Report on

“Javascript Mini Complier for ’for’,’if-else’ and ‘while’ constructs”

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

Compiler Design Laboratory

Bachelor of Technology

in

Computer Science & Engineering

Submitted by:

Jyosna.S	**PES1201700792**
Megha.M.	**PES1201700948**
Kona Supriya	**PES1201701117**

Under the guidance of

Kiran P

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:

A compiler is a computer program that transforms source code written in a programming language into another computer language (the target language), with the latter often having a binary form known as object code.

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

1. For Loop

2. If-else

3. While Loop

Given source program in JS can be translated to a symbol table, abstract syntax tree, intermediate code, optimized intermediate code and the target code (assemble language).

ARCHITECTURE OF LANGUAGE:

- Used lex to create the scanner for our language.
- Used 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)

- Ignore comments and white-spaces
    - Single line comments starting with //
    - Multi-line comments enclosed within /* ...... */
- Types: var

- Constructs 'for' loop 'while' loop, 'if' loop and 'if-else' loop.
- Includes function definition.
- No conflicts and errors in our code/grammar.
- Warnings and Error recovery:

  Errors:-
  - Use of undeclared identifiers
  - Redefinition of identifiers within the same scope
  - Use of undeclared identifiers
  - Invalid operands to the operators.
  - Missing braces for if-else,for loop and even function.
  - 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.

- Code Optimizations techniques used :
  - Common Subexpression elimination
  - Constant folding
  - Dead Code Elimination

LITERATURE SURVEY:

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

- https://www.lysator.liu.se/c/ANSI-C-grammar-y.html

- https://stackoverflow.com/questions/5175840/is-html-a-context-free-language
- https://stackoverflow.com/questions/2320402/how-to-define-a-grammar-for-a-programming-language - Helped us write the grammar for our compiler

- https://github.com/SiddhiKK/LexicalAnalyzer/blob/master/lexicalanalyzer.l -Reference link for writing the code and taking the ideas.

- https://www.tutorialspoint.com/compiler_design/compiler_design_code_generation.html -Reference link for target code generation.

CONTEXT-FREE GRAMMAR:

```
Body -> FunctionDeclaration
        | FunctionDeclaration Body
        | Statement
        | Statement Body

Statements -> Statement Statements |

Statement -> ';'
          | if_x Condition '{' Statements  '}'
          | if_x Condition '{' Statements '}'else_x '{' Statements'}'
          |for_x '('Statement Statement Expression ')' '{' Statements '}'
          |while_x Condition ‘{‘ Statements ‘}’
          |break_x';'
          |continue_x';'
          |return_x ExpressionOpt ';'
          |VariablesOrExpression ';'
          |Statements

Condition -> '(' Expression ')'

VariablesOrExpression -> var Variables
                      | Expression
```

Variables ->Variable
| Variable ',' Variables

Variable -> identifier
| identifier AssignmentOperator X

ExpressionOpt -> Expression
| number
| identifier

Expression -> X LogicalOperator X
| X RelationalOperator X
| X ArithematicOperator
| ArithematicOperator X
| X AssignmentOperator X |

X -> X '+' Y
| X '-' Y
| Y

Y -> Y '*' Z
| Y '/' Z
| Z

Z -> identifier
| number

```
FunctionDeclaration  -> function_x identifier '(' ParameterListOpt ')' '{'
                        Statements '}'

ParameterListOpt ->ParameterList |

ParameterList->identifier
                | identifier ',' ParameterList


identifier->  letter
            | identifier letter
            | identifier number

letter-> A...Z
        | a..z

number-> digit
          | number digit

digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9


LogicalOperator -> ||
                    | &&
                    | !
RelationalOperator -> < | > | <= | >= | == | !=
ArithematicOperator -> ++ | --
```

DESIGN AND IMPLEMENTATION:

Design
Language: Javascript
Tools : Lex and Yacc
Constructs : for ,if , if-else and function declaration

- 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), Scope , Datatype , Value

- **Abstract Syntax Tree** : We have implemented a binary tree to represent the abstract syntax tree internally , we have executed this for the 'for' ,'if' , 'if-else' 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.
- **Code Optimization** *:* To increase efficiency the code optimization is done on the generated ICG. We have implemented constant folding and variable propagation.
- **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 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.

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.
- Semantics Analysis uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct.
- 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.
- We have written appropriate rules to check for semantic validity (declare before use, appropriate open and close bracesetc.)
- Variables must be declared as var .
- 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.
- Using the output of the intermediate code generation we are generating the target code ie the assembly code as the final output.

Commands to execute the code:

