TABLE OF CONTENTS

Chapter No.	Title	Page No.
1.	INTRODUCTION	02
2.	ARCHITECTURE OF LANGUAGE:	07
3.	LITERATURE SURVEY	08
4.	CONTEXT FREE GRAMMAR	08
5.	DESIGN STRATEGY ● SYMBOL TABLE CREATION ● ABSTRACT SYNTAX TREE ● INTERMEDIATE CODE GENERATION ● CODE OPTIMIZATION ● ERROR HANDLING	13
6.	IMPLEMENTATION DETAILS : ● SYMBOL TABLE CREATION ● ABSTRACT SYNTAX TREE ● INTERMEDIATE CODE GENERATION ● CODE OPTIMIZATION ● ERROR HANDLING ● Provide instructions on how to build and run your computer	18
7.	RESULTS AND possible shortcomings of your Mini-Compiler	20
8.	SNAPSHOTS	20
9.	CONCLUSIONS	26
10.	FURTHER ENHANCEMENTS	26
REFERENCES/BIBLIOGRAPHY		27

INTRODUCTION
C MINI COMPILER
CONSTRUCTS : if-else and while

- The mini compiler is built for **C language** (for the constructs if-else and while)
- The compiler also handles other constructs like :
  - basic types like int, float and char.
  - 1D and 2D arrays

- The input of the compiler is a C program and the output of different stages are:
  - Tokens Generated
  - Validity of the Program (No errors/Error Handling)
  - C Program free of comments
  - Symbol Table
  - AST
  - Intermediate Code
  - Optimized Intermediate Code

- **Input file :**

```
int main()
{
    int a=10;
    int b=0;
    //comment 1
    while(a>0)
    {
     a=a-1;
     if(a==5)
     {
          b=1;
     }
    }
    /*
    multi line comment
    */
    return 0; }
```

1.1 Tokens Generated :

```
T_INT
T_MAIN
(
)
{
T_INT
T_IDENTIFIER
=
T_I_CONSTANT
;
T_INT
T_IDENTIFIER
=
T_I_CONSTANT
;
T_WHILE
(
T_IDENTIFIER
>
T_I_CONSTANT
)
{
T_IDENTIFIER
=
T_IDENTIFIER
-
T_I_CONSTANT
;
T_IF
(
T_IDENTIFIER
T_EQ_OP
T_I_CONSTANT
)
{
T_IDENTIFIER
=
T_I_CONSTANT
;
}
}
T_RETURN
T_I_CONSTANT
;
}
```

1.2 Validity of the program :

The validity of the program decides if the compiler can proceed to the next phases or not. Since the given input is valid, the compiler proceeds to the next phases.
Output : **Valid.**
We have made use of **bash scripts** for this purpose. In Order to complete all stages of compiler using just one command (If input is valid).

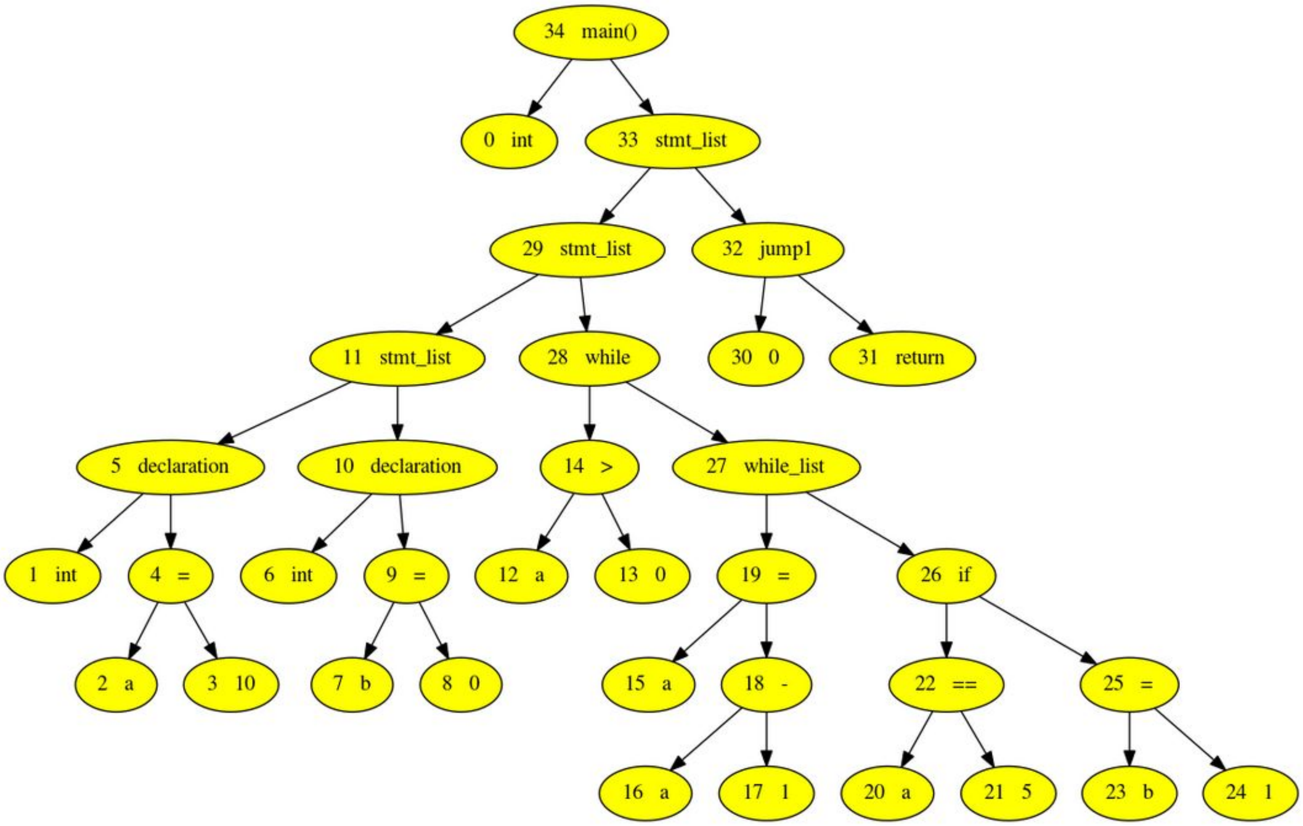
1.3 C Program free of comments :

```
int main()
{
    int a=10;
    int b=0;
    while(a>0)
    {
     a=a-1;
     if(a==5)
     {
         b=1;
     }
    }
    return 0;
}
```

