Mini compiler for PHP Language

A MINI PROJECT REPORT

Submitted by

Durga Chandana Sree M (RA2011028010099)
Chaluvadi Jwala Satya Saketh (RA2011028010071)
Bhargav Sandeep K (RA2011028010075)

Under the guidance of

Dr. Priyanka R

Assistant Professor, Department of Networking and Communications

INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

DEPARTMENT OF NETWORKING AND COMMUNICATIONS
FACULTY OF ENGINEERING AND TECHNOLOGY
SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
Kattankulathur, Kancheepuram MAY
2023

COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this Mini project report titled '**PHP Compiler**' for the course 18CSC304J – Compiler Design is the bonafide work of **Durga Chandana Sree M (RA2011028010099),Chaluvadi Jwala Satya Saketh (RA2011028010071),Bhargav Sandeep K (RA2011028010075)** undertook the task of completing the project within the allotted time

SIGNATURE

Dr.Priyanka R

Assistant Professor
Department of NWC Technologies

SIGNATURE

Dr. Annapurani Panaiyappan K
Head Of The Department
Professor
Department of Computing Technologies

ABSTRACT

The project aims to develop a PHP compiler as part of a comprehensive study in compiler design. The compiler will transform PHP source code, written in a high-level, dynamically-typed scripting language, into executable machine code, enabling efficient and optimized execution on various platforms. This abstract outlines the key components and objectives of the PHP compiler project.

The PHP compiler will consist of several phases, including lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization, and code generation. These phases will be implemented using established compiler design principles and techniques.

Lexical Analysis:
The first phase involves tokenizing the PHP source code, breaking it down into meaningful units called tokens. Lexical analysis will identify keywords, operators, literals, and other language constructs, thereby facilitating subsequent processing.

Syntax Analysis:
The next phase focuses on parsing the tokenized code to ensure it adheres to the defined grammar of the PHP language. The syntax analysis phase will employ techniques such as top-down parsing or bottom-up parsing to construct a parse tree, representing the hierarchical structure of the code.

Semantic Analysis:
After establishing the code's syntactic correctness, the semantic analysis phase will verify its semantic integrity. This involves checking for type compatibility, variable scoping, and other language-specific rules. Any errors or inconsistencies will be reported to the user.

Intermediate Code Generation:
Once the code has been verified, an intermediate representation (IR) will be generated. The IR serves as an abstraction layer, enabling further optimization and facilitating the generation of executable code.

Optimization:
The compiler will employ various optimization techniques to enhance the performance and efficiency of the generated code. Optimization may include constant folding, dead code elimination, register allocation, and loop optimization, among others.

Code Generation:

In the final phase, the optimized intermediate representation will be translated into executable machine code specific to the target platform. The code generation process will consider platform-specific constraints and optimizations to produce efficient and portable code.

Throughout the project, careful consideration will be given to error handling, reporting, and debugging mechanisms. The PHP compiler will strive to provide meaningful error messages and debug information to aid developers in identifying and resolving issues in their code.

TABLE OF CONTENTS

Introduction

1. Architecture of Language

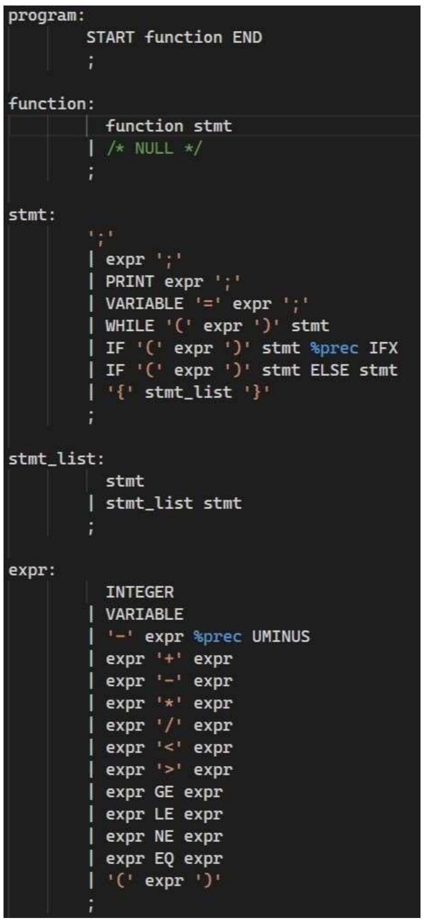
Our compiler supports the following language features :

- We handle variables of integer type only and it supports all integer operations.
- All types of arithmetic and logical expressions are handled.
- While and if-else statements are also handled.

