Report on

"Mini Compiler For Python-3 Using Lex and Yacc"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Himanshu Jain	**PES1201802828**
Rahil N Modi	**PES1201802826**
Sooryanath I T	**PES1201802827**

Under the guidance of

Preet Kanwal
Associate Professor , Department of CSE
PES University, Bengaluru

January – May 2021

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

- The compiler is built to serve and compile **python programs** of **version 3.x** with the assistance of tools like lex and yacc in union with few **C programming** constructs to serve as routines in implementing syntax-direct translations and optimizations.
- The constructs handled briefly include

    1. Arithmetic , boolean , relation operations.
    2. range() , lists .
    3. assignments , leader and suite blocks.
    4. While , if , elif and else block (suites) .
    5. module imports , break and continue statements.

- The sample input provided for the program is as follows

final_input_case1.txt

```
import math
import re

a = 10
b = 200
d = 20
f = a / 2

if a==3:
	print(a)
	a = b
	while(a < b):
		print(e)
		break
		a = a + 1
elif(a == 2):
	print(b)
	return y
else:
	print(c)
	a = a + b

a = True or False
c = d
e = a * f
```

Sample Output :

Lexer And Program Parsing Output (Syntax phase):

```
T_Import T_Package
T_Nl  T_Import T_Package
T_Nl  T_Id T_Asgn T_Num
T_Nl  T_Id T_Asgn T_Num
T_Nl  T_Id T_Asgn T_Num
T_Nl  T_Id T_Asgn T_Id T_Div T_Num
T_Nl  T_If T_Id T_Eq T_Num T_Colon
T_Nl  T_ID T_Print T_Op T_Id T_Cp
T_Nl  T_ND T_Id T_Asgn T_Id
T_Nl  T_ND T_While T_Op T_Id T_Lt T_Id T_Cp T_Colon
T_Nl  T_ID T_Print T_Op T_Id T_Cp
T_Nl  T_ND T_Break
T_Nl  T_ND T_Id T_Asgn T_Id T_Plus T_Num
T_Nl  T_Elif T_Op T_Id T_Eq T_Num T_Cp T_Colon
T_Nl  T_ID T_Print T_Op T_Id T_Cp
T_Nl  T_ND T_Return T_Id
T_Nl  T_Else T_Colon
T_Nl  T_ID T_Print T_Op T_Id T_Cp
T_Nl  T_ND T_Id T_Asgn T_Id T_Plus T_Id
T_Nl  T_Id T_Asgn T_True T_Or T_False
T_Nl  T_Id T_Asgn T_Id
T_Nl  T_Id T_Asgn T_Id T_Mul T_Id
T_Nl  T_EOF
VALID
```

Symbol Table Generation

SYMBOL TABLE

NAME	DATATYPE	VALUE	LINENO
a	BLN	1	[4,11,15,22,22]
b	INT	200	[5]
d	INT	20	[6]
t1	INT	5	[6]
f	INT	5	[8]
t2	BLR	0	[8]
t3	BLR	0	[11]
t4	INT	201	[15]
t5	BLR	0	[15]
t6	INT	401	[22]
t7	BLR	1	[22]
c	INT	20	[24]
t8	INT	5	[25]
e	INT	5	[25]

Three address code generation in quadruple format

```
-------------Three Address Code in Quadruple Format-------------

            Op    Arg1  Arg2  Res
      1     importNULL  NULL  math
      2     importNULL  NULL  re
      3     =     10    NULL  a
      4     =     200   NULL  b
      5     =     20    NULL  d
      6     /     a     2     t1
      7     =     t1    NULL  f
      8     ==    a     3     t2
      9     iffalse     t2    NULL  L5
      10    print NULL  NULL  a
      11    =     b     NULL  a
      12    label NULL  NULL  L2
      13    <     a     b     t3
      14    iffalse     t3    NULL  L1
      15    print NULL  NULL  e
      16    break NULL  NULL  NULL
      17    +     a     1     t4
      18    =     t4    NULL  a
      19    goto  NULL  NULL  L2
      20    label NULL  NULL  L1
      21    goto  NULL  NULL  L3
      22    label NULL  NULL  L5
      23    ==    a     2     t5
      24    iffalse     t5    NULL  L4
      25    print NULL  NULL  b
      26    returny     NULL  NULL
      27    goto  NULL  NULL  L3
      28    label NULL  NULL  L4
      29    print NULL  NULL  c
      30    +     a     b     t6
      31    =     t6    NULL  a
      32    label NULL  NULL  L3
      33    or    True  False t7
      34    =     t7    NULL  a
      35    =     d     NULL  c
      36    *     a     f     t8
      37    =     t8    NULL  e
```

