**SEMANTIC ANALYZER FOR C LANGUAGE**



NATIONAL INSTITUTE OF TECHNOLOGY SURATHKAL, KARNATAKA

REPORT 3

SUBMITTED TO:

Ms. Sushmita

Mrs. Uma Priya

# Group members:

|  |  |  |  |
| --- | --- | --- | --- |
| NAME | ROLL NUMBER | EMAIL ID | PHONE NUMBER |
| Akash Rao | 15CO202 | raoakash1997@outlook.com | +91 9480948212 |
| Siddharth Valliyodhan | 15CO135 | prerana12897@gmail.com | +91 8197206218 |

# Abstract

# A ​compiler is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language). The name ​compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.

# There are many different types of compilers.

# ● If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is a ​cross-compiler.

# ● A ​bootstrap compiler is written in the language that it intends to compile.

# A compiler is likely to perform many or all of the following operations:

# ● Preprocessing

# ● Lexical analysis

# ● Parsing

# ● Semantic analysis (syntax-directed translation)

# ● Conversion of input programs to an intermediate representation

# ● Code optimization

# ● Code generation

# Compilers implement these operations in phases that promote efficient design and correct transformations of source input to target output.

# Compilers are not the only translators used to transform source programs. An interpreter is computer software that transforms and then executes the indicated operations. The translation process influences the design of computer languages which leads to a preference of compilation or interpretation. In practice, an interpreter can be implemented for compiled languages and compilers can be implemented for interpreted languages.

# 

# Introduction

**Semantic analysis** or **context sensitive analysis** is a process in compiler construction, usually after parsing, to gather necessary semantic information from the source code. It usually includes type checking or makes sure a variable is declared before use which is impossible to describe in the extended Backus–Naur form and thus not easily detected during parsing.

It is done by adding certain rules to the already existing syntax phase grammars. To these grammars we add rules such as type checking functionality, function parameter matching etc. Once the code is free of these errors we allow it to pass to the Intermediate Code Generation phase.

In yacc this is achieved through the use of yacc’s existing architecture which allows the programmer to assign attributes to non terminals. Yacc only allows the use of S-attributed grammars which means that inherited attributes are not permitted.

Thus, our compiler recognizes the following semantic errors:

1. Variable Declaration

a. Duplicate declarations of a variable.

b. Array size less than 1

1. Function Declaration
   1. Duplicate declaration of a function.
   2. Parameters of type void
   3. No function definition
   4. Return type mismatch
2. Call expressions
   1. Parameter mismatch
   2. No such function in current scope
3. Select/While Statements
   1. Statement is not of type int
4. Expressions
   1. ID not declared in current scope
   2. Type mismatch
   3. Expression in array subscript not of type int
   4. More than 1 variable on the LHS

# Code – Basic.y

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "symtbl.h"

extern FILE \*yyin;

int yylineno;

/\* Scope Evaluation \*/

int level = 0;

int depth = 0;

int scope[1000];

//scope[0] = 0;

char\* yytext;

void yyerror(char\*);

int error = 0;

char datatype\_str[100];

char vars[100][1000];

int pnt[100] = { 0 };

int varpt = -1;

char multidec\_type[1000];

int iterative = 0;

% }

%union {

char name[1000];

int size;

char type[1000];

char params[1000];

}

%right '='

%left OR

%left AND

%left '|'

%left '^'

%left '&'

%left EQUAL NOTEQUAL

%left '<' '>' LESSER GREATER

%left '+' '-'

%left '\*' '/'

%left '$'

%right IF

%right ELSE

%token STRCONST INTCONST FLTCONST CHARCONST

%token AUTO BREAK CASE CHAR CONST CONTINUE DEFAULT DO DOUBLE ELSE ENUM EXTERN FLOAT FOR IF INT LONG REGISTER RETURN SHORT SIGNED SIZEOF STATIC STRUCT SWITCH TYPEDEF UNION UNSIGNED VOID VOLATILE WHILE

%token IDENTIFIER

%token INCREMENT DECREMENT AND OR EQUAL GREATER LESSER NOTEQUAL

%%

S

:func\_def S

| multidec';' S

|

;

func\_def

:modifiers datatype IDENTIFIER '(' params\_list')' '{'statement\_list'}'

{

insert\_fun($ < name>3, $ < name>2, scope, depth, $ < params>5);

//printf("dw%s",$<type>8);

if ((!strcmp($ < type>2, "void") && strlen($ < type>8) != 0) && (strcmp($ < type>2, $ < type>8)))

yyerror("Return type not matching");

node\* t = lookup($ < name>3, FUN\_TBL);

t->func\_def = 1;

}

| modifiers datatype IDENTIFIER '('')' '{'statement\_list'}'

{

insert\_fun($ < name>3, $ < name>2, scope, depth, NULL);

//printf("dw%s",$<type>7);

if ((!strcmp($ < type>2, "void") && strlen($ < type>7) != 0) && (strcmp($ < type>2, $ < type>7)))

yyerror("Return type not matching");

node\* t = lookup($ < name>3, FUN\_TBL);

t->func\_def = 1;

}

| modifiers datatype IDENTIFIER '(' params\_list')' ';'

{

insert\_fun($ < name>3, $ < name>2, scope, depth, $ < params>5);

}

| modifiers datatype IDENTIFIER '('')' ';'

{

insert\_fun($ < name>3, $ < name>2, scope, depth, NULL);

}

| modifiers datatype'\*' IDENTIFIER '(' params\_list')' '{'statement\_list'}'

{

char temp[1000];

strcpy(temp, $ < name>2);

strcat(temp, "\*");

insert\_fun($ < name>3, $ < name>2, scope, depth, $ < params>6);

if (strcmp(temp, "void") && strcmp(temp, $ < type>9))

yyerror("Return type not matching");

node\* t = lookup($ < name>3, FUN\_TBL);

t->func\_def = 1;

}

| modifiers datatype'\*' IDENTIFIER '('')' '{'statement\_list'}'

{

char temp[1000];

strcpy(temp, $ < name>2);

strcat(temp, "\*");

insert\_fun($ < name>3, $ < name>2, scope, depth, NULL);

if (strcmp(temp, "void") && strcmp(temp, $ < type>8))

yyerror("Return type not matching");

node\* t = lookup($ < name>3, FUN\_TBL);

t->func\_def = 1;

}

| modifiers datatype'\*' IDENTIFIER '(' params\_list')' ';'

{

char temp[1000];

strcpy(temp, $ < name>2);

strcat(temp, "\*");

insert\_fun($ < name>3, $ < name>2, scope, depth, $ < params>6);

}

| modifiers datatype'\*' IDENTIFIER '('')' ';'

{

char temp[1000];

strcpy(temp, $ < name>2);

strcat(temp, "\*");

insert\_fun($ < name>3, $ < name>2, scope, depth, NULL);

}

;

multidec

:modifiers datatype id\_chain

{

if (!strcmp($ < type>2 ,"void"))

yyerror("Variable of type void");

while (varpt >= 0)

{

if (pnt[varpt])

{

char temp[1000];

strcpy(temp,$ < name>2);

strcat(temp,"\*");

pnt[varpt] = 0;

node\* t = lookup(vars[varpt--], SYM\_TBL);

strcpy(t->type, temp);

}

else

{

node\* t = lookup(vars[varpt--], SYM\_TBL);

strcpy(t->type, $ < name>2);

}

}

}

;

;

datatype

:INT

{ strcpy($<type>$, "int"); }

| FLOAT

{ strcpy($<type>$, "float"); }

| CHAR

{ strcpy($<type>$, "char"); }

| DOUBLE

{ strcpy($<type>$, "double"); }

| VOID

{ strcpy($<type>$, "void"); }

| LONG

{ strcpy($<type>$, "long"); }

| SHORT

{ strcpy($<type>$, "short"); }

;

modifiers

:AUTO

| CONST

| EXTERN

| REGISTER

| SIGNED

| UNSIGNED

| VOLATILE

|

;

params\_dec

:modifiers datatype IDENTIFIER

{

if (!strcmp($ < type>2 ,"void"))

yyerror("Parameter of type void");

strcpy($<name>$,$ < name>3);

insert($ < name>3, $ < name>2, scope, depth, 0);

strcpy($<type>$,$ < type>2);

strcpy($<name>$,$ < name>3);

}

| modifiers datatype IDENTIFIER '['INTCONST']'

{

if (!strcmp($ < type>2, "void"))

yyerror("Parameter of type void");

strcpy($<name>$, $ < name>3);

char temp[1000];

strcpy(temp, $ < name>2);

strcat(temp, "\*");

insert($ < name>3, temp, scope, depth, 0);

strcpy($<type>$, temp);

int t\_array = atoi($ < name>5);

node\* t = lookup($ < name>3, SYM\_TBL);

t->array\_bound = t\_array;

strcpy($<name>$, $ < name>3);

}

| modifiers datatype IDENTIFIER '['']'

{

if (!strcmp($ < type>2, "void"))

yyerror("Parameter of type void");

strcpy($<name>$, $ < name>3);

char temp[1000]; strcpy(temp, $ < name>2);

strcat(temp, "\*");

insert($ < name>3, temp, scope, depth, 0);

strcpy($<type>$, temp);

strcpy($<name>$, $ < name>3);

}

| modifiers datatype '\*' IDENTIFIER

{

strcpy($<name>$,$ < name>3);

char temp[1000];

strcpy(temp,multidec\_type);

strcat(temp,"\*");

insert($ < name>4, temp, scope, depth, 0);

strcpy($<type>$,temp);

strcpy($<name>$,$ < name>4);

}

;

params\_list

:params\_dec

{

printf("\n Name %s ",$ < type>1);

change\_scope($ < name>1, level);

strcpy($<params>$, $ < type>1);

}

| params\_dec ',' params\_list

{ change\_scope($ < name>1, level);

char temp[1000]; strcpy(temp, $ < type>1);

strcat(temp, $ < params>3); strcpy($<params>$, temp);

}

;

brackets

:'('expression')'{strcpy($<type>$, $ < type>2); }

;

expression

:constant

{

node\* t = lookup($ < name>1, CONST\_TBL);

if (t != NULL)

strcpy($<type>$, t->type);

