**Semantic Analysis**

**Introduction:**

This report contains the details of the tasks finished in the Semantic Phase of the C compiler. We have developed a Parser for C language that uses the C lexer to parse the given C input file. In the previous phase, we checked if the given input code matches the language defined in the parser. We used a lexer to convert the input code into a stream of tokens provided to the parser. Parser matches the stream with the defined productions of the language. We used look-ahead for checking errors in comments and some other lexical errors. But lexical analyzer 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, the 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 a parse tree, whereas that of the semantic phase is annotated parse tree. Semantic analysis is done by modifications in the parser code only. Tasks performed in Semantic Analysis are:

1. Type Checking – Data types must be used in a manner that is consistent with their definition
2. Label Checking – Labels referenced in a program must exist.
3. Array Bounds Checking – The subscript should be adequately defined when declaring an array.

We have mentioned some of the semantic 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
3. Check if a variable is undeclared globally.
4. Check if a variable is visible in the current scope.
5. Reserved identifier misuse.
6. Function name and variable name cannot be the same.
7. Declaration of a keyword as a variable name.
8. Multiple declarations of a variable in scope.
9. Accessing an out-of-scope variable.
10. Actual and formal parameter mismatches

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

**Example Code Snippet:**

*void* insertSTnest(*char* \**s*, *int* *nest*)

    {

        if(lookupST(s) && ST[lookupST(s)].nestval != 9999)

        {

*int* pos = 0;

*int* value = hash(s);

            for (*int* i = value + 1 ; i!=value ; i = (i+1)%1001)

            {

                if(ST[i].length == 0)

                {

                    pos = i;

                    break;

                }

            }

            strcpy(ST[pos].name,s);

            strcpy(ST[pos].class,"Identifier");

            ST[pos].length = strlen(s);

            ST[pos].nestval = nest;

            ST[pos].params\_count = -1;

            ST[pos].lineno = yylineno;

        }

        else

        {

            for(*int* i = 0 ; i < 1001 ; i++)

            {

                if(strcmp(ST[i].name,s)==0 )

                {

                    ST[i].nestval = nest;

                }

            }

        }

    }

*void* insertSTparamscount(*char* \**s*, *int* *count*)

    {

        for(*int* i = 0 ; i < 1001 ; i++)

        {

            if(strcmp(ST[i].name,s)==0 )

            {

                ST[i].params\_count = count;

            }

        }

    }

**Declaration Section**:   
This section has included all the necessary header files, function declarations, and flags needed in the code. Between the declaration and rules section, we have listed all the tokens 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.

**Example Code Snippet:**

%{

*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;

    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;

%}

**Rules Section**:  
In this section, production rules for the entire C language are written. The grammar productions do the syntax analysis of the source code. Along with rules, semantic actions associated with the rules are also written and corresponding functions are called to do the necessary actions.

**Example Code Snippet:**

*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;} ;

**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.

**Example Code Snippet**

*int* main(*int* *argc* , *char* \*\**argv*)

{

    yyin = fopen(argv[1], "r");

    yyparse();

    if(flag == 0)

    {

        printf("Status: Parsing Complete - Valid\n");

        printf("%30s SYMBOL TABLE\n", " ");

        printf("%30s %s\n", " ", "------------");

        printST();

        printf("\n\n%30s CONSTANT TABLE\n", " ");

        printf("%30s %s\n", " ", "--------------");

        printCT();

    }

}

**Test Cases**

**test1.c** (without error)

#include<stdio.h>

int myfunc(int b){

    int x;

    return x;

}

void main(){

    int n,i;

    char ch;//Character Datatype

    int x;

    int a[10];

    for (i=0;i<10;i++){

        if(i<10){

            int x;

