Report on

“C MINI-COMPILER”

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Aravind Perichiappan	**PES1201700107**
Apoorve Gupta	**PES1201700038**
Pratyush Mishra	**PES1201700126**

Under the guidance of

Madhura V
Assistant Professor
PES University, Bengaluru

January – May 2020

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

Introduction

This project being a Mini Compiler for the C programming language, focuses on generating an intermediate code for the language for specific constructs.

It works for constructs such as conditional statements, loops and the if-else operator.

The main functionality of the project is to generate an optimized intermediate code for the given C source code.

This is done using the following steps:

i) Generate symbol table
ii) Generate Abstract Syntax Tree for the code
iii) Generate 3 address code
iv) Perform Code Optimization

The main tools used in the project include LEX which identifies predefined patterns and generates tokens for the patterns matched and YACC which parses the input for semantic meaning and generates an abstract syntax tree and intermediate code for the source code.

PYTHON is used to optimize the intermediate code generated by the parser.

Architecture of language

C constructs implemented:

1. Simple If
2. If-else
3. For-loop

- Arithmetic expressions with +, -, *, /, ++, -- are handled
- Boolean expressions with >,=,<=,== are handled
- Error handling reports undeclared variables
- Error handling also reports syntax errors with line numbers

Literature survey

- https://www.lysator.liu.se/c/ANSI-C-grammar-y.html
- http://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf
- http://dinosaur.compilertools.net/

Context free grammar

```
begin
        : external_declaration
        | begin external_declaration
        ;

primary_expression
        : IDENTIFIER { insertToHash($<str>1, data_type , yylineno); }
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
        ;

postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
        | postfix_expression '(' ')'
        | postfix_expression '(' argument_expression_list ')'
        | postfix_expression '.' IDENTIFIER
        | postfix_expression PTR_OP IDENTIFIER
        | postfix_expression INC_OP
        | postfix_expression DEC_OP
        ;

argument_expression_list
        : assignment_expression
        | argument_expression_list ',' assignment_expression
        ;

unary_expression
        : postfix_expression
        | INC_OP unary_expression
        | DEC_OP unary_expression
        | unary_operator cast_expression
        | SIZEOF unary_expression
```

```
	| SIZEOF '(' type_name ')'
	;

unary_operator
	: '&'
	| '*'
	| '+'
	| '-'
	| '~'
	| '!'
	;

cast_expression
	: unary_expression
	| '(' type_name ')' cast_expression
	;

multiplicative_expression
	: cast_expression
	| multiplicative_expression '*' cast_expression
	| multiplicative_expression '/' cast_expression
	| multiplicative_expression '%' cast_expression
	;

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

shift_expression
	: additive_expression
	| shift_expression LEFT_OP additive_expression
	| shift_expression RIGHT_OP additive_expression
	;

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

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

and_expression
	: equality_expression
	| and_expression '&' equality_expression
	;

exclusive_or_expression
	: and_expression
	| exclusive_or_expression '^' and_expression
	;

inclusive_or_expression
	: exclusive_or_expression
	| inclusive_or_expression '|' exclusive_or_expression
	;

logical_and_expression
	: inclusive_or_expression
	| logical_and_expression AND_OP inclusive_or_expression
	;

logical_or_expression
	: logical_and_expression
	| logical_or_expression OR_OP logical_and_expression
	;

conditional_expression
```

```
        : logical_or_expression
        | logical_or_expression '?' expression ':' conditional_expression
        ;

assignment_expression
        : conditional_expression
        | unary_expression assignment_operator assignment_expression
        ;

assignment_operator
        : '='
        | MUL_ASSIGN
        | DIV_ASSIGN
        | MOD_ASSIGN
        | ADD_ASSIGN
        | SUB_ASSIGN
        | LEFT_ASSIGN
        | RIGHT_ASSIGN
        | AND_ASSIGN
        | XOR_ASSIGN
        | OR_ASSIGN
        ;

expression
        : assignment_expression
        | expression ',' assignment_expression
        ;

constant_expression
        : conditional_expression
        ;

declaration
        : declaration_specifiers ';'
        | declaration_specifiers init_declarator_list ';'
        ;

declaration_specifiers
        : storage_class_specifier
        | storage_class_specifier declaration_specifiers
        | type_specifier
        | type_specifier declaration_specifiers
        ;

init_declarator_list
        : init_declarator
        | init_declarator_list ',' init_declarator
        ;

init_declarator
        : declarator
        | declarator '=' initializer
        ;

type_specifier
        : VOID
        | CHAR
        | SHORT
        | INT
        | LONG
        | FLOAT
        | DOUBLE
        | SIGNED
        | UNSIGNED
        ;

declarator
        : direct_declarator
        ;

direct_declarator
        : IDENTIFIER
        | '(' declarator ')'
        | direct_declarator '[' constant_expression ']'
        | direct_declarator '[' ']'
        | direct_declarator '(' parameter_list ')'
        | direct_declarator '(' identifier_list ')'
```

```
	| direct_declarator '(' ')'
	;

parameter_list
	: parameter_declaration
	| parameter_list ',' parameter_declaration
	;

parameter_declaration
	: declaration_specifiers declarator
	| declaration_specifiers
	;

identifier_list
	: IDENTIFIER
	| identifier_list ',' IDENTIFIER
	;

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

initializer_list
	: initializer
	| initializer_list ',' initializer
	;

statement
	: compound_statement
	| expression_statement
	| selection_statement
	| iteration_statement
	| jump_statement
	;

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 ';'
        ;

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

jump_statement
        : CONTINUE ';'
        | BREAK ';'
        | RETURN ';'
        | RETURN expression ';'
        ;

external_declaration
        : declaration
        ;
```

Design strategy

Lexical Analysis

- LEX tool was used to create a scanner for C language
- The scanner transforms the source file from a stream of bits and bytes into a series of meaningful tokens containing information that will be used by the later stages of the compiler.
- The scanner also scans for the comments (single-line and multiline comments) and writes the source file without comments onto an output file which is used in the further stages.
- All tokens included are of the form T_.Eg: T_pl for '+',T_min for '-' etc.
- A global variable 'yylavl' is used to record the value of each lexeme scanned. 'yytext' is the lex variable that stores the matched string.
- Skipping over white spaces and recognizing all keywords, operators, variables and constants is handled in this phase.
- Scanning error is reported when the input string does not match any rule in the lex file.
- The rules are regular expressions which have corresponding actions that execute on a match with the source input.

