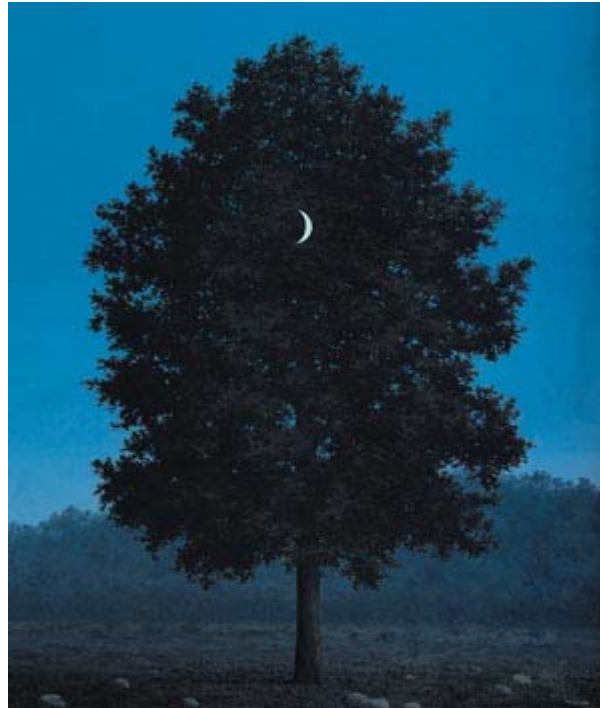


# Σύνταξη & Συντακτική Ανάλυση



Rene Magritte, *Le Seize Septembre*

Κωστής Σαγώνας <[kostis@cs.ntua.gr](mailto:kostis@cs.ntua.gr)>

# Μια γραμματική για τα Αγγλικά

---

Μια πρόταση αποτελείται από μια ουσιαστική φράση, ένα ρήμα, και μια ουσιαστική φράση

$$\langle S \rangle ::= \langle NP \rangle \langle V \rangle \langle NP \rangle$$

Μια ουσιαστική φράση αποτελείται από ένα άρθρο και ένα ουσιαστικό

$$\langle NP \rangle ::= \langle A \rangle \langle N \rangle$$

Ρήματα είναι τα εξής...

$$\langle V \rangle ::= \text{loves} \mid \text{hates} \mid \text{eats}$$

Άρθρα είναι τα εξής...

$$\langle A \rangle ::= \text{a} \mid \text{the}$$

Ουσιαστικά είναι τα εξής...

$$\langle N \rangle ::= \text{dog} \mid \text{cat} \mid \text{rat}$$

# Πως δουλεύει μια γραμματική

---

- Μια γραμματική είναι ένα σύνολο κανόνων που ορίζουν πώς κατασκευάζεται ένα **συντακτικό δένδρο**
- Ξεκινάμε βάζοντας το  $\langle S \rangle$  στη ρίζα του δένδρου
- Οι κανόνες της γραμματικής λένε πώς μπορούμε να προσθέσουμε παιδιά σε κάθε σημείο του δένδρου
- Για παράδειγμα, ο κανόνας

$$\langle S \rangle ::= \langle NP \rangle \langle V \rangle \langle NP \rangle$$

λέει ότι μπορούμε να προσθέσουμε κόμβους  $\langle NP \rangle$ ,  $\langle V \rangle$ , και  $\langle NP \rangle$ , με αυτή τη σειρά, ως παιδιά του κόμβου  $\langle S \rangle$

# Γραμματική για αριθμητικές εκφράσεις

---

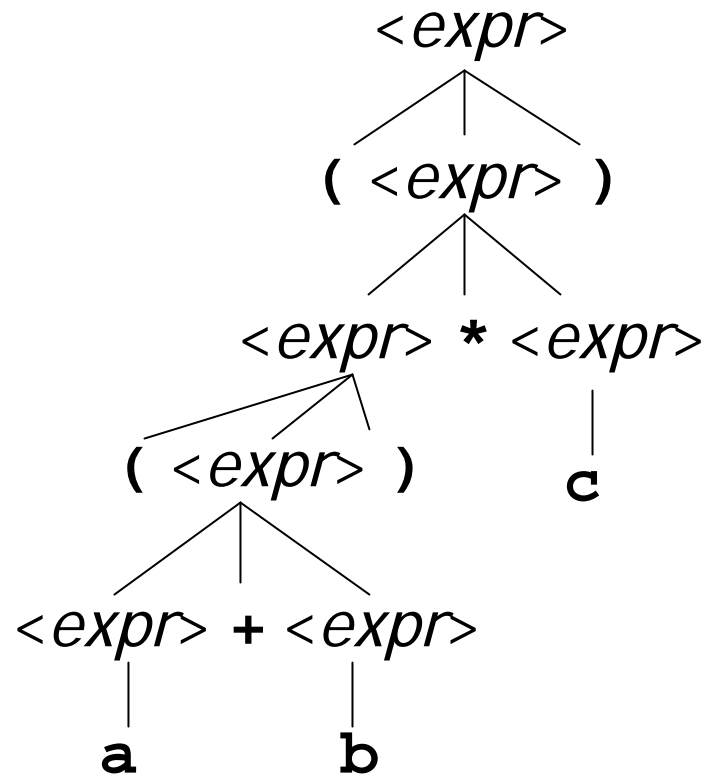
```
<expr> ::= <expr> + <expr>
          | <expr> * <expr>
          | ( <expr> )
          | a | b | c
```

Η παραπάνω γραμματική ορίζει ότι μια αριθμητική έκφραση μπορεί να είναι

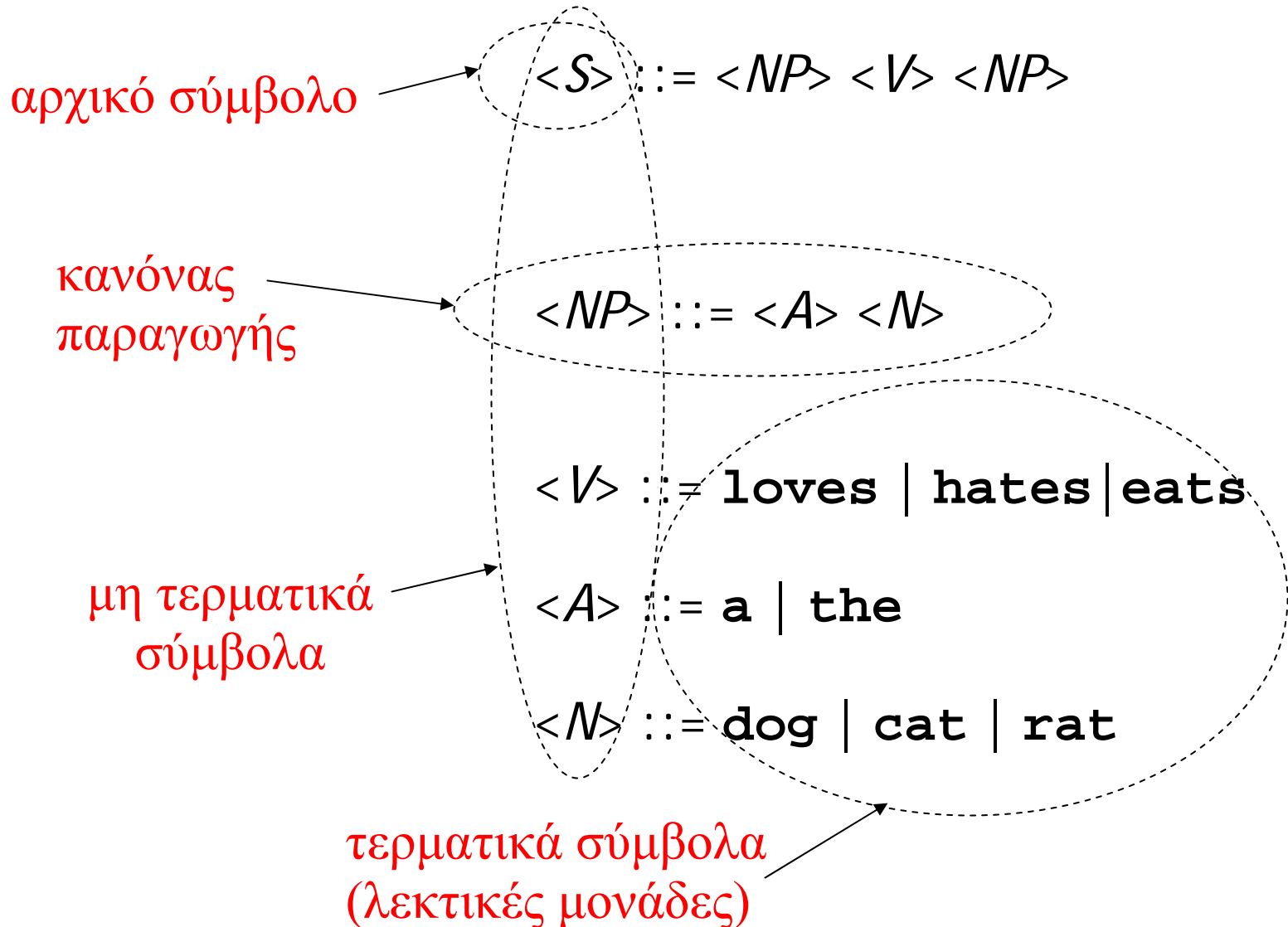
- το άθροισμα δύο άλλων εκφράσεων, ή
- το γινόμενο δύο εκφράσεων, ή
- μια έκφραση που περικλείεται από παρενθέσεις, ή
- κάποια από τις μεταβλητές **a**, **b**, ή **c**

