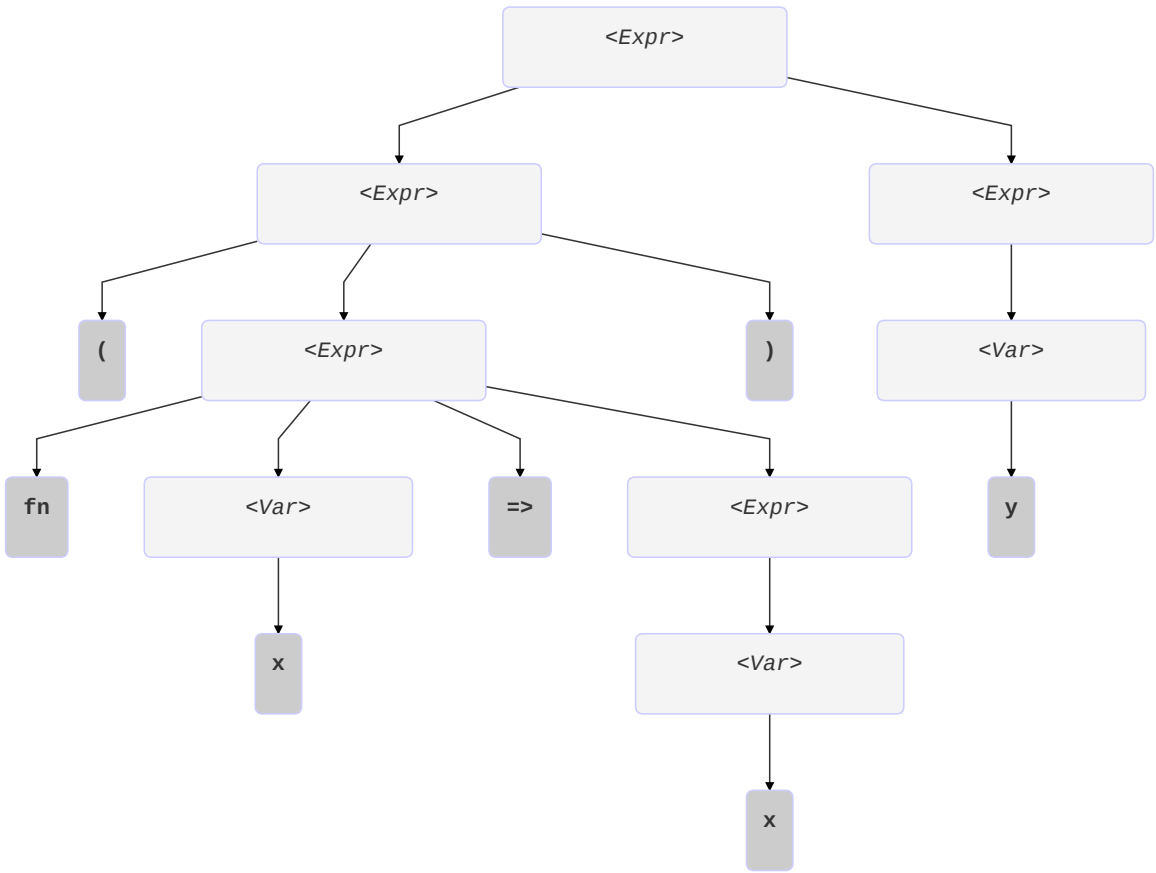
Γλώσσες Προγραμματισμού Ι: Λύσεις Επαναληπτικής '17

