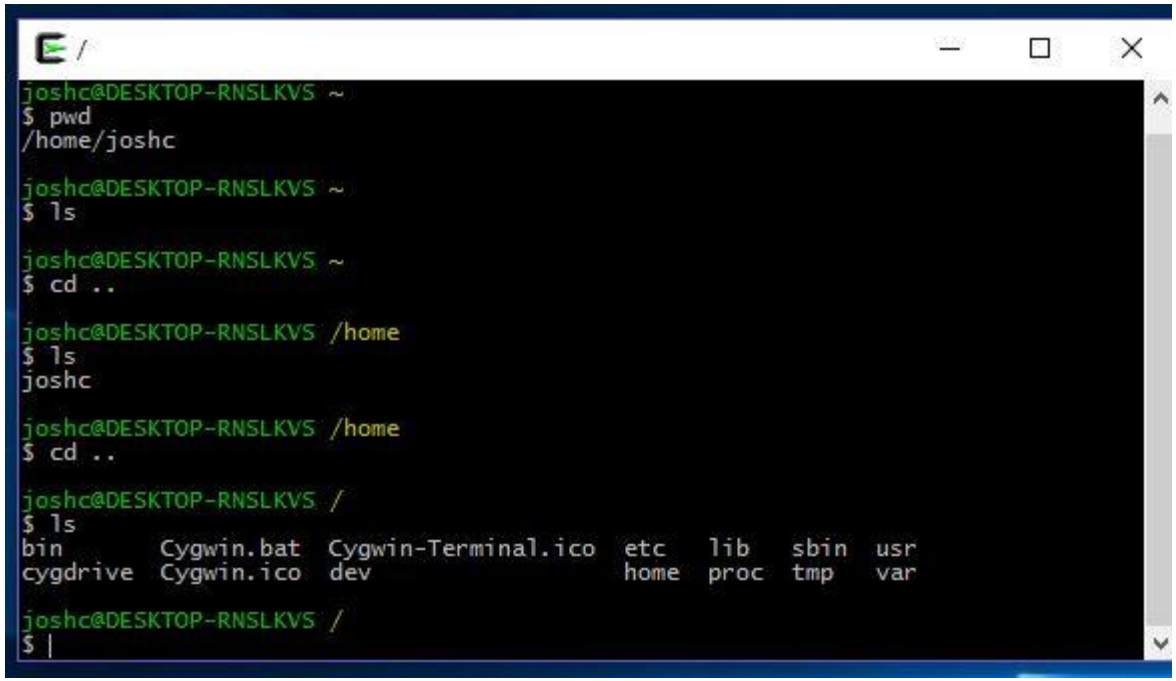
Flex -Bison

Αρχές Γλωσσών Προγραμματισμού και Μεταφραστών

Χρήσιμοι σύνδεσμοι

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

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



Lexing - Parsing

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

alpha = beta + gamma ;

Ο λεκτικός αναλυτής θα διαχωρίσει 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.**

Αναλυτές & Regular Expressions

Οι αναλυτές αναζητούν patterns χαρακτήρων

Για παράδειγμα στη C:

- Integer Constant: ένα ή παραπάνω ψηφία

- Variable Name: ένα γράμμα και στη συνέχεια ακολουθεί κανένα ή ένα και παραπάνω γράμματα ή και αριθμοί

Ένας τρόπος περιγραφής αυτών των patterns είναι μέσω regular expressions (regex – regexp) που τις γνωρίζουμε ήδη από το ed ή το vi στο Unix.

Αναλυτές & Regular Expressions

Ένα πρόγραμμα flex αποτελείται βασικά από μια λίστα regexps με οδηγίες σχετικά με τι πρέπει να γίνει όταν η είσοδος ταιριάζει με οποιαδήποτε από αυτές (actions)

Ένας αναλυτής γενικά έπειτα από το input, το ταιριάζει με όλες τις regexps επιλέγοντας το κατάλληλο action. Το Flex μεταφράζει όλα τα regexps ανάλογα με την είσοδο ταυτόχρονα, έτσι είναι τόσο γρήγορο για 100 patterns όσο και για ένα.

Παράδειγμα λεκτικής ανάλυσης

Θα χρησιμοποιήσουμε το flex για να αναπαράγουμε το γνωστό παράδειγμα που υπάρχει σε κάθε Unix σύστημα “Word Count”.

