ΑΤΕΙ ΘΕΣΣΑΛΟΝΙΚΗΣ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΜΑΘΗΜΑ: ΕΡΓΑΣΤΗΡΙΟ ΤΕΧΝΗΤΗΣ ΝΟΗΜΟΣΥΝΗΣ (Prolog)
ΕΞΑΜΗΝΟ: Δ΄- Εαρινό 2018-19
ΚΑΘΗΓΗΤΕΣ: Δ.ΣΤΑΜΑΤΗΣ, Μ.ΒΟΖΑΛΗΣ, Κ.ΚΡΑΒΑΡΗ

ΑΣΚΗΣΗ 6:
ΛΙΣΤΕΣ ΚΑΙ ΑΝΑΔΡΟΜΗ (2)

Αναδρομή και το κατηγόρημα append

Θεωρήστε για παράδειγμα το κατηγόρημα `append` το οποίο ενώνει τα στοιχεία δύο λιστών σε μια τρίτη

```
append([], L, L).
append([H|L1], L2, [H|L3]) :-
                append(L1, L2, L3).
```

Το κατηγόρημα αυτό είναι ίσως το πιο πολυχρησιμοποιούμενο της Prolog. (Η SWI-Prolog το φορτώνει αυτόματα – Αν θέλετε να δοκιμάσετε το δικό σας κώδικα πρέπει να δώσετε διαφορετικό όνομα, π.χ. my_append ή list_append).

Όπως σε όλα τα αναδρομικά κατηγορήματα το `append()` αποτελείται από παραπάνω από μία φράση.

Η πρώτη φράση είναι γεγονός και ορίζει την οριακή συνθήκη η οποία περιέχει όπως συνήθως (όχι όμως πάντα!!!) την κενή λίστα `[]`. Το γεγονός αυτό στη συγκεκριμένη περίπτωση λέει απλά ότι αν ενώσουμε μια κενή λίστα με μια λίστα `L` θα πάρουμε σαν αποτέλεσμα τη λίστα `L`.

Η δεύτερη φράση είναι κανόνας και περιγράφει την αναδρομική σχέση στην οποία υπακούει η ένωση δύο λιστών: αν ενώσουμε μια λίστα με πρώτο στοιχείο `H` και ουρά `L1` με μια λίστα `L2` τότε θα πάρουμε μια λίστα με πρώτο στοιχείο πάλι το `H` και ουρά την ένωση των `L1` και `L2` (που προκύπτει από την αναδρομική κλήση του append).

Ας δούμε με λεπτομέρεια τις ενέργειες της Prolog για να ικανοποιήσει την ερώτηση:

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

1. Καλείται ο κανόνας

```
append([H|L1], L2, [H|L3]) :-
        append(L1, L2, L3).
```

όπου έχουμε τις εξής ενοποιήσεις (δεσμεύσεις μεταβλητών με όρους): `H=a, L1=[b], L2=[c,d].`

Το παραπάνω ερώτημα αντικαθίσταται από την προυπόθεση του κανόνα:

```
        append(L1, L2, L3)
```

η οποία με βάση τις παραπάνω ενοποιήσεις οδηγεί στην νέα ερώτηση (Εφαρμογή της Αρχή της Ανάλυσης) :

```
?- append([b], [c,d], L3).
```

2. Καλείται εκ νέου ο κανόνας

```
append([H|L1], L2, [H|L3]) :-
        append(L1, L2, L3).
```

όπου έχουμε τις εξής ενοποιήσεις (δεσμεύσεις μεταβλητών): `H=b, L1=[], L2=[c,d].`

Το ερώτημα αντικαθίσταται από την προυπόθεση του κανόνα:

```
        append(L1, L2, L3).
```

η οποία με βάση τις παραπάνω ενοποιήσεις οδηγεί στην νέα ερώτηση

```
?- append([], [c,d], L3).
```

3. Καλείται το γεγονός

```
append([], L, L).
```

Με τη κλήση αυτή το τελευταίο ερώτημα επιτυγχάνει κάνοντας την ενοποίηση `L=[c,d].`

4. Μετά την επιτυχία του ερωτήματος αρχίζει η προς τα πίσω αντικατάσταση (back-substitution):

Το L3 του βήματος 2 είναι το L του βήματος 3 άρα

L3(βήμα-2) = [c, d]

Το L3 του βήματος 1 είναι το [H|L3] του βήματος 2 άρα

L3(βήμα-1) = [b, c, d]

Το L της ερώτησης είναι το [H|L3] του βήματος 1 άρα

L = [a, b, c, d]

Εναλλακτικές χρήσεις του append.

