Πανεπιστήμιο Πελοποννήσου
Σχολή Θετικών Επιστήμων και Τεχνολογίας
Τμήμα Επιστήμης και Τεχνολογίας Υπολογιστών

Μεταπτυχιακή Εργασία

«Μεθοδολογία Υλοποίησης Εμπρόσθιων Τμημάτων μεταγλωττιστή με τα Εργαλεία Flex και Bison»

Σταματία Σωτηράκου
ΑΜ:2010025

Επιβλέπων Καθηγητής
Κ^{ος} Κωνσταντίνος Μασσέλος

<Τρίπολη, Ιούλιος, 2012>

Ευχαριστίες

Θα θελα να ευχαριστήσω θερμά τον καθηγητή μου Κο Γρηγόριο Δημητρουλάκο κυρίως για την υπομονή που έκανε κατά τη διάρκεια της εκπόνησης της διπλωματικής εργασίας μου και για την πολύτιμη βοήθεια και καθοδήγηση του, για την επίλυση διαφόρων θεμάτων.

Σταματία Σωτηράκου

Πίνακας Περιεχομένων

Ευρετήριο Figures

Εκτεταμένη Περίληψη

Η παρούσα διπλωματική εργασία μελετά την μεθοδολογία υλοποίησης προγραμμάτων εμπρόσθιων τμημάτων μεταγλωττιστή με την χρήση των εργαλείων Flex και Bison.

Στο 1° κεφάλαιο, γίνεται αναφορά στην περιγραφή ενός λεξικογραφικού αναλυτή με την χρήση του Εργαλείου Flex. Αναλυτικότερα, αναφέρεται στην βασική λειτουργία ενός λεκτικού αναλυτή ,στη συμπεριφορά ,στη δομή και στη σύνταξης ενός προγράμματος εμπρόσθιου τμήματος μεταγλωττιστή.

Ακολουθεί το 2° κεφάλαιο, στο οποίο γίνεται η περιγραφή της ανάλυσης του συντακτικού αναλυτή με την χρήση του εργαλείου Bison. Στην συνέχεια, γίνεται αναφορά στη δομή και στη σύνταξη ενός συντακτικού αναλυτή, καθώς και στη συμπεριφορά ενός ολοκληρωμένου προγράμματος Bison.

Το 3° κεφάλαιο, επεξηγεί την συνεργασία των δύο εργαλείων για την παραγωγή και την μετάφραση ενός προγράμματος στο επίπεδο της λεκτικής και συντακτικής ανάλυσης. Καθώς επίσης γίνεται αναφορά στην συμπεριφορά των εργαλείων Flex και Bison στο προγραμματιστικό περιβάλλον του Microsoft Visual Studio.

Το τελευταίο Κεφάλαιο αναφέρεται στην ολοκληρωμένη σύνταξη προγραμμάτων με την χρήση των εργαλείων, με την παράθεση παραδειγμάτων. Συγκεκριμένα γίνεται ανάπτυξη λεκτικών και συντακτικών αναλυτών που αναγνωρίζουν και μεταγλωττίζουν ένα calculator.

Extended abstract

This thesis studies the methodology of implementing front-compiler programs using the Flex and Bison tools.

The first chapter refers to the description of a lexicographical analyzer using the Flex tool. More specifically, it refers to the basic function of the lexical analyzer, the behavior, the structure and the syntax of a front-compiler program and it also refers to the behavior of a complete Bison program.

The third chapter explains the cooperation between the two tools for the production and translation of a program

The second chapter first explains the description of the analysis of the parser, using Bison tool, continued by a reference to the structure and to the syntax of a parser analyzer using the Bison tool at the level of lexical and syntactic analysis. Finally it refers to the behavior of the flex and bison tools in the programming environment of Microsoft Visual studio.

The last chapter, through examples, demonstrates the complete syntax of programs using the two tools. Specifically, the development of lexical and parsing analyzers which identify and compile a calculator program is being shown.

ΚΕΦΑΛΑΙΟ 1^0

Εισαγωγή στο Flex

1.1 Τι είναι το Flex (fast lexical analyzer generator)

Το Flex ένα ελεύθερο εναλλακτικό λογισμικό του Lex(κλασική γεννήτρια παραγωγής λεκτικών αναλυτών που συνεργάζεται με την γεννήτρια συντακτικών αναλυτών yacc.) .Συνεργάζεται με την γεννήτρια συντακτικών αναλυτών Bison για την δημιουργία μεταγλωττιστών. Αρχικά το flex γράφτηκε σε C γλώσσα προγραμματισμού από τον Vern Paxson το 1987.Σήμερα υποστηρίζει την δημιουργία του κώδικα σε C και C++.

Το Flex είναι ένα εργαλείο που δημιουργεί προγράμματα που χειρίζονται δομημένες εισόδους .Είναι ένα γνήσιο εργαλείο σαρωτών : προγράμματα που αναγνωρίζουν λεξικογραφικά μοτίβα(patterns) σε ένα κείμενο και παράγει λεξικογραφικούς αναλυτές σε γλώσσες C/C++ ,που αποτελούν την πρώτη φάση της ανάλυσης της μεταγλώττισης.

Το Flex διαβάζει αρχεία εισόδου που έχουν δοθεί από το χρήστη ή μια standard είσοδο, στην περίπτωση που δεν έχει οριστεί όνομα αρχείου εισόδου, ώστε να γίνει η περιγραφή του λεξικογραφικού αναλυτή που θα παραχθεί. Η περιγραφή αποτελείται από κανονικές εκφράσεις (regular expressions) και κώδικα C που ονομάζονται κανόνες. Στην συνέχεια το Flex παράγει ως έξοδο ένα αρχείο C, `lex.yy.c' που καθορίζει την ρουτίνα yylex(),συνάρτηση λεξικής ανάλυσης .Το αρχείο αυτό μεταγλωττίζεται και αφού συνδεθεί με τη βιβλιοθήκη -lfl δημιουργεί ένα εκτελέσιμο το οποίο όταν εκτελείται ελέγχει αν υπάρχουν κανονικές εκφράσεις. Στην περίπτωση που βρει μια κανονική έκφραση εκτελεί τον αντίστοιχο κώδικα C.

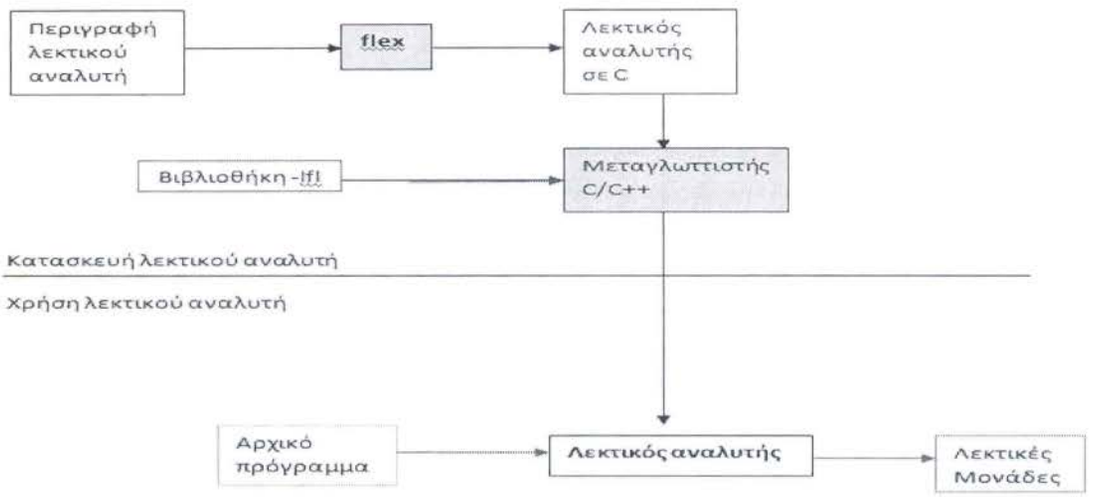

Figure 1.1 Λειτουργία ενός λεξικογραφικού αναλυτή.

Για την περιγραφή ενός λεξικογραφικού αναλυτή στο Flex χρησιμοποιούνται οι Κανονικές εκφράσεις (regular expressions ή regex) που περιγράφουν λεξικογραφικά πρότυπα μιας γλώσσας και οι Actions, δηλαδή κώδικας σε μια γλώσσα προγραμματισμού που εκτελείται όταν αναγνωριστεί μια κανονική έκφραση(στην οποία αντιστοιχεί η Action)

1.2Λεκτική ανάλυση

Η βασική λειτουργία ενός τυπικού λεκτικού αναλυτή είναι να διαβάζει από την είσοδό του έναν–έναν τους χαρακτήρες της εισόδου, να αναγνωρίζει ακολουθίες χαρακτήρων και να παράγει στην έξοδό του Λεκτικές Μονάδες (tokens). Ως παράδειγμα, η ακολουθία των παρακάτω γραμμάτων:

alpha =beta +gamma;

Ο scanner χωρίζει την παραπάνω πρόταση σε **tokens**: alpha, beta, gamma, ίσον (=), και (+), ερωτηματικό (;) . Στη συνέχεια, το πρόγραμμα ανάλυσης καθορίζει ότι οι beta + gamma είναι μια έκφραση, και ότι η έκφραση έχει εκχωρηθεί στο alpha(συντακτική ανάλυση).

alpha, beta, gamma	*τύπου προσδιοριστή (identifier)*
=, +	*τύπου πράξης*
;	*τύπου στίξης*

Υπάρχει τυπικός τρόπος για τον ορισμό κανόνων που θα χρησιμοποιηθούν για την αναγνώριση πρότυπων χαρακτήρων. Οι κανόνες αυτοί ονομάζονται Κανονικές Εκφράσεις (Regular Expressions). Με τη βοήθειά τους, μπορούμε να ορίσουμε τις ακολουθίες χαρακτήρων που ανήκουν στη γλώσσα, δηλαδή τις Λεκτικές Μονάδες τις οποίες θέλουμε να αναγνωρίζει ο λεκτικός αναλυτής.

1.3Γραμματική κανονικών εκφράσεων και σάρωση

Οι σαρωτές δουλεύουν κυρίως για την αναγνώριση προτύπων χαρακτήρων. Για να καταλάβουμε ακριβώς τα πρότυπα χαρακτήρων μπορούμε να παρατηρήσουμε ένα πρόγραμμα στην C.Για παράδειγμα, ένας ακέραιος αποτελείται από ένα ή περισσότερα ψηφία, το όνομα μίας μεταβλητής αποτελείται από ένα γράμμα ακολουθούμενο από μηδέν ή περισσότερα γράμματα ή ψηφία , και οι διάφοροι τελεστές είναι μεμονωμένοι χαρακτήρες ή ζεύγη χαρακτήρων. Ένας απλός τρόπος για να περιγράψουμε τα πρότυπα είναι οι κανονικές εκφράσεις. Ένα πρόγραμμα Flex αποτελείται από μια λίστα κανονικών εκφράσεων μαζί με οδηγίες που καθορίζουν τι πρέπει να κάνουν ,όταν ταιριαστεί η είσοδο με μια ή περισσότερες από τις κανονικές εκφράσεις.

Στην πραγματικότητα μία κανονική έκφραση χρησιμοποιείται για να περιγράψει μία κανονική γλώσσα. Η κανονική έκφραση αναπαριστά πρότυπα(συμβολοσειρές) που χρησιμοποιούνται σε μία μεταγλώσσα , μια γλώσσα που χρησιμοποιούμε για να περιγράψουμε ένα πρότυπο που θέλουμε να ταιριαστεί. Η μεταγλώσσα χρησιμοποιεί κανονικούς χαρακτήρες κειμένου.

Οι χαρακτήρες με ειδική σημασία στις κανονικές εκφράσεις είναι :

[ ] Αναγνωρίζει ένα μοναδικό χαρακτήρα εκτός από τους \, - και ^. Όταν θέλουμε να ορίσουμε ένα πρότυπο που μπορεί να ταυτιστεί με έναν από τους χαρακτήρες μιας ομάδας χαρακτήρων, τότε ορίζουμε μία *κλάση χαρακτήρων*. Μία κλάση χαρακτήρων ορίζεται από χαρακτήρες που περικλείονται μέσα σε ένα συνδυασμό αριστερής και δεξιάς αγκύλης. Έτσι, η συμβολοσειρά [abc] μπορεί να ταυτιστεί με το χαρακτήρα a ή με το χαρακτήρα b ή με το χαρακτήρα c. Ο χαρακτήρας – εκπροσωπεύει το διάστημα.

Για παράδειγμα το [0-9] σημαίνει όλους τους αριθμητικούς χαρακτήρες [0123456789], ενώ το [a-z] σημαίνει όλους τους μικρούς χαρακτήρες της αγγλικής αλφαβήτου. Οι σειρές χαρακτήρων ερμηνεύονται σε σχέση με τον χαρακτήρα κωδικοποίησης που χρησιμοποιείται, έτσι το εύρος [A-z] με ASCII κωδικοποίηση χαρακτήρων θα ταιριάζουν με όλους τους πεζούς και κεφαλαίους χαρακτήρες.

. Ταυτίζει όλους τους χαρακτήρες, εκτός από το τέλος γραμμής (\n).

"" Καθορίζουν την αρχή ή το τέλος μιας ακολουθίας χαρακτήρων. Για παράδειγμα το πρότυπο "abcd" αντιστοιχεί στην ακολουθία χαρακτήρων abcd. Τα εισαγωγικά χρησιμεύουν και στην περίπτωση που θέλουμε να δηλώσουμε κάθε άλλο μη αλφαριθμητικό χαρακτήρα ως χαρακτήρα. Μια ακόμα χρήση των (") είναι όταν θέλουμε να περιγράψουμε πρότυπο που περιέχει τον κενό χαρακτήρα ή τον οριζόντιο στηλογνώμονα (tab), τα οποία κανονικά στο Flex χωρίζουν τις Λεκτικές Μονάδες. Μέσα σε ζευγάρι διπλών εισαγωγικών (" ") μπορούν να χρησιμοποιηθούν και οι συμβολισμοί της C για ειδικούς χαρακτήρες, π.χ. καινούρια γραμμή \n, οριζόντιος στηλογνώμονας \t, πισωγύρισμα \b.

^ Χρησιμοποιείται όταν θέλουμε να ταιριάσουμε τον πρώτο χαρακτήρα της κανονικής έκφραση στην αρχή της γραμμής. Για παράδειγμα το πρότυπο ^a ταιριάζει την γραμμή που αρχίζει με τον χαρακτήρα a. Όταν όμως ο χαρακτήρας ^ περικλείεται σε αγκύλες η σημασία του αλλάζει, δηλώνει άρνηση. Το πρότυπο [^abc] ταιριάζει όλους τους υπόλοιπους χαρακτήρες, συμπεριλαμβανομένων και των ειδικών χαρακτήρων, εκτός από τους a,b,c.

**$** Ταιριάζει τον τελευταίο χαρακτήρα της κανονικής έκφρασης στο τέλος της γραμμής .Για παράδειγμα η έκφραση ab$ μπορεί να ταυτιστεί με την ακολουθία ab που βρίσκεται στο τέλος της γραμμής(δηλαδή ακολουθείται αλλαγή γραμμής \n).

\ Όταν θέλουμε να ταιριάξουμε ειδικούς χαρακτήρες όπως οι τελεστές σαν απλούς χαρακτήρες ,χρησιμοποιούμε τον χαρακτήρα διαφυγής(\). Για παράδειγμα το πρότυπο abc \+ αντιστοιχεί στην ακολουθία abc+

□ Όταν θέλουμε να ορίσουμε πρότυπα που να ταυτίζονται με 0 ή περισσότερες εμφανίσεις των αντίστοιχων εκφράσεων .Για παράδειγμα το a* σημαίνει κανένα ή περισσότερα a.

+ Όταν θέλουμε να ορίσουμε πρότυπα που να ταυτίζονται με τουλάχιστον μία ή περισσότερες εμφανίσεις των αντίστοιχων εκφράσεων .Για παράδειγμα το a+ σημαίνει ένα ή περισσότερα a. Αντίστοιχα το πρότυπο [a-z]+ ταιριάζει όλες τις συμβολοσειρές που αποτελούνται από μικρά γράμματα.

? Όταν θέλουμε να ορίσουμε πρότυπα που να ταυτίζονται με 0 ή μια εμφανίσεις των αντίστοιχων εκφράσεων. Ο τελεστής ? σημαίνει ότι το σύμβολο που προηγείται είναι προαιρετικό σε μια έκφραση. Για παράδειγμα το πρότυπο ab?c μπορεί ταιριαστεί με τις συμβολοσειρές ab, abc.

| Ορίζει την έννοια της εναλλαγής. Για παράδειγμα το πρότυπο ab|cd μπορεί να ταυτιστεί με το ab ή με το cd

( ) Οι παρενθέσεις τις χρησιμοποιούμε όταν θέλουμε να εκφράσουμε ομαδικότητα. Για παράδειγμα το πρότυπο (abc) θα ταυτιστεί με την συμβολοσειρά abc .Συνήθως τις χρησιμοποιούμε για να περιγράψουμε πολύπλοκες εκφράσεις ,όπως (abc) + που ταιρίαζει συμβολοσειρές που περιέχουν την ακολουθία abc μια φορά ή περισσότερες φορές .

/ Ορίζει ακολουθούμενα συμφραζόμενα. Για παράδειγμα η έκφραση ab/cd μπορεί να ταυτιστεί με την συμβολοσειρά ab εφόσον ακολουθείται από την συμβολοσειρά cd.

r{i} Αναγνωρίζει i ή περισσότερες εμφανίσεις της έκφρασης r. Για παράδειγμα το πρότυπο r{2} σημαίνει ότι η έκφραση r πρέπει να εμφανίζεται δύο φορές. Αντίστοιχα το πρότυπο r{2,3} αναγνωρίζει συμβολοσειρές που περιέχουν δύο ή τρεις φορές την έκφραση r.

EOF Δηλώνει το τέλος του αρχείου

1.3.1 Ταίριασμα και επίλυση ασαφών προτύπων

Όταν ο παραγόμενος σαρωτής τρέχει, αναλύει την είσοδο που του έχει δοθεί και ανιχνεύει τα πρότυπα που ταιριάζουν με την είσοδο. Στην περίπτωση που καθοριστεί ένα ταίριασμα προτύπου με την είσοδο ,η ενέργεια που αντιστοιχεί στο πρότυπο εκτελείται. Στην συνέχεια το υπόλοιπο μέρος της εισόδου ελέγχεται για άλλο ταίριασμα.

Ένα δεν βρεθεί κανένα πρότυπο να ταιριαστεί με την είσοδο, ο σαρωτής αντιγράφει την είσοδο στην έξοδο. Όταν βρεθεί ένα η περισσότερα πρότυπα που φαίνεται να ταιριάζουν την ίδια είσοδο, τα πράγματα περιπλέκονται και δημιουργούνται ασάφειες. Ο Flex επιλύει την ασάφεια με δύο απλούς κανόνες:

- Ταιριάζει το μακρύτερο πρόθεμα χαρακτήρων κάθε φορά που ο σαρωτής διαβάζει την είσοδο.
- Εάν ταιριάζουν τον ίδιο πρόθεμα χαρακτήρων ,επιλέγεται το πρότυπο που έχει οριστεί πρώτο.

Παράδειγμα

Έστω οι εξής κανόνες:

```
case      {return KEYWORD;}
[a-z]+    {return IDENTIFIER;}
```

Ως είσοδο έχουμε τη λέξη cases.Για το ταίριασμα θα προτιμηθεί ο δεύτερος κανόνας, γιατί ταυτίζεται με είσοδο μήκους 5, σε αντίθεση με τον πρώτο που ταυτίζεται με είσοδο μήκους 4. Αντίθετα, αν η είσοδος ήταν case, θα ταυτιζόταν με τον πρώτο κανόνα, γιατί αυτός έχει οριστεί πρώτος.

1.4Δομή ενός προγράμματος Flex

Ένα πρόγραμμα Flex αποτελείται από τρία τμήματα: το τμήμα των ορισμών(definition section),το τμήμα των κανόνων(rules section) και το τμήμα κώδικα χρήστη(user subroutines)

... definition section ...

%%

... rules section ...

%%

... user subroutines section ...

Τα τμήματα μεταξύ τους χωρίζονται με γραμμές αποτελούμενες από διπλά σύμβολα του ποσοστού(%%).Ένα πρόγραμμα Flex για να θεωρηθεί σωστό πρέπει να περιλαμβάνει οπωσδήποτε τα δύο πρώτα τμήματα ,ακόμα και στην περίπτωση που το ένα τμήμα είναι άδειο. Το τρίτο τμήμα θεωρείται προαιρετικό και μπορεί να παραληφθεί.

1.4.1 Τμήμα Ορισμών(Definition)

Στο τμήμα αυτό περιλαμβάνονται οι κανονικοί ορισμοί που χαρακτηρίζουν οικογένειες χαρακτήρων ή κανονικών εκφράσεων και έχουν την εξής μορφή :

Όνομα **Κανονική έκφραση**

Παράδειγμα

letter [a-z]+

digit [0-9]*

Στο τμήμα των Ορισμών περιλαμβάνεται και ο κώδικας C\C++ που θέλουμε να συμπεριληφθεί στον παραγόμενο λεκτικό αναλυτή ,όπως δηλώσεις μακροεντολών ,μεταβλητών και τύποι δεδομένων .Στο τμήμα μπορούμε να κάνουμε 'include' headers για να χρησιμοποιήσουμε συναρτήσεις που είναι υλοποιημένες σε μία βιβλιοθήκη .Επιπλέον μπορούμε να υλοποιήσουμε συναρτήσεις που θα χρησιμοποιούνται από τον παραγόμενο συντακτικό αναλυτή. Το κομμάτι αυτό εσωκλείεται με %{ %}.

Παράδειγμα

```
%{
#include<stdio.h>
#DEFINE TK_IF  1
int count=0;
```

%}

Στο τμήμα αυτό ορίζονται οι αρχικές καταστάσεις (start states). Ο χρήστης μπορεί να ορίσει δικές του "καταστάσεις"(conditions) και να ενεργοποιεί ορισμένους κανόνες μόνο εάν ο λεξικογραφικός αναλυτής βρίσκεται σε συγκεκριμένη user-defined κατάσταση.

Τέλος στο τμήμα αυτό περιλαμβάνονται παράμετροι που επηρεάζουν τα χαρακτηριστικά του παραγόμενου λεξικογραφικού αναλυτή και ορίζονται ως εξης :
%option .....

Παρακάτω επεξηγούμε μερικές παραμέτρους που θα συναντήσουμε σε υλοποιήσεις του Flex :

%option noyywrap ή – noyywrap(command line options)
Η παράμετρο noyywrap δεν επιτρέπει στον παραγόμενο λεξικογραφικό αναλυτή να χρησιμοποιήσει την συνάρτηση yywrap ,η οποία καλείται όταν τελειώσει το διάβασμα ενός αρχείου, υποθέτοντας ότι δεν υπάρχουν άλλα αρχεία, ωσότου ο χρήστης δώσει είσοδο(yyin) κάποιο αρχείο και ξανακαλέσει την συνάρτηση λεκτικής ανάλυσης yylex().

%option yylineno ή -- yylineno
Δηλώνει μια καθολική(global) μεταβλητή yylineno που κατευθύνει το Flex για την δημιουργία ενός σαρωτή ο οποίος θα διατηρεί τον αριθμό της τρέχουσας γραμμής του αρχείου εισόδου .

%option case-insensitive ή -- case-insensitive
Αναθέτει στον Flex να δημιουργήσει μια διάκριση πεζών και κεφαλαίων χαρακτήρων. Δηλαδή όταν διαβάζει την λέξη caSE να την ταυτίζει με την λέξη case.

%option nodefault ή -- nodefault

Καταργεί τον προεπιλογή κανόνα που τύπωνε την μη ταιριασμένη είσοδο στην έξοδο. Εάν ο σαρωτής συναντήσει μια είσοδο που δεν ταιριάζει με κανένα κανόνα την αποφεύγει με ένα error.

%option array ή --array(command line options)

Με την χρήση αυτής της παραμέτρου ,διευκρινίζουμε στο σαρωτή ότι το yytext(ταίριασμα της εισόδου) να είναι ένας απλός πίνακας και όχι δείκτης(char*)

%option stdout' ή `-t, --stdout (command line options)

Καθοδηγεί τον flex να εμφανίσει τον σαρωτή που έχει παραχθεί στην στάνταρ έξοδο αντί της lex.yy.c

1.4.2 Τμήμα κανόνων

Αποτελεί το κύριο τμήμα του λεξικογραφικού αναλυτή.

Ό βασικός τρόπος για να περιγράψουμε έναν κανόνα γίνεται με την χρήση κανονικών εκφράσεων της μορφής :

pattern {action}

κανονική έκφραση {ενέργεια}

Παράδειγμα

```
[0-9]* {printf("found a digit %s\n");}
```

Σε περίπτωση που βρει ένα αριθμό στην είσοδο του λεξικογραφικού αναλυτή, θα τυπώσει το μήνυμα 'found a digit'.

Όταν ικανοποιείται ένα pattern, ο κανόνας ενεργοποιείται και η ενέργεια που ακολουθεί εκτελείται.

Πρέπει να προσέξουμε ότι όταν θέλουμε να συντάξουμε ένα pattern δεν πρέπει να αφήνουμε κενό γιατί το δεξιότερο μέρος θα λειτουργήσει σαν action.

Στο τμήμα των κανόνων, τα patterns ή καλύτερα οι κανονικές εκφράσεις μπορούμε να τις αντικαταστήσουμε με κάποιο όνομα ανάμεσα σε {} ,το οποίο προηγουμένως το έχουμε προσδιορίσει στο τμήμα των ορισμών. Αυτό είναι χρήσιμο όταν θέλουμε να σπάσουμε σύνθετες κανονικές εκφράσεις. Για παράδειγμα:

```
Dig    [0-9]                ←———————  Τμήμα Ορισμών
Letter [a-z]
%%
{letter} {printf("found a letter%s\n");}  ←———————  Τμήμα Κανόνων
%%
```

Στο τμήμα αυτό περιλαμβάνει κώδικας C/C++ ανάμεσα σε %{ και %} στην αρχή του τμήματος κανόνων που χρησιμεύει για τη δήλωση τοπικών μεταβλητών στη συνάρτηση του λεκτικού αναλυτή yylex().Καθώς επίσης στο τμήμα αυτό συντάσσονται και start states (θα αναφερθούμε παρακάτω).
Το Flex αποτελείται από μία λίστα κανονικών εκφράσεων και με τις ενέργειες (actions) σχετικά με το τι πρέπει να κάνει όταν η είσοδο ταιριάσει με κάποια από τις κανονικές εκφράσεις. Η κάθε ενέργεια που χρησιμοποιείται είναι εντολή της C.
Η βασική λειτουργία του τμήματος αυτού είναι το ταίριασμα του μακρύτερου προθέματος της συμβολοσειράς εισόδου, που μπορεί να αναγνωριστεί από μια κανονική έκφραση και να εκτελεστεί η αντίστοιχη ενέργεια.

1.4.3 Τμήμα Κώδικα χρήστη

Το τμήμα αυτό μπορεί να είναι προαιρετικό . Ο κώδικας (C/C++) που χρησιμοποιείται σε αυτό το κομμάτι χρησιμοποιεί τις βοηθητικές συναρτήσεις που καλούνται από τον λεκτικό αναλυτή. Σημαντικό να αναφέρουμε ότι οι βοηθητικές συναρτήσεις είναι ορισμένες στο τμήμα των κανόνων.
Αν θέλουμε το αποτέλεσμα της επεξεργασίας του λεκτικού αναλυτή να χρησιμοποιηθεί ως αυτόνομο πρόγραμμα στο τμήμα αυτό περιλαμβάνεται η main(),διαφορετικά ο παραγόμενος κώδικας θα χρησιμοποιηθεί στο πλαίσιο άλλου προγράμματος.

Παράδειγμα

```
%%
void ERROR (const char msg [])
{
```

```
fprintf(stderr, "ERROR: %s\n", msg);

exit(1);

}
```

Ο κώδικας του τμήματος αυτού αντιγράφεται αυτολεξεί στο `lex.yy.c', το οποίο χρησιμοποιείται για τις συνοδευτικές διαδικασίες που απαιτεί ο σαρωτής. Το κείμενο εκτός των εσοχών ή το κείμενο που βρίσκεται εντός

1.4.4 Περιγραφή ενός ολοκληρωμένου προγράμματος Flex

Παρακάτω περιγράφεται ένα ολοκληρωμένο πρόγραμμα Flex ,το οποίο θα μας βοηθήσει να κατανοήσουμε την σύνταξη και την δομή ενός λεκτικού αναλυτή. Το παράδειγμα μετράει τις γραμμές ,τα γράμματα και τις λέξεις της εισόδου.

Παρατηρούμε ότι το πρόγραμμα συμπεριλαμβάνει και τα τρία τμήματα (το τρίτο τμήμα μπορεί να παραληφθεί). Στο τμήμα των ορισμών περιλαμβάνεται ο κώδικας C\C++ που θέλουμε να συμπεριληφθεί στον παραγόμενο λεκτικό αναλυτή και συγκεκριμένα περιλαμβάνει δηλώσεις μεταβλητών(chars,words και lines).

Το τμήμα των κανόνων απαρτίζεται από τρία patterns και τις αντίστοιχες actions.Το πρώτο pattern ταιριάζει μια λέξη(word). Οι χαρακτήρες που εσωκλείονται σε αγκύλες, γνωστό ως κλάση χαρακτήρων, ταιριάζουν ένα μικρό ή κεφαλαίο γράμμα, και το σύμβολο + σημαίνει ότι ταιριάζει ένα ή περισσότερα γράμματα ή λέξεις. Η action που του αντιστοιχεί απλώς ενημερώνει τον αριθμό των γραμμάτων και των λέξεων που ταιριάζει. Η μεταβλητή yytext έχει οριστεί για να δείχνει την είσοδο που έχει ταιριαστεί με το pattern. Το δεύτερο pattern απλώς ταιριάζει την νέα γραμμή και η action ενημερώνει τον αριθμό των γραμμάτων και των γραμμών. Το τελευταίο pattern ταιριάζει οποιονδήποτε χαρακτήρα και η action ενημερώνει τον αριθμό των χαρακτήρων.

Ο κώδικα C στο τρίτο τμήμα, καλεί την yylex(),δηλαδή την διαδικασία του σαρωτή και εκτυπώνει τα αποτελέσματα.

```
/* just like Unix wc */
%{
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);
}
```

Figure 1.4.4 Count words

1.5Αρχικές Καταστάσεις (Start states)

Η περιγραφή των λεκτικών μονάδων μόνο με την χρήση κανονικών εκφράσεων ,μερικές φορές δεν είναι απλή. Για το λόγο αυτό το Flex παρέχει το μηχανισμό των αρχικών καταστάσεων, με τον οποίο ενεργοποιούνται οι κανόνες υπό συνθήκη. Ο κανόνας, η κανονική έκφραση του οποίου ξεκινά με $<A_1,A_2,A_3,\ldots A_n>$, μπορεί να ενεργοποιηθεί μόνο αν η τρέχουσα κατάσταση είναι μια από τις $A_1,A_2,A_3,\ldots A_n$. Επιπλέον τις αρχικές καταστάσεις τις χρησιμοποιούμαι για να περιορίσουμε το πεδίο εφαρμογής κάποιων κανόνων ή για να αλλάξουμε τον τρόπο που ο Flex αντιμετωπίζει ένα μέρος αρχείου. Η αρχική κατάσταση κατά την έναρξη του λεκτικού αναλυτή ονομάζεται INITIAL και είναι κοινή. Η δήλωση των αρχικών καταστάσεων γίνεται στο τμήμα των Ορισμών(Definition) και διακρίνονται σε δυο είδη :

- Inclusive conditions(κοινές καταστάσεις) :Ορίζονται με την εντολή "%s condition_name" και ενεργοποιεί όλους του κανόνες με πρόθεμα < condition_name > και κανόνες χωρίς πρόθεμα.

- Exclusive conditions(αποκλειστικές καταστάσεις) : Ορίζονται με την εντολή "%x condition_name" και ενεργοποιεί όλους του κανόνες με πρόθεμα < condition_name >.