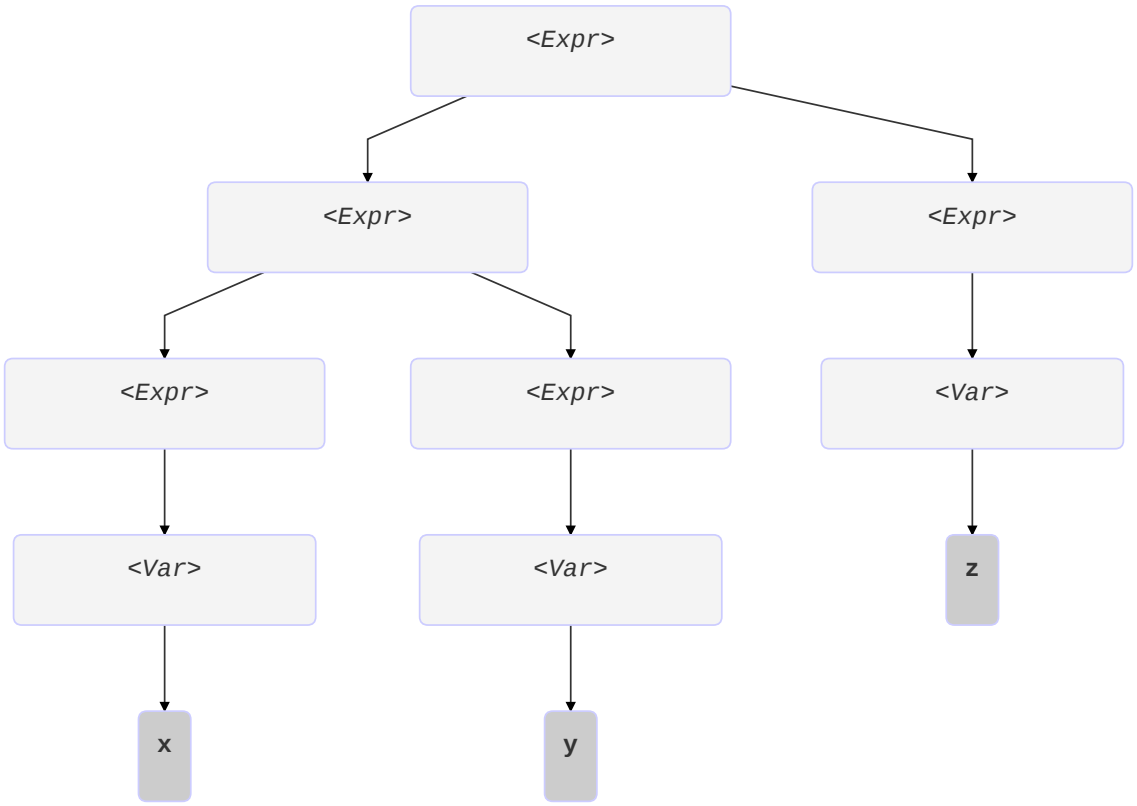
1. Γραμματικές

α) Το συντακτικό δέντρο για την έκφραση `(fn x => x) y`:

