



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



HY340 : ΓΛΩΣΣΕΣ ΚΑΙ ΜΕΤΑΦΡΑΣΤΕΣ

Φροντιστήριο 2ο Εισαγωγή στο YACC



Yacc

- Μια γεννήτρια συντακτικών αναλυτών για τις γλώσσες C/C++
- Μετατρέπει την περιγραφή μιας context-free γραμματικής σε ένα LALR συντακτικό αναλυτή γραμμένο
 - LALR = Look-Ahead Left-to-right parse, Rightmost-derivation
- Θα χρησιμοποιήσουμε το *bison*, μια βελτιωμένη έκδοση του yacc



Δομή Προγράμματος Yacc

%{

Πρόλογος

%}

Δηλώσεις yacc

%%

Περιγραφή γραμματικής

%%

Επίλογος (προαιρετικό)



Πρόλογος

- Ο πρόλογος μπορεί να περιέχει δηλώσεις macros, συναρτήσεων και μεταβλητών.
- Ότι προστίθεται σε αυτό το τμήμα αντιγράφεται χωρίς αλλαγές στην αρχή του παραγόμενου αρχείου .c που περιέχει τον κώδικα του συντακτικού αναλυτή
- Το τμήμα αυτό είναι προαιρετικό και μπορεί να παραληφθεί αφαιρώντας τα διαχωριστικά %{ και %}
- Μπορούμε επίσης να έχουμε περισσότερα από ένα τμήματα προλόγου ανάμεσα στα οποία παρεμβάλλονται δηλώσεις του yacc.

```
%{  
#include <stdio.h>  
#include "def.h"  
void print_token_value (FILE *, int, YYSTYPE);  
extern int lineno;  
%}
```



Δηλώσεις Yacc

- Σε αυτό το τμήμα δηλώνονται τα σύμβολα της γραμματικής καθώς και κάποια χαρακτηριστικά τους
 - Δήλωση τερματικών και μη τερματικών συμβόλων
 - Δήλωση αρχικού συμβόλου
 - Καθορισμός προτεραιότητας
- Επίσης δηλώνονται κάποιες παράμετροι που επηρεάζουν το συντακτικό αναλυτή
 - Κυρίως σε σχέση με τα ονόματα των παραγόμενων αρχείων του συντακτικού αναλυτή και των συναρτήσεων που προσφέρει



Δηλώσεις Yacc - Γραμματική (1/3)

- **%token TOKEN** – Ορίζει το τερματικό σύμβολο *TOKEN*
- **%union** – Ορίζει ένα union με τους τύπους που μπορούν να πάρουν τα σύμβολα (τερματικά και μη)
 - Π.χ. το **%union { int intValue; char * strValue; symrec *tPtr; }** δηλώνει ότι τα σύμβολα μπορούν να έχουν τύπο int, char * ή symrec
 - Τα ονόματα που δίνονται (intValue, strValue, tPtr) είναι για να μπορούμε να αναφερθούμε στους τύπους
- **%token <intVal> TOKEN** – Ορίζει το τερματικό σύμβολο *TOKEN*, με τύπο αυτό που αντιστοιχεί στο *intVal* πεδίο του union.
- **%type <intVal> expr** – Ορίζει το μη τερματικό σύμβολο *expr* με τύπο αυτό που αντιστοιχεί στο *intVal* πεδίο του union.



Δηλώσεις Yacc - Γραμματική (2/3)

- **%start symbol** – Ορίζει το αρχικό σύμβολο της γραμματικής
 - Αν παραληφθεί, αρχικό σύμβολο θεωρείται το πρώτο μη τερματικό σύμβολο που εμφανίζεται στο τμήμα της περιγραφής της γραμματικής
- **%destructor { code } symbols** – Ορίζει ένα τμήμα κώδικα που εκτελείται για τα δοθέντα σύμβολα όταν αυτά σταματήσουν να χρησιμοποιούνται
 - Προσφέρεται από το bison αλλά όχι από το yacc
 - Πολύ χρήσιμο για αποδέσμευση μνήμης σε περιπτώσεις λάθους
 - %union { char *string; }
%token <string> STRING
%destructor { free (\$\$); } STRING
- **%expect n** – Δηλώνει ότι αναμένουμε η γραμματική μας να έχει n conflicts
 - Ελέγχουμε πάντα όμως ότι ο yacc παίρνει τη σωστή απόφαση

Δηλώσεις Yacc - Γραμματική (3/3)

■ Προτεραιότητες

- **%left, %right, %nonassoc**
- Ορίζουν την προτεραιότητα στα token που ακολουθούν και βρίσκονται στην ίδια γραμμή, π.χ. %left ADD SUB
- Ίδια προτεραιότητα στα σύμβολα της ίδιας γραμμής
- Αυξανόμενη προτεραιότητα από πάνω προς τα κάτω, π.χ.
 - %left ADD, SUB /* + - */
 - %left MUL, DIV /* * / */
 - %right EXP /* ^ */
 - %nonassoc EQ /* == */
- To *left* σημαίνει ότι έχουμε αριστερή προσεταιριστικότητα
 - ◆ To $1+2+3$ σημαίνει $(1+2) + 3$
- To *right* σημαίνει ότι έχουμε δεξιά προσεταιριστικότητα
 - ◆ To 2^2^3 σημαίνει $2^{(2^3)}$
- To *nonassoc* σημαίνει ότι δεν υπάρχει προσεταιριστικότητα
 - ◆ To $1 == 2 == 3$ δεν επιτρέπεται
- To $1+2*3^4^5-6$ υπολογίζεται ως $(1 + (2 * (3 ^ (4 ^ 5)))) - 6$