1.4 Symbol Table :

ST	parent	id	name	type	declared at	used at	val
1	0	1	a	int	3	[4]	3
1	0	2	b	int	3	[4 8]	4
2	1	3	c	int	6	[7 8]	0
3	2	4	d	int	10	[ ]	0
4	1	5	e	int	15	[ ]	0

1.5 AST :



1.6 Intermediate Code :

```
main:       int a
            a = 10
            int b
            b = 0
BEGIN1 :    t1 = a > 0
            ifFalse t1 goto END1
L1 :        t2 = a - 1
            a = t2
            t3 = a == 5
            ifFalse t3 goto L3
            b = 1
L3:         goto BEGIN1
END1 :      return 0
next :
```

1.7 Code Optimization :

Code Optimizations such as Constant folding as Constant propagation
Sample input for Optimization (3 AC)

```
a = 10
b = a
c = b
d = c - 10
t0 = b + 10
a = c
b = 11
int f
t0 = c * 9
t1 = f * 20
print b
print t0
```

Optimized Output :

```
a = 10
b = 10
c = 10
d = 0
t0 = 20
a = c
b = 11
int f
t0 = 90
t1 = f * 20
print 11
print 90
```

2. ARCHITECTURE OF LANGUAGE:

The main constructs handled by the compiler include if-else statements and while statements. Along with that the compiler also handles basic types like int, float and char. It also handles 1D and 2D arrays. It takes care of syntax of printf statements and external declarations like header statements and #define statements.
The types of statements include :

- compound statement
- expression statement
- selection statement
- iteration statement
- return statement
- declaration statement
- print statement

2.1 Syntax of the language :

The yacc program also takes care of the following syntaxes :

- conversions from integer and real valued literals into integer and real valued numeric data.
- consumes any comments from the input stream and ignores them
- recognize all keywords and return the correct token
- arithmetic and Boolean expressions
- Postfix and Prefix expressions
- global/local variables declarations and initializations
- report syntax errors

2.2 Semantics of the language :

The yacc program handles the following syntaxes :

- Handles type checking of the variables. (including checking the dimensions of the variable i.e, 1D/2D/normal)
- Variables must be declared before use
- new declarations don't conflict with earlier ones, break statements only appear in loops

- handle errors related to scoping and declarations

3. LITERATURE SURVEY

- https://www.cs.utexas.edu/users/novak/lexpaper.htm
- http://dinosaur.compilertools.net/
- http://dinosaur.compilertools.net/yacc/
- https://www.geeksforgeeks.org/compiler-design-code-optimization/

4. CONTEXT FREE GRAMMAR

```
program
        : external_declaration function_definition
        | function_definition
        ;

external_declaration
        : declaration
        | include
        | external_declaration include
        | T_DEFINE T_IDENTIFIER primary_expression
        | external_declaration T_DEFINE T_IDENTIFIER
primary_expression
        | external_declaration declaration
        ;

include
        : T_INCLUDE '<' T_HEADER '>'
        | T_INCLUDE '"' T_HEADER '"'
        ;
```

```
function_definition
        : type_specifier T_MAIN '(' ')' compound_statement
        ;
primary_expression
        : T_IDENTIFIER
        | T_I_CONSTANT
        | T_F_CONSTANT
        | T_STRING_LITERAL
        | '(' expression ')'
        ;
postfix_expression
        : primary_expression
        | postfix_expression '[' T_I_CONSTANT ']'
        | postfix_expression T_INC_OP
        | postfix_expression T_DEC_OP
        ;

unary_expression
        : postfix_expression
        | T_INC_OP unary_expression
        | T_DEC_OP unary_expression
        | unary_operator unary_expression
        ;
unary_operator
        : '&' | '*' | '+' | '-' | '~' | '!'
        ;
multiplicative_expression
        : unary_expression
        | multiplicative_expression '*' unary_expression
        | multiplicative_expression '/' unary_expression
        | multiplicative_expression '%' unary_expression
        ;
additive_expression
        : multiplicative_expression
        | additive_expression '+' multiplicative_expression
        | additive_expression '-' multiplicative_expression
        ;
```