2. Literature Survey

- Yacc Documentation
- Lex Documentation

3. Context Free Grammar



```
program:
        START function END
        ;

function:
          function stmt
        | /* NULL */
        ;

stmt:
        ';'
        | expr ';'
        | PRINT expr ';'
        | VARIABLE '=' expr ';'
        | WHILE '(' expr ')' stmt
        | IF '(' expr ')' stmt %prec IFX
        | IF '(' expr ')' stmt ELSE stmt
        | '{' stmt_list '}'
        ;

stmt_list:
          stmt
        | stmt_list stmt
        ;

expr:
          INTEGER
        | VARIABLE
        | '-' expr %prec UMINUS
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | expr '<' expr
        | expr '>' expr
        | expr GE expr
        | expr LE expr
        | expr NE expr
        | expr EQ expr
        | '(' expr ')'
        ;
```

4. Design Strategy

Symbol Table Creation :

- There are usually multiple scopes in a program.
- Scope is decided on how many open curly braces we see before we store the variable in the symbol table.
- Whenever we see a variable declared or initialized with a constant value, we store it in the symbol table.

Abstract Syntax Tree :

- The abstract syntax tree is generated as we parse the program.
- A tree node is created based on the type of tokens parsed.
- We handle three basic types of nodes, that is - constant, identifier and operator.
- All these nodes are built from bottom up to form the abstract syntax tree.

Intermediate Code Generation :

- To generate intermediate code, we also make use of the parse tree indirectly.
- We have written quadruple form of code to a file based on the syntax we are currently parsing in the program.
- We make use of multiple stacks for this process.

Code Optimization :

- We have performed reduction in number of live registers and constant folding optimization.
- We have performed the optimization using C and Python.
- We analyze the program line by line and use string manipulation and stack to perform these optimizations.

Error Handling :

- The scanner doesn’t crash when it comes across unknown symbols.
- The parser doesn’t stop parsing on encountering error and prints a syntax error at the corresponding line number.
- Semantic analyzer produces an error on uninitialized variables, undeclared variables and re-declaration of variables.
- The code generator expects error free code to be passed to it.

Target Code Generation :

- Target code is generated using a simple load-use-store mechanism.

- This is done by looking at quadruple address code line by line.

5. <u>Implementation Details</u>

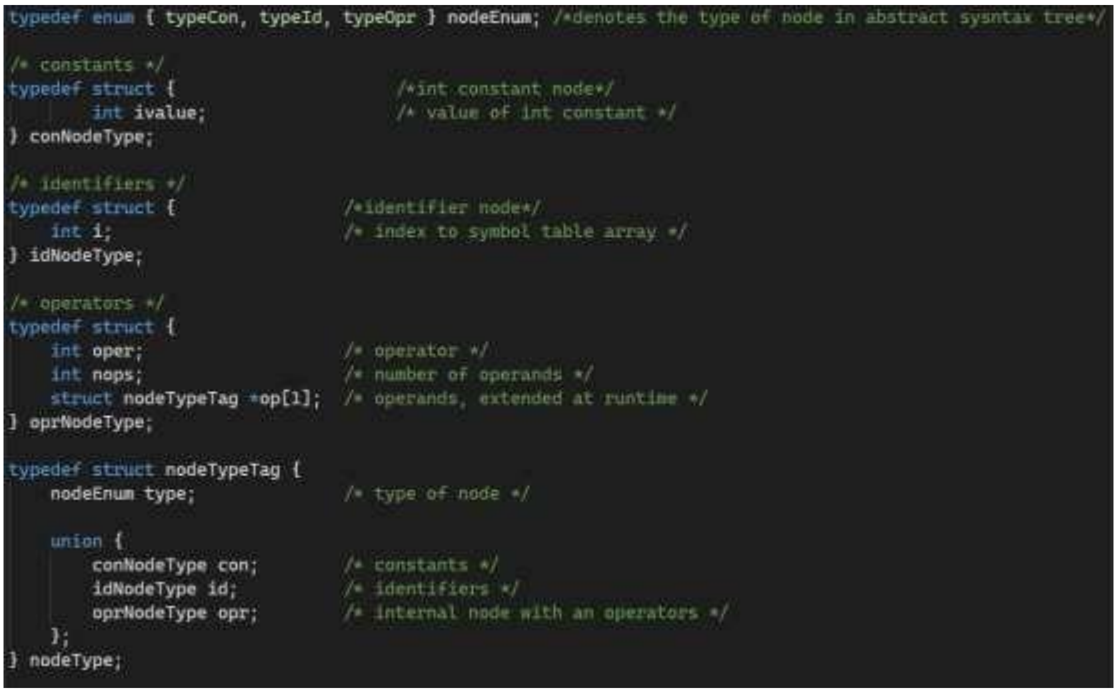
Symbol Table Creation :

