RV COLLEGE OF ENGINEERING ®
BENGALURU-560059
(Autonomous Institution Affiliated to VTU, Belagavi)

"Semantic Analysis"

Report

Compiler Design

(18IS54)

Submitted By

Ankit Kumar Singh (1RV18IS007)

Under the Guidance of

B. K. Srinivas

Asst. Professor

in partial fulfillment for the award of degree of

Bachelor of Engineering

in

INFORMATION SCIENCE AND ENGINEERING

2020-21

Abstract:

Lexical analysers cannot detect errors in the structure of a language (syntax), unbalanced parenthesis etc. These errors were handled by a parser. But in the syntax analysis phase, we don't check if the input is semantically correct. After the parser checks if the code is structured correctly, semantic analysis phase checks if that syntax structure constructed in the source program derives any meaning or not. The output of the syntax analysis phase is parse tree whereas that of semantic phase is annotated parse tree.
Semantic analysis is done by modifications in the parser code only. The following tasks are performed in semantic analysis:

1. Label Checking
2. Type Checking
3. Array Bounds Checking

Contents:

Introduction:

Semantic Analysis:

After the lexical analysis stage, we get the stream of tokens from source C code which is given as input to the parser. Parser verifies that a string of token names can be generated by the grammar of the source language. We expect the parser to check the structure of the input program and report any syntax errors. Semantic analysis phase checks the semantics of the language.
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.
Semantic analysis typically involves in following tasks:

1. Type Checking – Data types are used in a manner that is consistent with their definition (i. e., only with compatible data types, only with operations that are defined for them, etc.)
2. Label Checking – Labels references in a program must exist.
3. Array Bound Checking – When declaring an array, subscript should be defined properly.

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

1. Type mismatch
    a. Return type mismatch.
    b. Operations on mismatching variable types.
2. Undeclared variable
    a. Check if variable is undeclared globally.
    b. Check if variable is visible in current scope.
3. Reserved identifier misuse.
    a. Function name and variable name cannot be same.
    b. Declaration of keyword as variable name.
4. Multiple declaration of variable in a scope.
5. Accessing an out of scope variable.
6. Actual and formal parameter mismatch.

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. Lexer can be used to make a simple parser. But it needs making extensive use of the user-defined states.

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.

In the rules section, each grammar rule defines a symbol in terms of:

1. Other symbols
2. Tokens (or terminal symbols) which come from the lexer.

Each rule can have an associated action, which is executed *after* all the component symbols of the rule have been parsed. Actions are basically C-program statements surrounded by curly braces.

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:

- Compile the script using Yacc tool

```
$ yacc -d c_parser.y
```

- Compile the flex script using Flex tool

```
$ flex c_lexer.l
```

- After compiling the lex file, a lex.yy.c file is generated. Also, y.tab.c and y.tab.h files are generated after compiling the yacc script.
- The three files, lex.yy.c, y.tab.c and y.tab.h are compiled together with the options –ll and –ly

```
$ gcc -o compiler lex.yy.c y.tab.h y.tab.c -ll -ly
```

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

```
$ ./compiler test.c
```

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

Design of Programs:

Code:

Updated Lexer Code:

```
%{
     void yyerror(char* s);
     int yylex();
     #include "stdio.h"
     #include "stdlib.h"
     #include "ctype.h"
     #include "string.h"
     void ins();
     void insV();
     int flag=0;
     #define ANSI_COLOR_RED      "\x1b[31m"
     #define ANSI_COLOR_GREEN    "\x1b[32m"
     #define ANSI_COLOR_CYAN     "\x1b[36m"
     #define ANSI_COLOR_RESET    "\x1b[0m"
     extern char curid[20];
     extern char curtype[20];
     extern char curval[20];
     extern int currnest;
     void deletedata (int );
     int checkscope(char*);
     int check_id_is_func(char *);
     void insertST(char*, char*);
     void insertSTnest(char*, int);
     void insertSTparamscount(char*, int);
     int getSTparamscount(char*);
     int check_duplicate(char*);
     int check_declaration(char*, char *);
     int check_params(char*);
     int duplicate(char *s);
     int checkarray(char*);
```

```
    char currfunctype[100];
    char currfunc[100];
    char currfunccall[100];
    void insertSTF(char*);
    char gettype(char*,int);
    char getfirst(char*);
    extern int params_count;
    int call_params_count;
%}

%nonassoc IF
%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED
STRUCT
%token RETURN MAIN
%token VOID
%token WHILE FOR DO
%token BREAK
%token ENDIF
%expect 1

%token identifier array_identifier func_identifier
%token integer_constant string_constant float_constant
character_constant

%nonassoc ELSE

%right leftshift_assignment_operator
rightshift_assignment_operator
%right XOR_assignment_operator OR_assignment_operator
%right AND_assignment_operator modulo_assignment_operator
%right multiplication_assignment_operator
division_assignment_operator
%right addition_assignment_operator
subtraction_assignment_operator
%right assignment_operator

%left OR_operator
```

```
%left AND_operator
%left pipe_operator
%left caret_operator
%left amp_operator
%left equality_operator inequality_operator
%left lessthan_assignment_operator lessthan_operator
greaterthan_assignment_operator greaterthan_operator
%left leftshift_operator rightshift_operator
%left add_operator subtract_operator
%left multiplication_operator division_operator
modulo_operator

%right SIZEOF
%right tilde_operator exclamation_operator
%left increment_operator decrement_operator


%start program

%%
program
              : declaration_list;

declaration_list
              : declaration D

D
              : declaration_list
              | ;

declaration
              : variable_declaration
              | function_declaration

variable_declaration
              : type_specifier variable_declaration_list ';'
```

```
variable_declaration_list
              : variable_declaration_list ','
variable_declaration_identifier |
variable_declaration_identifier;

variable_declaration_identifier
              : identifier
{if(duplicate(curid)){printf("Duplicate\n");exit(0);}insertST
nest(curid,currnest); ins();  } vdi
                | array_identifier
{if(duplicate(curid)){printf("Duplicate\n");exit(0);}insertST
nest(curid,currnest); ins();  } vdi;



vdi : identifier_array_type | assignment_operator
simple_expression  ;

identifier_array_type
              : '[' initilization_params
              | ;

initilization_params
              : integer_constant ']' initilization {if($$ <
1) {printf("Wrong array size\n"); exit(0);} }
              | ']' string_initilization;

initilization
              : string_initilization
              | array_initialization
              | ;

type_specifier
              : INT | CHAR | FLOAT  | DOUBLE
              | LONG long_grammar
              | SHORT short_grammar
              | UNSIGNED unsigned_grammar
```

```
              | SIGNED signed_grammar
              | VOID  ;

unsigned_grammar
              : INT | LONG long_grammar | SHORT
short_grammar | ;

signed_grammar
              : INT | LONG long_grammar | SHORT
short_grammar | ;

long_grammar
              : INT  | ;

short_grammar
              : INT | ;

function_declaration
              : function_declaration_type
function_declaration_param_statement;

function_declaration_type
              : type_specifier identifier '('  {
strcpy(currfunctype, curtype); strcpy(currfunc, curid);
check_duplicate(curid); insertSTF(curid); ins(); };

function_declaration_param_statement
              : params ')' statement;

params
              : parameters_list | ;

parameters_list
              : type_specifier { check_params(curtype); }
parameters_identifier_list { insertSTparamscount(currfunc,
params_count); };
```

```
