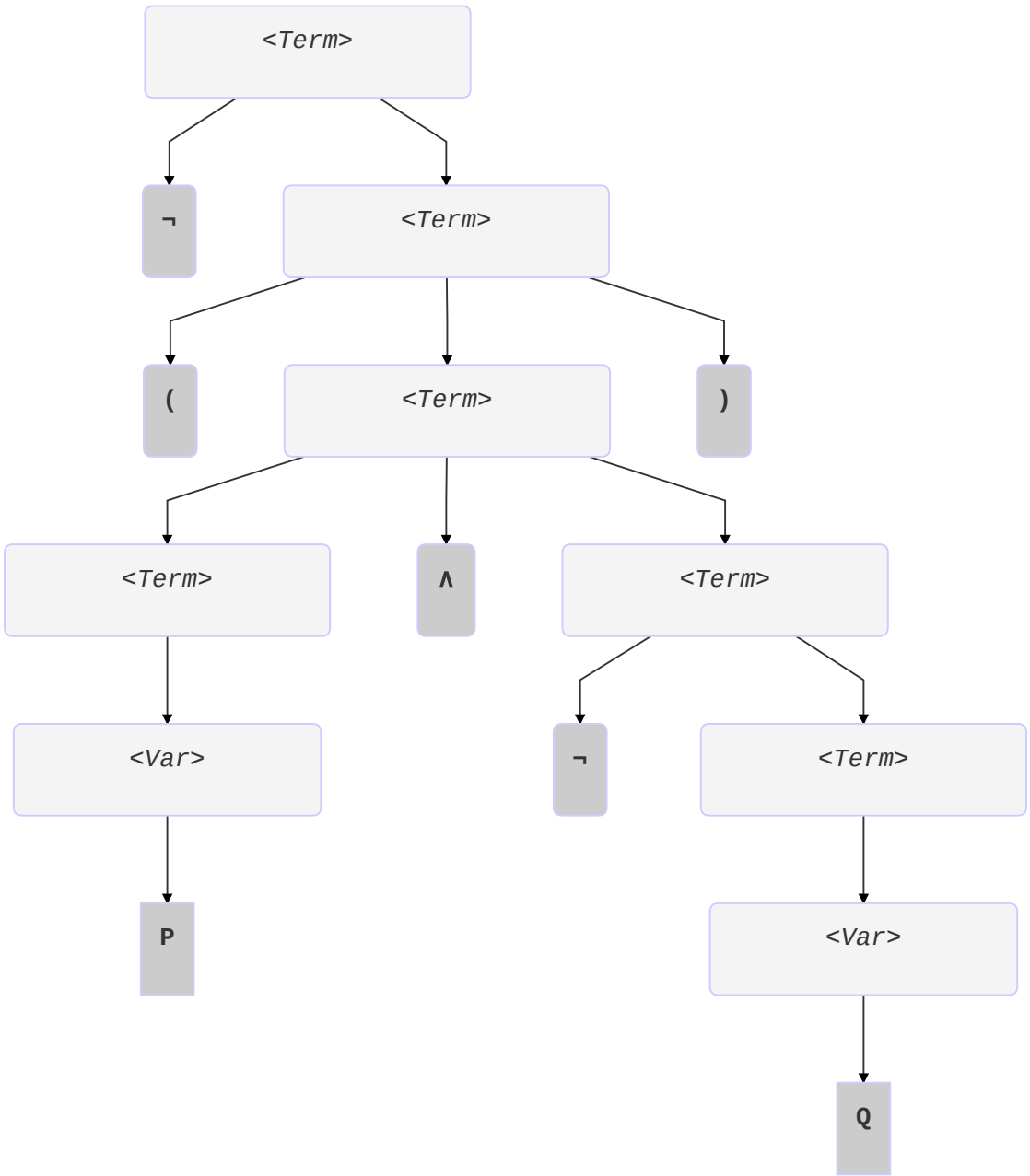
Γλώσσες Προγραμματισμού Ι: Λύσεις Κανονικής '17

