Report on

"C Mini Compiler for 'Ternary' and 'for' constructs"

Submitted in complete fulfillment of the requirements for ***Sem VI***

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Poorva Tiwari	**01FB16ECS251**
Rhiya Ramesh	**01FB16ECS301**
Rhythm Girdhar	**01FB16ECS302**

Under the guidance of

Kiran P
Professor
PES University, Bengaluru

January – May 2019

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

The language chosen is the C programming language. We have implemented the front end of the compiler for C language using Lex and Yacc for the following constructs :

1. For Loop
2. Ternary Operator

Using GNU C Compiler i.e gcc's latest version as our reference, we developed a mini-compiler that can handle basic C operations along with the ones chosen. Given source program in C can be translated to a symbol table, abstract syntax tree, intermediate code, and optimized intermediate code.

CHAPTER 2. ARCHITECTURE

- Used Flex/lex to create the scanner for our language.
- Used Bison/yacc to implement grammar rules to the token generated in the scanner phase.
- All token names are in capitals and everything else is in caps.
- The following are the operators and special characters implemented in our programming language:
    - Binary operators: + - * / %
    - Unary operators: ! ~ + - ++ -- (postfix and prefix)
    - Ternary operator: conditional operator
      variable = (expression) ? literal1 : literal2

- Ignore comments and white-spaces
    - Single line comments starting with //
    - Multi-line comments enclosed within /* …... */
- Types: int, float, char, void
- Constructs 'for' loop and ternary operators.
- Includes functions definitions and function calls.
- No conflicts and errors in our code/grammar.
- Maximum identifier length = 15
- Warnings and Error recovery:
    - Type Checking :
        - implicit conversion from float to char
        - implicit conversion from float to int
        - implicit conversion from char to float
    - Missing type specifier in the function definition. Defaults type to INT.
    - Division by zero. Assign INT_MAX and continue.
- Errors:

    - Redefinition of identifiers within the same scope
    - Use of undeclared identifiers
    - Invalid operands to ‘%’. Implicit conversion to INT
    - Truncate lengthy identifiers to length = 15
    - Syntax errors based on the specified grammar.
- All the errors and warnings are displayed along with line number
- If the same variable name is used within a nested scope, the *most closely nested loop* rule is used instead of giving an error (undeclared variable). It uses the previously defined value in the higher scope.
- Error handling related to scope and declaration.
- Since C doesn’t have a bool type, 0 and 1 are used to indicate false and true respectively. The test expression used in the ternary statement evaluates to 0 or 1 accordingly.
- Code Optimizations techniques used :
    - Common Subexpression elimination
    - Constant folding
    - Dead Code Elimination

CHAPTER 3. LITERATURE SURVEY

3.1
Course material shared for Compiler Design Course (especially ICG and Code optimisation)

3.2
https://www.lysator.liu.se/c/ANSI-C-grammar-y.html - Helped us write the grammar for our compiler

3.3
https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/ - Gave a quick overview of all the processes and codes involved in implementation of the compiler program.

3.4
https://softwareengineering.stackexchange.com/questions/165543/how-to-write-a-very-basic-compiler - A reference link for writing the code and taking ideas.

3.5
https://www.sigbus.info/how-i-wrote-a-self-hosting-c-compiler-in-40-days.html - A detailed guide link for writing the code and taking ideas.

CHAPTER 4. CONTEXT-FREE GRAMMAR

```
S : program
     ;
```

```
program
      : HASH INCLUDE '<' libraries '>' program
      | HASH INCLUDE HEADER_LITERAL   program
      | translation_unit
      ;


translation_unit
      : ext_dec
      | translation_unit ext_dec
      ;


ext_dec
      : declaration
      | function_definition
      ;


libraries
      : STDIO
      | STDLIB
      | MATH
      | STRING
      | TIME
      ;


compound_statement
      : '{' '}'
      | '{' block_item_list '}'
      ;


block_item_list
      : block_item
      | block_item_list block_item
      ;


block_item
      : declaration
      | statement
      | function_call ';'
      | RETURN expression_statement
      | printstat ';'
      ;


printstat
      : PRINT '(' STRING_LITERAL ')'
      | PRINT '(' STRING_LITERAL ',' expression ')'
      ;
```

```
declaration
      : type_specifier init_declarator_list ';'
      ;


statement
      : compound_statement
      | expression_statement
      | iteration_statement
      ;


iteration_statement
      : FOR '(' expression_statement expression_statement ')'
statement
      | FOR '(' expression_statement  expression_statement expression
')' statement
      | FOR '(' declaration expression_statement ')' statement

      | FOR '(' declaration expression_statement expression ')'
statement
      ;


type_specifier
      : VOID
      | CHAR
      | INT
      | FLOAT
      ;


init_declarator_list
      : init_declarator
      | init_declarator_list ',' init_declarator
      ;


init_declarator
      : IDENTIFIER  '=' assignment_expression

      | IDENTIFIER
      ;


assignment_expression
      : conditional_expression
      | unary_expression assignment_operator assignment_expression

      ;


assignment_operator
      : '='
      | ADD_ASSIGN
      | SUB_ASSIGN
      | MUL_ASSIGN
      | DIV_ASSIGN
      | MOD_ASSIGN
      ;


conditional_expression
      : equality_expression
      | equality_expression '?' expression ':' conditional_expression

      ;


expression_statement
      : ';'
      | expression ';'
      ;


expression
      : assignment_expression
      | expression ',' assignment_expression
      ;


primary_expression
      : IDENTIFIER
      | INTEGER_LITERAL
      | FLOAT_LITERAL
      | CHARACTER_LITERAL
      | '(' expression ')'
      ;


postfix_expression
      : primary_expression
      | postfix_expression INC_OP
      | postfix_expression DEC_OP
      ;


unary_expression
      : postfix_expression
      | unary_operator unary_expression
      ;


unary_operator
      : '+'
      | '-'
      | '!'
      | '~'
      | "INC_OP"
      | "DEC_OP"
```

```
        ;


equality_expression
        : relational_expression
        | equality_expression EQ_OP relational_expression
        | equality_expression NE_OP relational_expression
        ;


relational_expression
        : additive_expression
        | relational_expression '<' additive_expression
        | relational_expression '>' additive_expression
        | relational_expression LE_OP additive_expression
        | relational_expression GE_OP additive_expression
        ;


additive_expression
        : multiplicative_expression
        | additive_expression '+' multiplicative_expression
        | additive_expression '-' multiplicative_expression

        ;


multiplicative_expression
        : unary_expression
        | multiplicative_expression '*' unary_expression

        | multiplicative_expression '/' unary_expression

        | multiplicative_expression '%' unary_expression

        ;


function_definition
        : type_specifier declarator compound_statement

        | declarator compound_statement

        ;

function_call
        : declarator '(' identifier_list ')'
        | declarator '(' ')'
        ;

declarator
        : IDENTIFIER
        | declarator '(' parameter_list ')'
        | declarator '(' identifier_list ')'
        | declarator '(' ')'
```

```
	;


parameter_list
	: parameter_declaration
	| parameter_list ',' parameter_declaration
	;


parameter_declaration
	: type_specifier IDENTIFIER
	| type_specifier
	;


identifier_list
	: IDENTIFIER
	| identifier_list ',' IDENTIFIER
	;
```

