ΗΥ340 : ΓΛΩΣΣΕΣ ΚΑΙ ΜΕΤΑΦΡΑΣΤΕΣ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

```
VAR i:Integer;

    FUNCTION(Symbol) replicate

            x = (function(x,y){return x+y;});

              class DelFunctor: public std::unary_function<
```

ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης

Διάλεξη 8η
ΣΥΝΤΑΚΤΙΚΑ ΟΔΗΓΟΥΜΕΝΗ ΜΕΤΑΦΡΑΣΗ

Περιεχόμενα

- *Η αναγκαιότητα*
- Γραμματικές γνωρισμάτων
- Υλοποίηση σε LR parsers
- Υλοποίηση σε LL parsers
- Προγραμματιστική συμβουλή

Η αναγκαιότητα (1/5)

- Η συντακτική ανάλυση στοχεύει στον εντοπισμό του δέντρου συντακτικής ανάλυσης με διαδοχικά βήματα που εξομοιώνουν την κατασκευή του
  - *καθοδικά (top-down)* - LL(k) parsers
    - για predictive non-backtracking descent parser με πίνακα ανάλυσης η κατασκευή του δέντρου γίνεται top-down
    - για predictive / backtracking RDP η κατασκευή μπορεί να γίνει hacked είτε top-down ή bottom-up (σε περίπτωση back tracking πρέπει να «ακυρώσουμε» το λάθος τμήμα του δέντρου)
  - *ανοδικά (bottom-up)* - shift-reduce LR(k) parsers
    - η κατασκευή του δέντρου γίνεται πάντα bottom-up, χωρίς να απαιτείται ποτέ back-tracking

Η αναγκαιότητα (2/5)

- Για να εφαρμόσουμε μετάφραση, απλά και μόνο η συντακτική αναγνώριση της λέξης εισόδου δεν είναι επαρκής
  - Δεν έχει χρηστικότητα ένας C compiler που απλώς απαντά «*το πρόγραμμά σας είναι σωστό / λανθασμένο συντακτικά*»
  - Είναι προφανές ότι θα πρέπει να δρομολογούνται και και οι κατάλληλες ενέργειες μετάφρασης, δηλαδή να εκτελείται «κώδικας» σε κάποια σημεία της ανάλυσης

Η αναγκαιότητα (3/5)

- Πότε όμως είναι κατάλληλο να καλείται ο κώδικας που αναλαμβάνει να κάνει ότι χρειάζεται για μετάφραση;
  - Κάθε γραμματική παραγωγή (κανόνας) αντιπροσωπεύει και μία αντίστοιχη συντακτική δομή
  - Εάν κατά την αντιστοίχιση ενός τμήματος της λέξης εισόδου σε κάποια συντακτική δομή απαιτείται μετάφραση σε «κάτι» είναι απαραίτητο να μπορούμε να ορίσουμε κώδικα που εκτελείται ***ακριβώς με την αναγνώριση αυτής της συντακτικής δομής***
    - αυτό συμβαίνει σε περίπτωση επιτυχούς δοκιμής παραγωγής στους καθοδικούς αναλυτές
    - και στην περίπτωσης αναγωγής με κάποια παραγωγή στους ανοδικούς αναλυτές
- Θα δούμε ότι αυτή η πρακτική ανάγκη συσχέτισης κώδικα με γραμματικούς κανόνες (σε αναγνώριση ή αναγωγή) τυποποιείται θεωρητικά

Η αναγκαιότητα (4/5)

Η δυνατότητα μετάφρασης οδηγούμενης από την συντακτική ανάλυσης είναι θεμελιώδες στοιχείο όλων των μεθόδων κατασκευής συντακτικών αναλυτών

Από τον αλγόριθμο καθοδικής ανάλυσης με πρόβλεψη χωρίς αναδρομή

- Αλλιώς περιέχει μία γραμματική παραγωγή του X, έστω $X \rightarrow UVW$, δηλ. $M[X,a] = \{X \rightarrow UVW\}$. Αυτό ο αναλυτής το αντιμετωπίζει κάνοντας pop το X και push με τη σειρά W, V και U (δηλ. από δεξιά προς τα αριστερά). Σαν έξοδο μπορεί να βγάλει μήνυμα της παραγωγής που επεξεργάστηκε, αλλά και να εκτελέσει κάποια συνάρτηση (κώδικα) που μπορεί και αυτή να αποθηκεύεται στα στοιχεία πίνακα.

Ενσωματωμένος κώδικας μετάφρασης σε RDP για το *stmt→if (expr) stmt else stmt*

```
node* ifelse_f (void) {
    if (lookAhead!= IF_TOKEN)
        return (node*) 0;
    if (!match(OPENPAR_TOKEN)) {
        error(OPENPAR_TOKEN);
        return (node*) 0;
    }
    expr_node* cond = expr_f();
    if (!cond)
        return (node*) 0;
    if (!match(CLOSEPAR_TOKEN)) {
        error(CLOSEPAR_TOKEN);
        free(cond);
        return (node*) 0;
    }
    node* ifStmt = stmt_f();
    if (!ifStmt) {
        free(cond);
        return (node*) 0;
    }

    node* elseStmt = (node*) 0;
    if (lookAhead==ELSE_TOKEN && match(ELSE_TOKEN))
        elseStmt = stmt_f();
    return make_ifelse(cond, ifStmt, elseStmt);
};
```

Από τον αλγόριθμο ανοδικής συντακτικής ανάλυσης ενός LR parser

- Αν $action[s_m, a_i] = reduce\ A \rightarrow \beta$, τότε γίνεται κίνηση αναγωγής με νέο configuration $(s_0 X_1 s_1 X_2 s_2 \ldots X_{m-r} s_{m-r} A s, a_i a_{i+1} \ldots a_n\$)$ όπου r είναι το μήκος σε γραμματικά σύμβολα του β και $goto[s_{m-r}, A]=s$. Ο parser κάνει συνολικά pop $2r$ σύμβολα, αφήνοντας στην κορυφή της στοίβας την κατάσταση s_{m-r} και κάνοντας push το A και το $goto[s_{m-r}, A]$. Ταυτόχρονα, ο parser μπορεί να εκτελέσει και κώδικα που έχει συσχετιστεί με την παραγωγή $A \rightarrow \beta$, καθώς και να εκτυπώσει στην έξοδο την αναγωγή αυτή.

Η αναγκαιότητα (5/5)

IfElse → **if** (*expr*) *stmt*
IfElse →**if** (*expr*) *stmt* **else** *stmt*

```
node* ifelse_f (void) {
    if (lookAhead!= IF_TOKEN)
        return (node*) 0;
    if (!match(OPENPAR_TOKEN)) {
        error(OPENPAR_TOKEN);
        return (node*) 0;
    }
    expr_node* cond = expr_f();
    if (!cond)
        return (node*) 0;
    if (!match(CLOSEPAR_TOKEN)) {
        error(CLOSEPAR_TOKEN);
        free(cond);
        return (node*) 0;
    }
    node* ifStmt = stmt_f();
    if (!ifStmt) {
        free(cond);
        return (node*) 0;
    }

    node* elseStmt = (node*) 0;
    if (lookAhead==ELSE_TOKEN && match(ELSE_TOKEN))
        elseStmt = stmt_f();
    return make_ifelse(cond, ifStmt, elseStmt);
};
```

syntax analysis code

get expr value

get if stmt value

get else stmt value

set if_else stmt value

IfElse → **if** (*expr*) *stmt*
`{ return make_ifelse(expr, stmt, NULL); }`
IfElse →**if** (*expr*) *stmt* **else** *stmt*
`{ return make_ifelse(expr, stmt<1>, stmt<2>); }`

- Παρατηρούμε ότι σε κατασκευή RDP κάθε μη τερματικό σύμβολο αντιστοιχεί σε μία συνάρτηση, η κλήση της οποίας επιστρέφει και κάποια τιμή συγκεκριμένου τύπου δεδομένων
- Ουσιαστικά είναι σαν να έχουμε αντιστοιχήσει κάθε γραμματικό σύμβολο με κάποιο κατάλληλο τύπο δεδομένων
- Εάν δεν έχουμε RDP, αλλά έχουμε αναλυτή που παράγεται από κάποια γεννήτρια, τότε θα θέλαμε να χρησιμοποιούμε τα γραμματικά σύμβολα ως μεταβλητές
- Μπορούνε να κατανοήσουμε τι ρόλο παίζει ο διπλανός κώδικας καθώς και πότε περιμένουμε να καλείται

Περιεχόμενα

- Η αναγκαιότητα
- *Γραμματικές γνωρισμάτων*
- Υλοποίηση σε LR parsers
- Υλοποίηση σε LL parsers
- Προγραμματιστική συμβουλή

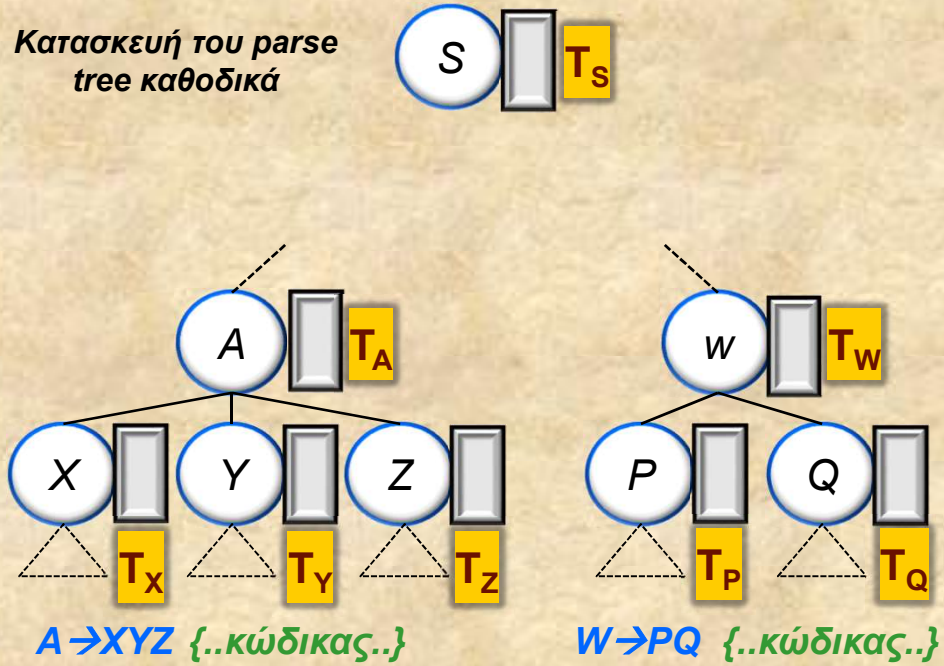
Γραμματικές γνωρισμάτων – γένεση

Παρατήρηση: Για την κατασκευή του δέντρου προφανώς απαιτείται μνήμη (κόμβοι και συνδέσεις)

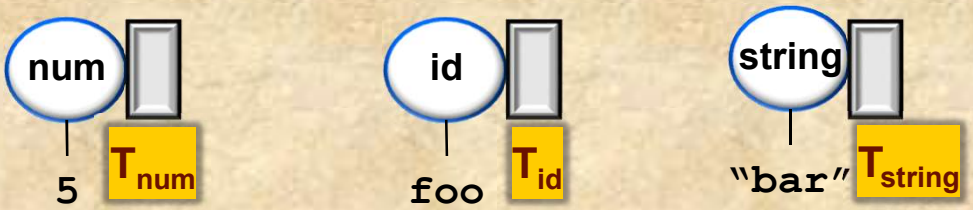
Ιδέα: Μήπως να δώσουμε επιπλέον μνήμη σε κάθε κόμβο;

Γιατί: Ώστε όταν ένας κόμβος κατασκευάζεται λόγω reduction (ή production) ο κώδικας που καλείται να μπορεί να χρησιμοποιεί τον κόμβο ως αποθηκευτικό χώρο!

Κατασκευή του parse tree καθοδικά



Κατασκευή του parse tree ανοδικά



Μέγεθος: Το μέγεθος της μνήμης του κάθε κόμβου πρέπει να είναι τόσο όσο χρειάζεται ο τύπος της τιμής θα αποθηκεύεται

Τύπος: Αυτό εξαρτάται από τον κώδικα που υπολογίζει την τιμή και ποικίλει για κάθε γραμματικό σύμβολο

Ορισμός: Ας ζητήσουμε από τον προγραμματιστή να ορίζει τον τύπο δεδομένων του κάθε γραμματικού συμβόλου

Υλοποίηση: Τότε αρκεί να κάνουμε *malloc* με το *sizeof* αυτού του τύπου κατά τη δημιουργία ενός κόμβου

$type\langle A\rangle : T_A$
$type\langle X\rangle : T_X$
$type\langle Y\rangle : T_Y$
$type\langle Z\rangle : T_Z$
$\kappa\lambda\pi\ldots$

Γραμματικές γνωρισμάτων (1/11)

Συντακτικά οδηγούμενοι ορισμοί. Πρόκειται για γενίκευση στις γραμματικές ανεξάρτητες συμφραζομένων όπου κάθε γραμματικό σύμβολο μπορεί να έχει ένα σύνολο από χαρακτηριστικά γνωρίσματα, χωρισμένα σε δύο υποσύνολα, τα συντιθέμενα γνωρίσματα (synthesized attributes) και τα κληρονομούμενα γνωρίσματα (inherited attributes).
Εάν φανταστούμε έναν κόμβο για κάποιο γραμματικό σύμβολο δέντρου συντακτικής ανάλυσης σαν μία εγγραφή, τότε τα πεδία αυτής της εγγραφής θα είναι τα γνωρίσματα του γραμματικού συμβόλου.

$$S \rightarrow \alpha A \beta \qquad A \rightarrow \gamma B \delta$$

Έστω A με γνωρίσματα $\alpha_1,\ldots,\alpha_n$ τύπων δεδομένων $t_1,\ldots,t_n$ αντίστοιχα. Τότε σε οποιοδήποτε δέντρο συντακτικής ανάλυσης της παραπάνω γραμματικής, οποιαδήποτε εμφάνιση του A (ως κόμβος) θα αντιστοιχεί πάντα σε μία συγκεκριμένη τιμή του *record* $A \{ t_1\ \alpha_1,\ t_2\ \alpha_2,..,\ t_n\ \alpha_n\}$

Γραμματικές γνωρισμάτων (2/11)

- Η τιμή ενός attribute για έναν κόμβο του συντακτικού δέντρου αποδίδεται από τον κώδικα της γραμματικής παραγωγής που χρησιμοποιήθηκε για τον κόμβο αυτό
  - Αυτόν τον κώδικα για μία παραγωγή τον λέμε *σημασιολογικό κανόνα*, σε αντιστοιχία με τον συντακτικό κανόνα που συνιστά η ίδια η παραγωγή
- Η τιμή ενός *συντιθέμενου γνωρίσματος* σε ένα κόμβο υπολογίζεται πάντα βάσει των τιμών των γνωρισμάτων των παιδιών του κόμβου στο δέντρο
- Η τιμή ενός *κληρονομημένου γνωρίσματος* σε ένα κόμβο υπολογίζεται πάντα βάσει των τιμών των γνωρισμάτων του γονικού κόμβου και των ιεραρχικών εταίρων του κόμβου αυτού
  - ***Δεν θα μας απασχολήσουν*** καθώς κάθε συντακτικά οδηγούμενος ορισμός με κληρονομημένα γνωρίσματα μπορεί να διατυπωθεί βάσει συντιθέμενων γνωρισμάτων

Γραμματικές γνωρισμάτων (3/11)

- Οι σημασιολογικοί κανόνες ορίζουν εξαρτήσεις μεταξύ των γνωρισμάτων γραμματικών συμβόλων – αυτές μπορούν να αναπαρίστανται μέσω ενός γράφου
  - Από αυτό το γράφο μπορούμε να εξάγουμε μια διάταξη αποτίμησης για τους σημασιολογικούς κανόνες
  - Η αποτίμηση των σημασιολογικών κανόνων δίνει τιμές στα γνωρίσματα των κόμβων του δέντρου συντακτικής ανάλυσης για τη λέξη εισόδου
- Οι σημασιολογικοί κανόνες μπορούν να έχουν και πλάγια αποτελέσματα - side effects
  - π.χ. εκτύπωση τιμών ή αλλαγές καθολικών μεταβλητών ή κλήσεις κάποιων συναρτήσεων
- Ένα δέντρο στο οποίο αποτυπώνονται και οι τιμές των γνωρισμάτων για κάθε κόμβο λέγεται *υποσημειωμένο δέντρο - annotated parse tree*

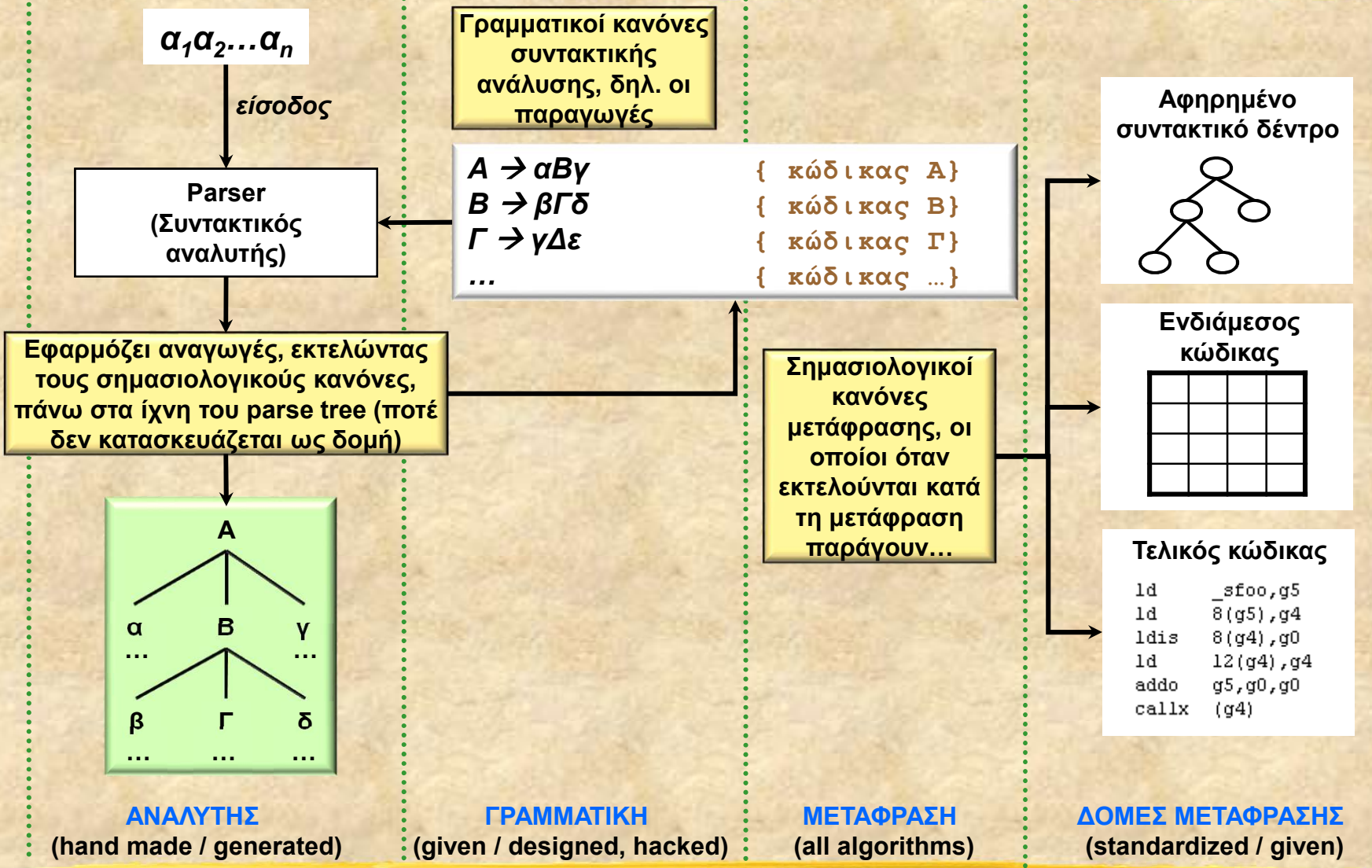
Γραμματικές γνωρισμάτων (4/11)

Γραμματικές γνωρισμάτων. Κάθε γραμματική παραγωγή της μορφής $A \rightarrow \alpha$ έχει ένα σύνολο από σημασιολογικούς κανόνες της μορφής: $b=f(c_1,\ldots,c_n)$ με f να είναι συνάρτηση και επιπλέον:

- b να είναι συντιθέμενο γνώρισμα του A και $c_1,\ldots,c_n$ να είναι γνωρίσματα των γραμματικών συμβόλων της παραγωγής.
- b να είναι κληρονομημένο γνώρισμα γραμματικού συμβόλου στο δεξί τμήμα της παραγωγής και $c_1,\ldots,c_n$ να είναι γνωρίσματα των γραμματικών συμβόλων της παραγωγής.

Σε οποιαδήποτε των δύο περιπτώσεων λέμε ότι το b *εξαρτάται* από τα γνωρίσματα $c_1,\ldots,c_n$. Μία γραμματική γνωρισμάτων είναι ένας συντακτικά οδηγούμενος ορισμός στον οποίο οι συναρτήσεις f στους σημασιολογικούς κανόνες δεν έχουν πλάγια αποτελέσματα.

Γραμματικές γνωρισμάτων (5/11)



Γραμματικές γνωρισμάτων (6/11)

Οι γραμματικές παραγωγές για συντακτικά οδηγούμενους ορισμούς, περιέχουν σημασιολογικούς κανόνες που χρησιμοποιούν τα γνωρίσματα των γραμματικών συμβόλων.

Για κάθε γραμματικό σύμβολο ***X***, το πεδίο ***X.sval*** είναι ο χώρος για την αποθήκευση της τιμής συντιθέμενου γνωρίσματος του *X* (*synthesized attribute value*), ενώ ***X.ival*** είναι αντίστοιχα ο χώρος αποθήκευσης τιμής κληρονομημένου γνωρίσματος (*inherited attribute value*).

Στην περίπτωση τερματικών συμβόλων (tokens) ***A***, το γνώρισμα είναι πάντα συντιθέμενο και έχει το αναγνωριστικό όνομα ***A.tval*** (token value).

