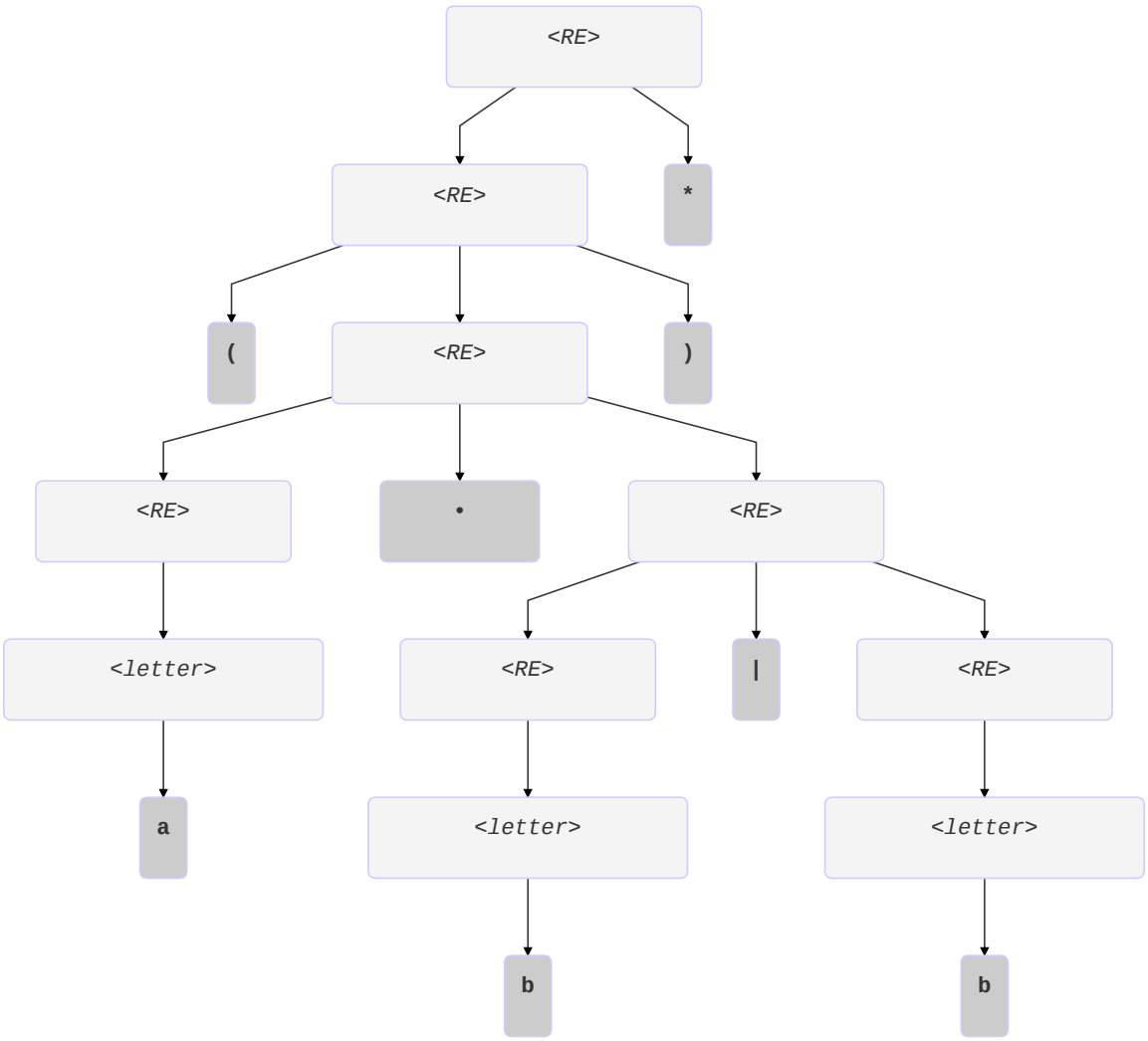
Γλώσσες Προγραμματισμού Ι: Λύσεις Κανονικής '16

