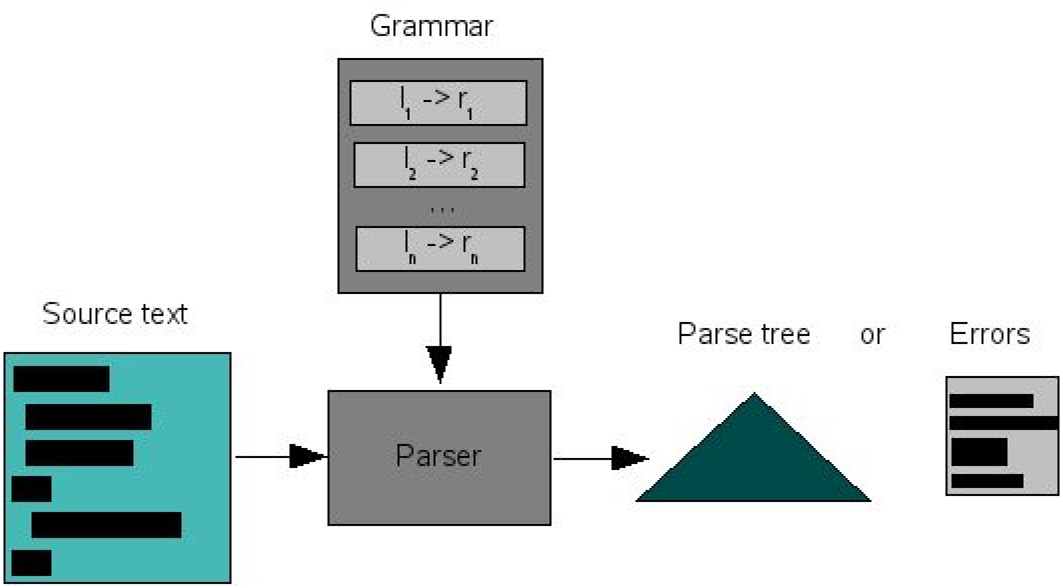
Compiler Design Lab

Parser

Phase #2



Submitted to : Ms. Meghana Pujar

Submitted by : Abhijith Anilkumar (13CO102)

Chirag Jamadagni (13CO117)

George CM (13CO119)

Table of Contents

List of Tables

List of Figures

Abstract

Compiler design for a language involves two phases - the analysis phase, and the synthesis phase. The aim of the second phase of the course project of building a compiler is to build a parser, using a tool called Yacc. The lexical analyzer generated in the first phase reads the source program and generates tokens that are given as input to the parser which then creates a syntax tree in accordance with the grammar, consequently leading to the generation of intermediate code that is fed into the synthesis phase, to obtain the correct, equivalent machine level code. Various test cases are also demonstrated, along with code snippets and output snapshots, in order to demonstrate the functioning of the parser.

Introduction

Syntactical Analyzer

Also known as the parser. Syntactic analysis is the second phase of a compiler. Parsing or syntactic analysis is the process of analysing a string of symbols, in computer languages, conforming to the rules of a formal grammar. It takes the tokens generated by the lexical analyser and a context free grammar and gives a parse tree corresponding to the grammar, with the tokens inputted, as the output. It basically gives a structural representation of the input, checking for correct syntax in the process.

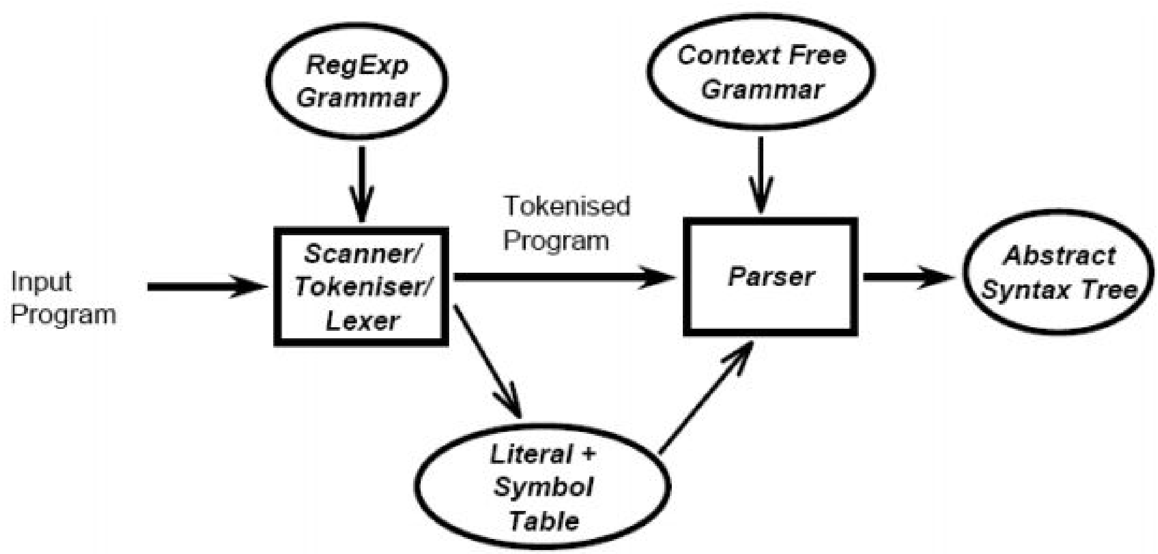


Fig 1: Flowchart of a parser

Yacc Script

Yacc(Yet Another Compiler-Compiler) provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine. The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions. Yacc is written in a portable dialect of C[1] and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

Yacc requires token names to be declared as such. In addition, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections:

declarations
%%
rules
%%
programs

The sections are separated by double percent "%%" marks. The percent "%" symbol is generally used in Yacc specifications as an escape character.