- We use lex, yacc and custom code to generate the symbol table.
- We hold the type, value and name of variables in an array of structures.
- These array of structures are different for different scopes.
- The symbol table therefore has an array of scopes which store an array of variables.
- We also use a stack to maintain the current scope.

Abstract Syntax Tree :

- The abstract syntax tree basically consists of only three types of nodes that we have defined

-



```
typedef enum { typeCon, typeId, typeOpr } nodeEnum; /*denotes the type of node in abstract sysntax tree*/

/* constants */
typedef struct {                  /*int constant node*/
        int ivalue;               /* value of int constant */
} conNodeType;

/* identifiers */
typedef struct {                /*identifier node*/
    int i;                      /* index to symbol table array */
} idNodeType;

/* operators */
typedef struct {
    int oper;                   /* operator */
    int nops;                   /* number of operands */
    struct nodeTypeTag *op[1];  /* operands, extended at runtime */
} oprNodeType;

typedef struct nodeTypeTag {
    nodeEnum type;              /* type of node */

    union {
        conNodeType con;        /* constants */
        idNodeType id;          /* identifiers */
        oprNodeType opr;        /* internal node with an operators */
    };
} nodeType;
```

- These nodes are built bottom up using yacc.

Intermediate Code Generation :

- We maintain a stack of all the operators and identifiers we parse. There’s also a stack for maintaining the labels.
- When a production symbol is completely parsed, we pop from the stack and print it to a file according to the symbol we parsed.

- We make use of the labels stack when dealing with the while loop and switch construct. Appropriately consuming the stack to print labels for loop and case statements.
- Also an arithmetic code generation function, which generates suitable code for any required operator.

Code Optimization :

- We use Python and basic string manipulation to convert three address code into optimized three address code.
- For constant folding we just inspect every statement and use raw string manipulation.
- For dead code removal, we use Python sets to find out variables not being used in the code.

Assembly Code Generation :

- We use Python and basic string manipulation to convert optimized three address code into target assembly code.
- We use a hash table just to maintain the required condition variable to be loaded for the while loop.

Error Handling :

- In the parser, we use yacc’s built-in error handling mechanism.
- In the semantic analyzer we use the symbol table to catch any errors.

6. Instructions for using the compiler

Installing dependencies :

sudo apt update sudo apt install
flex sudo apt install bison

- Change your directory to PhP-Compiler.
- Write your code in sample.php file.
- Then run bash compiler.sh in the terminal of your linux machine.

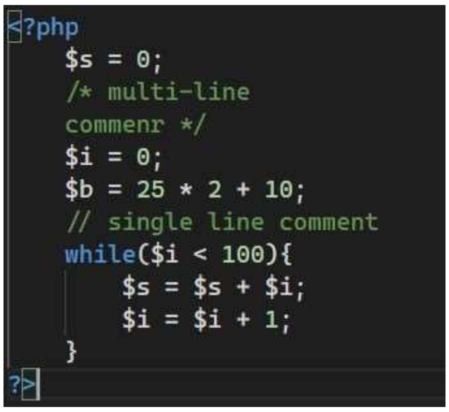
7. Results and Shortcomings

- Our compiler is a very minimal and basic, and handles programs which purely perform mathematical computations.
- Error handling of our compiler is not exhaustive and too simple to handle complicated errors.