}

| IDENTIFIER bin\_op expression

{

check\_scope($ < name>1, scope, depth, SYM\_TBL);

node\* t = lookup($ < name>1, SYM\_TBL);

if (strcmp(t->type, $ < type>3))

yyerror("Type mismatch");

strcpy($<type>$, t->type);

}

| constant bin\_op expression

{

strcpy($<type>$, $ < type>1);

node\* t = lookup($ < name>1, CONST\_TBL);

if (t != NULL)

{

if (strcmp(t->type, $ < type>3))

yyerror("Type mismatch");

strcpy($<type>$, t->type);

}

}

| IDENTIFIER

{

check\_scope($ < name>1, scope, depth, SYM\_TBL);

node\* t = lookup($ < name>1, SYM\_TBL);

strcpy($<type>$, t->type);

}

| un\_op IDENTIFIER

{

check\_scope($ < name>1, scope, depth, SYM\_TBL);

node\* t = lookup($ < name>2, SYM\_TBL);

strcpy($<type>$, t->type);

}

| un\_op constant

{ strcpy($<type>$, $ < type>2); }

| IDENTIFIER INCREMENT

{

check\_scope($ < name>1, scope, depth, SYM\_TBL);

node\* t = lookup($ < name>1, SYM\_TBL);

if (strcmp(t->type, "int"))

yyerror("Increment not possible");

strcpy($<type>$, t->type);

}

| IDENTIFIER DECREMENT

{

check\_scope($ < name>1, scope, depth, SYM\_TBL);

node\* t = lookup($ < name>1, SYM\_TBL);

if (strcmp(t->type, "int"))

yyerror("Increment not possible");

strcpy($<type>$, t->type);

}

| func\_call bin\_op expression

{

check\_scope($ < name>1, scope, depth, FUN\_TBL);

node\* t = lookup($ < name>1, FUN\_TBL);

if (strcmp(t->type, $ < type>3))

yyerror("Type mismatch");

strcpy($<type>$, t->type);

}

| func\_call

{

check\_scope($ < name>1, scope, depth, FUN\_TBL);

node\* t = lookup($ < name>1, FUN\_TBL);

strcpy($<type>$, t->type);

}

| un\_op func\_call

{

check\_scope($ < name>1, scope, depth, FUN\_TBL);

node\* t = lookup($ < name>2, FUN\_TBL);

strcpy($<type>$, t->type);

}

| brackets

{ strcpy($<type>$, $ < type>1); }

| un\_op brackets

{ strcpy($<type>$, $ < type>2); }

| brackets bin\_op expression

{

if (strcmp($ < type>1, $ < type>3))

yyerror("Type mismatch");

strcpy($<type>$, $ < type>3);

}

| IDENTIFIER'['expression']'

{

check\_scope($ < name>1, scope, depth, SYM\_TBL);

node\* t = lookup($ < name>1, SYM\_TBL);

if (strcmp($ < type>3, "int"))

yyerror("Array index not integer");

char temp[1000];

strcpy(temp, t->type);

temp[strlen(t->type) - 1] = 0;

strcpy($<type>$, temp);

}

| IDENTIFIER'['expression']' bin\_op expression

{

check\_scope($ < name>1, scope, depth, SYM\_TBL);

if (strcmp($ < type>3, "int"))

yyerror("Array index not integer");

node\* t = lookup($ < name>1, SYM\_TBL);

char temp[100];

strcpy(temp, t->type);

temp[strlen(t->type) - 1] = 0;

if (strcmp($ < type>3, temp))

yyerror("Type mismatch");

strcpy($<type>$,temp);

}

| un\_op IDENTIFIER'['expression']'

{

check\_scope($ < name>3, scope, depth, SYM\_TBL);

if (strcmp($ < type>4, "int"))

yyerror("Array index not integer");

node\* t = lookup($ < name>1, SYM\_TBL);

strcpy($<type>$, t->type);

}

| IDENTIFIER '['expression']' INCREMENT

{

check\_scope($ < name>1, scope, depth, SYM\_TBL);

if (strcmp($ < type>3, "int"))

yyerror("Array index not integer");

node\* t = lookup($ < name>1, SYM\_TBL);

char temp[100];

strcpy(temp, t->type);

temp[strlen(t->type) - 1] = 0;

if (strcmp($ < type>1, "int\*"))

yyerror("Type mismatch");

strcpy($<type>$, temp);

}

| IDENTIFIER '['expression']' DECREMENT

{

check\_scope($ < name>1, scope, depth, SYM\_TBL);

if (strcmp($ < type>3, "int"))

yyerror("Array index not integer");

node\* t = lookup($ < name>1, SYM\_TBL);

char temp[100];

strcpy(temp, t->type);

temp[strlen(t->type) - 1] = 0;

if (strcmp($ < type>1, "int\*"))

yyerror("Type mismatch");

strcpy($<type>$, temp);

}

;

const\_list

:constant

{ $<size>$ = 1; }

| constant ',' const\_list

{ $<size>$ = $ < size>3 + 1; }

;

statement\_list

:statement

{ strcpy($<type>$, $ < type>1); }

| statement statement\_list

{

strcpy($<type>$, $ < type>2);

if (strlen($ < type>1) > 0 && strlen($ < type>2) > 0 && strcmp($ < type>1, $ < type>2))

yyerror("Return type not matching");

if (strlen($ < type>1) > 0)

strcpy($<type>$, $ < type>1);

}

;

constant

:INTCONST

{

insert($<name>$, "int", scope, depth, 1);

strcpy($<type>$, "int");

}

| STRCONST

{

insert($<name>$, "string", scope, depth, 1);

strcpy($<type>$, "char\*");

}

| FLTCONST

{

insert($<name>$, "float", scope, depth, 1);

strcpy($<type>$, "float");

}

| CHARCONST

{

insert($<name>$, "char", scope, depth, 1);

strcpy($<type>$, "char");

}

;

bin\_op

:'+'

| '-'

| '\*'

| '/'

| AND

| OR

| EQUAL

| GREATER

| LESSER

| NOTEQUAL

| '%'

| '^'

| '|'

| '>'

| '<'

;

un\_op

:'!' %prec '$'

| '+' %prec '$'

| '-' %prec '$'

| '\*' %prec '$'

| '&' %prec '$'

| INCREMENT % prec '$'

| DECREMENT % prec '$'

;

func\_call

:IDENTIFIER '('expression\_list')'

{

check\_scope($ < name>1, scope, depth, FUN\_TBL);

node\* t = lookup($ < name>1, FUN\_TBL);

if (t->func\_def != 1)

yyerror("Function not defined");

if (strcmp($ < params>3, t->params))

yyerror("Parameters type mismatch");

strcpy($<name>$, $ < name>1);

}

| IDENTIFIER '('')'

{

check\_scope($ < name>1, scope, depth, FUN\_TBL);

node\* t = lookup($ < name>1, FUN\_TBL);

if (t->func\_def != 1)

yyerror("Function not defined");

if (t->params != NULL)

yyerror("Parameters type mismatch");

strcpy($<name>$, $ < name>1);

}

;

expression\_list

:expression

{ strcpy($<params>$, $ < type>1); }

| expression ',' expression\_list

{

char temp[1000];

strcpy(temp, $ < type>1);

strcat(temp, $ < params>3);

strcpy($<params>$, temp);

}

;

statement

:multidec ';'

{ strcpy($<type>$, ""); }

| conditional

{ strcpy($<type>$, ""); }

| iterative

{ strcpy($<type>$, ""); }

| assignment

{ strcpy($<type>$, ""); }

| RETURN expression';'

{ strcpy($<type>$, $ < type>2); }

| func\_call';'

{ strcpy($<type>$, ""); }

| func\_def

{ strcpy($<type>$, ""); }

| IDENTIFIER INCREMENT ';'

{

strcpy($<type>$, "");

check\_scope($ < name>1, scope, depth, SYM\_TBL);

}

| IDENTIFIER DECREMENT ';'

{

strcpy($<type>$, "");

check\_scope($ < name>1, scope, depth, SYM\_TBL);

}

| '{'statement\_list'}'

{ strcpy($<type>$, $ < type>2); }

| ';'

{ strcpy($<type>$, ""); }

| BREAK';'

{

if (iterative <= 0)

yyerror("Invalid use of break statement");

strcpy($<type>$, "");

}

;

id\_chain

:IDENTIFIER

{

varpt++; strcpy(vars[varpt],$ < name>1);

insert($ < name>1, multidec\_type, scope, depth, 0);

}

| IDENTIFIER '=' expression

{

varpt++; strcpy(vars[varpt],$ < name>1);

insert($ < name>1, multidec\_type, scope, depth, 0);

}

| '\*'IDENTIFIER

{

varpt++; strcpy(vars[varpt],$ < name>2);

pnt[varpt] = 1;

insert($ < name>2, multidec\_type, scope, depth, 0);

}

| id\_chain ',' '\*'IDENTIFIER

{

varpt++; strcpy(vars[varpt],$ < name>4);

pnt[varpt] = 1;

insert($ < name>4, multidec\_type, scope, depth, 0);

}

| '\*'IDENTIFIER '=' expression

{

varpt++; strcpy(vars[varpt],$ < name>2);

pnt[varpt] = 1;

insert($ < name>2, multidec\_type, scope, depth, 0);

}

| id\_chain ',' '\*'IDENTIFIER '=' expression

{

varpt++; strcpy(vars[varpt],$ < name>4);

pnt[varpt] = 1;

insert($ < name>4, multidec\_type, scope, depth, 0);

}

| id\_chain ',' IDENTIFIER '=' expression

{

varpt++; strcpy(vars[varpt],$ < name>3);

insert($ < name>3, multidec\_type, scope, depth, 0);

}

| id\_chain ',' IDENTIFIER

{

varpt++; strcpy(vars[varpt],$ < name>3);

insert($ < name>3, multidec\_type, scope, depth, 0);

}

| IDENTIFIER '['INTCONST']'

{

varpt++; strcpy(vars[varpt], $ < name>1);

pnt[varpt] = 1;

insert($ < name>1, multidec\_type, scope, depth, 0);

int t\_array = atoi($ < name>3);

if (t\_array <= 0)

yyerror("Invalid array size");

node\* t = lookup($ < name>1, SYM\_TBL);

t->array\_bound = t\_array;