Δηλώσεις Yacc – Παράμετροι (1/2)

■ **%defines**

- Παράγει ένα header file με τις δηλώσεις macros για τα σύμβολα της γραμματικής, καθώς και κάποιες επιπλέον δηλώσεις
- Αν το παραγόμενο αρχείο του συντακτικού αναλυτή είναι το *parser.c*, τότε το header file θα έχει όνομα *parser.h*

■ **%output=“file”**

- Ορίζει το όνομα του παραγόμενου αρχείου που θα περιέχει τον κώδικα του συντακτικού αναλυτή

■ **%file-prefix=“prefix”**

- Αλλάζει το πρόθεμα των αρχεία που παράγονται ώστε να είναι σαν το αρχείο εισόδου να λεγόταν *prefix.y*

■ **%name-prefix=“prefix”**

- Αλλάζει το πρόθεμα των συμβόλων που χρησιμοποιεί ο συντακτικός αναλυτής σε “*prefix*” αντί για “*yy*”



Δηλώσεις Yacc – Παράμετροι (2/2)

■ **%pure-parser**

- Ο παραγόμενος συντακτικός αναλυτής είναι reentrant (μπορούμε να κάνουμε νέα κλήση στην `yyparse` πριν τελειώσει η προηγούμενη). Προσοχή στο πρωτότυπο που θα πρέπει να έχει η `yylex`.

■ **%parse-param {argument-declaration}**

- Προσθέτει μια παράμετρο στο πρωτότυπο της συνάρτησης `yyparse`

■ **%lex-param {argument-declaration}**

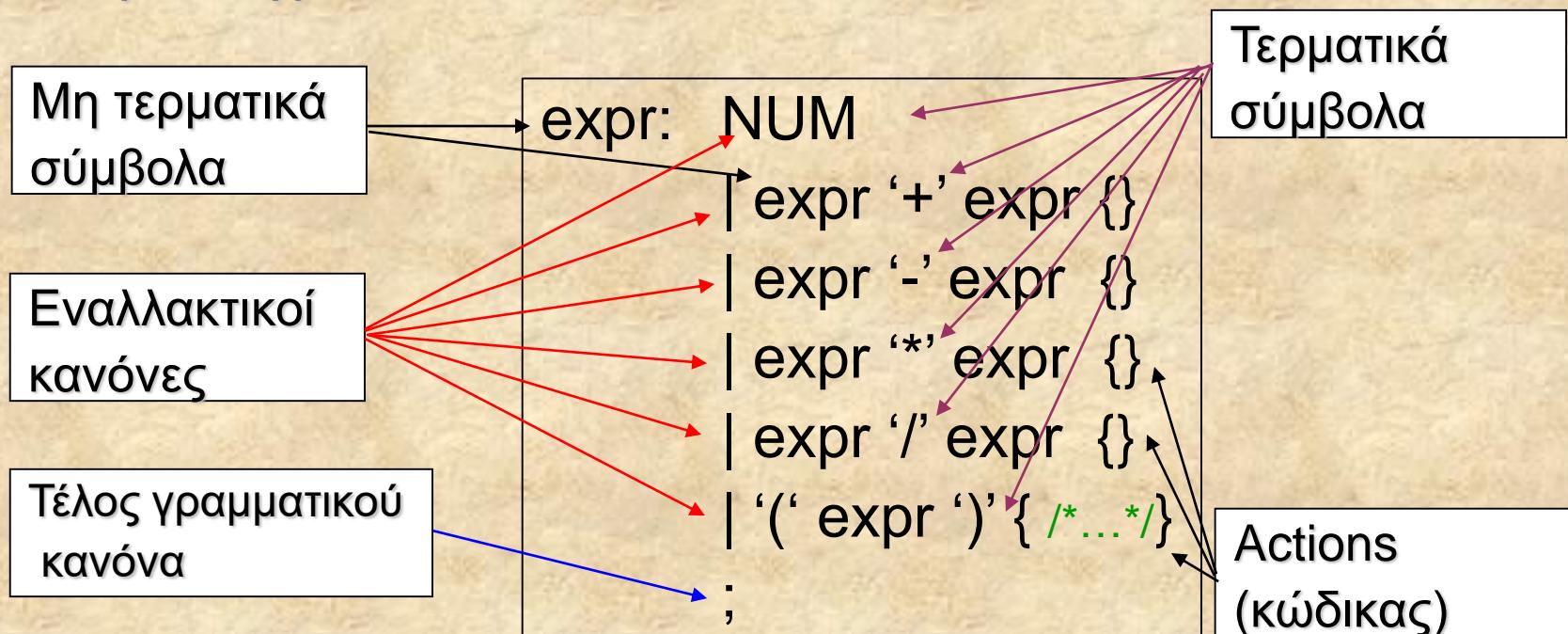
- Προσθέτει μια παράμετρο στο πρωτότυπο της συνάρτησης `yylex`

■ **%error-verbose**

- Χρησιμοποιείται για να πάρουμε πιο αναλυτικά μηνύματα λάθους στην κλήση της `yyerror`

Περιγραφή γραμματικής

- Σε αυτό το τμήμα δίνεται η περιγραφή μιας context-free γραμματικής μέσω γραμματικών κανόνων
- Παράδειγμα:



- Επιτρέπεται και ο κενός κανόνας που κάνει match το κενό string, π.χ. *program*: /*empty*/ | statements ;



