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:

Atul Anand Gopalkrishna	**PES1201701452**
Akhilarka Jayanthi	**PES1201700050**
Abhilash Balaji	**PES1201700041**

Under the guidance of

C.O. Prakash
Asst. Professor
PES University, Bengaluru

January – May 2020

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

6.	**IMPLEMENTATION DETAILS (TOOL AND DATA STRUCTURES USED in order to implement the following):** ● **SYMBOL TABLE CREATION** ● **ABSTRACT SYNTAX TREE (internal representation)** ● **INTERMEDIATE CODE GENERATION** ● **CODE OPTIMIZATION** ● **ASSEMBLY CODE GENERATION** ● **ERROR HANDLING - strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).** ● **Provide instructions on how to build and run your program.**	
7.	**RESULTS AND possible shortcomings of your Mini-Compiler**	
8.	**SNAPSHOTS (of different outputs)**	
9.	**CONCLUSIONS**	
10.	**FURTHER ENHANCEMENTS**	
REFERENCES/BIBLIOGRAPHY		

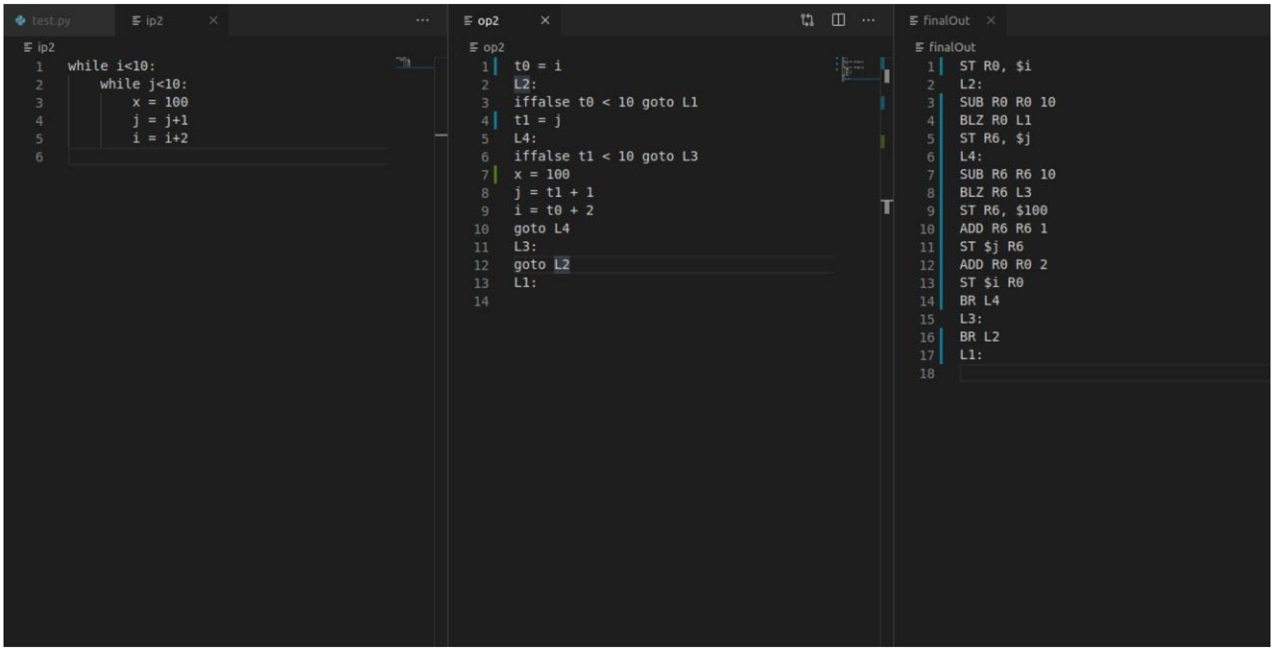
INTRODUCTION

We have built a mini compiler for the python programming language which supports the following constructs of the language.

- If else elif construct
- for loop construct
- while loop construct

The compiler takes the source program as input and produces a symbol table, an abstract syntax tree. After this it produces an intermediate code which is then optimized. This optimized intermediate code is then given as input to a program which generates a target assembly code as the final output.