Dead code eliminated 3 Address Code

```
-------------Post Dead Code Elimination -------------

-------------Three Address Code in Quadruple Format-------------

          Op    Arg1  Arg2  Res
     1    =     10    NULL  a
     2    =     200   NULL  b
     3    =     20    NULL  d
     4    /     a     2     t1
     5    =     t1    NULL  f
     6    ==    a     3     t2
     7    label NULL  NULL  L5
     8    ==    a     2     t5
     9    label NULL  NULL  L4
     10   print NULL  NULL  c
     11   +     a     b     t6
     12   =     t6    NULL  a
     13   label NULL  NULL  L3
     14   or    True  False t7
     15   =     t7    NULL  a
     16   =     d     NULL  c
     17   *     a     f     t8
     18   =     t8    NULL  e
```

ARCHITECTURE

- We have covered basic constructs of python which are used in day to day programming style such as if - else block and while statements which can be nested up to any finite extent.
- Indentations , nesting , arithmetic expressions , boolean expressions , leader - suites are also realised and implemented.
- imports , break keywords , pass keywords , range() and lists are handled to supplement the core features.
- Undefined variables , unknown symbols are notified with appropriate error diagnostics .
- The line of occurrences and the assigned expression values are folded and propagated , this is tracked and visible in the symbol table.
- Semantic rules are written to evaluate expressions and generate 3AC .

- Optimization techniques are applied over the quadruple format of 3AC.

LITERATURE SURVEY

1 . Yacc and Lex doc by Tom Niemann

2. The official documentation for Bison: https://www.gnu.org/software/bison/manual/ [1]

CONTEXT FREE GRAMMAR

```
                        Program Structure

Prog     :   Parsing T_EOF ;

Parsing :   T_Nl Parsing | Statements;

Constant    :   T_Num | T_String;

Variable    :   T_Id  %prec Only_Id ;
```

```
                        List and Iterables

List_Index  :   T_Id T_Osb Term T_Csb ;

List_Initialisation :   T_Osb List_elements T_Csb | T_List T_Etuple | T_Elist ;

List_elements   :   Constant T_Comma List_elements
                |   T_Id T_Comma List_elements
                |   List_Initialisation T_Comma List_elements
                |   Constant
                |   T_Id
                |   List_Initialisation
                ;

Range   :   T_Range T_Op T_Num T_Cp
        |   T_Range T_Op T_Num T_Comma T_Num T_Cp
        |   T_Range T_Op T_Num T_Comma T_Num T_Num T_Cp
        ;

Iterable    :   T_Id |   Range ;

Term    :   Variable |   Constant |   List_Index | Range ;
```

