Report on

Mini Compiler for If-Else and While Constructs in Python

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

Compiler Design

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Prathik B Jain	**PES2201800058**
Sathvik Saya	**PES2201800684**
Supreeth Ronad	**PES2201800705**

Under the guidance of

Prof. Swati
Assistant Professor
PES University, Bengaluru

January – May 2021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

FACULTY OF ENGINEERING
PES UNIVERSITY

TABLE OF CONTENTS

INTRODUCTION

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses.

A Mini Compiler is Built for Python and handles the If-Elif-Else and the While Constructs using c programming language. Lex and Flex are tools for generating scanners: programs which recognize lexical patterns in text. Flex is a faster version of Lex. Lex/Flex refers to either of the tools. Yacc and Bison are tools for generating parsers: programs which recognize the grammatical structure of programs. Bison is a faster version of Yacc.

The optimizer applies semantics preserving transformations to the annotated parse tree to simplify the structure of the tree and to facilitate the generation of more efficient code. The code generator transforms the simplified annotated parse tree into object code using rules which denote the semantics of the source language. The code generator may be integrated with the parser.

ARCHITECTURE OF THE LANGUAGE

Python has a very flexible syntax and we have tried to incorporate as much as possible from our experience of using python into the grammar. All lines of code terminate upon seeing a newline character. We have taken care of the following:

1. If-Elif-Else and While constructs
2. Print Statements
3. pass, break and void returns
4. Function definitions and Calls
5. Lists
6. All arithmetic operators and all boolean operators
7. Single Line Comments (#)

Semantically we have checked the following :

- Whether any variable used on the RHS is defined and in the current scope or any Enclosing Scope of the current scope.
- Whether a variable being indexed is List
- Whether all expressions in the If and While Clauses are Boolean expressions