Επίλογος

- Σκοπός του είναι μόνο η εύκολη και άμεση προσθήκη υλοποιήσεων των συναρτήσεων που χρησιμοποιούνται από τον παραγόμενο συντακτικό αναλυτή
- Ότι προστίθεται σε αυτό το τμήμα αντιγράφεται χωρίς αλλαγές στο τέλος του παραγόμενο αρχείο .c που περιέχει τον κώδικα του συντακτικού αναλυτή
- Το τμήμα αυτό είναι προαιρετικό και όταν παραλειφθεί μπορεί να παραλειφθεί και το δεύτερο σύμβολο “%%”

```
%%
```

```
int main(int argc, char **argv) {  
    yyparse();  
    return 0;  
}
```



Lex & Yacc

■ Lex

- Κάνει include το header file που παράγεται από το yacc για να δει τα tokens και τους τύπους τους
- Μέσα στα actions γράφουμε κώδικα που επιστρέφει στο yacc ένα-ένα τα tokens που αναγνωρίστηκαν
- Επίσης, φροντίζουμε να παρέχουμε τις τιμές για τα tokens που έχουν και κάποιο δηλωμένο τύπο
 - ◆ Π.χ. για τον ακέραιο 5 θα πρέπει να δώσουμε στο yacc το token INTEGER, αλλά και να του παρέχουμε αριθμητική τιμή 5

■ Yacc

- Στον πρόλογο δηλώνουμε τη συνάρτηση yylex που είναι υπεύθυνη για την λεξικογραφική ανάλυση (και παρέχεται από το lex)
- Εφαρμόζουμε τους κανόνες της γραμματικής ανάλογα με τα tokens που επιστρέφονται από την yylex
- Στα actions χρησιμοποιούμε και τις τιμές που έχουν τα tokens με συγκεκριμένους τύπους
 - ◆ Π.χ. εκτελούμε τον κανόνα για τον ακέραιο και κατόπιν στο action παίρνουμε και την αριθμητική τιμή του



Παράδειγμα (1/7)

- Γραμματική για έναν απλό υπολογιστή αριθμητικών εκφράσεων
 - Υποστηρίζουμε ανάθεση μίας αριθμητικής έκφρασης σε μία μεταβλητή
 - Υποστηρίζουμε αριθμητικές εκφράσεων που χωρίζονται με χαρακτήρες τέλους γραμμής
- Λεξικογραφικός αναλυτής
 - Σύμβολα +, -, *, /, (,), =, \n, ακέραιους, ids
- Συντακτικός αναλυτής
 - Αρχική λίστα με εκχωρήσεις σε μεταβλητές
 - Λίστα εκφράσεων που περιέχουν μεταβλητές και αριθμητικές εκφράσεις



Παράδειγμα - Lex (2/7)

```
%{  
    #include "parser.h" /* <- will be generated from parser.y */  
}  
  
/* Flex options */  
%option noyywrap  
%option yylineno  
  
/* Flex macros */  
id . . . [a-zA-Z][a-zA-Z_0-9]*  
integer. . [0-9]+  
  
%%  
  
"+". . . { return '+'; }  
"-". . . { return '-'; }  
"*". . . { return '*' ; }  
"/". . . { return '/' ; }  
"(". . . { return '(' ; }  
")". . . { return ')' ; }  
"=". . . { return '=' ; }  
[\\n]+. . . { return '\\n'; }  
  
{integer}. . . { return INTEGER; }  
{id}. . . { return ID; }  
  
[ \\t]+. . . {}  
  
. . . { fprintf(stderr, "Cannot match character '%s' with any rule\n", yytext); }
```

To header file *parser.h*
δημιουργείται από τον yacc

Μετατρέπει πολλαπλά
\n σε ένα

Οι τιμές των συμβολικών
ονομάτων INTEGER, ID
είναι ορισμένες στο
header file *parser.h*

Παράδειγμα – Πρόλογος Yacc (3/7)

```
%{  
    #include <stdio.h>  
    int yyerror (char* yaccProvidedMessage);  
    int yylex (void);  
  
    extern int yylineno;  
    extern char* yytext;  
    extern FILE* yyin;  
}  
  
%start program  
  
%token ID INTEGER  
  
%right      '='  
%left       ','  
%left       '+' '-'  
%left       '*' '/'  
%nonassoc   UMINUS  
%left       '(' ')' {  
%%
```

Καλείται από τον yacc με ένα μήνυμα όταν “ανακαλύψει” κάποιο λάθος

Αρχικό σύμβολο γραμματικής

Δήλωση των τερματικών συμβόλων που χρησιμοποιούνται από τον yacc και τον flex

Ορισμός προτεραιοτήτων και προσεταιριστικότητας

Παράδειγμα – Γραμματική Yacc (4/7)

```
program: . . . assignments expressions
. . . . . | /* empty */ ← ε κανόνας
. . .
. . .

expression: . . . INTEGER
. . . . . | ID
. . . . . | expression '+' expression
. . . . . | expression '-' expression
. . . . . | expression '*' expression
. . . . . | expression '/' expression
. . . . . | '(' expression ')'
. . . . . | expression %prec UMINUS ← Αλλαγή προτεραιότητας
. . .
. . .

expr: . . . expression '\n'
expressions: . . . expressions expr ← Δημιουργία μη κενής
. . . . . | expr λίστας expressions
. . .
. . .

assignment: . . . ID '=' expression '\n'
. . .
assignments: . . . assignments assignment ← Δημιουργία λίστας
. . . . . | /* empty */
. . .
. . .
```