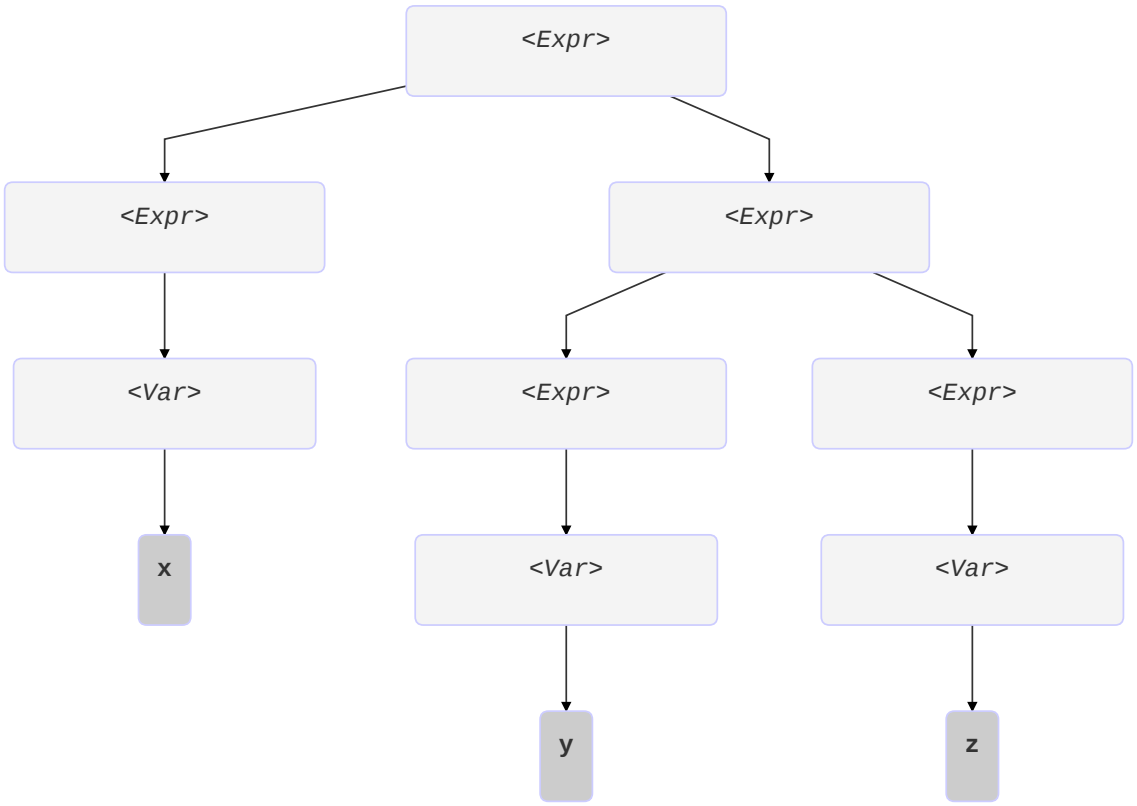


β) Για να είναι διφορούμενη η γραμματική θα πρέπει να υπάρχει μια συμβολοσειρά η οποία να αντιστοιχίζεται με παραπάνω από ένα συντακτικά δέντρα. Μία τέτοια συμβολοσειρά για τη δοθείσα γραμματική είναι η: `x y z`

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.

Την προσεταιριστικότητα ενός τελεστή μπορούμε να την επιβάλουμε, ορίζοντας με τέτοιον τρόπο την επέκταση των όρων του στη γραμματική, ώστε αυτή να γίνεται **μόνο από την πλευρά που επιθυμούμε**.

Έτσι καταλήγουμε στην παρακάτω γραμματική:

```
<Expr>  ::= fn <Var> => <Expr> | <Apply>
<Apply> ::= <Apply> <Paren> | <Paren>
<Paren> ::= <Var> | (<Expr>)
<Var>   ::= x | y | z
```

2. Προγραμματισμός σε ML

α)

Η `sizeUnbalanced` λειτουργεί αναδρομικά, βρίσκοντας για κάθε κόμβο το μέγεθος του αριστερού και δεξιού υποδέντρου του, καθώς και το πόσοι unbalanced κόμβοι υπάρχουν σε αυτά. Μέσω αυτών, βρίσκει το μέγεθος του δέντρου με ρίζα τον εαυτό του, και τους unbalanced κόμβους του. Αυτό προκύπτει από τις σχέσεις:

- Μέγεθος δέντρου με ρίζα έναν κόμβο = μέγεθος αριστερού υποδέντρου + μέγεθος δεξιού + 1
- Αν μέγεθος αριστερού = μέγεθος δεξιού, τότε:

  #unbalanced = #unbalanced αριστερού + #unbalanced δεξιού
- Αν έχουν διαφορετικό μέγεθος, τότε και ο ίδιος ο κόμβος είναι unbalanced, άρα:

  #unbalanced = #unbalanced αριστερού + #unbalanced δεξιού **+ 1**

```
fun sizeUnbalanced empty = (0, 0)
|   sizeUnbalanced (node (_, left, right)) =
        let
            val (sLeft, uLeft) = sizeUnbalanced left
            val (sRight, uRight) = sizeUnbalanced right
        in
            (
                sLeft + sRight + 1, (* number of nodes under this tree *)
                uLeft + uRight + (if sLeft = sRight then 0 else 1)
            )
        end;

fun countUnbalanced T = #2(sizeUnbalanced T)
```

β1) Η λύση είναι σωστή. Η `aux` χρησιμοποιεί εναλλάξ την `p` και την `not o p`, όπου στη θέση της `p` δίνεται μια συνάρτηση που ελέγχει αν ένας αριθμός είναι άρτιος. Αντίστοιχα η `not o p` ελέγχει αν ένας αριθμός είναι περιττός.