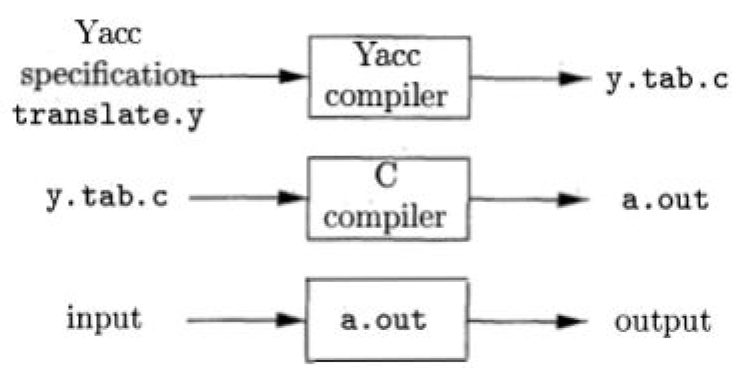


Fig 2: Creating a syntactical analyzer with Yacc

Design

Code

Lex can be used in two ways. The first one is where it is used for simple transformations, or for analysis and statistics gathering on a lexical level. It can also be used with a parser generator to perform the lexical and syntactical analysis phase, because it is particularly easy to interface Lex and Yacc. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but it requires a low level analyzer to recognize input tokens. When the tasks are divided into lexical and syntactical phases, Yacc is used to assign structure.

The rules section contains multiple grammar rules that specify the syntactical constructs of the source language. We have associated different actions to different rules, like returning values or obtaining values returned from previous actions.

In order to get the tokens from the lexical analyzer, the integer valued function yylex() must be specified. This function returns an integer which is the token number representing the kind of token that is read. If this token is associated with a value, it should be assigned to the external variable yylval.

The following are the key points in order to link the Lex file with the Yacc file:

- The token names in the Lex file should match with those of the Yacc file.
- In order that the Yacc file receives each of the tokens from the Lex file, we must also see to it that every token generated by the lexical analyzer is returned to the Yacc file, for it to check its rules and parse.
- The main function of the Lex file must also be modified, and a yyparse() function must be called from it, in order to return all the generated tokens to the Yacc file.

The following are the cases that arise while running the program:

- If the source program is free of syntactical errors, then the tokens generated by the lexical analyzer are generated and displayed.
    - In this case, along with the generated tokens, a table called the symbol table, is also generated, updated, and displayed which contains the token type and the token name.
    - Along with this, a table called the constant table, containing the various constant variables is also displayed.

- On the other hand, if the source program does contain syntax errors, then the tokens are properly generated up to the position of the error, after which an error message, generated by yyerror() appears, stating the line number at which the error has occurred.
    - The symbol table and constant table are not displayed in this case.

- *Errors:* For displaying the line numbers for the errors, we used a global variable lineno initialized to 1 and which keeps incrementing whenever a newline is scanned.

- *Symbol table and Constant table:* We have created a header file, “symbol.h” where there are functions to insert the lexemes into the symbol tables. When we see the <EOF> we print both the tables.

- *Shift-reduce conflicts:* There is only one shift-reduce conflict present in our grammar and no reduce-reduce conflicts at all. This conflict is due to the classic "dangling else" problem. The problem can be fixed by assigning precedence to the rules that causes conflict. The rule causing conflict is:

```
selection_statement : IF '(' expression ')' statement
                    | IF '(' expression ')' statement ELSE
statement
```

In order to fix this, we start by making ELSE and LOWER_THAN_ELSE (a pseudo-token) non associative:

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE
```

This gives ELSE more precedence over LOWER_THAN_ELSE simply because LOWER_THAN_ELSE is declared first.

Then in the conflicting rule you have to assign a precedence to either the shift or reduce action:

```
selection_stmt : IF '(' expression ')' statement %prec
LOWER_THAN_ELSE ; | IF '(' expression ')' statement ELSE statement ;
```

Here, higher precedence is given to shifting.

Parser.y

This is the main parser code. This is a yacc file which contains the declarations, rules and programs and defines the actions which should be taken for various cases.