ε κανόνας

Αλλαγή προτεραιότητας
κανόνα

Δημιουργία μη κενής
λίστας expressions

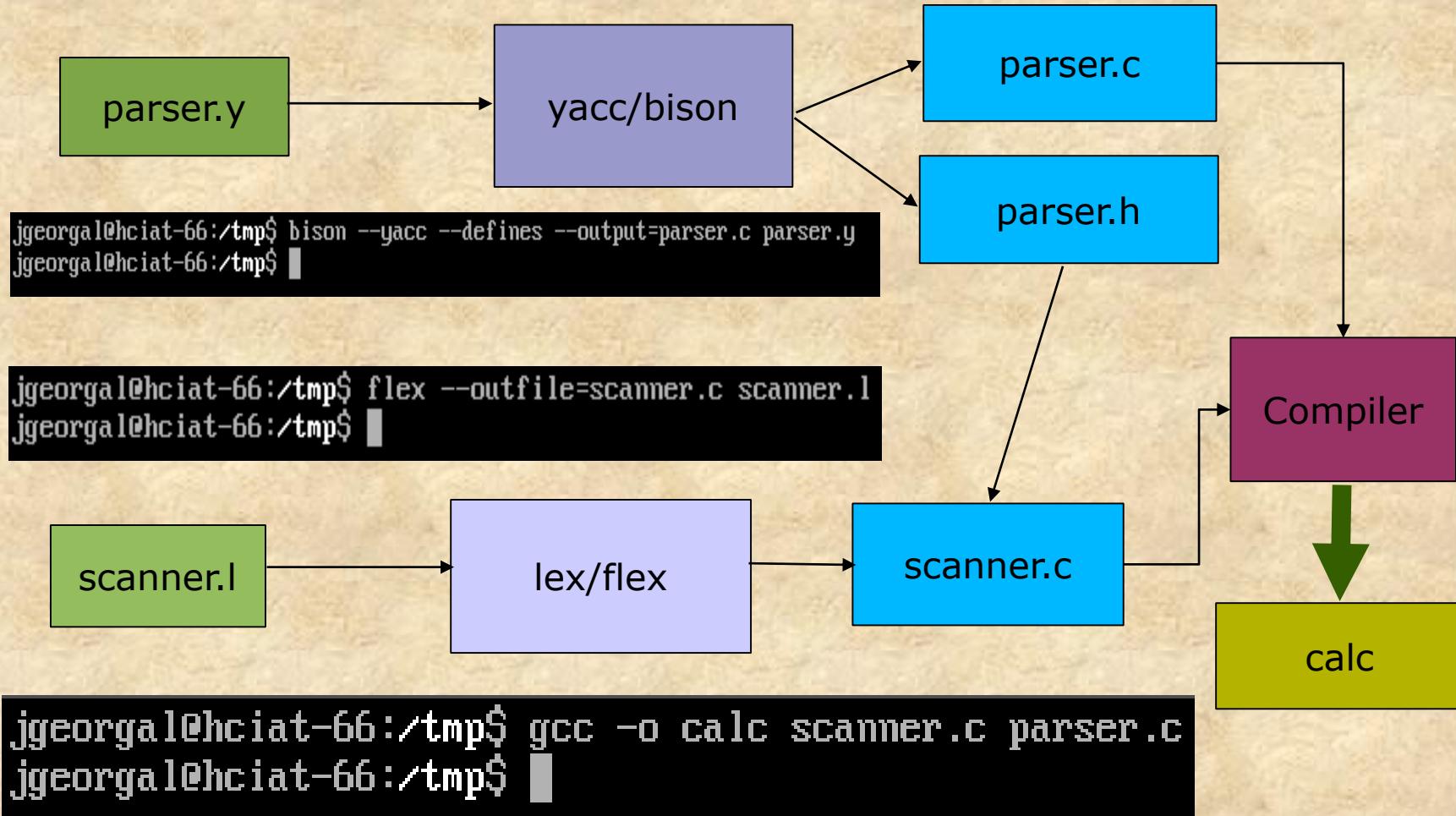
Δημιουργία λίστας
assignments που
μπορεί να είναι κενή



Παράδειγμα – Επίλογος Yacc (5/7)

```
%  
  
int yyerror (char* yaccProvidedMessage) ← Την παρέχουμε  
{  
    .   fprintf(stderr, "%s: at line %d, before token: %s\n", yaccProvidedMessage, yylineno, yytext);  
    .   fprintf(stderr, "INPUT NOT VALID\n");  
}  
  
//*****  
  
int main(int argc, char** argv)  
{  
    if (argc > 1) {  
        if (!(yyin = fopen(argv[1], "r"))){  
            .   fprintf(stderr, "Cannot read file: %s\n", argv[1]);  
            return 1;  
        }  
    }  
    else  
        yyin = stdin;  
  
    yyparse(); ← Παράγεται από  
    return 0;  
}
```

Παράδειγμα – Παραγωγή εκτελέσιμου (6/7)



Παράδειγμα – Test programs (7/7)

Έγκυρο πρόγραμμα

```
pi = 3141592  
|  
R = 321  
perifereia = 2 * pi * R  
  
1 + (8 / -6) * (7 / R) * (perifereia - R)  
  
perifereia + pi * R - 2 * 718 / 2  
  
1  
perifereia + foo / zoo * lala  
1 + 1
```

Μη έγκυρα προγράμματα

```
pi = 3141592  
|  
perifereia + pi * R - 2 * 718 / 2  
  
result = pi * 3  
|  
pi = 3141592  
R = 321  
perifereia = 2 * pi * R  
  
1 + (8 / -6) * (7 / R) * (perifereia - R)  
  
perifereia + pi * R - 2 * 718 / 2  
  
1  
perifereia + foo / zoo * lala  
1 + 1  
-2
```

Δήλωση μετά από εκφράσεις

η στις δηλώσεις



Actions (1/5)