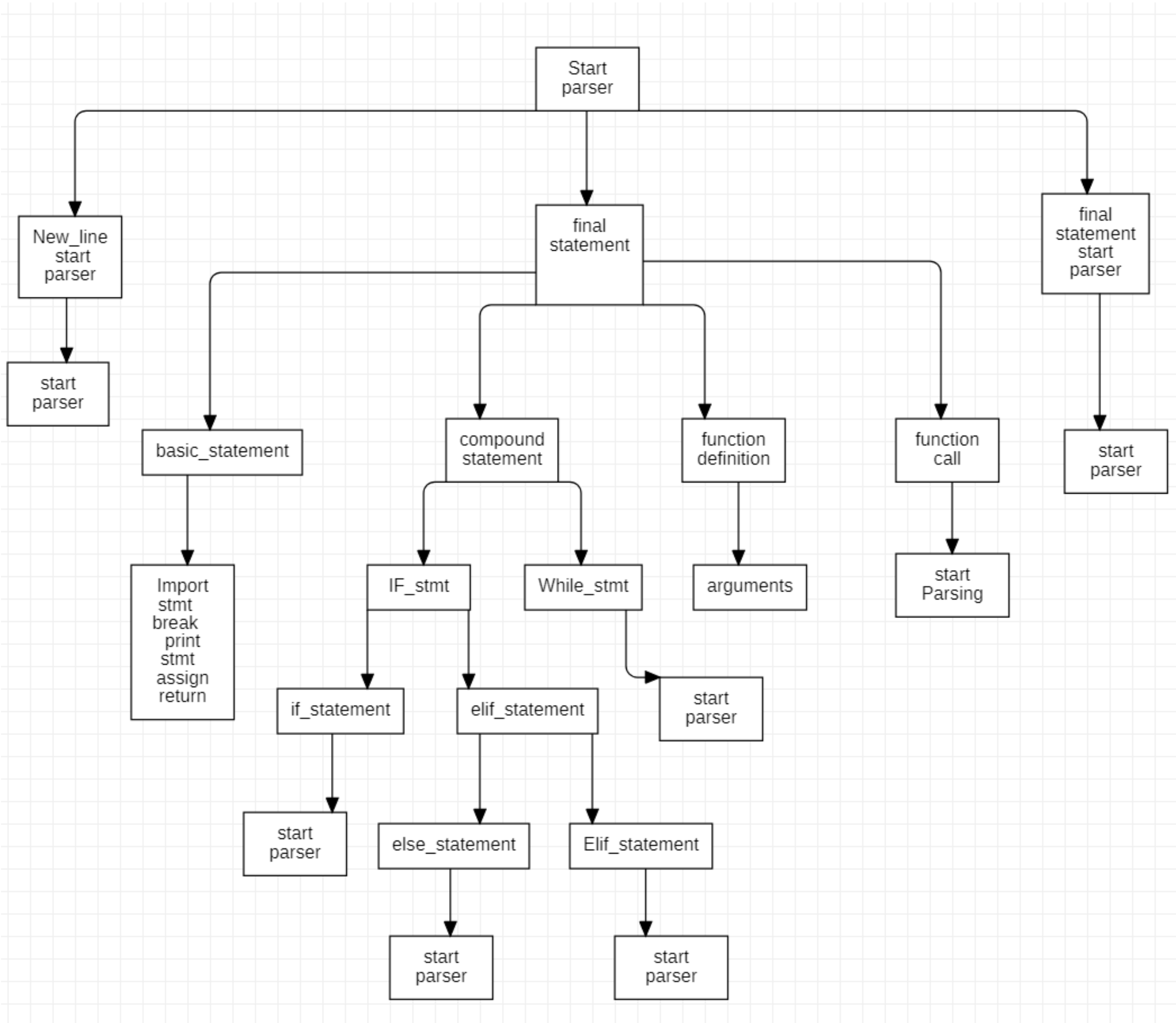
LITERATURE SURVEY

1. Lex and Yacc Doc by Tom Niemann
2. Official Bison Documentation : https://www.gnu.org/software/bison/manual
3. Stackoverflow : https://stackoverflow.com
4. Compiler Construction using Flex and Bison - Anthony A. Aaby, Walla Walla College, cs.wwc.edu, aabyan@wwc.edu, Version of February 25, 2004