ΣΗΜΑΣΙΟΛΟΓΙΚΟΙ ΚΑΝΟΝΕΣ ΜΕ ΣΥΝΤΙΘΕΜΕΝΑ ΓΝΩΡΙΣΜΑΤΑ

ΠΑΡΑΓΩΓΕΣ	ΣΗΜΑΣΙΟΛΟΓΙΚΟΙ ΚΑΝΟΝΕΣ	ΤΥΠΟΙ
$S \rightarrow expr$	**print**(*expr.sval*)	*χωρίς τύπο*
$expr \rightarrow expr_1 + term$	$expr.sval = expr_1.sval + term.sval$	*expr.sval*: **integer**
$expr \rightarrow term$	$expr.sval = term.sval$	
$term \rightarrow term_1 * prim$	$term.sval = term_1.sval * prim.sval$	*term.sval*: **integer**
$term \rightarrow prim$	$term.sval = prim.sval$	
$prim \rightarrow (expr)$	$prim.sval = expr.sval$	*prim.sval*: **integer**
$prim \rightarrow$ ***num***	$prim.sval =$ **num**$.tval$	*num.tval*: **integer**

Γραμματικές γνωρισμάτων (7/11)

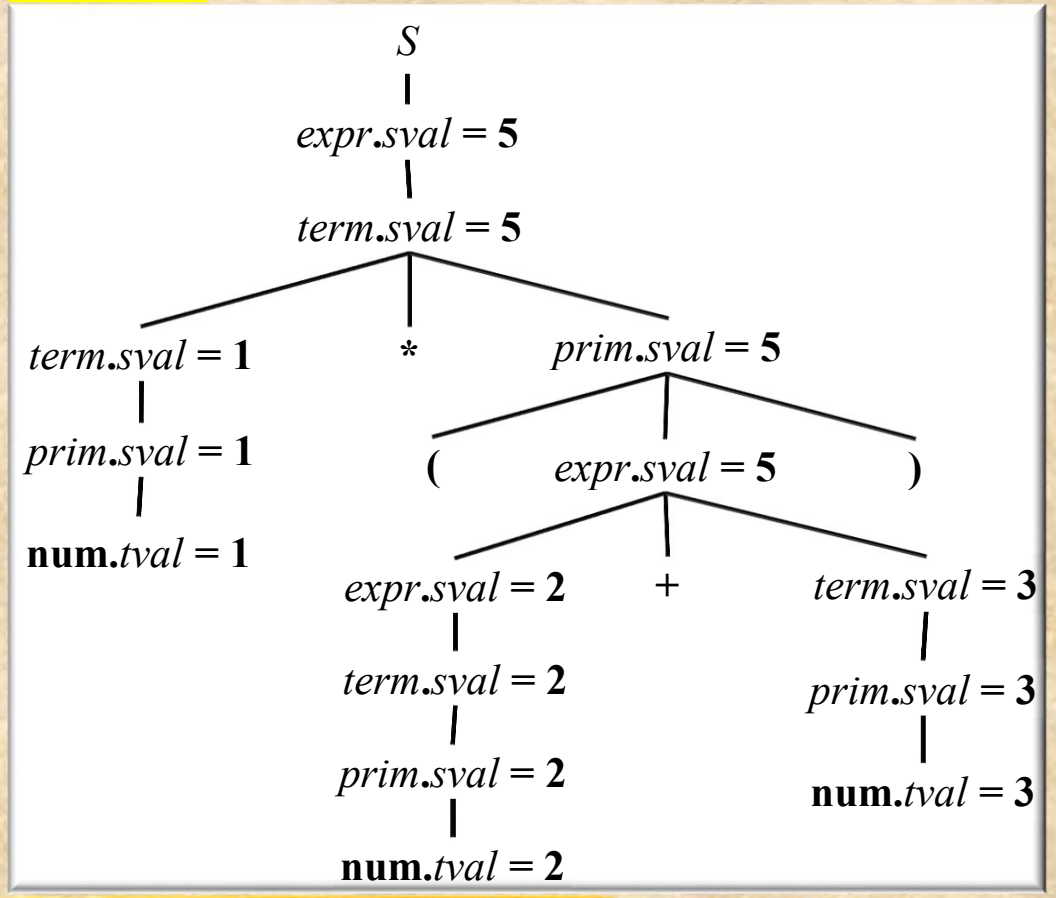
- Θα δούμε για κάποια λέξη εισόδου το υποσημειωμένο (annotated) δέντρο συντακτικής ανάλυσης που αντιστοιχεί στον συντακτικά οδηγούμενο ορισμό **1*(2+3)**

•Η τεχνική με synthesized attributes είναι η πιο διαδεδομένη στην πράξη.

•Ένας συντακτικά οδηγούμενος ορισμός που χρησιμοποιεί αποκλειστικά συντιθέμενα γνωρίσματα ονομάζεται S-ορισμός γνωρισμάτων (S-attributed definition).

•Ένα δέντρο συντακτικής ανάλυσης για S-ορισμό γνωρισμάτων μπορεί πάντα να υποσημειωθεί εκτελώντας τους σημασιολογικούς κανόνες ανοδικά.

•Σε μερικές περιπτώσεις ορισμένοι σημασιολογικοί κανόνες προκαλούν κάποια πλάγια αποτελέσματα, οπότε ορίζονται ως κώδικας ενώ μπορεί να μην χρησιμοποιείται αντίστοιχο γνώρισμα.



Γραμματικές γνωρισμάτων (8/11)

- Οι σημασιολογικοί κανόνες και ο τύπος των γνωρισμάτων δεν περιορίζονται σε απλές εκφράσεις και τύπους

expr → $expr_1$ + $expr_2$

```
if (expr1.sval.type == int_t && expr2.sval.type == int_t) then {
        expr.sval.type = int_t;
        expr.sval.val.intVal = expr1.sval.val.intVal+ expr2.sval.val.intVal;
}
else {
        error("type mismatch in +");
        expr.sval.type = error_t;
}
```

expr → $expr_1$ ***and*** $expr_2$

```
if (expr1.sval.type == bool_t && expr2.sval.type == bool_t) then {
        expr.sval.type = bool_t;
        expr.sval.val.boolVal = expr1.sval.val.boolVal && expr2.sval.val.boolVal;
}
else {
        error("type mismatch in &&");
        expr.sval.type = error_t;
}
```

Γραμματικές γνωρισμάτων (9/11)

expr → **bool**
expr.sval.type = *bool_t*;
expr.sval.val.boolVal = **bool**.*tval;*

expr → **num**
expr.sval.type = *int_t*;
expr.sval.val.intVal = **num**.*tval;*

expr → (*expr*$_1$)
expr.sval = *expr*$_1$.*sval*;

```
enum expr_t { int_t, bool_t, error_t };
struct expr {
     expr_t type;
     union {
          int                intVal;
          unsigned char      boolVal;
     } val;
};
```

expr.sval: `struct expr;`

- Οι σημασιολογικοί κανόνες γενικεύονται ως τμήματα προγράμματος τα οποία εκτελούνται από τον αναλυτή όταν εφαρμόζονται οι αντίστοιχες αναγωγές.
- Ο τύπος των γνωρισμάτων μπορεί να οριστεί όπως είναι επιθυμητό, ανάλογα με το είδος της πληροφορίας που θέλουμε να διοχετευτεί στα ανώτερα επίπεδα ανάλυσης (δεν μας απασχολεί ο τρόπος αποθήκευσης).
- Εάν ένα γραμματικό σύμβολο μπορεί να αντιπροσωπεύει πολλούς διαφορετικούς τύπους τότε χρησιμοποιούμε ένα τύπο που ενοποιεί τους επιμέρους τύπους (όπως π.χ. ένα C union)

Γραμματικές γνωρισμάτων (10/11)

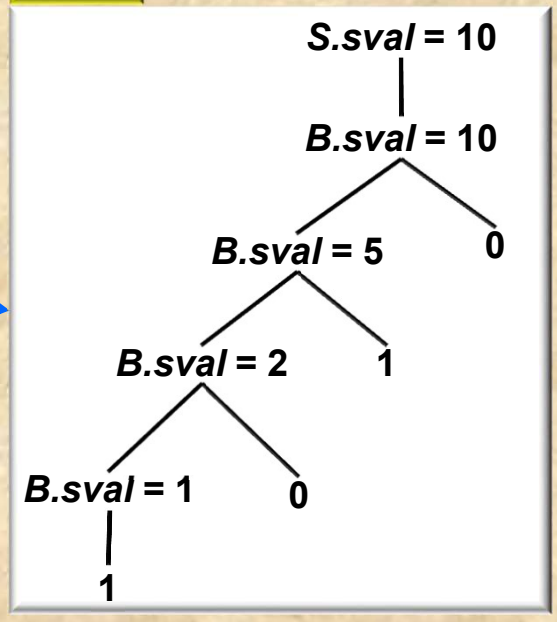
Παράδειγμα. Γραμματική και συντακτικά οδηγούμενη μετάφραση για αναγνώριση δυαδικών αριθμών με ταυτόχρονη μετατροπή σε τιμή δεκαδικού συστήματος.

$S \rightarrow B$	{ $S.sval = B.sval$; }
$B \rightarrow 0$	{ $B.sval = 0$; }
$B \rightarrow 1$	{ $B.sval = 1$; }
$B \rightarrow B0$	{ $B.sval = 2*B.sval$; }
$B \rightarrow B1$	{ $B.sval = 2*B.sval+1$; }

Κανόνες με συντιθέμενα γνωρίσματα

…περιμένουμε να έχουμε δέντρο τέτοιας δομής καθώς έχουμε αριστερά αναδρομική γραμματική.

1010



Υποσημειωμένο δέντρο

Γραμματικές γνωρισμάτων (11/11)

- *Οι σημασιολογικοί κανόνες περιέχουν τη λογική μετάφρασης στην περίπτωση αναγωγής (ανοδική ανάλυση) ή επιτυχούς αναγνώρισης δοκιμαζόμενης παραγωγής (καθοδική ανάλυση)*

Το παρακάτω σχήμα μετάφρασης με συντιθέμενα γνωρίσματα αποτυπώνει τη λογική μετάφρασης του RDP της 6ης διάλεξης, διαφάνεια 7. Εδικά στην κατασκευή RDP με back-tracking, ποτέ δεν φαίνεται στους σημασιολογικούς κανόνες η αλγοριθμική λογική ακύρωσης ενεργειών που έχουν ήδη γίνει λόγω μίας αποτυχημένης παραγωγής (όπως τα free actions του συγκεκριμένου παραδείγματος).

while → **while (** *expr* **)** *stmt*
{ *while.sval* = *make_while*(*expr.sval*, *stmt.sval*); }

if → **if (** *expr* **)** *stmt*
{ *if.sval* = *make_ifelse*(*expr.sval*, *stmt.sval*, *(node*) 0*); }

if → **if (** *expr* **)** $stmt_1$ **else** $stmt_2$
{ *if.sval* = *make_ifelse*(*expr.sval*, $stmt_1$*.sval*,$stmt_2$*.sval*); }

typeof<while.sval>	= **node***
typeof<if.sval>	= **node***
typeof<expr.sval>	= **node***
typeof<stmt.sval>	= **node***

Ο τύπος του κάθε γραμματικού συμβόλου αποφασίζεται ελεύθερα ανάλογα με τις ανάγκες της μετάφρασης. Εδώ είναι ίδιος με τις δομές που χρησιμοποιούνται στον αντίστοιχο RDP για την κατασκευή του αφηρημένου συντακτικού δέντρου (AST – abstract syntax tree)

Περιεχόμενα

- Η αναγκαιότητα
- Γραμματικές γνωρισμάτων
- *Υλοποίηση σε LR parsers*
- Υλοποίηση σε LL parsers
- Προγραμματιστική συμβουλή

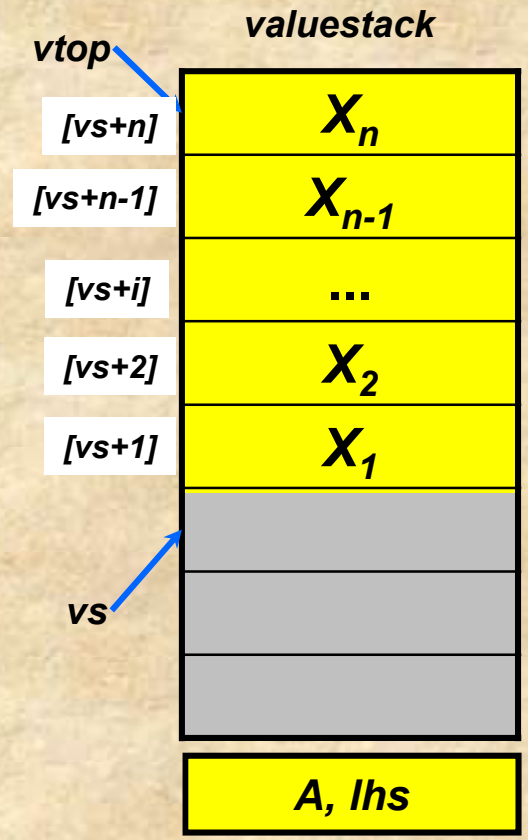
Υλοποίηση σε LR parsers (1/24)

- Η υποστήριξη των σημασιολογικών κανόνων σε shift-reduce parser είναι σχετικά εύκολη με απλές επεκτάσεις στη λειτουργία και εσωτερική δομή του parser
  - Προσθέτουμε μία *στοίβα τιμών* που διατηρείται παράλληλα με τη στοίβα των γραμματικών συμβόλων
  - Για κάθε σύμβολο στην στοίβα συμβόλων έχουμε την τιμή του στην αντίστοιχη θέση της στοίβας τιμών
  - Για τα *τερματικά σύμβολα* αποθηκεύονται οι *τιμές* των γνωρισμάτων όπως έχουν έρθει *από τον λεξικογραφικό αναλυτή*
  - Για τα *μη τερματικά σύμβολα* αποθηκεύονται οι *τιμές* όπως έχουν υπολογιστεί *από τους σημασιολογικούς κανόνες*
  - *Όταν τελειώσει η μετάφραση* η στοίβα τιμών θα έχει μόνο μία *τιμή*, αυτή *του* μη τερματικού *αρχικού συμβόλου*, η οποία και θα *αντιστοιχεί στη μετάφραση της λέξης εισόδου*

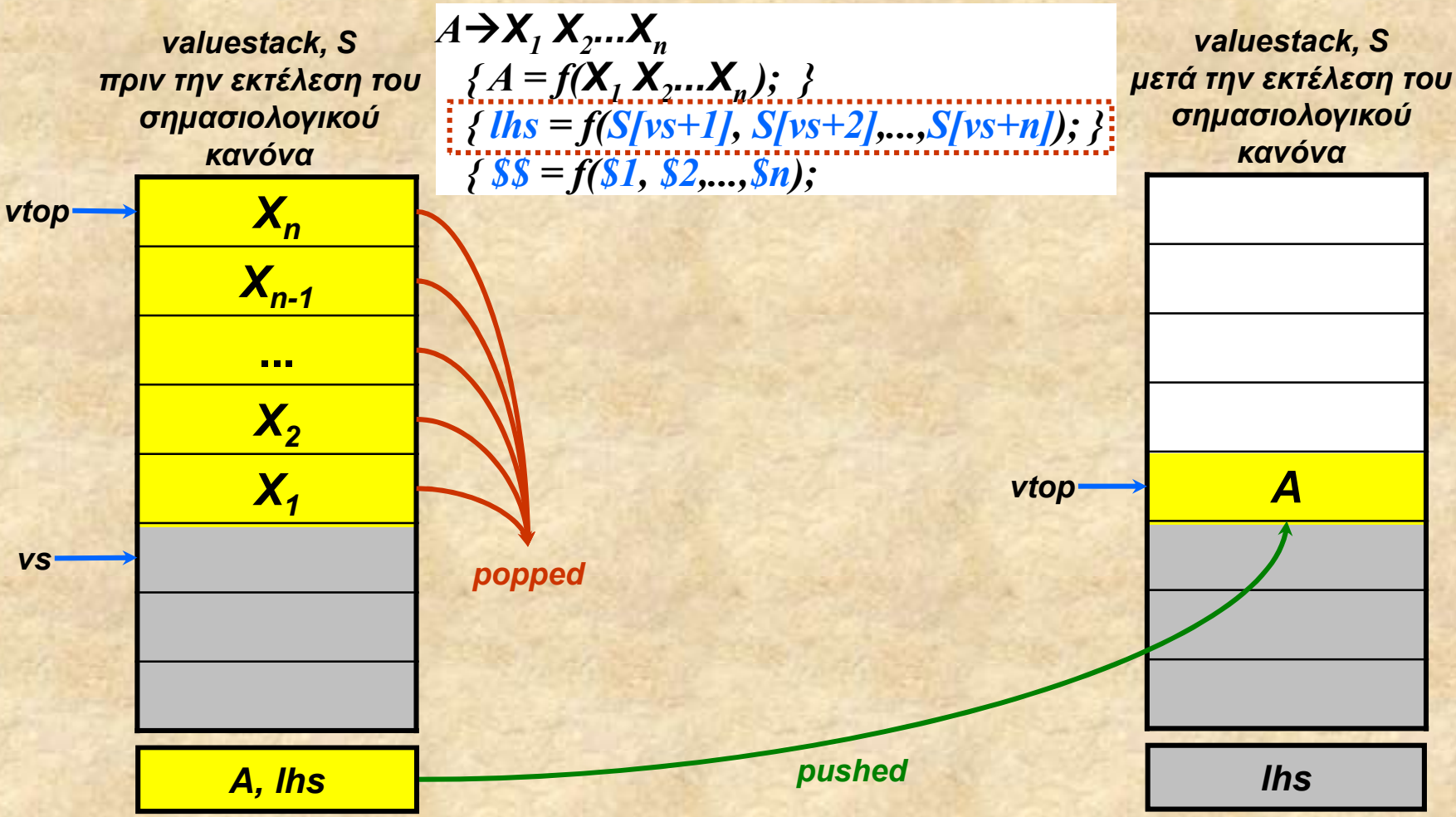
Υλοποίηση σε LR parsers (2/24)

Επιπλέον ενέργειες του LR parser για τους σημασιολογικούς κανόνες

- *shift*. Κάνε *push* την λεξικογραφική τιμή του τερματικού συμβόλου στη στοίβα τιμών. Εάν αυτό δεν αντιστοιχεί σε κάποια τιμή κάνε *push* μία γνωστή τιμή *void*.
- *reduce*. Έστω αναγωγή με τον κανόνα $A \rightarrow X_1\ X_2 ... X_n$. Τότε κάνε τα εξής:
  - $vs = vtop - n$, με *vtop* το δείκτη στην κορυφής της στοίβας τιμών.
  - Εκτέλεσε τον σημασιολογικό κανόνα για την παραπάνω παραγωγή, αντιστοιχώντας τις συμβολικές μεταβλητές γραμματικών συμβόλων ως εξής:
    - μεταφραστική τιμή του σύμβολου X_i στο στοιχείο *valuestack*[*vs*+*i*] της στοίβα τιμών, με *valuestack* την στοίβα τιμών.
    - το αριστερό μη-τερματικό σύμβολο στην προσωρινή μεταβλητή *lhs*.
  - Κάνε *pop n* τιμές από τη στοίβα τιμών.
  - Κάνε *push* στη στοίβα τιμών την τιμή του *lhs*.



Υλοποίηση σε LR parsers (3/24)



Υλοποίηση σε LR parsers (4/24)

- Φαίνεται λοιπόν ο τρόπος με τον οποίο αντιστοιχούν οι συμβολισμοί για τα γραμματικά σύμβολα στους σημασιολογικούς κανόνες σε πραγματικές μεταβλητές (κελιά της στοίβας τιμών), μέσω του βοηθητικού δείκτη *vs*
- Αυτό επιτρέπει στους προγραμματιστές να έχουν έναν κοινό συμβολικό τρόπο πρόσβασης στα σύμβολα ενός κανόνα, όπως *$$* για το μη τερματικό σύμβολο στο LHS, καθώς και *$i* για το *i*-οστό σύμβολο στο RHS (διαβάζοντας από αριστερά προς δεξιά).
- Όμως κατασκευαστικά, για να έχουμε μία στοίβα τιμών που να μπορεί σε ένα οποιοδήποτε κελί της να αποθηκεύει τιμή οποιουδήποτε γραμματικού συμβόλου, σημαίνει ότι έχω έναν κοινό τύπο για όλα τα γραμματικά σύμβολο.

Υλοποίηση σε LR parsers (5/24)

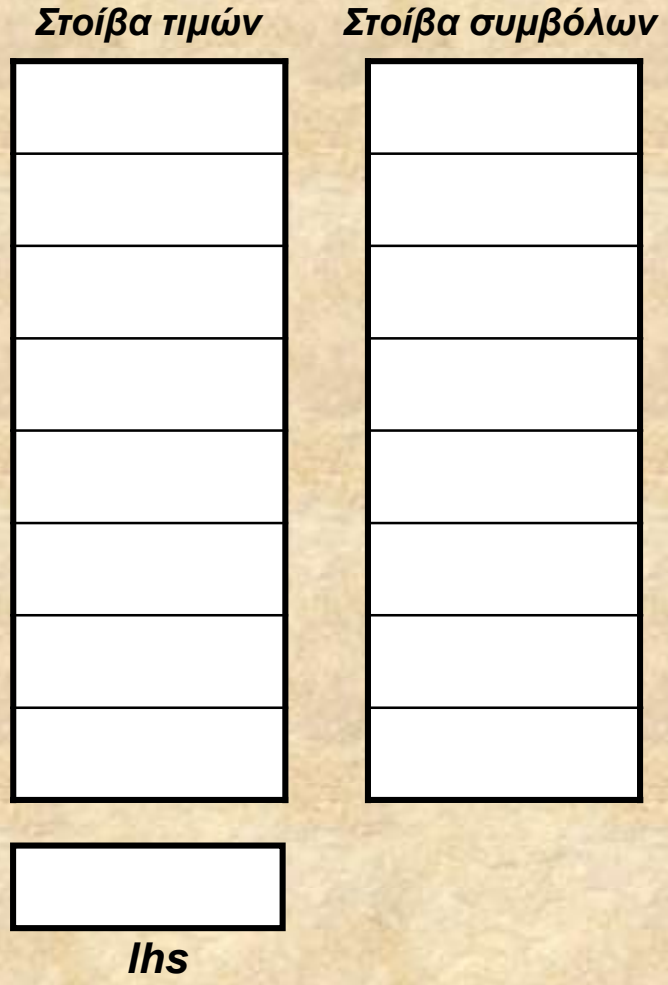
Έστω συνολικά n ο μέγιστος αριθμός γραμματικών συμβόλων τα οποία έχουν διαφορετικούς τύπους τιμών (γνωρισμάτων) μεταξύ τους και έστω T_i, i:$0...n$ να αντιστοιχεί σε κάθε τέτοιο ξεχωριστό τύπο. Τότε ορίζω ως τύπο τιμών στοίβας VT το παρακάτω *union* (δηλ. ενοποιημένο τύπο):

union VT { T_1 v_1; T_2 v_2; ... T_n v_n; };

Ορίζω επιπλέον την αντιστοιχία κάθε διαφορετικού γραμματικού συμβόλου X με τύπο γνωρισμάτων $type(X)= T_j$ στο πεδίο v_j. Προφανώς όλα τα γραμματικά σύμβολα ιδίου τύπου αντιστοιχούν στο ίδιο πεδίο του ενοποιημένου τύπου VT. Το αποθηκευτικό μέγεθος του VT είναι αυτό του μέγιστου αποθηκευτικού μεγέθους του τύπου T_i.

Κατά την εκτέλεση των σημασιολογικών κανόνων, στην αντιστοιχία οποιουδήποτε γραμματικού συμβόλου B (είτε στο LHS ή στο RHS) σε αποθηκευτικό χώρο του parser (*lhs* ή $S[vs+i]$), προσθέτω και το επίθεμα $.v_j$ όπου $typeof(B)= T_j$.

