Δ Ι Ε Θ Ν Ε Σ Π Α Ν Ε Π Ι Σ Τ Η Μ Ι Ο Τ Η Σ Ε Λ Λ Α Δ Ο Σ
Σ Χ Ο Λ Η Μ Η Χ Α Ν Ι Κ Ω Ν
Τ Μ Η Μ Α Μ Η Χ Α Ν Ι Κ Ω Ν Π Λ Η Ρ Ο Φ Ο Ρ Ι Κ Η Σ ,
Υ Π Ο Λ Ο Γ Ι Σ Τ Ω Ν Κ Α Ι Τ Η Λ Ε Π Ι Κ Ο Ι Ν Ω Ν Ι Ω Ν

ΛΕΚΤΙΚΟΣ ΚΑΙ ΣΥΝΤΑΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ

ΜΕΤΑΓΛΩΤΤΙΣΤΕΣ

Ομαδική Εργασία:

Αναστασιάδης Αλκίνοος (20003)

Ζήνα Ελένη (20046)

Επιβλέπων:

Λάντζος Θεόδωρος

ΣΕΡΡΕΣ, 2024

Περιεχόμενα

Εισαγωγή

Ο λεκτικός και συντακτικός αναλυτής είναι βασικά εργαλεία για την ανάπτυξη μεταγλωττιστών και γλωσσών προγραμματισμού. Το Flex χρησιμοποιείται για τη δημιουργία λεκτικών αναλυτών που αναγνωρίζουν μοτίβα κειμένου και τα μετατρέπουν σε tokens, ενώ το Bison χρησιμοποιείται για τη δημιουργία συντακτικών αναλυτών που οργανώνουν αυτά τα tokens σύμφωνα με συντακτικούς κανόνες της γλώσσας. Μέσω της συνδυαστικής χρήσης τους, μπορούμε να δημιουργήσουμε προγράμματα που κατανοούν και επεξεργάζονται σύνθετες γλωσσικές δομές, επιτρέποντας την ανάπτυξη πλήρως λειτουργικών μεταγλωττιστών.

Σκοπός της εργασίας

Ο σκοπός της εργασίας μας ήταν η δημιουργία ενός απλού λεκτικού και συντακτικού αναλυτή για μια νέα γλώσσα προγραμματισμού, την Greek Programming Language (GPL). Η GPL σχεδιάστηκε με λέξεις-κλειδιά και σύνταξη βασισμένα στην ελληνική γλώσσα, προσφέροντας μια πιο προσιτή και φιλική προς τους ελληνόφωνους χρήστες εμπειρία προγραμματισμού. Μέσω της χρήσης των εργαλείων Flex και Bison, αναπτύξαμε έναν αναλυτή που αναγνωρίζει τα μοτίβα και τις συντακτικές δομές της GPL, επιτρέποντας στους προγραμματιστές να γράφουν και να εκτελούν προγράμματα στη νέα αυτή γλώσσα.

Μεθοδολογία

Η μεθοδολογία που ακολουθήσαμε για την ανάπτυξη της GPL περιλάμβανε τη χρήση κανονικών εκφράσεων (regex) στο Flex για την αναγνώριση και εξαγωγή των λεκτικών μονάδων (tokens) από τον κώδικα που γράφτηκε στη γλώσσα μας. Αυτές οι λεκτικές μονάδες μεταβιβάστηκαν στη συντακτική ανάλυση, που υλοποιήθηκε με το Bison, όπου ορίσαμε τη σωστή σύνταξη των δομών της GPL. Μέσω αυτής της διαδικασίας, διασφαλίσαμε ότι κάθε πρόγραμμα που γράφτηκε στη GPL ακολουθεί τους προκαθορισμένους συντακτικούς κανόνες, επιτρέποντας την αναγνώριση και εκτέλεση έγκυρων προγραμμάτων στη νέα μας γλώσσα. Ο παρακάτω πίνακας δείχνει τις βασικές δομές που χρησιμοποιήσαμε στην GPL:

Δομή	**Περιγραφή**	**Παράδειγμα**
# lib	Δήλωση βιβλιοθηκών	(όπως είναι)
kyrio_meros() {…}	Κύρια συνάρτηση	kyrio_meros() {…}
grapse(“…”);	Εμφάνιση τιμών ή μηνυμάτων	grapse(“Enter a number:”); ή grapse(a,b,c);
diabase(“…”);	Εγγραφή τιμών	diabase(a,b,c);
epestrepse …;	Επιστροφή τιμών	epestrepse 0; ή epestrepse x;
(*…*)	Σχόλια	(*This is a comment*)
akeraios	Τύπος δεδομένων	akeraios x1,x2;
pragmatikos	Τύπος δεδομένων	pragmatikos y;
leksh	Τύπος δεδομένων	leksh name;
an (…) {…} alliws {…}	Δομή επανάληψης	an (x > 0) { grapse("Positive"); } alliws { grapse("Negative"); }
oso (…) {…}	Δομή επανάληψης	oso (x < 10) { x = x + 1; }
gia (…; …; …) {…}	Δομή επανάληψης	gia (i = 0; i < 10; i = i + 1) { grapse(i); }
+,-,*,/,=	Αριθμητικοί τελεστές	a = b + c;
<,>,==,<=,>=,!=	Συγκριτικοί τελεστές	an (a != b) { grapse("Different"); }
(,),{,},(κόμμα),(ερωτηματικό)	Σημεία στίξης	a = (b + c) * d;

Υλοποίηση

Κατά τη διαδικασία εκτέλεσης, ο λεκτικός αναλυτής διαβάζει το πρόγραμμα εισόδου και αναγνωρίζει τα διάφορα λεκτικά, ενώ ο συντακτικός αναλυτής χρησιμοποιεί τα tokens που επιστρέφονται για να αναγνωρίσει τη συντακτική δομή του προγράμματος. Ο κώδικας στον οποίο θα γίνει η ανάλυση, διαβάζεται από ένα αρχείο κειμένου και το αποτέλεσμα της ανάλυσης εκτυπώνεται στο τερματικό. Αν όλα πάνε καλά, το πρόγραμμα ολοκληρώνεται με επιτυχία, αλλιώς εμφανίζονται σφάλματα σύνταξης που πρέπει να διορθωθούν. Τα βήματα εκτέλεσης του λεκτικού και του συντακτικού αναλυτή είναι τα ακόλουθα:

1. **flex project.l**: Αυτή η εντολή χρησιμοποιεί το Flex (Fast Lexical Analyzer Generator) για να δημιουργήσει έναν λεκτικό αναλυτή από ένα αρχείο περιγραφής, το **project.l**. Το αρχείο αυτό περιέχει τους κανόνες που περιγράφουν πώς θα αναγνωρίζονται και θα επιστρέφονται τα tokens της γλώσσας προγραμματισμού που επεξεργάζεται το πρόγραμμα.
2. **bison -d project.y**: Αυτή η εντολή χρησιμοποιεί το Bison για να δημιουργήσει έναν συντακτικό αναλυτή από το αρχείο περιγραφής γραμματικής **project.y**. Αυτό το αρχείο περιέχει τους κανόνες που περιγράφουν τη σύνταξη της γλώσσας προγραμματισμού.
3. **gcc -o a.out project.tab.c lex.yy.c**: Αυτή η εντολή χρησιμοποιεί τον διερμηνέα GCC για να μεταγλωττίσει τα πηγαία αρχεία που παρήχθησαν από το Flex και το Bison, δηλαδή τα αρχεία **project.tab.c** και **lex.yy.c**, και να δημιουργήσει ένα εκτελέσιμο αρχείο που ονομάζεται **a.out**.
4. **./a.out**: Αυτή η εντολή εκτελεί το πρόγραμμα που παράχθηκε από το GCC, το εκτελέσιμο αρχείο **a.out**.

