Report on

“Title of the project”

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Hari Om	**PES1201700250**
Abhishek Banerji	**PES1201700745**
Divya N C	**PES1201802407**

Under the guidance of

Madhura V
Assistant Professor
PES University, Bengaluru

January – May 2020

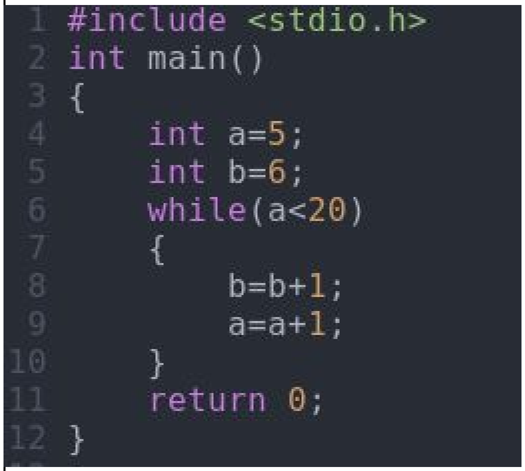
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

TABLE OF CONTENTS

Introduction

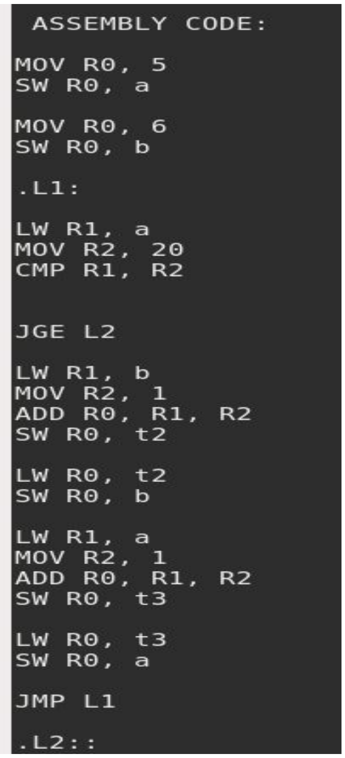
A compiler is a computer program that transforms source code written in a programming language to another computer language. We wrote a mini-compiler which compiles C source code to machine code. The compiler itself is written in Java, using Jflex and Byaccj as tools for parsing the source code.

Sample Input:



```
#include <stdio.h>
int main()
{
    int a=5;
    int b=6;
    while(a<20)
    {
        b=b+1;
        a=a+1;
    }
    return 0;
}
```

Sample Output:



```
 ASSEMBLY CODE:

MOV R0, 5
SW R0, a

MOV R0, 6
SW R0, b

.L1:

LW R1, a
MOV R2, 20
CMP R1, R2


JGE L2

LW R1, b
MOV R2, 1
ADD R0, R1, R2
SW R0, t2

LW R0, t2
SW R0, b

LW R1, a
MOV R2, 1
ADD R0, R1, R2
SW R0, t3

LW R0, t3
SW R0, a

JMP L1

.L2::
```

Architecture Of Language

This mini-compiler supports IF-ELSE, WHILE, DO-WHILE and FOR clauses. It has proper syntax and semantics handling, for example :

- It supports the basic data types: INT, FLOAT, VOID.
- It supports basic unary operators like: ‘~ ’ ‘ | ’ ‘ & ‘ ^ ’ .
- It supports binary operators like: ‘+’ ‘*’ ‘-’ ‘/’ ‘%’ ‘==’ ‘<=’ etc.
- It supports the assignment operator ‘=’
- It has scope handling.
- It checks for undeclared variables.
- It checks for redeclaration of variables.
- Checks for ‘ ; ’ at required places.
- Checks the type of the variable.
- Checks if the return type of a function matches with the datatype of variable/constant being returned.
- Checks if the function call matches its prototype.
- Checks if the index of an array is a positive integer or not.

Literature Survey

- For Grammar: https://www.lysator.liu.se/c/ANSI-C-grammar-y.html
- Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam,Ravi Sethi, Jeffrey D. Ullman
- JFlex: https://jflex.de/manual.html
- ByaccJ: http://byaccj.sourceforge.net/
- ICG: https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/
- Code Generation: https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lects/final.codegen.pdf