%s example1	%x example2
%%	%%
<example1>foo do_something();	<example2>foo do_something();
bar something_else();	<example2>bar something_else();

Figure 1.5 Exclusive-inclusive conditions

Στα παραπάνω παραδείγματα μπορούμε να αντιληφθούμε την διαφορά μεταξύ των Inclusive και Exclusive conditions . Στο δεύτερο παράδειγμα αν είχαμε παραλείψει < example2> στο πρότυπο bar, η αρχική κατάσταση example2 δεν θα ενεργοποιούσε το πρότυπο bar .Ενώ στο πρώτο παράδειγμα θα εκτελεστεί κανονικά το πρότυπο bar γιατί η αρχική κατάσταση βρίσκεται σε inclusive condition .

Με χρήση της εντολή BEGIN A σε κάποια ενέργεια ,όπου A condition_name,η τρέχουσα κατάσταση θα μεταβεί στην κατάσταση A .Ας δούμε ένα ποιο ολοκληρωμένο παράδειγμα μιας αρχικής κατάστασης.

```
%x comment
%%
 int line_num = 1;
"/*"         BEGIN(comment);
<comment>[^*\n]*
<comment>"*"+[^*/\n]*
<comment>\n                  ++line_num;
<comment>"*"+"/"             BEGIN(INITIAL);
```

Figure 1.5-a Απαλοιφή σχολίων

Στο παραπάνω παράδειγμα, η αρχική κατάσταση <comment> αναγνωρίζει τα σχόλια που θα υπάρξουν σε αρχείο εισόδου ,τα διαγράφει και παράλληλα κρατάει τον αριθμό της τρέχουσας γραμμής εισόδου, στην μεταβλητή line_num. Μετά την διαγραφή των σχολίων ενεργοποιείται η αρχική κατάσταση INITIAL.

Άς δούμε ένα τελευταίο παράδειγμα για να καταλάβουμε περισσότερο την σύνταξη μιας αρχικής κατάστασης.

```
%{
#include <math.h>
%}
%s expect
%%
expect-floats BEGIN(expect);
<expect>[0-9]+"."[0-9]+     {printf( "found a float, = %f\n",atof( yytext ) );}
<expect>\n                  { /* that's the end of the line, so
                             * we need another "expect-number"
                             * before we'll recognize any more
                             * numbers
                             */
                            BEGIN(INITIAL);
                            }
[0-9]+                  {printf( "found an integer, = %d\n",atoi( yytext ) );}
"."                     {printf( "found a dot\n" );}
```

Figure 1.5-b Float and Integer Number

Ο σαρωτής στο παραπάνω παράδειγμα παρέχει δυο διαφορετικές ερμηνείες στην περίπτωση που δοθεί ως είσοδο η συμβολοσειρά "123,524".Εξ'ορισμού θα αντιμετωπίσει την συμβολοσειρά ως δέντρο από tokens,τον ακέραιο αριθμό "123", την τελεία (.),και τον ακέραιο αριθμό "524". Ένα όμως στην συμβολοσειρά προηγείται η έκφραση " expect-floats", τότε θα γίνει η κλήση την αρχικής κατάστασης <expect> , και θα ταιριάξει την συμβολοσειρά "123,524" ως πραγματικό αριθμό(αριθμό κινητής υποδιαστολής),αντιμετωπίζοντάς την σαν ένα token.

1.6 Διαχείριση τμημάτων προτύπου

Στην ενότητα αυτή θα μιλήσουμε για την διαχείριση των Tokens.Τα Tokens όπως έχουμε αναφέρει σε παραπάνω ενότητα ,είναι πρότυπα (patterns),πρότυπα που είναι ταιριασμένα κατά την διαδικασία της λεκτική ανάλυση.

Το Flex παρέχει πολλούς τρόπους για να διαχειριστούμε το δεξί και το αριστερό τμήμα ενός προτύπου(αναγνωρισμένου Token). Μας παρέχει τρόπους που προσδιορίζουν και καθορίζουν την αλληλουχία των περιεχομένων των προτύπων ,όσον αφορά πιο περιεχόμενο προηγείται ακόμα και πιο token ακολουθείται

1.6.1 Αριστερό τμήμα του προτύπου

Το τμήμα αυτό μπορούμε να το χειριστούμε με τρείς απλούς τρόπους: τον ειδικό χαρακτήρα που καθορίζει την αρχή της γραμμής (^),τις αρχικές καταστάσεις και σαφής κώδικα.

Ο χαρακτήρας ^ λέει στο Flex, να ταιριάξει το pattern μόνο όταν βρίσκεται στην αρχή της γραμμής. Όπως καταλαβαίνουμε ο χαρακτήρας αυτός προσδιορίζει το περιεχόμενο το pattern και κυρίως το αριστερό τμήμα του προτύπου .

Η χρήση των αρχικών καταστάσεων μπορεί να καθορίσει πιο token προηγείται.

```
%s MYSTATE
%%
first { BEGIN MYSTATE; }
. . .
<MYSTATE>second { BEGIN 0; }
```

Figure 1.6.1 Καθορισμός Token

Στο παραπάνω παράδειγμα βλέπουμε ότι μεταξύ των δύο tokens υπάρχει μια διακριτική παρέμβαση. Το second Token αναγνωρίζεται εφόσον το first Token αναγνωριστεί πρώτο.

Όταν θέλουμε να μεταφέρουμε πληροφορίες από το ένα τμήμα ενός Token,στο τμήμα ενός άλλου Token ,μπορούμε να χρησιμοποιήσουμε flags.

```
%{
int flag = 0;
%}
%%
a { flag = 1; }
b { flag = 2; }
zzz {
switch(flag) {
                case 1: a_zzz_token(); break;
                case 2: b_zzz_token(); break;
               default: plain_zzz_token(); break;
             }
flag = 0;
}
```

Figure 1.6.1-a Flags

1.6.2 Δεξί τμήμα του προτύπου

Υπάρχουν τρείς τρόποι ,ώστε η αναγνώριση του Token να εξαρτάται από το δεξί τμήμα του: ο ειδικός χαρακτήρας $ που καθορίζει το τέλος της γραμμής, ο slash operator(/), και η συνάρτηση less().

Ο ειδικός χαρακτήρας $ όταν βρίσκεται στο τέλος ενός προτύπου, κάνει το ταίριασμα του token στο τέλος της γραμμής. Όπως και ο χαρακτήρας ^ έτσι και ο $ δεν αναφέρεται στο ταίριασμα οποιονδήποτε χαρακτήρα. Αυτό σημαίνει ότι προσδιορίζει το τμήμα του token και στην περίπτωσή μας προσδιορίζει το δεξί τμήμα ενός token.

Ο slash operator / ορίζει ακολουθούμενα συμφραζόμενα. Το πρότυπο ab/e θα ταιριαστεί εφόσον το ab ακολουθείται από το e. Επομένως είναι κατανοητό πιο τμήμα προσδιορίζεται όταν χρησιμοποιούμε το χαρακτήρα /.

Η συνάρτηση yyless() πρακτικά λέει στο Flex 'να σπρώξει προς τα πίσω' μέρος από το token που μόλις έχει διαβαστεί. Η yyless() πρακτικά είναι ένας αριθμός από τα τους χαρακτήρες του token που κρατάει. Για παράδειγμα : abcde {yyless(3);}
Στο παράδειγμα η yyless κρατάει τους πρώτους τρεις χαρακτήρες (abc) και επιστρέφει στην είσοδο τους υπόλοιπους (de) ,δηλαδή επιστρέφει το δεξί τμήμα του token που έχει αναγνωριστεί.

1.7Διεπαφή χρήστη

Το περιβάλλον του Flex παρέχει έτοιμες συναρτήσεις οι οποίες είναι προσβάσιμες και είναι δυνατόν να επαναπροσδιοριστούν από τον χρήστη. Οι λειτουργίες / μακροεντολές που περιγράφονται παρακάτω περιλαμβάνονται στην είσοδο του Flex και είναι προσβάσιμες από τον κώδικα χρήστη(user code).

int ***yylex***()

Αποτελεί την κύρια συνάρτηση του λεκτικού αναλυτή που παράγεται βάσει του αρχείου περιγραφής. Δίνεται από τον χρήστη ή παράγεται από το Flex. Καλείται όταν πρέπει να διαβαστεί μια νέα λεκτική μονάδα(token) και επιστρέφει συνήθως έναν ακέραιο που αντιστοιχεί στο αναγνωριστικό της λεκτικής μονάδας. Όταν ταιριαστεί ένα token η συνάρτηση yylex επιστρέφει μηδέν, αλλιώς επιστρέφει μη μηδενικές τιμές που καθορίζονται από τις actions που έχουν επιλεγεί.

```
%%
    main()
        {
        yylex();          <----------   'Τμήμα Κώδικα χρήστη'
        }
```

char* yytext

Περιέχει το τμήμα του αρχείου εισόδου που αναγνωρίστηκε κατά την τελευταία κλήση της yylex.

int yyleng

Περιέχει το πλήθος των χαρακτήρων που έχουν ικανοποιηθεί και περιέχονται στο yytext. Επιστρέφει έναν ακέραιο που περιλαμβάνει το μήκος του yytext και είναι ισοδύναμος με τον ακέραιο που επιστρέφει και strlen(yytext).

Void yymore()

Επισυνάπτει το επόμενο token που έχει αναγνωριστεί με το προηγούμενο token και το αντιμετωπίζει ως ένα. Η λειτουργία της συνάρτησης φαίνεται στο παρακάτω παράδειγμα.

```
%%
abc   {yymore ();}          <----------   Τμήμα Κανόνων
de    { printf("Token is %s\n", yytext);}
%%
```

Εάν δώσουμε σαν είσοδο την συμβολοσειρά abcde ,η έξοδο μας δεν θα περιέχει δύο ταιριασμένα tokens το abc και το cd, αλλά θα αναγνωριστεί ως ένα. Επομένως με την χρήση της yymore() η έξοδο που θα πάρουμε είναι 'Token is abcde'. Η συνάρτησης yymore() είναι χρήσιμη ,όταν θέλουμε να ορίσουμε συγκεκριμένα tokens με την χρήση κανονικών εκφράσεων.

Void yyless(n) όπου int n;

Διατηρεί τους πρώτους n χαρακτήρες και επιστρέφει στην έξοδο τους υπόλοιπους. Η συνάρτηση yyless() πρακτικά είναι χρήσιμη όταν θέλουμε να βάλουμε κάποιο όριο μεταξύ των tokens που τα εκφράζουμε με κανονικές εκφράσεις.

int input()

Διαβάζει ένα χαρακτήρα από το τρέχον αρχείο εισόδου .Κατά την διαδικασία της λεκτικής ανάλυση, για το ταίριασμα κάποιου token καλείται έμμεσα η συνάρτηση input() .Στην πραγματικότητα η συνάρτηση φέρνει τον τρέχον χαρακτήρα στον λεκτικό αναλυτή για να ταιριαστεί.

Συνήθως την συνάρτηση input() την χρησιμοποιούμε όταν θέλουμε να κάνουμε κάτι ιδιαίτερο με το κείμενο (text) που ακολουθεί ένα token.Στο παρακάτω παράδειγμα βλέπουμε ένα τρόπο για να χειριζόμαστε τα σχόλια. Στην πράξη καλείται η input() για να διαβαστούν οι χαρακτήρες έως ότου τελειώσει το αρχείο ή διαβάσει */.

```
"/*" { int c1 = 0, c2 = input ();
        for(;;) {
                if(c2 == EOF)          <-------- Τέλος αρχείου
                break;
                if(c1 == '*' && c2 == '/')  <---- Τέλος σχολίων (*/)  Τμήμα Κανόνων
                break;
                c1 = c2;
                c2 = input();
               }
    }
```

void unput(c) όπου char c;

Τοποθετεί τον χαρακτήρα c στην ακολουθία εισόδου. Αυτό σημαίνει ότι ο επόμενος χαρακτήρας που θα διαβάσει ο λεκτικός αναλυτής είναι το c.

Σημαντικό είναι αναφέρουμε δύο ειδικές ενέργειες που μπορεί να χρησιμοποιήσει ο χρήστης στο τμήμα Κανόνων.

- **ECHO**: εκτυπώνει την είσοδο που έχει ταιριαστεί, δηλαδή αντιγράφει τα περιεχόμενα του yytext στο τρέχον αρχείο εξόδου yyout. Είναι ισοδύναμο με το ακόλουθο:

fprintf(yyout, "%s", yytext);

Η προεπιλεγμένη δράση(default action) του flex για το κείμενο εισόδου που δεν έχει ταιριαστεί με κανένα pattern ,αντιγράφεται στην έξοδο,που ισοδυναμεί με την μακρο εντολή ECHO.Στο flex υπάρχει δυνατότητα ματαίωσης της προεπιλεγμένης δράσης(default action),η οποία είναι χρήσιμη στη περίπτωση που ο σαρωτής περιλαμβάνει patterns για να χειριστεί όλες τις πιθανές εισόδους. Η ματαίωση της default action μπορεί να γίνει με την χρήση της παραμέτρου %option nodefault ή -- nodefault σε γραμμή εντολών.

- **REJECT**: Κατευθύνει τον σαρωτή να επιλέξει το δεύτερο καλύτερο κανόνα που ταιριάζει την είσοδο. Με απλά λόγια μας επιτρέπει να ζητήσουμε από το μηχανισμό του Flex να επιλέξει την αμέσως καλύτερη ταύτιση κανόνα από αυτή που είχε επιλέξει γι' αυτή τη Δράση, επαναχρησιμοποιώντας έτσι τους χαρακτήρες.

Παράδειγμα

```
pink { npink++; REJECT; }

ink { nink++; REJECT; }

pin { npin++; REJECT; }

. |

\n ; /* discard other characters */
```

Στο παραπάνω παράδειγμα αν δίναμε σαν είσοδο την λέξη pink, θα διαπιστώναμε ότι και τα τρία patterns θα ταίριαζαν με την λέξη. Αν όμως δεν χρησιμοποιούσαμε το REJECT ,θα ταίριαζε η λέξη μόνο με το πρώτο pattern.

- **BEGIN(condition):** βάζει το λεξικογραφικό αναλυτή να μεταβεί στην κατάσταση με όνομα "condition"
- **yywrap()**

Όταν ο λεκτικός αναλυτής συναντά το τέλος ενός αρχείου, καλεί προαιρετικά την ρουτίνα yywrap() για να μάθει τι πρέπει να κάνει στη συνέχεια. Εάν η yywrap() επιστρέψει 0,ο σαρωτής συνεχίζει να διαβάζει ,ενώ αν η yywrap() επιστρέψει 1, ο σαρωτής επιστρέφει μηδέν token για να αναφέρει το τέλος του αρχείου.

1.8 Αρχεία I/O σε σαρωτές Flex

Υπάρχουν πολλοί τρόποι για να διαχειριστούμε την είσοδο και την έξοδο ενός σαρωτή. Οι μεταγλωττιστές Flex μπορούν να διαβάσουν από μια standard είσοδο (stdin), εκτός αν τους δώσουμε κάτι διαφορετικό. Συνήθως οι σαρωτές διαβάζουν από αρχεία. Ένας σαρωτής διαβάζει από το stdio FILE που κρατάει της πληροφορίες της εισόδου. Στους σαρωτές Flex καλείται ως yyin .

Επομένως για να μπορέσουμε να διαβάσουμε από ένα αρχείο ,αρκεί να κάνουμε τις σωστές ρυθμίσεις της εισόδου, δηλαδή το yyin πρίν γίνει το πρώτο κάλεσμα της yylex().Στο παρακάτω παράδειγμα μπορούμε να καταλάβουμε πως γίνεται ο καθορισμός ενός αρχείου ως είσοδο.

```
%option noyywrap
%{
int chars = 0;
int words = 0;
int lines = 0;
%}
%%
[a-zA-Z]+       { words++; chars += strlen(yytext); }
\n                   { chars++; lines++; }
.                      { chars++; }
%%
main(argc, argv)
int argc;
char **argv;
{
if(argc > 1) {
            if(!(yyin = fopen(argv[1], "r")))
              {
```

```
                perror(argv[1]);
                return (1);
            }
                }
yylex();
printf("%8d%8d%8d\n", lines, words, chars);
}
```

Figure 1.8 Count words of a file

Η μόνη διαφορά με το παράδειγμα Figure 1.4.4 Count words βρίσκεται στο τρίτο τμήμα(τμήμα κώδικα χρήστη).Στην main διαδικασία ανοίγει ένα όνομα αρχείου περασμένο στην γραμμή εντολών, εάν ο χρήστης δώσει ένα αρχείο ως είσοδο τότε εκχωρείται το αρχείο στο yyin.Διαφορετικά ένα δεν δοθεί κανένα αρχείο ως είσοδο, το yyin παραμένει απενεργοποιημένο, και στην περίπτωση αυτή η yylex()αυτόματα ορίζει ως είσοδο την stdin.

Όταν έχουμε να κάνουμε σάρωση σε περισσότερα από ένα αρχεία εισόδου τα πράγματα περιπλέκονται. Για τον λόγο αυτό το Flex παρέχει την διαδικασία yyrestart(). Η διαδικασία yyrestart(f) καθορίζει ποια είσοδο θα διαβάσει ο σαρωτής (stdio file f).

```
 %option noyywrap
%{
int chars = 0;
int words = 0;
int lines = 0;
int totchars = 0;
int totwords = 0;
int totlines = 0;
%}

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

```
char **argv;
{
int i;
if(argc < 2) { /* just read stdin */
yylex();
printf("%8d%8d%8d\n", lines, words, chars);
return 0;
}
for(i = 1; i < argc; i++) {
FILE *f = fopen(argv[i], "r");
if(!f) {
perror(argv[i]);
return (1);
}
yyrestart(f);
yylex();
fclose(f);
printf("%8d%8d%8d %s\n", lines, words, chars, argv[i]);
totchars += chars; chars = 0;
totwords += words; words = 0;
totlines += lines; lines = 0;
}
if(argc > 1) /* print total if more than one file */
printf("%8d%8d%8d total\n", totlines, totwords, totchars);
return 0;
}
```

Figure 1.8-a Reading Multiple Files-Count words

Στο παραπάνω παράδειγμα κάθε αρχείο "ανοίγεται" από την διαδικασία yyrestart(). Η yyrestart() προωθεί στο σαρωτή το αρχείο , και καλεί την yylex() για να διαβαστεί το αρχείο. Η διαδικασία επαναλαμβάνεται μέχρι να τελειώσουν τα αρχεία εισόδου. Εάν εκτελέσουμε το παραπάνω παράδειγμα τα αποτελέσματα που θα δούμε στην έξοδο είναι ο συνολικός αριθμός των λέξεων, χαρακτήρων και γραμμών του κάθε αρχείου ξεχωριστά καθώς και ο συνολικός αριθμός των λέξεων, χαρακτήρων και γραμμών όλων των αρχείων μαζί.

1.9 End-Of-File Κανόνας

Ο ειδικός κανόνας <<EOF>> υποδεικνύει ενέργειες(actions),οι οποίες πρέπει να λαμβάνονται όταν ο σαρωτής συναντά το τέλος του αρχείου και η ρουτίνα wrap() επιστρέφει 0.Η ενέργεια (action) θα τελειώσει εκτελώντας μια από τις παρακάτω λειτουργίες:

- Εκχώρηση στο yyin ένα νέο αρχείο εισόδου
- Εκτέλεση μιας επιστρεφόμενης δήλωσης
- Εκτέλεση της ειδικής ενέργειας yyterminate()
- Ή μετάβαση σε ένα νέο buffer χρησιμοποιώντας την μακροεντολή yy_switch_to_buffer()

Οι <<EOF>> κανόνες ίσως να μην μπορούν να χρησιμοποιηθούν με άλλα patterns ,αλλά μπορεί να είναι προσδιοριστικοί σε μια λίστα αρχικών καταστάσεων. Εάν δοθεί ένας κανόνας <<EOF>> και δεν έχει προσδιοριστεί που θα εκτελεστεί, τότε θα εφαρμοστεί σε όλες τις αρχικές καταστάσεις που θα έχουν δηλωθεί και ίσως να μην περιέχουν ενέργειες EOF.Επομένως για να ορίσουμε έναν <<EOF>> κανόνα που θα εκτελείται σε μια μόνο αρχική κατάσταση ,π.χ INITIAL start state , ακολουθούμε την εξής σύνταξη:

<INITIAL><<EOF>>

Οι <<EOF>> κανόνες θα εκτελεστούν όταν το πρόγραμμα βρίσκεται στην default INITIAL κατάσταση.

<S1,S2><<EOF>> θα εκτελεστούν όταν το πρόγραμμα βρίσκεται στις καταστάσεις S1,S2.

1.10 Δομή I/O ενός σαρωτή Flex

Βασικά, ένας σαρωτής Flex διαβάζει από μία πηγή εισόδου και γράφει τα αποτελέσματα σε μία έξοδο. Εξ'ορισμού , όταν μιλάμε για είσοδο και έξοδο συνήθως αναφερόμαστε στο stdin και stdout ,αλλά έχουμε παρατηρήσει ότι μερικές φορές αναφέρονται σε κάτι διαφορετικό.

1.10.1 Είσοδος ενός σαρωτή Flex(INPUT)

Στα προγράμματα που περιλαμβάνουν σαρωτές, η επίδοση της συχνότητας του σαρωτή καθορίζει την απόδοση ολόκληρου του προγράμματος. Οι προηγούμενες εκδόσεις του lex διάβαζαν από το yyin ένα χαρακτήρα την φορά. Από τότε, το Flex ανέπτυξε ένα ευέλικτο τριών επιπέδων σύστημα εισόδου, που επιτρέπει στους προγραμματιστές να προσαρμόζουν το κάθε επίπεδο ώστε να μπορέσουν να χειριστούν οποιαδήποτε δομή εισόδου. Τό σύστημα εισόδου τριών επιπέδων του flex αποτελείται:

- Ρύθμιση του yyin , ώστε να διαβαστούν τα αρχεία (ή αρχείο)
- Δημιουργία και χρήση της δομής YY_BUFFER_STATE
- Επαναπροσδιορισμός της YY_INPUT

Σε πολλές περιπτώσεις ,ένας σαρωτής Flex διαβάζει την είσοδο του, χρησιμοποιώντας ένα αρχείο ή μια standard είσοδο, όπου σαν είσοδο μπορεί να είναι και η κονσόλα χρήστη(user console). Υπάρχει μια λεπτή αλλά σημαντική διαφορά μεταξύ της εισόδου από ένα αρχείο και της εισόδου από την κονσόλα. Άν ο σαρωτής διαβάζει από ένα αρχείο, πρέπει τα μεγάλα κομμάτια να διαβαστούν όσο το δυνατόν γρηγορότερα. Αν όμως διαβάζει από την κονσόλα δημιουργείται μια καθυστέρηση στην διαδικασία του σαρωτή. Για να το καταλάβουμε ,ας υποθέσουμε ότι ο χρήστης πληκτρολογεί μια γραμμή την φορά, αυτό σημαίνει ότι ο σαρωτής θα διάβαζε κάθε γραμμή που είναι ήδη δακτυλογραφημένη. Συνεπώς δημιουργείται καθυστέρηση στον σαρωτή γιατί δεν μπορεί ο σαρωτής να διαβάσει μεγάλα κομμάτια. Στην περίπτωση αυτή η ταχύτητα δεν έχει μεγάλη σημασία, δεδομένου ότι ένας αργός σαρωτής είναι πιο γρήγορος από μια γρήγορη δακτυλογράφος, αφού διαβάζει έναν χαρακτήρα την φορά. Ευτυχώς το flex ελέγχει τι είσοδο θα χρησιμοποιηθεί και αυτόματα κάνει σωστές ρυθμίσει.

Για το χειρισμό της εισόδου , ο σαρωτής flex χρησιμοποιεί μια δομή, γνωστή ως YY_BUFFER_STATE, η οποία περιγράφει μια πηγή εισόδου. Περιέχει ένα string Buffer , σωρό από μεταβλητές και flags. Συνήθως περιέχει και ένα δείκτη (FILE*) που δείχνει από πιο αρχείο διαβάζει. όμως υπάρχει περίπτωση να δημιουργηθεί η δομή YY_BUFFER_STATE μη συνδεμένη με κάποιο αρχείο, αλλά για να σχετιστεί με μια συμβολοσειρά στην μνήμη.

Η συμπεριφορά της προεπιλεγμένη εισόδου(default input) ενός σαρωτή flex ,μπορεί κατά προσέγγιση να είναι ως εξής:

```
YY_BUFFER_STATE bp;

extern FILE* yyin;

... whatever the program does before the first call to the scanner

if(!yyin) yyin = stdin; default input is stdin

bp = yy_create_buffer(yyin,YY_BUF_SIZE );

YY_BUF_SIZE defined by flex, typically 16K

yy_switch_to_buffer(bp); tell it to use the buffer we just made

yylex(); or yyparse() or whatever calls the scanner
```

Εάν δεν έχει οριστεί η yyin ,δηλαδή η είσοδο, ορίζει ως είσοδο την default standard input(stdin). Έπειτα χρησιμοποιεί την συνάρτηση yy_create_buffer για να δημιουργήσει έναν καινούργιο buffer(ενδιάμεση μνήμη) που θα διαβάζει από το yyin,και την συνάρτηση yy_switch_to_buffer για να ενημερώσει τον σαρωτή να διαβάσει από τον νέο buffer και τέλος γίνεται η σάρωση .
Τέλος, για μέγιστη ευελιξία, μπορούμε να επαναπροσδιορίσουμε την μακροεντολή που χρησιμοποιεί το flex για να διαβάσει την είσοδο από τον τρέχον buffer:

```
#define YY_INPUT(buf,result,max_size) ...
```

Σε περίπτωση που ο τρέχον buffer ,που χρησιμοποιείται ως είσοδο, είναι άδειος, ο σαρωτής παραπέμπεται στην YY_INPUT, όπου buf είναι ο buffer και max_size το μέγεθος του buffer,αντίστοιχα result είναι το που θα τεθεί το πραγματικό μέγεθος που έχει διαβαστεί ή μηδέν (eof). Σήμερα η κύρια χρήση ενός συνηθισμένου YY_INPUT γίνεται σε event based συστήματα, όπου η είσοδος έρχεται από πηγή η οποία δεν μπορεί να την προ-φορτώσει (την είσοδο)σε ένα string buffer, πράγμα το οποίο σημαίνει οτι το stdio δεν μπορεί να την χειριστεί .Ο επαναπροσδιορισμός την μακροεντολής YY_INPUT, είναι χρήσιμος όταν η πηγή εισόδου δεν είναι αρχείο ή συμβολοσειρές.

1.10.2 Έξοδος ενός σαρωτή Flex(OUTPUT)

Η διαχείριση της εξόδου ενός σαρωτή είναι πολύ πιο απλή, από την διαχείριση της εισόδου, και εντελώς προαιρετική. Το flex λειτουργεί σαν να υπάρχει στο τέλος του σαρωτή ένας κανόνας που αντιγράφει την ταιριασμένη είσοδο, διαφορετικά θέτει την μη ταιριασμένη είσοδο στο yyout(default stdout).

. ECHO;

```
#define ECHO fwrite( yytext, yyleng, 1, yyout )
```

1.11 Input Buffers

Οι σαρωτές flex διαβάζουν την είσοδο από ένα buffer εισόδου.Ο buffer εισόδου μπορεί να συσχετιστεί με ένα stdio file, δηλαδή διαβάζει από ένα αρχείο, ή μπορεί να σχετίζεται με ένα string στην μνήμη. Ο τύπος YY_BUFFER_STATE, είναι ένας δείκτης που δείχνει στον buffer εισόδου του flex.

```
YY_BUFFER_STATE bp;

FILE *f;

f = fopen(..., "r");

bp = yy_create_buffer(f,YY_BUF_SIZE );   new buffer reading from f

yy_switch_to_buffer(bp);   use the buffer we just made

...

yy_flush_buffer(bp); discard buffer contents

...

void yy_delete_buffer (bp); free buffer
```

Σύμφωνα με το παραπάνω σχήμα, καλείται η ρουτίνα yy_create_buffer για να δημιουργηθεί ένας νέος buffer εισόδου που θα συσχετίζεται με ένα ανοιχτό stdio file και το δεύτερο argument YY_BUF_SIZE αναφέρεται στο μέγεθος του buffer. Στη συνέχεια καλείται η ρουτίνα yy_switch_to_buffer για να κατευθύνει τον σαρωτή να διαβάσει από τον buffer που έχει δημιουργηθεί. Μπορούμε να αλλάξουμε όσους buffer χρειάζονται. Ο τρέχον buffer είναι YY_CURRENT_BUFFER. Έπειτα καλείται η yy_flush_buffer για να απορριφτούν ότι είχε ο buffer. Η συνάρτηση αυτή

είναι χρήσιμη, για την ανάκτηση λαθών σε διαδραστικούς σαρωτές (interactive scanners). Και τέλος καλείται η ρουτίνα yy_delete_buffer για να διαγράψει τον Buffer.

1.11.1 Input from Strings

Συνήθως οι σαρωτές διαβάζουν από αρχείο, αλλά μπορεί να θελήσουμε να διαβάσει από μία διαφορετική πηγή όπως ένα string που βρίσκεται στην μνήμη.

```
bp = yy_scan_bytes(char *bytes, len); scan a copy of bytes
bp = yy_scan_string("string"); scan a copy of null-terminated string
bp = yy_scan_buffer (char *base, yy_size_t size); scan (size-2) bytes in place
```

Οι ρουτίνες yy_scan_bytes και yy_scan_string δημιουργούν ένα buffer ,ο οποίος διατηρεί ένα αντίγραφο του κειμένου, που πρόκειται να σαρωθεί. Μια ελαφρώς πιο γρήγορη ρουτίνα είναι η yy_scan_buffer, η οποία σαρώνει το κείμενο στην θέση του, εφόσον τα δυο τελευταία bytes του buffer είναι μηδενικά και δεν σαρώνονται. Ο τύπος yy_size_t είναι ο εσωτερικός τύπος του flex που χρησιμοποιείται για τα μεγέθη αντικειμένων.
Μόλις ένας buffer δημιουργήθηκε ,χρησιμοποιώντας την συνάρτηση yy_switch_to_buffer κατευθύνουμε τον σαρωτή να διαβάσει από αυτόν τον buffer.Χρησιμοποιώντας την συνάρτηση yy_delete_buffer ελευθερώνεται ο buffer και ενδεχομένως το αντίγραφο κειμένου που υπήρχε .Ο σαρωτής αντιμετωπίζει το τέλος του buffer ,ως το τέλος του αρχείου.

1.11.2 File Nesting(Εμφωλευμένα αρχεία)

Πολλές γλώσσες προγραμματισμού επιτρέπουν αρχεία εισόδου να συμπεριλαμβάνουν και άλλα αρχεία(εμφωλευμένα αρχεία), όπως #include στη C. Το flex παρέχει ζεύγη από συναρτήσεις που διαχειρίζονται στοίβες από αρχεία εισόδου(input files).

