*Report on*

# "JavaScript Mini - Compiler"

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

***Compiler Design Laboratory***

**Bachelor of Technology**
**in**
**Computer Science & Engineering**

***Submitted by:***

| | |
|---|---|
| **Alok Mehandale** | **PES1201701119** |
| **Navneetha Rajan** | **PES1201700161** |
| **Archana P** | **PES1201701543** |

*Under the guidance of*

**Mr. Prakash C O**

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 JavaScript programming language, focuses on generating an assembly code for the language for specific constructs. It works for constructs such as if-else statements and nested for loops. The main functionality of the project is to generate an assembly level code for the given JavaScript source code. This is done using the following steps:

i) Generate symbol table after performing expression evaluation
ii) Generate Abstract Syntax Tree for the code
iii) Generate 3 address code followed by corresponding quadruples
iv) Perform Code Optimization
v) Generate Assembly Code

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.
C is used to implement Abstract Syntax Tree using different data structures.
PYTHON is used to optimize the intermediate code generated by the parser. It is also used to generate assembly code.

## LITERATURE SURVEY AND OTHER 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://publicvoidlife.com/2014/12/18/yacc-program-implement-symbol-table-compiler/

## ARCHITECTURE OF LANGUAGE

JavaScript constructs implemented:

1. simple if
2. if-else
3. for-loop
4. if-else inside nested for-loop

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

## CONTEXT FREE GRAMMAR

```
STATEMENTS
    : STATEMENT STATEMENTS
    |     ;

STATEMENT
    : FOR_STATEMENT
    | BREAK SC
    | CONTINUE SC
    | VARIABLE_DECLARATION SC
    | ASSIGNMENT SC
    | IF_STATEMENT
    | SC
    ;

VARIABLE_DECLARATION
    : VAR VARIABLE_DECLARATOR
    ;

LITERAL
    : INT_LITERAL
    | STRING_LITERAL
    ;


VARIABLE_DECLARATOR
    :  IDENTIFIER MULTI_VAR
    | ASSIGNMENT MULTI_VAR
    ;
```

```
MULTI_VAR
    : ',' VARIABLE_DECLARATOR
    |
    ;

OPERAND
    : LITERAL
    | IDENTIFIER
    ;

ASSIGNMENT
    : IDENTIFIER '=' EXPRESSION
    ;

LOOP_INIT
    :
    SC
    | VARIABLE_DECLARATION SC
    | ASSIGNMENT SC
    | EXPRESSION SC
    ;

LOOP_COND
    : SC
    | EXPRESSION SC
    ;

FOR_STATEMENT
    : FOR '(' LOOP_INIT LOOP_COND EXPRESSION ')' BLOCK
    | error
    ;

BLOCK
    : '{' STATEMENTS '}'
    | SC
    ;

EXPRESSION
    : OPERAND
    | NUM_EXPRESSION
    | REL_EXPRESSION
    ;
NUM_EXPRESSION
    : OPERAND '+' OPERAND
    | OPERAND '-' OPERAND
    | OPERAND '*' OPERAND
    | OPERAND '/' OPERAND
    ;

REL_EXPRESSION
```

```
    : OPERAND G_OP OPERAND
    | OPERAND L_OP OPERAND
    | OPERAND GE1_OP OPERAND
    | OPERAND LE1_OP OPERAND
    | OPERAND EQ1_OP OPERAND
    | OPERAND NE1_OP OPERAND
    | OPERAND AND1_OP OPERAND
    | OPERAND OR1_OP OPERAND
    ;

IF_STATEMENT
    : IF '(' EXPRESSION ')' BLOCK ELSE_BLOCK
    ;

ELSE_BLOCK
    : ELSE BLOCK
    |
    ;

SC
    : ';'
    ;
```

## DESIGN STAGES AND IMPLEMENTATION

### Phase 1: (a)Lexical Analysis

- LEX tool was used to create a scanner for JavaScript language.
- The scanner transforms the source file 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 are defined in the Lex file.
- 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.

### Phase 1: (b) Syntax 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. Dynamic variable type
    3. Arithmetic and Boolean expressions containing relational operators
    4. Postfix and prefix expressions
    5. Constructs – nested if-else, nested for-loop

- Yacc tool is used for parsing. It reports shift-reduce and reduce-reduce conflicts on parsing an ambiguous grammar.