CHAPTER 5. DESIGN AND IMPLEMENTATION

5.1 Design

Language: C
Tools : Lex and Yacc
Data Types : int, float, char, void
Constructs : for and ternary

- Symbol Table : Symbol table is a data structure that tracks the current bindings of identifiers for performing semantic checks and generating code efficiently. We have implemented the symbol table as a linked list of structures. The members of the structures include variable name, line of declaration, data type, value, scope. Every new variable encountered in the program is entered into the symbol table.

  ```
  Symbol(identifier, param, function), Name, Type,
  Scope, Line , Value)
  ```

- Abstract Syntax Tree : We have implemented a binary tree to represent the abstract syntax tree internally , we have executed this for ternary expressions and the for construct, and output the tree in pre-order manner.

- Intermediate Code Generation : The intermediate code is generated on the fly , as we parse the code and check its grammar , the intermediate code is generated. The representation of ICG is done by using quadruple table. The quadruple table consists of operator, first parameter, second parameter and the result.

- Code Optimization : To increase efficiency the code optimization is done on the generated ICG. We have implemented constant folding, common subexpression elimination and dead code elimination.

- Error Handling : In case of syntax error, the compilation is halted, and an error message along with the line number where error occured is displayed. Semantic errors such as division by zero, multiple declaration of the same variable, invalid assignment, scope errors are also explicitly pointed out. All of which are specified as production rules within the grammar. Type checking is done and if the types do not match in the expressions, a warning is displayed and implicit conversion is done to recover from errors.

5.2 Implementation :

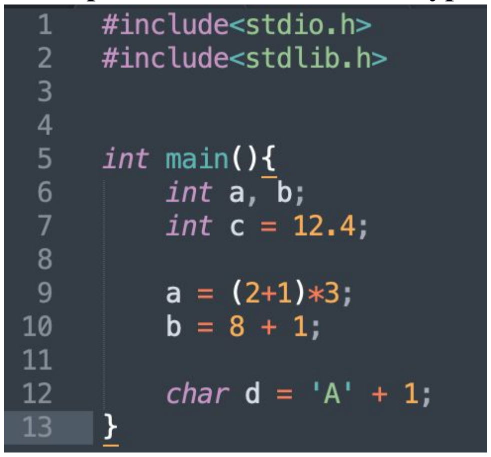
- The tools we have used for implementing the code are lex and yacc.
- The lex file has all the tokens specified with the help of regular expressions and the yacc file has grammar rules with corresponding actions.
- As the code is being parsed, the tokens are generated and comments and extra spaces are ignored. For every new variable encountered, it is entered into the symbol table along with its attributes.
- A union is used which consists of int, float and char and void type to use it for assigning data type of each variable.
- Semantics Analysis uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct (type checking) and update it accordingly.
- The scope check is done by having a variable which increments on every level of nesting. In this manner, the scope is checked for each variable and error messages are displayed if anything is used out of scope.
- Integer, float and char literal constants.
- We have written appropriate rules to check for semantic validity (type checking, declare before use, etc.)
- Type checking is done by checking type of each variable in the expression and implicit conversions are done to avoid errors along with display of warnings.
- Variables must be declared and can only be used in ways that are acceptable for the declared type.
- Once parsing is successful, we generate an abstract tree and it is shown in pre-order manner.
- The intermediate code generation also happens on the fly.
- After generating intermediate code, optimization is done by doing dead code elimination, constant folding and common subexpression elimination.
- Code optimisation uses information present in symbol table for machine dependent optimization.

Commands to execute the code:

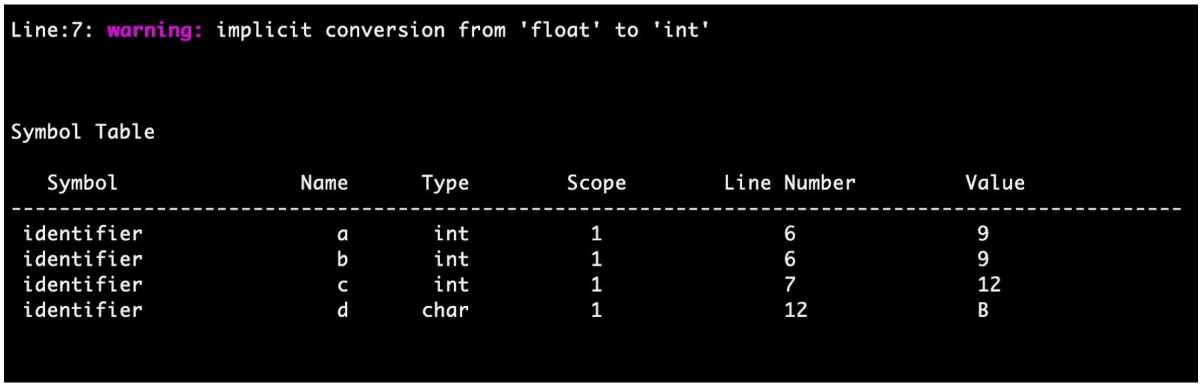
```
lex ast.l
yacc -d ast.y
gcc lex.yy.c y.tab.c -ll -ly -o ast.o
lex icg.l
yacc -d icg.y
gcc lex.yy.c y.tab.c -ll -ly -o icg.o


./ast.o <test1.c
./icg.o <test1.c
python optimize.py icg.txt
```