Syntax and Semantic Analysis

- Syntax analysis is only responsible for verifying that the sequence of tokens forms a valid sentence given the definition of your Programming Language grammar.
- The design implementation supports 1. Variable declarations and initializations 2. Variables of type int, float and char 3. Arithmetic and boolean expressions 4. Postfix and prefix expressions 5. Constructs - if-else, ternary, while loop and for loop
- Yacc tool is used for parsing. It reports shift-reduce and reduce-reduce conflicts on parsing an ambiguous grammar.
- Semantic analysis is also done in this phase to check for any semantical errors in the code.

Symbol table generation

- Four arrays are maintained to keep track of the variables, values, type and the line number in the input.

- As each line of the input file is parsed an entry is made in the symbol table for the declarations of identifiers.
- In case the variable is not initialized, by default all the variables are assigned 0.

Abstract Syntax Tree

- AST is generated using lex and yacc. A tree structure representing the syntactical flow of the code is generated in this phase. For expressions associativity is indicated using the %left and %right fields.
- To build the tree, a structure is maintained which has pointers to its children and a container for its data value.

```
typedef struct nodeTypeTag
{
        nodeEnum type; /* type of node */
        union
        {
                conNodeType con; /* constants */
                idNodeType id; /* identifiers */
                oprNodeType opr; /* operators */
        };
}nodeType;
```

- When every new token is encountered during parsing, the buildTree function takes in the value of the token, creates a node of the tree and attaches it to its parent (head of the reduced production). When the head production of the construct is reached the printTree function displays the tree for it.

Intermediate code generation

- 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. Intermediate code tends to be machine independent code. This is performed using lex and yacc tools.
- Three-Address Code – A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z, here x, y, z will have an address (memory location).

Code Optimizations

The code optimizer is a C program that maintains a key-value mapping using a struct that resembles the symbol table structure to keep track of variables and their values. This structure is used to perform dead code elimination and removal of common expressions.

Error Handling

Error handling is performed during the semantic analysis phase, right after syntactic analysis is over. This is implemented in the YACC parser by checking for scope validity, declarations and other semantic details.

Target Code Generation

A python script is used to convert the optimized intermediate code into assembly level instructions for the target machine. This is done by parsing the ICG code sequentially and generating appropriate assembly language instructions.

Implementation details

Symbol table generation

The symbol table is generated using lex and yacc. The symbol table is stored as an array of structs, each containing all the fields needed for a row of the symbol table as chars and ints. It is generated using a hash table data structure to store all the symbols in memory.

Abstract syntax tree

The AST contains nodes in the form of structs (shown above). A c program is used to graph all the nodes in a visual manner on the terminal. While parsing the code, we create and store the nodes as necessary.

Intermediate code generation

No additional data structures are used for generating intermediate code. The code input is parsed by the YACC parser and simultaneously the intermediate code is generated using conditional logic.

Code optimization

The code optimizer is a C program that goes through the ICG output line by line to optimize in two ways – removing dead code and removing common expressions. For this purpose, the ICG code is stored in a struct with two fields, one field containing the first operand and the other field containing all the other operands.

Assembly code generation

The assembly code is generated by a python script. It parses the input optimized ICG code and generates the equivalent assembly instructions. re module is used for this parsing. A dictionary is used to maintain the status of availability and usage of the target machine's registers.

Error Handling

Error handling is done in the syntax and semantic analysis stages using parsing and conditional logic. Discovered errors are displayed as terminal output.

Build & Execution Instructions

1. Semantic Analysis, Symbol Table generation and ICG:
```
lex parser.l
yacc parser.y
gcc y.tab.c -ll -ly
./a.out < tests/test1.c
```

2. ICG optimizer:
```
gcc opti.cpp -o opti
./opti < tests/test1.txt
```

3. Target code generator:
```
python generate_assembly.py tests/test1.txt
```

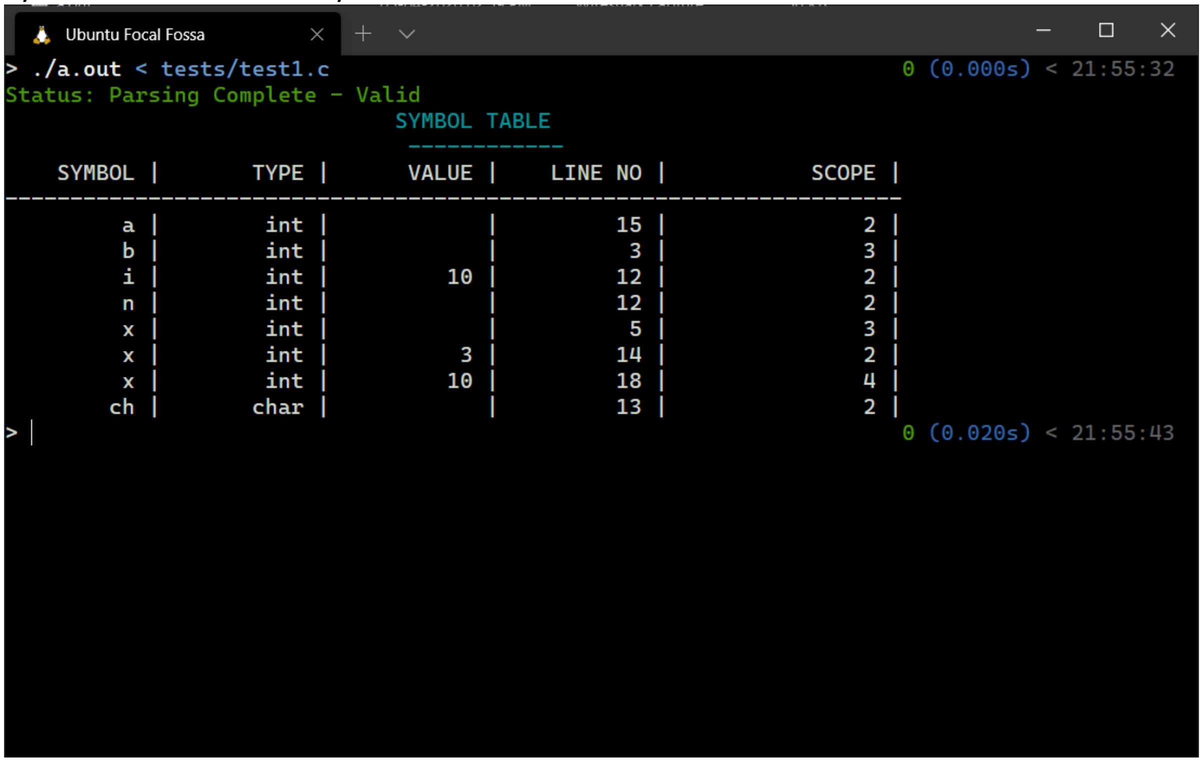
Results and possible shortcomings

Thus, we have seen the design strategies and implementation of the different stages involved in building a mini compiler and successfully built a working compiler that generates an assembly code, given a C code as input.

