Semantic Analyzer for the C Language

National Institute of Technology Karnataka,Surathkal

Date: 14th Feb 2019

Submitted To

Dr. P.Santhi Thilagam

Dept. of Computer Science

NITK Surathkal

Group Members:

1. Hardik Rana (16CO138)
2. Shushant Kumar (16CO143)
3. Anmol Horo (16CO206)

Abstract

This report contains the details of the tasks finished as a part of the Phase Three of Compiler design Lab.We have developed a semantic analyzer for C language which make use of parse tree generated in second phase(syntax analyzer) and will interpret symbols,their types and their relations with each other.It will also perform scope resolution,type checking and array-bound checking.

Contents

1. Introduction

The plain parse-tree constructed in that phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them.

For example

E → E + T

The above CFG production has no semantic rule associated with it, and it cannot help in making any sense of the production.

Semantic Analysis:

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

CFG + semantic rules = Syntax Directed Definitions

For example:

int a = “value”;

should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis.

Semantic analysis typically involves:

- Type Checking
- Scope Resolution
- Array-bound checking
- Flow control checks

Yacc Script:

Yacc 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 is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

The structure of our Yacc script is given below; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

Definition section

%{

%}

Rules section

%%

C code section

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.

The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

C Program:

This section describes the input C program which is fed to the yacc script for parsing. The workflow is explained as under:

1.Compile the script using Yacc tool

- **$ yacc –d parser.y -v**

2. Compile the flex script using Flex tool

- **$ lex lexl.l**

3. After compiling the lex file, lex.yy.c file is generated. Also, y.tab.c and y.tab.h files are generated after compiling the yacc script.

4. The three files, lex.yy.c, y.tab.c and y.tab.h are compiled together with the options –ll and –ly

- **$ gcc –w -g y.tab.c -ly -ll -o parser**

5. The executable file is generated, which on running parses the C file given as a command line input

- **$ ./compiler test.c**

6. The script also has an option to take standard input instead of taking input from a file.

2. Design Of Programs

Code:

The entire code for lexical analysis is broken down into 3 files: scanner.l and table.h. The scanner.l and table.h are same as in previous reports.