CHAPTER 6. SNAPSHOTS

6.1 Expression evaluation and type conversion.

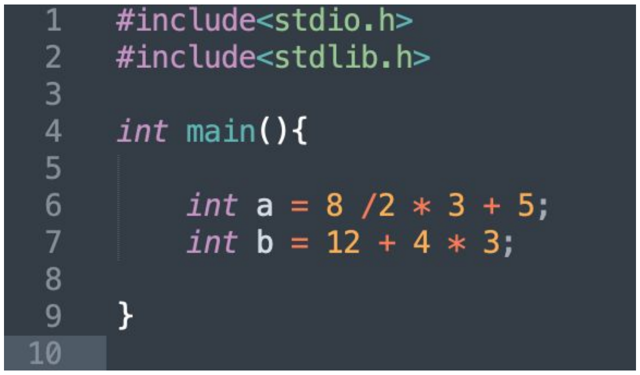


```
#include<stdio.h>
#include<stdlib.h>


int main(){
    int a, b;
    int c = 12.4;

    a = (2+1)*3;
    b = 8 + 1;

    char d = 'A' + 1;
}
```

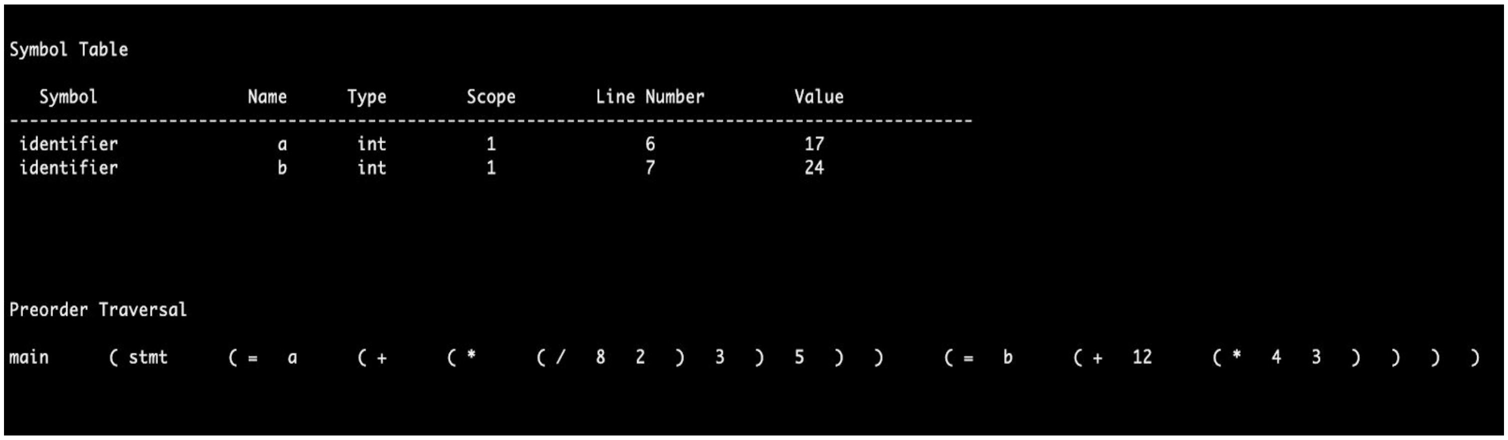


```
Line:7: warning: implicit conversion from 'float' to 'int'


Symbol Table

   Symbol                    Name       Type        Scope        Line Number          Value
-----------------------------------------------------------------------------------------------
 identifier                     a        int          1               6                9
 identifier                     b        int          1               6                9
 identifier                     c        int          1               7                12
 identifier                     d       char          1               12               B
```

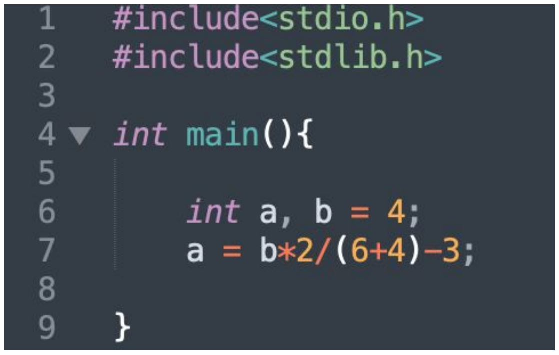
6.2 Expression evaluation and AST to demonstrate precedence and associativity. It also updates values in symbol table.



