PES
UNIVERSITY

Report on

"Mini Python Compiler(If and While Constructs)"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

G Ghana Gokul	**PES2201800077**
Tushar Dixit	**PES2201800138**
Waris K R	**PES2201800315**

Under the guidance of

Prof. Swati G
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

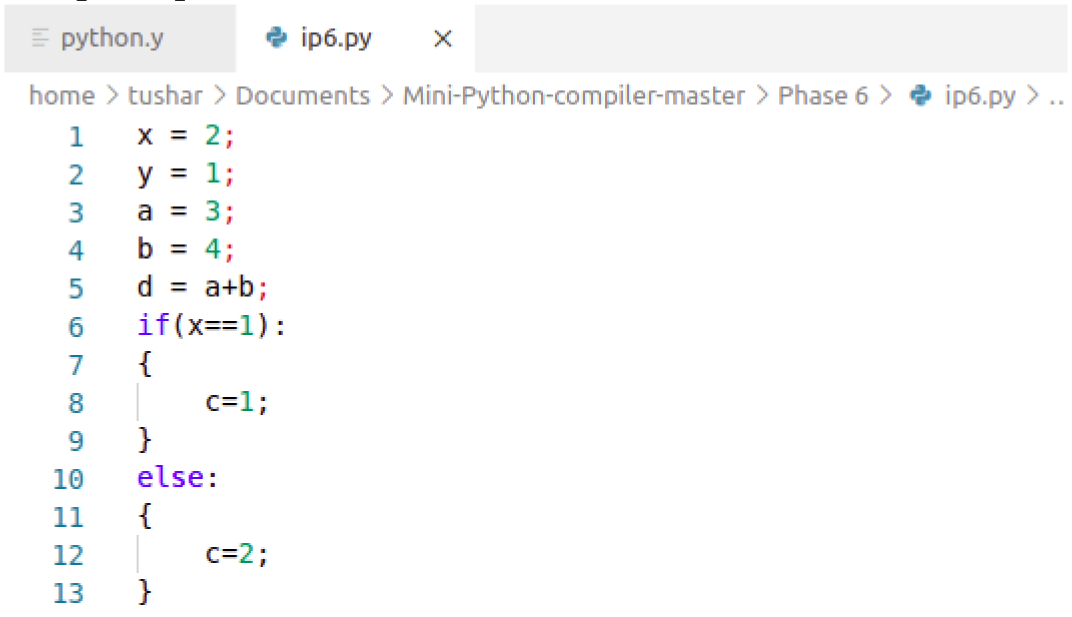
This project being a Mini Compiler for the Python programming language,
focuses on generating an intermediate code for the language for specific
constructs.
It works for constructs such as conditional statements (like if else) and
the ternary operator.
The main functionality of the project is to generate an optimized intermediate code for the given Python source code.
This is done using the following steps:

i) Generate symbol table after performing expression evaluation
ii) Generate Abstract Syntax Tree for the code
iii) Generate 3 address code followed by corresponding quadruples
iv) Perform Code Optimization

The main tools used in the project include LEX which identifies pre-defined patterns and generates tokens for the patterns matched and

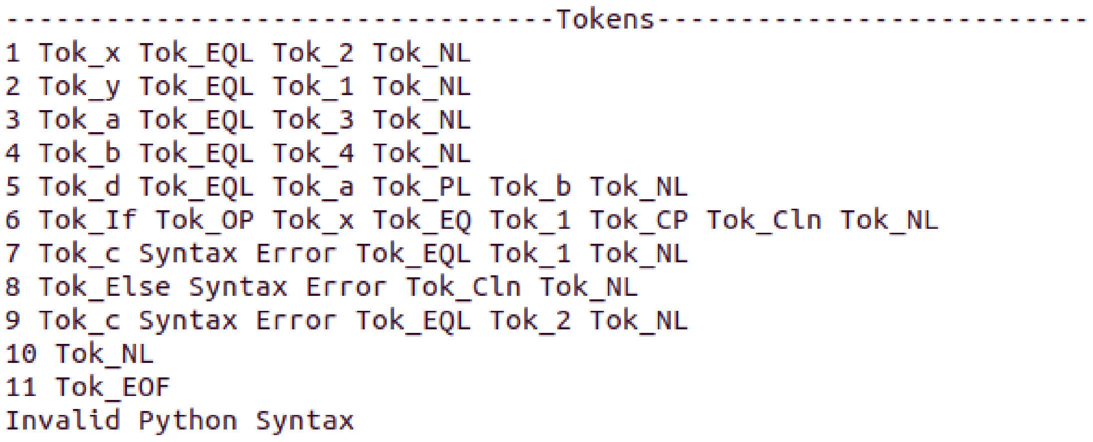
YACC which parses the input for semantic meaning and generates an abstract syntax tree and intermediate code code for the source code.