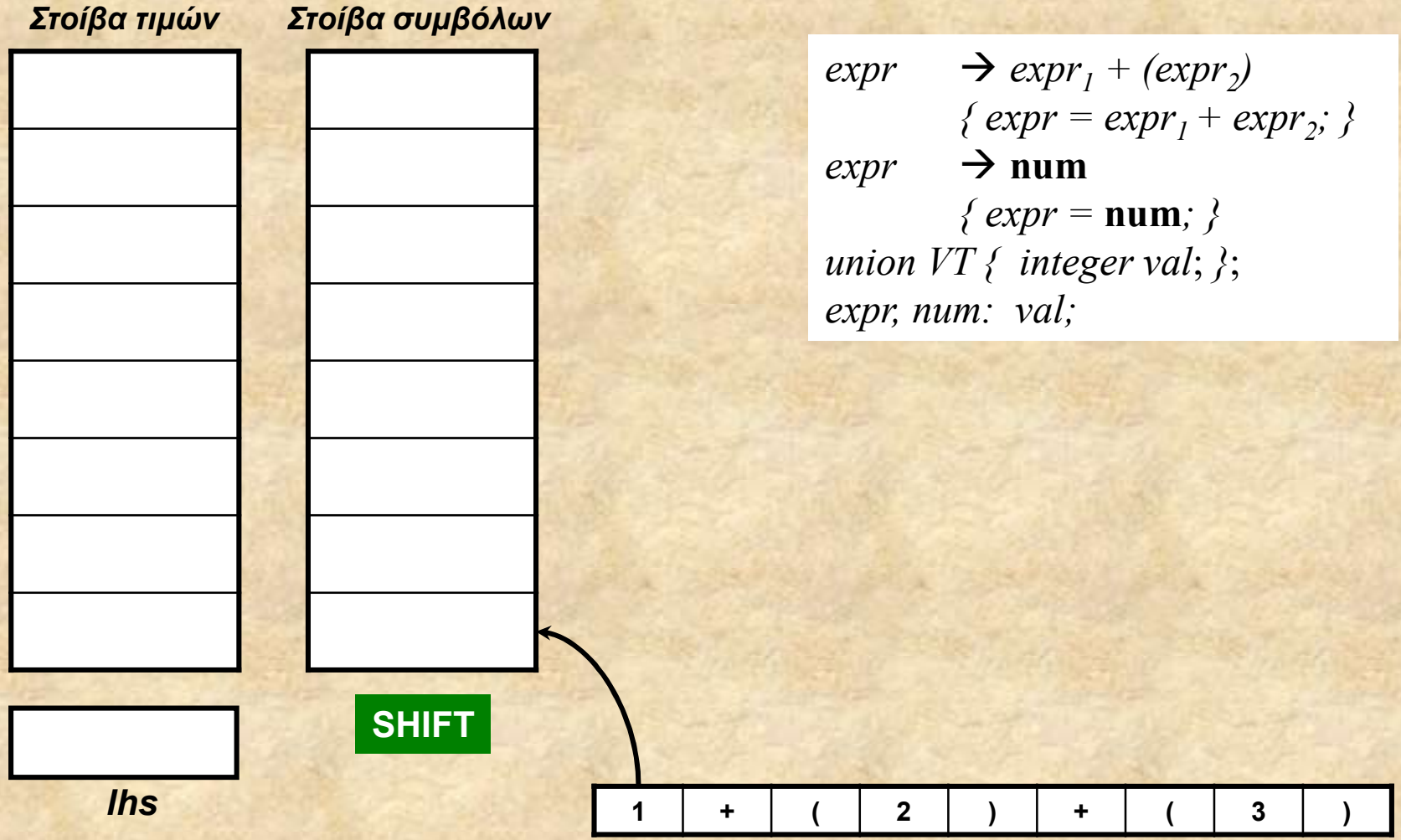
Υλοποίηση σε LR parsers (6/24)



expr → $expr_1$ + ($expr_2$)
{ *expr* = $expr_1$ + $expr_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

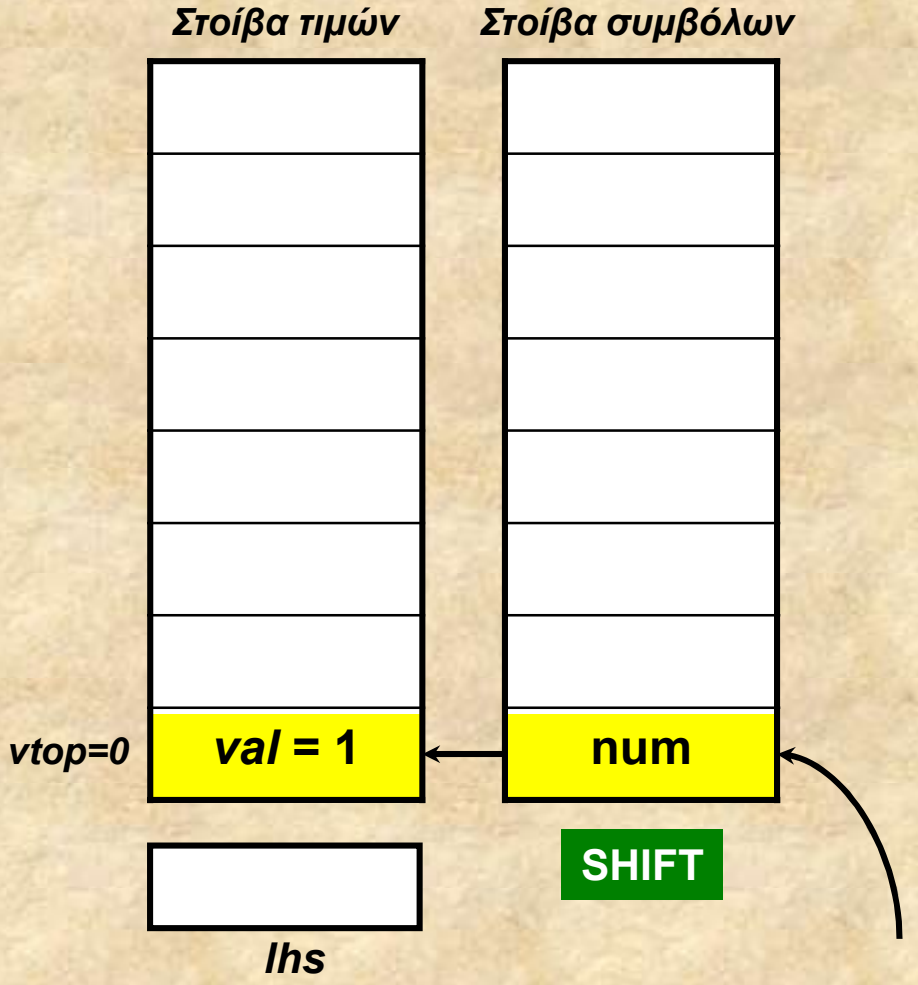
1	+	(	2	)	+	(	3	)

Υλοποίηση σε LR parsers (7/24)



expr → $expr_1$ + ($expr_2$)
{ *expr* = $expr_1$ + $expr_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

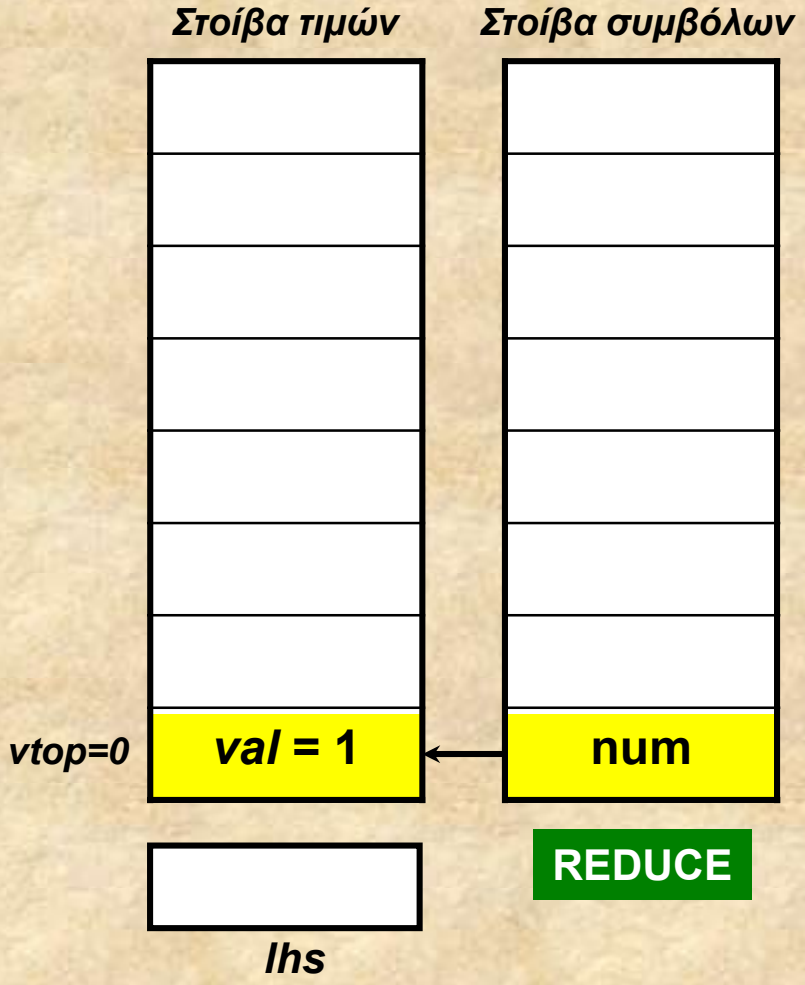
Υλοποίηση σε LR parsers (8/24)



expr → $expr_1$ + ($expr_2$)
{ *expr* = $expr_1$ + $expr_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

+	(	2	)	+	(	3	)

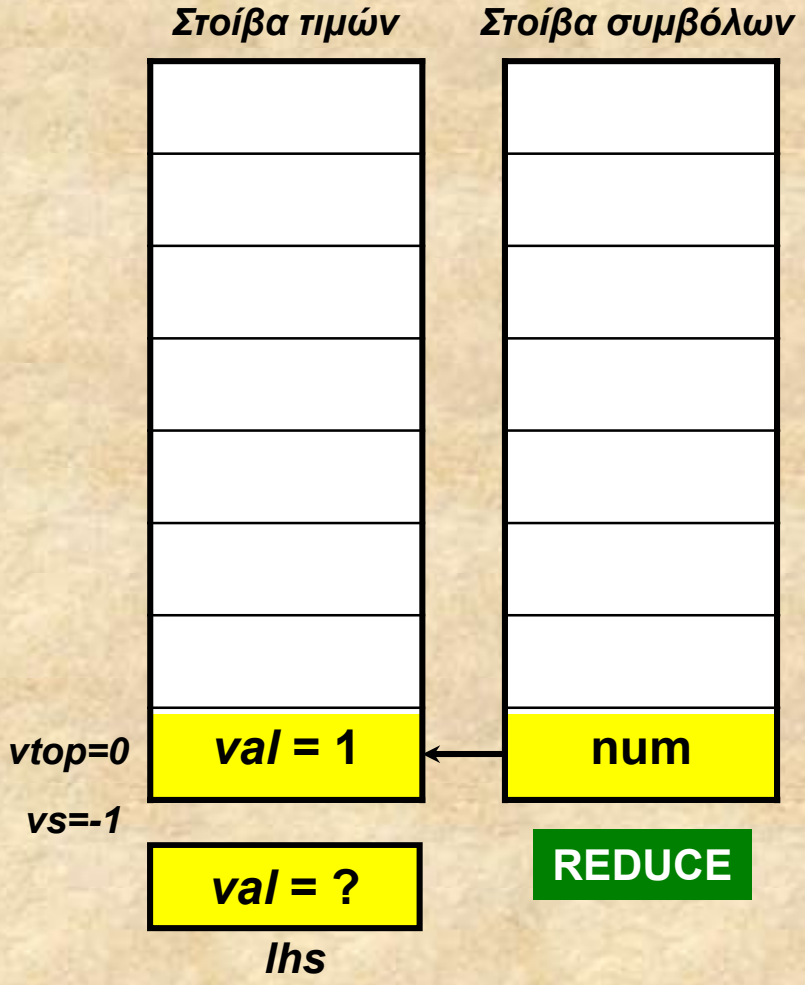
Υλοποίηση σε LR parsers (9/24)



$expr \rightarrow expr_1 + (expr_2)$
$\{ expr = expr_1 + expr_2; \}$
$expr \rightarrow$ **num**
$\{ expr =$ **num**$; \}$
union VT { integer val; };
expr, num: val;

+	(	2	)	+	(	3	)

Υλοποίηση σε LR parsers (10/24)



expr → *expr*$_1$ + (*expr*$_2$)
{ *expr* = *expr*$_1$ + *expr*$_2$; }

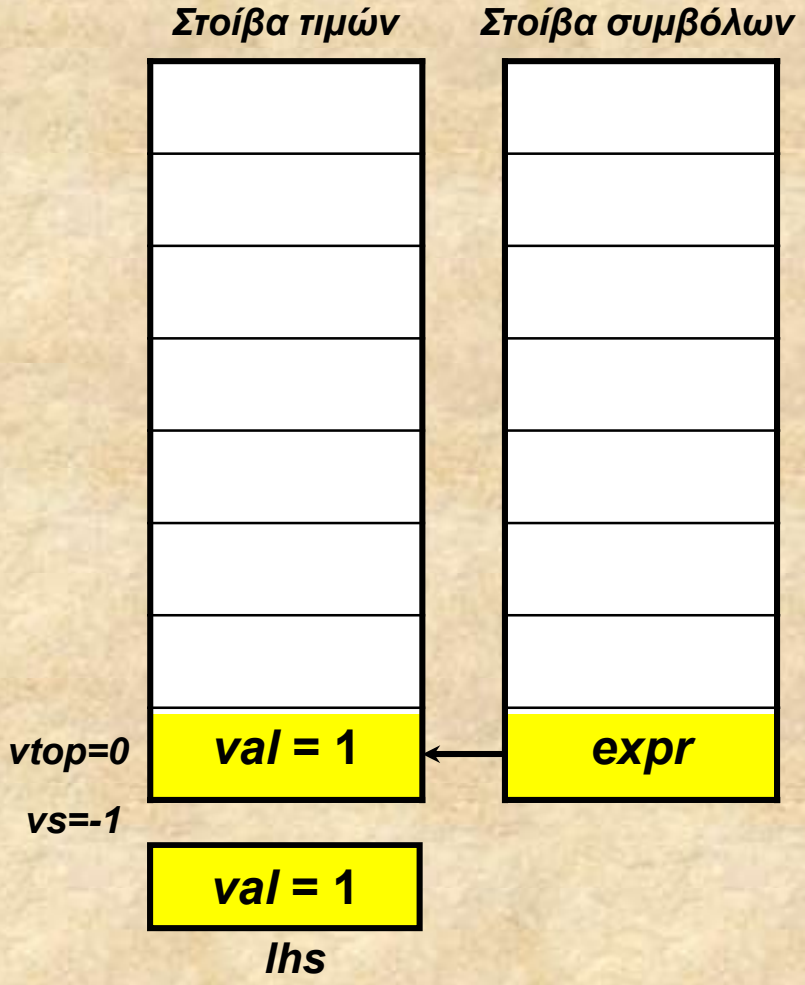
expr → **num**
{ *expr* = **num**; }

union VT { *integer val*; };
expr, num: val;

***vs* = *vtop*-|RHS| => *vs* = 0–1 => *vs* = -1**
{ *lhs.val* = *S*[*vs*+1].*val*; }

+	(	2	)	+	(	3	)

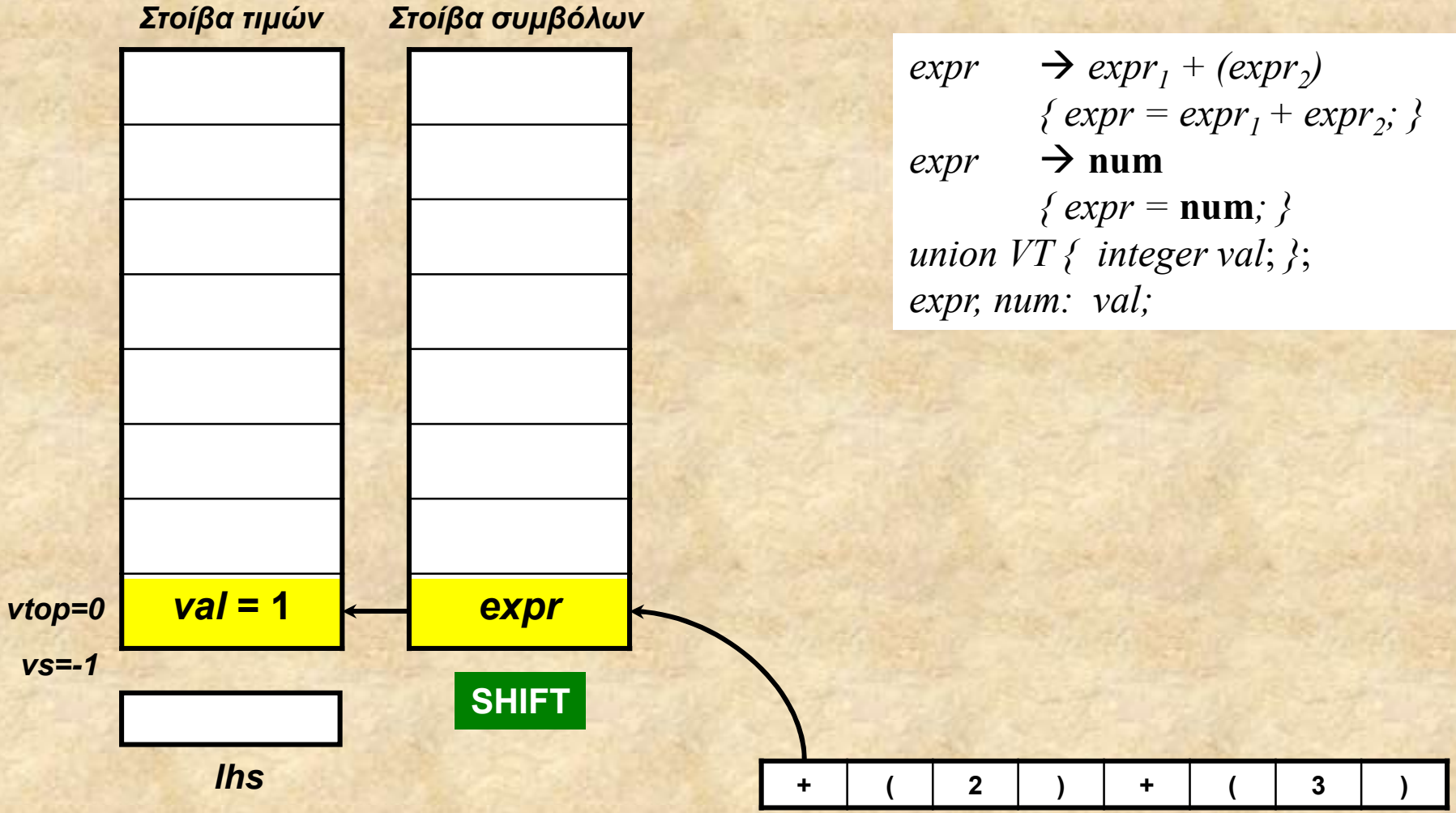
Υλοποίηση σε LR parsers (11/24)



expr → $expr_1$ + ($expr_2$)
{ *expr* = $expr_1$ + $expr_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

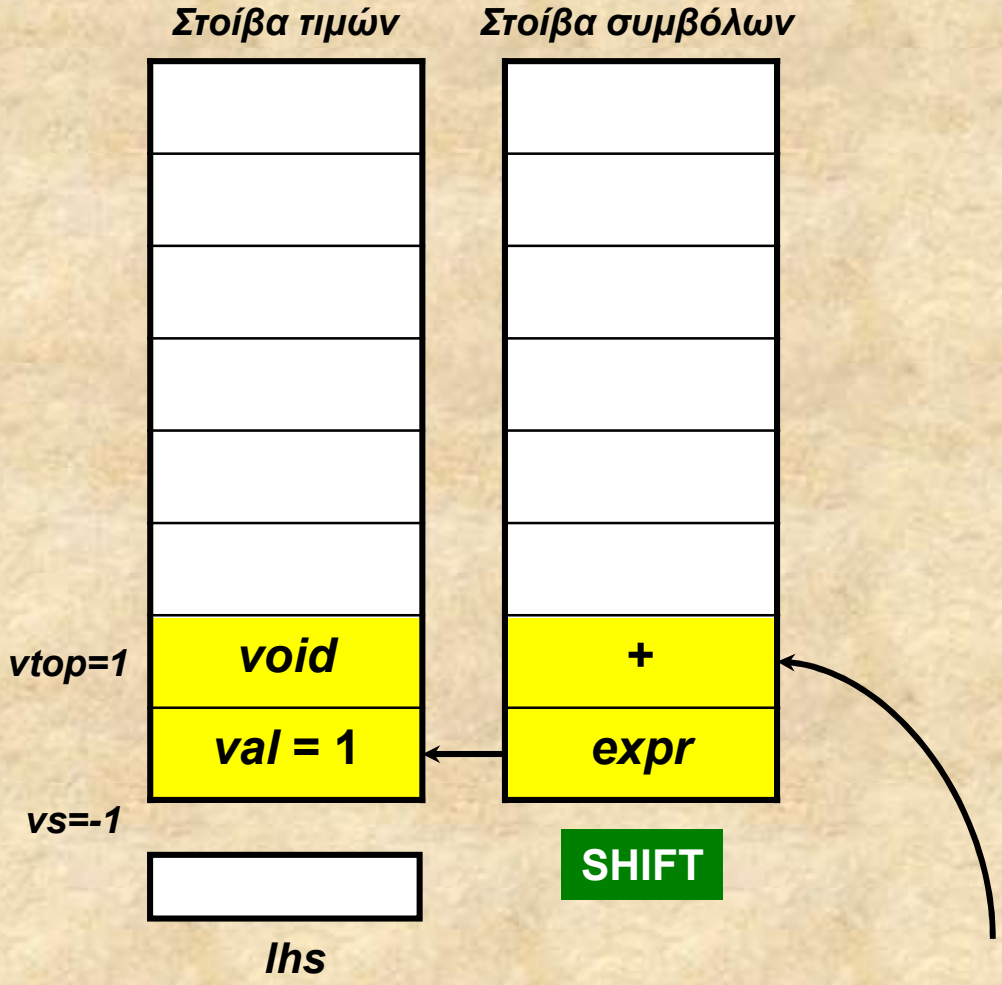
+	(	2	)	+	(	3	)

Υλοποίηση σε LR parsers (12/24)



expr → $expr_1$ + ($expr_2$)
{ *expr* = $expr_1$ + $expr_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