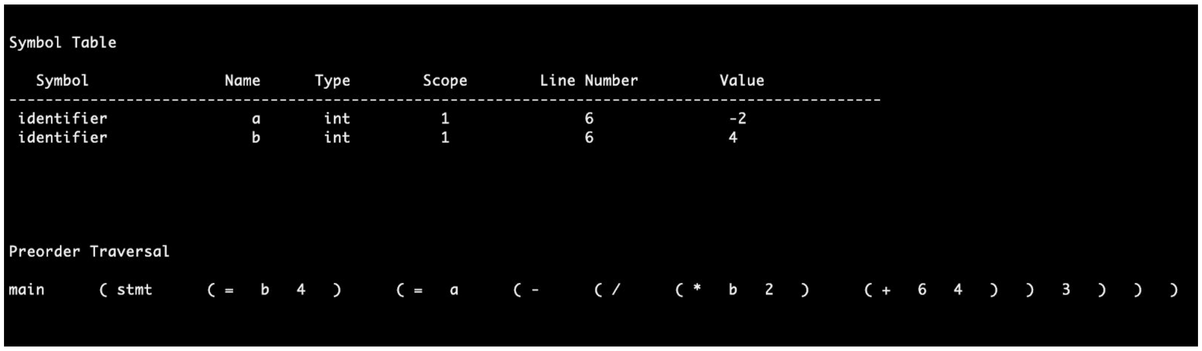
```c
#include<stdio.h>
#include<stdlib.h>

int main(){

    int a = 8 /2 * 3 + 5;
    int b = 12 + 4 * 3;

}
```



```
Symbol Table

   Symbol                Name      Type        Scope        Line Number         Value
-----------------------------------------------------------------------------------------------
 identifier                a       int           1             6                 17
 identifier                b       int           1             7                 24



Preorder Traversal

main     ( stmt     ( =   a     ( +     ( *     ( /   8   2   )   3   )   5   )   )     ( =   b     ( +   12     ( *   4   3   )   )   )   )
```

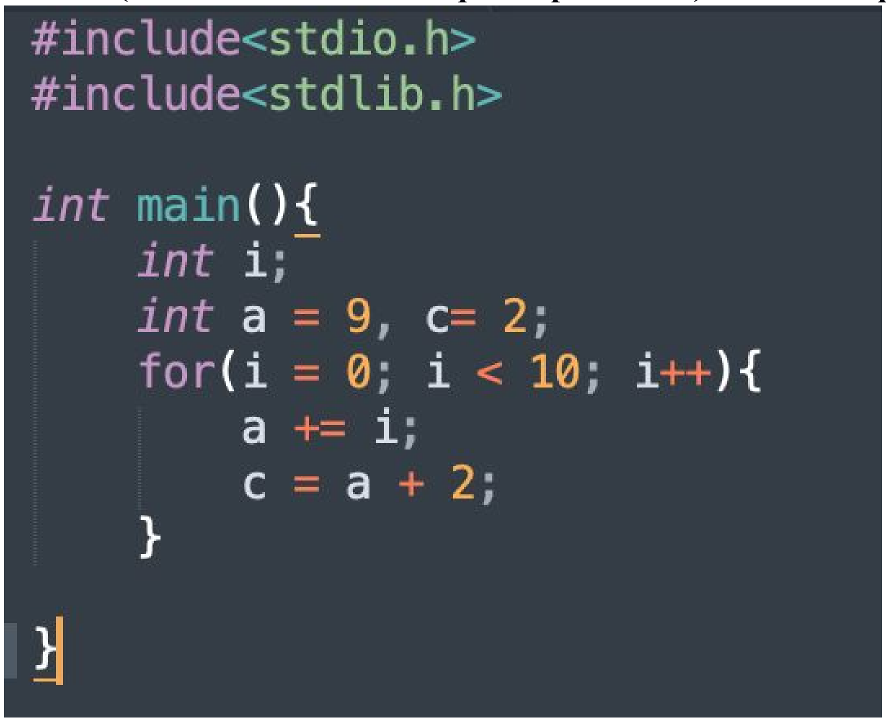


```c
#include<stdio.h>
#include<stdlib.h>

int main(){

    int a, b = 4;
    a = b*2/(6+4)-3;

}
```

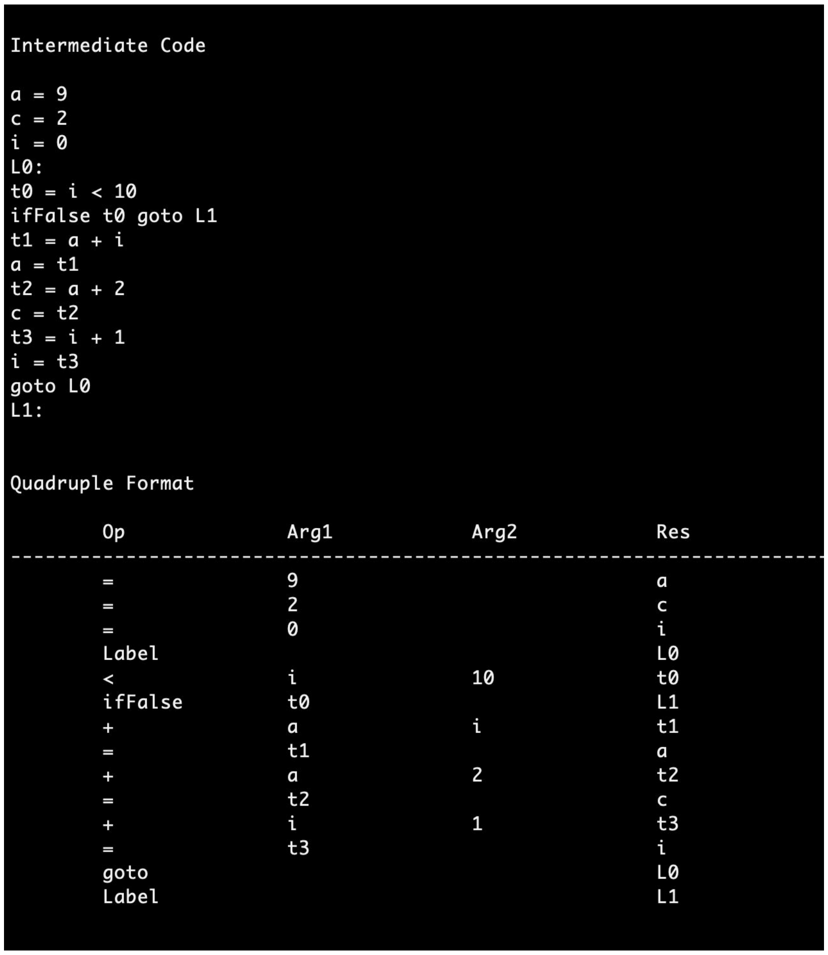


```
Symbol Table

   Symbol                Name      Type        Scope        Line Number         Value
-----------------------------------------------------------------------------------------------
 identifier                a       int           1             6                 -2
 identifier                b       int           1             6                 4



Preorder Traversal

main     ( stmt     ( =   b   4   )     ( =   a     ( -     ( /     ( *   b   2   )     ( +   6   4   )   )   3   )   )   )
```

