

Flex - Bison

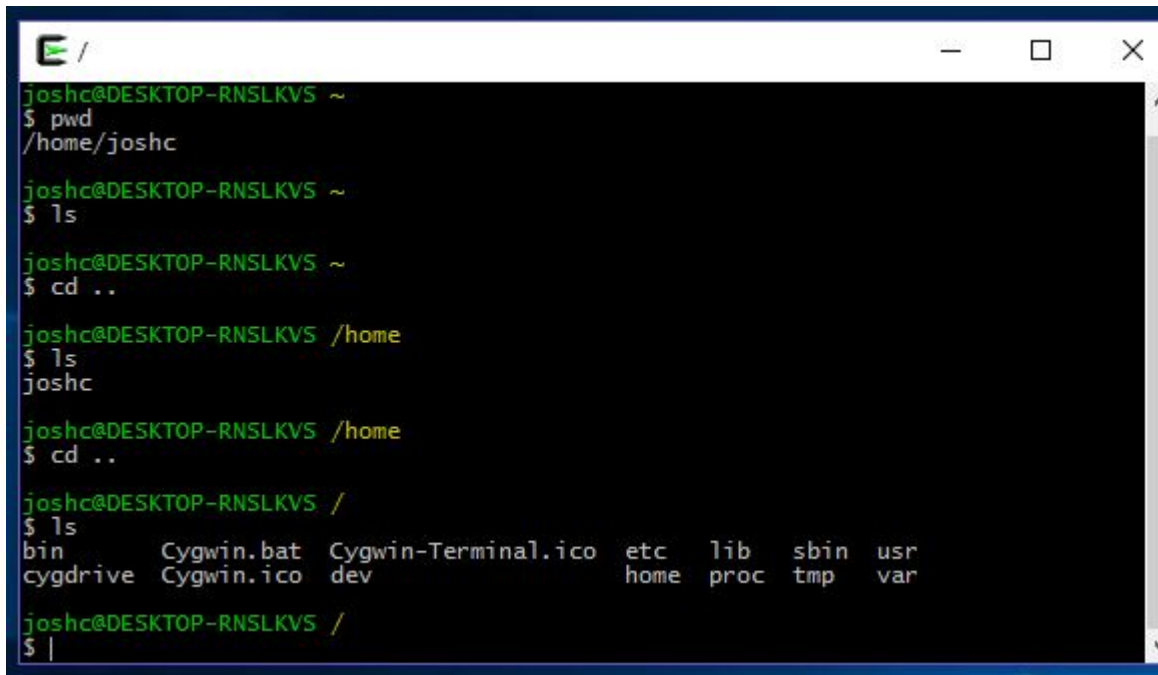
Αρχές Γλωσσών Προγραμματισμού και Μεταφραστών

Λειτουργικότητες και Προγραμματισμός
με Flex, Bison

Χρήσιμοι σύνδεσμοι

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

Για χρήστες με Windows - <http://www.cygwin.com/>



```
joshc@DESKTOP-RNSLKVS ~
$ pwd
/home/joshc

joshc@DESKTOP-RNSLKVS ~
$ ls

joshc@DESKTOP-RNSLKVS ~
$ cd ..

joshc@DESKTOP-RNSLKVS /home
$ ls
joshc

joshc@DESKTOP-RNSLKVS /home
$ cd ..

joshc@DESKTOP-RNSLKVS /
$ ls
bin          Cygwin.bat  Cygwin-Terminal.ico  etc  lib  sbin  usr
cygdrive     Cygwin.ico  dev                  home  proc tmp  var

joshc@DESKTOP-RNSLKVS /
$ |
```

Lexing – Parsing με μια ματιά...

- Λεκτική Ανάλυση (lexing)
- Συντακτική Ανάλυση (parsing)

alpha = beta + gamma ;

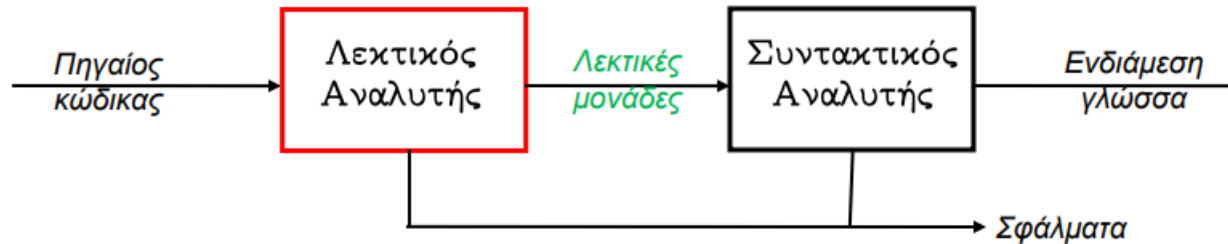
Ο **λεκτικός αναλυτής** θα αναγνωρίσει τα ακόλουθα tokens: alpha, equal_sign, beta, plus_sign, gamma, semicolon.

Ο **συντακτικός αναλυτής** θα αναγνωρίσει ότι το **beta + gamma** αποτελεί έκφραση και θα κάνει την ανάθεση στο **alpha**.



Λεκτική Ανάλυση

Φάση Λεκτικής Ανάλυσης (1)



Scanner (Λεκτικός Αναλυτής)

- **Είσοδος:** Το πηγαίο πρόγραμμα (μια συμβολοσειρά)
- **Έξοδος:** Λεκτικές μονάδες (tokens)

Token: αναπαρίσταται από ένα ζεύγος $\langle \text{tokenName}, \text{attribute} \rangle$

- **attribute:** αριθμός, συμβολοσειρά, δείκτης στον πίνακα συμβόλων, δομή κ.λπ. Το attribute είναι προαιρετικό.
- **tokenName:** ένα αφηρημένο σύμβολο που εκφράζει μια κλάση χαρακτήρων και το οποίο αποτελεί είσοδο στον parser.

