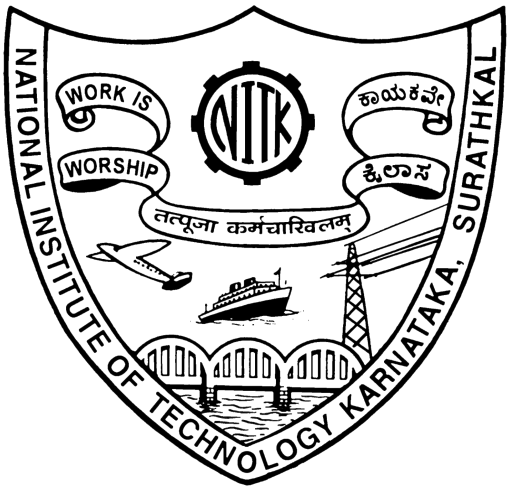
Semantic Analyser for the C Language



National Institute of Technology Karnataka Surathkal

Date: 14th March 2019

Submitted To:
Prof. P. Santhi Thilagam
CSE Dept, NITK

Group Members:
Namrata Ladda, 16CO121
Mehnaz Yunus, 16CO124
Sharanya Kamath, 16CO140

Abstract

This report contains the details of the tasks finished as a part of Phase Three of Compiler Design Lab. We have developed a Semantic Analyser for C language which makes use of the parser of the previous phase. The objective of this assignment is to perform semantic analysis such as type and scope analysis and declaration processing, and integrate such analyses with the parser.

The following functionalities are being checked in Semantic Analysis:

- Symbol table insertion for different scopes
- Multiple functions
- Assignment expression
- Undeclared variable
- Redeclaration in same scope
- Out of scope
- Type mismatch
- Redeclaration of pre defined function
- Return type of function mismatch
- Same variable different scopes
- Different data types
- Usage of non-array variable with subscript
- Out of bounds subscript
- Usage of array identifier without subscript

RESULTS

- Semantic Errors in the source program along with appropriate error messages
- Symbol table is displayed with appropriate attributes.

TOOLS USED

- Yacc
- Flex

Contents :

1. Introduction

1.1 Semantic Analysis

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of the semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array. The language specification may permit some type conversions called coercions. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

Semantics

The 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.

These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking
- Array-bound checking

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has a well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

1.2 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.

1.3 C Program

The workflow is explained as under:

- Compile the script using Yacc tool
  `$ yacc semantic.y`
- Compile the flex *s*cript using Flex tool
- `$ lex lexer.l`
- 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.

