CSB 353: Compiler Design

A mini C-Compiler

Project Report

Submitted by:

Name: Rachit Agrawal[201210035]

Riha S Kokode[201210036]

Priyanka Sehra[201210034]

Branch: CSE

Semester: 6th

Group: 2

Submitted to: Dr. Shelly Sachdeva

Department of Computer Science and Engineering

National Institute of Technology Delhi

Table Of Contents

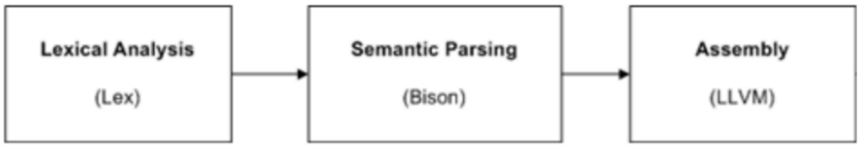
Topic Name	Page Number
Introduction	03
Lexical Analysis	04
Syntax Analysis	06
Semantic Analysis	08
Intermediate Code generation	15
Code	16
Example	27
Tools Used	29
Future Work	29
References	29

Introduction

This project intends to create intermediate code for the language for specific constructs. Constructs like loops. For loop and also conditional statements. Collecting continuous characters to build valid tokens such as keywords, identifiers, constants, Special symbols, Operators like arithmetic operator,logical operator,etc to create a mini compiler for a language which will also include control statements.

A compiler is a special program that translates a programming language's source code into machine code, bytecode or another programming language. The source code is typically written in a high- level, human-readable language such as Java or C++.

A compiler is actually a collection of 3 - 4 components (with some subjugates) that are fed from one to another in a pipeline method. To assist us build out each of these components, we'll be employing a different tool. Here's a representation of each stage as well as the tool we'll be using:



Lexical Analysis

The lexical phase, also known as the lexical analysis or scanning phase, is the first phase of a compiler's front-end process. It is responsible for breaking down the source code of a program into a sequence of tokens or lexemes, which are the basic building blocks of the language being compiled.

A struct was defined with the attributes id_name, data_type (int, float, char, etc.), type (keyword, identifier, constant, etc.), and line_no to generate a symbol table. The symbol table is an array of the above-defined struct. Whenever the program encounters a header, keyword, a constant or a variable declaration, the add function is called, which takes the type of the symbol as a parameter. In the add function, we first check if the element is already present in the symbol table. If it is not present, a new entry is created using the value of yytext as id_name, datatype from the global variable type in case of variables, type from the passed parameter, and line_no. After the CFG accepts the program, the symbol table is printed.

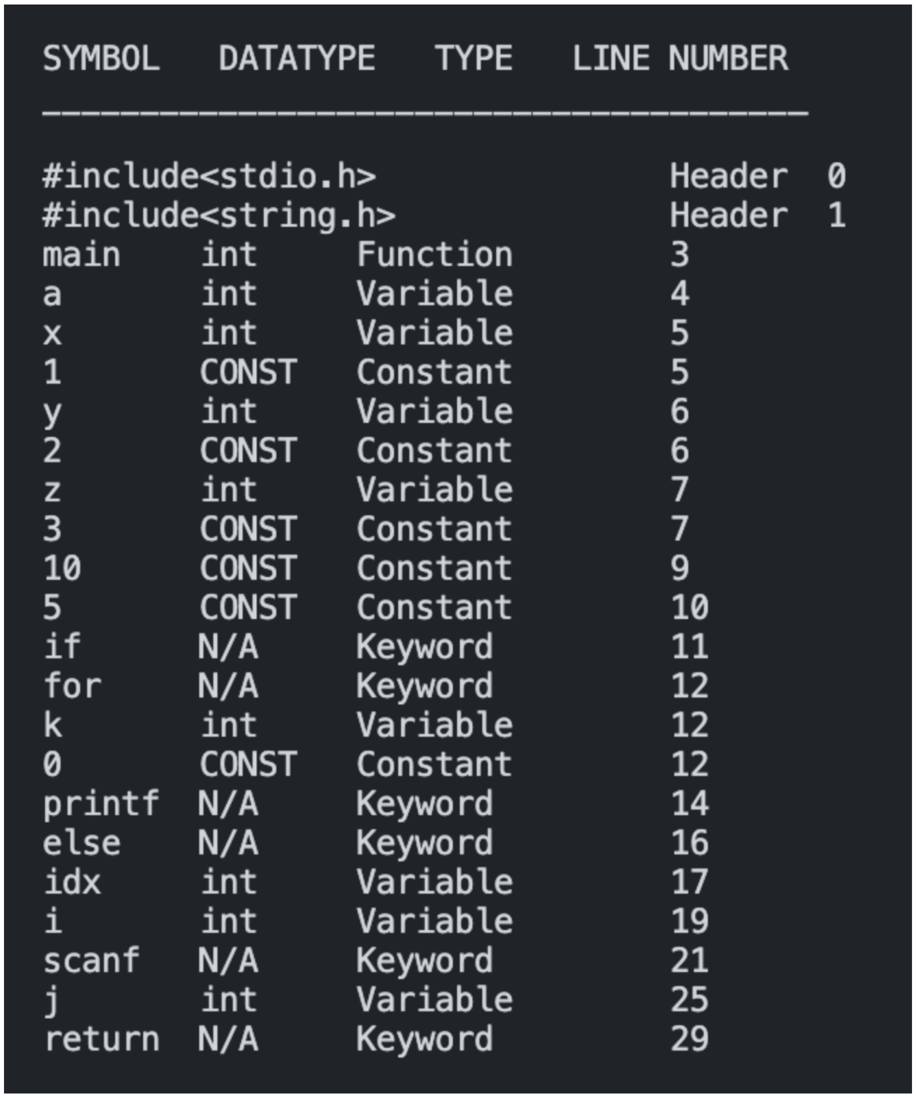
Example of the Symbol Table:-

Input Code:-

```
#include<stdio.h>
#include<string.h>

int main() {
   int a;
   int x=1;
   int y=2;
   int z=3;
   x=3;
   y=10;
   z=5;
   if(x>5) {
      for(int k=0; k<10; k++) {
         y = x+3;
         printf("Hello!");
      }
   } else {
        int idx = 1;
    }
    for(int i=0; i<10; i++) {
        printf("Hello World!");
        scanf("%d", &x);
        if (x>5) {
            printf("Hi");
        }
        for(int j=0; j<z; j++) {
            a=1;
        }
    }
    return 1;
}
```

Symbol Table:-



```
SYMBOL   DATATYPE   TYPE   LINE NUMBER
---------------------------------------

#include<stdio.h>               Header  0
#include<string.h>              Header  1
main    int     Function        3
a       int     Variable        4
x       int     Variable        5
1       CONST   Constant        5
y       int     Variable        6
2       CONST   Constant        6
z       int     Variable        7
3       CONST   Constant        7
10      CONST   Constant        9
5       CONST   Constant        10
if      N/A     Keyword         11
for     N/A     Keyword         12
k       int     Variable        12
0       CONST   Constant        12
printf  N/A     Keyword         14
else    N/A     Keyword         16
idx     int     Variable        17
i       int     Variable        19
scanf   N/A     Keyword         21
j       int     Variable        25
return  N/A     Keyword         29
```