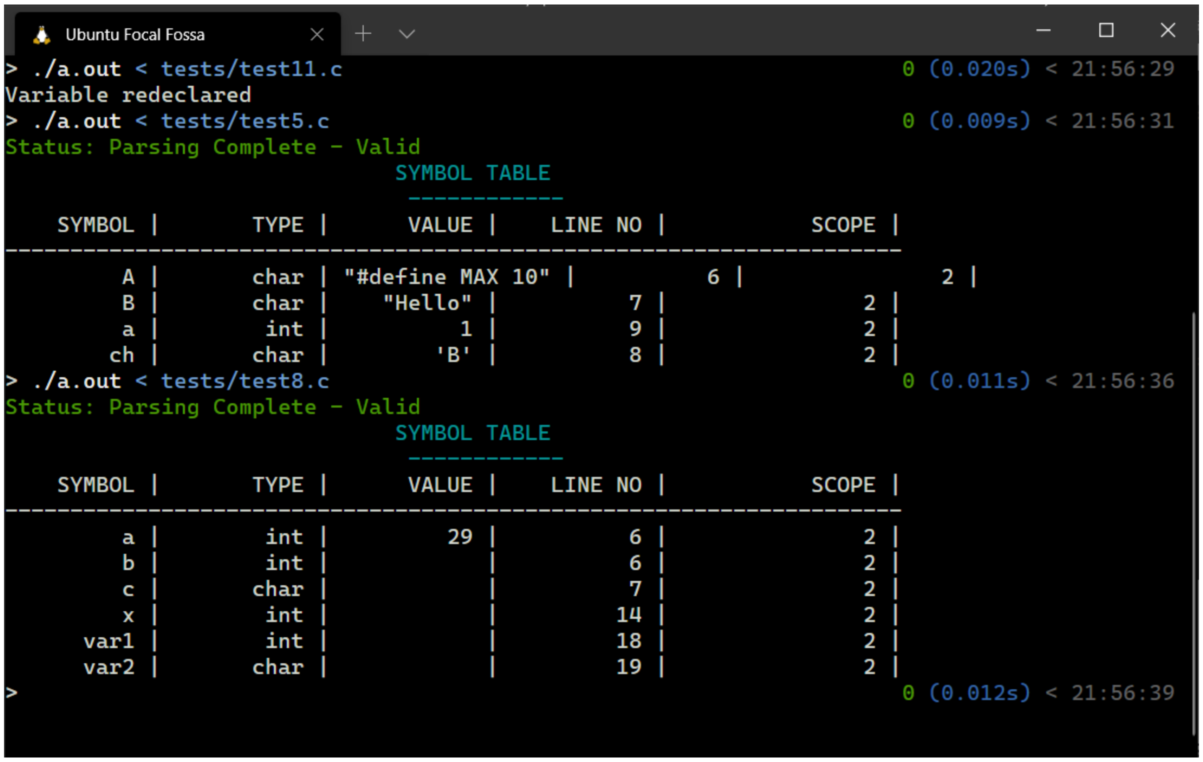
The major shortcomings of this compiler are that it does not support many constructs, and it does not optimize the code very well. Thus, the compiler is of limited use.

Output Snapshots

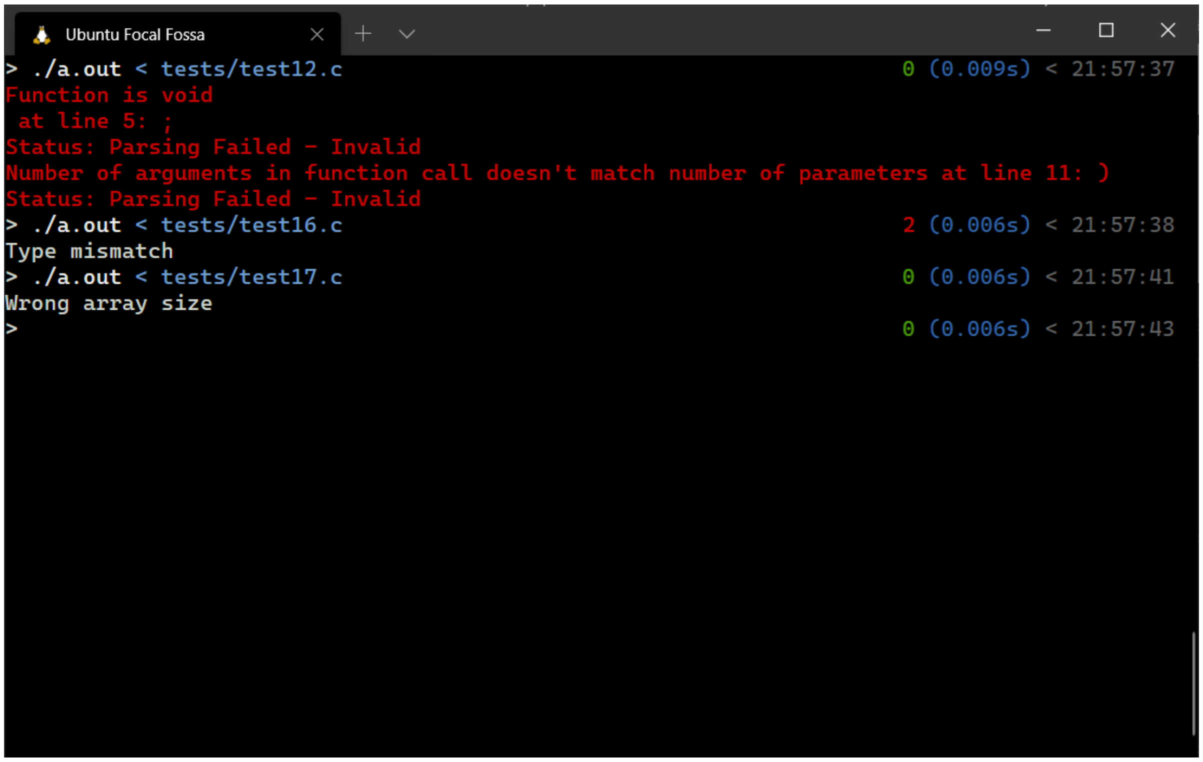
Syntax and semantic analysis:



```
> ./a.out < tests/test1.c                                              0 (0.000s) < 21:55:32
Status: Parsing Complete - Valid
                                               SYMBOL TABLE
                                                -------------

      SYMBOL |           TYPE |          VALUE |       LINE NO |                SCOPE |
---------------------------------------------------------------------------------------
           a |            int |                |            15 |                    2 |
           b |            int |                |             3 |                    3 |
           i |            int |             10 |            12 |                    2 |
           n |            int |                |            12 |                    2 |
           x |            int |                |             5 |                    3 |
           x |            int |              3 |            14 |                    2 |
           x |            int |             10 |            18 |                    4 |
          ch |           char |                |            13 |                    2 |
> |                                                                    0 (0.020s) < 21:55:43
```