- Μπορούμε να παρεμβάλλουμε ανάμεσα στα σύμβολα των δεξιών μερών των κανόνων κάποια actions, δηλαδή c, c++ κώδικα μέσα σε {}
 - Εκτελούνται κάθε φορά που ενεργοποιείται η συγκεκριμένη παραγωγή
 - Παράδειγμα

```
expr : ID { printf("Found ID\n"); }
        | expr '+' { int a = 3; } expr { int b; }
        | /* empty */ { printf("empty\n"); }
        ;
```

Actions (2/5)

- Μπορούμε μέσα από τα actions να αναφερθούμε σε όλα τα σύμβολα που έχουν δηλωμένο τύπο
 - Στο αριστερό μέρος ενός κανόνα αντιστοιχεί το σύμβολο `$$`
 - Κάθε δεξιό σύμβολο ενός κανόνα, μπορεί να αναφερθεί μέσω του συμβολικού ονόματος `$N` ($N = 1, 2, \dots$) ανάλογα με την σχετική του θέση
 - Προσοχή, καθώς στην αρίθμηση μετράνε και τα ίδια τα actions

```

expression::= INTEGER . . . . . { $$ = $1; }
. . . . | ID . . . . . { $$ = lookup($1); free($1); }
. . . . | expression '+' expression . { $$ = $1 + $3; }
. . . . | expression '-' expression . { $$ = $1 - $3; }
. . . . | expression '*' expression . { $$ = $1 * $3; }
. . . . | expression '/' expression . { $$ = $1 / $3; }
. . . . | '(' expression ')' . . . { $$ = $2; }
. . . . | '-' expression %prec UMINUS. { $$ = -$2; }
. . . . ;
  
```

non_terminal:	FOO { g_var = \$1; }	BAR { g_var2 = \$3; }		
<code>\$\$</code>	<code>\$1</code>	<code>\$2</code>	<code>\$3</code>	<code>\$4</code>

Actions (3/5)

- Οι τιμές στα τερματικά σύμβολα δίνονται από τον λεξικογραφικό αναλυτή
 - Ο yacc δηλώνει μία καθολική μεταβλητή, στιγμιότυπο του “union” που έχουμε ορίσει, με το όνομα *yyval*
 - Κάθε φορά που αναγνωρίζουμε ένα σύμβολο στον lex/flex, πριν επιστρέψουμε τον “κωδικό” του, αποθηκεύουμε στο “union” και την τιμή του (αν έχει)

```
%union {  
    char*      stringValue;  
    int        intValue;  
    double     realValue;  
}
```

```
{ integer}. { yyval.intValue = atoi(yytext); return INTEGER; }  
{ id}.   { yyval.stringValue = strdup(yytext); return ID; }
```



Actions (4/5)

■ Παράδειγμα με actions (lex)

```
%{  
    #include "parser.h" /* <- will be generated from parser.y */  
}  
  
/* Flex options */  
%option noyywrap  
%option yylineno  
  
/* Flex macros */  
id . . . [a-zA-Z][a-zA-Z_0-9]*  
integer . . [0-9]+  
  
%%  
  
"+". . . { return '+'; }  
"-". . . { return '-'; }  
"*". . . { return '*' ; }  
"/". . . { return '/' ; }  
"(". . . { return '(' ; }  
")". . . { return ')' ; }  
"=". . . { return '=' ; }  
";". . . { return ';' ; }  
  
{integer}. . . { yyval.intValue = atoi(yytext); return INTEGER; }  
(id). . . { yyval.stringValue = strdup(yytext); return ID; }  
  
[ \t\n]+. . . {}  
  
. . . { fprintf(stderr, "Cannot match character '%s' with any rule\n", yytext); }
```

Actions (5/5)

■ Παράδειγμα με actions (yacc)

```
/* Token Types Union */
%union {
    char* stringValue;
    int intValue;
}

/* Rest of the tokens */

%token <stringValue> ID
%token <intValue> INTEGER

%right   :=
%left   (+) (-)
%left   (*) (/)
%right  UMINUS
%left   ('(' ')')

%type <intValue> expression

%destructor { free($$); } ID

%%
program: . . . assignments expressions {};
        | /* empty */ {}

assignments: . . . ;
            | /* empty */ {}

expression: . . . INTEGER . . . . . . { $$ = $1; }
           | ID . . . . . . { $$ = lookup($1); free($1); }
           | expression '+' expression { $$ = $1 + $3; }
           | expression '-' expression { $$ = $1 - $3; }
           | expression '*' expression { $$ = $1 * $3; }
           | expression '/' expression { $$ = $1 / $3; }
           | '(' expression ')' . . . { $$ = $2; }
           | '-' expression %prec UMINUS { $$ = -$2; }

expr: . . . expression ';' { fprintf(stdout, "Result is: %d\n", $1); }

expressions: . . . expressions expr {};
            | expr {};

assignment: . . . ID '=' expression ';' { assign($1, $3); }
           | ;

assignments: . . . assignments assignment {};
            | /* empty */ {};
```



Ασάφειες στη γραμματική (1/3)

- Η περιγραφή μιας γραμματικής μπορεί να παράγει μία συγκεκριμένη ακολουθία συμβόλων με δύο ή περισσότερους τρόπους
- Π.χ. με αυτή τη γραμματική, η έκφραση $1+2^*3$ μπορεί να παραχθεί ως:

- expr \rightarrow expr '+' expr
 \rightarrow expr '+' expr '*' expr
 \rightarrow NUMBER '+' NUMBER '*' NUMBER
 \rightarrow 1+2*3
- expr \rightarrow expr '*' expr
 \rightarrow expr '+' expr '*' expr
 \rightarrow NUMBER '+' NUMBER '*' NUMBER
 \rightarrow 1+2*3