Sample Input:


python.y
ip6.py
home > tushar > Documents > Mini-Python-compiler-master > Phase 6 > ip6.py > ..

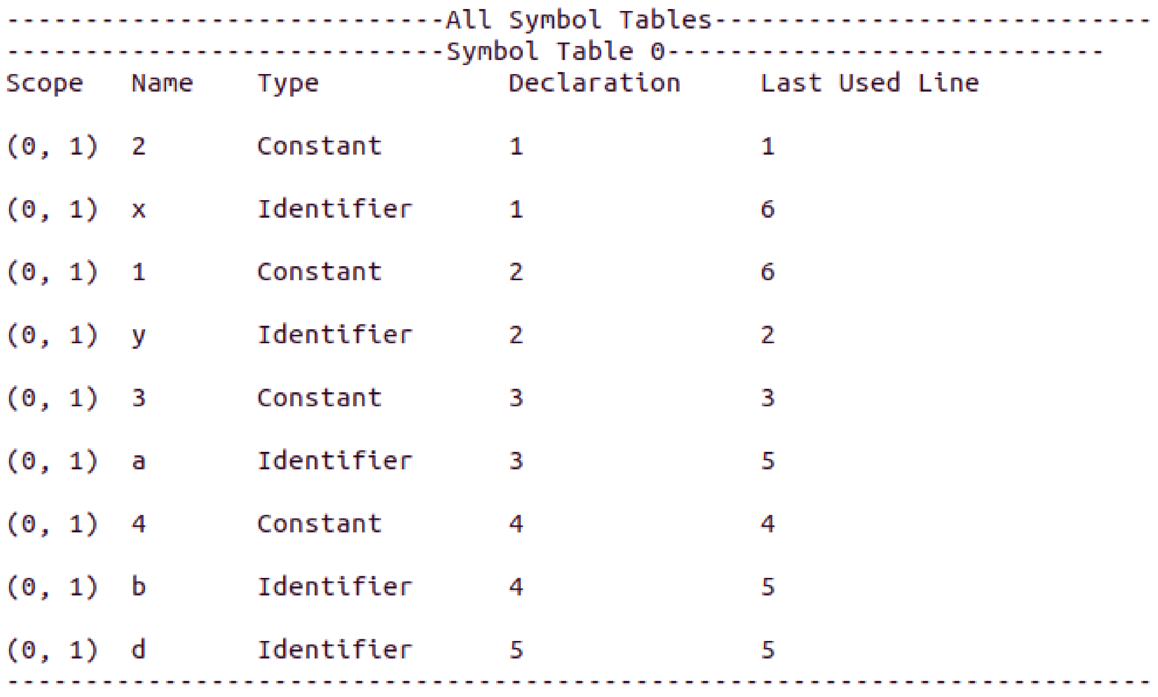
```
x = 2;
y = 1;
a = 3;
b = 4;
d = a+b;
if(x==1):
{
    c=1;
}
else:
{
    c=2;
}
```

Sample Output:
Token Sequence:



```
--------------------------------Tokens-------------------------
1 Tok_x Tok_EQL Tok_2 Tok_NL
2 Tok_y Tok_EQL Tok_1 Tok_NL
3 Tok_a Tok_EQL Tok_3 Tok_NL
4 Tok_b Tok_EQL Tok_4 Tok_NL
5 Tok_d Tok_EQL Tok_a Tok_PL Tok_b Tok_NL
6 Tok_If Tok_OP Tok_x Tok_EQ Tok_1 Tok_CP Tok_Cln Tok_NL
7 Tok_c Syntax Error Tok_EQL Tok_1 Tok_NL
8 Tok_Else Syntax Error Tok_Cln Tok_NL
9 Tok_c Syntax Error Tok_EQL Tok_2 Tok_NL
10 Tok_NL
11 Tok_EOF
Invalid Python Syntax
```