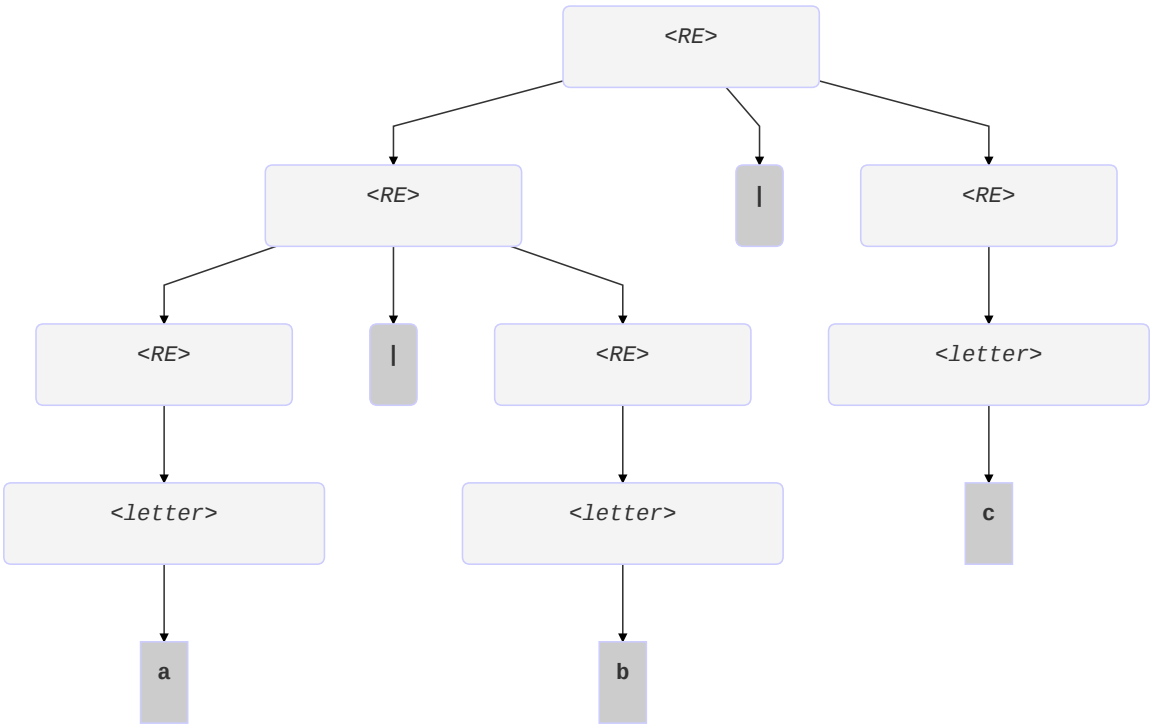
1. Γραμματικές

α) Το συντακτικό δέντρο για την έκφραση (a • b | b)*:

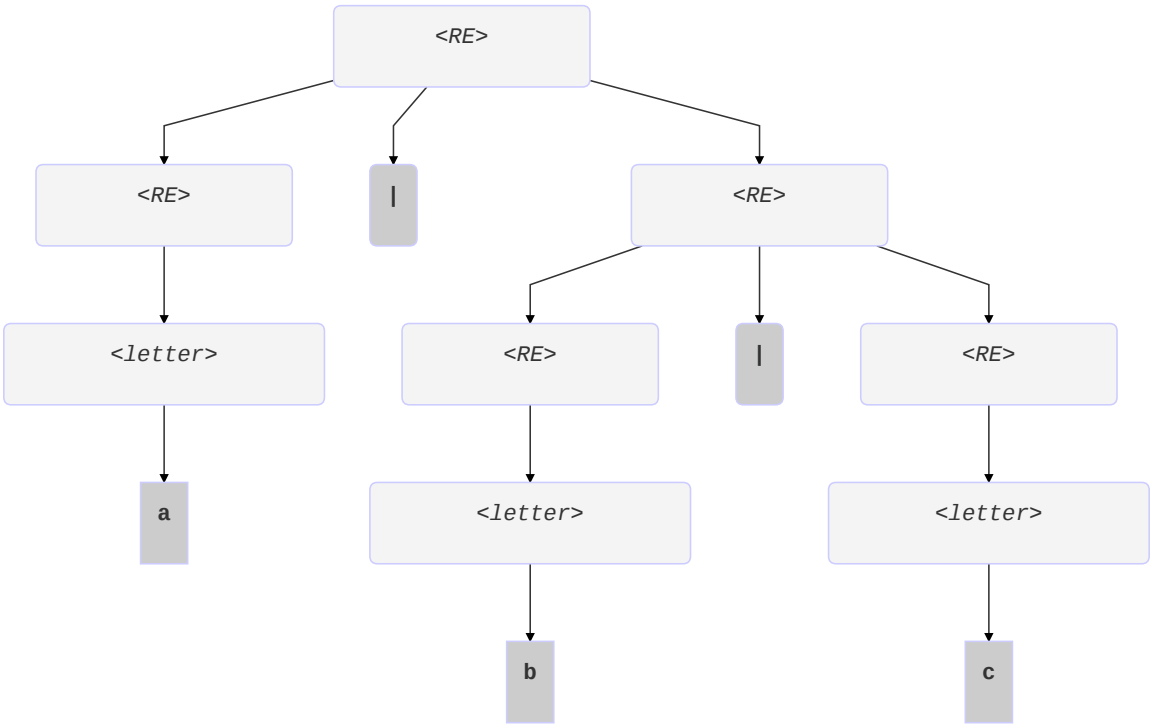


β) Για να είναι διφορούμενη η γραμματική θα πρέπει να υπάρχει μια συμβολοσειρά η οποία να αντιστοιχίζεται με παραπάνω από ένα συντακτικά δέντρα. Μία τέτοια συμβολοσειρά για τη δοθείσα γραμματική είναι η: a | b | c

1ο Συντακτικό Δέντρο:



2ο Συντακτικό Δέντρο:



γ) Η προτεραιότητα ενός τελεστή ορίζεται από το πόσο χαμηλά εμφανίζεται στο συντακτικό δέντρο. Όσο **χαμηλότερα** βρίσκεται, τόσο **υψηλότερη** είναι η προτεραιότητά του. Αντίστοιχα, όσο πιο κοντά βρίσκεται στην κορυφή, τόσο χαμηλότερη είναι η προτεραιότητά του.

Έχοντας το παραπάνω υπ'όψιν θέλουμε να τροποποιήσουμε τη δοθείσα γραμματική ώστε η προτεραιότητα των τελεστών να είναι η εξής, από τη χαμηλότερη στην υψηλότερη: | • *

Έτσι καταλήγουμε στην παρακάτω γραμματική (με '|' εννοείται το σύμβολο | της γραμματικής):

```
<RE>     ::= <Or> | <RE> '|' <Or>
<Or>     ::= <And> | <Or> • <And>
<And>    ::= <Expr> | <Expr>*
<Expr>   ::= <letter> | (<RE>)
<letter> ::= a | b | ... | z
```

Για την προτεραιότητα, φτιάξαμε νέα μη-τερματικά σύμβολα για κάθε τελεστή και ξεκινώντας από αυτόν με τη χαμηλότερη προτεραιότητα (|) φτάσαμε με εμφωλιασμούς σε αυτόν με την υψηλότερη (*).

δ) Η γραμματική που φτιάξαμε δίνει αριστερή προσεταιριστικότητα στη διάζευξη και την παράθεση, αφού επιτρέπει την αναφορά στον εαυτό τους μόνο από το αριστερό μέρος, ενώ στο δεξί μέρος επιβάλει την αναφορά μόνο σε κατώτερα επίπεδα.

2. Προγραμματισμός σε ML

α) Εξήγηση:

Ορίζουμε βοηθητική συνάρτηση `traverse`. Η `traverse` κρατάει έναν **accumulator** με τις **ομάδες** στις οποίες έχει χωριστεί ανά πάσα στιγμή η λίστα. Διαπερνάει σειριακά τη λίστα και για κάθε στοιχείο `el` που συναντάει, ελέγχει αν το `el` είναι μεγαλύτερο ή ίσο με το τελευταίο στοιχείο που μπήκε στην τελευταία ομάδα. Αν είναι, τότε μπορεί να μπει στην ομάδα διατηρώντας τη μη φθίνουσα σειρά, οπότε το κάνει **prepend**. Αν δεν ανήκει, δημιουργεί μία **νέα** ομάδα με μοναδικό στοιχείο το `el` και την προσθέτει στην **κορυφή** του accumulator. Η κατασκευή των ομάδων γίνεται επίτηδες ανάποδα, ώστε να μπορούμε κάθε φορά να βρούμε εύκολα ποιο ήταν το τελευταίο στοιχείο που μπήκε στον accumulator, και σε ποια ομάδα ανήκει.

