**Semantic Analyzer for the C Language**



**National Institute of Technology Karnataka Suratkal**

**Date: 30th March 2017**

**GROUP MEMBERS:**

**V B VINEETH REDDY**

**14CO151**

**VIGNESH R**

**14CO153**

**MANOJ YADAV**

**14CO156**

## 

## **Abstract:**

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. The file that is created contains what are called the source statements. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements. When executing (running), the compiler first parses (or analyzes) all of the language statements syntactically one after the other and then, in one or more successive stages or "passes", builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code.

**Language: C**

The lexer shall recognize the following key words of c language.

|  |  |
| --- | --- |
| **Data Types:** | **I** int, float |
| **D Derived Data Types:** | struct |
| **Compounded Loops:** | for(){  } |
| **Operators and Relational Operators:** | +,-,/,\*,&&,||,<,>,<=,>=,!= etc. |
| **Separators:** | ;,{},(),[] etc. |

Lexer shall also recognize keywords like **return**, **void** and all the **identifiers** and **constants** in the code. Lexer shall recognize **functions** its return type and its parameters Lex/Flex tool is used to implement lexical analyzer for c

**Syntax analysis:**

Also known as parsing, it is the second phase of a compiler. We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

**Semantic Analysis:**

The semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interprets 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 analyzer here checks for following errors:

1. Type mismatch
2. Undeclared variable
3. Multiple declaration of variable in a scope.
4. Accessing an out of scope variable.

**Intermediate Code Generation:**

Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers. If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.

The second part of compiler, synthesis, is changed according to the target machine. It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

**Three-Address Code:**

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.

**Quadruples**

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result.

**Triples**

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

**Indirect Triples**

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizer to freely re-position the sub-expression to produce an optimized code.

**Declarations**

A variable or procedure has to be declared before it can be used. Declaration involves allocation of space in memory and entry of type and name in the symbol table. A program may be coded and designed keeping the target machine structure in mind, but it may not always be possible to accurately convert a source code to its target language.

Taking the whole program as a collection of procedures and sub-procedures, it becomes possible to declare all the names local to the procedure. Memory allocation is done in a consecutive manner and names are allocated to memory in the sequence they are declared in the program. We use offset variable and set it to zero {offset = 0} that denote the base address.

The source programming language and the target machine architecture may vary in the way names are stored, so relative addressing is used. While the first name is allocated memory starting from the memory location 0 {offset=0}, the next name declared later, should be allocated memory next to the first one.

### **Attribute Grammar**

An attribute grammar is a special form of context-free grammar where some additional information (attributes) is 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.

An 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.

### **YACC Script**

Yacc provides a general tool for imposing structure on the input to a computer program. The yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

*Definition section*

*%%*

*Rules section*

*%%*

*C code section*

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

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

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

### **C Code**

This section describes the input C program which is fed to the flex script in order to generate the lex file after taking all the rules mentioned in the account. Finally, a file called lex.yy.c is generated, which when executed recognizes the tokens present in the C program which was given as an input. The script also has an option to take standard input instead of taking input from a file.

## **Design of Programs:**

### **Flex Script:**

alpha [a-zA-Z]

digit [0-9]

%%

[ \t] ;

[ \n] { yylineno = yylineno + 1;}

int { yylval.iValue = INT; return INT; }

float { yylval.iValue = FLOAT; return FLOAT;}

void {yylval.iValue=VOID; return VOID;}

sizeof return SIZEOF;

typedef return TYPEDEF;

for return FOR;

while return WHILE;

if return IF;

else return ELSE;

return return RETURN;

printf return PRINTF;

scanf return SCANF;

struct return STRUCT;

^"#include ".+ return PREPROC;

{digit}+ {yylval.str=strdup(yytext); return NUM;}

({digit}\*\.{digit}+)|({digit}+\.{digit}\*) {yylval.str=strdup(yytext); return REAL;}