Δομή αρχείου:

- **Τμήμα 1:** Declarations (ορισμοί ονομάτων, κώδικας για 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

```
%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);
}
```

Παράδειγμα λεκτικής ανάλυσης

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

Στο πρώτο τμήμα εισάγουμε τις δηλώσεις. Στην ενότητα δήλωσης, ο κώδικας εντός του % {και%} αντιγράφεται μέσω λέξεων κοντά στην αρχή του δημιουργημένου αρχείου πηγής C. Σε αυτήν την περίπτωση απλώς ρυθμίζει μεταβλητές για γραμμές, λέξεις και χαρακτήρες.

Παράδειγμα λεκτικής ανάλυσης

Στο δεύτερο τμήμα εισάγουμε μια λίστα με patterns και actions. Κάθε pattern βρίσκεται στην αρχή μιας γραμμής, ακολουθούμενο από κώδικα C για εκτέλεση όταν γίνεται match το pattern.

Ο κώδικας C μπορεί να είναι μία δήλωση ή πιθανώς ένα μπλοκ πολλαπλών γραμμών 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.

Παράδειγμα λεκτικής ανάλυσης

```
%%
[a-zA-Z]+ { words++; chars += strlen(yytext); }
\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 + σημαίνει να ταιριάζει με ένα ή περισσότερα από τα προηγούμενα, που σημαίνει εδώ μια σειρά από γράμματα ή μια λέξη. Ο κωδικός δράσης ενημερώνει τον αριθμό των λέξεων και των χαρακτήρων που εμφανίζονται. Σε οποιαδήποτε ενέργεια, η μεταβλητή yytext ρυθμίζεται ώστε να δείχνει στο κείμενο εισαγωγής που ταιριάζει ακριβώς με το pattern. Σε αυτήν την περίπτωση, το μόνο που μας ενδιαφέρει είναι πόσους χαρακτήρες ήταν έτσι ώστε να μπορούμε να ενημερώσουμε κατάλληλα τον αριθμό των χαρακτήρων.

Παράδειγμα λεκτικής ανάλυσης

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

- Το **δεύτερο** pattern **\n** ταιριάζει ακριβώς με μια νέα γραμμή.

Η ενέργεια ενημερώνει τον αριθμό γραμμών και χαρακτήρων.

- Το **τρίτο** pattern είναι μια **τελεία .** η οποία είναι regex-ese για οποιονδήποτε χαρακτήρα (Είναι παρόμοιο με το **?** σε ένα shell script).

Η ενέργεια ενημερώνει τον αριθμό των χαρακτήρων.

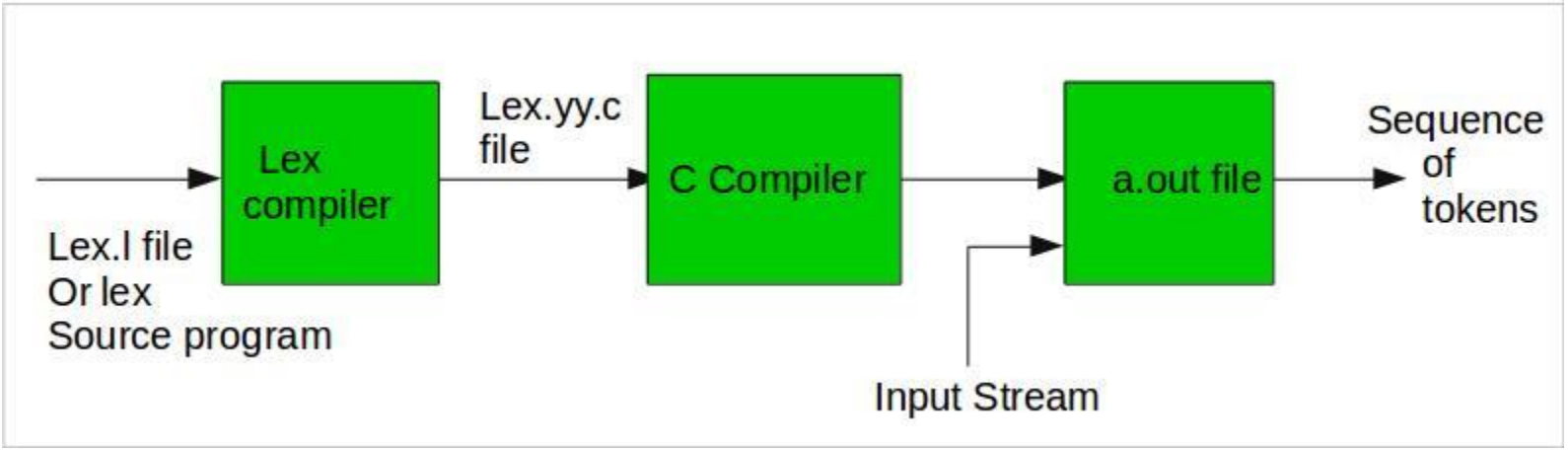
Για το συγκεκριμένο παράδειγμα αυτά είναι όλα τα patterns που χρειαζόμαστε.

Παράδειγμα λεκτικής ανάλυσης

- 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).
- 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);
}
```