π.χ. η ακολουθία χαρακτήρων $x = x + y ;$ γίνεται:

$\langle \text{id}, x \rangle \langle \text{eq}, = \rangle \langle \text{id}, x \rangle \langle \text{plus_op}, + \rangle \langle \text{id}, y \rangle \langle \text{sc}, ; \rangle$

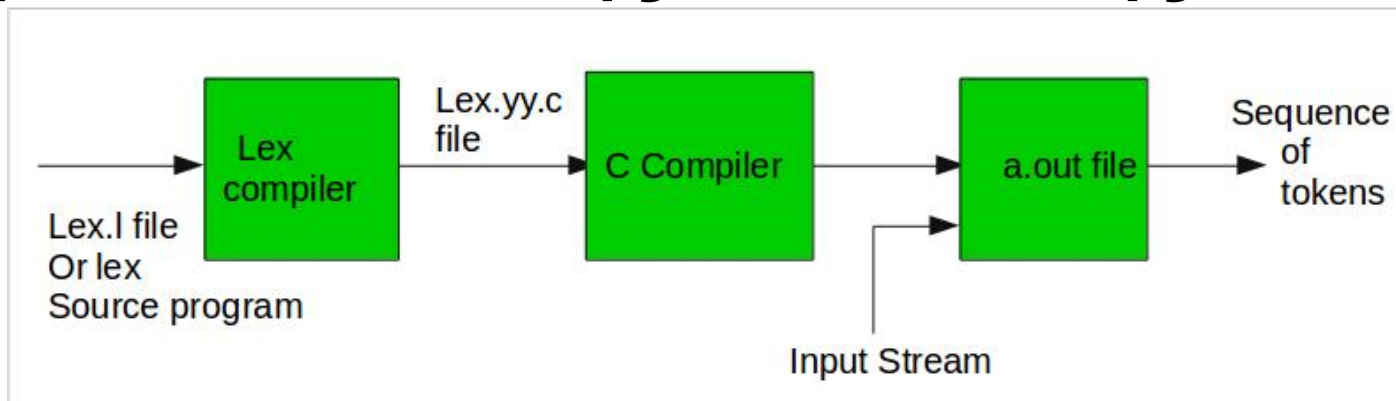
Φάση Λεκτικής Ανάλυσης (2)

Pattern: αποτελεί μια περιγραφή της μορφής που μπορούν να πάρουν τα λεξήματα (π.χ. μια κανονική έκφραση – περισσότερο στην συνέχεια...)

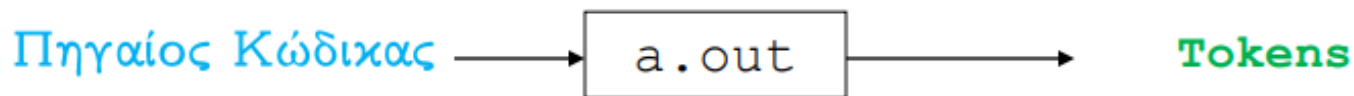
Lexeme (ή λέξημα): μια ακολουθία χαρακτήρων που ταιριάζει με ένα pattern (π.χ. το keyword *while*, ή το string “*Ceid*”)

Λεκτική ανάλυση: Διαβάζει χαρακτήρες από την είσοδο και τους ταιριάζει με **patterns**, παράγοντας **lexemes**, τα οποία αντιστοιχίζονται σε **tokens**, που αποτελούν την είσοδο του συντακτικού αναλυτή.

Βήματα Λεκτικής Ανάλυσης



1. Αρχικά, μέσω του FLEX compiler γίνεται η μετάφραση στο .l αρχείο (περισσότερα στη συνέχεια για τη δομή του...) - αν δεν υπάρχουν λάθη δεν θα εμφανιστεί κάποιο μήνυμα στο terminal. *flex wc.l*
2. Εάν η μετάφραση είναι επιτυχής, τότε έχει παραχθεί το πρόγραμμα του Λεκτικού Αναλυτή *lex.yy.c*, το οποίο θα κάνουμε compile μέσω gcc. *gcc lex.yy.c*
3. Αν το compiling γίνει με επιτυχία, έχει παραχθεί το εκτελέσιμο αρχείο *a.out*, το οποίο αντιστοιχεί στον τελικό λεκτικό αναλυτή. *Εκτέλεση ως ./a.out*



όπου **Πηγαίος Κώδικας** το δοκιμαστικό αρχείο που θέλετε να «περάσετε» από Λεκτική Ανάλυση

Κανονικές Εκφράσεις (Regular Expressions) (1)

Για την παραγωγή των tokens, ο αναλυτής θα αναζητήσει patterns χαρακτήρων στην συμβολοσειρά εισόδου.

Η περιγραφή των patterns αυτών γίνεται με τη χρήση Κ.Ε.

Κανονική Έκφραση	Ερμηνεία
x	Ο χαρακτήρας x
.	Οποιοσδήποτε χαρακτήρας, εκτός του χαρακτήρα αλλαγής γραμμής
“abc”	Η ακολουθία χαρακτήρων abc
[abc]	Οποιοσδήποτε από τους χαρακτήρες a, b, c
[a-z]	Οποιοσδήποτε από τους χαρακτήρες από a έως z
[^a-z]	Οποιοσδήποτε χαρακτήρας εκτός των a έως z
r*	Καμία ή περισσότερες επαναλήψεις της Κ.Ε. r
r+	Μία ή περισσότερες επαναλήψεις της Κ.Ε. r
r?	Μία ή καμία επανάληψη της Κ.Ε. r

Κανονικές Εκφράσεις (Regular Expressions) (2)

Κανονική Έκφραση	Ερμηνεία
$r\{i, j\}$	Από i έως j ($0 < i < j$) επαναλήψεις της r
rs	Ακολουθίες που προκύπτουν από παράθεση των Κ.Ε. r, s
(r)	Οι παρενθέσεις ορίζουν την εφαρμογή των τελεστών
$r s$	Ακολουθίες που ικανοποιούν είτε την Κ.Ε. r , είτε την s
r	Ικανοποιείται μόνο αν η r βρίσκεται στην αρχή της γραμμής
$r\$$	Ικανοποιείται μόνο αν η r βρίσκεται στο τέλος της γραμμής
$<<EOF>>$	Ικανοποιείται όταν συναντηθεί το EOF

Για τη χρήση των ειδικών χαρακτήρων $\$, |, ^$, κ.τ.λ. ως κυριολεκτικών πρέπει να προηγηθεί backslash \backslash

Το Flex μεταφράζει όλες τις RegEx ανάλογα με την είσοδο ταυτόχρονα, έτσι είναι το ίδιο γρήγορο για 100 patterns όσο και για ένα.