- Assembly code output will be correct for any type of program that our grammar parses. However, the cost of the program is high due to a simple assembly generation algorithm.

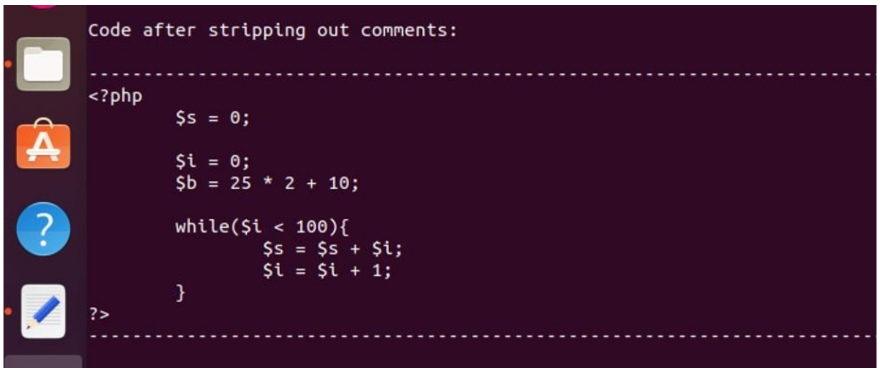
8. Snapshots

Sample input :



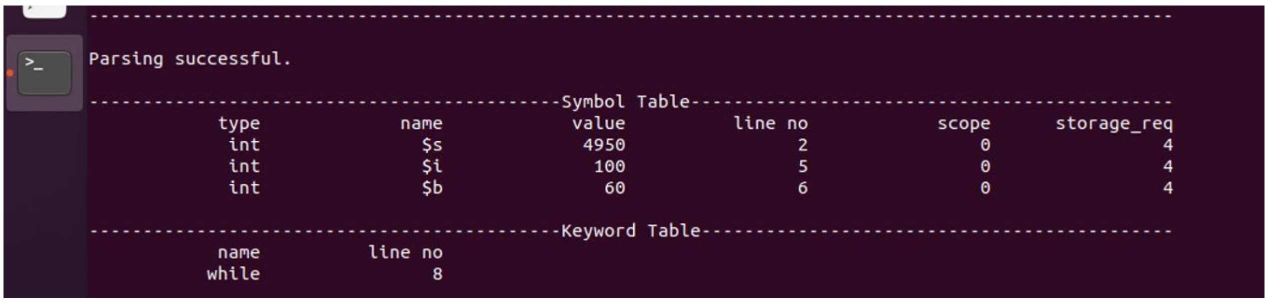
```
<?php
    $s = 0;
    /* multi-line
    commenr */
    $i = 0;
    $b = 25 * 2 + 10;
    // single line comment
    while($i < 100){
        $s = $s + $i;
        $i = $i + 1;
    }
?>
```

Code after stripping out comments :



