Report on

"Mini Compiler for Python"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Darshan D	**PES1201801456**
Drishti Hoskote	**PES1201801283**
Ameya Bhamare	**PES1201800351**

Under the guidance of

Ms. Madhura V
Asst. Professor
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 Mini Compiler has been built for the Python Language.
The constructs handled for the python language are if, if-else and for.

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently whereas other languages use punctuation, and it has fewer syntactic constructions than other languages.It provides very high-level dynamic data types and supports dynamic type checking.

A *compiler* is a program that translates a source program written in some high-level programming language into machine code for some computer architecture. The generated machine code can be later executed many times against different data each time.

Sample Input :

test.py

```
b = 3
c = 5
a = b + c
d = b*5
f = a/6
q = a + (b - d) / f * 4
```

Sample Output:

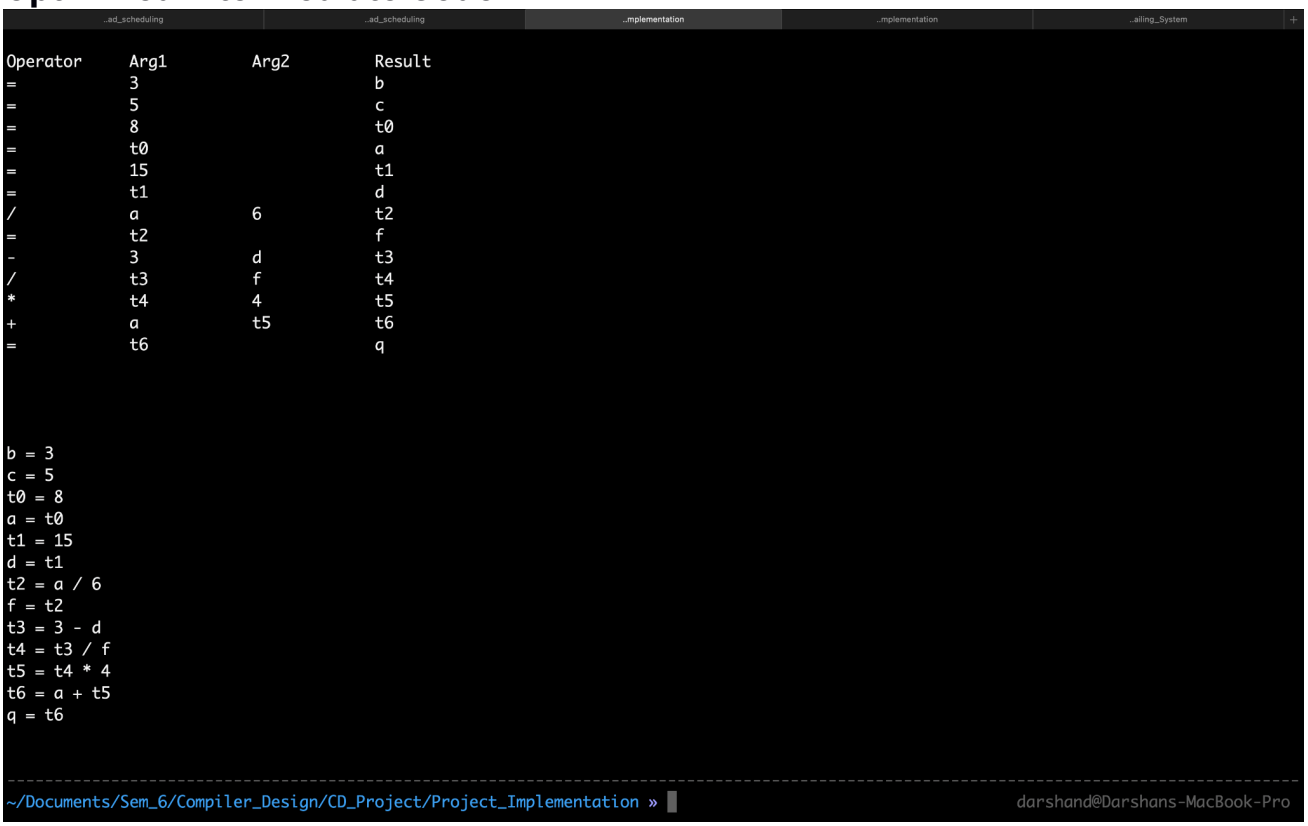
Symbol table:

```
Parsing is successful


SYMBOL TABLE:


Symbol Table Number: 0.0
-------------------------------------------------------------------------------------------------------------
SNo     |   Token       |               Value           |    Line No    |       Column No       |       Scope
-------------------------------------------------------------------------------------------------------------
1           t6                  a+t5                            6                   4           0
2           t3                  b-d                             6                   9           0
3           t4                  t3/f                            6                   8           0
4           t2                  a/6                             5                   4           0
5           t5                  t4*4                            6                   8           0
6           d                   b*5                             4                   0           0
7           a                   b+c                             3                   0           0
8           t1                  b*5                             4                   4           0
9           f                   a/6                             5                   0           0
10          q                   a+(b-d)/f*4                     6                   0           0
11          t0                  b+c                             3                   4           0
12          c                   5                               2                   0           0
13          b                   3                               1                   0           0
```