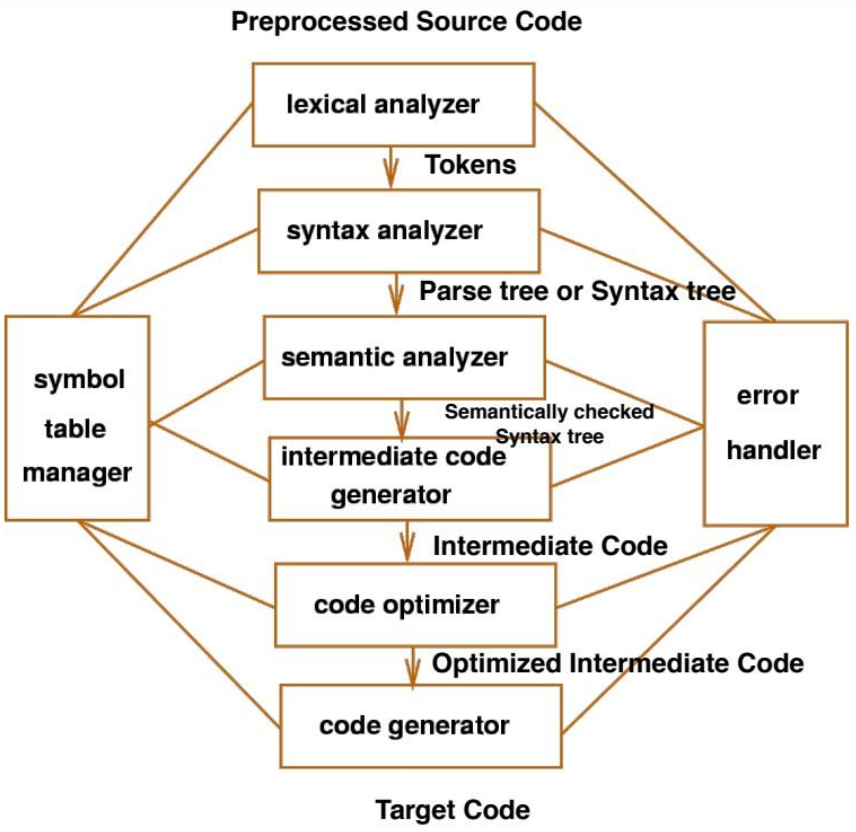
```
relational_expression
        : additive_expression
        | relational_expression '<' additive_expression
        | relational_expression '>' additive_expression
        | relational_expression T_LE_OP additive_expression
        | relational_expression T_GE_OP additive_expression
        ;
equality_expression
        : relational_expression
        | equality_expression T_EQ_OP relational_expression
        | equality_expression T_NE_OP relational_expression
        ;
logical_and_expression
        : equality_expression
        | logical_and_expression T_AND_OP equality_expression
        ;
logical_or_expression
        : logical_and_expression
        | logical_or_expression T_OR_OP logical_and_expression
        ;
assignment_expression
        : logical_or_expression
        | unary_expression assignment_operator assignment_expression
        ;

assignment_operator
        : '='
        | T_MUL_ASSIGN
        | T_DIV_ASSIGN
        | T_MOD_ASSIGN
        | T_ADD_ASSIGN
        | T_SUB_ASSIGN
        ;
expression
        : assignment_expression
        | expression ',' assignment_expression
        ;
```

```
declaration
        : type_specifier init_declarator_list ';'
        ;

init_declarator_list
        : init_declarator
        | init_declarator_list ',' init_declarator
        ;

init_declarator
        : declarator
        | declarator '=' initializer
        ;

type_specifier
        : T_CHAR
        | T_INT
        | T_FLOAT
        | T_DOUBLE
        ;

declarator
        : T_IDENTIFIER
        | declarator '[' T_I_CONSTANT ']'
        | declarator '[' ']'
        ;

initializer
        : assignment_expression
        | '{' initializer_list '}'
        | '{' initializer_list ',' '}'
        ;

initializer_list
        : initializer
        | initializer_list ',' initializer
        ;
```

```
statement
        : compound_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        | T_PRINT '(' T_STRING_LITERAL ')' ';'
        ;

compound_statement
        : '{' '}'
        | '{' statement_list '}'
        | '{' declaration_list '}'
        | '{' declaration_list statement_list '}'
        ;

declaration_list
        : declaration
        | declaration_list declaration
        ;

statement_list
        : statement
        | statement_list statement
        ;

expression_statement
        : ';'
        | expression ';'
        ;

selection_statement
        : T_IF '(' expression ')' statement
        | T_IF '(' expression ')' statement T_ELSE statement
        ;
```

```
iteration_statement
        : T_WHILE '(' expression ')' statement
        ;

jump_statement
        : T_CONTINUE ';'
        | T_BREAK ';'
        | T_RETURN ';'
        | T_RETURN expression ';'
        ;
```

5.DESIGN STRATEGY

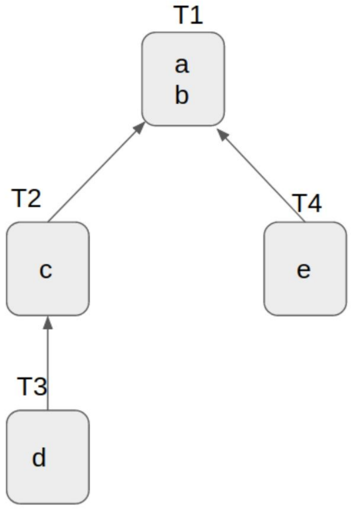


5.1 SYMBOL TABLE CREATION

- The symbol table implemented has a scopewise structure.

- Each time a declaration statement is encountered the symbol table is queried if the variable is already defined in the current scope and if not , it creates an entry in the symbol table
- Each scope has its own mini symbol table which points to its parent symbol table. An example of which is shown below.
  Here, the scope is handled by just incrementing / decrementing a scope counter everytime we encounter a ‘{‘ (inc) or ‘}’ (dec)
- The symbol **entry** for each of the variables consists of the following information :
  1. id - unique for each variable
  2. name - (l-value) of the variable
  3. type - the type of the variable
  4. declared at - specifies in which line a particular variable is declared
  5. used at - specifies in which lines the variable has been used
  6. length - specifies the length of the arrays (both 1D and 2D)
  7. breadth - specifies whether an array is 1D or 2D
  8. ptr - to store the data

```
int main()
{
    int a=3,b=4;
    if(a>b)
    {
     int c;
     c=5;
     if(b<c)
     {
          int d;
     }
    }
    else
    {
     int e;
    }
}
```



5.2 ABSTRACT SYNTAX TREE

Abstract Syntax tree is a tree representation of the abstract syntactic structure of source code. The nodes contains some data (a token and its value). However, it also contains some very specific pointers. If else construct requires 3 children. Since maximum of 3 children are required for all the constructs, the structure has 3 child pointers.

Every time a token other than an operator or punctuation is encountered, a leaf node is created. Whenever an operator is seen, an internal node is created and initialized with its children pointers. The tree is complete with abstract syntax tree generated for all the basic necessary constructs apart from while and if-else.

The tree is visualised using **pydot**. Pydot is a Python module provides with a full interface to create handle modify and process graphs in Graphviz's dot language. The tree structure is output into an external file as and when the tree nodes are created and initialized with child pointers (while parsing). This file is then loaded using python. A helper function is also being written to convert the file into a pydot graph.

5.3 INTERMEDIATE CODE GENERATION

After the creation of symbol table and AST, the next phase is ICG. Setting the inherited attributes of the variables was a main challenge as yacc does not support setting these values before the non terminal is on the stack. In order to tackle this the following approach was followed:

```
if_stat : if( {$$.false=”L1”} Cond ‘{‘ stmt ‘}’
cond : E OR E {//should access false of cond as
$0}
```

This approach is actually setting the false value of a temporary marker non-terminal (M) that is being created internally by yacc and $0 is accessing its false value(and not actually condition’s.)
We have used inherited attributes such as next,false,true and synthesized attributes such as code and addr.