Context Free Grammar

```
"begin : external_declaration",
"begin : begin external_declaration",
"begin : Define begin",
"begin : error",
"primary_expression : IDENTIFIER",
"primary_expression : CONSTANT",
"primary_expression : STRING_LITERAL",
"primary_expression : '(' expression ')'",
"Define : DEFINE",
"postfix_expression : primary_expression",
"postfix_expression : postfix_expression '[' expression ']'",
"postfix_expression : postfix_expression '(' ')'",
"postfix_expression : postfix_expression '(' argument_expression_list ')'",
"postfix_expression : postfix_expression '.' IDENTIFIER",
"postfix_expression : postfix_expression PTR_OP IDENTIFIER",
"postfix_expression : postfix_expression INC_OP",
"postfix_expression : postfix_expression DEC_OP",
"argument_expression_list : assignment_expression",
"argument_expression_list : argument_expression_list ',' assignment_expression",
"unary_expression : postfix_expression",
"unary_expression : INC_OP unary_expression",
"unary_expression : DEC_OP unary_expression",
"unary_expression : unary_operator cast_expression",
"unary_expression : SIZEOF unary_expression",
"unary_expression : SIZEOF '(' type_name ')'",
"unary_operator : '&'",
"unary_operator : '*'",
"unary_operator : '+'",
"unary_operator : '-'",
"unary_operator : '~'",
"unary_operator : '!'",
"cast_expression : unary_expression",
"cast_expression : '(' type_name ')' cast_expression",
"multiplicative_expression : cast_expression",
"multiplicative_expression : multiplicative_expression '*' cast_expression",
"multiplicative_expression : multiplicative_expression '/' cast_expression",
"multiplicative_expression : multiplicative_expression '%' cast_expression",
"additive_expression : multiplicative_expression",
"additive_expression : additive_expression '+' multiplicative_expression",
"additive_expression : additive_expression '-' multiplicative_expression",
"shift_expression : additive_expression",
"shift_expression : shift_expression LEFT_OP additive_expression",
"shift_expression : shift_expression RIGHT_OP additive_expression",
"relational_expression : shift_expression",
"relational_expression : relational_expression '<' shift_expression",
"relational_expression : relational_expression '>' shift_expression",
"relational_expression : relational_expression LE_OP shift_expression",
"relational_expression : relational_expression GE_OP shift_expression",
"equality_expression : relational_expression",
"equality_expression : equality_expression EQ_OP relational_expression",
"equality_expression : equality_expression NE_OP relational_expression",
"and_expression : equality_expression",
"and_expression : and_expression '&' equality_expression",
"exclusive_or_expression : and_expression",
"exclusive_or_expression : exclusive_or_expression '^' and_expression",
"inclusive_or_expression : exclusive_or_expression",
"inclusive_or_expression : inclusive_or_expression '|' exclusive_or_expression",
"logical_and_expression : inclusive_or_expression",
"logical_and_expression : logical_and_expression AND_OP inclusive_or_expression",
"logical_or_expression : logical_and_expression",
"logical_or_expression : logical_or_expression OR_OP logical_and_expression",
"conditional_expression : logical_or_expression",
"conditional_expression : logical_or_expression '?' expression ':' conditional_expression",
"assignment_expression : conditional_expression",
"assignment_expression : unary_expression assignment_operator assignment_expression",
"assignment_operator : '='",
"assignment_operator : MUL_ASSIGN",
"assignment_operator : DIV_ASSIGN",
"assignment_operator : MOD_ASSIGN",
"assignment_operator : ADD_ASSIGN",
"assignment_operator : SUB_ASSIGN",
"assignment_operator : LEFT_ASSIGN",
"assignment_operator : RIGHT_ASSIGN",
"assignment_operator : AND_ASSIGN",
"assignment_operator : XOR_ASSIGN",
"assignment_operator : OR_ASSIGN",
"expression : assignment_expression",
"expression : expression ',' assignment_expression",
"constant_expression : conditional_expression",
"declaration : declaration_specifiers ';'",
"declaration : declaration_specifiers init_declarator_list ';'",
"declaration_specifiers : storage_class_specifier",
"declaration_specifiers : storage_class_specifier declaration_specifiers",
"declaration_specifiers : type_specifier",
"declaration_specifiers : type_specifier declaration_specifiers",
"init_declarator_list : init_declarator",
"init_declarator_list : init_declarator_list ',' init_declarator",
"init_declarator : declarator",
"init_declarator : declarator '=' initializer",
"storage_class_specifier : TYPEDEF",
"storage_class_specifier : EXTERN",
"storage_class_specifier : STATIC",
"storage_class_specifier : AUTO",
"storage_class_specifier : REGISTER",
"type_specifier : VOID",
"type_specifier : CHAR",
"type_specifier : SHORT",
"type_specifier : INT",
"type_specifier : LONG",
"type_specifier : FLOAT",
"type_specifier : DOUBLE",
"type_specifier : SIGNED",
"type_specifier : UNSIGNED",
"type_specifier : struct_or_union_specifier",
"specifier_qualifier_list : type_specifier specifier_qualifier_list",
"specifier_qualifier_list : type_specifier",
"specifier_qualifier_list : CONST specifier_qualifier_list",
"specifier_qualifier_list : CONST",
"struct_or_union_specifier : struct_or_union IDENTIFIER '{' struct_declaration_list '}' ';'",
"struct_or_union_specifier : struct_or_union '{' struct_declaration_list '}' ';'",
"struct_or_union_specifier : struct_or_union IDENTIFIER ';'",
"struct_or_union : STRUCT",
"struct_or_union : UNION",
"struct_declaration_list : struct_declaration",
"struct_declaration_list : struct_declaration_list struct_declaration",
"struct_declaration : specifier_qualifier_list struct_declarator_list ';'",
"struct_declarator_list : declarator",
"struct_declarator_list : struct_declarator_list ',' declarator",
"declarator : pointer direct_declarator",
"declarator : direct_declarator",
"direct_declarator : IDENTIFIER",
"direct_declarator : '(' declarator ')'",
"direct_declarator : direct_declarator '[' constant_expression ']'",
"direct_declarator : direct_declarator '[' ']'",
"direct_declarator : direct_declarator '(' parameter_list ')'",
"direct_declarator : direct_declarator '(' identifier_list ')'",
"direct_declarator : direct_declarator '(' ')'",
"pointer : '*'",
"pointer : '*' pointer",
"parameter_list : parameter_declaration",
"parameter_list : parameter_list ',' parameter_declaration",
"parameter_declaration : declaration_specifiers declarator",
"parameter_declaration : declaration_specifiers",
"identifier_list : IDENTIFIER",
"identifier_list : identifier_list ',' IDENTIFIER",
"type_name : specifier_qualifier_list",
"type_name : specifier_qualifier_list declarator",
"initializer : assignment_expression",
"initializer : '{' initializer_list '}'",
"initializer : '{' initializer_list ',' '}'",
"initializer_list : initializer",
"initializer_list : initializer_list ',' initializer",
"statement : compound_statement",
"statement : expression_statement",
"statement : selection_statement",
"statement : iteration_statement",
"statement : jump_statement",
"compound_statement : '{' '}'",
"compound_statement : '{' statement_list '}'",
"compound_statement : '{' declaration_list '}'",
"compound_statement : '{' declaration_list statement_list '}'",
"declaration_list : declaration",
"declaration_list : declaration_list declaration",
"statement_list : statement",
"statement_list : statement_list statement",
"expression_statement : ';'",
"expression_statement : expression ';'",
"selection_statement : IF '(' expression ')' statement",
"selection_statement : IF '(' expression ')' statement ELSE statement",
"iteration_statement : WHILE '(' expression ')' statement",
"iteration_statement : FOR '(' expression_statement expression_statement ')' statement",
"iteration_statement : FOR '(' expression_statement expression_statement expression ')'
statement",
"jump_statement : CONTINUE ';'",
"jump_statement : BREAK ';'",
"jump_statement : RETURN ';'",
"jump_statement : RETURN expression ';'",
"external_declaration : function_definition",
"external_declaration : declaration",
"function_definition : declaration_specifiers declarator declaration_list
compound_statement",
"function_definition : declaration_specifiers declarator compound_statement",
"function_definition : declarator declaration_list compound_statement",
"function_definition : declarator compound_statement",
```