{alpha}({alpha}|{digit})\* { yylval.str = strdup(yytext); return ID ;}

"<=" {yylval.str=strdup(yytext); return LE;}

">=" { yylval.str=strdup(yytext);return GE;}

"==" {yylval.str=strdup(yytext); return EQ;}

"!=" {yylval.str=strdup(yytext); return NE;}

">" {yylval.str=strdup(yytext); return GT;}

"<" {yylval.str=strdup(yytext); return LT;}

"." return DOT;

"&&" return AND;

"||" return OR;

\/\/.\* ;

\/\\*(.\*\n)\*.\*\\*\/ ;

\"(\\.|[^"])\*\" {yylval.str=strdup(yytext); return STRING;}

. return yytext[0];

%%

### 

### **Yacc Script:**

%{

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include "symbol.cpp"

int g\_addr = 1000;

extern "C" {

int yylex();

void yyerror(char \*);

}

string str;

%}

%token<str> ID NUM REAL

%token PTR DOT SIZEOF

%token TYPEDEF STRUCT

%token<iValue> INT FLOAT VOID

%token IF ELSE WHILE RETURN FOR

%token PRINTF SCANF

%token STRING

%token PREPROC

%token ARRAY FUNCTION

%token MAIN

%token<str> GT LT LE GE NE EQ

%left GT LT LE GE NE EQ

%left AND OR

%right '='

%left '+' '-'

%left '\*' '/'

%type<iValue> Type

%type<str> statement array

%union {

int iValue; /\* integer value \*/

float realValue;

char \*str; /\* identifier name \*/

}

%%

start: Function start

| declare start

| PREPROC start

|

;

declare: Type statement ';' { if(redeclare($2))

{insert($2,$1,g\_addr); g\_addr+=4;}

else

{printf("Redecleration %s \n",$2);} }

| statement ';'

| ID ';' { printf("Undeclared Variable %s\n",$1);}

| function\_call ';'

| Type function\_call '{' statement\_list '}'

| array ';'

| Type array ';' { insert($2,ARRAY,g\_addr);

insert($2,$1,g\_addr); g\_addr+=4; }

| StructStmt ';'

| error

;

block: '{'{printf("Entered New scope\n");} statement\_list {printf("Exited New Scope\n");}'}'

;

statement\_list: statement\_list Stmt

|

;

Stmt: while

| declare

| For

| If

| print

| ';'

| RETURN ';'{printf("Wrong return type\n");}

| RETURN statement ';'

;

expression:

| expression LE expression

| expression GE expression

| expression NE expression

| expression EQ expression

| expression GT expression

| expression LT expression

| statement

| array

;

statement: ID{store($1);} '=' {store("=");} statement {assign();}

| ID ',' statement

| NUM ',' statement

| ID{store($1);} '+'{store("+");} statement {temp\_assign();}

| ID{store($1);} '-'{store("-");} statement {temp\_assign();}

| ID{store($1);} '\*'{store("\*");} statement {temp\_assign();}

| ID{store($1);} '/'{store("/");} statement {temp\_assign();}

| '(' statement ')'

| NUM {$$ = $1; store($1);}

| REAL {$$ = $1; store($1);}

| ID {$$=$1;store($1);}

| array

| array '=' statement {assign();}

| array '+'{store("+");} statement {temp\_assign();}

| array '-'{store("-");} statement {temp\_assign();}

| array '\*'{store("\*");} statement {temp\_assign();}

| array '/'{store("/");} statement {temp\_assign();}

| function\_call

;

function\_call : ID'('')' {func($1);}

| ID'('statement')' {func($1);}

;

array : ID'['statement']' {array($1,$3);}

;

Function: Type ID '(' ArgListOpt ')' block { insert($2,FUNCTION,g\_addr); insert($2,$1,g\_addr);g\_addr+=4; if($1==268) printf("Wrong return type\n"); }

;

ArgListOpt: ArgList

|

;

ArgList: ArgList ',' Arg

| Arg

;