5.4 CODE OPTIMIZATION

We have used python for code optimization .The input for this is the 3 Address Code. The python script parses through the input file line by line and whenever there is an assignment it tries to optimize it using some structures that are shown later. We have implemented Constant folding and propagation

5.5 ERROR HANDLING

The following kind of errors have been taken care of :

- Wrong use of variable or undeclared :
  Example 1 :
  ```
  char a[10]=”string”;
  a=a+1;
  ```
  Example 2 :
  ```
  //use before declare
  int a=0;
  b=a;
  ```
  Both of these can be handled by querying the symbol table to check if the variable exists and to get the type of the variable if it exists.

- Division By Zero is not possible
  Example :
  ```
  int a = 10/0;
  ```
  This can be handled by just looking at the yylval of the token and if it is 0, the error is reported
- Mod Operation is possible only for positive integers
  Example :
  ```
  int a = 10 % 1.5;
  int b = 10 % 0;
  int c = 10 % -2;
  ```
  All of the above operations are not allowed. It is also not allowed when a variable whose value is 0 is given. This can be handled either by looking at yylval or by querying the symbol table for the variable’s value.
- Dimensional Error
  ```
  int b = {1,2};
  int a[10]=1;
  int c[2][2]={1,2};
  ```
  These kind of errors can be handled by length and breadth of the variable present in the symbol table.
- Type mismatch has occurred
  ```
  char a[10] = “string”;
  int b = a + 10;
  ```
  handled by type checking.
- Wrong Initialization/ assignment :
  ```
  int a = “string” ;
  ```
  handled by type checking.
- Error in String Assignment
  Example :
  ```
  char a[2] = “long string “ ;
  ```
  The length of the string is longer than array size .
  Handled by comparing the size of the string and the array.
- Either keyword or variable is already present
  Example :
  ```
  int printf = 0;
  int a=0;
  int a= 10;
  ```

Keywords are preloaded into an array and each new variable is checked if its a keyword or not. Redeclaration can be handled by the use of symbol table and checking if the entry is already present.

6. IMPLEMENTATION DETAILS

6.1 SYMBOL TABLE CREATION

1. **The structures used are :**

   struct Variable -

```
typedef struct Variable
{
        float fVal; #yylval of the variable
        int idVal; #id of the variable
        int type; #type of the variable
}Variable;
```

2. **The helper functions used are :**

- multi_comment() - removes multi line comments
- init_symtab() - initializes the symbol table for a specific scope
- create_symboltable () -creates the symbol table
- present() - checks if a particular keyword is present in the symbol table or not.
- add_to_symtab() - adds a variable to symbol table if it is not already existing in the symbol table
- used() - returns the line numbers at which a variable is used
- display_symtab() - displays the symbol table
- find_var() - checks if a particular variable is present in the symbol table or not
- check_type() - checks the type of variable
- assign() - assigns a particular value to a variable
- check_dim() - checks and returns the dimensions of an array

6.2 ABSTRACT SYNTAX TREE

1. The structure used:

```
struct Node {
        char token[100];
        char* num ;
        struct Node* c1;
        struct Node* c2;
        struct Node* c3;
};
```

2. The helper functions used are :

- create_leaf() - to create leaf nodes
- create_node() - to create internal nodes and initialise the child pointers
- show_graph() - to create virtual nodes and display the graph (pydot)

6.3 INTERMEDIATE CODE GENERATION

1. The structure used:

```
typedef struct node
{
        char code[400];   //synthesized
        char addr[50];    //synthesised attribute
        char true[5];     //inherited attribute
        char false[5];    //inherited attribute
        char next[5];     //inherited attribute
}NODE;
```

2. The helper functions used are :

- check_type() - used to calculate proper indexes for the arrays based on the type of the array
- inbuilt sprintf() to concatenate code generated at each rule in yacc.

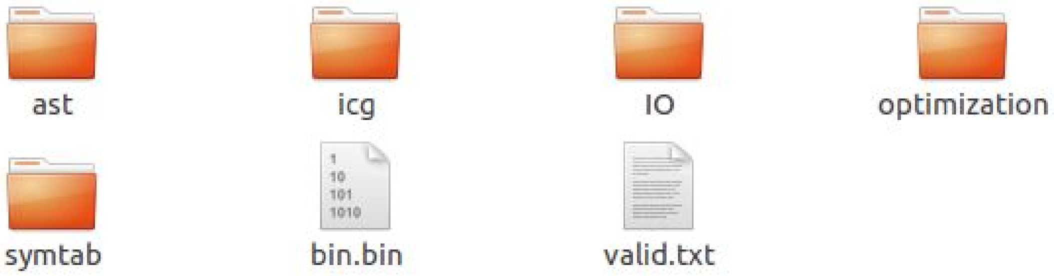
6.4 CODE OPTIMIZATION

We have used python for the purpose of code optimization. A table is used to keep track of the constants and each time a variable is assigned with the variables whose values are already present in the table, that variable is given the constant value based on the value in the table.

6.5 ERROR HANDLING

We have used inbuilt yyerror function for error handling and a string to it indicating which error has occurred . A global variable keeps track of the line numbers, and the exact line number where the error has occurred along with the type of the error is displayed.

Instructions on how to build and run your computer :



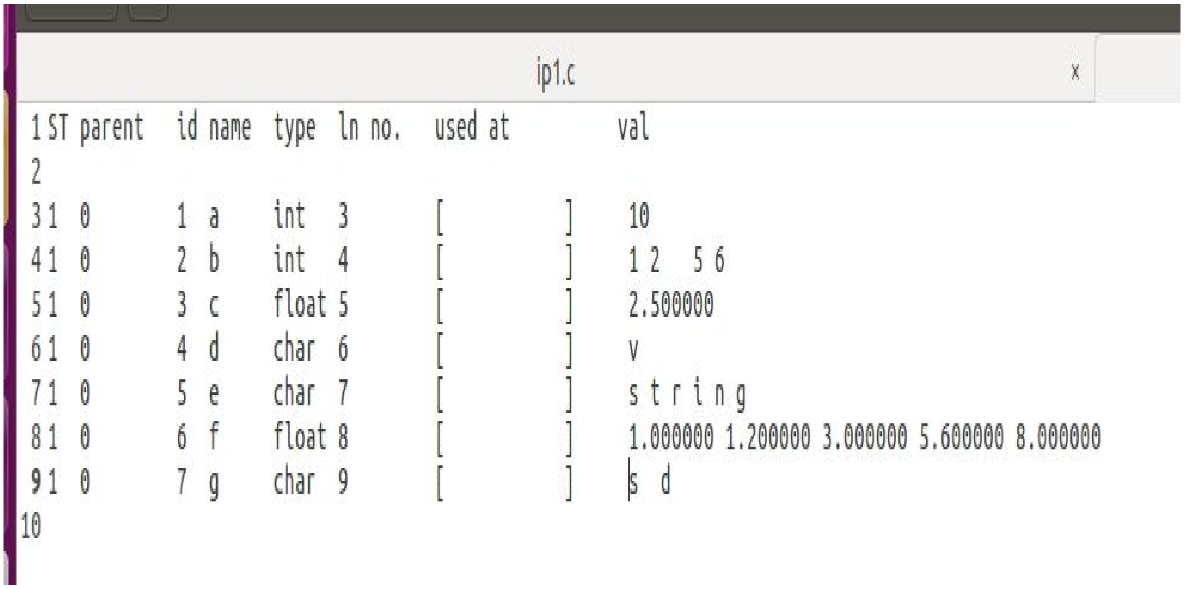
In symtab folder :
run : **bash ex.sh input_file.c**
The rest of the stages will be executed based on the validity of the program which is stored in valid.txt.
The bin.bin has the symbol table for future reference.