THE CONTEXT FREE GRAMMAR

<table>
<tr><th>Grammar</th><th></th><th>Expression</th><th>Legend</th></tr>
<tr><td>digit</td><td>→</td><td>[0-9]</td><td>CLN : Colon (' : ')</td></tr>
<tr><td>constant</td><td>→</td><td>[digit]+ | QUOTE [_a-zA-Z]+ QUOTE</td><td>NL : New Line (' \n ')</td></tr>
<tr><td>variable</td><td>→</td><td>[_a-zA-Z][_a-zA-Z0-9]*</td><td>EQL : Equal (' = ')</td></tr>
<tr><td>term</td><td>→</td><td>variable | constant</td><td>L_Paren : '('</td></tr>
<tr><td></td><td></td><td></td><td>R_Paren : ')'</td></tr>
<tr><td>bool_exp</td><td>→</td><td>True | False | arith_exp ROP arith_exp</td><td>ROP : Relational Operators ('<', '>', '==' ..)</td></tr>
<tr><td></td><td></td><td>bool_exp or bool_exp | bool_exp and bool_exp |</td><td>ID : Indent</td></tr>
<tr><td></td><td></td><td>not bool_exp |</td><td>DD : Dedent</td></tr>
<tr><td></td><td></td><td>L_Paren bool_exp R_Paren</td><td>QUOTE : "</td></tr>
<tr><td>arith_exp</td><td>→</td><td>term | L_Paren arith_exp R_Paren | arith_exp + arith_exp |</td><td>STR : Any String</td></tr>
<tr><td></td><td></td><td>arith_exp - arith_exp |</td><td>Trip_Quote : '''</td></tr>
<tr><td></td><td></td><td>arith_exp * arith_exp |</td><td>HASH : #</td></tr>
<tr><td></td><td></td><td>arith_exp / arith_exp</td><td></td></tr>
<tr><td>assign_stmt</td><td>→</td><td>variable EQL term</td><td></td></tr>
<tr><td>pass_stmt</td><td>→</td><td>"pass"</td><td></td></tr>
<tr><td>import_stmt</td><td>→</td><td>"import" <pkg_name></td><td></td></tr>
<tr><td>expression_stmt</td><td>→</td><td>bool_exp | arith_exp</td><td></td></tr>
<tr><td>basic_stmt</td><td>→</td><td>expression_stmt | assign_stmt | pass_stmt</td><td></td></tr>
<tr><td>stmt_list</td><td>→</td><td>basic_stmt NL stmt_list | basic stmt</td><td></td></tr>
<tr><td>suite</td><td>→</td><td>stmt_list NL | NL ID stmt + DD</td><td></td></tr>
<tr><td>stmt</td><td>→</td><td>basic_stmt NL | cmpd_stmt</td><td></td></tr>
<tr><td>cmpd_stmt</td><td>→</td><td>if_stmt | while_stmt</td><td></td></tr>
<tr><td>while_stmt</td><td>→</td><td>"while" bool_exp CLN NL ID stmt_list DD</td><td></td></tr>
<tr><td>if_stmt</td><td>→</td><td>"if" bool_exp CLN suite ("elif" bool_exp CLN suite)* ["else" CLN suite]</td><td></td></tr>
<tr><td>comment</td><td>→</td><td>HASH STR | Trip_Quote STR multiline_comment</td><td></td></tr>
<tr><td>multiline_comment</td><td>→</td><td>Trip_Quote | STR NL multiline_comment</td><td></td></tr>
</table>

THE CFG Flow Diagram



DESIGN STRATEGY

First and foremost, every connection points back to the Symbol Table, as per our Design Strategy. This ensures that the Symbol table is linked to the necessary nodes of the Abstract Syntax tree and the required Quadruples in the Intermediate code. The symbol table also contains all of the Temporaries provided by the compiler.

As you can see in the sample output, the Symbol Table stores 'Records' with four columns, namely

1. Scope: Scope of each variable contained in record
2. Name: Value/Name of each variable contained in record.
3. Type : Type of each variable contained in record
    i. PackageName
    ii. Func_Name
    iii. Identifier
    iv. Constant

v. ListTypeID
vi. ICGTempVar
vii. ICGTempLabel

4. Declaration : Line of Declaration of each variable contained in record
5. Last Use Line

The scope is a property of indentation depth, and we have a tuple of the parent's scope and the current scope measured using the indentation depth to make it distinct.

There are two types of nodes in the Abstract Syntax Tree: Leaf nodes and Internal nodes. Depending on the construct it serves, the nodes will have a variable number of children (0-3). Take the If-Else Statement for example.

If
Condition CodeBlock Else

We take the AST and store it as a matrix of levels in order to represent it. Each stage of the AST has been printed, as seen in the sample output. In addition, next to each Internal node is a number enclosed in brackets that represents the number of children they have in the next step. The identifiers, constants, Lists, and packages leaf nodes in the AST point to a symbol table record.

Recursively stepping through the AST generates the intermediate code. Each line is saved as a quadruple (Operation, Arg1, Arg2, Result) so that code can be quickly optimised in the steps that follow.

In the whole code, we remove dead code, explicitly unused variables.For example, if we have the following lines of code:

a = 10
b = 10
c = a + b

And these 3 variables are not used on any other RHS, then these 3 lines of code are Eliminated during optimization. All the dead variables in the code are removed. We iterate through the Quads to do this.

We introduced a stack and used three tokens to take control of the indentation-based code structure and scoping. The top of the stack always points to the current indentation value; if that value doesn't adjust when scanning the next line, it means we're in the same scope, and we return the token ND, which stands for "No-dent." If the value increases, we are entering a sub-scope, and we return the token ‘ID,' which stands for ‘Indent,' and if the value reduces, we are returning to one of the enclosing scopes, and we return ‘DD,' which stands for ‘Dedent.'