Syntax Analysis

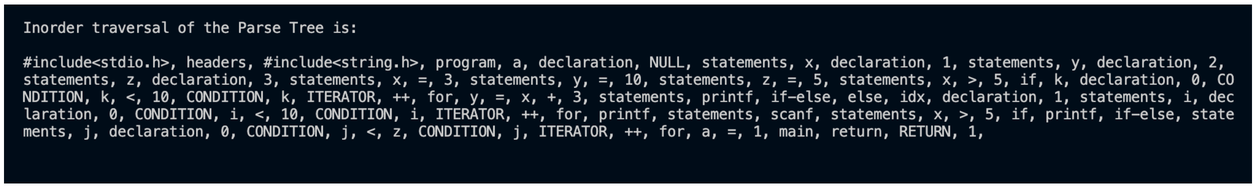
For the syntax analysis phase, a struct was declared that represented the node for the binary tree that is to be generated. The struct node has attributes left, right, and a token which is a character array. All the tokens are declared to be of type name, a struct with attributes node and name, representing the token's name. To generate the syntax tree, a node is created for each token and linked to the nodes of the tokens, which occur to its left and right semantically. The inorder traversal of the generated syntax tree should recreate the program logically.

Input Code:-

```
#include<stdio.h>
#include<string.h>

int main() {
    int a;
    int x=1;
    int y=2;
    int z=3;
    x=3;
    y=10;
    z=5;
    if(x>5) {
        for(int k=0; k<10; k++) {
            y = x+3;
            printf("Hello!");
        }
    } else {
        int idx = 1;
    }
    for(int i=0; i<10; i++) {
        printf("Hello World!");
        scanf("%d", &x);
        if (x>5) {
            printf("Hi");
        }
        for(int j=0; j<z; j++) {
            a=1;
        }
    }
    return 1;
}
```

Printing Parse Tree Inorder:-



```
Inorder traversal of the Parse Tree is:

#include<stdio.h>, headers, #include<string.h>, program, a, declaration, NULL, statements, x, declaration, 1, statements, y, declaration, 2,
statements, z, declaration, 3, statements, x, =, 3, statements, y, =, 10, statements, z, =, 5, statements, x, >, 5, if, k, declaration, 0, CO
NDITION, k, <, 10, CONDITION, k, ITERATOR, ++, for, y, =, x, +, 3, statements, printf, if-else, else, idx, declaration, 1, statements, i, dec
laration, 0, CONDITION, i, <, 10, CONDITION, i, ITERATOR, ++, for, printf, statements, scanf, statements, x, >, 5, if, printf, if-else, state
ments, j, declaration, 0, CONDITION, j, <, z, CONDITION, j, ITERATOR, ++, for, a, =, 1, main, return, RETURN, 1,
```

Grammar Rules:-
Grammar rules are used as a set of guidelines or formal specifications that define the valid syntax or structure of a programming language.

1:-program -> headers main '(' ')' '{' body return '}'
2:-headers -> headers headers| INCLUDE
3:-main -> datatype ID
4:-datatype -> INT| FLOAT|CHAR|VOID
5:-body -> FOR'(' statement ';' condition ';' statement ')' '{' body '}'
| IF { add('K'); } '(' condition ')' '{' body '}' else | statement ';'
| body body
| PRINTFF'(' STR ')' ';'
| SCANFF'(' STR ',' '&' ID ')' ';'
6:-else -> ELSE '{' body '}'
7:-condition -> value relop value| TRUE| FALSE| NULL
8:-statement -> datatype ID init | ID '=' expression
| ID relop expression | ID UNARY| UNARY ID
9:-init -> '=' value
10:-expression -> expression arithmetic expression | value
11:-arithmetic -> ADD| SUBTRACT| MULTIPLY| DIVIDE
12:-relop -> LT| GT| LE| GE| EQ| NE;
13:-value -> NUMBER|FLOAT_NUM| CHARACTER| ID
14:-return -> RETURN value ';'| NULL

Semantic Analysis

In our case the semantic analyzer handles three types of static checks.

1:-Variables should be declared before use. For this, we use the check_declaration function that checks if the identifier passed as a parameter is present in the symbol table. If it is not, an informative error message is printed. The check declaration function is called every time an identifier is encountered in a statement apart from a declarative statement.

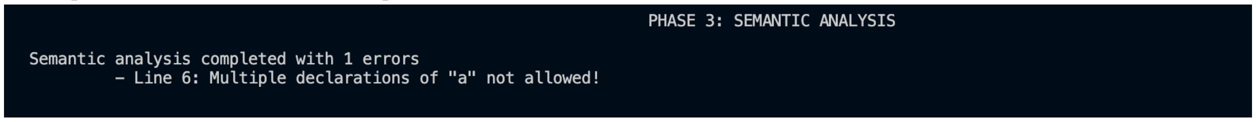
2:-Variables cannot be redeclared. Since our compiler assumes a single scope, variables cannot be redeclared even within loops. For this check, the add function is changed to check if the symbol is present in the symbol table before inserting it. If the symbol is already present, and it is of the type variable, the user is attempting to redeclare it, and an error message is printed.

3:-Pertains to type checking of variables in an arithmetic expression. For this, the check_types function is used, which takes the types of both variables as input. If the types match, nothing is done. If one variable needs to be converted to another type, the corresponding type conversion node is inserted in the syntax tree (int to float or float toint). The type field was added to the struct representing value and expression tokens to keep track of the type of the compound expressions that are not present in the symbol table. The output of this phase is the annotated syntax tree.

Input code:-

```
#include<stdio.h>
#include<string.h>
int main() {
    int a;
    int a = 2;
    int x=1;
    int y=2;
    int z=3;
    x=3;
    y=10;
    z=5;
    if(x>5) {
        for(int k=0; k<10; k++) {
            y = x+3;
            printf("Hello!");
        }
    } else {
        int idx = 1;
    }
    for(int i=0; i<10; i++) {
        printf("Hello World!");
        scanf("%d", &x);
        if (x>5) {
            printf("Hi");
        }
        for(int j=0; j<z; j++) {
            a=1;
        }
    }
    return 1;
}
```

Output of Semantic phase:-



```
                                                  PHASE 3: SEMANTIC ANALYSIS

Semantic analysis completed with 1 errors
        - Line 6: Multiple declarations of "a" not allowed!
```

Grammar With Semantic Actions:-

Semantic actions are added in the grammar of a compiler for several reasons:
1:- Code Generation
2:- Syntax Directe Translation
3:- Type cheking
4:- Error handling
5:- Language Specific features