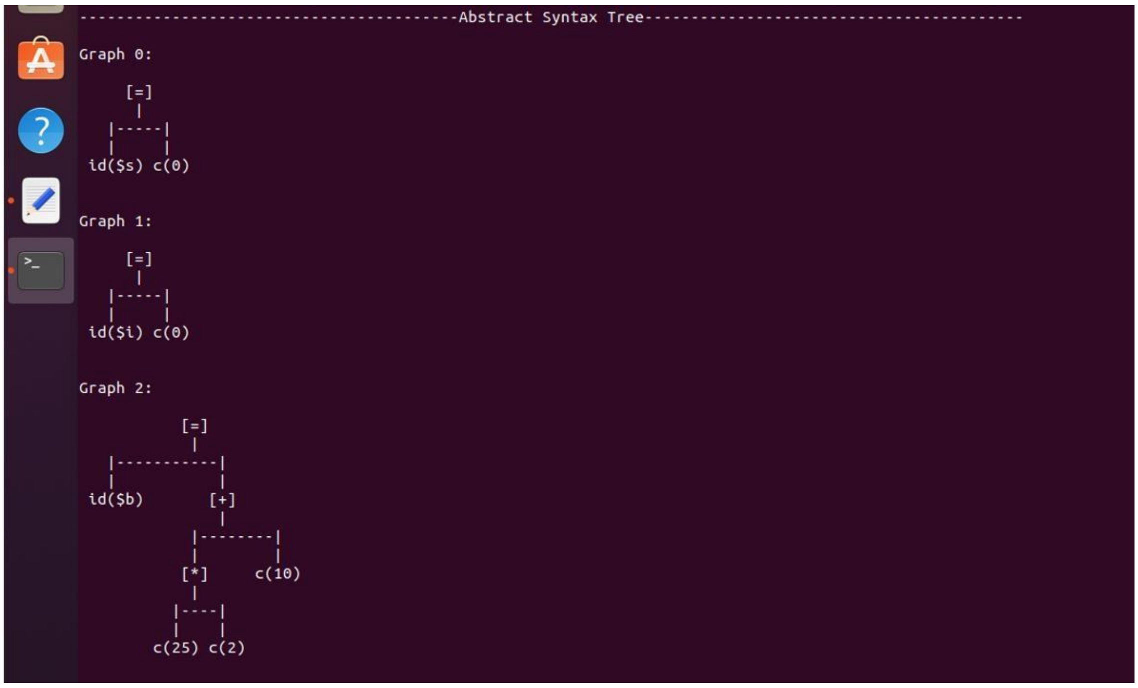
```
Code after stripping out comments:

------------------------------------------------------------------------
<?php
        $s = 0;

        $i = 0;
        $b = 25 * 2 + 10;

        while($i < 100){
                $s = $s + $i;
                $i = $i + 1;
        }
?>
------------------------------------------------------------------------
```

Symbol and keyword table :

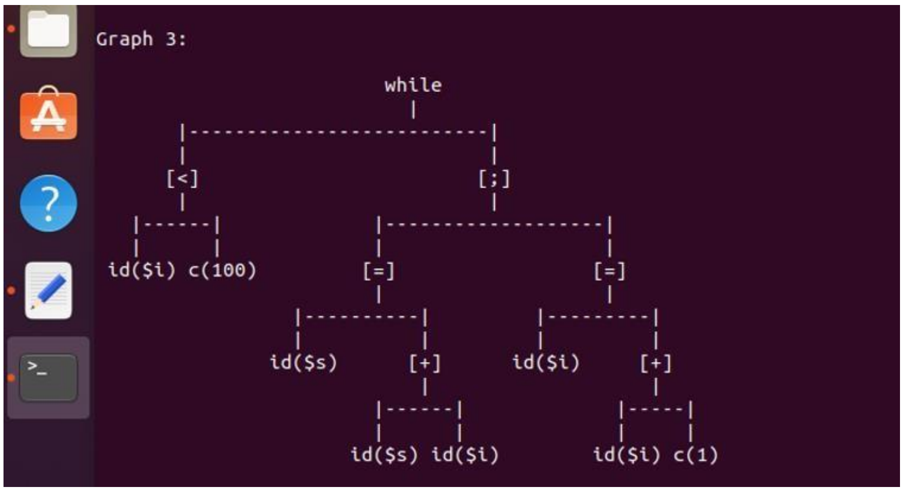


```
-----------------------------------------------------------------------------------------------------------

Parsing successful.

------------------------------------------Symbol Table------------------------------------------------------
            type            name            value            line no            scope        storage_req
             int              $s             4950                  2                0                  4
             int              $i              100                  5                0                  4
             int              $b               60                  6                0                  4

------------------------------------------Keyword Table-----------------------------------------------------
            name         line no
           while               8
```

Abstract syntax tree :