6.3 ICG (three address code and quadruple format) for FOR loop

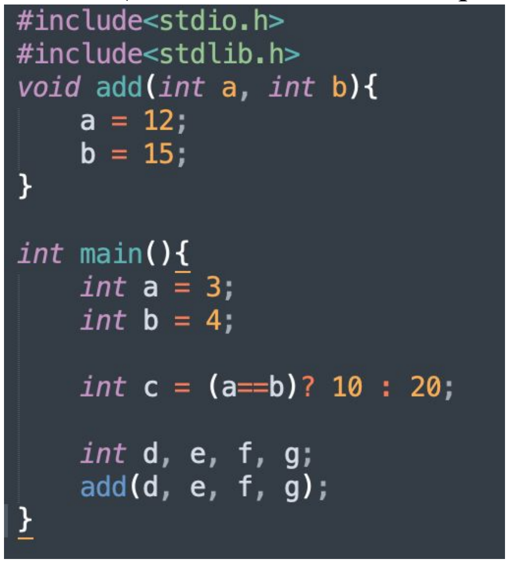


```c
#include<stdio.h>
#include<stdlib.h>

int main(){
    int i;
    int a = 9, c= 2;
    for(i = 0; i < 10; i++){
        a += i;
        c = a + 2;
    }

}
```

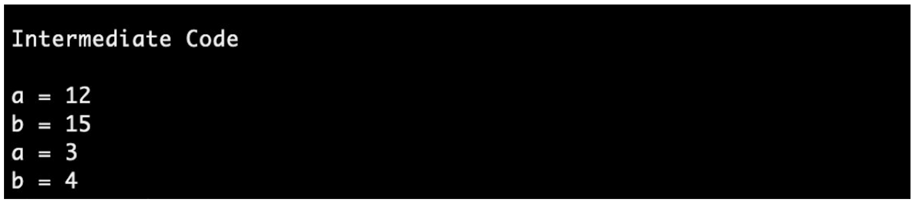


```
Intermediate Code

a = 9
c = 2
i = 0
L0:
t0 = i < 10
ifFalse t0 goto L1
t1 = a + i
a = t1
t2 = a + 2
c = t2
t3 = i + 1
i = t3
goto L0
L1:


Quadruple Format

        Op              Arg1            Arg2            Res
-----------------------------------------------------------------------
        =               9                               a
        =               2                               c
        =               0                               i
        Label                                           L0
        <               i               10              t0
        ifFalse         t0                              L1
        +               a               i               t1
        =               t1                              a
        +               a               2               t2
        =               t2                              c
        +               i               1               t3
        =               t3                              i
        goto                                            L0
        Label                                           L1
```

6.4 ICG (three address code and quadruple format) for ternary loop and function calls.

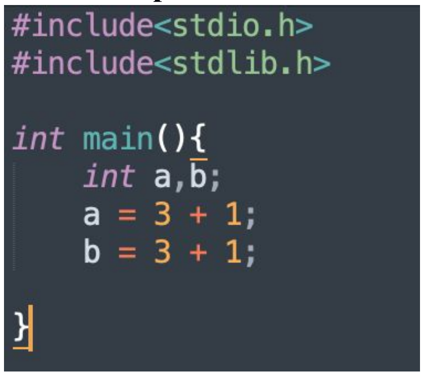


```c
#include<stdio.h>
#include<stdlib.h>
void add(int a, int b){
    a = 12;
    b = 15;
}

int main(){
    int a = 3;
    int b = 4;

    int c = (a==b)? 10 : 20;

    int d, e, f, g;
    add(d, e, f, g);
}
```

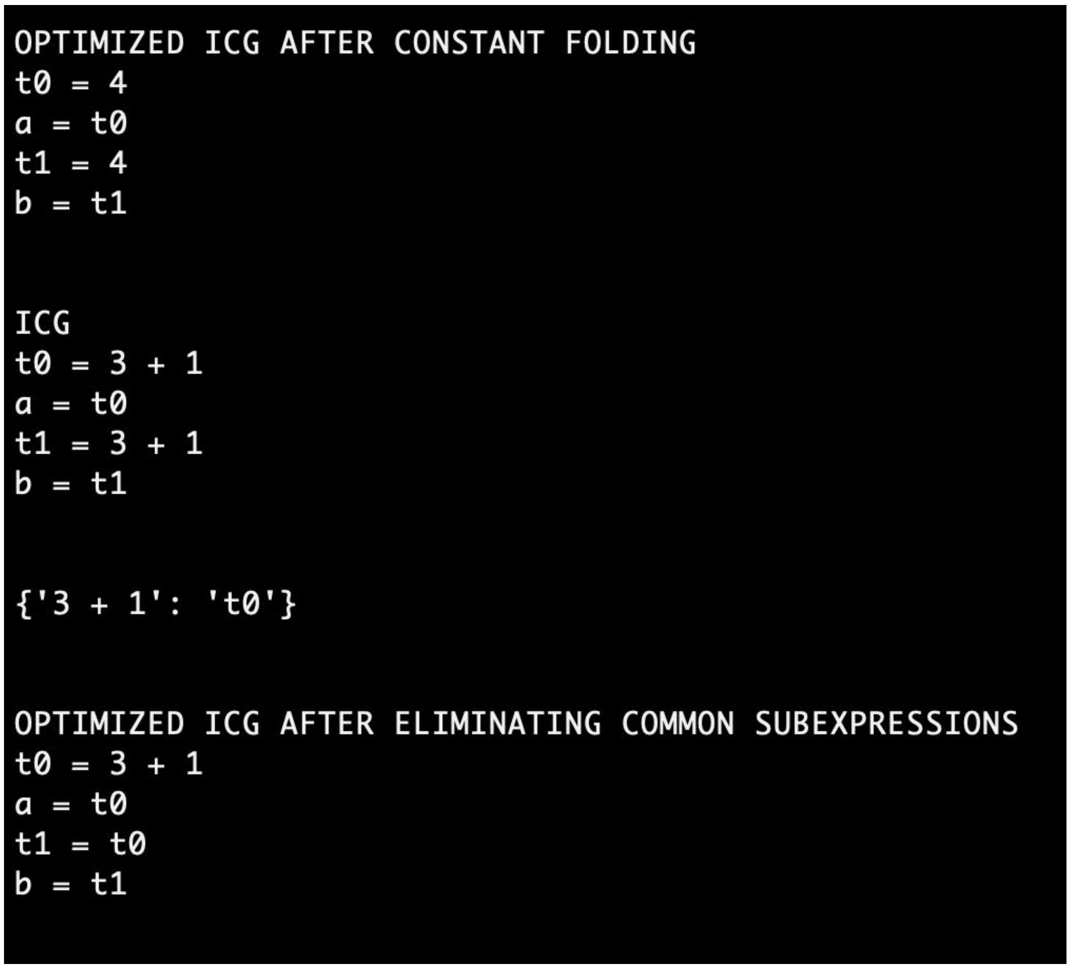


```
Intermediate Code

a = 12
b = 15
a = 3
b = 4
```