- Compilation is done with the options –ll, –ly and -w

2. Design of Programs

2.1 Code

```
%{
#include <string.h>
#include<stdio.h>
#include<stdlib.h>
#define YYSTYPE char*
extern int yylineno;
char type[200];
int scope = 0;
int n = 1;
char funcReturnType[100][100];
int currType = 0;
int var = 0; //no of variables declared in one statement

struct declaration_info {
    char id[100];
    char value[100];
}dec_info[100];

struct symbolTable
{
    char token[100];
    char type[100];
    int tn;
    float fvalue;
    int value;
    int scope;
    int isFunction;
    int fType[100];
    char paramType[100];
    int numParams;
}table[100];

void saveReturnType(int currType, char* returnType) {
    strcpy(funcReturnType[currType], returnType);
}

int isPresent(char* token){ //check if token is already present in
symbol table
    int i;
```

```
    for(i = 1; i < n; ++i)
        if(!strcmp(table[i].token, token))
            return i;
    return 0;
}

int getNumParams(char* token){
    int i;
    for(i = 1; i < n; ++i)
        if(strcmp(table[i].token, token) == 0)
            return table[i].numParams;
}

char* getParamType(char* token){
    int i;
    for(i = 1; i <n; ++i)
        if(strcmp(table[i].token, token) == 0)
            return table[i].paramType;
}

char* getDataType(char* token){
    int i;
    for(i = 1; i < n; ++i)
        if(strcmp(table[i].token, token) == 0)
            return table[i].type;
}

int getMinScope(char* token) {
    int i;
    int min = 999;
    for(i = 1; i < n; ++i){
        if(strcmp(table[i].token, token) == 0)
            if(table[i].scope < min)
                min = table[i].scope;
    }
    return min;
}

void storeValue(char* token, char* value, int scope){
    int i;
    for(i = 1; i < n; ++i){
        if(strcmp(table[i].token, token) == 0 && table[i].scope ==
scope) {
            if(strcmp(table[i].type, "float") == 0)
                table[i].fvalue = atof(value);
```

```
            if(strcmp(table[i].type, "int") == 0)
                table[i].value = atoi(value);
            if(strcmp(table[i].type, "char") == 0)
                table[i].value = value;
            break;
        }
    }
}

void insert(char* token, char* type, int scope, char* value) {
    if(!isPresent(token) || table[isPresent(token)].scope != scope){
        strcpy(table[n].token, token);
        strcpy(table[n].type, type);
        table[n].numParams = 0;
        table[n].isFunction = 0;
        table[n].scope = scope;
        if(strcmp(type, "float") == 0)
            table[n].fvalue = atof(value);
        if(strcmp(type, "int") == 0)
            table[n].value = atoi(value);
        n++;
    }
    return;
}

void insertFunction(char* token, char* type, char* paramType, int
numParams){
    if(!isPresent(token)){
        strcpy(table[n].token, token);
        strcpy(table[n].type, type);
        strcpy(table[n].paramType, paramType);
        table[n].numParams = numParams;
        table[n].isFunction = 1;
        table[n].scope = scope;
        n++;
    }
    return;
}

void showSymbolTable(){
    int i;

printf("\n\n-----------------------------------------------------------------
-------------------------------\n\t\t\t\tSYMBOL
TABLE\n-----------------------------------------------------------------
```

```
---------------------------\n");
    printf("\n\nToken\tType\tParameterType\tNo of
Parameters\tisFunction\tValue\t\tScope");

printf("\n-----------------------------------------------------------------
-----------------------------------\n");
    for(i = 1; i < n; ++i){
        printf("\n%s\t%s\t%s\t\t%d\t\t\t%d\t\t",
        table[i].token,
        table[i].type,
        (table[i].isFunction ? table[i].paramType : "-"),
        (table[i].isFunction ? table[i].numParams : 0),
        table[i].isFunction);

        !strcmp(table[i].type, "int") ?
table[i].value==9999?printf("-\t\t"):printf("%d\t\t",table[i].value) :
table[i].fvalue==9999.000000?printf("-\t\t"):printf("%f\t",table[i].fval
ue);
        printf("%d",table[i].scope);
    }
    printf("\n");
}

void end() {
    showSymbolTable();
    printf("\nParsing Failed\n");
    exit(0);
}

%}

%left '<' '>' '='
%left '+' '-'
%left '*' '/' '%'
%token IDENTIFIER
%token INT FOR RETURN PRINTF FLOAT VOID WHILE CHAR
%token STRING_CONST INT_CONST FLOAT_CONST
%token ADD_ASSIGN

%start start_state
%glr-parser

%%

start_state:
```

```
    function_definition_list
    ;

function_definition_list:
    function_definition_list function_definition
    | function_definition
    ;

function_definition:
    type_specifier IDENTIFIER '(' ')' compound_statement
      {
        if(strcmp($1, funcReturnType[currType-1]) != 0){
            printf("\nError: Type mismatch at line no %d\n", yylineno);
            end();
        }
        else if(isPresent($2)) {
            printf("\nError: Redefinition of function at line no %d\n",
yylineno);
            end();
        }
        else
        insertFunction($2, $1, "-", 0);
    }
   | type_specifier IDENTIFIER '(' type_specifier_fn IDENTIFIER ')'
compound_statement
    {
        if(strcmp($1, funcReturnType[currType-1]) != 0){
            printf("\nError Type mismatch at line no %d\n", yylineno);
            end();
        }
        else if(isPresent($2)) {
            printf("\nError: Redefinition of function at line no %d\n",
yylineno);
            end();
        }
        else
            insertFunction($2, $1, $4, 1);
    }
    ;

type_specifier:
      INT    { $$ = "int"; strcpy(type, $1); }
    | FLOAT { $$ = "float"; }
    | VOID { $$ = "void"; }
    | CHAR { $$ = "char"; }
```

```
    ;
type_specifier_fn:
      INT    { $$ = "int"; }
    | FLOAT { $$ = "float"; }
    ;


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

statement_list:
      statement
    | statement_list statement
    ;

statement:
      compound_statement
    | expression_statement
    | iteration_statement
    | jump_statement
    | print_statement
   | function_call_statement
    ;

expression_statement:
      expression ';'
    | ';'