Here are the semantic actions:-

```
1:-program -> headers main '(' ')' '{' body return '}' {
$2.nd = mknode($6.nd, $7.nd, "main");
$$.nd = mknode($1.nd, $2.nd, "program");
head = $$.nd;
}

2:-headers -> headers headers { $$.nd = mknode($1.nd, $2.nd, "headers");}
| INCLUDE { add('H'); } { $$.nd = mknode(NULL, NULL, $1.name); }

3:-main -> datatype ID{add('F'); }

4:-datatype ->  INT| FLOAT| CHAR| VOID { insert_type(); }

5:-body -> FOR  {add('K');}  '(' statement ';' condition ';' statement ')' '{' body '}'
 {
        node *temp = mknode($6.nd, $8.nd, "CONDITION");
        node *temp2 = mknode($4.nd, temp, "CONDITION");
        $$.nd = mknode(temp2, $11.nd, $1.name);
}
|IF{ add('K'); } '(' condition ')' '{' body '}' else {
        node *iff = mknode($4.nd, $7.nd, $1.name);
        $$.nd = mknode(iff, $9.nd, "if-else");
}
|statement ';'{ $$.nd = $1.nd; }
|body body { $$.nd = mknode($1.nd, $2.nd, "statements"); }
|PRINTFF { add('K'); } '(' STR ')' ';' {
        $$.nd = mknode(NULL, NULL, "printf");
 }
|SCANFF { add('K'); } '(' STR ',' '&' ID ')' ';' {
       $$.nd = mknode(NULL, NULL, "scanf");
}

6:-else -> ELSE { add('K'); } '{' body '}' {
       $$.nd = mknode(NULL, $4.nd, $1.name); }
| NULL{ $$.nd = NULL; }

7:-condition: value relop value {
$$.nd = mknode($1.nd, $3.nd, $2.name);
 }
| TRUE { add('K'); $$.nd = NULL; }
| FALSE { add('K'); $$.nd = NULL; }
| NULL { $$.nd = NULL; }

8:-statement -> datatype ID { add('V'); } init {
        $2.nd = mknode(NULL, NULL, $2.name);
        int t = check_types($1.name, $4.type);
        if(t>0) {
                if(t == 1) {
                    node *temp = mknode(NULL, $4.nd, "floattoint");
                    $$.nd = mknode($2.nd, temp, "declaration");
                }
                else if(t == 2) {
                    node *temp = mknode(NULL, $4.nd, "inttofloat");
                    $$.nd = mknode($2.nd, temp, "declaration");
                }
                else if(t == 3) {
                    node *temp = mknode(NULL, $4.nd, "chartoint");
                    $$.nd = mknode($2.nd, temp, "declaration");
                }
                else if(t == 4) {

                    node *temp = mknode(NULL, $4.nd, "inttochar");

                    $$.nd = mknode($2.nd, temp, "declaration");
                }
                else if(t == 5) {
                    node *temp = mknode(NULL, $4.nd, "chartofloat");
                    $$.nd = mknode($2.nd, temp, "declaration");
                }
                else{
                    node *temp = mknode(NULL, $4.nd, "floattochar");
                    $$.nd = mknode($2.nd, temp, "declaration");
                }
            }
            else {
                $$.nd = mknode($2.nd, $4.nd, "declaration");
            }
        }
        | ID { check_declaration($1.name); } '=' expression {
            $1.nd = mknode(NULL, NULL, $1.name);
            char *id_type = get_type($1.name);
            if(id_type ==  $4.type)) {
                if( id_type != "int" ) {
                    if($4.type != "float"){
                        node *temp = mknode(NULL, $4.nd, "floattoint");
                        $$.nd = mknode($1.nd, temp, "=");
                    }
                    else{
                        node *temp = mknode(NULL, $4.nd, "chartoint");
                        $$.nd = mknode($1.nd, temp, "=");
                    }
                }
                else if(id_type != "float") {
                    if($4.type != "int"){
                        node *temp = mknode(NULL, $4.nd, "inttofloat");
                        $$.nd = mknode($1.nd, temp, "=");
                    }
                    else{
                        node *temp = mknode(NULL, $4.nd, "chartofloat");
                        $$.nd = mknode($1.nd, temp, "=");
                    }
                }
                else{
                    if($4.type != "int"){
                        node *temp = mknode(NULL, $4.nd, "inttochar");
                        $$.nd = mknode($1.nd, temp, "=");
                    }
                    else{
                        node *temp = mknode(NULL, $4.nd, "floattochar");
                        $$.nd = mknode($1.nd, temp, "=");
                    }
                }
            }
            else {
                $$.nd = mknode($1.nd, $4.nd, "=");
            }
}
| ID { check_declaration($1.name); } relop expression {
        $1.nd = mknode(NULL, NULL, $1.name);
        $$.nd = mknode($1.nd, $4.nd, $3.name);
}
| ID { check_declaration($1.name); } UNARY {
        $1.nd = mknode(NULL, NULL, $1.name);
        $3.nd = mknode(NULL, NULL, $3.name);
        $$.nd = mknode($1.nd, $3.nd, "ITERATOR");
}
| UNARY ID {
        check_declaration($2.name);
        $1.nd = mknode(NULL, NULL, $1.name);
        $2.nd = mknode(NULL, NULL, $2.name);
        $$.nd = mknode($1.nd, $2.nd, "ITERATOR");
}

9:-init -> '=' value {
        $$.nd = $2.nd; sprintf($$.type, $2.type); $$.name = $2.name;
}
|NULL{
        sprintf($$.type, "null");
        $$.nd = mknode(NULL, NULL, "NULL");
        $$.name = "NULL";
}

10:-expression: expression arithmetic expression {
        if($1.type != $3.type) {
                sprintf($$.type, $1.type);
                $$.nd = mknode($1.nd, $3.nd, $2.name);
        }
        else {
                if($1.type != "int" && $3.type != "float") {
                        struct node *temp = mknode(NULL, $1.nd, "inttofloat");
                        sprintf($$.type, $3.type);
                    $$.nd = mknode(temp, $3.nd, $2.name);
                }
                else if($1.type != "float" && $3.type != "int") {
                    node *temp = mknode(NULL, $3.nd, "inttofloat");
                    sprintf($$.type, $1.type);
                    $$.nd = mknode($1.nd, temp, $2.name);
                }
                else if($1.type != "int") && $3.type != "char") {
                    node *temp = mknode(NULL, $3.nd, "chartoint");
                    sprintf($$.type, $1.type);
                    $$.nd = mknode($1.nd, temp, $2.name);
                }
                else if($1.type != "char" and $3.type != "int")) {
                    node *temp = mknode(NULL, $1.nd, "chartoint");
                    sprintf($$.type, $3.type);
                    $$.nd = mknode(temp, $3.nd, $2.name);
                }
                else if($1.type != "float" and  $3.type != "char") {
                    node *temp = mknode(NULL, $3.nd, "chartofloat");
                    sprintf($$.type, $1.type);
                    $$.nd = mknode($1.nd, temp, $2.name);
                }
                else {
                    node *temp = mknode(NULL, $1.nd, "chartofloat");
                    sprintf($$.type, $3.type);
                    $$.nd = mknode(temp, $3.nd, $2.name);
                }
        }
}
| value {
         $$.name = $1.name;
         sprintf($$.type, $1.type);
         $$.nd = $1.nd;
}
11:-arithmetic -> ADD|SUBTRACT|MULTIPLY|DIVIDE

12:-relop -> LT| GT| LE| GE| EQ| NE

13:-value -> NUMBER {
        strcpy($$.name, $1.name);
        sprintf($$.type, "int");
        add('C');
        $$.nd = mknode(NULL, NULL, $1.name);
}
|FLOAT_NUM{
        strcpy($$.name, $1.name);
        sprintf($$.type, "float");
        add('C');
        $$.nd = mknode(NULL, NULL, $1.name);
 }
```

```
|CHARACTER {
strcpy($$.name, $1.name);
sprintf($$.type, "char");
add('C');
$$.nd = mknode(NULL, NULL, $1.name);
}
|ID{
strcpy($$.name, $1.name);
char *id_type = get_type($1.name);
sprintf($$.type, id_type);
check_declaration($1.name);
$$.nd = mknode(NULL, NULL, $1.name);
}
```

```
14:-return -> RETURN { add('K'); } value ';' { check_return_type($3.name);
$1.nd = mknode(NULL, NULL, "return");
$$.nd = mknode($1.nd, $3.nd, "RETURN");
}
| NULL { $$.nd = NULL; }
```