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

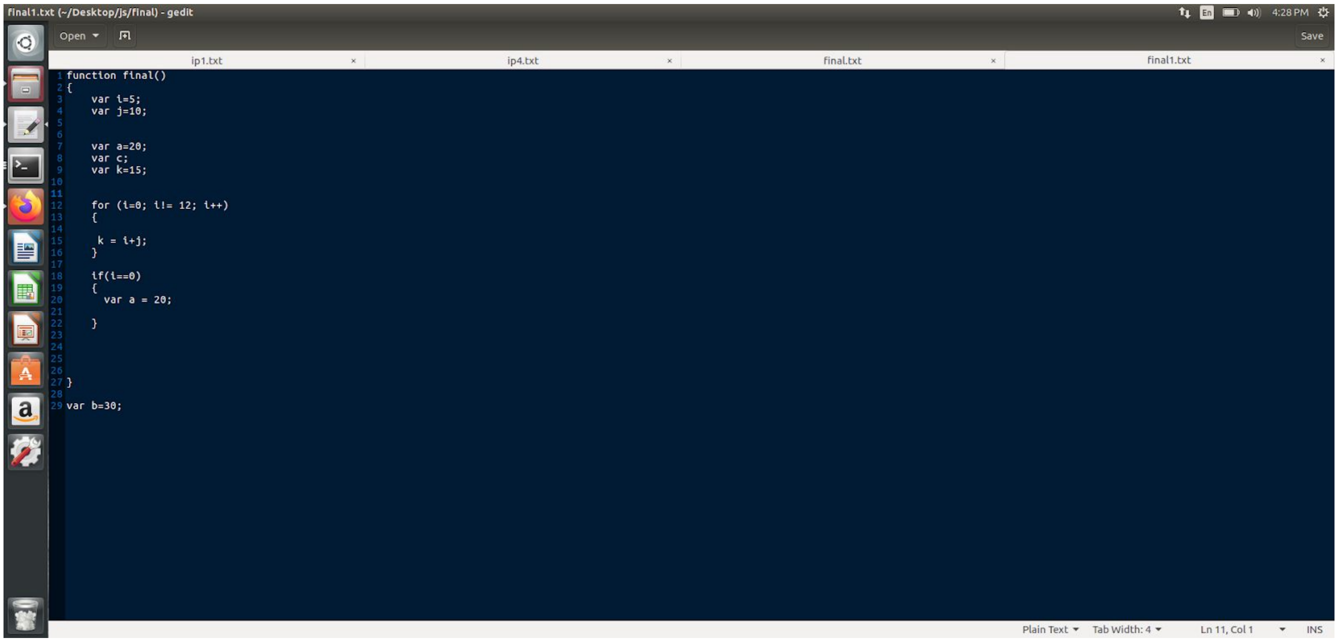
After parsing, if there are errors then the line numbers of those errors are displayed along with a ‘parsing failed’ on the terminal. Otherwise, a ‘parsing complete’ message is displayed on the console. The symbol table with stored & updated values is always displayed, irrespective of errors. Also, the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

SHORTCOMINGS:

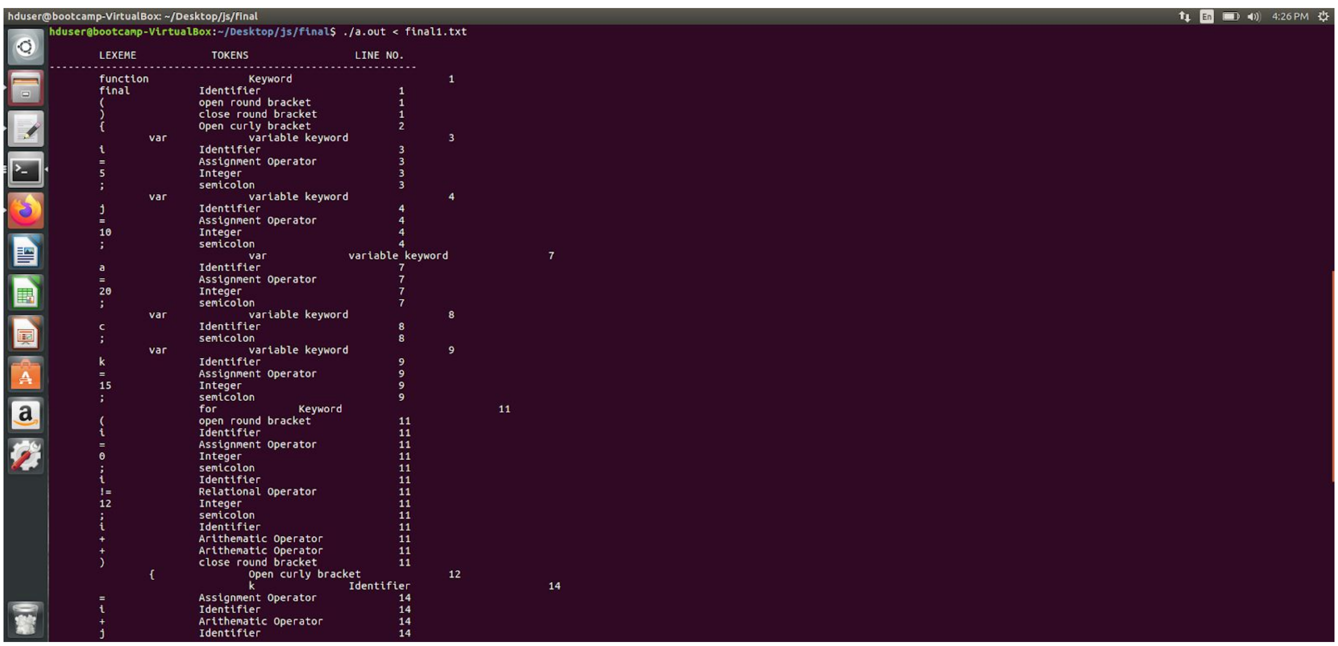
- Traversing the symbol table is time consuming as we have implemented a linked list,no random access possible.
- Currently, the abstract syntax tree is represented as a flat list in pre-order manner. The interpretation might be difficult in this case.

SNAPSHOTS:

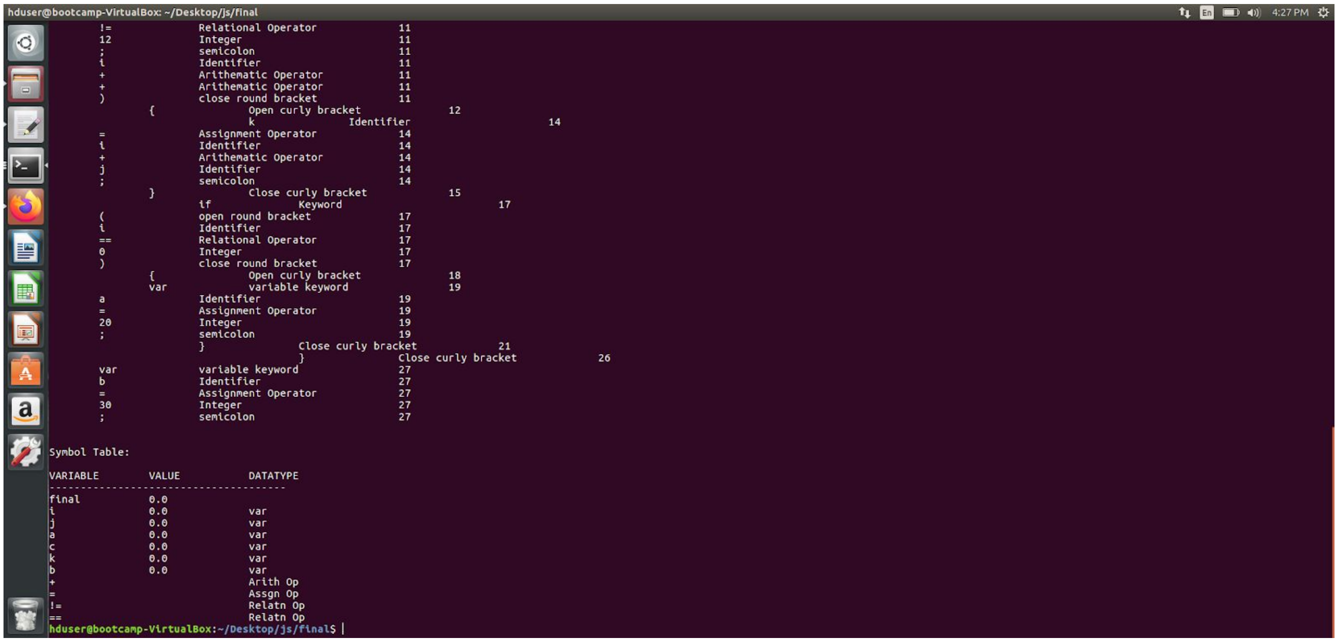
1. **Token generation:**



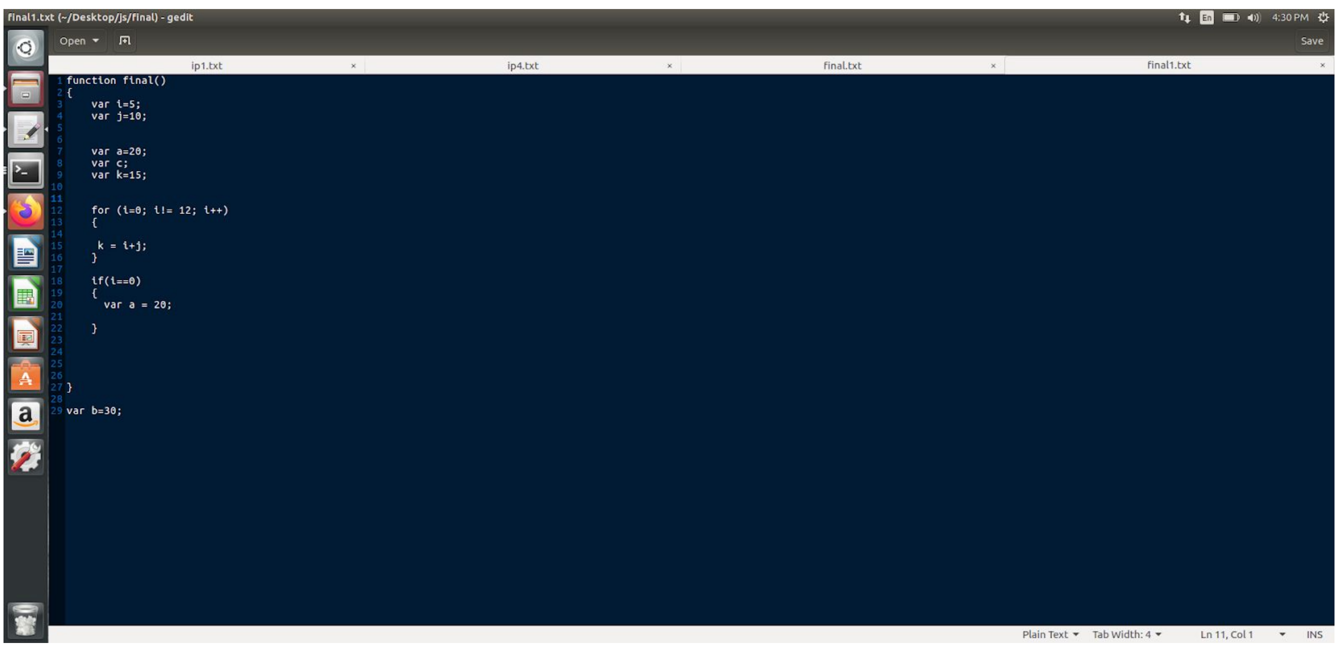

```
function final()
{
    var i=5;
    var j=10;


    var a=20;
    var c;
    var k=15;


    for (i=0; i!= 12; i++)
    {

     k = i+j;
    }

    if(i==0)
    {
      var a = 20;

    }




}

var b=30;
```