6.5 Code Optimization

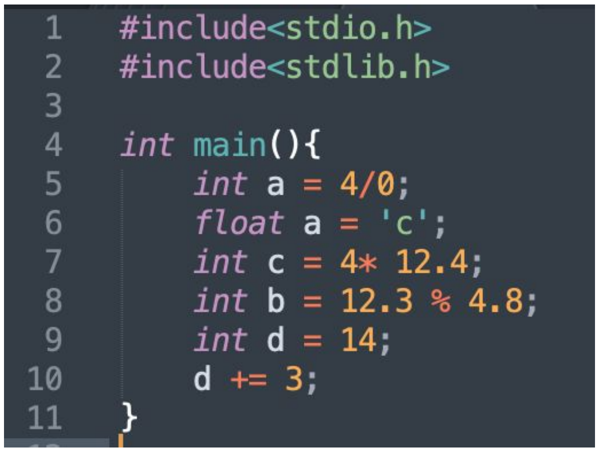


```c
#include<stdio.h>
#include<stdlib.h>

int main(){
    int a,b;
    a = 3 + 1;
    b = 3 + 1;

}
```

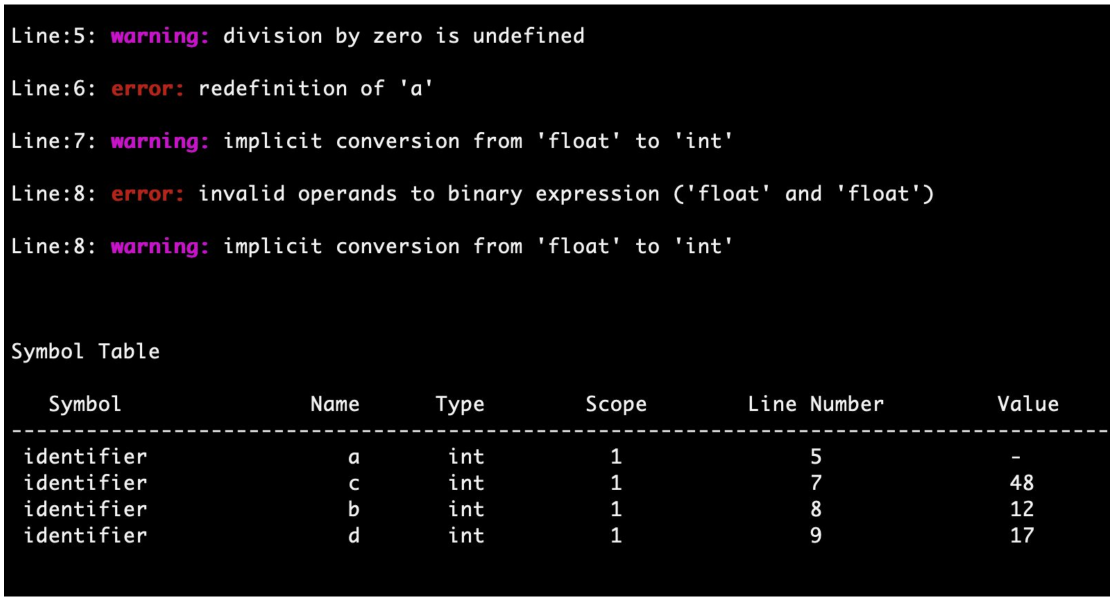


```
OPTIMIZED ICG AFTER CONSTANT FOLDING
t0 = 4
a = t0
t1 = 4
b = t1


ICG
t0 = 3 + 1
a = t0
t1 = 3 + 1
b = t1


{'3 + 1': 't0'}


OPTIMIZED ICG AFTER ELIMINATING COMMON SUBEXPRESSIONS
t0 = 3 + 1
a = t0
t1 = t0
b = t1
```

6.6 Implicit conversion and invalid operations. Redefinition error

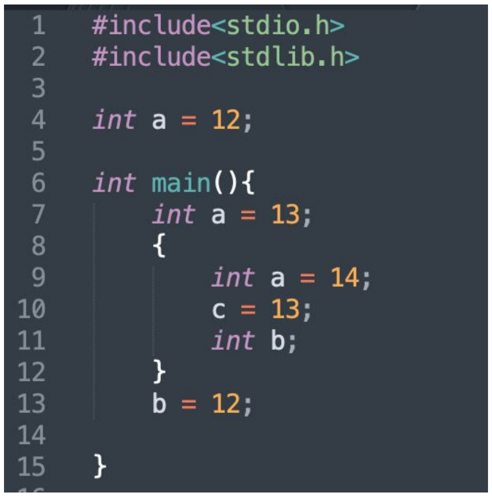


```
1   #include<stdio.h>
2   #include<stdlib.h>
3
4   int main(){
5       int a = 4/0;
6       float a = 'c';
7       int c = 4* 12.4;
8       int b = 12.3 % 4.8;
9       int d = 14;
10      d += 3;
11  }
```