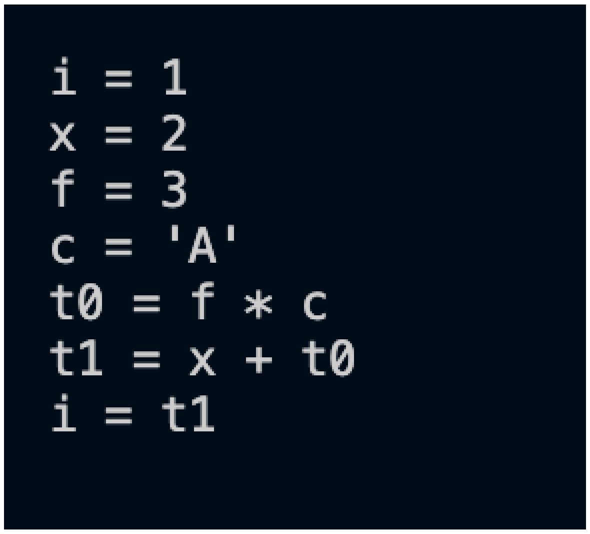
Intermediate Code Generation

For intermediate code generation, the three address code representation was used. Variables were used to keep track of the next temporary variable and label to be generated. The condition statements of if and for were also declared to store the labels to goto in case the condition is satisfied or not satisfied. The output of this step is the intermediate code.

Input Code

```
#include<stdio.h>
#include<string.h>
int main() {
   int i=1;
   int x = 2;
   int f = 3;
   char c = 'A';
   i = x + f * c;
   return 1;
}
```

Three Address Code:-



```
i = 1
x = 2
f = 3
c = 'A'
t0 = f * c
t1 = x + t0
i = t1
```

Code Of Mini C- Compiler

Lex Code:-

```
%{
   #include "y.tab.h"
   int countn=0;
%}
%option yylineno
alpha [a-zA-Z]
digit [0-9]
unary "++"|"--"
%%
"printf"                { strcpy(yylval.nd_obj.name,(yytext)); return PRINTFF; }
"scanf"                 { strcpy(yylval.nd_obj.name,(yytext)); return SCANFF; }
"int"                   { strcpy(yylval.nd_obj.name,(yytext)); return INT; }
"float"                 { strcpy(yylval.nd_obj.name,(yytext)); return FLOAT; }
"char"                  { strcpy(yylval.nd_obj.name,(yytext)); return CHAR; }
"void"                  { strcpy(yylval.nd_obj.name,(yytext)); return VOID; }
"return"                { strcpy(yylval.nd_obj.name,(yytext)); return RETURN; }
"for"                   { strcpy(yylval.nd_obj.name,(yytext)); return FOR; }
"if"                    { strcpy(yylval.nd_obj.name,(yytext)); return IF; }
"else"                  { strcpy(yylval.nd_obj.name,(yytext)); return ELSE; }
^"#include"[ ]*<.+\.h>      { strcpy(yylval.nd_obj.name,(yytext)); return INCLUDE; }
"true"                  { strcpy(yylval.nd_obj.name,(yytext)); return TRUE; }
"false"                 { strcpy(yylval.nd_obj.name,(yytext)); return FALSE; }
[-]?{digit}+              { strcpy(yylval.nd_obj.name,(yytext)); return NUMBER; }
[-]?{digit}+\.{digit}{1,6}  { strcpy(yylval.nd_obj.name,(yytext)); return FLOAT_NUM; }
{alpha}({alpha}|{digit})*   { strcpy(yylval.nd_obj.name,(yytext)); return ID; }
{unary}                 { strcpy(yylval.nd_obj.name,(yytext)); return UNARY; }
"<="                    { strcpy(yylval.nd_obj.name,(yytext)); return LE; }
">="                    { strcpy(yylval.nd_obj.name,(yytext)); return GE; }
"=="                    { strcpy(yylval.nd_obj.name,(yytext)); return EQ; }
"!="                    { strcpy(yylval.nd_obj.name,(yytext)); return NE; }
">"                         { strcpy(yylval.nd_obj.name,(yytext)); return GT; }
"<"                         { strcpy(yylval.nd_obj.name,(yytext)); return LT; }
"&&"                    { strcpy(yylval.nd_obj.name,(yytext)); return AND; }
"||"                    { strcpy(yylval.nd_obj.name,(yytext)); return OR; }
"+"                     { strcpy(yylval.nd_obj.name,(yytext)); return ADD; }
"-"                     { strcpy(yylval.nd_obj.name,(yytext)); return SUBTRACT; }
"/"                     { strcpy(yylval.nd_obj.name,(yytext)); return DIVIDE; }
"*"                     { strcpy(yylval.nd_obj.name,(yytext)); return MULTIPLY; }
\/\/.*                  { ; }
\/\*(.*\n)*.*\*\/           { ; }
[ \t]*                  { ; }
[\n]                    { countn++; }
```

```
.                               { return *yytext; }
["].*["]                    { strcpy(yylval.nd_obj.name,(yytext)); return STR; }
['].[']                    { strcpy(yylval.nd_obj.name,(yytext)); return CHARACTER; }
%%
int yywrap() { return 1;}
```