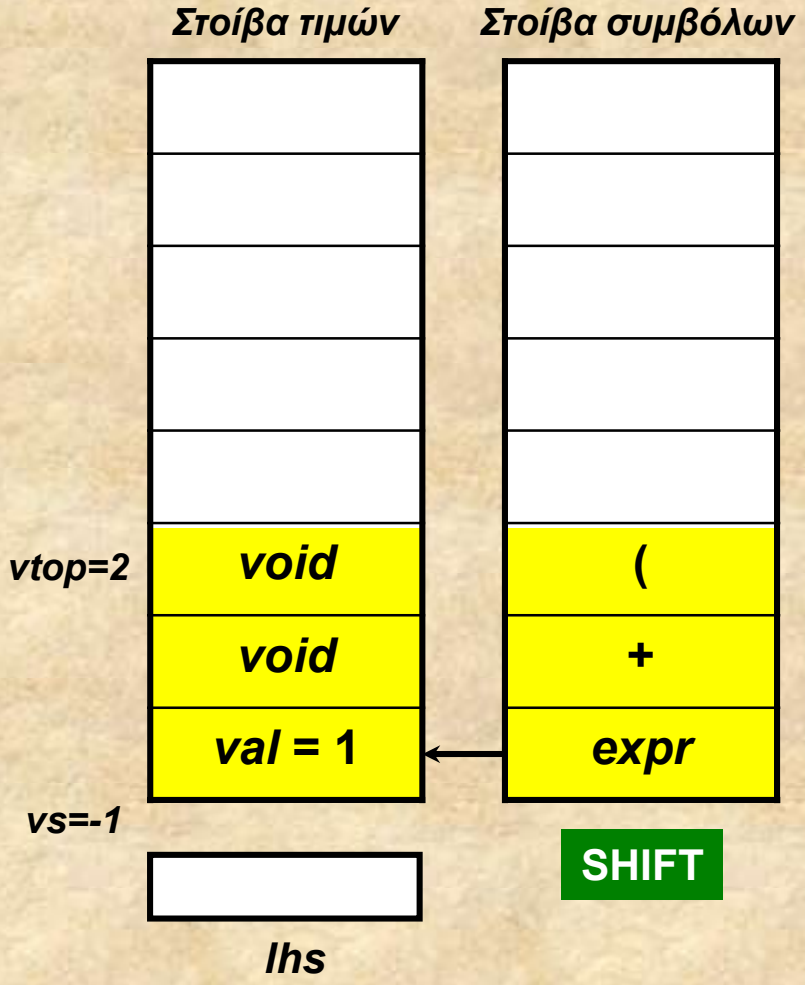
Υλοποίηση σε LR parsers (13/24)



expr → $expr_1$ + ($expr_2$)
{ *expr* = $expr_1$ + $expr_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

(	2	)	+	(	3	)

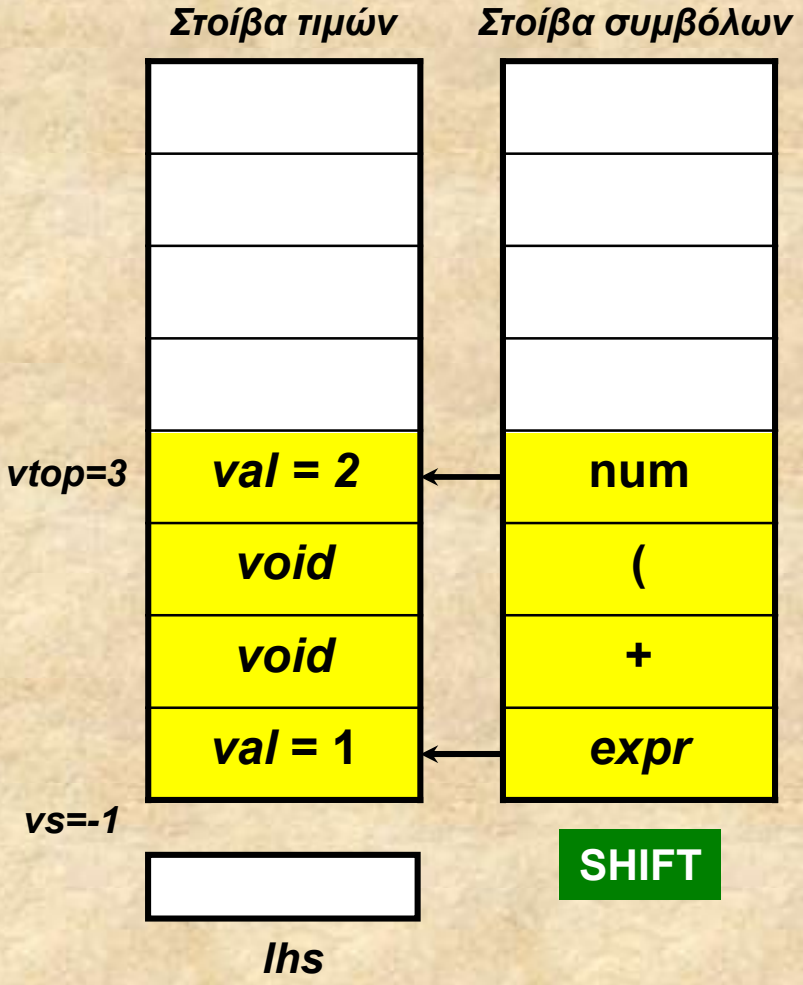
Υλοποίηση σε LR parsers (14/24)



expr → $expr_1$ + ($expr_2$)
{ *expr* = $expr_1$ + $expr_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

2	)	+	(	3	)

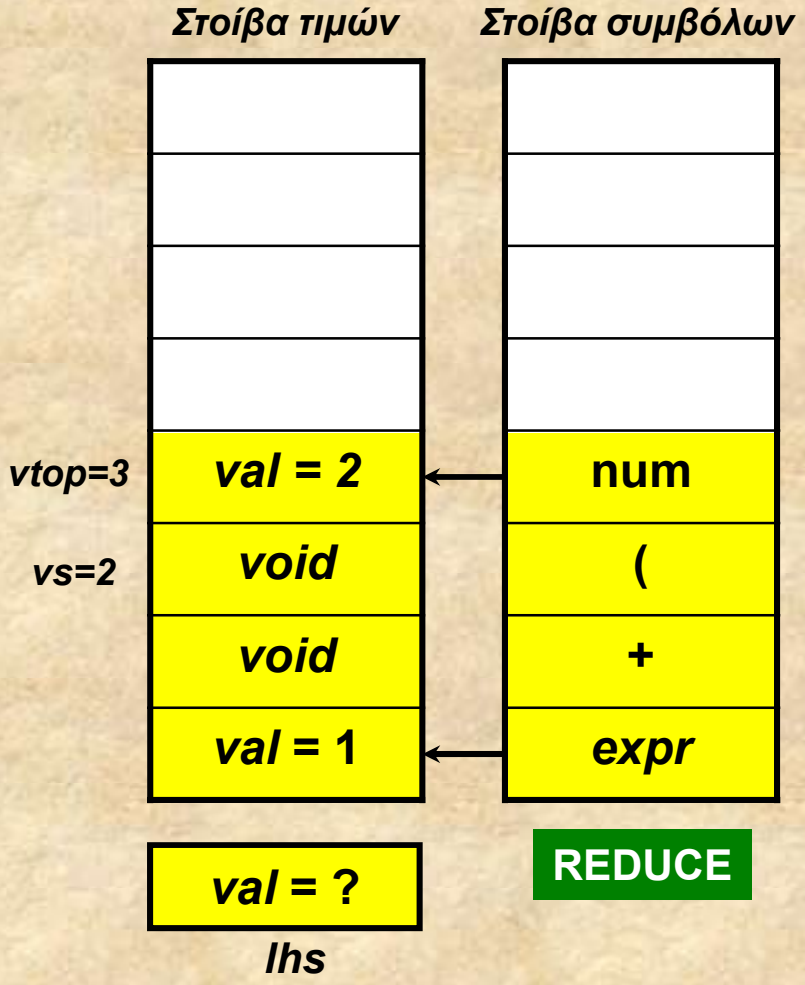
Υλοποίηση σε LR parsers (15/24)



expr → *expr*$_1$ + (*expr*$_2$)
{ *expr* = *expr*$_1$ + *expr*$_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

)	+	(	3	)

Υλοποίηση σε LR parsers (16/24)

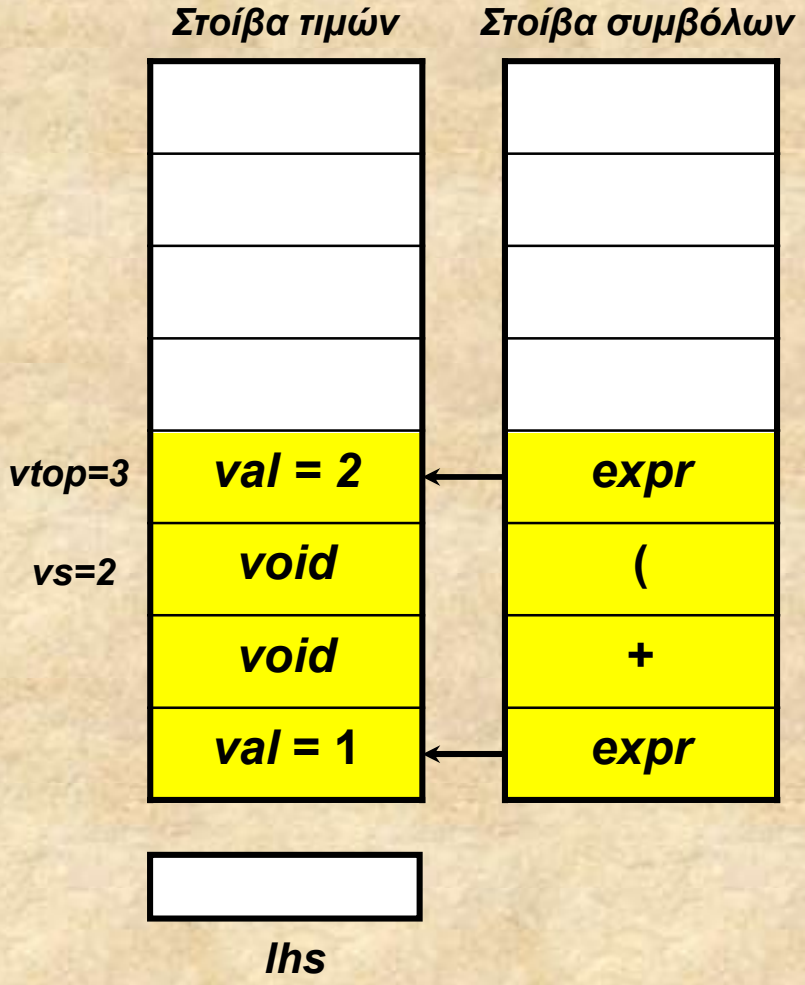


expr → *expr*$_1$ + (*expr*$_2$)
{ *expr* = *expr*$_1$ + *expr*$_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

***vs* = *vtop*-|RHS| => *vs* = 3–1 => *vs* = 2**
{ *lhs.val* = *S*[*vs*+1].*val*; }

)	+	(	3	)

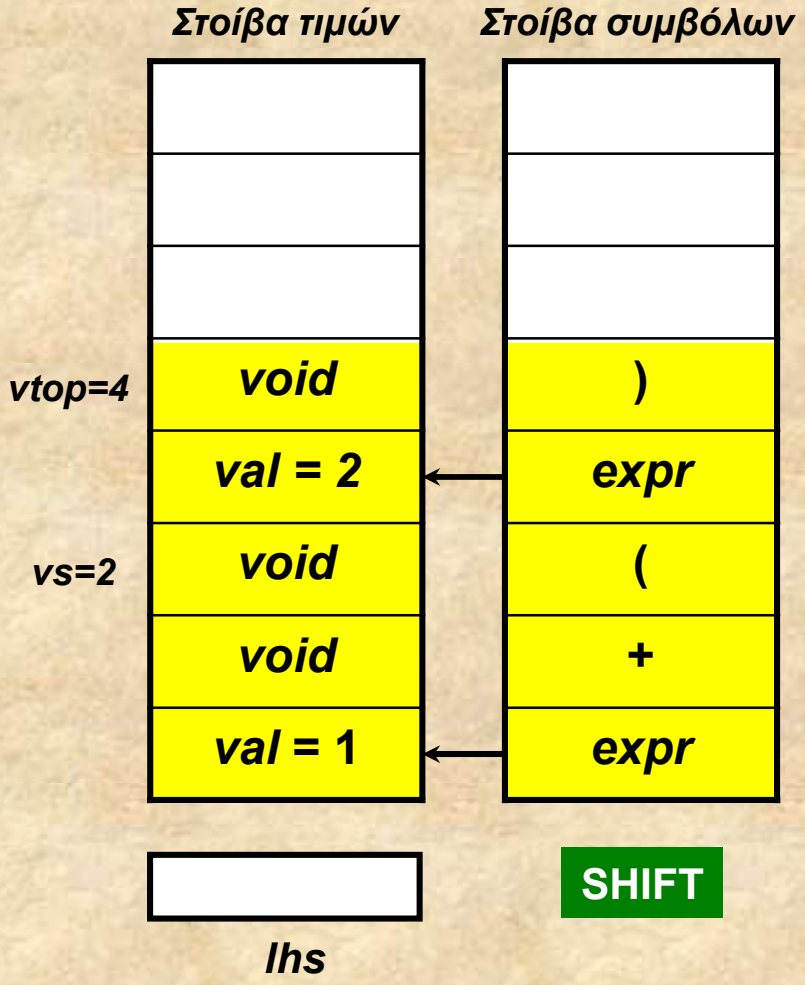
Υλοποίηση σε LR parsers (17/24)



expr → $expr_1$ + ($expr_2$)
{ *expr* = $expr_1$ + $expr_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

)	+	(	3	)

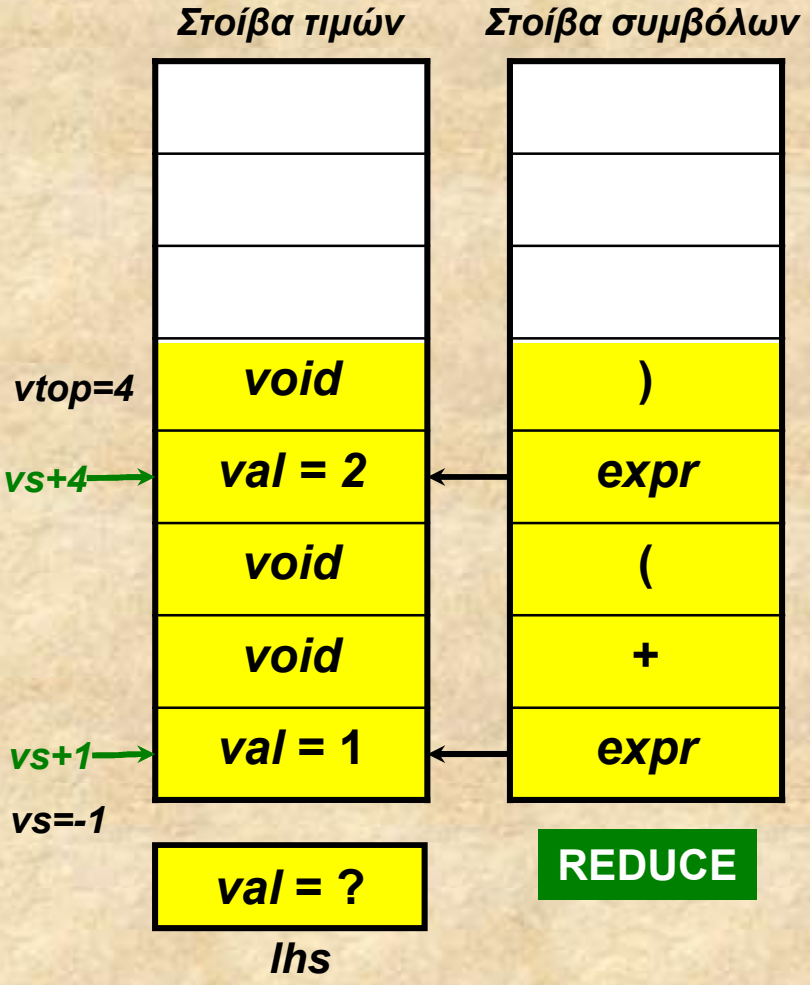
Υλοποίηση σε LR parsers (18/24)



expr → *expr*$_1$ + (*expr*$_2$)
{ *expr* = *expr*$_1$ + *expr*$_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

+	(	3	)

Υλοποίηση σε LR parsers (19/24)



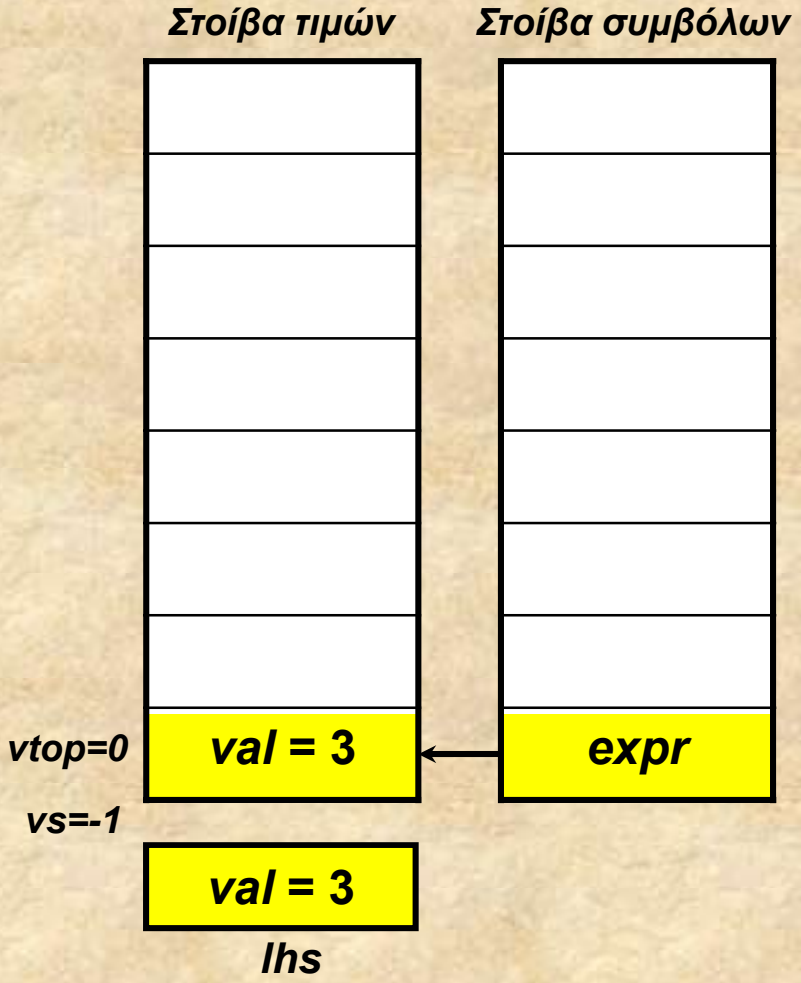
Το RHS έχει 5 σύμβολα

expr → *expr*$_1$ + (*expr*$_2$)
{ *expr* = *expr*$_1$ + *expr*$_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

***vs* = *vtop*-|RHS| => *vs* = 4–5 => *vs* = -1**
{ *lhs.val* = *S*[*vs*+1].*val* + *S*[*vs*+4].*val*; }

+	(	3	)

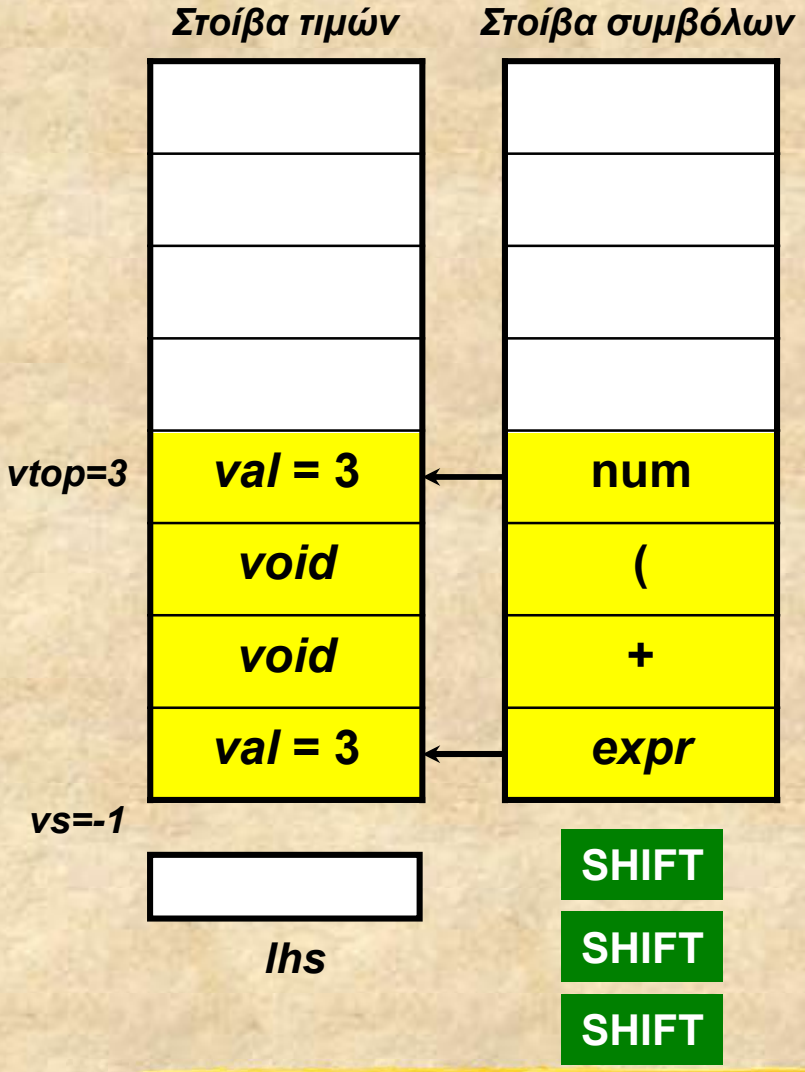
Υλοποίηση σε LR parsers (19/24)



expr → $expr_1$ + ($expr_2$)
{ *expr* = $expr_1$ + $expr_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

+	(	3	)

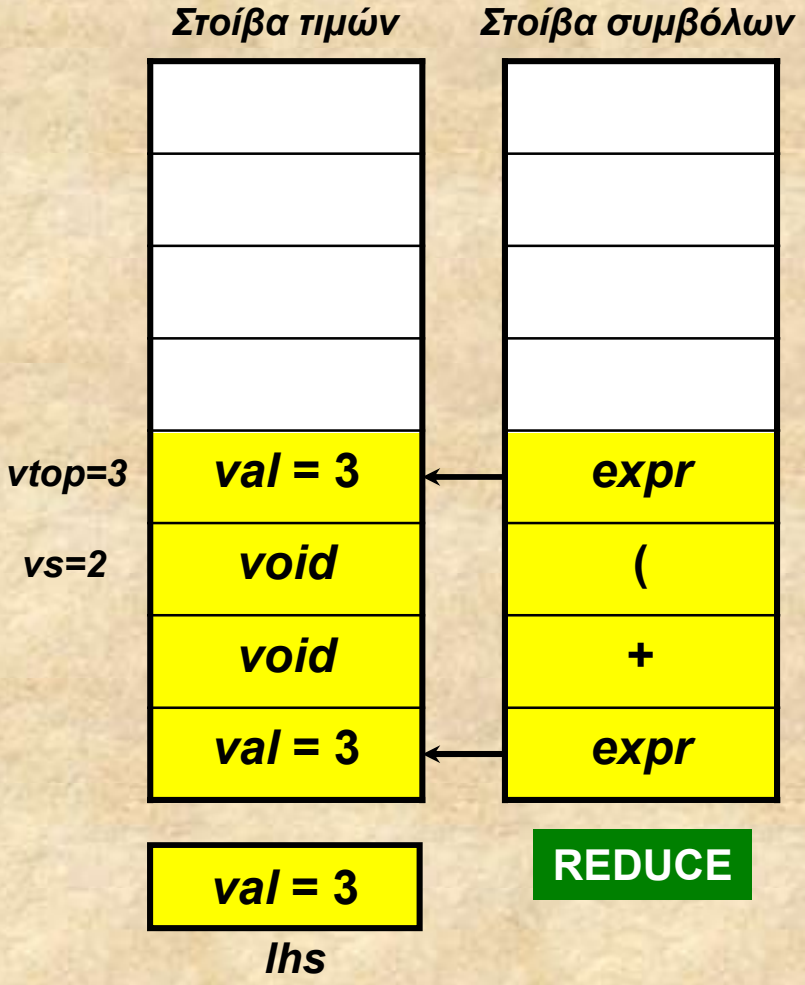
Υλοποίηση σε LR parsers (20/24)



expr → *expr*$_1$ + (*expr*$_2$)
{ *expr* = *expr*$_1$ + *expr*$_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