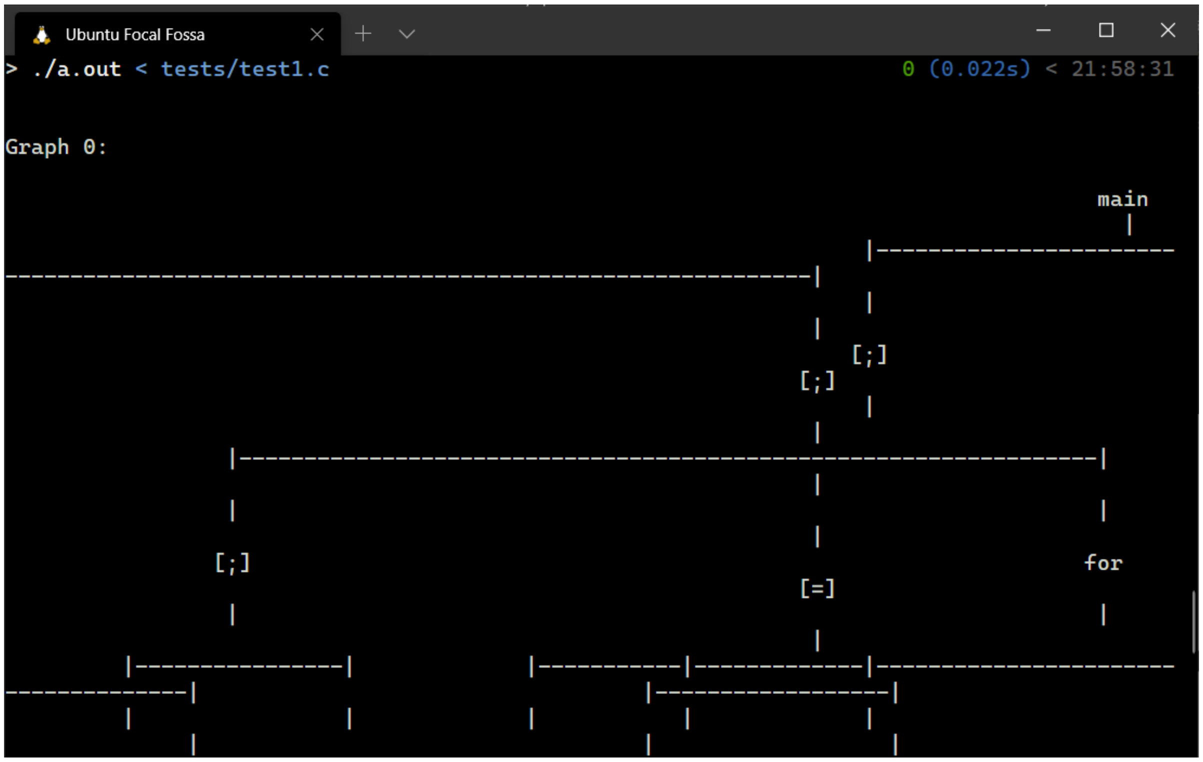


```
Ubuntu Focal Fossa
> ./a.out < tests/test11.c                                                   0 (0.020s) < 21:56:29
Variable redeclared
> ./a.out < tests/test5.c                                                    0 (0.009s) < 21:56:31
Status: Parsing Complete - Valid
                              SYMBOL TABLE
                               ------------

    SYMBOL |          TYPE |         VALUE |    LINE NO |           SCOPE |
----------------------------------------------------------------------
         A |          char | "#define MAX 10" |          6 |              2 |
         B |          char |       "Hello" |          7 |               2 |
         a |           int |             1 |          9 |               2 |
        ch |          char |           'B' |          8 |               2 |
> ./a.out < tests/test8.c                                                    0 (0.011s) < 21:56:36
Status: Parsing Complete - Valid
                              SYMBOL TABLE
                               ------------

    SYMBOL |          TYPE |         VALUE |    LINE NO |           SCOPE |
----------------------------------------------------------------------
         a |           int |            29 |          6 |               2 |
         b |           int |               |          6 |               2 |
         c |          char |               |          7 |               2 |
         x |           int |               |         14 |               2 |
      var1 |           int |               |         18 |               2 |
      var2 |          char |               |         19 |               2 |
>                                                                            0 (0.012s) < 21:56:39
```