# Συντακτικό δένδρο

---



# Συστατικά μιας γραμματικής



# Ορισμός γραμματικών σε μορφή Backus-Naur

---

Μια γραμματική σε μορφή Backus-Naur αποτελείται από

- Ένα σύνολο από **λεκτικές μονάδες (tokens)**
  - Συμβολοσειρές που αποτελούν τα μικρότερα αδιαίρετα κομμάτια της σύνταξης του προγράμματος
- Ένα σύνολο από **μη τερματικά σύμβολα (non-terminals)**
  - Συμβολοσειρές που εγκλείονται σε αγκύλες, π.χ.  $\langle NP \rangle$ , και αντιπροσωπεύουν κομμάτια του συντακτικού της γλώσσας
  - Δε συναντιούνται στο πρόγραμμα, αλλά είναι σύμβολα που βρίσκονται στο αριστερό μέρος κάποιων κανόνων της γραμματικής
- Το **αρχικό σύμβολο (start symbol)** της γραμματικής
  - Ένα συγκεκριμένο μη τερματικό σύμβολο που αποτελεί τη ρίζα του συντακτικού δένδρου για κάθε αποδεκτό από τη γλώσσα πρόγραμμα
- Ένα σύνολο από **κανόνες παραγωγής (production rules)**

# Κανόνες παραγωγής

---

- Οι κανόνες παραγωγής χρησιμοποιούνται για την κατασκευή του συντακτικού δένδρου
- Κάθε κανόνας έχει τη μορφή  $A ::= \Delta$ 
  - Το αριστερό μέρος  $A$  αποτελείται από ένα μη τερματικό σύμβολο
  - Το δεξί μέρος  $\Delta$  είναι μια ακολουθία από τερματικά (λεκτικές μονάδες) και μη τερματικά σύμβολα
- Κάθε κανόνας προσδιορίζει έναν πιθανό τρόπο κατασκευής του συντακτικού υποδένδρου που
  - έχει ως ρίζα του το μη τερματικό σύμβολο στο αριστερό μέρος  $A$  του κανόνα και
  - έχει ως παιδιά αυτής της ρίζας (με την ίδια σειρά εμφάνισης) τα σύμβολα στο δεξί μέρος  $\Delta$  του κανόνα



# Επιλογές στη γραφή των κανόνων παραγωγής

- Όταν υπάρχουν περισσότεροι από ένας κανόνες παραγωγής με το ίδιο αριστερό μέρος, μπορούμε να κάνουμε χρήση της παρακάτω συντομογραφίας
- Στη BNF γραμματική μπορούμε να δώσουμε το αριστερό μέρος, το διαχωριστή  $::=$ , και μετά μια ακολουθία από δεξιά μέρη που διαχωρίζονται από το ειδικό σύμβολο  $|$
- Οι δύο γραμματικές είναι ίδιες

```
<expr> ::= <expr> + <expr>  
         | <expr> * <expr>  
         | ( <expr> )  
         | a | b | c
```

```
<expr> ::= <expr> + <expr>  
<expr> ::= <expr> * <expr>  
<expr> ::= ( <expr> )  
<expr> ::= a  
<expr> ::= b  
<expr> ::= c
```

# Κανόνες παραγωγής του κενού

---

- Το ειδικό μη τερματικό *<empty>* χρησιμοποιείται σε περιπτώσεις που θέλουμε κάποιος κανόνας να μην παράγει τίποτα
- Για παράδειγμα, οι παρακάτω κανόνες ορίζουν τη δομή **if-then** των περισσότερων γλωσσών, η οποία επιτρέπει την ύπαρξη ενός προαιρετικού **else**

```
<if-stmt> ::= if <expr> then <stmt> <else-part>  
<else-part> ::= else <stmt> | <empty>
```

# Κατασκευή συντακτικών δένδρων

---

- Αρχίζουμε την κατασκευή βάζοντας το αρχικό σύμβολο της γραμματικής στη ρίζα του δένδρου
- Προσθέτουμε παιδιά σε κάθε μη τερματικό σύμβολο, χρησιμοποιώντας *κάποιον* από τους *κανόνες παραγωγής* της γλώσσας για το συγκεκριμένο μη τερματικό
- Η διαδικασία τερματίζει όταν όλα τα φύλλα του δένδρου αποτελούνται από λεκτικές μονάδες
- Η συμβολοσειρά που αντιστοιχεί στο δένδρο που κατασκευάσαμε βρίσκεται διαβάζοντας τα φύλλα του δένδρου από αριστερά προς τα δεξιά

# Παραδείγματα

```
<expr> ::= <expr> + <expr>  
         | <expr> * <expr>  
         | ( <expr> )  
         | a | b | c
```

- Τα συντακτικά δένδρα για τις παρακάτω εκφράσεις

a+b

(a+b)

(a+(b))

a+b\*c

- Η κατασκευή των συντακτικών δένδρων είναι η δουλειά του συντακτικού αναλυτή ενός μεταγλωττιστή
- Υπάρχουν διάφοροι αποδοτικοί αλγόριθμοι και εργαλεία για (ημι)αυτόματη κατασκευή του συντακτικού αναλυτή

# Τυπικός ορισμός σύνταξης γλωσσών

---

- Για να ορίσουμε τη σύνταξη των γλωσσών προγραμματισμού χρησιμοποιούμε γραμματικές
- Η γλώσσα που ορίζεται από μια γραμματική είναι το σύνολο των συμβολοσειρών για τα οποία η γραμματική μπορεί να παράξει συντακτικά δένδρα
- Τις περισσότερες φορές το σύνολο αυτό είναι άπειρο (παρόλο που η γραμματική είναι πεπερασμένη)
- Η κατασκευή μιας γραμματικής για μια γλώσσα μοιάζει λίγο με προγραμματισμό...