```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "symbol.h"
#include "y.tab.h"
int errorFlag=0;
extern FILE *yyin;
extern FILE *yyout;
extern int line,cnt;
extern symrec *sym_table;
extern symrec *const_table;
extern char *tempid;
%}
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%start translation_unit
%%

primary_expression
    : IDENTIFIER  { putsym(tempid, 'v', line); }
    | CONSTANT    { putsym(tempid, 'c', line);}
    | STRING_LITERAL  { putsym(tempid, 'c', line);}
    | '(' expression ')'
                                                              1,120         Top
```

Figure 3

```
    ;

postfix_expression
    : primary_expression
    | postfix_expression '[' expression ']'
    | postfix_expression '(' ')'
    | postfix_expression '(' argument_expression_list ')'
    | postfix_expression '.' IDENTIFIER { putsym(tempid, 'v', line);}
    | postfix_expression PTR_OP IDENTIFIER { putsym(tempid, 'v', line);}
    | postfix_expression INC_OP  { putsym(tempid, 'o', line);}
    | postfix_expression DEC_OP  { putsym(tempid, 'o', line);}
    ;

argument_expression_list
    : assignment_expression
    | argument_expression_list ',' assignment_expression { putsym(",",'p', line); }
    ;

unary_expression
    : postfix_expression
    | INC_OP unary_expression { putsym("++",'o', line); }
    | DEC_OP unary_expression { putsym("--",'o', line); }
    | unary_operator cast_expression
    | SIZEOF unary_expression { putsym("sizeof",'o', line); }
    | SIZEOF '(' type_name ')' { putsym("sizeof",'o', line); }
    ;

unary_operator
    : '&' { putsym("&",'o', line); }
    | '*' { putsym("*",'o', line); }
    | '+' { putsym("+",'o', line); }
    | '-' { putsym("-",'o', line); }
    | '~' { putsym("~",'o', line); }
    | '!' { putsym("!",'o', line); }
    ;

cast_expression
                                                                39,104        8%
```

Figure 4

```
    : unary_expression
    | '(' type_name ')' cast_expression
    ;

multiplicative_expression
    : cast_expression
    | multiplicative_expression '*' cast_expression { putsym("*",'o', line); }
    | multiplicative_expression '/' cast_expression { putsym("/",'o', line); }
    | multiplicative_expression '%' cast_expression { putsym("%",'o', line); }
    ;

additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression { putsym("+",'o', line); }
    | additive_expression '-' multiplicative_expression { putsym("-",'o', line); }
    ;

shift_expression
    : additive_expression
    | shift_expression LEFT_OP additive_expression { putsym("<<",'o', line); }
    | shift_expression RIGHT_OP additive_expression { putsym(">>",'o', line); }
    ;

relational_expression
    : shift_expression
    | relational_expression '<' shift_expression
    | relational_expression '>' shift_expression
    | relational_expression LE_OP shift_expression { putsym("<=",'o', line); }
    | relational_expression GE_OP shift_expression { putsym(">=",'o', line); }
    ;

equality_expression
    : relational_expression
    | equality_expression EQ_OP relational_expression { putsym("==",'o', line); }
    | equality_expression NE_OP relational_expression { putsym("!=",'o', line); }
    ;

                                                                111,116       17%
```

Figure 5

```
and_expression
    : equality_expression
    | and_expression '&' equality_expression { putsym("&", 'o', line);}
    ;

exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression { putsym("^", 'o', line); }
    ;

inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression { putsym("|", 'o', line); }
    ;

logical_and_expression
    : inclusive_or_expression
    | logical_and_expression AND_OP inclusive_or_expression { putsym("&&", 'o', line); }
    ;

logical_or_expression
    : logical_and_expression
    | logical_or_expression OR_OP logical_and_expression { putsym("||", 'o', line); }
    ;

conditional_expression
    : logical_or_expression
    | logical_or_expression '?' expression ':' conditional_expression { putsym("?:",'o', line); }
    ;

assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression
    ;

assignment_operator
    : '=' { putsym("=",'o', line); }
                                                                        148,113-116    25%
```

Figure 6

```
    | MUL_ASSIGN { putsym("*=",'o', line); }
    | DIV_ASSIGN { putsym("/=",'o', line); }
    | MOD_ASSIGN { putsym("%=",'o', line); }
    | ADD_ASSIGN { putsym("+=",'o', line); }
    | SUB_ASSIGN { putsym("-=",'o', line); }
    | LEFT_ASSIGN { putsym("<<=",'o', line); }
    | RIGHT_ASSIGN { putsym(">==",'o', line); }
    | AND_ASSIGN { putsym("&=",'o', line); }
    | XOR_ASSIGN { putsym("^=",'o', line); }
    | OR_ASSIGN { putsym("|=",'o', line); }
    ;

expression
    : assignment_expression
    | expression ',' assignment_expression { putsym(",", 'p', line); }
    ;

constant_expression
    : conditional_expression
    ;

declaration
    : declaration_specifiers ';' { putsym(";", 'p', line); }
    | declaration_specifiers init_declarator_list ';' { putsym(";", 'p', line); }
    ;

declaration_specifiers
    : storage_class_specifier
    | storage_class_specifier declaration_specifiers
    | type_specifier
    | type_specifier declaration_specifiers
    | type_qualifier
    | type_qualifier declaration_specifiers
    ;

init_declarator_list
    : init_declarator
                                                                        185,113-116    34%
```

Figure 7

