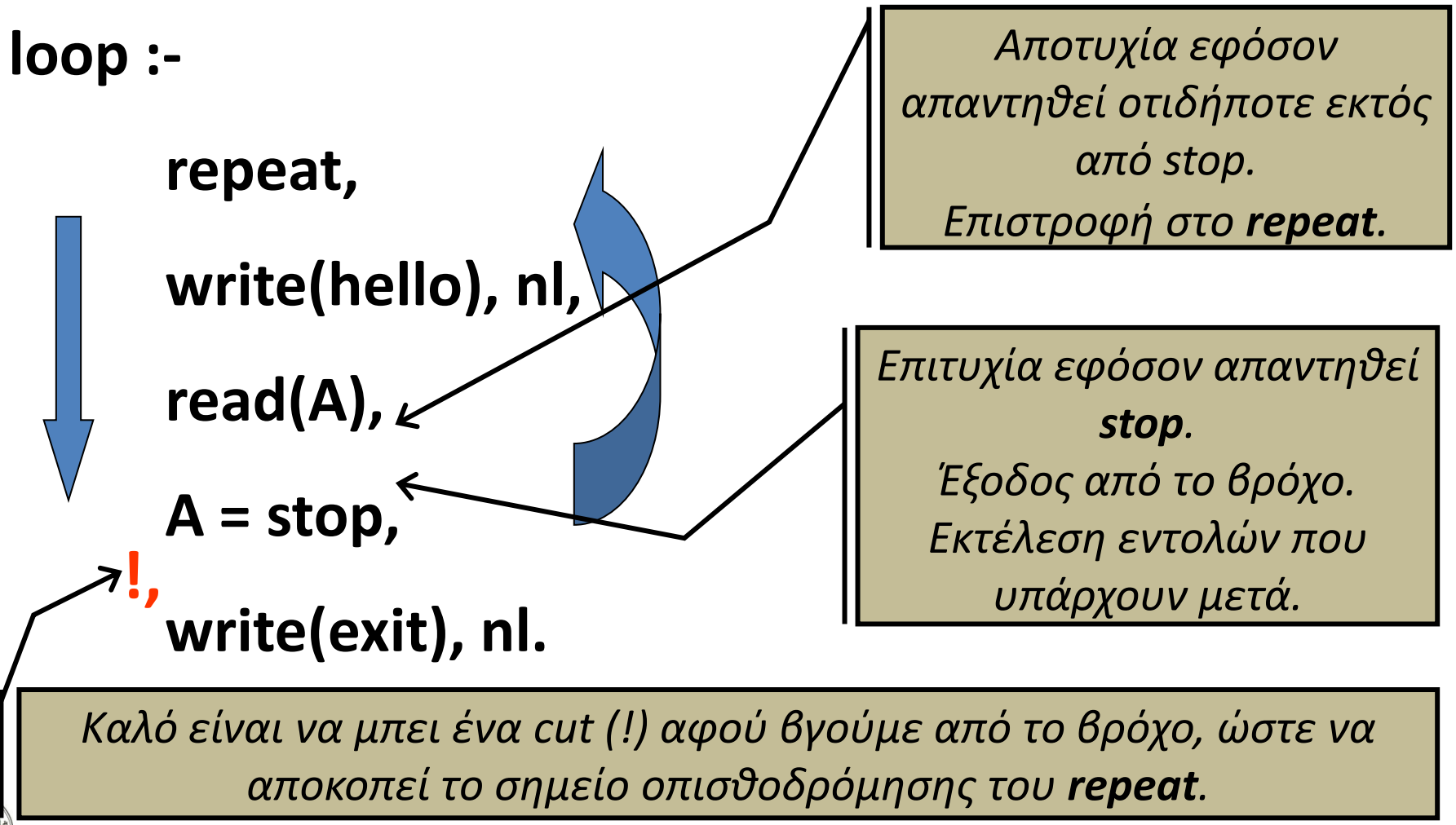
```
loop :-
    repeat,
    write(hello), nl,
    fail.
```