YACC Code: parser.y file

```
%{
	#include <stdio.h>

	void log_check(int,int);
	void arith_check(int,int);
	void assi_check(int,int);

	#include "symboltable.h"
	#include "lex.yy.c"
	table_t stable[MAX_SCOPE];
	entry_type** ctable;

	int arg_list[15], arg_length = 0, func_composite_diff = 0
	,cur_dtype,assign_right_check = 0, declaration_status = 0, loop_status = 0,
	function_status = 0, function_type ;

	int yyerror(char *msg);
%}

%union
{
	int dtype;
	entry_type* entry;
}

%token SHORT INT LONG CONST VOID CHAR FLOAT
%token <entry> decc hexc charc floatc
%token IF FOR WHILE CONTINUE BREAK RETURN
%token <entry> ID
```

```
%token STRING
%token and or leq geq eq neq
%token mul_asn div_asn mod_asn add_asn sub_asn
%token incr decr
%type <entry> id const arr_index

%type <dtype> sub_expr unary_expr arithm_expr func_call assign_expr arr
expr_left

%left ','
%right '='
%left or
%left and
%left eq neq
%left '<' '>' leq geq
%left '+' '-'
%left '*' '/' '%'
%right '!'
%nonassoc LTE
%nonassoc ELSE

%start starter


%%


for_stmt :FOR '(' expr_stmt  expr_stmt ')' {loop_status = 1;} stmt {loop_status = 0;}
                  |FOR '(' expr_stmt expr_stmt expr ')' {loop_status = 1;} stmt
{loop_status = 0;};

while_stmt : WHILE '(' expr         ')' {loop_status = 1;} stmt {loop_status = 0;};

expr_stmt: expr ';' | ';' ;

expr: expr ',' sub_expr| sub_expr;

sub_expr: sub_expr '>' sub_expr {log_check($1,$3); $$ = $1;}
              |sub_expr '<' sub_expr        {log_check($1,$3); $$ = $1;}
              |sub_expr eq sub_expr         {log_check($1,$3); $$ = $1;}
              |sub_expr neq sub_expr {log_check($1,$3); $$ = $1;}
              |sub_expr leq sub_expr {log_check($1,$3); $$ = $1;}
              |sub_expr geq sub_expr {log_check($1,$3); $$ = $1;}
              |sub_expr and sub_expr {log_check($1,$3); $$ = $1;}
```

```
            |sub_expr or sub_expr {log_check($1,$3); $$ = $1;}
            |'!' sub_expr {$$ = $2;} |arithm_expr {$$ = $1;}
            |assign_expr {$$ = $1;}
            |unary_expr {$$ = $1;}
   ;


assign_expr : expr_left asn_opr  arithm_expr {assi_check($1,$3); $$ = $3;
                  assign_right_check=0;}
          | expr_left asn_opr  arr {assi_check($1,$3); $$ = $3;assign_right_check=0;}
          |expr_left asn_opr  func_call {assi_check($1,$3); $$ =
                  $3;assign_right_check=0;}
          | expr_left asn_opr  unary_expr {assi_check($1,$3); $$ =
                  $3;assign_right_check=0;}
          |unary_expr asn_opr  unary_expr {assi_check($1,$3); $$ =
                  $3;assign_right_check=0;}
   ;

unary_expr:  id incr {$$ = $1->dtype;}
             | id decr  {$$ = $1->dtype;}
             | decr id {$$ = $2->dtype;}
             | incr id {$$ = $2->dtype;}
         ;

expr_left: id  {$$ = $1->dtype;}
             | arr        {$$ = $1;}
          ;

Asn_opr: '=' {assign_right_check=1;}
            |add_asn {assign_right_check=1;}
            |sub_asn {assign_right_check=1;}
            |mul_asn {assign_right_check=1;}
            |div_asn {assign_right_check=1;}
            |mod_asn {assign_right_check=1;}
   ;

arithm_expr: arithm_expr '+' arithm_expr {arith_check($1,$3);}
                |arithm_expr '-' arithm_expr {arith_check($1,$3);}
                |arithm_expr '*' arithm_expr {arith_check($1,$3);}
                |arithm_expr '/' arithm_expr {arith_check($1,$3);}
                |arithm_expr '%' arithm_expr {arith_check($1,$3);}
                |'(' arithm_expr ')' {$$ = $2;} |id    {$$ = $1->dtype;}
                |const  {$$ = $1->dtype;}
   ;
```

```
declaration:      type  declaration_list ';' {declaration_status = 0; }
                  | declaration_list ';'
                  | unary_expr ';'

declaration_list: declaration_list ',' sub_decl     |sub_decl ;

sub_decl: assign_expr |id |arr ;

id: ID { if(declaration_status && !assign_right_check) {
           $1 = insert(stable[current_scope].symbol_table,yytext,INT_MAX,cur_dtype);
           if($1 == NULL) yyerror("This variable is redeclared");
        }
        else {
           $1 = search_recursive(yytext);
           if($1 == NULL) yyerror("This variable not declared");
        }
        $$ = $1;
        }
    ;

const: decc {$1->is_constant=1; $$ = $1;}
             | hexc    {$1->is_constant=1; $$ = $1;}
             | charc {$1->is_constant=1; $$ = $1;}
             | floatc {$1->is_constant=1; $$ = $1;}
    ;

stmts:stmts stmt | ;

stmt:composite_stmt|sole_stmt;

composite_stmt : '{'
                  { if(!func_composite_diff) current_scope = create_new_scope();
                    else func_composite_diff = 0;}
               stmts