```
    | init_declarator_list ',' init_declarator { putsym(",", 'p', line); }
    ;

init_declarator
    : declarator
    | declarator '=' initializer { putsym("=", 'o', line); }
    ;

storage_class_specifier
    : TYPEDEF { putsym("typedef", 'k', line);}
    | EXTERN { putsym("extern", 'k', line);}
    | STATIC { putsym("static", 'k', line);}
    | AUTO { putsym("auto", 'k', line);}
    | REGISTER { putsym("register", 'k', line);}
    ;

type_specifier
    : VOID { putsym("void", 'k', line);}
    | CHAR { putsym("char", 'k', line);}
    | SHORT { putsym("short", 'k', line);}
    | INT { putsym("int", 'k', line);}
    | LONG { putsym("long", 'k', line);}
    | FLOAT { putsym("float", 'k', line);}
    | DOUBLE { putsym("double", 'k', line);}
    | SIGNED { putsym("signed", 'k', line);}
    | UNSIGNED { putsym("unsigned", 'k', line);}
    | struct_or_union_specifier
    | enum_specifier
    | TYPE_NAME
    ;

struct_or_union_specifier
    : struct_or_union IDENTIFIER '{' struct_declaration_list '}'
    | struct_or_union '{' struct_declaration_list '}'
    | struct_or_union IDENTIFIER
    ;

                                                                    222,124       42%
```

Figure 8

```
struct_or_union
    : STRUCT { putsym("struct", 'k', line);}
    | UNION { putsym("union", 'k', line);}
    ;

struct_declaration_list
    : struct_declaration
    | struct_declaration_list struct_declaration
    ;

struct_declaration
    : specifier_qualifier_list struct_declarator_list ';' { putsym(";", 'p', line); }
    ;

specifier_qualifier_list
    : type_specifier specifier_qualifier_list
    | type_specifier
    | type_qualifier specifier_qualifier_list
    | type_qualifier
    ;

struct_declarator_list
    : struct_declarator
    | struct_declarator_list ',' struct_declarator { putsym(",", 'p', line); }
    ;

struct_declarator
    : declarator
    | ':' constant_expression { putsym(":", 'p', line); }
    | declarator ':' constant_expression { putsym(":", 'p', line); }
    ;

enum_specifier
    : ENUM '{' enumerator_list '}' { putsym("enum", 'k', line);}
    | ENUM IDENTIFIER '{' enumerator_list '}' { putsym("enum", 'k', line); putsym(tempid, 'v', line); }
    | ENUM IDENTIFIER { putsym("enum", 'k', line); putsym(tempid, 'v', line); }
    ;
                                                                    259,113-116   51%
```

Figure 9

```
enumerator_list
    : enumerator
    | enumerator_list ',' enumerator { putsym(",", 'p', line); }
    ;

enumerator
    : IDENTIFIER { putsym(tempid, 'v', line); }
    | IDENTIFIER '=' constant_expression { putsym("=", 'o', line); putsym("tempid", 'v', line); }
    ;

type_qualifier
    : CONST { putsym("const", 'k', line); }
    | VOLATILE { putsym("volatile", 'k', line); }
    ;

declarator
    : pointer direct_declarator
    | direct_declarator
    ;

direct_declarator
    : IDENTIFIER { putsym(tempid, 'v', line); }
    | '(' declarator ')'
    | direct_declarator '[' constant_expression ']'
    | direct_declarator '[' ']'
    | direct_declarator '(' parameter_type_list ')'
    | direct_declarator '(' identifier_list ')'
    | direct_declarator '(' ')'
    ;

pointer
    : '*' { putsym("*", 'o', line); }
    | '*' type_qualifier_list { putsym("*", 'o', line); }
    | '*' pointer { putsym("*", 'o', line); }
    | '*' type_qualifier_list pointer { putsym("*", 'o', line); }
    ;
                                                              296,125-128    59%
```

Figure 10

```
type_qualifier_list
    : type_qualifier
    | type_qualifier_list type_qualifier
    ;


parameter_type_list
    : parameter_list
    | parameter_list ',' ELLIPSIS { putsym(",", 'p', line); putsym("::", 'o', line); }
    ;

parameter_list
    : parameter_declaration
    | parameter_list ',' parameter_declaration { putsym(",", 'p', line); }
    ;

parameter_declaration
    : declaration_specifiers declarator
    | declaration_specifiers abstract_declarator
    | declaration_specifiers
    ;

identifier_list
    : IDENTIFIER {putsym(tempid, 'v', line);}
    | identifier_list ',' IDENTIFIER { putsym(tempid, 'v', line); putsym(",", 'p', line); }
    ;

type_name
    : specifier_qualifier_list
    | specifier_qualifier_list abstract_declarator
    ;

abstract_declarator
    : pointer
    | direct_abstract_declarator
    | pointer direct_abstract_declarator
                                                              333,121-124    68%
```

Figure 11

