Μεταγλωττιστές

Μετάφραση Οδηγούμενη από το Συντακτικό

Δημήτρης Μιχαήλ

Τμήμα Πληροφορικής και Τηλεματικής
Χαροκόπειο Πανεπιστήμιο

Μετάφραση Οδηγούμενη από το Συντακτικό

Syntax-Directed Translation

Μια πολύ σημαντική τεχνική για την ανάπτυξη μεταγλωττιστών ονομάζεται:

"μετάφραση οδηγούμενη από το συντακτικό"

(i) Συσχετίζουμε κάθε *τερματικό* και *μη-τερματικό* με κάποιες *ιδιότητες* (attributes).

(ii) Διατυπώνουμε *εξισώσεις ιδιοτήτων* ή *σημασιολογικούς κανόνες* που εκφράζουν το πως οι τιμές των ιδιοτήτων σχετίζονται με τους κανόνες της γραμματικής.

Μετάφραση Οδηγούμενη από το Συντακτικό

Syntax-Directed Translation

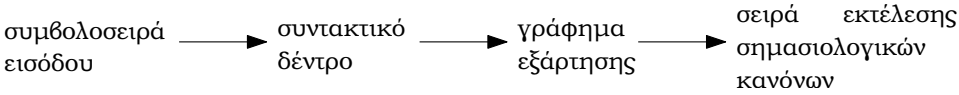
Υπάρχουν δύο τρόποι αναπαράστασης της συσχέτισης σημασιολογικών κανόνων με τους κανόνες της γραμματικής:

(i) **Ορισμοί Οδηγούμενοι από το Συντακτικό** (syntax-directed definitions).
Είναι υψηλού επιπέδου και κρύβουν λεπτομέρειες υλοποίησης.

(ii) **Μεταφραστικά Σχήματα** (translation schemes).
Περιγράφουν την σειρά εκτέλεσης των σημασιολογικών κανόνων, οπότε περιέχουν περισσότερες λεπτομέρειες υλοποίησης.

Μετάφραση Οδηγούμενη από το Συντακτικό

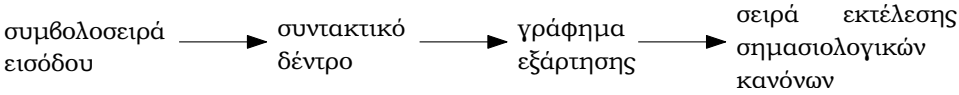
Syntax-Directed Translation



Οι σημασιολογικές ρουτίνες που αντιστοιχούν σε κανόνες της γραμματικής είναι υπεύθυνες για την διαχείριση του πίνακα συμβόλων, έλεγχο τύπων, παραγωγή μηνυμάτων λάθους, παραγωγή κώδικα, κ.τ.λ.

Μετάφραση Οδηγούμενη από το Συντακτικό

Syntax-Directed Translation



Οι σημασιολογικές ρουτίνες που αντιστοιχούν σε κανόνες της γραμματικής είναι υπεύθυνες για την διαχείριση του πίνακα συμβόλων, έλεγχο τύπων, παραγωγή μηνυμάτων λάθους, παραγωγή κώδικα, κ.τ.λ.

1. Το συντακτικό δέντρο και το γράφημα εξαρτήσεων δεν πρέπει να κατασκευαστούν υποχρεωτικά, είναι θέμα υλοποίησης.
2. Ανάλογα με τις ανάγκες του συστήματος μπορούμε να υλοποιήσουμε αυτή την τεχνική είτε σε ένα πέρασμα είτε σε περισσότερα.

Ορισμοί Οδηγούμενοι από το Συντακτικό

Ιδιότητες

Ένας ορισμός οδηγούμενος από το συντακτικό είναι μια γενικοποίηση των γραμματικών χωρίς συμφραζόμενα όπου κάθε σύμβολο της γραμματικής έχει ένα σύνολο από ιδιότητες (attributes).

(i) Φανταστείτε τον κόμβο ενός συμβόλου μιας γραμματικής σε ένα συντακτικό δέντρο ως μια εγγραφή με πεδία που περιέχουν πληροφορία. Οι ιδιότητες είναι τα ονόματα αυτών των πεδίων.

(ii) Μια ιδιότητα μπορεί να αναπαριστά ότι επιλέξουμε, μια συμβολοσειρά, έναν αριθμό, έναν τύπο, μια διεύθυνση μνήμης, κ.τ.λ.

Η τιμή μιας ιδιότητας σε έναν κόμβο ενός συντακτικού δέντρου ορίζεται από έναν σημασιολογικό κανόνα που σχετίζεται με τον κανόνα παραγωγής της γραμματικής που αφορά αυτό τον κόμβο.

Οι ιδιότητες των συμβόλων της γραμματικής χωρίζονται σε δύο κατηγορίες.

Τύποι Ιδιοτήτων

1. **συνθέσιμες ιδιότητες (synthesized attributes)**, των οποίων η τιμή τους σε ένα κόμβο του δέντρου μπορεί να υπολογισθεί με βάση τις τιμές των απογόνων του
2. **κληρονομήσιμες ιδιότητες (inherited attributes)**, που υπολογίζονται με βάση τις τιμές των προγόνων και των αδελφών κόμβων.

Συνθέσιμη ιδιότητα

Μια συνθέσιμη ιδιότητα για ένα μη-τερματικό σύμβολο A σε έναν κόμβο N ενός συντακτικού δέντρου ορίζεται από έναν σημασιολογικό κανόνα που σχετίζεται με την παραγωγή στον κόμβο N.

Η παραγωγή πρέπει να έχει το A στην αριστερή της πλευρά.

Μία συνθέσιμη ιδιότητα στον κόμβο N ορίζεται μόνο συναρτήσει ιδιοτήτων του ιδίου κόμβου N και ιδιοτήτων των παιδιών του N.

Κληρονομήσιμη ιδιότητα

Μια κληρονομήσιμη ιδιότητα για ένα μη-τερματικό σύμβολο B σε έναν κόμβο N ορίζεται από έναν σημασιολογικό κανόνα που σχετίζεται με μια παραγωγή στο πατέρα του N.

Η παραγωγή πρέπει να έχει το B στο δεξιό μέρος της.

Μια κληρονομήσιμη ιδιότητα σε έναν κόμβο N ορίζεται συναρτήσει μόνο τιμών ιδιοτήτων του πατέρα του N, του ίδιου του N και των αδελφών του N.

Ορισμοί Οδηγούμενοι από το Συντακτικό

Παράδειγμα: Απλή Αριθμομηχανή

(i) Συσχετίζουμε τα μη-τερματικά σύμβολα της γραμματικής E, T και F με μια συνθέσιμη ιδιότητα που πέρνει ακέραιες τιμές και έχει όνομα *val*.

(ii) Το τερματικό σύμβολο **digit** συσχετίζεται με μια συνθέσιμη ιδιότητα *yylval* που υποθέτουμε πως έρχεται από τον λεκτικό αναλυτή.

κανόνας	σημασιολογικός κανόνας
$L \to E$ **;**	$print(E.val)$
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to$ **(** E **)**	$F.val := E.val$
$F \to$ **digit**	$F.val :=$ **digit**$.yylval$

(iii) Θεωρούμε πως ο πρώτος σημασιολογικός κανόνας ορίζει την τιμή μιας ”ψεύτικης” ιδιότητας (dummy) η οποία δεν φαίνεται.

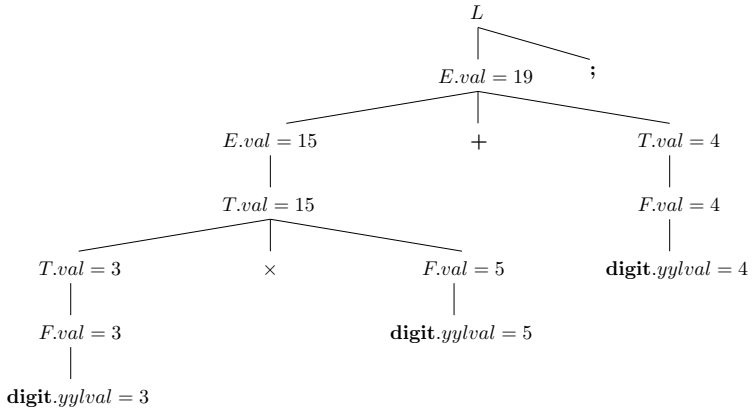
(i) Ο ορισμός αυτός περιέχει μόνο συνθέσιμες ιδιότητες. Ορισμοί που χρησιμοποιούν αποκλειστικά συνθέσιμες (synthesized) ιδιότητες, ονομάζονται **ορισμοί S-ιδιοτήτων** (S-attributed definition).

κανόνας	σημασιολογικός κανόνας
$L \to E$ **;**	$print(E.val)$
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to$ **(** E **)**	$F.val := E.val$
$F \to$ **digit**	$F.val :=$ **digit**$.yylval$

(iii) Σε έναν ορισμό που είναι S-ιδιοτήτων μπορούμε να υπολογίσουμε τις ιδιότητες με μια ανοδική διάσχιση του συντακτικού δέντρου. Για παράδειγμα με μια post-order διάσχιση.