Symbol Tables:

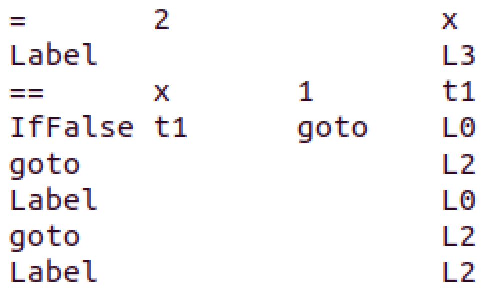


```
----------------------------All Symbol Tables----------------------------
----------------------------Symbol Table 0--------------------------
Scope   Name    Type            Declaration     Last Used Line

(0, 1)  2       Constant        1               1

(0, 1)  x       Identifier      1               6

(0, 1)  1       Constant        2               6

(0, 1)  y       Identifier      2               2

(0, 1)  3       Constant        3               3

(0, 1)  a       Identifier      3               5

(0, 1)  4       Constant        4               4

(0, 1)  b       Identifier      4               5

(0, 1)  d       Identifier      5               5
-------------------------------------------------------------------------
```

Unoptimized Intermediate Code:

```
=       2               x


=       1               y


=       3               a


=       4               b


+       a       b       t0

=       t0              d

Label                   L3


==      x       1       t1
IfFalse t1      goto    L0

=       1               c

goto                    L2
Label                   L0


=       2               c

goto                    L2
Label                   L2
```

Optimized Intermediate Code:



```
=        2               x
Label                    L3
==       x       1       t1
IfFalse  t1      goto    L0
goto                     L2
Label                    L0
goto                     L2
Label                    L2
```

Architecture Of 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.

But we have not taken care of semicolons, so a semicolon will result in an error while parsing. All lines of code terminate upon seeing a newline character.

We have taken care of the following:

● If-Elif-Else and While constructs
● Print Statements
● pass, break and void returns
● Function definitions and Calls
● Lists
● All arithmetic operators and all boolean operators except standalone ‘!’
(“!=” is taken care of)
● 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/

Context Free Grammar

Grammar to Parse While And If Statements in Python

Grammar	Expression	Legend
StartDebugger	StartParse T_EndOfFile	T_Import : "import"
constant	T_Number | T_String	T_Print : "print"
T_ID	[_a-zA-Z][_a-zA-Z0-9]*	T_Pass : "pass"
term	T_ID | constant | list_index	T_If : "if"
list_index	T_ID T_OB constant T_CB	T_While : "while"
StartParse	T_NL StartParse | finalStatements T_NL StartParse | finalStatements T_NL	T_Break : "break"
basic_smt	pass_stmt | break_stmt | import_stmt | assign_stmt | bool_exp | arith_exp | print_stmt | return_stmt	T_And : "and"
arith_exp	term | arith_exp T_PL arith_exp |	T_Or : "or"
	arith_exp T_MN arith_exp |	T_Not : "not"
	arith_exp T_ML arith_exp |	T_Elif : "elif"
	arith_exp T_DV arith_exp |	T_Else : "else
	T_OP arith_exp T_CP	T_Def : "def"
ROP	T_GT | T_LT | T_ELT | T_EGT	T_In : "in"
bool_exp	bool_term | bool_term T_Or bool_term | arith_exp T_LT arith_exp | bool_term T_And bool_term	T_Cln : ":"
	arith_exp T_GT arith_exp | arith_exp T_ELT arith_exp | arith_exp T_EGT arith_exp | arith_exp T_In T_ID	T_GT : ">"
bool_term	bool_factor | arith_exp T_EQ arith_exp | T_True | T_False	T_LT : "<"
bool_factor	T_Not bool_factor | T_OP bool_exp T_CP	T_EGT : ">="
import_stmt	T_Import T_ID	T_ELT : "<="
assign_stmt	T_ID T_EQL arith_exp | T_ID T_EQL bool_exp | T_ID T_EQL func_call | T_ID T_EQL T_OB T_CB |	T_EQ : "=="
pass_stmt	T_Pass	T_NEQ : "!="
break_stmt	T_Break	T_True : "True"
return_stmt	T_Return	T_False : "False"
print_stmt	T_Print T_OP constant T_CP	T_PL : "+"
finalStatements	basic_stmt | cmpd_stmt | func_def | func_call	T_MN : "-"
cmpd_stmt	if_stmt | while_stmt	T_ML : "*"
if_stmt	T_If bool_exp T_Cln start_suite | T_If bool_exp T_Cln start_suite elif_stmts	T_DV : "/"
elif_stmts	T_Elif bool_exp T_Cln start_suite elif_stmts | else_stmt	T_OP : "("
else_stmt	T_Else T_Cln start_suite	T_CP : ")"
while_stmt	T_While bool_exp T_Cln start_suite	T_OB : "["
start_suite	basic_stmt | T_NL ID finalStatements suite	T_CB : "]"
suite	T_NL ND finalStatements suite | T_NL end_suite	T_Comma : ","
end_suite	DD finalStatements | DD |	T_EQL : "="
args	T_ID args_list |	T_NL : "\n"
args_list	T_Comma T_ID args_list |	T_String : String
call_list	T_Comma term call_list |	T_Number : Number
call_args	term call_list |	ND : Nodent
func_def	T_Def T_ID T_OP args T_CP T_Cln start_suite	ID : Indent
func_call	T_ID T_OP call_args T_CP	DD : Dedent
		T_EOF : End Of File