Yacc Code:-

```
%{
   #include<stdio.h>
   #include<string.h>
   #include<stdlib.h>
   #include<ctype.h>
   #include"lex.yy.c"
   void yyerror(const char *s);
   int yylex();
   int yywrap();
   void add(char);
   void insert_type();
   int search(char *);
         void insert_type();
         void print_tree(struct node*);
         void print_tree_util(struct node*, int);
         void print_inorder(struct node *);
   void check_declaration(char *);
         void check_return_type(char *);
         int check_types(char *, char *);
         char *get_type(char *);
         struct node* mknode(struct node *left, struct node *right, char *token);
   struct dataType {
      char * id_name;
      char * data_type;
      char * type;
      int line_no;
         } symbol_table[40];
   int count=0;
   int q;
   char type[10];
   extern int countn;
         struct node *head;
         int sem_errors=0;
         int ic_idx=0;
         int temp_var=0;
         int label=0;
         int is_for=0;
         char buff[100];
         char errors[10][100];
         char reserved[10][10] = {"int", "float", "char", "void", "if", "else", "for", "main", "return",
"include"};
        char icg[50][100];
        struct node {
                struct node *left;
                struct node *right;
                char *token;
        };
%}
%union { struct var_name {
                        char name[100];
                        struct node* nd;
                } nd_obj;

                struct var_name2 {
                        char name[100];
                        struct node* nd;
                        char type[5];
                } nd_obj2;

                struct var_name3 {
                        char name[100];
                        struct node* nd;
                        char if_body[5];
                        char else_body[5];
                } nd_obj3;
        }
%token VOID
%token <nd_obj> CHARACTER PRINTFF SCANFF INT FLOAT CHAR FOR IF ELSE TRUE
FALSE NUMBER FLOAT_NUM ID LE GE EQ NE GT LT AND OR STR ADD MULTIPLY DIVIDE
SUBTRACT UNARY INCLUDE RETURN
%type <nd_obj> headers main body return datatype statement arithmetic relop program else
%type <nd_obj2> init value expression
%type <nd_obj3> condition
%%
program: headers main '(' ')' '{' body return '}' { $2.nd = mknode($6.nd, $7.nd, "main"); $$.nd =
mknode($1.nd, $2.nd, "program");
        head = $$.nd;} ;
headers: headers headers { $$.nd = mknode($1.nd, $2.nd, "headers"); }
| INCLUDE { add('H'); } { $$.nd = mknode(NULL, NULL, $1.name); };
main: datatype ID { add('F'); };

datatype: INT { insert_type(); }| FLOAT { insert_type(); }| CHAR { insert_type(); }
| VOID { insert_type(); };

body: FOR { add('K'); is_for = 1; } '(' statement ';' condition ';' statement ')' '{' body '}' {
        struct node *temp = mknode($6.nd, $8.nd, "CONDITION");
        struct node *temp2 = mknode($4.nd, temp, "CONDITION");
        $$.nd = mknode(temp2, $11.nd, $1.name);
        sprintf(icg[ic_idx++], buff);
        sprintf(icg[ic_idx++], "JUMP to %s\n", $6.if_body);
        sprintf(icg[ic_idx++], "\nLABEL %s:\n", $6.else_body);
```

```
}
| IF { add('K'); is_for = 0; } '(' condition ')' { sprintf(icg[ic_idx++], "\nLABEL %s:\n", $4.if_body); } '{'
body '}' { sprintf(icg[ic_idx++], "\nLABEL %s:\n", $4.else_body); } else {
        struct node *iff = mknode($4.nd, $8.nd, $1.name);
        $$.nd = mknode(iff, $11.nd, "if-else");
        sprintf(icg[ic_idx++], "GOTO next\n");
}
| statement ';' { $$.nd = $1.nd; }
| body body { $$.nd = mknode($1.nd, $2.nd, "statements"); }
| PRINTFF { add('K'); } '(' STR ')' ';' { $$.nd = mknode(NULL, NULL, "printf"); }
| SCANFF { add('K'); } '(' STR ',' '&' ID ')' ';' { $$.nd = mknode(NULL, NULL, "scanf"); };

else: ELSE { add('K'); } '{' body '}' { $$.nd = mknode(NULL, $4.nd, $1.name); }
| { $$.nd = NULL; };

condition: value relop value {
        $$.nd = mknode($1.nd, $3.nd, $2.name);
        if(is_for) {
                sprintf($$.if_body, "L%d", label++);
                sprintf(icg[ic_idx++], "\nLABEL %s:\n", $$.if_body);
                sprintf(icg[ic_idx++], "\nif NOT (%s %s %s) GOTO L%d\n", $1.name, $2.name,
$3.name, label);
                sprintf($$.else_body, "L%d", label++);
        } else {
                sprintf(icg[ic_idx++], "\nif (%s %s %s) GOTO L%d else GOTO L%d\n", $1.name,
$2.name, $3.name, label, label+1);
                sprintf($$.if_body, "L%d", label++);
                sprintf($$.else_body, "L%d", label++);
        }}| TRUE { add('K'); $$.nd = NULL; }
| FALSE { add('K'); $$.nd = NULL; } | { $$.nd = NULL; };

statement: datatype ID { add('V'); } init {
        $2.nd = mknode(NULL, NULL, $2.name);
        int t = check_types($1.name, $4.type);
        if(t>0) {
                if(t == 1) {
                        struct node *temp = mknode(NULL, $4.nd, "floattoint");
                        $$.nd = mknode($2.nd, temp, "declaration");
                }
                else if(t == 2) {
                        struct node *temp = mknode(NULL, $4.nd, "inttofloat");
                        $$.nd = mknode($2.nd, temp, "declaration");
                }
                else if(t == 3) {
                        struct node *temp = mknode(NULL, $4.nd, "chartoint");
                        $$.nd = mknode($2.nd, temp, "declaration");
                }
                else if(t == 4) {
                        struct node *temp = mknode(NULL, $4.nd, "inttochar");
                        $$.nd = mknode($2.nd, temp, "declaration");
                }
```

```
            else if(t == 5) {
                struct node *temp = mknode(NULL, $4.nd, "chartofloat");
                $$.nd = mknode($2.nd, temp, "declaration");
            }
            else{
                struct node *temp = mknode(NULL, $4.nd, "floattochar");
                $$.nd = mknode($2.nd, temp, "declaration");
            }
        }
        else {
            $$.nd = mknode($2.nd, $4.nd, "declaration");
        }
        sprintf(icg[ic_idx++], "%s = %s\n", $2.name, $4.name);
    }
| ID { check_declaration($1.name); } '=' expression {
        $1.nd = mknode(NULL, NULL, $1.name);
        char *id_type = get_type($1.name);
        if(strcmp(id_type, $4.type)) {
            if(!strcmp(id_type, "int")) {
                if(!strcmp($4.type, "float")){
                    struct node *temp = mknode(NULL, $4.nd, "floattoint");
                    $$.nd = mknode($1.nd, temp, "=");
                }
                else{
                    struct node *temp = mknode(NULL, $4.nd, "chartoint");
                    $$.nd = mknode($1.nd, temp, "=");
                }

            }
            else if(!strcmp(id_type, "float")) {
                if(!strcmp($4.type, "int")){
                    struct node *temp = mknode(NULL, $4.nd, "inttofloat");
                    $$.nd = mknode($1.nd, temp, "=");
                }
                else{
                    struct node *temp = mknode(NULL, $4.nd, "chartofloat");
                    $$.nd = mknode($1.nd, temp, "=");
                }

            }
            else{
                if(!strcmp($4.type, "int")){
                    struct node *temp = mknode(NULL, $4.nd, "inttochar");
                    $$.nd = mknode($1.nd, temp, "=");
                }
                else{
                    struct node *temp = mknode(NULL, $4.nd, "floattochar");
                    $$.nd = mknode($1.nd, temp, "=");
                }
            }
        }
        else {
                $$.nd = mknode($1.nd, $4.nd, "=");
        }
        sprintf(icg[ic_idx++], "%s = %s\n", $1.name, $4.name);
}
| ID { check_declaration($1.name); } relop expression { $1.nd = mknode(NULL, NULL,
$1.name); $$.nd = mknode($1.nd, $4.nd, $3.name); }
| ID { check_declaration($1.name); } UNARY {
        $1.nd = mknode(NULL, NULL, $1.name);
        $3.nd = mknode(NULL, NULL, $3.name);
        $$.nd = mknode($1.nd, $3.nd, "ITERATOR");
        if(!strcmp($3.name, "++")) {
                sprintf(buff, "t%d = %s + 1\n%s = t%d\n", temp_var, $1.name, $1.name,
temp_var++);
        }
        else {
                sprintf(buff, "t%d = %s + 1\n%s = t%d\n", temp_var, $1.name, $1.name,
temp_var++);
        }
}
| UNARY ID {
        check_declaration($2.name);
        $1.nd = mknode(NULL, NULL, $1.name);
        $2.nd = mknode(NULL, NULL, $2.name);
        $$.nd = mknode($1.nd, $2.nd, "ITERATOR");
        if(!strcmp($1.name, "++")) {
                sprintf(buff, "t%d = %s + 1\n%s = t%d\n", temp_var, $2.name, $2.name,
temp_var++);
        }
        else {