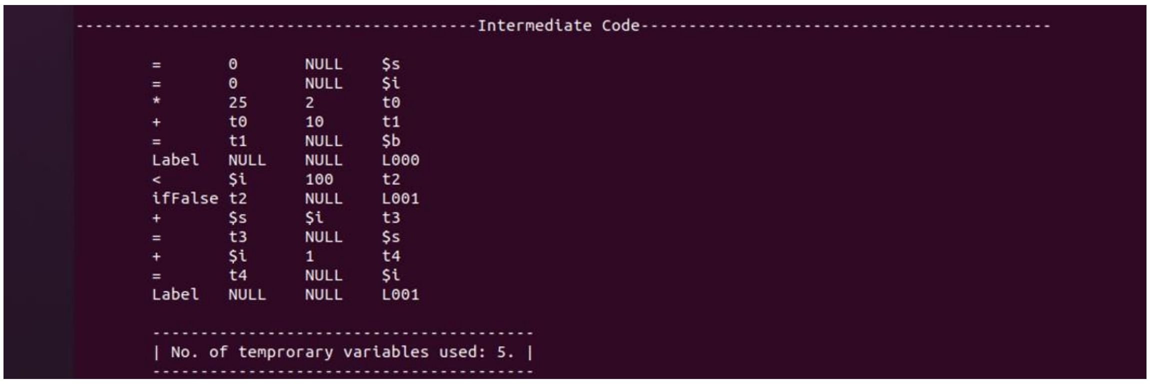


```
----------------------------------------Abstract Syntax Tree----------------------------------------
Graph 0:

     [=]
      |
   |-----|
   |     |
id($s) c(0)

Graph 1:

     [=]
      |
   |-----|
   |     |
id($i) c(0)

Graph 2:

           [=]
            |
   |-----------|
   |           |
id($b)        [+]
               |
          |--------|
          |        |
         [*]      c(10)
          |
        |----|
        |    |
      c(25) c(2)
```



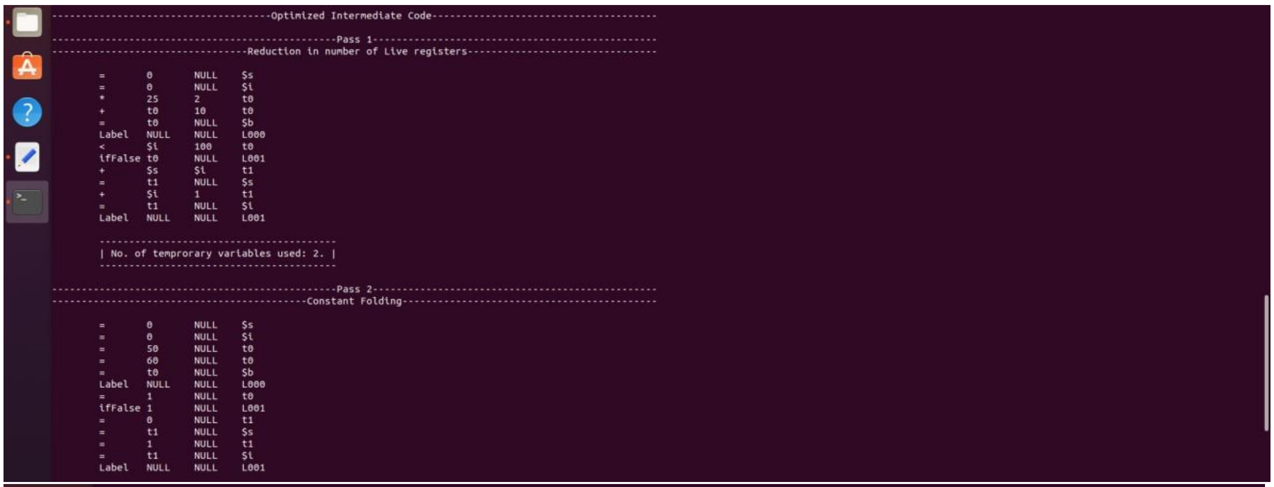
```
Graph 3:

                          while
                            |
       |----------------------------|
       |                            |
      [<]                          [;]
       |                            |
   |-------|              |-------------------|
   |       |              |                   |
id($i) c(100)            [=]                 [=]
                          |                   |
                |-----------|          |----------|
                |           |          |          |
              id($s)       [+]       id($i)      [+]
                            |                     |
                         |------|              |-----|
                         |      |              |     |
                      id($s) id($i)         id($i) c(1)
```