Arg: Type ID

;

Type: INT

| FLOAT

| VOID

;

while: WHILE{w\_gen1();} '(' expression ')'{w\_gen2();} block{w\_gen3();}

;

For: FOR '(' expression ';'{f\_gen1();} expression ';'{f\_gen2();} expression ')'{f\_gen3();} block {f\_gen4();}

;

If : IF '(' expression ')'{if\_gen1();} block{if\_gen2();} ELSE block{if\_gen3();}

;

StructStmt : STRUCT ID '{' Type statement ';' '}' { insert($2,STRUCT,g\_addr); g\_addr+=4; }

;

print : PRINTF '(' expression ')' ';'

;

%%

#include "lex.yy.c"

#include <ctype.h>

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

{

FILE \*file;

file = fopen(argv[1], "r");

if (!file)

{

fprintf(stderr, "Could not open %s\n", argv[1]);

exit(1);

}

yyin = file;

if(!yyparse())

{

printf("\nParsing done\n");

printsym();

}

else

printf("\nParsing failed\n");

fclose(yyin);

return 0;

}

void yyerror(char \*s)

{

printf("%d : %s %s \n",yylineno,s,yytext);

}

**Symbol Table code :**

#include <bits/stdc++.h>

using namespace std;

struct symbolTable

{

/\* data \*/

string name;

vector<int> type;

int addr;

};

map<string,symbolTable> st;

char icg[100][100];

bool lookup(string name)

{

if(!st.count(name))

return true;

else

return false;

}

void insert(char \*n, int type, int addr)

{

string name(n);

struct symbolTable sym;

if(lookup(name))

{

sym.name = name;

sym.type.push\_back(type);

sym.addr = addr;

st[name] = sym;

}

else

{

sym = st[name];

if(sym.addr == addr)

{

sym.type.push\_back(type);

st[name] = sym;

}

}

return;

}

int top=-1,lno=0,ltop=0;

int lab\_tags[20];

int in = 0;

char c[100];

char temp[100]="t";

void printsym()

{

//cout<<"hello"<<endl;

map<string,symbolTable>::iterator it;

struct symbolTable s;

for(it=st.begin();it!=st.end();++it)

{

s=it->second;

cout<<s.addr<<" "<<s.name<<" ";

for(int i=0;i<s.type.size();++i)

{

switch(s.type[i])

{

case 266 : printf("Integer ");

break;

case 265 : printf("Structure ");

break;

case 258 : printf("Identifier ");

break;

case 279 : printf("Function ");

break;

case 278 : printf("Array ");

break;

case 267 : printf("Float ");

break;

case 268 : printf("Void ");

}

}

cout<<endl;

}

return;

}

bool check(string s1, string s2){

struct symbolTable symbolTable1, symbolTable2;

symbolTable1 = st[s1];

symbolTable2 = st[s2];

return symbolTable1.type[0] == symbolTable2.type[0];

}

bool redeclare(char\* name){

if(!st.count(name))

return true;

else

return false;

}

void store (char\* str)

{

strcpy(icg[++top],str);

}

void printStack()

{

int i;

printf("Stack\n");

for(i=1;i<top;i++)

{

printf("---%s---\n",icg[i]);

}

}

void assign()

{

//printStack();

printf("%s = %s\n",icg[top-2],icg[top]);

top= top-2;

}

void temp\_assign()

{

//printStack();

strcpy(temp,"t");

sprintf(c,"%d",in);

strcat(temp,c);

if(strcmp(icg[top],"")!=0)

{ printf("%s := %s '%s' %s\n",temp,icg[top-2],icg[top-1],icg[top]); top=top-2;}

else

{printf("%s := %s '%s' %s\n",temp,icg[top-3],icg[top-2],icg[top-1]);

top=top-3;}

strcpy(icg[top],temp);

in++;

}

void f\_gen1()

{

printf("F%d: ",lno++ );

}

void f\_gen2()