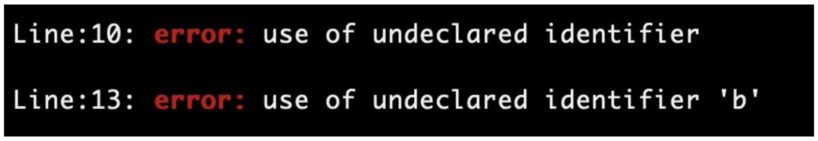


```
Line:5: warning: division by zero is undefined

Line:6: error: redefinition of 'a'

Line:7: warning: implicit conversion from 'float' to 'int'

Line:8: error: invalid operands to binary expression ('float' and 'float')

Line:8: warning: implicit conversion from 'float' to 'int'



Symbol Table

   Symbol                 Name      Type        Scope        Line Number          Value
-----------------------------------------------------------------------------------------
 identifier                 a        int          1              5                  -
 identifier                 c        int          1              7                 48
 identifier                 b        int          1              8                 12
 identifier                 d        int          1              9                 17
```

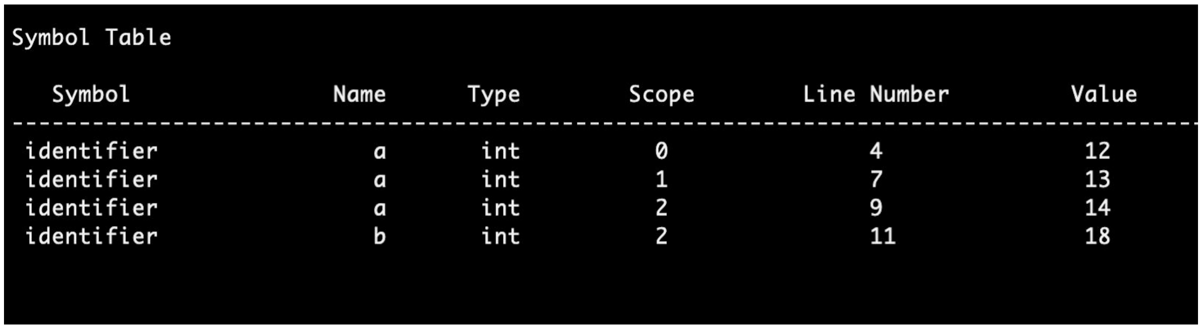
6.7 Handling errors related to the scope



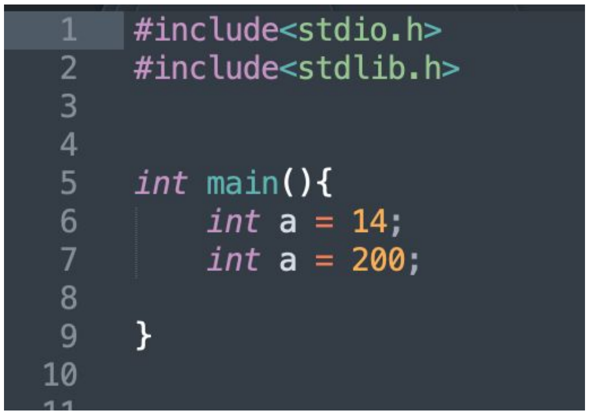
```
#include<stdio.h>
#include<stdlib.h>

int a = 12;

int main(){
    int a = 13;
    {
        int a = 14;
        c = 13;
        int b;
    }
    b = 12;

}
```



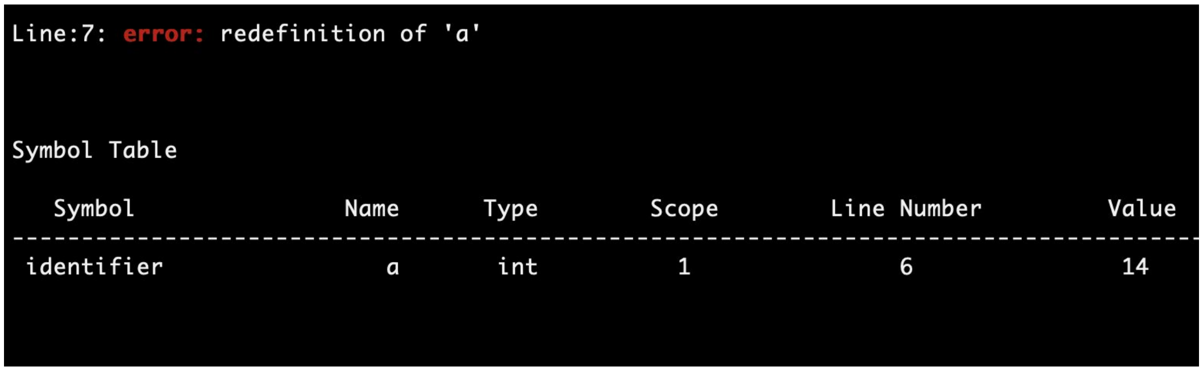
```
Line:10: error: use of undeclared identifier

Line:13: error: use of undeclared identifier 'b'
```



```
Symbol Table

   Symbol                  Name       Type         Scope        Line Number         Value
------------------------------------------------------------------------------------------
 identifier                   a        int           0              4                12
 identifier                   a        int           1              7                13
 identifier                   a        int           2              9                14
 identifier                   b        int           2              11               18
```