```
    ;

direct_abstract_declarator
    : '(' abstract_declarator ')'
    | '[' ']'
    | '[' constant_expression ']'
    | direct_abstract_declarator '[' ']'
    | direct_abstract_declarator '[' constant_expression ']'
    | '(' ')'
    | '(' parameter_type_list ')'
    | direct_abstract_declarator '(' ')'
    | direct_abstract_declarator '(' parameter_type_list ')'
    ;

initializer
    : assignment_expression
    | '{' initializer_list '}'
    | '{' initializer_list ',' '}'
    ;

initializer_list
    : initializer
    | initializer_list ',' initializer { putsym(",", 'p', line); }
    ;

statement
    : labeled_statement
    | compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    ;

labeled_statement
    : IDENTIFIER ':' statement  { putsym(tempid, 'v', line);  }
    | CASE constant_expression ':'  statement { putsym(":", 'p', line); putsym("case", 'k', line);}
                                                                                   370,117-120    76%
```

Figure 12

```
    | DEFAULT ':' statement { putsym(":", 'p', line); putsym("default", 'k', line); }
    ;

compound_statement
    : '{' '}'
    | '{' statement_list '}'
    | '{' declaration_list '}'
    | '{' declaration_list statement_list '}'
    ;

declaration_list
    : declaration
    | declaration_list declaration
    ;

statement_list
    : statement
    | statement_list statement
    ;

expression_statement
    : ';' { putsym(";", 'p', line); }
    | expression ';' { putsym(";", 'p', line); }
    ;

selection_statement
    : IF '(' expression ')' statement    %prec LOWER_THAN_ELSE { putsym("if", 'k', line); }
  | IF '(' expression ')' statement ELSE statement { putsym("if", 'k', line);  putsym("else", 'k', line);}
    | SWITCH '(' expression ')' statement
    ;

iteration_statement
    : WHILE '(' expression ')' statement  { putsym("while", 'k', line);}
    | DO statement WHILE '(' expression ')' ';'
    | FOR '(' expression_statement expression_statement ')' statement  { putsym("for", 'k', line);}
    | FOR '(' expression_statement expression_statement expression ')' statement { putsym("for", 'k', line); }
    ;
                                                                                   407,117-120    85%
```

Figure 13

```
jump_statement
    : GOTO IDENTIFIER ';' { putsym("goto", 'k', line); putsym(";", 'p', line); putsym(tempid, 'v', line);}
    | CONTINUE ';' { putsym("continue", 'k', line); putsym(";", 'p', line); }
    | BREAK ';'  { putsym("break", 'k', line); putsym(";", 'p', line);}
    | RETURN ';'  { putsym("return", 'k', line); putsym(";", 'p', line);}
    | RETURN expression ';'{ putsym("return", 'k', line); putsym(";", 'p', line);}
    ;

translation_unit
    : external_declaration
    | translation_unit external_declaration
    ;

external_declaration
    : function_definition
    | declaration
    ;

function_definition
    : declaration_specifiers declarator declaration_list compound_statement
    | declaration_specifiers declarator compound_statement
    | declarator declaration_list compound_statement
    | declarator compound_statement
    ;

%%
#include <stdio.h>
extern char yytext[];
extern int column;
yyerror(s)
char *s;
{
    errorFlag=1;
    fflush(stdout);
    printf("\nSyntax error at line: %d and pos: %d\n", line,cnt);
}
                                                                                                    444,128        93%
```

Figure 14

```
extern int column;
yyerror(s)
char *s;
{
    errorFlag=1;
    fflush(stdout);
    printf("\nSyntax error at line: %d and pos: %d\n", line,cnt);
}
main(argc,argv)
char **argv;
int argc;
{
    if(argc<=1)
    {
        printf("Arguments missing ! correct format : ./a.out filename \n");
        return 0;
    }
    yyin=fopen(argv[1],"r");
    yyout=fopen("out.c","w");
    yyparse();

    if(!errorFlag)
  {
        printf("Compilation Successful!\n");
    FILE *sym_tab=fopen("symbol_table.txt","w");
    fprintf(sym_tab,"\n    Symbol Table \n================== \n \t\t\t\t\t\t\t\tName\t\t\t\t\t\t\t\tToken Class\t\t\t\t\t\t\t\t\t\t\
t\t\t\t\t\t\t\t\t\t\t\t\t\t\tLine Number\n");
    symrec *ptr;
    for(ptr=sym_table;ptr!=(symrec *)0;ptr=(symrec *)ptr->next)
        fprintf(sym_tab,"\n%20s%30.30s%60s",ptr->name,ptr->type,ptr->line);
    fprintf(sym_tab,"\n\n\n\n    Constant Table \n================== \n \t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\tValue\t\t\t\t\t\t\t\t\
t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\tLine Number\n");
    for(ptr=const_table;ptr!=(symrec *)0;ptr=(symrec *)ptr->next)
        fprintf(sym_tab,"\n%50s%60s",ptr->name,ptr->line);
    fclose(sym_tab);
    }
}
                                                                                                    471,120        Bot
```

Figure 15

Symbol.h

Contains functions to input a token into the symbol or constant table and output in on the standard output. Basically, this file contains the hash implementation of the symbol and constant table.