Flex input file

Δομή αρχείου εισόδου

declarations

%%

translation rules

%%

Auxiliary functions

Παράδειγμα λεκτικής ανάλυσης (1)

Για να κατανοήσουμε τη δομή του αρχείου, θα χρησιμοποιήσουμε το flex για να αναπαράγουμε το παράδειγμα του “Word Count”, που υπάρχει σε κάθε Unix σύστημα.

Δομή αρχείου:

- **Τμήμα 1: Declarations** (ορισμοί ονομάτων, κώδικας για τον παραγόμενο λεκτικό αναλυτή όπως δηλώσεις μακροεντολών, μεταβλητών και τύπων δεδομένων μέσα σε `{ %}`).
- **Τμήμα 2: Translations rules** (αποτελείται από κανόνες – κύριο τμήμα προγράμματος – όταν ικανοποιείται ένα pattern, ενεργοποιείται ο κανόνας και εκτελείται το action).
- **Τμήμα 3: Auxiliary functions**: (κώδικας ορισμένος από το χρήστη για εισαγωγή στο παραγόμενο πρόγραμμα C).

Παράδειγμα λεκτικής ανάλυσης (2)

```
%option noyywrap
%{
int chars = 0;
int words = 0;
int lines = 0;
%}
%%
[a-zA-Z]+ { words++; chars += strlen(yytext); }
\n { chars++; lines++; }
. { chars++; }
%%
main(int argc, char **argv)
{
yylex();
printf("%8d%8d%8d\n", lines, words, chars);
}
```

Τμήμα 1: **declarations** (ορισμοί)

- **Ορισμοί ονομάτων** που χρησιμοποιούνται ως συντομογραφίες Κ.Ε. και έχουν τη μορφή:

`name regular_expression`

π.χ.

DIGIT [0-9]

ID [a-z][a-z0-9]*

- **Κώδικας** προς συμπερίληψη στον παραγόμενο λεκτικό αναλυτή, συνήθως δηλώσεις μακροεντολών, μεταβλητών και τύπων δεδομένων μέσα σε ‘%{ %}’.

Παράδειγμα λεκτικής ανάλυσης (3)

Στο παράδειγμα του Word Count:

```
%{  
int chars = 0;  
int words = 0;  
int lines = 0;  
%}
```

Στο πρώτο τμήμα εισάγουμε τις δηλώσεις. Στην ενότητα δήλωσης, ο κώδικας εντός του `%{ %}` αντιγράφεται ως έχει στην έξοδο, για τον παραγόμενο λεκτικό αναλυτή. Στο παράδειγμα, απλώς ρυθμίζει μεταβλητές για γραμμές, λέξεις και χαρακτήρες.

Τμήμα 2: rules (κανόνες)

- Το τμήμα των κανόνων είναι το κύριο τμήμα του προγράμματος και αποτελείται από κανόνες της μορφής:

`pattern` `{action}`

- Κάθε `pattern` βρίσκεται στην αρχή μιας γραμμής, ακολουθούμενο από κώδικα C για εκτέλεση όταν γίνεται match το pattern.
- Όταν ικανοποιείται ένα `pattern`, ο κανόνας ενεργοποιείται και ο κώδικας C (`action`) εκτελείται.
- Ο κώδικας C μπορεί να είναι μία δήλωση ή πιθανώς ένα μπλοκ πολλαπλών εντολών σε αγκύλες, `{}`.
- Κάθε μοτίβο πρέπει να ξεκινά στην αρχή της γραμμής, αφού το flex θεωρεί ότι οποιαδήποτε γραμμή που ξεκινά με κενό διάστημα είναι κώδικας για εισαγωγή στο παραγόμενο πρόγραμμα C.
- Όταν ικανοποιούνται περισσότεροι κανόνες ταυτόχρονα, επιλέγεται αυτός που καταναλώνει περισσότερους χαρακτήρες. Εάν καταναλώνουν τον ίδιο αριθμό, επιλέγεται αυτός που έχει δηλωθεί πρώτος στη λίστα.

Παράδειγμα λεκτικής ανάλυσης (4)

Στο παράδειγμα του Word Count:

```
%%  
[a-zA-Z]+ { words++; chars += strlen(ytext); }  
\n { chars++; lines++; }  
. { chars++; }  
%%
```

Σε αυτό το πρόγραμμα, υπάρχουν μόνο τρία patterns. Το **πρώτο**, [a-zA-Z]+, ταιριάζει με μια λέξη.

Οι χαρακτήρες σε αγκύλες, γνωστοί ως κλάση χαρακτήρων, ταιριάζουν με οποιοδήποτε μεμονωμένο κεφαλαίο ή πεζό γράμμα και το σύμβολο **+** σηματοδοτεί το ταίριασμα με ένα ή περισσότερα από τα στοιχεία που εφαρμόζεται, που σημαίνει εδώ μια σειρά από γράμματα ή μια λέξη. Μέσω του action, ενημερώνεται ο αριθμός των λέξεων και των χαρακτήρων που εμφανίζονται.

Σε οποιαδήποτε ενέργεια, η μεταβλητή `ytext` περιέχει το κομμάτι του κειμένου που έχει ικανοποιήσει την Κ.Ε. (λέξημα). Στο παράδειγμα, η μεταβλητή που αφορά το πλήθος των χαρακτήρων προσαυξάνεται με το μήκος (# χαρακτήρων μέσω της `strlen`) του λεξιήματος.

Παράδειγμα λεκτικής ανάλυσης (5)

Στο παράδειγμα του Word Count:

```
%%  
[a-zA-Z]+ { words++; chars += strlen(yytext); }  
\n { chars++; lines++; }  
. { chars++; }  
%%
```

- Το **δεύτερο** pattern `\n` ταιριάζει ακριβώς με μια νέα γραμμή.
 - Η ενέργεια (action) ενημερώνει τον αριθμό γραμμών και χαρακτήρων.
- Το **τρίτο** pattern είναι μια **τελεία** `.`. Το pattern αυτό θα ενεργοποιηθεί για οποιονδήποτε άλλον χαρακτήρα, που δεν έχει ενεργοποιήσει κάποιο από τα 2 προηγούμενα.
 - Η ενέργεια (action) ενημερώνει τον αριθμό των χαρακτήρων.
- Για το συγκεκριμένο παράδειγμα αυτά είναι όλα τα patterns που χρειάζονται.