Είναι πάρα πολύ σημαντικές οι εναλλακτικές χρήσεις του κατηγορήματος append. Ως γνωστόν το append ενώνει δύο λίστες σε μια τρίτη. Καθώς όμως η Prolog δεν ορίζει ποια ορίσματα είναι γνωστά (και άρα είσοδοι στο append) και ποια ορίσματα είναι άγνωστα (και άρα έξοδοι του append) μπορούμε να χρησιμοποιήσουμε το κατηγόρημα αυτό με άγνωστες τις δύο λίστες που πρέπει να συνενωθούν και γνωστό μόνο το αποτέλεσμα της συνένωσης, να κόψουμε τη λίστα που αποτελεί το τρίτο όρισμά του σε υπολίστες που ενωμένες μας δίνουν το τρίτο όρισμα. Π.χ.

```
?- append(A,B,[a,b,c]).
A = []
B = [a,b,c];

A = [a]
B = [b,c];

A = [a,b]
B = [c];

A = [a,b,c]
B = [];

No (False)
```

Ποιες είναι οι πιθανές λίστες A, B που ενωμένες μας δίνουν τη λίστα [a,b,c]?

Με άλλα λόγια:
κόψε το τρίτο όρισμα της append (δηλαδή τη λίστα [a,b,c]) σε 2 υπολίστες.

Αναδρομή και το κατηγόρημα `reverse()`

Καθώς ο μόνος τρόπος προσπέλασης των στοιχείων μιας λίστας είναι ο χωρισμός της σε κεφαλή και σε ουρά, η επεξεργασία των λιστών γίνεται σχεδόν πάντα με αναδρομικά κατηγορήματα. Αυτά τα κατηγορήματα αφού υποθέτουν ότι έχουν λύσει το πρόβλημα για την ουρά της λίστας βρίσκουν τη λύση για ολόκληρη την εν λόγω λίστα, δηλαδή μαζί με την κεφαλή της. Η λύση για την ουρά δε βρίσκεται φυσικά δια μαγείας αλλά με την **αναζήτηση** σε ένα δένδρο: αναζητείται διαδοχικά η λύση της ουράς της ουράς, η λύση της ουράς της ουράς της ουράς, κλπ, μέχρι που η ουρά να αποκτήσει λίγα ή κανένα στοιχείο (=κενή λίστα) οπότε καλείται μια οριακή συνθήκη και κατόπιν με τη μέθοδο της **πίσω-αντικατάστασης** (**back-substitution**) συμπληρώνονται οι λύσεις που είχαν μείνει σε εκκρεμότητα.

Για παράδειγμα, έστω ότι θέλουμε να ορίσουμε το κατηγόρημα reverse(L,L1) το οποίο αντιστρέφει τη σειρά των στοιχείων της λίστας L και βάζει το αποτέλεσμα στην L1, πχ.:

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

Η υλοποίηση είναι απλά η παρακάτω:

```
reverse([],[]).

reverse([H|T],L) :-
    reverse(T,L1),
    append(L1,[H],L).
```

Οριακή συνθήκη: αν η λίστα είναι κενή τότε η αντίστροφή της είναι επίσης κενή.

Αναδρομική συνθήκη: έστω ότι L είναι το reverse της λίστας [H|T]. Στην περίπτωση αυτή …

…αν η reverse της T είναι η λίστα L1…

…τότε η L προκύπτει από την L1 αν στο τέλος της προσθέσουμε το στοιχείο H. Αυτό τεχνικά γίνεται με την append όπως φαίνεται εδώ.

Έτσι για παράδειγμα, η ερώτηση

```
?-reverse([a,b,c],LA).
```

θα καλέσει, μέσω του αναδρομικού κανόνα, τις προυποθέσεις του (ενοποιήσεις: H=a, T=[b,c], LA=L)

(1) **?-reverse([b,c],L1),append(L1,[a],LA).**

Εδώ η απάντηση `L1` εκκρεμεί. Για να βρεθεί θα κληθούν, μέσω του αναδρομικού κανόνα, εκ νέου οι προυποθέσεις του (ενοποιήσεις: H'=β, T'=[c], L1=L' % Προσοχή πρόκειται για νέα στιγμιότυπα των H,T και L). Η ερώτηση μετασχηματίζεται στην:

(2) **?-reverse([c],L1'), append(L1',[b],L1),** `append(L1,[a],LA).`

Και πάλι η απάντηση `L1'` εκκρεμεί. Για να βρεθεί καλείται, μέσω του αναδρομικού κανόνα, το γεγονός (οριακή συνθήκη της αναδρομής) και η ερώτηση μετασχηματίζεται στην:

(3)**?-reverse([],L1''), append(L1'',[c],L1'),**`append(L1',[b],L1), append(L1,[a],LA).`

Με την ενεργοποίηση της οριακής συνθήκη έχουμε **L1''=[].**

Από το σημείο αυτό αρχίζει η προς τα πίσω αντικατάσταση και αποκαθιστώνται οι εκκρεμότητες που είχαμε δημιουργήσει.