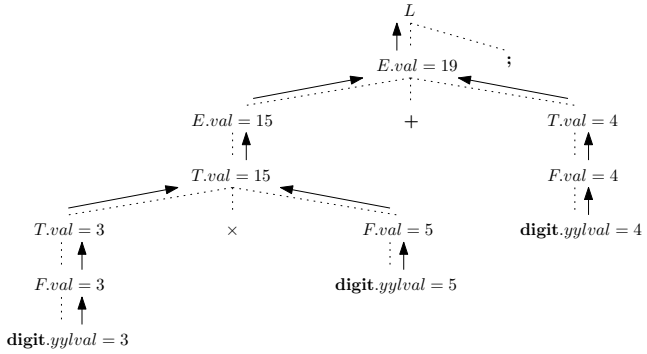
Ορισμοί Οδηγούμενοι από το Συντακτικό

Δέντρο Έκφρασης $3 \times 5 + 4;$



Ορισμοί Οδηγούμενοι από το Συντακτικό

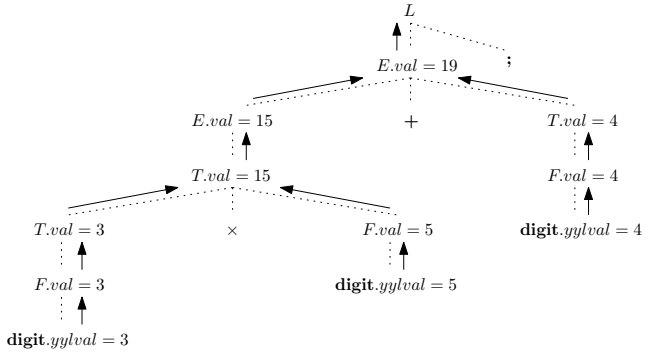
Δέντρο Έκφρασης $3 \times 5 + 4;$



(i) Από τους σημασιολογικούς κανόνες μπορούμε να αναπαραστήσουμε τις εξαρτήσεις των ιδιοτήτων ως ένα κατευθυνόμενο γράφημα.

Ορισμοί Οδηγούμενοι από το Συντακτικό

Δέντρο Έκφρασης $3 \times 5 + 4;$

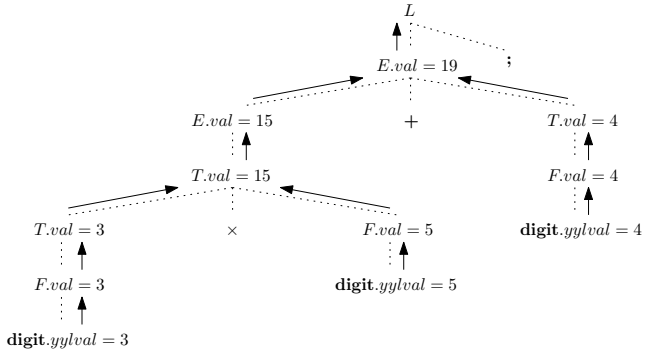


(i) Από τους σημασιολογικούς κανόνες μπορούμε να αναπαραστήσουμε τις εξαρτήσεις των ιδιοτήτων ως ένα κατευθυνόμενο γράφημα.

(ii) Κόμβοι είναι οι ιδιότητες και μια ακμή (a, b) σημαίνει πως η ιδιότητα a πρέπει να υπολογισθεί πριν από την ιδιότητα b

Ορισμοί Οδηγούμενοι από το Συντακτικό

Δέντρο Έκφρασης $3 \times 5 + 4;$



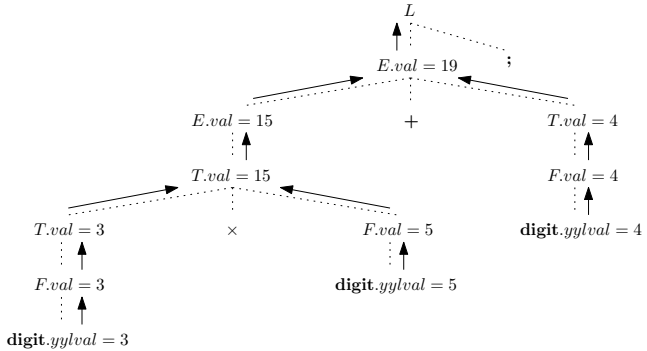
(i) Από τους σημασιολογικούς κανόνες μπορούμε να αναπαραστήσουμε τις εξαρτήσεις των ιδιοτήτων ως ένα κατευθυνόμενο γράφημα.

(ii) Κόμβοι είναι οι ιδιότητες και μια ακμή (a, b) σημαίνει πως η ιδιότητα a πρέπει να υπολογισθεί πριν από την ιδιότητα b

(iii) Το γράφημα ονομάζεται **γράφημα εξαρτήσεων** και πρέπει να είναι ακυκλικό (directed acyclic graph ή DAG).

Ορισμοί Οδηγούμενοι από το Συντακτικό

Δέντρο Έκφρασης $3 \times 5 + 4;$



(i) Από τους σημασιολογικούς κανόνες μπορούμε να αναπαραστήσουμε τις εξαρτήσεις των ιδιοτήτων ως ένα κατευθυνόμενο γράφημα.

(ii) Κόμβοι είναι οι ιδιότητες και μια ακμή (a, b) σημαίνει πως η ιδιότητα a πρέπει να υπολογισθεί πριν από την ιδιότητα b

(iii) Το γράφημα ονομάζεται **γράφημα εξαρτήσεων** και πρέπει να είναι ακυκλικό (directed acyclic graph ή DAG).

(iv) Σε αυτή την περίπτωση, μια τοπολογική διάταξη των κόμβων του DAG μας επιτρέπει τον σωστό υπολογισμό των ιδιοτήτων.

Ορισμοί Οδηγούμενοι από το Συντακτικό

Παράδειγμα: Δήλωση Μεταβλητών

(i) Στο παράδειγμα αυτό χρησιμοποιούμε κληρονομήσιμες ιδιότητες.

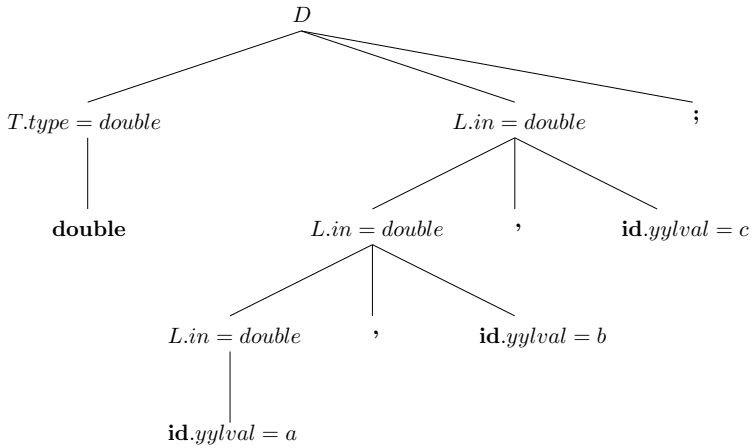
(ii) Το μη-τερματικό T έχει μια συνθέσιμη ιδιότητα *type*.

(iii) Το μη-τερματικό L έχει μια κληρονομήσιμη ιδιότητα *in* που χρησιμοποιείται για να στείλουμε τον τύπο της δήλωσης προς τα κάτω στο συντακτικό δέντρο.

κανόνας	σημασιολογικός κανόνας
$D \rightarrow T\ L$ **;**	$L.in := T.type$
$T \rightarrow$ **int**	$T.type := int$
$T \rightarrow$ **double**	$T.type := double$
$L \rightarrow L_1$ **, id**	$L_1.in := L.in$
	$addtype(\mathbf{id}.yylval, L.in)$
$L \rightarrow$ **id**	$addtype(\mathbf{id}.yylval, L.in)$

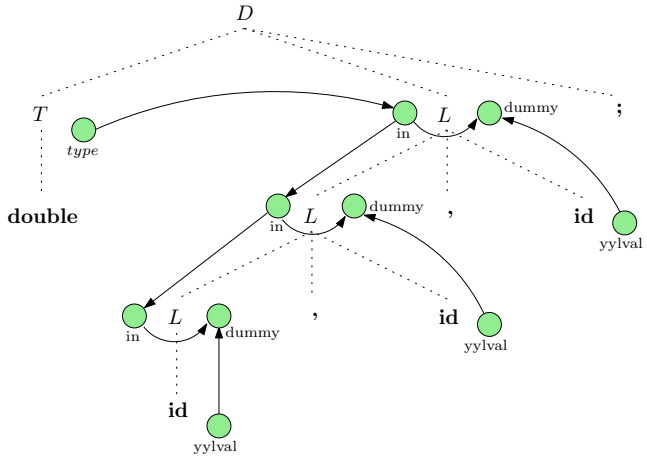
Ορισμοί Οδηγούμενοι από το Συντακτικό

Δέντρο Δήλωσης `double a, b, c;`



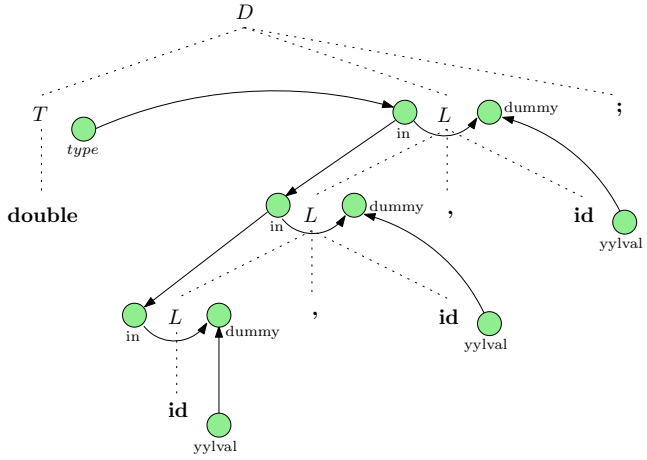
Ορισμοί Οδηγούμενοι από το Συντακτικό

Γράφημα Εξαρτήσεων Δήλωσης `double a, b, c;`



Ορισμοί Οδηγούμενοι από το Συντακτικό

Γράφημα Εξαρτήσεων Δήλωσης `double a, b, c;`



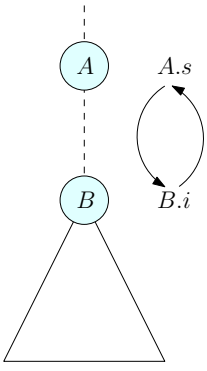
- Μια τοπολογική διάταξη του DAG μας επιτρέπει να εκτελέσουμε τους σημασιολογικούς κανόνες στην σωστή σειρά.