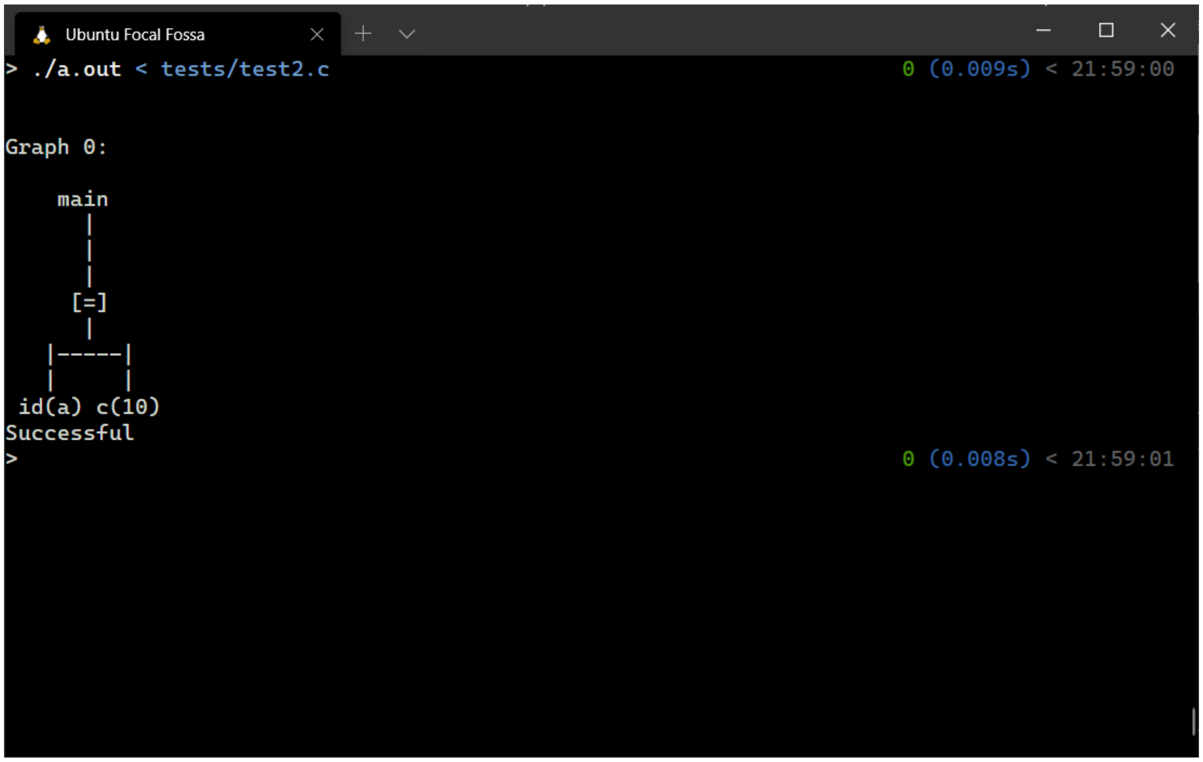


```
Ubuntu Focal Fossa
> ./a.out < tests/test12.c                                                   0 (0.009s) < 21:57:37
Function is void
 at line 5: ;
Status: Parsing Failed - Invalid
Number of arguments in function call doesn't match number of parameters at line 11: )
Status: Parsing Failed - Invalid
> ./a.out < tests/test16.c                                                   2 (0.006s) < 21:57:38
Type mismatch
> ./a.out < tests/test17.c                                                   0 (0.006s) < 21:57:41
Wrong array size
>                                                                            0 (0.006s) < 21:57:43
```

AST:

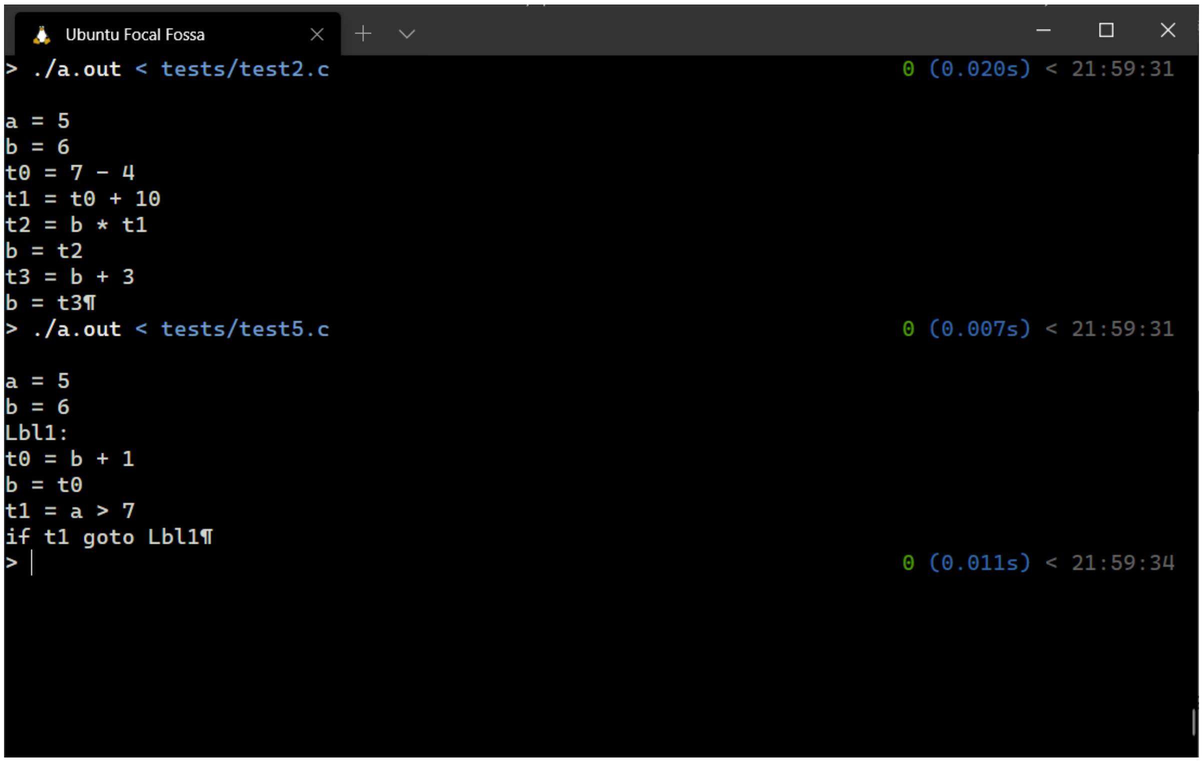


```
> ./a.out < tests/test1.c                                            0 (0.022s) < 21:58:31

Graph 0:

                                                                                    main
                                                                                      |
                                                                  |-----------------------
--------------------------------------------------------------|
                                                                  |
                                                              |
                                                                 [;]
                                                             [;]
                                                                  |
                                                              |
                 |------------------------------------------------------------------|
                                                              |
                 |                                                                  |
                                                              |
                [;]                                                                for
                                                             [=]
                 |                                                                  |
                                                              |
         |----------------|             |-----------|-------------|-----------------------
--------------|                                  |------------------|
         |                |             |           |             |
              |                                  |                  |
```