{

strcpy(temp,"t");

sprintf(c,"%d" ,in);

strcat(temp,c);

printf("%s = not %s\n",temp,icg[top] );

printf("if %s goto F%d\n",temp,lno );

in++;

lab\_tags[++ltop]=lno;

lno++;

printf("goto F%d\n",lno );

lab\_tags[++ltop]=lno;

printf("F%d: ",++lno );

}

void f\_gen3()

{

int x;

x=lab\_tags[ltop--];

printf("goto F1\n");

printf("F%d: ",x );

}

void f\_gen4()

{

int x;

x=lab\_tags[ltop--];

printf("goto F%d\n",lno);

printf("F%d: ",x );

}

void w\_gen1()

{

printf("W%d: ",lno++ );

}

void w\_gen2()

{

strcpy(temp,"t");

sprintf(c,"%d" ,in);

strcat(temp,c);

printf("%s = not %s\n",temp,icg[top] );

printf("if %s goto W%d\n",temp,lno );

in++;

}

void w\_gen3()

{

printf("goto W1\n");

printf("W%d: ",lno);

}

void if\_gen1()

{

lno++;

strcpy(temp,"t");

sprintf(c,"%d" ,in);

strcat(temp,c);

printf("%s = not %s\n",temp,icg[top] );

printf("if %s goto IF%d\n",temp,lno );

in++;

lab\_tags[++ltop]=lno;

}

void if\_gen2()

{

int x;

lno++;

x=lab\_tags[ltop--];

printf("goto IF%d\n",lno);

printf("IF%d: ",x );

lab\_tags[++ltop]=lno;

}

void if\_gen3()

{

int z;

z=lab\_tags[ltop--];

printf("IF%d\n",z );

}

void array(char \*id,char \*ind)

{

strcpy(temp,"t");

sprintf(c,"%d" ,in);

strcat(temp,c);

printf("%s := %s '\*' 4\n",temp,ind);

char t[100];

strcpy(t,temp);

in++;

strcpy(temp,"t");

sprintf(c,"%d" ,in);

strcat(temp,c);

printf("%s := %s '['%s']'\n",temp,id,t );

strcpy(icg[top++],temp);

in++;

}

void func(char \*fid)

{

printf("Function Call Encountered : ");

printf("goto %s\n",fid);

}

**Test Case:**

**Input :**

#include <stdio.h>

struct abc

{

int a;

};

int foobar()

{

a=a+5;

return 0;

}

void main()

{

int n=5;

int arr[10];

float x;

n=n+b\*x+arr[1];

arr[1]=arr[2]+x+4;

if(n<x)

{

x = b + a;