Ορισμοί Οδηγούμενοι από το Συντακτικό

Ύπαρξη Σωστής Σειράς Υπολογισμού;

Εαν ένας ορισμός έχει και συνθέσιμες και κληρονομήσιμες ιδιότητες, μπορεί να μην υπάρχει σωστή σειρά υπολογισμού των ιδιοτήτων σε ένα συντακτικό δέντρο.

π.χ εαν έχουμε δύο μη-τερματικά A και B με συνθέσιμη ιδιότητα $A.s$ και κληρονομήσιμη $B.i$



κανόνας	σημασιολογικός κανόνας
$A \to B$	$A.s = B.i$
	$B.i = A.s + 1$

το γράφημα εξαρτήσεων δεν είναι DAG αφού έχει κύκλο.

Ορισμοί Οδηγούμενοι από το Συντακτικό

Ύπαρξη Σωστής Σειράς Υπολογισμού;

Είναι ένα υπολογιστικά δύσκολο πρόβλημα δεδομένου ενός οδηγούμενου από το συντακτικό ορισμού, να βρούμε εαν υπάρχει μια σωστή σειρά υπολογισμού για κάθε δυνατό συντακτικό δέντρο.

Ορισμοί Οδηγούμενοι από το Συντακτικό

Ύπαρξη Σωστής Σειράς Υπολογισμού;

Είναι ένα υπολογιστικά δύσκολο πρόβλημα δεδομένου ενός οδηγούμενου από το συντακτικό ορισμού, να βρούμε εαν υπάρχει μια σωστή σειρά υπολογισμού για κάθε δυνατό συντακτικό δέντρο.

Αντιμετωπίζουμε το πρόβλημα αυτό χρησιμοποιώντας μόνο υποκλάσεις των ορισμών όπως

- ορισμούς S-ιδιοτήτων και
- ορισμούς L-ιδιοτήτων που θα δούμε στις επόμενες διαφάνειες.

Τα γραφήματα εξαρτήσεων που προκύπτουν από αυτούς τους ορισμούς είναι από κατασκευή DAGs και άρα έχουν σωστή σειρά υπολογισμού των ιδιοτήτων.

Ο ορισμός για την δήλωση μεταβλητών που είδαμε ανήκει σε μια κατηγορία ορισμών που ονομάζονται **ορισμοί *L*-ιδιοτήτων**.

Η ιδέα πίσω από τους ορισμούς αυτούς είναι πως μεταξύ των ιδιοτήτων μιας παραγωγής οι ακμές στο γράφημα εξαρτήσεων πάνε από αριστερά προς τα δεξιά, αλλά όχι από δεξιά προς τα αριστερά.

Ορισμοί L-ιδιοτήτων

Ένας ορισμός ονομάζεται L-ιδιοτήτων εαν κάθε ιδιότητα είναι

1. συνθέσιμη,
2. κληρονομήσιμη αλλά με τους επιπλέον κανόνες ως εξής.

   Έστω ο κανόνας παραγωγής $A \to X_1 X_2 \cdots X_n$ και μια κληρονομήσιμη ιδιότητα $X_i.a$ που υπολογίζεται από έναν σημασιολογικό κανόνα που σχετίζεται με αυτή την παραγωγή. Ο κανόνας αυτός μπορεί να χρησιμοποιήσει μόνο

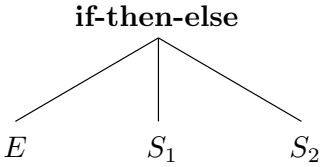
   - κληρονομήσιμες ιδιότητες που σχετίζονται με το A,
   - είτε κληρονομήσιμες είτε συνθέσιμες ιδιοτήτων που σχετίζονται με στιγμιότυπα των συμβόλων $X_1, X_2, \ldots, X_{i-1}$ που βρίσκονται αριστερά του X_i.
   - κληρονομήσιμες ή συνθέσιμες ιδιότητες που σχετίζονται με το συγκεκριμένο στιγμιότυπο του X_i, αλλά μόνο με τέτοιο τρόπο ώστε να μην υπάρχει κύκλος στον γράφημα εξαρτήσεων που δημιουργείται από τις ιδιότητες του X_i.

Θα δούμε ένα παράδειγμα όπου μέσα από ορισμούς οδηγούμενους από το συντακτικό μπορούμε να ορίσουμε την κατασκευή ”αφηρημένων συντακτικών δέντρων”.

Αφηρημένα Συντακτικά Δέντρα

Ένα αφηρημένο συντακτικό δέντρο (abstract syntax tree) είναι μια συνοπτική μορφή ενός συντακτικού δέντρου που προέκυψε από την συντακτική ανάλυση (parse tree).

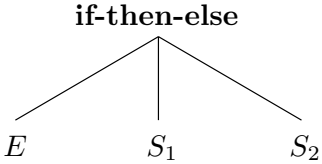
Κανόνες παραγωγής όπως $S \rightarrow \textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2$ μπορεί να εμφανιστούν ως:



Αφηρημένα Συντακτικά Δέντρα

Ένα αφηρημένο συντακτικό δέντρο (abstract syntax tree) είναι μια συνοπτική μορφή ενός συντακτικού δέντρου που προέκυψε από την συντακτική ανάλυση (parse tree).

Κανόνες παραγωγής όπως $S \to \textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2$ μπορεί να εμφανιστούν ως:



- Σε γενικές γραμμές αφαιρούμε ότι δεν δίνει επιπλέον πληροφορία, έχοντας υπόψη πως έχουμε ήδη κάνει συντακτική ανάλυση. Για παράδειγμα, ένα ερωτηματικό (semicolon) δεν έχει κάποια επιπλέον πληροφορία να μας δώσει.

Χρήση Αφηρημένων Συντακτικών Δέντρων

Θα δούμε στον κεφάλαιο της ενδιάμεσης αναπαράστασης πως με την χρήση αφηρημένων συντακτικών δέντρων μπορούμε να απεμπλέξουμε τις επόμενες φάσεις μεταγλώττισης από την συντακτική ανάλυση.

- Μετά την συντακτική ανάλυση φτιάχνουμε ένα αφηρημένο συντακτικό δέντρο στην μνήμη επάνω στο οποίο λειτουργούν οι υπόλοιπες φάσεις του μεταγλωττιστή.

Αυτή η τεχνική είναι πιο εύκολα εφαρμόσιμη σήμερα όπου οι υπολογιστές μας έχουν αρκετή μνήμη διαθέσιμη.

Αφηρημένα Συντακτικά Δέντρα

Τα αφηρημένα συντακτικά δέντρα είναι μια απλοποιημένη μορφή του συντακτικού δέντρου που δεν περιέχει χαρακτηριστικά της γλώσσας που δεν είναι χρήσιμα μετά την φάση της συντακτικής ανάλυσης.

(i) Επίσης η δομή ενός συντακτικού δέντρου εξαρτάται πολύ από την δομή της γραμματικής, και συνεπώς από τους μετασχηματισμούς που έγιναν όπως αφαίρεση αριστερής αναδρομής, αφαίρεση κοινού προθέματος, κ.τ.λ.

(ii) Στο αφηρημένο συντακτικό δέντρο θέλουμε να επιστρέψουμε στην πιο απλή γραμματική. Δεν μας πειράζει π.χ άμα είναι διφορούμενη αφού ξέρουμε ήδη το συντακτικό δέντρο.

Ο συντακτικός αναλυτής για μια γλώσσα προγραμματισμού μπορεί να κάνει την ανάλυση με βάση την γραμματική

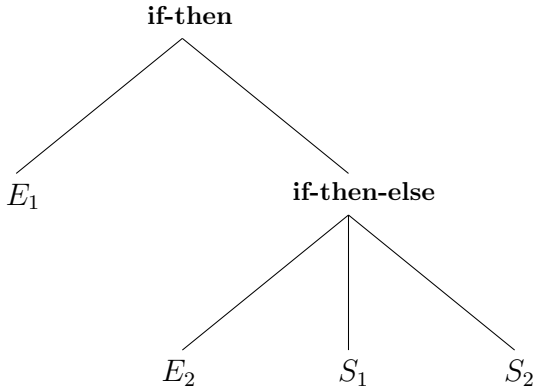
stmt	$\rightarrow$	*matched_stmt*
	|	*open_stmt*
matched_stmt	$\rightarrow$	**if** *expr* **then** *matched_stmt* **else** *matched_stmt*
	|	other
open_stmt	$\rightarrow$	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *matched_stmt* **else** *open_stmt*

διαβάζοντας όμως την συμβολοσειρά

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2$$

θέλουμε να δώσουμε στην επόμενη φάση ένα αφηρημένο συντακτικό δέντρο που να μην περιέχει τις λεπτομέρειες.

if E_1 **then if** E_2 **then** S_1 **else** S_2



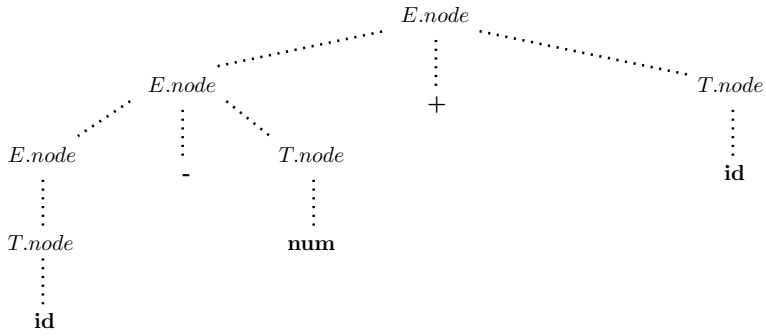
Ο παρακάτω οδηγούμενος από το συντακτικό ορισμός κατασκευάζει αφηρημένα συντακτικά δέντρα για εκφράσεις.