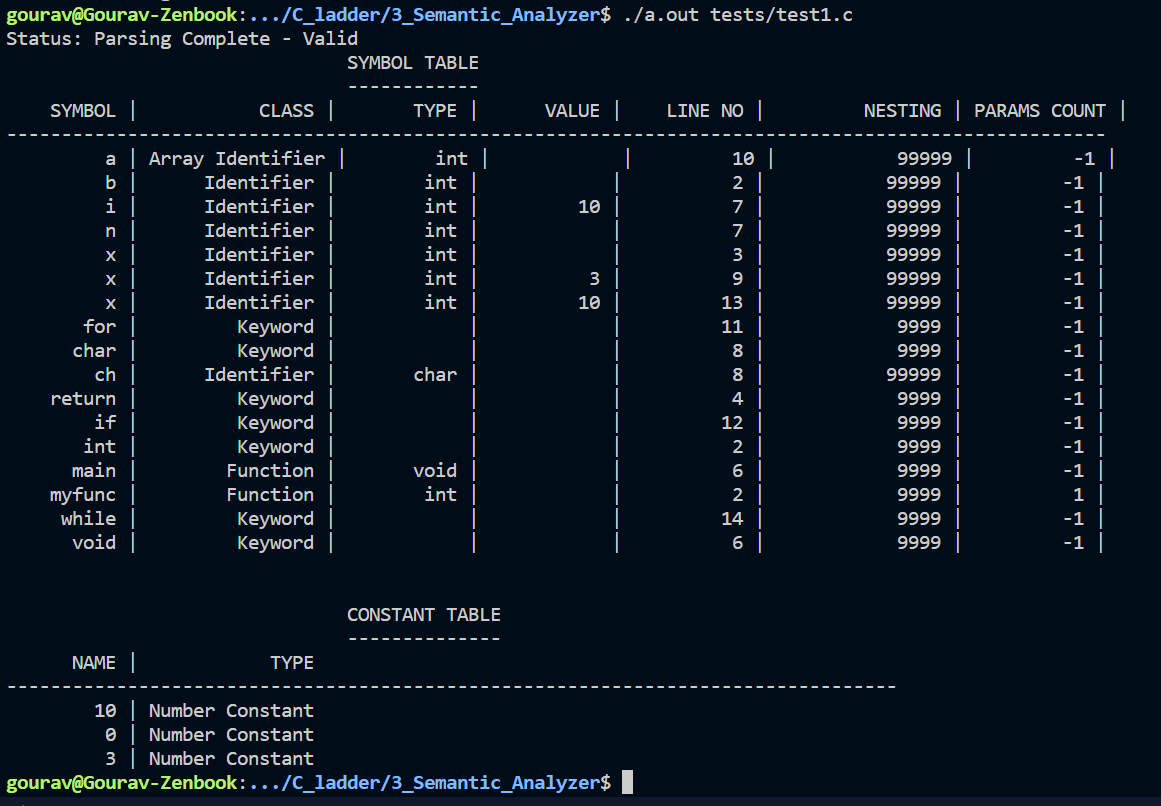
            while(x<10){

                x++;

            }}}

   x=3;

}



**test2.c** (without error)

#include<stdio.h>

int main(){

    int a = 5;

    while(a>0){

        printf("Hello world");

        a--;

    }

    a=4;

    while(a>0){

        printf("%d",a);

        a--;

        int b;

        b= 4;

        while(b>0){

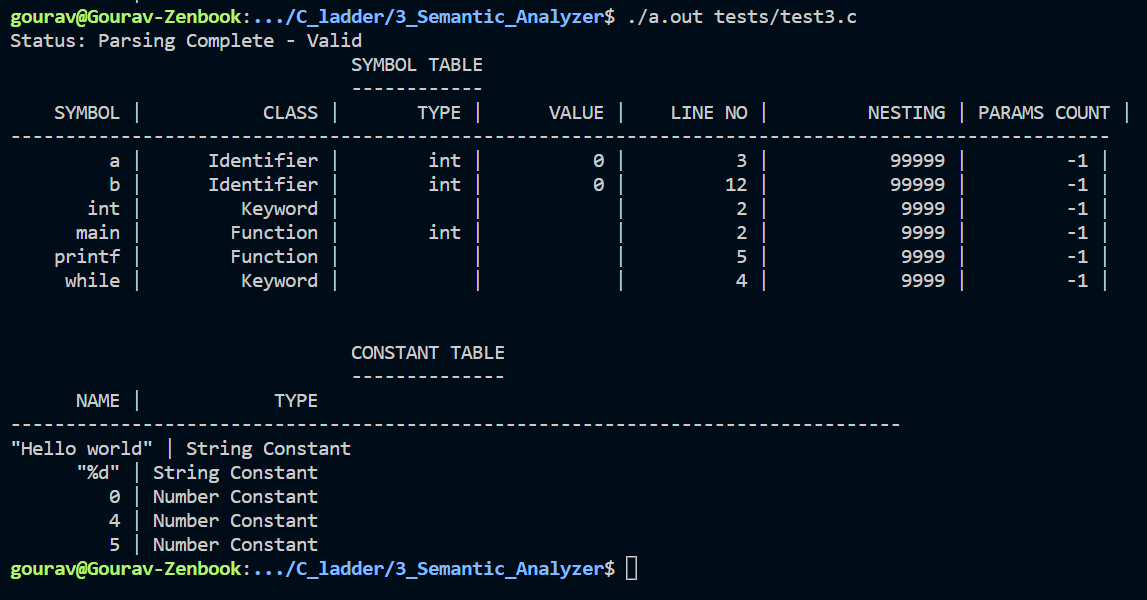
            printf("%d", a\*b);

            b--;

        }

    }

}



**test12.c** (with error)

 #include <stdio.h>

void func(int a, int b){

    return a;

}

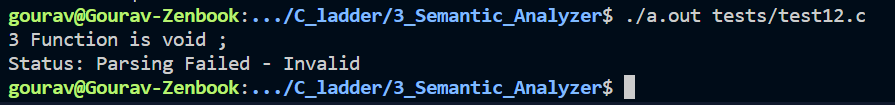
int main(){

    int z = 5;

    func(5,z,z);

    printf("wow\n");

}



**Code Optimization**

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e., CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following objectives:

* The optimization must be correct, it must not, in any way, change the meaning of the program.
* Optimization should increase the speed and performance of the program.
* The compilation time must be kept reasonable.
* The optimization process should not delay the overall compiling process.

**Types of Code Optimization –**The optimization process can be broadly classified into two types:

1. **Machine Independent Optimization –** This code optimization phase attempts to improve the **intermediate code** to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.
2. **Machine Dependent Optimization –** Machine-dependent optimization is done after the **target code** has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum **advantage** of the memory hierarchy.

**Phases of Optimization**  
There are generally two phases of optimization:

* **Global Optimization:**  
  Transformations are applied to large program segments that includes functions,procedures and loops.
* **Local Optimization:**  
  Transformations are applied to small blocks of statements.The local optimization is done prior to global optimization.