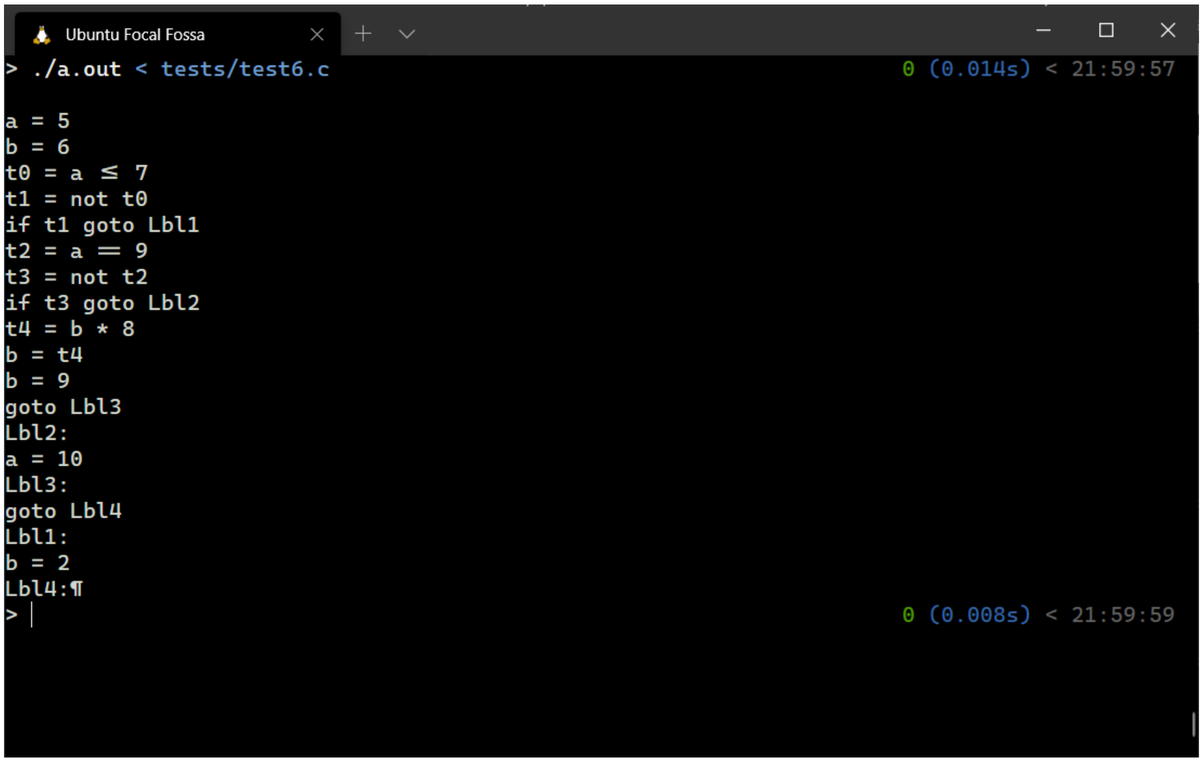
```
> ./a.out < tests/test2.c                                            0 (0.009s) < 21:59:00

Graph 0:

    main
      |
      |
      |
     [=]
      |
   |-----|
   |     |
 id(a) c(10)
Successful
>                                                                    0 (0.008s) < 21:59:01
```

ICG:



Ubuntu Focal Fossa

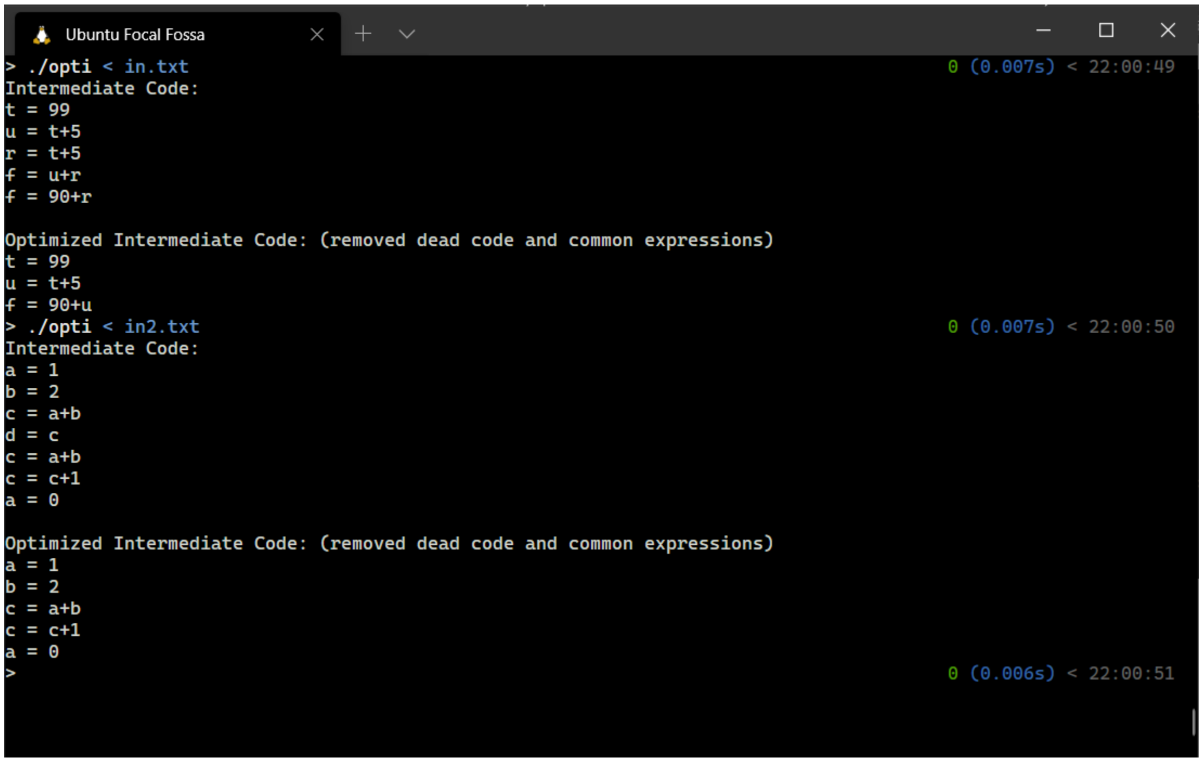
```
> ./a.out < tests/test2.c                                     0 (0.020s) < 21:59:31

a = 5
b = 6
t0 = 7 - 4
t1 = t0 + 10
t2 = b * t1
b = t2
t3 = b + 3
b = t3¶
> ./a.out < tests/test5.c                                     0 (0.007s) < 21:59:31

a = 5
b = 6
Lbl1:
t0 = b + 1
b = t0
t1 = a > 7
if t1 goto Lbl1¶
>                                                             0 (0.011s) < 21:59:34
```