Design Strategy

The Design Strategy we have implemented is that firstly, every links back to the Symbol Table. This means that the required nodes of the Abstract Syntax tree and the required Quadruples in the Intermediate code have a link to the Symbol table. All the compiler generated Temporaries are also stored in the symbol table.

The Symbol Table stores ‘Records’ having 4 columns as you can see in the sample
output, ie,

- ❖ Scope : Scope of each variable contained in record
- ❖ Name : Value/Name of each variable contained in record
- ❖ Type : Type of each variable contained in record
    - ➢ PackageName
    - ➢ Func_Name
    - ➢ Identifier
    - ➢ Constant
    - ➢ ListTypeID
    - ➢ ICGTempVar
    - ➢ ICGTempLabel
- ❖ Declaration : Line of Declaration of each variable contained in record
- ❖ Last Use Line

The scope is a function of Indentation depth and to make it unique we have a tuple of the scope of the parent and the current scope calculated using the Indentation depth.

For the Abstract Syntax Tree, We have 2 Types of Nodes, Leaf nodes and Internal
nodes. The nodes can have variable number of children (0-3) depending upon the
construct it represents. Take the example of the If-Else Statement,

Condition
If
CodeBlock
Else
To display the AST, We take the AST and store it as a matrix of levels. As we can see in the sample output, we have printed each level of the AST. All Internal nodes also have a number enclosed in brackets next to them, which represents the number of children they have in the next level. Leaf nodes in the AST representing identifiers, constants, Lists, packages point to a record in the symbol table.

The Intermediate code is generated by recursively stepping through the AST. Each line is stored as a quadruple (Operation, Arg1, Arg2, Result) so that we may easily optimize code in the following steps.

Optimizations done:
1. Dead code elimination
2. Constant Folding
3. Constant Propagation
4. Copy Propagation
The implementation of these are present in opt1.py file as we have used python to do all code optimizations.

We Eliminate dead code, specifically unused variables in the whole program. 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.

To take care of the Indentation based code structure and scoping we have
implemented a stack and we use 3 tokens. The top of the stack always points to the
current indentation value, if that value on scanning the next line, doesn't change, it
implies that we're in the same scope and hence, we return the token 'ND', i.e,
'No-dent'. If the value increases, it means we are entering a sub-scope and we
return the token 'ID' , i.e. 'Indent' and finally, if the value decreases, it means we're
returning to one of the enclosing scope and we return 'DD', i.e. 'Dedent'.
For Error Handling, Whenever the parser encounters an error, It prints the Line no
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” structure represents each record in the symbol table. Each symbol table can contain a maximum of “MAXRECST” records, MAXRECST is a macro.

The “STable” structure represents one symbol table. A new Symbol table is made for every scope. A Maximum of “MAXST” symbol tables and hence scopes can exist, MAXST is a macro.

The “STable” structure represents one symbol table. A new Symbol table is made for every scope. A Maximum of “MAXST” symbol tables and hence scopes can exist, MAXST is a macro.

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 structure takes care of both leaf nodes as well as Internal"Operator" Nodes. The respective values are set depending upon the type of node. Each node can have 0-3 children. We print the AST by first storing it in a Matrix of Order "MAXLEVELS" x "MAXCHILDREN" and printing the matrix Levelwise. This matrix, is a matrix of pointers to the AST. The "noOps" element of the Node gives the number of children of that node.