```
void yypush_buffer_state(bp); switch to bp, stack old buf
void yypop_buffer_state(); delete current buffer, return to previous
```

Στην πράξη ,οι παραπάνω συναρτήσεις είναι ανεπαρκείς για οποιονδήποτε που θα τις χρησιμοποιήσει για εμφωλευμένα αρχεία εισόδου, δεδομένου ότι δεν διατηρούν βοηθητικές πληροφορίες που σχετίζονται με τα αρχεία που στοιβάζονται, όπως ο αριθμός γραμμή ή το όνομα του τρέχοντος αρχείου. Παρόλα αυτά, δεν είναι δύσκολο να συμπεριλάβει κάποιος πληροφορίες στον κώδικά του χρησιμοποιώντας, διάφορα τεχνάσματα όπως δομές(struct), start state και το ειδικό token <<EOF>> , το οποίο ταιριάζει το τέλος του αρχείου, μετά το κάλεσμα της συνάρτησης yywrap().Για να καταλάβουμε των χειρισμό εμφωλευμένων αρχείων και την διαχείριση εισόδου-εξόδου στο flex, αρκεί να κατανοήσουμε το παρακάτω χαρακτηριστικό παράδειγμα

```
%option noyywrap
%x IFILE

%{
struct bufstack {
struct bufstack *prev;                    /* previous entry */
YY_BUFFER_STATE bs;           /* saved buffer */
int lineno;                                   /* saved line
number */
char *filename;                             /* name of this
file */
FILE *f;                                      /* current file
*/
} *curbs = 0;
char *curfilename;                        /* name of current input
file */
int newfile(char *fn);
int popfile(void);
%}

%%
match #include statement up through the quote or <
^"#"[ \t]*include[ \t]*[\ "<] { BEGIN IFILE; }

handle filename up to the closing quote, >, or end of line
<IFILE>[^ \t\n\">]+ {
                        { int c;
                          while((c = input()) && c != '\n') ;
                        }
                        yylineno++;
                        if(!newfile(yytext))
                        yyterminate();          /* no such file */
                        BEGIN INITIAL;
```

```
                    }

handle bad input in IFILE state
<IFILE>.|\n         { fprintf(stderr, "%4d bad include line\n",
yylineno); yyterminate();}

pop the file stack at end of file, terminate if it's the
outermost file
<<EOF>>          { if(!popfile()) yyterminate(); }

print the line number at the beginning of each line
and bump the line number each time a \n is read
^.              { fprintf(yyout, "%4d %s", yylineno, yytext); }
^\n            { fprintf(yyout, "%4d %s", yylineno++, yytext); }
\n              { ECHO; yylineno++; }
.               { ECHO; }

%%
main(int argc, char **argv)
{
if(argc < 2) {
  fprintf(stderr, "need filename\n");
  return 1;
                    }
if(newfile(argv[1]))
  yylex();
}

int newfile(char *fn)
  {
    FILE *f = fopen(fn, "r");
    struct bufstack *bs = malloc(sizeof(struct bufstack));

  /* die if no file or no room */
  if(!f) { perror(fn); return 0; }
  if(!bs) { perror("malloc"); exit(1); }

  /* remember state */
  if(curbs)curbs->lineno = yylineno;
  bs->prev = curbs;

  /* set up current entry */
  bs->bs = yy_create_buffer(f, YY_BUF_SIZE);
  bs->filename = fn;
  yy_switch_to_buffer(bs->bs);
  curbs = bs;
  yylineno = 1;
  curfilename = fn;
  return 1;
```

```
  }

int popfile(void)
{
  struct bufstack *bs = curbs;
  struct bufstack *prevbs;

  if(!bs) return 0;

  /* get rid of current entry
  fclose(bs->f);
  yy_delete_buffer(bs->bs);

  /* switch back to previous */
  prevbs = bs->prev;
  free(bs);

  if(!prevbs) return 0;

  yy_switch_to_buffer(prevbs->bs);
  curbs = prevbs;
  yylineno = curbs->lineno;
  curfilename = curbs->filename;
  return 1;
 }
```

Figure 1.11.2 Nested Files

Το παραπάνω πρόγραμμα τυπώνει τα εισαγόμενα αρχεία μαζί με τον αριθμό της κάθε γραμμής του αρχείου. Για να γίνει αυτό εφικτό ,το πρόγραμμα διατηρεί μια στοίβα με τα εμφωλευμένα αρχεία εισόδου και τους αριθμούς γραμμών, ωθώντας μια νέα καταχώρηση στην στοίβα κάθε φορά που συναντά μια #include καταχώρηση και αφαιρώντας την από την στοίβα αφού φτάσει στο τέλος κάποιου αρχείου.
Στο πρώτο τμήμα ορίζεται η αρχική κατάσταση (IFILE) και ο κώδικας C για την δήλωση της δομής bufstack που θα κρατήσει την καταχώρηση στην λίστα των αποθηκευμένων αρχείων εισόδου.
Το πρώτο pattern ταιριάζει μια #include δήλωση μέσα σε διπλό εισαγωγικό (<), η οποία παραπέμπει σε όνομα αρχείου καθώς επίσης επιτρέπει προαιρετικά να υπάρχει κάποιο κενό. Στην συνέχεια μεταβαίνει στην IFILE κατάσταση για να διαβάσει το επόμενο όνομα του αρχείου εισόδου. Στην κατάσταση IFILE, το δεύτερο pattern ταιριάζει το όνομα αρχείου, χαρακτήρες που προηγούνται από το κλειστό εισαγωγικό(>),το κενό ή το τέλος του αρχείου. Το όνομα του νέου αρχείου περνάει

στο newfile() για να οριστεί ως τρέχον αρχείο εισόδου και να δημιουργηθεί το επόμενο επίπεδο της εισόδου. Το επόμενο pattern ασχολείται με την περίπτωση που δεν έχει σχηματιστεί σωστά η γραμμή μετά το #include και δεν σχηματίζεται κάποιο όνομα αρχείου. Στην περίπτωση αυτή τυπώνεται ένα μήνυμα σφάλματος (error) και χρησιμοποιείται η μακροεντολή yyterminate() η οποία επιστρέφει αμέσως από τον σαρωτή.

Το επόμενο pattern που ακολουθείται είναι το ειδικό pattern <<EOF>>, το οποίο ταιριάζει το τέλος του αρχείου. Στην περίπτωση που ταιριαστεί το pattern , καλείται η popfile() για να επιστρέψει στο προηγούμενο αρχείο εισόδου. Εάν η μακροεντολή επιστρέψει 0 ,σημαίνει ότι αυτό είναι το τελευταίο αρχείο. Διαφορετικά ,ο σαρωτής θα συνεχίσει την ανάγνωση του προηγούμενου αρχείου .

Τα τέσσερα τελευταία patterns κάνουν την πραγματική δουλεία της εκτύπωσης κάθε γραμμής με τον αριθμό της προηγούμενης γραμμής. Toflex παρέχει την μεταβλητή yylineno, η οποία επιδιώκει να εντοπίσει τον αριθμό γραμμής. Το pattern ^. ταιριάζει οποιονδήποτε χαρακτήρα στην αρχή της γραμμής ,έτσι ώστε η action να εκτυπώνει τον τρέχον αριθμό της γραμμής και τον χαρακτήρα. Το pattern ^\n ταιριάζει την νέα γραμμή από την αρχή της γραμμής ,που σημαίνει ότι η νέα γραμμή είναι άδεια ,έτσι ώστε ο κώδικας εκτυπώνει τον αριθμό γραμμής και την νέα γραμμή και αυξάνει τον αριθμό γραμμής. Η νέα γραμμή ή οποιοσδήποτε χαρακτήρας που δεν βρίσκεται στην αρχή της γραμμής ,εκτυπώνεται με την εντολή ECHO,αυξάνοντας τον αριθμό για την νέα γραμμή.

Η ρουτίνα newfile(fn) προετοιμάζει το διάβασμα από το αρχείο με το όνομα fn ,αποθηκεύοντας το προηγούμενο αρχείο εισόδου. Αυτό το επιτυγχάνεται με την τήρηση μιας συνδεδεμένης λίστας με τις bufstack δομές, όπου η κάθε μία συνδέεται με την προηγούμενη bufstack δομή μαζί με την αποθηκευμένη yylineno και το όνομα του αρχείου. Ανοίγει το αρχείο, δημιουργεί και αλλάζει τον flex buffer και αποθηκεύει το προηγούμενο ανοιχτό αρχείο, το όνομα και τον buffer.

Η ρουτίνα popfile() κλείνει το ανοιχτό αρχείο, διαγράφει τον τρέχον flex buffer ,και στην συνέχεια αποκαθιστάει τον buffer, το όνομα αρχείου και τον αριθμό γραμμής από την προηγούμενη καταχώρηση της στοίβας.

1.12 Βιβλιοθήκη του flex

Το flex προέρχεται από μία σχετικά μικρή βιβλιοθήκη αλλά με χρήσιμες ρουτίνες. Μπορούμε να συνδεθούμε με την βιβλιοθήκη γράφοντας –lfl στο τέλος της γραμμής εντολή του unix ή άλλων ισοδύναμων συστημάτων. Περιέχει εκδόσεις της main() και της wrap().

Το flex συνοδεύεται από ένα μικρό main πρόγραμμα, το οποίο μπορεί να είναι χρήσιμο για γρήγορα προγράμματα και δοκιμές και από το 'στέλεχος' wrap().

```
main(int ac, char **av)

{

return yylex();

}

int yywrap() { return 1; }
```

1.13 Διαχείριση Πίνακα συμβόλων και γεννήτρια πινάκων συμφραζομένων

Σχεδόν κάθε πρόγραμμα flex και bison χρησιμοποιεί έναν πίνακα συμβόλων για να κρατάει πληροφορίες σχετικές με τα ονόματα που διαβάζονται στην είσοδο. Ο πίνακας διατρέχεται κάθε φορά που διαβάζεται ένα όνομα και εάν αυτό δεν υπάρχει προστίθεται στην κατάλληλη θέση.

Υπάρχουν αρκετοί τρόποι κατασκευής πινάκων συμβόλων. Οι πιο σημαντικότεροι είναι οι γραμμικές λίστες (linear lists) και οι πίνακες κατακερματισμού (hash tables). Οι γραμμικές λίστες είναι εύκολο να προγραμματιστούν ενώ οι πίνακες κατακερματισμού είναι δυσκολότεροι αλλά γρηγορότεροι. Σημειώνεται επίσης ότι οι δομές δεδομένων που υλοποιούν τους πίνακες συμβόλων θα πρέπει να είναι δυναμικές (να μπορούν να αλλάξουν μέγεθος κατά τη διάρκεια της μεταγλώττισης), διότι διαφορετικά θα πρέπει να επιλέγεται ένα μέγιστο μέγεθος της δομής.

Θα ξεκινήσουμε με ένα απλό πρόγραμμα που θα περιέχει ένα πίνακα συμβόλων για συμφραζόμενα τα οποία θα κρατούν κάθε λέξη ,τα αρχεία που την περιλαμβάνουν και τους αριθμούς γραμμών τους που την περιλαμβάνουν. Το παρακάτω παράδειγμα Figure 1.13 Concordance generator(πχ ευρετήριο) δείχνει το τμήμα τον ορισμών μιας γεννήτρια πινάκων συμφραζομένων(Concordance Generator)

```
/* fb2-4 text concordance */
%option noyywrap nodefault yylineno case-insensitive

/* the symbol table */
%{
struct symbol {                                  /* a word */
  char *name;
  struct ref *reflist;
 };

struct ref {
  struct ref *next;
  char *filename;
  int flags;
  int lineno;
 };

/* simple symtab of fixed size */
#define NHASH 9997
struct symbol symtab[NHASH];

struct symbol *lookup(char*);
void addref(int, char*, char*,int);

char *curfilename;                    /* name of current input
file */
%}
%%
```

Figure 1.13 Concordance generator –Definition Section

Η γραμμή των %options περιέχει τέσσερις παραμέτρους. Η παράμετρος %yylineno λέει στον flex να ορίσει την ακέραια μεταβλητή yylineno που θα διατηρεί το τρέχον αριθμό γραμμής. Αυτό σημαίνει ,ότι κάθε φορά που ο σαρωτής διαβάζει έναν χαρακτήρα νέας γραμμή, το yylineno αυξάνεται, ενώ όταν ο σαρωτής επιστρέφει στην προηγούμενη γραμμή ,το yylineno μειώνεται. Η case-insensitive λέει στο flex να δημιουργήσει έναν σαρωτή που θα αντιμετωπίζει τα πεζά και τα κεφαλαία ίδια. Η noyywrap λέει στο flex να μην καλέσει την wrap μετά το τέλος του αρχείου και τέλος η παράμετρος nodefault δεν επιστρέφει στη έξοδο καμία μη ταιριασμένη είσοδο.

Ο πίνακας συμβόλων είναι απλά ένας πίνακας από δομές συμβόλων ,που η κάθε μια περιλαμβάνει έναν δείκτη για το όνομα και μια λίστα από αναφορές. Οι αναφορές είναι μια λίστα συνδεδεμένη με τους αριθμούς γραμμών και τους δείκτες για το όνομα του αρχείου. Επίσης βλέπουμε στο παράδειγμά μας ότι έχει οριστεί η

μεταβλητή curfilename, η οποία είναι ένας δείκτης(static pointer) που δείχνει το όνομα του τρέχοντος αρχείου και χρησιμοποιείται για την πρόσθεση αναφοράς.

```
%%
/* rules for concordance generator */
/* skip common words */
a |
an |
and |
are |
as |
at |
be |
but |
for |
in |
is |
it |
of |
on |
or |
that |
the |
this |
to                              /* ignore */

[a-z]+(\'(s|t))?          { addref(yylineno, curfilename,
yytext, 0); }
.|\n                           /* ignore everything else */
%%
```

Figure 1.13-a Concordance generator –Rules Section

Αντιλαμβανόμαστε ότι συνήθως τα συμφραζόμενα δεν περιέχουν μικρές κοινές λέξεις, και για αυτό στο παραπάνω παράδειγμα τα πρώτα δεκαεννέα patterns τις ταιριάζουν και τις αγνοούν. Το επόμενο pattern ταιριάζει μια συμβολοσειρά από γράμματα [a-z]+ ακολουθουμένη προαιρετικά από ένα απόστροφο και με ένα s ή t, για να ταιριάξει λέξεις όπως owner's και can't. Κάθε αντιστοιχισμένη λέξη καταχωρείται στην ρουτίνα addref() μαζί με το όνομα του τρέχοντος αρχείου που την περιλαμβάνει καθώς και τον αριθμό γραμμής. Το τελευταίο pattern ταιριάζει και αγνοεί ότι δεν έχει ταιριαστεί από τα προηγούμενα patterns.

```
main(argc, argv)
int argc;
char **argv;
{
int i;
if(argc < 2) {                          /* just read stdin */
```

```
    curfilename = "(stdin)";
    yylineno = 1;
    yylex();
} else
for(i = 1; i < argc; i++) {
    FILE *f = fopen(argv[i], "r");
        if(!f) {
        perror(argv[1]);
        return (1);
                }
   curfilename = argv[i];            /* for addref */
   yyrestart(f);
   yylineno = 1;
   yylex();
   fclose(f);
   }
printrefs();
}
```

Figure 1.13-b Concordance generator –main routine

Στη κύρια ρουτίνα ,ανοίγεται το κάθε αρχείο με την σειρά, χρησιμοποιώντας το yyrestart για να διαβάσει το αρχείο και καλείται η yylex().Οι προσθήκες καθορίζονται στην curfilename με το όνομα του τρέχοντος αρχείου ,που θα χρησιμοποιηθεί στην δημιουργία της λίστα των αναφορών και γίνεται ο καθορισμός του yylineno σε 1 για κάθε αρχείο .Τέλος η ρουτίνα printrefs() βάζει σε αλφαβητική σειρά τον πίνακα συμβόλων και εκτυπώνει τις αναφορές.

1.14 Πολλαπλοί Λεκτικοί Αναλυτές(Lexer) σε ένα πρόγραμμα

Μπορούμε να έχουμε λεκτικούς αναλυτές(Lexers) για δύο εν μέρει ή εντελώς διαφορετικές συντάξεις μιας λεκτικής μονάδας(token) στο ίδιο πρόγραμμα. Για παράδειγμα ένας διαδραστικός διερμηνέας αποσφαλμάτωση(Interactive Debugging Interpreter) μπορεί να έχει ένα Lexer για τον προγραμματισμό γλώσσας και έναν δεύτερο για τις εντολές εντοπισμού σφαλμάτων.

Υπάρχουν δύο βασικές προσεγγίσεις για τον χειρισμών δύο Lexers σε ένα πρόγραμμα:

- Συνδυάζοντας τους σε ένα ενιαίο Lexer
- Ή χρησιμοποιώντας δυο πλήρεις Lexers στο ίδιο πρόγραμμα

1.14.1 Συνδυασμός των Lexers

Για να μπορέσουμε να συνδυάσουμε δύο Lexers σε ένα χρησιμοποιούμε αρχικές καταστάσεις. Όλα τα patterns για του κάθε Lexer έχουν πρώτο συνθετικό ένα μοναδικό σετ από αρχικές καταστάσεις. Όταν ο Lexer ξεκινάει, χρειάζεται να προσθέσεις ένα κομμάτι κώδικά για να θέσεις τον Lexer σε κατάσταση έναρξης για το συγκεκριμένο λεκτικό που χρησιμοποιείται, για παράδειγμα ο παρακάτω κώδικας.

```
%s INITA INITB INITC
%%

%{
      extern first_tok, first_lex;
      if(first_lex) {
        BEGIN first_lex;
        first_lex = 0;
     }
    if(first_tok) {
     int holdtok = first_tok;
     first_tok = 0;
     return holdtok;
     }
%}
```

Figure 1.14.1 Code to put the Lexer into the appropriate Initial State

Στην περίπτωση αυτή, πριν καλέσουμε τον Lexer ,θέτουμε το first_lex σε initial κατάσταση του Lexer. Θα χρησιμοποιούσαμε συνήθως ένα συνδυασμό Lexer σε συνδυασμό με ένα συνδυασμένο πρόγραμμα ανάλυσης yacc, ώστε ο κώδικα να αναγκάζει το αρχικό token(initial token) να κατευθύνει τον parser(αναλυτή), για το ποια γραμματική θα χρησιμοποιήσει.

Το πλεονέκτημα αυτής της προσέγγισης είναι ότι ο αντικειμενικός κώδικας(object code) είναι σχετικά μικρός, δεδομένου ότι υπάρχει ένα αντίγραφο του Lexer code.Επιπλέον τα διαφορικά σετ κανόνων μπορούν να χρησιμοποιήσουν κοινούς κανόνες .

Το μειονέκτημα που θα μπορούσαμε να αναφέρουμε είναι ότι πρέπει να είμαστε προσεκτικοί για να χρησιμοποιήσουμε τις σωστές αρχικές καταστάσεις, αφού δεν μπορούν δυο Lexer να είναι ενεργοποιημένοι σε μία αρχική κατάσταση. Άλλο ένα

μειονέκτημα είναι ότι δεν μπορούμε να χρησιμοποιήσουμε διαφορετικές πηγές εισόδου για διαφορετικούς Lexer.

1.14.2 Πολλαπλοί Lexers

Η άλλη προσέγγιση είναι να περιλαμβάνονται δυο πλήρη Lexers στο ίδιο πρόγραμμα. Το κόλπο είναι να αλλάξουμε τα ονόματα που χρησιμοποιεί ο lex για τις λειτουργίες και τις μεταβλητές του, ώστε οι δύο lexers να μπορούν να δημιουργηθούν ξεχωριστά από το flex και στη συνέχεια να μεταγλωττίζονται μαζί σε ένα πρόγραμμα.

Το Flex παρέχει γραμμή εντολών και πρόγραμμα επιλογής για την αλλαγή του προθέματος που χρησιμοποιείται σχετικά για τα ονόματα του σαρωτή που παράγεται. Για παράδειγμα οι options μπορούν να καθορίσουν τον flex να χρησιμοποιήσει το πρόθεμα "foo" αντί για "yy" και να θέσει τον παραγόμενο σαρωτή στο foolex.c.

%option prefix="foo"

%option outfile="foolex.c"

Αντίστοιχα και σε command line:

$ flex --outfile=foolex.c --prefix=foo foo.l

Είτε έτσι , είτε αλλιώς ,ο παραγόμενος σαρωτής έχει σημείο εισόδου την foolex(),η οποία διαβάζει από το stdio file fooin, και ούτε καθεξής. Κάπως συγκεχυμένα, το Flex παράγει ένα σύνολο από #define μακροεντολών στο μπροστινό μέρος του lexer, που επαναπροσδιορίζουν τα στάνταρ "yy" ονόματα στο επιλεγμένο πρόθεμα. Αυτό μας επιτρέπει να γράφουμε στον lexer ,χρησιμοποιώντας τα στάνταρ ονόματα, αλλά τα εξωτερικά ορατά ονόματα θα χρησιμοποιούν το πρόθεμα που έχει επιλεγεί.

```
#define yy_create_buffer foo_create_buffer
#define yy_delete_buffer foo_delete_buffer
#define yy_flex_debug foo_flex_debug
#define yy_init_buffer foo_init_buffer
#define yy_flush_buffer foo_flush_buffer
#define yy_load_buffer_state foo_load_buffer_state
#define yy_switch_to_buffer foo_switch_to_buffer
#define yyin fooin
#define yyleng fooleng
#define yylex foolex
```