- Αφού βρέθηκε το reverse της λίστας`[]` -> `[]` βρίσκουμε μέσω της `append([],[c],L1')` το `L1'=[c].`
- Αφού βρήκαμε το reverse της λίστας `[c]` -> `[c]` βρίσκουμε μέσω της `append([c],[b],L1)` το `L1=[c,b].`
- Αφού βρήκαμε το reverse της λίστας `[b,c]` -> `[c,b]` βρίσκουμε μέσω της `append([c,b],[a],LA)` το `LA=[c,b,α].`

Αλλα κατηγορήματα που θα χρειαστήτε για την άσκηση:

Το κατηγόρημα `not/1`

Το κατηγόρημα `not()` παίρνει σαν όρισμα ένα κατηγόρημα και επιτυγχάνει όταν το όρισμα επιτυγχάνει ενώ επιτυγχάνει όταν το όρισμα αποτυγχάνει. Π.χ.

```
?- not(member(2,[1,2,3])).
No
?- not(member(4,[1,2,3])).
Yes
```

Τα κατηγορήματα `atomic/1`

Το κατηγόρημα `atomic/1` είναι ενσωματωμένο στην Prolog και επιτυγχάνει αν το όρισμά του είναι άτομο ή αριθμός, ενώ αποτυγχάνει όταν το όρισμά της είναι λίστα ή μεταβλητή. Για παράδειγμα,

```
?- atomic(a).
Yes
?- atomic([1,2]).
No
```

```
?- atomic(2).
Yes
?- atomic(X).
No
```

Η ανώνυμη ή άγνωστη μεταβλητή _ (μεταβλητή don't care)

Όπως ήδη είπαμε το σύμβολο _ αντιστοιχεί σε μια άγνωστη μεταβλητή. Η μεταβλητή αυτή έχει μια ιδιαιτερότητα σε σχέση με όλες τις άλλες αγνώστους, όπως π.χ. την μεταβλητή `X`, στο ότι δεν μας ενδιαφέρει η τιμή που θα πάρει. Η μεταβλητή _ χρησιμεύει και σε κανόνες που κάποιο όρισμα δεν μας ενδιαφέρει, για παράδειγμα, η φράση:

```
member(X,[X|_]).
```

Αν βάζαμε σαν όρισμα το `[H|T]` τότε η SWI-Prolog θα ανίχνευε το γεγονός ότι στα ορίσματα εμφανίζεται η μεταβλητή `T` η οποία δεν χρησιμοποιείται πουθενά και θα μας έδινε το μήνυμα

```
[WARNING: (όνομα αρχείου:γραμμή κώδικα)
      Singleton variables: T]
```

Αν και δεν δηλώνει κανένα ουσιαστικό πρόβλημα το μήνυμα αυτό, μπορούμε να το αποφύγουμε με χρήση της an;vnymhw μεταβλητής _ (underscore).

ΤΙ ΠΡΕΠΕΙ ΝΑ ΚΑΝΕΤΕ

(α) Γράψτε το κατηγόρημα **`posneg(L,LP,LN)`** που ξεχωρίζει τα στοιχεία μίας λίστας σε θετικά και αρνητικά και παράγει δύο νέες λίστες με τα στοιχεία αυτά αντίστοιχα. Για παράδειγμα,

```
?- posneg([13,-51,-11,29], LP,LN).
LP=[13,29] LN=[-51,-11]
?- posneg([1, 3, 21], LP,LN).
LP=[1,3,21] LN=[]
```

(β) Γράψτε το κατηγόρημα **`sumlist(L,X)`** όπου το πρώτο όρισμα είναι μία λίστα από λίστες, και στο δεύτερο όρισμα επιστρέφεται το άθροισμα των μηκών όλων των λιστών. Για παράδειγμα,

```
?- sumlist([[2,3],[1,s,d,a],[3]], X).
X=7
?-sumlist([[],[1,3,2],[a,d,s]], X).
X=6
```

(γ) Γράψτε το κατηγόρημα **`enwsh(L1,L2,L)`** που συνενώνει δύο λίστες έτσι ώστε στη λίστα που προκύπτει να μην υπάρχουν στοιχεία που επαναλαμβάνονται (ένωση συνόλων?). Για παράδειγμα,

```
?- enwsh([3,5,9,11],[4,3,10,9], L).
L=[5,11,4,3,10,9]
?- enwsh([c,e,f,a,w],[a,b,c,d], X).
X=[e,f,w,a,b,c,d]
```

(δ) Γράψτε το κατηγόρημα **`flat(L1,L2)`** όπου το δεύτερο όρισμα `L2` είναι η πεπλατυσμένη λίστα των στοιχείων της `L1`: αν η `L1` περιέχει άλλες λίστες μέσα της τότε η `L2` περιέχει όλα τα άτομα που περιέχει η `L1` χωρίς όμως να περιέχει άλλες υπολίστες. Για παράδειγμα,

```
?- flat([[a, e], [[[b], c]]], L).
L2 = [a, e, b, c]
?- flat([a, [[b,c], [[d,e], f]], g], L2).
L2 = [a, b, c, d, e, f, g]
```

Υπόδειξη: Χρησιμοποιήστε το atomic/1.

(ε) Γράψτε το κατηγόρημα **`memberlist(X,L)`** όπου το δεύτερο όρισμα είναι μία λίστα από λίστες, και βρίσκει αν το στοιχείο `X` είναι μέλος οποιασδήποτε λίστας που περιέχεται μέσα στην L. Για παράδειγμα,

```
?- memberlist(a, [[b,c], [d,e], [f,2,a,3]]).
Yes
?- memberlist(4, [[b,c], [[w]], [f,2,a,3]]).
No
```