7. RESULTS

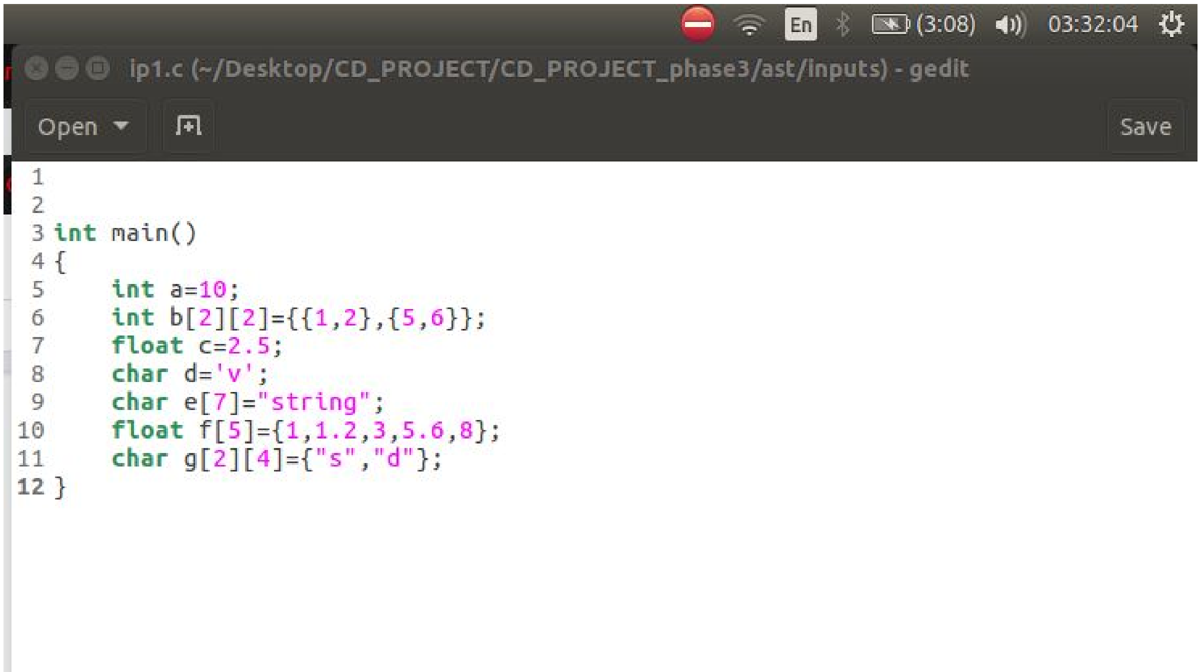
The final outcome of this project is a mini compiler for C which performs all the basic responsibilities of a compiler.

8. SNAPSHOTS

1. symbol table generation:

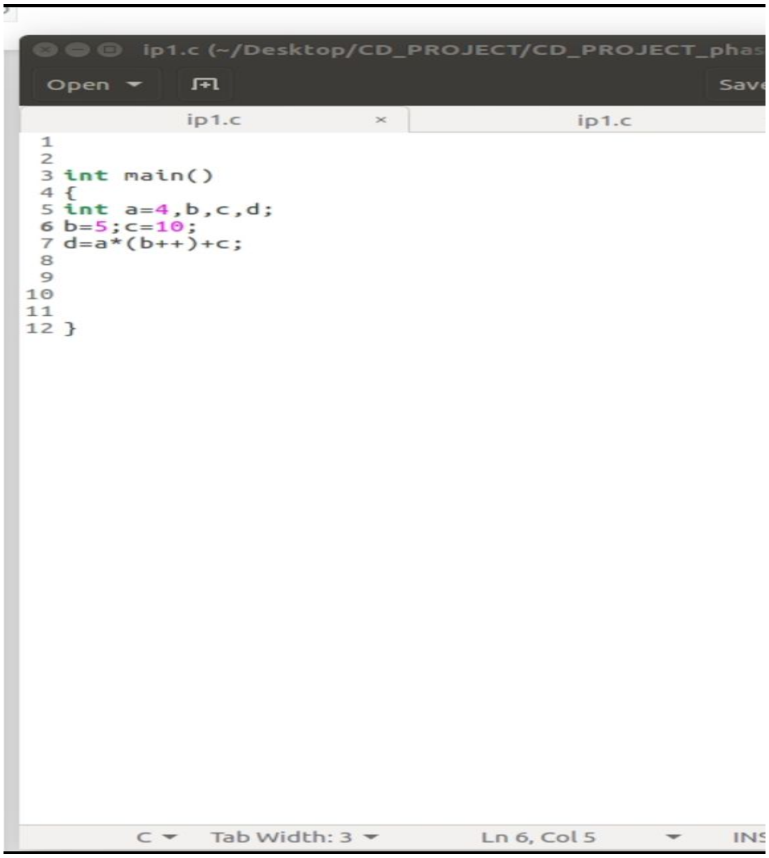


```
ST parent   id name  type  ln no.  used at       val

1  0        1  a     int   3       [          ]  10
1  0        2  b     int   4       [          ]  1 2  5 6
1  0        3  c     float 5       [          ]  2.500000
1  0        4  d     char  6       [          ]  v
1  0        5  e     char  7       [          ]  s t r i n g
1  0        6  f     float 8       [          ]  1.000000 1.200000 3.000000 5.600000 8.000000
1  0        7  g     char  9       [          ]  s  d
```