)

Υλοποίηση σε LR parsers (21/24)

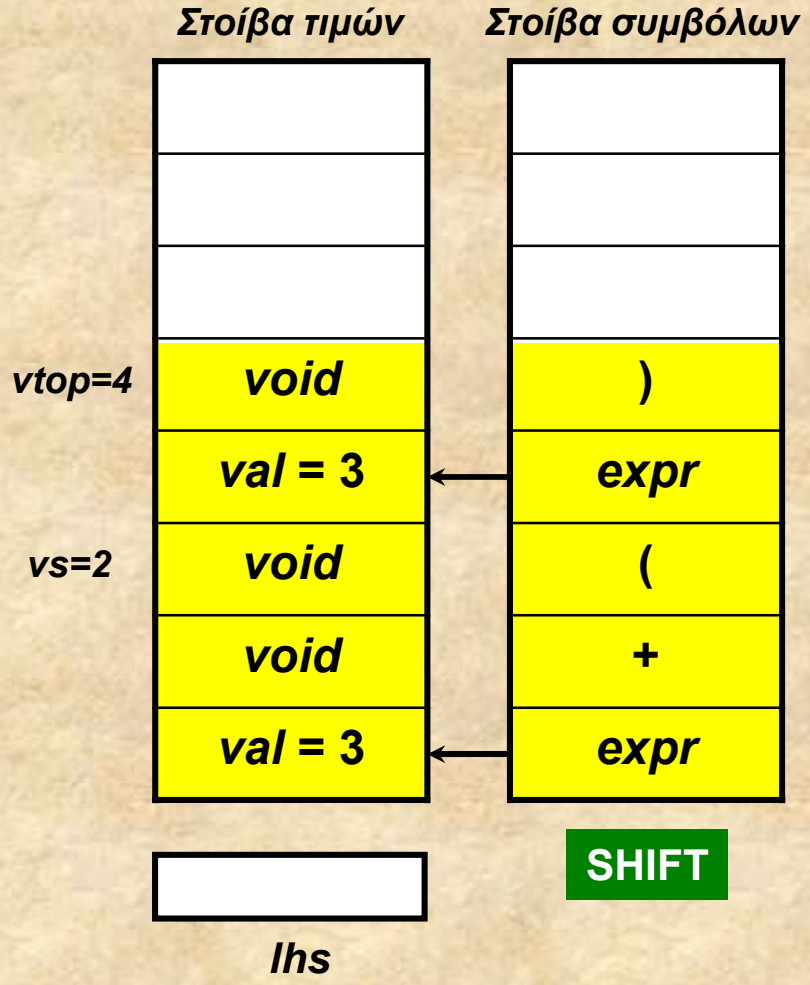


expr → *expr*$_1$ + (*expr*$_2$)
{ *expr* = *expr*$_1$ + *expr*$_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

***vs* = *vtop*-|RHS| => *vs* = 3–1 => *vs* = 2**
{ *lhs.val* = *S*[*vs*+1].*val*; }

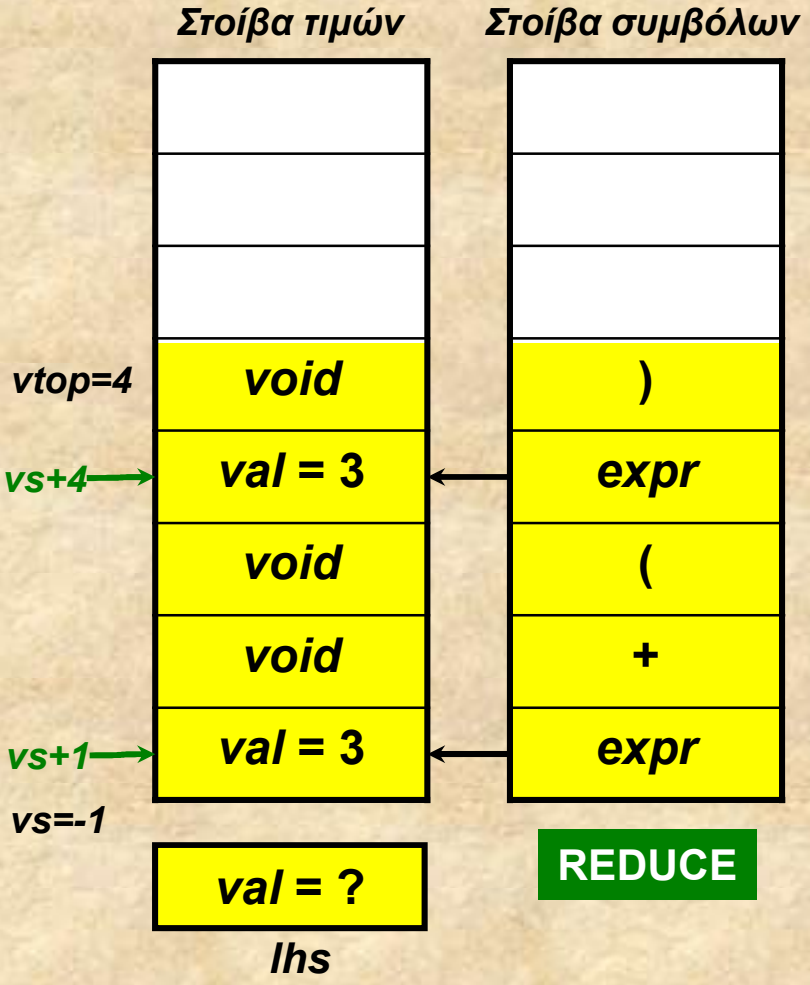
)

Υλοποίηση σε LR parsers (22/24)



expr → *expr*$_1$ + (*expr*$_2$)
{ *expr* = *expr*$_1$ + *expr*$_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

Υλοποίηση σε LR parsers (23/24)

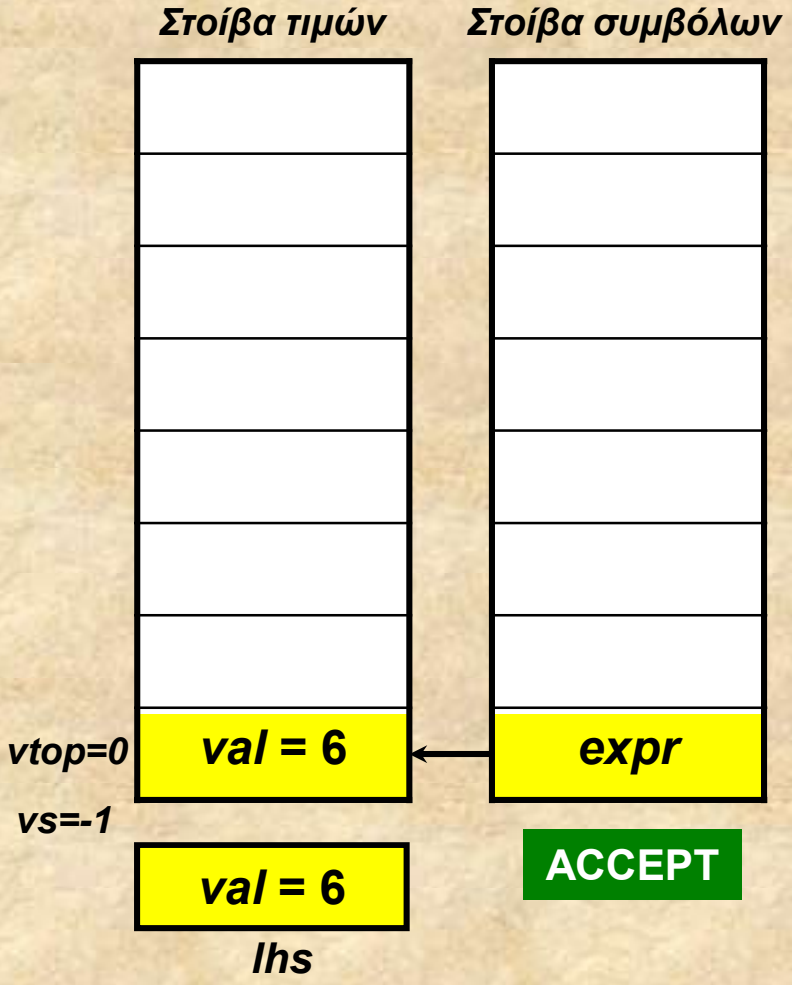


Το RHS έχει 5 σύμβολα

expr → $expr_1$ + ($expr_2$)
{ *expr* = $expr_1$ + $expr_2$; }
expr → **num**
{ *expr* = **num**; }
union VT { *integer val*; };
expr, num: val;

***vs* = *vtop*-|RHS| => *vs* = 4–5 => *vs* = -1**
{ *lhs.val* = *S*[*vs*+1].*val* + *S*[*vs*+4].*val*; }

Υλοποίηση σε LR parsers (24/24)



$expr \rightarrow expr_1 + (expr_2)$
$\{ expr = expr_1 + expr_2; \}$
$expr \rightarrow$ **num**
$\{ expr =$ **num**$; \}$
union VT { integer val; };
expr, num: val;

Περιεχόμενα

- Η αναγκαιότητα
- Γραμματικές γνωρισμάτων
- Υλοποίηση σε LR parsers
- *Υλοποίηση σε LL parsers*
- Προγραμματιστική συμβουλή

Υλοποίηση σε LL parsers (1/24)

- Δεν είναι προφανής η ενσωμάτωση σε predictive top-down parser με πίνακα συντακτικής ανάλυσης
  - καθώς το δέντρο συντακτικής ανάλυσης κατασκευάζεται καθοδικά
  - ενώ η συντακτικά οδηγούμενη μετάφραση υπολογίζεται ανοδικά
  - το πρακτικό πρόβλημα είναι ότι όταν ο καθοδικός αναλυτής δοκιμάζει μία παραγωγή, δεν έχουν υπολογιστεί τα γνωρίσματα των συμβόλων στο RHS

    - Αλλιώς περιέχει μία γραμματική παραγωγή του X, έστω $X \rightarrow UVW$, δηλ. $M[X,a] = \{X \rightarrow UVW\}$. Αυτό ο αναλυτής το αντιμετωπίζει κάνοντας pop το X και push με τη σειρά W, V και U (δηλ. από δεξιά προς τα αριστερά). Σαν έξοδο μπορεί να βγάλει μήνυμα της

Διάλεξη 5, διαφάνεια 45

- Μία λύση θα μπορούσε να ήταν η κατασκευή του συντακτικού δέντρου και έπειτα η εκτέλεση των σημασιολογικών κανόνων
  - είναι πάρα πολύ αργό και δεν εφαρμόζεται ποτέ στην πράξη

Υλοποίηση σε LL parsers (2/24)

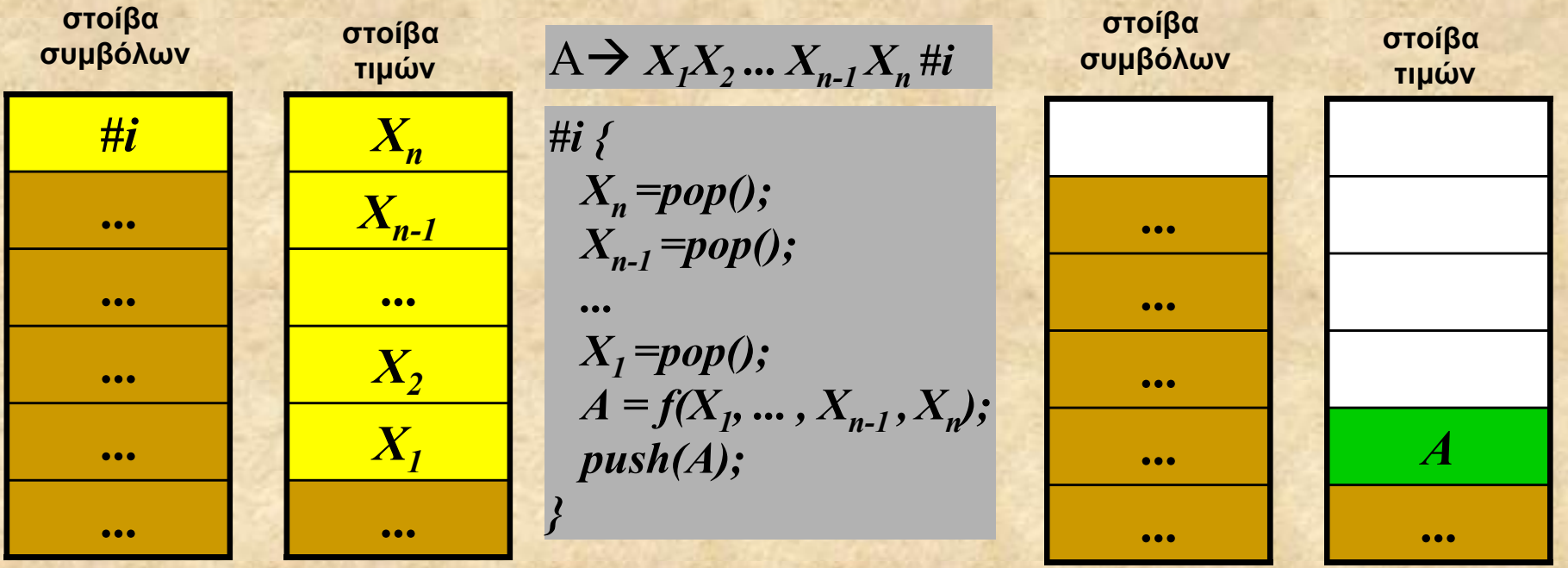
- Η λύση είναι η επέκταση της στοίβας συμβόλων ώστε να περιέχει και σημασιολογικούς κανόνες / actions
  - Θεωρώντας τους ως γραμματικά σύμβολα της παραγωγής **πάντα στο τέλος του RHS**. Θα εκτελούνται μόνο όταν βρίσκονται στην κορυφή της στοίβας συμβόλων, ενώ έπειτα γίνονται popped
- Ταυτόχρονα προσθέτουμε και στοίβα τιμών, που περιέχει τις τιμές των γνωρισμάτων για τα γραμματικά σύμβολα
  - Ένας σημασιολογικός κανόνας παραγωγής κάνει τα εξής:
    - *pop* τις τιμές των γραμματικών συμβόλων του RHS
    - Υπολογισμός και *push* της τιμής του LHS μη τερματικού συμβόλου
- Λόγω του αλγόριθμου καθοδικής ανάλυσης
  - σε μία δοκιμαζόμενη παραγωγή **πρώτα γίνεται *push*** ο σημασιολογικός κανόνας και έπειτα τα σύμβολα στο RHS
    - A→**XYZ *{ κώδικας }*** σημαίνει *push* ***{ κώδικας } ZYX***
  - ενώ η στοίβα πάνω από έναν σημασιολογικό κανόνα έχει γίνει popped (δηλ. προς εκτέλεση) μόνο όταν έχουν γίνει matched όλα τα σύμβολα του RHS του αντίστοιχου κανόνα

Υλοποίηση σε LL parsers (3/24)

Έστω μία παραγωγή $A \rightarrow X_1 \ldots X_n \ \#i$, με $\#i$ να συμβολίζει τον σημασιολογικό κανόνα (action) αυτής της παραγωγής, τον οποίο θεωρούμε ως ένα γραμματικό σύμβολο στο *τέλος* της παραγωγής. Τότε θα φροντίσουμε να υλοποιήσουμε με τέτοιο τρόπο τη διαχείριση της στοίβας τιμών και της εκτέλεσης των σημασιολογικών κανόνων ώστε:

- Όταν το $\#i$ είναι στην κορυφή της στοίβας, δηλ. πρόκειται να εκτελεστεί, οι τιμές των $X_1 \ldots X_n$ βρίσκονται στη στοίβα τιμών ως εξής: X_n στην κορυφή, έπειτα το X_{n-1} και τελευταίο το X_1 (από κάτω του μπορούν να υπάρχουν και άλλες τιμές).
- Η εκτέλεση του $\#i$ θα υπολογίσει την τιμή του A βάσει των τιμών $X_1 \ldots X_n$ της στοίβας, κάνοντας pop όλα τα X και push την τιμή του A.

Υλοποίηση σε LL parsers (4/24)



Υλοποιούμε συντακτικά οδηγούμενη μετάφραση με συντιθέμενα γνωρίσματα, οι τιμές των οποίων είναι διαθέσιμες όταν έχουν γίνει επιτυχώς matched τα αντίστοιχα γραμματικά σύμβολα της παραγωγής. Ο σημασιολογικός κανόνας κανόνας είναι το τελευταίο γραμματικό σύμβολο το οποίο γίνεται πάντα matched, εκτελώντας τον κώδικά του.

Υλοποίηση σε LL parsers (5/24)

Παράδειγμα. Μετάφραση με μέτρηση παρενθέσεων σε γραμματική ισορροπημένων παρενθέσεων και αγκυλών. Θα δούμε τα δύο βασικά βήματα μετατροπής της γραμματικής και των μεταφραστικών κανόνων (translation rules) σε μεταφραστικές ενέργειες του καθοδικού αναλυτή και ενσωμάτωσης τους στη γραμματική. Παρακάτω ακολουθεί η αρχική γραμματική με τους μεταφραστικούς κανόνες.

$E \rightarrow \varepsilon$	$\{ E.total = 0; \}$
$E \rightarrow (E_1)$	$\{ E.total = E_1.total + 1; \}$
$E \rightarrow [E_1]$	$\{ E.total = E_1.total; \}$

Υλοποίηση σε LL parsers (6/24)

ΒΗΜΑ Ι. Αντικατάσταση των μεταφραστικών κανόνων με ενέργειες στον καθοδικό αναλυτή, όπως έχει οριστεί πριν. Δηλ. (α) *pop* τις τιμές των συμβόλων στο RHS, (β) υπολογισμός και *push* της τιμής του συμβόλου του LHS.

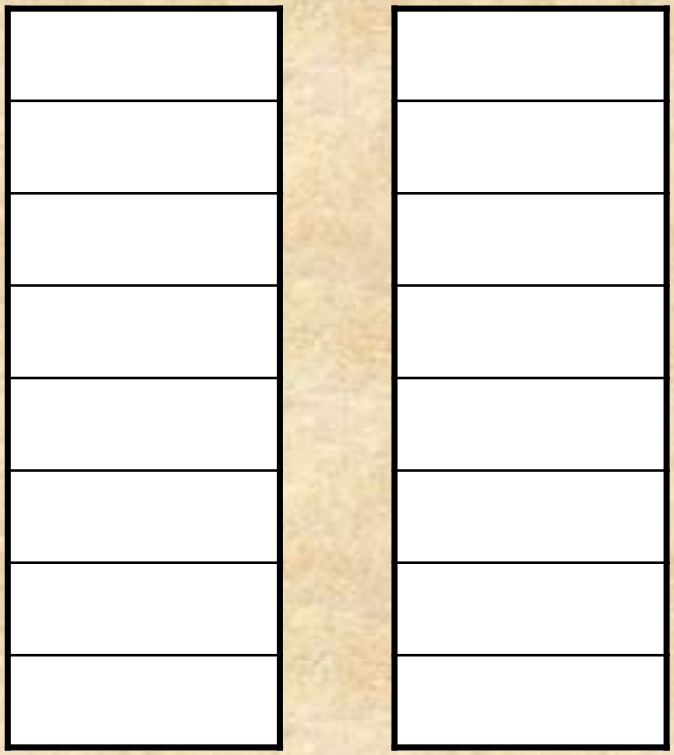
$E \rightarrow \varepsilon$	{ *push(*0); }
$E \rightarrow (E_1)$	{ n = *pop*(); *push*(n+1); }
$E \rightarrow [E_1]$	{ n = *pop*(); *push*(n); }

ΒΗΜΑ ΙΙ. Αντιστοιχούμε κάθε μεταφραστικό κανόνα σε ένα μοναδικά αριθμημένο γραμματικό σύμβολο, το οποίο γίνεται τμήματα του γραμματικού κανόνα.