Τμήμα 3: *Auxiliary functions*

Στο τρίτο τμήμα είναι ο κώδικας C που αντιγράφεται στο αρχείο που θα παραχθεί (περιέχει συνήθως μικρές ρουτίνες ή συναρτήσεις που σχετίζονται με τον κώδικα στα actions του τμήματος 2).

Στο παράδειγμα του Word Count:

Ο κώδικας C στο τέλος είναι ένα κύριο πρόγραμμα που καλεί τη **yylex()**, τη lexer function που παράγεται από το flex και στη συνέχεια εκτυπώνει τα αποτελέσματα.

```
main(int argc, char **argv)
{
  yylex();
  printf("%8d%8d%8d\n", lines, words, chars);
}
```

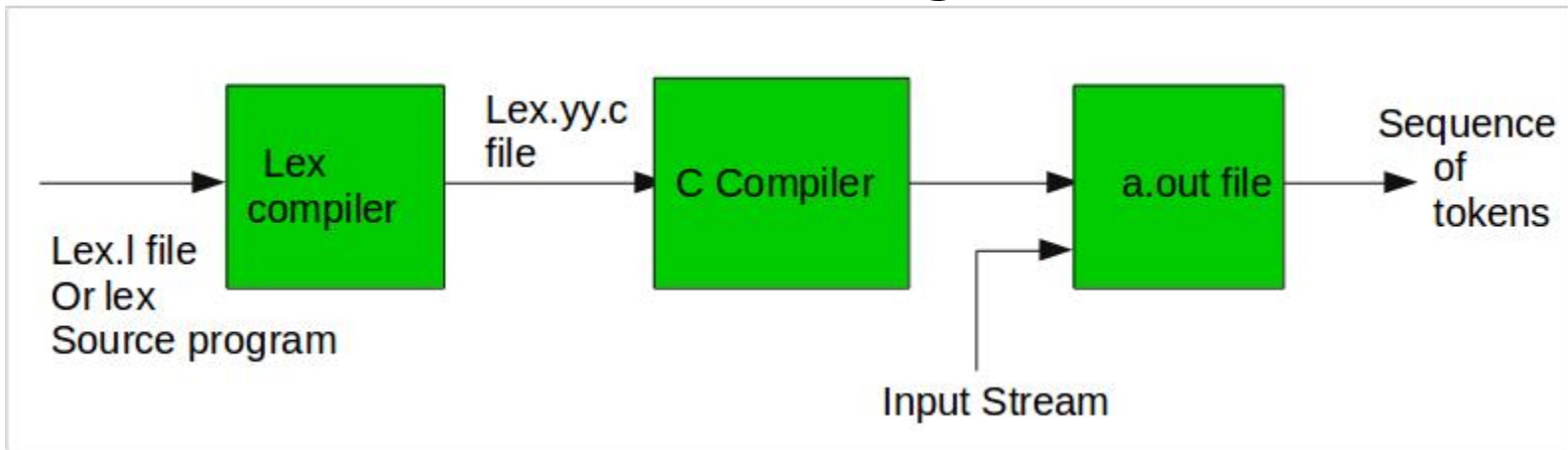
2^ο Παράδειγμα Λεκτικής Ανάλυσης (1)

```
/* recognize tokens for the calculator and print them out */  
%%  
"+" { printf("PLUS\n"); }  
"- " { printf("MINUS\n"); }  
"*" { printf("TIMES\n"); }  
"/" { printf("DIVIDE\n"); }  
"|" { printf("ABS\n"); }  
[0-9]+ { printf("NUMBER %s\n", yytext); }  
\n { printf("NEWLINE\n"); }  
[ \t] { }  
. { printf("Character %s\n", yytext); }  
%%
```

2^ο Παράδειγμα Λεκτικής Ανάλυσης (2)

- Τα πρώτα πέντε patterns είναι τελεστές, γραμμένοι ως συμβολοσειρές, και τα αντίστοιχα actions, απλώς τυπώνουν ένα μήνυμα που αναφέρει τι έχει ταιριαστεί.
- Το έκτο pattern ταιριάζει έναν ακέραιο. Το μοτίβο με αγκύλες [0-9] αντιστοιχεί σε οποιοδήποτε μονοψήφιο και το σύμβολο + συμβολίζει την αντιστοίχιση ενός ή περισσότερων από το προηγούμενο στοιχείο, δηλαδή μια συμβολοσειρά ενός ή περισσότερων ψηφίων. Το αντίστοιχο action τυπώνει τη συμβολοσειρά που έχει ταιριάξει.
- Το έβδομο pattern ταιριάζει μια αλλαγή γραμμής, που συμβολίζεται από τη συνήθη ακολουθία \n.
- Το όγδοο pattern αφορά τα whitespaces. Ταιριάζει με οποιοδήποτε κενό διάστημα ή tab (\t). Το action είναι κενό, άρα μόλις «ενεργοποιηθεί» το συγκεκριμένο pattern δεν θα γίνει κάποια ενέργεια.
- Το τελικό pattern ενεργοποιείται για οτιδήποτε δεν ενεργοποίησε κάποιο από τα προηγούμενα. Μέσω του action του εκτυπώνεται κατάλληλο μήνυμα.

Βήματα μετάφρασης αρχείου calc.l



```
$ flex calc.l
```

```
$ gcc lex.yy.c
```

```
$ ./a.out
```

1. Πρώτα, μέσω του flex κάνουμε τη μετάφραση στο πρόγραμμα.
2. Στη συνέχεια, κάνουμε compile το lex.yy.c, δηλαδή το πρόγραμμα C που δημιουργήθηκε από το προηγούμενο βήμα, με το gcc.
3. Παράγεται το εκτελέσιμο αρχείο a.out, το οποίο εκτελούμε γράφοντας ./a.out.