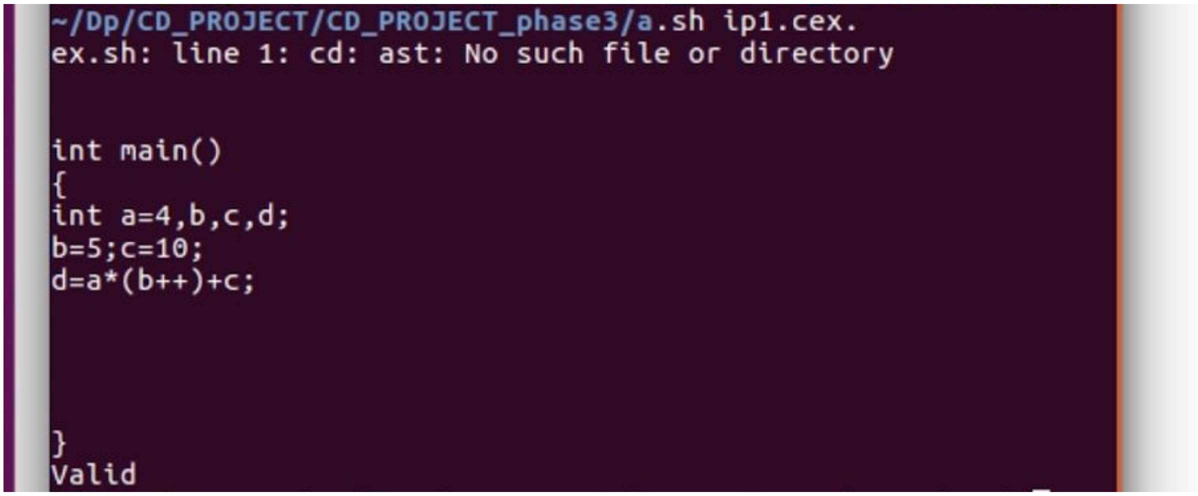

En
(3:08)
03:32:04
ip1.c (~/Desktop/CD_PROJECT/CD_PROJECT_phase3/ast/inputs) - gedit
Open
Save

```c


int main()
{
    int a=10;
    int b[2][2]={{1,2},{5,6}};
    float c=2.5;
    char d='v';
    char e[7]="string";
    float f[5]={1,1.2,3,5.6,8};
    char g[2][4]={"s","d"};
}
```

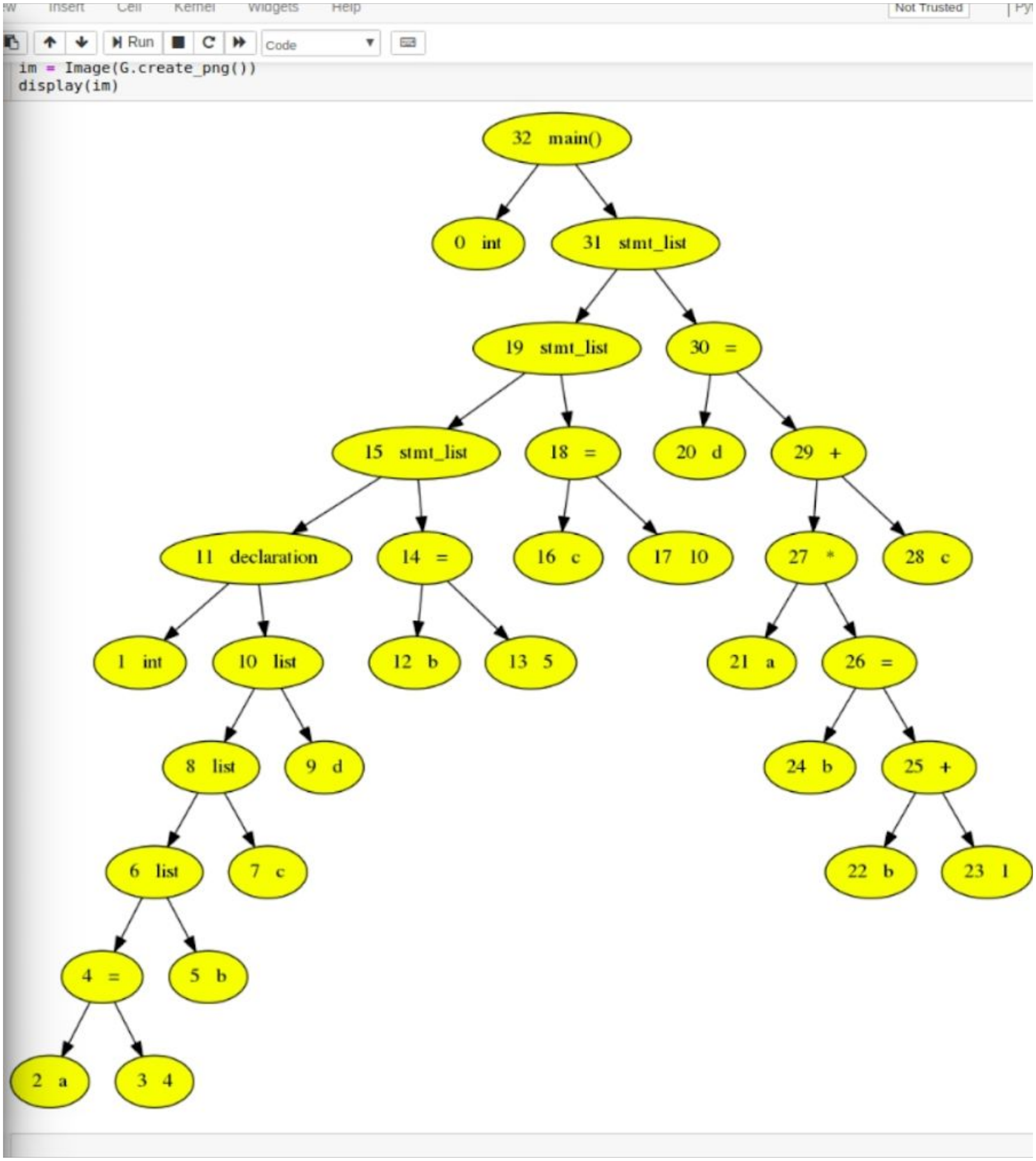
2. AST :



```


int main()
{
int a=4,b,c,d;
b=5;c=10;
d=a*(b++)+c;




}
```