Βήματα εκτέλεσης αρχείου flex



$ flex wc.l

$ gcc lex.yy.c $

./a.out

The boy stood on the burning deck
shelling peanuts by the peck
^D
2 12 63

Πρώτα μέσω του flex κάνουμε τη μετάφραση στο πρόγραμμά (προσοχή αν δεν υπάρχουν λάθη δεν θα μας ενημερώσει με κάποιο output).
Στη συνέχεια, κάνουμε compile το lex.yy.c, το πρόγραμμα C lex.yy.c που δημιουργήθηκε από το προηγούμενο βήμα με το gcc.
Παράγεται το εκτελέσιμο αρχείο a.out που το εκτελούμε με ./a.out

Παράδειγμα 2 λεκτικής ανάλυσης

```
/* 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); }
%%
```

Παράδειγμα 2 λεκτικής ανάλυσης

- Τα πρώτα πέντε patterns είναι τελεστές, γραμμένοι ως συμβολοσειρές που αναφέρονται, και τα actions, προς το παρόν, απλά τυπώνουν ένα μήνυμα που λέει τι ταιριάζει.

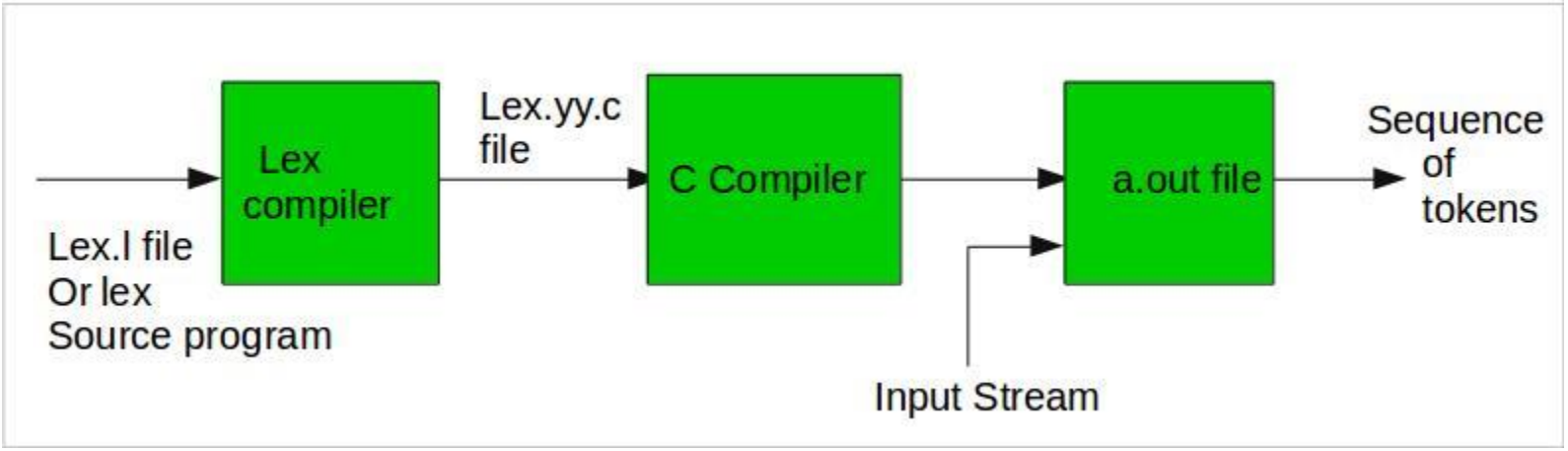
- Το έκτο pattern ταιριάζει έναν ακέραιο. Το μοτίβο με αγκύλες [0-9] αντιστοιχεί σε οποιοδήποτε μονοψήφιο και το ακόλουθο σύμβολο + σημαίνει αντιστοίχιση ενός ή περισσότερων από το προηγούμενο στοιχείο, που σημαίνει μια συμβολοσειρά ενός ή περισσότερων ψηφίων. Το action τυπώνει τη συμβολοσειρά που ταιριάζει.