}

| IDENTIFIER'['']' '=' '{'const\_list'}'

{

varpt++; strcpy(vars[varpt], $ < name>1);

pnt[varpt] = 1;

insert($ < name>1, multidec\_type, scope, depth, 0);

node\* t = lookup($ < name>1, SYM\_TBL);

t->array\_bound = $ < size>6;

}

| IDENTIFIER '['INTCONST']' '=' '{'const\_list'}'

{

varpt++; strcpy(vars[varpt], $ < name>1);

pnt[varpt] = 1;

insert($ < name>1, multidec\_type, scope, depth, 0);

int t\_array = atoi($ < name>3);

if (t\_array < $ < size>7)

yyerror("Array index out of bound");

node\* t = lookup($ < name>1, SYM\_TBL);

t->array\_bound = t\_array;

}

| id\_chain ',' IDENTIFIER'['INTCONST']'

{

varpt++; strcpy(vars[varpt], $ < name>3);

pnt[varpt] = 1;

insert($ < name>3, multidec\_type, scope, depth, 0);

int t\_array = atoi($ < name>5);

if (t\_array <= 0)

yyerror("Invalid array size");

node\* t = lookup($ < name>3, SYM\_TBL);

t->array\_bound = t\_array;

}

| id\_chain ',' IDENTIFIER '['']' '=' '{'const\_list'}'

{

varpt++; strcpy(vars[varpt], $ < name>3);

pnt[varpt] = 1;

insert($ < name>3, multidec\_type, scope, depth, 0);

node\* t = lookup($ < name>3, SYM\_TBL);

t->array\_bound = $ < size>8;

}

| id\_chain ',' IDENTIFIER '['INTCONST']' '=' '{'const\_list'}'

{

varpt++; strcpy(vars[varpt], $ < name>3);

pnt[varpt] = 1;

insert($ < name>3, multidec\_type, scope, depth, 0);

int t\_array = atoi($ < name>5);

if (t\_array < $ < size>9)

yyerror("Array index out of bound");

node\* t = lookup($ < name>3, SYM\_TBL);

t->array\_bound = t\_array;

}

;

conditional

:IF '('expression')' statement %prec IF

{

if (strcmp($ < type>3,"int"))

yyerror("Invalid expression");

}

| IF '('expression')' statement ELSE statement

{

if (strcmp($ < type>3,"int"))

yyerror("Invalid expression");

}

;

iterative

:WHILE'('expression')' statement

{

iterative--;

if (strcmp($ < type>3,"int"))

yyerror("Invalid expression");

}

;

assignment

:IDENTIFIER '=' expression ';'

{check\_scope($ < name>1, scope, depth, SYM\_TBL); }

| '\*'IDENTIFIER '=' expression';'

{check\_scope($ < name>1, scope, depth, SYM\_TBL); }

| IDENTIFIER'['expression']' '=' expression';'

{

if (strcmp($ < type>3, "int"))

yyerror("Invalid array index");

}

;

%%

void yyerror(char\* s)

{

error = 1;

//printf("ERROR: %s\n", s);

fprintf(stderr, "\nLINE %d: %s \nERROR: %s\n", yylineno, s, yytext);

//exit(0);

}

int main()

{

//yyin = fopen("test\_cases/semantic/return\_type.c", "r");

yyin = fopen("test\_cases/program.c", "r");

yyparse();

if (error)

printf("\nUNSUCCESSFUL\n");

else

printf("\nSUCCESS!\n");

display();

return 0;

}

# Code – symtbl.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "symtbl.h"

int hash(char\* x, int M)

{

int i, sum;

for (sum = 0, i = 0; i < strlen(x); i++)

sum += x[i];

return sum % M;

}

node\* lookup(char\* x, int table)

{

int idx = hash(x, 100);

node\* t = NULL;

if (table == SYM\_TBL)

{

if (sym\_tbl[idx] == NULL)

return NULL;

t = sym\_tbl[idx];

}

else if (table == CONST\_TBL)

{

if (const\_tbl[idx] == NULL)

return NULL;

t = const\_tbl[idx];

}

else if (table == FUN\_TBL)

{

if (fun\_tbl[idx] == NULL)

return NULL;

t = fun\_tbl[idx];

}

while (t != NULL)

{

if (strcmp(t->name, x) == 0)

return t;

t = t->next;

}

return NULL;

}

void insert(char\* x, char\* type, int\* scope\_in, int depth\_in, int table)

{

if (lookup(x, table) != NULL && table != 1)

{

char error\_msg[100];

strcpy(error\_msg, "Variable already declared: ");

strcat(error\_msg, x);

yyerror(error\_msg);

return;

}

int idx = hash(x, 100);

node\* cell = (node\*)malloc(sizeof(node));

cell->name = (char\*)malloc(strlen(x));

strcpy(cell->name, x);

cell->type = (char\*)malloc(strlen(type));

strcpy(cell->type, type);

if (depth\_in == 0)

{

cell->scope = NULL;

cell->depth = 0;

}

else

{

cell->scope = (int\*)malloc(sizeof(int)\*depth\_in);

cell->depth = depth\_in;

}

for (int i = 0; i<depth\_in; i++)

{

cell->scope[i] = scope\_in[i];

}

cell->array\_bound = 0;

cell->next = NULL;

node\* t = NULL;

if (table == SYM\_TBL)

{

if (sym\_tbl[idx] == NULL)

{

sym\_tbl[idx] = cell;

return;

}

t = sym\_tbl[idx];

}

else

{

if (const\_tbl[idx] == NULL)

{

const\_tbl[idx] = cell;

return;

}

t = const\_tbl[idx];

}

while (t->next != NULL)

t = t->next;

t->next = cell;

}

void insert\_fun(char\* x, char\* type, int\* scope\_in, int depth\_in, char\* param\_list\_in)

{

if (lookup(x, 2) != NULL)

{

char error\_msg[100];

strcpy(error\_msg, "Function already declared: ");

strcat(error\_msg, x);

yyerror(error\_msg);

return;

}

int idx = hash(x, 100);

node\* cell = (node\*)malloc(sizeof(node));

cell->name = (char\*)malloc(strlen(x));

strcpy(cell->name, x);

cell->type = (char\*)malloc(strlen(type));

strcpy(cell->type, type);

if (depth\_in == 0)

{

cell->scope = NULL;

cell->depth = 0;

}

else

{

cell->scope = (int\*)malloc(sizeof(int)\*depth\_in);

cell->depth = depth\_in;

}

for (int i = 0; i<depth\_in; i++)

{

cell->scope[i] = scope\_in[i];

}

if (param\_list\_in == NULL)

cell->params = NULL;

else

{

cell->params = (char\*)malloc(strlen(param\_list\_in));

strcpy(cell->params, param\_list\_in);

}

cell->func\_def = 0;

cell->next = NULL;

node\* t = NULL;

if (fun\_tbl[idx] == NULL)

{

fun\_tbl[idx] = cell;

return;

}

t = fun\_tbl[idx];

while (t->next != NULL)

t = t->next;

t->next = cell;

}

void check\_scope(char\* x, int\* scope\_in, int depth\_in, int table)

{

char error\_msg[100];

node\* t = lookup(x, table);

if (t == NULL)

{

strcpy(error\_msg, "Identifier undeclared: ");

strcat(error\_msg, x);

yyerror(error\_msg);

return;

}

int i;

if (depth\_in<t->depth)

{

strcpy(error\_msg, "Identifier out of scope: ");

strcat(error\_msg, x);

yyerror(error\_msg);

return;

}

for (i = 0; i<t->depth; i++)

{

if (scope\_in[i] != t->scope[i])

{

strcpy(error\_msg, "Identifier out of scope: ");

strcat(error\_msg, x);

yyerror(error\_msg);

return;

}

}

}

void change\_scope(char\* x, int level\_in)

{

char error\_msg[100];

node\* t = lookup(x, 0);

if (t == NULL)

return;

t->depth = t->depth + 1;

int \*temp = (int\*)malloc(sizeof(int)\*(t->depth));

for (int i = 0; i<t->depth - 1; i++)

{

temp[i] = t->scope[i];

}

t->scope = temp;

t->scope[t->depth - 1] = level\_in + 1;

}

void display()

{

printf("\n------------------------------------------------\n\t\tSymbol table\n------------------------------------------------\n");

printf("Value\t\t-\tType\t-\tScope\n------------------------------------------------\n");

int i, j;

for (i = 0; i<100; i++)

{

if (sym\_tbl[i] == NULL)

continue;

node\* t = sym\_tbl[i];

while (t != NULL)

{

printf("%s\t\t-\t%s\t-\t", t->name, t->type);

if (t->depth == 0)

printf("Global");

for (j = 0; j<t->depth; j++)

printf("%d ", t->scope[j]);

printf("\n");

t = t->next;

}

}

printf("\n\n-------------------------------------------------------------------------\n\t\tFunction table\n--------------------------------------------------------------------------\n");

printf("Value\t\t-\tType\t-\tScope\t\t-\tParams\n--------------------------------------------------------------------------\n");

for (i = 0; i<100; i++)

{

if (fun\_tbl[i] == NULL)

continue;

node\* t = fun\_tbl[i];

while (t != NULL)

{

printf("%s\t\t-\t%s\t-\t", t->name, t->type);

if (t->depth == 0)

printf("Global");

for (j = 0; j<t->depth; j++)

printf("%d ", t->scope[j]);

printf("\t\t-\t%s\n", t->params);

t = t->next;

}

}

printf("\n\n----------------------------\n\tConstant table\n----------------------------\n");

printf("Value\t\t-\tType\n----------------------------\n");

for (i = 0; i<100; i++)

{

if (const\_tbl[i] == NULL)

continue;

node\* t = const\_tbl[i];

while (t != NULL)

{

printf("%s\t\t-\t%s\n", t->name, t->type);

t = t->next;

}

}

}

# Code- scanner.l

%{

#include "y.tab.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

extern int level;

extern int depth;

extern int scope[1000];

extern int yylineno;

extern int iterative;

%}

%%

"//".\*\n{ /\*printf("Single line comment\n");\*/ yylineno++; }

"/\*"([^\*] | \\*+[^\* / ])\*"\*/" { /\*printf("Multi line comment\n");\*/ }

\"(\\.|[^"\n\\])\*\" {/\*printf("String const:\t\t%s\n", yytext); insert(yytext, 1, CONST\_TBL);\*/ strcpy(yylval.name,yytext); return STRCONST; }