```c
#include <stdlib.h>
#define t_void  1
#define t_char  2
#define t_int   3
#define t_float 4
struct symrec
{
    char *name,type[20],line[100];
    struct symrec *next;
};
typedef struct symrec symrec;
symrec *sym_table = (symrec *)0;
symrec *const_table = (symrec *)0;

void *putsym(char *sym_name,char sym_type, int linec)
{
    symrec *ptr,*newptr;
    char line[39],linen[19];
    snprintf(linen, 19, "%d", linec);
    strcpy(line," ");
    strcat(line,linen);
    char type[20];
    switch(sym_type)
    {
            case 'c':
                    strcpy(type,"Constant");
                    break;
            case 'v':
                    strcpy(type,"Identifier");
                    break;
            case 'p':
                    strcpy(type,"Punctuator");
                    break;
            case 'o':
                    strcpy(type,"Operator");
                    break;
            case 'k':
                                                                                                                                  6,128          Top
```

Figure 16

```
                    strcpy(type,"Keyword");
                    break;
            case 's':
                    strcpy(type,"String Literal");
                    break;
            case 'd':
                    strcpy(type,"Preprocessor Statement");
                    break;
    }
    if(sym_type == 'c')
    {
    for(newptr=const_table;newptr!=(symrec *)0;newptr=(symrec *)newptr->next)
     if(strcmp(newptr->name,sym_name)==0)
     {
       strcat(newptr->line,line);
       return;
     }
        ptr=(symrec *)malloc(sizeof(symrec));
        ptr->name=(char *)malloc(strlen(sym_name)+1);
        strcpy(ptr->name,sym_name);
        strcpy(ptr->type,type);
    strcpy(ptr->line,line);
        ptr->next=(struct symrec *)const_table;
        const_table=ptr;
    }
    else
    {
    for(newptr=sym_table;newptr!=(symrec *)0;newptr=(symrec *)newptr->next)
     if(strcmp(newptr->name,sym_name)==0)
     {
       strcat(newptr->line,line);
       return;
     }
        ptr=(symrec *)malloc(sizeof(symrec));
        ptr->name=(char *)malloc(strlen(sym_name)+1);
        strcpy(ptr->name,sym_name);
        strcpy(ptr->type,type);
                                                                          74,110-116    74%
```

Figure 17

```
     {
       strcat(newptr->line,line);
       return;
     }
        ptr=(symrec *)malloc(sizeof(symrec));
        ptr->name=(char *)malloc(strlen(sym_name)+1);
        strcpy(ptr->name,sym_name);
        strcpy(ptr->type,type);
    strcpy(ptr->line,line);
        ptr->next=(struct symrec *)const_table;
        const_table=ptr;
    }
    else
    {
    for(newptr=sym_table;newptr!=(symrec *)0;newptr=(symrec *)newptr->next)
     if(strcmp(newptr->name,sym_name)==0)
     {
       strcat(newptr->line,line);
       return;
     }
        ptr=(symrec *)malloc(sizeof(symrec));
        ptr->name=(char *)malloc(strlen(sym_name)+1);
        strcpy(ptr->name,sym_name);
        strcpy(ptr->type,type);
    strcpy(ptr->line,line);
        ptr->next=(struct symrec *)sym_table;
        sym_table=ptr;
    }
}
void *getsym(char *sym_name)
{
    symrec *ptr;
    for(ptr=sym_table;ptr!=(symrec *)0;ptr=(symrec *)ptr->next)
    if(strcmp(ptr->name,sym_name)==0)
    return ptr;
    return 0;
}
                                                                          87,112        Bot
```

Figure 18

Run

This is a shell script which automates the compilation and execution of the code.

7 lines (6 sloc) | 117 Bytes

```
#!/bin/sh
yacc -d parser.y
lex scanner.l
gcc lex.yy.c y.tab.c -w -g
./a.out $1
rm -rf lex.yy.c y.tab.c y.tab.h a.out
```

Figure 19

Test Cases

Serial No.	Test Case	Expected Output	Status

1	#include<stdio.h>
	void main()
	{
	int a,,b;
	}
		line no.4:syntax error	Working

2	#include<stdio.h>
	void main()
	{
	int a,c,b
	c=a+b;
	}
		line no.4:syntax error	Working

3	#include<stdio.h>
	void main()
	{
	int a,b,c;
	
	if(a>b);
	a = b-c;
	else
	a = b+c;
	}
		line no.8:syntax error	Working

4	#include<stdio.h>
	void main()
	{
	int a,b,c;
	
	for(i=0;i++)
	a = a+b;
	}
		line no.6:syntax error	Working

5	`#include<stdio.h>`
	`void main()`
	`{`
	`   int a,b,c;`
	
	`   for(i=0;i<b;i++)`
	`   {`
	`     for(j=0;j<c;j--)`
	`     {`
	`       a=a+b;`
	`     }`
	`   }`
		line no.12:syntax error	Working

6	`#include<stdio.h>`
	`void main()`
	`{`
	` int a,b,c;`
	
	` for(i=0;i<b;i++)`
	` {`
	` a=b*;`
	` }`
	`}`
		line no.8:syntax error	Working

7	`#include<stdio.h>`
	`void main()`
	`{`
	` int a,b,c[10; //syntax error`
	`}`
		ine no.4:syntax error	Working

8	<pre>#include<stdio.h>
	void main()
	{
	      int a[10],i,j;
	
	      while(i<10)
	      {
	            scanf("%d",&a[i]);
	      }
	
	      for(i=0;i<10;i++)
	      {
	            for(j=0;j<9-i;j++)
	            {
	              if(a[j]>a[j+1])
	              {
	
	swap(&a[j],&a[j+1]);
	               }
	             }
	      }
	}
	
	void swap(int *a,int *b)
	{
	      *a=*a+*b;
	      *b=*a-*b;
	      *a=*a-*b;
	}</pre>
		Prints the symbol and constants table	Working

Implementation

The implementation is as follows:

- Firstly, a file is created with a "*.y*" extension, which represents a yacc file. The file created here is "*parser.y*".
- The file, like all yacc files, contains three main sections: Declarations, Rules, and Programs.
- All the definitions used throughout the program are defined under this section, including the header file that is created for the implementation of the symbol table, i.e., "*symbol.h*".
- The Rules section contains the various grammatical rules that are to be followed while parsing.
- The Programs section is where the *yyparse()* and the *yyerror()* functions are called, in order to generate the error message at the line where the syntax error has occurred.
- The header file "*symbol.h*" contains functions to insert a token into the symbol table and also to output it on stdout. "*symbol.h*" essentially consists of the hash implementation of the symbol and constant tables.
- Once the file is made, it is run using the shell script by giving the following command: *./run*

Extensive Test Cases

Case 1 : Positive test case with no errors

Input