Whenever the parser encounters an error, it prints the line number and column number of the error. We also display the following errors:

- Identifier <var> Not declared in scope
- Identifier <var> Not Indexable

All Comments are removed from the code before parsing.

IMPLEMENTATION DETAILS

The Symbol table uses two Structures,

```
typedef struct record {
	char *type;
	char *name;
	int decLineNo;
	int lastUseLine;
} record;

typedef struct STable {
	int no;
	int noOfElems;
	 int scope;
	record *Elements;
	int Parent;
} STable;
```

The “record” arrangement reflects each record in the symbol table. Each symbol table will have a maximum of “MAXRECST” documents, MAXRECST is a macro. One symbol table is represented by the “STable” arrangement. For each scope, a new Symbol table is created. MAXST is a macro that allows a maximum of “MAXST” symbol tables and hence scopes to occur.The Abstract Syntax Tree uses one structure,

```
typedef struct ASTNode {
	int nodeNo;
	/*Operator*/
	char *NType;
	int noOps;
	struct ASTNode** NextLevel;
	 /*Identifier or Const*/  record *id;
} node;
```

This ASTNode arrangement looks after all leaf nodes and internal "operator" nodes. Based on the form of the node, the appropriate values are set.

Each node may have a maximum of three children. The AST is printed by storing it in a Matrix of Order "MAXLEVELS" x "MAXCHILDREN" and then printing the matrix Levelwise. This Matrix is a pointer matrix to the AST. The Node's "noOps" element specifies the number of children it has.

The Three-Address Code is represented and stored as Quads that are given by the structure,

```
typedef struct Quad {
	char *R;
	char *A1;
	char *A2;
	char *Op;
	int I;
} Quad;
```

During code optimization, the last element, the integer 'I,' is used. In an array of Quads, all of the Three-Address codes are stored as Quads. A limit of "MAXQUADS" quadruples can be used.

We loop around the code until there is no more code that can be removed for dead code removal. To see if a quadruple has dead code, look to see if the result parameter/element of that quad exists as an argument to all other quads that haven't been deleted. If not, we consider the quad to be dead code and assign the value "-1" to the variable "I".

Scope checking is accomplished by recursively stepping through enclosing scopes and locating the variable's most recent definition. If no definition is found, the error is printed.

If you wish to run the code,
Install flex and bison using the below commands(if not installed),

```
sudo apt-get upgrade
sudo apt-get install bison flex
```

```
lex grammar.l
yacc -dv grammar.y
gcc lex.yy.c y.tab.c -g -ll -o Test.out
./Test.out < InputFile.txt
```

RESULTS AND SHORTCOMINGS

We now have a mini compiler that parses grammar corresponding to basic Python syntax and produces an efficient intermediate representation as a result. Below are some of the places where our mini compiler fell short:

- Just one very simple optimization is used, and it does not result in a significant reduction in code density.
- There are a few memory leaks in the programme, but the majority of them have been fixed.

SNAPSHOTS

1. TestCase_1

1.1 Input File

```
pb@ubuntu:~/Downloads/Mini-Python-Compiler-in-Lex-and-Yacc-master$ cat input3.txt
#Basic Code

import scipy

x = 2
y = 1

a = 3
b = 4
d = a+b

if(x==1):
        c=1
elif(y==1):
        c=2
else:
        c=1

pb@ubuntu:~/Downloads/Mini-Python-Compiler-in-Lex-and-Yacc-master$ 
```