```
#define yylineno foolineno
#define yyout fooout
#define yyrestart foorestart
#define yytext footext
#define yywrap foowrap
#define yyalloc fooalloc
#define yyrealloc foorealloc
#define yyfree foofree
```

Figure 1.14.2 #define macro-Επαναπροσδιορισμός των standard names

ΚΕΦΑΛΑΙΟ 2

Εισαγωγή και Χρήση του Bison

2.1 Εισαγωγή του εργαλείου Bison

Το Bison είναι συμβατό με το εργαλείο το UNIX Yacc και αποτελεί μια βελτιωμένη έκδοση του Yacc.Το Bison αρχικά γράφτηκε από τον Robert Corbett. Ο Richard Stallman έκανε το Bison συμβατό με το Yacc και ο Wilfred Hansen από το πανεπιστήμιο Carnegie Mellon πρόσθεσε την έννοια multi-character string και άλλα χαρακτηριστικά στο εργαλείο. Από τότε, εξελίχθηκαν πολλά νέα χαρακτηριστικά του Bison χάρη της σκληρής δουλειάς εθελοντών.

Το εργαλείο Bison παράγει συντακτικούς αναλυτές που μετατρέπουν γραμματικές χωρίς συμφραζόμενα σε ντετερμινιστικούς LR (διαβάζουν την είσοδο από τα αριστερά στα δεξιά) ή γενικευμένους LR αναλυτές (Στους μεταγλωττιστές χρησιμοποιούνται συνήθως Συντακτικοί Αναλυτές που δεν οπισθοδρομούν, διότι δε χρειάζεται να δοκιμάσουν τους εναλλακτικούς κανόνες της Γραμματικής. Αυτοί οι Συντακτικοί Αναλυτές ονομάζονται ντετερμινιστικοί αναλυτές χρησιμοποιώντας LALR(1) αναλυτές πινάκων (parser tables).

Όταν το Flex αναγνωρίζει κανονικές εκφράσεις, το Bison αναγνωρίζει ολόκληρες γραμματικές. Το flex χωρίζει την είσοδό του σε κομμάτια (Tokens),και στην συνέχεια το Bison παίρνει τα κομμάτια αυτά και τα ομαδοποιεί λογικά, σύμφωνα με την γραμματική που έχει οριστεί. Παρακάτω περιγράφεται σχηματικά η λειτουργία του Bison.

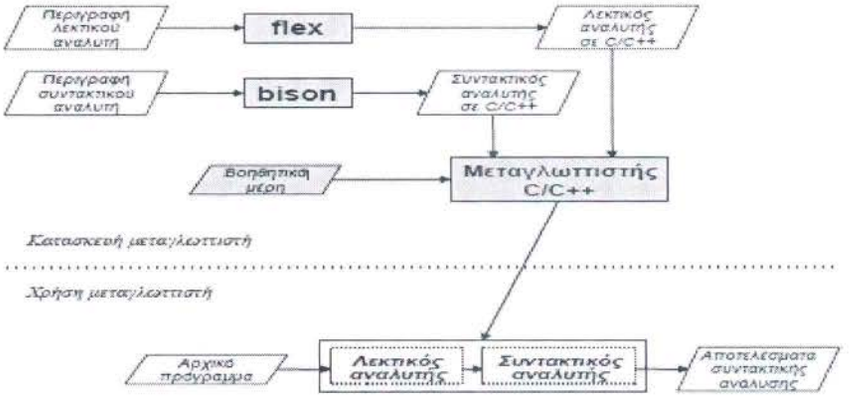

Figure 2.1 Λειτουργιά του Bison

2.2 Συντακτική Ανάλυση-Ταίριασμα Εισόδου

Η συντακτική ανάλυση είναι η διαδικασία ανάλυσης μιας ακολουθίας tokens που καθορίζουν μια γραμματική δομή. Ένα πρόγραμμα συντακτικής ανάλυσης, είναι ένα από τα συστατικά ενός interpreter ή compiler, το οποίο συλλαμβάνει την είσοδο κειμένου και το μετατρέπει σε μια μορφή κατάλληλη για περαιτέρω επεξεργασία και ελέγχει για συντακτικά λάθη. Η συντακτική ανάλυση χρησιμοποιεί συχνά έναν λεκτικό αναλυτή για να δημιουργήσει τα tokens. Οι συντακτικές αναλυτές μπορούν να προγραμματιστεί με το χέρι ή μπορεί να δημιουργηθεί με την χρήση κάποιου εργαλείου(bison).

Κάθε γλώσσα προγραμματισμού έχει κανόνες που περιγράφουν με ακρίβεια την συντακτική δομή των προγραμμάτων. Η σύνταξη αυτή μπορεί να περιγραφεί με γραμματικές μη εξαρτώμενες από τα συμφραζόμενα (context free grammar) ή σε BNF (Backus-Naur-Form) μορφή. Οι γραμματικές προσφέρουν μεγάλα πλεονεκτήματα τόσο στο σχεδιασμό γλωσσών όσο και στη κατασκευή αντίστοιχων μεταγλωττιστών. Μια γραμματική δίνει τη δυνατότητα να περιγραφεί η σύνταξη μιας γλώσσας με ακρίβεια. Από μερικές γραμματικές μπορούμε να κατασκευάσουμε αυτόματα αποτελεσματικούς συντακτικούς αναλυτές.

Η συντακτική ανάλυση ελέγχει αν ένα αρχικό πρόγραμμα ανήκει στη γλώσσα της οποίας η σύνταξη έχει ορισθεί από κάποια συγκεκριμένη Γραμματική χωρίς συμφραζόμενα (Context Free). Στην ίδια φάση κατασκευάζεται και το αντίστοιχο συντακτικό δένδρο που έχει στη ρίζα του το αρχικό σύμβολο της Γραμματικής και στα φύλλα του τις λεξικές μονάδες του αρχικού προγράμματος. Το κομμάτι του μεταγλωττιστή που κάνει αυτή τη δουλειά είναι ο Συντακτικός Αναλυτής.

Η είσοδος του Συντακτικού αναλυτή είναι ένα αρχικό πρόγραμμα σαν μία συμβολοσειρά λεξικών μονάδων και η έξοδός του είναι ένα συντακτικό δένδρο, κάποια μορφή κώδικα τριών διευθύνσεων ή μία ένδειξη ότι το αρχικό πρόγραμμα δεν είναι συντακτικά ορθό.

Το Bison για να μπορέσει να αναλύσει μια δοθείσα είσοδο , πρέπει πρωτίστως να έχει περιγραφθεί με ένα πλαίσιο γραμματικής χωρίς συμφραζόμενα (context-free grammar, είναι η γραμματική όπου ο κάθε κανόνας παράγωγης έχει την μορφή V->w ,όπου V είναι nonterminal symbol και το w είναι ένα string από terminals και/ή nontrminals symbols).Αυτό σημαίνει ότι πρέπει να καθοριστούν συντακτικά

ομαδοποιημένοι κανόνες , τους οποίους θα χρησιμοποιήσει ο αναλυτής. Με λίγα λόγια η γραμματική αποτελείται από μια σειρά κανόνων, με τους οποίους ο αναλυτής αναγνωρίζει την ισχύουσα συντακτικά είσοδο. Στην γλώσσα C μια πρόταση μπορεί να είναι συντακτικά σωστή αλλά σημασιολογικά να μην είναι έγκυρη ,για παράδειγμα σε ένα πρόγραμμα μπορεί να εκχωρηθεί σε ένα string μια μεταβλητή int και να θεωρείται συντακτικά σωστή πρόταση σύμφωνα με τους κανόνες που έχουν καθοριστεί στο συντακτικό αναλυτή. Άρα ο συντακτικός αναλυτής καθορίζει την σύνταξη και όχι την σημασιολογία μιας πρότασης. Η σημασιολογία εξαρτάται αποκλειστικά από τον προγραμματιστή. Παρακάτω περιγράφεται μια έκδοση της γραμματικής που θα χρησιμοποιήσουμε σε επόμενες ενότητες.

statement: NAME '=' expression
expression: NUMBER '+' NUMBER
| NUMBER '−' NUMBER

Η κάθετη γραμμή |, σημαίνει ότι υπάρχουν δύο πιθανότητες για το ίδιο σύμβολο(symbol). Αυτό αποτελεί μα έκφραση (expression) που μπορεί να είναι μία πρόσθεση ή μια αφαίρεση .Το σύμβολο που βρίσκεται αριστερά από την άνω-κάτω τελεία (:), είναι γνωστό ως left-hand side του κανόνα και συχνά το συναντάμε ως LHS, ενώ το δεξί τμήμα είναι γνωστό ως right-hand left του κανόνα και γνωστό ως RHS. Τα σύμβολα που εμφανίζονται στην είσοδο, και επιστρέφονται από τον lexer(λεκτικό αναλυτή) είναι tokens ή terminal symbols.Αυτά που εμφανίζονται στο left-hand side του κανόνα αποτελούν τα nonterminal symbols, ενώ τα σύμβολα που εμφανίζονται στο αριστερό τμήμα του κανόνα μπορεί να είναι μηδέν, terminals και nonterminal symbols. Τα terminals και τα nonterminal symbols πρέπει να είναι διαφορετικά. Είναι λάθος να γράψουμε ένα κανόνα με ένα token να βρίσκεται στην αριστερά πλευρά του.

Nonterminal symbol Terminal symbol

expression: NUMBER + expression ;

Στις γραμματικές του Bison η έννοια symbols αντιπροσωπεύει τις γραμματικές κατηγοριοποιήσεις ή ταξινομήσεις της γλώσσας δηλαδή τις «λέξεις» της γραμματικής. Η έννοια terminal symbols αντιπροσωπεύει μια κλάση από ισοδύναμα

tokens, δηλαδή τύποι λεκτικών μονάδων που σχηματίζουν μια συμβολοσειρά, ενώ nonterminal symbols αντιπροσωπεύει μια κλάση από συντακτικές ισοδύναμες ομάδες δηλαδή είναι ειδικά σύμβολα που υποδηλώνουν συμβολοσειρές οι οποίες εκφράζουν την συντακτική δομή της γλώσσας.

Ένας τρόπος για να καταλάβουμε πώς γίνεται η ανάλυση μιας εισόδου σύμφωνα με το Bison, είναι η αναπαράσταση ενός δέντρου(parse tree). Για παράδειγμα ,εάν έχουμε ως είσοδο fred=12+13 και χρησιμοποιήσουμε την γραμματική που συντάξαμε παραπάνω, το δέντρο θα έχει την παρακάτω μορφή(Figure 2.2).

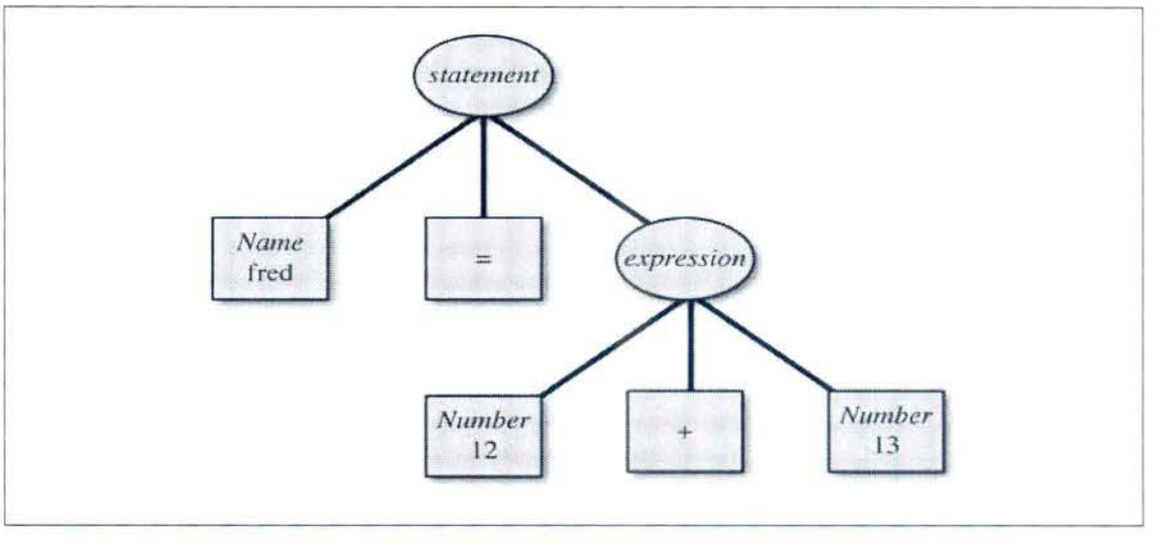

Figure 2.2 Expression parse tree

Στο παράδειγμα το 12+13 αποτελεί την έκφραση(expression) και fred=expression αποτελεί την δήλωση(statement). Κάθε γραμματική περιλαμβάνει ένα start symbol, το οποίο είναι μοναδικό και αποτελεί την ρίζα του δέντρου. Στην γραμματική που χρησιμοποιήσαμε στο παραπάνω παράδειγμα ,το start symbol είναι η δήλωση(statement). Οι κανόνες μιας γραμματικής μπορούν άμεσα ή έμμεσα να παραπέμπονται στους ίδιου τους κανόνες. Η δυνατότητα αυτή είναι πολύ σημαντική γιατί μπορούμε να αναλύσουμε αυθαίρετα μεγάλες ακολουθίες εισόδου. Επεκτείνοντας την γραμματική που χρησιμοποιήσαμε παραπάνω μπορούμε να χειριστούμε μεγαλύτερες αριθμητικές εκφράσεις.

expression: NUMBER

| expression + NUMBER

| expression – NUMBER

Σύμφωνα με τους παραπάνω κανόνες μπορούμε να αναλύσουμε μεγαλύτερες αριθμητικές εκφράσεις όπως fred = 14 + 23 − 11 + 7 εφαρμόζοντας επανειλημμένα τους κανόνες έκφρασης όπως φαίνεται στο σχήμα Figure 2.2-a. Όσο ο αναλυτής διαβάζει τα tokens , κάθε ένα token που διαβάζεται και δεν ολοκληρώνει έναν κανόνα, το ωθεί σε μια εσωτερική στοίβα και αλλάζει σε μια νέα κατάσταση η οποία αντικατοπτρίζει το token που έχει διαβαστεί. Αυτή η ενέργεια είναι γνωστή ως shift. Όταν βρει όλα τα σύμβολα που αποτελούν το δεξιό τμήμα του κανόνα (left-hand side), βγάζει από την στοίβα τα σύμβολα του δεξιού τμήματος, ωθεί στην στοίβα το σύμβολο του αριστερού τμήματος του κανόνα και αλλάζει σε νέα κατάσταση η οποία αντικατοπτρίζει το νέο σύμβολο της στοίβας. Η διαδικασία αυτή είναι γνωστή ως reduction, αφού μειώνει τον αριθμό αντικειμένων από την στοίβα. Όταν ο parser μειώνει έναν κανόνα, εκτελεί τον κώδικα χρήστη που τον ακολουθεί.

Shift:Ένας parser μετατοπίζει ένα σύμβολο εισόδου, τοποθετώντας το στη στοίβα αναλυτή με την προσδοκία ότι το σύμβολο θα ταιριάξει με έναν από τους κανόνες της γραμματικής.

Reduce: Όταν η είσοδος ταιριαστεί με την λίστα των συμβόλων του RHS τμήματος του κανόνα, ο αναλυτής κάνει μείωση του κανόνα μετακινώντας τα σύμβολα του RHS τμήματος από την στοίβα και αντικαθιστώντας τα με τα σύμβολα του LHS.

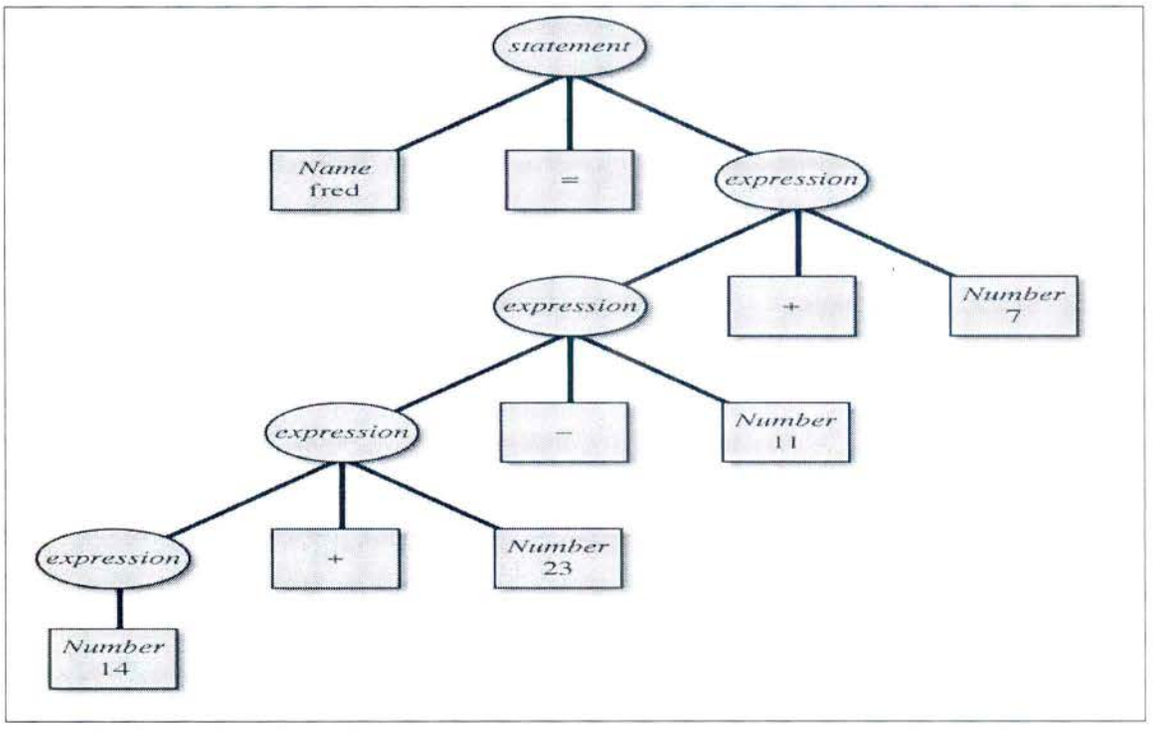

Figure 2.2-a Parse tree with recursive rules

Παρακάτω εξηγούμε πως ο parser αναλύει την είσοδο fred=12+13 χρησιμοποιώντας τους απλούς κανόνες του Figure 2.2.Ο parser ξεκινάει την μετατόπιση των tokens (shift) με την σειρά στην εσωτερική στοίβα:

fred shift

fred = shift

fred = 12 shift

fred = 12 + shift

fred = 12 + 13 reduce

Στο σημείο αυτό μειώνει(reduction) τον κανόνα expression: NUMBER + NUMBER, και βγάζει από την στοίβα το 12,+,13 και το αντικαθιστά με το expression.

fred=expression

Στη συνέχεια μειώνει τον κανόνα statement: NAME = expression και βγάζει από την στοίβα το fred,= και το expression και το αντικαθιστά με το statement.Έχουμε φτάσει πλέον στο τέλος της εισόδου και η στοίβα έχει μειωθεί περιέχοντας το start symbol(statement).Επομένως η είσοδο απεδείχθη έγκυρη σύμφωνα με την γραμματική. Ακριβώς την ίδια διαδικασία ακολουθούμε και με την είσοδο fred=12+23-11+7.Αναλυτικότερα μετατοπίσει τα tokens(shift) στην στοίβα:

fred shift

fred= shift

fred=12 reduce

Στο σημείο αυτό ο parser μειώνει τα αντικείμενα σύμφωνα με τον κανόνα expression:NUMBER και βγάζει από την στοίβα το 12 και το αντικαθιστά με το expression.

fred=expression

fred=expression+ shift

fred=expression+23 reduce

Ο αναλυτής μειώνει και σε αυτό το σημείο τα αντικείμενα σύμφωνα με τον κανόνα expression:expression+NUMBER και τα βγάζει από την στοίβα και τα αντικαθιστά

με την expression.

fred=expression

fred=expression-11 reduce

Μειώνει τα αντικείμενα αφού ικανοποιείται ο κανόνας expression:expression-NUMBER και τα βγάζει από την στοίβα και τα αντικαθιστά με την expression.

fred=expression

fred=expression+ shift

fred=expression+7 reduce

Και σε αυτό το σημείο ικανοποιείται ο κανόνας expression:expression+NUMBER, οπότε τα αντικείμενα βγαίνουν από την στοίβα και αντικαθιστώνται με το expression.

fred=expression

Στη συνέχεια μειώνει τον κανόνα statement: NAME = expression και βγάζει από την στοίβα τα αντικείμενα fred,= και το expression και τα αντικαθιστά με το statement. Διαπιστώνουμε ότι και σε αυτή την περίπτωση ότι είσοδο απεδείχθη έγκυρη σύμφωνα με την γραμματική.

2.2.1 Τύποι συγκρούσεων(Types of Conflicts)

Στην 2.2 ενότητα αναφερθήκαμε στις διαδικασίες shift και reduction που χρησιμοποιεί το Bison κατά την ανάλυση μιας εισόδου. Υπάρχουν δύο είδη συγκρούσεων που μπορεί να συμβούν όταν το Bison προσπαθεί να δημιουργήσει έναν αναλυτή : shift/reduce και reduce/reduce.

H Shift/Reduce Conflict προκύπτει όταν υπάρχουν δύο πιθανές αναλύσεις για μια string είσοδο και η μία ανάλυση ολοκληρώνει τον κανόνα (reduce) και η άλλη ανάλυση δεν ολοκληρώνει τον κανόνα (shift). Για παράδειγμα :

e: 'X'

| e '+' e

;

Για την είσοδο X+X+X υπάρχουν δύο πιθανές αναλύσεις: (X+X)+X ή X+(X+X). Λαμβάνοντας την επιλογή reduce ο parser θα επιλέξει την πρώτη ανάλυση και λαμβάνοντας την επιλογή shift ο parser θα επιλέξει την δεύτερη ανάλυση. Συνήθως το Bison χρησιμοποιεί την επιλογή Shift εκτός αν ο χρήστης παρέμβει με έναν operator που θα καθορίζει την προτεραιότητα των δηλώσεων.

Η reduce/reduce Conflict προκύπτει όταν το ίδιο token ολοκληρώνει δύο διαφορετικούς κανόνες. Για παράδειγμα:

thing: expa|expb;

expa:X;

expb:X;

Το token X μπορεί να είναι expa ή expb. Το bison κάνει ελάττωση(reduce) με τον κανόνα που έχει περιγραφεί πρώτος στο αρχείο της γραμματικής.
Τα περισσότερα reduce/reduce Conflicts είναι λιγότερο προφανής από το παραπάνω, αλλά συνήθως δείχνουν τα λάθη γραμματικής.

2.2.2 Διαχειριστές Προτεραιότητα (Operator Precedence)

Bison προσφέρει έναν έξυπνο τρόπο για να περιγράψει το προβάδισμα χωριστά από τους κανόνες της γραμματικής, η οποία καθιστά την γραμματική και τον parser μικρότερους και πιο εύκολα να διατηρηθούν.

Αν η είσοδό μας είναι 2+3*4 , και η γραμματική που θα χρησιμοποιήσουμε είναι:
exp: exp '+' exp
| exp '*' exp
ενδεχομένως η είσοδό μας μπορεί να αντιμετωπιστεί και ως (2+3)*4 ή 2+(3*4) κατά την ανάλυση.

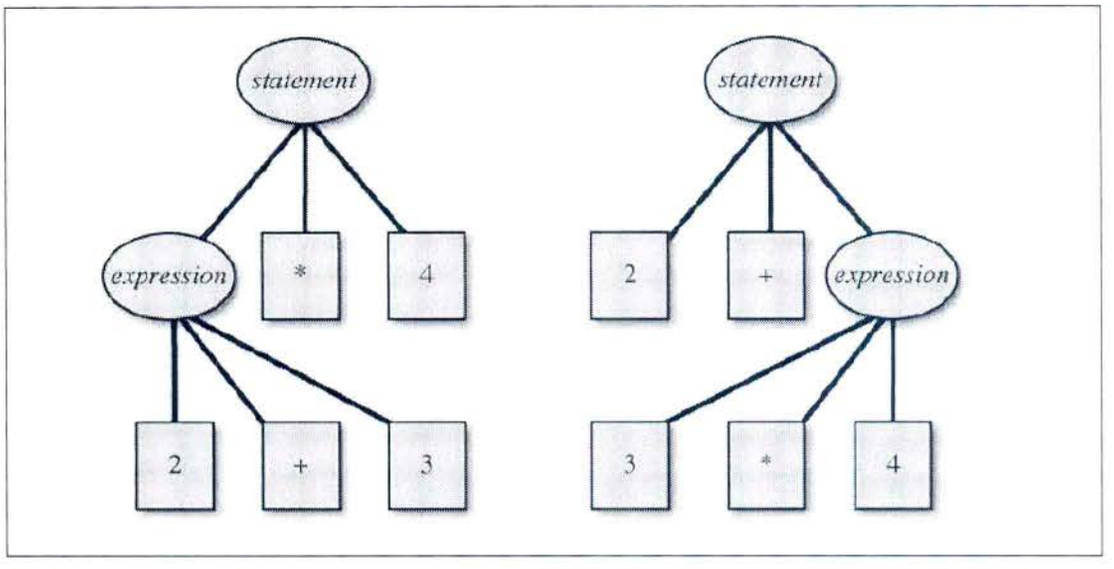

Figure 2.2.2 Two expression parse trees

Αν μεταγλωττίσουμε το παραπάνω παράδειγμα, το Bison θα μας ενημερώσει για shift\reduce συγκρούσεις, οι οποίες αποτελούν καταστάσεις. Το πρόβλημα που δημιουργείται είναι ότι το Bison δεν ξέρει αν πρέπει να κάνει shift σε ένα token στην στοίβα ή να κάνει reduce έναν κανόνα. Αναλύοντας την είσοδο 2+3*4 ακολουθούμε τα συγκεκριμένα βήματα (Για λόγους συντομίας χρησιμοποιούμε το E αντί για το exp) :

2 shift NUMBER

E reduce E → NUMBER

E + shift +

E + 3 shift NUMBER

E + E reduce E → NUMBER

Στο σημείο αυτό, ο parser βλέπει το * και θα μπορούσε είτε να κάνει reduce το 2+3 χρησιμοποιώντας τον κανόνα: exp: exp '+' exp

Είτε να κάνει shift το * έτσι ώστε να μπορέσει αργότερα να κάνει reduce με τον κανόνα: exp: exp '*' exp

Στην περίπτωση αυτή θα μπορούσαμε να λύσουμε το πρόβλημα ενημερώνοντας το Bison για την προτεραιότητα και την προσεταιριστικότητα των τελεστών. Η προτεραιότητα ελέγχει ποιος τελεστής θα εκτελεστεί πρώτος. Η προσεταιριστικότητα ελέγχει την ομαδοποίηση των τελεστών που βρίσκονται στο ίδιο επίπεδο προτεραιότητας. Οι τελεστές μπορούν να ομαδοποιηθούν στα αριστερά, για παράδειγμα a-b-c στην C σημαίνει a-(b-c) ή στα δεξιά για παράδειγμα a=b=c στην C σημαίνει a=(b=c).

Το Bison μας επιτρέπει να ορίσουμε την προτεραιότητα, προσθέτοντας τις παρακάτω γραμμές στο τμήμα των ορισμών που θα ενημερώνει πώς να επιλύονται οι συγκρούσεις:

```
%left '+' '-'
%left '*' '/'
%nonassoc '|' UMINUS
%type <a> exp
%%
...
exp: exp '+' exp { $$ = newast('+', $1,$3); }
| exp '-' exp { $$ = newast('-', $1,$3);}
| exp '*' exp { $$ = newast('*', $1,$3); }
| exp '/' exp { $$ = newast('/', $1,$3); }
| '|' exp { $$ = newast('|', $2, NULL); }
| '(' exp ')' { $$ = $2; }
| '-' exp %prec UMINUS{ $$ = newast('M', NULL, $2); }
| NUMBER { $$ = newnum($1); }
;
```

Κάθε μια από τις δηλώσεις ορίζουν ένα επίπεδο προτεραιότητας. Οι δηλώσεις %left, %nonassoc και %right ορίζουν την σειρά προτεραιότητας από το χαμηλό στο υψηλό επίπεδο. Τα σύμβολα + και – έχουν μικρότερη προτεραιότητα από τα σύμβολα * και /. Τα σύμβολα | και UMINUS δεν έχουν προσεταιριστικότητα και έχουν την μεγαλύτερη προτεραιότητα. Όταν εμπλέκονται κανόνες σε μια σύγκρουση, το Bison συμβουλεύεται τον πίνακα προτεραιότητας για την επίλυση του.
Ο «τροποποιητής» %prec δηλώνει την προτεραιότητα ενός συγκεκριμένου κανόνα, καθορίζοντας ένα τερματικό σύμβολο το οποίο η προτεραιότητα θα πρέπει να χρησιμοποιείται για τον εν λόγω κανόνα. Στο παράδειγμά μας, ο κανόνας '-' exp %prec UMINUS δηλώνει ότι ο μοναδικός τελεστής – ,παρόλο που έχει χαμηλότερη προτεραιότητα, θα αντιμετωπιστεί με την μεγαλύτερη προτεραιότητα για την ικανοποίηση του συγκεκριμένου κανόνα.

2.3 Bison parsers-Μέθοδοι Ανάλυσης

Οι Bison parsers συνήθως χρησιμοποιούν δύο μέθοδοι ανάλυσης την LALR(1) (Look Ahead Left to Right with a one-token lookahead) και την μέθοδο GLR (Generalized Left to Right).Οι περισσότεροι parsers χρησιμοποιούν την μέθοδο LARL(1), η οποία είναι λιγότερη ισχυρή, αλλά γρηγορότερη και εύκολη στην χρήση σε σχέση με την GLR.

Παρόλο που η LARL είναι αρκετά ισχυρή, μπορούμε να γράψουμε γραμματικές που δεν μπορεί να τις διαχειριστεί. Δεν μπορεί να χειριστεί διφορούμενες γραμματικές, δηλαδή γραμματικές που ταιριάζουν την ίδια είσοδο με περισσότερα από ένα parse tree. Επίσης δεν μπορεί να χειριστεί γραμματικές που χρειάζονται περισσότερα από ένα token για να μπορέσει να προχωρήσει στο ταίριασμα του κανόνα. Για παράδειγμα:

phrase: cart_animal AND CART

| work_animal AND PLOW

cart_animal: HORSE | GOAT

work_animal: HORSE | OX

Η παραπάνω γραμματική δεν είναι διφορούμενη, αφού υπάρχει ένα πιθανό parse tree για κάθε έγκυρη είσοδο. Από τη άλλη , το Bison δεν μπορεί να διαχειριστεί την γραμματική γιατί απαιτούνται δύο σύμβολα για να προχωρήσει στην μεταγλώττιση. Η μέθοδο ανάλυσης LARL(1) δεν είναι η καταλληλότερη να χειριστεί συγκρούσεις. Ειδικότερα, αν είχαμε ως είσοδο HORSE AND CART , η γραμματική δεν μπορεί το HORSE να το αναγνωρίσει ως cart_animal ή work_animal μέχρι να διαβαστεί το CART . Το Bison σύμφωνα με την LARL(1) διαβάζει το HORSE και επιπλέον μπορεί να δει ένα σύμβολο μπροστά δηλαδή το AND , όμως δεν είναι αρκετό για να καταλάβει αν το HORSE είναι cart_animal ή work_animal.Διαπιστώνουμε ότι σε αυτή την περίπτωση η μέθοδο LARL(1)δεν μπορεί να είναι χρήσιμη. Αν αλλάζαμε τον πρώτο κανόνα ως εξής:

phrase: cart_animal CART

| work_animal PLOW

Το Bison μπορεί να διαχειριστεί την είσοδο, αφού μπορεί να δει ένα token μπροστά για να ελέγξει αν η είσοδο HORSE ακολουθείται από το CART ή ακολουθημένο από

το PLOW στην περίπτωση που είναι work_animal. Αν η είσοδο ήταν HORSE CART οι κανόνες θα επαληθευόντουσαν χωρίς να δημιουργηθούν συγκρούσεις. Στην πράξη, οι κανόνες που μπορεί να χειριστεί το Bison δεν είναι τόσο περίπλοκοι και δεν δημιουργούν σύγχυση όπως στο παραπάνω παράδειγμα. Ένας λόγος είναι, ότι το Bison γνωρίζει ακριβώς ποια γραμματική μπορεί να αναλύσει και ποιες όχι.

Από την άλλη, η μέθοδος GLR, επιτρέπει στους Bison parsers να διαχειριστούν οποιανδήποτε γραμματική και να δημιουργήσουν έναν parser που θα αναλύει την επίλυση συγκρούσεων κατά την μεταγλώττιση. Μπορούν να διαχειριστούν διφορούμενες γραμματικές Επομένως θα μπορούσε το παραπάνω παράδειγμα, χωρίς να γίνει η αλλαγή του κανόνα phrase ,να αναλυθεί από ένα parser GLR χωρίς προβλήματα. Όμως είναι πολύ αργή μέθοδο και δύσκολη στη χρήση.

2.4 Δομή ενός προγράμματος Bison

Η γραμματική του Bison απαιτεί την σύνταξη τριών τμημάτων: το τμήμα των ορισμών, το τμήμα των κανόνων και το τμήμα κώδικα.

... definition section ...

%%

... rules section ...

%%

... user subroutines section ...

%{
Δηλώσεις C/C++
%}

Δηλώσεις bison

%%

Τα τμήματα χωρίζονται από γραμμές που αποτελούνται από δύο σύμβολα τοις εκατό(%%).Το πρώτο και το δεύτερο τμήμα απαιτούνται, αν και ένα τμήμα μπορεί να είναι κενός. Το τρίτο τμήμα καθώς και το %% μπορούν να παραληφθούν.

Το τμήμα των ορισμών περιέχει ορισμούς macro και δηλώσεις μεταβλητών και συναρτήσεων που χρησιμοποιούνται στις ενέργειες (actions) που λαμβάνουν χώρα όταν αναγνωρίζεται ένας γραμματικός κανόνας από τον συντακτικό αναλυτή. Οι ορισμοί και οι δηλώσεις τυπώνονται στην αρχή του παραγόμενου C αρχείου πριν τον ορισμό της συνάρτησης yyparse που υλοποιεί την συντακτική ανάλυση της

περιγραφόμενης γλώσσας. Είναι επίσης δυνατή η χρήση '#include' για τη λήψη δηλώσεων συναρτήσεων και μεταβλητών που περιλαμβάνονται σε header αρχεία. Εάν δεν υπάρχουν C δηλώσεις, είναι δυνατή η παράλειψη των διαχωριστών '%{' και '%}'. Μπορεί να υπάρχουν δηλώσεις όπως %union, %start, %token, %type, %left, %right, και %nonassoc. Στο τμήμα αυτό μπορεί να περιέχει σχόλια της μορφής της C ,που περιβάλλονται από */ και /*.

Το τμήμα ορισμών και συγκεκριμένα στις δηλώσεις Bison ,περιέχει δηλώσεις για τον καθορισμό τερματικών και μη τερματικών συμβόλων, προτεραιοτήτων τελεστών, τύπων δεδομένων των σημασιολογικών τιμών κ.τ.λ. Όλα αυτά είναι προαιρετικά ,το τμήμα αυτό μπορεί να είναι κενό.

Δήλωση των τερματικών συμβόλων	Δηλώσεις των μη τερματικών συμβόλων
%token TK_NAME **TK_NAME** :Όνομα τερματικού συμβόλου	%type expression **expression:** όνομα μη τερματικού συμβόλου

Figure 2.4.1 Δηλώσεις Συμβόλων

Το τμήμα των κανόνων περιλαμβάνει κανόνες γραμματικής και δράσεις που περιέχουν κώδικα C. Η περιγραφή της γραμματικής της γλώσσας γίνεται με κανόνες παραγωγής διατυπωμένους στην εξής μορφή:

Αριστερό τμήμα : Δεξιό τμήμα

Παράδειγμα

arithmetic_expr: NUM

|arithmetic_expr'+'arithmetic_expr

|arithmetic_expr'-'arithmetic_expr

|arithmetic_expr'*'arithmetic_expr

|arithmetic_expr'/'arithmetic_expr

|'('arithmetic_expr ')'

;

Στο τρίτο τμήμα , το Bison αντιγράφει αυτολεξεί τα περιεχόμενα που βρίσκονται στο τμήμα κώδικα ,στο αρχείο C που παράγεται μετά την εκτέλεση του Bison . Το τμήμα αυτό περιέχονται οι ρουτίνες που καλούνται από τις δράσεις. Χρησιμεύει συνήθως για τον ορισμό βοηθητικών συναρτήσεων που καλούνται από τις διάφορες σημασιολογικές ρουτίνες.

Στο παρακάτω παράδειγμα ενός προγράμματος συντακτικού αναλυτή Bison μπορούμε να καταλάβουμε την δομή και τον διαχωρισμό των τμημάτων. Αρχικά συναντάμε το τμήμα των ορισμών με τις δηλώσεις και στην συνέχεια την γραμματική. Στο τμήμα των κανόνων ,η γραμματική περιλαμβάνει πρόσθεση, αφαίρεση, πολλαπλασιασμό, διαίρεση, εισαγωγή παρενθέσεων και αναγνώριση λαθών. Και στο τρίτο μέρος συναντάμε την main που καλεί την yyparse()για να κάνει την ανάλυση της εισόδου που θα της δοθεί.

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

%token NUMBER
%%
lines: lines expr '\n'{ printf("%g\n", $2); }
      | lines '\n'
      | /* e */
      | error '\n'{ yyerror("reenter last line:"); yyerrok(); }
          ;
expr: expr '+' term           { $$ = $1 + $3; }
      | expr '-' term         { $$ = $1 - $3; }
      | term
      ;

term: term '*' factor   { $$ = $1 * $3; }
      | term '/' factor { $$ = $1 / $3; }
      | factor
      ;
factor:'(' expr ')'           { $$ = $2; }
       | '(' expr error { $$ = $2; yyerror("missing ')'"); yyerrok();
}
       | '-' factor           { $$ = -$2; }
       | NUMBER
      ;
%%
int main(void)
{
      return yyparse();
}
```

Figure 2.5 Παράδειγμα Συντακτικού Αναλυτή

2.5Ειδικοί χαρακτήρες που χρησιμοποιεί το Bison

% Η γραμμή με δύο σύμβολα τοις εκατό διαχωρίζουν τα τμήματα της γραμματικής του Bison. Όλες οι δηλώσεις στο τμήμα τον ορισμών ξεκινούν με %, including %{ %}, %start, %token, %type, %left, %right, %nonassoc, και %union.

$ Στις ενέργειες το σύμβολο του δολαρίου εισάγει μια τιμή αναφοράς, για παράδειγμα $3,για την τιμή του τρίτου συμβόλου στο δεξί τμήμα του κανόνα.

@ Στις δράσει ,το σύμβολο @ εισάγει ένα σημείο αναφοράς(location), όπως το @2 δηλώνει την θέση του δεύτερου συμβόλου του δεξί τμήματος του κανόνα(RHS).

' Τα literal token περικλείονται σε μονά εισαγωγικά, για παράδειγμα '\'.

" Το Bison επιτρέπει μέσα σε διπλά εισαγωγικά να δηλώνουμε συμβολοσειρές ως ψευδώνυμα για tokens, όπως "!=".

< > Σε μια τιμή αναφοράς μπορούμε να παραλείψουμε τον προεπιλεγμένο τύπο της τιμής αναφοράς εσωκλείοντας το όνομα του τύπου σε < >,για παράδειγμα $<type>3.

{ } Στις δράσεις ο κώδικας C περικλείεται σε άγκιστρα.

; Στο τέλος του κάθε κανόνα περιλαμβάνει ερωτηματικό, εκτός αν ακολουθείται κάθετη γραμμή(|).

| Όταν δύο διαδοχικές κανόνες έχουν την ίδια αριστερή πλευρά, ο δεύτερος κανόνας μπορεί να αντικαταστήσει το σύμβολο και την άνω κάτω τελεία με μια κάθετη γραμμή.

: Σε κάθε κανόνα το αριστερό τμήμα του κανόνα ακολουθείται με άνω κάτω τελεία.

_ Τα σύμβολα μπορούν να περιλαμβάνουν κάτω παύλες μαζί με γράμματα, αριθμούς και διαστήματα.

. Τα σύμβολα μπορούν να περιλαμβάνουν τελείες μαζί με γράμματα, αριθμούς και παύλες. Αυτό μπορεί να δημιουργήσει προβλήματα ,επειδή C identifiers δεν μπορούν να περιλαμβάνουν τελείες.

$end Η γραμματική του Bison χρησιμοποιεί το ψευδο_token $end για να δηλώσει το τέλος της εισόδου

2.6Σύνταξη Κανόνων

Η γραμματική του Bison απαιτεί την σύνταξη κανόνων. Η περιγραφή της γραμματικής της γλώσσας γίνεται με κανόνες παραγωγής διατυπωμένους σε BNF γενική μορφή. Κάθε κανόνας ξεκινάει με ένα non-terminal σύμβολο, με άνω κάτω τελεία και ακολουθείται από μια λίστα από σύμβολα(μπορεί να είναι και κενή), literal

tokens και ενέργειες. Στο τέλος του κανόνα ακολουθείται ένα ελληνικό ερωτηματικό, αν και το ερωτηματικό μπορεί να θεωρηθεί τεχνικά προαιρετικό. Για παράδειγμα:
exp: A '+'B ;
Ο κανόνας λέει ότι το exp είναι non-terminal και αποτελείται από τα terminal σύμβολα(tokens) **A**,+ και το σύμβολο **B** (τα σύμβολα **A και B** πρέπει να ορισθούν σε άλλο σημείο της γραμματικής). Όπως έχουμε αναφέρει παραπάνω το αριστερό τμήμα του κανόνα, στην περίπτωσή μας το exp: , καλείται LHS και το δεξί τμήμα του κανόνα, στην περίπτωσή μας το **A '+'B**,καλείται RHS. Το δεξί τμήμα του κανόνα μπορεί να είναι κενό.
Το κενό (whitespace) στον κανόνα είναι σημαντικός για των διαχωρισμό των συμβόλων, επομένως μπορούμε να προσθέσουμε όσο κενό χώρο θέλουμε μεταξύ των συμβόλων ,χωρίς να επηρεάσει τον κανόνα.
Εάν πολλοί διαδοχικοί κανόνες γραμματικής έχουν ίδιο το LHS,ο δεύτερος και οι επόμενοι κανόνες μπορούν να ξεκινήσουν με κάθετη γραμμή(|), αντί όνομα του non-terminal και άνω κάτω τελεία. Τα παρακάτω δύο κομμάτια είναι ισοδύναμα:

exp: number '+' number
| number '-' number;

Το ερωτηματικό παραλείπεται πριν την κάθετη γραμμή

exp: number '+' number ;
exp: number '-' number ;

Αντιθέτως, κανόνες πού έχουν το ίδιο RHS δεν χρειάζεται να εμφανίζονται μαζί.

2.7Ενέργειες (Actions)

Η σημασιολογία μια γλώσσας ορίζεται από τις ενέργειες που εκτελούνται. Μια ενέργεια είναι κώδικας της C που εκτελείται όταν το Bison ταιριάξει τον αντίστοιχο κανόνα γραμματικής. Η ενέργεια πρέπει να είναι μια σύνθετη εντολή της C,για παράδειγμα:

date: month '/' day '/' year { printf("date found"); } ;

Οι ενέργειες μπορεί να παραπέμπονται σε τιμές αναφοράς που είναι συνδεδεμένες με τα σύμβολα του κανόνα, χρησιμοποιώντας το σύμβολο του δολαρίου ($) ακολουθημένο με έναν αριθμό. Η ψευδο-μεταβλητή $$ αναπαριστά τη σημασιολογική τιμή αναφοράς του μη τερματικού συμβόλου που βρίσκεται αριστερά του ' : ' σε έναν κανόνα(LHS). Οι τιμές των τερματικών και μη τερματικών

συμβόλων που βρίσκονται δεξιά του ' : '(RHS) αναφέρονται σαν $1, $2 και ούτω καθεξής. Για παράδειγμα :

```
exp: ...
     | exp '+' exp
     { $$ = $1 + $3; }                                    action

date: month '/' day '/' year { printf("date %d-%d-%d found", $1, $3, $5); };
      ↓      ↓      ↓      ↘
      $$     $1     $3     $5
```

Για κανόνες που δεν περιέχουν ενέργειες, το Bison Χρησιμοποιεί μια Default value.