               '}' {current_scope = exit_scope();} ;


sole_stmt :if_stmt
          |for_stmt
          |while_stmt |declaration |func_call ';'
         |CONTINUE ';' {if(!loop_status) {yyerror("Illegal use of continue");}} |BREAK ';'
         {if(!loop_status) {yyerror("Illegal use of break");}}
         |RETURN ';'    { if(function_status)
```

```
                                        {       if(function_type != VOID)
                                                yyerror("return type (VOID) does not
match func_defination type");}

                                        else
                                                yyerror("return statement not inside
func_defination definition");
                                  }
              |RETURN sub_expr ';' {    if(function_status)  {
                                        if(function_type != $2)
                                        yyerror("return type does not match
func_defination type");}
                                  else
                                        yyerror("return statement not in
func_defination definition");
                                  };


if_stmt :IF '(' expr ')' stmt %prec LTE |IF '(' expr ')' stmt ELSE stmt ;

type : dtype ptr {declaration_status = 1; } | dtype {declaration_status = 1; } ;

ptr: '*' ptr | '*'        ;

dtype :INT {cur_dtype = INT;} |SHORT  {cur_dtype = SHORT;} |LONG  {cur_dtype =
LONG;} |CHAR {cur_dtype = CHAR;} |FLOAT {cur_dtype = FLOAT;}
              |VOID{cur_dtype =VOID;};


func_defination: type id {  function_type = cur_dtype; declaration_status = 0;
current_scope = create_new_scope();}
       '(' arg_list ')' {
func_composite_diff=1;fill_parameter_list($2,arg_list,arg_length); arg_length = 0;
function_status = 1;declaration_status = 0;}
       composite_stmt { function_status = 0;  };


arg_list : arg_list ',' arg | arg | ;

arg : type id {arg_list[arg_length++] = $2->dtype;};

func_call: id '(' parameter_list ')'{ $$ = $1->dtype;
                           verify_arg_list($1,arg_list,arg_length);
                           arg_length = 0;       }
```

```
    | id '(' ')' { $$ = $1->dtype;
                    verify_arg_list($1,arg_list,arg_length);
                    arg_length = 0;     } ;

parameter_list: parameter_list ','  parameter  | parameter ;

parameter: sub_expr     {arg_list[arg_length++] = $1;} | STRING
{arg_list[arg_length++] = STRING;} ;

arr: id '[' arr_index ']'{
                if(declaration_status)
                {
                    if($3->value <= 0)
                        yyerror("size of array is not positive");
                    else
        if($3->is_constant && !assign_right_check)
                        $1->array_dimension = $3->value;
                    else if(assign_right_check){
                        if($3->value > $1->array_dimension)
                            yyerror("Array index out of bound");
                        if($3->value < 0)
                            yyerror("Array index cannot be negative");
                    }
                }

                else if($3->is_constant)
                {
                    if($3->value > $1->array_dimension)
                        yyerror("Array index out of bound");

                    if($3->value < 0)
                        yyerror("Array index cannot be negative");
                }
                $$ = $1->dtype;
            }

arr_index: const    {$$ = $1;} | id     {$$ = $1;} ;

starter: starter block | block ;

block: func_defination | declaration ;
```

```
%%

void log_check(int left, int right)
{
        if(left != right) yyerror("Mismatch of data types in logical expr");
}

void arith_check(int left, int right)
{
        if(left != right) yyerror("Mismatch of data types in arithmetic expr");
}
void assi_check(int left, int right)
{
        if(left != right) yyerror("Mismatch of data types in assignment expr");
}

int main(int argc, char *argv[])
{
         int i;
         for(i=0; i<MAX_SCOPE;i++){
          stable[i].symbol_table = NULL;
          stable[i].parent = -1;
         }

        ctable = create_table();
        stable[0].symbol_table = create_table();
        yyin = fopen("test.c", "r");

        if(!yyparse()) printf("\nParsing completed successfully\n\n\n");
        else printf("\nParsing not completed\n\n\n");

        printf("SYMBOL TABLE VIEW\n\n");
        stable_disp();

        printf("CONSTANT TABLE VIEW");
        ctable_disp(ctable);

        fclose(yyin);
        return 0;
}

int yyerror(char *msg)
{
        printf("Line no: %d Error message: %s Token: %s\n", yylineno, msg, yytext);
```

```
        exit(0);
}
```