κανόνας	σημασιολογική ρουτίνα
$E \to E_1 + T$	$E.node = new\ Node('+', E_1.node, T.node)$
$E \to E_1 - T$	$E.node = new\ Node('-', E_1.node, T.node)$
$E \to T$	$E.node = T.node$
$T \to (E)$	$T.node = E.node$
$T \to id$	$T.node = new\ Leaf(id, id.yylval)$
$T \to num$	$T.node = new\ Leaf(num, num.yylval)$

Είναι ορισμός S-ιδιοτήτων αφού έχει μόνο συνθέσιμες ιδιότητες.

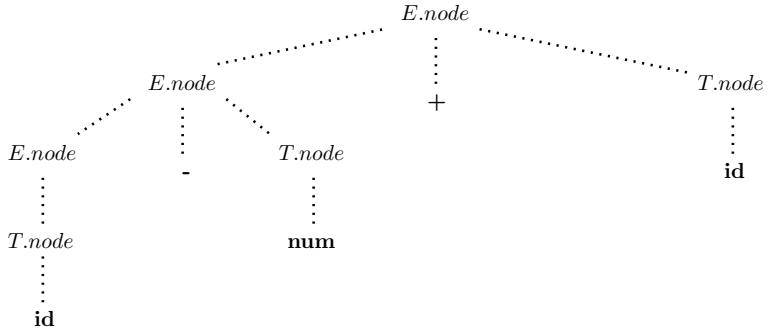
Αφηρημένα Συντακτικά Δέντρα

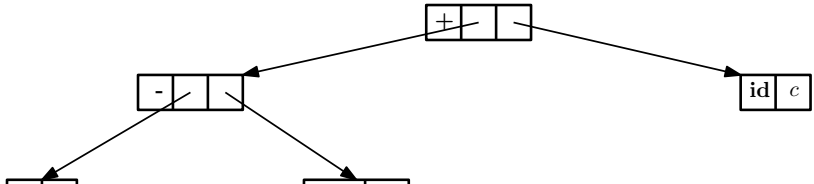
Η έκφραση $a - 4 + c$



Αφηρημένα Συντακτικά Δέντρα

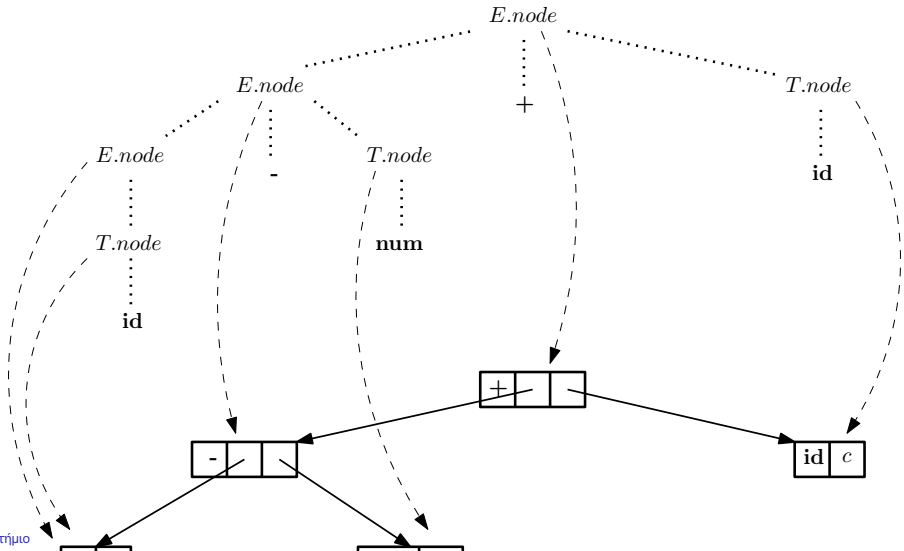
Η έκφραση $a - 4 + c$





Αφηρημένα Συντακτικά Δέντρα

Η έκφραση $a - 4 + c$



Μεταφραστικό Σχήμα

Ένα μεταφραστικό σχήμα οδηγούμενο από το συντακτικό είναι μια γραμματική χωρίς συμφραζόμενα με επιπλέον κομμάτια κώδικα μέσα στους κανόνες παραγωγής.

Τα κομμάτια κώδικα αυτά λέγονται σημασιολογικές πράξεις και μπορεί να εμφανιστούν σε οποιαδήποτε σημείο ενός κανόνα.

Από σύμβαση περικλείουμε τις σημασιολογικές πράξεις με { και }.

Μεταφραστικά Σχήματα

Παράδειγμα: Απλή Αριθμομηχανή

L	$\rightarrow$	E **;**	{	print(E.val);	}
E	$\rightarrow$	E_1 **+** T **;**	{	E.val = E_1.val + T.val;	}
E	$\rightarrow$	T	{	E.val = T.val;	}
T	$\rightarrow$	T_1 * F	{	T.val = T_1.val $\times$ F.val;	}
T	$\rightarrow$	F	{	T.val = F.val;	}
F	$\rightarrow$	**(** E **)**	{	F.val = E.val;	}
F	$\rightarrow$	**digit**	{	F.val = **digit**.yylval	}

Το παρακάτω μεταφραστικό σχήμα μετατρέπει εκφράσεις όπως

$$5 + 3 * 2 + 3 * (2 + 3)$$

σε

$$+ + 5 * 3\ 2 * 3 + 2\ 3$$

$L \rightarrow E$ **;**
$E \rightarrow$ { print('+'); } E_1 **+** T **;**
$E \rightarrow T$
$T \rightarrow$ { print('*'); } T_1 ***** F
$T \rightarrow F$
$F \rightarrow$ **(** E **)**
$F \rightarrow$ **digit** { print(**digit**.yylval); }

Υλοποίηση

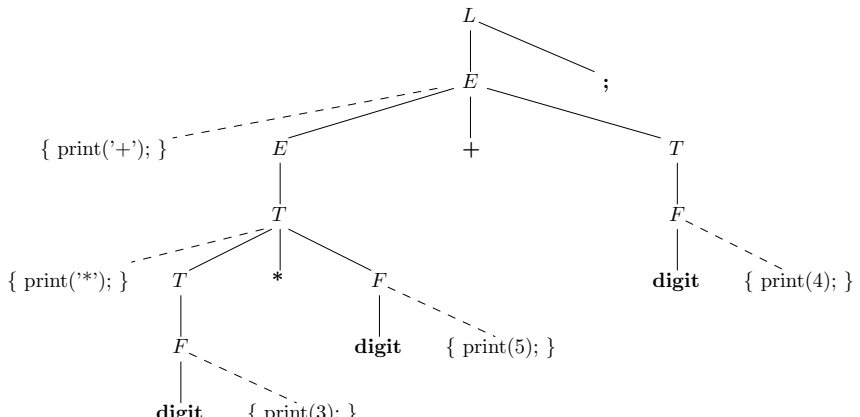
Οποιοδήποτε μεταφραστικό σχήμα μπορεί να υλοποιηθεί πρώτα κατασκευάζοντας το συντακτικό δέντρο και μετά εκτελώντας όλες τις σημασιολογικές πράξεις σε μια από αριστερά-προς-δεξιά πρώτα-σε-βάθος (DFS) διάσχιση.

Η έκφραση

$$3 * 5 + 4$$

έχει το εξής συντακτικό δέντρο. Προσθέτοντας τις σημασιολογικές πράξεις και εκτελώντας σε μια αριστερά-προς-δεξιά διάσχιση κατά βάθος, έχουμε ως αποτέλεσμα την έκφραση

$$+ * 3\ 5\ 4$$



Μεταφραστικά Σχήματα
Translation Schemes

Συνήθως τα μεταφραστικά σχήματα υλοποιούνται μέσω της συντακτικής ανάλυσης χωρίς να φτιάχνουμε το συντακτικό δέντρο.

Σημαντικές Περιπτώσεις

Δύο σημαντικές περιπτώσεις που μας ενδιαφέρουν είναι όταν

1. η γραμματική είναι *LR* και ο οδηγούμενος από το συντακτικό ορισμός είναι *S*-ιδιοτήτων
2. η γραμματική είναι *LL* και ο οδηγούμενος από το συντακτικό ορισμός είναι *L*-ιδιοτήτων.

Στις περιπτώσεις αυτές, οι σημασιολογικοί κανόνες του ορισμού μπορούν να μετατραπούν σε ένα μεταφραστικό σχήμα όπου τα κομμάτια κώδικα εκτελούνται την σωστή στιγμή.

Δεν θα μπούμε σε περισσότερες λεπτομέρειες.