    ;

expression:
      assignment_expression
    | expression ',' assignment_expression
    ;

assignment_expression:
      all_expression
    | assignment_expression assignment_operator all_expression
    {
```

```
        storeValue($1, $3, scope);
    }
    ;

assignment_operator:
      '='
    | ADD_ASSIGN
    ;

all_expression:
      fundamental_expression
    | all_expression '>' all_expression
    | all_expression '<' all_expression
    | all_expression '+' all_expression
    {
        char ans[100];
        int a = atoi($1) + atoi($3);
        sprintf(ans, "%d", a);
        strcpy($$, ans);
    }
    | all_expression '-' all_expression
    {
        char ans[100];
        int a = atoi($1) - atoi($3);
        sprintf(ans, "%d", a);
        strcpy($$, ans);
    }
    | all_expression '/' all_expression
    {
        char ans[100];
        int a = atoi($1) / atoi($3);
        sprintf(ans, "%d", a);
        strcpy($$, ans);
    }
    | all_expression '*' all_expression
    {
        char ans[100];
        int a = atoi($1) * atoi($3);
        sprintf(ans, "%d", a);
        strcpy($$, ans);
    }
    | all_expression '%' all_expression
    {
        char ans[100];
        int a = atoi($1) % atoi($3);
```

```
        sprintf(ans, "%d", a);
        strcpy($$, ans);
    }
    ;

fundamental_expression:
    IDENTIFIER
    {
        if(!isPresent($1)) {
            printf("\nError: Undeclared variable at line %d\n",
yylineno);
            end();
        }
        else{
            int minScope = getMinScope($1);
            if(scope < minScope){
                printf("\nError Variable out of scope at line %d\n",
yylineno);
                end();
            }
        }

    }
    | STRING_CONST
    | INT_CONST
    | FLOAT_CONST
    | '(' expression ')'
    ;

iteration_statement:
      FOR '(' expression statement expression statement expression ')'
    | FOR '(' expression statement expression statement ')'
    | WHILE '(' expression ')'
    ;

jump_statement:
      RETURN INT_CONST ';'
      {
        strcpy(funcReturnType[currType], "int");
        currType++;
      }
    | RETURN FLOAT_CONST ';'
    { saveReturnType(currType, "float"); currType++; }
    | RETURN ';'
    {
```

```
        saveReturnType(currType, "void");
        currType++;
    }
    | RETURN IDENTIFIER ';'
    {
        char* types = getDataType($2);
        printf("Type%s", types);
        saveReturnType(currType, types);
        currType++;
    }
    ;

print_statement:
      PRINTF '(' STRING_CONST ',' init_declarator_list ')' ';'
    ;

function_call_statement:
      IDENTIFIER '(' ')' ';'
    {
        if(!isPresent($1)){
            printf("\nError Undeclared function at line no %d\n",
yylineno);
            end();
        }
        else if(getNumParams($1) != 0) {
            printf("\nError at line no %d\n", yylineno);
            end();
        }
    }
   | IDENTIFIER '(' IDENTIFIER ')' ';'
    {
        if(!isPresent($1)){
            printf("Error Undeclared function at line no %d\n",
yylineno);
            end();
        }
        else if(getNumParams($1) != 1){
            printf("Error No of parameters does not match signature at
line no %d\n", yylineno);
            end();
        }
        else if(strcmp(getParamType($1), getDataType($3)) != 0){
            printf("Error Parameter type does not match signature at
line no %d\n", yylineno);
            end();
```

```
        }

    }
   | IDENTIFIER '(' INT_CONST ')'';'
    {
        if(!isPresent($1)){
            printf("\nError Undeclared function at line no %d\n",
yylineno);
            end();
        }
        else if(getNumParams($1) != 1){
            printf("\nError No of parameters does not match signature at
line no %d\n", yylineno);
            end();
        }
        else if(strcmp(getParamType($1), "int") != 0){
            printf("\nError Parameter type does not match signature at
line no %d\n", yylineno);
            end();
        }

    }
    | IDENTIFIER '(' FLOAT_CONST ')' ';'
    {
        if(!isPresent($1)){
            printf("Error Undeclared function at line no %d\n",
yylineno);
            end();
        }
        else if(getNumParams($1) != 1){
            printf("Error No of parameters does not match signature at
line no %d\n", yylineno);
            end();
        }
        else if(strcmp(getParamType($1), "float") != 0){
            printf("Error Parameter type does not match signature at
line no %d\n", yylineno);
            end();
        }

    }
    ;

declaration_list:
      declaration
```

```
    | declaration_list declaration
    ;

declaration:
    type_specifier init_declarator_list ';'
    {
        for(int i = 0; i < var; ++i){
            if(!isPresent(dec_info[i].id) ||
table[isPresent(dec_info[i].id)].scope != scope)
                insert(dec_info[i].id, $1, scope, dec_info[i].value);
            else{
                printf("\nError: Redeclaration of variable at line no
%d\n", yylineno);
                end();
            }
        }
        var = 0;
    }
    ;

init_declarator_list:
      init_declarator
    | init_declarator_list ',' init_declarator
    ;

init_declarator:
    variable
    {
    strcpy(dec_info[var].id, $1);
    strcpy(dec_info[var].value, "9999");
    var++;
    }
    | variable '=' assignment_expression
    {
        strcpy(dec_info[var].id, $1);
        strcpy(dec_info[var].value, $3);
        var++;
    }
    ;

variable:
      IDENTIFIER
    ;
%%
```

```c
#include "lex.yy.c"
#include <stdio.h>
#include <string.h>
#include <ctype.h>


int err = 0;
int main(int argc, char* argv[]) {
    yyin = fopen(argv[1], "r");
    yyparse();

    if(err == 0)
        printf("\nParsing Complete\n");
    else
        printf("\nParsing Failed\n");

    fclose(yyin);

    showSymbolTable();
}

extern char* yytext;
extern int yylineno;
yyerror() {
    err = 1;
    printf("\nYYError at line no %d\n", yylineno);
    exit(0);
}

void startBlock() {
    scope++;
    return;
}

void endBlock() {
    scope--;
    return;
}
```