0[xX][0 - 9a - fA - F] + {/\*printf("Int const:\t\t%s\n", yytext); insert(yytext, 2, CONST\_TBL);\*/ strcpy(yylval.name, yytext); return INTCONST; }

[0 - 9] + {/\*printf("Int const:\t\t%s\n", yytext); insert(yytext, 2, CONST\_TBL);\*/ strcpy(yylval.name, yytext); return INTCONST; }

(([0 - 9] + ) | ([0 - 9] \* \.[0 - 9] + )([eE][-+] ? [0 - 9] + ) ? ) {/\*printf("Float const:\t\t%s\n", yytext); insert(yytext, 3, CONST\_TBL);\*/ strcpy(yylval.name, yytext); return FLTCONST;

}

'([^'\\\n] | \\.)' {/\*printf("Char const:\t\t%s\n", yytext); insert(yytext, 4, CONST\_TBL);\*/ strcpy(yylval.name,yytext); return CHARCONST; }

' {/\*printf("\n Error: Unterminated Character constant\n\n");\*/ }

'' {/\*printf("Char const:\t\t%s\n", yytext); insert(yytext, 1, CONST\_TBL);\*/ }

'([^'\\\n] | \\.) + ' {/\*printf("\n Error: Character constant too long\n\n");\*/ }

"auto" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return AUTO; }

"break" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return BREAK; }

"case" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return CASE; }

"char" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ strcpy(yylval.name, yytext); return CHAR; }

"const" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return CONST; }

"continue" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return CONTINUE; }

"default" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return DEFAULT; }

"do" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return DO; }

"double" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ strcpy(yylval.name, yytext); return DOUBLE; }

"else" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return ELSE; }

"enum" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return ENUM; }

"extern" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return EXTERN; }

"float" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ strcpy(yylval.name, yytext); return FLOAT; }

"for" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return FOR; }

"if" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return IF; }

"int" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ strcpy(yylval.name, yytext); return INT; }

"long" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ strcpy(yylval.name, yytext); return LONG; }

"register" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return REGISTER; }

"return" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return RETURN; }

"short" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ strcpy(yylval.name, yytext); return SHORT; }

"signed" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return SIGNED; }

"sizeof" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return SIZEOF; }

"static" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return STATIC; }

"struct" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return STRUCT; }

"switch" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return SWITCH; }

"typedef" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return TYPEDEF; }

"union" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return UNION; }

"unsigned" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return UNSIGNED; }

"void" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ strcpy(yylval.name, yytext); return VOID; }

"volatile" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ return VOLATILE; }

"while" { /\*printf("Keyword:\t\t%s\n", yytext);\*/ iterative++; return WHILE; }

[a - zA - Z\_][a - zA - Z0 - 9\_] \* { /\*printf("Identifier:\t\t%s\n", yytext);\*/ /\*insert(yytext, 0, SYM\_TBL);\*/ strcpy(yylval.name,yytext); return IDENTIFIER; }

"\*/" { /\*printf("\n Error: Unexpected end of comment\n\n");\*/ }

"/\*" { /\*printf("\n Error: Unterminated Multi line comment\n\n");\*/ }

"++" { /\*printf("Operator:\t\t%s\n", yytext);\*/ return INCREMENT; }

"--" { /\*printf("Operator:\t\t%s\n", yytext);\*/ return DECREMENT; }

"&&" { /\*printf("Operator:\t\t%s\n", yytext);\*/ return AND; }

"||" { /\*printf("Operator:\t\t%s\n", yytext);\*/ return OR; }

"==" { /\*printf("Operator:\t\t%s\n", yytext);\*/ return EQUAL; }

">=" { /\*printf("Operator:\t\t%s\n", yytext);\*/ return GREATER; }

"<=" { /\*printf("Operator:\t\t%s\n", yytext);\*/ return LESSER; }

"!=" { /\*printf("Operator:\t\t%s\n", yytext);\*/ return NOTEQUAL; }

"!" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '!'; }

"%" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '%'; }

"^" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '^'; }

"&" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '&'; }

"\*" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '\*'; }

"(" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '('; }

")" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return ')'; }

"-" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '-'; }

"+" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '+'; }

"=" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '='; }

"{" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ depth++; level++; scope[depth - 1] = level; return '{'; }

"}" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ scope[depth] = 0; depth--; return '}'; }

"|" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '|'; }

"~" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '~'; }

"[" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '['; }

"]" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return ']'; }

";" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return ';'; }

":" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return ':'; }

"<" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '<'; }

">" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '>'; }

"?" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '?'; }

"," { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return ','; }

"." { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '.'; }

"/" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '/'; }

"#" { /\*printf("Punctuator:\t\t%s\n", yytext);\*/ return '#'; }

"\n" { /\*printf("\n");\*/ yylineno++; }

" " { /\*printf(" ");\*/ }

"\t" { /\*printf("\t");\*/ }

"\"" { /\*printf("\n Error: Unmatched quotation\n\n");\*/ }

. { /\*printf("\n Error: Invalid token \n\n");\*/ }

%%

int yywrap()

{

return 1;

}

Implementation Details

For scope resolution, we maintained a stack by the name scope, which keeps tracks of the current scope state. Variable named level is used to indicate different blocks, each of the block gets a unique level variable. The variable depth is used for maintaining the nested level of the blocks currently the scope state is in. Every time we encounter a new variable declaration, we store the current stack state along with the current depth to maintain the scope of the variable.

Functions are now stored seperately from regular varibles(in a seperate table). The parameter list is also stored in the table. Parameter list attribute records the parameter count and type of each parameters in the function. This is done by concatenating all the parameters' datatype. The return type is also saved as an attribute. Whenever the function is called, the respective feilds of the function call and function definition is matched. An error message is returned in case of inconsistency.

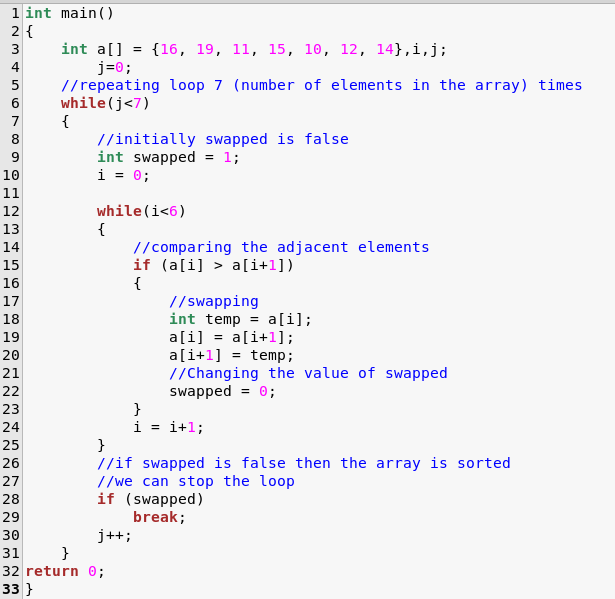
Also if the function has multiple return statements, the type of all the returned entity is checked against each other.

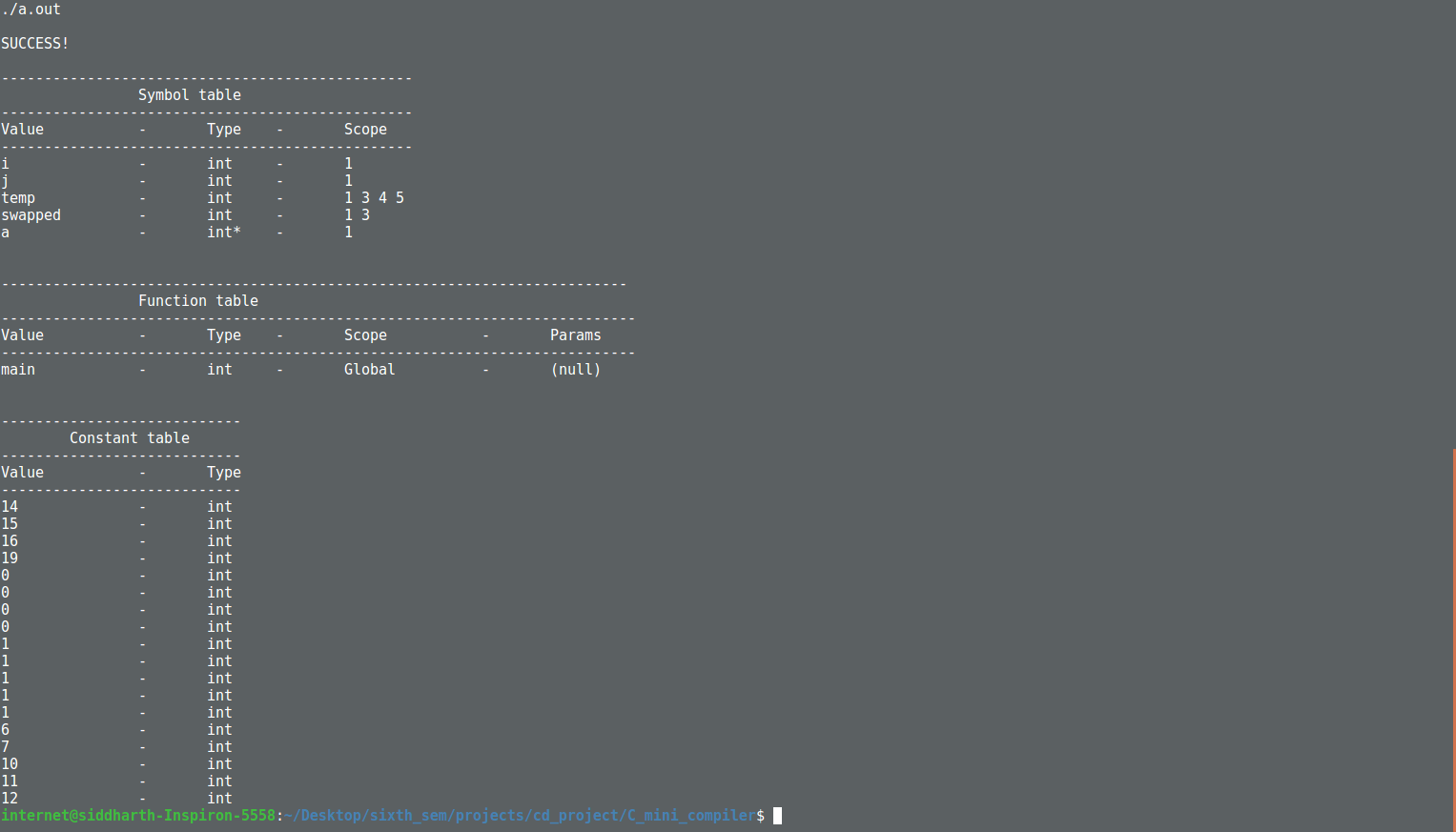
The current type of the expression is always maintained. This way the if construct is evaluted only if the condition expression is an integer. Also, accessing the array elements is possible only if the datatype of the iterator is integer.

