Report on

"Building a Mini Compiler based on Perl"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:
K. Anjali Kamath

Under the guidance of

Prof. Preet Kanwal
Assistant Professor, CSE
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

1. Introduction

A compiler is a software that converts code written in one programming language into a low level language(Machine/Target language). There are two kinds of compilers: A cross compiler is a compiler that is capable of creating code for a platform other than the one on which the compiler is running. The second type is a source-to-source compiler/transpiler/transcompiler that translates source code written in one programming language into the source code of another language. In our project, we aim to build a mini compiler for the Perl Programming language using the concepts learnt in the compiler design course. We have built this compiler using Lex and Yacc as tools for tokenizing and parsing different phases of the compiler.

There are 4 steps for building the compiler:

1. Lexical Analysis
2. Semantic Analysis
3. Intermediate Code Generation
4. Target Code Generation

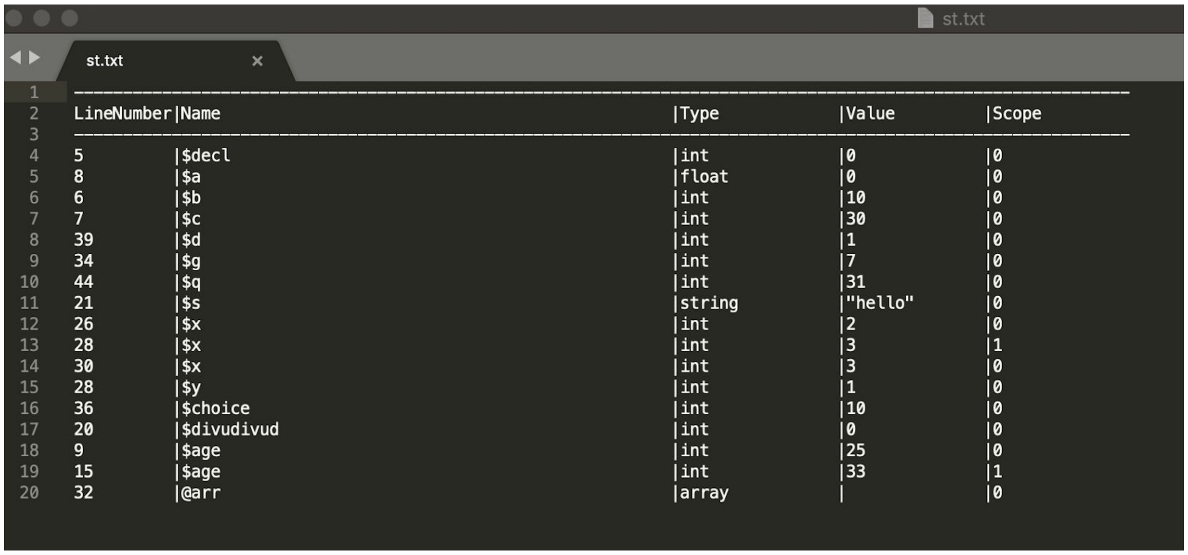
The project has been divided into 5 phases to ease the building of the compiler:
Phase 1: Building of context free grammar
Phase 2: Lexical analysis with error handling, error recovery, scope handling and symbol table generation.
Phase 3: Abstract syntax tree generation and intermediate code generation(3 address code).
Phase 4: Intermediate Code optimization.
Phase 5: Target code generation- Conversion of ICG to assembly code using a hypothetical target machine model.

Sample input: Perl program.
Sample output: Symbol table, Abstract Syntax Tree, 3-address code, optimized 3-address code, Assembly code

1. Input program:

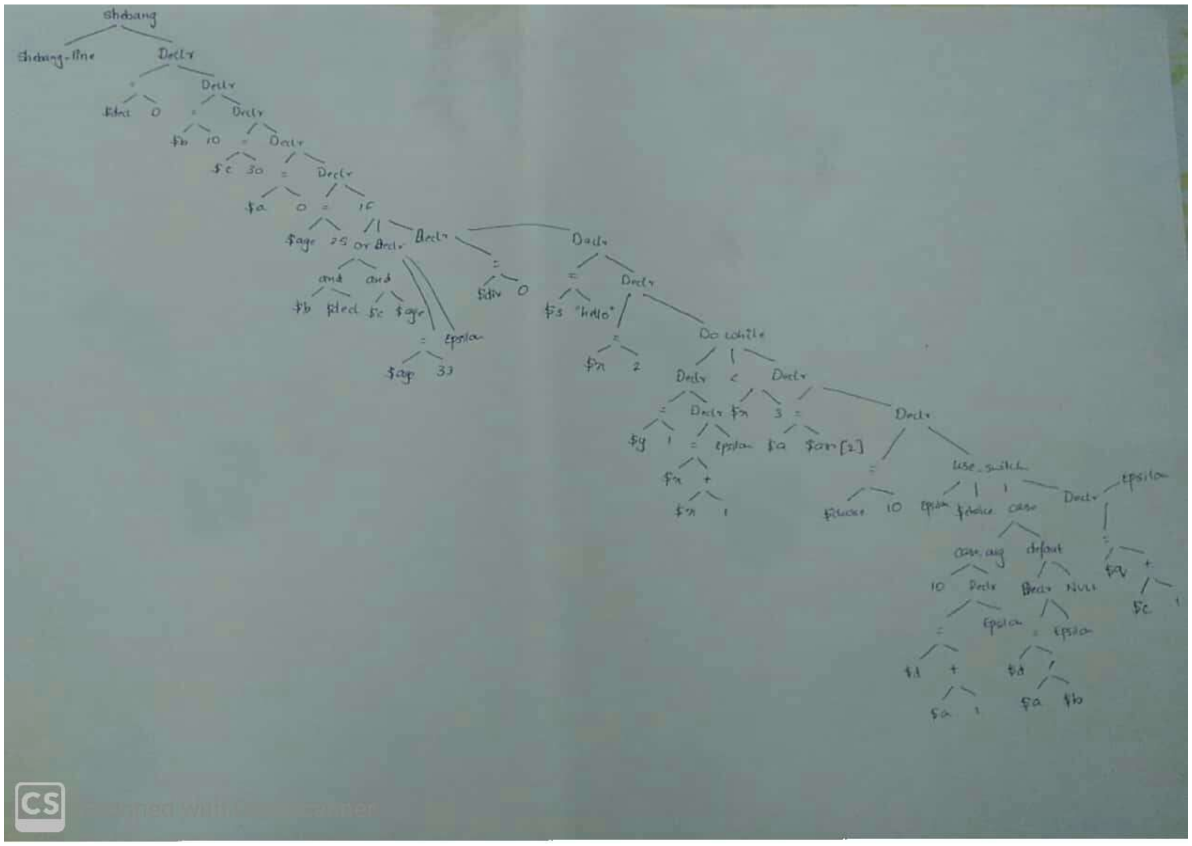
```
#!/usr/bin/perl
#start of input file
$decl = 0;
$b = 10;
$c = 30;
$a = 0;
$age = 25;
#if loop
if($b && $decl || $c && $age) {
  # if condition is true then do the following
  $age = 33;
}
#variable greater than 31 is truncated
$divudivudivudivudivudivudivudivudivu = 0;
$s = "hello";
#syntax error: not properly defined variable
def = 5;
#do while loop
$x = 2;
do{
  $y = 1;
  $x = $x + 1;
} while($x < 3);

@arr = (3, 5, 7, 9);
$a = $arr[2];
$choice = 10;
use Switch;
switch($choice){
  case 10        { $d = $a + 1; }
  else          { $d = $a / $b; }

}

$q = $c++;
```