Design Strategy

Symbol Table Creation:

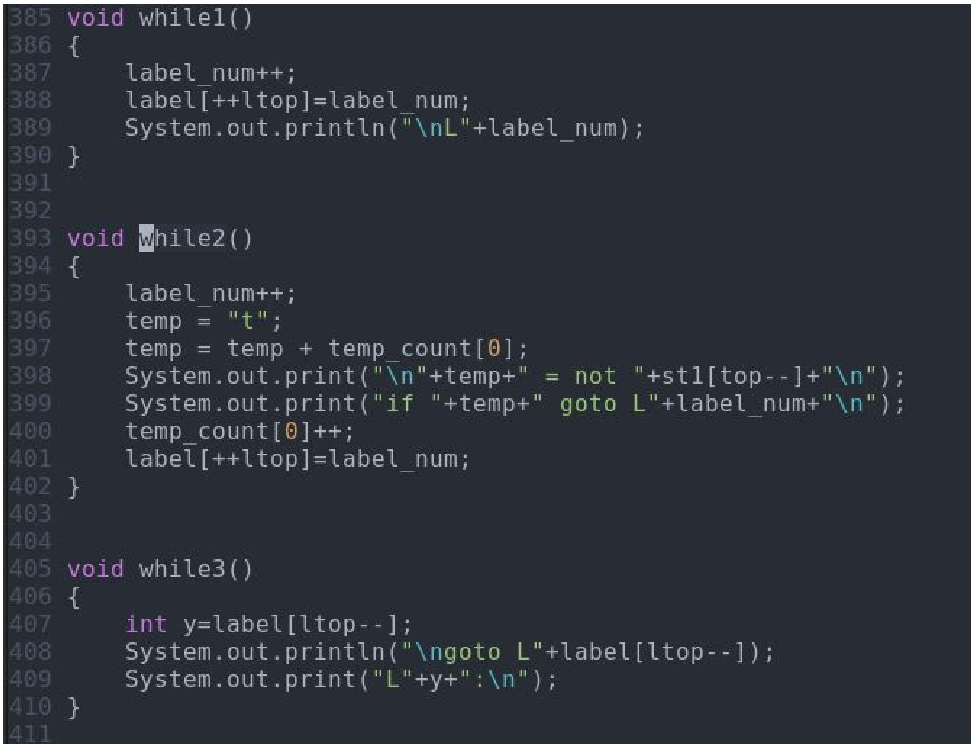
The symbol table is populated in the syntax phase and updated in semantic phase. It contains the following:

- Serial No - The number of entries in the table.
- Identifier - Name of the variables.
- Scope - Scope of the variable.
- Value - Mathematical value of the variable if defined.
- Type - Data Type of the variable.
- Parameter type - Data type of the function parameters.

Intermediate Code Generation:

The intermediate code is generated as it passes through the grammar, with help of some special sub-routines defined for the language clauses such as FOR, WHILE..etc. These routines are called as mid-rule actions.

Subroutine example for the WHILE Clause:



```
void while1()
{
    label_num++;
    label[++ltop]=label_num;
    System.out.println("\nL"+label_num);
}


void while2()
{
    label_num++;
    temp = "t";
    temp = temp + temp_count[0];
    System.out.print("\n"+temp+" = not "+st1[top--]+"\n");
    System.out.print("if "+temp+" goto L"+label_num+"\n");
    temp_count[0]++;
    label[++ltop]=label_num;
}


void while3()
{
    int y=label[ltop--];
    System.out.println("\ngoto L"+label[ltop--]);
    System.out.print("L"+y+":\n");
}
```

Code Optimization:

The input Intermediate Code is scanned and any values that are available at compile time are propagated forward. Then another pass is made to check if any arithmetic instructions can be done before moving to the nest phase.

Error Handling:

Scanner:

For any tokens which don’t match the lexical rules, a -1 is returned to the parser.

Parser:

The error recovery is done using the ***‘error’*** keyword which is included in the grammar at certain places. There is a function yyerror() which handles the error, if the grammar isn’t matched. It prints the line number at which the error occurs. It handles the error and continues parsing the rest of the code.

Semantic Analyzer:

The error handling here is done by writing certain mid-rule actions, which are basically java code, to check if the token passed makes semantic sense according to the language.