```c
#include<stdio.h>

void main()
{
    int a[10],i,j;

    while(i<10)
    {
        scanf("%d",&a[i]);
    }


    for(i=0;i<10;i++)
    {
        for(j=0;j<9-i;j++)
        {
            if(a[j]>a[j+1])
            {
                swap(&a[j],&a[j+1]);
            }
        }
    }
}

void swap(int *a,int *b)
{
    *a=*a+*b;
    *b=*a-*b;
    *a=*a-*b;
}
```

Figure 20

Output

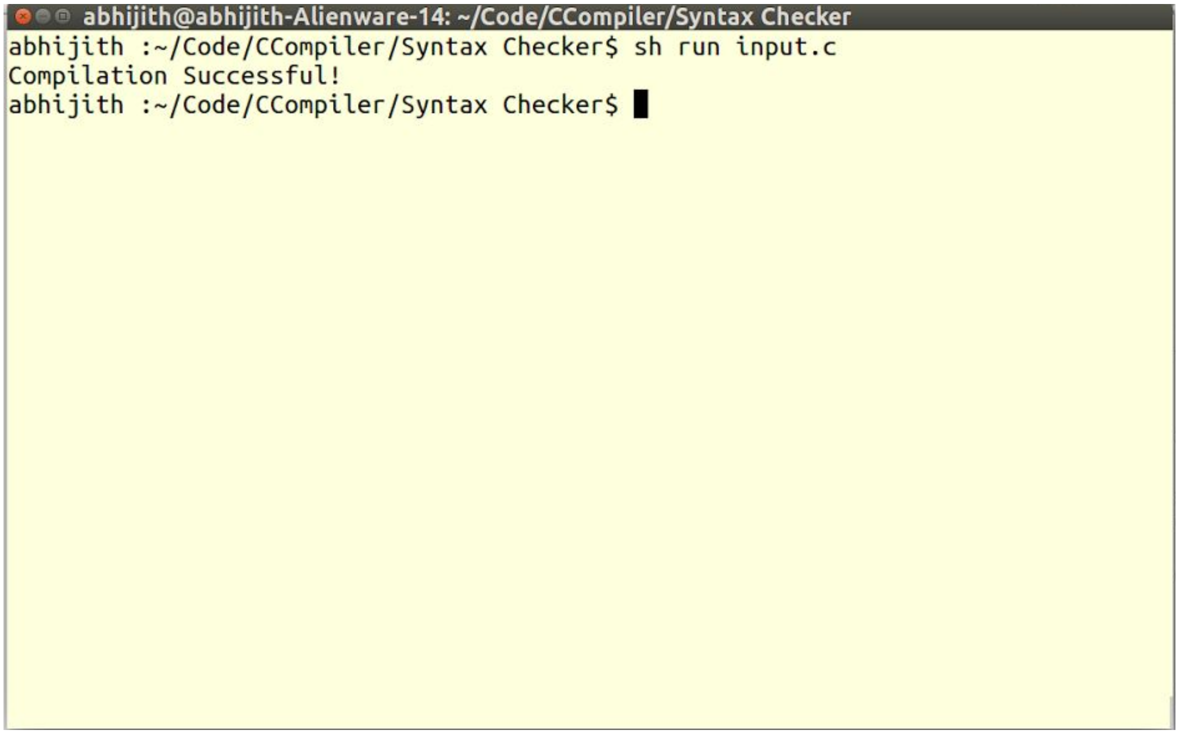


Figure 21

Symbol Table
===================

Name	Token Class	Line Number
}	Punctuator	22
return	Keyword	21
if	Keyword	21
/	Operator	17
*	Operator	17 17
=	Operator	17
&	Operator	8 10 16
scanf	Identifier	8 10 16
printf	Identifier	7 9 15 18 20
;	Punctuator	6 7 8 9 10 15 16 17 18 20 21
]	Punctuator	6
[	Punctuator	6
num	Identifier	6
ee	Identifier	6
si	Identifier	6 17 18
time	Identifier	6 16 17
,	Punctuator	6 6 6 6 6 8 10 16 18
rate	Identifier	6 10 17
amount	Identifier	6 8 17
{	Punctuator	5
)	Punctuator	5 7 8 9 10 15 16 17 18 20 21 21
(	Punctuator	5 7 8 9 10 15 16 17 18 20 21 21
main	Identifier	5
int	Keyword	5 6

Figure 22

Constant Table
==================

Value	Line Number
0	21
"Hello"	20
1	19
"\nSimple Interest : %d"	18
"\nEnter Period of Time : "	15
"\nEnter Rate of Interest : "	9
"%d"	8 10 16
"\nEnter Principal Amount : "	7
100	6 17

Figure 23

Case 2 : Negative test case with syntax errors

Input