Θα δούμε όμως

(i) το εργαλείο παραγωγής συντακτικών αναλυτών Yacc
(**Yet Another Compiler-Compiler**)

(ii) πως περιγράφουμε μεταφραστικά σχήματα με το Yacc

(iii) πως αποθηκεύει τις σημασιολογικές τιμές και πότε εκτελεί τις σημασιολογικές πράξεις.

Yacc/Bison και Εργαλεία Παραγωγής Συντακτικών Αναλυτών

Το εργαλείο Yacc είναι ένα πρόγραμμα που δέχεται ένα σύνολο κανόνων παραγωγής μιας γραμματικής και παράγει έναν $LALR(1)$ συντακτικό αναλυτή.

(i) Τα τερματικά και μη-τερματικά μπορούν να έχουν μια σημασιολογική τιμή.

(ii) Κάθε κανόνας της γραμματικής μπορεί να έχει επιπλέον κάποιες σημασιολογικές πράξεις που είναι κώδικας *C*.

Bison είναι το όνομα της υλοποίησης του Yacc από την GNU (“GNU’s Not Unix”).

Γεννήτριες Συντακτικών Αναλυτών

LALR

Yacc/Bison

- http://dinosaur.compilertools.net/yacc
- https://en.wikipedia.org/wiki/GNU_bison

CUP

- http://www2.cs.tum.edu/projects/cup/

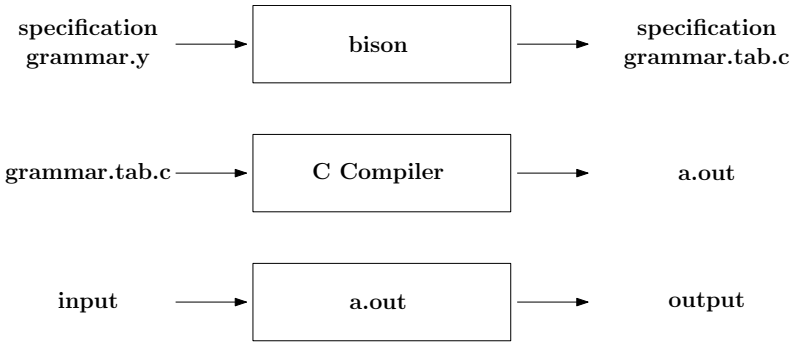
LL

ANTLR

- https://www.antlr.org/

Και πολλά περισσότερα εργαλεία που μπορείτε να τα βρείτε μέσω της ιστοσελίδας http://catalog.compilertools.net/ .

Bison


specification
grammar.y
bison
specification
grammar.tab.c
grammar.tab.c
C Compiler
a.out
input
a.out
output

Bison

Μορφή Προγραμμάτων

```
%{
δηλώσεις μεταβλητών C
%}
δηλώσεις bison
%%
μεταφραστικοί κανόνες
%%
βοηθητικές συναρτήσεις C
```

Ένα σύνολο κανόνων

$$head \rightarrow body_1 | body_2 | \cdots | body_n$$

έχει την εξής μορφή

```
head      :   body-1   { semantic-action-1 }
          |   body-2   { semantic-action-2 }
                ...
          |   body-n   { semantic-action-n }
          ;
```

Συμβολοσειρές που δεν είναι σε εισαγωγικά και δεν έχουν δηλωθεί ως λεκτικές μονάδες χρησιμοποιούνται ως μη-τερματικά.

Σημασιολογικές Τιμές

Το Bison υλοποιεί τις σημασιολογικές τιμές χρησιμοποιώντας την στοίβα.

- Όταν κάνει ολίσθηση (shift) ενός συμβόλου στην στοίβα, προσθέτει μαζί και την σημασιολογική τιμή αυτού του συμβόλου.

Σημασιολογικές Τιμές

Το Bison υλοποιεί τις σημασιολογικές τιμές χρησιμοποιώντας την στοίβα.

(i) Όταν κάνει ολίσθηση (shift) ενός συμβόλου στην στοίβα, προσθέτει μαζί και την σημασιολογική τιμή αυτού του συμβόλου.

(ii) Μπορείτε να φανταστείτε πως διατηρεί δύο στοίβες, ή πως κάθε στοιχείο της στοίβας είναι μια εγγραφή με σύμβολο και σημασιολογική τιμή.

Σημασιολογικές Τιμές

Το Bison υλοποιεί τις σημασιολογικές τιμές χρησιμοποιώντας την στοίβα.

(i) Όταν κάνει ολίσθηση (shift) ενός συμβόλου στην στοίβα, προσθέτει μαζί και την σημασιολογική τιμή αυτού του συμβόλου.

(ii) Μπορείτε να φανταστείτε πως διατηρεί δύο στοίβες, ή πως κάθε στοιχείο της στοίβας είναι μια εγγραφή με σύμβολο και σημασιολογική τιμή.

(iii) Τέτοιες σημασιολογικές τιμές έχουν τύπο *YYSTYPE*. Επαναορίζοντας την σταθερά αυτή μπορούμε να αλλάξουμε τον τύπο των σημασιολογικών τιμών. Μια κλασική τεχνική είναι να ορίσουμε το *YYSTYPE* ως `union`.

Σημασιολογικές Τιμές

Το Bison υλοποιεί τις σημασιολογικές τιμές χρησιμοποιώντας την στοίβα.

(i) Όταν κάνει ολίσθηση (shift) ενός συμβόλου στην στοίβα, προσθέτει μαζί και την σημασιολογική τιμή αυτού του συμβόλου.

(ii) Μπορείτε να φανταστείτε πως διατηρεί δύο στοίβες, ή πως κάθε στοιχείο της στοίβας είναι μια εγγραφή με σύμβολο και σημασιολογική τιμή.

(iii) Τέτοιες σημασιολογικές τιμές έχουν τύπο *YYSTYPE*. Επαναορίζοντας την σταθερά αυτή μπορούμε να αλλάξουμε τον τύπο των σημασιολογικών τιμών. Μια κλασική τεχνική είναι να ορίσουμε το *YYSTYPE* ως `union`.

(iv) Οι σημασιολογικές τιμές ενός τερματικού συμβόλου, διαβάζονται από την μεταβλητή *yylval*, η οποία έχει αποκτήσει τιμή από τον λεκτικό αναλυτή.

Οι σημασιολογικές πράξεις είναι ακολουθίες C εντολών.

(i) Σε μία σημασιολογική ρουτίνα έχουμε πρόσβαση στη σημασιολογική τιμή του μη-τερματικού στην αριστερή πλευρά του κανόνα παραγωγής με το σύμβολο $$.

(ii) Με $$i$ αναφερόμαστε στην τιμή που σχετίζεται με το i-οστό σύμβολο στο δεξιό μέρος του κανόνα παραγωγής.

Θα φτιάξουμε έναν συντακτικό αναλυτή για μία απλή αριθμομηχανή με την παρακάτω γραμματική.

$$
\begin{array}{lcl}
E & \Rightarrow & E + T \\
E & \Rightarrow & T \\
T & \Rightarrow & T * F \\
T & \Rightarrow & F \\
F & \Rightarrow & (\ E\ ) \\
F & \Rightarrow & \textbf{digit}
\end{array}
$$

Bison

Παράδειγμα Απλής Αριθμομηχανής

```
%{
#include <stdio.h>
#include <ctype.h>

void yyerror (char const *s)
{
    fprintf (stderr, "%s\n", s);
}
%}

%token DIGIT

%%
```

- Στο πρώτο μέρος κάνουμε απαραίτητες δηλώσεις για την γλώσσα C.
- Η συνάρτηση yyerror() καλείται από τον συντακτικό αναλυτή άμα δεν μπορέσει να διαβάσει την είσοδο
- Δηλώνουμε επίσης τα διάφορα τερματικά σύμβολα.

```
line:  expr  '\n'   { printf("%d\n", $1); }

expr:  expr  '+'  term  { $$ = $1 + $3; }
     | term
     ;

term:  term '*' factor { $$ = $1 * $3; }
     | factor
     ;

factor: '(' expr ')'    { $$ = $2; }
      | DIGIT
      ;
%%
```

Στο δεύτερο μέρος γράφουμε την γραμματική. Κάθε κανόνας μπορεί να έχει και σημασιολογικές πράξεις.
Η έκφραση $$ αφορά το σύμβολο που είναι στην αριστερή πλευρά ενός κανόνα. Οι εκφράσεις $1, $2, κ.τ.λ. αφορά τα σύμβολα που είναι στην δεξιά πλευρά του κανόνα.

```
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c - '0';
        return DIGIT;
    }
    return c;
}
```

Στο τελευταίο μέρος μπορούμε να δώσουμε επιπλέον βοηθητικές συναρτήσεις.

- Πρέπει να υπάρχει ένας λεκτικός αναλυτής με μια συνάρτηση yylex().
- Στο παρόν παράδειγμα υλοποιούμε εμείς ένα πολύ βασικό λεκτικό αναλυτή.

Για να χρησιμοποιήσουμε το προηγούμενο μεταπρόγραμμα καλούμε

```
> bison grammar.y
> gcc -c grammar.tab.c
```

Στην συνέχεια με ένα απλό πρόγραμμα όπως το παρακάτω

```
int yyparse (void);

int main() {
    return yyparse();
}
```

καλούμε τον συντακτικό αναλυτή.