Boolean and Arithmetic Expressions

```
Boolean_Exp :   Boolean_Term T_And Boolean_Term
            |   Boolean_Term T_Or Boolean_Term
            |   Boolean_Term
            ;

Boolean_Term    :   T_False
                |   T_True
                |   Arithmetic_Exp T_In Iterable
                |   Arithmetic_Exp T_Lt Arithmetic_Exp
                |   Arithmetic_Exp T_Lte Arithmetic_Exp
                |   Arithmetic_Exp T_Gt Arithmetic_Exp
                |   Arithmetic_Exp T_Gte Arithmetic_Exp
                |   Arithmetic_Exp T_Neq Arithmetic_Exp
                |   Arithmetic_Exp T_Eq Arithmetic_Exp
                |   Boolean_Next
                ;

Boolean_Next    :   T_Not Boolean_Term
                |   T_Op Boolean_Exp T_Cp
                |   T_Op T_Id T_Cp
                |   T_Op T_Not T_Id T_Cp
                ;

Arithmetic_Exp  :   Arithmetic_Exp T_Add Arithmetic_Term
                |   Arithmetic_Exp T_Minus Arithmetic_Term
                |   Arithmetic_Term
                ;

Arithmetic_Term :   Arithmetic_Term T_Mul Arithmetic_Factor
                |   Arithmetic_Term T_Div Arithmetic_Factor
                |   Arithmetic_Term T_Mod Arithmetic_Factor
                |   Arithmetic_Factor
                ;

Arithmetic_Factor   :   Term
                    |   T_Op Arithmetic_Exp T_Cp
                    ;

Assignment_Stmt :   Variable T_Asgn Arithmetic_Exp
                |   Variable T_Asgn List_Initialisation
                ;
```

```
                                        Statements


Pass_Stmt    : T_Pass ;
Import_Stmt :    T_Import T_Package ;
Print_Stmt  :    T_Print T_Op Term T_Cp ;
Break_Stmt  : T_break ;
Return_Stmt : T_Return ;
Expression_Stmt : Boolean_Exp | Arithmetic_Exp ;

Basic_Stmt  :    Expression_Stmt
            |    Assignment_Stmt
            |    Pass_Stmt
            |    Print_Stmt
            |    Break_Stmt
            |    Return_Stmt
            |    Import_Stmt
            ;

Suite    :   T_Nl T_ID Statements
         ;
Statements  :    Basic_Stmt T_Nl T_ND Statements
            |    Compound_Stmt T_Nl T_ND Statements
            |    Basic_Stmt T_Nl Statements
            |    Compound_Stmt T_Nl Statements
            |    Basic_Stmt T_Nl T_ID
            |    Basic_Stmt T_Nl
            |    Compound_Stmt
            ;
```

```
                                        Conditions and Loops


Compound_Stmt   :   If_Stmt |   While_Stmt |    For_Stmt ;

While_Stmt  :   T_While Boolean_Exp T_Colon Suite T_DD
            |   T_While Boolean_Exp T_Colon Suite   %prec While_without_T_DD
            ;

If_Stmt :   T_If Boolean_Exp T_Colon Suite T_DD Elif
        |   T_If Boolean_Exp T_Colon Suite TEMP Elif
        ;

Elif    :   T_Elif Boolean_Exp T_Colon Suite T_DD Elif
        |   T_Elif Boolean_Exp T_Colon Suite TEMP Elif
        |   Else
        ;

Else    : T_Else T_Colon Suite T_DD
        | T_Else T_Colon Suite {Pop_Tabspaces();}   %prec Else_without_T_DD
        |       %prec Lambda
        ;

For_Stmt    :   T_For T_Id T_In Iterable T_Colon Suite T_DD  %prec No_Else_In_For
            |   T_For T_Id T_In Iterable T_Colon Suite {Pop_Tabspaces();} %prec For_without_T_DD
            ;
```

DESIGN STRATEGY

SYMBOL TABLE CREATION

- Symbol table stores the values of variables and temporaries
- The structure of the symbol table is as follows:
  - name (String)
  - datatype (String)
  - value (String)
  - line (List of lineno)
- The following functions have been implemented for symbol table:
  - display = Displays the entire symbol table
  - search = Search for an entry based on the name field
  - insert = Insert a new record if does not exists
  - insert_exist = Determine if a record exists in the symbol table

INTERMEDIATE CODE GENERATION

- Three address code for the input program is generated and stored in the **Quadruple** format:
  - Operator
  - Arg1
  - Arg2
  - Result

CODE OPTIMIZATION

- The following optimizations have been implemented
  - **Constant Folding**

Expression evaluation is performed using the semantic rules and the result is stored in the symbol table for both variables and the temporaries. This is done for both arithmetic expressions and boolean expressions.

a = 1 + 2

After constant folding, a = 3

- **Constant Propagation**

  The values computed using constant folding are used in the program wherever the variables are used.

  a = 2

  b = a + 3

  After constant propagation, b = 2 + 3

- **Strength Reduction**

  strength reduction is a compiler optimization where expensive operations are replaced with equivalent but less expensive operations.

  1. **before** : a = b * 64

     after : a = b << 6 (equivalent of multiplying with 2^6)
  2. **before** : a = b / 8

     after : a = b >> 3

  bit shifting is faster and less expensive in terms of clock cycles than traditional multiplication and division.