Input: 12+34

NUMBER 12

PLUS

NUMBER 34

NEWLINE

Input: 5 6 / 7q

NUMBER 5

NUMBER 6

DIVIDE

NUMBER 7

character q

NEWLINE

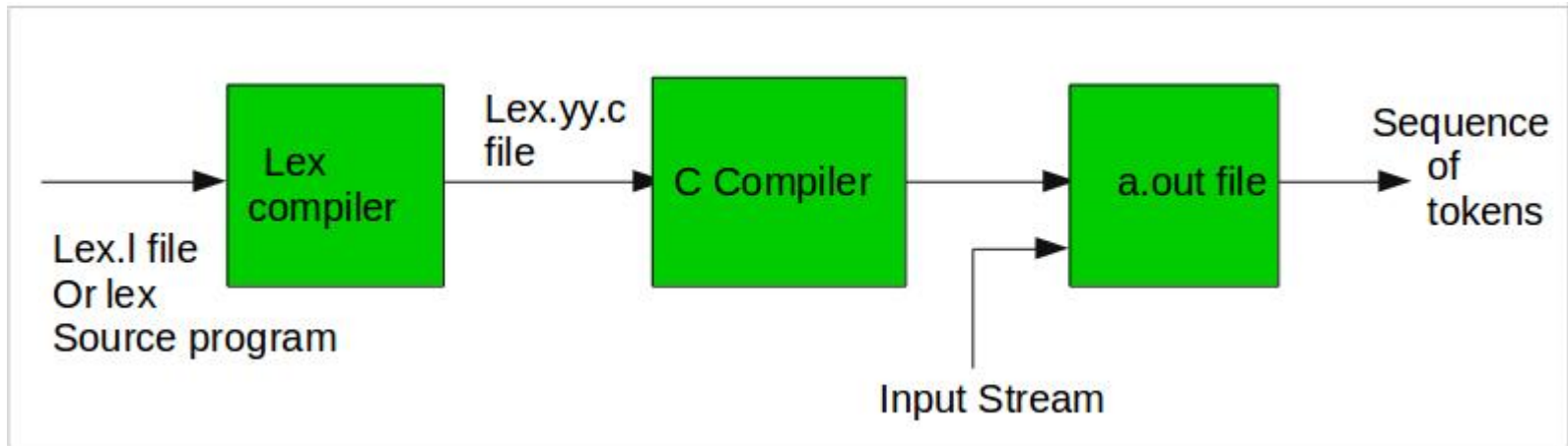
Tokens

```
/* recognize tokens for the calculator and print them out */
%{
enum ytokentype {
NUMBER = 258,
ADD = 259,
SUB = 260,
MUL = 261,
DIV = 262,
ABS = 263,
EOL = 264
};
int yylval;
}%
%%
"+" { return ADD; }
"-" { return SUB; }
"*" { return MUL; }
"/" { return DIV; }
|" { return ABS; }
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
\n { return EOL; }
[\t] { /* ignore whitespace */ }
. { printf("Mystery character %c\n", *yytext); }
%%
main(int argc, char **argv)
{
int tok;
while(tok = yylex()) {
printf("%d", tok);
if(tok == NUMBER) printf(" = %d\n", yylval);
else printf("\n");
}
```

Όταν ένας λεκτικός αναλυτικής επιστρέφει μια ροή από tokens, το καθένα έχει δύο μέρη, το token και την τιμή του token.

Το token γενικά είναι ένας ακέραιος αριθμός.

Βήματα μετάφρασης αρχείου calc2.l



```
$ flex calc2.l
```

```
$ gcc lex.yy.c
```

```
$ ./a.out
```

1. Πρώτα, μέσω του flex κάνουμε τη μετάφραση στο πρόγραμμα.
2. Στη συνέχεια, κάνουμε compile το lex.yy.c, δηλαδή το πρόγραμμα C που δημιουργήθηκε από το προηγούμενο βήμα, με το gcc.
3. Παράγεται το εκτελέσιμο αρχείο a.out, το οποίο εκτελούμε γράφοντας ./a.out.

Input: a / 34 + |45

Mystery character a

262

258 = 34

259

263

258 = 45

264

Καταστάσεις στο Flex

- Το Flex υποστηρίζει την υπό συνθήκη ενεργοποίηση ενός κανόνα μέσω της έννοιας της **κατάστασης**.
- Αρχικά το Flex βρίσκεται στην κατάσταση *INITIAL*.
- Οι καταστάσεις δηλώνονται στο τμήμα ορισμών – 1^ο τμήμα.
(π.χ. `%s A_STATE`)
- Στο τμήμα των κανόνων, ένας υπό συνθήκη ενεργός κανόνας γράφεται ως εξής:
 - `<A_STATE>pattern action`
ή ως:
 - `<A_STATE, B_STATE>pattern action`
όπου ο κανόνας είναι ενεργός σε οποιαδήποτε από τις δυο καταστάσεις.
- Η μετάβαση σε μια κατάσταση γίνεται γράφοντας στο τμήμα `action` κάποιου κανόνα: `BEGIN(A_STATE);`
- Η επιστροφή στην αρχική κατάσταση γίνεται γράφοντας `BEGIN(INITIAL);`

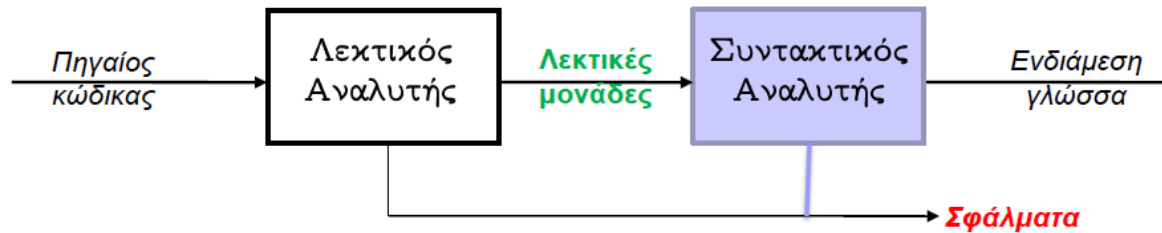
Επιπλέον στοιχεία περιβάλλοντος Flex

- Μερικές από τις κυριότερες διαθέσιμες μεταβλητές/συναρτήσεις του περιβάλλοντος:
 - `char *yytext`: περιέχει το κομμάτι του κειμένου που έχει ικανοποιήσει την κανονική έκφραση (λέξημα)
 - `int yyleng`: ένας ακέραιος που δηλώνει το μέγεθος του `yytext`
 - `FILE *yyin`: το προκαθορισμένο αρχείο εισόδου
 - `int yylex()`: η παραγόμενη συνάρτηση λεκτικής ανάλυσης. Επιστρέφει το αναγνωριστικό της λεκτικής μονάδας που διαβάζει.
- Περισσότερα μπορείτε να βρείτε και στο Manual του Flex (Eclass)



Συντακτική Ανάλυση

Συντακτικός Αναλυτής Bison



Parser (Συντακτικός Αναλυτής)

- Ελέγχει την ακολουθία των **λεκτικών μονάδων** (*tokens*) για γραμματική ορθότητα.