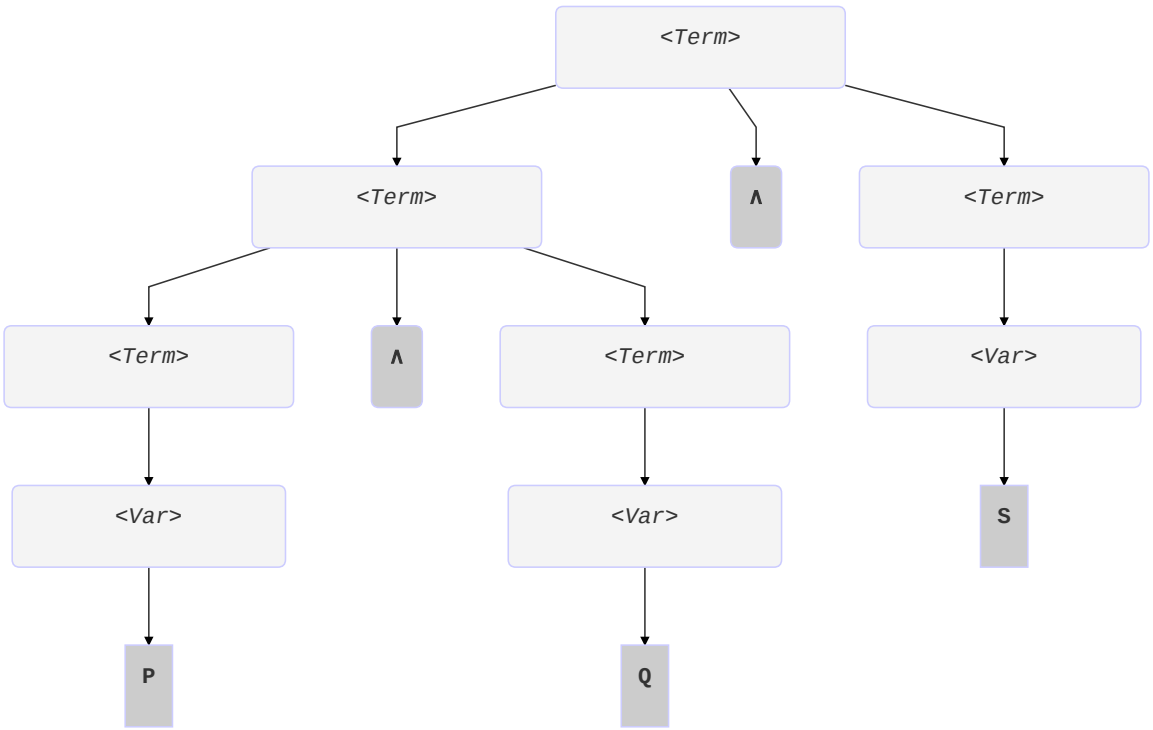
1. Γραμματικές

α) Το συντακτικό δέντρο για την έκφραση ¬(P∧¬Q):