{$$=$1;}

Εάν έχουμε έναν κανόνα χωρίς RHS σύμβολα και το LHS περιέχει ένα δηλωμένο τύπο, θα πρέπει να γράψουμε μια ενέργεια για να ορίσουμε την τιμή του**(value).**

2.7.1 Ενσωματωμένες Ενέργειες

Αν και η τεχνική ανάλυσης του Bison επιτρέπει τις ενέργειες μόνο στο τέλος του κανόνα, το Bison μπορεί επίσης να προσημειώσει ενέργειες ενσωματωμένες στο πλαίσιο του κανόνα. Αν γράψουμε μια ενέργεια μέσα στον κανόνα, το Bison θα επινοήσει έναν κανόνα με το δεξιό τμήμα του να είναι άδειο και το αριστερό τμήμα αποτελούμενο από όνομα (μην ξεχνάμε ότι statement:Name=exp), κάνοντας την ενσωματωμένη ενέργεια σε εκτελέσιμη ενέργεια για αυτόν τον κανόνα, και αντικαθιστώντας την ενέργεια στον αρχικό κανόνα με το όνομα της. Για παράδειγμα:

```
thing: A { printf("seen an A"); } B ;
          ↘ Ενσωματωμένη ενέργεια
```

Για να γίνει αντιληπτό ,ας θεωρήσουμε ότι η είσοδο είναι AB.Ο parser μόλις πάρει το Α από την είσοδο θα τρέξει αυτόματα η action και θα συνεχίσει την ανάλυση, δηλαδή θα δεί το επόμενο token B και θα ολοκληρώσει το ταίριασμα του κανόνα.

Η ενσωματωμένη ενέργεια αντιμετωπίζεται ως σύμβολο μέσα στο κανόνα και η τιμή της είναι διαθέσιμη στο τέλος του κανόνα-ενέργεια όπως γίνεται σε κάθε σύμβολο:

```
thing: A { $$ = 17; } B C
{ printf("%d", $2); }
;
```

Το παράδειγμα τυπώνει ¨17¨.Οποιαδήποτε ενέργεια μπορεί να αναφέρεται στην τιμή αναφοράς του συμβόλου A ως $1, στο τέλος του κανόνα-ενέργεια αναφέρεται η τιμή της ενσωματωμένης ενέργειας ως $2 και μπορεί να αναφέρεται στην τιμή ανάφορας του B ως $3 και του C ως $4.

Οι ενσωματωμένες ενέργειες μπορούν να προκαλέσουν συγκρούσεις(shift/reduce ή reduce/reduce conflicts). Για παράδειγμα η παρακάτω γραμματική δεν δημιουργεί κανένα πρόβλημα:

%%

thing: abcd | abcz ;

abcd: 'A' 'B' 'C' 'D' ;

abcz: 'A' 'B' 'C' 'Z' ;

Αν προσθέσουμε μια ενσωματωμένη ενέργεια στην παραπάνω γραμματική, θα δημιουργήσει shift/reduce σύγκρουση.

%%

thing: abcd | abcz ;

abcd: 'A' 'B' { somefunc(); } 'C' 'D' ;

abcz: 'A' 'B' 'C' 'Z' ;

Στην πρώτη περίπτωση ο parser δεν χρειάζεται να αποφασίσει αν θα αναλύσει το abcd ή το abcz μέχρι να δει και τέσσερα token.Στην δεύτερη περίπτωση, ο parser θα πρέπει να αποφασίσει όταν αναλύσει το B, όμως στο σημείο αυτό δεν έχει δει «αρκετά» από την είσοδο για να αποφασίσει ποιος κανόνας θα το αναλύσει. Εάν η ενσωματωμένη action ήταν μετά το C ,τότε δεν θα υπήρχε πρόβλημα ,δεδομένου ότι το Bison βλέπει ένα token μπροστά ,στο παράδειγμά μας θα έβλεπε το D ή Z επόμενο token.

Ο κανόνας που ο parser αρχίζει να αναλύει πρώτο ονομάζεται αρχικός κανόνας(start rule). Εάν θελήσουμε να ξεκινήσει ο parser με κάποιον άλλο κανόνα, πρέπει στο τμήμα των ορισμών (δηλώσεις Bison) να γράψουμε το εξής:

%start name

Να ξεκινάει από τον κανόνα **name**, ως όνομα του κανόνα ορίζεται το non-terminal που βρίσκεται στο αριστερό μέρος του(LHS).

2.8 Αναδρομικοί κανόνες

Ένας κανόνας ονομάζεται αναδρομικός, όταν στο αποτέλεσμα του εμφανίζεται μη τερματικό σύμβολο και επίσης εμφανίζεται στο δεξιό τμήμα. Σχεδόν όλες οι γραμματικές Bison χρησιμοποιούν την αναδρομή, γιατί είναι ο μόνος τρόπος για να ορίσουμε μια συχνότητα από οποιοδήποτε αριθμό για ένα συγκεκριμένο πράγμα. Στο παρακάτω παράδειγμα αναλύεται μια μη κενή λίστα από εκφράσεις που διαχωρίζονται από κόμμα., με το σύμβολο expr να ορίζεται σε άλλο σημείο της γραμματικής:

```
exprlist: expr
        | exprlist ',' expr
        ;
```

Είναι επίσης δυνατόν να έχουμε αμοιβαίους αναδρομικούς κανόνες δηλαδή το μη τερματικό σύμβολο του κάθε κανόνα αναφέρεται στο δεξί τμήμα του άλλο.

```
expr: primary
    | primary '+' primary
    ;
primary: constant
       | '(' expr ')'
       ;
```

Όταν γράφουμε έναν αναδρομικό κανόνα ,μπορούμε να βάλουμε την αναδρομική αναφορά στο δεξί όπου έχουμε δεξιά αναδρομή (Right Recursion) ή στο αριστερό μέρος του κανόνα όπου έχουμε αριστερή αναδρομή (Left Recursion), για παράδειγμα :

```
exprlist: exprlist ',' expr ; /* left recursion */
exprlist: expr ',' exprlist ; /* right recursion */
```

Το Bison χειρίζεται πιο αποτελεσματικά την δεξιά αναδρομή από την αριστερή αναδρομή.

2.9 Symbols

Κάθε σύμβολο που χρησιμοποιεί το Bison,είτε είναι terminal ή non-terminal σύμβολο, πρέπει να έχει μια value που σχετίζεται με αυτό. Για να καταλάβουμε στην περίπτωση αυτή την έννοια value, ας υποθέσουμε ότι έχουμε ένα token που

παραπέμπει σε αριθμό. Το token μπορεί να είναι ένας ακέραιος ή πραγματικός αριθμός. Στην γλώσσα προγραμματισμού C ο τύπος του token μπορεί να είναι int ή double. Η value αναφέρεται στους διάφορους τύπους των συμβόλων. (Value: Κάθε token όσον αφορά την γραμματική του Bison έχει syntactic και semantic value. Semantic value(σημασιολογική τιμή) είναι το πραγματικό περιεχόμενο των δεδομένων του token. Για παράδειγμα, ο συντακτικός τύπος μιας λειτουργίας μπορεί να είναι integer, αλλά η σημασιολογική της τιμή μπορεί να είναι το 3.)
Στο Bison parser η value των συμβόλων που έχουν τον ίδιο τύπο ορίζονται με την macro YYSTYPE στο τμήμα των ορισμών. Εξ' ορισμού η macro YYSTYPE είναι int αλλά αν θέλουμε να την δηλώσουμε με διαφορετικό τύπο γράφουμε το εξής:

```
%{
#define YYSTYPE double
%}
```

Τμήμα ορισμών-Δηλώσεις C\C++

Στην περίπτωση που οι values είναι διαφορετικού τύπου για διαφορετικά σύμβολα, οι τύποι δεν δηλώνονται με την macro YYSTYPE όπως παραπάνω, αλλά χρησιμοποιούμε την συνήθης ένωση(union). Για παράδειγμα:

```
%Union{
    int      i;
    double   f;
    char *   str[50];
        }
```

Τμήμα ορισμών-Δηλώσεις Bison

Τα σύμβολα που χρησιμοποιούνται στο Bison, πρέπει να ορίζεται ο τύπος τους. Για να δηλώσουμε τους τύπους των συμβόλων αρκεί να προσθέσουμε <τύπος> στην αρχική δήλωση των συμβόλων.(Όπως γνωρίζουμε τα non-terminal και τα terminal σύμβολα δηλώνονται στο τμήμα των ορισμών και συγκεκριμένα στις δηλώσεις Bison). Για παράδειγμα:

```
%token <f> NUMBER
%type <i> expression
```

Εναλλακτικά , μπορούμε να καθορίσουμε τον τύπο δεδομένων ,όταν αναφερόμαστε στην τιμή αναφοράς του ,χρησιμοποιώντας < type> μετά το σύμβολο $.Για παράδειγμα:

```
%union {
        int itype;
        double dtype;
}
```

Μπορούμε να γράψουμε $<itype>1 για να αναφερθούμε στην πρώτη υποενότητα του κανόνα ως ακέραιος ή $<dtype>1 για να αναφερθούμε στην δεύτερη υποενότητα του κανόνα ως πραγματικός.

2.10 Tokens

Τα tokens είναι terminal symbols που έχει αναγνωρίσει ο lexer και που τα περνάει στον parser.Κάθε φορά που ένας Bison parser χρειάζεται ένα token, καλεί την yylex() που επιστρέφει το επόμενο token από την είσοδο. Στο τέλος της εισόδου, η yylex() επιστρέφει μηδέν.

Τα tokens είναι τα σύμβολα που δηλώνονται με το πρόθεμα %token. Το Bison αντιμετωπίζει τους μεμονωμένους χαρακτήρες που εσωκλείονται σε μονά εισαγωγικά ως tokens. Για παράδειγμα:

```
expr: '(' expr ')' ;
```

Οι παρενθέσεις είναι literal tokens. Οι χαρακτήρες όπως +,-,*,/,(,) είναι tokens που δεν χρειάζονται να δηλωθούν με το πρόθεμα %token.
Το Bison μας επιτρέπει να δηλώσουμε strings ως ψευδώνυμα για tokens για παράδειγμα:

```
%token NE "!="
%%
...
exp: exp "!=" exp ;
```

Σύμφωνα με το παραπάνω παράδειγμα δηλώνεται το token NE και μας επιτρέπεται να χρησιμοποιήσουμε εναλλακτικά το NE και != στον parser.

Μεταξύ του lexer και του parser, τα tokens προσδιορίζονται από μικρούς ακεραίους. Ο αριθμός ενός literal token καθορίζεται από το τοπικό σύνολο χαρακτήρων ASCII. Τα συμβολικά tokens έχουν τιμές(value) που αποδίδονται από το Bison, γεγονός που τους δίνει αριθμούς υψηλότερους από έναν πιθανό κωδικό χαρακτήρα ,οπότε δεν ερχονται σε σύγκρουση με κάποιο literal token. Το Bison μας επιτρέπει να ορίσουμε μόνοι μας τους αριθμούς των tokens .

%token UP 50 DOWN 60

Είναι λάθος να ορίσουμε δυο tokens στον ίδιο αριθμό. Στις περισσότερες περιπτώσεις είναι καλό να αφήνουμε το Bison να ορίζει τους αριθμούς των tokens. Ο lexer πρέπει να γνωρίζει τους αριθμούς των tokens, προκειμένου να επιστρέψει τις κατάλληλες τιμές στον parser. Για τα literal tokens χρησιμοποιεί την αντίστοιχη σταθερά του χαρακτήρα (C character constant). Για τα συμβολικά tokens μπορούμε με την επιλογή –d της γραμμής εντολών να δημιουργήσουμε ένα C header file με τους ορισμούς όλων των αριθμών των tokens. Εάν κάνουμε #include το header file στον lexer ,μπορούμε να χρησιμοποιήσουμε τα σύμβολα(tokens) στον C κώδικά του. Το header file συνήθως καλείται ως xxx.tab.h ένα το αρχείο εισόδου μας είναι xxx.y ή μπορούμε να το μετονομάσουμε με το %define ή - -define=filename (σε γραμμή εντολών)

%defines "xxxsyms.h"

2.11 Locations

Για να υποβοηθηθεί μια αναφορά σφάλματος, το bison καθορίζει τις θέσεις(locations) των tokens, ένα χαρακτηριστικό για να «παρακολουθεί» την γραμμή του πηγαίου κώδικα και το εύρος της στήλης για κάθε token στον parser. Οι θέσεις ενεργοποιούνται με το %location ή έμμεσα με αναφορά στην θέση που βρίσκεται στην action code. Ο lexer είναι υποχρέωσή του να «παρακολουθεί» την τρέχουσα γραμμή και στήλης και να ρυθμίζει το εύρος της θέση του κάθε token στο yyloc, πριν

επιστρέψει το επόμενο token.(Flex lexer αυτόματα παρακολουθεί τον αριθμό γραμμής , αλλά ο αριθμός στήλης εξαρτάται από εμάς) .

Μέσα σε μια action code , η θέση για κάθε σύμβολο αναφέρεται ως @$ για το σύμβολο που βρίσκεται στο αριστερό τμήμα του κώδικα(LHS) και ως @1 για το σύμβολο που βρίσκεται στο δεξί τμήμα του κώδικα(RHS) .Η Location είναι στην πραγματικότητα μια δομή που μπορεί να αναφέρεται σε όποιο πεδίο θέλουμε. Για παράδειγμα:

```
exp:   ...
       | exp '/' exp
          {
           @$.first_column = @1.first_column;
           @$.first_line = @1.first_line;
           @$.last_column = @3.last_column;
           @$.last_line = @3.last_line;
           if ($3)
            $$ = $1 / $3;
           else
            {
             $$ = 1;
             fprintf (stderr,
                   "Division by zero, l%d,c%d-l%d,c%d",
                   @3.first_line, @3.first_column,
                   @3.last_line, @3.last_column);
            }
          }
```

2.12 Παραλλαγή και Πολλαπλές γραμματικές(Parsers)

Μπορεί να θέλουμε parsers για δύο μερικώς ή εντελώς διαφορετικές γραμματικές στο ίδιο πρόγραμμα. Για παράδειγμα, ένας διαδραστικός διερμηνέας αποσφαλμάτωση (interactive debugging interpreter) θα μπορούσε να έχει έναν parser για τη γλώσσα προγραμματισμού και έναν άλλο για τις εντολές εντοπισμού σφαλμάτων (debugger commands).

Υπάρχουν δύο τρόποι για να χειριστούμε δυο γραμματικές σε ένα πρόγραμμα: συνδυάζοντάς τις γραμματικές σε έναν parser ή βάζοντας δύο πλήρεις parsers στο πρόγραμμα.

2.12.1 Συνδυασμός Αναλυτών (parsers)

Αν έχουμε μερικές παρόμοιες γραμματικές μπορούμε να τις συνδυάσουμε σε μία προσθέτοντας έναν νέο αρχικό κανόνα που θα εξαρτάται από το πρώτο διάβασμα του token. Για παράδειγμα:

```
%token CSTART PPSTART
%%
combined: CSTART cgrammar
| PPSTART ppgrammar
;
cgrammar: . . .
ppgrammar: . . .
```

Στη περίπτωση αυτή, αν το πρώτο token είναι CSTART ,θα αναλυθεί η γραμματική της οποίας ο αρχικός κανόνας είναι cgrammar, ενώ αν το πρώτο token είναι CS PPSTART ,θα αναλυθεί η γραμματική της οποίας ο αρχικός κανόνας είναι ppgrammar.

Θα πρέπει επίσης να τεθεί κώδικας στον lexer που θα επιστρέφει το κατάλληλο special token την πρώτη φορά που ο parser ζητάει από τον lexer token:

```
%%
%{
    extern first_tok;
    if(first_tok) {
            int holdtok = first_tok;
            first_tok = 0;
            return holdtok;
            }
%}
```

. . . <the rest of the lexer>

Στην περίπτωση αυτή, μπορούμε να ορίσουμε το first_tok με το κατάλληλο token πριν από την κλήση του yyparse().

Ένα πλεονέκτημα αυτής της προσέγγισης είναι ότι το πρόγραμμα είναι μικρότερο από ότι θα ήταν με πολλαπλά προγράμματα ανάλυσης(parsers), αφού υπάρχει μόνο ένα αντίγραφο του κώδικα ανάλυσης. Το μειονέκτημα είναι ότι δεν μπορούμε να καλέσουμε έναν parser, ενώ ένας άλλος είναι ενεργός, εκτός εάν δημιουργήσουμε ένα καθαρό αναλυτή (pure parser) όπου θα πρέπει να χρησιμοποιούν διαφορετικά σύμβολα και στις δύο γραμματικές.

2.12.2 Πολλαπλοί Αναλυτές

Η άλλη προσέγγιση είναι να περιλαμβάνει δύο πλήρη parsers σε ένα ενιαίο πρόγραμμα. Κάθε Bison parser έχει την ίδιο σημείο εισόδου,yyparse(), και καλεί τον ίδιο lexer, yylex(),ο οποίος χρησιμοποίει την ίδια τιμή του token στην μεταβλητή yyval (μέσω της συγκεκριμένης μεταβλητής γίνεται η επικοινωνία με τον lexer. Η μεταβλητή περιέχει την τιμή του token). Επίσης ο parse table και ο parse stack είναι καθολικές μεταβλητές με αντίστοιχα ονόματα yyact και yyv. Αν μεταφράσουμε δυο γραμματικές και μεταγλωττίσουμε και συνδέσουμε τα αρχεία που προκύπτουν ,θα έχουμε έναν μακρύ κατάλογο από συμβόλων. Για να το αποφύγουμε, μπορούμε να αλλάξουμε τα ονόματα που χρησιμοποιεί το Bison για τις συναρτήσεις και τις μεταβλητές.

-%name-prefix or the -p Flag: Μπορούμε να χρησιμοποιήσουμε μια δήλωση στον πηγαίο κώδικα του Bison για να αλλάξουμε το πρόθεμα που χρησιμοποιείται για τα ονόματα του parser που παράγεται από το Bison.

%name-prefix "pdq"

Αυτό παράγει έναν parser με σημείο εισόδου pdrparse(), το οποίο καλεί τον lexer pdqlex() και ούτε καθεξής. Συγκεκριμένα τα ονόματα που επηρεάζονται είναι το yyparse(), yylex(), yyerror(), yylval, yychar, και yydebug.(Η μεταβλητή yychar κρατάει το ποιο πρόσφατο token που έχει διαβαστεί, η οποία μερικές φορές είναι χρήσιμη όταν εκτυπώνουμε μηνύματα λαθών). Επίσης, μπορούμε να καθορίσουμε το πρόθεμα όχι στο αρχείο προέλευσης αλλά μέσω της γραμμής εντολών με την –p flag

και με την –b flag καθορίζουμε το πρόθεμα του αρχείου που παράγεται. Για παράδειγμα:

bison -d -p pdq -b pref mygram.y

Θα παραχθούν pref.tab.c και pref.tab.h με parser του οποίου το σημείο εισόδου είναι pdqparse.

2.13 y.output Files

Το Bison μπορεί να δημιουργήσει ένα αρχείο καταγραφής, γνωστό ως y.output ή name.output, δείχνει όλες τις καταστάσεις του parser και τις μεταβάσεις από την μια στην άλλη κατάσταση, απεικονίζει μια κατάσταση, τους κανόνες που είναι υποψήφιοι για αναγνώριση (τι βρίσκεται στη στοίβα), και τις ενέργειες ανάλογα με το σύμβολο που θα διαβαστεί. Για να παραχθεί ένα αρχείο καταγραφής αρκεί να γράψουμε την flag --report=all.

Παρακάτω περιγράφεται ένα μέρος του αρχείου καταγραφής του parser του κεφαλαίου 4.

```
state 3
  10 term: NUMBER .
  $default reduce using rule 10 (term)
state 4
  11 term: ABS . term
  NUMBER shift, and go to state 3
  ABS shift, and go to state 4
  term go to state 9
state 5
  2 calclist: calclist calc . EOL
  EOL shift, and go to state 10
```

Figure 2.13 Αρχείο καταγραφής

Η τελεία σε κάθε κατάσταση δείχνει κατά πόσο ο parser έχει αναλύσει τον κανόνα στην κατάσταση που βρίσκεται. Όταν ο parser βρίσκεται στην κατάσταση 4 , για παράδειγμα, έχει δει το NUMBER token ,θα το κάνει shift στην στοίβα και θα μεταβεί στην κατάσταση 3. Μόλις δει το token ABS, το κάνει shift και επιστρέφει στην κατάσταση 4. Εάν ακολουθεί reduction που επιστρέφει στην κατάσταση 4 με το

term στην κορυφή της στοίβας, τότε μεταβαίνει στην κατάσταση 9. Στην κατάσταση 3 , μειώνει (reduce) πάντα τον κανόνα 10. Μετά το reduction, το NUMBER αντικαθίσταται στη στοίβα του parser με term όπου ο parser μεταβαίνει στην κατάσταση 4 και στη συνέχεια μεταβαίνει στην κατάσταση 9.

Στην περίπτωση που υπάρχουν συγκρούσεις(conflicts), οι καταστάσεις δείχνουν τις συγκρούσεις των shift και reduce actions.

```
State 19 conflicts: 3 shift/reduce
state 19
   5 exp: exp . ADD exp
   5 | exp ADD exp .
   6 | exp . SUB factor
   7 | exp . ABS factor

   ADD shift, and go to state 12
   SUB shift, and go to state 13
   ABS shift, and go to state 14
   ADD [reduce using rule 5 (exp)]
   SUB [reduce using rule 5 (exp)]
   ABS [reduce using rule 5 (exp)]
   $default reduce using rule 5 (exp)
```

Figure 2.13-a Καταστάσεις με conflicts

Στην περίπτωση αυτή έχουμε shift/reduce conflict όταν parser βλέπει το σύμβολο '+'."

2.14 Βιβλιοθήκη του Bison

Το Bison κληρονομεί μια βιβλιοθήκη με χρήσιμες ρουτίνες από τον προκάτοχο του yacc. Για να συμπεριλάβουμε την βιβλιοθήκη προσθέτουμε το flag –ly τέλος της cc command line. Η βιβλιοθήκη περιέχει την ρουτίνα main() και yyerror().

main()

Η βιβλιοθήκη έχει ένα ελάχιστο πρόγραμμα, το οποίο μερικές φορές είναι χρήσιμο για γρήγορα προγράμματα και για δοκιμές.

main(ac, av)

```
{
        yyparse();
        return 0;
}
```

Όπως και με οποιανδήποτε ρουτίνα της βιβλιοθήκης, μπορούμε να παρέχουμε δική μας main(). Σχεδόν σε κάθε χρήσιμη εφαρμογή θα θέλουμε η main() να δέχεται παραμέτρους γραμμής εντολών, flags, αρχεία και ελέγχους για λάθη.

yyerror()

Το Bison παρέχει μια απλή ρουτίνα error-reporting.

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

Συνήθως αυτό επαρκεί, αλλά μια καλύτερη ρουτίνα λάθους που θα αναφέρει τουλάχιστον τον αριθμό της γραμμής και το πιο πρόσφατο token θα διευκολύνει τον parser.

YYABORT

Η ειδική δήλωση YYBORT; σε μια ενέργεια (action) κάνει την ρουτίνα yyparse() να επιστρέφει μια μη μηδενική τιμή, υποδεικνύοντας αποτυχία. Είναι χρήσιμη όταν μια ρουτίνα μιας ενέργειας εντοπίσει σφάλμα τόσο σοβαρό έτσι ώστε να πρέπει να σταματήσει η μεταγλώττιση. Δεδομένου ότι ο parser μπορεί να δεί ένα token πιο μπροστά , ο κανόνας που περιέχει την YYBORT μπορεί να μην μειωθεί(reduce) έως ότου ο parser διαβάσει ένα άλλο token.

YYACCEPT

Η ειδική δήλωση YYACCEPT; σε μια ενέργεια (action) κάνει την ρουτίνα yyparse() να επιστρέφει μηδενική τιμή, υποδεικνύοντας επιτυχία. Είναι χρήσιμη στην περίπτωση που ο lexer δεν μπορεί να ενημερώσει πότε τα δεδομένα εισόδου τελειώνουν, ενώ ο parser μπορεί. Δεδομένου ότι ο parser μπορεί να δει ένα token πιο μπροστά , ο κανόνας που περιέχει την YYACCEPT μπορεί να μην μειωθεί(reduce) έως ότου ο parser διαβάσει ένα άλλο token.

YYBACKUP

Η macro YYBACKUP μας επιτρέπει να μην κάνουμε shift στο τρέχον token και να αντικατασταθεί με κάτι άλλο. Η σύνταξή της είναι εξής:

sym: TOKEN { YYBACKUP(newtok, newval); }

Δεν κάνει shift στο token. Η macro αυτή επιτρέπεται μόνο για κανόνες που μειώνουν μια μόνο τιμή(value) και μόνο όταν δεν υπάρχει lookahead token(το επόμενο token που έχει δει ο parser).

yyclearin

Απορρίπτει το τρέχον lookahead token. Η macro αυτή είναι χρήσιμη σε κανόνες error και στην ανάκτηση λάθος σε ένα διαδραστικό πρόγραμμα ανάλυσης για να τεθεί το πρόγραμμα ανάλυσης σε μια γνωστή κατάσταση μετά από ένα λάθος.

stmtlist: stmt | stmtlist stmt ;

stmt: error { reset_input(); yyclearin; } ;

Μετά από ένα λάθος, αυτό καλεί τον χρήστη ρουτίνα reset_input (), η οποία θέτει κατά πάσα πιθανότητα την είσοδο σε μια γνωστή κατάσταση και στην συνέχεια χρησιμοποιείται το yyclearin για να προετοιμάσει τον parser έτσι ώστε να ξεκινήσει το διάβασμα των tokens εκ'νέου.

YYDEBUG

Ορίζεται σε μη μηδενική τιμή η macro YYDEBUG, όταν μεταγλωττίζεται ο parser. Μπορούμε να χρησιμοποιήσουμε '-DYYDEBUG=1' στην option του μεταγλωττιστή ή '#define YYDEBUG 1' στο τμήμα των ορισμών και συγκεκριμένα στις δηλώσεις C/C++.

yydebug

Η ακέραια μεταβλητή yydebug στην λειτουργία του parser ελέγχει αν ο parser παράγει στην πραγματικότητα το αποτέλεσμα της αποσφαλμάτωσης. Εάν η τιμή της μεταβλήτης δεν είναι μηδενική, ο parser παράγει αναφορά αποσφαλμάτωσης , ενώ αν η τιμή είναι μηδέν δεν παράγει κάποια αναφορά .

yyerrok

Όταν ο parser ανιχνεύσει κάποιο συντακτικό λάθος, απέχει στο να αναφέρει κάποιο λάθος μέχρι να μετατοπίσει τρία συνεχόμενα tokens χωρίς κάποιο λάθος. Αυτό μετριάζει τα πολλαπλά μηνύματα λάθους που δημιουργούνται από ένα λάθος, καθώς ο parser συγχρονιστεί. Η macro yyerrok ενημερώνει τον parser για να επιστρέψει σε μια κανονική κατάσταση.

Για παράδειγμα, ας υποθέσουμε ότι έχουμε έναν command interpreter, στον οποίο όλες οι εντολές είναι σε χωριστές γραμμές.

```
cmdlist: cmd | cmdlist cmd ;

cmd: error '\n' { yyerrok; } ;
```

Ο κανόνας με το errok προσπερνάει την είσοδο μετά από ένα λάθος μέχρι την αλλαγή γραμμής. Η macro yyerrok ενημερώνει τον parser ότι η αποκατάσταση του λάθους ολοκληρώθηκε.

YYERROR

Μερικές φορές ο κώδικας της ενέργειας ενός κανόνα μπορεί να ανιχνεύσει ευαίσθητα λάθη(detect context-sensitive syntax errors) και ο parser δεν μπορεί. Εάν ανιχνευτεί ένα συντακτικό λάθος, μπορούμε να καλέσουμε την YYERROR για να παράγει το ίδιο αποτέλεσμα , όπως θα έκανε ο parser όταν θα διάβαζε ένα token που δεν επιτρεπόταν από την γραμματική. Μόλις επικαλεστούμε την macro YYERROR, ο parser μεταβαίνει σε κατάσταση αποκατάστασης λάθους ψάχνοντας μια κατάσταση για να μετατοπίσει το λάθος token.

yyerror

Όταν ο parser ανιχνεύσει ένα συντακτικό λάθος, καλεί την συνάρτηση yyerror() για να ενημερώσει τον χρήστη για το λάθος, περνώντας το σε μια παράμετρο: ένα string που περιγράφει το λάθος.(Μπορούμε να παράγουμε ένα μήνυμα λάθους χρησιμοποιώντας στον κώδικά μας την yyerror().). Η προεπιλεγμένη έκδοση της yyerror() στην βιβλιοθήκη του Bison τυπώνει την παράμετρο στην τυπική έξοδο.

```
yyerror(const char *msg)
{
```

```
printf("%d: %s at '%s'\n", yylineno, msg, yytext);
}
```

yyparse()

Το σημείο εισόδου ενός παραγόμενου Bison parser η συνάρτηση yyparse(). Όταν το πρόγραμμα καλεί την yyparse(), επιχειρεί να αναλύσει συντακτικά την ουρά εισόδου. Ο parser επιστρέφει μια τιμή μηδέν όταν η συντακτική ανάλυση πετύχει, διαφορετικά επιστρέψει μη μηδενική τιμή. Συνήθως η συνάρτηση yyparse() δεν παίρνει παραμέτρους αλλά όταν ο parser χρειάζεται να εισάγει πληροφορίες από το περιβάλλον του προγράμματος ,χρησιμοποιεί είτε global μεταβλητές είτα παραμέτρους για τον ορισμών τους:

```
%parse-param {char *modulename}

%parse-param {int intensity}
```

Μας επιτρέπει να καλέσουμε την συνάρτηση yyparse("mymodule",42) και αναφέρεται στο modulename και στο intensity της action code του parser.

Κάθε φορά που καλείται η yyparse(), ο parser ξεκινάει εκ'νέου συντακτική ανάλυση, ξεχνώντας ότι κατάσταση θα μπορούσε να ήταν την τελευταία φορά που επέστρεψε. Σε αντίθεση με την συνάρτηση yylex() παράγεται από το Flex, η οποία κρατάει το σημείο που σταμάτησε κάθε φορά που θα την καλέσει ο χρήστης.

YYRECOVERING()

Όταν το Bison ανιχνεύσει ένα συντακτικό λάθος, κανονικά μπαίνει σε κατάσταση ανάκτησης , η οποία απέχει από κάποια άλλη αναφορά λάθους μέχρι να γίνει shift τριών συνεχόμενων tokens χωρίς αναφορά λάθους . Αυτό μετριάζει τα πολλαπλά μηνύματα λάθους που δημιουργούνται από ένα λάθος, καθώς ο parser συγχρονιστεί. Η macro YYRECOVERING() επιστρέφει μη μηδενική τιμή όταν ο parser βρίσκεται σε κατάσταση ανάκτηση λαθών, διαφορετικά επιστρέφει μηδέν.

ΚΕΦΑΛΑΙΟ 3^0

Συνεργασία-Μεταγλώττιση προγραμμάτων Flex και Bison σε Περιβάλλον του Microsoft Visual studio

3.1 Λεκτική και συντακτική ανάλυση (Flex/Bison)

Σκοπός της λεκτικής ανάλυσης είναι να διαβάζει μια πηγή εισόδου ως ένα ρεύμα χαρακτήρων και να παράγει λεκτικές μονάδες tokens. Από την άλλη η συντακτική ανάλυση χρησιμοποιεί ως είσοδο τις λεκτικές μονάδες που παράγει η λεκτική ανάλυση.

Τα εργαλεία Flex και Bison που αντιστοιχούν στην δημιουργία λεκτικού και συντακτικού αναλυτή κατασκευάστηκαν με σκοπό να συνεργαστούνε.

Για να επιτευχθεί η συνεργασία μεταξύ τους, ο χρήστης πρέπει να ακολουθήσει τα εξής βήματα:

- Στο αρχείο που έχουμε συντάξει στο αρχείο Flex αφαιρούμε τις δηλώσεις των λεκτικών μονάδων(tokens) και στη θέση τους προσθέτουμε το αρχείο που παράγεται από το Bison –d
  #include <filename>.tab.h
- Επιπλέον αφαιρούμε από το αρχείο Flex την main()
- Στο αρχείο Bison προσθέτουμε στο τρίτο μέρος του ,το αρχείο #include <filename>.lex.c που παράγεται από το αρχείο Flex. Πλέον η συνάρτηση yylex() καλείται μέσα της συνάρτησης yyparse().