Features

- Checks for undeclared variables
- Checks for redeclaration of variables
- Type checking of variables
- Mathematical operations compatibility
- Checks if the return type of a function matches with the datatype of the variable/constant being returned
- Checks if the number of parameters in a function call matches with the number of parameters in the function definition
- Checks if data types of parameters in function call and its definition match
- Checks for scope of variables and reports an error if a variable is out of scope
- Identifies the dimensionality of arrays and checks if index of an array is a positive integer or not

3. Test Cases

Without Errors:

S.No	Test Case	Expected Output	Status

1	int foo(int a)
	{
	a= a + 1;
	return a;
	}
		Parsing completed successfully
		
		SYMBOL TABLE VIEW
		
		Scope: 0
		--- 	
		lexeme data_type array_dim num_args parameter_list
		
		--- 	
			PASS

		foo 259 -1 1 259 main 262 -1 0 -- ------------------ Scope: 1 -- ------------------ lexeme data_type array_dim num_args parameter_list -- ------------------ a 259 -1 0 -- ------------------ Scope: 2 -- ------------------ lexeme data_type array_dim num_args parameter_list -- ------------------ c 259 -1 0 a 259 -1 0 b 259 -1 0 -- ------------------	

<table>
<tr>
<td>2</td>
<td>#include <stdio.h>
int main()
{
int a=5;
{
int a = 10;
{
int a = 100;
}
}
return 0;
}</td>
<td>Parsing completed successfully

SYMBOL TABLE VIEW

Scope: 0

----
lexeme data_type array_dim num_args parameter_list

----
main 259 -1 0

Scope: 1
----
lexeme data_type array_dim num_args parameter_list

----
a 259 -1 0

Scope: 2
----
lexeme data_type array_dim num_args parameter_list

---</td>
<td>PASS</td>
</tr>
</table>

		a 259 -1 0 --- Scope: 3 --- lexeme data_type array_dim num_args parameter_list --- a 259 -1 0 ---	

With Errors:

S.No	Test Case	Expected Output	Status

1	int main()
	{
	int a=4;
	b=9;
	return 0;
	}
		Error message: This variable not declared Token: b	PASS

2	int a[0];	Error message: size of array is not positive Token:]	PASS

3	int main()
	{
	foo();
	Return 0;
		Error message: This variable not declared Token: foo	PASS

	}		

4	int foo(int a)
	{
	float s;
	return s;
	}
		Error message: return type does not match func_defination type Token: ;	PASS

5	int a = 5.4;	Error message: Mismatch of data types in assignment expr Token: ;	PASS

Test Case 1:

```
#include<stdio.h>

int foo(int a)
{
        a= a + 1;
        return a;
}

void main()
{
   int a,b,c;
        b=5;
        c=foo(b);
        return;
}
```

Output 1:

```
Parsing completed successfully


SYMBOL TABLE VIEW

Scope: 0

-------------------------------------------------------------------------------------------------
 lexeme       data_type     array_dim     num_args      parameter_list

-------------------------------------------------------------------------------------------------
 foo         259          -1            1           259
 main         262          -1            0

-------------------------------------------------------------------------------------------------


Scope: 1

-------------------------------------------------------------------------------------------------
 lexeme       data_type     array_dim     num_args      parameter_list

-------------------------------------------------------------------------------------------------
 a          259          -1            0

-------------------------------------------------------------------------------------------------


Scope: 2

-------------------------------------------------------------------------------------------------
 lexeme       data_type     array_dim     num_args      parameter_list

-------------------------------------------------------------------------------------------------
 c          259          -1            0
```

```
a          259        -1          0
b          259        -1          0

--------------------------------------------------------------------------------------------------------



CONSTANT TABLE VIEW
-------------------------
 lexeme     data-type


-------------------------
 5        259
 1        259


-------------------------
```

Test Case 2:

```
#include <stdio.h>
int main()
{
        int a=4;
        b=9;            //undeclared variable
        a=10;
        return 0;
}
```

Output 2 :

```
Line no: 5 Error message: This variable not declared Token: b
```

Test Case 3:

```
#include <stdio.h>
void main()
{
        int a[0];       //array index error (has to be greater than zero)
        int b[8.5];     //array index cannot be a float value
        int c=5;
        float d;
      int i;
      int c=3; //redeclaration of variable 'c'
      int sum;
      sum=0;

        for(i=0;i<12;i++)
        {
                sum=sum+i;
        }
        return;
}
```

Output 3 :

```
Line no: 4 Error message: size of array is not positive Token: ]
```

Test Case 4:

```
#include <stdio.h>
int main()
{
        int a=4;
        int b=5;
        {
                int d=56;
        }
        int sum;
        sum=a+b;
        {
                int d=20;
        }
        foo();
        d=89;        //variable 'd' out of scope
}
```

Output 4 :

```
Line no: 14 Error message: This variable not declared Token: foo
```