## Phase 2: Symbol Table creation

- A structure is maintained to keep track of the variables, constants, operators and the keywords in the input. The parameters of the structure are the name of the token, the line number of occurrence, the category of the token (constant, variable, keyword, operator) and the value that it holds the datatype.

```
typedef struct symtable
{
    char *name;
    char *type;
    char *value;
    int lineno;
} SymbolTable;
```

- As each line is parsed, the actions associated with the grammar rules are executed. Symbol table functions such as createEntry, putkwd, and get_Pos are called appropriately with each production rule.

- $1 is used to refer to the first token in the given production and $$ is used to refer to the resultant of the given production.

- Expressions are evaluated and the values of the used variables are updated accordingly.

- At the end of the parsing, the updated symbol table is displayed.

- For the sake of this project, we declare variables using **var** keyword and this generally only identifies them to a global scope. Since we are not implementing functions, scope is not included in the symbol table.

## Phase 3: Abstract Syntax Tree

- A structure is maintained to keep track of each AST node. This uses an op code mainly to identify itself as a different node

- op code tells us what the node currently holds. It can hold numbers, strings, operators, and different kinds of complex statements. If there are more than 2 children it needs to hold, then it is set as LIST type, and children are recursively added as child nodes.

```
typedef struct abstract_syntax_tree {
   enum code op;
   int val;
   struct symbol *sym;
   struct abstract_syntax_tree *left, *right;
   char *str;
} AST;
```

- op code tells us what the node currently holds. It can hold numbers, strings, operators, and different kinds of complex statements. The type that is used to hold these is the LIST type.

- eg. **a=1** would be 2 AST nodes.
  - One node to hold numeric 1 value, of type NUM
  - One node pointing to symbol table entry for variable **a**

- AST is created using the symbol table when required. Once the tree is generated, we can get a complete picture of the execution order of the program and comprehend the logic it implements.

- At the end of this, all symbol table values have been updated and code is ready to be optimized from the intermediate stage.

## Phase 4: 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.

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 statements are of the form x = y op z, here x, y, z will have an address (memory location).

Example – The three address code for the expression a + b * c + d:

T 1 = b * c
T 2 = a + T 1
T 3 = T 2 + d
T1, T2, T3 are temporary variables.

The data structure used to represent Three address Code is the Quadruples. It is shown with 4 columns- operator, operand1, operand2, and result.

For every assignment operation, the values are stored as quadruples. Then various functions are called in the corresponding grammar to convert it to an intermediate code.

## Phase 5: Code Optimization

The code optimizer maintains a key-value mapping that resembles the symbol table structure to keep track of variables and their values (possibly after expression evaluation). This structure is used to perform constant propagation and constant folding in sequential blocks followed by dead code elimination.

| Sample Input(Quadruples) | Sample Input(3 Address Code) |
|---|---|
| = 3 NULL a<br>+ a 5 b<br>+ a b c<br>* c e d<br>= 8 NULL a<br>* a 2 f<br>if x NULL L0 | a = 3<br>b = a + 5<br>c = a + b<br>d = c * e<br>a = 8<br>f = a * 2<br>if ( x ) L0: |

## Phase 6: ASSEMBLY CODE GENERATION

The assembly code was generated with the help of a python file that takes quadruples generated from the previous phase of code optimisation. In this python file, we have an entry function that identifies what kind of a statement the quadruple represents, and accordingly calls another function to convert that line into assembly code.

We have different functions defined for different kinds of statements, namely, Branch, Mov, mathematical operations like ADD, SUB,etc. and for lines with relational operators.

In each of these functions, we load the value from the memory, compute the needed values and then store the register values back to the memory. This way we use a maximum of just 3 registers for any input. However, this would be inefficient as we load from the memory each time, instead of reusing register values.

A dictionary mapping of variables to registers and their values would allow us to reuse registers, but without a lookahead scheme, would require for us to have a limit on the number of registers used.

## 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 intermediate code, given a JavaScript code as input.

There are a few shortcomings with respect to our implementation. The symbol table structure is same across all types of tokens (constants, identifiers and operators).