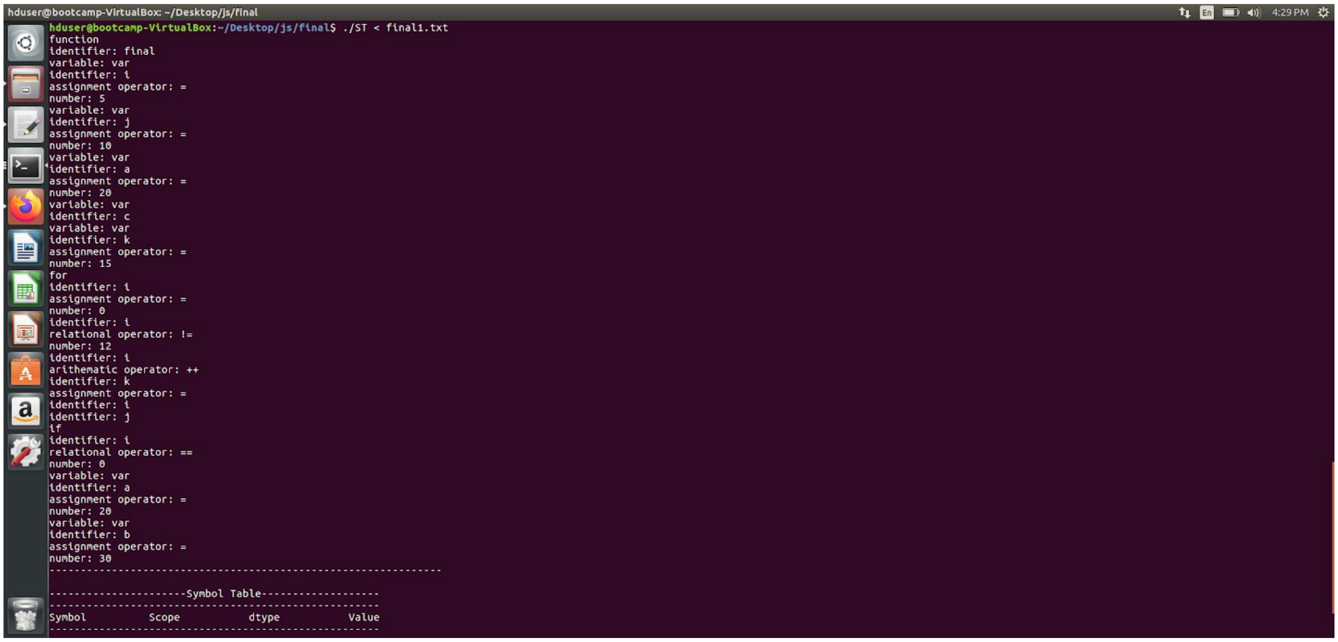



```
hduser@bootcamp-VirtualBox:~/Desktop/js/final$ ./a.out < final1.txt

        LEXEME            TOKENS                LINE NO.
-----------------------------------------------------------
        function                Keyword                         1
        final           Identifier                      1
        (               open round bracket              1
        )               close round bracket             1
        {               Open curly bracket              2
                var             variable keyword                3
        i               Identifier                      3
        =               Assignment Operator             3
        5               Integer                         3
        ;               semicolon                       3
                var             variable keyword                4
        j               Identifier                      4
        =               Assignment Operator             4
        10              Integer                         4
        ;               semicolon                       4
                        var             variable keyword                7
        a               Identifier                      7
        =               Assignment Operator             7
        20              Integer                         7
        ;               semicolon                       7
                var             variable keyword                8
        c               Identifier                      8
        ;               semicolon                       8
                var             variable keyword                9
        k               Identifier                      9
        =               Assignment Operator             9
        15              Integer                         9
        ;               semicolon                       9
                        for             Keyword                         11
        (               open round bracket              11
        i               Identifier                      11
        =               Assignment Operator             11
        0               Integer                         11
        ;               semicolon                       11
        i               Identifier                      11
        !=              Relational Operator             11
        12              Integer                         11
        ;               semicolon                       11
        i               Identifier                      11
        +               Arithematic Operator            11
        +               Arithematic Operator            11
        )               close round bracket             11
                {               Open curly bracket              12
                        k               Identifier                      14
        =               Assignment Operator             14
        i               Identifier                      14
        +               Arithematic Operator            14
        j               Identifier                      14
```