}

else{

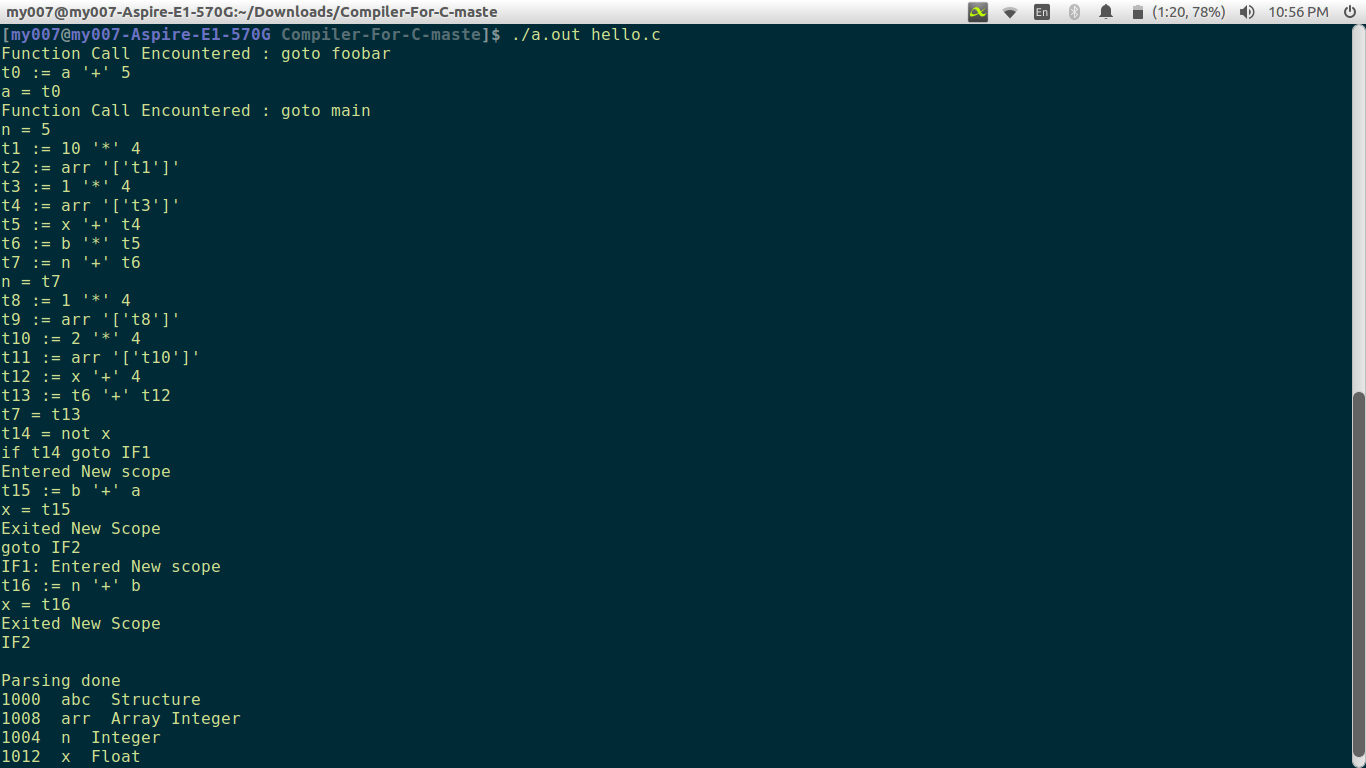
x = n + b;

}

return 0;

}

**Output :**

****

**Input -2:**

#include <stdio.h>

struct abc

{

int a;

};

int foobar()

{

a=a+5;

return 0;

}

void main()

{

int n=5;

int arr[10];

float x;

n=n+b\*x+arr[1];

arr[1]=arr[2]+x+4;