# Παράδειγμα κατασκευής γραμματικής (1)

---

Συνήθως γίνεται με χρήση της τεχνικής “διαίρει και βασίλευε” (divide and conquer)

- Παράδειγμα: κατασκευή της γλώσσας των δηλώσεων της Java (η οποία είναι παρόμοια με αυτή της C):
  - αρχικά, η δήλωση έχει ένα όνομα τύπου
  - στη συνέχεια μια ακολουθία από μεταβλητές που διαχωρίζονται με κόμματα (όπου κάθε μεταβλητή μπορεί να πάρει μια αρχική τιμή)
  - και στο τέλος ένα ερωτηματικό (semicolon)

```
float a;  
boolean a, b, c;  
int a = 1, b, c = 1 + 2;
```

## Παράδειγμα κατασκευής γραμματικής (2)

---

- Αρχικά, ας αγνοήσουμε την πιθανή ύπαρξη αρχικοποιητών:

```
<var-decl> ::= <type-name> <declarator-list> ;
```

- Ο κανόνας για τα ονόματα των πρωτόγονων τύπων (primitive types) της Java είναι απλούστατος:

```
<type-name> ::= boolean | byte | short | int  
                | long | char | float | double
```

Σημείωση: δεν παίρνουμε υπόψη κατασκευασμένους τύπους (constructed types): ονόματα κλάσεων, ονόματα διεπαφών (interfaces), και τύπους πινάκων

## Παράδειγμα κατασκευής γραμματικής (3)

---

- Η ακολουθία των μεταβλητών που διαχωρίζονται με κόμματα έχει ως εξής:

```
<declarator-list> ::= <declarator>  
                    | <declarator> , <declarator-list>
```

- Όπου ξανά, έχουμε προς το παρόν αγνοήσει τους πιθανούς αρχικοποιητές των μεταβλητών



## Παράδειγμα κατασκευής γραμματικής (4)

---

- Οι δηλωτές μεταβλητών, με ή χωρίς αρχικοποιήσεις, ορίζονται ως:

```
<declarator> ::= <variable-name>  
                | <variable-name> = <expr>
```

- Για ολόκληρη τη Java:
  - Πρέπει να επιτρέψουμε και ζεύγη από αγκύλες μετά το όνομα των μεταβλητών για τη δήλωση των πινάκων
  - Πρέπει επίσης να ορίσουμε και τη σύνταξη των αρχικοποιητών πινάκων
  - (Φυσικά θέλουμε και ορισμούς για τα μη τερματικά σύμβολα *<variable-name>* και *<expr>*)

# Τί αποτελεί λεκτική μονάδα (token);

---

- Όποια κομμάτια της γλώσσας επιλέξουμε να θεωρήσουμε ως μη κατασκευαζόμενα από μικρότερα κομμάτια
- Μεταβλητές (`i`, `j`), λέξεις κλειδιά (`if`), τελεστές (`==`, `++`), σταθερές (`123.4`), κ.λπ.
- Οι γραμματικές που έχουμε ορίσει δίνουν τη δομή των φράσεων (*phrase structure*): πως το πρόγραμμα κατασκευάζεται από μια σειρά λεκτικών μονάδων
- Πρέπει επιπλέον να ορίσουμε και τη λεκτική δομή (*lexical structure*): πως ένα αρχείο χωρίζεται σε λεκτικές μονάδες



# Δύο επιλογές ορισμού της λεκτικής δομής

---

- Με μια κοινή γραμματική
  - Οι χαρακτήρες είναι οι μοναδικές λεκτικές μονάδες
  - Συνήθως δεν ακολουθείται αυτή η επιλογή: κενά και σχόλια περιπλέκουν αρκετά τη γραμματική και την καθιστούν μη αναγνώσιμη
- Με ξεχωριστές γραμματικές
  1. Μία που ορίζει πώς προκύπτουν οι λεκτικές μονάδες από ένα αρχείο με χαρακτήρες
    - Η γραμματική αυτή συνήθως είναι μια **κανονική γραμματική (regular grammar)** και χρησιμοποιείται από το **λεκτικό αναλυτή (scanner)**
  2. Μία που ορίζει πώς προκύπτουν τα συντακτικά δένδρα από μία ακολουθία λεκτικών μονάδων
    - Η γραμματική αυτή συνήθως είναι μια **γραμματική ελεύθερη συμφραζομένων (context-free grammar)** και χρησιμοποιείται από το **συντακτικό αναλυτή (parser)**

# Ιστορικές σημειώσεις (1)

---

- Παλιά, κάποιες γλώσσες προγραμματισμού δε διαχώριζαν τη λεκτική από την φραστική δομή
  - Παλιές εκδόσεις της Fortran και της Algol επέτρεπαν κενά σε οποιοδήποτε σημείο, ακόμα και στο μέσο μιας λέξης κλειδί!
  - Άλλες γλώσσες, π.χ. η PL/I, επιτρέπουν τη χρήση λέξεων κλειδιών ως μεταβλητές
    - (Το ίδιο συμβαίνει και στην ML, αλλά εκεί δεν αποτελεί πρόβλημα.)
- Τα παραπάνω δεν είναι καλή ιδέα...
  - προσθέτουν δυσκολία στην λεκτική και συντακτική ανάλυση και
  - μειώνουν την αναγνωσιμότητα των προγραμμάτων που υλοποιούν αυτές τις φάσεις της μεταγλώττισης

## Ιστορικές σημειώσεις (2)

- Κάποιες γλώσσες έχουν **λεκτική δομή σταθερής μορφής** (*fixed-format*)—τα κενά είναι σημαντικά
  - Μία εντολή ανά γραμμή (π.χ. της διάτρητης κάρτας)
  - Οι πρώτες 7 θέσεις κάθε γραμμής για την ταμπέλα (label)
- Οι πρώτες διάλεκτοι της Fortran, Cobol, και της Basic
- Σχεδόν οι περισσότερες μοντέρνες γλώσσες είναι **ελεύθερης μορφής** (*free-format*): τα κενά αγνοούνται
  - Π.χ. Algol, Pascal, Java, ...
  - Μερικές άλλες (C, C++) διαφέρουν λίγο λόγω του προεπεξεργαστή
  - Σε κάποιες όμως (Python, Haskell), τα κενά είναι σημαντικά



# Άλλες μορφές γραμματικών

---

- Μικρές διαφοροποιήσεις της μορφής Backus-Naur (BNF)
  - Χρήση  $\rightarrow$  ή  $=$  αντί για  $::=$
  - Όχι  $\langle \rangle$  αλλά κάποιο ειδικό font ή χρήση αποστρόφων για τις λεκτικές μονάδες ώστε να ξεχωρίζονται εύκολα από τα μη τερματικά σύμβολα
- Επεκτάσεις της μορφής Backus-Naur (EBNF)
  - Πρόσθετος συμβολισμός για την απλοποίηση κάποιων κανόνων:
    - $\{x\}$  υποδηλώνει μηδέν ή περισσότερες επαναλήψεις του  $x$
    - $[x]$  υποδηλώνει ότι το  $x$  είναι προαιρετικό (δηλαδή  $x \mid \langle empty \rangle$ )
    - $()$  για ομαδοποίηση
    - $|$  οπουδήποτε για να υποδηλώσει επιλογή
    - Αποστρόφους γύρω από τις λεκτικές μονάδες ούτως ώστε να ξεχωρίζονται από τα παραπάνω μετασύμβολα
- Συντακτικά διαγράμματα

# Παραδείγματα EBNF

---

$\langle \text{if-stmt} \rangle ::= \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stmt} \rangle [\text{else } \langle \text{stmt} \rangle]$

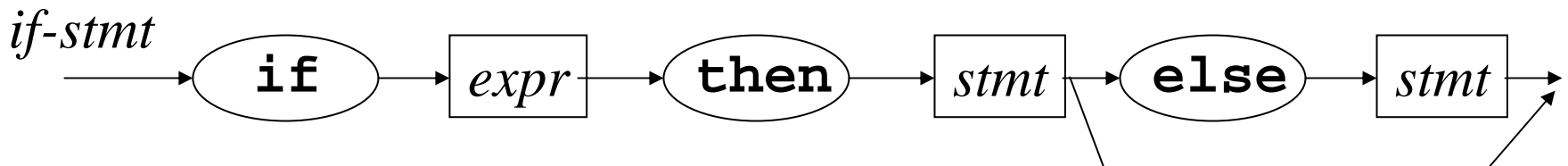
$\langle \text{stmt-list} \rangle ::= \langle \text{stmt} \rangle \{ ; \langle \text{stmt} \rangle \}$

$\langle \text{thing-list} \rangle ::= (\langle \text{stmt} \rangle \mid \langle \text{decl} \rangle) \{ ; (\langle \text{stmt} \rangle \mid \langle \text{decl} \rangle) \}$

# Συντακτικά διαγράμματα (1)

- Έστω ότι έχουμε μια γραμματική σε EBNF
- Ο κάθε κανόνας παραγωγής μετατρέπεται σε μια σειρά από κουτιά
  - Ορθογώνια για τα μη τερματικά σύμβολα
  - Οβάλ για τα τερματικά σύμβολα
  - Τα παραπάνω ενώνονται με βέλη
  - (Πιθανώς κάποια βέλη να παρακάμπτουν κάποια από τα κουτιά.)