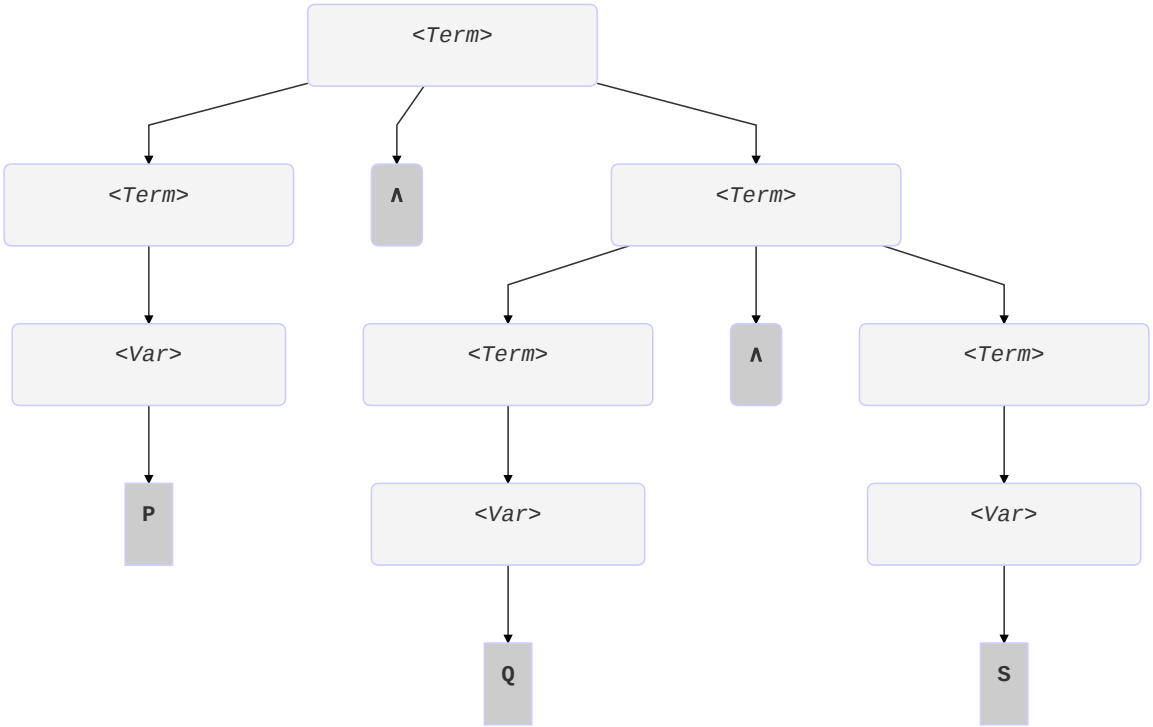


β) Για να είναι διφορούμενη η γραμματική θα πρέπει να υπάρχει μια συμβολοσειρά η οποία να αντιστοιχίζεται με παραπάνω από ένα συντακτικά δέντρα. Μία τέτοια συμβολοσειρά για τη δοθείσα γραμματική είναι η: P∧Q∧S

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**.

Έχοντας τα παραπάνω υπ'όψιν θέλουμε να τροποποιήσουμε τη δοθείσα γραμματική ώστε να ικανοποιούνται τα εξής κριτήρια:

- Σειρά προτεραιοτήτων τελεστών (από τη χαμηλότερη στην υψηλότερη):

  $\rightarrow$, $\vee$, $\wedge$, $\neg$
- Η σύζευξη και η διάζευξη να είναι αριστερά προσεταιριστικές
- Η συνεπαγωγή να είναι δεξιά προσεταιριστική

Έτσι καταλήγουμε στην παρακάτω γραμματική:

```
<Term>  ::= <Impl> | <Impl> → <Term>
<Impl>  ::= <Disj> | <Impl> ∨ <Disj>
<Disj>  ::= <Conj> | <Disj> ∧ <Conj>
<Conj>  ::= <Expr> | ¬ <Conj>
<Expr>  ::= <Var> | ⊥ | (<Term>)
<Var>   ::= P | Q | ... | S
```

EDIT: Το `<Conj> ::= <Expr> | ¬ <Expr>` έγινε `<Conj> ::= <Expr> | ¬ <Conj>`. Διόρθωση από Γ. Δάρα *~~2018/09/18*

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 (¬). Σε κάθε επίπεδο εμφωλιασμού, δεν επιτρέψαμε την αναφορά σε προηγούμενα επίπεδα, εκτός από την περίπτωση `(<Term>)`.

Για την προσεταιριστικότητα, επιβάλαμε πχ. στον τελεστή ∨ (αριστερά προσεταιριστικός), η επέκταση (δηλ. η εμφάνιση του `<Term>`) να γίνεται μόνο στο αριστερό μέρος.