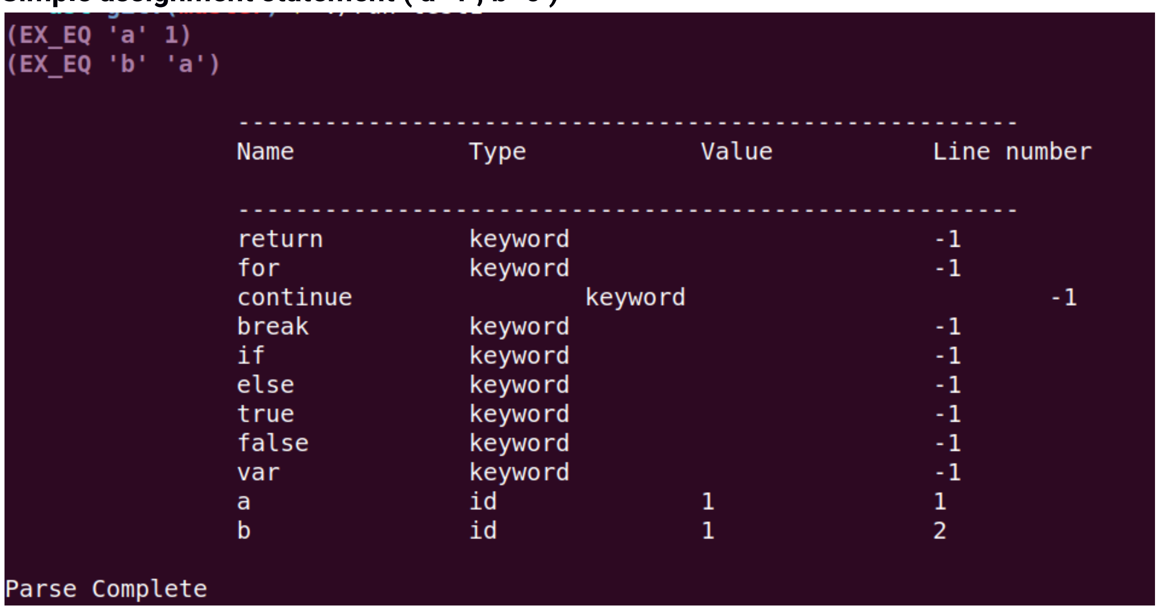
This leads to some fields being empty for some of the tokens. This can be optimized by using a better representation.

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

The Assembly code generated, loads and stores values from the memory for every computation. This can be avoided be using a lookahead scheme with a mapping of register values and memory locations in order to reuse the loaded register values.

## SNAPSHOTS4

**Simple assignment statement ( a=1 ; b=0 )**



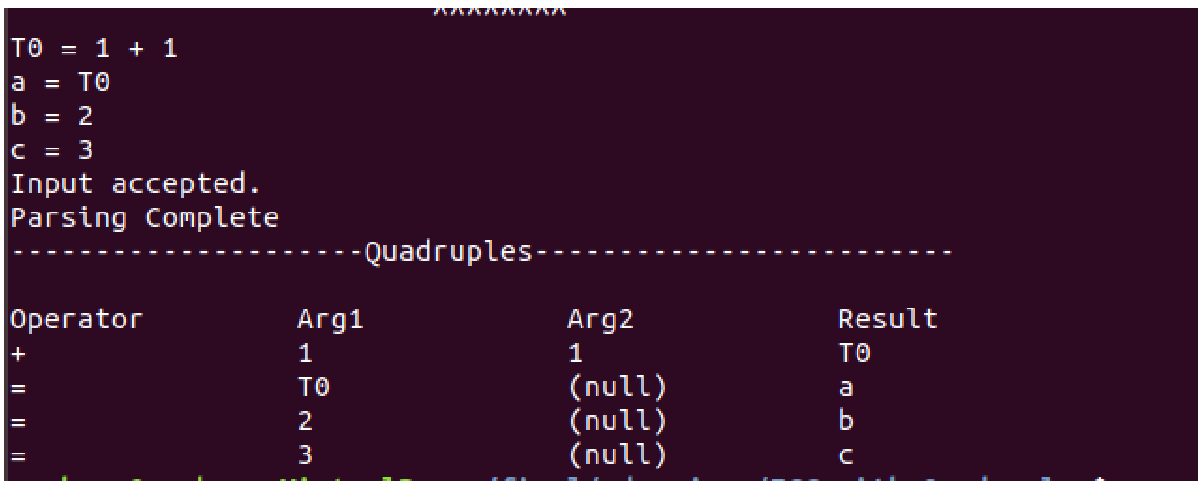
```
(EX_EQ 'a' 1)
(EX_EQ 'b' 'a')

                ------------------------------------------------------
                Name            Type            Value           Line number

                ------------------------------------------------------
                return          keyword                         -1
                for             keyword                         -1
                continue                keyword                         -1
                break           keyword                         -1
                if              keyword                         -1
                else            keyword                         -1
                true            keyword                         -1
                false           keyword                         -1
                var             keyword                         -1
                a               id              1               1
                b               id              1               2

Parse Complete
```