Ubuntu Focal Fossa

```
> ./a.out < tests/test6.c                                     0 (0.014s) < 21:59:57

a = 5
b = 6
t0 = a ≤ 7
t1 = not t0
if t1 goto Lbl1
t2 = a == 9
t3 = not t2
if t3 goto Lbl2
t4 = b * 8
b = t4
b = 9
goto Lbl3
Lbl2:
a = 10
Lbl3:
goto Lbl4
Lbl1:
b = 2
Lbl4:¶
>                                                             0 (0.008s) < 21:59:59
```

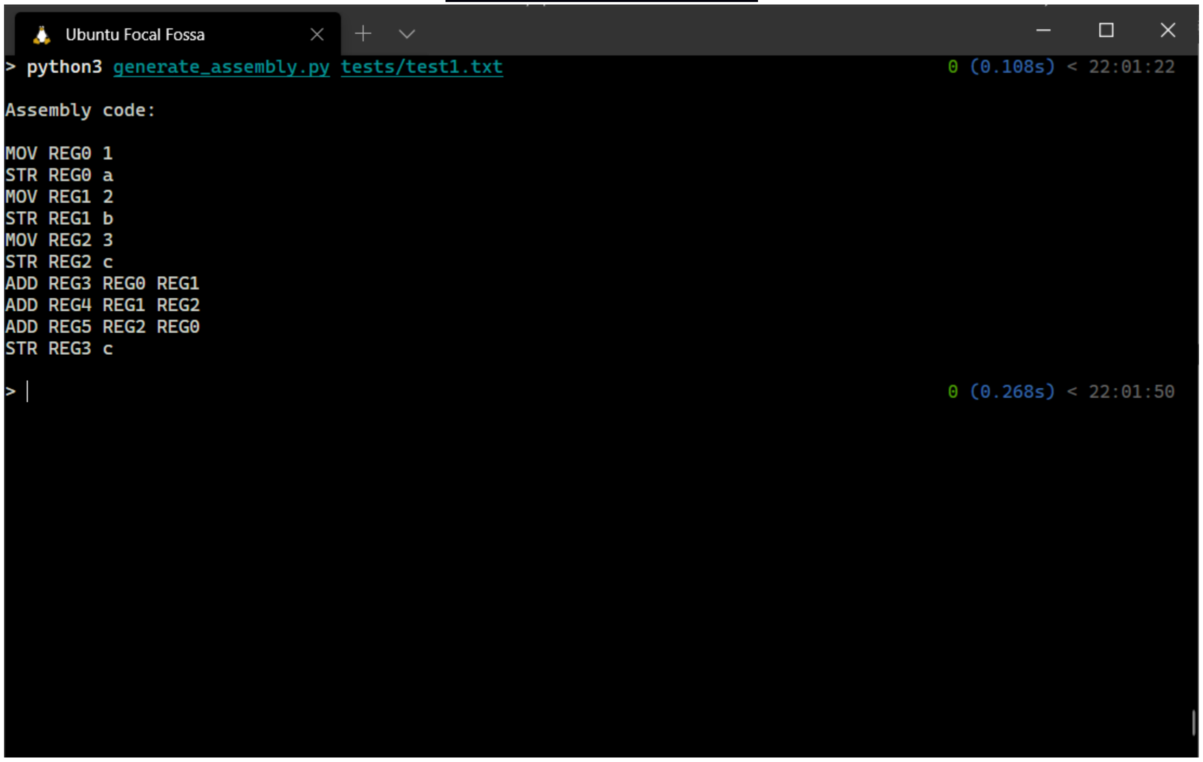
Optimized ICG code:



Ubuntu Focal Fossa

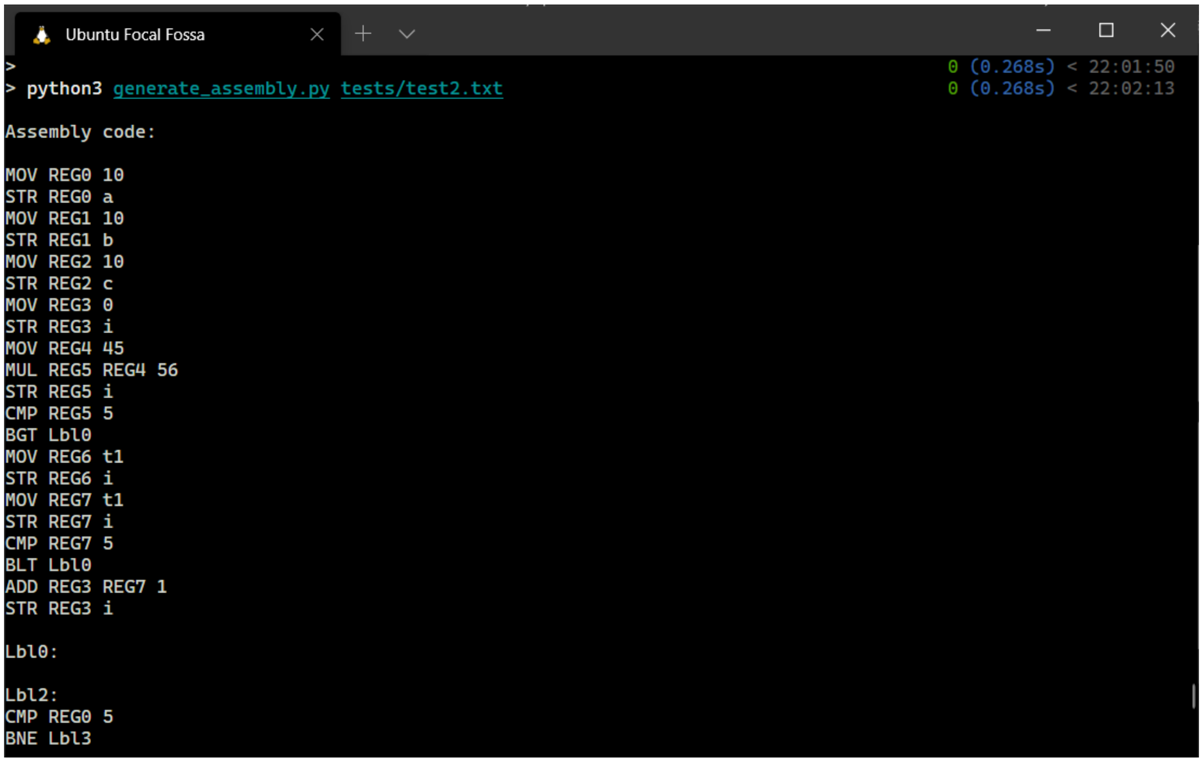
```
> ./opti < in.txt                                                                              0 (0.007s) < 22:00:49
Intermediate Code:
t = 99
u = t+5
r = t+5
f = u+r
f = 90+r

Optimized Intermediate Code: (removed dead code and common expressions)
t = 99
u = t+5
f = 90+u
> ./opti < in2.txt                                                                             0 (0.007s) < 22:00:50
Intermediate Code:
a = 1
b = 2
c = a+b
d = c
c = a+b
c = c+1
a = 0

Optimized Intermediate Code: (removed dead code and common expressions)
a = 1
b = 2
c = a+b
c = c+1
a = 0
>                                                                                              0 (0.006s) < 22:00:51
```