Screenshots/Test Case Explanations

1) Arrays.c

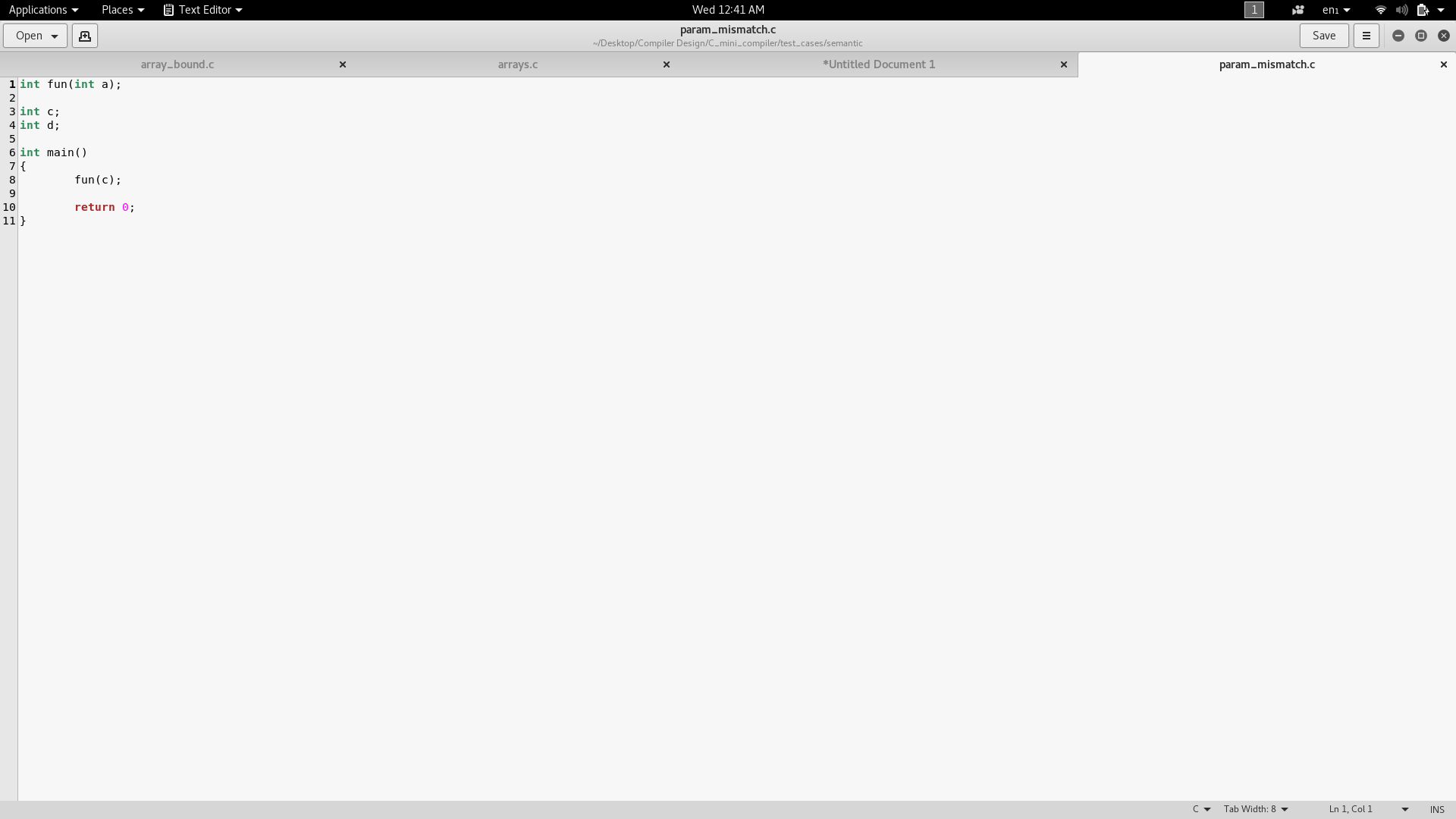
A normal C code to use bubble sort to sort a list of given numbers in ascending order. This tests the array index restrictions, the type checking mechanisms, the break validity checks etc.