```
#include<stdio.h>
/* This is a multi
line comment*/
int main() {
    int amount, rate, time, si, 56ee;
    //This is a single line comment
    printf("\nEnter Principal Amount : ");
    scanf("%d", &amount);

    printf("\nEnter Rate of Interest : ");
    scanf("%d", &rate);

    printf("\nEnter Period of Time   : ");
    scanf("%d", &time);
    scanf("",);
    si = (amount * rate * time / 100;
    printf("\nSimple Interest : %d", si);
    printf("",);
    return(0);
}
}
"I am a
```

Figure 24

Output

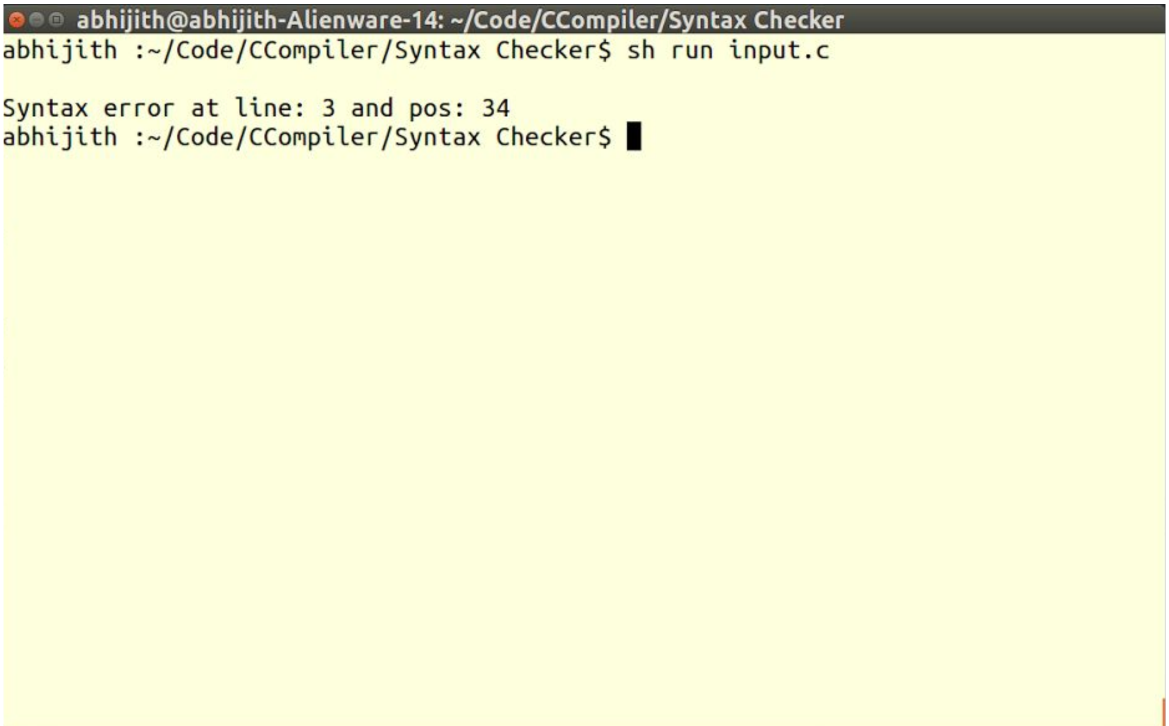


Figure 25

Results/ Future Work

The outcome of the second phase of this project is a syntactical analyzer that takes in the tokens generated by the lexical analyzer built in the first phase as input, and displays syntax error on the corresponding line number, and also symbol table and constant table.

Our next aim would be to implement a add a semantic checker to the existing compiler, and build the symbol table to be used during checking and code generation, in order to work our way towards making a full-fledged compiler.

References

1. http://www.tldp.org/HOWTO/Lex-YACC-HOWTO-6.html
2. http://easywaysnow.blogspot.in/2012/10/running-lex-and-yacc-program-inubuntu.html?m=1
3. https://www.lysator.liu.se/c/ANSI-C-grammar-y.html
4. http://dinosaur.compilertools.net/yacc/