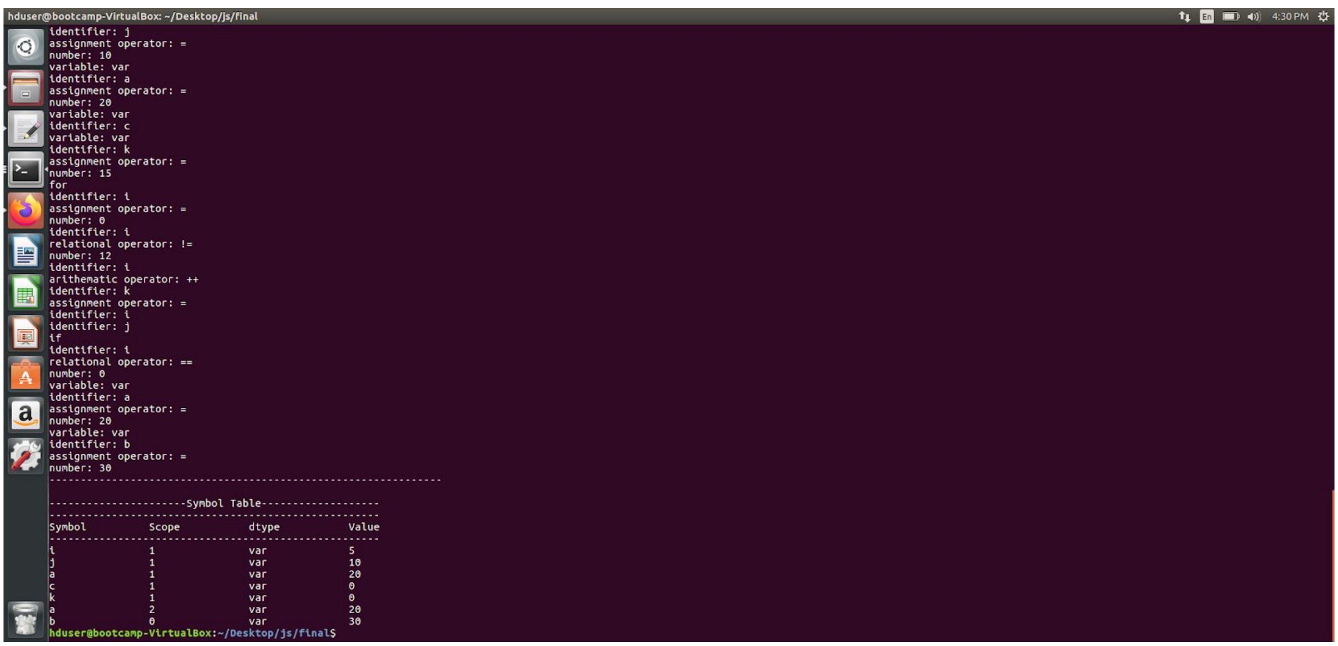


2. Symbol Table:



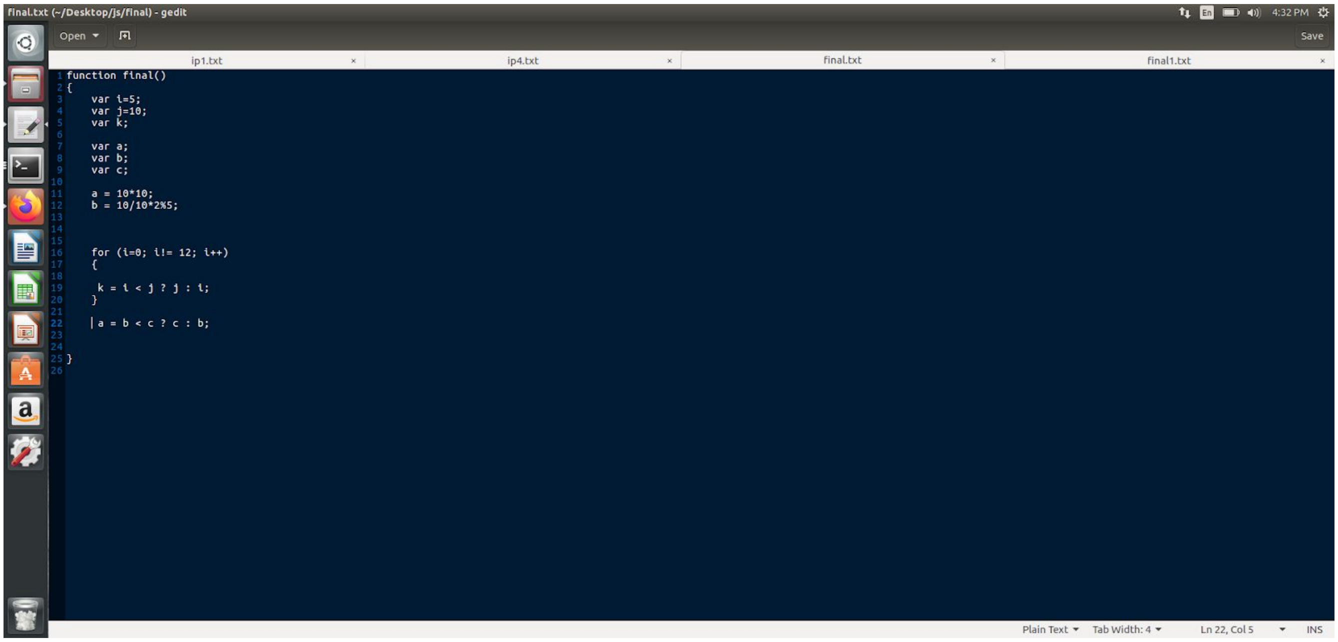


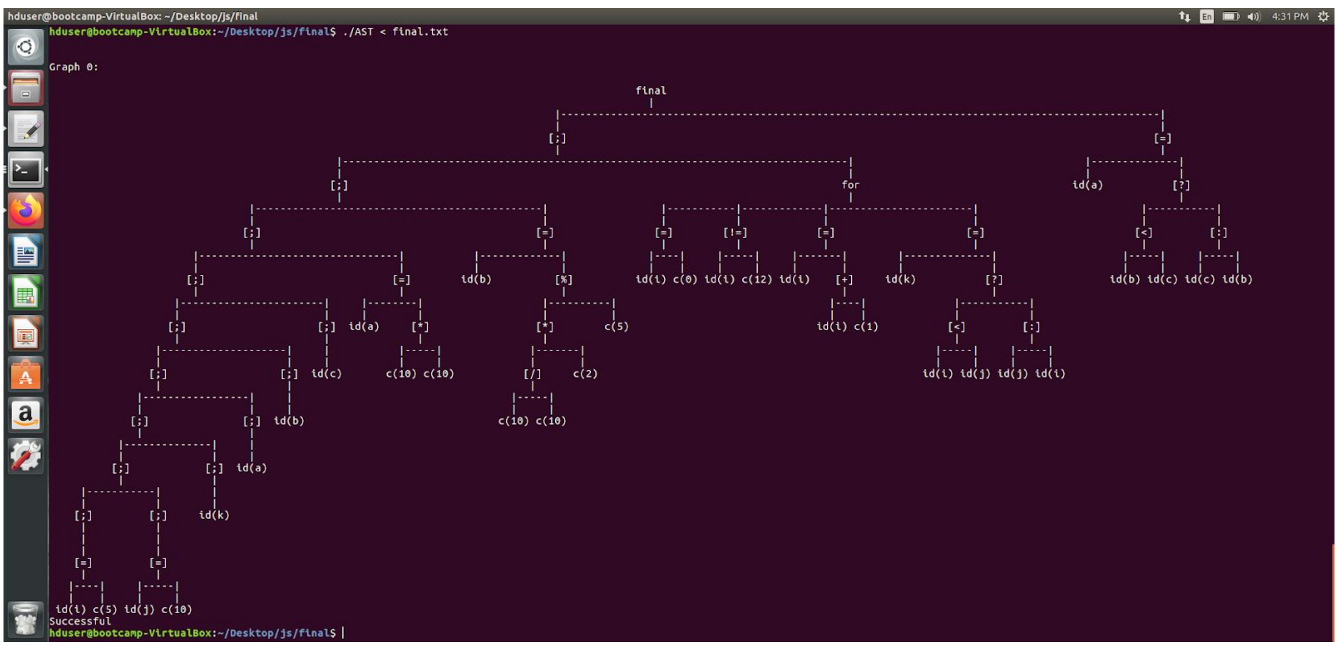
```
hduser@bootcamp-VirtualBox: ~/Desktop/js/final
hduser@bootcamp-VirtualBox:~/Desktop/js/final$ ./ST < final1.txt
function
identifier: final
variable: var
identifier: i
assignment operator: =
number: 5
variable: var
identifier: j
assignment operator: =
number: 10
variable: var
identifier: a
assignment operator: =
number: 20
variable: var
identifier: c
variable: var
identifier: k
assignment operator: =
number: 15
for
identifier: i
assignment operator: =
number: 0
identifier: i
relational operator: !=
number: 12
identifier: i
arithematic operator: ++
identifier: k
assignment operator: =
identifier: i
identifier: j
if
identifier: i
relational operator: ==
number: 0
variable: var
identifier: a
assignment operator: =
number: 20
variable: var
identifier: b
assignment operator: =
number: 30
------------------------------------------------------------

---------------------Symbol Table------------------
---------------------------------------------------
Symbol          Scope           dtype           Value
---------------------------------------------------
```