$E \rightarrow$	ε #1
$E \rightarrow$	(E_1) #2
$E \rightarrow$	[E_1] #3

#1	{ *push(*0); }
#2	{ n = *pop*(); *push*(n+1); }
#3	{ n = *pop*(); *push*(n); }

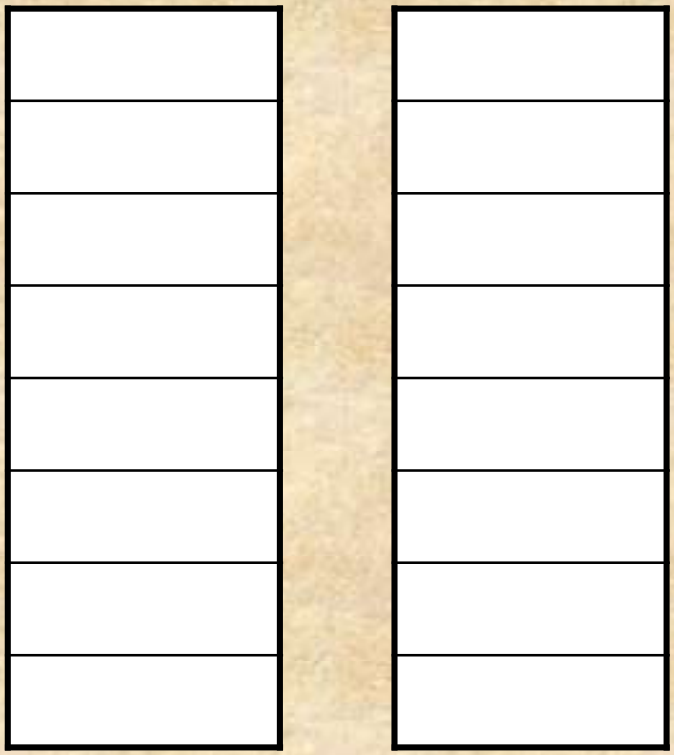
Υλοποίηση σε LL parsers (7/24)



$E \rightarrow \quad \varepsilon$ #1
$E \rightarrow \quad (\ E_1\ )$ #2
$E \rightarrow \quad [\ E_1\ ]$ #3
#1 { $push(0)$; }
#2 { $n = pop()$; $push(n+1)$; }
#3 { $n = pop()$; $push(n)$; }

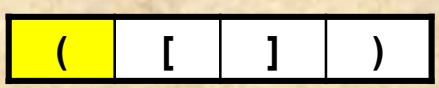
(	[	]	)

Υλοποίηση σε LL parsers (8/24)

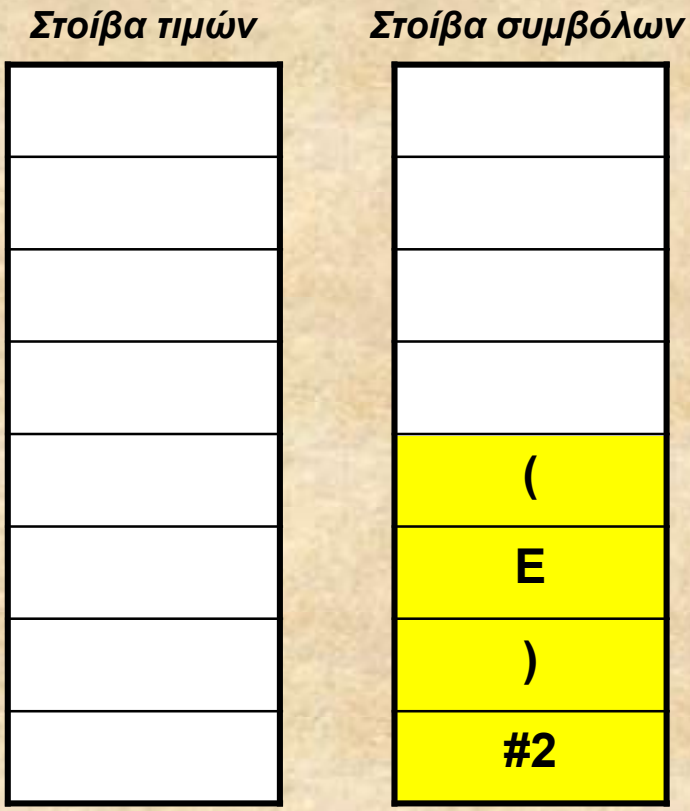


$E \rightarrow \quad \varepsilon$ #1
$E \rightarrow \quad (E_1)$ #2
$E \rightarrow \quad [E_1]$ #3
#1 { $push(0)$; }
#2 { $n = pop()$; $push(n+1)$; }
#3 { $n = pop()$; $push(n)$; }

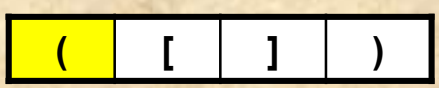
PREDICT



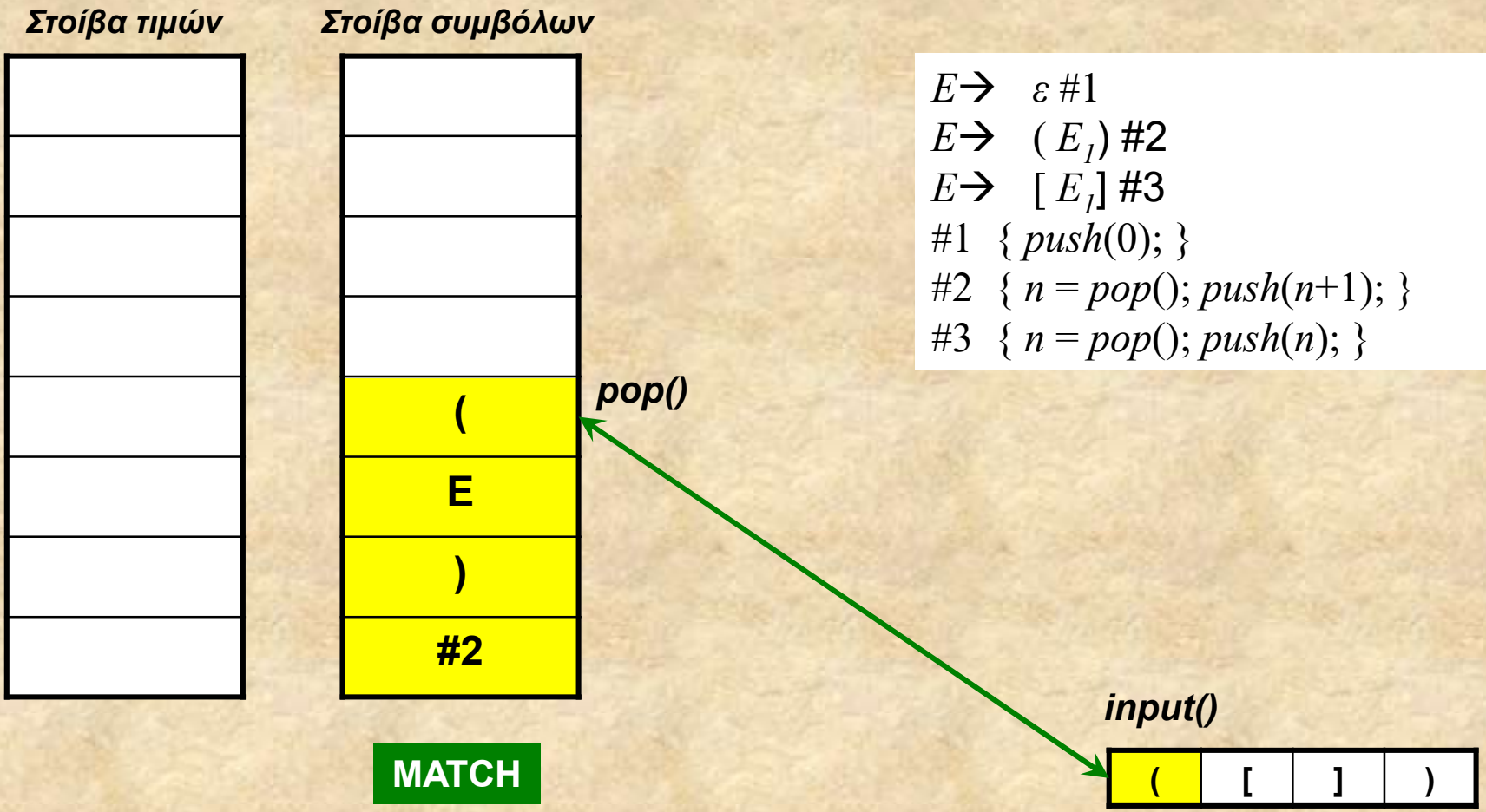
Υλοποίηση σε LL parsers (9/24)



$E \rightarrow \quad \varepsilon$ #1
$E \rightarrow \quad (E_1)$ #2
$E \rightarrow \quad [E_1]$ #3
#1 { $push(0)$; }
#2 { $n = pop()$; $push(n+1)$; }
#3 { $n = pop()$; $push(n)$; }

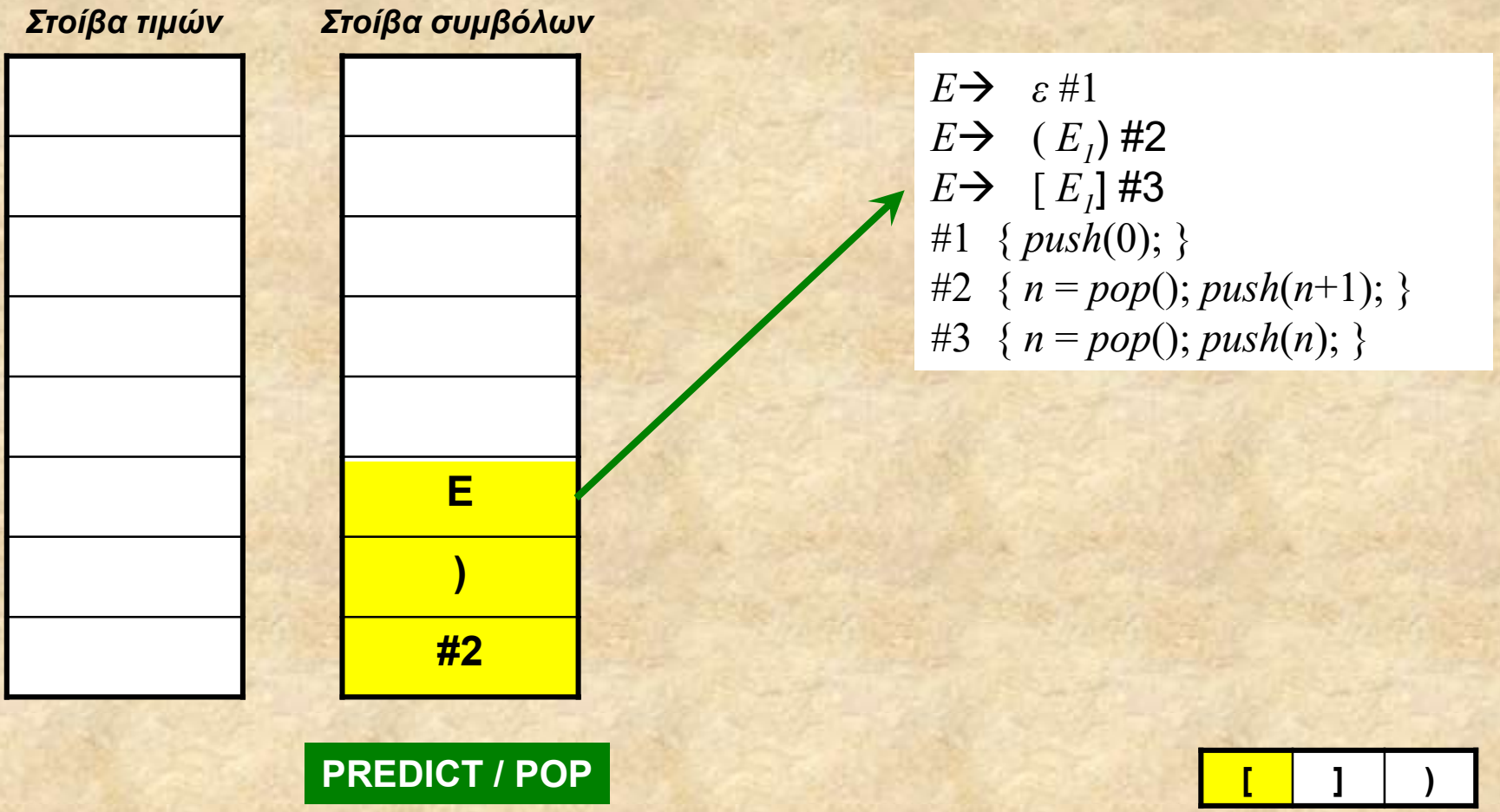


Υλοποίηση σε LL parsers (10/24)

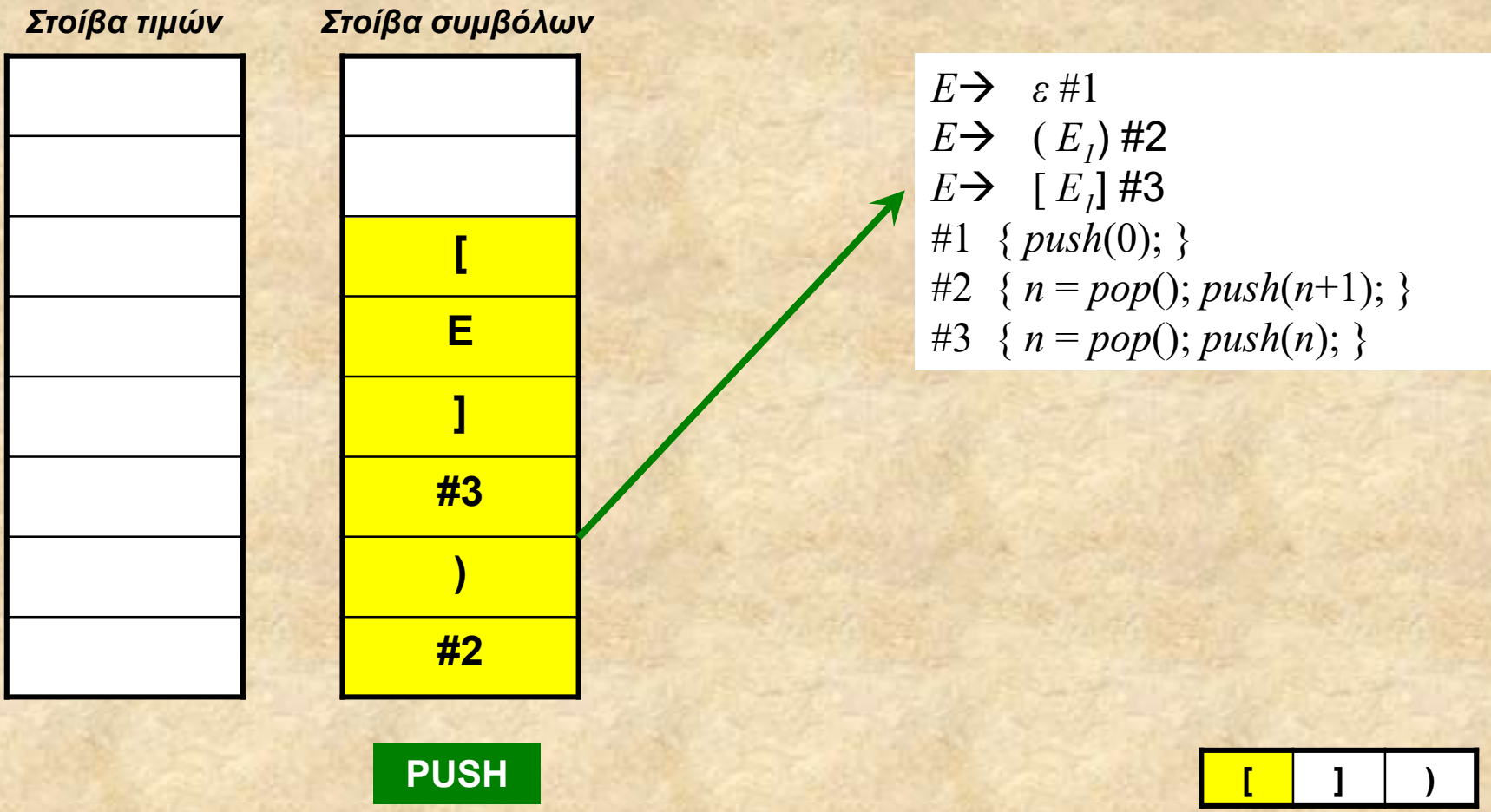


$E \rightarrow \quad \varepsilon$ #1
$E \rightarrow \quad (E_1)$ #2
$E \rightarrow \quad [E_1]$ #3
#1 { $push(0)$; }
#2 { $n = pop()$; $push(n+1)$; }
#3 { $n = pop()$; $push(n)$; }

Υλοποίηση σε LL parsers (11/24)

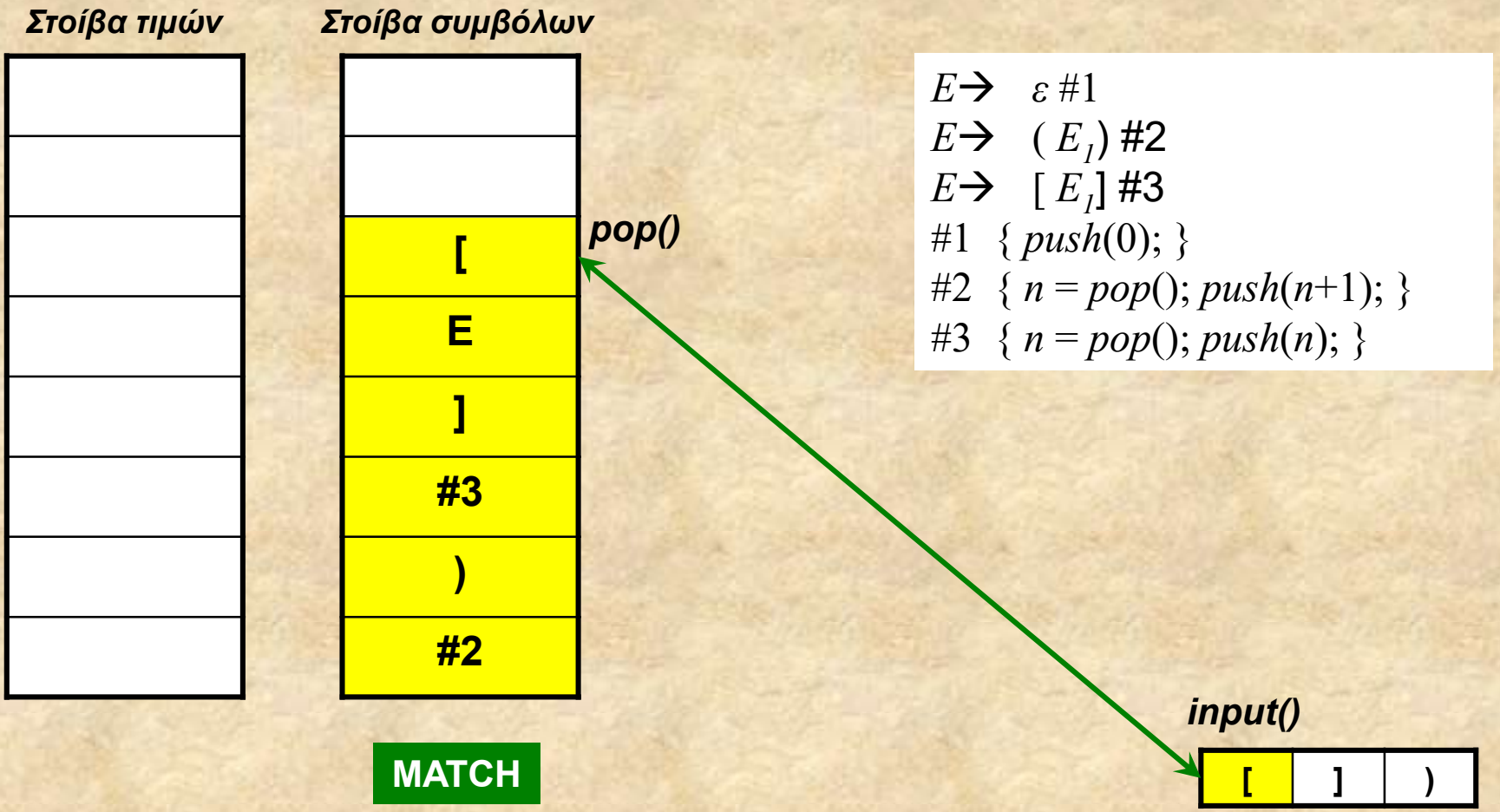


Υλοποίηση σε LL parsers (12/24)



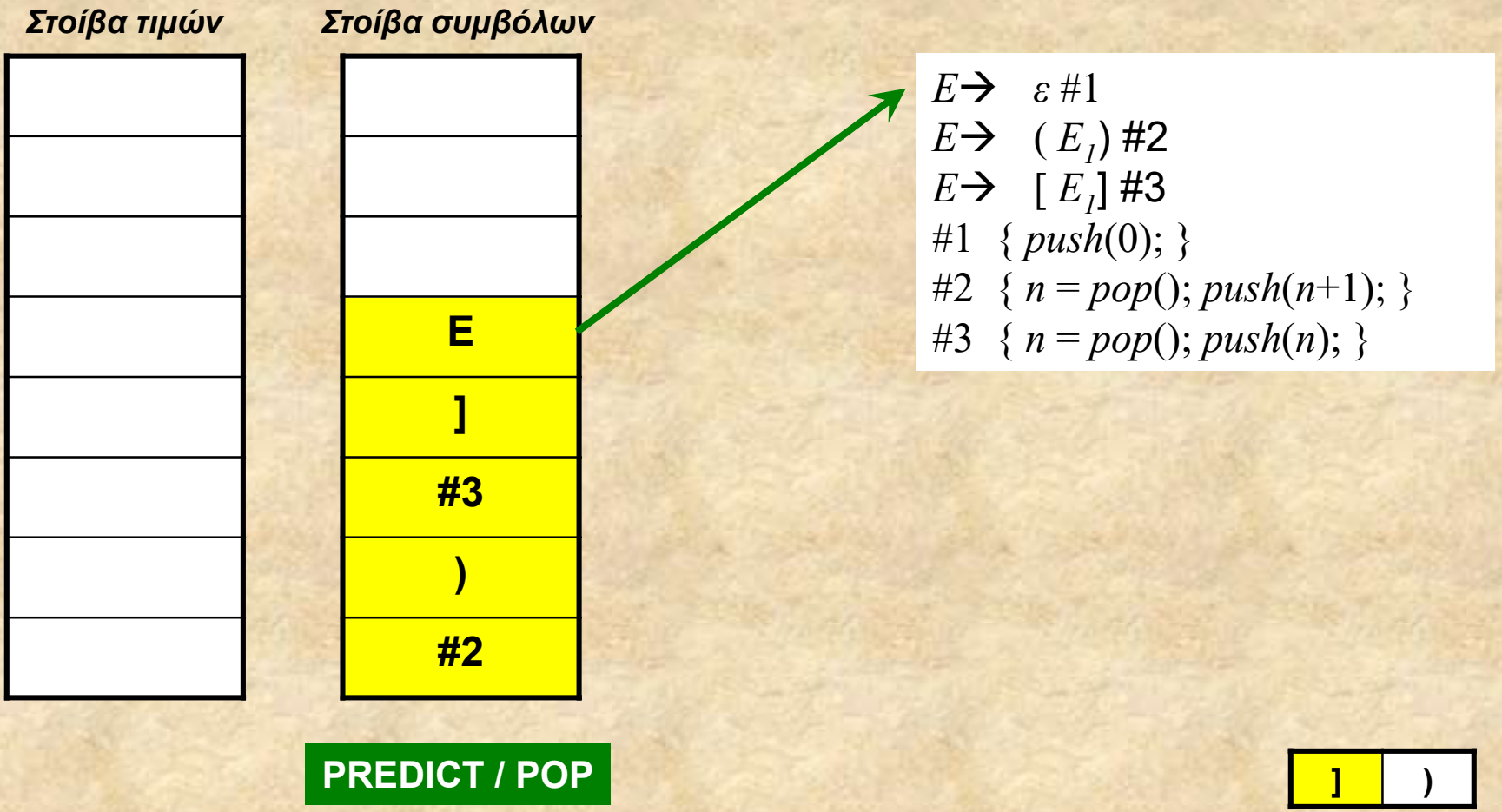
$E \rightarrow \quad \varepsilon$ #1
$E \rightarrow \quad (E_1)$ #2
$E \rightarrow \quad [E_1]$ #3
#1 { $push(0)$; }
#2 { $n = pop()$; $push(n+1)$; }
#3 { $n = pop()$; $push(n)$; }

Υλοποίηση σε LL parsers (13/24)