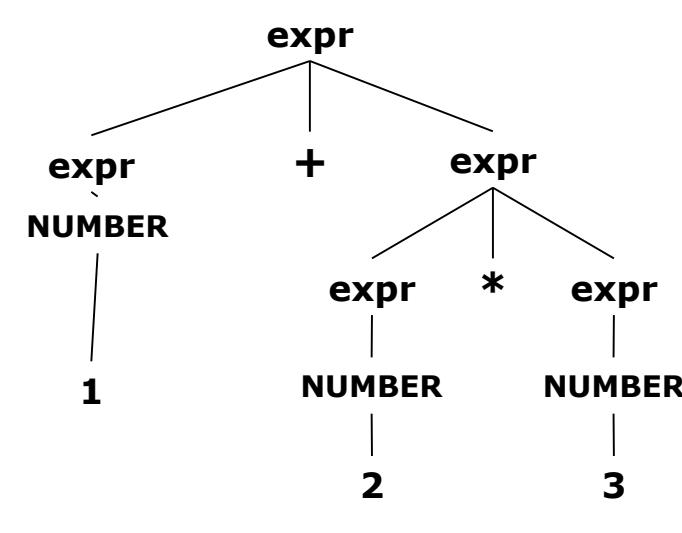
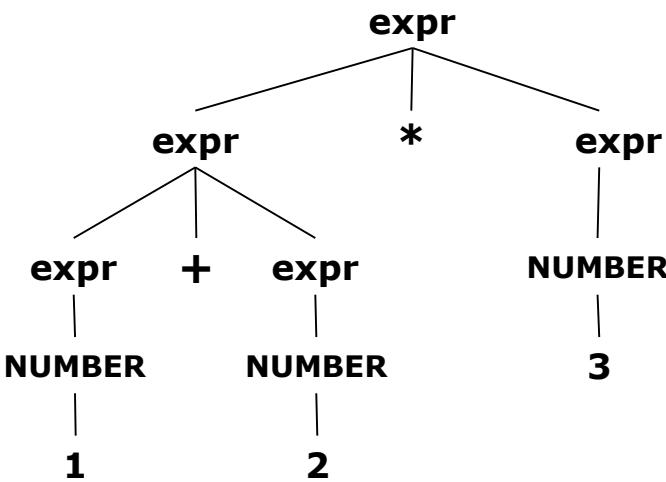
```
expr : expr '+' expr  
      | expr '*' expr  
      | NUMBER  
      ;
```

Ασάφειες στη γραμματική (2/3)

- Η ασάφεια αυτή φαίνεται καλύτερα στα συντακτικά δέντρα (syntax trees)
- Το προηγούμενο παράδειγμα δίνει τα παρακάτω συντακτικά δέντρα

$$1+2*3 \rightarrow (1+2) * 3$$

$$1+2*3 \rightarrow 1+ (2 * 3)$$





Ασάφειες στη γραμματική (3/3)

- Αυτή η αμφισημία στις γραμματικές δεν είναι επιθυμητή επειδή μπορούμε να καταλήξουμε σε λάθος συμπεράσματα
- Απαλοιφή ασάφειας
 - Με προτεραιοτήτων τελεστών
 - Με αλλαγή της γραμματικής
 - ◆ Στο προηγούμενο παράδειγμα ας πούμε έχουμε:

```
expr : expr '+' expr1  
      | expr1;  
  
expr1: expr1 '*' NUMBER  
      | NUMBER;
```

```
expr → expr '+' expr1  
      → expr1 '+' expr1  
      → expr1 '+' expr1 '*' NUMBER  
      → expr1 '+' NUMBER '*' NUMBER  
      → NUMBER '+' NUMBER '*' NUMBER  
      → 1+2*3
```

Πώς δουλεύει ο Yacc (1/5)

- Το parsing μίας ακολουθίας εισόδου γίνεται στην πραγματικότητα από κάτω προς τα πάνω (bottom-up parsing)
 - π.χ η έκφραση $3^*4^*(3+2)$ ανάγεται σε expr ως εξής:

◆ $3^*4^*(3+2)$

→ NUMBER '*' 4 '*' '(' 3 '+' 2 ')'
→ expr '*' 4 '*' '(' 3 '+' 2 ')'
→ expr '*' NUMBER '*' '(' 3 '+' 2 ')'
→ expr '*' expr '*' '(' 3 '+' 2 ')'
→ expr '*' '(' 3 '+' 2 ')'
→ expr '*' '(' NUMBER '+' 2 ')'
→ expr '*' '(' expr '+' 2 ')'
→ expr '*' '(' expr '+' NUMBER ')'
→ expr '*' '(' expr '+' expr ')'
→ expr '*' '(' expr ')'
→ expr '*' expr
→ expr

```
expr : expr '+' expr
      | expr '*' expr
      | '(' expr ')'
      | NUMBER
      ;
```



Πώς δουλεύει ο Yacc (2/5)

- Για την υλοποίηση μίας “από κάτω προς τα πάνω” συντακτικής ανάλυσης θα χρησιμοποιήσουμε
 - Μία στοίβα στην οποία θα αποθηκεύονται τα σύμβολα που αποτελούν τη γλώσσα
 - Μία ενέργεια “shift” η οποία θα τοποθετεί ένα σύμβολο από την είσοδο στη στοίβα
 - Μία ενέργεια “reduce” η οποία θα αντικαθιστά ένα σύμβολο ή μία ακολουθία συμβόλων που εμφανίζονται στο δεξιό μέρος ενός κανόνα με το σύμβολο που εμφανίζεται στο αριστερό μέρος του κανόνα