Intermediate code :

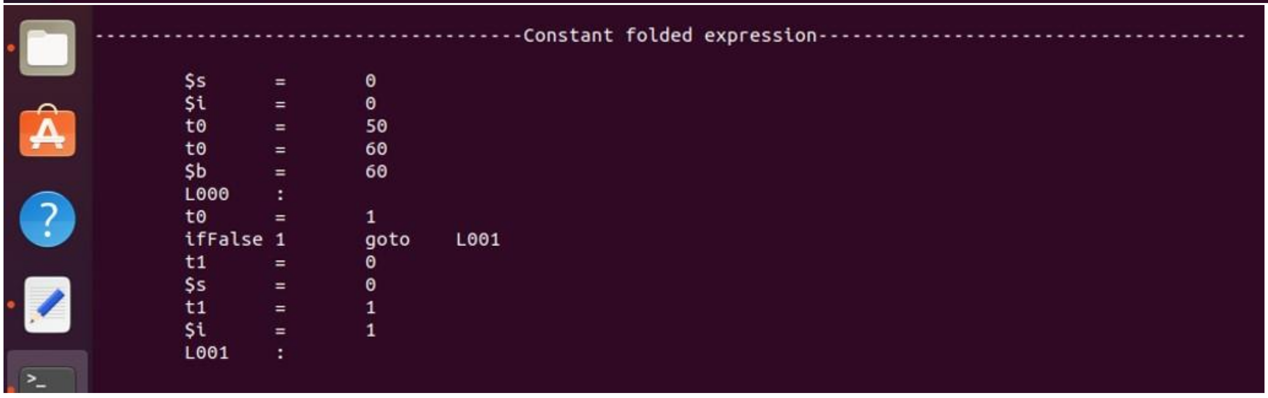


```
-----------------------------------------Intermediate Code------------------------------------------

        =       0       NULL    $s
        =       0       NULL    $i
        *       25      2       t0
        +       t0      10      t1
        =       t1      NULL    $b
        Label   NULL    NULL    L000
        <       $i      100     t2
        ifFalse t2      NULL    L001
        +       $s      $i      t3
        =       t3      NULL    $s
        +       $i      1       t4
        =       t4      NULL    $i
        Label   NULL    NULL    L001

        ---------------------------------------
        | No. of temprorary variables used: 5. |
        ---------------------------------------
```

Optimized intermediate code :