Έτσι, ξεκινώντας από το πρώτο στοιχείο της λίστας `h` ελέγχει αν είναι άρτιο. Αν δεν είναι, τερματίζει με τιμή false λόγω βραχυκύκλωσης. Αν είναι, η τιμή που επιστρέφει προκύπτει από την `aux` εφαρμοσμένη στην ουρά της λίστας (δηλαδή χωρίς το `h`) αλλά αυτή τη φορά το πρώτο στοιχείο της ουράς θα ελεγχθεί για περιττότητα.

Η διαδικασία επαναλαμβάνεται αναδρομικά μέχρι είτε η συνθήκη να αποτύχει για έναν αριθμό, οπότε και όλες οι επιμέρους `aux` που έχουν ελεγχθεί θα επιστρέψουν `false`, είτε η αναδρομή να διασχίσει όλες τις "ουρές" της αρχικής λίστας φτάνοντας τελικά στην κενή για την οποία επιστρέφεται `true`.

β2)

> ΣΗΜΕΙΩΣΗ: Έχει γίνει διόρθωση από Δημήτρη Σ, σε σχέση με την προηγούμενη έκδοση του αρχείου που ανέφερε λανθασμένα τη λύση ως αποδοτική.

Η λύση δεν είναι αποδοτική, αφού όσο προχωράει η λίστα, η `aux` δέχεται ως όρισμα την `p`, μετά την `not o p`, μετά την `not o (not o p)` κ.ό.κ. Επομένως πρέπει κάθε φορά να κάνει τον υπολογισμό μιας έκφρασης με αριθμό όρων που αυξάνεται γραμμικά. Έτσι, η πολυπλοκότητά της προκύπτει ότι είναι τετραγωνική.

3. Συμπερασμός τύπων στην ML

Α. `fun foo x y = x (y + 1) y`

- x: a = int -> int -> c (αφού καλείται με ορίσματα τα `y + 1`, `y`)
- y: b = int (αφού προστίθεται με το 1)
- foo: c (αφού c είναι το αποτέλεσμα της `x (y + 1) y`)

Άρα foo: (int -> int -> c) -> int -> c ή `foo = fn : (int -> int -> 'a) -> int -> 'a` αν γραφεί όπως στην ML.

Β. `fun bar x y = y (bar y x)`

- x: a = b = c -> d (αφού τα `x`, `y` χρησιμοποιούνται εναλλάξ στην `bar`) = c -> c
- y: b = c -> d = c -> c (`d = c` αφού το αποτέλεσμα της `y (bar y x)` είναι το αποτέλεσμα της `bar`)
- bar: a -> b -> c

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

Γ. `fun doh x y z = y (z x) + 1`

- x: a
- y: b = d -> int (αφού καλείται με όρισμα το `z x` και μετά το αποτελεσμά της αθροίζεται με το 1)
- z: c = a -> d (αφού καλείται με όρισμα το `x`)
- doh: a -> b -> c -> int (αφού το αποτελεσμά της είναι άθροισμα)

Άρα doh: a -> (d -> int) -> (a -> d) -> int ή `doh = fn : 'a -> ('b -> int) -> ('a -> 'b) -> int` αν γραφεί όπως στην ML.

Δ. `fun ugh x y z = z x (y x)`

- x: a
- y: b = a -> d (αφού καλείται με όρισμα το `x`)
- z: c = a -> d -> e (αφού καλείται με ορίσματα τα `x` και `y x`)
- ugh: a -> b -> c -> e (αφού e είναι το αποτέλεσμα της `z x (y x)`)

Άρα ugh: a -> (a -> d) -> (a -> d -> e) -> e ή `ugh = fn : 'a -> ('a -> 'b) -> ('a -> 'b -> 'c) -> 'c` αν γραφεί όπως στην ML.