Sample Input ICG Sample Output



```
while i<10:
    while j<10:
        x = 100
        j = j+1
        i = i+2
```

```
t0 = i
L2:
iffalse t0 < 10 goto L1
t1 = j
L4:
iffalse t1 < 10 goto L3
x = 100
j = t1 + 1
i = t0 + 2
goto L4
L3:
goto L2
L1:
```

```
ST R0, $i
L2:
SUB R0 R0 10
BLZ R0 L1
ST R6, $j
L4:
SUB R6 R6 10
BLZ R6 L3
ST R6, $100
ADD R6 R6 1
ST $j R6
ADD R0 R0 2
ST $i R0
BR L4
L3:
BR L2
L1:
```

ARCHITECTURE

1. Lex tool was used to scan the input program and generate tokens for it. The Lex program takes the program as input and scans for constructs using regular expressions when a particular construct is matched; the appropriate token is generated and the Lex file constructs a symbol table using these tokens. The Lex file also has regular expressions to ignore certain constructs such as comments which either start with #(single line comment) or “””,’’’ (multi line comment).
2. The yacc tool was used to parse the grammar for the language using the tokens generated by the lex tool. The tokens generated by lex are passed to the yacc tool which tries to verify the grammar using the rules specified for the if, for and while constructs. If an error is detected in the syntax, the program keeps skipping characters until a valid statement is found.

3. After parsing this grammar and verifying its validity we generate an intermediate code (three address code) using rules specified along with the grammar. This intermediate code is also optimized, the techniques used for optimization are as follows.
    - Common subexpression elimination.
    - Constant Folding.
4. After this the optimized intermediate code is passed to a program that generates target assembly code.

LITERATURE SURVEY:

- https://www.geeksforgeeks.org/code-optimization-in-compiler-design/
- Compilers: Principles, Techniques, and Tools:Alfred Aho, Ravi Sethi, Jeffrey Ullman, Monica S. Lam
- https://docs.python.org/3/reference/grammar.html

CONTEXT-FREE GRAMMAR:

start: list_stmt start |%empty

list_stmt: if_stmt | for_stmt | while_stmt ;

if_stmt: "if" if_test ":" "\n" if_suite elif_stmt optional_else

elif_stmt: %empty | "elif" if_test ":" "\n" if_suite elif_stmt ;

optional_else: %empty | "else" ":" "\n" if_suite ;

if_test:
IDENTIFIER logical_op IDENTIFIER | IDENTIFIER logical_op
DIGIT| IDENTIFIER

if_suite: if_assign_id_digit if_suite | assign_id_digit suite
|%empty ;

if_assign_id_digit:
IDENTIFIER "=" DIGIT "\n" | IDENTIFIER "=" IDENTIFIER "\n"

for_stmt: "for" for_test ":" "\n" suite

for_test: IDENTIFIER "in" "range" "(" DIGIT ")"

while_stmt: "while" test ":" "\n" suite

test:IDENTIFIER logical_op IDENTIFIER | IDENTIFIER logical_op DIGIT| IDENTIFIER

logical_op: "==" | "!=" | "<" | ">" | "<=" | ">=" ;

suite: assign_id_digit suite | %empty ;

assign_id_digit: IDENTIFIER "=" DIGIT "\n" | IDENTIFIER "=" IDENTIFIER "\n" |IDENTIFIER "=" IDENTIFIER ARITHMETIC_OP IDENTIFIER NEWLINE |IDENTIFIER "=" IDENTIFIER ARITHMETIC_OP DIGIT NEWLINE | IDENTIFIER "=" DIGIT ARITHMETIC_OP IDENTIFIER NEWLINE | IDENTIFIER "=" DIGIT ARITHMETIC_OP DIGIT NEWLINE

ARITHMETIC_OP : "+" | "-" | "/" | "*" | "**"

letter : "A"|"B"...............|"z"|"_"
DIG : 0|1|2.......|9
DIGIT : DIG(DIG)*
IDENTIFIER : letter|(letter|digit)*

DESIGN AND IMPLEMENTATION:

Language: C
Tools : Lex and Yacc
Types :
Constructs : for , while , and if statements

1. Symbol Table :
   The symbol table data structure is implemented as array of structures. It contains fields for construct , construct abstract token, value, scope and line no. All identifiers and tokens are inserted into the symbol table.
2. Abstract Syntax Tree :

The abstract syntax tree is built using node structures which have field for the current node value and links to 4 more children. The yacc file build the this tree while parsing through the grammar and the terr is listed out in a level order manner.