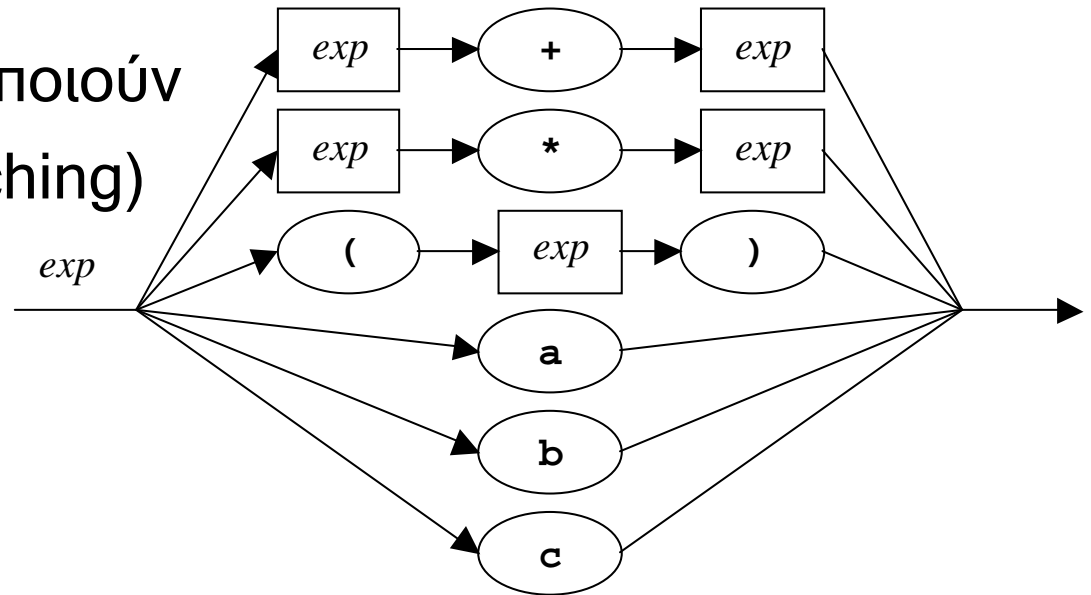
$\langle if-stmt \rangle ::= \mathbf{if} \langle expr \rangle \mathbf{then} \langle stmt \rangle [ \mathbf{else} \langle stmt \rangle ]$





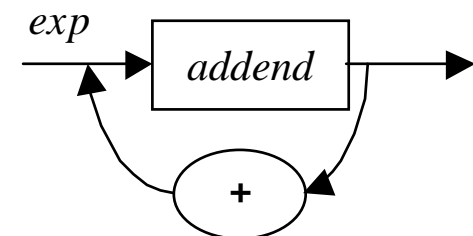
## Συντακτικά διαγράμματα (2)

- Πολλαπλοί κανόνες παραγωγής χρησιμοποιούν διακλαδώσεις (branching)



- Η επανάληψη υποδηλώνεται με χρήση βρόχων

$\langle exp \rangle ::= \langle addend \rangle \{ + \langle addend \rangle \}$



# Παράδειγμα διαφορετικού συμβολισμού EBNF

---

*WhileStatement.*

**while** ( *Expression* ) *Statement*

*DoStatement.*

**do** *Statement* **while** ( *Expression* );

*ForStatement.*

**for** ( *ForInit*<sub>opt</sub> ; *Expression*<sub>opt</sub> ; *ForUpdate*<sub>opt</sub> )  
*Statement*

από το βιβλίο *The Java™ Language Specification*, James Gosling *et al.*

# Από τη Σύνταξη προς τη Σημασιολογία

# Τρεις “ισοδύναμες” γραμματικές

---

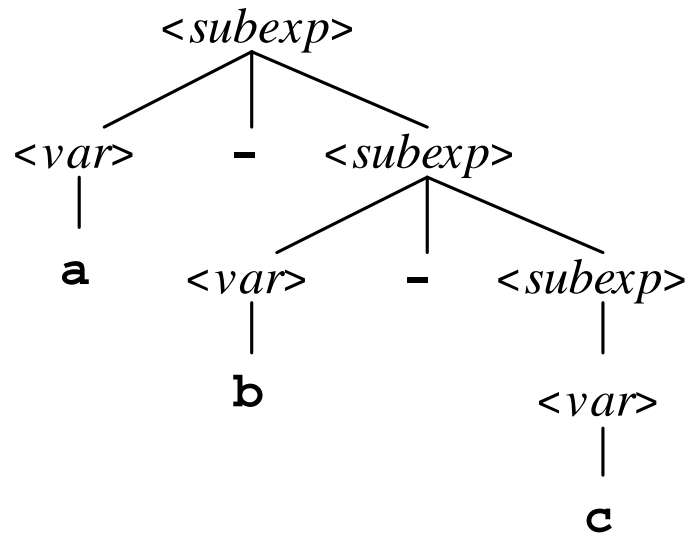
G1:  $\langle subexp \rangle ::= \langle subexp \rangle - \langle subexp \rangle \mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$

G2:  $\langle subexp \rangle ::= \langle var \rangle - \langle subexp \rangle \mid \langle var \rangle$   
 $\langle var \rangle ::= \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$

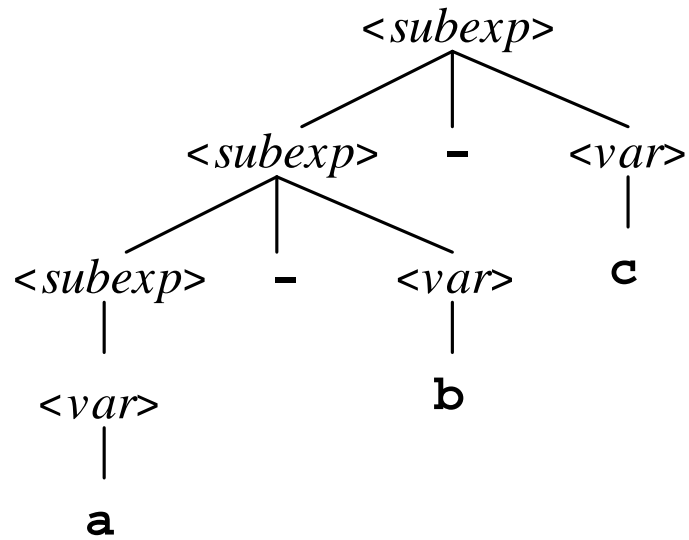
G3:  $\langle subexp \rangle ::= \langle subexp \rangle - \langle var \rangle \mid \langle var \rangle$   
 $\langle var \rangle ::= \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$

- Και οι τρεις γραμματικές ορίζουν την ίδια γλώσσα: τη γλώσσα όλων των συμβολοσειρών που περιλαμβάνουν ένα ή περισσότερα **a**, **b**, ή **c** τα οποία διαχωρίζονται μεταξύ τους από ένα μείον. Αλλά...