2. Symbol Table:



```
------------------------------------------------------------------------------------------------------------------
LineNumber|Name                                                  |Type             |Value            |Scope
------------------------------------------------------------------------------------------------------------------
5         |$decl                                                 |int              |0                |0
8         |$a                                                    |float            |0                |0
6         |$b                                                    |int              |10               |0
7         |$c                                                    |int              |30               |0
39        |$d                                                    |int              |1                |0
34        |$g                                                    |int              |7                |0
44        |$q                                                    |int              |31               |0
21        |$s                                                    |string           |"hello"          |0
26        |$x                                                    |int              |2                |0
28        |$x                                                    |int              |3                |1
30        |$x                                                    |int              |3                |0
28        |$y                                                    |int              |1                |0
36        |$choice                                               |int              |10               |0
20        |$divudivud                                            |int              |0                |0
9         |$age                                                  |int              |25               |0
15        |$age                                                  |int              |33               |1
32        |@arr                                                  |array            |                 |0
```

3. Abstract Syntax Tree:



4. Intermediate Code:

```
$arr = [ #3 , #5 , #7 , #9 ]
$decl = #0
$b = #10
$c = #30
$a = #0
$age = #25
t0 = $b && $decl
t1 = $c && $age
t2 = t0 || t1
t3 = not t2
IF t3 GOTO L0
$age = #33
LABEL L0
$divudivud = #0
$s = "hello"
$x = #2
LABEL L1
$y = #1
t4 = $x + #1
$x = t4
t5 = $x < #3
IF t5 GOTO L1
t6 = #2
t7 = #4
t8 = t6 * t7
$a = $arr [ t8 ]
$choice = #10
t9 = $choice
t10 = #10
t11 = #10
t12 = t10 == t11
IF t12 GOTO L2
ELSE GOTO L3
LABEL L2
t13 = $a + #1
$d  =  t13
GOTO L5
LABEL L3
t14 = $a / $b
$d = t14
```

```
LABEL L5
t15 = $c + #1
$c = t15
$q = $c
```

5. Target/Assembly code for intermediate code:

```
        .TEXT
        MOV R2,0
        STR R2,$decl
        MOV R3,10
        STR R3,$b
        MOV R4,30
        STR R4,$c
        MOV R5,0
        STR R5,$a
        MOV R6,25
        STR R6,$age
        AND R7,R3,R2
        AND R8,R4,R6
        OR R9,R7,R8
        R10 = NOT R11
        BNEZ R10 GOTO L0
        MOV R6,33
        STR R6,$age
L0:
        MOV R12,0
        STR R12,$divudivud
        MOV R13,2
        STR R13,$x
L1:
        MOV R14,1
        STR R14,$y
        ADD R5,R13,1
        STR $x,R5
        SUB R3,R13,3
        BLT R3 GOTO L1
        MOV R2,2
        MOV R4,4
        MUL R9,R2,R4
        LOAD R5,ARR[R9]
        MOV R7,10
        STR R7,$choice
        LOAD R8,$choice
        MOV R11,10
        MOV  R10,10
        SUB R6,R11,R10
        BEQ R6 GOTO L2
        BR L3
L2:
        ADD R12,R5,1
        STR $d,R12
        BR L5
L3:
        DIV R14,R5,R3
        STR $d,R14
L5:
        ADD R13,R4,1
        STR $c,R13
        LOAD R2,$c

.DATA
        ARR : .WORD 3 , 5 , 7 , 9
        S : .ASCIZ          "hello"
```

6. Optimized Intermediate Code:

```
$arr = [ #3 , #5 , #7 , #9 ]
$decl = #0
$b = #10
$c = #30
$a = #0
$age = #25
t0 = #0
t1 = #1
t2 = #1
t3 = not t2
IF t3 GOTO L0
$age = #33
LABEL L0
$divudivud = #0
$s = "hello"
$x = #2
$y = #1
LABEL L1
```

```
$x = $x + #1
t4 = $x < #3
IF t4 GOTO L1
t5 = #2
t6 = #4
t7 = #8
$a = $arr[t7]
$choice = #10
t8 = $choice
t9 = #10
t10 = #10
t11 = #1
IF t1 GOTO L2
ELSE GOTO L3
LABEL L2
$d = #1
GOTO L5
LABEL L3
$d = #0
LABEL L5
$c ++
$q = $c
```

7. Target/Assembly Code For optimized intermediate code:

```
.TEXT
        MOV R2,0
        STR R2,$decl
        MOV R3,10
        STR R3,$b
        MOV R4,30
        STR R4,$c
        MOV R5,0
        STR R5,$a
        MOV R6,25
        STR R6,$age
        MOV R7,0
        MOV R8,1
        MOV R9,1
        R10 = NOT R11
        BNEZ R10 GOTO L0
        MOV R6,33
        STR R6,$age
L0:
        MOV R12,0
        STR R12,$divudivud
        MOV R13,2
        STR  R13,$x
        MOV R14,1
        STR R14,$y
L1:
        ADD R13,R13,1
        STR R13,$x
        SUB R2,R13,3
        BLT R2 GOTO L1
        MOV R3,2
        MOV R4,4
        MOV R5,8
        LOAD R5,$arr[t7]
        MOV R7,10
        STR R7,$choice
        LOAD R8,$choice
        MOV R9,10
        MOV R11,10
        MOV R10,1
        R8 GOTO L2
        BR L3
L2:
        MOV R6,1
        STR  R6,$d
        BR L5
L3:
        MOV R6,0
        STR R6,$d
L5:
        ADD R4,R4,1
        STR R4,$c
        LOAD R12,$c

.DATA
        ARR : .WORD 3 , 5 , 7 , 9
        S : .ASCIZ        "hello"
```

2. Architecture of language

Perl language is fundamentally interpreted. The following are the programming constructs considered:

1. Do..while loop
2. Switch..case

Other structures handled are:

1. If loop
2. Unary operators
3. Arithmetic and Logical operators
4. Comments (Single and Multiline)
5. Arrays
6. Strings
7. Concatenation operator
8. Relational Operators

3. Literature Survey

[1]M. S. Bajwa, A. P. Agarwal and N. Gupta, "Code optimization as a tool for testing software," *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, New Delhi, 2016, pp. 961-967.