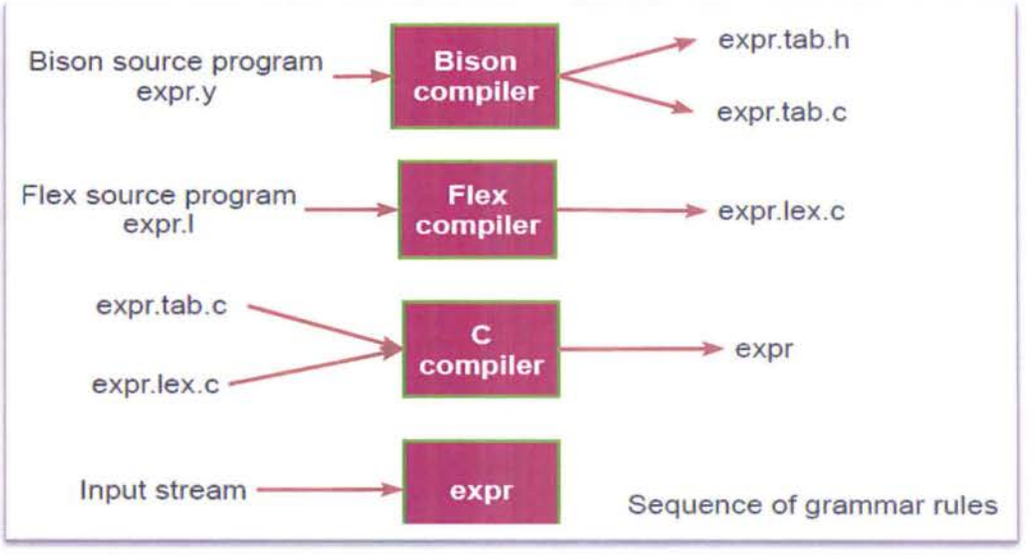

Figure 3.1 Διαδικασία παραγωγής Αναλυτή

Την συνεργασία των δύο εργαλείων θα δούμε αναλυτικά μέσω προγραμμάτων στο επόμενο κεφάλαιο.

3.2Μεταγλώττιση και εκτέλεση προγραμμάτων flex/bison

Για να κατανοήσουμε στην πράξη όσα έχουν αναφερθεί στις παραπάνω ενότητες και για να μπορέσουμε να συντάξουμε ένα πρόγραμμα με το εργαλείο flex και bison θα πρέπει να χρησιμοποιήσουμε ένα λογισμικό. Για τις ανάγκες της μεταγλώττισης στην παρούσα διπλωματική εργασία ,θα χρησιμοποιήσουμε το Microsoft visual studio .

3.2.1 Microsoft Visual Studio

Υποστηρίζει την μεταγλώττιση προγραμμάτων σε γλώσσα C ή C++ σε κώδικα γραμμής (command line). Οι εφαρμογές αυτές λέγονται εφαρμογές κονσόλας (console applications). Υπάρχουν δύο ειδών console applications:

- Win32 console applications που μεταγλωττίζουν σε native code και χρησιμοποιούνται για μεταγλώττιση προγραμμάτων γραμμένα κατά το πρότυπο ISO/ANSIC++
- OI CLR console applications που στοχεύουν στο NET. Framework

Εμείς θα ασχοληθούμε για την σύνταξη των προγραμμάτων μας με το πρότυπο ISO/ANSIC++. Για να τρέξουμε το πρόγραμμα μας πρέπει να ακολουθήσουμε την εξής διαδικασία:

1. Ανοίγουμε το Microsoft visual studio

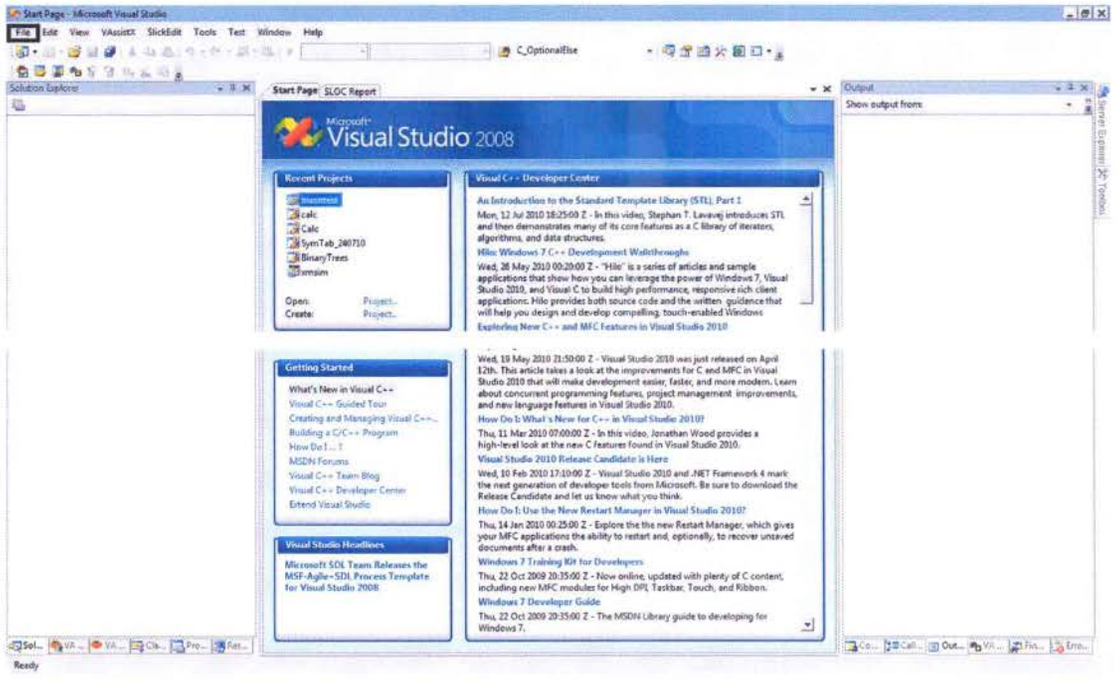

Figure 3.2.1-a Αρχική σελίδα του Visual Studio

2. Δημιουργούμε ένα χώρο εργασίας για την υλοποίηση μιας Win32 console applications. Για να γίνει αυτό ανοίγουμε από το menu το File->New->Project οπότε ανοίγει το ακόλουθο παράθυρο

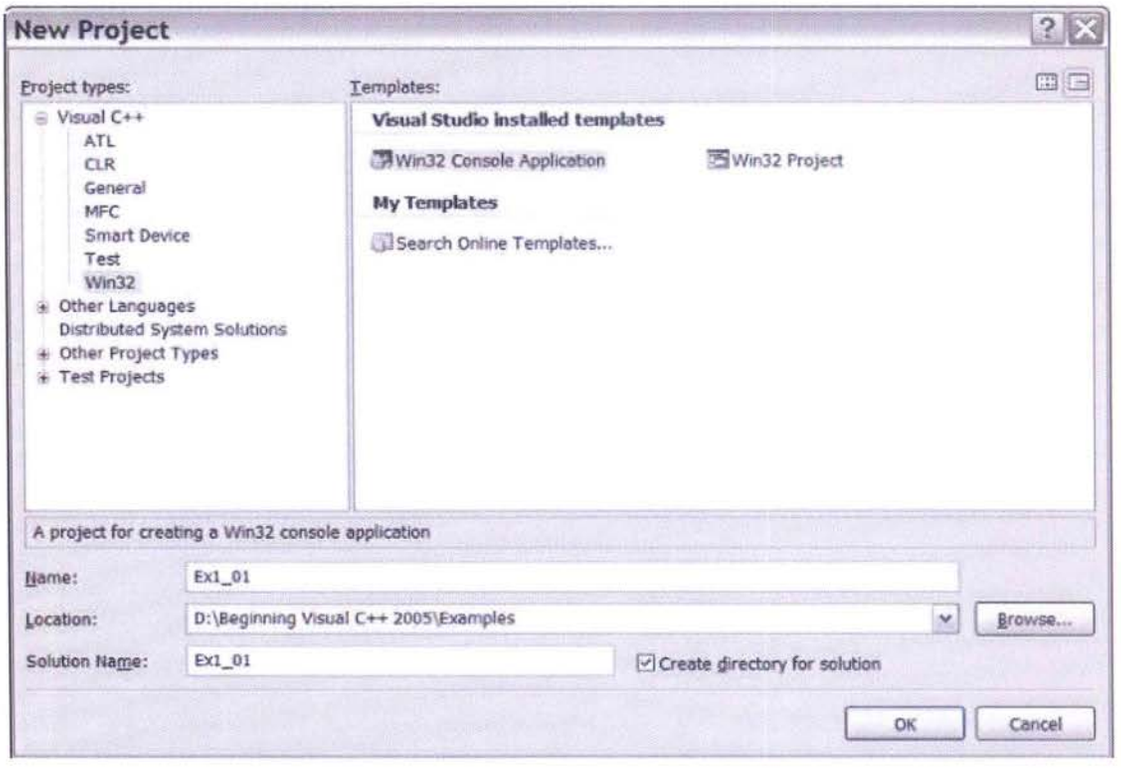

Figure 3.2.1-b Αρχικό παράθυρο προσδιορισμού του νέου χώρου εργασίας

3. Δίνουμε όνομα στο χώρο εργασίας μας στο πεδίο Name. Επιλέγουμε την διαδρομή μέσα στην οποία θέλουμε να δημιουργηθεί ο χώρος εργασίας (project) στο πεδίο Location. Ξε-τσεκάρουμε το κουμπί Create directory for solution. Το αποτέλεσμα είναι η δημιουργία ενός φακέλου (directory) μέσα στην διαδρομή που έχουμε υποδείξει το πεδίο Location. Στην συνέχεια πατάμε ok και προκύπτει το ακόλουθο παράθυρο

Figure 3.2.1-c Προσδιορισμός παραμέτρων του χώρου εργασίας

4. Επιλέγουμε το πεδίο Aplication setting και εμφανίζεται το παρακάτω παράθυρο με τις επιλογές όπως φαίνονται στο σχήμα.

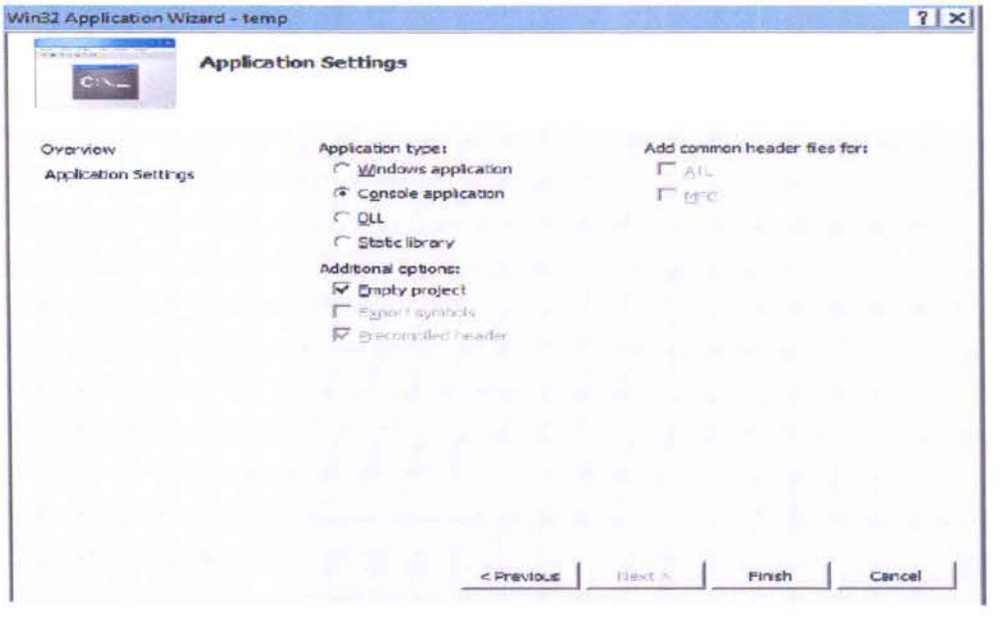

Figure 3.2.1-d Προσδιορισμός παραμέτρων του χώρου εργασίας

5. Πατάμε finish για την ολοκλήρωση δημιουργίας του χώρου εργασίας. Ο χώρος εργασίας φαίνεται στο παραπάνω σχήμα Figure 3.2.1-d .

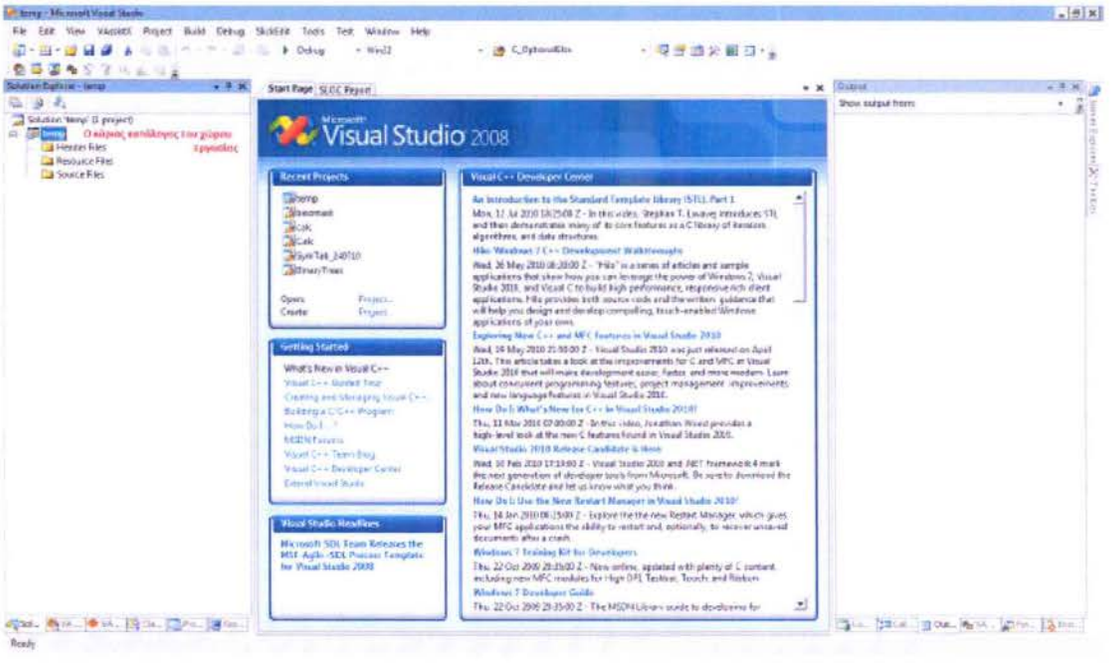

Figure 3.2.1-e Χώρος εργασίας

6. Αριστερά στο σχήμα μπορούμε να οργανώσουμε τα αρχεία της εφαρμογής μας ανάλογα με την χρησιμότητά τους. Η οργάνωση των αρχείων στο visual studio δεν ανταποκρίνεται στην πραγματική οργάνωση τους στο δίσκο. Στην πράξη η οργάνωση αυτή είναι εικονική και υφίσταται μόνο μέσα στο visual studio. Ο φυσικός χώρος των αρχείων στο δίσκο είναι στον κατάλογο που ορίστηκε στο παράθυρο του σχήματος Figure 3.2.1.e.

7. Πατώντας δεξί κλίκ πάνω στους εικονικούς καταλόγους του visual studio είναι δυνατόν να προσθέσουμε είτε νέους καταλόγους , είτε να προσθέσουμε νέα αρχεία προς μεταγλώττιση κ.τ.λ. Η πρακτική που θα γίνει θα οδηγήσει και στην απαιτούμενη εξοικείωση με τις λειτουργίες αυτές.

8. Τα αρχεία που βρίσκονται στους καταλόγους του χώρου εργασίας μεταγλωττίζονται κατά την κλήση C/C++ compiler από το visual studio .

Επίσης ο χώρος εργασίας του visual studio δεν περιορίζεται μόνο στην κλήση μεταγλωττιστών του συγκεκριμένου πακέτου λογισμικού αλλά και μεταγλωττιστές που ορίζει ο χρήστης. Το περιβάλλον επιτρέπει την διαμόρφωση των παραμέτρων για την κλήση και άλλων μεταγλωττιστών ή εργαλείων, στην περίπτωσή μας το flex και το bison όπου η μεταγλώττιση των αρχείων γίνεται κάτω από ένα ενιαίο περιβάλλον εργασίας.

3.2.2 Μεταγλώττιση στο Microsoft Visual Studio

Η μεταγλώττιση στο visual studio μπορεί να γίνει μέσω της επιλογής Build του κυρίου μενού (Figure 3.2.2-a)του περιβάλλοντος. Η επιλογή Build Solution μεταγλωττίζει μόνο αρχεία τα οποία έχουν ενημερωθεί μετά την τελευταία μεταγλώττιση συν τα αρχεία που εξαρτώνται από αυτά. Η επιλογή Rebuild Solution εκτελεί πλήρη μεταγλώττιση σε όλα τα αρχεία που ανήκουν στον χώρο εργασίας.

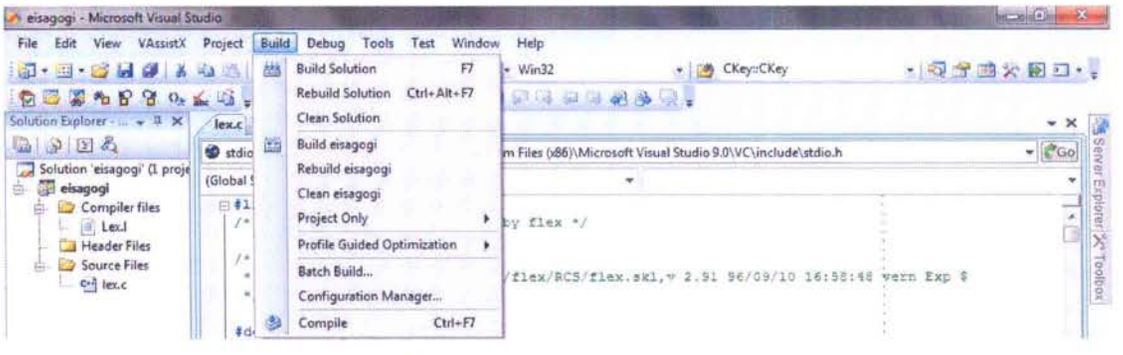

Figure 3.2.2-a Μενού μεταγλώττισης

Σε ότι αφορά την μεταγλώττιση προγραμμάτων σε C/C++ . Η μεταγλώττιση αφορά το πέρασμα του μεταγλωττιστή και του Linker. Ο μεταγλωττιστής ελέγχει συντακτικά και σημασιολογικά το πρόγραμμα ,ενώ ο linker διασύνδεει και ελέγχει τα object αρχεία που έχουν προκύψει από το στάδιο της μεταγλώττισης. Η αποσφαλμάτωση ενός προγράμματος που έχει λάθη μεταγλώττισης έγκειται στην καλή γνώση της C/C++. Στο Figure 3.2.2-b φαίνεται η περιγραφή ενός στοιχειώδους λεκτικού αναλυτή που έχει ένα συντακτικό λάθος. Το λάθος αφορά την παράλειψη μιας αγκύλης στην συνάρτηση main(). Ο μεταγλωττιστής μας πληροφορεί για τα εξής:

- Ότι υπάρχει λάθος
- Σε ποιο αρχείο βρίσκεται (lex.1)
- Σε ποια γραμμή (10)

- Μια υπόδειξη για το πώς μπορεί να επιλυθεί

Πρέπει να σημειωθεί ότι οι υποδείξεις του μεταγλωττιστή δεν είναι πάντοτε ξεκάθαρες και εξαρτώνται από την πολυπλοκότητα του λάθους. Η επίλυση τέτοιων λαθών απαιτεί επίμονη πρόσβαση στο διαδίκτυο και εμπειρία.

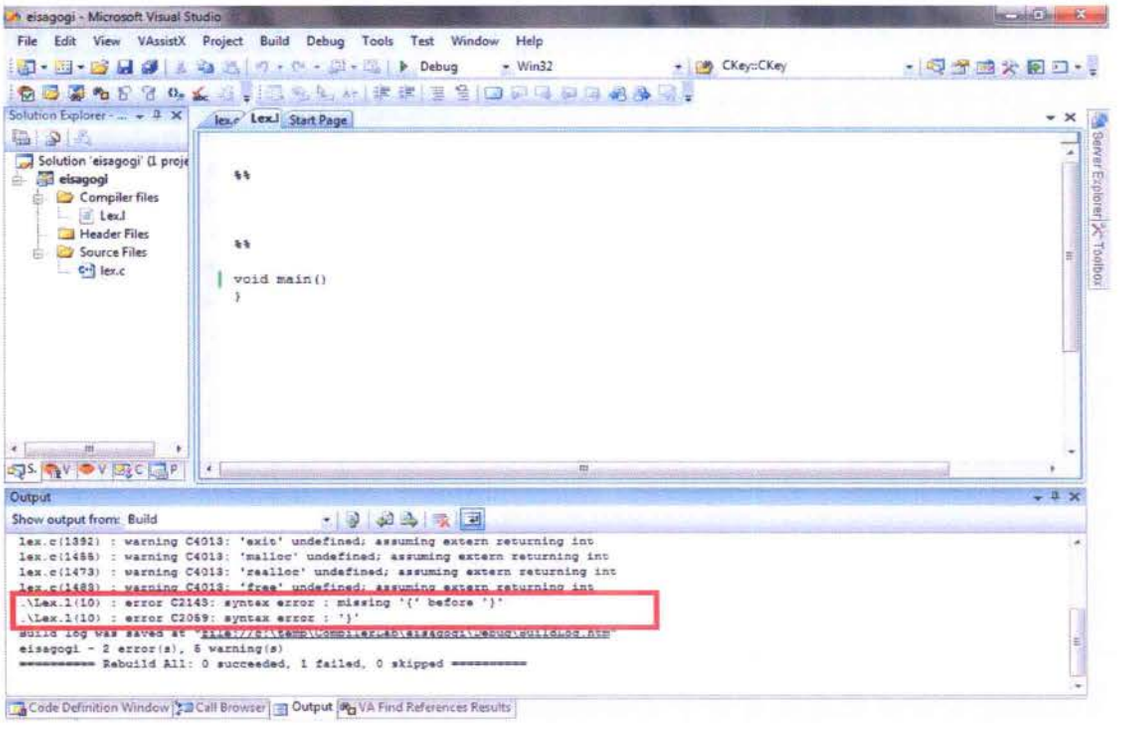

Figure 3.2.2-b Παράδειγμα λάθους μεταγλώττισης

3.2.3 Εργαλείο Flex

Για να μπορέσουμε να χρησιμοποιήσουμε το εργαλείο flex στα διάφορα προγράμματά μας πρέπει πρώτα να εγκαταστήσουμε το εργαλείο flex. Το εργαλείο flex για windows μπορεί να βρεθεί στην ιστοσελίδα http://gnuwin32.sourceforge.net/packages/flex.htm.

Η εγκατάσταση είναι απλή. Τρέχουμε το εκτελέσιμο αρχείο και κάνουμε εγκατάσταση του προγράμματος σε κατάλογο που δεν έχει κενό χαρακτήρα π.χ αποφεύγουμε να το κάνουμε εγκατάσταση στον default κατάλογο C:\Program Files\GnuWin32. Για να αποφύγουμε την δυσλειτουργία του εργαλείου τοποθετούμε την εγκατάσταση στον κατάλογο π.χ. C:\GnuWin32

Την πρώτη φορά που θα γίνει η εγκατάσταση του εργαλείου flex, bison ή m4 στα windows θα πρέπει η διαδρομή εγκατάστασης του flex,bison και m4 να

συμπεριληφθεί στην μεταβλητή PATH του συστήματος. Για να γίνει αυτό πρέπει (για μια φορά) να ακολουθηθεί η παρακάτω διαδικασία:

1. Μέσα από τον πίνακα ελέγχου (control panel) πηγαίνουμε στις ρυθμίσεις του συστήματος(System) και από εκεί επιλέγουμε το μενού (advanced tab) των προχωρημένων ρυθμίσεων ,οπότε προκύπτει το ακόλουθο παράθυρο (Figure 3.2.3-b) από πού επιλέγουμε το κουμπί Enviroment Variables (Μεταβλητές Συστήματος).

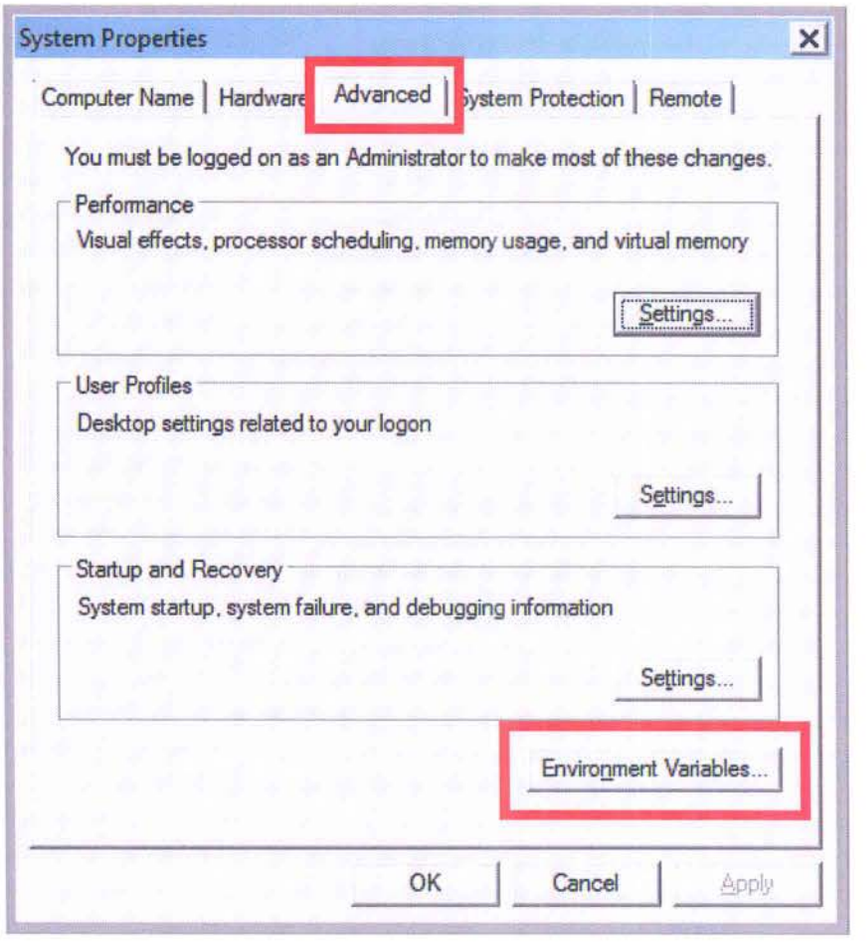

Figure 3.2.3-b Μενού προχωρημένων ρυθμίσεων

2. Προκύπτει το παράθυρο του σχήματος Figure 3.2.3-c. Επιλέγουμε EDIT για να εισάγουμε την διαδρομή όπου βρίσκονται τα εκτελέσιμα αρχεία του flex,bison και m4. Τα αρχεία αυτά βρίσκονται μέσα στο κατάλογο .../bin.

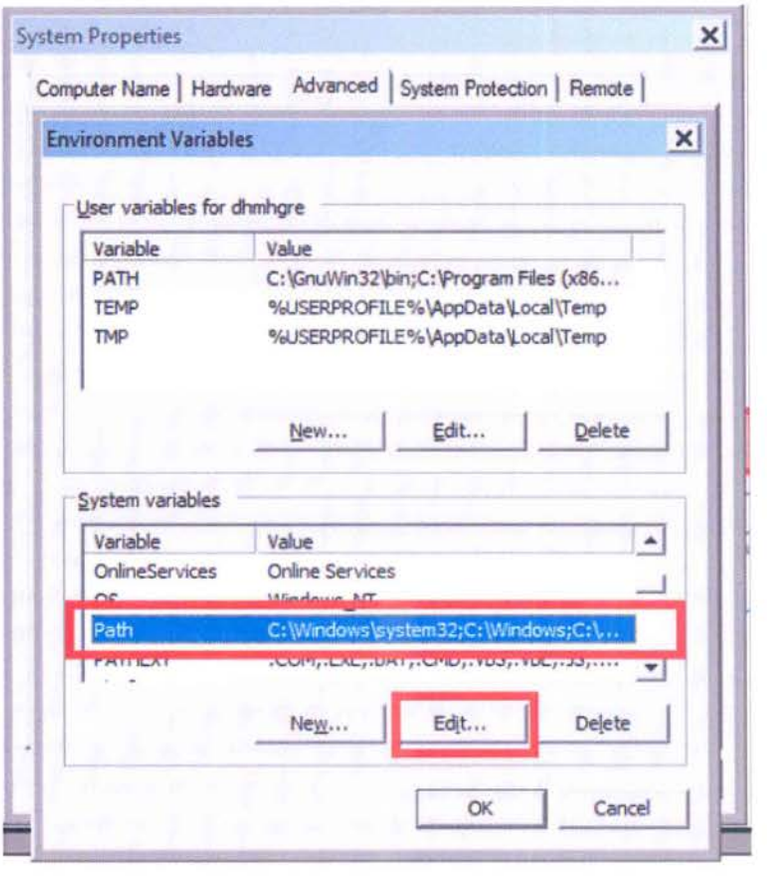

Figure 3.2.3-c Παράθυρο μεταβλητών περιβάλλοντος

3. Μέσα στο variable value συμπληρώνουμε την διαδρομή των εκτελέσιμων αρχείων όπως φαίνεται στο παρακάτω παράθυρο.Προσοχή!Οι διαδρομές στο πεδίο αυτό χωρίζονται με ελληνικό ερωτηματικό.