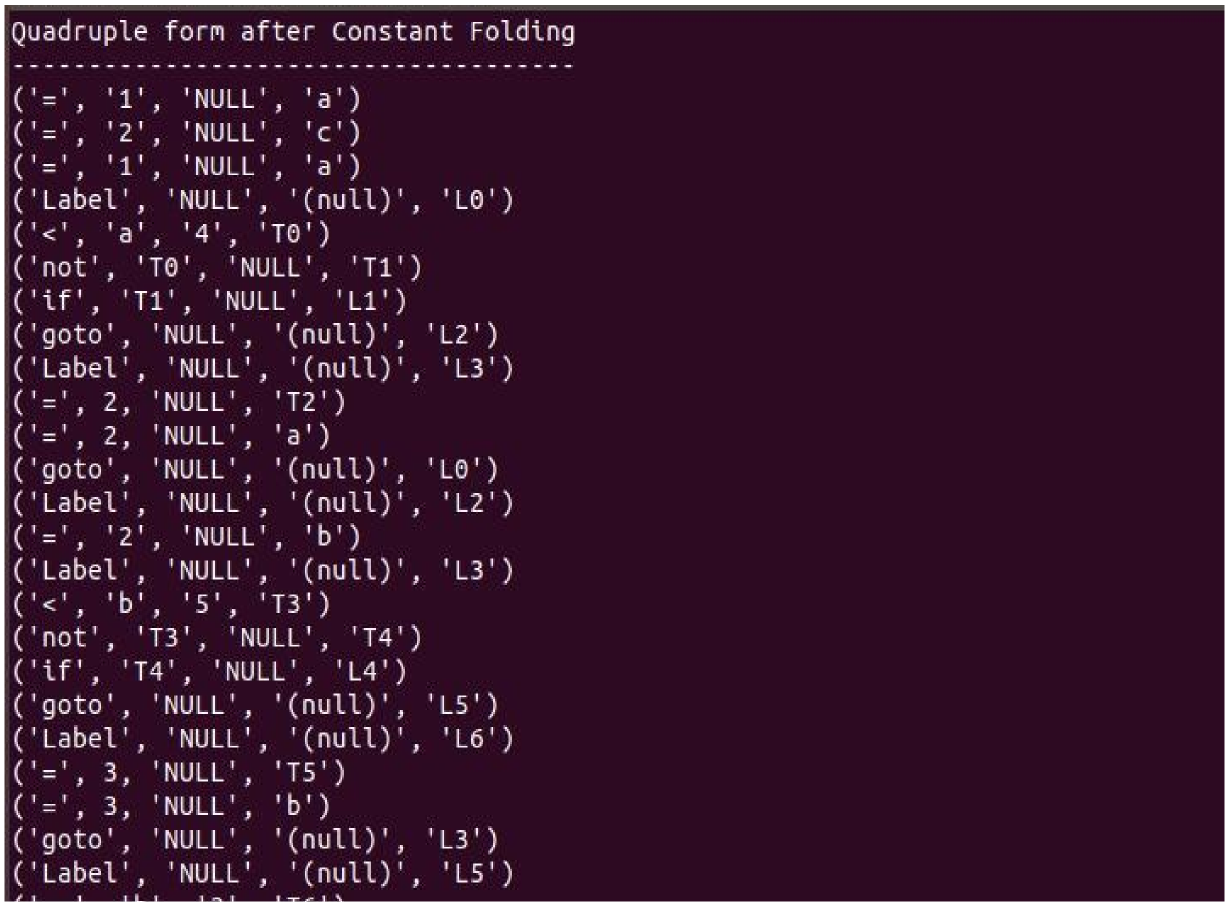
**Intermediate code with Quadruples**



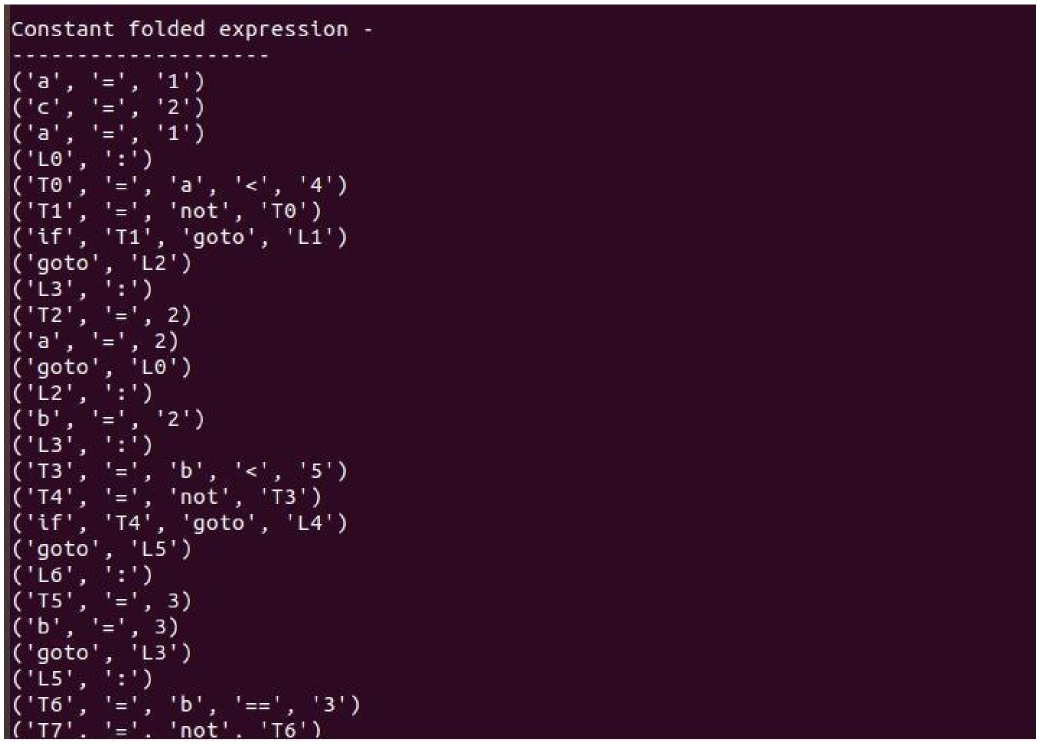
```
T0 = 1 + 1
a = T0
b = 2
c = 3
Input accepted.
Parsing Complete
---------------------Quadruples------------------------

Operator         Arg1            Arg2            Result
+                1               1               T0
=                T0              (null)          a
=                2               (null)          b
=                3               (null)          c
```