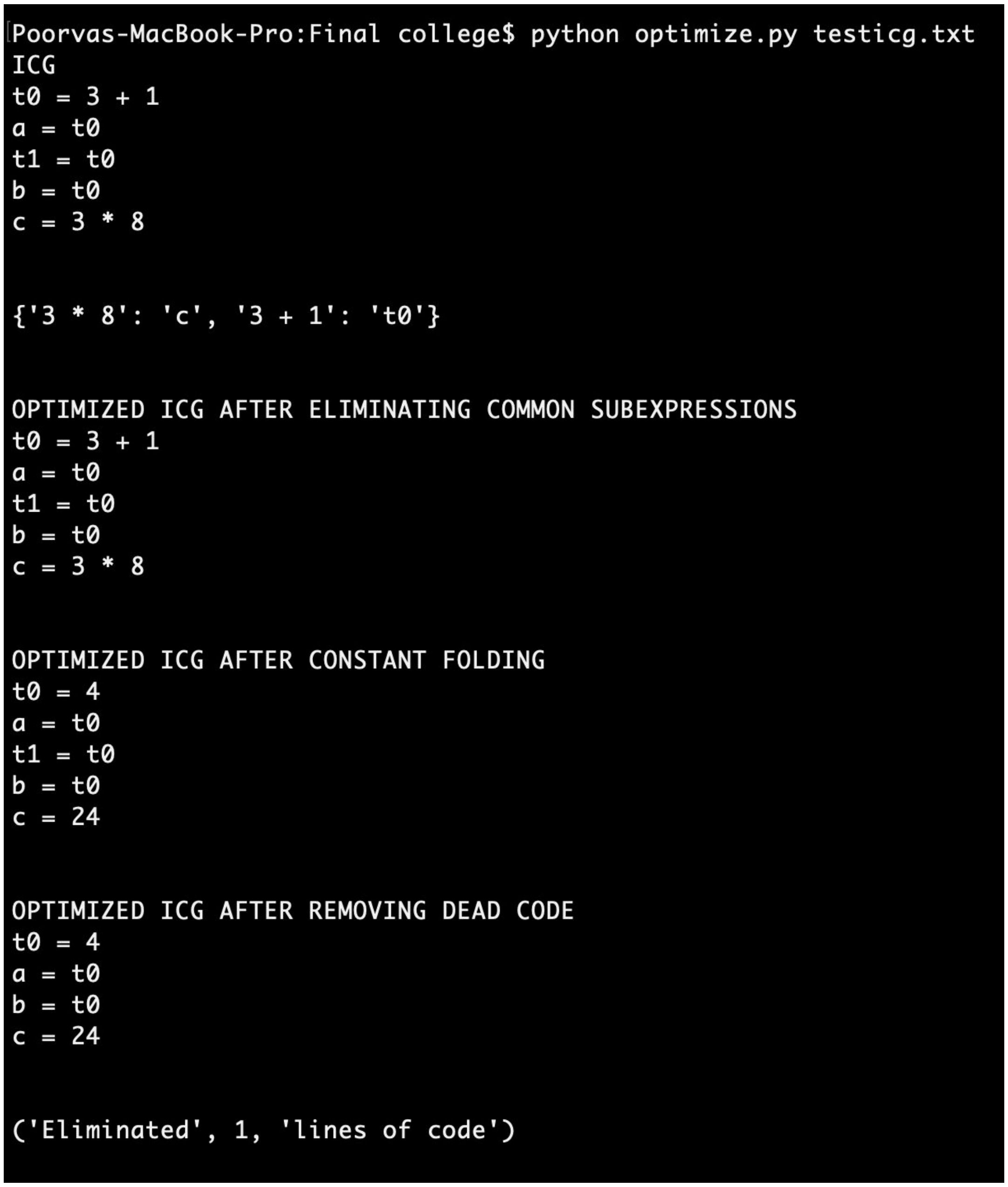


```
#include<stdio.h>
#include<stdlib.h>


int main(){
    int a = 14;
    int a = 200;

}
```



```
Line:7: error: redefinition of 'a'


Symbol Table

   Symbol                  Name       Type         Scope         Line Number         Value
------------------------------------------------------------------------------------------
 identifier                   a        int           1               6                 14
```

6.8 Code optimization



```
[Poorvas-MacBook-Pro:Final college$ python optimize.py testicg.txt
ICG
t0 = 3 + 1
a = t0
t1 = t0
b = t0
c = 3 * 8


{'3 * 8': 'c', '3 + 1': 't0'}


OPTIMIZED ICG AFTER ELIMINATING COMMON SUBEXPRESSIONS
t0 = 3 + 1
a = t0
t1 = t0
b = t0
c = 3 * 8


OPTIMIZED ICG AFTER CONSTANT FOLDING
t0 = 4
a = t0
t1 = t0
b = t0
c = 24


OPTIMIZED ICG AFTER REMOVING DEAD CODE
t0 = 4
a = t0
b = t0
c = 24


('Eliminated', 1, 'lines of code')
```

CHAPTER 7. RESULTS AND CONCLUSION

The lex and yacc codes are compiled and executed by the following terminal commands to parse the given input file.

```
lex ast.l
yacc -d ast.y
gcc lex.yy.c y.tab.c -ll -ly -o ast.o
lex icg.l
yacc -d icg.y
gcc lex.yy.c y.tab.c -ll -ly -o icg.o
```

```
./ast.o <test1.c
./icg.o <test1.c
python optimize.py icg.txt
```

After parsing, if there are errors then the line numbers of those errors are displayed along with a ‘parsing failed’ on the terminal. Otherwise, a ‘parsing complete’ message is displayed on the console. The symbol table with stored & updated values is always displayed, irrespective of errors. Also, the three address codes along with the temporary variables are also displayed along with the flow of the conditional and iterative statements.

CHAPTER 8. SHORTCOMINGS

- Traversing the symbol table is time consuming as we have implemented a linked list, no random access possible.
- Currently, the abstract syntax tree is represented as a flat list in pre-order manner. The interpretation might be difficult in this case.

CHAPTER 9. FUTURE ENHANCEMENTS

Include:
- Other looping constructs like while, do-while.
- Conditional jumps like goto, continue and break.
- Conditional statements like if-else and switch case.
- More data types.

REFERENCES

a. Compilers – Principles, Techniques, and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
b. https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/
c. http://web.cs.wpi.edu/~kal/courses/compilers/
d. https://www.tutorialspoint.com/compiler_design/compiler_design_intermediate_c ode_generations.htm