for(b=5;b<10;b=b+1){

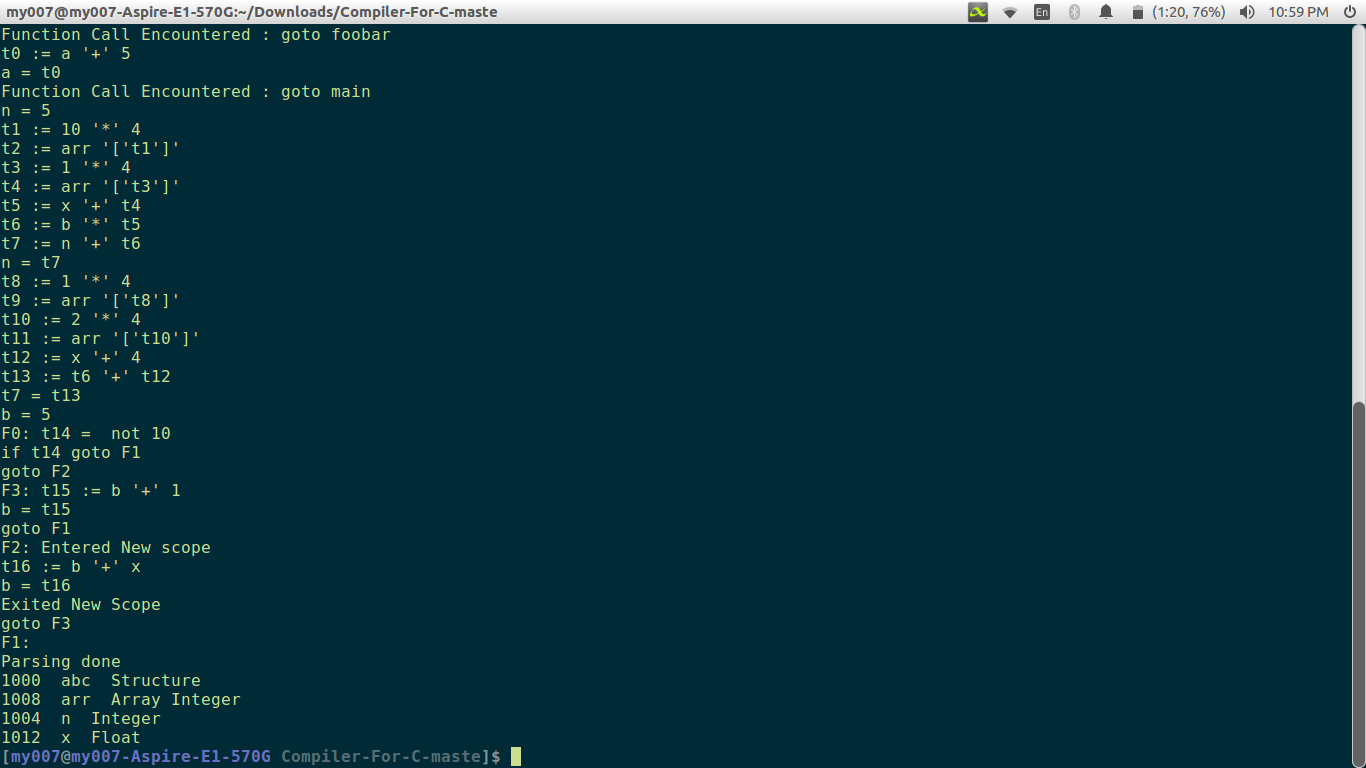
b = b + x;

}

return 0;

}

**Output-2:**

****

**Input -3:**

#include <stdio.h>

struct abc

{

int a;

};

int foobar()

{

a=a+5;

return 0;

}

void main()

{

int n=5;

int arr[10];

float x;

n=n+b\*x+arr[1];

arr[1]=arr[2]+x+4;

while( b < 5)