3. Intermediate Code Generation :

   The Intermediate code is generated in the yacc program. This is done by specifying rules along with the grammar that converts the corresponding structure to a three address code format while the grammar is being parsed.

4. Code Optimization :

   We have implemented two optimizations to generate more efficient intermediate code. One of the optimizations was compile time evaluation, for example is we have a statement a = 1 + 2 since the right hand side only consists of digits this can be evaluated at compile time and the three address code generated need not have a add operation, this reduces the number of basic arithmetic operation in the target code. The basic techniques used for this are common subexpression elimination and constant folding.

5. Error Handling :

   For error handling we have used panic mode recovery. While the yacc file parses the grammar, when an error is seen in the syntax of the token received by the program we skip the rest of the tokens until another valid syntactical match is identified.

6. Target Code Generation:

   For Target Code Generation we have used LRU register selection algorithm along with redundancy failsafes , a python script analyzes and parses the intermediate code with regular expressions and grouping , LRU is implemented with a register - line key value dictionary.

IMPLEMENTATION:

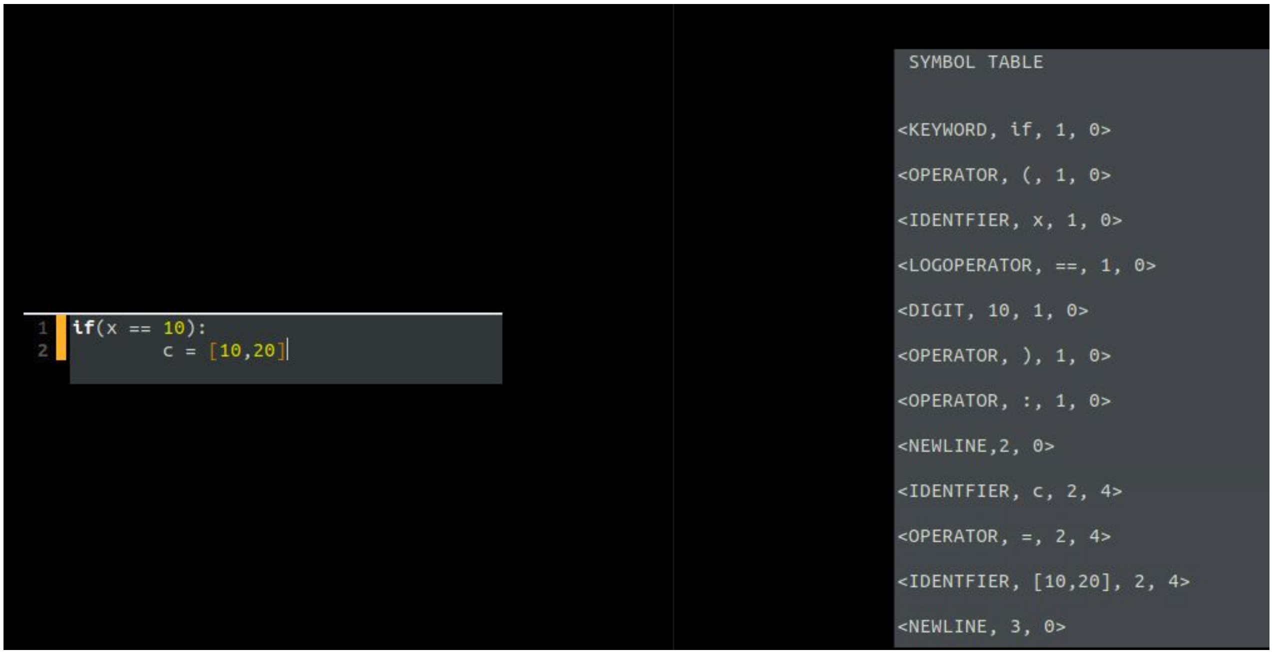
1. The lex program is used to scan the python file. It first removes all unnecessary lines such as blank lines or line starting with # or ‘’’,”” which declare single and multiline comments respectively. After this the lex file uses the regular expressions which are used to identify constructs belonging to the language. After these constructs are identified respective tokens are generated which are then sent to the yacc for further steps such as parsing. Furthermore when constructs are identified they are entered into the symbol table structure which consists of the actual token, its abstract name, its value, the line number where it was found and its scope which is determined by the indentation level of the statement.
2. After this the yacc file takes the tokens generated by the lex program as an input and does further processing. Firstly the tokens which are received are first verified by the grammar defined for the language to ensure the validity of the stream of tokens and to verify that it belongs to the python grammar. If any errors are generated in case of wrong