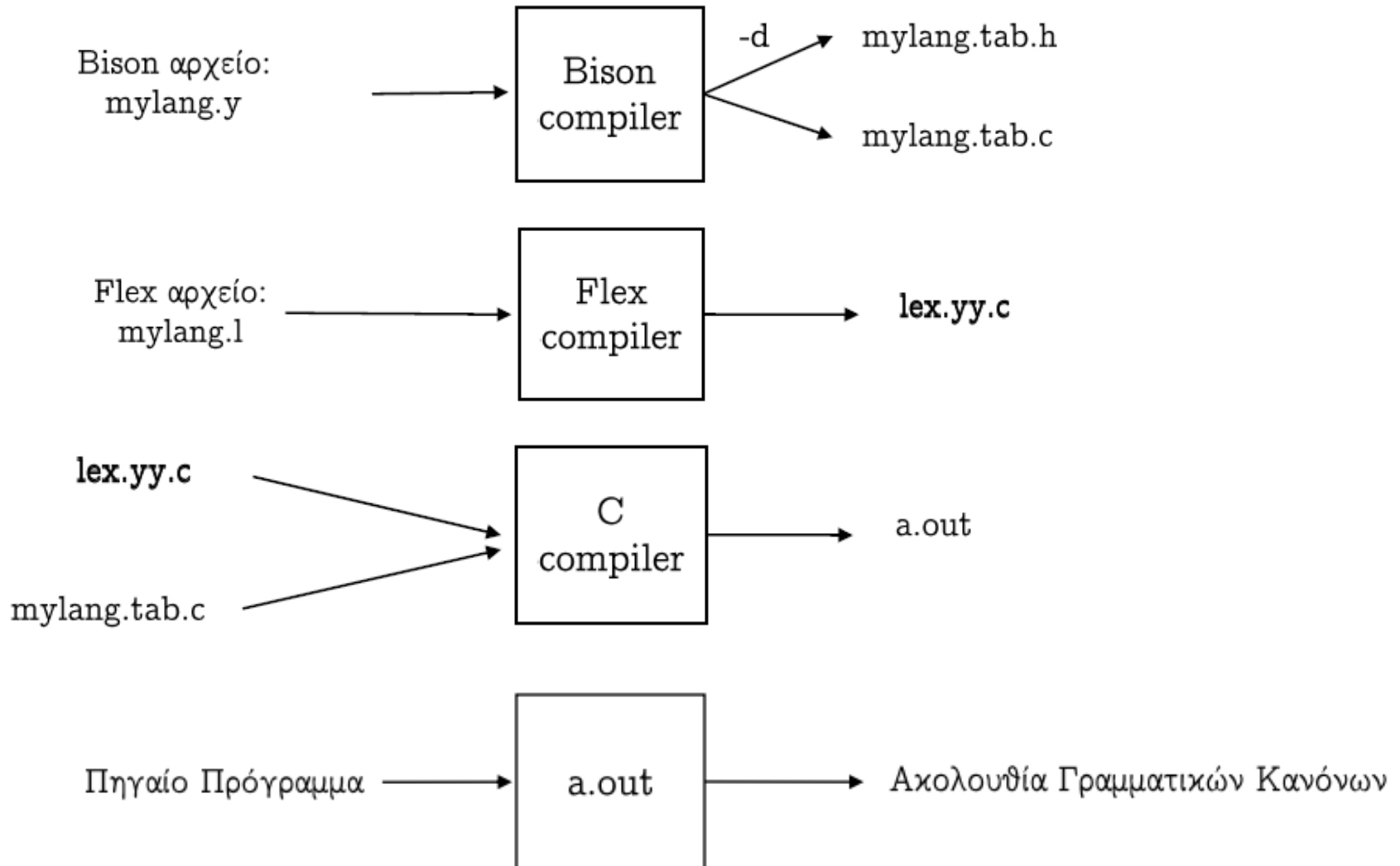
Προσοχή η συντακτική ανάλυση είναι υπολογιστικά αρκετά δυσκολότερη από τη λεκτική – Επομένως είναι προτιμότερο να συμπεριλάβουμε όσο περισσότερους κανόνες μπορούμε στο λεκτικό αναλυτή (π.χ. χειρισμός whitespaces).

Bison: Βελτίωση του εργαλείου yacc του Unix.

Yacc: γεννήτρια συντακτικών αναλυτών -> δέχεται μια γραμματική χωρίς συμφραζόμενα (LALR(1)) και παράγει έναν συντακτικό αναλυτή σε C.

Η παραγόμενη συνάρτηση `yyparse` αναγνωρίζει τις συμβολοσειρές εισόδου, κατασκευάζει το συντακτικό δέντρο και εκτελεί τις ενέργειες που περιγράφονται στο πρόγραμμα.

Διαδικασία Παραγωγής Συντακτικού Αναλυτή



Παρατηρήσεις

- Το header file που παράγεται από τον Bison Compiler (*.tab.h) πρέπει να συμπεριληφθεί στο αρχείο του flex (*.l).
Με αυτό τον τρόπο επιτυγχάνεται η επικοινωνία μεταξύ του λεκτικού και συντακτικού αναλυτή.
- Δηλώσεις μεταβλητών που θα χρειαστούν και στους δύο αναλυτές θα πρέπει να οριστούν με τη δήλωση extern (π.χ. yyinval).