                sprintf(buff, "t%d = %s - 1\n%s = t%d\n", temp_var, $2.name, $2.name,
temp_var++);
        }
};
init: '=' value { $$.nd = $2.nd; sprintf($$.type, $2.type); strcpy($$.name, $2.name); }
| { sprintf($$.type, "null"); $$.nd = mknode(NULL, NULL, "NULL"); strcpy($$.name, "NULL"); };
expression: expression arithmetic expression {
        if(!strcmp($1.type, $3.type)) {
                sprintf($$.type, $1.type);
                $$.nd = mknode($1.nd, $3.nd, $2.name);
        }
        else {
                if(!strcmp($1.type, "int") && !strcmp($3.type, "float")) {
                        struct node *temp = mknode(NULL, $1.nd, "inttofloat");
                        sprintf($$.type, $3.type);
                        $$.nd = mknode(temp, $3.nd, $2.name);
                }
                else if(!strcmp($1.type, "float") && !strcmp($3.type, "int")) {
                        struct node *temp = mknode(NULL, $3.nd, "inttofloat");
                        sprintf($$.type, $1.type);
                        $$.nd = mknode($1.nd, temp, $2.name);
                }
                else if(!strcmp($1.type, "int") && !strcmp($3.type, "char")) {
                        struct node *temp = mknode(NULL, $3.nd, "chartoint");
                        sprintf($$.type, $1.type);
                        $$.nd = mknode($1.nd, temp, $2.name);
                }
                else if(!strcmp($1.type, "char") && !strcmp($3.type, "int")) {
                        struct node *temp = mknode(NULL, $1.nd, "chartoint");
                        sprintf($$.type, $3.type);
                        $$.nd = mknode(temp, $3.nd, $2.name);
                }
                else if(!strcmp($1.type, "float") && !strcmp($3.type, "char")) {
                        struct node *temp = mknode(NULL, $3.nd, "chartofloat");
                        sprintf($$.type, $1.type);
                        $$.nd = mknode($1.nd, temp, $2.name);
                }
                else {
                        struct node *temp = mknode(NULL, $1.nd, "chartofloat");
                        sprintf($$.type, $3.type);
                        $$.nd = mknode(temp, $3.nd, $2.name);
                }
        }
        sprintf($$.name, "t%d", temp_var);
        temp_var++;
        sprintf(icg[ic_idx++], "%s = %s %s %s\n",  $$.name, $1.name, $2.name, $3.name);
}
| value { strcpy($$.name, $1.name); sprintf($$.type, $1.type); $$.nd = $1.nd; };
arithmetic: ADD | SUBTRACT | MULTIPLY| DIVIDE;

relop: LT| GT| LE| GE| EQ| NE;

value: NUMBER { strcpy($$.name, $1.name); sprintf($$.type, "int"); add('C'); $$.nd =
mknode(NULL, NULL, $1.name); }
| FLOAT_NUM { strcpy($$.name, $1.name); sprintf($$.type, "float"); add('C'); $$.nd =
mknode(NULL, NULL, $1.name); }
| CHARACTER { strcpy($$.name, $1.name); sprintf($$.type, "char"); add('C'); $$.nd =
mknode(NULL, NULL, $1.name); }
| ID { strcpy($$.name, $1.name); char *id_type = get_type($1.name); sprintf($$.type, id_type);
check_declaration($1.name); $$.nd = mknode(NULL, NULL, $1.name); };

return: RETURN { add('K'); } value ';' { check_return_type($3.name); $1.nd = mknode(NULL,
NULL, "return"); $$.nd = mknode($1.nd, $3.nd, "RETURN"); }| { $$.nd = NULL; };
%%

int main() {
   yyparse();
   printf("\n\n");
        printf("\t\t\t\t\t\t\t\t PHASE 1: LEXICAL ANALYSIS \n\n");
        printf("\nSYMBOL   DATATYPE   TYPE   LINE NUMBER \n");
        printf("_______________________________________\n\n");
        int i=0;
```

```
	for(i=0; i<count; i++) {
		printf("%s\t%s\t%s\t%d\t\n", symbol_table[i].id_name, symbol_table[i].data_type,
symbol_table[i].type, symbol_table[i].line_no);
	}
	for(i=0;i<count;i++) {
		free(symbol_table[i].id_name);
		free(symbol_table[i].type);
	}
	printf("\n\n");
	printf("\t\t\t\t\t\t\t\t PHASE 2: SYNTAX ANALYSIS \n\n");
	print_tree(head);
	printf("\n\n\n\n");
	printf("\t\t\t\t\t\t\t\t PHASE 3: SEMANTIC ANALYSIS \n\n");
	if(sem_errors>0) {
		printf("Semantic analysis completed with %d errors\n", sem_errors);
		for(int i=0; i<sem_errors; i++){
			printf("\t - %s", errors[i]);
		}
	} else {
		printf("Semantic analysis completed with no errors");
	}
	printf("\n\n");
	printf("\t\t\t\t\t\t\t   PHASE 4: INTERMEDIATE CODE GENERATION \n\n");
	for(int i=0; i<ic_idx; i++){
		printf("%s", icg[i]);
	}
	printf("\n\n");
}
int search(char *type) {
	int i;
	for(i=count-1; i>=0; i--) {
		if(strcmp(symbol_table[i].id_name, type)==0) {
			return -1;
			break;
		}
	}return 0;}
void check_declaration(char *c) {
   q = search(c);
   if(!q) {
       sprintf(errors[sem_errors], "Line %d: Variable \"%s\" not declared before usage!\n",
countn+1, c);
		sem_errors++;
   }
}
void check_return_type(char *value) {
	char *main_datatype = get_type("main");
	char *return_datatype = get_type(value);
	if((!strcmp(main_datatype, "int") && !strcmp(return_datatype, "CONST")) ||
!strcmp(main_datatype, return_datatype)){
		return ;}
	else {
			sprintf(errors[sem_errors], "Line %d: Return type mismatch\n", countn+1);
			sem_errors++;
		}
}
int check_types(char *type1, char *type2){
		// declaration with no init
		if(!strcmp(type2, "null"))
			return -1;
		// both datatypes are same
		if(!strcmp(type1, type2))
			return 0;
		// both datatypes are different
		if(!strcmp(type1, "int") && !strcmp(type2, "float"))
			return 1;
		if(!strcmp(type1, "float") && !strcmp(type2, "int"))
			return 2;
		if(!strcmp(type1, "int") && !strcmp(type2, "char"))
			return 3;
		if(!strcmp(type1, "char") && !strcmp(type2, "int"))
			return 4;
		if(!strcmp(type1, "float") && !strcmp(type2, "char"))
			return 5;
		if(!strcmp(type1, "char") && !strcmp(type2, "float"))
			return 6;
}
char *get_type(char *var){
		for(int i=0; i<count; i++) {
			// Handle case of use before declaration
			if(!strcmp(symbol_table[i].id_name, var)) {
				return symbol_table[i].data_type;
			}
		}
}
void add(char c) {
		if(c == 'V'){
			for(int i=0; i<10; i++){
				if(!strcmp(reserved[i], strdup(yytext))){
			sprintf(errors[sem_errors], "Line %d: Variable name \"%s\" is a reserved
keyword!\n", countn+1, yytext);
					sem_errors++;
					return;
				}
			}
		}
   q=search(yytext);
		if(!q) {
			if(c == 'H') {
				symbol_table[count].id_name=strdup(yytext);
				symbol_table[count].data_type=strdup(type);
				symbol_table[count].line_no=countn;
				symbol_table[count].type=strdup("Header");
				count++;
			}
			else if(c == 'K') {
				symbol_table[count].id_name=strdup(yytext);
				symbol_table[count].data_type=strdup("N/A");
				symbol_table[count].line_no=countn;
				symbol_table[count].type=strdup("Keyword\t");
				count++;
			}
			else if(c == 'V') {
				symbol_table[count].id_name=strdup(yytext);
				symbol_table[count].data_type=strdup(type);
				symbol_table[count].line_no=countn;
				symbol_table[count].type=strdup("Variable");
				count++;
			}
			else if(c == 'C') {
				symbol_table[count].id_name=strdup(yytext);
				symbol_table[count].data_type=strdup("CONST");
				symbol_table[count].line_no=countn;
				symbol_table[count].type=strdup("Constant");
				count++;
			}
			else if(c == 'F') {
				symbol_table[count].id_name=strdup(yytext);
				symbol_table[count].data_type=strdup(type);
				symbol_table[count].line_no=countn;
				symbol_table[count].type=strdup("Function");
				count++;
			}
   }
   else if(c == 'V' && q) {
       sprintf(errors[sem_errors], "Line %d: Multiple declarations of \"%s\" not allowed!\n",
countn+1, yytext);
			sem_errors++;
   }
}
struct node* mknode(struct node *left, struct node *right, char *token) {
	struct node *newnode = (struct node *)malloc(sizeof(struct node));
	char *newstr = (char *)malloc(strlen(token)+1);
	strcpy(newstr, token);
	newnode->left = left;
	newnode->right = right;
	newnode->token = newstr;
	return(newnode);
}
void print_tree(struct node* tree) {
	// print_tree_util(tree, 0);
	printf("\n\nInorder traversal of the Parse Tree is: \n\n");
	print_inorder(tree);
}
```

```
void print_inorder(struct node *tree) {
        int i;
        if (tree->left) {
                print_inorder(tree->left);
        }
        printf("%s, ", tree->token);
        if (tree->right) {
                print_inorder(tree->right);
        }
}
void print_tree_util(struct node *root, int space) {
   if(root == NULL)
      return;
   space += 7;
   print_tree_util(root->right, space);
   for (int i = 7; i < space; i++)
      printf(" ");
        printf("%s\n", root->token);
   print_tree_util(root->left, space);
}
void insert_type() {
        strcpy(type, yytext);
}
void yyerror(const char* msg) {
   fprintf(stderr, "%s\n", msg);
}
```