Όλος ο κώδικας, τα παραπάνω βήματα, καθώς και πολλά έγκυρα παραδείγματα εκτέλεσης, βρίσκονται στην ιστοσελίδα του GitHub όπου έχουμε το δικό μας repository [1].

project.l:

```
%option caseless

%{
#include <stdio.h>
#include "project.tab.h"
void ret_print();
void yyerror();
%}

%x COMMENT2
digit [0-9]


%%
"(*" BEGIN(COMMENT2);
<COMMENT2>[^)*\n]+
<COMMENT2>\n
<COMMENT2><<EOF>> yyerror("EOF in comment");
<COMMENT2>"*)" BEGIN(INITIAL);
<COMMENT2>[*)]

"akeraios" { ret_print(); return AKER; }
"pragmatikos" { ret_print(); return PRAG; }
"leksh" { ret_print(); return LEKSH; }
"grapse" { ret_print(); return GRAPSE; }
"diabase" { ret_print(); return DIABASE; }
"epestrepse" { ret_print(); return EPESTREPSE; }

"kyrio_meros()" { ret_print(); return KYRIO_MEROS; }

"oso" { ret_print(); return OSO; }
"gia" { ret_print(); return GIA; }
"an" { ret_print(); return AN; }
"alliws" { ret_print(); return ALLIWS; }

"#"  {ret_print(); return INC; }
"lib"  {ret_print(); return LIB; }
```

```
"+"  { ret_print(); return '+'; }
"-"  { ret_print(); return '-'; }
"*"  { ret_print(); return '*'; }
"/"  { ret_print(); return '/'; }

"("  { ret_print(); return '('; }
")"  { ret_print(); return ')'; }
"{"  { ret_print(); return LBRACE; }
"}"  { ret_print(); return RBRACE; }

","  { ret_print(); return ','; }
";"  { ret_print(); return SEMI; }
"="  { ret_print(); return '='; }

"<"  { ret_print(); return '<'; }
">"  { ret_print(); return '>'; }
"<="  { ret_print(); return LEQ; }
">="  { ret_print(); return GEQ; }
"=="  { ret_print(); return EQ; }
"!="  { ret_print(); return NEQ; }

{digit}+ {yylval = atoi(yytext); ret_print(); return NUMBER;}
[a-zA-Z][a-zA-Z0-9]*    { ret_print(); return SYMBOL; }
\"[^\"]*\"  { ret_print(); return MESSAGE; }
[ \t\n]+ {}

%%

void ret_print(){
printf("%s\t\n", yytext);
fprintf(yyout,"%s\t\n", yytext);
}

int yywrap() {
    yylex();
    return 0;
}
```

Αυτός ο κώδικας Flex ορίζει τους κανόνες για την αναγνώριση των λεκτικών μονάδων (είναι case sensitive) σε ένα πρόγραμμα γραμμένο στη GPL. Οι κανόνες περιλαμβάνουν κανονικές εκφράσεις για αναγνώριση αριθμητικών, συμβόλων, λέξεων-κλειδιών, και άλλων στοιχείων της γλώσσας. Κάθε φορά που αναγνωρίζεται ένα λεκτικό, εκτελείται μια συνάρτηση που τυπώνει το λεκτικό και στη συνέχεια το επιστρέφει στον συντακτικό αναλυτή. Επιπλέον, υπάρχει ένα μηχανισμός για αναγνώριση σχολίων που αγνοούνται κατά την ανάλυση. Αυτός ο κώδικας είναι το πρώτο βήμα στη διαδικασία ανάλυσης ενός προγράμματος.

project.y:

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define YYSTYPE double
extern FILE *yyin, *yyout;

void print_token(const char *token, const char *message) {
    printf("%s %s\n\n", token, message);
    fprintf(yyout, "%s %s\n\n", token, message);
}

void yyerror(const char *msg){
    fprintf(stderr, "%s\n", msg);
}

int my_fun();
int yylex();

%}

%token NUMBER SEMI SYMBOL
%token NEQ LEQ GEQ EQ
%token AKER PRAG LEKSH
%token OSO GIA AN ALLIWS
%token KYRIO_MEROS GRAPSE DIABASE EPESTREPSE
%token INC LIB
%token LBRACE RBRACE MESSAGE

%start program
%right '='
%left '+' '-'
%left '*' '/'
%left '<' '>'
%left NEQ LEQ GEQ EQ
%right UMINUS