Intermediate Code:

```
Generated Intermediate Code:

b=3
c=5
t0=b+c
a=t0
t1=b*5
d=t1
t2=a/6
f=t2
t3=b-d
t4=t3/f
t5=t4*4
t6=a+t5
q=t6



Quadruple form of Intermediate Code:

Operator       Arg 1          Arg 2          Result

=              3                             b
=              5                             c
+              b              c              t0
=              t0                            a
*              b              5              t1
=              t1                            d
/              a              6              t2
=              t2                            f
-              b              d              t3
/              t3             f              t4
*              t4             4              t5
+              a              t5             t6
=              t6                            q
```

Optimized Intermediate Code



```
Operator     Arg1         Arg2         Result
=            3                         b
=            5                         c
=            8                         t0
=            t0                        a
=            15                        t1
=            t1                        d
/            a            6            t2
=            t2                        f
-            3            d            t3
/            t3           f            t4
*            t4           4            t5
+            a            t5           t6
=            t6                        q



b = 3
c = 5
t0 = 8
a = t0
t1 = 15
d = t1
t2 = a / 6
f = t2
t3 = 3 - d
t4 = t3 / f
t5 = t4 * 4
t6 = a + t5
q = t6

~/Documents/Sem_6/Compiler_Design/CD_Project/Project_Implementation »        darshand@Darshans-MacBook-Pro
```

ARCHITECTURE

In terms of the syntax and semantics, we have handled the following for the python language:

- Single line and multi line comments are ignored by the lexical analyzer
- Blank lines are ignored by the lexical analyzer
- Import statements have been handled
- Syntax and semantic analysis has been handled to take care of the if, if-else, if-elif-else conditional constructs and the looping construct - for.
- Support has been provided to handle the arithmetic operators like ‘+’, ‘-’, ‘*’, ‘/’ and ‘%’
- Support has been provided to handle the relational operators like ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’ and ‘!=’

- Support has been provided to handle the logical operators - ‘not’, ‘and’, ‘or’. Notably the operators ‘and’ and ‘or’ have been implemented as short circuit operators and the intermediate code generated supports the short circuit functionality as well.

- Support has been provided to recognize keywords and return the appropriate token. Some of the keywords handled includes the following exhaustive list:

    - import
    - True
    - False
    - not
    - and
    - or
    - if
    - else
    - elif
    - for
    - in

Care has also been taken to ensure that keywords cannot be used as identifiers. A syntax error is generated in such a scenario

- Support has been provided to handle the assignment operator - ‘=’

- Support has been provided to handle punctuators and separators like ‘;’, ‘:’, ‘,’, ‘[‘, ‘]’, ‘(‘, ‘)’

- Support has been provided to handle the range function with all its variations possible

- Support has been provided to handle indentation by the use of tabs and spaces. A stack has been used to handle indentation accurately and to generate the Indent and Dedent tokens properly. Care has also been taken to take care of specific cases like generation of required number of Dedent tokens at the end of the file, generation of required number of Dedent tokens when there is a decrease in indentation but there is no set of spaces or tabs to match as such.

- Support has been provided to recognize identifiers properly. The length of the identifiers has been limited to 79 characters for readability purposes as per the conventions that python follows. If the length of an identifier is greater

than 79 characters, then an error message is displayed and the identifier is truncated to the first 79 characters

- Support has been provided to handle numbers and strings accurately
- Support has also been provided to handle numbers written in the scientific notation. In such cases the corresponding floating number is generated. For example, 3.1415E+3 is converted to floating point number 3141.500000 where the default precision after the decimal point is 6 digits.
- Support has been provided to handle precedence and associativity of operators as followed by the python language

LITERATURE SURVEY

- The official grammar as used by Python 3.9 version, provided in the official python documentation: https://docs.python.org/3/reference/grammar.html
- A comprehensive introduction to YACC - https://www.cs.ccu.edu.tw/~naiwei/cs5605/YaccBison.html
- More resources for YACC - https://silcnitc.github.io/yacc.html#yylex
- Python documentation: https://docs.python.org/3/reference/expressions.html
- Information about Lexical analysis for Python as provided by the official python documentation: https://docs.python.org/3/reference/lexical_analysis.html#indentation

CONTEXT FREE GRAMMAR:

The following is the Context Free Grammar used to implement our project:

```
prog_start : input_file;


input_file  : T_Newline input_file
        | statement input_file
        | {;}
;
statement   : simple_statement
        | compound_statement
```

```
;
simple_statement   : small_statement next_simple_statement;

next_simple_statement   : T_Newline
                  | T_semicolon T_Newline
                  | T_semicolon small_statement next_simple_statement
;

small_statement   : expr_statement
             | import_statement
;

import_statement   : T_import T_identifier;

expr_statement   : assignment_statement
             | or_test
;

assignment_statement    : T_identifier T_assignment assignment_expr;

assignment_expr : or_test
            | T_list T_left_par T_right_par
            | T_left_sq_b T_right_sq_b
;

or_test   : or_test T_or and_test
               | and_test
```

```
;

and_test   : and_test T_and not_test
           | not_test
;

not_test   : T_not not_test
       | comparison
;

comparison  : comparison T_LT arith_exp
        | comparison T_GT arith_exp
        | comparison T_EQ arith_exp
        | comparison T_GTE arith_exp
        | comparison T_LTE arith_exp
        | comparison T_NEQ arith_exp
        | comparison T_in arith_exp
        | arith_exp
;

arith_exp   : arith_exp T_plus arith_exp2
            | arith_exp T_minus arith_exp2
            | arith_exp2
;

arith_exp2  : arith_exp2 T_star factor
        | arith_exp2 T_divide factor
```

```
        | arith_exp2 T_modulus factor
        | factor
;

factor  : T_plus factor
        | T_minus factor
        | term
;

term    : T_identifier
        | constant
        | list_index
        | T_left_par or_test T_right_par
;

constant    : T_number
        | T_string
        | T_True
        | T_False
;

list_index  : T_identifier T_left_sq_b or_test T_right_sq_b;

compound_statement  : if_statement
                | for_statement
;
```

```
if_statement    : T_if test T_colon suite elif_statement optional_else;

test    : or_test;

suite   : simple_statement
      | T_Newline T_Indent suite1
;

suite_for   : simple_statement
                  | T_Newline T_Indent suite1
;

suite1  : statement T_Dedent
      | statement repeat_statement T_Dedent
;

repeat_statement    : statement repeat_statement

                | statement
                | T_Newline repeat_statement
                | T_Newline
;

elif_statement  : {;}
            | T_elif test T_colon suite elif_statement
;
```

```
optional_else :   {;}
            | T_else T_colon suite
;

for_statement   : T_for exprlist T_in testlist T_colon suite_for
;

exprlist    : first_exprlist last_exprlist;

first_exprlist  : T_identifier;

last_exprlist   : {;}
            | T_comma
            | T_comma first_exprlist last_exprlist
;

testlist    : range_fn;

range_fn : T_range T_left_par range_term T_right_par

       | T_range T_left_par range_term T_comma range_term T_right_par

       | T_range T_left_par range_term T_comma range_term T_comma
range_term T_right_par
;

range_term  : T_identifier
```

```
        | T_number
        | list_index
;
```

DESIGN STRATEGY

- **SYMBOL TABLE CREATION:**
    - For every different scope value, a separate symbol table is created. Thus we have a symbol table per scope
    - Everytime an identifier is encountered that falls under a particular scope, the symbol table associated with that scope is accessed and a new record for this identifier is created accordingly.
    - Similarly to retrieve values of identifiers on encountering them, the symbol table corresponding to the scope is accessed and the value is retrieved by accessing the record corresponding to this identifier.

- **INTERMEDIATE CODE GENERATION:**
    - Intermediate code generation has been implemented by associating attributes with the non terminals used in the grammar rules.
    - Based on the grammar rule, the equivalent intermediate code is generated by concatenating the different strings received from the attributes. This is then propagated upwards through the attributes associated with the non terminals in the grammar rules
    - In each of these action rules, records are inserted into the quadruple data structure with appropriate information.
    - The symbol table is updated with the temporaries generated

- **CODE OPTIMIZATION**
    - The quadruple data structure created due to the generation of intermediate code is taken as input for the optimization phase. This

phase performs the different optimizations and gives the resulting quadruple data structure as the output.

- The following code optimizations have been performed:
  - Constant folding
    - Expressions that can be evaluated at compile time as the arguments forming the expressions are constants, are evaluated and the resulting value is assigned to the appropriate variable.
    - Algebraic identities are also constant folded by this optimization. Algebraic identities are equations that are always true regardless of the value assigned to the variables
    - Example for algebraic identity constant folding:
      - a + 0 = 0 + a = a
  - Constant propagation
    - If the value of the variable is a constant that is known at compile time, this value is propagated and substituted whenever this variable is encountered.
    - This is usually followed by constant folding
  - Common Subexpression Elimination
    - An occurrence of an expression E is called a common subexpression, if E is previously computed and the values in E have not changed since the previous computation.
    - All such future occurrences of the expression can be eliminated as there is no need to recompute the value of the expression
    - The Variables that are assigned to these future occurrences of the expression are assigned to the temporary that holds the value of the original expression E
  - Strength Reduction
    - Here an expensive operation is replaced by a cheaper operation.

- The cost being talked about here is with respect to the evaluation of the expression by the underlying hardware
- Example:
  - a*2 => a<<1
  - a/2 => a>>1

- **ERROR HANDLING:**

  The following error handling strategies have been used:

  - If a keyword is used as an identifier, an error is prompted to the user
  - If there is mismatch with respect to the indentation, an error is prompted to the user.
  - If the length of an identifier exceeds 79 characters, an error message prompting this is shown to the user and the identifier is truncated to the first 79 characters
  - If an undeclared variable is used as part of an expression, a syntax error is thrown prompting the use of an undeclared variable
  - If an invalid operator like the increment, decrement operator etc is used as part of an expression, a syntax error is thrown

IMPLEMENTATION DETAILS

- **SYMBOL TABLE CREATION:**
  - The symbol table has been implemented as a vector of vector of unordered map from string to a node. A node represents a record in the symbol table and is a class consisting of attributes like identifier name, line number, value associated with the identifier, scope value and the column number.
  - Each vector inside the outermost vector corresponds to a particular scope. Each such vector is a collection of unordered map containers where each container corresponds to a symbol table of a particular scope value

- The unordered map is basically a hashed container where the actual records of the symbol table are stored. The identifier is used to get the hash value and it is mapped to a node.