  3. **before** : b = expression_a and expression_b

     after : b = (expression_a == 0) 0 : expression_b
  4. **before** : b = expression_a or expression_b

after : b = (expression_a == 0)?expression_b:expression_a

- **Dead Code Elimination**

  Eliminating unused imported modules:

  When the interpreter executes the above `import` statement, it searches for the module in the current directory , but what if the module isn't used at all ? This leads to wastage of dynamic loading of the module.

  before :
```
import re
a = 10
b = 10
c = a+ b
print(c)
```
  after :
```
a = 10
b = 10
c  = 20 #constant fold + prop
print(20)
```

  Eliminating unreachable code beyond break and continue statements:

  When the statements are placed after terminating statements such as break or continue , then they do not get executed at all , so any block of code beyond break and continue isn't reachable.

  before :
```
if (a ==100):
    while a < 100:
        a = a -1
        break
        print("garbage")
```
  after :
```
if (a == 100):
    while a<100:
        a = a-1
        break
```

  loop optimizations and skipping decisions loops

  if the loop entry criteria can be computed at compile time , then the loop can be either skipped or retained based on the truth value at compile time.

  before :
```
a = 99
if a > 100:
```
  after :
```
a =99
print("less than 100")
```

```
print("greater than 100")
else:
    print("less than 100")
```

ERROR HANDLING

- Division by zero
- Variable not defined
- Syntax Error
- Indentation Error
- Lexical Error with Panic mode recovery

IMPLEMENTATION DETAILS

SYMBOL TABLE CREATION

```
typedef struct SymbolTable
{                                  // a = 10
    char* name;                    // a
    char* datatype;                // INT
    char* value;                   // 10
    list* line;                    // 1
}SymbolTable;

typedef struct list
{
    int lineno;
    struct list* next;
}list;
SymbolTable* symTable[SIZE];
```

A structure is created to store individual entries in the symbol table and the entire symbol table is viewed as an array of structures.

Hashing technique is used to obtain an index for an element. The following in the hash function used :

```
strlen(name) % SIZE;
```

INTERMEDIATE CODE GENERATION

```
typedef struct Quadruple{
      char Operator[10];
      char Arg1[10];
      char Arg2[10];
      char Result[100];
      struct Quadruple *next;
}Q;
```

Three address code is generated and its Quadruple format is stored in the table. Further the semantic rules are used to generate the code and put it in appropriate fields in Quadruple format. (Used as a data structure to store Three Address Code).

CODE OPTIMIZATION

- Constant folding and constant propagation is implemented by using the semantic rules and entries created in the symbol table.
- Strength reduction
- Dead Code Elimination

Steps to run the program

- Download and place **lexer.l** and **parser.y** files in a directory
    - Github repository to download the files : https://github.com/CD-2021/2826-2827-2828

- Open the terminal in the same directory
- Run the following commands to get the output
    - ./automate.sh
      (OR)
    - yacc -d -v parser.y
      lex lexer.l
      gcc lex.yy.c y.tab.c -ll
      ./a.out < final_input_case1.txt

RESULTS AND SNAPSHOTS (Outputs)

- The whole compiler design process has seven phases divided into front end, machine independent optimization (optional phase), back end. Symbol Table and Error Handler are connected to all phases.
- Front End includes:
    - Lexical Analyser - Its output is tokens (Eg: T_Not, T_And, T_Or, etc.)

```
"from"                  {update("From");return T_From;}
"as"                    {update("As");return T_As;}
"if"                    {update("If");return T_If;}
"elif"                  {update("Elif");return T_Elif;}
"else"                  {update("Else");return T_Else;}
"while"                 {update("While");return T_While;}
"for"                   {update("For");return T_For;}
"import"                {update("Import");return T_Import;}
"print"                 {update("Print");return T_Print;}
"list"                  {update("List");return T_List;}
"tuple"                 {update("Tuple");return T_Tuple;}
"def"                   {update("Def");return T_Def;}
"False"                 {update("False");return T_False;}
"True"                  {update("True");return T_True;}
"class"                 {update("Class");return T_Class;}
"continue"              {update("Continue");return T_Continue;}
"break"                 {update("Break");return T_break;}
"in"                    {update("In");return T_In;}
"pass"                  {update("Pass");return T_Pass;}
```