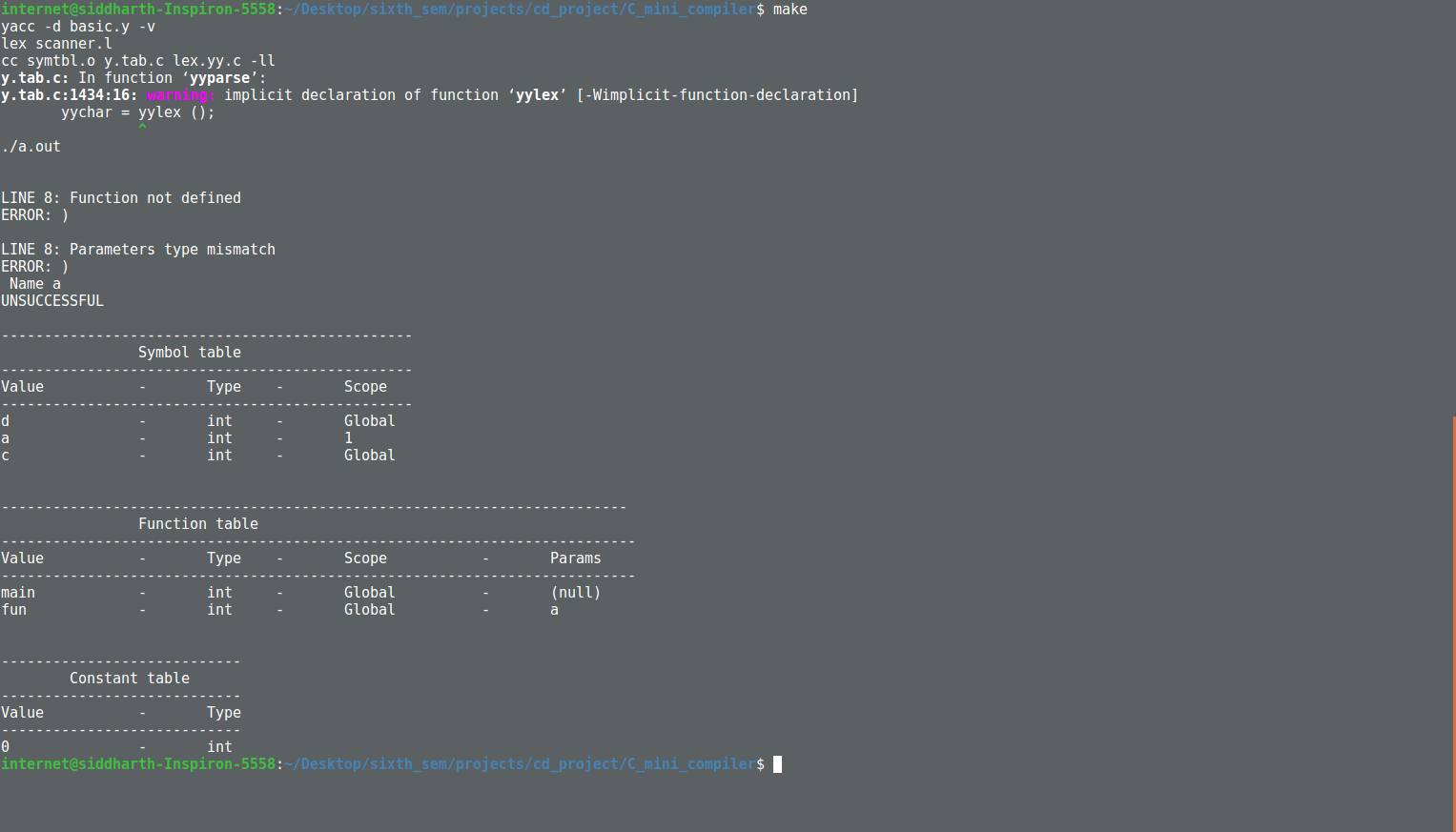




2) Parameter Mismatch

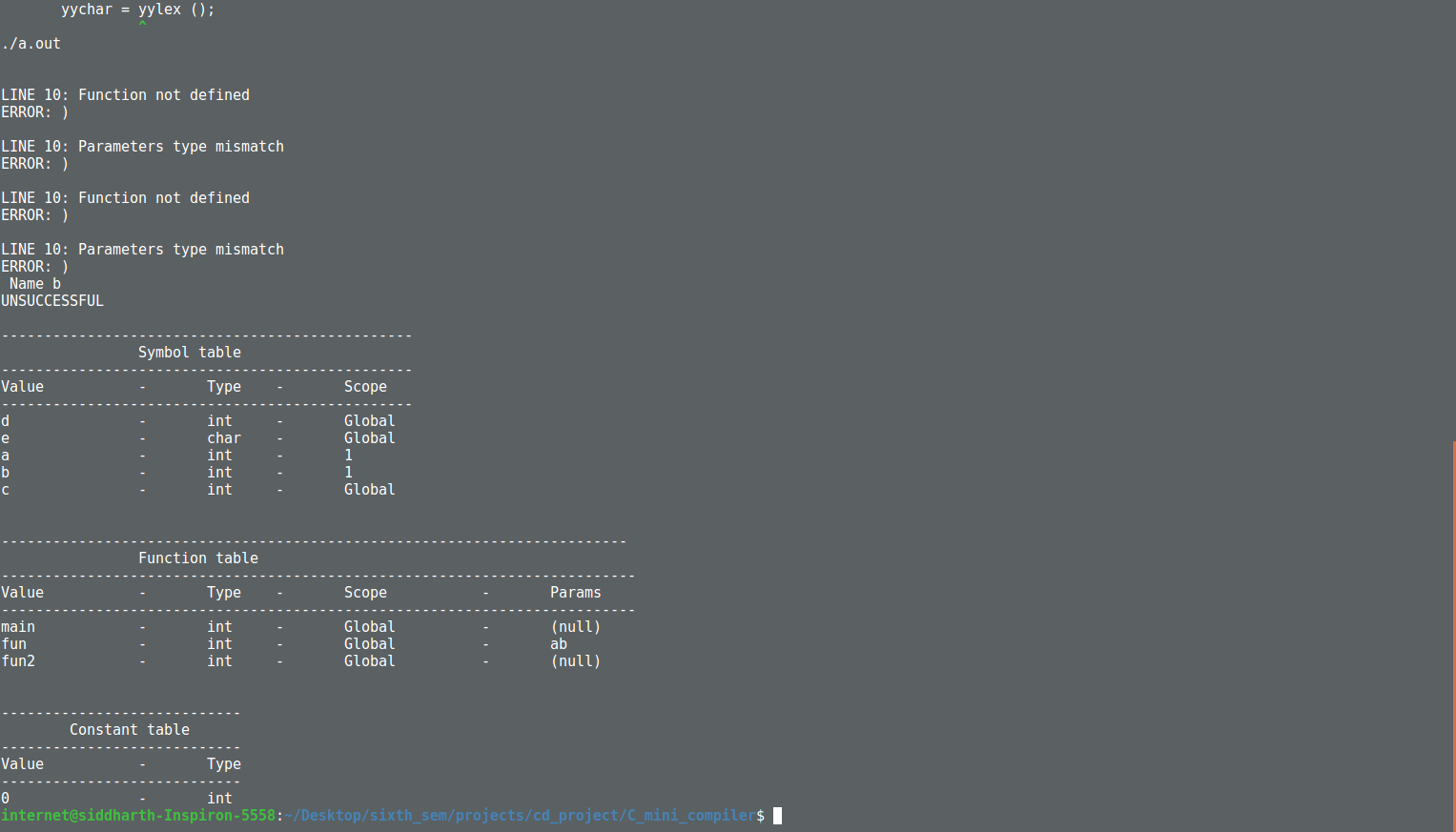
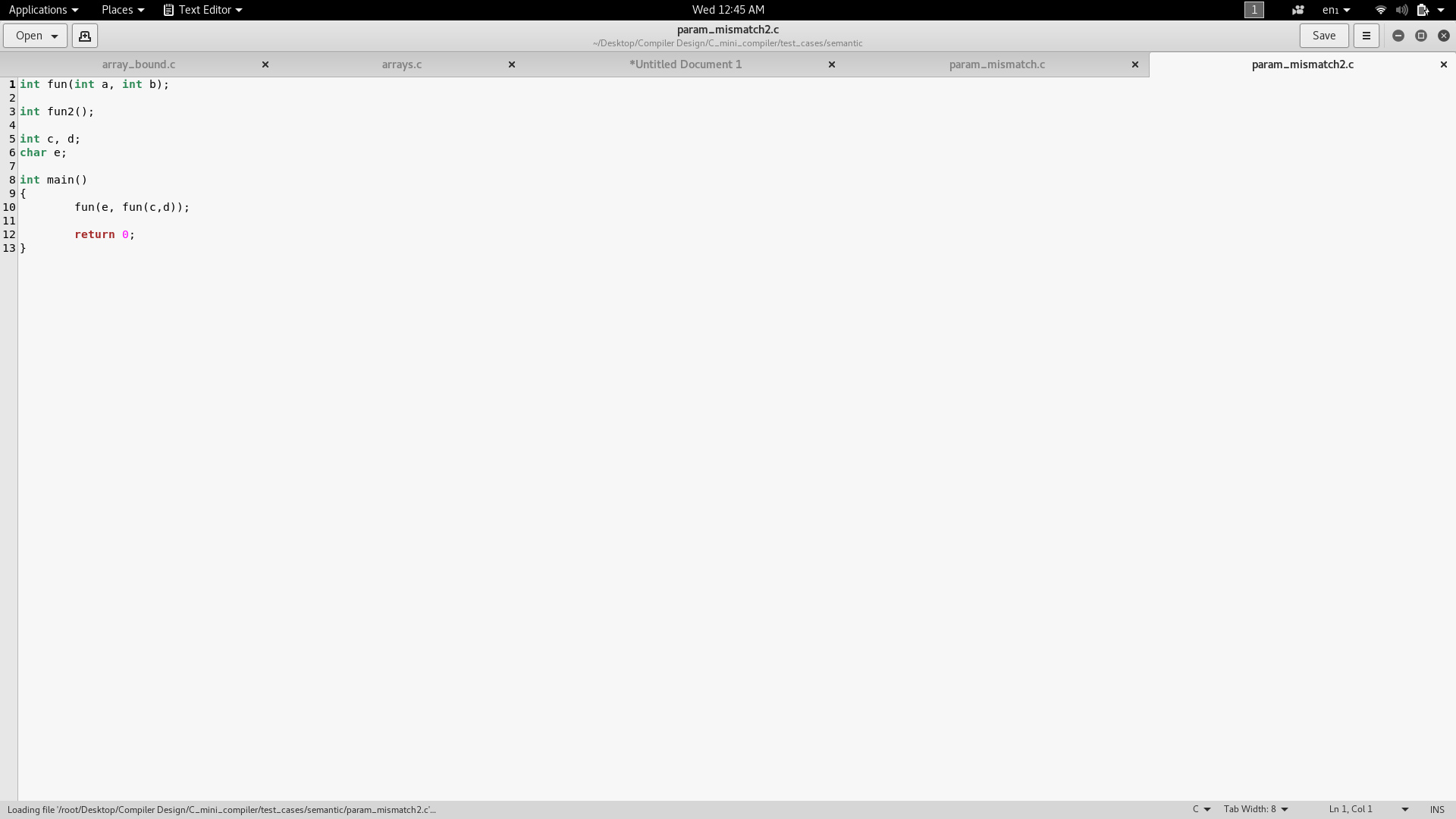
This is a simple case of mismatching parameters designed to check the effectiveness of the parameter checking constraints.