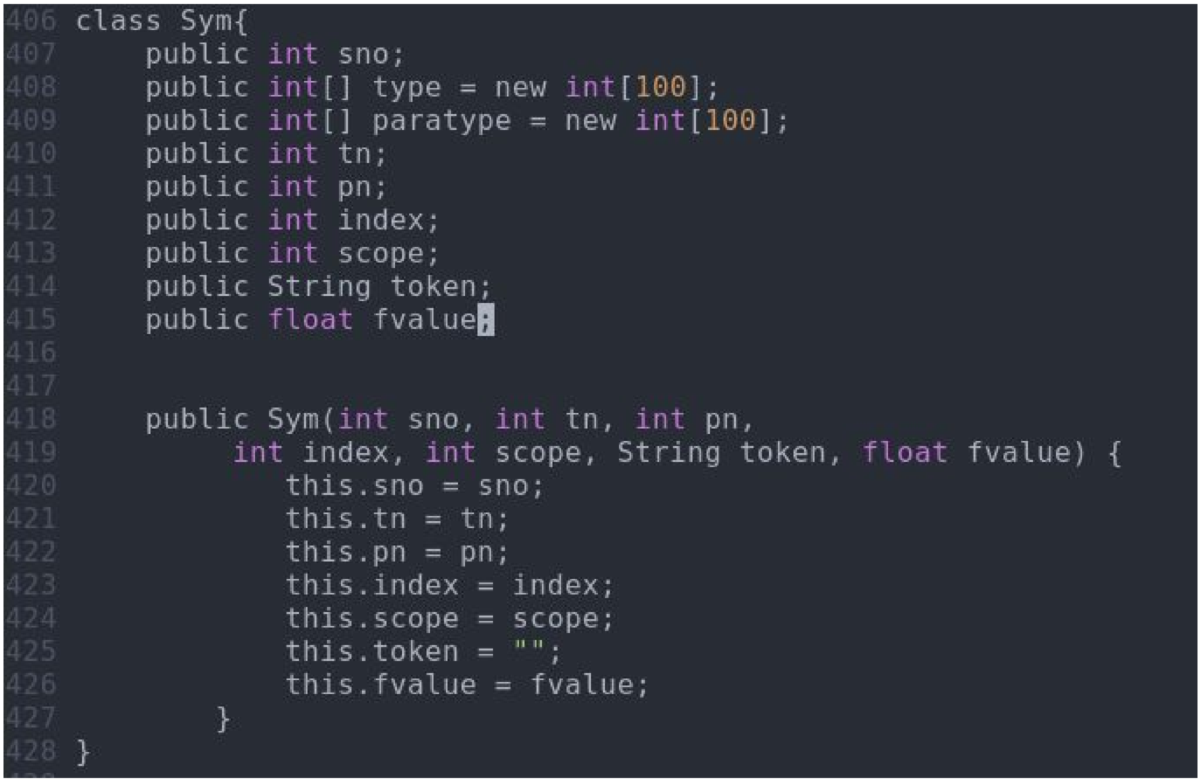
Target Code Generation:

There is no error handling in this phase. It is dependent on the previous phases to catch all errors.

Implementation Details

Symbol Table Creation:

The object of the following class was populated accordingly.



```
class Sym{
    public int sno;
    public int[] type = new int[100];
    public int[] paratype = new int[100];
    public int tn;
    public int pn;
    public int index;
    public int scope;
    public String token;
    public float fvalue;


    public Sym(int sno, int tn, int pn,
         int index, int scope, String token, float fvalue) {
            this.sno = sno;
            this.tn = tn;
            this.pn = pn;
            this.index = index;
            this.scope = scope;
            this.token = "";
            this.fvalue = fvalue;
        }
}
```

Intermediation Code Generation:

The intermediate code while getting generated is printed out directly. But the ideal way to do it would be to store it in a Quadruples table.

Code Optimization:

First the input icg gets scanned so that all known values are stored in a dictionary which can later be propagated to other lines in the code.

Then the icg is scanned again to check for any arithmetic instructions that can be evaluated then and there. This step is done after the variable propagation is done.

Assembly Code Generation:

re

Error Handling:

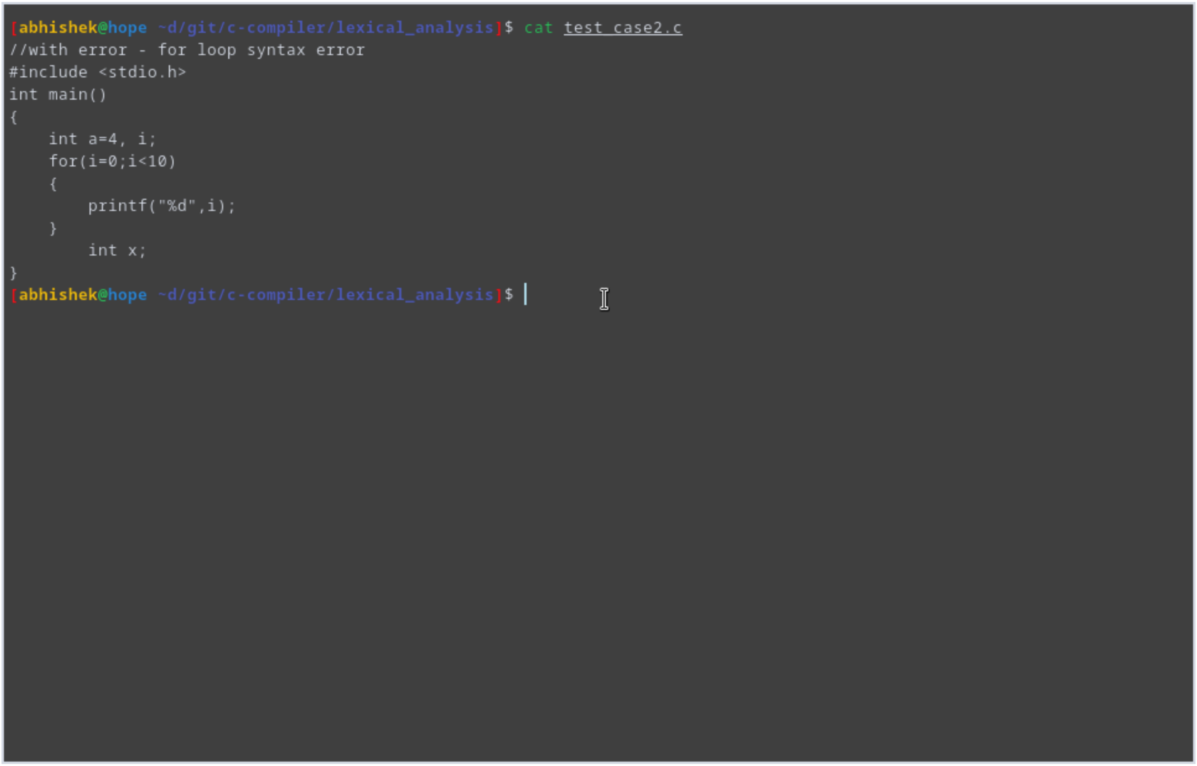
As mentioned in the Design Strategy.