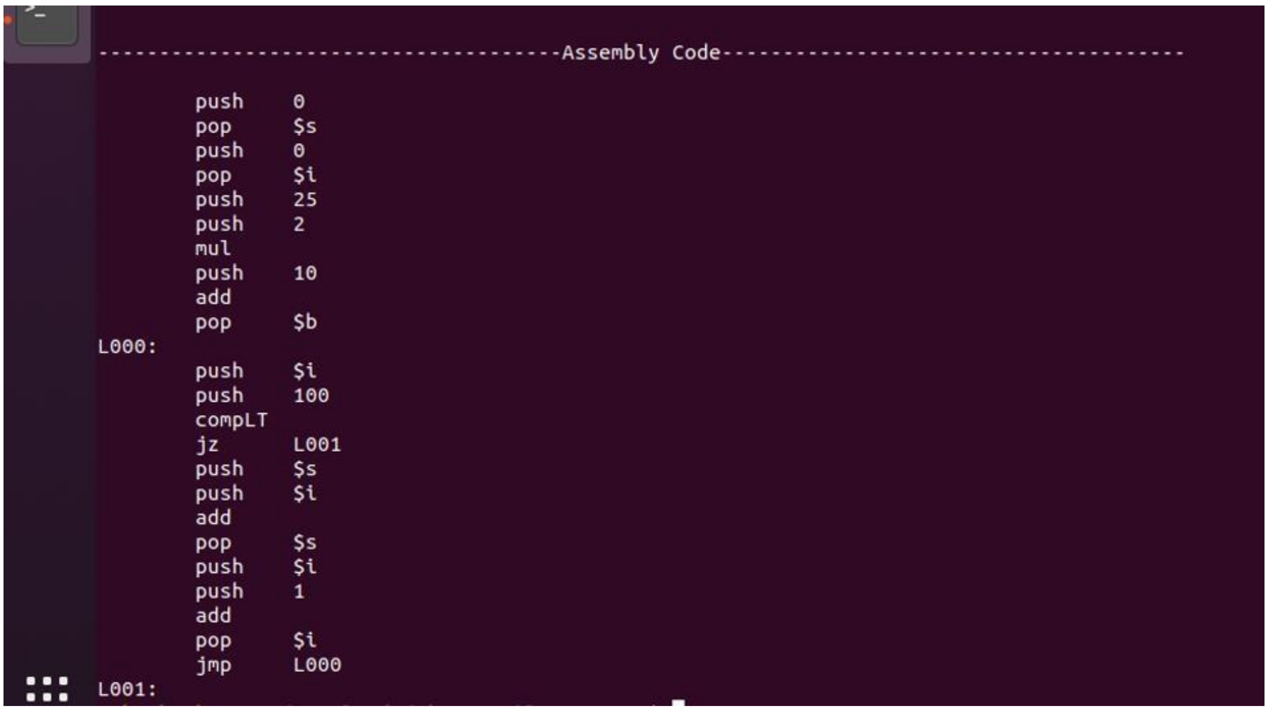


```
------------------------------------Optimized Intermediate Code------------------------------------

----------------------------------------------Pass 1----------------------------------------------
--------------------------------Reduction in number of Live registers-----------------------------

        =       0       NULL    $s
        =       0       NULL    $i
        *       25      2       t0
        +       t0      10      t0
        =       t0      NULL    $b
        Label   NULL    NULL    L000
        <       $i      100     t0
        ifFalse t0      NULL    L001
        +       $s      $i      t1
        =       t1      NULL    $s
        +       $i      1       t1
        =       t1      NULL    $i
        Label   NULL    NULL    L001

        ---------------------------------------
        | No. of temprorary variables used: 2. |
        ---------------------------------------

----------------------------------------------Pass 2----------------------------------------------
------------------------------------------Constant Folding----------------------------------------

        =       0       NULL    $s
        =       0       NULL    $i
        =       50      NULL    t0
        =       60      NULL    t0
        =       t0      NULL    $b
        Label   NULL    NULL    L000
        =       1       NULL    t0
        ifFalse 1       NULL    L001
        =       0       NULL    t1
        =       t1      NULL    $s
        =       1       NULL    t1
        =       t1      NULL    $i
        Label   NULL    NULL    L001
```



```
-------------------------------------Constant folded expression-------------------------------------

        $s      =       0
        $i      =       0
        t0      =       50
        t0      =       60
        $b      =       60
        L000    :
        t0      =       1
        ifFalse 1       goto    L001
        t1      =       0
        $s      =       0
        t1      =       1
        $i      =       1
        L001    :
```

Assembly code :



```
-----------------------------------Assembly Code-----------------------------------

        push    0
        pop     $s
        push    0
        pop     $i
        push    25
        push    2
        mul
        push    10
        add
        pop     $b
L000:
        push    $i
        push    100
        compLT
        jz      L001
        push    $s
        push    $i
        add
        pop     $s
        push    $i
        push    1
        add
        pop     $i
        jmp     L000
L001:
```

9. **Conclusions**

- It’s very easy to type a command to compile a program, but writing and understanding all phases of a compiler is challenging.
- Powerful tools like Lex and Yacc can be used in order to replicate or build a compiler.
- Working on this project has helped us grasp the internals and all the phases of a compiler.

10. **Further Enhancements**

- Handling more data types.
- Handling arrays, pointers etc.
- Function calls and argument parsing.
- More efficient assembly code generator.

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/
- https://www.javatpoint.com/code-generation
- https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lect s/final.codegen.pdf