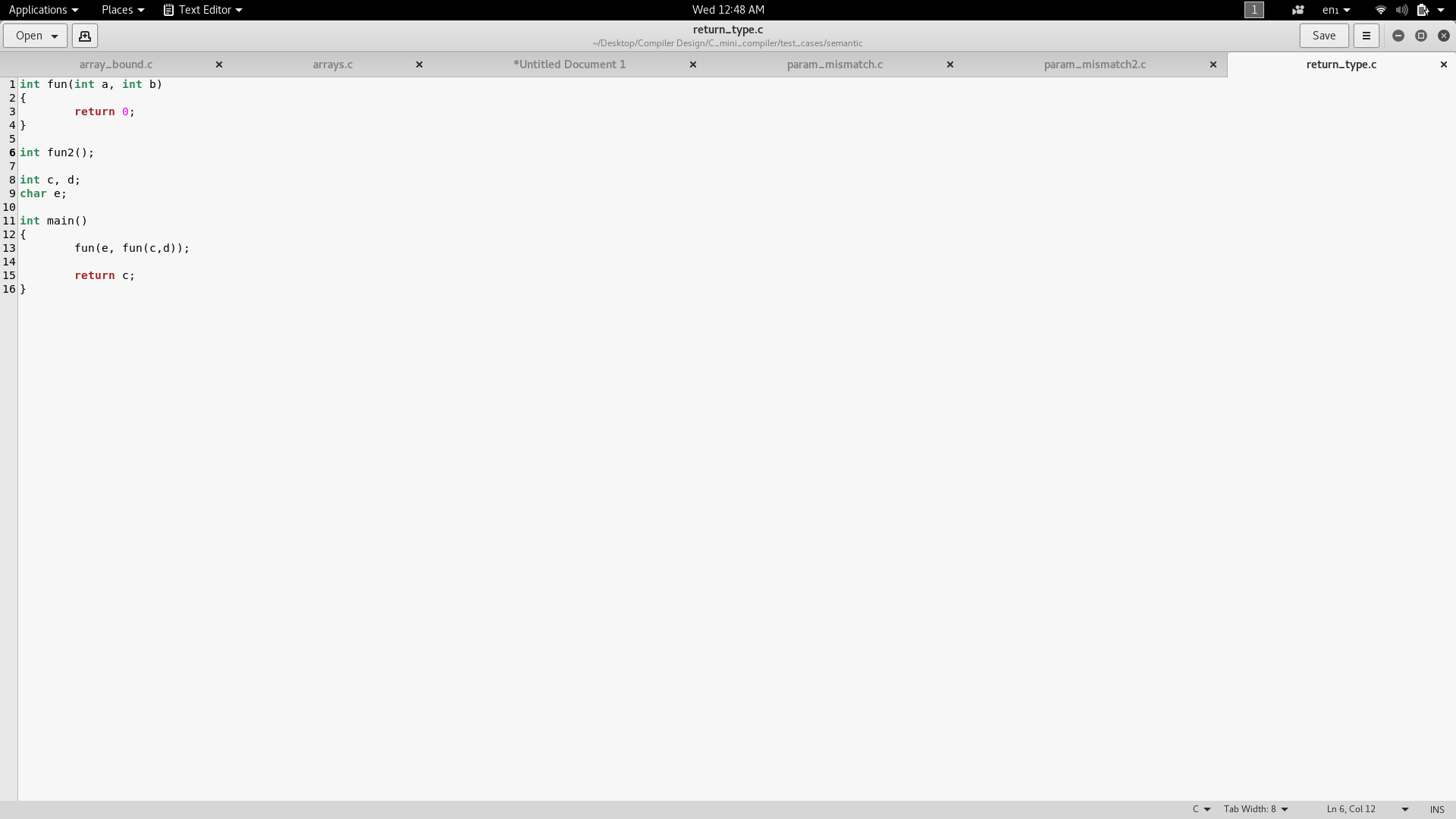
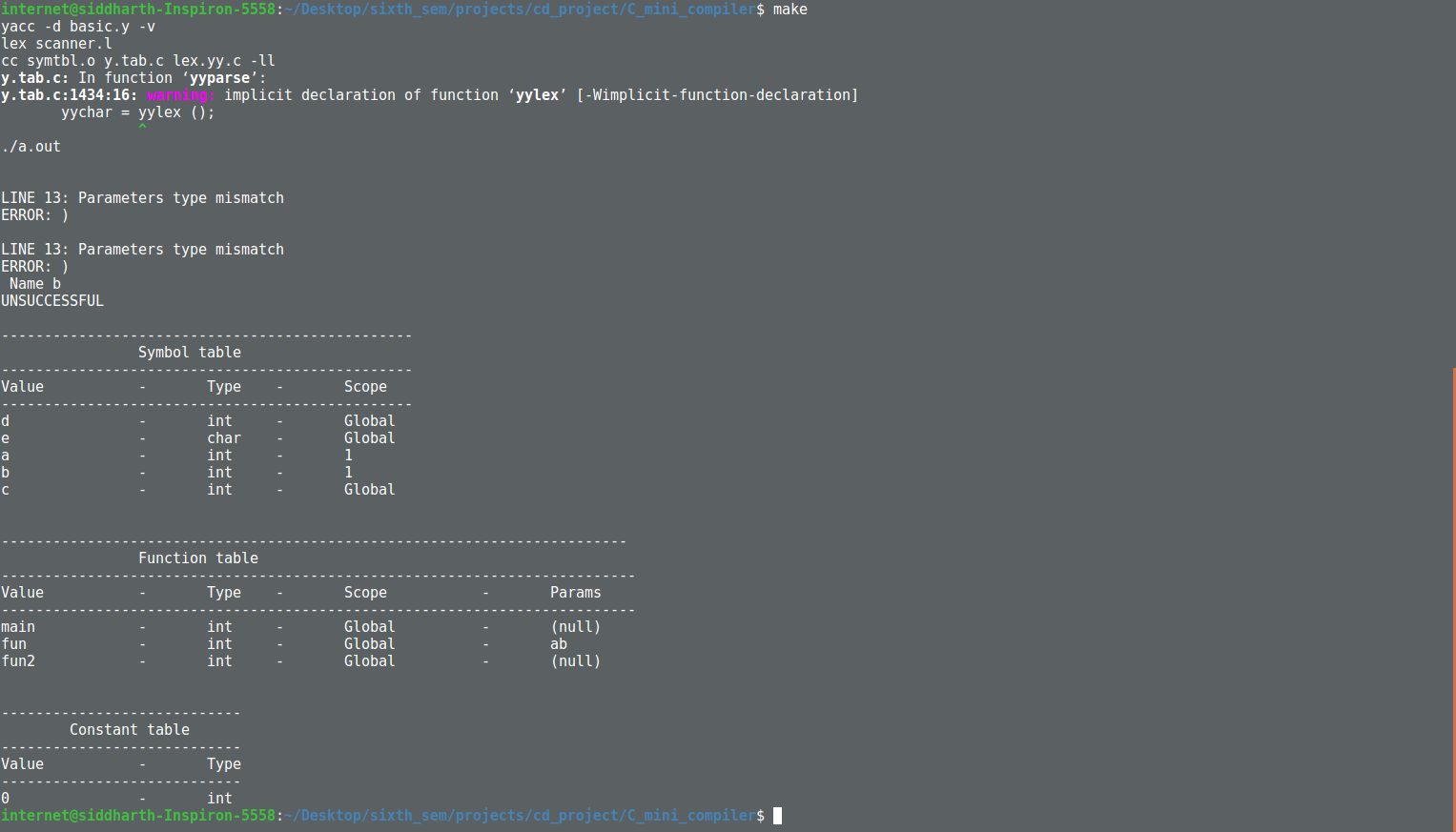
3) Parameter Mismatch 2

This is a more complex case of a parameter mismatch where a function’s return is used as a parameter to another function



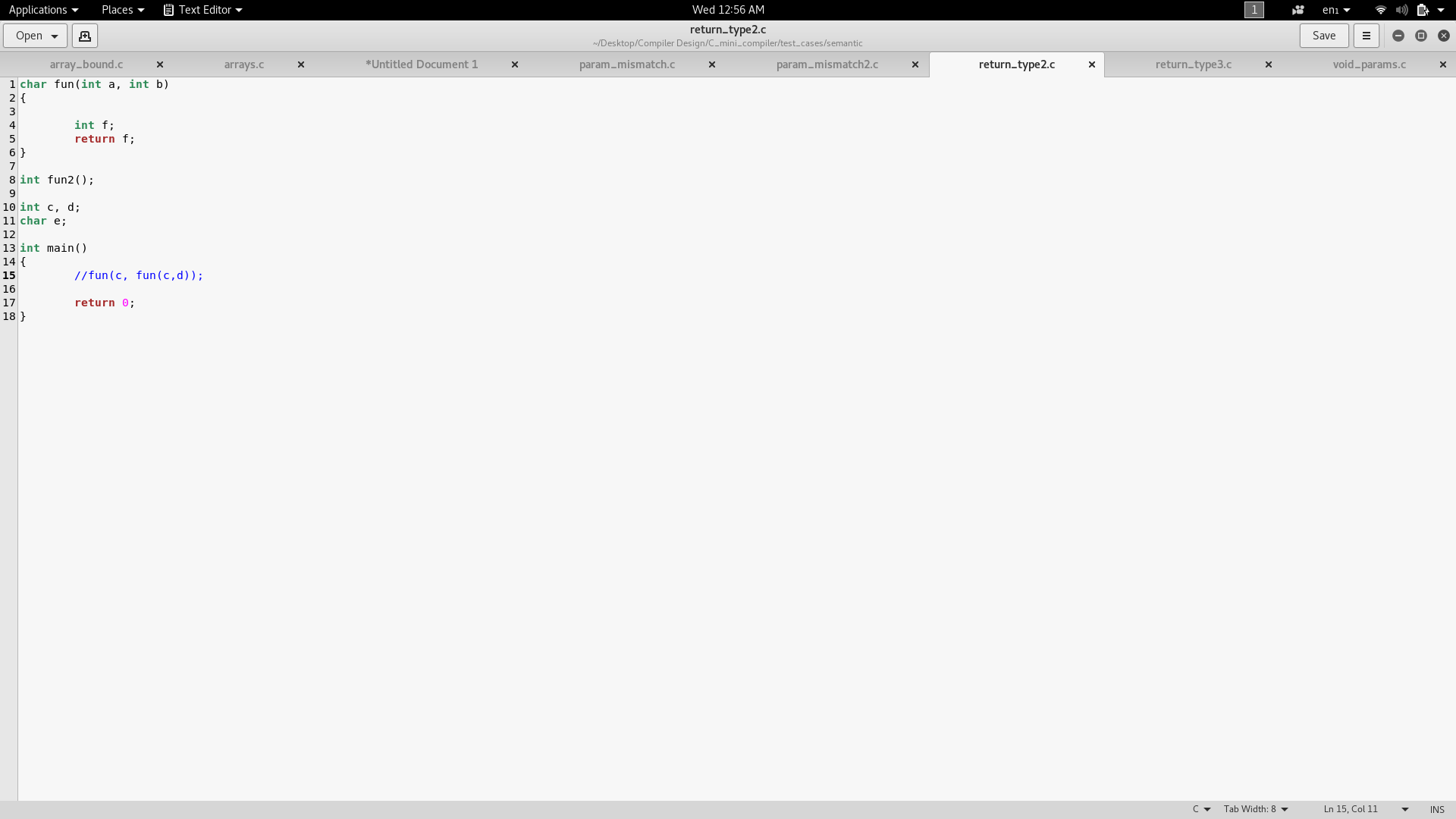
4) Return Type

This code tests the working of the return statement matching with the original return statement of the function. It also checks the function definition error.



5) Return type\_2

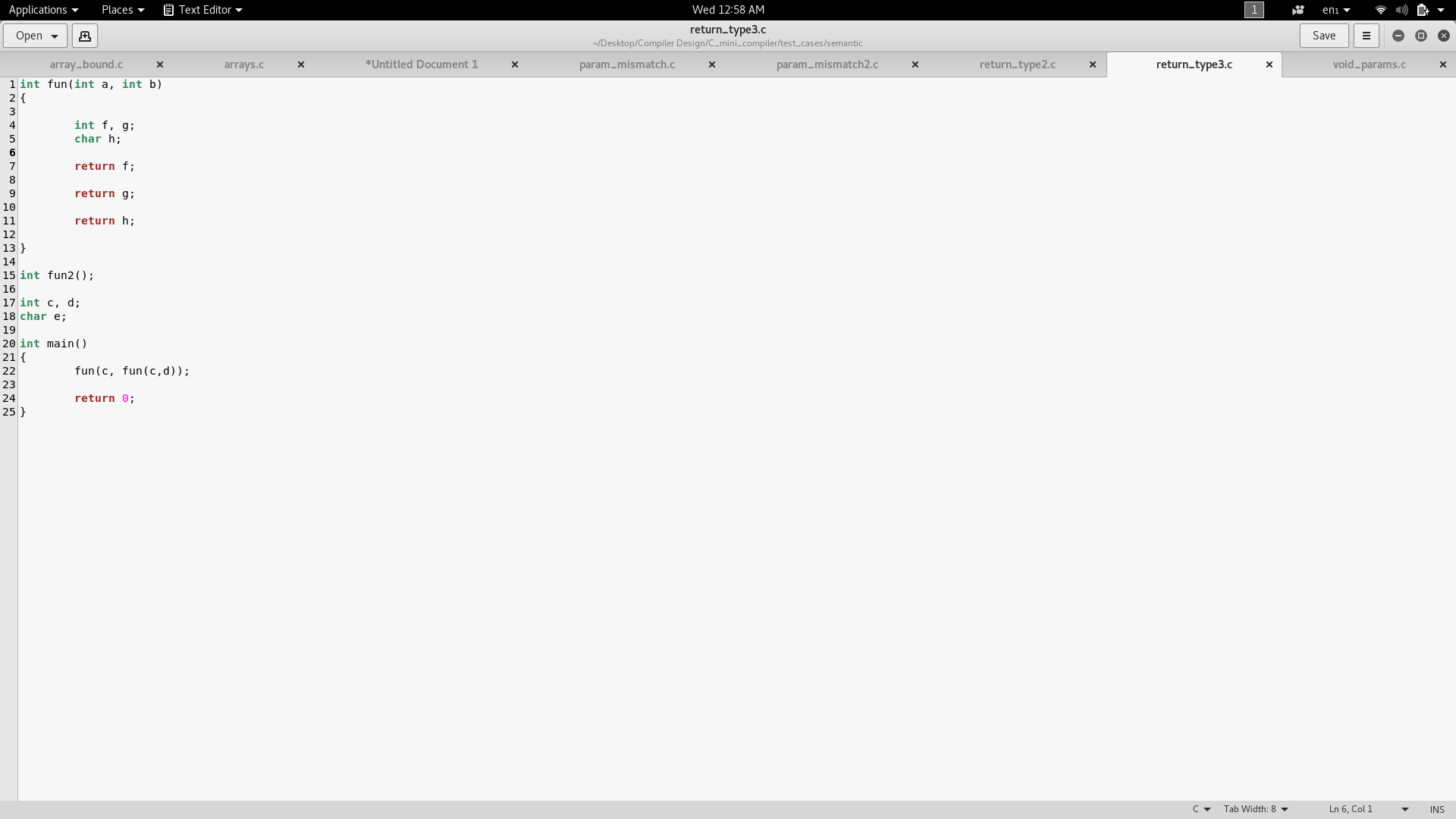
A slightly more complex check to the return type checks we have provided