parameters_identifier_list
              : param_identifier
parameters_identifier_list_breakup;

parameters_identifier_list_breakup
              : ',' parameters_list
              | ;

param_identifier
              : identifier { ins();insertSTnest(curid,1);
params_count++; } param_identifier_breakup;

param_identifier_breakup
              : '[' ']'
              | ;

statement
              : expression_statment | compound_statement
              | conditional_statements |
iterative_statements
              | return_statement | break_statement
              | variable_declaration;

compound_statement
              : {currnest++;} '{'  statment_list  '}'
{deletedata(currnest);currnest--;}  ;

statment_list
              : statement statment_list
              | ;

expression_statment
              : expression ';'
              | ';' ;

conditional_statements
              : IF '(' simple_expression ')'
```

```
{if($3!=1){printf("Condition checking is not of type
int\n");exit(0);}} statement conditional_statements_breakup;

conditional_statements_breakup
              : ELSE statement
              | ;

iterative_statements
              : WHILE '(' simple_expression ')'
{if($3!=1){printf("Condition checking is not of type
int\n");exit(0);}} statement
              | FOR '(' expression ';' simple_expression ';'
{if($5!=1){printf("Condition checking is not of type
int\n");exit(0);}} expression ')'
              | DO statement WHILE '(' simple_expression
')'{if($5!=1){printf("Condition checking is not of type
int\n");exit(0);}} ';';
return_statement
              : RETURN ';' {if(strcmp(currfunctype,"void"))
{printf("Returning void of a non-void function\n");
exit(0);}}
              | RETURN expression ';' {
if(!strcmp(currfunctype, "void"))
                                                    {

yyerror("Function is void");
                                                    }


if((currfunctype[0]=='i' || currfunctype[0]=='c') && $2!=1)
                                                    {

printf("Expression doesn't match return type of function\n");
exit(0);
                                                    }

                                          };
```

```
break_statement
              : BREAK ';' ;

string_initilization
              : assignment_operator string_constant
{insV();} ;

array_initialization
              : assignment_operator '{'
array_int_declarations '}';

array_int_declarations
              : integer_constant
array_int_declarations_breakup;

array_int_declarations_breakup
              : ',' array_int_declarations
              | ;

expression
              : mutable assignment_operator expression
{

                    if($1==1 && $3==1)

                    {

$$=1;

}

else

{$$=-1; printf("Type mismatch\n"); exit(0);}

}
```

```
              | mutable addition_assignment_operator
expression     {

                    if($1==1 && $3==1)

$$=1;

else

{$$=-1; printf("Type mismatch\n"); exit(0);}

}
              | mutable subtraction_assignment_operator
expression  {

                    if($1==1 && $3==1)

$$=1;

else

{$$=-1; printf("Type mismatch\n"); exit(0);}

}
              | mutable multiplication_assignment_operator
expression {

                    if($1==1 && $3==1)

$$=1;

else

{$$=-1; printf("Type mismatch\n"); exit(0);}

}
              | mutable division_assignment_operator
```

```
expression        {

                    if($1==1 && $3==1)

$$=1;

else

{$$=-1; printf("Type mismatch\n"); exit(0);}

}
              | mutable modulo_assignment_operator
expression        {

                    if($1==1 && $3==1)

$$=1;

else

{$$=-1; printf("Type mismatch\n"); exit(0);}

}
              | mutable increment_operator
              {if($1 == 1) $$=1; else $$=-1;}
              | mutable decrement_operator
              {if($1 == 1) $$=1; else $$=-1;}
              | simple_expression {if($1 == 1) $$=1; else
$$=-1;} ;


simple_expression
              : simple_expression OR_operator and_expression
{if($1 == 1 && $3==1) $$=1; else $$=-1;}
              | and_expression {if($1 == 1) $$=1; else
$$=-1;};
```

```
and_expression
              : and_expression AND_operator
unary_relation_expression {if($1 == 1 && $3==1) $$=1; else
$$=-1;}
                |unary_relation_expression {if($1 == 1)
$$=1; else $$=-1;} ;


unary_relation_expression
              : exclamation_operator
unary_relation_expression {if($2==1) $$=1; else $$=-1;}
              | regular_expression {if($1 == 1) $$=1; else
$$=-1;} ;

regular_expression
              : regular_expression relational_operators