Results and Shortcomings

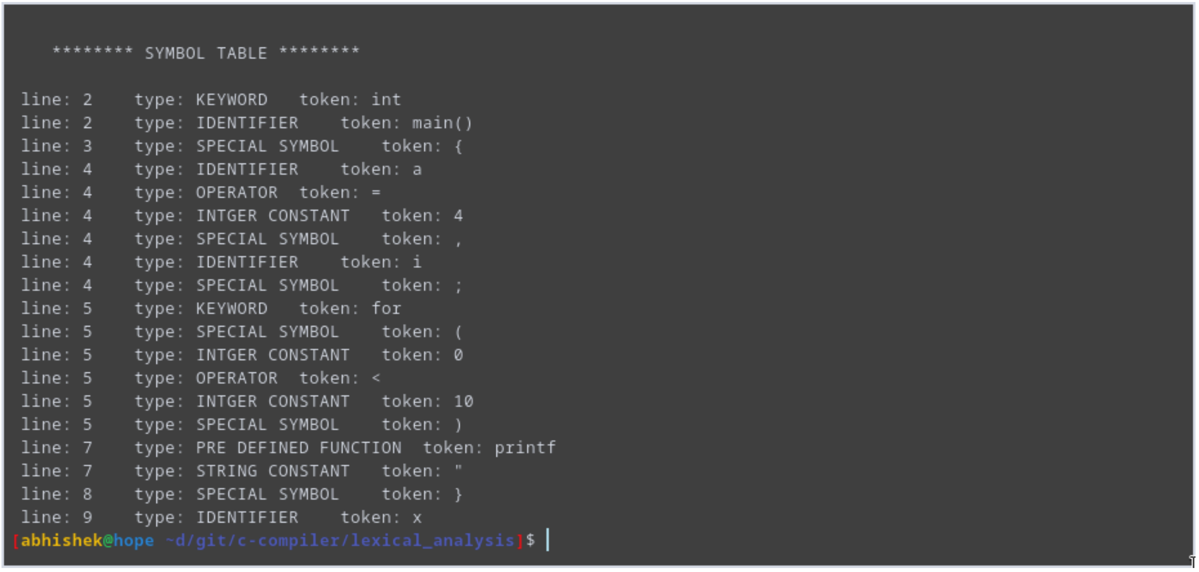
The mini-compiler does a good job in compiling real basic C-code to a hypothetical target code. It handles the error well, and has proper semantic
handling. It also does certain code optimizations.
Few shortcoming would be that it doesn’t store the ICG in Quadruples table. Also, due to our lack of familiarity with Java, we weren’t able to implement abstract syntax tree and hence scraped it.

Snapshots

Lexical Analysis

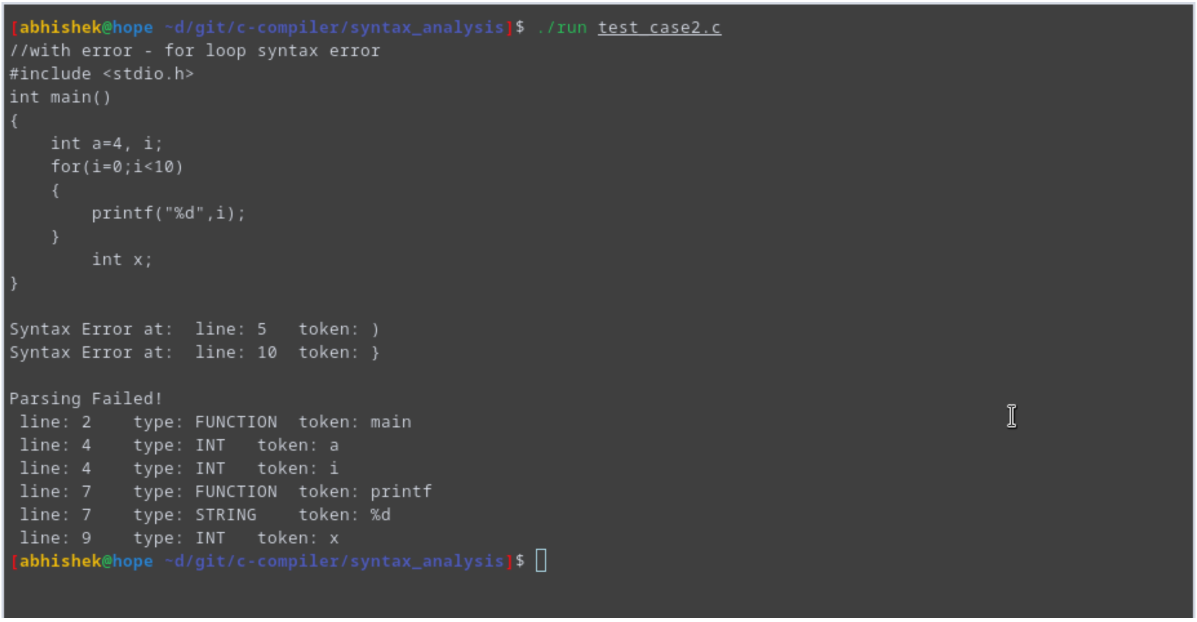


```
[abhishek@hope ~d/git/c-compiler/lexical_analysis]$ cat test_case2.c
//with error - for loop syntax error
#include <stdio.h>
int main()
{
    int a=4, i;
    for(i=0;i<10)
    {
        printf("%d",i);
    }
        int x;
}
[abhishek@hope ~d/git/c-compiler/lexical_analysis]$
```