1.2 Token Sequence

```
---------------------------------Token Sequence--------------------------
1 T_NL
2 T_NL
3 T_IMPT T_scipy T_NL
4 T_NL
5 T_x T_EQL T_2 T_NL
6 T_y T_EQL T_1 T_NL
7 T_NL
8 T_a T_EQL T_3 T_NL
9 T_b T_EQL T_4 T_NL
10 T_d T_EQL T_a T_PL T_b T_NL
11 T_NL
12 T_If T_OP T_x T_EQ T_1 T_CP T_Cln T_NL
13 T_ID T_c T_EQL T_1 T_NL
14 T_Elif T_OP T_y T_EQ T_1 T_CP T_Cln T_NL
15 T_ID T_c T_EQL T_2 T_NL
16 T_Else T_Cln T_NL
17 T_ID T_c T_EQL T_1 T_NL
18 T_NL
19 T_EOF

  $$ PARSING COMPLETED $$
```

1.3 Abstract Syntax Tree

```
------------------------Abstract Syntax Tree------------------------
 NewLine(2)
 import(1)   NewLine(2)
 scipy    =(2)   NewLine(2)
      x    2    =(2)   NewLine(2)
        y    1    =(2)   NewLine(2)
          a    3    =(2)   NewLine(2)
            b    4    =(2)   If(3)
              d    +(2)   ==(2)   BeginBlock(2)   Elif(3)
               a    b    x    1    =(2)   EndBlock    ==(2)   BeginBlock(2)   Else(1)
                              c    1    y    1    =(2)   EndBlock    BeginBlock(2)
                                          c    2    =(2)   EndBlock
                                                    c    1
```

1.4 Optimized Quads

```
---------------------------------All Quads---------------------------------
0       import  scipy   -       -
1       =       2       -       T2
2       =       T2      -       x
3       =       1       -       T5
4       =       T5      -       y
13      =       x       -       T19
14      =       1       -       T20
15      ==      T19     T20     T21
16      If False        T21     -       L0
19      goto    -       -       L1
20      Label   -       -       L0
21      =       y       -       T27
22      =       1       -       T28
23      ==      T27     T28     T29
24      If False        T29     -       L0
27      goto    -       -       L1
28      Label   -       -       L0
31      Label   -       -       L1
32      Label   -       -       L1
----------------------------------------------------------------------------
```

1.5 Intermediate Code

```
import scipy
T2 = 2
x = T2
T5 = 1
y = T5
T8 = 3
a = T8
T11 = 4
b = T11
T14 = a
T15 = b
T16 = T14 + T15
d = T16
T19 = x
T20 = 1
T21 = T19 == T20
If False T21 goto L0
BeginBlockT22 = 1
c = T22
EndBlockgoto L1
L0: T27 = y
T28 = 1
T29 = T27 == T28
If False T29 goto L0
BeginBlockT30 = 2
c = T30
EndBlockgoto L1
L0: BeginBlockT35 = 1
c = T35
EndBlockL1: L1:
```

1.6 All Symbol Tables

```
----------------------------All Symbol Tables----------------------------
Scope   Name    Type          Declaration     Last Used Line
(0, 1)  scipy   PackageName   3               3
(0, 1)  2       Constant      5               5
(0, 1)  x       Identifier    5               12
(0, 1)  1       Constant      6               12
(0, 1)  y       Identifier    6               14
(0, 1)  3       Constant      8               8
(0, 1)  a       Identifier    8               10
(0, 1)  4       Constant      9               9
(0, 1)  b       Identifier    9               10
(0, 1)  d       Identifier    10              10
(0, 1)  T2      ICGTempVar    -1              -1
(0, 1)  T5      ICGTempVar    -1              -1
(0, 1)  T8      ICGTempVar    -1              -1
(0, 1)  T11     ICGTempVar    -1              -1
(0, 1)  T14     ICGTempVar    -1              -1
(0, 1)  T15     ICGTempVar    -1              -1
(0, 1)  T16     ICGTempVar    -1              -1
(0, 1)  T19     ICGTempVar    -1              -1
(0, 1)  T20     ICGTempVar    -1              -1
(0, 1)  T21     ICGTempVar    -1              -1
(0, 1)  L0      ICGTempLabel  -1              -1
(0, 1)  T22     ICGTempVar    -1              -1
(0, 1)  L1      ICGTempLabel  -1              -1
(0, 1)  T27     ICGTempVar    -1              -1
(0, 1)  T28     ICGTempVar    -1              -1
(0, 1)  T29     ICGTempVar    -1              -1
(0, 1)  T30     ICGTempVar    -1              -1
(0, 1)  T35     ICGTempVar    -1              -1
(0, 2)  1       Constant      13              14
(0, 2)  c       Identifier    13              13
(1, 4)  2       Constant      15              15
(1, 4)  c       Identifier    15              15
(1, 8)  1       Constant      17              17
(1, 8)  c       Identifier    17              17
-------------------------------------------------------------------------
```