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` ανήκει στην ίδια "ομάδα" με το στοιχείο που μπήκε **τελευταίο** στον accumulator. Αν ανήκει στην ίδια ομάδα, 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 ls x =
    let
        (* Αν είναι το 1ο στοιχείο που εξετάζουμε *)
        fun traverse (el, [nil]) =
            if el < x then [[el]] else [[el], []]

        (* Αλλιώς, αν έχουν μπει ήδη στοιχεία *)
        |   traverse (el, (lastNum :: lastGroup) :: rest) =
            if (lastNum < x) = (el < x) then
                (* Μπαίνει στην ίδια ομάδα *)
                (el :: lastNum :: lastGroup) :: rest
            else
                (* Δημιουργεί νέα ομάδα *)
                [el] :: (lastNum :: lastGroup) :: rest
        (* Τρέχουμε την traverse σειριακά στην ls μέσω foldl *)
        let result = foldl traverse [[]] ls
    in
        (* Αντιστρέφουμε τη σειρά ομάδων και των στοιχείων τους για να βγει το σωστό
αποτέλεσμα *)
        rev (map rev result)
    end
```

β)

Αλγόριθμος του Kadane [1]:

```
fun maxSumSublist L =
    let
        fun traverse (el, (max_till_prev, curr_max)) =
            let
                val max_here = Int.max(max_till_prev + el, el)
                val max = Int.max(curr_max, max_here)
            in
                (max_here, max)
            end
        val (_, result) = foldl traverse (0, 0) L
    in
        result
    end
```

3. Συμπερασμός τύπων στην ML

Α. `fun foo x y z = z (y x + 1)`

- x: a
- y: b = a -> int (αφού καλείται η y με όρισμα x: a, και μετά το αποτέλεσμα `y x` προστίθεται με το 1: int)
- z: c = int -> d (αφού καλείται με όρισμα το `(y x + 1)` που είναι int)
- foo: a -> b -> c -> d (αφού d είναι το αποτέλεσμα της `z (y x + 1)`)

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

Β. `fun bar x y = x (bar y x)`

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

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

Γ. `fun doh x y z = z (x y) (y + 1)`

- x: a = b -> e (αφού καλείται με όρισμα y: b) = int -> e
- y: b = int (αφού προστίθεται με το 1: int)
- z: c = e -> int -> f (αφού καλείται με ορίσματα `x y` : e, και `y + 1` : int)
- doh: a -> b -> c -> f (αφού f είναι το αποτέλεσμα της `z (x y) (y + 1)`)

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

Δ. `fun ugh x y z = x z (y z)`

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

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

4. Εμβέλειες

α) Απάντηση στις διαφάνειες (Ονόματα και εμβέλεια, διαφ. 28).

β) Παράδειγμα στις διαφάνειες (Ονόματα και εμβέλεια, διαφ. 32-34).

γ) Ο μεταγλωττιστής δεν μπορεί να γνωρίζει κατά τη μεταγλώττιση τον τύπο που θα έχει μία μεταβλητή με δυναμική εμβέλεια. Αυτό συμβαίνει επειδή μόνο κατά την εκτέλεση το πρόγραμμα μαθαίνει σε ποια τιμή αναφέρεται το όνομα κάθε μεταβλητής στο σημείο που αυτή χρησιμοποιείται.

Αυτό φαίνεται στο παρακάτω παράδειγμα: *(Σημείωση: κανονικά δεν μπορεί να γραφεί τέτοιο πρόγραμμα, αφού η ML δεν έχει δυναμική εμβέλεια.)*

```
fun f () = x
fun example () =
    let
        val x = 42
    in
        f ()
    end
```

Εδώ, η f επιστρέφει την τιμή του x. Αυτή η τιμή όμως δίνεται μόνο κατά την εκτέλεση, όπου καλείται η f στο σώμα της example και η x παίρνει την τιμή 42 μέσω δυναμικής εμβέλειας.

Ο τύπος της x μπορεί να είναι οποιοσδήποτε, και ο έλεγχός του δεν μπορεί να γίνει κατά τη μεταγλώττιση όσον αφορά τη συνάρτηση f.

5. Πέρασμα παραμέτρων

α) Κλήση με τιμή (by value)

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

β) Κλήση κατ' αναφορά (by reference)

f: x	f: y	f: A[0]	f: A[1]	main: k	main: A[0]	main: A[1]
1	3	3	0	1	3	0

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

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

δ) Κλήση κατ' όνομα (by name)

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

6. Προγραμματισμός σε Prolog

α)