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;
```

The last element, the integer 'I', is used during code optimization. All the Three-Address codes are stored as Quads in an array of Quads. There can be a maximum of "MAXQUADS" quadruples.

For the dead code elimination we continuously loop through the code till no more code can be eliminated. To check if a quadruple represents dead code we see if the result parameter/element of that quad appears as any arguments to any other subsequent quads which have not been eliminated, if not, we consider that quad to represent dead code and mark the element 'I' with the value "-1". The scope checking is handled by recursively stepping through enclosing scopes and finding the most recent definition of the variable. If no definition is found, we print the error.

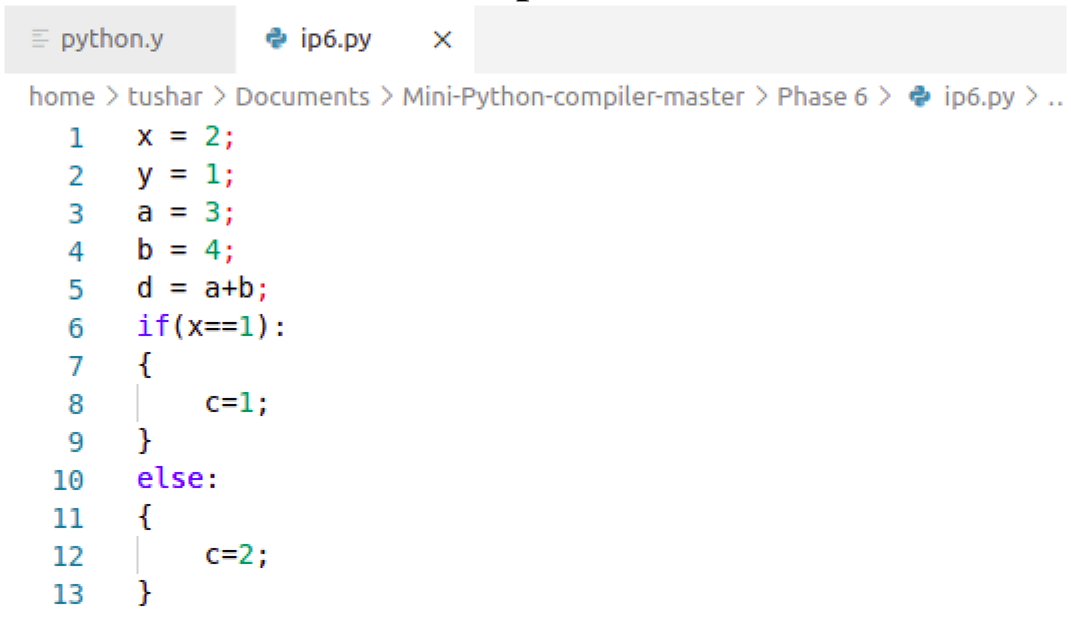
Lastly, To compile the code and execute the code we have provided a makefile and shell file.
If you wish to only see the unoptimized ICG,
```
./python.sh -i0 input.py
```
and for optimized ICG,
```
./python.sh -i1 input.py
```
These will generate the output file input.I which contains the ICG.

Results And Shortcomings

The result achieved is that we have a mini compiler which parses grammar
corresponding to basic python syntax and generates finally, an optimized intermediate representation.
The areas where our mini compiler falls short are,
● Only one very basic optimization is implemented that does not reduce code
density by a huge margin.
● The Program has a few memory leaks, although most of them have been taken care of.

Snapshots

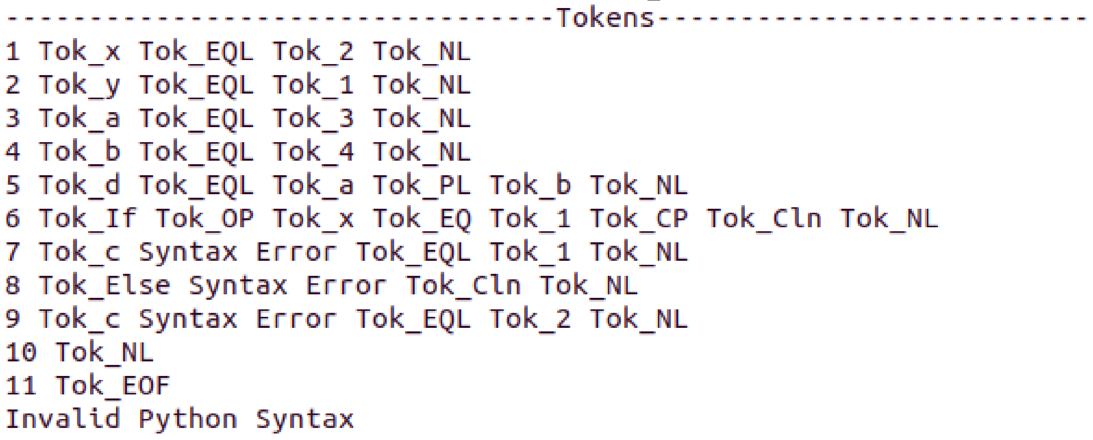
Input:


python.y
ip6.py
home > tushar > Documents > Mini-Python-compiler-master > Phase 6 > ip6.py > ..