{

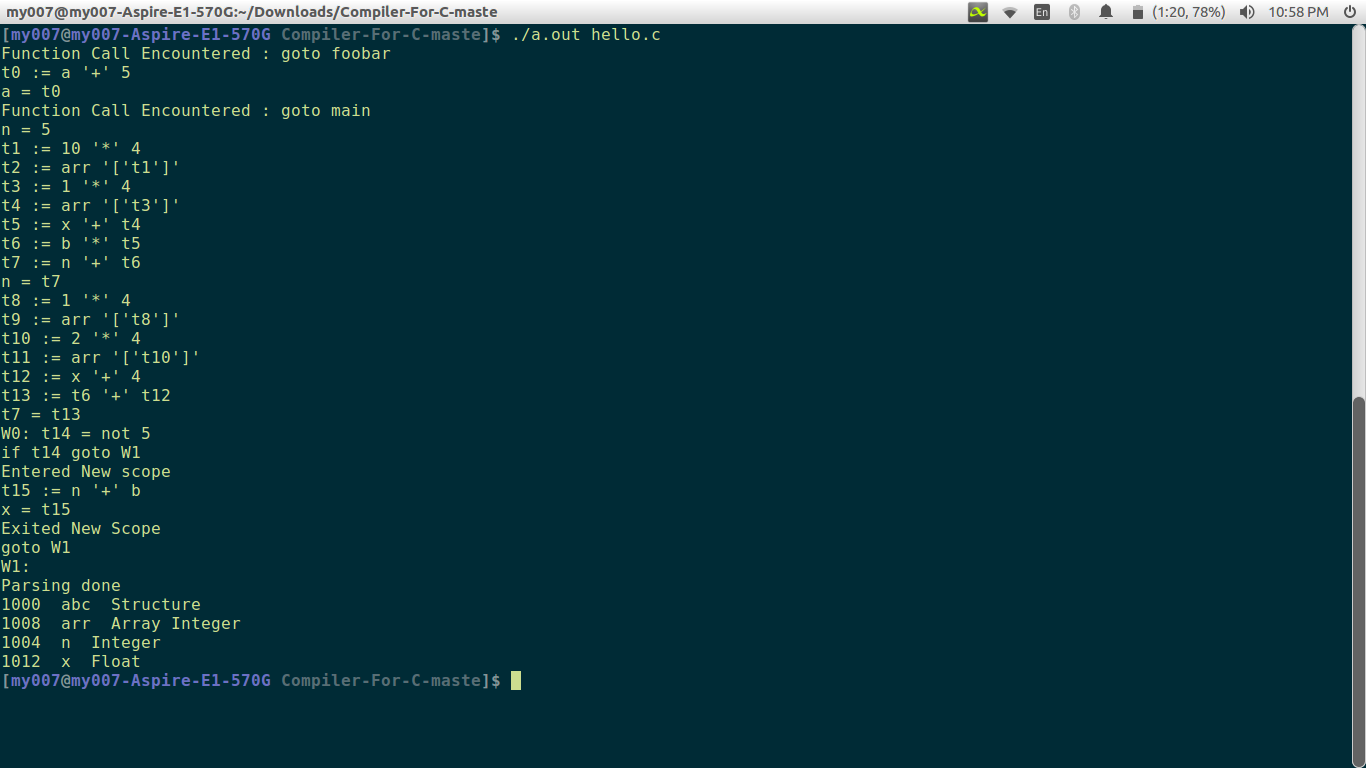
x = n+b;

}

return 0;

}

**Output -3:**

****

## 

## **Implementation:**

### **Symbol Table:**

* For handling scope of variables we use the following algorithm.
* Create a stack which will store the symbol entry or entry of each token till found ‘{‘.
* Create a copy of the top of the stack. Use that as current symbol table.
* Whenever ‘}’ is encountered. pop() from the stack. So that we get the scope of the previous code region.
* Create ICG stack which stores all the operators and the identifiers.
* When an operation (except assignment operation) is finished, a new temporary variable is assigned to it and the three address code is printed to stdout.
* For assignment operation, assign the topmost temporary variable in the stack to the identifier.
* For ‘for’ and ‘while’ loops, we use four and three generator functions respectively to implement initialization, conditional and update statements.
* For if-else statement, we use three generator functions and assign temporary variables which would be stored in the stack.
* For arrays, we multiply the array index by 4 (assuming integer occupies four bytes), calculate the array index address and assign it to a temporary variable. Use this temporary variable to access the array element.
* For functions, we implemented function calls grammer, which calls a separate function to print the goto label.

### **Semantic Errors:**

* Undeclared variables: Check in the current scope symbol table. If the entry is not present, report undeclared token.
* Re-declaration of variables: Check in the current scope symbol table (as it will contain the token of the parent/global code region), check if the variable was initialized before. If yes, print re-declaration.
* Type mismatch: For every assignment, relational or arithmetic operation, check the types of each of the token involved. If the types aren’t same, throw the error.
* Scope of a variable check: As we maintain a stack of token entries/symbol table at each code region, we can directly check for a given token. If previously declared at a higher scope, report the error.

## 

## **Results**

Compiler works properly until the intermediate code generation phase. It gives errors whenever there are type mismatch, scope issues, and re-declaration of a variable or undeclared variables, and syntax errors. The compiler provides appropriate errors along with the line number so that the user can know where the error is and may aid him in rectifying it.