%%
```

```
program: main
       | library main
       ;

main : KYRIO_MEROS LBRACE stmt_list RBRACE { print_token("kyrio_meros", "Syntax ok");}
     ;

stmt_list : stmt SEMI {print_token("; Semicolon", "Syntax ok"); }
          | stmt_list stmt SEMI {print_token("; Semicolon", "Syntax ok"); }
          | loops
          | stmt_list loops
          ;

library : INC LIB { print_token("libraries", "Syntax ok");}
        ;

condition : expr '<' expr {$$ = $1 < $3; print_token("< Comparison operator", "Syntax ok");}
          | expr '>' expr {$$ = $1 > $3; print_token("> Comparison operator", "Syntax ok");}
          | expr NEQ expr {$$ = $1 != $3; print_token("!= Comparison operator", "Syntax ok");}
          | expr EQ expr {$$ = $1 == $3; print_token("== Comparison operator", "Syntax ok");}
          | expr LEQ expr {$$ = $1 <= $3; print_token("<= Comparison operator", "Syntax ok");}
          | expr GEQ expr {$$ = $1 >= $3; print_token(">= Comparison operator", "Syntax ok");}
          ;

expr : expr '+' expr {$$ = $1 + $3; print_token("+ Arithmetic operator", "Syntax ok");}    Sem
     | expr '-' expr {$$ = $1 - $3; print_token("- Arithmetic operator", "Syntax ok");}
     | expr '*' expr {$$ = $1 * $3; print_token("* Arithmetic operator", "Syntax ok");}
     | expr '/' expr  {$$ = $1 / $3; print_token("/ Arithmetic operator", "Syntax ok");}
     | expr '=' expr {$$ = $3; print_token("= Arithmetic operator", "Syntax ok");}
     | '(' expr ')' {$$ = $2; print_token("() Punctuation", "Syntax ok");}
     | '-' expr %prec UMINUS {$$ = -$2; print_token("UMINUS", "Syntax ok");}
     | NUMBER { $$ = $1; print_token("number", "Syntax ok"); }
     | SYMBOL { $$ = 0; print_token("symbol", "Syntax ok"); }
     ;
```

```
stmt : AKER var_list {print_token("akeraios Data type", "Syntax ok");}
     | PRAG var_list {print_token("pragmatikos Data type", "Syntax ok");}
     | LEKSH var_list { print_token("leksh Data type", "Syntax ok"); }
     | DIABASE '(' var_list ')' { print_token("diabase Keyword", "Syntax ok");}
     | GRAPSE '(' var_list ')' { print_token("grapse Keyword", "Syntax ok");}
     | GRAPSE '(' MESSAGE ')' { print_token("grapse message Keyword", "Syntax ok");}
     | EPESTREPSE expr { print_token("epestrepse Keyword", "Syntax ok");}
     | expr
     ;
```

```
loops: OSO '(' condition ')' LBRACE stmt_list RBRACE {print_token("OSO Loop type", "Syntax ok");}
     | AN '(' condition ')' LBRACE stmt_list RBRACE ALLIWS LBRACE stmt_list RBRACE {print_token("AN Loop type", "Syntax ok");}
     | GIA '(' expr SEMI condition SEMI expr ')' LBRACE stmt_list RBRACE {print_token("GIA Loop type", "Syntax ok");}

var_list : expr
         | var_list ',' expr { print_token(", Comma", "Syntax ok"); }
         ;

%%
```

```c
int my_fun() {
    int c;
    while ((c = getchar()) == ' ');
    if ((c == '.') || (isdigit(c))) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    } else if (isalpha(c)) {
        ungetc(c, stdin);
        scanf("%*c");
        yylval = 0;
        return SYMBOL;
    }
    return c;
}

int main() {
    yyin = fopen("wll1.txt", "r");
    yyout = fopen("wll_analysis.txt", "w");
    yyparse();
    fclose(yyin);
    fclose(yyout);
    return 0;
}
```

Αυτός ο κώδικας Bison ορίζει τους κανόνες για τη συντακτική ανάλυση του προγράμματος γραμμένο στη GPL. Οι κανόνες αυτοί καθορίζουν πώς πρέπει να σχηματιστούν οι δομές του προγράμματος, όπως η κύρια συνάρτηση, οι βιβλιοθήκες, οι δηλώσεις μεταβλητών, οι εντολές εκχώρησης, οι συνθήκες σύγκρισης και οι δομές επανάληψης. Κάθε φορά που αναγνωρίζεται μια δομή, εκτελείται μια συνάρτηση που τυπώνει τον τύπο της δομής και ένα μήνυμα επιβεβαίωσης (print_token()). Επίσης, υπάρχει και μια βοηθητική συνάρτηση my_fun() για την αναγνώριση των διαφορετικών τύπων των λεκτικών μονάδων, αριθμούς (number) και συμβολοσειρές (symbol). Αυτός ο κώδικας είναι υπεύθυνος για την κατανόηση της δομής του προγράμματος και τη δημιουργία μιας αναπαράστασης του στο τερματικό του περιβάλλοντος του VS Code.

Παρακάτω παρατίθενται 1 έγκυρο παράδειγμα για την κατανόηση της ανάλυσης, καθώς και 1 παράδειγμα με λάθη ως προς τη συντακτική δομή των λεκτικών μονάδων:

wll1.txt:

```
# lib
kyrio_meros() {

    grapse("Auto to programma ypologizei ton meso oro dio arithmon.");
    akeraios i;
    grapse("Poses fores theleis na trexeis auto to programma?");


    diabase(i);

    leksh string1;
    (*string1 = "To apotelesma einai";*)

    oso(i>0){

        pragmatikos number1, number2, sum;
        akeraios count;

        grapse("Enter two numbers: ");
        (*diabase(number1, number2);*)
        gia(i=1;i<3;i=i+1){
            diabase(numberi);
        }

        (*calculate the average*)
        sum = number1 + number2;
        count=2;

        grapse(string1);
        grapse(number1, number2, sum/count);
        i=i-1;

    }
    epestrepse 0;

}
```

Αποτέλεσμα:

```
#
lib
libraries Syntax ok

kyrio_meros()
{
grapse
(
"Auto to programma ypologizei ton meso oro dio arithmon."
)
grapse message Keyword Syntax ok

;
; Semicolon Syntax ok

akeraios
i
symbol Syntax ok

;
akeraios Data type Syntax ok

; Semicolon Syntax ok

grapse
(
"Poses fores theleis na trexeis auto to programma?"
)
grapse message Keyword Syntax ok

;
; Semicolon Syntax ok

diabase
(
i
symbol Syntax ok

)
diabase Keyword Syntax ok

;
; Semicolon Syntax ok

leksh
```

1

```
string1
symbol Syntax ok

;
leksh Data type Syntax ok

; Semicolon Syntax ok

oso
(
i
symbol Syntax ok

>
0
number Syntax ok

)
> Comparison operator Syntax ok

{
pragmatikos
number1
symbol Syntax ok

,
number2
symbol Syntax ok

,
, Comma Syntax ok

sum
symbol Syntax ok

;
, Comma Syntax ok

pragmatikos Data type Syntax ok

; Semicolon Syntax ok

akeraios
count
symbol Syntax ok
```

2

```
;
akeraios Data type Syntax ok

; Semicolon Syntax ok

grapse
(
"Enter two numbers: "
)
grapse message Keyword Syntax ok

;
; Semicolon Syntax ok

gia
(
i
symbol Syntax ok

=
1
number Syntax ok

;
= Arithmetic operator Syntax ok

i
symbol Syntax ok

<
3
number Syntax ok

;
< Comparison operator Syntax ok

i
symbol Syntax ok

=
i
symbol Syntax ok

+
1
```

3

```
number Syntax ok

)
+ Arithmetic operator Syntax ok

= Arithmetic operator Syntax ok

{
diabase
(
numberi
symbol Syntax ok

)
diabase Keyword Syntax ok

;
; Semicolon Syntax ok

}
GIA Loop type Syntax ok

sum
symbol Syntax ok

=
number1
symbol Syntax ok

+
number2
symbol Syntax ok

;
+ Arithmetic operator Syntax ok

= Arithmetic operator Syntax ok

; Semicolon Syntax ok

count
symbol Syntax ok

=
2
```

4

```
number Syntax ok

;
= Arithmetic operator Syntax ok

; Semicolon Syntax ok

grapse
(
string1
symbol Syntax ok

)
grapse Keyword Syntax ok

;
; Semicolon Syntax ok

grapse
(
number1
symbol Syntax ok

,
number2
symbol Syntax ok

,
, Comma Syntax ok

sum
symbol Syntax ok

/
count
symbol Syntax ok

/ Arithmetic operator Syntax ok

)
, Comma Syntax ok

grapse Keyword Syntax ok

;
```

5

```
; Semicolon Syntax ok

i
symbol Syntax ok

=
i
symbol Syntax ok

-
1
number Syntax ok

;
- Arithmetic operator Syntax ok

= Arithmetic operator Syntax ok

; Semicolon Syntax ok

}
OSO Loop type Syntax ok

epestrepse
0
number Syntax ok

;
epestrepse Keyword Syntax ok

; Semicolon Syntax ok

}
kyrio_meros Syntax ok
```

6

wll4.txt (λανθασμένο):