- Το έβδομο pattern ταιριάζει μια αλλαγή γραμμής, που αντιπροσωπεύεται από τη συνήθη ακολουθία στη C **\n**.

- Το όγδοο pattern αφορά το whitespace. Ταιριάζει με οποιοδήποτε κενό διάστημα ή tab (**\t**) και δεν έχει καθόλου κώδικα στο action.

- Το τελικό pattern είναι για να ταιριάζει με οτιδήποτε δεν είχαν τα άλλα patterns. Το action του είναι να τυπώνει το κατάλληλο μήνυμα.

Βήματα εκτέλεσης αρχείου flex



$ flex calc.l

$ gcc lex.yy.c $

./a.out

Πρώτα μέσω του flex κάνουμε τη μετάφραση στο πρόγραμμά (προσοχή αν δεν υπάρχουν λάθη δεν θα μας ενημερώσει με κάποιο output).
Στη συνέχεια, κάνουμε compile το lex.yy.c, το πρόγραμμα C lex.yy.c που δημιουργήθηκε από το προηγούμενο βήμα με το gcc.
Παράγεται το εκτελέσιμο αρχείο a.out που το εκτελούμε με ./a.out.

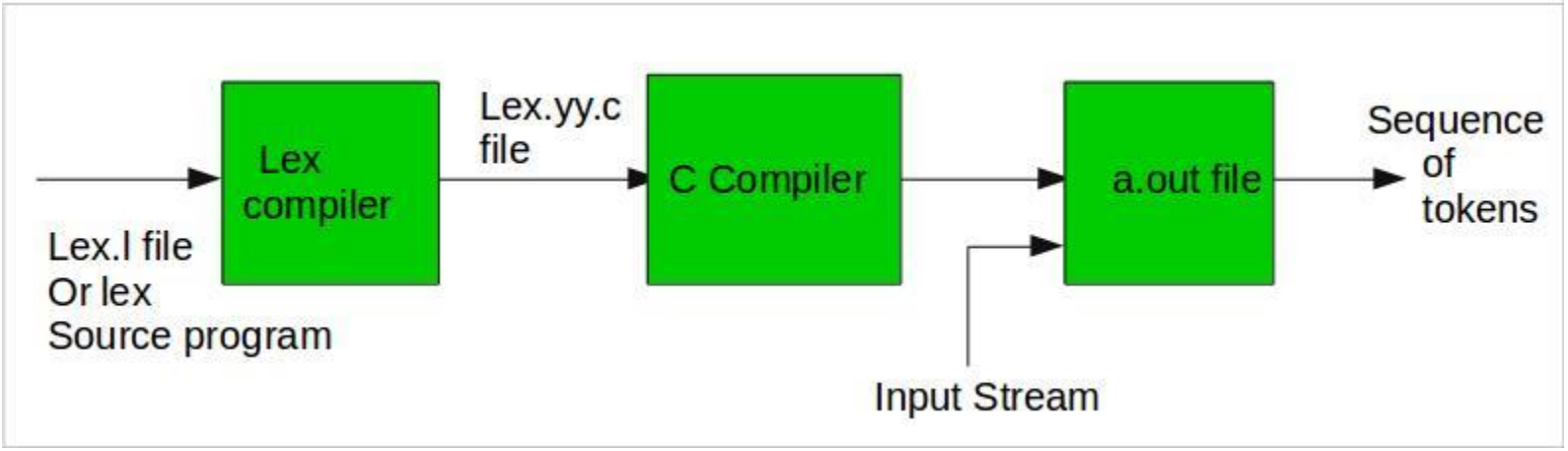
12+34
NUMBER 12
PLUS
NUMBER 34
NEWLINE
5 6 / 7q
NUMBER 5
NUMBER 6
DIVIDE
NUMBER 7
character q
NEWLINE