G2 parse tree:



G3 parse tree:



# Γιατί είναι σημαντικά τα συντακτικά δένδρα;

---

- Θέλουμε η δομή του συντακτικού δένδρου να αντικατοπτρίζει τη σημασιολογία της συμβολοσειράς που αντιπροσωπεύει
- Αυτό κάνει το σχεδιασμό της γλώσσας πιο δύσκολο:
  - ενδιαφερόμαστε για τη δομή του κάθε συντακτικού δένδρου
  - όχι μόνο για τη συμβολοσειρά των φύλλων του

Τα συντακτικά δένδρα είναι το μέρος που η σύνταξη αρχίζει να συναντά τη σημασιολογία των γλωσσών

# Τελεστές (operators)

---

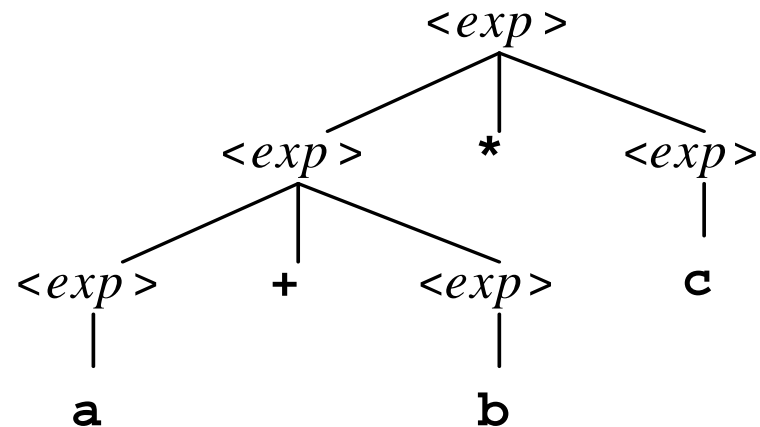
- Τελεστές χρησιμοποιούνται για λειτουργίες που γίνονται συχνά, π.χ. πρόσθεση, αφαίρεση, πολλαπλασιασμό, ...
- Ο όρος τελεστής αναφέρεται τόσο στη λεκτική μονάδα (π.χ. +, \*) όσο και στη λειτουργία αυτή καθ' αυτή
  - **Μοναδιαίοι** τελεστές δέχονται ένα όρισμα:  $-42$
  - **Διαδικοί** τελεστές δέχονται δύο ορίσματα:  $4+2$
  - **Τριαδικοί** τελεστές δέχονται τρία ορίσματα:  $a?b:c$
- Στις περισσότερες γλώσσες
  - Οι δυαδικοί τελεστές γράφονται σε infix μορφή: π.χ.  $4+2$
  - Οι μοναδιαίοι τελεστές γράφονται σε prefix ( $-42$ ) ή σε postfix μορφή ( $i++$ )

# Προτεραιότητα (precedence) τελεστών

- Έστω η γραμματική G4:

```
<exp> ::= <exp> + <exp>
         | <exp> * <exp>
         | ( <exp> )
         | a | b | c
```

- Ένα συντακτικό δένδρο για τη συμβολοσειρά  $a+b*c$  είναι το



- Στο δένδρο αυτό η πρόσθεση γίνεται πριν από τον πολλαπλασιασμό, κάτι που δεν είναι σε αρμονία με τις συνήθεις προτεραιότητες των τελεστών + και \*



# Προτεραιότητα τελεστών στη γραμματική

---

- Για να έχουμε τη σωστή προτεραιότητα τελεστών, αλλάζουμε τη γραμματική με τέτοιο τρόπο ώστε ο τελεστής με την μεγαλύτερη προτεραιότητα να καταλήγει “πιο κάτω” στο συντακτικό δένδρο

G5:

$$\begin{array}{l} \langle expr \rangle ::= \langle expr \rangle + \langle expr \rangle \mid \langle mulexpr \rangle \\ \langle mulexpr \rangle ::= \langle mulexpr \rangle * \langle mulexpr \rangle \\ \quad \quad \quad \mid ( \langle expr \rangle ) \\ \quad \quad \quad \mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \end{array}$$

# Παραδείγματα προτεραιότητας τελεστών

---

- C (15 επίπεδα προτεραιότητας—πάρα πολλά;)

```
a = b < c ? * p + b * c : 1 << d ( )
```

- Pascal (5 επίπεδα—όχι αρκετά;)

```
a <= 0 or 100 <= a
```

**Συντακτικό λάθος!**

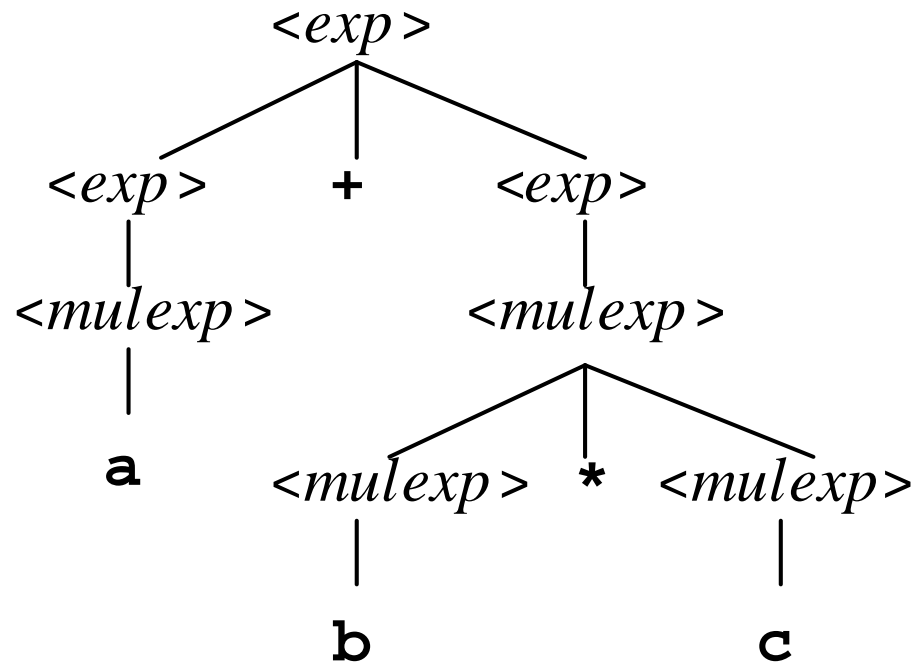
- Smalltalk (1 επίπεδο για όλους τους δυαδικούς τελεστές)

```
a + b * c
```

# Συντακτικό δένδρο με σωστή προτεραιότητα

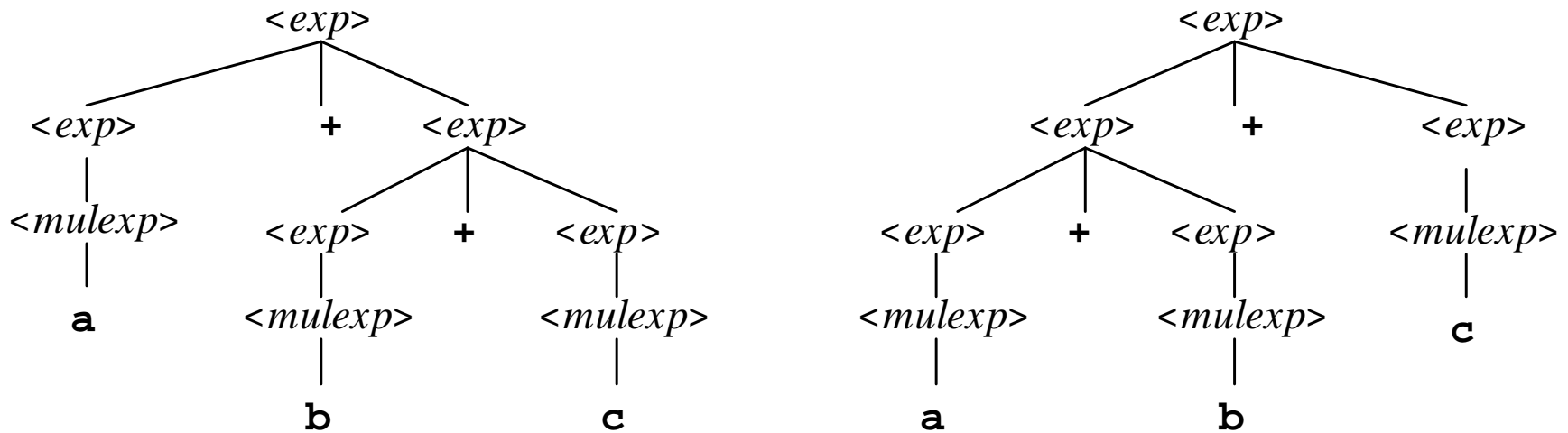
---

G5 parse tree:



Η γραμματική G5 παράγει μόνο αυτό το δένδρο για  $a+b*c$   
Αναγνωρίζει την ίδια γλώσσα με τη G4, αλλά δεν παράγει πλέον δένδρα με λάθος προτεραιότητα τελεστών

# Προσεταιριστικότητα (associativity) τελεστών



Η γραμματική G5 παράγει τα παραπάνω δένδρα για  $a+b+c$

Το πρώτο από αυτά δεν αντικατοπτρίζει τη συνήθη  
προσεταιριστικότητα του τελεστή  $+$

# Προσεταιριστικότητα τελεστών

---

- Έχουν χρησιμότητα όταν η σειρά της αποτίμησης δεν καθορίζεται από παρενθέσεις ή προτεραιότητα τελεστών
- Οι **αριστερά προσεταιριστικοί** τελεστές ομαδοποιούν από αριστερά προς τα δεξιά:  $a+b+c+d = ((a+b)+c)+d$
- Οι **δεξιά προσεταιριστικοί** τελεστές ομαδοποιούν από δεξιά προς τα αριστερά:  $a+b+c+d = a+(b+(c+d))$
- Στις περισσότερες γλώσσες, οι περισσότεροι τελεστές είναι αριστερά προσεταιριστικοί, αλλά υπάρχουν και εξαιρέσεις

# Παραδείγματα προσηταιριστικότητας

---

- C

**a<<b<<c** — τελεστές είναι αριστερά προσηταιριστικοί  
**a=b=0** — δεξιά προσηταιριστικός (ανάθεση)

- ML

**3-2-1** — τελεστές είναι αριστερά προσηταιριστικοί  
**1::2::nil** — δεξιά προσηταιριστικός (κατασκευή λίστας)

- Fortran

**a/b\*c** — τελεστές είναι αριστερά προσηταιριστικοί  
**a\*\*b\*\*c** — δεξιά προσηταιριστικός (ύψωση σε δύναμη)

# Προσεταιριστικότητα στη γραμματική

G5:

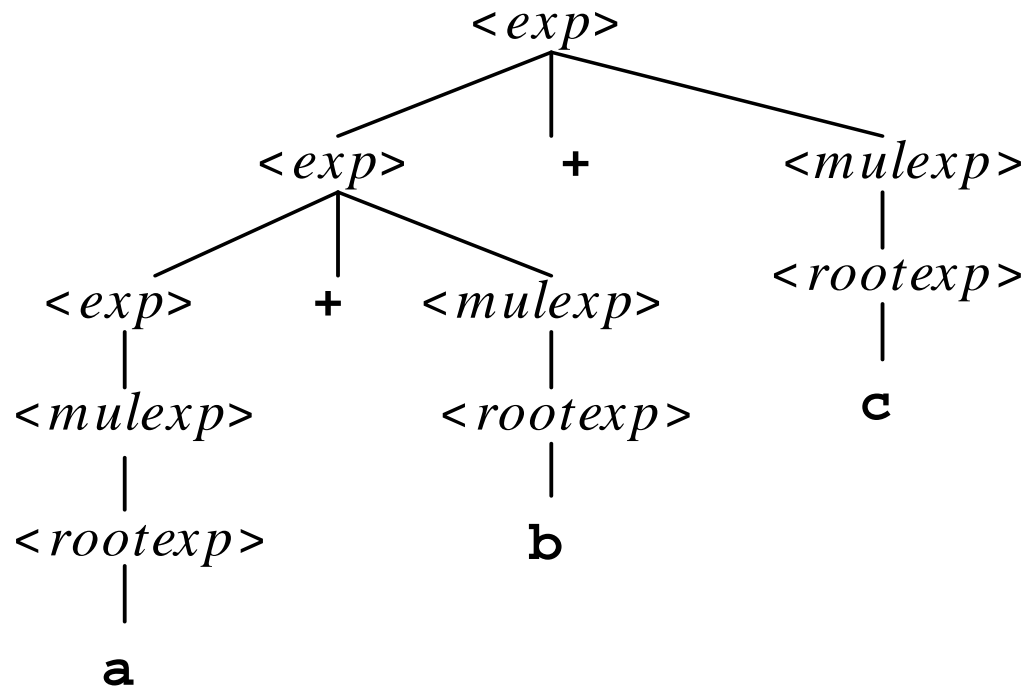
$$\begin{aligned} \langle expr \rangle & ::= \langle expr \rangle + \langle expr \rangle \mid \langle mulexp \rangle \\ \langle mulexp \rangle & ::= \langle mulexp \rangle * \langle mulexp \rangle \\ & \mid ( \langle expr \rangle ) \\ & \mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \end{aligned}$$

- Για να διορθώσουμε το πρόβλημα, τροποποιούμε τη γραμματική ώστε να κάνουμε δένδρα με τελεστές + να μεγαλώνουν προς τα αριστερά (παρόμοια για τον \*)

G6:

$$\begin{aligned} \langle expr \rangle & ::= \langle expr \rangle + \langle mulexp \rangle \mid \langle mulexp \rangle \\ \langle mulexp \rangle & ::= \langle mulexp \rangle * \langle rootexp \rangle \mid \langle rootexp \rangle \\ \langle rootexp \rangle & ::= ( \langle expr \rangle ) \\ & \mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \end{aligned}$$

# Δένδρο με σωστή προσειριστικότητα



Η γραμματική  $G_6$  παράγει μόνο αυτό το δένδρο για  $a+b+c$

Παράγει την ίδια γλώσσα με τη  $G_5$ , αλλά δεν παράγει

πλέον δένδρα με λάθος προσειριστικότητα τελεστών



# Διφορούμενη ερμηνεία (ambiguity)

---

- Η γραμματική G4 είναι **διφορούμενη (ambiguous)**: διφορούμενη σημαίνει ότι παράγει περισσότερα από ένα συντακτικά δένδρα για την ίδια συμβολοσειρά
- Όμως η επιδιόρθωση των προβλημάτων προτεραιότητας και προσηταιριστικότητας των τελεστών εξαφάνισε όλη την ασάφεια στη συγκεκριμένη γραμματική
- Αυτό είναι επιθυμητό: το συντακτικό δένδρο υποδηλώνει τη σημασιολογία του προγράμματος και δε θέλουμε αυτή να είναι διφορούμενη
- Όμως υπάρχουν και άλλοι λόγοι που μια γραμματική είναι διφορούμενη, όχι μόνο λόγοι σχετικοί με τους τελεστές