- The node and symbol table have been implemented as classes with the appropriate attributes, appropriate constructors to initialize the members and functions to provide the functionality of inserting into the symbol table and a few other helper functions

- **INTERMEDIATE CODE GENERATION:**

  - First the union corresponding to yylval is modified to aid in the generation of intermediate code.

  - The union has the following 4 fields:
    - indentation level of type int: This is used to keep track of the indentation and increase or decrease the scope count accordingly
    - text of type char* : All the tokens and non terminals that have a char * value associated with it, use this field
    - structure inter_code having the following 4 fields:
      - char *addr: This is used to store the name of the temporary created as part of the action corresponding to a grammar rule
      - char *code: This is used to store the intermediate code generated as part of the action corresponding to a grammar rule
      - char *true_l: This is used to store the label created that corresponds to the true case
      - char *false_l: This is used to store the label created that corresponds to the false case
    - structure range_icg having the following 4 fields:
      - char *start_r: This is used to store the start value corresponding to the range function
      - char *end_r: This is used to store the end value corresponding to the range function
      - char *step_r: This is used to store the step value corresponding to the range function
      - char *sym_tab_info: This is used to store the information that will be used to update the symbol table

  - The intermediate code generation is done by using a vector of strings inside each action corresponding to a grammar rule. The strings are

then concatenated to get one string, which is assigned to the attribute code and propagated upwards. The other fields like addr, true_l and false_l are also appropriately filled and propagated up the grammar rules. This simulates the synthesized attributes concept learnt by us in Compiler Design class.

- A doubly linked list is used to implement the quadruple data structure which is used to store the intermediate code generated. This doubly linked list is essentially a collection of nodes where each node corresponds to one record in the quadruple data structure. A node of the quadruple data structure has the following attributes:
  - char *op: This field stores the operator
  - char *arg1: This field stores the argument 1 value
  - char *arg2: This field stores the argument 2 value
  - char *res: This field stores the variable or temporary that holds the result
  - node *next: This field stores a pointer to the next record in the doubly linked list. Stores NULL if it is the last node in the list.
  - node *prev: This field stores a pointer to the previous record in the doubly linked list. Stores NULL if it is the first node in the list.

- **CODE OPTIMIZATION:**

  - The code optimization phase takes the quadruple data structure as input and provides the optimized quadruple data structure as the output.

  - The input quadruple data structure is stored in a list of dictionaries. Each element of this list (dictionary) corresponds to a record in the quadruple data structure. The dictionary has fields like ‘op’, ‘arg1’, ‘arg2’ and ‘res’ to store the appropriate attributes from the records of the quadruple data structure

  - Constant folding:
    - Uses the list of dictionaries created.
    - Traverses them and performs folding on encountering constants as arguments

  - Constant propagation:
    - Uses the list of dictionaries created.
    - Traverses them and performs constant propagation when a favourable case is encountered.

        - This is usually followed by constant folding

    - Common Subexpression Elimination:
        - Uses the list of dictionaries created
        - Creates a deep copy of this list to work with while traversing the actual list. This is because it is unsafe to remove records from an iterable while iterating through it

    - Strength Reduction:
        - Uses the list of dictionaries created

- **ERROR HANDLING:**

    - Everytime an error is encountered, the error message that is prompted to the user contains the exact line number and the column number where the error has occurred. This is achieved by using the first_line, first_column, last_line, last_column fields associated with yylloc.

    - To retrieve the value of an identifier, the symbol table associated with that scope is checked to see if the identifier has been previously declared. If not, an error is prompted.

    - yyleng is used to keep track of the length of the string that has matched the regular expression mentioned in the lex file. This is used to check if the length of an identifier exceeds 79 characters or not. If it does, the identifier is truncated to the first 79 characters

- **Instructions on how to build and run our program**

    - First the yacc utility is used to compile the yacc file: **yacc -d -t --verbose test1_y.y**
        - Here -d is used to generate the y.tab.h file which is used by the lex file
        - Here -t is used to track the generation of tokens
        - Here --verbose is used to generate the y.output file

    - Next the lex utility is used to compile the lex file: **lex test1_l.l**

    - The generated files are linked along with helper cpp program: **g++ -std=c++11 lex.yy.c y.tab.c test1.cpp -ll -ly -w**

    - The executable is run with the required input: **./a.out < test.py**

- The intermediate code and the quadruple data structure generated are stored in separate text files. This is used by the optimisations python program which is run using the following command: **python3 optimisations.py**

RESULTS AND POSSIBLE SHORTCOMINGS:

- All the different cases mentioned in the Architecture section have been handled and the working of the mini - compiler is as expected for these cases
- The basic constructs such as if, if-else, if-elif-else and for have been handled
- The Intermediate code generation is quite comprehensive with respect to the different cases it can handle accurately
- The optimization phase is quite comprehensive as well, with respect to the different optimizations it can perform
- The mini-compiler built by us is obviously not complete and can't be used as an alternative to the original python compiler and interpreter. This is because the number of constructs, different variations and cases we could handle were restricted due to the time constraints involved.
- There are quite a few cases for which the original python compiler would work but our mini compiler wouldn't work
- All the constructs and expressions used can only be compile time dependent. Run time dependency couldn't be introduced as support for input from the user has not been provided.