```
x = 2;
y = 1;
a = 3;
b = 4;
d = a+b;
if(x==1):
{
    c=1;
}
else:
{
    c=2;
}
```

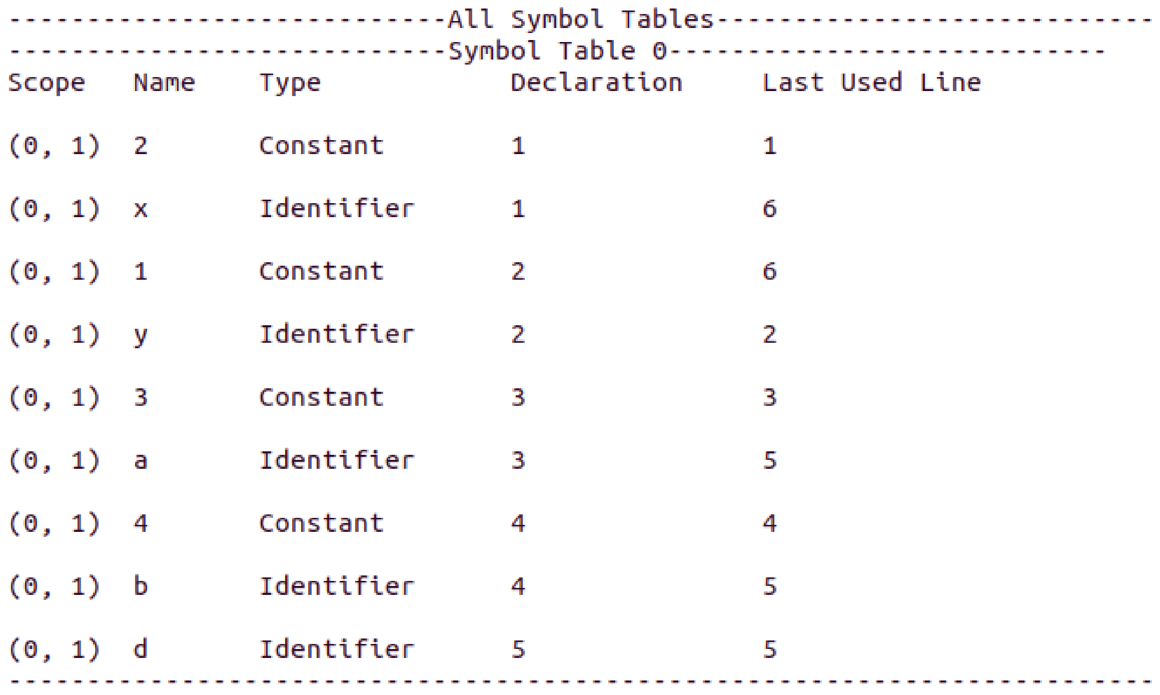
Outputs:

1. Token Sequence:



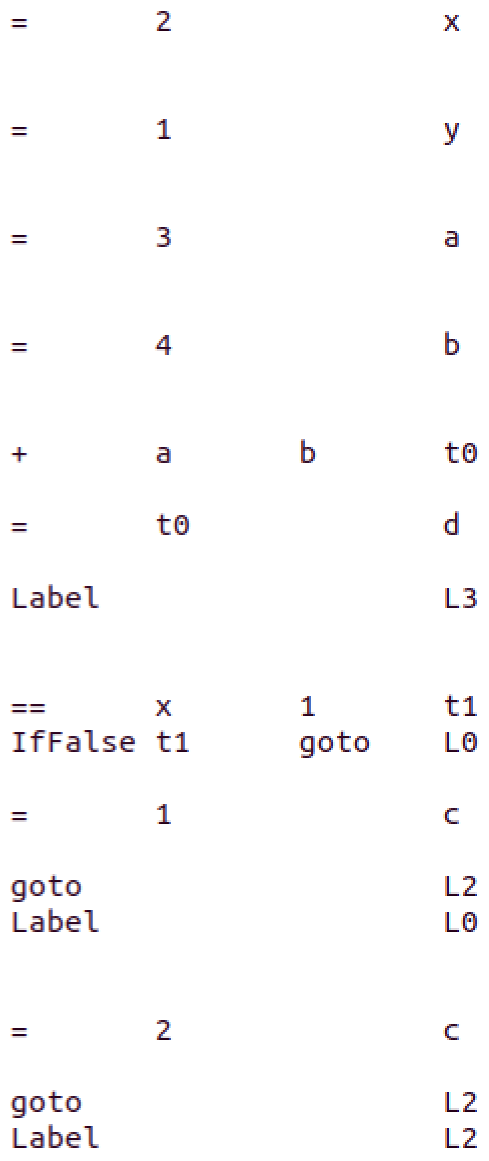
```
---------------------------------Tokens-------------------------
1 Tok_x Tok_EQL Tok_2 Tok_NL
2 Tok_y Tok_EQL Tok_1 Tok_NL
3 Tok_a Tok_EQL Tok_3 Tok_NL
4 Tok_b Tok_EQL Tok_4 Tok_NL
5 Tok_d Tok_EQL Tok_a Tok_PL Tok_b Tok_NL
6 Tok_If Tok_OP Tok_x Tok_EQ Tok_1 Tok_CP Tok_Cln Tok_NL
7 Tok_c Syntax Error Tok_EQL Tok_1 Tok_NL
8 Tok_Else Syntax Error Tok_Cln Tok_NL
9 Tok_c Syntax Error Tok_EQL Tok_2 Tok_NL
10 Tok_NL
11 Tok_EOF
Invalid Python Syntax
```

2. Symbol Tables



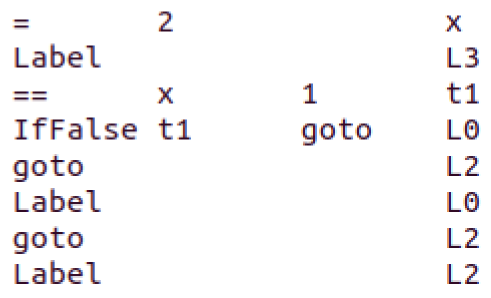
```
----------------------------All Symbol Tables----------------------------
----------------------------Symbol Table 0----------------------------
Scope   Name    Type            Declaration     Last Used Line