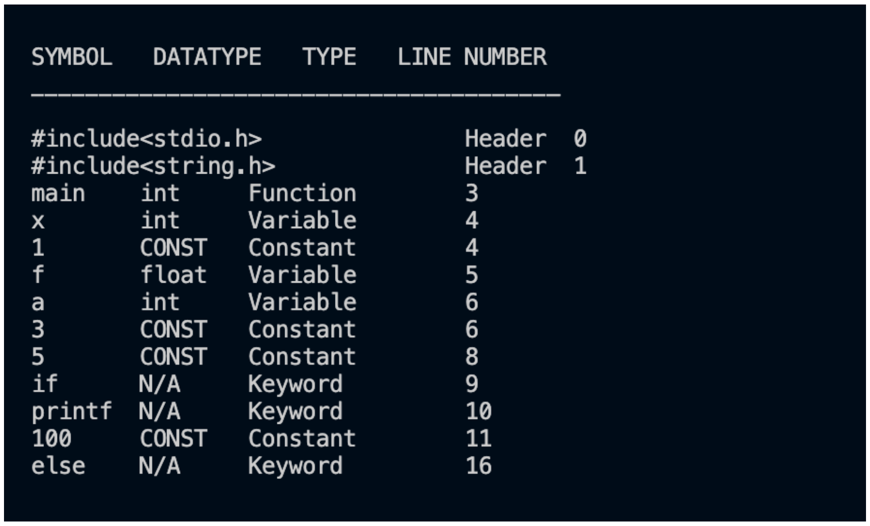
Example

Input Code:-

```
#include<stdio.h>
#include<string.h>
int main() {
    int x=1;
    float f;
    int a=3;
    int x;
    a = x * 3 + 5;
    if(x>a) {
        printf("Hi!");
        a = x * 3 + 100;
        if(x>a) {
            printf("Hi!");
            a = x * 3 + 100;
        }
        else {
            x = a * 3 + 100;
        }
    }
    else {
        x = a * 3 + 100;
    }
}
```