sum_expression {if($1 == 1 && $3==1) $$=1; else $$=-1;}
                | sum_expression {if($1 == 1) $$=1; else
$$=-1;} ;

relational_operators
              : greaterthan_assignment_operator |
lessthan_assignment_operator | greaterthan_operator
              | lessthan_operator | equality_operator |
inequality_operator ;

sum_expression
              : sum_expression sum_operators term  {if($1 ==
1 && $3==1) $$=1; else $$=-1;}
              | term {if($1 == 1) $$=1; else $$=-1;};

sum_operators
              : add_operator
              | subtract_operator ;

term
              : term MULOP factor {if($1 == 1 && $3==1)
```

```
$$=1; else $$=-1;}
              | factor {if($1 == 1) $$=1; else $$=-1;} ;

MULOP
              : multiplication_operator | division_operator
| modulo_operator ;

factor
              : immutable {if($1 == 1) $$=1; else $$=-1;}
              | mutable {if($1 == 1) $$=1; else $$=-1;} ;

mutable
              : identifier {
                             if(check_id_is_func(curid))
                             {printf("Function name used as
Identifier\n"); exit(8);}
                           if(!checkscope(curid))

{printf("%s\n",curid);printf("Undeclared\n");exit(0);}
                           if(!checkarray(curid))

{printf("%s\n",curid);printf("Array ID has no
subscript\n");exit(0);}
                           if(gettype(curid,0)=='i' ||
gettype(curid,1)== 'c')
                           $$ = 1;
                           else
                           $$ = -1;
                           }
              | array_identifier
{if(!checkscope(curid)){printf("%s\n",curid);printf("Undeclar
ed\n");exit(0);}} '[' expression ']'
                                {if(gettype(curid,0)=='i'
|| gettype(curid,1)== 'c')
                                $$ = 1;
                                else
                                $$ = -1;
```

```
                                    };

immutable
              : '(' expression ')' {if($2==1) $$=1; else
$$=-1;}
              | call
              | constant {if($1==1) $$=1; else $$=-1;};

call
              : identifier '('{
                            if(!check_declaration(curid,
"Function"))
                            { printf("Function not
declared"); exit(0);}
                            insertSTF(curid);
                              strcpy(currfunccall,curid);
                            } arguments ')'
                              {
if(strcmp(currfunccall,"printf"))
                                    {

if(getSTparamscount(currfunccall)!=call_params_count)
                                          {
                                                yyerror("Number of
arguments in function call doesn't match number of
parameters");
                                                //printf("Number
of arguments in function call %s doesn't match number of
parameters\n", currfunccall);
                                                exit(8);
                                          }
                                    }
                                  };

arguments
              : arguments_list | ;
```

```
arguments_list
              : expression { call_params_count++; } A ;

A
              : ',' expression { call_params_count++; } A
              | ;

constant
              : integer_constant      {  insV(); $$=1; }
              | string_constant  {  insV(); $$=-1;}
              | float_constant   {  insV(); }
              | character_constant{  insV();$$=1; };

%%

extern FILE *yyin;
extern int yylineno;
extern char *yytext;
void insertSTtype(char *,char *);
void insertSTvalue(char *, char *);
void incertCT(char *, char *);
void printST();
void printCT();

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

     if(flag == 0)
     {
          printf(ANSI_COLOR_GREEN "Status: Parsing Complete -
Valid" ANSI_COLOR_RESET "\n");
          printf("%30s" ANSI_COLOR_CYAN "SYMBOL TABLE"
ANSI_COLOR_RESET "\n", " ");
          printf("%30s %s\n", " ", "------------");
          printST();
```

```
          printf("\n\n%30s" ANSI_COLOR_CYAN "CONSTANT TABLE"
ANSI_COLOR_RESET "\n", " ");
          printf("%30s %s\n", " ", "--------------");
          printCT();
     }
}

void yyerror(char *s)
{
     printf(ANSI_COLOR_RED "%d %s %s\n", yylineno, s,
yytext);
     flag=1;
     printf(ANSI_COLOR_RED "Status: Parsing Failed -
Invalid\n" ANSI_COLOR_RESET);
     exit(7);
}

void ins()
{
     insertSTtype(curid,curtype);
}

void insV()
{
     insertSTvalue(curid,curval);
}

int yywrap()
{
     return 1;
}
```