Πρόγραμμα:

```
fun listify L =
    let
        (* Αν είναι το πρώτο στοιχείο που εξετάζουμε *)
        fun traverse (el, [nil]) = [[el]]

        (* Αν έχουν ήδη μπει στοιχεία *)
        |   traverse (el, (lastNum :: lastGroup) :: rest) =
            (* Αν διατηρείται η μη φθίνουσα σειρά, πρόσθεσέ το *)
            if el >= lastNum then
                (el :: lastNum :: lastGroup) :: rest
            else
            (* Αλλιώς, φτιάξε νέα ομάδα *)
                [el] :: (lastNum :: lastGroup) :: rest
```

```
        (* Διάσχιση στοιχείων σειριακά μέσω της foldl *)
        val result = foldl traverse [[]] L
    in
        (* Αντιστρέφουμε τη σειρά ομάδων και των στοιχείων τους, για να βγει το σωστό
αποτέλεσμα *)
        rev (map rev result)
    end
```

β)

> Στην προηγούμενη έκδοση του αρχείου υπήρχε άλλη, πιο περίπλοκη, λύση στο ερώτημα. Μπορείτε να την δείτε στο Παράρτημα. Η παρούσα λύση είναι του Βασίλη Ξ.

Τα βήματα του αλγορίθμου είναι τα εξής:

- Ταξινομούμε τις λίστες ως προς το μήκος τους (από το μικρότερο στο μεγαλύτερο) μέσω της `cmp`
  πχ. `[[1, 2, 3], [4], [5, 6], [7]]` --> `[[4], [7], [5, 6], [1, 2, 3]]`
- Με την `groupLists` διατρέχουμε σειριακά τις λίστες και ομαδοποιούμε αυτές που έχουν ίδιο μήκος. Όποτε συναντάμε λίστα με ίδιο μήκος με την προηγούμενή της, τις βάζουμε στην ίδια ομάδα. Μόλις αυξηθεί το μήκος, δημιουργούμε νέα ομάδα.
  πχ. `[[4], [7], [5, 6], [1, 2, 3]]` --> `[ [[4], [7]], [[5, 6]], [[1, 2, 3]] ]`
  Οι `[4], [7]` έχουν ίδιο μήκος και ανήκουν στην ίδια ομάδα. Οι υπόλοιπες είναι μόνες τους.
- Παρατηρούμε ότι το μήκος κάθε **ομάδας** αναπαριστά τη συχνότητα μήκους των λιστών που περιέχει! Για παράδειγμα η ομάδα `[[4], [7]]` έχει μήκος 2, μιας και ο αριθμός των αρχικών λιστών που είχαν μήκος 1, ήταν 2.
- Κάνουμε πάλι ταξινόμηση ως προς το μήκος, αλλά αυτή τη φορά για τις ομάδες. Με αυτόν τον τρόπο έχουμε καταφέρει οι λίστες να εμφανίζονται με τη σειρά που θέλουμε, δηλαδή ανάλογα με τη συχνότητα εμφάνισης του μήκους τους.
- Τέλος, χρησιμοποιώντας την `List.concat` ενοποιούμε τις ομάδες και παράγουμε την τελική λίστα.

```
fun lenfreqsort L =
    let
        (* Ταξινόμηση ως προς μήκος *)
        fun cmp (L1, L2) = (length L1) > (length L2)
        val sorted = sort cmp L

        (* Ομαδοποίηση λιστών ίδιου μήκους *)
        fun groupLists (lst, nil) = [[lst]]
        |   groupLists (lst, (lastList :: lastGroup) :: rest) =
            (* Αν είναι ίδιου μήκους με πριν, βάλ'την στην ίδια ομάδα *)
            if (length lst) = (length lastList) then
                (lst :: lastList :: lastGroup) :: rest
            else
            (* Αλλιώς φτιάξε νέα ομάδα *)
                [lst] :: (lastList :: lastGroup) :: rest

        (* Ταξινόμηση ομάδων ως προς μήκος *)
        val grouped = foldl groupLists [] sorted
        val sortGrouped = sort cmp grouped
```

```
        (* Συνένωση ομάδων μεταξύ τους *)
        val answer = List.concat sortGrouped
    in
        answer
    end
```

3. Συμπερασμός τύπων στην ML

A. `fun foo x y z = x (y (z + 42))`

x: a = e -> d (αφού καλείται με όρισμα το `y (z + 42)` τύπου e)

y: b = int -> e (αφού καλείται με όρισμα το `z + 42` τύπου int)

z: c = int (αφού προστίθεται με το 42)

foo: a -> b -> c -> d (αφού έχει ως αποτέλεσμα την τιμή του `x (y (z + 42))`)

Άρα foo: (e -> d) -> (int -> e) -> int -> d ή `foo = fn : ('a -> 'b) -> (int -> 'a) -> int -> 'b` όπως γράφεται στην ML.