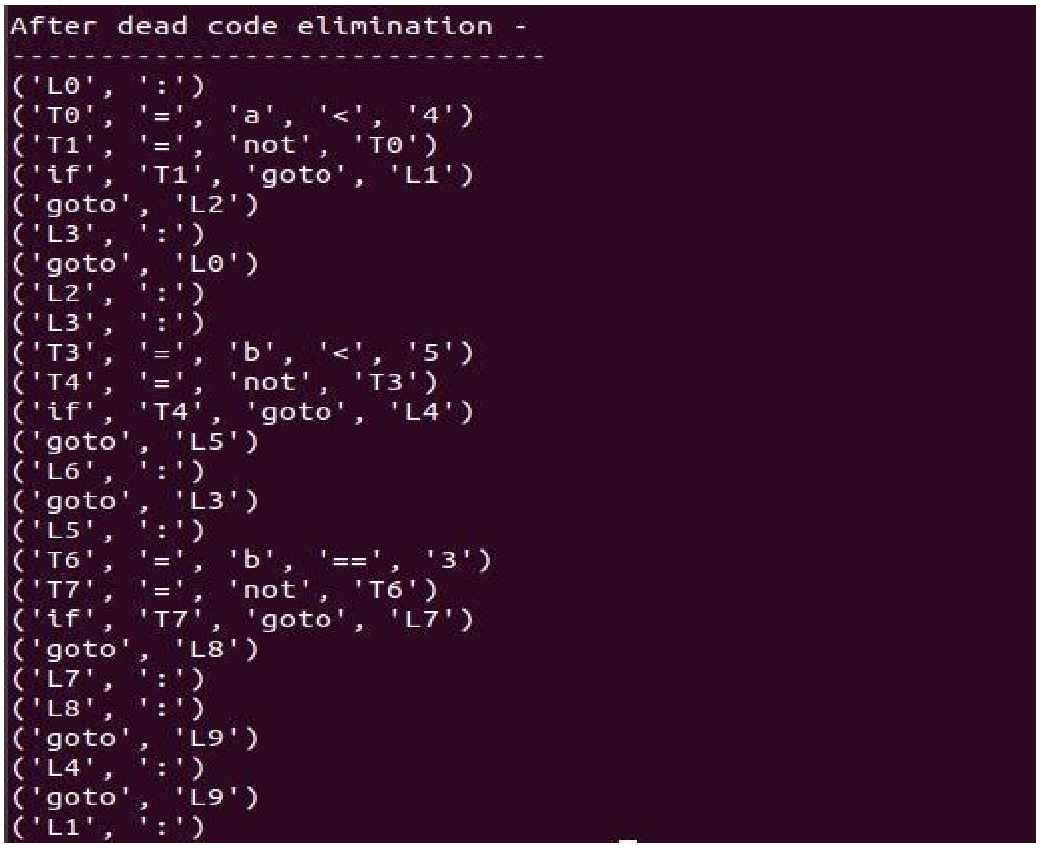
**Optimisations_____**



```
Quadruple form after Constant Folding
-------------------------------------
('=', '1', 'NULL', 'a')
('=', '2', 'NULL', 'c')
('=', '1', 'NULL', 'a')
('Label', 'NULL', '(null)', 'L0')
('<', 'a', '4', 'T0')
('not', 'T0', 'NULL', 'T1')
('if', 'T1', 'NULL', 'L1')
('goto', 'NULL', '(null)', 'L2')
('Label', 'NULL', '(null)', 'L3')
('=', 2, 'NULL', 'T2')
('=', 2, 'NULL', 'a')
('goto', 'NULL', '(null)', 'L0')
('Label', 'NULL', '(null)', 'L2')
('=', '2', 'NULL', 'b')
('Label', 'NULL', '(null)', 'L3')
('<', 'b', '5', 'T3')
('not', 'T3', 'NULL', 'T4')
('if', 'T4', 'NULL', 'L4')
('goto', 'NULL', '(null)', 'L5')
('Label', 'NULL', '(null)', 'L6')
('=', 3, 'NULL', 'T5')
('=', 3, 'NULL', 'b')
('goto', 'NULL', '(null)', 'L3')
('Label', 'NULL', '(null)', 'L5')
```