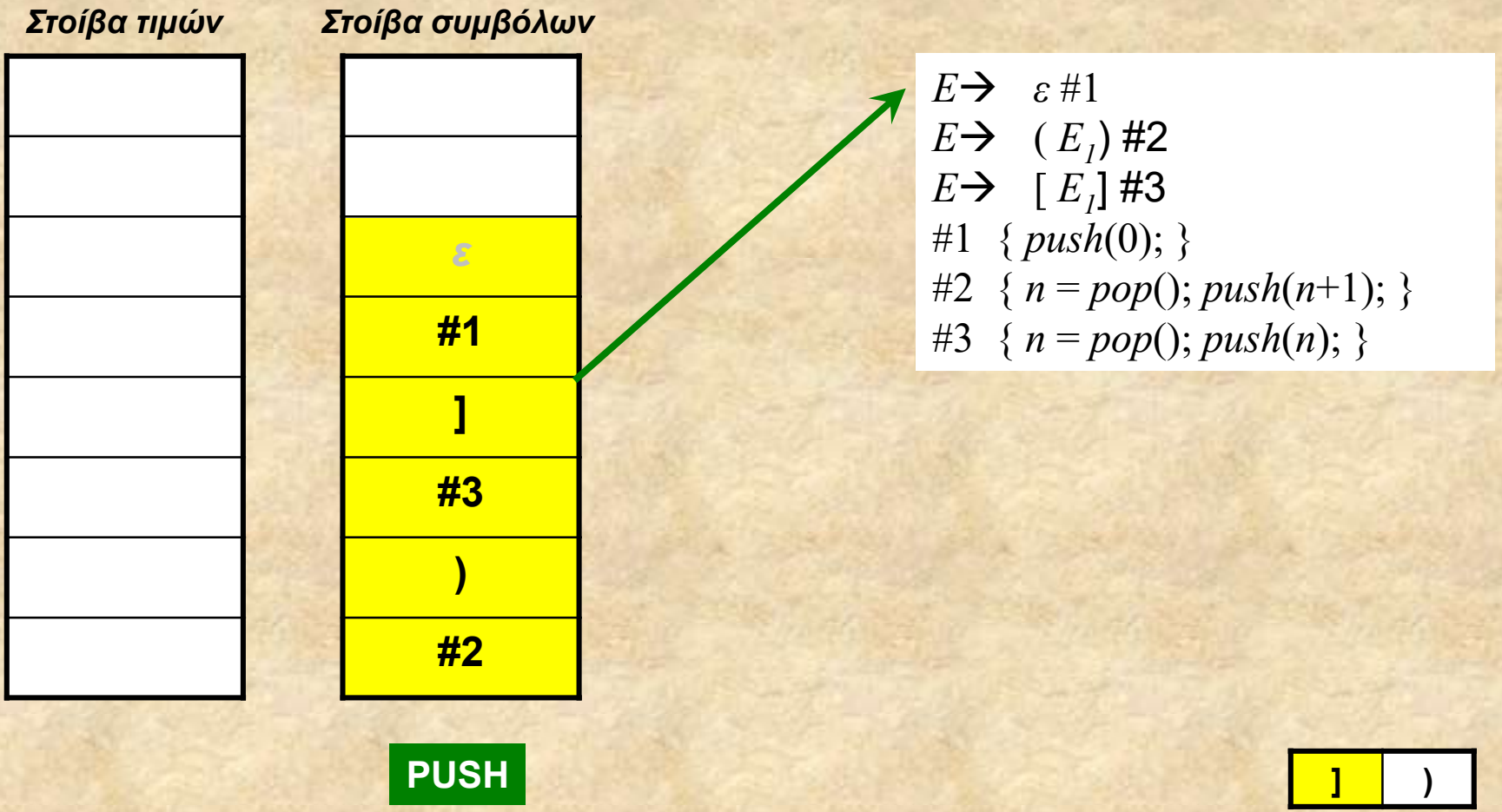


$E \rightarrow \quad \varepsilon$ #1
$E \rightarrow \quad (E_1)$ #2
$E \rightarrow \quad [E_1]$ #3
#1 { $push(0)$; }
#2 { $n = pop()$; $push(n+1)$; }
#3 { $n = pop()$; $push(n)$; }

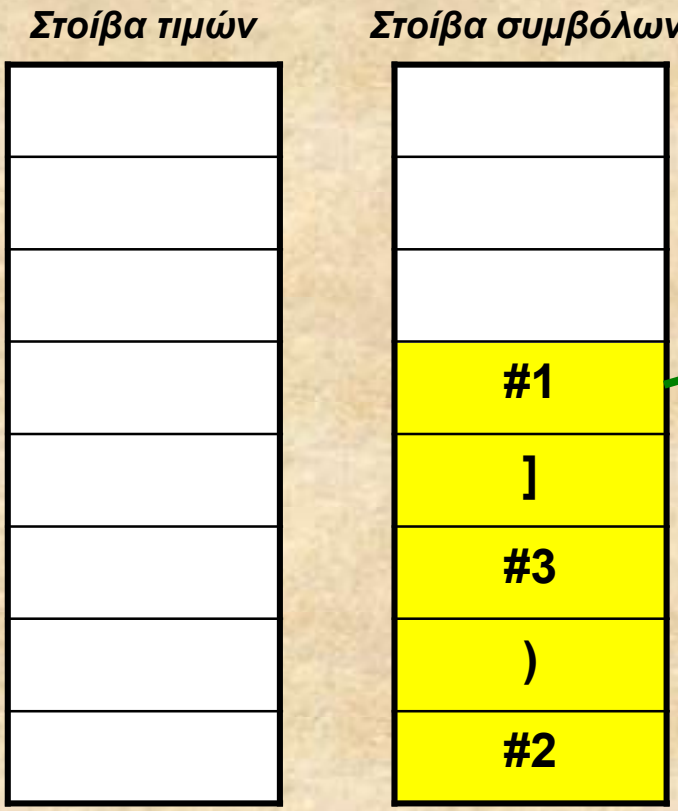
Υλοποίηση σε LL parsers (14/24)



Υλοποίηση σε LL parsers (15/24)



Υλοποίηση σε LL parsers (16/24)

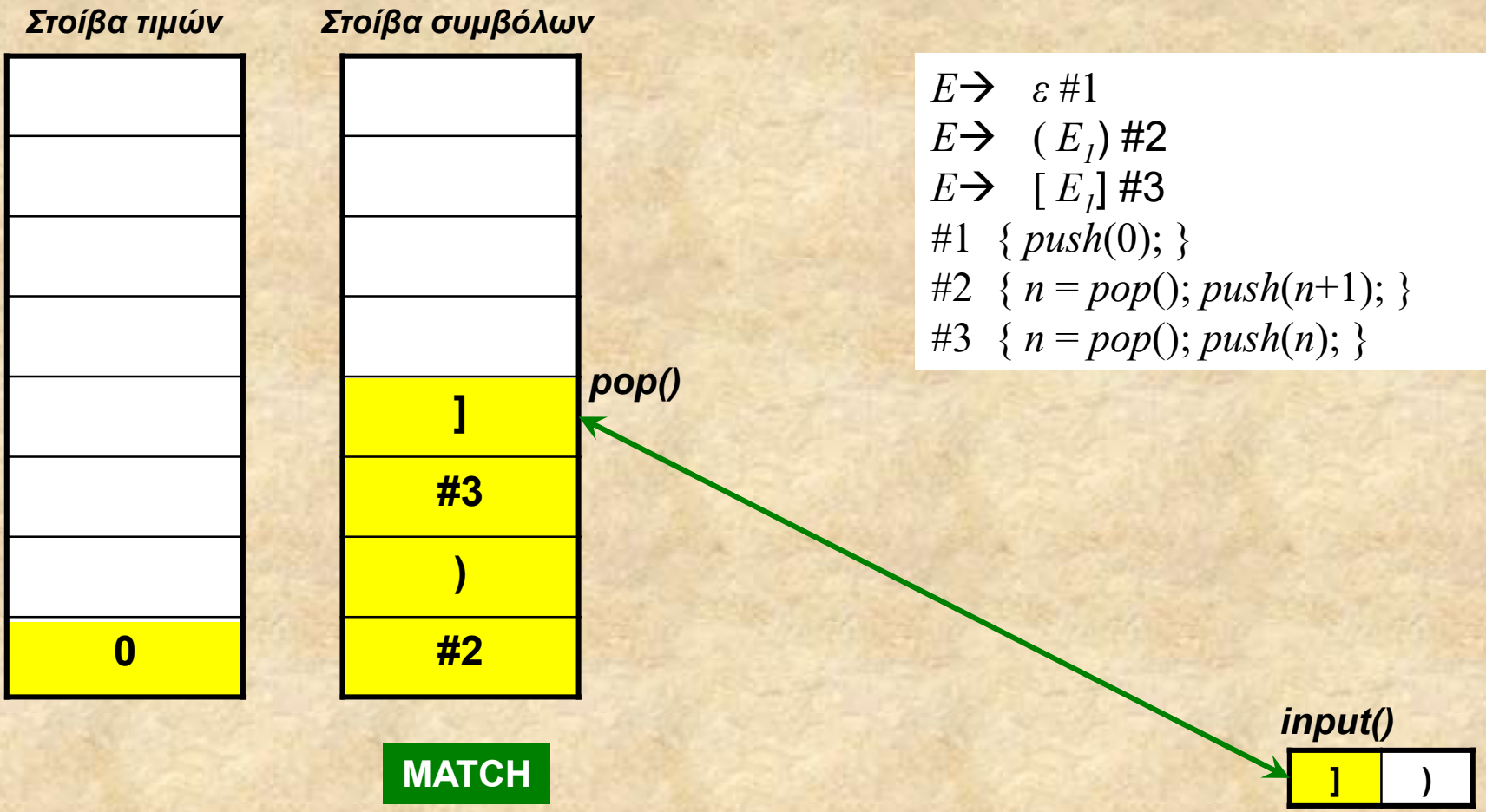


$E \rightarrow \quad \varepsilon$ #1
$E \rightarrow \quad (E_1)$ #2
$E \rightarrow \quad [E_1]$ #3
#1 { $push(0)$; }
#2 { $n = pop()$; $push(n+1)$; }
#3 { $n = pop()$; $push(n)$; }

EXECUTE, POP

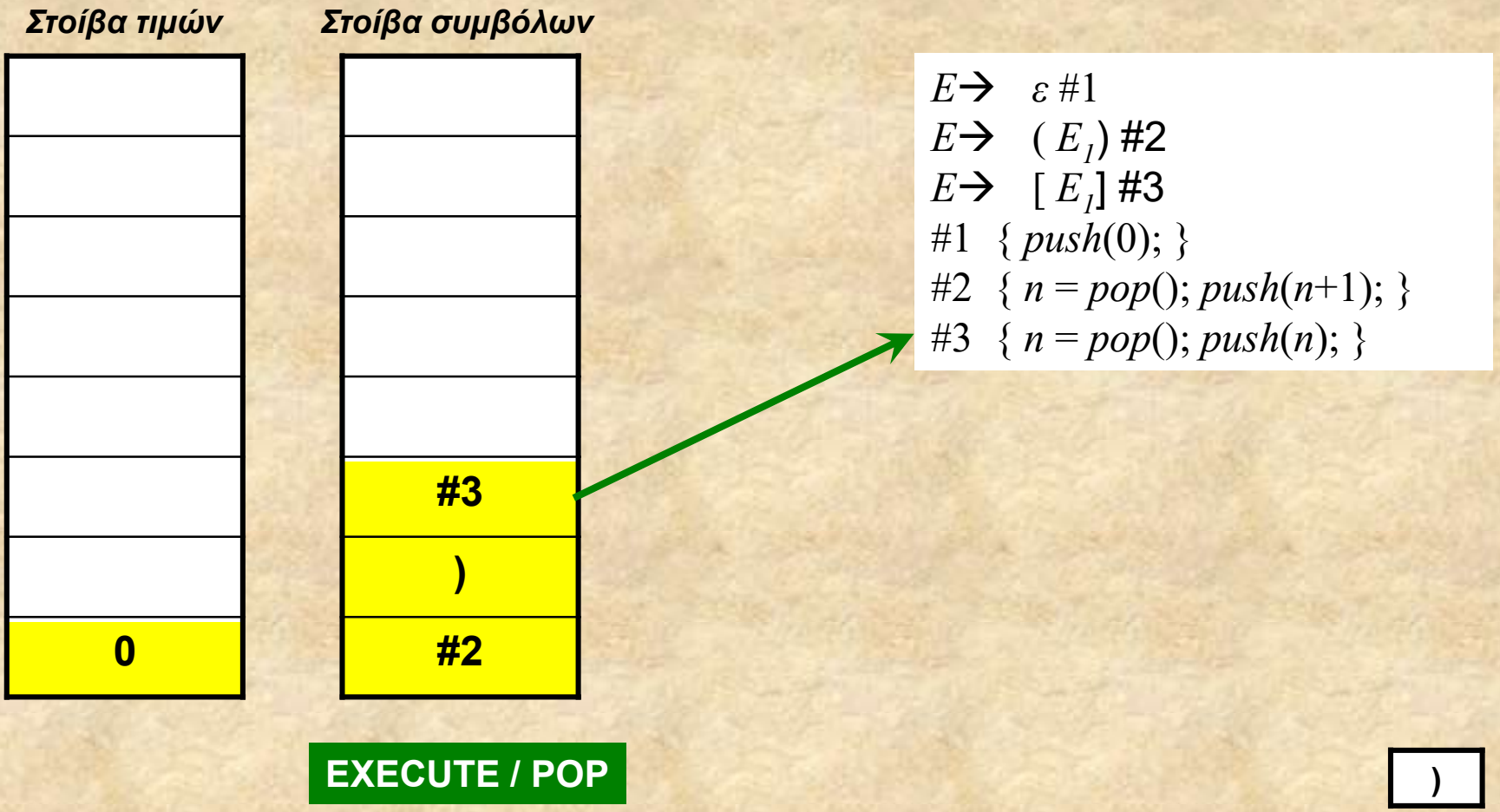
]	)

Υλοποίηση σε LL parsers (17/24)

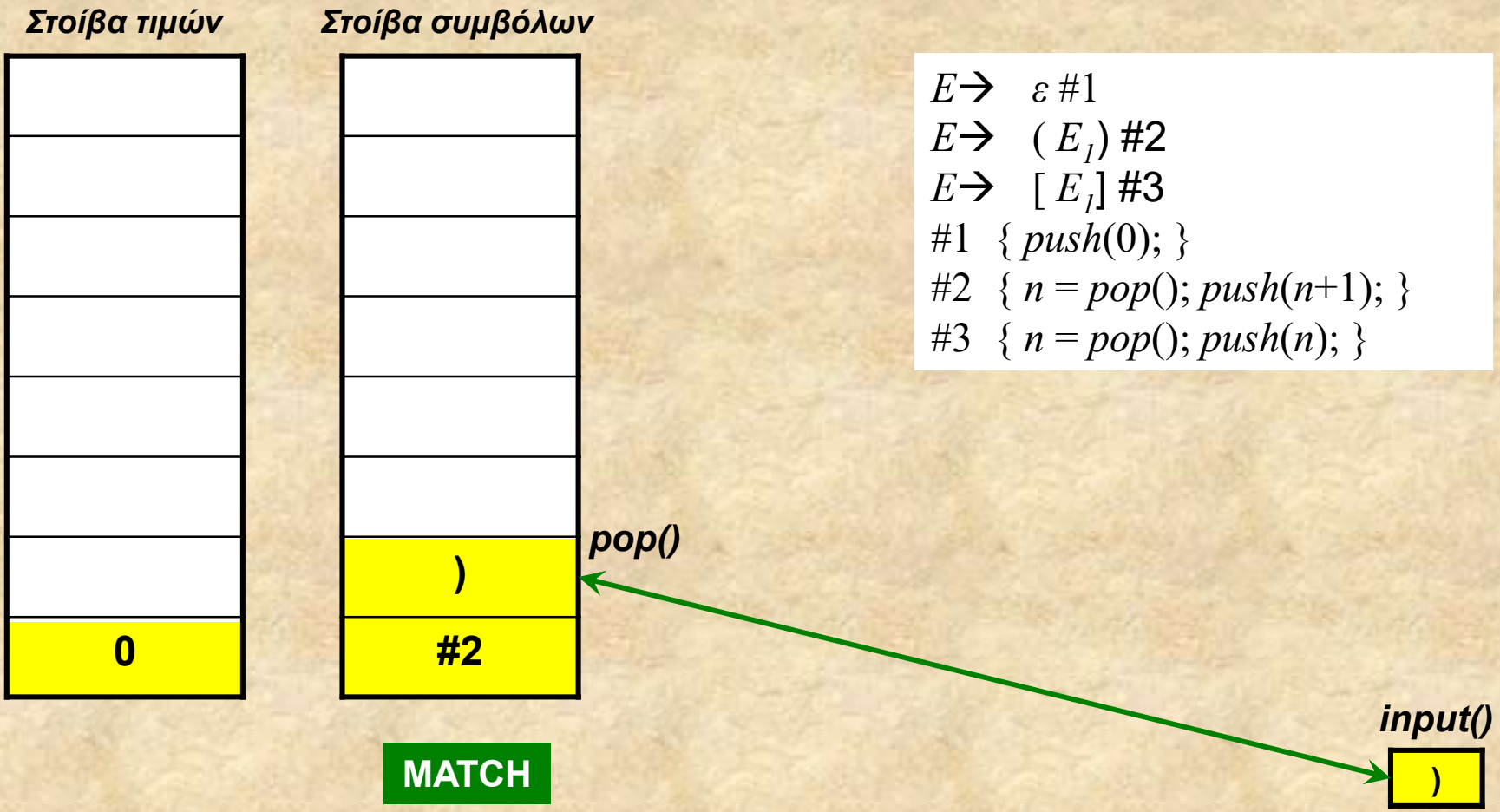


$E \rightarrow \quad \varepsilon$ #1
$E \rightarrow \quad (E_1)$ #2
$E \rightarrow \quad [E_1]$ #3
#1 { $push(0)$; }
#2 { $n = pop()$; $push(n+1)$; }
#3 { $n = pop()$; $push(n)$; }

Υλοποίηση σε LL parsers (18/24)

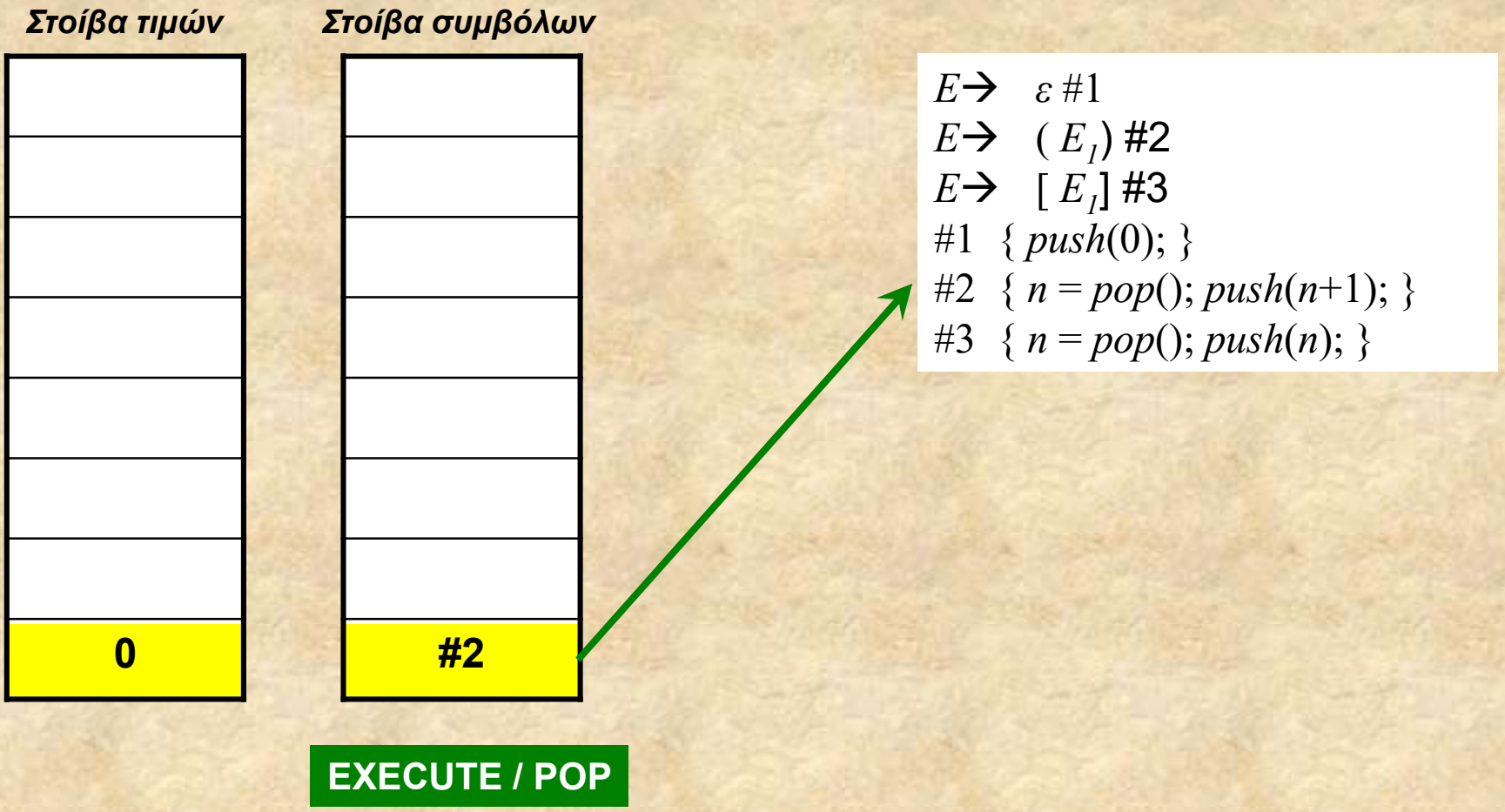


Υλοποίηση σε LL parsers (19/24)



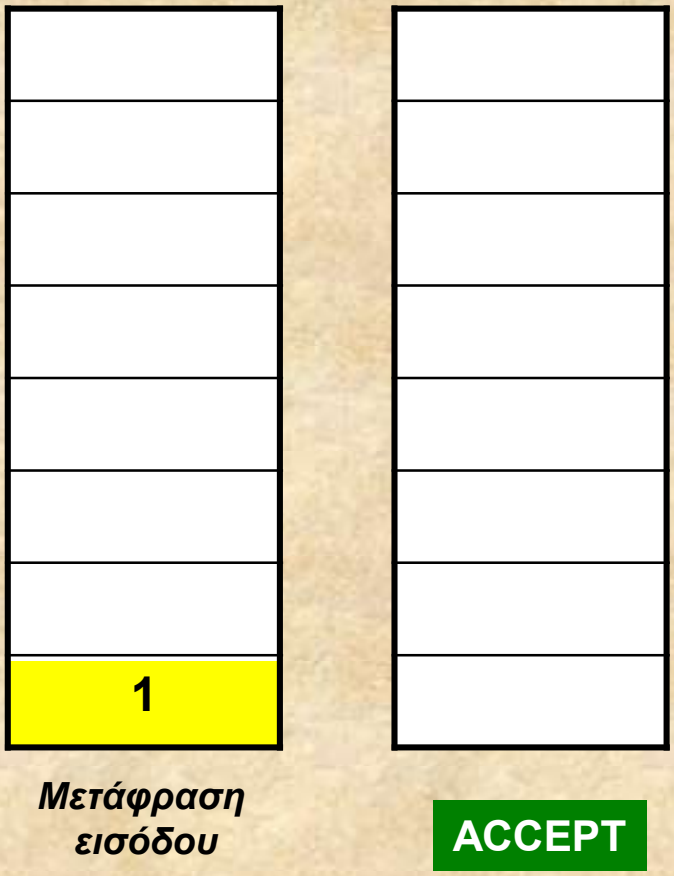
$E \rightarrow \quad \varepsilon$ #1
$E \rightarrow \quad (E_1)$ #2
$E \rightarrow \quad [E_1]$ #3
#1 { *push*(0); }
#2 { n = *pop*(); *push*(n+1); }
#3 { n = *pop*(); *push*(n); }

Υλοποίηση σε LL parsers (20/24)



$E \rightarrow \quad \varepsilon$ #1
$E \rightarrow \quad (E_1)$ #2
$E \rightarrow \quad [E_1]$ #3
#1 { $push(0)$; }
#2 { $n = pop()$; $push(n+1)$; }
#3 { $n = pop()$; $push(n)$; }

Υλοποίηση σε LL parsers (21/24)