B. `fun bar x y = x y (y + 42)`

x: a = int -> int -> c (αφού καλείται με ορίσματα τα `y`, `y + 42`)

y: b = int (αφού προστίθεται με το 42)

bar: a -> b -> c (αφού έχει ως αποτέλεσμα την τιμή του `x y (y + 42)`)

Άρα bar: (int -> int -> c) -> int -> c ή `bar = fn : (int -> int -> 'a) -> int -> 'a` όπως γράφεται στην ML.

Γ. `fun baz x y = x (y(x))`

x: a = c -> d (αφού καλείται με όρισμα το `y(x)`)

y: b = a -> c (αφού καλείται με όρισμα το `x`) = (c -> d) -> c

baz: a -> b -> d (αφού έχει ως αποτέλεσμα την τιμή του `x (y(x))`)

Άρα baz: (c -> d) -> ((c -> d) -> c) -> d ή `baz = fn : ('a -> 'b) -> (('a -> 'b) -> 'a) -> 'b` όπως γράφεται στην ML.

4. Εγγραφές δραστηριοποίησης

α) Από απάντηση του Nick V στο group Ροή Λ Fora

> Στο πρόγραμμα συμβαίνουν τα εξής:
>
> - καλείται η συνάρτηση `a`
> - η συνάρτηση `a` ορίζει μία τοπική μεταβλητή `m`
> - ανατίθεται η τιμή 0 στη μεταβλητή `m`
> - η συνάρτηση a επιστρέφει τη συνάρτηση `addm`, η οποία κάνει capture τη μεταβλητή `m`
> - η μεταβλητή `m` βγαίνει εκτός εμβέλειας
>
> Τώρα λοιπόν έχεις στα χέρια σου τη συνάρτηση `addm` η οποία έχει κάνει capture τη μεταβλητή `m`, η οποία όμως μεταβλητή είναι εκτός εμβέλειας (αφού έχει γίνει pop το activation record της `a` από το stack). Οπότε έχει στα χέρια της ένα dangling reference προς τη μεταβλητή `m`... Και όταν εκτελεστεί, το `m + n` (με `n` = 2) θα έχει απροσδιόριστη τιμή, αφού η `m` θα έχει απροσδιόριστη τιμή.
>
> Για να εκτελεστεί σωστά ένα τέτοιο πρόγραμμα πρέπει το activation record της `a` να αποθηκευτεί στο σωρό και όχι στη στοίβα.

Μικρή σημείωση (με επιφυλάξεις για το σε ποιες γλώσσες ισχύει)

Η διαφορά μεταξύ σωρού και στοίβας στην οποία οφείλεται η αδυναμία χρήσης μόνο της στοίβας, είναι η εξής: η στοίβα (stack) χρησιμοποιείται για δεδομένα που είναι γνωστό κατά τη μεταγλώττιση ότι θα χρειαστούν δέσμευση μνήμης, και τα οποία αποδεσμεύονται αυτόματα μόλις βγουν εκτός εμβέλειας. Αντίθετα ο σωρός (heap) χρησιμοποιείται για δυναμική δέσμευση μνήμης κατά την εκτέλεση.

β) Αν χρησιμοποιηθεί ο σωρός, η `addm(n = 2)` επιστρέφει `m + n`, όπου το `m` έχει πάρει τιμή 0 από την κλήση της `a`. Άρα η έξοδος θα είναι `m + n = 0 + 2 = 2`.

5. Πέρασμα παραμέτρων

α) Κλήση κατ' αναφορά (by reference)

A[0]	**A[1]**
3	2

β) Κλήση με τιμή-αποτέλεσμα (by value-result)

A[0]	**A[1]**
3	2

γ) Κλήση με επέκταση μακροεντολών (by macro-expansion)

A[0]	**A[1]**
1	3

δ) Κλήση κατ' όνομα (by name)

A[0]	A[1]
1	3

6. Σχεδιασμός γλωσσών προγραμματισμού

No thanks.