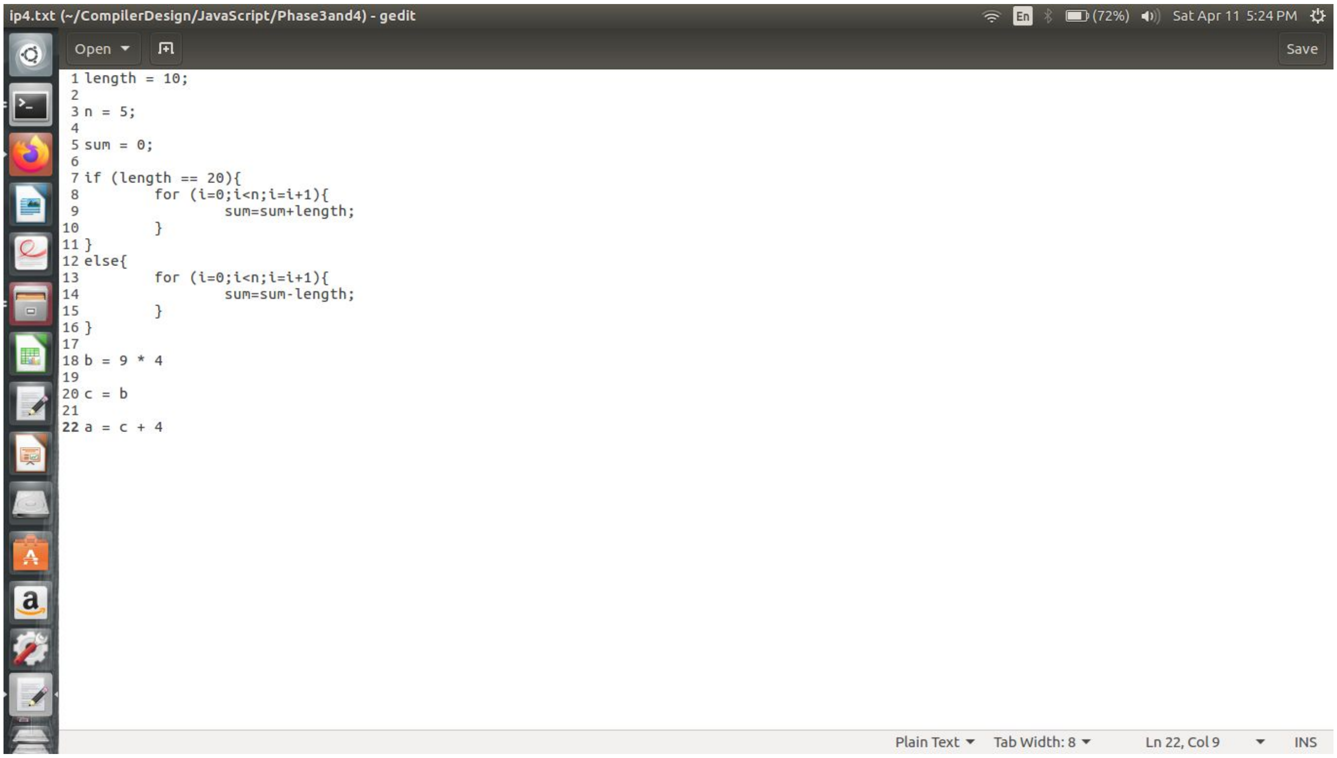
```
hduser@bootcamp-VirtualBox: ~/Desktop/js/final
identifier: j
assignment operator: =
number: 10
variable: var
identifier: a
assignment operator: =
number: 20
variable: var
identifier: c
variable: var
identifier: k
assignment operator: =
number: 15
for
identifier: i
assignment operator: =
number: 0
identifier: i
relational operator: !=
number: 12
identifier: i
arithematic operator: ++
identifier: k
assignment operator: =
identifier: i
identifier: j
if
identifier: i
relational operator: ==
number: 0
variable: var
identifier: a
assignment operator: =
number: 20
variable: var
identifier: b
assignment operator: =
number: 30
------------------------------------------------------------

---------------------Symbol Table------------------
---------------------------------------------------
Symbol          Scope           dtype           Value
---------------------------------------------------
i               1               var             5
j               1               var             10
a               1               var             20
c               1               var             0
k               1               var             0
a               2               var             20
b               0               var             30
hduser@bootcamp-VirtualBox:~/Desktop/js/final$
```