```
running_sum([], [], _).
running_sum([X0 | X], [Y0 | Y], N) :-
    Y0 is X0 + N,
    running_sum(X, Y, Y0).

running_sum(L, S) :- running_sum(L, S, 0).
```

Το `N` εδώ αναπαριστά το άθροισμα των στοιχείων που βρίσκονταν πριν το `X0` στην **αρχική λίστα**. Ξεκινάει με τιμή 0, αφού αρχικά δεν υπάρχουν στοιχεία πριν την `L`.

Στη συνέχεια πρέπει να διαδίδεται η σωστή τιμή του `N`, ώστε να υπολογίζεται το `Y0` ως άθροισμα του στοιχείου που εξετάζεται τώρα και του `N`, του αθροίσματος των στοιχείων που προηγήθηκαν.

Αυτή η υλοποίηση είναι μόνο προς την "κατεύθυνση" των τρέχοντων αθροισμάτων S, αφού υπολογίζουμε κάθε φορά το `Y0` με την εντολή `Y0 is X0 + N`.

β) Το κατηγόρημα δεν είναι ευέλικτο, αφού λειτουργεί μόνο όταν είναι γνωστή η λίστα L. Μπορεί να χρησιμοποιηθεί με δύο γνωστές λίστες, όπου θα επιστρέψει true ή false, ανάλογα με το αν η S είναι η λίστα τρέχοντων αθροισμάτων της L.

Για να κάνουμε το κατηγόρημα να λειτουργεί και για την άλλη "κατεύθυνση", δηλαδή όταν είναι γνωστή μόνο η S και πρέπει να βρούμε την L, πρέπει να κάνουμε την εξής προσθήκη:

```
running_sum([], [], _).
running_sum([X0 | X], [Y0 | Y], N) :-
    (var(X0) ->
        X0 is Y0 - N,
        running_sum(X, Y, Y0).
     ;
        Y0 is X0 + N,
        running_sum(X, Y, Y0).
    ).

running_sum(L, S) :- running_sum(L, S, 0).
```

Υπενθύμιση: το `(condition -> if_true ; if_false)` *είναι το αντίστοιχο* `if-else` *της Prolog.*

Με τη χρήση του `var(X0)` μπορούμε να βρούμε ποια από τις δύο λίστες είναι γνωστές. Αν ισχύει το `var(X0)`, η `X0` είναι ελεύθερη μεταβλητή, άρα πρέπει να βρούμε τη λίστα των αριθμών από τη λίστα αθροισμάτων. Γνωρίζουμε την `Y0` που πρέπει να είναι το άθροισμα της `X0` με το `N`, άρα από αυτό βρίσκουμε ότι `X0 is Y0 - N`.

Αν δεν ισχύει το `var(X0)`, τότε η `X0` δεν είναι ελεύθερη μεταβλητή, άρα πρέπει να βρούμε τη λίστα αθροισμάτων. Έτσι, εδώ κάνουμε το ίδιο με το ερώτημα (α).

γ)

```
subseq([], []).
subseq([X | L], [X | S]) :-
    subseq(L, S).
subseq([_ | L], S) :-
    subseq(L, S).

subset_sum(L, Sum, S) :-
    subseq(L, S),
    sum_list(S, Sum).
```

Το `subseq/2` είναι το γνωστό κατηγόρημα που βρίσκει όλα τα πιθανά υποσύνολα μιας λίστας `L`.

Βάσει αυτού ορίζεται το κατηγόρημα `subset_sum/3`, αφού για να είναι αληθές αρκεί το `S` να είναι υποσύνολο της `L` και το άθροισμα των στοιχείων του `S` να είναι ίσο με το `Sum` (χρήση του `sum_list/2`).