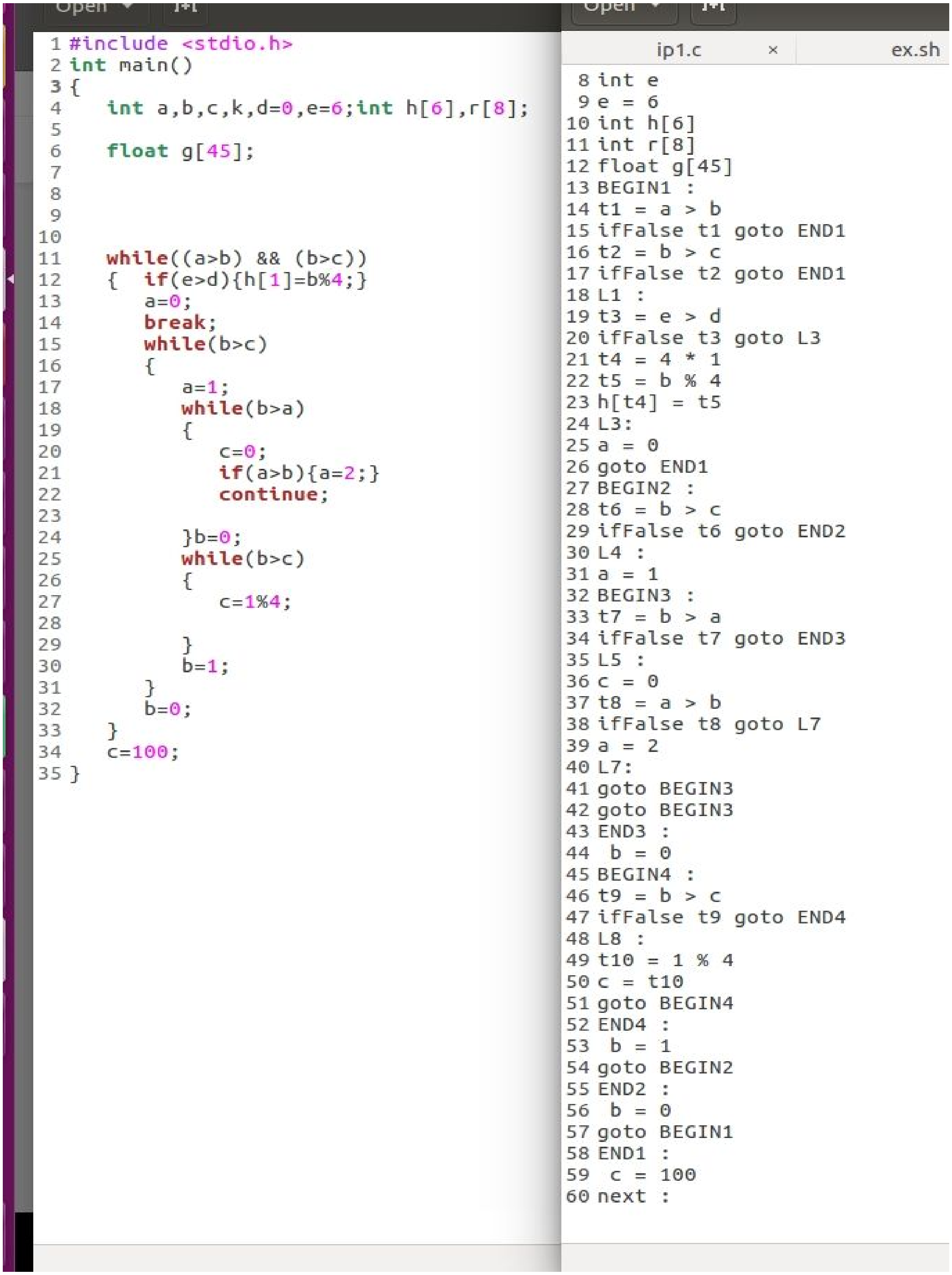


```
~/Dp/CD_PROJECT/CD_PROJECT_phase3/a.sh ip1.cex.
ex.sh: line 1: cd: ast: No such file or directory


int main()
{
int a=4,b,c,d;
b=5;c=10;
d=a*(b++)+c;




}
Valid
```