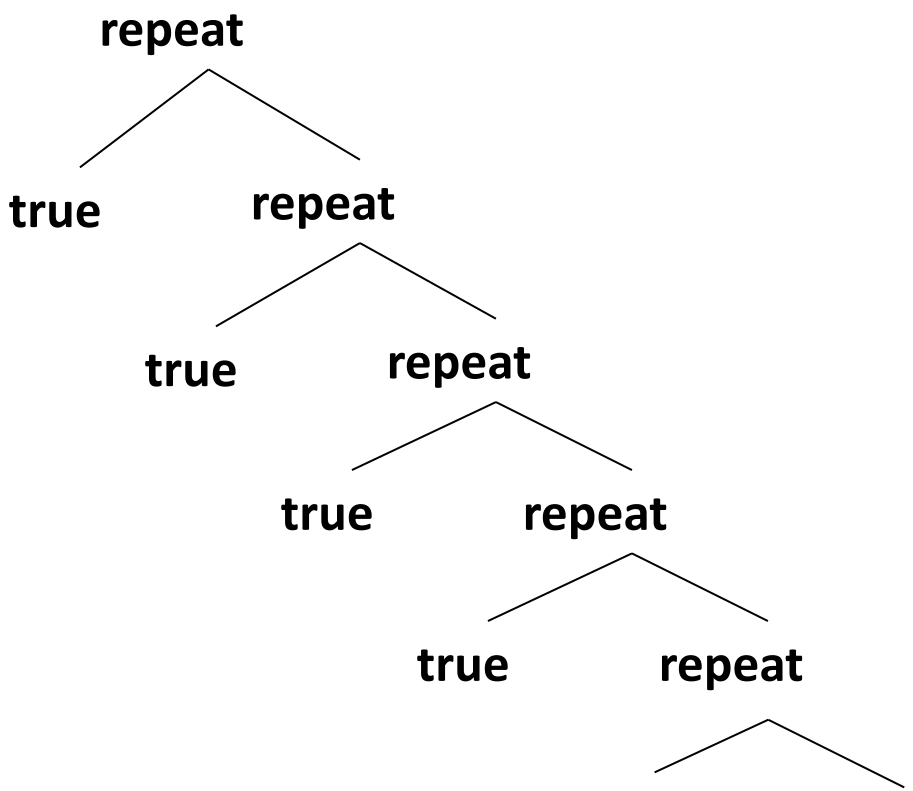

```
?- loop.
hello
hello
...
```

Άπειρες φορές

Βρόχος *repeat/until* που τερματίζει



Δένδρο εκτέλεσης repeat



repeat.

repeat :- repeat.

Μεικτοί Βρόχοι (1/2)

- Με συνδυασμό των δύο τεχνικών, μπορούμε να έχουμε μεικτούς βρόχους, δηλαδή βρόχους που μπορεί να τερματίσουν είτε από πάνω, είτε από κάτω.

```
loop :-
        member(X,[1,2,3,4,5]),
        write(X), nl,
        X > 2,
        !, write(exit), nl.
loop.
```

- *Θα τυπωθούν οι αριθμοί* **1, 2, 3** *και η λέξη* **exit.**
- *Ο βρόχος θα τερματίσει από κάτω.*

Μεικτοί Βρόχοι (2/2)

```
loop :-
        member(X,[1,2,3,4,5]),
        write(X), nl,
        X > 5,
        !, write(exit), nl.
loop.
```

- *Θα τυπωθούν οι αριθμοί* **1, 2, 3, 4, 5** *(ΟΧΙ η λέξη* **exit***).*
- *Ο βρόχος θα τερματίσει από πάνω.*

Σημείωμα Αναφοράς

Σημείωμα Αδειοδότησης

Τέλος ενότητας

Επεξεργασία: Εμμανουήλ Ρήγας

Θεσσαλονίκη, Εαρινό Εξάμηνο 2013-2014

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Σημειώματα

Διατήρηση Σημειωμάτων

Οποιαδήποτε αναπαραγωγή ή διασκευή του υλικού θα πρέπει να συμπεριλαμβάνει:

- το Σημείωμα Αναφοράς
- το Σημείωμα Αδειοδότησης
- τη δήλωση Διατήρησης Σημειωμάτων
- το Σημείωμα Χρήσης Έργων Τρίτων (εφόσον υπάρχει)

μαζί με τους συνοδευόμενους υπερσυνδέσμους.