SNAPSHOTS

Test cases to show cases handled by Intermediate Code Generation:

1. **Test Case 1:** Test case involves arithmetic expressions. The intermediate code generated follows the rule of precedence and associativity as followed by python

```
b = 3
c = 5
a = b + c
d = b*5
f = a/6
q = a + b - d / f * 4
```

Symbol Table:

```
Parsing is successful


SYMBOL TABLE:


Symbol Table Number: 0.0
-------------------------------------------------------------------------------------------------------------
SNo     |   Token       |              Value           |    Line No    |       Column No       |       Scope
-------------------------------------------------------------------------------------------------------------
1            t6                t3-t5                          6                4           0
2            t3                a+b                            6                4           0
3            t4                d/f                            6                12          0
4            t2                a/6                            5                4           0
5            t5                t4*4                           6                12          0
6            d                 b*5                            4                0           0
7            a                 b+c                            3                0           0
8            t1                b*5                            4                4           0
9            f                 a/6                            5                0           0
10           q                 a+b-d/f*4                      6                0           0
11           t0                b+c                            3                4           0
12           c                 5                              2                0           0
13           b                 3                              1                0           0
```

Intermediate Code:

```
..ad_scheduling    ..ad_scheduling    ..mplementa

Generated Intermediate Code:

b=3
c=5
t0=b+c
a=t0
t1=b*5
d=t1
t2=a/6
f=t2
t3=a+b
t4=d/f
t5=t4*4
t6=t3-t5
q=t6



Quadruple form of Intermediate Code:

Operator       Arg 1          Arg 2          Result

=              3                             b
=              5                             c
+              b              c              t0
=              t0                            a
*              b              5              t1
=              t1                            d
/              a              6              t2
=              t2                            f
+              a              b              t3
/              d              f              t4
*              t4             4              t5
-              t3             t5             t6
=              t6                            q
```

2. **Test Case 2:** Test case to show intermediate code generation for if, elif and else constructs

```
a = 5
if a>5:
   b = 6
elif a<9:
   b = 7
elif a == 4:
   b = 9
else:
   b = 10
```

Symbol table:

```
..ad_scheduling  ..ad_scheduling  ..mplementation  ..mplementation
SYMBOL TABLE:


Symbol Table Number: 0.0
-------------------------------------------------------------------------------------------------------------
SNo     |   Token       |             Value          |    Line No    |       Column No       |       Scope
-------------------------------------------------------------------------------------------------------------
1           t1              a<9                          4                9               0
2           t2              a==4                         6                9               0
3           t0              a>5                          2                3               0
4           a               5                            1                0               0


Symbol Table Number: 1.0
-------------------------------------------------------------------------------------------------------------
SNo     |   Token       |             Value          |    Line No    |       Column No       |       Scope
-------------------------------------------------------------------------------------------------------------
1           b               6                            3                4               1

Symbol Table Number: 1.1
-------------------------------------------------------------------------------------------------------------
SNo     |   Token       |             Value          |    Line No    |       Column No       |       Scope
-------------------------------------------------------------------------------------------------------------
1           b               7                            5                4               1

Symbol Table Number: 1.2
-------------------------------------------------------------------------------------------------------------
SNo     |   Token       |             Value          |    Line No    |       Column No       |       Scope
-------------------------------------------------------------------------------------------------------------
1           b               9                            7                4               1

Symbol Table Number: 1.3
-------------------------------------------------------------------------------------------------------------
SNo     |   Token       |             Value          |    Line No    |       Column No       |       Scope
-------------------------------------------------------------------------------------------------------------
1           b               10                           9                4               1
```

Intermediate Code Generated:

```
Generated Intermediate Code:

a=5
t0=a>5
ift0gotoL0
gotoL1
L0:
b=6
goto L6
L1:
t1=a<9
ift1gotoL2
gotoL3
L2:
b=7
goto L6
L3:
t2=a==4
ift2gotoL4
gotoL5
L4:
b=9
goto L6
L5:
b=10
L6:
```

Intermediate Code in Quadruple Format:

```
Quadruple form of Intermediate Code:

Operator       Arg 1          Arg 2          Result

=              5                             a
>              a              5              t0
Label                                        L0
if             t0                            L0
Label                                        L1
goto                                         L1
=              6                             b
<              a              9              t1
Label                                        L2
if             t1                            L2
Label                                        L3
goto                                         L3
=              7                             b
==             a              4              t2
Label                                        L4
if             t2                            L4
Label                                        L5
goto                                         L5
=              9                             b
=              10                            b
Label                                        L6
goto                                         L6
```

3. **Test Case 3:** Test Case to show intermediate code generation for the ‘for’ looping construct

```
a = 8
for i in range (5, 10, 2):
        b = 6
```

Symbol Table:

```
Parsing is successful


SYMBOL TABLE:


Symbol Table Number: 0.0
-------------------------------------------------------------------------------------------------------------
SNo     |   Token       |               Value           |    Line No    |       Column No       |       Scope
-------------------------------------------------------------------------------------------------------------
1            a                8                               1                0            0



Symbol Table Number: 1.0
-------------------------------------------------------------------------------------------------------------
SNo     |   Token       |               Value           |    Line No    |       Column No       |       Scope
-------------------------------------------------------------------------------------------------------------
1            i                5                               2                4            1
2            t1               i<10                            2                0            1
3            t0               i+2                             2                0            1
4            b                6                               3                4            1
```