# Το πρόβλημα του “ξεκρέμαστου else”

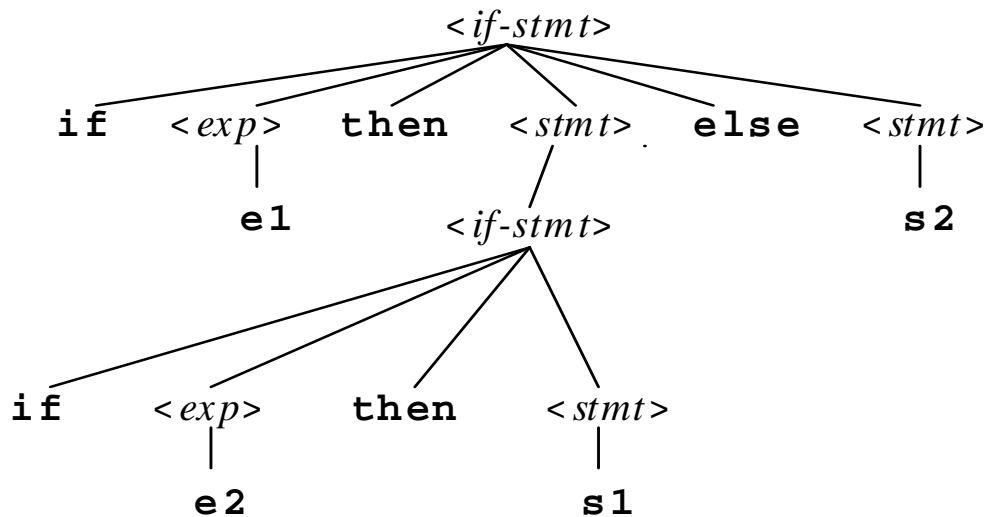
---

```
<stmt> ::= <if-stmt> | s1 | s2
<if-stmt> ::= if <expr> then <stmt> else <stmt>
            | if <expr> then <stmt>
<expr> ::= e1 | e2
```

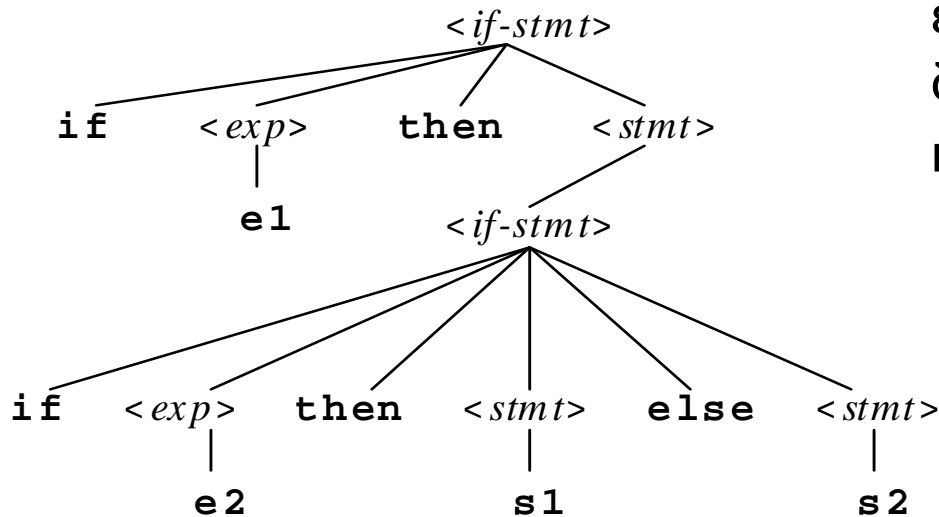
- Η γραμματική αυτή είναι διφορούμενη ως προς το “ξεκρέμαστο else” (“dangling-else ambiguity”).
- Η παρακάτω εντολή

```
if e1 then if e2 then s1 else s2
```

έχει δύο συντακτικά δένδρα



Οι περισσότερες γλώσσες που έχουν αυτό το πρόβλημα επιλέγουν το κάτω συντακτικό δένδρο: το `else` πηγαίνει με το κοντινότερο αταίριαστο `then`



# Διόρθωση του προβλήματος (1)

```
<stmt> ::= <if-stmt> | s1 | s2  
<if-stmt> ::= if <expr> then <stmt> else <stmt>  
           | if <expr> then <stmt>  
<expr> ::= e1 | e2
```

- Θέλουμε να επιβάλλουμε ότι εάν αυτό επεκταθεί σε ένα **if**, τότε το **if** πρέπει ήδη να έχει το δικό του **else**.  
Πρώτα, δημιουργούμε ένα νέο μη τερματικό *<full-stmt>* που παράγει όλες τις εντολές που παράγονται από το *<stmt>*, αλλά απαγορεύει τις **if** εντολές χωρίς **else**

```
<full-stmt> ::= <full-if> | s1 | s2  
<full-if> ::= if <expr> then <full-stmt> else <full-stmt>
```

## Διόρθωση του προβλήματος (2)

---

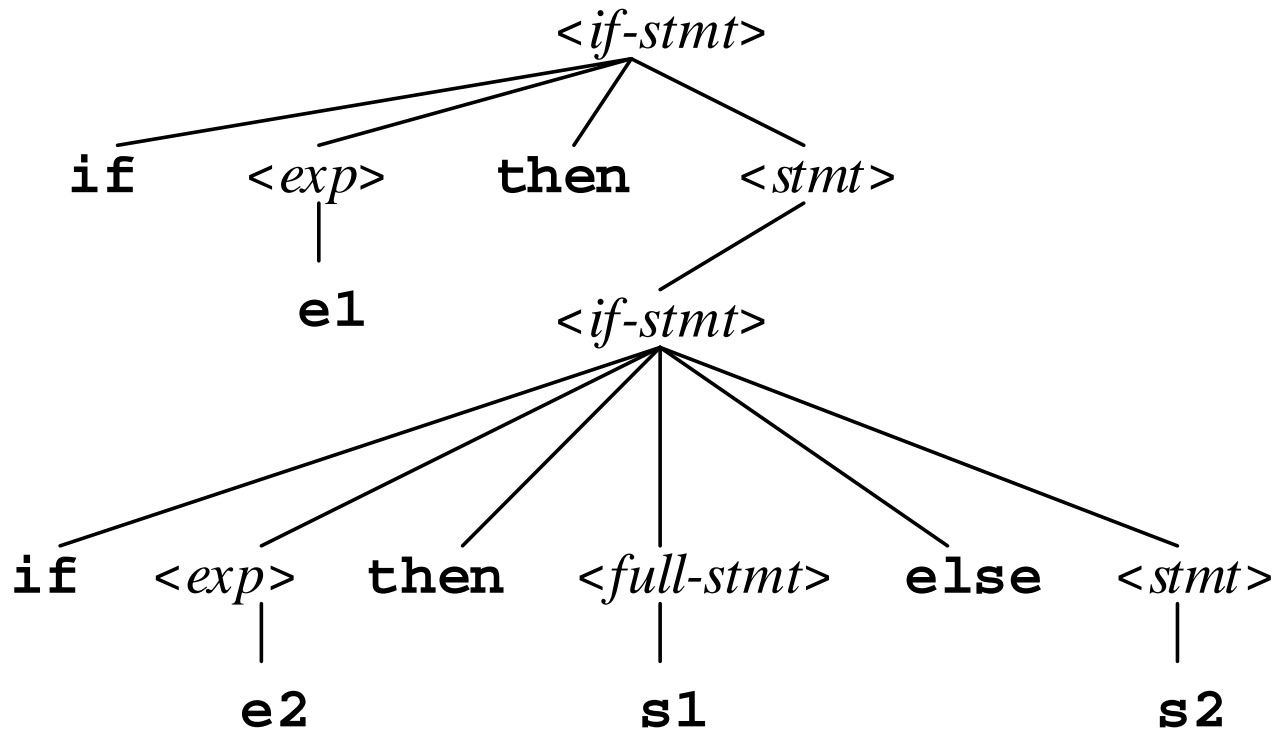
- Μετά χρησιμοποιούμε το νέο μη τερματικό εδώ

```
<stmt> ::= <if-stmt> | s1 | s2  
<if-stmt> ::= if <expr> then <full-stmt> else <stmt>  
           | if <expr> then <stmt>  
<expr> ::= e1 | e2
```

- Το αποτέλεσμα είναι ότι η παραπάνω γραμματική μπορεί να ταιριάζει ένα **else** με ένα **if** μόνο όταν όλα τα κοντινά **if** έχουν ήδη κάποιο **else** ως ταίρι τους.