On removal of the commented line, the output is,

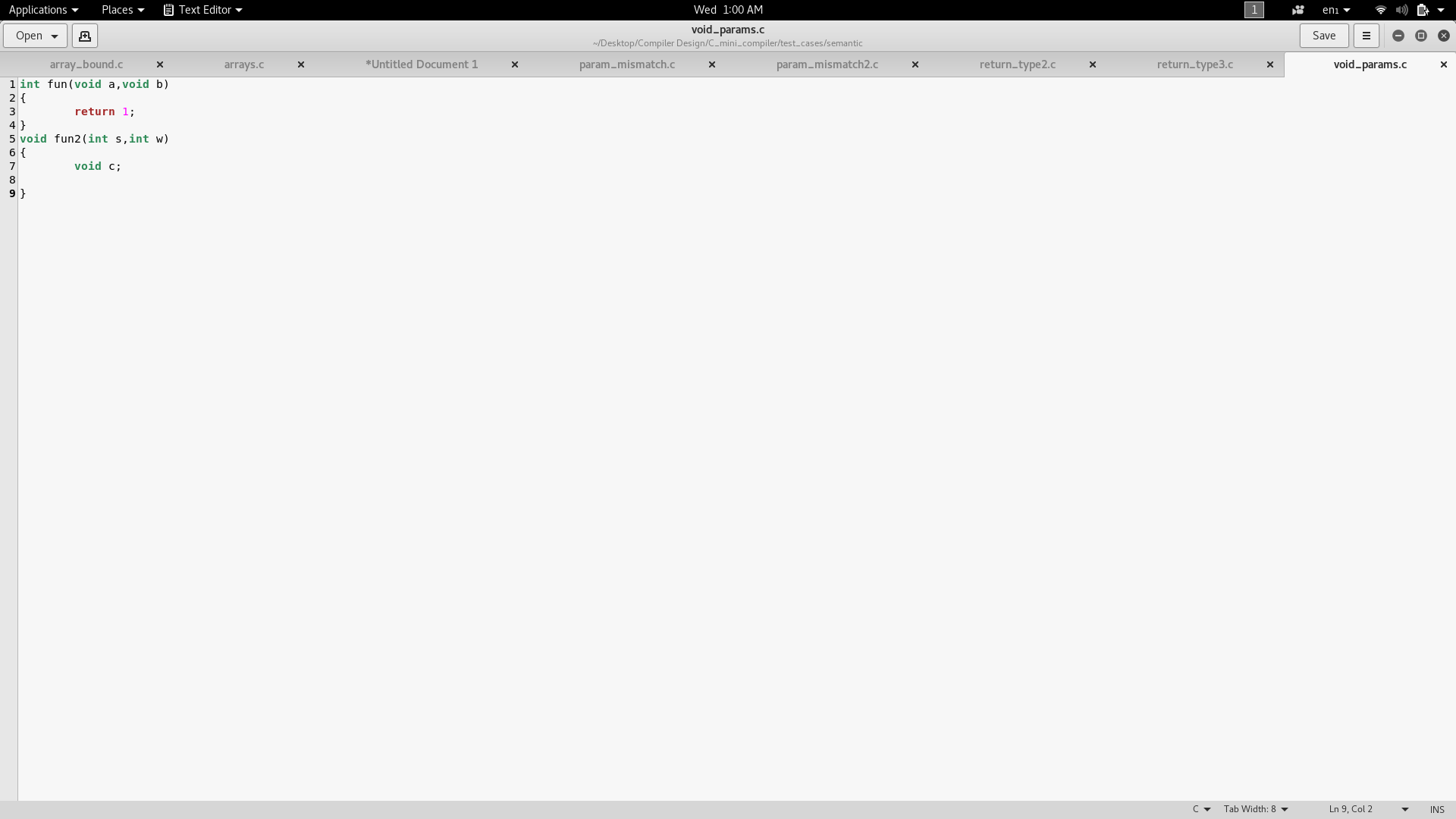
Return Type 3

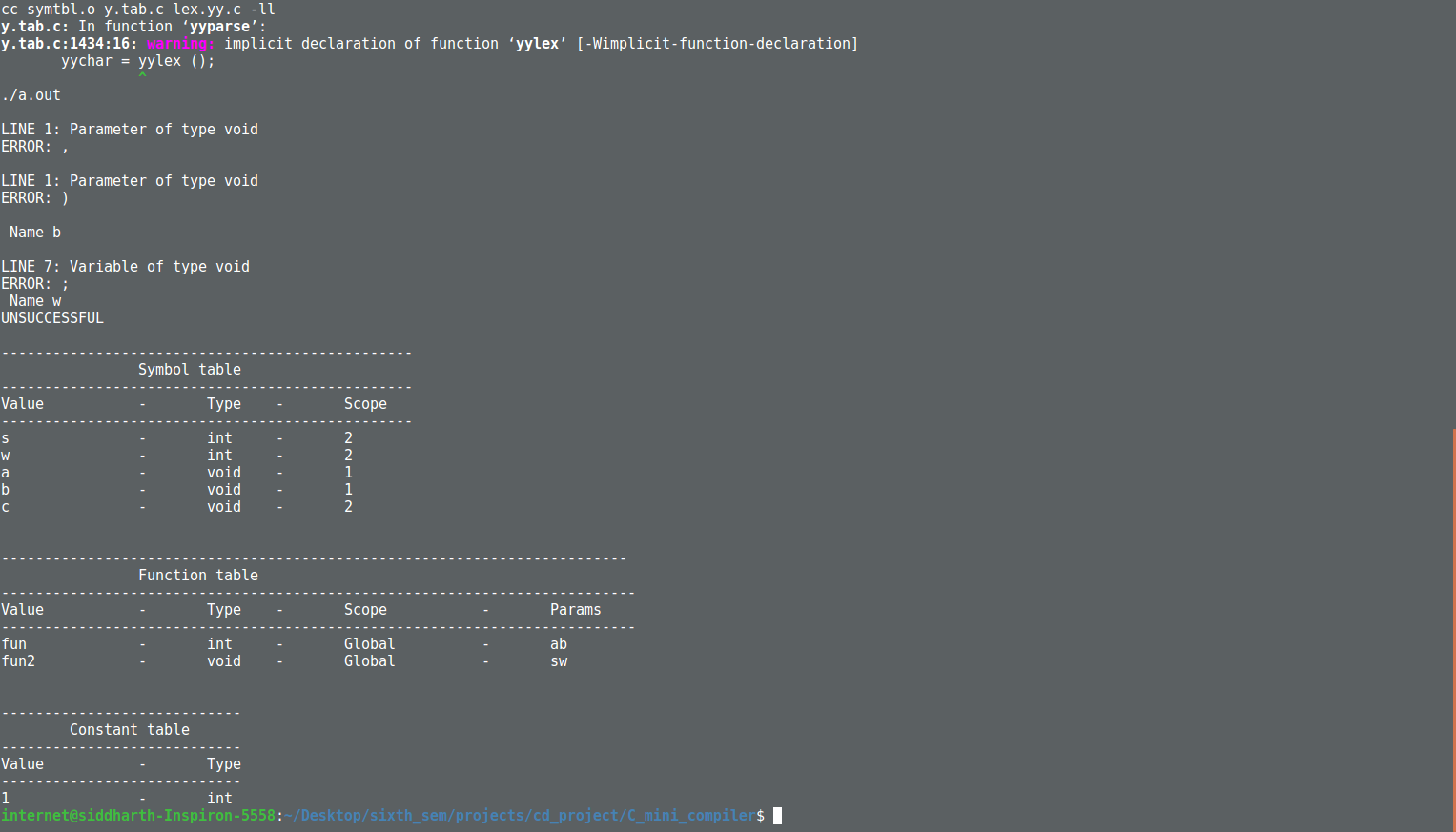
This is the most complicated test case we have to test the robustness of our return statement checks.



Void Parameters

This test case returns an error due to the presence of void parameters and void variables.





Conclusions

At the end of the third phase of the Compiler design project, we are able to successfully

parse through C programs which do not have any semantic errors. The program was also

able to identify and parse through all the C-language constructs mentioned in the abstract,

that is:

While loops

Arrays

Pointers

Basic data types(int, float,char) with relevant modifiers

The identifier symbol table contained the type of the identifier along with its name and scope. This allowed for scope related and type related error checking mechanisms.

The function symbol table contained the return type of the function along with the parameters it takes and its scope. This allowed us to create return type checks and parameter match checks when calling functions.

With the addition of all these features we have acheived the detection of all the errors mentioned in the introduction of this document. For future work, we intend to add more rules to try and convert this grammar into an Intermediate Code form such as 3 address code.