Need and Requirement of optimization

- The basic necessity of code optimization is to increase its efficiency and to consume less and less time without affecting the quality of code.
- The Project is better in terms of optimization if it's smaller, use less memory, less operations and usage of resources, execute all code quickly.
- Now comes the output phase, without compromising with output the optimization need to produce same output in comparison to the un optimized result.

[2] C. H. Bhatt and H. B. Bhadka, "Extension of Context Free Grammar for Intermediate Code and Peephole Optimization Rule Parsers," *2015 Fifth International Conference on Advanced Computing & Communication Technologies*, Haryana, 2015, pp. 162-164, doi: 10.1109/ACCT.2015.79.

Importance of correct CFG Grammar:

CFG grammar provides helps to generate top-down parsers as well as with required modification it can be also useful for other parser generation specifically and it also requires studying other intermediate code. It is also possible to implement other optimization techniques by specifying rules in rules file and making appropriate change in the grammar accordingly to accommodate them. The parser generated using this grammar can also be enhanced for other parsing mechanism by adding additional non-terminal symbols in the grammar.

4. Context Free Grammar

Some of the tokens handled by our compiler are:

```
%token T_If T_Print T_Do T_While T_Switch T_Else T_Case T_Assignop T_Pluseq
T_Shebang T_Minuseq T_Exp T_String T_True T_False T_Use T_Arr

T_Id T_Int T_Float T_Double T_Inc T_Dec T_Lesseq T_Greteq T_Or T_And T_Not
T_Eq T_Noteq

%left '>' '<' '=' T_Pluseq T_Minuseq

%left '+' '-' '*' '/' '%' ","

%left T_Inc T_Dec
```

Grammar:

```
P              ->    Shebang S
S              ->     Declaration;S  | Assign expr;S
                     | do{S} while(cond);S
                     | us; S; switch(arg) {st} S | print “string”;S |  B;S
                     | if(cond) {S} | ue;S |ArrayDecl;S| ε
ArrayDecl    ->    @id=() | @id=(D)
D              ->    num,D | string,D | num | string
Declaration ->    L
B              ->     $id.$id
L              ->     L,X     | X
X              ->     $id    |       Assignment expr
Assign expr ->    $id=E
Cond           ->    Cond || C  |   C
C              ->    C&&D | D
D              ->    not D | M
M              ->    (cond)| relexp | true | false
relexp ->      relexp relop E|E|id|num
relop          ->     < | > | <= | >= | == | !=
E              ->     E+T | E-T | T | ue
T              ->     T*F | T/F | F
F              ->     N**F | N
N              ->    $id | num|(E)
```

us	->	use Switch;
arg	->	$id | num
st	->	case Y O
Y	->	K {S}
K	->	num | $id | ”character”
O	->	else {S}| st
ue	->	$id++ \|$id--| ++$id\| --$id

5. Design Strategy

a. Symbol table generation

The input is parsed and every time it encounters a token, it gets stored in the symbol table. The symbol table consists of line_number, type of token, value of token and the scope.

b. Abstract Syntax Tree

The tree is formed as the input is parsed. A node in the tree has the following structure:

- Root
- Number of children
- Pointer to the children

A node of the tree can have two or more children. The terminals are stored as the leaves of the tree.Depending on the operator or token, different types of nodes are created.The tree is printed when parsing is complete.

c. Intermediate Code Generation

The 3-address code was generated with temporary variables [T0...Tn] and labels [L0...Ln]. The intermediate values were stored in temporary variables.The conditional statements and loops were reduced to If statements while generating the three address code and labels were used for routing/branching during conditional statements.

d. Code Optimization

1. Constant Propagation

   When an identifier is encountered, we check the symbol table to see if an entry exists. If the entry exists we perform constant propagation, that is ,replace every occurrence of the variable with it

2. Constant Folding

   When an arithmetic expression is encountered, we check to see if all the operands contain digits and are not identifiers. If all the operands are numbers we evaluate the expression.

3. Loop invariant code motion

   Statements or expressions that can be moved outside the loop and it should not affect the semantic of the program. Such statements/expressions were targeted.

e. Error handling

A function called yyerror was created. Panic mode recovery was used to handle the error. When an error occurs, the type of error and lineno and the line is printed and that line is skipped and the parser continues with the next input line.

f. Target code generation

We followed MIPS architecture to generate the assembly code for the Perl language. We first generate the optimized code .This optimized code is then converted to assembly code by parsing the intermediate code. The target code handles operations such as MOV,LOAD,STORE, conditional and unconditional branch as well as all types of operators.

We also implemented an LRUsystem to reuse registers when all the available registers are used up.

6. Implementation Details

- **Symbol table generation**

Every time a variable declaration is done, the parser identifies the token from the pattern described in the lex file and adds it to the symbol table. The structure of symbol table is as follows:

```
typedef struct Node{
char name[100];
char type[100];
struct Node* next;
int lno;
 int scope;
 char value;
}node;
```

The ‘name’ field of the node contains the name of the variable. The array ‘type’ stores the datatype, ‘next’ is a pointer to the next node in the symbol table,’lno’ stores the line number , ‘scope’ contains the scope in which the variable occurs and ‘value’ stores the value assigned to the variable.

```
void insert(const char name[], const char type[], int l, int scope );
node* lookup(const char s[]);
```

The ‘lookup’ function searches the symbol table to check whether an entry for the variable already exists in the same scope. If it does, a pointer to the entry is returned. If not, a new entry is created using the ‘insert’ function.

- **Abstract Syntax Tree**

```
typedef struct AST{
        char lexeme[100];
        int NumChild;
        struct AST **child;
}AST_node;
```

The basic node of the abstract syntax tree has three components. The character array lexeme contains the name of the token or variable, the NumChild field is the number of children that the node has. The node also contains pointers to its children.

The 'YYSTYPE ' was redefined to the following:

```
%union {
        char var_type[100];
        char text[100];
        struct AST *node;
}
```

%union is used to modify the type of yylval.
The var_type and text attributes are used for numbers,identifiers and strings. The 'node' pointer points to the node associated with the particular $$.

```
struct AST* make_if_node(char* root, AST_node* child1, AST_n
ode* child2, AST_node* child3);
struct AST* make_switch_node(char* root, AST_node* child1,
AST_node* child2, AST_node* child3,AST_node* child4);
struct AST * make_node(char*, AST_node*, AST_node*);
struct AST* make_leaf(char* root);
void AST_print(struct AST *t);
```

The above functions were used for the implementation of AST.
The 'make_if_node' function was used to create a node for the if and do_while constructs.Similarly, the 'make_switch_node' function creates a node for the 'switch' construct. 'make_node' is

a more general function which creates nodes for logical, arithmetic,binary,unary operators and other tokens like Declaration, assignment expression etc. The ‘make_leaf’ function is used to create the leaves of the tree i.e the nodes which contain the terminal values. Lastly, the ‘AST_print’ function is used to print the abstract syntax tree once we are done parsing the input.