2. TestCase_2

2.1 Input File

```
pb@ubuntu:~/Downloads/Mini-Python-Compiler-in-Lex-and-Yacc-master$ cat input2.txt
import hWorld
x=10
y=10

#Comment1
listX = []
while(x==10):
        x = 1
#Comment2
if(x==y):
        x=10
else:
        x=10

if(x==y):
        x=10
else:
        y=10
```

2.2 Token Sequence

```
---------------------------------Token Sequence--------------------------
1 T_IMPT T_hWorld T_NL
2 T_x T_EQL T_10 T_NL
3 T_y T_EQL T_10 T_NL
4 T_NL
5 T_NL
6 T_listX T_EQL T_OB T_CB T_NL
7 T_While T_OP T_x T_EQ T_10 T_CP T_Cln T_NL
8 T_ID T_x T_EQL T_1 T_NL
9 T_NL
10 T_If T_OP T_x T_EQ T_y T_CP T_Cln T_NL
11 T_ID T_x T_EQL T_10 T_NL
12 T_Else T_Cln T_NL
13 T_ID T_x T_EQL T_10 T_NL
14 T_NL
15 T_If T_OP T_x T_EQ T_y T_CP T_Cln T_NL
16 T_ID T_x T_EQL T_10 T_NL
17 T_Else T_Cln T_NL
18 T_ID T_y T_EQL T_10 T_NL
19 T_NL
20 T_EOF

  $$ PARSING COMPLETED $$
```

2.3 Abstract Syntax Tree

```
------------------------Abstract Syntax Tree------------------------
  NewLine(2)
  import(1)   NewLine(2)
  hWorld    =(2)   NewLine(2)
        x    10    =(2)   NewLine(2)
           y    10    listX    NewLine(2)
                   While(2)   NewLine(2)
                   ==(2)   BeginBlock(2)   If(3)   If(3)
                   x    10    =(2)   EndBlock    ==(2)   BeginBlock(2)   Else(1)   ==(2)   BeginBlock(2)   Else(1)
                              x    1    x    y    =(2)   EndBlock    BeginBlock(2)   x    y    =(2)   EndBlock    BeginBlock(2)
                                                    x    10    =(2)   EndBlock    x    10    =(2)   EndBlock
                                                                          x    10    y    10
```

2.4 Optimized Quads

```
--------------------------------All Quads--------------------------------
0       import  hWorld  -       -
1       =       10      -       T2
2       =       T2      -       x
3       =       10      -       T5
4       =       T5      -       y
5       =       x       -       T9
6       =       10      -       T10
7       ==      T9      T10     T11
8       Label   -       -       L0
9       If False        T11     -       L1
10      =       1       -       T12
11      =       T12     -       x
12      goto    -       -       L0
13      Label   -       -       L1
14      =       x       -       T18
15      =       y       -       T19
16      ==      T18     T19     T20
17      If False        T20     -       L4
18      =       10      -       T21
19      =       T21     -       x
20      goto    -       -       L5
21      Label   -       -       L4
22      =       10      -       T26
23      =       T26     -       x
24      Label   -       -       L5
25      =       x       -       T33
26      =       y       -       T34
27      ==      T33     T34     T35
28      If False        T35     -       L6
29      =       10      -       T36
30      =       T36     -       x
31      goto    -       -       L7
32      Label   -       -       L6
33      =       10      -       T41
34      =       T41     -       y
35      Label   -       -       L7
--------------------------------------------------------------------------
```