(0, 1)  2       Constant        1               1

(0, 1)  x       Identifier      1               6

(0, 1)  1       Constant        2               6

(0, 1)  y       Identifier      2               2

(0, 1)  3       Constant        3               3

(0, 1)  a       Identifier      3               5

(0, 1)  4       Constant        4               4

(0, 1)  b       Identifier      4               5

(0, 1)  d       Identifier      5               5
-------------------------------------------------------------------------
```

3. Unoptimized ICG



```
=       2               x


=       1               y


=       3               a


=       4               b


+       a       b       t0

=       t0              d

Label                   L3


==      x       1       t1
IfFalse t1      goto    L0

=       1               c

goto                    L2
Label                   L0


=       2               c

goto                    L2
Label                   L2
```

4. Optimized ICG



```
=       2               x
Label                   L3
==      x       1       t1
IfFalse t1      goto    L0
goto                    L2
Label                   L0
goto                    L2
Label                   L2
```

Conclusions

In Conclusion, A Compiler for Python was implemented. In addition to the constructs specified, the basic python constructs were implemented and function definitions and calls were supported.

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.

Further Enhancements

The Compiler can be further enhanced by adding,

- Support for 'For' Loops
- Better Memory Management
- More efficient optimization techniques
- Semantic analysis for Parameter Matching