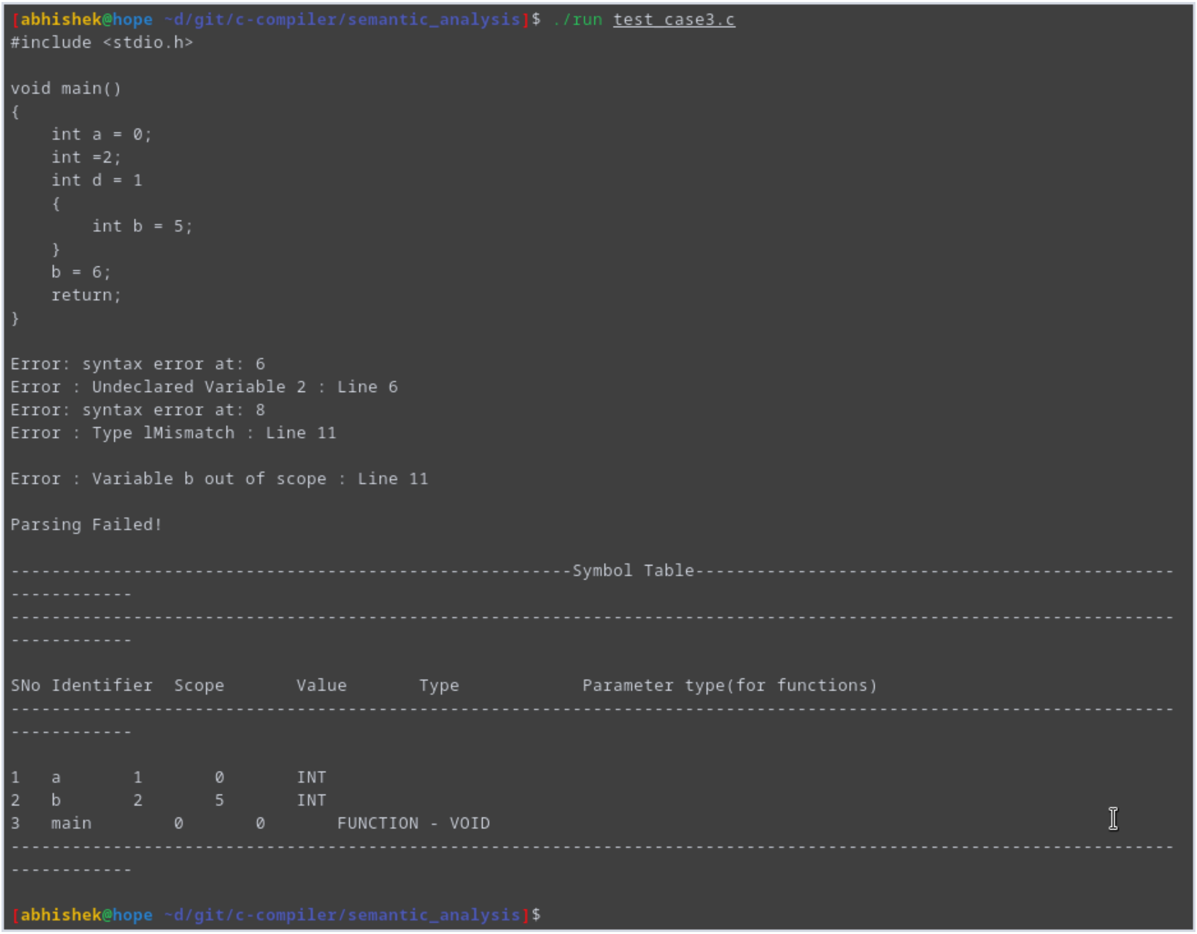
```
   ******** SYMBOL TABLE ********

 line: 2    type: KEYWORD   token: int
 line: 2    type: IDENTIFIER    token: main()
 line: 3    type: SPECIAL SYMBOL    token: {
 line: 4    type: IDENTIFIER    token: a
 line: 4    type: OPERATOR  token: =
 line: 4    type: INTGER CONSTANT   token: 4
 line: 4    type: SPECIAL SYMBOL    token: ,
 line: 4    type: IDENTIFIER    token: i
 line: 4    type: SPECIAL SYMBOL    token: ;
 line: 5    type: KEYWORD   token: for
 line: 5    type: SPECIAL SYMBOL    token: (
 line: 5    type: INTGER CONSTANT   token: 0
 line: 5    type: OPERATOR  token: <
 line: 5    type: INTGER CONSTANT   token: 10
 line: 5    type: SPECIAL SYMBOL    token: )
 line: 7    type: PRE DEFINED FUNCTION  token: printf
 line: 7    type: STRING CONSTANT   token: "
 line: 8    type: SPECIAL SYMBOL    token: }
 line: 9    type: IDENTIFIER    token: x
[abhishek@hope ~d/git/c-compiler/lexical_analysis]$
```