2.5 Intermediate Code

```
import hWorld
T2 = 10
x = T2
T5 = 10
y = T5
T9 = x
T10 = 10
T11 = T9 == T10
L0: If False T11 goto L1
BeginBlockT12 = 1
x = T12
EndBlockgoto L0
L1: T18 = x
T19 = y
T20 = T18 == T19
If False T20 goto L4
BeginBlockT21 = 10
x = T21
EndBlockgoto L5
L4: BeginBlockT26 = 10
x = T26
EndBlockL5: T33 = x
T34 = y
T35 = T33 == T34
If False T35 goto L6
BeginBlockT36 = 10
x = T36
EndBlockgoto L7
L6: BeginBlockT41 = 10
y = T41
EndBlockL7:
```

2.6 All Symbol Tables

```
---------------------------All Symbol Tables---------------------------
Scope    Name    Type           Declaration     Last Used Line
(0, 1)  hWorld  PackageName     1               1
(0, 1)  10      Constant        2               7
(0, 1)  x       Identifier      2               7
(0, 1)  y       Identifier      3               15
(0, 1)  listX   ListTypeID      6               6
(0, 1)  T2      ICGTempVar      -1              -1
(0, 1)  T5      ICGTempVar      -1              -1
(0, 1)  T9      ICGTempVar      -1              -1
(0, 1)  T10     ICGTempVar      -1              -1
(0, 1)  T11     ICGTempVar      -1              -1
(0, 1)  L0      ICGTempLabel    -1              -1
(0, 1)  L1      ICGTempLabel    -1              -1
(0, 1)  T12     ICGTempVar      -1              -1
(0, 1)  T18     ICGTempVar      -1              -1
(0, 1)  T19     ICGTempVar      -1              -1
(0, 1)  T20     ICGTempVar      -1              -1
(0, 1)  L4      ICGTempLabel    -1              -1
(0, 1)  T21     ICGTempVar      -1              -1
(0, 1)  L5      ICGTempLabel    -1              -1
(0, 1)  T26     ICGTempVar      -1              -1
(0, 1)  T33     ICGTempVar      -1              -1
(0, 1)  T34     ICGTempVar      -1              -1
(0, 1)  T35     ICGTempVar      -1              -1
(0, 1)  L6      ICGTempLabel    -1              -1
(0, 1)  T36     ICGTempVar      -1              -1
(0, 1)  L7      ICGTempLabel    -1              -1
(0, 1)  T41     ICGTempVar      -1              -1
(0, 2)  1       Constant        8               8
(0, 2)  x       Identifier      8               15
(1, 4)  10      Constant        11              11
(1, 4)  x       Identifier      11              11
(1, 8)  10      Constant        13              13
(1, 8)  x       Identifier      13              13
(1, 16) 10      Constant        16              16
(1, 16) x       Identifier      16              16
(1, 32) 10      Constant        18              18
(1, 32) y       Identifier      18              18
-----------------------------------------------------------------------
```

CONCLUSION

A Compiler for Python was implemented using lex and yacc tools.. In addition to the constructs specified, the basic python constructs were implemented and function definitions and calls were supported. We have also shown the tokens generated , the abstract syntax tree ,intermediate code and the optimized code separately.

The compiler also reports the basic errors and gives the line number and column number. The Intermediate code was represented by quads , which was later optimized to remove dead code and reduce the usage of temporary variables.

FURTHER ENHANCEMENTS

The compiler can be further enhanced by adding support for ‘For' Loops, classes, and other types of loops. Improved memory management for registers, More effective optimization strategies Semantic analysis for parameter matching, Error recovery can be implemented to enhance the current work.