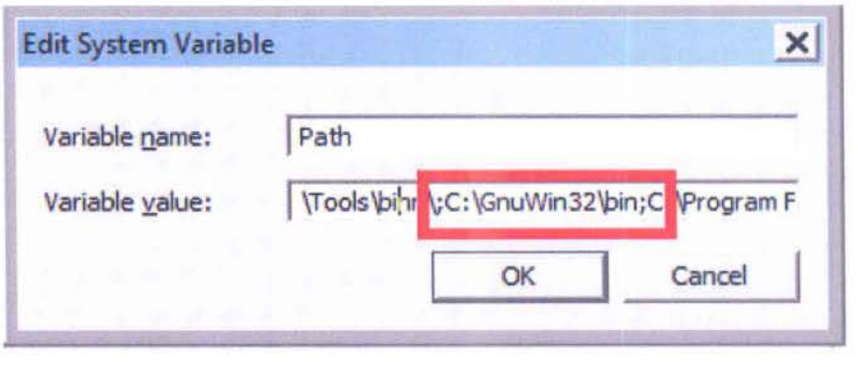

Figure 3.2.3-d Ενημέρωση της μεταβλητής path

4. Στη συνέχεια απαιτείται LogOff και μετά LogOn για να ενημερωθεί η μεταβλητή path. Το εργαλείο flex είναι ένα εργαλείο που η κλήση του

γίνεται από την γραμμή εντολής (commandd line) και έχει την παρακάτω μορφή:

Flex [διακόπτες][αρχείο εισόδου]

Η κλήση του εργαλείου Flex μέσα από visual studio ακολουθεί την ίδια ακριβώς σύνταξη , με την διαφορά ότι ο χρήστης κατά την μεταγλώττιση του χώρου εργασίας καλεί αυτόματα το flex με την ενδεδημένη γραμμή εντολής χωρίς να είναι απαραίτητο ο χρήστης να κάνει κλήση του εργαλείου από την γραμμή εντολής. Ο τρόπος κλήσης του εργαλείου flex δεν είναι σταθερος αλλά μπορεί να τροποποιειθεί από τον χρήστη ακολουθώντας τα παρακάτω βήματα:

1)Δεξί κλίκ στο αρχείο του flex(filename.l) και πάτημα στο αριστερό κουμπί από το context menu της επιλογής Properties

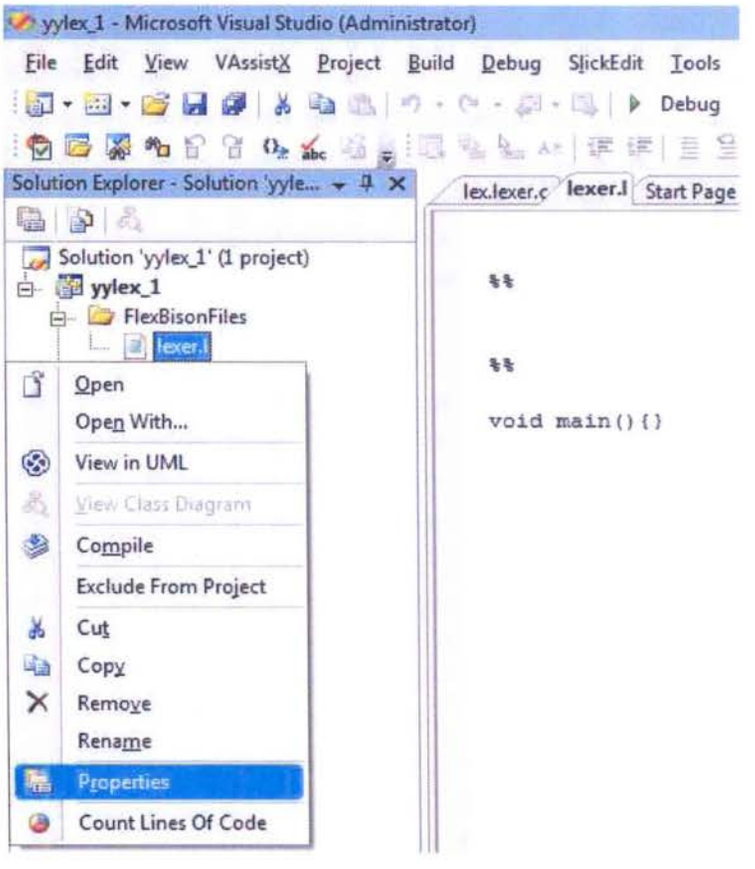

Figure 3.2.3-e Επιλογές που καθορίζουν την κλήση του εργαλείου Flex

2)Στο παράθυρο διαλόγου που προκύπτει φαίνεται ο τρόπος κλήσης του εργαλείου flex μέσα στο Command line tab. Στα υπόλοιπα 2tabs(Generak και Performance) ο χρήστης βρίσκει διακόπτες της κλήση του εργαλείου flex(Figureure)

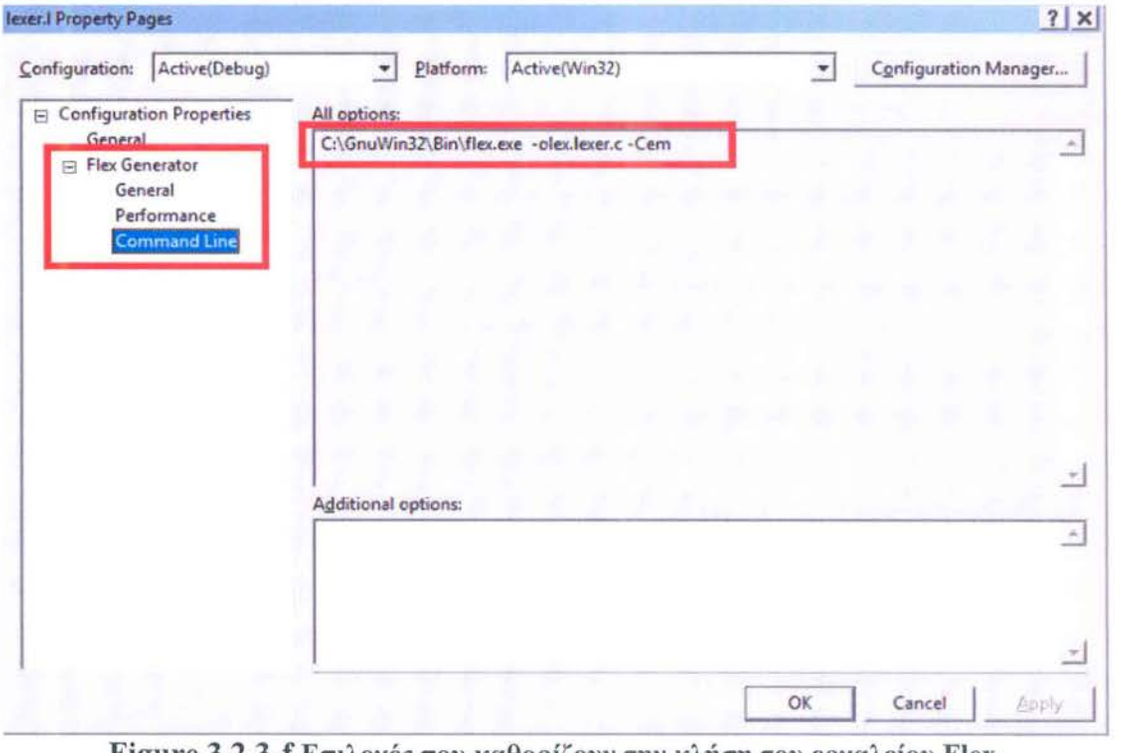

Figure 3.2.3-f Επιλογές που καθορίζουν την κλήση του εργαλείου Flex

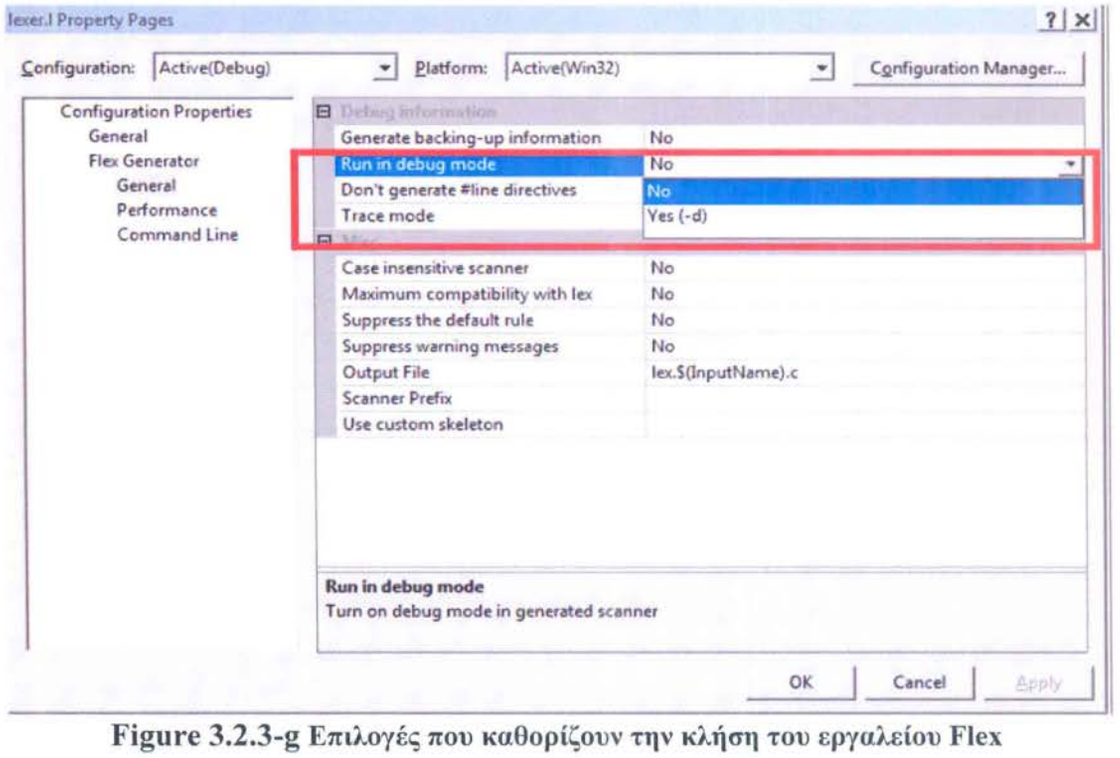

Figure 3.2.3-g Επιλογές που καθορίζουν την κλήση του εργαλείου Flex

Στο παράδειγμα του σχήματος **Figure 3.2.3-g** φαίνεται η τροποποίηση του διακόπτη «Run in debug mode». Η ενεργοποίηση αυτής της λειτουργίας προσθέτει την παράμετρο –d στην γραμμής εντολής όπως φαίνεται στο παρακάτω σχήμα **Figure 3.2.3-h**.

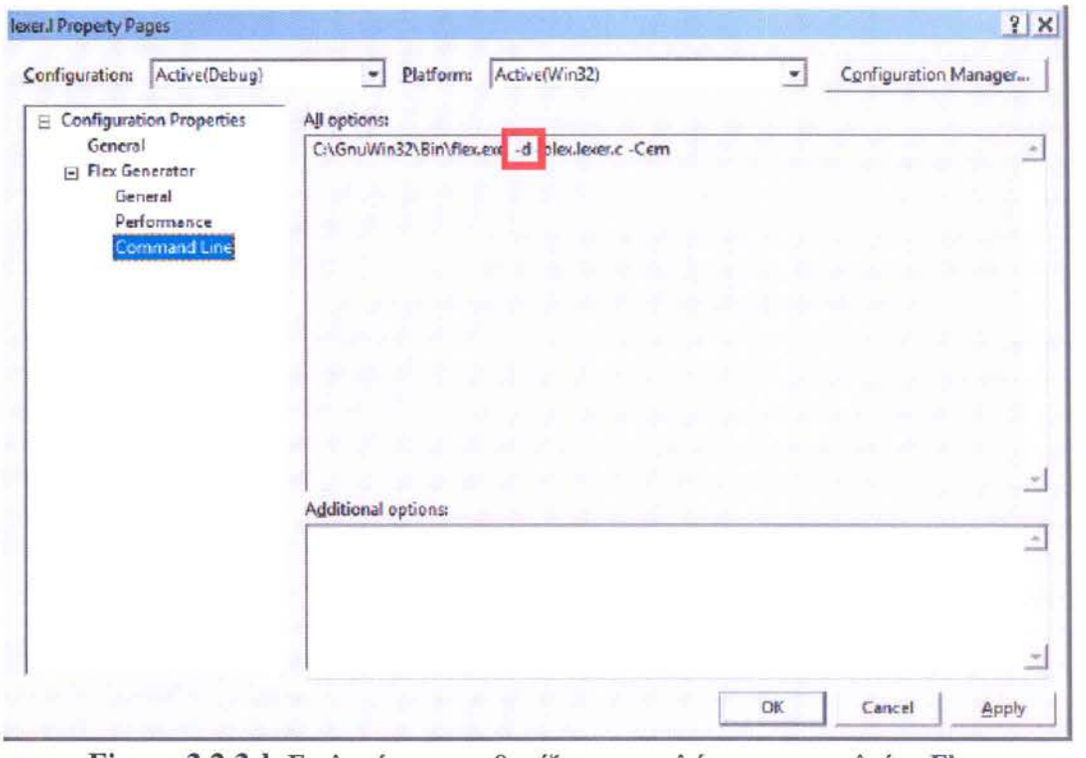

Figure 3.2.3-h Επιλογές που καθορίζουν την κλήση του εργαλείου Flex

3.2.4 Εργαλείο Bison

Το εργαλείο Bison συμβάλει στην αυτόματη ανάπτυξη συντακτικών αναλυτών πέρνωντας ως είσοδε context free grammars σε Baucks Naur form BNF μορφή. Η διαδικασία έχει ως εξής:

1 Περιγράφουμε την λειτουργηκότητα του συντακτικού αναλυτή με την βοήθεια BNF εκφράσεων
2 Μεταγλωττίζουμε την περιγραφή αυτή χτησιμοποιώντας το εργαλείο Bison
3 Παίρνοντας ως έξοδο τον συντακτικό αναλυτή σε γλώσσα C/C++. Το εργαλείο Bison για windows μπορεί να βρεθεί στην ιστοσελίδα http://gnuwin32.sourceforge.net/packages/bison.htm

Η εγκατάσταση είναι απλή. Τρέχουμε το εκτελέσιμο αρχείο και κάνουμε εγκατάσταση του προγράμματος σε κατάλογο που δεν έχει κενό χαρακτήρα π.χ αποφεύγουμε να το κάνουμε εγκατάσταση στον default κατάλογο C:\Program Files\GnuWin32. Για να αποφύγουμε την δυσλειτουργία του εργαλείου τοποθετούμε την εγκατάσταση στον κατάλογο π.χ. C:\GnuWin32. Το Bison απαιτεί το εργαλείο M4 που περιγράφεται στην παρακάτω υποενότητα. Αυτό που πρέπει να προσέξουμε είναι ότι την πρώτη φορά που θα γίνει η εγκατάσταση των προγραμμάτων Flex, Bison

και Μ4 απαιτείται να ακολουθηθεί η διαδικασία που περιγράφεται στην υποενότητα 3.2.3 για την ενημέρωση των μεταβλητών του συστήματος.

3.2.5 M4 Processor

Το εργαλείο Μ4 για windows μπορεί να βρεθεί στην ιστοσελίδα http://gnuwin32.sourceforge.net/packages/m4.htm.Η εγκατάσταση είναι απλή. Τρέχουμε το εκτελέσιμο αρχείο και κάνουμε εγκατάσταση του προγράμματος σε κατάλογο που δεν έχει κενό χαρακτήρα π.χ αποφεύγουμε να το κάνουμε εγκατάσταση στον default κατάλογο C:\Program Files\GnuWin32. Για να αποφύγουμε την δυσλειτουργία του εργαλείου τοποθετούμε την εγκατάσταση στον κατάλογο π.χ. C:\GnuWin32.

3.2.6 FlexBison Rules για Visual Studio

Το Visual Studio παρέχει ένα ολοκληρωμένο περιβάλλον ανάπτυξης εφαρμογών το οποίο επιτρέπει να χρησιμοποιηθούν εξωτερικά εργαλεία κατά την μεταγλώττιση των εφαρμογών. Τα εργαλεία Flex και Bison που καλούνται από το επίπεδο εντολής κάνουν κατάλληλη την χρήση διακοπτών και παραμέτρων προκειμένου να παραχθεί το επιθυμητό αποτέλεσμα.

Η κλήση των δύο εργαλείων Flex και Bison από την γραμμή εντολής μπορεί να αποφευχθεί συντονίζοντας το Visual Studio κατά τέτοιο τρόπο ώστε να είναι δυνατή η αυτόματη κλήση των εργαλείων πριν την μεταγλώττιση του C/C++ κώδικα. Η διαμόρφωση του Visual Studio μπορεί να γίνει αυτόματα κατεβάζοντας το αρχείο FlexBison.Rules από την ακόλουθη ιστοσελίδα(προτείνεται η εγκατάσταση να γίνει στο κατάλογο C:\GnuWin) :
http://msdn.microsoft.com/enus/library/aa730877(VS.80).aspx

Μετά την εγκατάσταση θα είναι δυνατόν να γράφονται οι περιγραφές για τους λεκτικούς και συντακτικούς αναλυτές μέσα στο περιβάλλον του Visual Studio και οι κλήσεις των εργαλείων Flex και Bison θα γίνονται αυτόματα κατά την μεταγλώττιση του project.

Για την ενεργοποίηση των εργαλείων Flex και Bison επιλέγουμε από το κύριο μενού Project->Custom Build Rules όπου προκύπτει το παράθυρο του σχήματος Figureure 3.2.6. Στο παράθυρο αυτό τσεκάρουμε το check box των εργαλείων τα οποία θέλουμε

να ενεργοποιήσουμε και πατάμε ok. Στη συνέχεια όποτε γίνεται η μεταγλώττιση θα καλούνται αυτόματα τα εργαλεία Flex και Bison.

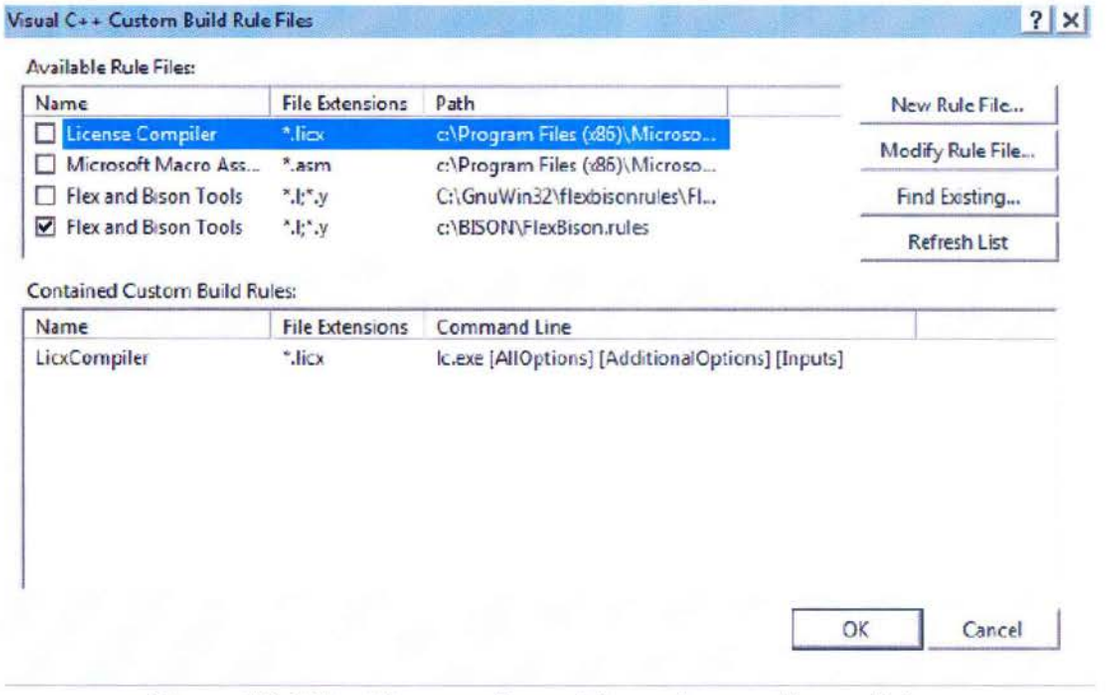

Figure 3.2.6 Παράθυρο κανόνων κλήσης εξωτερικών εργαλείων

ΚΕΦΑΛΑΙΟ 4^0

Δημιουργία προγραμμάτων με την Χρήση των εργαλείων Flex και Bison

4.1 Δημιουργία scanner και parser ενός απλού calculator

Στα παρακάτω παραδείγματα θα δούμε την δημιουργία ενός απλού calculator ,δημιουργώντας ένα scanner(Lexer) και ένα parser .Θα δούμε τις αλλαγές που γίνονται στην πράξη κατά την εκτέλεση του Lexer και του parser μαζί.
Αρχικά δημιουργούμε έναν scanner όπου θα αναγνωρίζει tokens που έχουμε ορίσει στο τμήμα των κανόνων. Όπως έχουμε αναφέρει στο πρώτο κεφάλαιο, κάθε φορά που το πρόγραμμα χρειάζεται έναν token καλεί την yylex(), η οποία διαβάζει μια μικρή είσοδο και επιστρέφει το token που έχει αναγνωρίσει. Όταν χρειαστεί και άλλο token καλεί την yylex() ξανά. Ο scanner λειτουργεί σαν μια υπορουτίνα , αυτό σημαίνει ότι κάθε φορά που επιστρέφει ένα token, θυμάται σε πιο σημείο σταμάτησε και στην επόμενη κλήση του yylex() συνεχίζει από αυτό το σημείο.
Ο scanner στην πράξη επιστρέφει ένα ρεύμα από tokens όπου θα διαχειριστεί ο parser.Τα tokens που επιστρέφει ο scanner, αποτελούνται από δύο μέρη: το token και μια τιμή του. Οι τιμές των tokens αποθηκεύονται στην μεταβλητή yyval. Όταν το Bison παράγει τον parser, αναθέτει αυτόματα τις τιμές των token και δημιουργεί το **.h** με των ορισμών των αριθμών των tokens .Στο παράδειγμά μας 4.1έχουμε αναθέσει τιμές στα tokens με το χέρι.

```
1  /* recognize tokens for the calculator and print them out */
2  %{
3  enum yytokentype {
4  NUMBER = 258,
5  ADD = 259,
6  SUB = 260,
7  MUL = 261,
8  DIV = 262,
9  ABS = 263,
10 EOL = 264
11 };
12 int yylval;
13 %}
14 %%
15 "+"        { 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");
    }
}
```

Figure 4.1Calculator scanner fb4-1.l

Στη συνέχεια δημιουργούμε τον parser για να διαχειριστεί τα tokens που αναγνωρίζονται και επιστρέφονται από τον scanner.

```
%{
#include <stdio.h>
%}
/* declare tokens */
%token NUMBER
%token ADD SUB MUL DIV ABS
%token EOL
%%
calclist: /* nothing */                matches at beginning of input
| calclist exp EOL { printf("= %d\n", $1); }end of an expression
;
exp: factor default $$ = $1
| exp ADD factor { $$ = $1 + $3; }
| exp SUB factor { $$ = $1 - $3; }
;
factor: term default $$ = $1
| factor MUL term { $$ = $1 * $3; }
| factor DIV term { $$ = $1 / $3; }
;
term: NUMBER default $$ = $1
| ABS term { $$ = $2 >= 0? $2 : - $2; }
;
%%
```

```
24  main(int argc, char **argv)
25  {
26   yyparse();
27  }
28   yyerror(char *s)
29  {
30       fprintf(stderr, "error: %s\n", s);
31  }
```

Figure 4.2Simple calculator parser fb4-2.y

Για να μπορέσουμε να εκτελέσουμε τον scanner μέσω του parser πρέπει να κάνουμε κάποιες αλλαγές ώστε να επιτευχθεί η συνεργασία τους. Αρχικά περιλαμβάνουμε στον scanner ένα header file που παράγεται από τον parser, το οποίο περιλαμβάνει τους ορισμούς των tokens και τον ορισμό της yyval. Επίσης διαγράφουμε την testing main() routine από το τρίτο μέρος του scanner.

%{
# include "fb4-2.tab.h" Έχει παραχθεί από τον parser
%}
%% Ίδιοι κανόνες του παραδείγματος 4.1, και καθόλου κώδικα του τρίτου μέρους

4.2Βελτιωμένη έκδοση calculator που δημιουργεί ASTs

Μια από τις πιο ισχυρές δομές δεδομένων που χρησιμοποιούνται στους μεταγλωττιστές είναι το abstract syntax tree (AST). Ένα AST είναι ένα δέντρο μεταγλώττισης που παραλείπει τους κόμβους με τους κανόνες που δεν έχουν κάποιο ενδιαφέρον.

Το παρακάτω παράδειγμα περιγράφει ένα βελτιωμένο calculator που δημιουργεί ASTs parse trees. Λόγω του μεγέθους του, θα το χωρίσουμε σε μερικά αρχεία(sourse files) και θα βάλουμε το μεγαλύτερο μέρος του κώδικα C σε ξεχωριστό αρχείο, το οποίο σημαίνει ότι θα χρειαστούμε επίσης ένα header file C για να δηλώσει τις ρουτίνες και τις δομές δεδομένων που θα χρησιμοποιούνται στα διάφορα αρχεία.

```
1  /*
2  * Declarations for a calculator fb4-3
3  */
4  /* interface to the lexer */
5  extern int yylineno; /* from lexer */
6  void yyerror(char *s, ...);
7  /* nodes in the abstract syntax tree */
```

```
 8  struct ast {
 9          int nodetype;
10          struct ast *l;
11          struct ast *r;
12  };
13  struct numval {
14          int nodetype; /* type K for constant */
15          double number;
16  };
17  /* build an AST */
18  struct ast *newast(int nodetype, struct ast *l, struct ast *r);
19  struct ast *newnum(double d);
20  /* evaluate an AST */
21  double eval(struct ast *);
22  /* delete and free an AST */
23  void treefree(struct ast *);
```

Figure 4.3 Calculator that builds an AST: header fb4-3.h

Η ρουτίνα yyerror έχει ελαφρώς ενισχυθεί για να λαβαίνει arguments στο ύφος του printf.

Το AST αποτελείται από κόμβους. Ο κάθε κόμβος έχει ένα node type και οι διαφορετικοί κόμβοι έχουν διαφορετικά πεδία αλλά προς το παρόν θα περιέχουν μόνο δύο: ένα για ένα pointer που θα δείχνει έως δύο subnodes και ένα που θα περιέχει έναν αριθμό. Οι δύο ρουτίνες newast και newnum δημιουργούν τους ASTs κόμβους. Η eval διασχίζει το AST και επιστρέφει την τιμή της expression που αντιστοιχεί. Η treefree διασχίζει το AST και διαγράφει τους κόμβους του δέντρου.

```
 1  /* calculator with AST */
 2  %{
 3  # include <stdio.h>
 4  # include <stdlib.h>
 5  # include "fb4-3.h"
 6  %}
 7  %union {
 8       struct ast *a;
 9       double d;
10  }
11  /* declare tokens */
12  %token <d> NUMBER
13  %token EOL
14  %type <a> exp factor term
15  %%
16  calclist: /* nothing */
17   | calclist exp EOL {
18        printf("= %4.4g\n", eval($2)); evaluate and print the AST
19        treefree($2);                          free up the AST
20        printf("> ");
21  }
```

```
22  | calclist EOL { printf("> "); } /* blank line or a comment */
23  ;
24  exp: factor
25   | exp '+' factor { $$ = newast('+', $1,$3); }
26   | exp '-' factor { $$ = newast('-', $1,$3);}
27  ;
28  factor: term
29   | factor '*' term { $$ = newast('*', $1,$3); }
30   | factor '/' term { $$ = newast('/', $1,$3); }
31  ;
32  term: NUMBER { $$ = newnum($1); }
33  | '|' term { $$ = newast('|', $2, NULL); }
34  | '(' exp ')' { $$ = $2; }
35  | '-' term { $$ = newast('M', $2, NULL); }
36  ;
37  %%
```

Figure 4-4 Bison parser for AST calculator fb4-4.y

Το παραπάνω παράδειγμα δείχνει τον Bison parser για το calculator AST. Στο πρώτο τμήμα του parser χρησιμοποιεί την δομή %union για να δηλώσει τις τιμές των συμβόλων που χρησιμοποιούνται στο parser. Στον Bison parser κάθε σύμβολο, token και nonterminal έχουν μια τιμή που συνδέεται με αυτό(Κεφάλαιο 2^0). Εξ' ορισμού όλες οι τιμές είναι ακέραιοι αριθμοί. Η δομή %union αναφέρεται σε δύο αριθμούς, τον a που είναι ένας pointer στο AST και τον d ο οποίος είναι ένας double αριθμός. Το δεύτερο τμήμα του parser βλέπουμε τους κανόνες που θα χρησιμοποιηθούν για το calculator AST.Σε αυτή την έκδοση του calculator δημιουργούμε ένα AST για κάθε expression και μετά το αξιολογούμε, εκτυπώνουμε τα αποτελέσματα και στο τέλος το ελευθερώνουμε. Καλούμε την newest() για να δημιουργήσει τον κάθε κόμβο στο AST. Κάθε κόμβος έχει ένα τελεστή, οποίος είναι ένα token.

```
1   /* recognize tokens for the calculator */
2   %option noyywrap nodefault yylineno
3   %{
4   # include "fb4-3.h"
5   # include "fb4-4.tab.h"
6   %}
7   /* float exponent */
8   EXP ([Ee][-+]?[0-9]+)
9   %%
10  "+" |
11  "-" |
12  "*" |
13  "/" |
14  "|" |
15  "(" |
16  ")"             { return yytext[0]; }
```

```
17  [0-9]+"."[0-9]*{EXP}? |
18  "."?[0-9]+{EXP}?{ yylval.d = atof(yytext); return NUMBER; }
19  \n              { return EOL; }
20  "//".*
21  [ \t]           { /* ignore whitespace */ }
22  .               { yyerror("Mystery character %c\n", *yytext); }
23  %%
```

Figure 4-5 Lexer for AST calculator

Στο παραπάνω παράδειγμα ,ο lexer χειρίζεται τους τελεστές με τον ίδιο κανόνα και επιστρέφει yytext[0], δηλαδή επιστρέφει τον ιδιο χαρακτήρα ως token,τους αριθμούς που περιλαμβάνουν κινητή υποδιαστολή και αλλάζει την εσωτερική αναπαράσταση των αριθμών αυτών σε double(πραγματικούς). Οι τιμές των double αριθμών τους αναθέτει στο yylval.d.(Το Flex δεν μπορεί να διαχειριστεί αυτόματα τις τιμές των tokens ,όπως ο Bison).

```
1  # include <stdio.h>
2  # include <stdlib.h>
3  # include <stdarg.h>
4  # include "fb4-3.h"
5  struct ast *
6  newast(int nodetype, struct ast *l, struct ast *r)
7  {
8      struct ast *a = malloc(sizeof(struct ast));
9      if(!a) {
10     yyerror("out of space");
11     exit(0);
12   }
13   a->nodetype = nodetype;
14   a->l = l;
15   a->r = r;
16   return a;
17 }
18 struct ast *
19 newnum(double d)
20 {
21   struct numval *a = malloc(sizeof(struct numval));
22    if(!a) {
23    yyerror("out of space");
24    exit(0);
25   }
26   a->nodetype = 'K';
27   a->number = d;
28   return (struct ast *)a;
29 }
```

```
eval(struct ast *a)
{
double v; calculated value of this subtree
switch(a->nodetype) {
case 'K': v = ((struct numval *)a)->number; break;
case '+': v = eval(a->l) + eval(a->r); break;
case '-': v = eval(a->l) - eval(a->r); break;
case '*': v = eval(a->l) * eval(a->r); break;
case '/': v = eval(a->l) / eval(a->r); break;
case '|': v = eval(a->l); if(v < 0) v = -v; break;
case 'M': v = -eval(a->l); break;
default: printf("internal error: bad node %c\n", a->nodetype);
 }
  return v;
}
void
treefree(struct ast *a)
{
switch(a->nodetype) {
   /* two subtrees */
case '+':
case '-':
case '*':
case '/':
treefree(a->r);
   /* one subtree */
case '|':
case 'M':
treefree(a->l);
   /* no subtree */
case 'K':
free(a);
break;
default: printf("internal error:free badnode %c\n",a->nodetype);
  }
}
void
yyerror(char *s, ...)
{
 va_list ap;
 va_start(ap, s);
 fprintf(stderr, "%d: error: ", yylineno);
 vfprintf(stderr, s, ap);
 fprintf(stderr, "\n");
}
int
main()
{
  printf("> ");
  return yyparse();
}
```

Figure 4-6 C routines for AST calculator

Το παραπάνω αρχείο αποτελείται από ρουτίνες και καλείται από τον parser. Αρχικά χρησιμοποιούνται δύο ρουτίνες newnum και numval που δημιουργούν AST κόμβους, δεσμεύοντας μνήμη όπου και τους τοποθετούν. Όλοι οι AST κόμβοι έχουν nodetype, έτσι ώστε μια «βόλτα» στο δέντρο μπορεί κανείς να διαπιστώσει το είδος των κόμβων. Στη συνέχεια χρησιμοποιούνται δύο ρουτίνες eval και treefree που διασχίζουν το δέντρο. Η κάθε μία διασχίζει το δέντρο κατά βάθος, επισκέπτοντας αναδρομικά το υπο-δέντρο(subtree) του κάθε κόμβου και στην συνέχεια τον ίδιο τον κόμβο. Η ρουτίνα eval επιστρέφει την τιμή του δέντρου ή του υποδέντρου μετά από κάθε κλήση της, ενώ treefree δεν έχει να επιστρέψει τίποτα. Στο τέλος συναντάμε την yyerror και την main. Η yyerror δέχεται μια λίστα από arguments, η οποία αποδεικνύεται βολική όταν παράγει μηνύματα λάθους.