```
(*Auto to txt exei lathi etsi oste na fanei i leitourgia tou project*)

# lib
kyrio_meros() {

    grapse("Auto to programma ypologizei ton meso oro dio arithmon.");
    akeraio i; (*edo*)
    grapse("Poses fores theleis na trexeis auto to programma?");


    diavase(i);  (*edo*)

    leksh string1;
    (*string1 = "To apotelesma einai";*)

    oso(i-0){  (*edo*)

        pragmatikos number1, number2, sum;
        akeraios count;

        grapse("Enter two numbers: ");
        (*diabase(number1, number2);*)
        gia(i==1;i<3;i=i+1){                (*edo*)
            diabase(numberi);
        }

        (*calculate the average*)
        sum = number1 + number2;
        count=2;

        grapse(string1);
        grapse(number1, number2, sum/count);
        i=i-1;

    }
    epestrepse 0;

}
```

Αποτέλεσμα:

```
#
lib
libraries Syntax ok

kyrio_meros()
{
grapse
(
"Auto to programma ypologizei ton meso oro dio arithmon."
)
grapse message Keyword Syntax ok

;
; Semicolon Syntax ok

akeraio
symbol Syntax ok

i
syntax error
```

Στον κώδικα υπάρχουν πολλά λάθη, όμως η συντακτική ανάλυση θα ανιχνεύσει το πρώτο σφάλμα που θα δει και θα σταματήσει. Το λάθος εδώ είναι ότι ο κώδικας προσπαθεί να δηλώσει μια μεταβλητή i ως ακέραια, αλλά υπάρχει ορθογραφικό λάθος και το εκλαμβάνει ως σύμβολο, με αποτέλεσμα να μην αναγνωρίζει αυτήν την συντακτική δομή και να σταματάει.

Συμπεράσματα

Συνολικά, η ανάπτυξη του αναλυτή για τη GPL αποτελεί μια επιτυχημένη διαδικασία. Μέσω της χρήσης εργαλείων όπως το Flex και το Bison, καταφέραμε να δημιουργήσουμε ένα εργαλείο που αναγνωρίζει και επεξεργάζεται τον κώδικα στη γλώσσα μας. Με αυτή τη διαδικασία, αποκτήσαμε εμπειρία και κατανόηση σχετικά με την ανάπτυξη γλωσσών προγραμματισμού και τη χρήση αναλυτικών εργαλείων.

Μελλοντικές επεκτάσεις

Στο μέλλον, μπορούμε να εξετάσουμε τη δυνατότητα επέκτασης του αναλυτή για την υποστήριξη περισσότερων χαρακτηριστικών της γλώσσας, όπως η προσθήκη νέων τύπων δεδομένων ή η βελτίωση της ανίχνευσης σφαλμάτων, καθώς ο κώδικας είναι εύκολα επεκτάσιμος. Επιπλέον, μπορούμε να εξετάσουμε τη δυνατότητα ενσωμάτωσης εργαλείων για την ανάλυση σημασιολογίας, προσθέτοντας έτσι δυνατότητες όπως ο έλεγχος τύπων και η βελτιστοποίηση του κώδικα.

Βιβλιογραφία

[1] GitHub Repository
https://github.com/Helen1Z/lexical-and-syntax-analyzer

[2] Compiler Construction using Flex and Bison
https://dlsiis.fi.upm.es/traductores/Software/Flex-Bison.pdf

[3] Flex – Bison Compiler
https://github.com/nooyooj/flex-bison-compiler

[4] Introducing Flex and Bison
https://www.oreilly.com/library/view/flex-bison/9780596805418/ch01.html