Για να μεταγλωττίσετε το παραπάνω αρχείο με όνομα simple.c

```
> gcc simple.c grammar.tab.o -o simple-calc
```

Εκτέλεση Σημασιολογικών Κανόνων

Όταν ένας συντακτικός αναλυτής που έχει δημιουργηθεί από τον Bison κάνει ελάττωση (reduce) με τον κανόνα

$$A \to X_1 X_2 \cdots X_k \ \ \{ \text{ C κώδικας } \}$$

αφαιρεί από την στοίβα τα σύμβολα $X_1, X_2, \ldots, X_k$ και τις σημασιολογικές τιμές τους και προσθέτει το σύμβολο A με την σημασιολογική τιμή του (τιμή $\$\$$).

(i) Η εκτέλεση του κώδικα γίνεται πριν γίνει η ελάττωση.

(ii) Οποιαδήποτε αναφορά του κώδικα στα $\$1$, $\$2$, ..., $\$k$ χρησιμοποιεί τις σημασιολογικές τιμές των k στοιχείων που βρίσκονται υψηλότερα στην στοίβα.

Πράξεις στην Μέση των Κανόνων

Ο Bison εκτελεί τις σημασιολογικές πράξεις όταν κάνει ελάττωση με βάση έναν κανόνα παραγωγής.

Για να υποστηρίξει και πράξεις στο μέσο των κανόνων, όπως

```
A : X1 { action1 } X2 X3 { action2 }
```

προσθέτει ένα επιπλέον μη-τερματικό σύμβολο *M* και αντικαθιστά τον προηγούμενο κανόνα με

```
A : X1 M X2 X3 { action2 }
M : /* empty */ { action1 }
```

Πράξεις στην Μέση των Κανόνων

Ο Bison εκτελεί τις σημασιολογικές πράξεις όταν κάνει ελάττωση με βάση έναν κανόνα παραγωγής.

Για να υποστηρίξει και πράξεις στο μέσο των κανόνων, όπως

```
A : X1 { action1 } X2 X3 { action2 }
```

προσθέτει ένα επιπλέον μη-τερματικό σύμβολο *M* και αντικαθιστά τον προηγούμενο κανόνα με

```
A : X1 M X2 X3 { action2 }
M : /* empty */ { action1 }
```

Σε περίπτωση που θέλουμε να αποφύγουμε τον κανόνα με μόνο ϵ στην δεξιά πλευρά μπορούμε να μετασχηματίσουμε ισοδύναμα σε:

```
A : B X2 X3 {action2 }
B : X1 {action1 }
```

Σειρά Εκτέλεσης Σημασιολογικών Πράξεων

Ο τρόπος υπολογισμού των σημασιολογικών τιμών την ώρα της ελάττωσης, σημαίνει πως πρέπει να προσέχουμε ώστε να έχουν υπολογιστεί πριν τις χρησιμοποιήσουμε.

Πιο συγκεκριμένα πρέπει να μην αναφερόμαστε σε σημασιολογικές τιμές που αφορούν σύμβολα δεξιότερα σε έναν κανόνα παραγωγής.

```
A: X1 X2 { $4 = $2; } X3 X4 X5 X6
```

Η παραπάνω χρήση θα δώσει μήνυμα λάθους.

Το παράδειγμα δήλωσης μεταβλητών περιέχει μία κληρονομήσιμη ιδιότητα καθώς θέλουμε να στείλουμε τον τύπο προς τα κάτω στο συντακτικό δέντρο.

κανόνας	σημασιολογικός κανόνας
$D \rightarrow T\ L$ **;**	$L.in := T.type$
$T \rightarrow$ **int**	$T.type := int$
$T \rightarrow$ **double**	$T.type := double$
$L \rightarrow L_1$ **, id**	$L_1.in := L.in$
	$addtype(\mathbf{id}.yylval, L.in)$
$L \rightarrow$ **id**	$addtype(\mathbf{id}.yylval, L.in)$

Ένας τρόπος υλοποίησης αυτού του ορισμού σε Bison χρησιμοποιεί μια εξωτερική μεταβλητή της *C*.

Το παράδειγμα δήλωσης μεταβλητών περιέχει μία κληρονομήσιμη ιδιότητα καθώς θέλουμε να στείλουμε τον τύπο προς τα κάτω στο συντακτικό δέντρο.

```
D : T {curType=$1;}  L ;

T : int {$$ = INTEGER;}
  | double {$$ = DOUBLE;}

L : L, id { addToSymbolTable($3,curType); }
  | id { addToSymbolTable($1,curType); }
```

Μπορούμε να δηλώσουμε μια μεταβλητή *curType* στο πρώτο μέρος ενός προγράμματος Bison.

Ο Bison παρέχει και μια πιο προχωρημένη τεχνική που μας βοηθάει να υλοποιήσουμε κληρονομήσιμες ιδιότητες.

(i) Στις σημασιολογικές πράξεις μπορούμε να αναφερθούμε σε θέσεις της στοίβας που βρίσκονται χαμηλότερα από όλα τα σύμβολα της δεξιάς πλευράς ενός κανόνα.

(ii) Χρησιμοποιούμε $\$0, \$-1, \ldots$

```
D : T L ;

T : int {$$ = INTEGER;}
  | double {$$ = DOUBLE;}

L : L, id { addToSymbolTable($3,$0); }
  | id { addToSymbolTable($1,$0); }
```

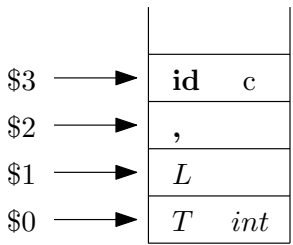
Δήλωση Μεταβλητών με Bison

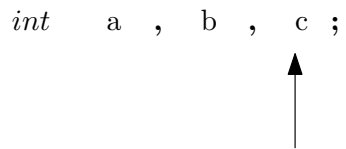
Κληρονομήσιμες Ιδιότητες

```
D : T L ;

T : int {$$ = INTEGER;}
  | double {$$ = DOUBLE;}

L : L, id { addToSymbolTable($3,$0); }
  | id { addToSymbolTable($1,$0); }
```





```
D : T L ;

T : int {$$ = INTEGER;}
  | double {$$ = DOUBLE;}

L : L, id { addToSymbolTable($3,$0); }
  | id { addToSymbolTable($1,$0); }
```

Η τεχνική αυτή θέλει ιδιαίτερη προσοχή, μιας και πρέπει να είμαστε σίγουροι πως πριν γίνει η ελάττωση υπάρχει όντως το αντίστοιχο σύμβολο πιο κάτω στην στοίβα.

Μπορούμε επίσης να υλοποιήσουμε την συγκεκριμένη περίπτωση κατασκευάζοντας μια λίστα από προσδιοριστικά και χρησιμοποιώντας μόνο συνθέσιμες ιδιότητες.

```
D : T L { forall( i, $2 )
              addToSymbolTable( i, $1 ); }

T : int {$$ = INTEGER;}
  | double {$$ = DOUBLE;}

L : L, id { list L = create_list($3);
            L->append($1);
            $$ = L; }
  | id { $$ = create_list($1); }
```

Θα κάνουμε τις εξής αλλαγές:

- πολλές γραμμές με εκφράσεις
- άδειες γραμμές μεταξύ των εκφράσεων

Θα φτιάξουμε έναν συντακτικό αναλυτή για μία απλή αριθμομηχανή με την παρακάτω γραμματική

$$E \quad \rightarrow \quad E + E \mid E - E \mid E * E \mid E/E \mid -E \mid (E) \mid \text{number}$$

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double  /* double type for Bison stack */

void yyerror (char const *s)
{
    fprintf (stderr, "%s\n", s);
}
%}
%token NUMBER

%left '+' '-'
%left '*' '/'
%right UMINUS
%%

lines: lines expr '\n'   { printf("%g\n", $2); }
     | lines '\n'
     | /* empty */
     ;
```

Bison

Προχωρημένη Αριθμομηχανή (Συνέχεια)

```
expr: expr '+' expr           { $$ = $1 + $3; }
    | expr '-' expr           { $$ = $1 - $3; }
    | expr '*' expr           { $$ = $1 * $3; }
    | expr '/' expr           { $$ = $1 / $3; }
    | '(' expr ')'            { $$ = $2; }
    | '-' expr %prec UMINUS   { $$ = -$2; }
    | NUMBER
    ;

%%

yylex() {
    int c;
    while ( (c=getchar()) == ' ' );
    if ( (c=='.') || (isdigit(c)) ) {
        ungetc(c,stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
```

Εαν δεν χρησιμοποιήσουμε προτεραιότητα τελεστών ο Bison παραπονιέται πως υπάρχουν conflicts.

Καλώντας τον bison με -v δημιουργείται ένα επιπλέον αρχείο grammar.output που περιέχει επιπλέον πληροφορίες για τις καταστάσεις και τα reduce/reduce και shift/reduce conflicts.

Ο Bison επιλύει τις συγκρούσεις με τους παρακάτω κανόνες:

1. Μια σύγκρουση ελάττωσης/ελάττωσης (reduce/reduce) επιλύεται επιλέγοντας τον κανόνα που εμφανίζεται πρώτος στο αρχείο μετα-προγράμματος.

Ο Bison επιλύει τις συγκρούσεις με τους παρακάτω κανόνες:

1. Μια σύγκρουση ελάττωσης/ελάττωσης (reduce/reduce) επιλύεται επιλέγοντας τον κανόνα που εμφανίζεται πρώτος στο αρχείο μετα-προγράμματος.
2. Μια σύγκρουση ολίσθησης/ελάττωσης επιλύεται πάντα κάνοντας ολίσθηση.
   Με αυτό τον τρόπο λύνεται σωστά το πρόβλημα του ξεκρέμαστου (dangling) else.

Ο τρόπος επίλυσης των συγκρούσεων δεν είναι πάντα ο επιθυμητός. Ο Bison μας επιτρέπει λοιπόν να καθορίσουμε την προσεταιριστικότητα και προτεραιότητα των τερματικών.

