ΔΙΕΘΝΕΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΕΛΛΑΔΟΣ

INTERNATIONAL HELLENIC UNIVERSITY
SCHOOL OF ENGINEERING
DEPARTMENT OF INFORMATICS, COMPUTER
AND TELECOMMUNICATIONS ENGINEERING

LEXICAL AND SYNTAX ANALYZER

COMPILERS

Project Team:

Anastasiades Alkinoos (20003)

Zina Eleni (20046)

Supervisor:

Lantzos Theodoros

SERRES, 2024

Contents

Introduction

The lexical and syntax parser are basic tools for developing compilers and programming languages. Flex is used to create lexical parsers that recognize text patterns and convert them into tokens, while Bison is used to create parsers that organize these tokens according to syntactic rules of the language. Through their combined use, we can create programs that understand and process complex language structures, allowing the development of fully functional compilers.

Purpose of the project

The purpose of our project was to create a simple lexical and syntax analyzer for a new programming language, the Greek Programming Language (GPL). GPL was designed with keywords and syntax based on the Greek language, offering a more accessible and Greek-speaking programming experience. Through the use of the Flex and Bison tools, we developed an analyzer that recognizes GPL patterns and syntactic structures, allowing developers to write and execute programs in this new language.

Methodology

Our methodology for developing the GPL involved using regular expressions (regex) in Flex to identify and extract tokens from code written in our language. These tokens were passed on to the syntactic analysis implemented with Bison, where we defined the correct syntax of GPL structures. Through this process, we have ensured that every program written in the GPL follows the predefined syntax rules, allowing valid programs to be recognized and run in our new language. The following table shows the basic structures we used in the GPL:

Structure	**Description**	**Example**
# lib	Library Declaration	(as it is)
kyrio_meros() {…}	Main function	kyrio_meros() {…}
grapse("…");	Prints values or messages	grapse("Enter a number:"); or grapse(a,b,c);
diabase("…");	Reads values	diabase(a,b,c);
epestrepse …;	Returns values	epestrepse 0; or epestrepse x;
(*…*)	Comments	(*This is a comment*)
akeraios	Data type	akeraios x1,x2;
pragmatikos	Data type	pragmatikos y;
leksh	Data type	leksh name;
an (…) {…} alliws {…}	Loop type	an (x > 0) { grapse("Positive"); } alliws { grapse("Negative"); }
oso (…) {…}	Loop type	oso (x < 10) { x = x + 1; }
gia (…; …; …) {…}	Loop type	gia (i = 0; i < 10; i = i + 1) { grapse(i); }
+,-,*,/,=	Arithmetic operators	a = b + c;
<,>,==,<=,>=,!=	Comparative operators	an (a != b) { grapse("Different"); }
(,),{,},(comma),(question mark)	Punctuation	a = (b + c) * d;

Implementation

During the execution process, the lexical analyzer reads the input program and recognizes the various lexemes, while the syntax analyzer/parser uses the tokens returned to identify the syntax structure of the program. The code in which the analysis will be performed is read from a text file and the result of the analysis is printed in the terminal. If everything goes well, the program completes successfully, otherwise syntax errors appear that need to be fixed. The steps to run the lexical and syntax analyzers are as follows:

1. **flex project.l**: This command uses Flex (Fast Lexical Analyzer Generator) to create a lexical analyzer from a description file, **project.l**. This file contains rules that describe how to identify and return the programming language tokens that the program processes.
2. **bison -d project.y**: This command uses Bison to create a parser from the **project.y** grammar description file. This file contains the rules that describe the syntax of the programming language.
3. **gcc -o a.out project.tab.c lex.yy.c**: This command uses the GCC interpreter to compile the source files produced by Flex and Bison, namely **project.tab.c** and **lex.yy.c**, and create an executable file called **a.out**.
4. **./a.out**: This command runs the program produced by GCC, the **a.out** executable file.

The whole code, the steps above, and many valid execution examples can be found on the GitHub website where we have our own 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;
}
```

This Flex Code defines the rules for identifying tokens (case sensitive) in a program written in the GPL. Rules include regular expressions to identify numeric, symbolic, keywords, and other elements of the language. Each time a lexeme is recognized, a function is executed that prints the lexeme and then returns it to the parser. In addition, there is a mechanism for identifying comments that are ignored during parsing. This code is the first step in the process of a program analysis.

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"); }
         ;

%%
```

```
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;
}
```

This Bison code defines the rules for syntax analysis of the program written in the GPL. These rules define how program structures such as the main function, libraries, variable declarations, assignment commands, comparison conditions, and iteration structures should be formed. Each time a form of structure is recognized, a function is executed that prints the type of structure and a confirmation message (print_token()). There is also an auxiliary function my_fun() to identify different types of tokens, numbers and symbols. This code is responsible for understanding the structure of the program and creating a representation of it in the terminal of the VS Code environment.

Below are 1 valid example for understanding analysis, as well as 1 example with errors in the syntax structure of tokens:

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;

}
```

Result:

```
#
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 (incorrect):

```
(*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;

}
```

Result:

```
#
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
```

There are many errors in the code, but the syntax analysis will detect the first error it sees and stop. The error here is that the code tries to declare an i variable as an integer, but there is a spelling error and it takes it as a symbol, so it does not recognize this syntax structure and stops.

Conclusion

Overall, developing the analyzer for the GPL is a successful process. Through the use of tools like Flex and Bison, we were able to create a tool that recognizes and processes code in our language. With this process, we gained experience and understanding about developing programming languages and using analytical tools.

Future expansions

In the future, we may consider extending the parser to support more features of the language, such as adding new data types or improving error detection, as the code is easily extensible. In addition, we can consider the possibility of integrating tools for semantic analysis, thus adding features such as type checking and code optimization.

Bibliography

[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