- **Intermediate Code Generation**
  Stack method has been used to generate the 3-address code. Labels and temporaries were incrementally generated.
  Structure is as follows:

```
struct OPT{
char op[10];
char arg1[10];
char arg2[10];
char result[10];
};

struct OPT QIC[100];
```

The structure ‘OPT’ contains four fields. The array ‘op’ stores the operator, ‘arg1’ contains the first argument or operand in the expression, ‘arg2’ stores the second operand and the ‘result’ array stores the result of the operation. The array of structures QIC is used to represent the intermediate code in the form of a table. Whenever any variable or identifier is encountered, it is pushed into the stack.We write suitable functions to then generate the corresponding intermediate code for every statement in the input file.The output is represented in the quadruple format with ‘operator’, ’arg1’ , ‘arg2’ , ‘result’ as the columns of the table.We also output the intermediate code in a non-tabular form as well for readability.

- **Code Optimization**
  struct OPT OPT[100];

As a continuation of the same file, the above array is used to output the optimized intermediate code in a tabular format.

1) Constant propagation
   To achieve constant propagation,we check if a statement is an assignment expression (where a number is getting assigned).We then loop through the table and replace every occurence of that variable with its value.

2) Loop invariant code
   To remove loop invariant code, we first checked whether after the occurence of a label ,a simple assignment statement occurs.If it does,we pop that line from the stack, shift the stack contents and push the assignment expression after the loop is done.

3) Constant folding
   To achieve constant folding,we check whether the arguments in the table generated are numbers.If they are numbers,we obtain the operation to be performed from the operator column and perform the operation.This result is then stored back in the table.

Target code generation
For generating the assembly code, the optimized ICG is parsed and for each expression/statement the corresponding assembly code is generated. Registers R0 to R14 are used. General operations like LOAD, STORE, ADD , SUB , MOV, MUL,DIV,AND,OR IF B ,Conditional Branch(all branch types) are used to perform the MIPS operations. Array storage and operations (load and store) have also been implemented. LRU(Least Recently Used) method is used to reuse registers.

A dictionary is used to keep track of the available and occupied registers. The input file is loaded and read and each line of the input file is stored in a list. From this list, we extract the temporaries and variables using a regex.
Initially, all registers are initialized as available.Each of the extracted temporaries and variables are assigned registers from the available dictionary and the corresponding register is then marked unavailable.The allocated registers are stored in a separate 'assigned' dictionary.

We implemented the least recently used algorithm for reuse of registers.Another list 'used' which had all the used registers was used for this purpose.When a register is used, it is removed from the list and appended to the end of the list.This way, the most recently used registers are at the back and the registers that can be reused are at the front of the list. We begin reusing registers when the last register, i.e R14 is allocated.Also, we do not allocate registers to arrays unless they are explicitly loaded.This also helps us in not wasting registers.

A translation function is used to convert the optimized intermediate code to assembly code.Based on the length of the line and the operators and symbols used ,we fetch the registers allocated to the temporary/variables if they occur in the line and appropriate assembly instructions are generated. The arrays and strings are stored in the .DATA portion at the end of the file.The rest of the code is stored in the .TEXT portion of the file.

- **Error handling**
  The following function has been used to identify the error and printing it:

```
void yyerror(const char *s) {
 printf("\nError occurred at %d\n ",yylineno);
 fprintf(stderr, "%s\n\n", s);
}
```

When an error occurs,for example a syntax error,the above function is called and it prints the line number that the error has occurred in and the error to the console and continues parsing.

- **Steps to execute**
  For each phase following needs to be executed on the terminal:
  ```
  lex lexer.l
  yacc -d parser.y
  gcc y.tab.h
  ./a.out input.pl
  ```
  To generate assembly code:
  ```
  python3 assembly.py
  ```

7. Result and possible shortcomings

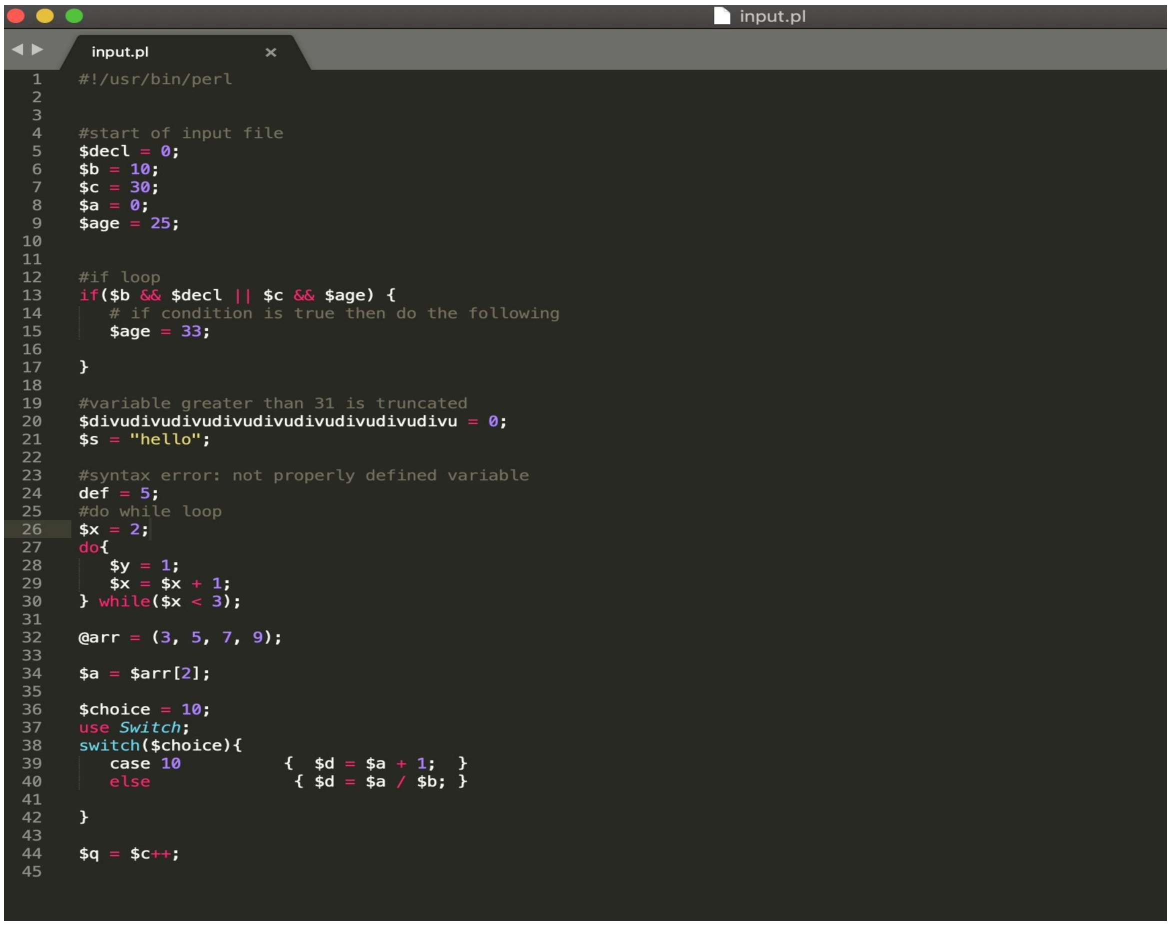
The end result of the project is the development of a mini compiler which replicates an actual compiler to a great extent. The compiler built in this project was for perl programming language. Similarly, the above methodologies can be used for any programming language and a similar compiler can be built for different constructs. Basic errors like syntax errors and division by 0 are displayed when the compiler encounters them.Also,when the length of a variable name is more than 31 characters,it is truncated.