4. Πέρασμα παραμέτρων

α) Κλήση με τιμή (by value)

f: x	f: y	f: A[0]	f: A[1]	f: A[2]	main: A[0]	main: A[1]	main: A[2]
1	2	0	1	1	0	1	1

β) Κλήση κατ' αναφορά (by reference)

f: x	f: y	f: A[0]	f: A[1]	f: A[2]	main: A[0]	main: A[1]	main: A[2]
1	2	1	1	2	1	1	2

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

f: x	f: y	f: A[0]	f: A[1]	f: A[2]	main: A[0]	main: A[1]	main: A[2]
1	2	0	1	1	1	1	2

δ) Κλήση κατ' όνομα (by name)

f: x	f: y	f: A[0]	f: A[1]	f: A[2]	main: A[0]	main: A[1]	main: A[2]
1	1	1	2	1	1	2	1

5. Ερωτήσεις πολλαπλής επιλογής

α1) Α **α2)** Γ **β1)** Α **β2)** Β

6. Προγραμματισμός σε Prolog

α1) Η λύση είναι σωστή. Το κατηγόρημα `helper` δεσμεύει σωστά την κορυφή κάθε ουράς του `S` (`H2`) με το κατηγόρημα `H2 is H1 + A`, και έτσι η Prolog μπορεί να βρει τη λίστα `S` για μία δεδομένη λίστα `L`.

Αν και στην αναδρομική κλήση περνιέται η μη αποτιμημένη έκφραση `+(H1, A)`, αυτό δεν αποτελεί πρόβλημα, αφού αυτή η έκφραση θα είναι το νέο `A'` στην εμφωλευμένη αναφορά του κατηγορήματος `helper`, και έτσι θα αποτιμηθεί σωστά στο επόμενο `H2 is H1 + A`.

α2) Η λύση δεν είναι αποδοτική, αφού όσο προχωράει η διάσχιση των λιστών, το μη αποτιμημένο `A` μεγαλώνει σε μέγεθος και "κουβαλάει" όλα τα προηγούμενα στοιχεία. Αυτό μετατρέπει τη λύση από μία γραμμική που θα ήταν κανονικά (αν αντί για `+(H1, A)` περνιόταν το `H2`) σε μία τετραγωνική $\mathcal{O}(N^2)$.

β)

Η λύση είναι bottom-to-top, καθώς για έναν κόμβο σε βάθος `D`, αναζητούνται πρώτα λύσεις σε καθένα από τα παιδιά του για βάθος `DD`. Αν βρεθούν `X`, `DD` που να οδηγούν σε λύση για κάποιο από τα παιδιά του, τότε αυτή αποτελεί και λύση για τον ίδιο τον κόμβο, και έτσι το βάθος του `D` ενοποιείται με την τιμή `DD + 1`.

Επιπλέον χρειάζεται μία "βάση", και αυτή είναι η `find_depth([X|_], X, 1)`, αφού κάθε δέντρο με ρίζα τιμής `X` ικανοποιεί τα ερωτήματα που ψάχνουν για την τιμή `X` σε βάθος 1.

> Note: Η σειρά των κατηγορημάτων στο σώμα του `find_depth` παίζει πολύ σημαντικό ρόλο, αφού είναι η σειρά με την οποία η Prolog θα αναζητήσει τις λύσεις. Για κάθε κόμβο, πρώτα βρίσκει τα πιθανά παιδιά του, μετά βρίσκεις πιθανές λύσεις στα παιδιά, και μετά αν αυτές βρεθούν γίνεται η ενοποίηση του D.

```
find_depth([X|_], X, 1).
find_depth([_|Children], X, D) :-
    member(Child, Children),
    find_depth(Child, X, DD),
    D is DD + 1.
```