    - Syntactic Analyser - Its output is the semantic validity of a given input file. Whether it is valid or a syntax error.

```
T_Import T_Package
T_Nl  T_Import T_Package
T_Nl  T_Id T_Asgn T_Num
T_Nl  T_Id T_Asgn T_Num
T_Nl  T_Id T_Asgn T_Num
T_Nl  T_Id T_Asgn T_Id T_Div T_Num
T_Nl  T_If T_Id T_Eq T_Num T_Colon
T_Nl  T_ID T_Print T_Op T_Id T_Cp
T_Nl  T_ND T_Id T_Asgn T_Id
T_Nl  T_ND T_While T_Op T_Id T_Lt T_Id T_Cp T_Colon
T_Nl  T_ID T_Print T_Op T_Id T_Cp
T_Nl  T_ND T_Break
T_Nl  T_ND T_Id T_Asgn T_Id T_Plus T_Num
T_Nl  T_Elif T_Op T_Id T_Eq T_Num T_Cp T_Colon
T_Nl  T_ID T_Print T_Op T_Id T_Cp
T_Nl  T_ND T_Return T_Id
T_Nl  T_Else T_Colon
T_Nl  T_ID T_Print T_Op T_Id T_Cp
T_Nl  T_ND T_Id T_Asgn T_Id T_Plus T_Id
T_Nl  T_Id T_Asgn T_True T_Or T_False
T_Nl  T_Id T_Asgn T_Id
T_Nl  T_Id T_Asgn T_Id T_Mul T_Id
T_Nl  T_EOF
VALID
```

- Semantic Analyser - It is used to find the type and value of variables and store them appropriately in the symbol table.

SYMBOL TABLE

NAME	DATATYPE	VALUE	LINENO
a	BLN	1	[4,11,15,22,22]
b	INT	200	[5]
d	INT	20	[6]
t1	INT	5	[6]
f	INT	5	[8]
t2	BLR	0	[8]
t3	BLR	0	[11]
t4	INT	201	[15]
t5	BLR	0	[15]
t6	INT	401	[22]
t7	BLR	1	[22]
c	INT	20	[24]
t8	INT	5	[25]
e	INT	5	[25]

Note : The line number of temporaries is wrt 3AC

- Intermediate Code Generation - It generates Three address code partly using semantic rules and storing it in Quadruple form.

-------------Three Address Code in Quadraple Format-------------

	Op	Arg1	Arg2	Res
1	import	NULL	NULL	math
2	import	NULL	NULL	re
3	=	10	NULL	a
4	=	200	NULL	b
5	=	20	NULL	d
6	/	a	2	t1
7	=	t1	NULL	f
8	==	a	3	t2
9	iffalse	t2	NULL	L5
10	print	NULL	NULL	a
11	=	b	NULL	a
12	label	NULL	NULL	L2
13	<	a	b	t3
14	iffalse	t3	NULL	L1
15	print	NULL	NULL	e
16	break	NULL	NULL	NULL
17	+	a	1	t4
18	=	t4	NULL	a
19	goto	NULL	NULL	L2
20	label	NULL	NULL	L1
21	goto	NULL	NULL	L3
22	label	NULL	NULL	L5
23	==	a	2	t5
24	iffalse	t5	NULL	L4
25	print	NULL	NULL	b
26	return	y	NULL	NULL
27	goto	NULL	NULL	L3
28	label	NULL	NULL	L4
29	print	NULL	NULL	c
30	+	a	b	t6
31	=	t6	NULL	a
32	label	NULL	NULL	L3
33	or	True	False	t7
34	=	t7	NULL	a
35	=	d	NULL	c
36	*	a	f	t8
37	=	t8	NULL	e

- Machine Independent Code optimization - This phase takes 3AC as the input and produces optimized code as output. Four optimizations are implemented here:
  - Constant Folding

```
e = 5
f = 6
d = 7
r = (e + f)*(d - f) # r = 11 (Constant Folding)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
t3    INT    11    [3]
r     INT    11    [5]
a     INT    1     [6]
```

  - Constant Propagation

```
a = 1
b = (a * 2) + a     # b = 3  (Constant Propagation)
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

    t5      INT     3       [8]
    b       INT     3       [8]
    t6      BLR     1       [8]
```

- ○ Strength Reduction: Doing multiplication using left and right shifts to reduce cycles.

```
int multiply(int m , int n)
{
    int answer = 0, counter = 0;
    while (m)
    {
        // check for set bit and left
        // shift n, count times
        if (m % 2 == 1)
            answer += n << counter;

        // increment of place value (count)
        counter++;
        //divide m by 2 , aka right shift
        m >>= 1;
    }
    return answer;

}
```