Πώς δουλεύει ο Yacc (3/5)

- Μία ενέργεια “accept” που εφαρμόζεται όταν διαβαστεί ολόκληρη η είσοδος και έχει αναγνωριστεί το αρχικό σύμβολο
 - Σηματοδοτεί την εγκυρότητα της ακολουθίας εισόδου σύμφωνα με την γραμματική
- Μία ενέργεια “error” που εφαρμόζεται όταν δεν μπορεί να εφαρμοστεί καμία ενέργεια “reduce” πάνω στη στοίβα, δηλαδή τα σύμβολα που υπάρχουν σε αυτήν δεν υπάρχουν στο δεξί μέρος κανενός κανόνα
 - Σηματοδοτεί την αδυναμία της γραμματικής να περιγράψει την ακολουθία εισόδου



Πώς δουλεύει ο Yacc (4/5)

- Έτσι ο αλγόριθμος που εφαρμόζεται για την συντακτική ανάλυση της ακολουθίας εισόδου, είναι ο εξής
 - Ανάλογα με την κατάσταση στην οποία βρίσκεται, ο συντακτικός αναλυτής, αποφασίζει εάν χρειάζεται το επόμενο σύμβολο από την είσοδο (lookahead token) για να επιλέξει την επόμενη ενέργεια
 - Ανάλογα με την τρέχουσα κατάσταση και το επόμενο σύμβολο (εάν χρειάζεται) αποφασίζει να εκτελέσει μία από τις ενέργειες “shift”, “reduce”, “accept” ή “error”



Πώς δουλεύει ο Yacc (5/5)

■ Παραδείγματα

```
expr  : expr '+' expr  
      | expr '*' expr  
      | '(' expr ')'   
      | NUMBER  
      ;
```

Στοίβα	Είσοδος	Ενέργεια
	$3^{**}4^{*(3+2)}$	shift
3	$^{**}4^{*(3+2)}$	reduce
expr	$^{**}4^{*(3+2)}$	shift
expr *	$4^{*(3+2)}$	shift
expr * *	$4^{*(3+2)}$	error

Ανεπιτυχής συντακτική ανάλυση

Στοίβα	Είσοδος	Ενέργεια
	$3^*4^*(3+2)$	shift
3	$4^*(3+2)$	reduce
expr	$4^*(3+2)$	shift
expr *	$4^{*(3+2)}$	shift
expr * 4	$(3+2)$	reduce
expr * expr	$(3+2)$	reduce
expr	$(3+2)$	shift
expr *	$(3+2)$	shift
expr * ($3+2)$	shift
expr * (3	$+2)$	reduce
expr * (expr	$+2)$	shift
expr * (expr +	$2)$	shift
expr * (expr + 2)	reduce
expr * (expr + expr)	reduce
expr * (expr)	shift
expr * (expr)		reduce
expr * expr		reduce
expr		accept

Επιτυχής συντακτική ανάλυση



Yacc Conflicts (1/6)

- Συμπεριφορά του yacc όταν για μία παραγωγή υπάρχουν δύο ή περισσότεροι “δρόμοι”
 - Ουσιαστικά όταν υπάρχουν δύο ή παραπάνω διαφορετικές παραγωγές για μία ακολουθία εισόδου, ο yacc σε κάποιο σημείο θα βρεθεί στο δίλημμα να κάνει “reduce” με τον κατάλληλο κανόνα ή να κάνει “shift” και να κάνει “reduce” με έναν επόμενο κανόνα (ο οποίος περιέχει και όλα τα σύμβολα του πρώτου κανόνα)
 - Το δίλημμα αυτό λέγεται ***shift/reduce conflict***

Yacc Conflicts (2/6)

■ Παράδειγμα

```
expr : ifelse
      | NUMBER
      ;
ifelse: IF '(' NUMBER ')' expr else
      ;
else   : ELSE expr
      | /* empty */
      ;
```

Στοίβα	Είσοδος	Ενέργεια
	IF (4) IF (5) 5 ELSE 6	shift
IF	(4) IF (5) 5 ELSE 6	shift
IF (4) IF (5) 5 ELSE 6	shift
IF (4) IF (5) 5 ELSE 6	shift
IF (4)	IF (5) 5 ELSE 6	shift
...	...	shift ...
IF (4) IF (5) 5	ELSE 6	shift ή reduce



Yacc Conflicts (3/6)

- Μπορεί να έχουμε και δύο διαφορετικούς κανόνες με ίδιο δεξιό μέρος
- Στην περίπτωση αυτή, όταν ο yacc αναγνωρίσει την ακολουθία συμβόλων που περιγράφεται από αυτό, δε θα ξέρει ποιο κανόνα να χρησιμοποιήσει για να κάνει “reduce”
 - Το δίλημμα αυτό λέγεται ***reduce/reduce conflict***



Yacc Conflicts (4/6)

- Ο yacc παράγει συντακτικούς αναλυτές, ακόμα κι αν η γραμματική περιέχει shift/reduce ή reduce/reduce conflicts
 - Κατά τη διάρκεια shift/reduce conflict ο yacc κάνει εξ' ορισμού “shift”
 - Κατά τη διάρκεια reduce/reduce conflict ο yacc κάνει “reduce” με τον κανόνα που βρίσκεται πιο κοντά στην αρχή του αρχείου “.y”
- Εάν τρέξουμε τον yacc (ή bison) με την παράμετρο “-v”, ο τελευταίος δημιουργεί ένα αρχείο με κατάληξη “.output” στο οποίο επισημαίνει τα σημεία των κανόνων που οδηγούν σε conflicts