Intermediate Code Generated:

```
Generated Intermediate Code:

a=8
i=5
L0:
b=6
t0 = i + 2
i=t0
t1=i < 10
if t1 goto L0
goto L1
L1:



Quadruple form of Intermediate Code:

Operator       Arg 1          Arg 2          Result

=              8                             a
=              6                             b
=              5                             i
+              i              2              t0
=              t0                            i
<              i              10             t1
Label                                        L0
if             t0                            L0
Label                                        L1
goto                                         L1
```

Test cases to show the Optimizations performed:

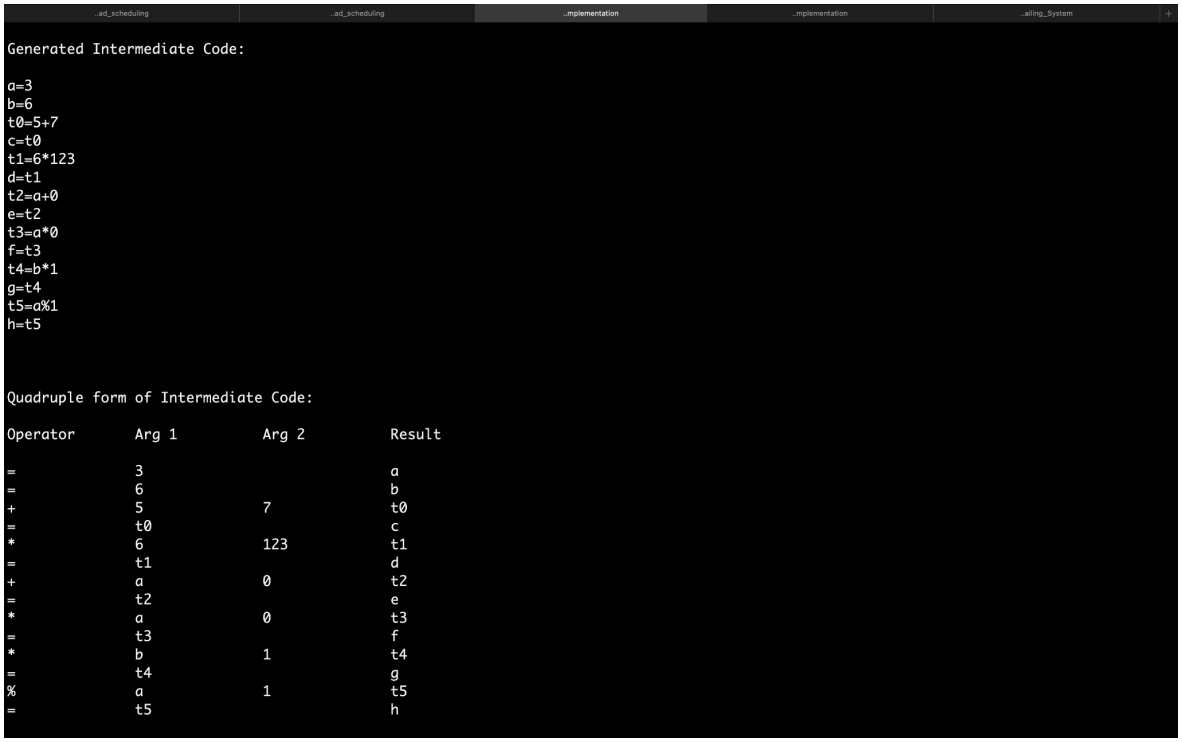
1. **Test Case 1: Constant Folding Optimization**

a = 3
b = 6
c = 5 + 7
d = 6*123
e = a + 0
f = a * 0
g = b*1
h = a%1

Symbol Table:

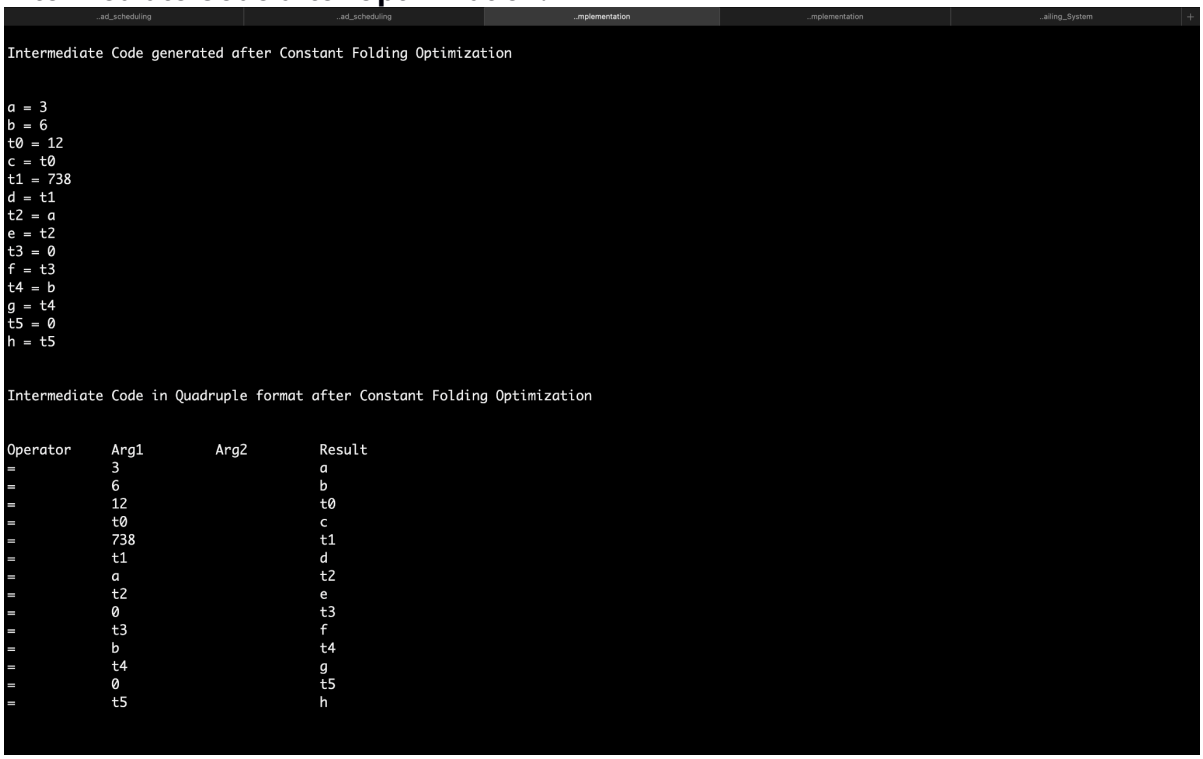
```
Parsing is successful


SYMBOL TABLE:


Symbol Table Number: 0.0
-------------------------------------------------------------------------------------------------------------
SNo     |   Token       |               Value           |    Line No    |       Column No       |       Scope
-------------------------------------------------------------------------------------------------------------
1           g               b*1                             7               0           0
2           t3              a*0                             6               4           0
3           h               a%1                             8               0           0
4           t4              b*1                             7               4           0
5           t2              a+0                             5               4           0
6           t5              a%1                             8               4           0
7           e               a+0                             5               0           0
8           d               6*123                           4               0           0
9           a               3                               1               0           0
10          t1              6*123                           4               4           0
11          f               a*0                             6               0           0
12          c               5+7                             3               0           0
13          t0              5+7                             3               4           0
14          b               6                               2               0           0
```

Intermediate Code Generated (Before Optimization):



```
Generated Intermediate Code:

a=3
b=6
t0=5+7
c=t0
t1=6*123
d=t1
t2=a+0
e=t2
t3=a*0
f=t3
t4=b*1
g=t4
t5=a%1
h=t5



Quadruple form of Intermediate Code:

Operator       Arg 1          Arg 2          Result

=              3                             a
=              6                             b
+              5              7              t0
=              t0                            c
*              6              123            t1
=              t1                            d
+              a              0              t2
=              t2                            e
*              a              0              t3
=              t3                            f
*              b              1              t4
=              t4                            g
%              a              1              t5
=              t5                            h
```

Intermediate Code after Optimization:



```
Intermediate Code generated after Constant Folding Optimization


a = 3
b = 6
t0 = 12
c = t0
t1 = 738
d = t1
t2 = a
e = t2
t3 = 0
f = t3
t4 = b
g = t4
t5 = 0
h = t5


Intermediate Code in Quadruple format after Constant Folding Optimization


Operator     Arg1         Arg2         Result
=            3                         a
=            6                         b
=            12                        t0
=            t0                        c
=            738                       t1
=            t1                        d
=            a                         t2
=            t2                        e
=            0                         t3
=            t3                        f
=            b                         t4
=            t4                        g
=            0                         t5
=            t5                        h
```

2. Test Case 2: Constant Propagation Optimization

a = 5
b = 6
c = a + b
d = a*b
e = b - a
f = b/a

Symbol Table:

```
Parsing is successful


SYMBOL TABLE:


Symbol Table Number: 0.0
-------------------------------------------------------------------------------------------------------------
SNo     |   Token       |              Value            |    Line No    |       Column No       |       Scope
-------------------------------------------------------------------------------------------------------------
1             t3                b/a                             6               4             0
2             t2                b-a                             5               4             0
3             d                 a*b                             4               0             0
4             a                 5                               1               0             0
5             t1                a*b                             4               4             0
6             f                 b/a                             6               0             0
7             e                 b-a                             5               0             0
8             c                 a+b                             3               0             0
9             t0                a+b                             3               4             0
10            b                 6                               2               0             0
```

Intermediate Code Generated (Before Optimization):

```
Generated Intermediate Code:

a=5
b=6
t0=a+b
c=t0
t1=a*b
d=t1
t2=b-a
e=t2
t3=b/a
f=t3


Quadruple form of Intermediate Code:

Operator       Arg 1          Arg 2          Result

=              5                             a
=              6                             b
+              a              b              t0
=              t0                            c
*              a              b              t1
=              t1                            d
-              b              a              t2
=              t2                            e
/              b              a              t3
=              t3                            f
```

Intermediate Code after Optimization:

```
Intermediate Code generated after Constant Propagation Optimization


a = 5
b = 6
t0 = 11
c = t0
t1 = 30
d = t1
t2 = 1
e = t2
t3 = 1.2
f = t3


Intermediate Code in Quadruple format after Constant Propagation Optimization


Operator     Arg1         Arg2         Result
=            5                         a
=            6                         b
=            11                        t0
=            t0                        c
=            30                        t1
=            t1                        d
=            1                         t2
=            t2                        e
=            1.2                       t3
=            t3                        f
```