```
%left '+' '-'
%right '^'
%nonassoc '<'
%right UMINUS
```

Τερματικά στην ίδια δήλωση έχουν την ίδια προτεραιότητα. Οι δηλώσεις που βρίσκονται χαμηλότερα έχουν μεγαλύτερη προτεραιότητα.

Ο Bison επιλύει τις συγκρούσεις ολίσθησης/ελάττωσης συσχετίζοντας με μια προτεραιότητα και προσεταιριστικότητα κάθε κανόνα και κάθε τερματικό που σχετίζεται με την σύγκρουση.

Εάν πρέπει να διαλέξει μεταξύ

- ολίσθησης ενός συμβόλου a
- ελάττωσης με βάση ένα κανόνα $B \to \beta$

επιλέγει την ελάττωση αν η προτεραιότητα του κανόνα είναι μεγαλύτερη από του συμβόλου a, ή αν η προτεραιότητα είναι η ίδια αλλά η προσεταιριστικότητα του κανόνα είναι αριστερή. Σε αντίθετη περίπτωση επιλέγει ολίσθηση.

Κανονικά η προτεραιότητα ενός κανόνα είναι ίδια με την προτεραιότητα του τελευταίου τερματικού από δεξιά.

π.χ με τους κανόνες

$$\begin{array}{lcl} E & \rightarrow & E + E \\ E & \rightarrow & E * E \end{array}$$

αν βλέπουμε lookahead '+' γίνεται ελάττωση αφού το '+' στον κανόνα έχει την ίδια προτεραιότητα αλλά είναι αριστερά προσεταιριστικό.

αν βλέπουμε lookahead '*' γίνεται ολίσθηση αφού το '*' του lookahead έχει μεγαλύτερη προτεραιότητα από το '+' στον κανόνα.

Υπάρχουν περιπτώσεις που το τελευταίο τερματικό δεν παρέχει την απαραίτητη προτεραιότητα σε έναν κανόνα.

Μπορούμε να επιβάλλουμε συγκεκριμένη προτεραιότητα προσθέτοντας στο τέλος του κανόνα

```
%prec <terminal>
```

Όπου έχουμε δώσει προτεραιότητα στο συγκεκριμένο τερματικό. Το συγκεκριμένο τερματικό δεν είναι ανάγκη να επιστρέφεται από τον λεκτικό αναλυτή.

Σύνταξη ενός Αρχείου CUP

Java

Η είσοδος στο εργαλείο παραγωγής συντακτικών αναλυτών CUP έχει 4 μέρη:

1. δηλώσεις
   κώδικας που αντιγράφεται στον παραγώμενο συντακτικό αναλυτή
2. δήλωση τερματικών και μη-τερματικών και δήλωση του τύπου την σημασιολογικής τιμής τους
3. προτεραιότητα και προσεταιριστικότητα των τερματικών
   χρησιμοποιείται για να επιλυθούν shift/reduce conflicts
4. γραμματική
   σε μορφή BNF με επιπλέον σημασιολογικές πράξεις σε Java

Παράδειγμα Αρχείου CUP

```
import java_cup.runtime.*;

/* Terminals (tokens returned by the scanner). */
terminal            SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal            UMINUS, LPAREN, RPAREN;
terminal Integer    NUMBER;

/* Non terminals */
non terminal            expr_list, expr_part;
non terminal Integer    expr, term, factor;

/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;
```

Παράδειγμα Αρχείου CUP

```
/* The grammar */
expr_list ::= expr_list expr_part
            | expr_part
                ;

expr_part ::= expr SEMI;

expr      ::= expr PLUS expr
            | expr MINUS expr
            | expr TIMES expr
            | expr DIVIDE expr
            | expr MOD expr
            | MINUS expr %prec UMINUS
            | LPAREN expr RPAREN
            | NUMBER
              ;
```

- Η σύνταξη ενός αρχείου CUP μοιάζει πολύ με την σύνταξη ενός αρχείου Yacc/Bison.

Παράδειγμα Αρχείου CUP

Με Σημασιολογικές Ρουτίνες

```
// CUP specification for a simple expression evaluator (w/ actions)

import java_cup.runtime.*;

/* Terminals (tokens returned by the scanner). */
terminal           SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal           UMINUS, LPAREN, RPAREN;
terminal Integer   NUMBER;

/* Non-terminals */
non terminal            expr_list, expr_part;
non terminal Integer    expr;

/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

/* The grammar */
expr_list ::= expr_list expr_part
            | expr_part;
```

Παράδειγμα Αρχείου CUP

Με Σημασιολογικές Ρουτίνες

```
expr_part ::= expr:e
              {: System.out.println("= " + e); :}
              SEMI
               ;

expr      ::= expr:e1 PLUS expr:e2
              {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
            |
              expr:e1 MINUS expr:e2
              {: RESULT = new Integer(e1.intValue() – e2.intValue()); :}
            |
              expr:e1 TIMES expr:e2
              {: RESULT = new Integer(e1.intValue() * e2.intValue()); :}
            |
              expr:e1 DIVIDE expr:e2
              {: RESULT = new Integer(e1.intValue() / e2.intValue()); :}
            |
              expr:e1 MOD expr:e2
              {: RESULT = new Integer(e1.intValue() % e2.intValue()); :}
            |
              NUMBER:n  {: RESULT = n; :}
            |
              MINUS expr:e {: RESULT = new Integer(0 – e.intValue()); :}
              %prec UMINUS
            |
              LPAREN expr:e RPAREN {: RESULT = e; :}
                ;
```

Αφηρημένα Συντακτικά Δέντρα

Παράδειγμα Αριθμομηχανής

Έστω πως θέλουμε να υλοποιήσουμε μια μικρή αριθμομηχανή που να καταλαβαίνει την εξής γραμματική:

```
Program ::= Expr SEMICOLON
            ;

Expr    ::= Expr PLUS Expr
          |
            Expr MINUS Expr
          |
            Expr MULT Expr
          |
            MINUS Expr
          |
            LPAREN Expr RPAREN
          |
            INTEGER_LITERAL
            ;
```

Θα φτιάξουμε μια αντικειμενοστραφή έκδοση ενός αφηρημένου συντακτικού δέντρου.

Ο λεκτικός αναλυτής πρέπει να μας επιστρέφει τα εξής:

- `SEMICOLON PLUS MINUS MULT LPAREN RPAREN`
- `INTEGER_LITERAL` μαζί με την ακέραια τιμή

Μοντελοποιούμε την έκφραση με μια abstract κλάση σε Java (ή με ένα interface).

```
package org.hua.abstractsyntax;

public abstract class Expr
{
    public abstract int eval();
}
```

Η συνάρτηση `eval`() επιστρέφει την τιμή της έκφρασης.

Θα χρησιμοποιήσουμε την παρακάτω κλάση για να αναπαραστήσουμε μια σταθερά τύπου ακεραίου. Ο λεκτικός αναλυτής θα μας δώσει και την τιμή της σταθεράς.

```
package org.hua.abstractsyntax;

public class IntegerLiteral extends Expr {

    private int value;

    public IntegerLiteral(int value) {
        this.value = value;
    }

    @Override
    public int eval() {
        return value;
    }

}
```

Τα φύλλα του αφηρημένου συντακτικού δέντρου θα είναι αντικείμενα αυτής της κλάσης.

Αφηρημένα Συντακτικά Δέντρα

Παράδειγμα Αριθμομηχανής

Φτιάχνουμε μια κλάση για κάθε κόμβο που μπορεί να υπάρχει στο αφηρημένο συντακτικό δέντρο:

```
package org.hua.abstractsyntax;

public class PlusExpr extends Expr {

    private Expr e1, e2;

    public PlusExpr(Expr e1, Expr e2) {
        this.e1 = e1;
        this.e2 = e2;
    }

    @Override
    public int eval() {
        return e1.eval() + e2.eval();
    }

}
```

Αφηρημένα Συντακτικά Δέντρα

Παράδειγμα Αριθμομηχανής

Φτιάχνουμε μια κλάση για κάθε κόμβο που μπορεί να υπάρχει στο αφηρημένο συντακτικό δέντρο:

```
package org.hua.abstractsyntax;

public class MinusExpr extends Expr {

    private Expr e1, e2;

    public MinusExpr(Expr e1, Expr e2) {
        this.e1 = e1;
        this.e2 = e2;
    }

    @Override
    public int eval() {
        return e1.eval() − e2.eval();
    }

}
```

Αφηρημένα Συντακτικά Δέντρα

Παράδειγμα Αριθμομηχανής

Φτιάχνουμε μια κλάση για κάθε κόμβο που μπορεί να υπάρχει στο αφηρημένο συντακτικό δέντρο:

```
package org.hua.abstractsyntax;

public class TimesExpr extends Expr {

    private Expr e1, e2;

    public TimesExpr(Expr e1, Expr e2) {
        this.e1 = e1;
        this.e2 = e2;
    }

    @Override
    public int eval() {
        return e1.eval() * e2.eval();
    }

}
```

Φτιάχνουμε μια κλάση για κάθε κόμβο που μπορεί να υπάρχει στο αφηρημένο συντακτικό δέντρο:

```
package org.hua.abstractsyntax;

public class UnaryMinusExpr extends Expr {

    private Expr e;

    public UnaryMinusExpr(Expr e) {
        this.e = e;
    }

    @Override
    public int eval() {
        return - e.eval();
    }

}
```

Αφηρημένα Συντακτικά Δέντρα

Παράδειγμα Αριθμομηχανής

Φτιάχνουμε μια κλάση για κάθε κόμβο που μπορεί να υπάρχει στο αφηρημένο συντακτικό δέντρο:

```
package org.hua.abstractsyntax;

public class Program {

    private Expr e;

    public Program(Expr e) {
        this.e = e;
    }

    @Override
    public int eval() {
        return e.eval();
    }

}
```

Αφηρημένα Συντακτικά Δέντρα

Παράδειγμα Αριθμομηχανής

Δεδομένου των κλάσεων αυτών μπορούμε να μεταφράσουμε εκφράσεις σε αφηρημένα συντακτικά δέντρα μέσω της γραμματικής:

```
package org.hua.parser;

import org.hua.abstractsyntax.*;

terminal LPAREN, RPAREN, SEMICOLON, EQ;
terminal TIMES, PLUS, MINUS, UMINUS;
terminal Integer INTEGER_LITERAL;

non terminal org.hua.abstractsyntax.Program Program;
non terminal org.hua.abstractsyntax.Expr Expr;

precedence left PLUS, MINUS;
precedence left TIMES;
precedence left UMINUS;

start with Program;
```

Δεδομένου των κλάσεων αυτών μπορούμε να μεταφράσουμε εκφράσεις σε αφηρημένα συντακτικά δέντρα μέσω της γραμματικής:

```
Program ::= Expr:e
            {:
                Program program = new Program( e );
                RESULT = program;
                System.out.println("Value = " + program.eval());
            :}
            SEMICOLON
            ;
```

Αφηρημένα Συντακτικά Δέντρα

Παράδειγμα Αριθμομηχανής

Δεδομένου των κλάσεων αυτών μπορούμε να μεταφράσουμε εκφράσεις σε αφηρημένα συντακτικά δέντρα μέσω της γραμματικής:

```
Expr    ::= Expr:e1 PLUS Expr:e2
            {: RESULT = new PlusExpr(e1, e2); :}
          |
            Expr:e1 MINUS Expr:e2
            {: RESULT = new MinusExpr(e1, e2); :}
          |
            Expr:e1 MULT Expr:e2
            {: RESULT = new MultExpr(e1, e2); :}
          |
            MINUS Expr:e
            {: RESULT = new UnaryMinusExpr(e); :}
            %prec UMINUS
          |
            LPAREN Expr:e RPAREN
            {: RESULT = e; :}
          |
            INTEGER_LITERAL:n
            {:
               RESULT = new IntegerLiteral( n.intValue() );
            :}
        ;
```

cscope

- βάση δεδομένων που αποθηκεύει σχέσεις ως στήλες (column-store)
- ερευνητικό πρωτότυπο `http://db.csail.mit.edu/projects/cstore/`
- commencial version `http://www.vertica.com/`

Ο SQL parser είναι γραμμένος με YACC σε γλώσσα C++ και χρησιμοποιεί αφηρημένα συντακτικά δέντρα.

Στις επόμενες διαφάνειες μπορούμε να δούμε ένα μικρό μέρος της περιγραφής του συντακτικού αναλυτή.

Αφηρημένα Συντακτικά Δέντρα

Παράδειγμα SQL Συντακτικού Αναλυτή

```
1   %{
2
3   #include <iostream>
4   #include "PObject.h"
5
6   #define YYSTYPE PObject*
7
8   #include "ListProjections.h"
9   #include "ListSelections.h"
10  #include "ListColumnValues.h"
11  #include "TVariable.h"
12  #include "Expression.h"
13  #include "Query.h"
14  #include "PIdent.h"
15
16  #include "Expressions/EColumn.h"
17  #include "Expressions/EString.h"
18  #include "Expressions/ENumber.h"
19  #include "Expressions/ENumbers/EInteger.h"
20  #include "Expressions/ENumbers/EFloat.h"
21  #include "Expressions/EArithmetic.h"
22  #include "Expressions/EArithmetics/EPlus.h"
23  #include "Expressions/EAgg.h"
24  #include "Expressions/EAgg/EAggCount.h"
25  #include "BExpression.h"
26  #include "BExpressions/BBool.h"
27  #include "BExpressions/BIsNull.h"
28  #include "Queries/QCommit.h"
29  #include "Queries/QDelete.h"
30  #include "Queries/QInsert.h"
31  #include "Queries/QSelect.h"
32  #include "Queries/QMerge.h"
```

Αφηρημένα Συντακτικά Δέντρα

Παράδειγμα SQL Συντακτικού Αναλυτή

```
33  using namespace std;
34
35  int yylex( void );
36
37  extern "C" void yyerror( char *str ) {
38      parser_error( "Syntax error" );
39  }
40
41  extern "C" int yywrap( void ) {
42      return 1;
43  }
44
45  void parser_error( string message ) {
46      cerr << "error: " << message << endl;
47      exit( 1 );
48  }
49
50  extern Parser* thisptr;
51
52  %}
53
54  %left OR
55  %left AND
56  %left NOT
57  %left OP_EQ OP_NE OP_LT OP_GT OP_LE OP_GE
58  %left '+' '-'
59  %left '*' '/'
60  %nonassoc UMINUS
```

Αφηρημένα Συντακτικά Δέντρα

Παράδειγμα SQL Συντακτικού Αναλυτή

```
61  %token AND AVG MIN MAX SUM COUNT AS ASC BETWEEN BY DESC DISTINCT FROM GROUP
62  %token HAVING IN IS NOT NULLX OR ORDER SELECT WHERE BOOL_TRUE BOOL_FALSE
63  %token AGG_COUNT AGG_MIN AGG_MAX AGG_SUM AGG_AVG
64  %token STRING FLOATNUM INTNUM NAME
65  %token INSERT INTO VALUES DELETE COMMIT LIMIT OFFSET MERGE
66
67  %%
68
69  sql:
70      statement ';'^^I{
71          thisptr->setQuery( dynamic_cast<Query*>( $1 ) );
72          YYACCEPT;
73      }
74
75  statement:
76      merge_statement { $$ = $1; }
77      |
78      commit_statement { $$ = $1; }
79      |
80      delete_statement_searched { $$ = $1; }
81      |
82      insert_statement { $$ = $1; }
83      |
84      query_spec { $$ = $1; }
85  ^^I;
```

Αφηρημένα Συντακτικά Δέντρα

Παράδειγμα SQL Συντακτικού Αναλυτή

```
86   commit_statement:
87       COMMIT {
88           Query* q = QCommit::create();
89           $$ = q;
90       };
91
92   delete_statement_searched:
93       DELETE FROM projection_list opt_where_clause {
94           ListProjections* cl_from = dynamic_cast<ListProjections*>( $3 );
95           BExpression*  cl_where = dynamic_cast<BExpression*>( $4 );
96
97           Query* q;
98           cl_where->typeCheck(cl_from);
99           q = QDelete::create();
100
101          list<Projection*> projections = cl_from->getProjections();
102
103          if (projections.size() > 0)
104          {
105               ( (QDelete*) q)->setProjection(projections.front());
106          }
107          ( (QDelete*) q)->setWherePredicate(cl_where);
108
109          $$ = q;
110      };
```

Αφηρημένα Συντακτικά Δέντρα

Παράδειγμα SQL Συντακτικού Αναλυτή

```
111  /* ... */
112
113  query_spec:
114      SELECT selection FROM projection_list opt_where_clause opt_ordering {
115          /* ... */
116      }
117      |
118      SELECT selection FROM projection_list opt_where_clause
119      GROUP BY group_by_clause opt_having_clause opt_ordering {
120          /* ... */
121      };
122
123  selection:
124      scalar_exp_commalist  { $$ = $1; }
125  ^^I|
126      '*'^^I{ /* ... */ }
127  ^^I;
128
129  scalar_exp_commalist:
130      expression^^I{ /* ... */ }
131      |
132      expression AS NAME^^I{ /* ... */ }
133      |
134      scalar_exp_commalist ',' expression^^I{ /* ... */ }
135      |
136      scalar_exp_commalist ',' expression AS NAME^^I{ /* ... */ }
137      ;
138
139  /* ... */
```

Αφηρημένα Συντακτικά Δέντρα

Παράδειγμα SQL Συντακτικού Αναλυτή

```
140   opt_where_clause:
141       /* empty */ {
142           BBool* lp = BBool::create( 1 );
143           $$ = lp;
144       }
145       |^^Iwhere_clause { $$ = $1; }
146       ;
147
148   where_clause: WHERE search_condition { $$ = $2; } ;
149
150   search_condition:
151       |
152       search_condition OR search_condition^^I{
153           BExpression* l = dynamic_cast<BExpression*>($1);
154           BExpression* r = dynamic_cast<BExpression*>($3);
155           BDisjunction* lp = BDisjunction::create( l, r );
156           $$ = lp;
157       }
158       |
159       search_condition AND search_condition^^I{
160           BExpression* l = dynamic_cast<BExpression*>($1);
161           BExpression* r = dynamic_cast<BExpression*>($3);
162           BConjunction* lp = BConjunction::create( l, r );
163           $$ = lp;
164       }
165       |^^INOT search_condition { /* ... */ }
166       |^^Iboolean_literal { $$ = $1; }
167       |^^I'(' search_condition ')' { $$ = $2; }
168       |^^Ipredicate { $$ = $1; }
169   ^^I;
170   /* ... */
171   %%
```