4.3Βελτιωμένη έκδοση ενός calculator

Στην ενότητα αυτή, θα παρουσιάσουμε ένα τελευταίο παράδειγμα, χωρισμένο σε αρχεία ,που επεκτείνει ένα calculator και γίνεται εμφανής η χρήση και η συνεργασία των εργαλείων Flex και Bison. Θα προσθέσουμε ονόματα μεταβλητών, εκχωρήσεων, συγκριτικούς τελεστές, ελέγχους ροής με την χρήση των if/the/else και των while/do, θα δημιουργήσουμε και θα ορίσουμε συναρτήσεις που θα μπορεί ο τελικός χρήστης να έχει πρόσβαση(build-in) και τέλος θα συμπεριλάβουμε μια μικρή ανάκαμψη σφαλμάτων. Στο παρακάτω παράδειγμα ,προσδιορίζουμε μια συνάρτηση και στην συνέχεια την καλούμαι χρησιμοποιώντας μια ενσωματωμένη συνάρτηση ως ένα από τα ορίσματα της συνάρτησης avg (sqrt) .

```
> let avg(a,b) = (a+b)/2;
Defined avg
> avg(3, sqrt(25))
= 4
```

Figure 4-7 Advanced calculator header fb4-6.h

```
/*
* Declarations for a calculator fb4-6
*/
/* interface to the lexer */
extern int yylineno; /* from lexer */
void yyerror(char *s, ...);
/* symbol table */
struct symbol {    /* a variable name */
  char *name;
  double value;
  struct ast *func;  /* stmt for the function */
  struct symlist *syms;   /* list of dummy args */
};
  /* simple symtab of fixed size */
#define NHASH 9997
struct symbol symtab[NHASH];
struct symbol *lookup(char*);
  /* list of symbols, for an argument list */
struct symlist {
  struct symbol *sym;
  struct symlist *next;
};
struct symlist*newsymlist(struct symbol
                          *sym, struct symlist *next;
void symlistfree(struct symlist *sl);
```

Στον πίνακα των συμβόλων, να σύμβολο μπορεί να είναι μεταβλητή και ορισμένη συνάρτηση(user defined function). Στον παραπάνω κώδικα το πεδίο value κρατάει την τιμή του συμβόλου ως μεταβλητή, το πεδίο func δείχνει στο AST που αφορά τον κώδικα της συνάρτησης και το πεδίο syms δείχνει μια συνδεδεμένη λίστα των arguments, τα οποία είναι σύμβολα. Οι συναρτήσεις newsymlist και symlistfree δημιουργεί και ελευθερώνει τα ορίσματα(arguments).

```
/* node types
* + - * / |
* 0-7 comparison ops, bit coded 04 equal, 02 less, 01 greater
* M unary minus
* L expression or statement list
* I IF statement
* W WHILE statement
* N symbol ref
* = assignment
* S list of symbols
* F built in function call
* C user function call
*/
```

```
38  */
39  enum bifs { /* built-in functions */
40  B_sqrt = 1,
41  B_exp,
42  B_log,
43  B_print
44  };
45  /* nodes in the abstract syntax tree */
46  /* all have common initial nodetype */
47  struct ast {
48    int nodetype;
49    struct ast *l;
50    struct ast *r;
51  };
52  struct fncall {      /* built-in function */
53    int nodetype;     /* type F */
54    struct ast *l;
55    enum bifs functype;
56  };
57  struct ufncall {         /* user function */
58    int nodetype;      /* type C */
59    struct ast *l;    /* list of arguments */
60    struct symbol *s;
61  };
62  struct flow {
63    int nodetype;     /* type I or W */
64    struct ast *cond;          /* condition */
65    struct ast *tl;        /* then branch or do list */
66    struct ast *el;   /* optional else branch */
67  };
68  struct numval {
69    int nodetype;         /* type K */
70    double number;
71  };
72  struct symref {
73    int nodetype;         /* type N */
74    struct symbol *s;
75  };
76  struct symasgn {
77    int nodetype;         /* type = */
78    struct symbol *s;
79    struct ast *v;         /* value */
80  };
81  /* build an AST */
82  struct ast *newast(int nodetype, struct ast *l, struct ast *r);
83  struct ast *newcmp(int cmptype, struct ast *l, struct ast *r);
84  struct ast *newfunc(int functype, struct ast *l);
85  struct ast *newcall(struct symbol *s, struct ast *l);
86  struct ast *newref(struct symbol *s);
87  struct ast *newasgn(struct symbol *s, struct ast *v);
88  struct ast *newnum(double d);
89  struct ast *newflow(int nodetype,
```

```
90          struct ast *cond, struct ast *tl, struct ast *tr);
91    /* define a function */
92  void dodef(struct symbol *name,
93          struct symlist *syms, struct ast *stmts);
94    /* evaluate an AST */
95  double eval(struct ast *);
96    /* delete and free an AST */
97  void treefree(struct ast *);
98    /* interface to the lexer */
99  extern int yylineno; /* from lexer */
100 void yyerror(char *s, ...);
```

Αυτή η έκδοση του calculator έχει σημαντικά περισσότερα είδη κόμβων AST. Όπως έχουμε αναφέρει και στο προηγούμενο παράδειγμα κάθε είδος κόμβου ξεκινάει από το nodetype ,όπου μία διάσχιση του δέντρου μπορεί να μας πει το είδος του κάθε κόμβου. Ο βασικός κόμβος ast χρησιμοποιείται επίσης για συγκρίσεις, όπου κάθε είδος σύγκρισης είναι ένας διαφορετικός τύπος και εκφράσεις των λιστών.
Κάθε ενσωματωμένη συνάρτηση έχει ένα fncall κόμβο και αποτελεί ένα όρισμα(function's argument), από την άλλη το enum μας λέει ποια ενσωματωμένη συνάρτηση είναι. Υπάρχουν τρείς βασικές συνάρτησης sqrt,exp και η log, καθώς και η print, μια συνάρτηση που τυπώνει το όρισμά της και επιστρέφει το όρισμα ώ τιμή(value). Οι κλήσεις των συναρτήσεων έχουν τον unfncall κόμβο, ο οποίος είναι είσοδος στον πίνακα συμβόλων και στην AST ,η οποία είναι λίστα από ορίσματα.
Όσον αφορά οι εκφράσεις ελέγχου ροής, if/then/else και while/do χρησιμοποιούν τον flow κόμβο , οποίος περιλαμβάνει την έκφραση του ελέγχου, το κλαδί ή λίστα και το προαιρετικό κλαδί else.
Οι σταθερές παραπέμπονται στο numval, οι αναφορές στα σύμβολο παραπέμπονται στο πεδίο symref(δείχνει το σύμβολο μέσα στο πίνακα συμβόλων) ,οι αναθέσεις (assignments) παραπέμπονται στο symasgn που δείχνει το σύμβολο που του έχει ανατεθεί μια τιμή και δείχνει την τιμή AST. Κάθε AST έχει μια τιμή(value). Η τιμή των if/then/else είναι η τιμή του κλαδιού που αναφέρονται, ενώ η τιμή των while/do είναι η τελευταία τιμή της λίστας. Η τιμή της λίστας των εκφράσεων είναι η τελευταία έκφραση. Τέλος έχουμε διαδικασίες C για την δημιουργία κάθε είδους κόμβου της AST και μια διαδικασία για την δημιουργία καθορισμένων συναρτήσεων.

4.3.1 Δημιουργία parser της βελτιωμένης έκδοσης calculator

Συνεχίζοντας το παράδειγμα μας, παρακάτω παρουσιάζουμε τον parser της βελτιωμένης έκδοσης του calculator.

```
/* calculator with AST */
%{
# include <stdio.h>
# include <stdlib.h>
# include "fb4-6.h"
%}
%union {
  struct ast *a;
  double d;
  struct symbol *s;                /* which symbol */
  struct symlist *sl;
  int fn;                          /* which function */
}
/* declare tokens */
%token <d> NUMBER
%token <s> NAME
%token <fn> FUNC
%token EOL
%token IF THEN ELSE WHILE DO LET
%nonassoc <fn> CMP
%right '='
%left '+' '-'
%left '*' '/'
%nonassoc '|' UMINUS
%type <a> exp stmt list explist
%type <sl> symlist
%start calclist
%%
stmt: IF exp THEN list { $$ = newflow('I', $2, $4, NULL); }
| IF exp THEN list ELSE list { $$ = newflow('I', $2, $4, $6); }
| WHILE exp DO list { $$ = newflow('W', $2, $4, NULL); }
| exp
;
list: /* nothing */ { $$ = NULL; }
| stmt ';' list { if ($3 == NULL)
```

```
36      $$ = $1;
37     else
38      $$ = newast('L', $1, $3);
39     }
40 ;
41 exp: exp CMP exp { $$ = newcmp($2, $1, $3); }
42 | exp '+' exp { $$ = newast('+', $1,$3); }
43 | exp '-' exp { $$ = newast('-', $1,$3);}
44 | exp '*' exp { $$ = newast('*', $1,$3); }
45 | exp '/' exp { $$ = newast('/', $1,$3); }
46 | '|' exp { $$ = newast('|', $2, NULL); }
47 | '(' exp ')' { $$ = $2; }
48 | '-' exp %prec UMINUS { $$ = newast('M', $2, NULL); }
49 | NUMBER { $$ = newnum($1); }
50 | NAME { $$ = newref($1); }
51 | NAME '=' exp { $$ = newasgn($1, $3); }
52 | FUNC '(' explist ')' { $$ = newfunc($1, $3); }
53 | NAME '(' explist ')' { $$ = newcall($1, $3); }
54 ;
55 explist: exp
56 | exp ',' explist { $$ = newast('L', $1, $3); }
57 ;
58 symlist: NAME { $$ = newsymlist($1, NULL); }
59 | NAME ',' symlist { $$ = newsymlist($1, $3); }
60 ;
61 calclist: /* nothing */
62 | calclist stmt EOL {
```

```
63         printf("= %4.4g\n> ", eval($2));
64         treefree($2);
65     }
66 | calclist LET NAME '(' symlist ')' '=' list EOL {
67         dodef($3, $5, $8);
68         printf("Defined %s\n> ", $3->name); }
69 | calclist error EOL { yyerrok; printf("> "); }
70 ;
```

Figure 4-8 Advanced calculator parser fb4-7.y

Η δομή %union ορίζει τα διάφορα είδη των τιμών των συμβόλων. Η τιμή ενός συμβόλου μπορεί να είναι ένας δείκτης στον πίνακα συμβόλων, μία λίστα από σύμβολα ή και μια υποκατηγορία μιας σύγκρισης.

Ο παραπάνω parser δηλώνει ένα νέο token FUNC για τις ενσωματωμένες συναρτήσεις, όπου η τιμή του υποδεικνύει ποια συνάρτηση είναι και έξι δεσμευμένες λέξεις(IF ,THEN, ELSE, WHILE, DO, LET). Το token CMP δηλώνει

τους έξι τελεστές σύγκρισης και η τιμή του υποδεικνύει ποιος τελεστής είναι. Η λίστα προτεραιότητας των δηλώσεων ξεκινάει με ένα νέο CMP και τον τελεστή '='. Στη συνέχεια στο τμήμα των δηλώσεων το %start προσδιορίζει το ανώτερο επίπεδο των κανόνων, που σημαίνει ότι δεν χρειάζεται να το συντάξουμε στην αρχή του parser.

Στο τμήμα των κανόνων , η γραμματική διακρίνεται από τις δηλώσεις(stmt) και τις εκφράσεις(exp). Μια δήλωση μπορεί να είναι είτε μια ροή ελέγχου(if/then/else ή while/do), είτε μία έκφραση. Οι δηλώσεις if και while δέχονται λίστες δηλώσεων, όπου κάθε δήλωση της λίστας ακολουθείται από ερωτηματικό. Κάθε κανόνας όταν ταιριάζει μια δήλωση καλεί μια ρουτίνα για να δημιουργήσει τον κατάλληλο κόμβο AST. Όσον αφορά τις εκφράσεις, στον πρώτο κανόνα το CMP χειρίζεται τους έξι τελεστές χρησιμοποιώντας την τιμή του για πει ποιος τελεστής είναι και ο κανόνας των αναθέσεων δημιουργεί ένα κόμβο ανάθεσης.

Υπάρχουν ξεχωριστοί κανόνες για τις ενσωματωμένες συναρτήσεις που προσδιορίζονται από το δεσμευμένο όνομα(FUNC) και από τις συναρτήσεις που προσδιορίζονται από ένα σύμβολο(NAME).

Ο κανόνας explist, λίστα από εκφράσεις, δημιουργεί το AST των εκφράσεων που χρησιμοποιούνται για τα arguments της συνάρτησης που καλείται. Ο κανόνας symlist, λίστα από σύμβολα, δημιουργεί μια συνδεδεμένη λίστα συμβόλων που σχετίζονται με τα εικονικά arguments που ορίζονται σε μια συνάρτηση.

Το τελευταίο κομμάτι της γραμματικής είναι η ανώτερη βαθμίδα, η οποία αναγνωρίζει μια λίστα από δηλώσεις και δηλώσεις μια συνάρτησης . ταξινομεί ένα AST μιας δήλωσης ,τυπώνει το αποτέλεσμα και στην συνέχεια απελευθερώνει το AST.

4.3.2 Δημιουργία Lexer της βελτιωμένης έκδοσης calculator

```
1  /* recognize tokens for the calculator */
2  %option noyywrap nodefault yylineno
3  %{
4    # include "fb4-6.h"
5    # include "fb4-7.tab.h"
6  %}
7  /* float exponent */
8  EXP ([Ee][-+]?[0-9]+)
9  %%
```

```
10  /* single character ops */
11  "+" |
12  "-" |
13  "*" |
14  "/" |
15  "=" |
16  "|" |
17  "," |
18  ";" |
19  "(" |
20  ")"     { return yytext[0]; }
21  /* comparison ops, all are a CMP token */
22  ">"     { yylval.fn = 1; return CMP; }
23  "<"     { yylval.fn = 2; return CMP; }
24  "<>"    { yylval.fn = 3; return CMP; }
25  "=="    { yylval.fn = 4; return CMP; }
26  ">="    { yylval.fn = 5; return CMP; }
27  "<="    { yylval.fn = 6; return CMP; }
28  /* keywords */
29  "if"    { return IF; }
30  "then"  { return THEN; }
31  "else"  { return ELSE; }
32  "while" { return WHILE; }
33  "do"    { return DO; }
34  "let"   { return LET;}
35       /* built-in functions */
36  "sqrt"  { yylval.fn = B_sqrt; return FUNC; }
37  "exp"   { yylval.fn = B_exp; return FUNC; }
38  "log"   { yylval.fn = B_log; return FUNC; }
39  "print" { yylval.fn = B_print; return FUNC; }


40      /* names */
41  [a-zA-Z][a-zA-Z0-9]*{ yylval.s = lookup(yytext); return NAME; }
42  [0-9]+"."[0-9]*{EXP}? |
43  "."?[0-9]+{EXP}?    { yylval.d = atof(yytext); return NUMBER; }
44  "//".*
45  [ \t]     /* ignore whitespace */
46  \\\n    { printf("c> "); } /* ignore line continuation */
47  \n      { return EOL; }
48  .       { yyerror("Mystery character %c\n", *yytext); }
49  %%
```

Figure 4-9 Advanced calculator lexer fb4-8.l

Στο τμήμα των κανόνων του Lexer υπάρχουν νέοι μεμονωμένοι χαρακτήρες τελεστές. Οι έξι τελεστές σύγκρισης επιστρέφουν όλοι ένα CMP token με την αντίστοιχη λεκτική τιμή του κάθε τελεστή. Οι έξι λέξεις κλειδιά(keywords) και οι τέσσερις ενσωματωμένες συναρτήσεις αναγνωρίζονται από τα αντίστοιχα literal patterns. Η

νέα γραμμή(EOL) σηματοδοτεί το τέλος μιας συμβολοσειράς της εισόδου. Το ταίριασμα του backslash και η αλλαγή γραμμής δεν επιστρέφουν τίποτα στον parser.

4.3.3 Δημιουργία των βοηθητικών συναρτήσεων του βελτιωμένου calculator

Τέλος, έχουμε ένα αρχείο με τον βοηθητικό κώδικα. Μερικά σημεία του αρχείου είναι ίδια με το παράδειγμα που παρουσιάσαμε στην ενότητα 4.2. Αρχικά , βλέπουμε στον κώδικά μας, την διαχείριση του πίνακα συμβόλων.

Figure 4-10 Advanced calculator helper functions fb4-9func.c

```
/*
* helper functions for fb4-9
*/
# include <stdio.h>
# include <stdlib.h>
# include <stdarg.h>
# include <string.h>
# include <math.h>
# include "fb4-6.h"
/* symbol table */
/* hash a symbol */
static unsigned
symhash(char *sym)
{
  unsigned int hash = 0;
  unsigned c;
  while(c = *sym++) hash = hash*9 ^ c;
  return hash;
}
struct symbol *
lookup(char* sym)
{
  struct symbol *sp = &symtab[symhash(sym)%NHASH];
  int scount = NHASH; /* how many have we looked at */
  while(--scount >= 0) {
    if(sp->name && !strcmp(sp->name, sym)) { return sp; }
    if(!sp->name) { /* new entry */
        sp->name = strdup(sym);
        sp->func = NULL;
        sp->syms = NULL;
```

```
31          return sp;
32         }
33      if(++sp >= symtab+NHASH) sp = symtab; /* try the next entry *
34     }
35     yyerror("symbol table overflow\n");
36     abort(); /* tried them all, table is full */
37  }
```

```
38  struct ast *
39  newast(int nodetype, struct ast *l, struct ast *r)
40  {
41    struct ast *a = malloc(sizeof(struct ast));
42    if(!a) {
43       yyerror("out of space");
44       exit(0);
45     }
46    a->nodetype = nodetype;
47    a->l = l;
48    a->r = r;
49    return a;
50  }
51  struct ast *
52  newnum(double d)
53  {
54    struct numval *a = malloc(sizeof(struct numval));
55    if(!a) {
56          yyerror("out of space");
57          exit(0);
58    }
59  a->nodetype = 'K';
60     a->number = d;
61     return (struct ast *)a;
62  }
```

Ακολουθούν οι διαδικασίες για την δημιουργία των AST κόμβων και του symlists. Όλες διαθέτουν κόμβο και στη συνέχεια συμπληρώνουν τα πεδία κατάλληλα για το τύπο του κόμβου. Παρατηρούμε μιε εκτεταμένη έκδοση του treefree, η οποία διασχίζει αναδρομικά το AST και ελευθερώνει όλους τους κόμβου από το δέντρο.

```
63  struct ast *
64  newcmp(int cmptype, struct ast *l, struct ast *r)
65  {
66    struct ast *a = malloc(sizeof(struct ast));
67    if(!a) {
68        yyerror("out of space");
69        exit(0);
70    }
71    a->nodetype = '0' + cmptype;
72    a->l = l;
73    a->r = r;
74    return a;
75  }
76  struct ast *
77  newfunc(int functype, struct ast *l)
78  {
79    struct fncall *a = malloc(sizeof(struct fncall));
```

```
79    struct fncall *a = malloc(sizeof(struct fncall));
80    if(!a) {
81     yyerror("out of space");
82         exit(0);
83       }
84    a->nodetype = 'F';
85    a->l = l;
86    a->functype = functype;
87    return (struct ast *)a;
88  }
89  struct ast *
90  newcall(struct symbol *s, struct ast *l)
91  {
92    struct ufncall *a = malloc(sizeof(struct ufncall));
93    if(!a) {
94       yyerror("out of space");
95       exit(0);
96    }
97    a->nodetype = 'C';
98    a->l = l;
99    a->s = s;
100   return (struct ast *)a;
101 }
102 struct ast *
103 newref(struct symbol *s)
104 {
105   struct symref *a = malloc(sizeof(struct symref));
106   if(!a) {
107      yyerror("out of space");
108      exit(0);
```

```
109     }
110     a->nodetype = 'N';
111     a->s = s;
112     return (struct ast *)a;
113   }
114   struct ast *
115   newasgn(struct symbol *s, struct ast *v)
116   {
117     struct symasgn *a = malloc(sizeof(struct symasgn));
118     if(!a) {
119        yyerror("out of space");
120        exit(0);
121     }
122     a->nodetype = '=';
123     a->s = s;
124     a->v = v;
125     return (struct ast *)a;
126   }

127   struct ast *
128   newflow(int nodetype, struct ast *cond,
129      struct ast *tl, struct ast *el)
130   {
131     struct flow *a = malloc(sizeof(struct flow));
132     if(!a) {
133        yyerror("out of space");
134       exit(0);
135     }
136     a->nodetype = nodetype;
137     a->cond = cond;
138     a->tl = tl;
139     a->el = el;
140     return (struct ast *)a;
141   }
142   /* free a tree of ASTs */
143   void
144   treefree(struct ast *a)
145   {
146     switch(a->nodetype) {
147         /* two subtrees */
148     case '+':
149     case '-':
150     case '*':
151     case '/':
152     case '1': case '2': case '3': case '4': case '5': case '6':
153     case 'L':
154         treefree(a->r);
```

```
155       /* one subtree */
156     case '|':
157     case 'M': case 'C': case 'F':
158         treefree(a->l);
159      /* no subtree */
160     case 'K': case 'N':
161         break;
162     case '=':
163        free( ((struct symasgn *)a)->v);
164       break;
165      /* up to three subtrees */
166     case 'I': case 'W':
167       free( ((struct flow *)a)->cond);
168       if( ((struct flow *)a)->tl)treefree(((struct flow *)a)->tl);
169       if( ((struct flow *)a)->el)treefree(((struct flow *)a)->el);
170       break;
171    default:printf("internal error:free bad node%c\n",a->nodetype);
172     }
173     free(a); /* always free the node itself */
174   }
175   struct symlist *
176   newsymlist(struct symbol *sym, struct symlist *next)

177   {
178      struct symlist *sl = malloc(sizeof(struct symlist));
179      if(!sl) {
180         yyerror("out of space");
181         exit(0);
182     }
183     sl->sym = sym;
184     sl->next = next;
185     return sl;
186   }
187   /* free a list of symbols */
188   void
189   symlistfree(struct symlist *sl)
190   {
191     struct symlist *nsl;
192     while(sl) {
193        nsl = sl->next;
194        free(sl);
195        sl = nsl;
196     }
197   }
```

Το βασικό μέρος του calculator είναι η eval, η οποία ταξινομεί την ανάπτυξη του AST στον parser. Οι συγκρίσεις που γίνονται επιστρέφουν 0 ή 1 αναλόγως αν επιτύχουν. Όσο αφορά τις εκφράσεις ,διασχίζουμε κατά βάθος το δέντρο για να υπολογίσουμε τις τιμές. Το AST είναι εύκολο να εφαρμόσει τα if/then/else: αξιολογώντας την κατάσταση του AST αποφασίζει ποιο κλαδί θα πάρει και στη συνέχεια αξιολογεί το AST σύμφωνα με το μονοπάτι που έχει επιλέξει. Για την

αξιολόγηση των βρόγχων while/do (ένας βρόγχος στην eval αξιολογεί την κατάσταση του AST και έπειτα το σώμα του) επαναλαμβάνεται μέχρι η κατάσταση του AST παραμένει true(αληθής).Κάθε AST, που οι αναφορές των μεταβλητών έχουν αλλάξει από μια ανάθεση και έχουν αλλάξει τιμές, αξιολογείται.

```
198 static double callbuiltin(struct fncall *);
199 static double calluser(struct ufncall *);
200 double
201 eval(struct ast *a)
202 {
203   double v;
204   if(!a) {
205     yyerror("internal error, null eval");
206     return 0.0;
207   }
208   switch(a->nodetype) {
209     /* constant */
210   case 'K': v = ((struct numval *)a)->number; break;
211     /* name reference */
212   case 'N': v = ((struct symref *)a)->s->value; break;
213    /* assignment */
214   case '=': v = ((struct symasgn *)a)->s->value =
215      eval(((struct symasgn *)a)->v); break;
216    /* expressions */
217   case '+': v = eval(a->l) + eval(a->r); break;
218   case '-': v = eval(a->l) - eval(a->r); break;
219   case '*': v = eval(a->l) * eval(a->r); break;
220   case '/': v = eval(a->l) / eval(a->r); break;
221   case '|': v = fabs(eval(a->l)); break;
222   case 'M': v = -eval(a->l); break;
223    /* comparisons */
224   case '1': v = (eval(a->l) > eval(a->r))? 1 : 0; break;
225   case '2': v = (eval(a->l) < eval(a->r))? 1 : 0; break;
226   case '3': v = (eval(a->l) != eval(a->r))? 1 : 0; break;
227   case '4': v = (eval(a->l) == eval(a->r))? 1 : 0; break;
228   case '5': v = (eval(a->l) >= eval(a->r))? 1 : 0; break;
229   case '6': v = (eval(a->l) <= eval(a->r))? 1 : 0; break;
230     /* control flow */
231 /* null expressions allowed in the grammar,so check for them */
232    /* if/then/else */
233   case 'I':
234   if( eval( ((struct flow *)a)->cond) != 0){check the condition
235      if( ((struct flow *)a)->tl) { the true branch
236            v = eval( ((struct flow *)a)->tl);
237         } else
238           v = 0.0; /* a default value */
239       } else {
240       if( ((struct flow *)a)->el) { the false branch
241          v = eval(((struct flow *)a)->el);
```

```
242         } else
243             v = 0.0; /* a default value */
244     }
245     break;
246         /* while/do */
247      case 'W':
248      v = 0.0; /* a default value */
249      if( ((struct flow *)a)->tl) {
250      /*evaluatethe condition*/
251      while( eval(((struct flow *)a)->cond) != 0)
252      /*evaluate the target statements*/
253      v = eval(((struct flow *)a)->tl);
254      }
255      break; /* value of last statement is value of while/do */
256           /* list of statements */
257       case 'L': eval(a->l); v = eval(a->r); break;
258       case 'F': v = callbuiltin((struct fncall *)a); break;
259       case 'C': v = calluser((struct ufncall *)a); break;
260       default: printf("internal error: bad node %c\n", a->nodetype);
261        }
262       return v;
263     }
```

Η ενσωμάτωση των συναρτήσεων είναι σχετικά απλή. Αρχικά θα καθοριστεί ποια συνάρτηση είναι και έπειτα θα γίνει η κλήση του αντίστοιχου κώδικα της συνάρτησης.

```
264  static double
265  callbuiltin(struct fncall *f)
266  {
267     enum bifs functype = f->functype;
268     double v = eval(f->l);
269    switch(functype) {
270    case B_sqrt:
271       return sqrt(v);
272    case B_exp:
273      return exp(v);
274    case B_log:
275      return log(v);
276    case B_print:
277      printf("= %4.4g\n", v);
278      return v;
279    default:
280     yyerror("Unknown built-in function %d", functype);
281     return 0.0;
282     }
283  }
```

Ο ορισμός της συνάρτησης αποτελείται από το όνομα της συνάρτησης, από την λίστα των εικονικών arguments και ένα AST που αντιπροσωπεύει το σώμα της συνάρτησης. Ο καθορισμός μιας συνάρτησης απλά αποθηκεύει την λίστα των arguments και την είσοδο του AST στο πίνακα συμβόλων της συνάρτησης.

```
284   /* define a function */
285 void
286 dodef(struct symbol *name,struct symlist *syms,struct ast *func)
287 {
288   if(name->syms) symlistfree(name->syms);
289   if(name->func) treefree(name->func);
290   name->syms = syms;
291   name->func = func;
292 }
```

Ας υποθέσουμε ότι ορίζουμε μια συνάρτηση για τον υπολογισμό του μέγιστου μεταξύ των arguments:

> let max(x,y) = if x >= y then x; else y;;

> max(4+5,6+7)

Η συνάρτηση έχει δύο εικονικά arguments, x και y. Όταν καλείται η συνάρτηση ο ταξινομητής κάνει τα εξής βήματα:

- Αξιολογεί τα πραγματικά arguments, 4+5 και 6+7 στην περίπτωση μας.
- Αποθηκεύει τις τρέχουσες τιμές των εικονικών arguments και αναθέτει τις τιμές των πραγματικών στις εικονικές.
- Αξιολογεί το σώμα της συνάρτησης, όπου θα χρησιμοποιεί τις τιμές των πραγματικών arguments όταν αναφέρεται στα εικονικά arguments.
- Τοποθετεί τις παλιές τιμές των εικονικών.
- Επιστρέφει την τιμή της έκφρασης του σώματος της συνάρτησης.

Ο παρακάτω κώδικας κάνει τα εξής:την μέτρηση των arguments , διαθέτει δύο προσωρινούς πίνακες για τις παλιές και τις νέες τιμές των εικονικών arguments, και στην συνέχεια κάνει τα παραπάνω βήματα.

```
293 static double
294 calluser(struct ufncall *f)
295 {
296   struct symbol *fn = f->s; /* function name */
297   struct symlist *sl; /* dummy arguments */
298   struct ast *args = f->l; /* actual arguments */
299   double *oldval, *newval; /* saved arg values */
300   double v;
301   int nargs;
302   int i;
303   if(!fn->func) {
304      yyerror("call to undefined function", fn->name);
305      return 0;
306  }
307     /* count the arguments */
308   sl = fn->syms;
309   for(nargs = 0; sl; sl = sl->next)
310     nargs++;
311   /* prepare to save them */
312   oldval = (double *)malloc(nargs * sizeof(double));
313   newval = (double *)malloc(nargs * sizeof(double));
314   if(!oldval || !newval) {
315       yyerror("Out of space in %s", fn->name); return 0.0;
316 }
317 /* evaluate the arguments */
318 for(i = 0; i < nargs; i++) {
319     if(!args) {
320         yyerror("too few args in call to %s", fn->name);
321         free(oldval); free(newval);
322         return 0.0;
323      }
324     if(args->nodetype == 'L') { /* if this is a list node */
325          newval[i] = eval(args->l);
326          args = args->r;
```

```
327       } else { /* if it's the end of the list */
328           newval[i] = eval(args);
329           args = NULL;
330       }
331   }
332       /* save old values of dummies, assign new ones */
333       sl = fn->syms;
334       for(i = 0; i < nargs; i++) {
335           struct symbol *s = sl->sym;
336           oldval[i] = s->value;
337           s->value = newval[i];
338          sl = sl->next;
339       }
340       free(newval);
341          /* evaluate the function */
342       v = eval(fn->func);
343          /* put the real values of the dummies back */
344       sl = fn->syms;
345       for(i = 0; i < nargs; i++) {
346           struct symbol *s = sl->sym;
347           s->value = oldval[i];
348          sl = sl->next;
349       }
350       free(oldval);
351       return v;
352   }
```

Η χρήση του βελτιωμένου calculator που προαναφερθήκαμε είναι αρκετά ευέλικτο για να κάνουμε χρήσιμους υπολογισμούς. Στο παρακάτω παράδειγμα παρουσιάζουμε μια συνάρτηση που υπολογίζει την τετραγωνική ρίζα επαναληπτικά με την μέθοδο του Newton.

Παράδειγμα 4-10 Computing square roots with the calculator

Defined sq

> let avg(a,b)=(a+b)/2;

Defined avg

> sq(10)

= 3.162

> sqrt(10)

= 3.162

> sq(10)-sqrt(10)

= 0.000178 accurate to better than the .001 cutoff

Βιβλιογραφία

[1] John Levine, flex & bison, Text Processing Tools (2009), Εκδόσεις O'Reilly Media

[2] *The flex Manual* documents flex version 2.5.35 , September 2007. http://flex.sourceforge.net/manual/index.html#Top

[3 Bison 2.5 http://www.gnu.org/software/bison/manual/html_node/index.html

[4] Flex, version 2.5, A fast scanner generator, http://www.cs.princeton.edu/~appel/modern/c/software/flex/flex.html

[5] Parsing, http://www.multilingualarchive.com/ma/enwiki/en/Parsing

[6]Flex and BISON, http://aquamentus.com/flex_bison.html#4

[7] Yacc/Bison , http://oss.sgi.com/LDP/LDP/LG/issue87/ramankutty.html

[8] Vern Paxson, Flex version 2.5 A fast scanner generator, Edition 2.5, March 1995, http://www-scf.usc.edu/~csci410/handouts/flex.pdf

.

[9] Lan Gao, Flex Tutorial, http://alumni.cs.ucr.edu/~lgao/teaching/flex.html

[10] Νικόλαος Σ. Παπασπύρου και Εμμανουήλ Σ. Σκορδαλάκης, Μεταγλωττιστές(2003), Εκδόσεις Συμμετρία

[11] http://en.wikipedia.org/

[12] Anthony A. Aaby,Compiler Construction using Flex and Bison, Version of April 22, 2005

[13] M.E.Lesk & E.Schmidt, Lex – A Lexical Analyzer Generator:

[14] Charles Donnelly and Richard Stallman, Bison, The Yacc-compatible Parser Generator, Bison Version 2.5, 14 May 2011

[15] Παναγιώτης Πιντέλας , Μεταγλωττιστές, Πανεπιστημιακές σημειώσεις ,Ε.Α.Π (2003)