3. AST:

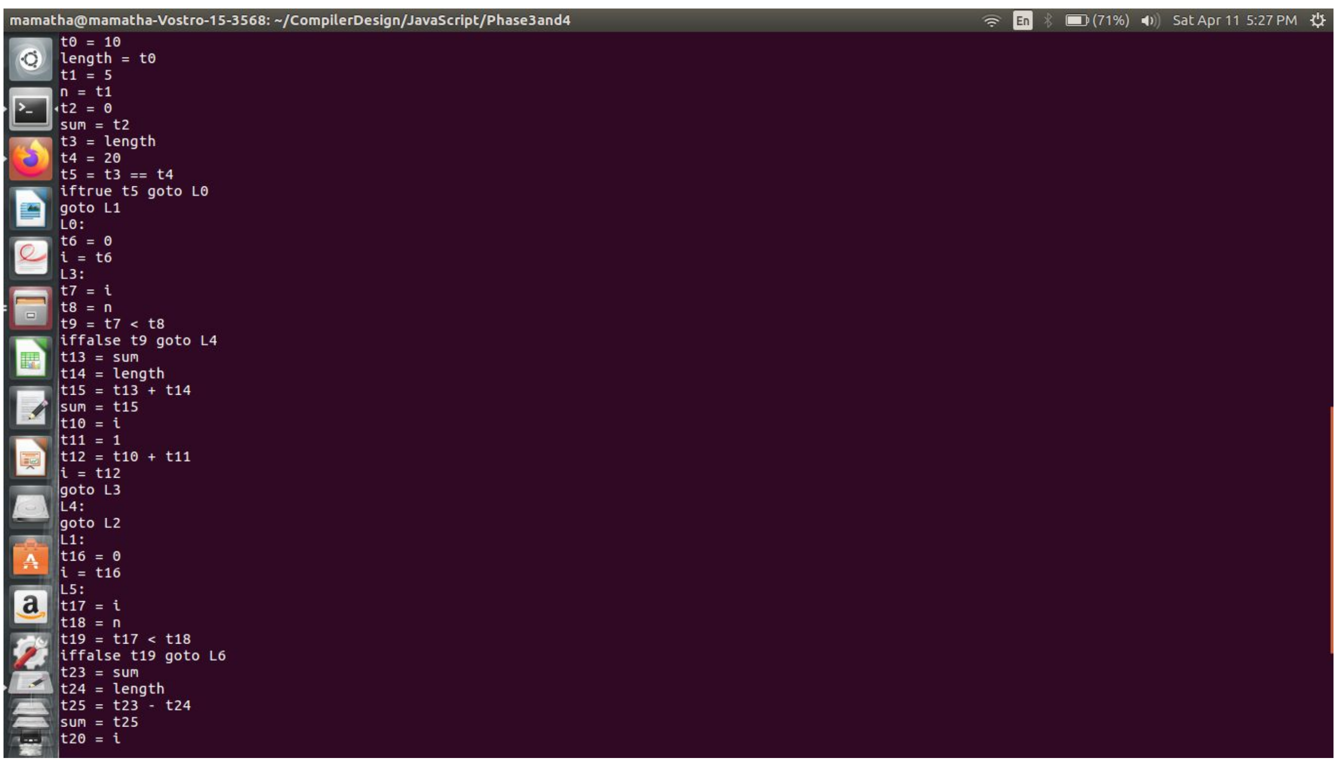

final.txt (~/Desktop/js/final) - gedit
Open
Save
ip1.txt
ip4.txt
final.txt
final1.txt
function final()
{
var i=5;
var j=10;
var k;
var a;
var b;
var c;
a = 10*10;
b = 10/10*2%5;
for (i=0; i!= 12; i++)
{
k = i < j ? j : i;
}
a = b < c ? c : b;
}
Plain Text
Tab Width: 4
Ln 22, Col 5
INS
4:32 PM


hduser@bootcamp-VirtualBox: ~/Desktop/js/final
hduser@bootcamp-VirtualBox:~/Desktop/js/final$ ./AST < final.txt
Graph 0:
final
[;]
[=]
for
id(a)
[?]
Successful
hduser@bootcamp-VirtualBox:~/Desktop/js/final$
4:31 PM

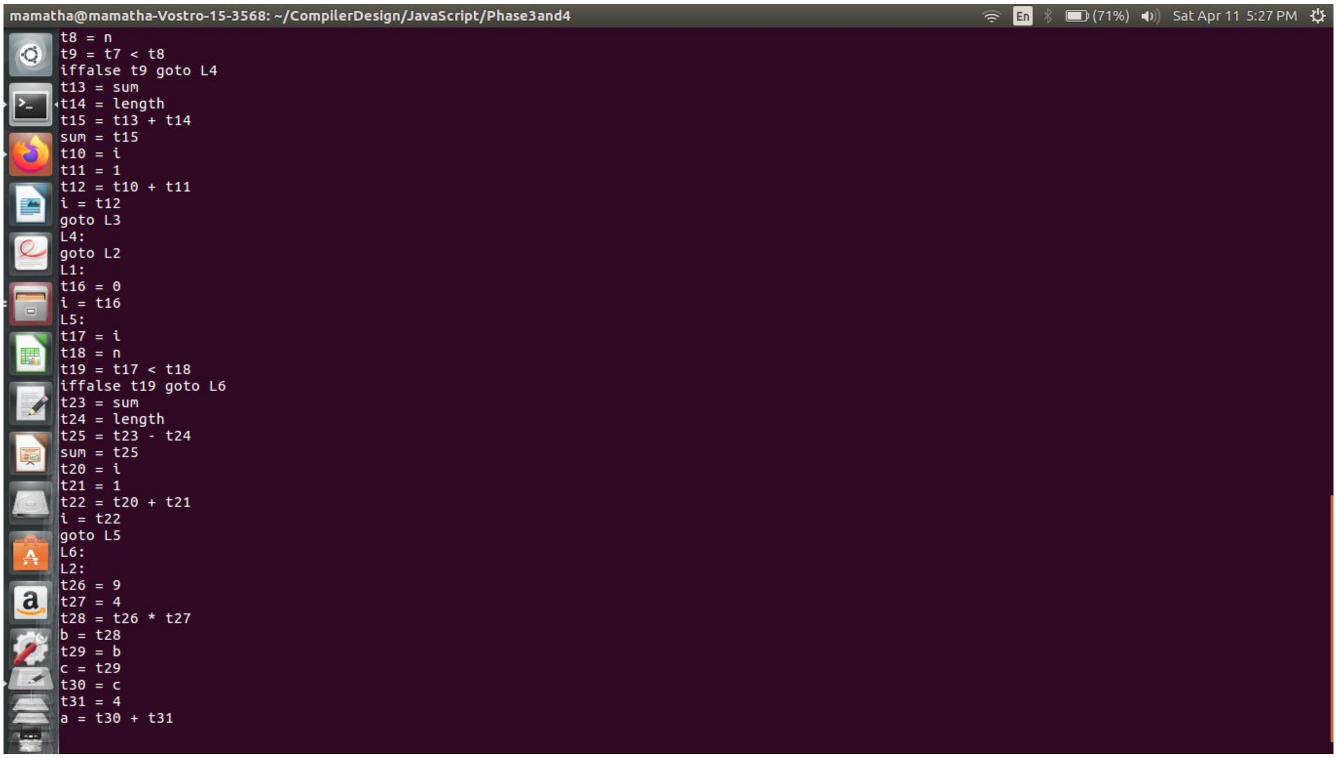
4. ICG:

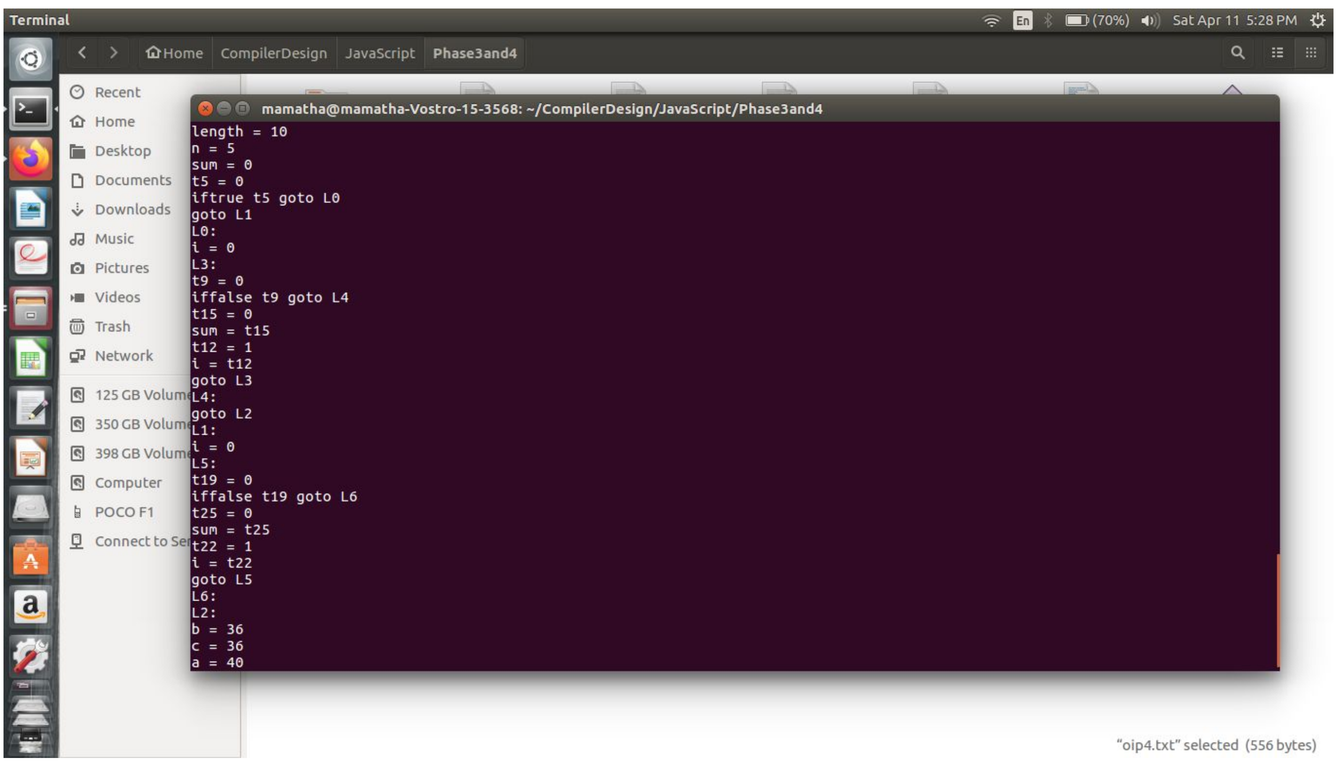



```
length = 10;

n = 5;

sum = 0;

if (length == 20){
        for (i=0;i<n;i=i+1){
                sum=sum+length;
        }
}
else{
        for (i=0;i<n;i=i+1){
                sum=sum-length;
        }
}

b = 9 * 4

c = b

a = c + 4
```




```
t0 = 10
length = t0
t1 = 5
n = t1
t2 = 0
sum = t2
t3 = length
t4 = 20
t5 = t3 == t4
iftrue t5 goto L0
goto L1
L0:
t6 = 0
i = t6
L3:
t7 = i
t8 = n
t9 = t7 < t8
iffalse t9 goto L4
t13 = sum
t14 = length
t15 = t13 + t14
sum = t15
t10 = i
t11 = 1
t12 = t10 + t11
i = t12
goto L3
L4:
goto L2
L1:
t16 = 0
i = t16
L5:
t17 = i
t18 = n
t19 = t17 < t18
iffalse t19 goto L6
t23 = sum
t24 = length
t25 = t23 - t24
sum = t25
t20 = i
```





FUTURE ENHANCEMENTS

Include:

- **Other looping constructs like while, do-while.**
- **Conditional jumps like goto, continue and break.**
- **Conditional statements like switch case.**

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_intermediatecode_generations.html