$E \rightarrow \quad \varepsilon$ #1
$E \rightarrow \quad (E_1)$ #2
$E \rightarrow \quad [E_1]$ #3
#1 { *push*(0); }
#2 { n = *pop*(); *push*(n+1); }
#3 { n = *pop*(); *push*(n); }

Υλοποίηση σε LL parsers (22/24)

Αφήσαμε μία λεπτομέρεια που πρέπει τώρα να αντιμετωπιστεί. Ο κανόνας που δώσαμε αναφέρεται στον υπολογισμό των τιμών μόνο για μη τερματικά σύμβολα. Τι γίνεται στην περίπτωση των τερματικών συμβόλων; πως γίνεται push η τιμή του γνωρίσματός τους στη στοίβα τιμών; Η απάντηση βρίσκεται στον τρόπο λειτουργίας του καθοδικού αναλυτή:

- Εάν στην κορυφή της στοίβας συμβόλων είναι το X και το επόμενο σύμβολο εισόδου είναι το α, με $X=\alpha$, τότε κάνε *pop* το X και *input*(). Η ενέργεια αυτή είναι γνωστή ως *match*.
- Το παραπάνω σημαίνει ότι μετά το match, η τιμή του τερματικού έχει χαθεί καθώς η είσοδος προχωράει στο επόμενο σύμβολο. Άρα, η μόνη λύση για να κρατήσουμε την τιμή του κάθε matched τερματικού συμβόλου είναι να έχουμε πριν από κάθε match, ένα push της τιμής του εκάστοτε συμβόλου εισόδου στη στοίβα τιμών.
- Αυτό σημαίνει ότι στους γραμματικούς κανόνες όπου θέλουμε να χρησιμοποιήσουμε τις τιμές κάποιων τερματικών συμβόλων στους σημασιολογικούς κανόνες θα πρέπει πριν το τερματικό σύμβολο να προσθέσουμε έναν κανόνα της μορφής «*push(current token)*»

Υλοποίηση σε LL parsers (23/24)

Γραμματική	Μετάφραση	Καθοδική μετάφραση
prim→**num** *prim*→(*expr*)	*prim*→**num** { *prim*.val = num.tval;} *prim*→(*expr*) { *prim.val* = *expr.val*; }	*prim*→#1 **num** *prim*→(*expr*) #2 #1 { *push*(*lookAhead*); } #2 { *e* = *pop*(); *push*(*e*); } ή {}

Καθώς πάντα υπάρχει το πρόβλημα ότι μία γραμματική πρέπει να μετατραπεί σε *LL(1)*, π.χ. απαιτείται εξάλειψη αριστερής αναδρομής, η εξαγόμενη γραμματική δεν είναι πρόσφορη για μετάφραση. Π.χ. ο αριστερά αναδρομικός κανόνας:

$$E \rightarrow E + T \text{ γίνεται } E \rightarrow T\,E' \text{ και } E \rightarrow \varepsilon \mid + T\,E'$$

Η ενσωμάτωση κανόνων μετάφρασης στους δύο κανόνες που προκύπτουν δεν είναι καθόλου προφανής. Η λύση είναι η εξής:

- Αντιμετωπίζουμε τους μεταφραστικούς κανόνες ως γραμματικά σύμβολα, ενώ η ενσωμάτωσή τους εφαρμόζεται στην αρχική γραμματική.
- Έπειτα μετατρέπουμε τη γραμματική σε LL(1) θεωρώντας τους κανόνες ως γραμματικά σύμβολα.
- Δεν μας απασχολεί σε ποιες παραγωγές και σε ποια θέση θα μεταφερθούν τελικά οι μεταφραστικοί κανόνες.

Υλοποίηση σε LL parsers (24/24)

Η αρχική γραμματική

$expr \rightarrow expr + term$
$expr \rightarrow term$
$term \rightarrow term * prim$
$term \rightarrow prim$

Η γραμματική με μεταφραστικούς κανόνες καθοδικής ανάλυσης ως γραμματικά σύμβολα

$expr \rightarrow expr + term\ \#1$
$expr \rightarrow term\ \#2$
$term \rightarrow term * prim\ \#3$
$term \rightarrow prim\ \#4$

Υλοποίηση των μεταφραστικών κανόνων

#1 { t=pop; e=pop(); push(e+t); }
#2 { t = pop(); push(t); } ή {}
#3 { p=pop; t=pop(); push(t+p); }
#4 { p = pop(); push(p); } ή {}

Η τροποποιημένη γραμματική με εξάλειψη αριστερής αναδρομής με τα ειδικά γραμματικά σύμβολα για τους μεταφραστικούς κανόνες

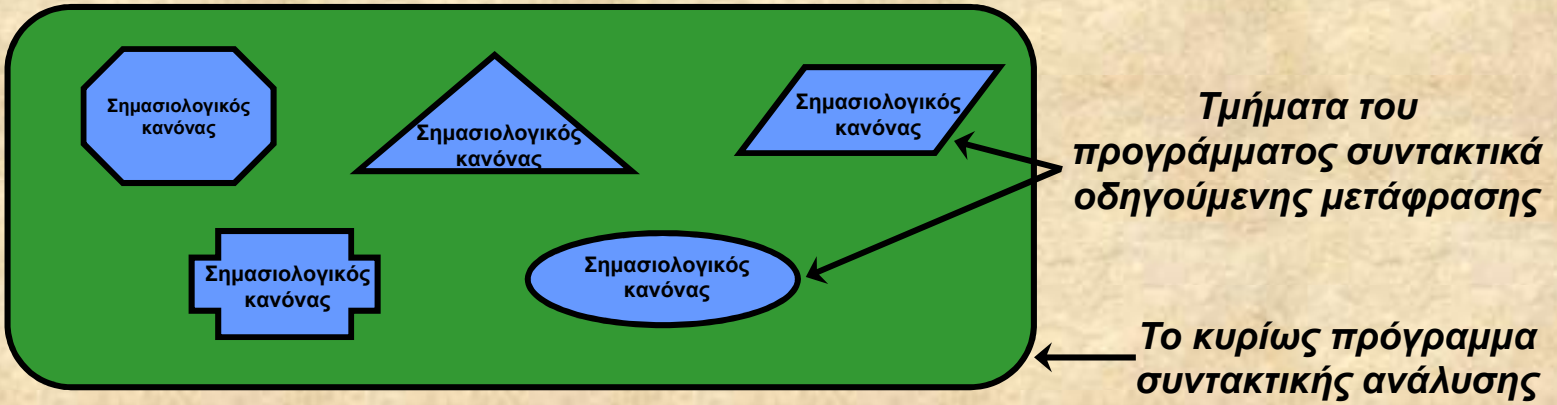
$expr \rightarrow term\ \#2\ expr'$
$expr \rightarrow \varepsilon$
$expr' \rightarrow + term\ \#1\ expr'$
$term \rightarrow prim\ \#4\ term'$
$term' \rightarrow * prim\ \#3\ term'$
$term' \rightarrow \varepsilon$

Περιεχόμενα

- Η αναγκαιότητα
- Γραμματικές γνωρισμάτων
- Υλοποίηση σε LR parsers
- Υλοποίηση σε LL parsers
- *Προγραμματιστική συμβουλή*

Προγραμματιστική συμβουλή (1/8)

- Η υλοποίηση μεταγλωττιστή με συντακτικά οδηγούμενη μετάφραση είναι η καλύτερη σχετική κατασκευαστική τεχνική σήμερα
- Αυτή η τεχνική βασίζεται στο συνδυασμό δύο ανεξάρτητων προγραμμάτων, το ένα ωστόσο ενσωματωμένο μέσα στο άλλο
  - το πρόγραμμα συντακτικής ανάλυσης, που είναι και το κυρίως πρόγραμμα – main program
  - το πρόγραμμα συντακτικά οδηγούμενης μετάφρασης που είναι ενσωματωμένο στο πρόγραμμα συντακτικής ανάλυσης – embedded program



Προγραμματιστική συμβουλή (2/8)

- Ο κώδικας για συντακτικά οδηγούμενη μετάφραση ποτέ δεν συνιστά αυτούσιο ανεξάρτητο πρόγραμμα, σε αντίθεση με τον συντακτικό αναλυτή
- Το γεγονός αυτό πολλές φορές οδηγεί σε κατασκευαστικές συνήθειες που πλήττουν την κατασκευαστική και οργανωτική ποιότητα των σημασιολογικών κανόνων
- Θα παρουσιαστεί μία τακτική η οποία μπορεί να εφαρμοστεί σε όλες τις περιπτώσεις συντακτικά οδηγούμενης μετάφρασης (LL ή LR) με πολλά πλεονεκτήματα
- Η τακτική αυτή έχει εφαρμοστεί επιτυχώς σε μεταγλωττιστές μεγάλης κλίμακας για ειδικές γλώσσες τέταρτης γενιάς, όπως τη γλώσσα I-*GET*.
  - http://www.ics.forth.gr/hci/files/plang/iget_language.pdf
  - http://www.ics.forth.gr/hci/files/plang/IGET_WINDOWS.zip

Προγραμματιστική συμβουλή (3/8)

Περιγραφή τακτικής. Έστω ο γραμματικός κανόνας $A \rightarrow X_1 \ldots X_n$ με έναν αντίστοιχο σημασιολογικό κανόνα που αποδίδει είτε κάποια τιμή στο συντιθέμενο γνώρισμα του Α ή / και προκαλεί κάποιο πλάγιο αποτέλεσμα σε μεταβλητές του προγράμματος συντακτικά οδηγούμενης μετάφρασης. Τότε ο σημασιολογικός κανόνας υλοποιείται μέσω μίας μοναδικά ονομαζόμενης συνάρτησης ως εξής:

T_A **Manage_**A**_**$X_1 X_2 \ldots X_{n-1} X_n$ (T_{X1} $p1$, T_{X2} $p2, \ldots, T_{Xn}$ p_n),
με *typeof*(A) = T_A και *typeof*(X_j) = T_{Xj}

Εναλλακτικά το επιστρεφόμενο όρισμα μπορεί να αντικατασταθεί ως πρώτο όρισμα τύπου pointer και η συνάρτηση να γίνει void. Π.χ., έστω οι γραμματικοί κανόνες για το *lvalue* στη γλώσσα *alpha*. Τότε θα οριστούν συγκεκριμένες συναρτήσεις σημασιολογικής ανάλυσης για κάθε κανόνα.

Προγραμματιστική συμβουλή (4/8)

expr Manage_lvalue_id* (char* *p1*);
expr Manage_lvalue_localid* (char* *p1*);
expr Manage_lvalue_globalid* (char* *p1*);
expr Manage_lvalue_tableitem* (expr* *p1*);

lvalue → **id**
{ $*lvalue* = *Manage_lvalue_id*(**id.name**); }
lvalue → **local id**
{ $*lvalue* = *Manage_lvalue_localid*(**id.name**); }
lvalue →**:: id**
{ $*lvalue* = *Manage_lvalue_globalid*(**id.name**); }
lvalue → *tableitem*
{ $*lvalue* = *Manage_lvalue_tableitem*($*tableitem*); }

Προγραμματιστική συμβουλή (5/8)

Τα κατασκευαστικά πλεονεκτήματα είναι πολλά

- Οι αυθεντικοί γραμματικοί κανόνες μένουν «καθαροί» και αναγνώσιμοι.
- Το σύνολο των απαιτούμενων συναρτήσεων συντακτικά οδηγούμενης μετάφρασης (API) μπορεί να οριστεί από πριν και πολύ γρήγορα.
- Η κατανομή φόρτου σε ανεξάρτητους προγραμματιστές είναι έτσι τετριμμένη, ενώ όλοι εργάζονται σε διαφορετικά τμήματα και αρχεία κώδικα (και όχι σε ένα, π.χ, θηριώδες YACC specification file).
- Η επιδιόρθωση λαθών είναι ευκολότερη καθώς το trace μέσα σε σημασιολογικούς κανόνες που είναι ενωμένοι με τον κώδικα του αναλυτή είναι περισσότερο επίπονο και χρονοβόρο.
- Το coding είναι ευκολότερο καθώς δεν χρειάζεται να δουλεύετε σε ένα μεγάλο αρχείο βλέποντας κώδικα και ορισμούς που δεν σας απασχολούν, ενώ οι ειδικοί editors που αναγνωρίζουν C/C++ αρχεία δεν θα «μπερδεύονται» με το mixed syntax (π.χ. YACC+C), προσφέροντας σας όλες τις ευκολίες όπως syntax highlighting, type information, κλπ.
- Εάν χρειαστεί να αλλάζετε τους σημασιολογικές κανόνες δεν είναι αναγκαίο να κάνετε parser re-generation κάθε φορά.
- Η επαναχρησιμοποίηση σημασιολογικών κανόνων για κατασκευή διαφορετικών compilers γίνεται ιδιαίτερα εύκολη.

Προγραμματιστική συμβουλή (6/8)

```
Lvalue:
      ID
        { Manage_Lvalue_ID($1, &($$)); }
    | LEXICAL'(' ID ')' SCOPERES ID DOT ID
        { Manage_Lvalue_LEXICAL_ID($3, $6, $8, &($$)); }
    | VIRTUAL SCOPERES ID DOT ID
        { Manage_Lvalue_VIRTUAL_ID($3, $5, &($$)); }
    | AGENT SCOPERES ID DOT ID
        { Manage_Lvalue_AGENT_ID($3, $5, &($$)); }
    | AgentScopeRes ID
        { Manage_Lvalue_AgentScopeResID($1, $2, &($$)); }
    | Lvalue '[' Expression ']'
        { Manage_Lvalue_Array($1, $3, &($$)); }
    | '*' Lvalue %prec USTAR
        { Manage_Lvalue_Pointer($2, &($$)); }
    | Lvalue DOT ID
        { Manage_Lvalue_Dot_ID($1,$3, &($$)); }
    | Lvalue ARROW ID
        { Manage_Lvalue_Arrow_ID($1, $3, &($$)); }
        | '(' Lvalue ')'
        { Manage_Lvalue_Parenthesis($2,&($$)); }
    | ObjRef DOT ID
        { Manage_Lvalue_ObjRef_ID($1,$3, &($$)); }
        ;
```

```
EventBlocks:
      EventBlock
        { Manage_EventBlocks($1, &($$)); }
    | EventBlocks EventBlock
        { Manage_EventBlocks($1, $2, &($$)); }
        ;

EventBlock_EventClass: ID
        { Manage_EventBlock_EventClass($1, &($$)); }
        ;

EventBlock:
        EventBlockPrefix EventBlock_EventClass Compound
        { Manage_EventBlock($1, $2, $3, &($$)); }
        ;

EventBlockPrefix:
      VariableDeclarations '(' Expression ')'
        { Manage_EventBlockPrefix($1, $3, &($$)); }
    | '(' Expression ')'
        { Manage_EventBlockPrefix((SymbolList*) GNULL, $2, &($$)); }
        ;
```

```
VirtualClassPrefix:
        VIRTUAL ID '(' IdListAlsoEmpty ')' '[' IncScope
        { Manage_VirtualClassPrefix($2, $4, &($$)); }
        ;
VirtualClass:
        VirtualClassPrefix
        InGenesisSpecs Compound OutConstructor
        Destructor Compound ']' OutDestructor
        DecScope
        {
            Manage_VirtualClass($1,$2,$3,$6);
            IN_VIRTUAL=0; POP_CURR_CLASS(); OUT_CLASS_SCOPE();
            $$=$1;
        }
        ;
```

```
Constraint: Lvalue EQUALS Expression ';'
        { Manage_Constraint($1, $3, &($$)); }
        ;

Monitor: LvalueList ':' Compound
        { Manage_Monitor($1, $3, &($$)); }
        ;

Method:  METHOD ObjRef DOT ID Compound
        { Manage_Method($2,$4,$5, &($$)); }
        ;

MethodNotify: ObjRef ARROW ID ';'
        { $$=Manage_MethodNotify($1,$3); }
        ;
```

Μικρά αποσπάσματα από την γέννηση και εφαρμογή της τεχνικής (με παραλλαγές) στην υλοποίηση του I-*GET* compiler, A. Savidis, 1996

Προγραμματιστική συμβουλή (7/8)

Το πρόγραμμα YACC σας επιτρέπει να έχετε κώδικα και ορισμούς μέσα στο YACC specification file. *Μην χρησιμοποιήσετε ποτέ αυτήν την ευκολία*.

```
%{
/*
 * I-GET compiler YACC (BISON) implementation for the I-GET language.
 * Start March 27, 1996.
 * Anthony Savidis, 1996 (frozen in June).
 */

extern "C" {
#include <malloc.h>
#include "misch.h"
#include "code.h"
}
#include "symbol.hh"
#include "parse.hh"
#include "voidptr.h"
#include "export.hh"
#include "exprlval.hh"
#include "precond.hh"
#include "evhandle.hh"
#include "codegen.hh"
#include "apistmt.hh"
#include "ioevstmt.hh"
#include "method.hh"
#include "stmts.hh"
#include "hook.hh"

extern int yylex (void);
extern int yylineno;
%}


%union {
    ConstValue          cval;       // constant value
    Symbol*             sym;        // symbol item
    BaseType            builtin;    // builtin data type
    int                 intnum;     // when an integer number is needed
    Bracket             bracket;    // for BracketCollection
    LayerSchemeList*    schemes;    // for VrefRestDefs
    LayerScheme         laysch;     // for VrefRestDef
    char*               str;        // use of string constant
    SymbolList*         symlist;    // anywhere list of symbols
    ExportQualifier     exportType; // for ExportPrefix
    SpecLocalDecl       specloc;    // for SpecLocalDecl
    InLexSpec           lexspec;    // for InLexSpecs, InLexSpe
    InGenSpec           genspec;    // for InGenesisSpecs, InGe
    InAgentSpec         agnspec;    // for InAgentSpecs, InAgen
    InstSpec            instspec;   // for InstSpec, InstSpecs,
    Scheme*             scheme;     // for InstantiationScheme
    SchemeList*         schlist;    // for InstantiationSchemes
    Instantiation*      inst;       // for LexicalInstantiatio
    Code*               code;       // when code is constructe
    Lvalue*             lval;       // for Lvalue
    VobjDef             vobjdef;    // for VobjRestDef, VobjRes
    Expression*         expr;       // All types of expressions
    LvalueList*         lvalues;    // when list of lvalues is
    Initializer*        init;       // for initializers
    InitializerList*    initlist;   // for initializer lists
    MatchTid            mid;        // MatchingTransactionId
    Precondition*       precond;    // for all types of precon
    ExpressionList*     exprlist;   // for ExpressionList
    EventBlock          block;      // for EventBlock
    EventBlockList*     blocks;     // for EventBlocks
    Stmt*               stmt;       // for Stmt
    StringList*         strlist;    // for StringConstList
}


%start  IGET_File

%type   <builtin>       BuiltIn
%type   <intnum>        StarCollection
%type   <bracket>       BracketCollection
%type   <sym>           Pointer Array ArrayOfPointers Variable
%type   <sym>           Enumerated Structure ForwardStruct Type
%type   <sym>           FuncTypeHandle FuncProto FuncIssue Func
%type   <sym>           QualifiedFuncProto QualifiedFuncImpl
```

Στην κατασκευή του I-GET compiler, όλοι οι ορισμοί είναι σε κατάλληλα header files. Έτσι αποφεύγετε τη «μόλυνση» του YACC αρχείου με λογική πέρα της συντακτικής δομής και των καλούμενων σημασιολογικών κανόνων και τα άσκοπα re-generations

1

2

Ιδιαίτερα σε compilers σχετικά μεγάλης κλίμακας και πολυπλοκότητας όπως αυτός (~50KLOC) ο καλός τεμαχισμός ξεκινά ήδη από το YACC file.

Προγραμματιστική συμβουλή (8/8)

```
%type   <agnspec>       InAgentSpecs InAgeSpec InAgeSpecs
%type   <code>          Compound
%type   <sym>           VirtualInstance LexicalInstance VirtualClass
%type   <sym>           Constraint Monitor Method
%type   <scheme>        InstantiationScheme
%type   <schlist>       InstantiationSchemes
%type   <inst>          LexicalInstantiation
%type   <sym>           InputEvent OutputEvent OutputEventPrefix Age
%type   <symlist>       EventParameters InArgs OutArgs
%type   <lval>          Lvalue ObjRef
%type   <expr>          Expression OptionalParent Primary Term Assig
%type   <vobjdef>       VobjRestDef VobjRestDefs
%type   <lvalues>       LvalueList
%type   <init>          Initializer InitializerDef ArrayStructInitia
%type   <initlist>      InitializerList
%type   <str>           TransactionId SharedId
%type   <mid>           MatchingTransactionId MatchingSharedId
%type   <precond>       APIEvent MsgEvent ShUpdateEvent
%type   <precond>       ShCreateEvent ShDestroyEvent Precondition
%type   <exprlist>      ExpressionList
%type   <expr>          FunctionExpr InstantiateAgentExpr
%type   <sym>           EventBlock_EventClass EventHandlerSpec
%type   <block>         EventBlock
%type   <blocks>        EventBlocks
%type   <expr>          EventBlockPrefix EventHandlerSpec_Prefix
%type   <code>          MsgSend CreateSharedObject UpdateSharedObjec
%type   <code>          ReadSharedObject DestroySharedObject
%type   <code>          DoOutputEvent DoInputEvent
%type   <str>           MethodNotify
%type   <stmt>          Stmt Loop Conditional Case ExportedStmt
%type   <code>          Stmts CasePairs CasePair CasePostfix
%type   <intnum>        StoreCaseId
%type   <expr>          StoreCaseExpr
%type   <str>           StringConst CPPCodeHook BridgeIGETtoCPP Brid
%type   <strlist>       StringConstList

%token  <sym> ID
%token  <cval> CONST
```

3

```
%token  <sym> ID
%token  <cval> CONST
%type   <str> HOOK

%token  AGENT BOOL BREAK BRIDGE CASE CHANNEL CHAR CONSTRUCTOR C
%token  DAID DEFAULT DESTROY DESTRUCTOR ELSE ENUM EXPORT EXTERN
%token  FOR HEADERS HOOK HOOKSTMT IF IN INPUTEVENT INSTANTIATIO
%token  LONGINT LONGWORD LONGREAL ME
%token  MESSAGE METHOD MYAGENT NEW NIL NOEXPORT POBJECTID OF OU
%token  PARENT PRIVATE PUBLIC REAL REF RELEASE RETURN SCHEME SH
%token  SHDESTROY SHREAD SHUPDATE STRING STRUCT TERMINATE TYPE
%token  WHILE WORD

%token  OR AND PLUSPLUS MINUSMINUS GT LT LE GE EQ NE
%token  ARROW LEFTARROW DOT ADD_A SUB_A MUL_A DIV_A MOD_A SCOPE

%token  ':' '?' ';' ',' '[' ']' '(' ')' '{' '}' '@'
%token  '=' '^' '+' '-' '*'  '/' '%' '!' '&' '|'


%left   ','
%right  '=' ADD_A SUB_A MUL_A DIV_A MOD_A
%left   OR
%left   AND
%left   '^'
%left   '&'
%left   EQ NE
%left   GT LT GE LE
%left   '+' '-'
%left   '*' '/' '%'
%right  '!' PLUSPLUS MINUSMINUS UMINUS UPLUS USTAR ADDROF
%left   ARROW DOT
%nonassoc SCOPERES


%%

Constructor:
```

Από εδώ και κάτω αρχίζουν οι γραμματικοί κανόνες

4