3. Test Case 3: Common Subexpression Elimination Optimization

a = 2
b = 3
c = a/b
q = 0
r = a/b
we = 123
eiur = 342
dsf = 3242
fg = a/b

Symbol Table:

```
Parsing is successful


SYMBOL TABLE:


Symbol Table Number: 0.0
-------------------------------------------------------------------------------------------------------------
SNo     |   Token       |              Value           |    Line No    |       Column No       |       Scope
-------------------------------------------------------------------------------------------------------------
1            fg               a/b                            9                0            0
2            b                3                              2                0            0
3            eiur             342                            7                0            0
4            dsf              3242                           8                0            0
5            we               123                            6                0            0
6            t2               a/b                            9                5            0
7            t1               a/b                            5                4            0
8            r                a/b                            5                0            0
9            t0               a/b                            3                4            0
10           q                0                              4                0            0
11           c                a/b                            3                0            0
12           a                2                              1                0            0
```

Intermediate Code Generated (Before Optimization):

```
Generated Intermediate Code:

a=2
b=3
t0=a/b
c=t0
q=0
t1=a/b
r=t1
we=123
eiur=342
dsf=3242
t2=a/b
fg=t2


Quadruple form of Intermediate Code:

Operator       Arg 1          Arg 2          Result

=              2                             a
=              3                             b
/              a              b              t0
=              t0                            c
=              0                             q
/              a              b              t1
=              t1                            r
=              123                           we
=              342                           eiur
=              3242                          dsf
/              a              b              t2
=              t2                            fg
```

Intermediate Code after Optimization:

```
Intermediate Code generated after Common Subexpression Elimination Optimization


a = 2
b = 3
t0 = a / b
c = t0
q = 0
r = t0
we = 123
eiur = 342
dsf = 3242
fg = t0


Intermediate Code in Quadruple format after Common Subexpression Elimination Optimization


Operator     Arg1         Arg2         Result
=            2                         a
=            3                         b
/            a            b            t0
=            t0                        c
=            0                         q
=            t0                        r
=            123                       we
=            342                       eiur
=            3242                      dsf
=            t0                        fg
```

4. Test Case 4: Strength Reduction Optimization

a = 3
b = 5
c = a*2

d = a*64
e = b/128

Symbol Table:

```
Parsing is successful


SYMBOL TABLE:


Symbol Table Number: 0.0
-------------------------------------------------------------------------------------------------------------
SNo     |   Token       |               Value           |    Line No    |       Column No       |       Scope
-------------------------------------------------------------------------------------------------------------
1             t2                b/128                           5                4             0
2             d                 a*64                            4                0             0
3             a                 3                               1                0             0
4             t1                a*64                            4                4             0
5             e                 b/128                           5                0             0
6             c                 a*2                             3                0             0
7             t0                a*2                             3                4             0
8             b                 5                               2                0             0
```

Intermediate Code Generated (Before Optimization):

```
Generated Intermediate Code:

a=3
b=5
t0=a*2
c=t0
t1=a*64
d=t1
t2=b/128
e=t2


Quadruple form of Intermediate Code:

Operator       Arg 1          Arg 2          Result

=              3                             a
=              5                             b
*              a              2              t0
=              t0                            c
*              a              64             t1
=              t1                            d
/              b              128            t2
=              t2                            e
```

Intermediate Code after Optimization:

```
Intermediate Code generated after Strength Reduction Optimization


a = 3
b = 5
t0 = a << 1
c = t0
t1 = a << 6
d = t1
t2 = b >> 7
e = t2


Intermediate Code in Quadruple format after Strength Reduction Optimization


Operator     Arg1         Arg2         Result

=            3                         a
=            5                         b
<<           a            1            t0
=            t0                        c
<<           a            6            t1
=            t1                        d
>>           b            7            t2
=            t2                        e
```

CONCLUSION

By means of this project, we have learnt how to build a compiler for a high level language like Python. Being limited by the duration of the semester, we did not build a full fledged compiler. We handled a list of constructs like if, else and for. Besides, we generated Intermediate Code in Quadruple format and four optimizations were implemented.

We would like to thank the Dept. of CSE for giving us this opportunity to implement our own compiler. This gave us a feel for how compilers run any code that we write. Special thanks to Prof. Madhura V for her insightful lectures and project guidance from time to time.

FUTURE ENHANCEMENTS

- We would like to handle more constructs that python supports like the while and do while constructs, switch construct, provide support for functions, classes.
- We would also like to support a few more data types specific to python like sets, dictionaries, tuples etc.

REFERENCES / BIBLIOGRAPHY

- A lot of the concepts have been learnt from the resources provided for the Compiler Design Course by PES University
- The material provided explicitly for the CD project with respect to the Lex / YACC learning material has been quite useful in learning the basics
- The official Python documentation has been our primary resource to refer to all things about Python
- A few references used are as follows:
    - The official grammar as used by Python 3.9 version, provided in the official python documentation: https://docs.python.org/3/reference/grammar.html
    - A comprehensive introduction to YACC - https://www.cs.ccu.edu.tw/~naiwei/cs5605/YaccBison.html
    - More resources for YACC - https://silcnitc.github.io/yacc.html#yylex
    - Python documentation: https://docs.python.org/3/reference/expressions.html
    - Information about Lexical analysis for Python as provided by the official python documentation: https://docs.python.org/3/reference/lexical_analysis.html#indentation