2.2 Explanation

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

3. Test Cases

Without Errors:

1. **test0.c**
   Symbol Table insertion for different scopes.

```c
#include <stdio.h>
int main()
{
	int y=5;
	float z;
	{
		int y;
		while(y<10)
		{
			float w;
		}
	}
	return 0;
}
```

Output

```
$ ./a.out test1.c

Inserting function main

Parsing Complete


------------------------------------------------------------------------------------------------
                                SYMBOL TABLE
------------------------------------------------------------------------------------------------

Token    Type     ParameterType    No of Parameters        isFunction      Value           Scope
------------------------------------------------------------------------------------------------

y        int      -                0                       0               5               1
z        float    -                0                       0               -               1
y        int      -                0                       0               -               2
w        float    -                0                       0               -               3
main     int      -                0                       1               0               0
$
```

2. **test1.c**
   Multiple Functions.

```c
#include <stdio.h>
int sum()
{
	int var;
	return 1;
}

float func(int x)
{
	return 1.0;
}

int main()
{
	int a=5;
	while(1){
		int x;
	}
}
```

Output

```
$ ./a.out test2.c

Inserting function sum

Inserting function func

Error: Type mismatch at line no 19


------------------------------------------------------------------------------------------------
                                SYMBOL TABLE
------------------------------------------------------------------------------------------------

Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
------------------------------------------------------------------------------------------------

var     int     -               0                       0               -               1
sum     int     -               0                       1               0               0
func    float   int             1                       1               0.000000        0
a       int     -               0                       0               5               1
x       int     -               0                       0               -               2

Parsing Failed
$
```

3. **test2.c**

Assignment of expression

```c
#include <stdio.h>
int main()
{
	int a=10;
	float b=4.5;
	a=5+9;
	return 0;
}
```

Output

```
$ ./a.out test10.c
$$ 14
Parsing Complete


------------------------------------------------------------------------------------------------
                                SYMBOL TABLE
------------------------------------------------------------------------------------------------

Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
------------------------------------------------------------------------------------------------

a       int     -               0                       0               14              1
b       float   -               0                       0               4.500000        1
main    int     -               0                       1               0               0
$
```

4. **test7.c**

Same variable different scopes

```c
#include <stdio.h>
int main()
{
	int x;
	x=10;
	while(1)
	{
		while(1)
		{
			int x=4;
		}
	}
	int g;
	return 0;
}
```