7. Προγραμματισμός σε Prolog

α) Θα χρειαστούμε τα γνωστά κατηγορήματα:

- `select/3` για την αφαίρεση ενός στοιχείου από μία λίστα
- `permut/2` για την εύρεση όλων των πιθανών μεταθέσεων μιας λίστας

Έτσι, το `magic/2` μπορεί να προκύψει από τους εξής περιορισμούς:

- Οι αριθμοί του τετραγώνου πρέπει να είναι οι ακέραιοι 1-9, και καθένας πρέπει να εμφανίζεται μια φορά. Αυτό διασφαλίζεται με το `permut` που ορίζουμε.
- Όλα τα αθροίσματα πρέπει να είναι ίσα με το 15. Αυτό προκύπτει ως εξής:
  - $V_1 + V_2 + V_3 = V_4 + V_5 + V_6 = V_7 + V_8 + V_9 = N$
  - $\sum_{i=1}^{9} V_i = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45 \implies 3N = 45 \implies N = 15$

```
select(X, [X|T], T).
select(X, [H|T1], [H|T2]) :- select(X, T1, T2).

permut([], []).
permut([H|T], P) :- permut(P1, T), select(H, P, P1).

magic([V1, V2, V3, V4, V5, V6, V7, V8, V9]) :-
    permut([V1, V2, V3, V4, V5, V6, V7, V8, V9], [1, 2, 3, 4, 5, 6, 7, 8, 9]),
    V1 + V2 + V3 =:= 15,
    V4 + V5 + V6 =:= 15,
    V7 + V8 + V9 =:= 15,
    V1 + V4 + V7 =:= 15,
    V2 + V5 + V8 =:= 15,
    V3 + V6 + V9 =:= 15,
    V1 + V5 + V9 =:= 15,
    V3 + V5 + V7 =:= 15.
```

β) Εξήγηση:

- Το `atlevel` βρίσκει όλους τους κόμβους οι οποίοι βρίσκονται στο ίδιο επίπεδο. Είναι tail-recursive, αφού το `Level` ενεργεί ως accumulator. Κατεβαίνει κάθε φορά αναδρομικά κατά ένα επίπεδο. Όταν

συμπληρώσει `Level - 1` βήματα από την κορυφή, τότε το Level που έχει περαστεί στην αναδρομική κλήση των κόμβων που βρίσκονται στο ζητούμενο επίπεδο, θα έχει μειωθεί στο 1, και έτσι κάθε τέτοιος κόμβος θα επιστρέψει την τιμή του. Κάθε πατέρας των κόμβων θα συγκεντρώσει τις τιμές τους από το αριστερό και δεξί υποδέντρο του και τελικά αυτές θα φτάσουν στην κορυφή.

- Το `tree_height` είναι το γνωστό κατηγόρημα εύρεσης του ύψους ενός δυαδικού δέντρου.
- Το `bfs_level` περιορίζει μέσω του κατηγορήματος `between/3` τις τιμές του `Level` ώστε να αναζητηθούν μόνο επίπεδα βάθους μεταξύ του 0 και του ύψους του δέντρου, που υπολογίζεται μέσω του `tree_height`. Για κάθε τέτοιο `Level`, επιστρέφει τους κόμβους του μέσω του `atlevel`.

```
atlevel(nil, _, []).
atlevel(node(X, _, _), 1, [X]).
atlevel(node(_, Left, Right), Level, List) :-
    Level > 1,
    Lm1 is Level - 1,
    atlevel(Left, Lm1, ListL),
    atlevel(Right, Lm1, ListR),
    append(ListL, ListR, List).

tree_height(nil, 0).
tree_height(node(_, Left, Right), Height) :-
    tree_height(Left, H1),
    tree_height(Right, H2),
    Height is 1 + max(H1, H2).

bfs_level(Tr, L) :-
    tree_height(Tr, MaxHeight),
    between(0, MaxHeight, Level),
    atlevel(Tr, Level, L).
```

Παράρτημα

Προηγούμενη λύση για το 2β

> Η λύση είναι λίγο περίπλοκη και μάλλον υπάρχει απλούστερη. Bear with me :)

Τα βήματα του αλγορίθμου είναι τα εξής:

- Ταξινομούμε τις λίστες ως προς το μήκος τους (από το μικρότερο στο μεγαλύτερο) μέσω της `cmp` πχ. [[1, 2, 3], [4], [5, 6], [7]] --> [[4], [7], [5, 6], [1, 2, 3]]
- Τώρα μπορούμε σειριακά να βρούμε πόσες φορές εμφανίζεται το κάθε μήκος. Με την `freq_traverse`, όποτε συναντάμε λίστα ίδιου μήκους με αυτήν που είδαμε προηγουμένως, αυξάνουμε τη συχνότητα. Αν δούμε λίστα μεγαλύτερου μήκους, ορίζουμε νέα συχνότητα με τιμή 1.
- Άρα τώρα έχουμε μία λίστα συχνοτήτων: [(2, 1), (1, 2), (1, 3)]. Για κάθε δυάδα, ο $1^{ος}$ αριθμός δείχνει τη συχνότητα και ο $2^{ος}$ το μήκος. Έτσι ξέρουμε ότι έχουμε 2 πίνακες μήκους 1, 1 πίνακα μήκους 2, κ.ό.κ.

- Η `expand` δημιουργεί μία λίστα ίδιου μήκους με την αρχική. Παίρνει κάθε στοιχείο από τη λίστα συχνοτήτων και το βάζει σε μια λίστα τόσες φορές όσες είναι η συχνότητά του. Με αυτόν τον τρόπο έχουμε καταφέρει να έχουμε μία 1:1 αντιστοίχιση κάθε στοιχείου από την ταξινομημένη λίστα με τη συχνότητα που εμφανίζεται το μήκος του.

  Πράγματι, για την ταξινομημένη [[4], [7], [5, 6], [1, 2, 3]] η `expand` δίνει τη λίστα [2, 2, 1, 1], αφού το μήκος του [4], εμφανίζεται 2 φορές, του [7] πάλι δύο φορές κ.ό.κ.
- Με την `zip` "ζευγαρώνουμε" τις δύο λίστες `sorted` και `freq_expanded`, ώστε στη νέα λίστα που θα προκύψει κάθε στοιχείο να είναι σε ζευγάρι με τη συχνότητα με την οποία εμφανίζεται το μήκος του. Αυτό θα χρειαστεί για το τελικό στάδιο της ταξινόμησης.
- Ταξινομούμε μέσω της `freq_cmp` τη λίστα που προέκυψε για να βρούμε την απάντηση. Η ταξινόμηση γίνεται με πρώτο κλειδί τη συχνότητα και δεύτερο το μήκος της λίστας, ώστε οι λίστες με ίδιο μήκος να εμφανίζονται συνεχόμενα.

```
fun lenfreqsort L =
    let
        (* Ταξινόμηση ως προς μήκος *)
        fun cmp (L1, L2) = (length L1) > (length L2)
        val sorted = sort cmp L

        (* Δημιουργία λίστας συχνοτήτων *)
        fun freq_traverse (el, nil) =
            [(1, length el)]
        |   freq_traverse (el, (freq, len) :: rest) =
            (* Αν είναι ίδιου μήκους με πριν, αύξησε τη συχνότητα *)
            if (length el) = len then
                (freq + 1, len) :: rest
            else
            (* Αλλιώς νέα συχνότητα = 1 *)
                (1, length el) :: (freq, len) :: rest

        val freqs = foldl freq_traverse [] sorted

        (* Επέκταση λίστας συχνοτήτων *)
        fun expand ((freq, len), accum) = List.tabulate(freq, fn _ => freq) @ accum
        val freq_expanded = foldl expand [] freqs

        (* Ζευγάρωμα επεκτεταμένης λίστας συχνοτήτων με την ταξινομημένη *)
        fun zip nil nil = nil
        |   zip (A1 :: A) (B1 :: B) = (A1, B1) :: (zip A B)

        val expanded = zip freq_expanded sorted

        (* Τελική ταξινόμηση ως προς συχνότητα και μήκος *)
        fun freq_cmp ((f1, L1), (f2, L2)) =
            f1 > f2 orelse (f1 = f2 andalso (length L1) > (length L2))

        val final_sorted = sort freq_cmp expanded
    in
        (* Θέλουμε να επιστρέφονται μόνο οι λίστες, χωρίς να φαίνεται η συχνότητα του
μήκους τους *)
```

```
        map (fn (freq, lst) => lst) final_sorted
    end
```