# Τώρα παράγουμε μόνο το συντακτικό δένδρο

---



# Ξεκρέμαστα `else` και αναγνωσιμότητα

---

- Διορθώσαμε τη γραμματική, αλλά...
- Το πρόβλημα στη γραμματική αντικατοπτρίζει κάποιο πρόβλημα στη γλώσσα, την οποία δεν αλλάξαμε
- Μια ακολουθία από `if-then-else` δεν είναι εύκολα αναγνώσιμη, ειδικά εάν κάποια από τα `else` λείπουν

```
int a=0;  
if (0==0)  
    if (0==1) a=17;  
else a=42;
```

Ποια είναι η τιμή του `a` μετά την εκτέλεση του προγράμματος;

# Καλύτερα στυλ προγραμματισμού

---

```
int a=0;
if (0==0)
    if (0==1) a=17;
    else a=42;
```

Καλύτερο: σωστή στοίχιση

```
int a=0;
if (0==0) {
    if (0==1) a=17;
    else a=42;
}
```

Ακόμα καλύτερο: η χρήση μπλοκ δείχνει καθαρά τη δομή του κώδικα



# Γλώσσες χωρίς ξεκρέμαστα `else`

---

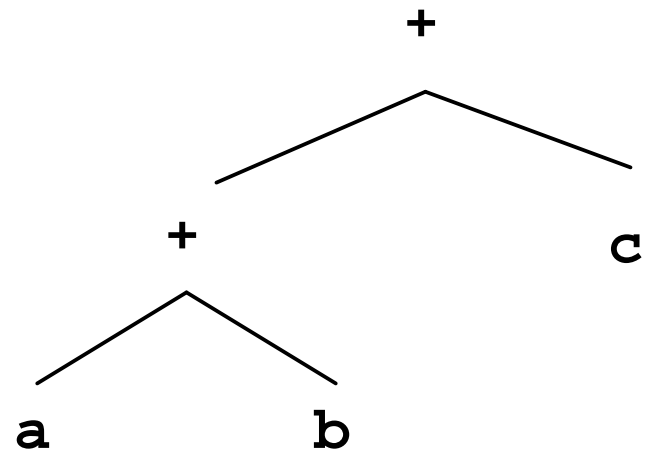
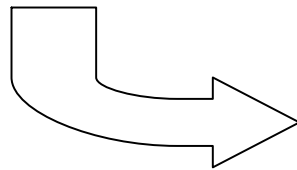
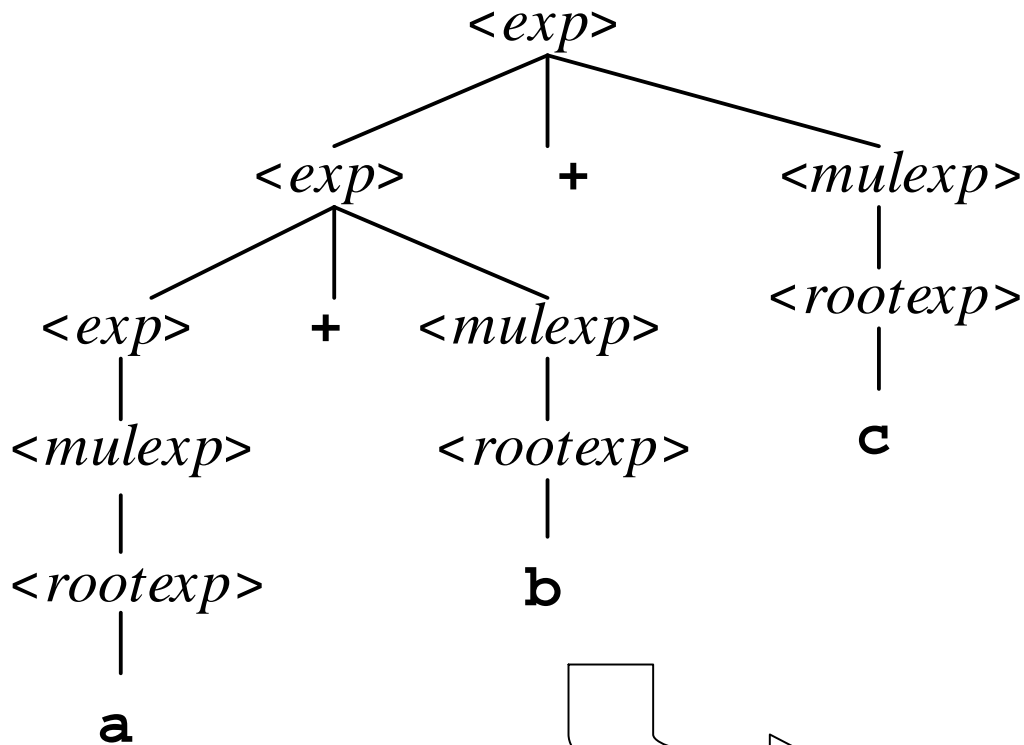
- Μερικές γλώσσες ορίζουν τα `if-then-else` με τρόπο τέτοιο που να επιβάλλει τη σαφήνεια στη χρήση τους
- Η Algol δεν επιτρέπει μέσα στο `then` να αρχίζει άμεσα κάποιο άλλο `if`
  - (αλλά μπορεί να αρχίζει ένα μπλοκ με ένα άλλο `if`)
- Η Algol 68 επιβάλλει σε κάθε `if` να τερματίζεται με ένα `fi` (υπάρχουν επίσης `do-od` και `case-esac`)
- Η Ada επιβάλλει σε κάθε `if` να τερματίζεται με ένα `end if`

# Γραμματική για ολόκληρη τη γλώσσα

---

- Κάθε ρεαλιστική γλώσσα περιέχει πολλά μη τερματικά σύμβολα
- Ειδικά εάν από τη γραμματική έχει εξαλειφθεί η ασάφεια
- Τα επιπλέον μη τερματικά καθορίζουν τον τρόπο με τον οποίο προκύπτει ένα μοναδικό συντακτικό δένδρο
- Όταν κατασκευαστεί το συντακτικό δένδρο, τα επιπλέον μη τερματικά δεν έχουν κάποια χρησιμότητα πλέον
- Οι υλοποιήσεις των γλωσσών συνήθως αποθηκεύουν μια συνεπτυγμένη μορφή του συντακτικού δένδρου, η οποία ονομάζεται **αφηρημένο συντακτικό δένδρο**

# Συγκεκριμένο συντακτικό δένδρο



# Αφηρημένο συντακτικό δένδρο

# Συμπερασματικά

---

- Για τον ορισμό του συντακτικού (και όχι μόνο!) των γλωσσών προγραμματισμού χρησιμοποιούμε γραμματικές
- Οι γραμματικές ορίζουν
  - το ποια είναι τα επιτρεπτά προγράμματα μιας γλώσσας, αλλά και
  - το συντακτικό δένδρο για αυτά τα προγράμματα
  - το δένδρο με τη σειρά του ορίζει τη σειρά εκτέλεσης των εντολών
  - και κατά συνέπεια συνεισφέρει στον ορισμό της σημασιολογίας
- Υπάρχει ισχυρή σύνδεση μεταξύ θεωρίας και πράξης
  - Δύο γραμματικές, δύο μέρη του μεταγλωττιστή (compiler)
  - Σημείωση: Υπάρχουν προγράμματα, γεννήτριες λεκτικών και συντακτικών αναλυτών (lexers and parser generators), που μπορούν να δημιουργήσουν αυτόματα τον κώδικα του λεκτικού και του συντακτικού αναλυτή από γραμματικές μιας γλώσσας