Output

```
$ ./a.out test7.c

Inserting function main

Parsing Complete


--------------------------------------------------------------------------------------------
                                SYMBOL TABLE
--------------------------------------------------------------------------------------------

Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
--------------------------------------------------------------------------------------------

x       int     -               0                       0               10              1
x       int     -               0                       0               4               3
g       int     -               0                       0               -               1
main    int     -               0                       1               0               0
$
```

5. **test8.c**

Different data types

```c
#include <stdio.h>

int main()
{
	int a=10;
	float b=3.14;
	return 0;
}
```

Output

```
$ ./a.out test8.c

Inserting function main

Parsing Complete


------------------------------------------------------------------------------------------------
                                SYMBOL TABLE
------------------------------------------------------------------------------------------------

Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
------------------------------------------------------------------------------------------------

a       int     -               0                       0               10              1
b       float   -               0                       0               3.140000        1
main    int     -               0                       1               0               0
$
```

With Errors:

6. **test3.c**
   Undeclared variable.

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

Output

```
$ ./a.out test3.c

Error: Undeclared variable at line 5


------------------------------------------------------------------------------------------------
                                SYMBOL TABLE
------------------------------------------------------------------------------------------------

Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
------------------------------------------------------------------------------------------------

a       int     -               0                       0               4               1

Parsing Failed
$
```

7. **test4.c**

   Redeclaration of variable in same scope.

```c
#include <stdio.h>
int main()
{
	int x=4;
	int y=9;
	int x=10;
	return 0;
}
```

Output

```
$ ./a.out test4.c

Error: Redeclaration of variable at line no 6


------------------------------------------------------------------------------------------
                              SYMBOL TABLE
------------------------------------------------------------------------------------------


Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
------------------------------------------------------------------------------------------

x       int     -               0                       0               4               1
y       int     -               0                       0               9               1

Parsing Failed
$
```

8. **test5.c**

   Accessing out of scope variable.

```c
#include <stdio.h>
int main()
{
	int a=4;
	while(1)
	{
		int num=10;
	}
	a=5;
	num=8;
	return 0;
}
```

Output

```
$ ./a.out test5.c

Error Variable out of scope at line 10


------------------------------------------------------------------------------------------------
                                SYMBOL TABLE
------------------------------------------------------------------------------------------------

Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
------------------------------------------------------------------------------------------------

a       int     -               0                       0               5               1
num     int     -               0                       0               10              2

Parsing Failed
$
```

9. **test6.c**

   Return type of function mismatch.

```c
#include <stdio.h>
int sum(int b)
{
	return 3.14;
}

int main()
{
	sum(10);
	//float z,y=7.0;
	return 0;
}
```

Output

```
$ ./a.out test6.c

Error Type mismatch at line no 5


--------------------------------------------------------------------------------------------
                                SYMBOL TABLE
--------------------------------------------------------------------------------------------

Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
--------------------------------------------------------------------------------------------


Parsing Failed
$
```

10. **test9.c**

    Parameter mismatch

```c
#include<stdio.h>

int square(int a)
{
	return 1;
}

int main()
{
	square();
	return 0;
}
```

Output

```
$ ./a.out test9.c

Error at line no 10


--------------------------------------------------------------------------------------------
                                SYMBOL TABLE
--------------------------------------------------------------------------------------------

Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
--------------------------------------------------------------------------------------------

square  int     int             1                       1               0               0

Parsing Failed
$
```

4. Implementation

The yacc scripts takes the stream of tokens recognized by lexer. The following semantic error are checked in this phase :

- Undeclared variable
- Redeclaration of variable in same scope
- Variable out of scope
- Return type of function misatch
- Number of parameters in a function

5. Results

Tokens recognized by the lexer are successfully parsed in the parser. The output displays the set of identifiers and constants present in the program with their types. The parser generates error messages in case of any syntactical or semantic errors in the test program.

6. Future Work

The yacc script presented in this report takes care of all the semantic rules of C language but is not fully exhaustive in nature. Our future work would include making the script even more robust in order to handle function parameters and other corner cases and thus make it more effective. Also we would work on Intermediate Code Generator.

7. References

- http://dinosaur.compilertools.net/yacc/
- https://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/tutorial-large.pdf
- Compilers Principles Techniques and Tools book