Tokens

```
/* recognize tokens for the calculator and print them out */
%{
enum yytokentype {
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 εκτέλεσης αρχείου flex



$ flex calc2.l

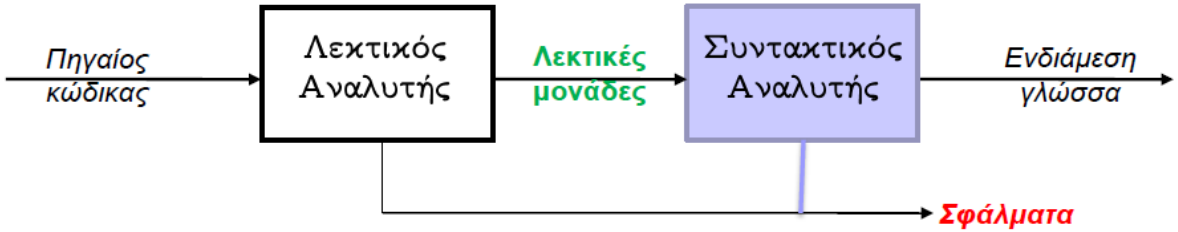
$ gcc lex.yy.c $

./a.out

Πρώτα μέσω του flex κάνουμε τη μετάφραση στο πρόγραμμά (προσοχή αν δεν υπάρχουν λάθη δεν θα μας ενημερώσει με κάποιο output).
Στη συνέχεια, κάνουμε compile το lex.yy.c, το πρόγραμμα C lex.yy.c που δημιουργήθηκε από το προηγούμενο βήμα με το gcc.
Παράγεται το εκτελέσιμο αρχείο a.out που το εκτελούμε με ./a.out.

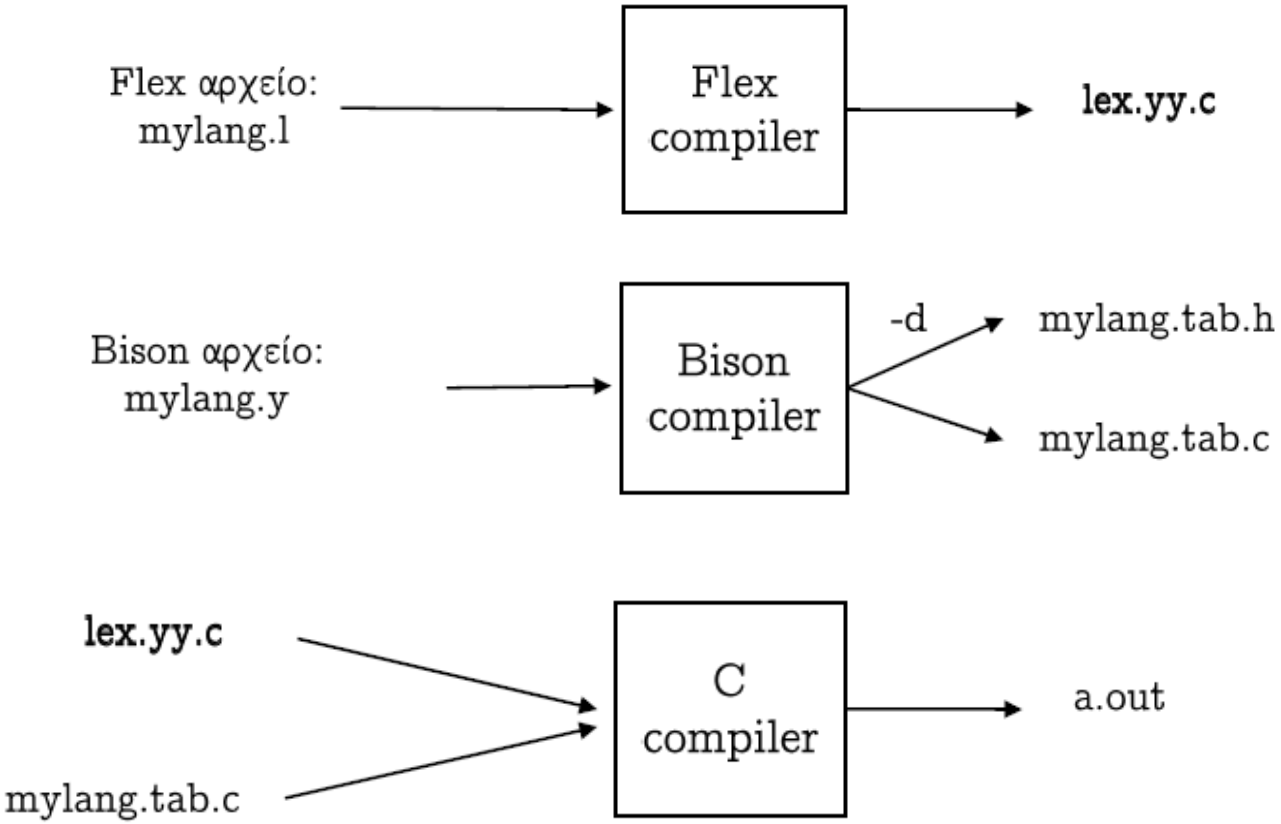
a / 34 + |45
Mystery character a
262
258 = 34
259
263
258 = 45
264

Συντακτικός Αναλυτής Bison



- Για ένα πρόγραμμα συντακτικής ανάλυσης, χρειαζόμαστε τρόπο για να περιγράψουμε τους κανόνες που χρησιμοποιεί ο αναλυτής για να μετατρέψει μια ακολουθία tokens σε ένα parse tree.
- **Προσοχή η συντακτική ανάλυση είναι υπολογιστικά αρκετά δυσκολότερη από τη λεκτική – Επομένως είναι προτιμότερο να συμπεριλάβουμε όσο περισσότερους κανόνες μπορούμε στο λεκτικό αναλυτή.**
- Για τη συγκεκριμένη διαδικασία χρησιμοποιούμε τη γραμματική 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 μη τερματικό σύμβολο.
- Το δεξιό μέλος μπορεί να περιέχει μηδέν ή περισσότερα τερματικά και μη τερματικά σύμβολα και εντολές C σε { }.

Διαδικασία Παραγωγής Compiler



Δηλώσεις Bison

- Το header file που παράγει ο bison πρέπει να συμπεριληφθεί στο αρχείο του flex. 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.

Δομή Προγράμματος Bison