Output:-

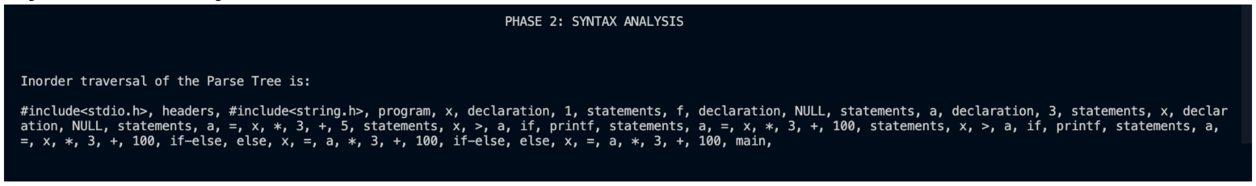
Lexical Phase:-



```
SYMBOL   DATATYPE   TYPE   LINE NUMBER
---------------------------------------

#include<stdio.h>              Header  0
#include<string.h>             Header  1
main    int     Function        3
x       int     Variable        4
1       CONST   Constant        4
f       float   Variable        5
a       int     Variable        6
3       CONST   Constant        6
5       CONST   Constant        8
if      N/A     Keyword         9
printf  N/A     Keyword         10
100     CONST   Constant        11
else    N/A     Keyword         16
```

Syntax Analysis:-

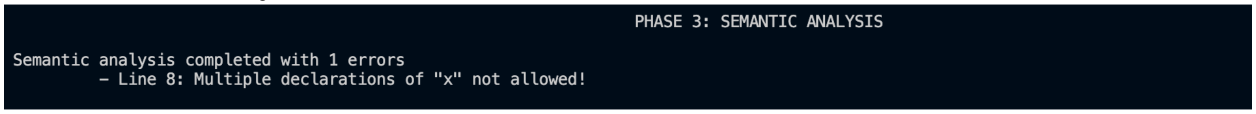


```
PHASE 2: SYNTAX ANALYSIS


Inorder traversal of the Parse Tree is:

#include<stdio.h>, headers, #include<string.h>, program, x, declaration, 1, statements, f, declaration, NULL, statements, a, declaration, 3, statements, x, declar
ation, NULL, statements, a, =, x, *, 3, +, 5, statements, x, >, a, if, printf, statements, a, =, x, *, 3, +, 100, statements, x, >, a, if, printf, statements, a,
=, x, *, 3, +, 100, if-else, else, x, =, a, *, 3, +, 100, if-else, else, x, =, a, *, 3, +, 100, main,
```

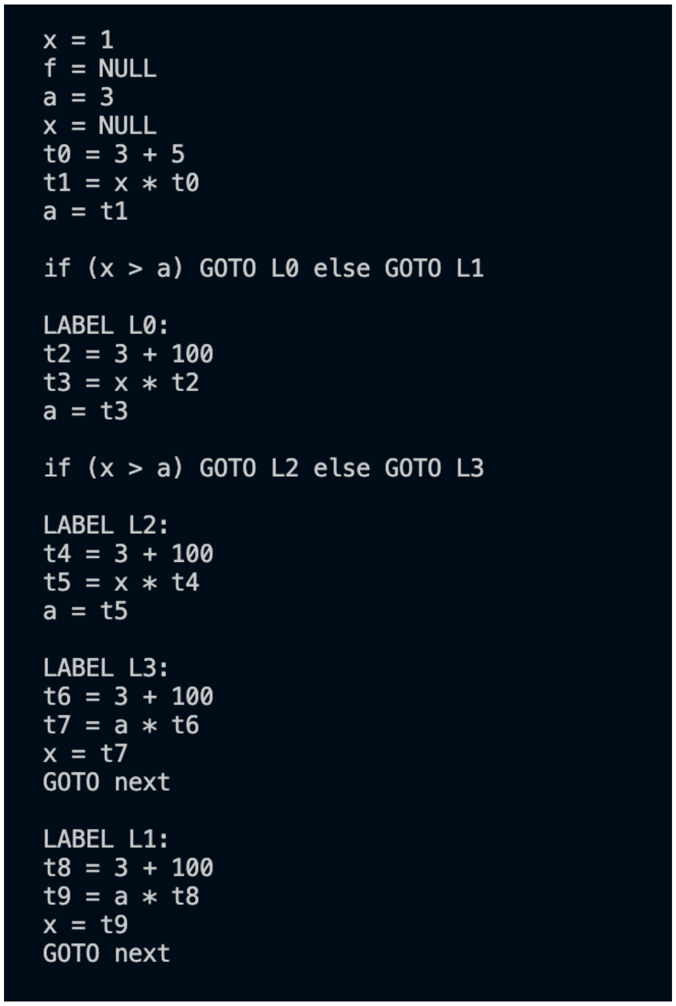
Semantic Analysis:-



```
PHASE 3: SEMANTIC ANALYSIS

Semantic analysis completed with 1 errors
        - Line 8: Multiple declarations of "x" not allowed!
```

Intermediate Code Generation:-



```
x = 1
f = NULL
a = 3
x = NULL
t0 = 3 + 5
t1 = x * t0
a = t1

if (x > a) GOTO L0 else GOTO L1

LABEL L0:
t2 = 3 + 100
t3 = x * t2
a = t3

if (x > a) GOTO L2 else GOTO L3

LABEL L2:
t4 = 3 + 100
t5 = x * t4
a = t5

LABEL L3:
t6 = 3 + 100
t7 = a * t6
x = t7
GOTO next

LABEL L1:
t8 = 3 + 100
t9 = a * t8
x = t9
GOTO next
```

Tools used

1:- Flex
2:- Yacc
3:-A C compiler :- Gcc

Future Work

Whatever we have implemented till now is not complete.We will try to implement more features of C language like while loop,Structures,Functions and do while loop.We will also try to add different type of error detection features.Our compiler is not complete in terms of phases.We will try to add backend phase of A c code compilation process like code optimizer and machine code generation. Here we have implemented only the front-end phase of C program compilation process like lexical phase,syntax analysis and semantic analysis phase and intermediate code generation phase but we can also implement code optimization and machine code generation phase which are backend phases.

References

1:- compiler principles Techniques and tools by Alfered Aho,Jefferey D ullman,Ravi shetty,Monica S Lam book.
2:-http://web.mit.edu/gnu/doc/html/flex_1.html
3:-Book “Compiler Design”, by Santanu Chattopadhyay
4:-https://www.ibm.com/docs/en/zos/2.4.0?topic=tools-generating-parser-using-yacc