Δομή Προγράμματος Bison

%{

Κώδικας C

(μακροεντολές, τύποι δεδομένων, δηλώσεις μεταβλητών και συναρτήσεων)

%}

Δηλώσεις Bison

%%

Κανόνες παραγωγής γραμματικής

%%

Κώδικας C

(υλοποίηση συναρτήσεων, main())

Κανόνες Παραγωγής Γραμματικής (1)

- Για την περιγραφή της γραμματικής της γλώσσας θα χρησιμοποιηθούν **κανόνες παραγωγής** διατυπωμένοι σε **BNF** (*Backus-Naur Form*).
 - Κάθε γραμμή είναι ένας κανόνας για τη δημιουργία ενός branch στο parse tree.
- Η γενική μορφή των κανόνων είναι: **αριστερό μέλος**: **δεξιό μέλος**;
- Το **αριστερό μέλος** είναι ένα μη τερματικό σύμβολο.
- Το **δεξιό μέλος** μπορεί να περιέχει μηδέν ή περισσότερα τερματικά και μη τερματικά σύμβολα και εντολές C σε { }.

```
Program : {count=0;} block _list
          {printf(" Counted %d block(s) \n",count);}
          ;
block _list : /* nothing */
            | block _list block {count++;}
            ;
block      : BEGIN block _list END
            ;
```

Κανόνες Παραγωγής Γραμματικής (2)

Παράδειγμα με **actions**:

```
statement: NAME '=' expression
          | expression { printf("= %d\n", $1); }
          ;
expression: NUMBER '*' NUMBER { $$ = $1 * $3 }
          | NUMBER           { $$ = $1 }
          ;
```

- \$1, \$3 αναφέρονται σε τιμές του δεξιού μέλους. Το \$\$ θέτει τιμή στο αριστερό μέλος. Η έκφραση $$$ = \$1 * \$3$ θέτει την τιμή της (expression) σε $\text{NUMBER}(\$1) * \text{NUMBER}(\$3)$.
- Το **action** ενός κανόνα εκτελείται όταν ο parser ελαττώσει (reduce) τον κανόνα.
- Ο λεκτικός αναλυτής θα πρέπει να έχει επιστρέψει μια τιμή μέσω της `yylval`.

Δηλώσεις Bison (1)

- Τμήμα Ορισμών
 - Τα Tokens της γραμματικής θα πρέπει να οριστούν.
 - Ορίζονται με `%token TOKEN_NAME`
 - Από τα ορισμένα tokens, ο Bison θα δημιουργήσει ένα κατάλληλο header file (include στο αρχείο του flex)
 - Χαρακτήρες σε απλά εισαγωγικά μπορούν να χρησιμοποιηθούν σαν tokens χωρίς να δηλωθούν, π.χ. '+', '=' κ.λπ.

Δηλώσεις Bison (2)

```
%nonassoc '=' '<' '>'
```

```
%left '+' '-'
```

```
%left '*' '/' TK_div, TK_mod
```

- Δηλώσεις τελεστών της γλώσσας και της προτεραιότητας και προσεταιριστικότητάς τους.
- Ισχύουν τα εξής:
 - Τα tokens που εμφανίζονται στην ίδια γραμμή έχουν την ίδια προτεραιότητα.
 - Η προτεραιότητα αυξάνεται από πάνω προς τα κάτω
 - Τα '+' και '-' έχουν μικρότερη προτεραιότητα από τα '*' και '/'
 - Η έκφραση $a+b+c$ υπολογίζεται ως $(a+b)+c$
 - Το %nonassoc χρησιμοποιείται για τελεστές που δεν μπορούν να συνδυαστούν μεταξύ τους, π.χ. το '<'
 - Χρησιμοποιούνται για αποφυγή της παραγωγής μιας έκφρασης με περισσότερους του ενός τρόπους.

Τμήμα κώδικα C

Σε αυτό το τμήμα δηλώνονται συναρτήσεις όπως `yerror`, `main` κ.λπ.

```
yerror(char *errmsg)
{
    fprintf(stderr, "%s\n", errmsg);
}
```

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

Βασικές συναρτήσεις Bison

- `int yyparse();` → Η συνάρτηση αυτή υλοποιεί τον Συντακτικό Αναλυτή. Επιστρέφει 0 αν αναγνωρισθεί η συμβολοσειρά εισόδου ή την τιμή 1 σε περίπτωση συντακτικού λάθους.
- `void yyerror(const char *message);` → Η συνάρτηση αυτή πρέπει να υλοποιείται υποχρεωτικά στο μεταπρόγραμμα του bison. Καλείται αυτόματα όταν εντοπιστεί κάποιο συντακτικό σφάλμα.
- `int yylex();` → Η συγκεκριμένη συνάρτηση, όπως αναφέρθηκε, υλοποιεί τον Λεκτικό Αναλυτή. Καλείται από το συντακτικό αναλυτή κάθε φορά που πρέπει να διαβαστεί μια νέα λεκτική μονάδα από τη συμβολοσειρά εισόδου και να εκλεχθεί. Με λίγα λόγια, η `yyparse` κάθε φορά καλεί την `yylex` προς αναγνώριση κάποιας λεκτικής μονάδας.
- Μπορείτε να βρείτε περισσότερα στο εγχειρίδιο του Bison (Eclass)

Παράδειγμα αξιοποίησης Flex-Bison

Ας υποθέσουμε ότι θέλουμε να γράψουμε έναν μεταφραστή για μια απλή γλώσσα αριθμητικών εκφράσεων.

Η είσοδος θα πρέπει να διαβάζεται από ένα αρχείο *input* και τα αποτελέσματα της εξόδου να γράφονται σε ένα αρχείο *output*.

Π.χ. για είσοδο $7+5*8$, στο αρχείο *output* θα πρέπει να γραφεί 47

Βήμα 1: σχεδιάζουμε το bnf της γλώσσας

- Στην περίπτωση μας θεωρούμε πως κάθε πρόγραμμα της γλώσσας πρέπει να αποτελείται από μόνο μία αριθμητική έκφραση, ενώ οι μόνες επιτρεπόμενες πράξεις είναι: '+', '*' μεταξύ ακεραίων.
- $\langle \text{πρόγραμμα} \rangle ::= \langle \text{έκφραση} \rangle$
 $\langle \text{έκφραση} \rangle ::= \text{ΑΚΕΡΑΙΟΣ}$
| $\langle \text{έκφραση} \rangle + \langle \text{έκφραση} \rangle$
| $\langle \text{έκφραση} \rangle * \langle \text{έκφραση} \rangle$

Βήμα 2: μεταφέρουμε το bnf στο Bison

```
■ program: expr { fprintf(yyout, "%i\n", $1); }  
           ;  
expr: INT  
     | expr '+' expr      { $$ = $1 + $3; }  
     | expr '*' expr      { $$ = $1 * $3; }  
           ;
```

- Ωστόσο η γλώσσα μας δεν είναι LL(1) . Συνεπώς θα πρέπει είτε να τροποποιήσουμε τη γραμματική ώστε να αποφύγουμε την αριστερή αναδρομή, είτε να ορίσουμε προτεραιότητες στους τελεστές ώστε να γνωρίζει ο bison πώς να αναλύσει τη συμβολοσειρά εισόδου.
- ΠΡΟΣΟΧΗ: πολλές φορές η αλλαγή της γραμματικής είναι μονόδρομος (πχ αριστερή παραγοντοποίηση,...)