```
%{
    Κώδικας C
    (μακροεντολές, τύποι δεδομένων, δηλώσεις μεταβλητών και
  συναρτήσεων)
%}
    Δηλώσεις Bison
%%
    Κανόνες παραγωγής γραμματικής
%%
    Κώδικας C
    (υλοποίηση συναρτήσεων, main() )
```

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.

- int 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 ορίζει την λεκτική ανάλυση. Θα πρέπει όμως να αναφερθεί πως καλείται από την yyparse 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.

Βήμα 1: σχεδιάζουμε το bnf της γλώσσας

- Στην περίπτωσή μας θεωρούμε πως κάθε πρόγραμμα της γλώσσας πρέπει να αποτελείται από μόνο μία αριθμητική έκφραση, ενώ οι μόνες επιτρεπόμενες πράξεις είναι: ‘+’, ‘*’ μεταξύ ακεραίων.
- <πρόγραμμα> ::= <έκφραση>
  <έκφραση> ::= ΑΚΕΡΑΙΟΣ
  | <έκφραση> + <έκφραση>
  | <έκφραση> * <έκφραση>

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 γραματικής 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: main, yyerror συναρτήσεις (1/2)

- 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 4: main, yyerror συναρτήσεις (2/2)

```
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**, που ορίζουν στο bison τα ονόματα των tokens. 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
  b) 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 στάδιο θα πρέπει να οριστεί εκ νέου **κώδικας C** για την χρήση της κύριας συνάρτησης main καθώς και άλλων βασικών συναρτήσεων όπως yyerror κτλ.

Παρακάτω παρουσιάζεται ένα ενδεικτικό παράδειγμα της δομής:

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

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