Syntax Analysis: (wrong for loop syntax)



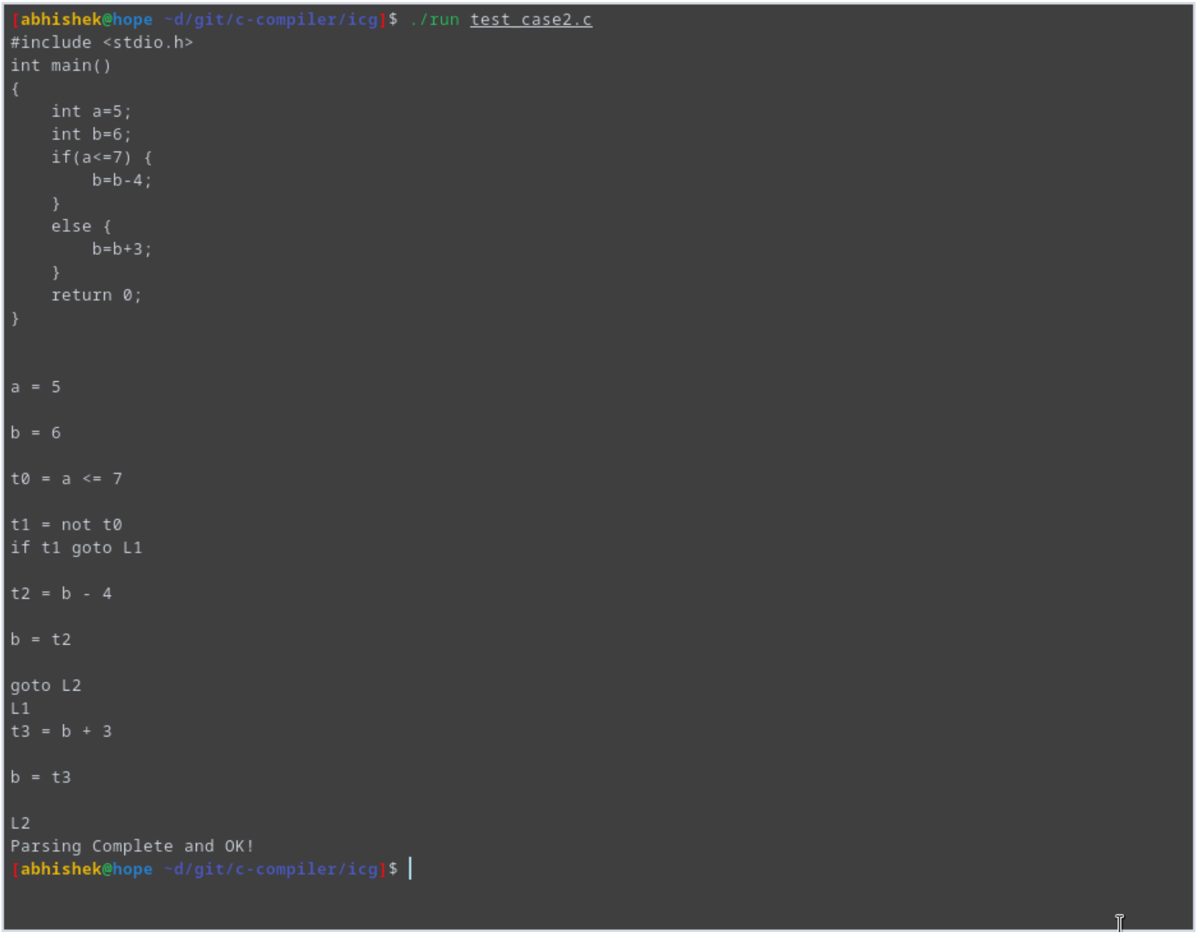
```
[abhishek@hope ~d/git/c-compiler/syntax_analysis]$ ./run test_case2.c
//with error - for loop syntax error
#include <stdio.h>
int main()
{
    int a=4, i;
    for(i=0;i<10)
    {
        printf("%d",i);
    }
        int x;
}

Syntax Error at:  line: 5   token: )
Syntax Error at:  line: 10  token: }

Parsing Failed!
 line: 2    type: FUNCTION  token: main
 line: 4    type: INT   token: a
 line: 4    type: INT   token: i
 line: 7    type: FUNCTION  token: printf
 line: 7    type: STRING    token: %d
 line: 9    type: INT   token: x
[abhishek@hope ~d/git/c-compiler/syntax_analysis]$ 
```

Semantic Analysis: (scope of variable b, semicolon missing)