Βήμα 3: Δηλώσεις & Προτεραιότητες

- `%token INT`
`%left '+'`
`%left '*'`
- Προσοχή το `%left '*'` τοποθετείται δεύτερο διότι ο πολλαπλασιασμός έχει μεγαλύτερη προτεραιότητα από την πρόσθεση.

Βήμα 4: Δημιουργία λεκτικού αναλυτή (1)

- Θα πρέπει έπειτα να γράψουμε έναν λεκτικό αναλυτή που να μετατρέπει τη συμβολοσειρά εισόδου σε μία ακολουθία από token. Οπότε έχουμε:

- ```
digit [0-9]
num {digit}+
%%
{num} { yyval = atoi(yytext); return INT; }
"+" { return '+'; }
"*" { return '*'; }
"\n" { return '\n'; }
. ;
%%
```
- Θέτει τιμή για χρήση στα actions
- Επιστρέφει το token που αναγνώρισε

## Βήμα 4: Δημιουργία λεκτικού αναλυτή (2)

Επίσης, θα πρέπει να συμπεριλάβουμε το header file που παράχθηκε από το Bison, καθώς και άλλες απαραίτητες/χρήσιμες βιβλιοθήκες:

```
%{
#include "y.tab.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
%}
```

## Βήμα 5: main, yyerror συναρτήσεις (1)

- Στο αρχείο του bison θα πρέπει επίσης να ορισθούν οι συναρτήσεις main και yyerror
- Επίσης θα πρέπει να ορίσουμε τις μεταβλητές που χειρίζονται τα αρχεία εισόδου/εξόδου
- ```
%{  
#include <stdio.h>  
#include <math.h>  
void yyerror(char *);  
extern FILE *yyin;  
extern FILE *yyout;  
%}
```

Βήμα 5: main, yyerror συναρτήσεις (2)

- ```
void yyerror(char *s) {
 fprintf(stderr, "%s\n", s);
}
```
- ```
int main ( int argc, char **argv ) {  
    ++argv; --argc;  
    if ( argc > 0 )  
        yyin = fopen( argv[0], "r" );  
    else  
        yyin = stdin;  
    yyout = fopen ( "output", "w" );  
    yyparse ();  
    return 0;  
}
```

Παράδειγμα συντακτικής ανάλυσης (1/3)

Κώδικας σε C

```
/* calculator */  
%{  
    #include <stdio.h>  
%}
```

Δηλώσεις Bison

```
/* declare tokens */  
%token NUMBER  
%token ADD SUB MUL DIV ABS
```

- Στο πρώτο τμήμα, οι δηλώσεις περιλαμβάνουν κώδικα C που θα αντιγραφεί στην αρχή του δημιουργημένου C parser, και πάλι περικλείεται στο `%{` και `%}`.
- Ακολουθούν οι δηλώσεις για τα token: `%token ONOMA_TOKEN`. Κατά συνθήκη, τα διακριτικά έχουν κεφαλαία ονόματα.

Παράδειγμα συντακτικής ανάλυσης (2/3)

Κανόνες Γραμματικής

start: addition | subtraction | product | division

addition : NUMBER ADD NUMBER | addition ADD NUMBER;

subtraction : NUMBER SUB NUMBER | subtraction SUB NUMBER ;

product: NUMBER MUL NUMBER | product MUL NUMBER;

division: NUMBER DIV NUMBER | division DIV NUMBER;

- Με βάση την παραπάνω γραμματική το σύμβολο **start** δηλώνει το αρχικό σύμβολο της Γραμματικής.
 - Το μη-τερματικό σύμβολο **start** μπορεί να ορίσει κάποιο από τους κανόνες
addition | subtraction | product | division
 - Στη περίπτωση π.χ. του **addition** θα μπορούσε να γίνει πρόσθεση ενός αριθμού με έναν άλλον ή πρόσθεση κάποιου αριθμού σε κάποιο ήδη άθροισμα:
 - α) με χρήση του κανόνα **NUMBER ADD NUMBER** που χρησιμοποιεί τερματικά σύμβολα – tokens και συγκεκριμένα τα NUMBER, ADD
 - β) με χρήση του κανόνα **addition ADD NUMBER** που χρησιμοποιεί το μη-τερματικό σύμβολο **addition** και τα τερματικά σύμβολα (tokens) NUMBER, ADD
- Ομοίως, τα παραπάνω ισχύουν και για τις άλλες πράξεις!

Ενδεικτικές **επιτρεπτές** πράξεις: $5+5$, $3*3*3$ κτλ.

Ενδεικτικές **μη-επιτρεπτές** πράξεις: $5+5/2$, $3*3-4$ κτλ.

Παράδειγμα συντακτικής ανάλυσης (3/3)

Στο τελευταίο τμήμα μπορεί να οριστεί εκ νέου **κώδικας C** για την χρήση της κύριας συνάρτησης `main` καθώς και άλλων βασικών συναρτήσεων όπως της `yyerror` κτλ.

Παρακάτω παρουσιάζεται ένα ενδεικτικό παράδειγμα της δομής:

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

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

Διαχείριση σφαλμάτων στο Bison

- Προκαθορισμένη συμπεριφορά κατά την ανίχνευση σφάλματος:
 - Κλήση `yyerror()` για την εκτύπωση διαγνωστικού μηνύματος
 - **Τερματισμός** εκτέλεσης
- `%error-verbose`: οδηγία για πιο κατατοπιστικά μηνύματα
- Η προκαθορισμένη συμπεριφορά δεν υλοποιεί Error Recovery -> Μπορείτε να βρείτε περισσότερες σχετικές πληροφορίες στο manual