Yacc Conflicts (5/6)

- Το αρχείο είναι χωρισμένο στα παρακάτω τμήματα
 - Αχρησιμοποίητοι κανόνες, τερματικά και μη-τερματικά σύμβολα
 - Καταστάσεις που περιέχουν shift/reduce και reduce/reduce conflicts
 - Η γραμματική που αναγνώρισε ο yacc
 - Τερματικά και μη-τερματικά σύμβολα
 - Όλες οι καταστάσεις του αυτόματου
- Κάθε κατάσταση του αυτόματου περιέχει
 - Τον κανόνα μαζί με μια τελεία (**cursor**) που χωρίζει το κομμάτι του κανόνα που έχει ήδη αναγνωριστεί, από το κομμάτι που υπολείπετε
 - Τις πιθανές εισόδους, ακολουθούμενες από τις ενέργειες που θα εκτελέσει ο yacc
 - Τέλος, μπορεί να υπάρχει το '\$default', που υποδηλώνει οποιαδήποτε άλλη είσοδο



Yacc Conflicts (6/6)

- Για παράδειγμα, έστω η δίπλα γραμματική, στην οποία ξεχάσαμε να βάλουμε προτεραιότητα στο '/'
 - Στο output αρχείο εμφανίζονται τα παρακάτω:

State 11

1 exp: exp . '+' exp
2 | exp . '-' exp
3 | exp . '*' exp
4 | exp . '/' exp
4 | exp '/' exp .

'+' shift, and go to state 4
'-' shift, and go to state 5
'*' shift, and go to state 6
'/' shift, and go to state 7

'+' [reduce using rule 4 (exp)]
'-' [reduce using rule 4 (exp)]
'*' [reduce using rule 4 (exp)]
'/' [reduce using rule 4 (exp)]
\$default reduce using rule 4 (exp)

State 8 conflicts: 1 shift/reduce

State 9 conflicts: 1 shift/reduce

State 10 conflicts: 1 shift/reduce

State 11 conflicts: 4 shift/reduce

State 8

1 exp: exp . '+' exp

1 | exp '+' exp .

NUM + NUM / NUM: is it

2 | exp . '-' exp

NUM + (NUM / NUM) or

3 | exp . '*' exp

(NUM + NUM) / NUM?

4 | exp . '/' exp

'*'

shift, and go to state 6

'/'

shift, and go to state 7

'/'

[reduce using rule 1 (exp)]

\$default

reduce using rule 1 (exp)

```
%token NUM
%left '+'
%left '-'
%left '*'
%%
exp: exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | exp '/' exp
    | NUM;
```

Oι ενέργειες μέσα σε [] απορρίπτονται από τον bison



Error Handling (1/4)

- Ο yacc εξ' ορισμού, όταν δεν μπορεί να αναγνωρίσει την ακολουθία συμβόλων εισόδου, καλεί την συνάρτηση `yyerror` και τερματίζει την λειτουργία του
- Πολλές φορές δε θέλουμε να τερματίζουμε τη λειτουργία του προγράμματος μας στο πρώτο συντακτικό λάθος
 - Ειδικά σε “interactive” προγράμματα όπως το “bc” του UNIX



Error Handling (2/4)

- Η επαναφορά της λειτουργίας του συντακτικού αναλυτή μετά από συντακτικό λάθος (error recovery) είναι πολύ δύσκολη διαδικασία, επειδή πρέπει να “μαντέψουμε” που τελειώνει το κομμάτι που δεν συμφωνεί με την γραμματική ... και που ξεκινάει το κομμάτι που είναι εν δυνάμει σωστό
 - Ο yacc προσφέρει ένα σύμβολο με το όνομα **error** το οποίο καταναλώνει τα περιεχόμενα της στοίβας και τα επόμενα σύμβολα εισόδου, μέχρι να ανακαλύψει τουλάχιστον τρία σύμβολα που ξεκινούν ένα άλλο κανόνα

```
n_term: FOO  
| error  
;
```



Error Handling (3/4)

- Μπορούμε επίσης να κατευθύνουμε την διαδικασία επαναφοράς προσθέτοντας σύμβολα στην παραγωγή που περιέχει το σύμβολο error και προσθέτοντας actions
 - Παράδειγμα
 - ◆ Όταν συμβεί λάθος ο συντακτικός αναλυτής αγνοεί όλα τα σύμβολα μέχρι να συναντήσει ';' :

```
expr: . . . expression ';' { fprintf(stdout, "Result is: %d\n", $1); }
. . . | error ';'
```



Error Handling (4/4)

- Ο yacc προσφέρει τα εξής macros που μπορούν να χρησιμοποιηθούν σε κανόνες
 - **yyerrok** – εξαναγκάζει τον συντακτικό αναλυτή να θεωρήσει ότι βγήκε από την κατάσταση επαναφοράς μετά από λάθος και έτσι να αρχίσει κατευθείαν να ειδοποιεί για καινούρια λάθη
 - **yyclearin** – καθαρίζει το lookahead

```
expr: . . . expression ';' { fprintf(stdout, "Result is: %d\n", $1); }
      | error ';' { yyerrok; };
```



References

■ Bison Home Page

- <http://www.gnu.org/software/bison>

■ Bison Manual

- [http://www.gnu.org/software/bison/manual/biso
n.html](http://www.gnu.org/software/bison/manual/bison.html)

■ Bison for Windows

- <http://gnuwin32.sourceforge.net/packages/bison.htm>