- ○ Dead Code Elimination

```
-------------Post Dead Code Elimination -------------


-------------Three Address Code in Quadraple Format-------------

                Op      Arg1    Arg2    Res
        1       =       10      NULL    a
        2       =       200     NULL    b
        3       =       20      NULL    d
        4       /       a       2       t1
        5       =       t1      NULL    f
        6       ==      a       3       t2
        7       label   NULL    NULL    L5
        8       ==      a       2       t5
        9       label   NULL    NULL    L4
        10      print   NULL    NULL    c
        11      +       a       b       t6
        12      =       t6      NULL    a
        13      label   NULL    NULL    L3
        14      or      True    False   t7
        15      =       t7      NULL    a
        16      =       d       NULL    c
        17      *       a       f       t8
        18      =       t8      NULL    e
```

- Symbol Table: It is a table used to store all variables and temporaries with values, type and line numbers.

```
                         SYMBOL TABLE

          NAME      DATATYPE      VALUE          LINENO

          a         BLN           1              [4,11,15,22,22]
          b         INT           200            [5]
          d         INT           20             [6]
          t1        INT           5              [6]
          f         INT           5              [8]
          t2        BLR           0              [8]
          t3        BLR           0              [11]
          t4        INT           201            [15]
          t5        BLR           0              [15]
          t6        INT           401            [22]
          t7        BLR           1              [22]
          c         INT           20             [24]
          t8        INT           5              [25]
          e         INT           5              [25]

 Note : The line number of temporaries is wrt 3AC
```

ALL TEST CASES:

- Test Case 1: If-Elif-Else

```
final_input_case1.txt
import math
import re

a = 10
b = 200
d = 20
f = a / 2

if a==3:
    print(a)
    a = b
    while(a < b):
        print(e)
        break
        a = a + 1
elif(a == 2):
    print(b)
    return y
else:
    print(c)
    a = a + b

a = True or False
c = d
e = a * f

```

- Test Case 2: While loop

```
final_input_case2.txt
c = 50
d = 40
g = 100
a = 10
b = 20
while a==b:
    a = a + b
    b = b - a
    print("hello")
    while b==c:
        print("efghi")
        print("abghjk")
        while(b==c):
            print("Happy")
            while(b!=c):
                print("Sad")
                if(a==b):
                    print(d)
            a = b + c
        d = g
    a = b
c = d
a = b + d

```

- Test Case 3: Other statements

final_input_case3.txt

```
import math
a = 50
b = 30
a > b
c = 80
e = 40
c < e
(a+b) == (c+d)
(not b)
not ((a >= b) and a < b)
c = a + b * 5 / (10 % 30)
d = c - b
print("Hello World")
return a
pass
break
a = [1,2,3,4,5]
c = b[10]
b[11] = d
range(10)
range(10,20)

```

- Test Case 4: Optimizations

final_input_case4.txt

```
e = 5
f = 6
d = 7
r = (e + f)*(d - f) # r = 11 (Constant Folding)

a = 1
b = (a * 2) + a     # b = 3  (Constant Propagation)

c = True or False
d = c + b           # c = 1; d = 4 (Constant Propagation)

f = e               # e is not defined

e = 0

e = 2
f = (True and (False or True))
f = f + (d / e)     # f = 3

h = "Hello"
i = h

```

SHORTCOMINGS:

- The compiler design only includes front end phases and optional phases. Further it can be extended to the backend phase.
- Only few constructs like while, if and if-elif-else are used which can be extended to for loop, functions, classes and objects, etc.
- Compilers that we have developed have little intelligence when compared to original compilers with multiple lookaheads.
- Only four simple optimization strategies are implemented which can be extended to multiple complex ones.
- Using functions scope of variables can also be handled in the symbol table.

CONCLUSIONS:

- All 5 phases of the compiler till the machine independent optimization is implemented.
- Theory concepts are very useful and exactly explain the precise way to implement these.
- The compiler design project was a very innovative and useful project from all other projects done in so many semesters.
- Had a lot of things to learn and wonder which would help us in the future in any field we work in.

FUTURE ENHANCEMENTS:

- Extend the project to include backend phases - target code generation and machine dependent optimization.
- Include other python constructs like for loops, functions, classes and objects.
- Multiple optimization techniques which are not implemented here can be implemented to make it more intelligent.
- Further we can include knowledge of assemblers to generate machine level code.
- Finally the same knowledge can be used to develop a compiler for a new language.

REFERENCES / BIBLIOGRAPHIES:

[1] https://www.gnu.org/software/bison/manual/