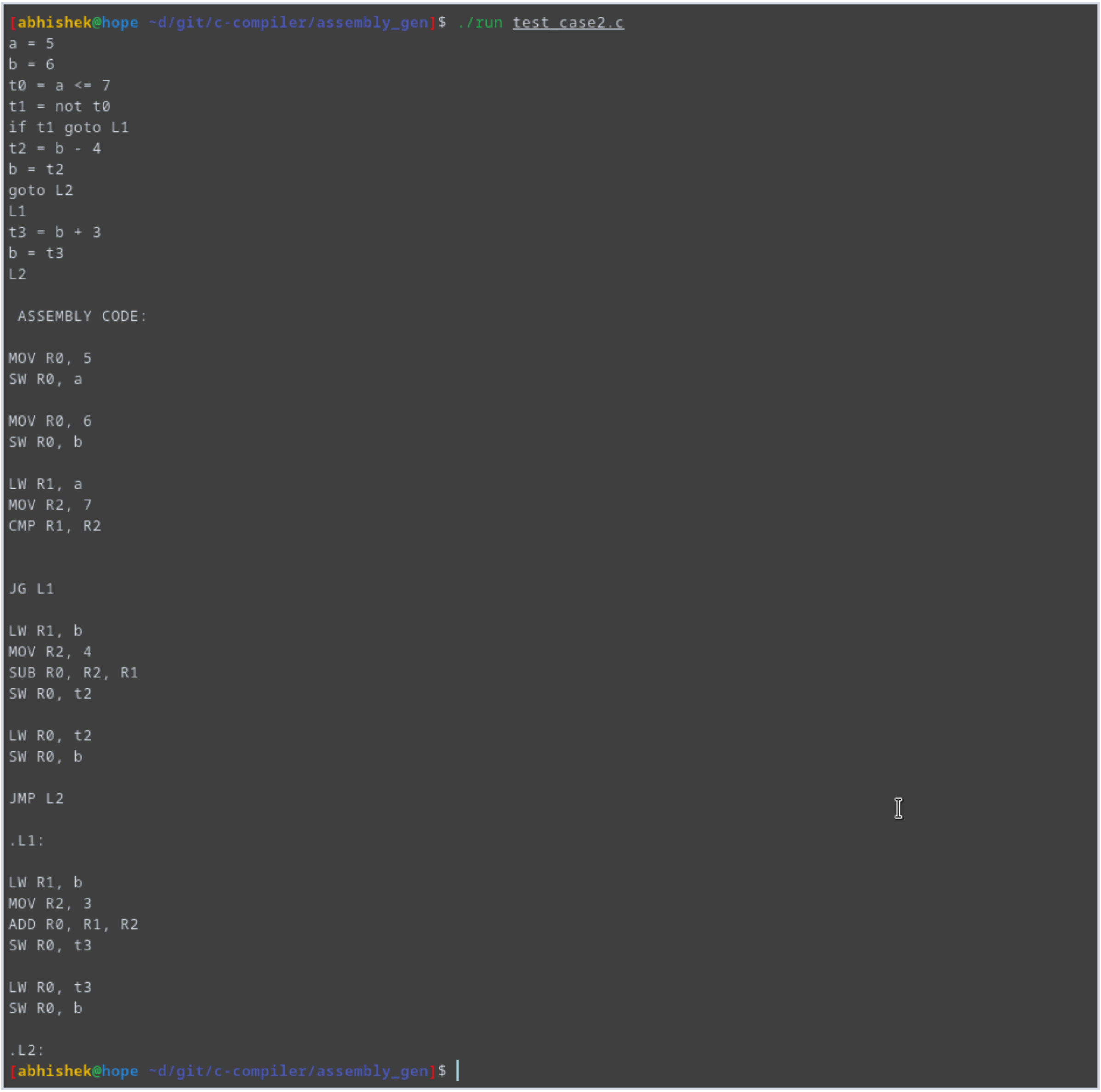
```
[abhishek@hope ~d/git/c-compiler/semantic_analysis]$ ./run test_case3.c
#include <stdio.h>

void main()
{
    int a = 0;
    int =2;
    int d = 1
    {
        int b = 5;
    }
    b = 6;
    return;
}

Error: syntax error at: 6
Error : Undeclared Variable 2 : Line 6
Error: syntax error at: 8
Error : Type 1Mismatch : Line 11

Error : Variable b out of scope : Line 11

Parsing Failed!

-------------------------------------------------------Symbol Table-----------------------------------------------
------------
------------------------------------------------------------------------------------------------------------------
------------

SNo Identifier  Scope       Value       Type            Parameter type(for functions)
------------------------------------------------------------------------------------------------------------------
------------

1   a       1       0       INT
2   b       2       5       INT
3   main        0       0       FUNCTION - VOID
------------------------------------------------------------------------------------------------------------------
------------

[abhishek@hope ~d/git/c-compiler/semantic_analysis]$
```

Intermediate Code Generation:



```
[abhishek@hope ~d/git/c-compiler/icg]$ ./run test_case2.c
#include <stdio.h>
int main()
{
    int a=5;
    int b=6;
    if(a<=7) {
        b=b-4;
    }
    else {
        b=b+3;
    }
    return 0;
}


a = 5

b = 6

t0 = a <= 7

t1 = not t0
if t1 goto L1

t2 = b - 4

b = t2

goto L2
L1
t3 = b + 3

b = t3

L2
Parsing Complete and OK!
[abhishek@hope ~d/git/c-compiler/icg]$
```

Assembly Code Generation:



```
[abhishek@hope ~d/git/c-compiler/assembly_gen]$ ./run test case2.c
a = 5
b = 6
t0 = a <= 7
t1 = not t0
if t1 goto L1
t2 = b - 4
b = t2
goto L2
L1
t3 = b + 3
b = t3
L2

 ASSEMBLY CODE:

MOV R0, 5
SW R0, a

MOV R0, 6
SW R0, b

LW R1, a
MOV R2, 7
CMP R1, R2


JG L1

LW R1, b
MOV R2, 4
SUB R0, R2, R1
SW R0, t2

LW R0, t2
SW R0, b

JMP L2

.L1:

LW R1, b
MOV R2, 3
ADD R0, R1, R2
SW R0, t3

LW R0, t3
SW R0, b

.L2:
[abhishek@hope ~d/git/c-compiler/assembly_gen]$
```

Conclusions:

The mini-compiler supports the clauses mentioned at the start of the project. It parses the source code, displays any errors with line number mentioned if found. Optimizes the code if possible and generates the target machine code. This concludes our project.

Future Enhancements:

In future, we can extend it to support a larger subset of the C-grammar. Also we could generate actual machine code for some architecture like x86 or amd64.