A Perl program when given as input to the compiler gives us a symbol table showing lexical analysis, an abstract syntax tree, 3-address code and finally the assembly code. Thus,we get a low-level language from a high-level language.

The possible shortcomings would be the complexity of the compiler and the performance of the compiler.

8. Snapshots

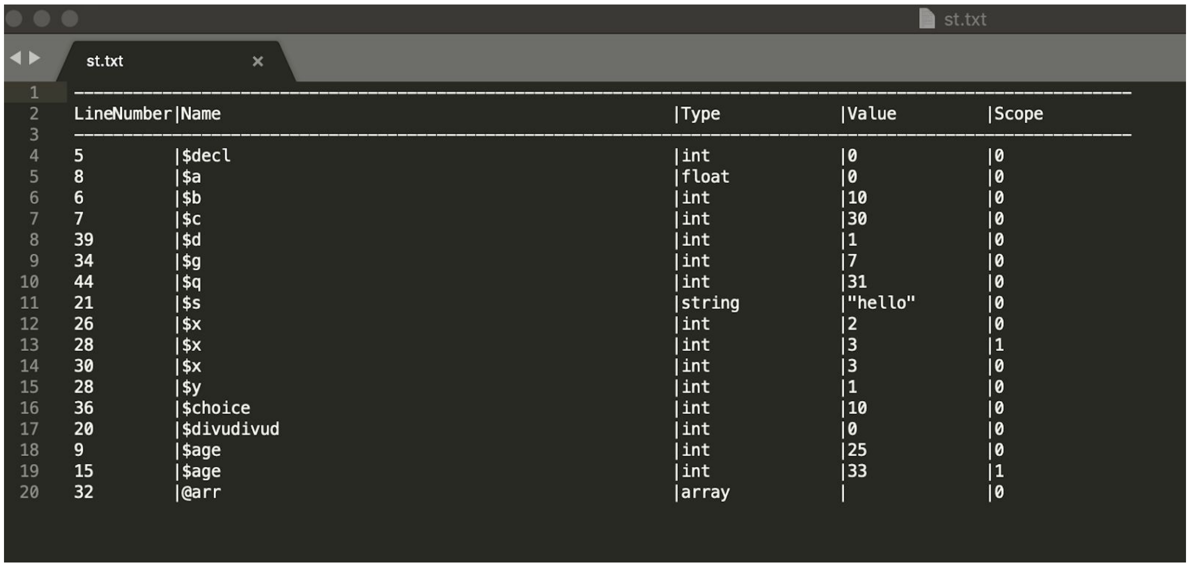
Input file:



input.pl

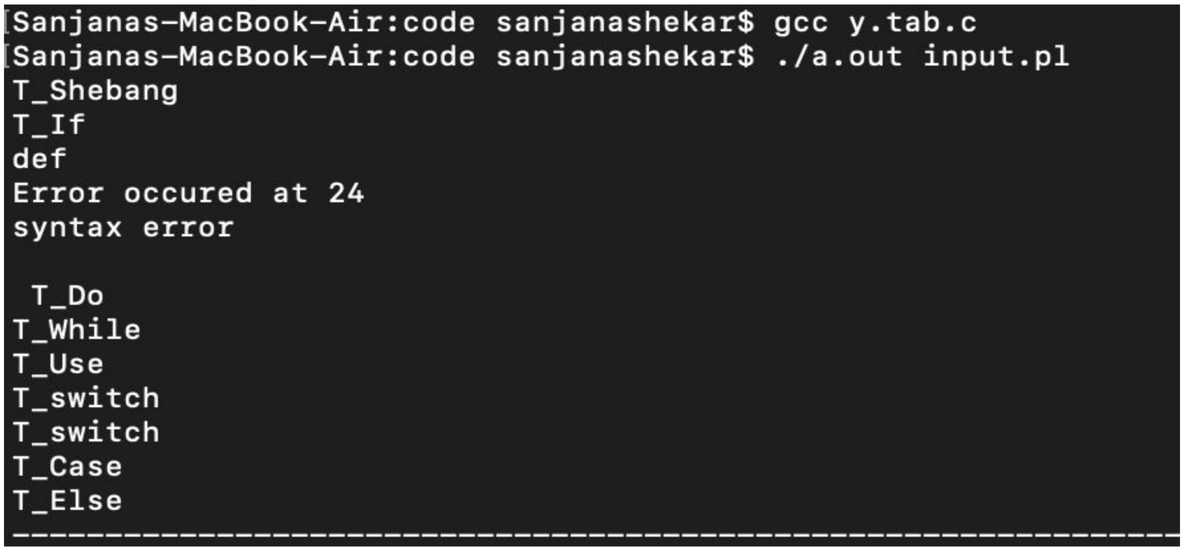
```
#!/usr/bin/perl


#start of input file
$decl = 0;
$b = 10;
$c = 30;
$a = 0;
$age = 25;


#if loop
if($b && $decl || $c && $age) {
    # if condition is true then do the following
    $age = 33;

}

#variable greater than 31 is truncated
$divudivudivudivudivudivudivudivudivu = 0;
$s = "hello";

#syntax error: not properly defined variable
def = 5;
#do while loop
$x = 2;
do{
    $y = 1;
    $x = $x + 1;
} while($x < 3);

@arr = (3, 5, 7, 9);

$a = $arr[2];

$choice = 10;
use Switch;
switch($choice){
    case 10             {  $d = $a + 1;  }
    else                 { $d = $a / $b; }

}

$q = $c++;

```

Symbol Table:



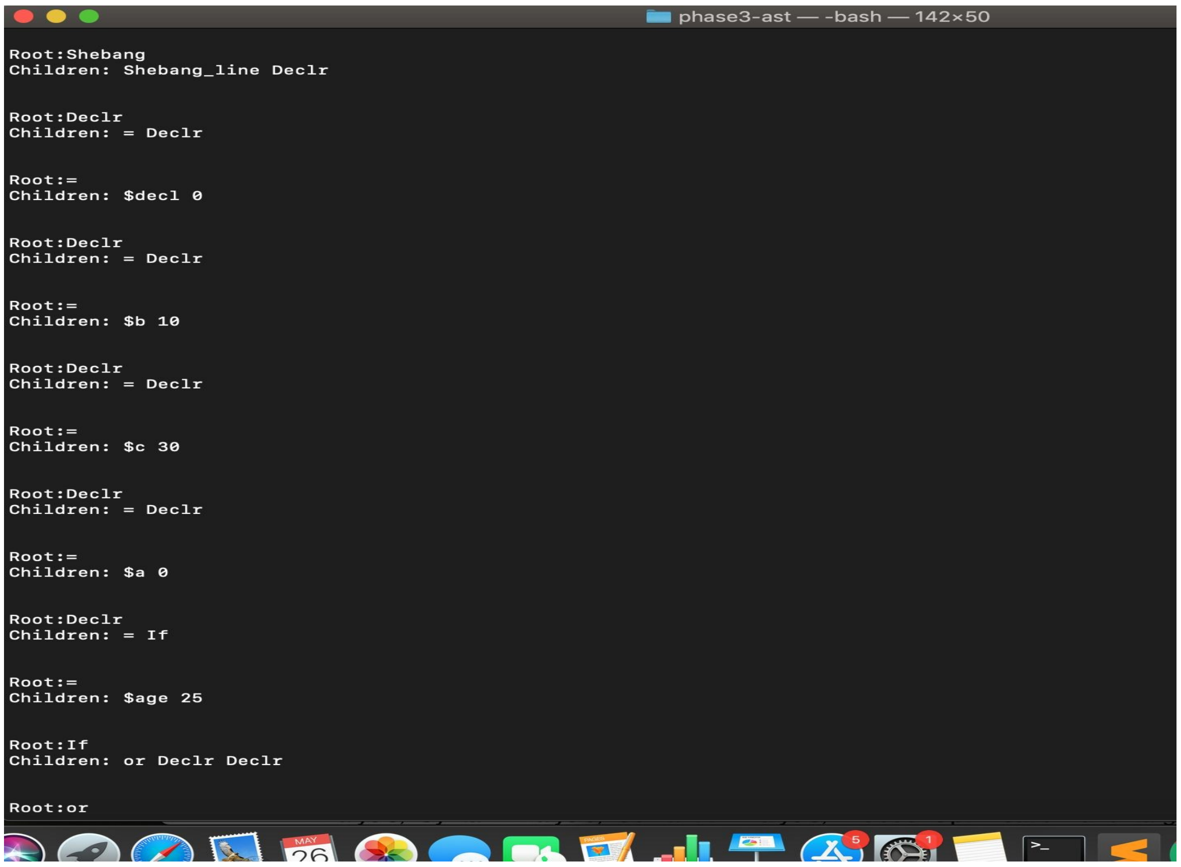
```
st.txt
--------------------------------------------------------------------------------------------------------------
LineNumber|Name                                                    |Type            |Value          |Scope
--------------------------------------------------------------------------------------------------------------
5         |$decl                                                   |int             |0              |0
8         |$a                                                      |float           |0              |0
6         |$b                                                      |int             |10             |0
7         |$c                                                      |int             |30             |0
39        |$d                                                      |int             |1              |0
34        |$g                                                      |int             |7              |0
44        |$q                                                      |int             |31             |0
21        |$s                                                      |string          |"hello"        |0
26        |$x                                                      |int             |2              |0
28        |$x                                                      |int             |3              |1
30        |$x                                                      |int             |3              |0
28        |$y                                                      |int             |1              |0
36        |$choice                                                 |int             |10             |0
20        |$divudivud                                              |int             |0              |0
9         |$age                                                    |int             |25             |0
15        |$age                                                    |int             |33             |1
32        |@arr                                                    |array           |               |0
```

Error handling:



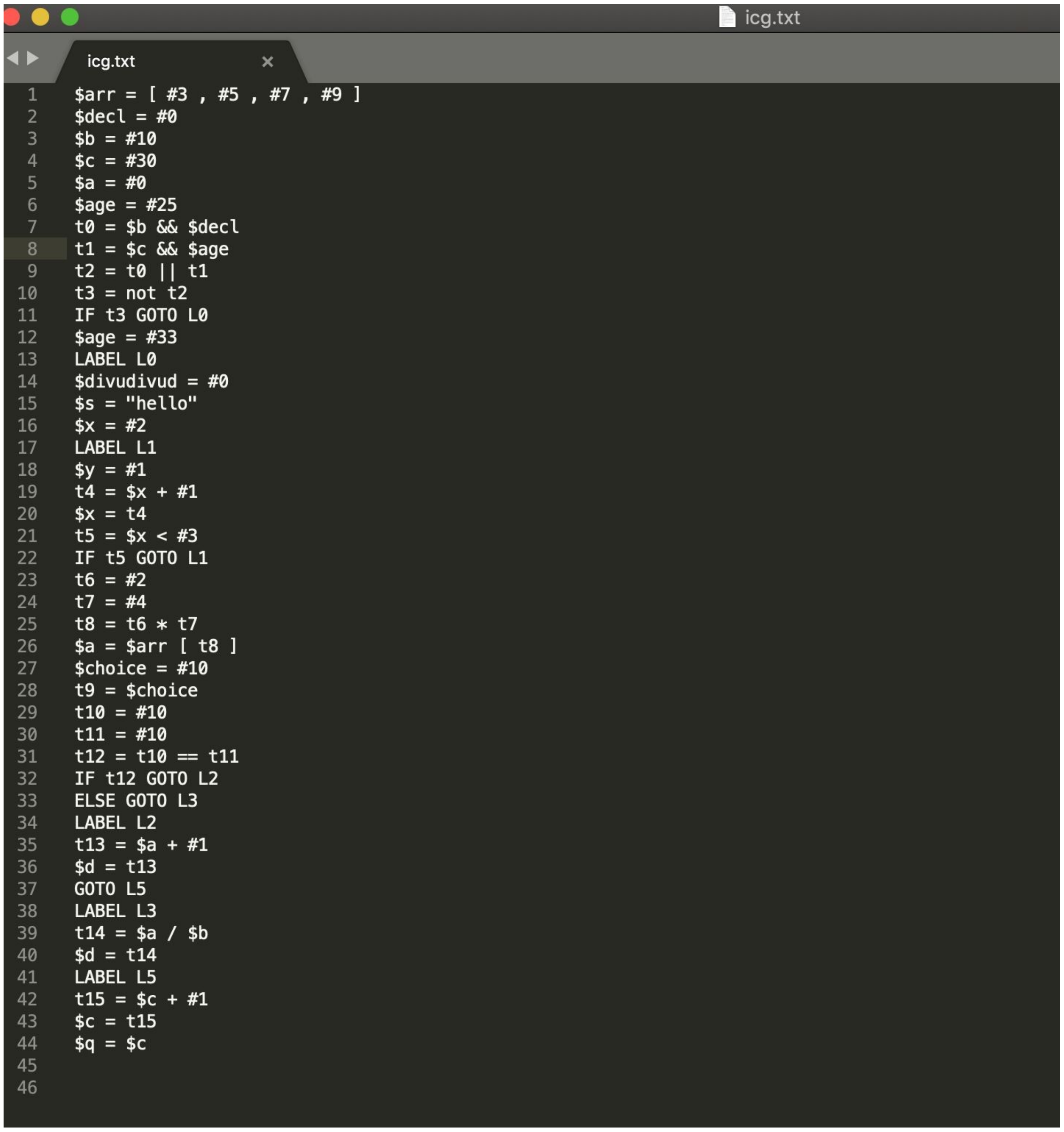
```
Sanjanas-MacBook-Air:code sanjanashekar$ gcc y.tab.c
Sanjanas-MacBook-Air:code sanjanashekar$ ./a.out input.pl
T_Shebang
T_If
def
Error occured at 24
syntax error

 T_Do
T_While
T_Use
T_switch
T_switch
T_Case
T_Else
--------------------------------------------------------------
```