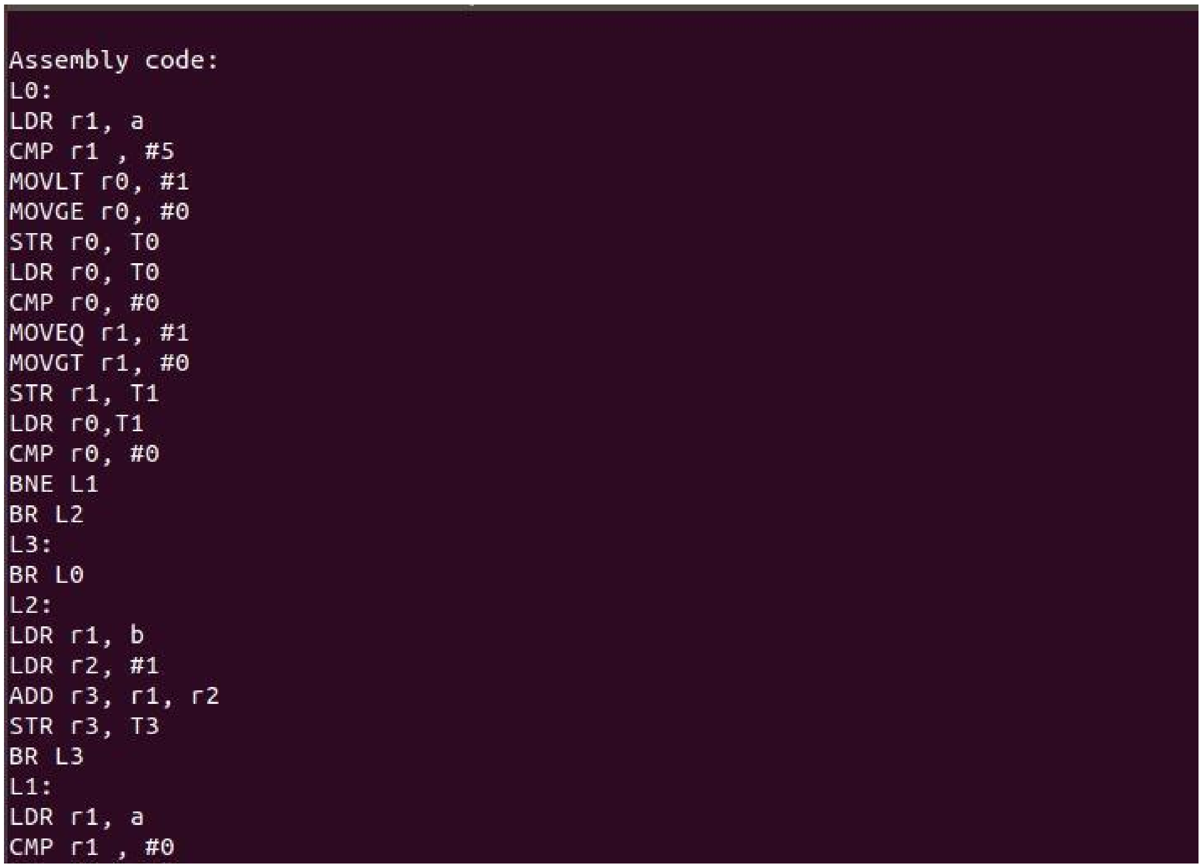


```
Constant folded expression -
--------------------
('a', '=', '1')
('c', '=', '2')
('a', '=', '1')
('L0', ':')
('T0', '=', 'a', '<', '4')
('T1', '=', 'not', 'T0')
('if', 'T1', 'goto', 'L1')
('goto', 'L2')
('L3', ':')
('T2', '=', 2)
('a', '=', 2)
('goto', 'L0')
('L2', ':')
('b', '=', '2')
('L3', ':')
('T3', '=', 'b', '<', '5')
('T4', '=', 'not', 'T3')
('if', 'T4', 'goto', 'L4')
('goto', 'L5')
('L6', ':')
('T5', '=', 3)
('b', '=', 3)
('goto', 'L3')
('L5', ':')
('T6', '=', 'b', '==', '3')
('T7', '=', 'not', 'T6')
```



```
After dead code elimination -
------------------------------
('L0', ':')
('T0', '=', 'a', '<', '4')
('T1', '=', 'not', 'T0')
('if', 'T1', 'goto', 'L1')
('goto', 'L2')
('L3', ':')
('goto', 'L0')
('L2', ':')
('L3', ':')
('T3', '=', 'b', '<', '5')
('T4', '=', 'not', 'T3')
('if', 'T4', 'goto', 'L4')
('goto', 'L5')
('L6', ':')
('goto', 'L3')
('L5', ':')
('T6', '=', 'b', '==', '3')
('T7', '=', 'not', 'T6')
('if', 'T7', 'goto', 'L7')
('goto', 'L8')
('L7', ':')
('L8', ':')
('goto', 'L9')
('L4', ':')
('goto', 'L9')
('L1', ':')
```

**Assembly Code**



```
Assembly code:
L0:
LDR r1, a
CMP r1  , #5
MOVLT r0, #1
MOVGE r0, #0
STR r0, T0
LDR r0, T0
CMP r0, #0
MOVEQ r1, #1
MOVGT r1, #0
STR r1, T1
LDR r0,T1
CMP r0, #0
BNE L1
BR L2
L3:
BR L0
L2:
LDR r1, b
LDR r2, #1
ADD r3, r1, r2
STR r3, T3
BR L3
L1:
LDR r1, a
CMP r1  , #0
```

## FUTURE ENHANCEMENTS

As mentioned above, we can use separate structures for the different types of tokens and then declare a union of these structures. This way, memory will be properly utilized.

For constant propagation at branches, we need to implement SSA form of the code. This will work well in all cases and yield the right output.

An ideal implementation in assembly code for reusing register values would be with a lookahead scheme, which would make it more efficient.