Target code generation:

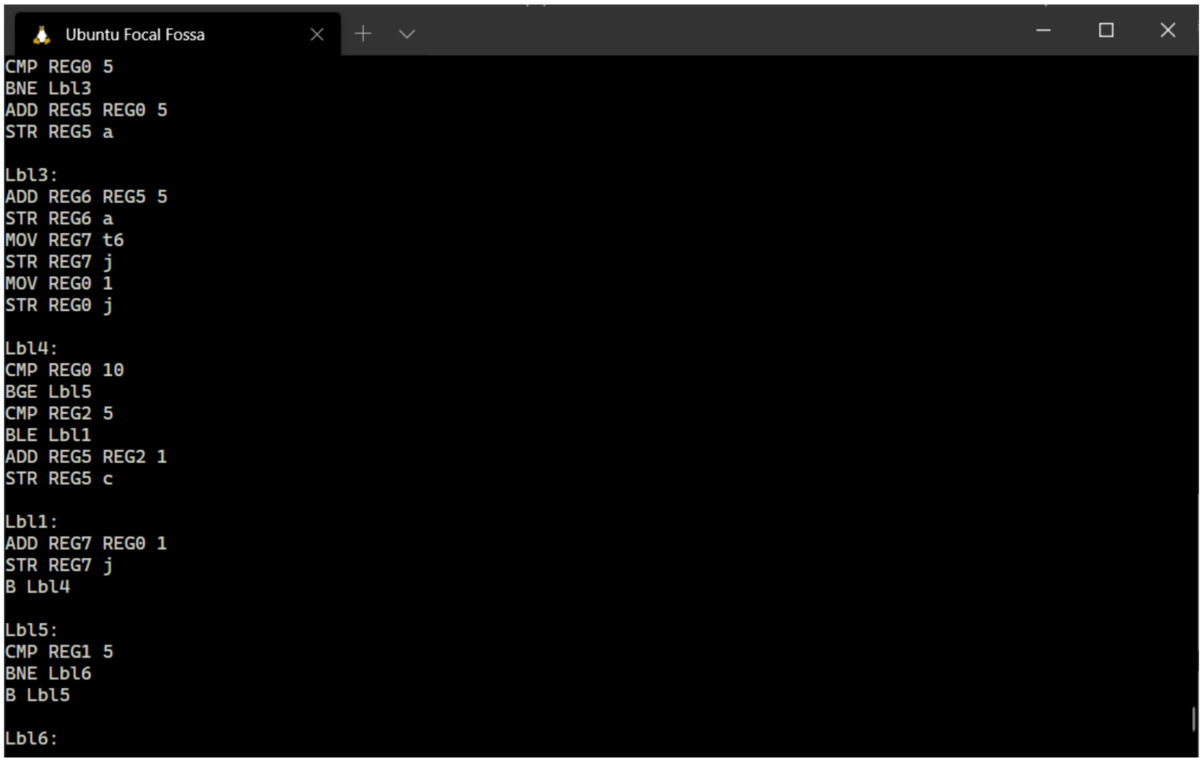


Ubuntu Focal Fossa

```
> python3 generate_assembly.py tests/test1.txt                                                 0 (0.108s) < 22:01:22

Assembly code:

MOV REG0 1
STR REG0 a
MOV REG1 2
STR REG1 b
MOV REG2 3
STR REG2 c
ADD REG3 REG0 REG1
ADD REG4 REG1 REG2
ADD REG5 REG2 REG0
STR REG3 c

>                                                                                              0 (0.268s) < 22:01:50
```



```
>                                                                       0 (0.268s) < 22:01:50
> python3 generate_assembly.py tests/test2.txt                          0 (0.268s) < 22:02:13

Assembly code:

MOV REG0 10
STR REG0 a
MOV REG1 10
STR REG1 b
MOV REG2 10
STR REG2 c
MOV REG3 0
STR REG3 i
MOV REG4 45
MUL REG5 REG4 56
STR REG5 i
CMP REG5 5
BGT Lbl0
MOV REG6 t1
STR REG6 i
MOV REG7 t1
STR REG7 i
CMP REG7 5
BLT Lbl0
ADD REG3 REG7 1
STR REG3 i

Lbl0:

Lbl2:
CMP REG0 5
BNE Lbl3
```



```
CMP REG0 5
BNE Lbl3
ADD REG5 REG0 5
STR REG5 a

Lbl3:
ADD REG6 REG5 5
STR REG6 a
MOV REG7 t6
STR REG7 j
MOV REG0 1
STR REG0 j

Lbl4:
CMP REG0 10
BGE Lbl5
CMP REG2 5
BLE Lbl1
ADD REG5 REG2 1
STR REG5 c

Lbl1:
ADD REG7 REG0 1
STR REG7 j
B Lbl4

Lbl5:
CMP REG1 5
BNE Lbl6
B Lbl5

Lbl6:
```

Conclusions

Thus, we have implemented a basic mini-compiler for C that takes input C code and compiles assembly for the target machine by processing the code through various phases as described in this report.

Further enhancements

- The symbol table structure is same across all types of tokens (constants, identifiers and operators). This leads to some fields being empty for some of the tokens. This can be optimized by using a better representation.

- All the different phases of the compiler can be integrated better to make the process of compiling the code easier.

- The Code optimizer does not optimize well when propagating constants across branches (At if statements and loops). It works well only in sequential programs. This can be rectified.

- More constructs can be supported by the compiler.

References

- https://www.lysator.liu.se/c/ANSI-C-grammar-y.html
- http://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf
- http://dinosaur.compilertools.net/
- http://sandbox.mc.edu/~bennet/cs404/outl/cbnf.html