Abstract Syntax Tree:



```
Root:Shebang
Children: Shebang_line Declr

Root:Declr
Children: = Declr

Root:=
Children: $decl 0

Root:Declr
Children: = Declr

Root:=
Children: $b 10

Root:Declr
Children: = Declr

Root:=
Children: $c 30

Root:Declr
Children: = Declr

Root:=
Children: $a 0

Root:Declr
Children: = If

Root:=
Children: $age 25

Root:If
Children: or Declr Declr

Root:or
```

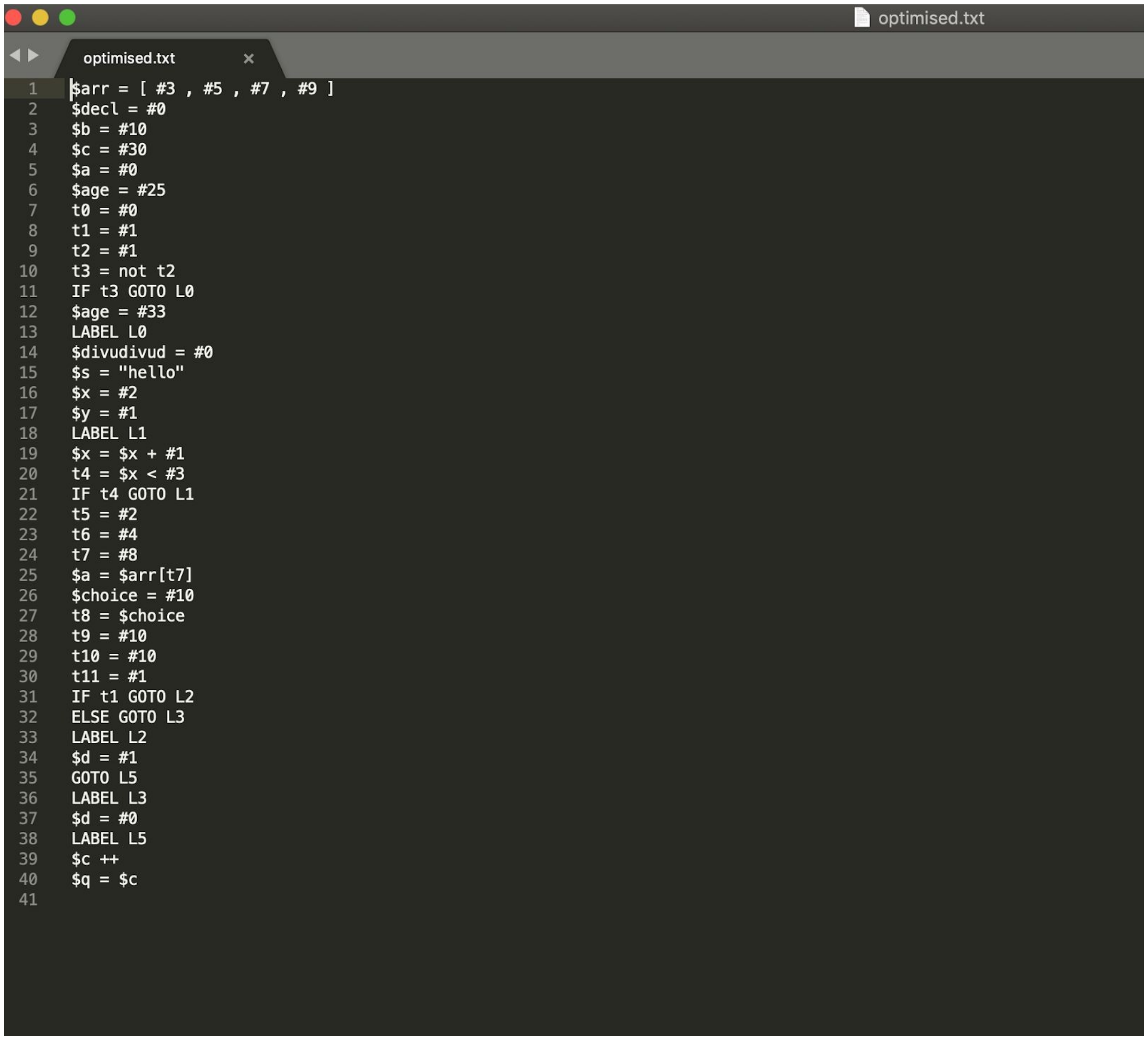
Intermediate Code:



icg.txt

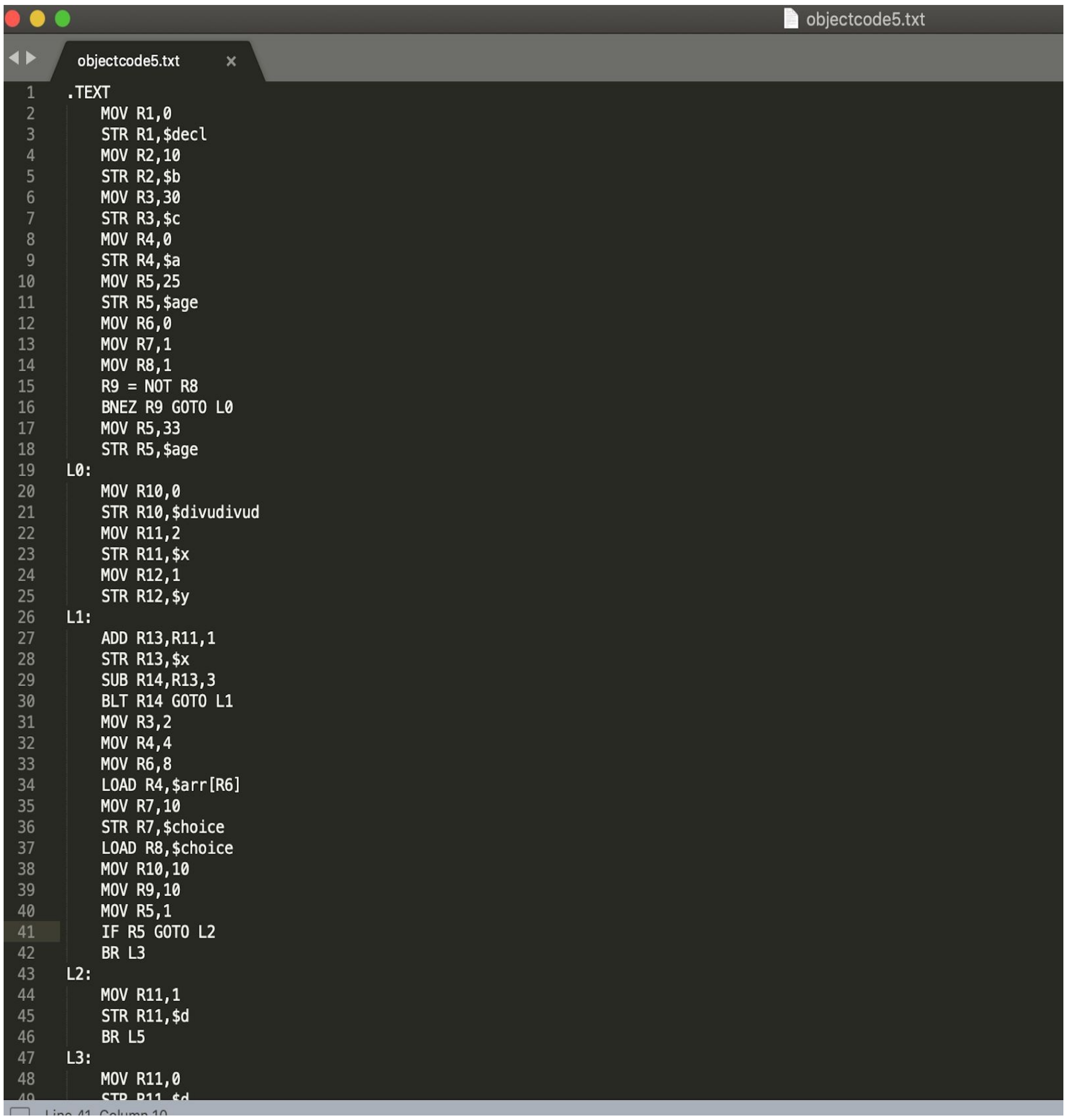
```
$arr = [ #3 , #5 , #7 , #9 ]
$decl = #0
$b = #10
$c = #30
$a = #0
$age = #25
t0 = $b && $decl
t1 = $c && $age
t2 = t0 || t1
t3 = not t2
IF t3 GOTO L0
$age = #33
LABEL L0
$divudivud = #0
$s = "hello"
$x = #2
LABEL L1
$y = #1
t4 = $x + #1
$x = t4
t5 = $x < #3
IF t5 GOTO L1
t6 = #2
t7 = #4
t8 = t6 * t7
$a = $arr [ t8 ]
$choice = #10
t9 = $choice
t10 = #10
t11 = #10
t12 = t10 == t11
IF t12 GOTO L2
ELSE GOTO L3
LABEL L2
t13 = $a + #1
$d = t13
GOTO L5
LABEL L3
t14 = $a / $b
$d = t14
LABEL L5
t15 = $c + #1
$c = t15
$q = $c
```

Optimised ICG:



```
$arr = [ #3 , #5 , #7 , #9 ]
$decl = #0
$b = #10
$c = #30
$a = #0
$age = #25
t0 = #0
t1 = #1
t2 = #1
t3 = not t2
IF t3 GOTO L0
$age = #33
LABEL L0
$divudivud = #0
$s = "hello"
$x = #2
$y = #1
LABEL L1
$x = $x + #1
t4 = $x < #3
IF t4 GOTO L1
t5 = #2
t6 = #4
t7 = #8
$a = $arr[t7]
$choice = #10
t8 = $choice
t9 = #10
t10 = #10
t11 = #1
IF t1 GOTO L2
ELSE GOTO L3
LABEL L2
$d = #1
GOTO L5
LABEL L3
$d = #0
LABEL L5
$c ++
$q = $c
```

Assembly Code for Optimised ICG:



```
.TEXT
    MOV R1,0
    STR R1,$decl
    MOV R2,10
    STR R2,$b
    MOV R3,30
    STR R3,$c
    MOV R4,0
    STR R4,$a
    MOV R5,25
    STR R5,$age
    MOV R6,0
    MOV R7,1
    MOV R8,1
    R9 = NOT R8
    BNEZ R9 GOTO L0
    MOV R5,33
    STR R5,$age
L0:
    MOV R10,0
    STR R10,$divudivud
    MOV R11,2
    STR R11,$x
    MOV R12,1
    STR R12,$y
L1:
    ADD R13,R11,1
    STR R13,$x
    SUB R14,R13,3
    BLT R14 GOTO L1
    MOV R3,2
    MOV R4,4
    MOV R6,8
    LOAD R4,$arr[R6]
    MOV R7,10
    STR R7,$choice
    LOAD R8,$choice
    MOV R10,10
    MOV R9,10
    MOV R5,1
    IF R5 GOTO L2
    BR L3
L2:
    MOV R11,1
    STR R11,$d
    BR L5
L3:
    MOV R11,0
    STR R11,$d
```

9. Conclusion

This report has discussed the building of a compiler for Perl programming language. The sample input is a Perl program and through the above mentioned methodologies and techniques we attempt to reach a replication of what a Perl compiler produces.

As a part of each stage, a part of the compiler was built (Symbol Table, Abstract Syntax Tree, Intermediate Code, Assembly code). Each of these components are required to compile code successfully.

The compiler building was achieved by breaking it down into the phases:

Lexical Analysis
Semantic Analysis
Intermediate Code Generation
Target Code Generation

By referring to the topics taught during the course of compiler design, we were able to build this compiler and it has helped us get a very good hands-on experience for the same. This project has helped us improve knowledge related to compiler design and has greatly benefitted us to improve our skills in computer science related fields.

10. Further Enhancements

Future enhancements to the project would be:

1. Easy to use UI to help the user get a better picture of the execution.
2. Much better optimization of the code.
3. Add a few more constructs like for loop and while loop.
4. Print the syntax tree in a better way.
5. More detailed error messages.
6. Handling nested loops.

11. Acknowledgement

We would like to thank the Department of Computer Science and Engineering for giving us this opportunity to build a compiler for Perl for our 6th semester Compiler Design course.
We would like to thank Mrs. Preet Kanwal for her constant guidance and support during the course of completion of the project. We would also like to express our gratitude to Dr. Shylaja S and the staff of the computer science department for their support.

12. References/Bibliography

[1] https://www.geeksforgeeks.org/compiler-design-detection-of-a-loop-in-three-address-code/?ref=rp

[2] https://www.geeksforgeeks.org/introduction-to-yacc/

[3] https://www.slideshare.net/TahaM21/lex-yacc-77849608

[4]https://www.tutorialspoint.com/compiler_design/compiler_design_code_optimization.htm

[5]https://www.geeksforgeeks.org/introduction-of-lexical-analysis/

[6]https://www.geeksforgeeks.org/symbol-table-compiler/

[7]https://cs.nyu.edu/courses/spring11/G22.2130-001/lecture9.pdf

[8]https://www.sciencedirect.com/topics/computer-science/assembly-code