Test Case 5:

```
#include<stdio.h>
int foo(int a)
{
        float s;
        return s;    //return type mismatch
}
int sum(int b,int c)
{
```

```
        int s;
        s=b+c;
        return s;
}
void main()
{
        int p=3;
        int q=4;
        float f=4.5;
        int d;
        d=sum(p,f);  //parameter type mismatch
        return;
}
```

Output 5 :

```
Line no: 5 Error message: return type does not match func_defination type Token: ;
```

Test Case 6:

```
#include <stdio.h>
int main()
{
        int a=5;
        {
                int a = 10;
                {
                        int a = 100;
                }
        }
        return 0;
```

```
}
```

Output 6 :

```
Parsing completed successfully


SYMBOL TABLE VIEW

Scope: 0

-----------------------------------------------------------------------------------------------
 lexeme        data_type     array_dim     num_args      parameter_list

-----------------------------------------------------------------------------------------------
 main          259           -1            0

-----------------------------------------------------------------------------------------------



Scope: 1

-----------------------------------------------------------------------------------------------
 lexeme        data_type     array_dim     num_args      parameter_list

-----------------------------------------------------------------------------------------------
 a             259           -1            0

-----------------------------------------------------------------------------------------------
```

Scope: 2

----

lexeme	data_type	array_dim	num_args	parameter_list
a	259	-1	0	

Scope: 3

----

lexeme	data_type	array_dim	num_args	parameter_list
a	259	-1	0	

CONSTANT TABLE VIEW

lexeme	data-type
5	259
10	259
100	259
0	259

```
-------------------------
```

Test Case 7:

```
#include <stdio.h>
int main()
{
	int a = 5.4;	//type mismatch
	int b=67;
	float f=6.7;
	b=f;		//type mismatch (int = float)
	return 0;
}
```

Output 7 :

```
Line no: 4 Error message: Mismatch of data types in assignment expr Token: ;
```

4. Implementation

The semantic analyzer built for the subset of C language reports various errors present (if any) and has the following functionalities:

1. Checks for undeclared variables
2. Checks for redeclaration of variables
3. Type checking of variables
4. Mathematical operations compatibility
5. Checks if the return type of a function matches with the datatype of the variable/constant being returned
6. Checks if the number of parameters in a function call matches with the number of parameters in the function definition
7. Checks if data types of parameters in function call and its definition match
8. Checks for scope of variables and reports an error if a variable is out of scope

9. Identifies the dimensionality of arrays and checks if index of an array is a positive integer or not

5.Results and Future Works:

Results:

The lexical analyzer, syntax analyzer and the semantic analyzer for a subset of C language, which include selection statements, compound statements, iteration statements (for, while and do-while) and user defined functions is generated. It is important to define unambiguous grammar in the syntax analysis phase. The semantic analyzer performs type checking, reports various errors such as undeclared variable, type mismatch, errors in function call (number and datatypes of parameters mismatch) and errors in array indexing.

The lex file (parser.l) and yacc (parser.y) are compiled using following commands:

```
#!/bin/bash
lex lexer.l
yacc -d parser.y -v
gcc -w -g y.tab.c -ly -ll -o semantic_analyser
./semantic_analyser
```

After parsing, if there are errors then the line numbers of those errors are displayed along with a ‘parsing failed’ on the terminal. Otherwise, a ‘parsing complete’ message is displayed on the console. The symbol table with stored & updated values is always displayed, irrespective of errors.

Future Works:

We have implemented the parser and semantic analyzer for most of the sets of C language. The future work may include defining the grammar and specifying the semantics for switch statements, predefined functions (like string functions, file read and write functions), jump statements and enumerations.

6.References

1. Lex and Yacc By John R. Levine, Tony Mason, Doug Brown

2. Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

3. https://en.wikipedia.org/wiki/Symbol_table

4.https://www.cse.iitb.ac.in/~br/courses/cs699-autumn2013/refs/lextut-victor-eijkhout.pdf

5.https://www2.cs.arizona.edu/~debray/Teaching/CSc453/DOCS/tutorial-large.pdf

6.http://epaperpress.com/lexandyacc/download/LexAndYaccTutorial.pdf - Lex and Yacc Tutorial by Tom Nieman

7.Jenkins Hash Function on Wikipedia :
https://en.wikipedia.org/wiki/Jenkins_hash_function#one-at-a-time