Explanation

The lex code is detecting the tokens from the source code and returning the corresponding token to the parser. In phase 1 we were just printing the token and now we are returning the token so that the parser uses it for further computation. We are using the symbol table and constant table of the previous phase only. We added functions like insertSTnest(),insertSTparamscount(),checkscope(), deletedata(), duplicate() etc., in order to check the semantics.In the production rules of the grammar semantic actions are written and these are performed by the functions listed above.

Declaration Section

In this section we have included all the necessary header files,function declaration and flag that was needed in the code.

Between declaration and rules section we have listed all the tokens which are returned by the lexer according to the precedence order. We also declared the operators here according to their associativity and precedence.This ensures the grammar we are giving to the parser is unambiguous as LALR(1) parser cannot work with ambiguous grammar.

Rules Section

In this section production rules for the entire C language are written. The grammar production does the syntax analysis of the source code. When a complete statement with proper syntax is matched by the parser. Along with rules semantic actions associated with the rules are also written and corresponding functions are called to do the necessary actions.

C-Program Section

In this section the parser links the extern functions,variables declared in the lexer, external files generated by the lexer etc. The main function takes the input source code file and prints the final symbol table.

Test Cases:

Valid Test Cases:

```
================================  Running TestCase 1  ================================
Status: Parsing Complete - Valid
                          SYMBOL TABLE
-------------------------------------------------------------------------------
    SYMBOL |          CLASS |       TYPE |      VALUE |    LINE NO | PARAMS COUNT |
-------------------------------------------------------------------------------
         a | Array Identifier |        int |            |         15 |         -1 |
         b |      Identifier |        int |            |          3 |         -1 |
         i |      Identifier |        int |         10 |         12 |         -1 |
         n |      Identifier |        int |            |         12 |         -1 |
         x |      Identifier |        int |            |          5 |         -1 |
         x |      Identifier |        int |          3 |         14 |         -1 |
         x |      Identifier |        int |         10 |         18 |         -1 |
       for |         Keyword |            |            |         16 |         -1 |
      char |         Keyword |            |            |         13 |         -1 |
        ch |      Identifier |       char |            |         13 |         -1 |
    return |         Keyword |            |            |          6 |         -1 |
        if |         Keyword |            |            |         17 |         -1 |
       int |         Keyword |            |            |          3 |         -1 |
      main |        Function |       void |            |         10 |         -1 |
    myfunc |        Function |        int |            |          3 |          1 |
     while |         Keyword |            |            |         19 |         -1 |
      void |         Keyword |            |            |         10 |         -1 |


                          CONSTANT TABLE
-------------------------------------------------------------------------------
      NAME |           TYPE
-------------------------------------------------------------------------------
        10 | Number Constant
         0 | Number Constant
         3 | Number Constant
```

```
====================================  Running TestCase 2  ====================================
Status: Parsing Complete - Valid
                              SYMBOL TABLE
-------------------------------------------------------------------------------------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO | PARAMS COUNT |
-------------------------------------------------------------------------------------------
         a |      Identifier |        int |          0 |          5 |         -1 |
         b |      Identifier |        int |          0 |         17 |         -1 |
       int |         Keyword |            |            |          3 |         -1 |
      main |        Function |        int |            |          3 |         -1 |
    printf |        Function |            |            |          8 |         -1 |
     while |         Keyword |            |            |          6 |         -1 |


                              CONSTANT TABLE
-------------------------------------------------------------------------------------------
      NAME |            TYPE
-------------------------------------------------------------------------------------------
"Hello world" | String Constant
      "%d" | String Constant
         0 | Number Constant
         4 | Number Constant
         5 | Number Constant
```

```
================================  Running TestCase 3  ================================
Wrong array size


================================  Running TestCase 4  ================================
Status: Parsing Complete - Valid
                          SYMBOL TABLE
-------------------------------------------------------------------------------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO | PARAMS COUNT |
-------------------------------------------------------------------------------------
         a |      Identifier |        int |         29 |          6 |         -1 |
         b |      Identifier |        int |            |          6 |         -1 |
         c |      Identifier |       char |            |          7 |         -1 |
         x |      Identifier |        int |            |         14 |         -1 |
       for |         Keyword |            |            |          8 |         -1 |
      char |         Keyword |            |            |          7 |         -1 |
        if |         Keyword |            |            |         10 |         -1 |
       int |         Keyword |            |            |          4 |         -1 |
      main |        Function |        int |            |          4 |         -1 |
      var1 |      Identifier |        int |            |         18 |         -1 |
      var2 |      Identifier |       char |            |         19 |         -1 |
    printf |        Function |            |            |         11 |         -1 |

                          CONSTANT TABLE
-------------------------------------------------------------------------------------
      NAME |            TYPE
-------------------------------------------------------------------------------------
"Hello World" | String Constant
      "%d" | String Constant
        15 | Number Constant
        29 | Number Constant
         0 | Number Constant
```

Invalid Test Cases:

```
================================  Running TestCase 5  ================================
========
Function not declared

================================  Running TestCase 6  ================================
========
6 Function is void ;
Status: Parsing Failed - Invalid


================================  Running TestCase 7  ================================
========
12 Number of arguments in function call doesn't match number of parameters )
Status: Parsing Failed - Invalid


================================  Running TestCase 8  ================================
========
Type mismatch
```