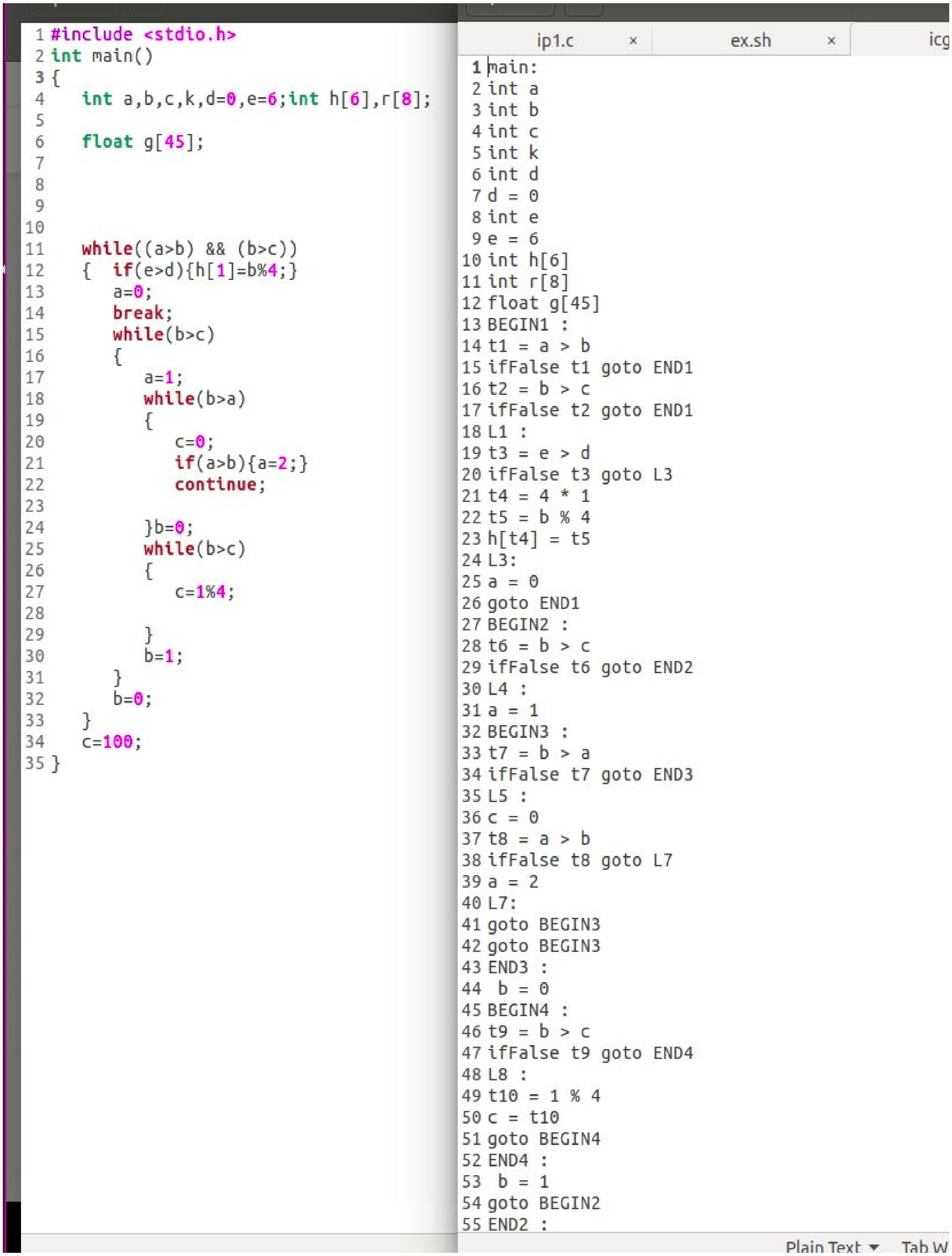
```
im = Image(G.create_png())
display(im)
```

3. ICG:



```
#include <stdio.h>
int main()
{
    int a,b,c,k,d=0,e=6;int h[6],r[8];

    float g[45];




    while((a>b) && (b>c))
    {   if(e>d){h[1]=b%4;}
        a=0;
        break;
        while(b>c)
        {
            a=1;
            while(b>a)
            {
                c=0;
                if(a>b){a=2;}
                continue;

            }b=0;
            while(b>c)
            {
                c=1%4;

            }
            b=1;
        }
        b=0;
    }
    c=100;
}
```

```
int e
e = 6
int h[6]
int r[8]
float g[45]
BEGIN1 :
t1 = a > b
ifFalse t1 goto END1
t2 = b > c
ifFalse t2 goto END1
L1 :
t3 = e > d
ifFalse t3 goto L3
t4 = 4 * 1
t5 = b % 4
h[t4] = t5
L3:
a = 0
goto END1
BEGIN2 :
t6 = b > c
ifFalse t6 goto END2
L4 :
a = 1
BEGIN3 :
t7 = b > a
ifFalse t7 goto END3
L5 :
c = 0
t8 = a > b
ifFalse t8 goto L7
a = 2
L7:
goto BEGIN3
goto BEGIN3
END3 :
 b = 0
BEGIN4 :
t9 = b > c
ifFalse t9 goto END4
L8 :
t10 = 1 % 4
c = t10
goto BEGIN4
END4 :
 b = 1
goto BEGIN2
END2 :
 b = 0
goto BEGIN1
END1 :
 c = 100
next :
```



```
#include <stdio.h>
int main()
{
    int a,b,c,k,d=0,e=6;int h[6],r[8];

    float g[45];




    while((a>b) && (b>c))
    {  if(e>d){h[1]=b%4;}
       a=0;
       break;
       while(b>c)
       {
          a=1;
          while(b>a)
          {
             c=0;
             if(a>b){a=2;}
             continue;

          }b=0;
          while(b>c)
          {
             c=1%4;

          }
          b=1;
       }
       b=0;
    }
    c=100;
}
```

```
main:
int a
int b
int c
int k
int d
d = 0
int e
e = 6
int h[6]
int r[8]
float g[45]
BEGIN1 :
t1 = a > b
ifFalse t1 goto END1
t2 = b > c
ifFalse t2 goto END1
L1 :
t3 = e > d
ifFalse t3 goto L3
t4 = 4 * 1
t5 = b % 4
h[t4] = t5
L3:
a = 0
goto END1
BEGIN2 :
t6 = b > c
ifFalse t6 goto END2
L4 :
a = 1
BEGIN3 :
t7 = b > a
ifFalse t7 goto END3
L5 :
c = 0
t8 = a > b
ifFalse t8 goto L7
a = 2
L7:
goto BEGIN3
goto BEGIN3
END3 :
 b = 0
BEGIN4 :
t9 = b > c
ifFalse t9 goto END4
L8 :
t10 = 1 % 4
c = t10
goto BEGIN4
END4 :
 b = 1
goto BEGIN2
END2 :
```

2. Optimization :

Input :

```
a = 10
b = 9
t0 = a + b
t1 = t0
e = t1 + 4
f = e + 7
t1 = 1 + f
print t1
int z
t3 = z * 3
```

Output :

```
a = 10
b = 9
t0 = 19
t1 = 19
e = 23
f = 30
t1 = 31
print 31
int z
t3 = z * 3
```

9. CONCLUSIONS

By doing this mini project we have learnt all the phases of the compiler and the challenges faced in designing a simple compiler.

10. FURTHER ENHANCEMENTS :

Further improvements could be:

- To handle multi-dimensional arrays (not only 1D and 2D)
- To handle other constructs like for and switch

- To perform other optimizations such as common sub-expression elimination using DAG.

11. Bibliography and references :

- https://www.cs.utexas.edu/users/novak/lexpaper.htm

- http://dinosaur.compilertools.net/

- http://dinosaur.compilertools.net/yacc/

- https://www.geeksforgeeks.org/compiler-design-code-optimization/