syntax this is handled by panic mode recovery where tokens are skipped until another syntactically valid grammar statement is encountered.

3. During the parsing the syntactically valid statements are then put into the form of an abstract syntax tree. This is done using rules specified alongside the grammar to insert into the tree and adding child nodes to a parent node in the tree. Once generated this tree is then presented in the level order representation.
4. Also During this phase intermediate code is generated. This is done by specifying appropriate rules along with the grammar to convert the respective statement to its equivalent three address code format. While this is happening transformations are made to the code to make it more efficient. Such as ......
5. Finally this optimized intermediate code is passed to the program to generate the assembly target code , this python program Parses the code using Regular expressions and converts the intermediate code into assembly target code (MIPS in our case).LRU register selection is also implemented.

SNAPSHOTS:

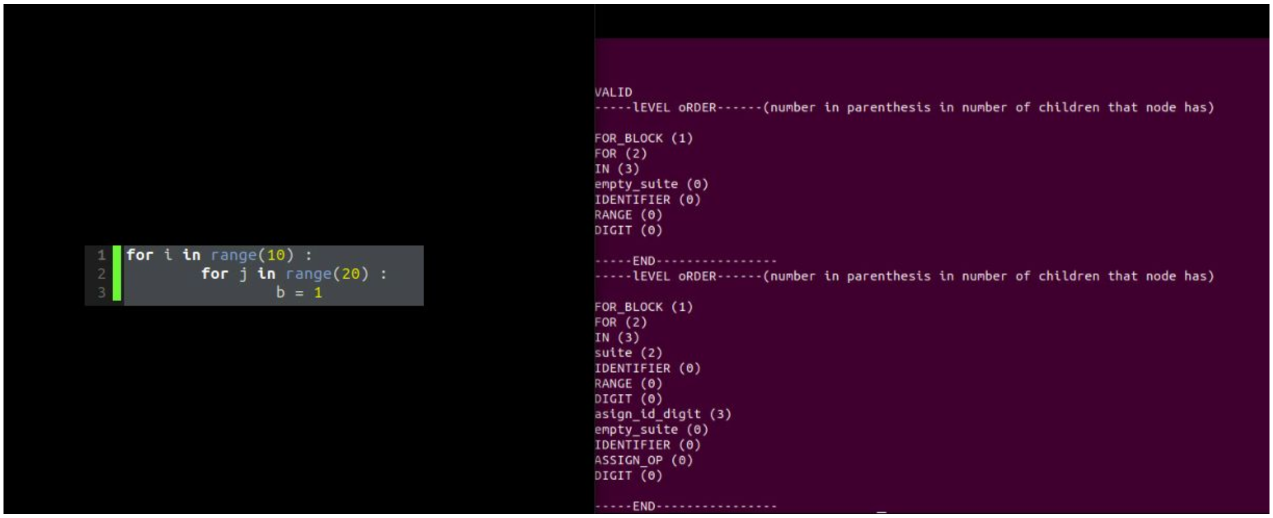
Symbol Table :



```
if(x == 10):
        c = [10,20]
```

```
SYMBOL TABLE

<KEYWORD, if, 1, 0>
<OPERATOR, (, 1, 0>
<IDENTFIER, x, 1, 0>
<LOGOPERATOR, ==, 1, 0>
<DIGIT, 10, 1, 0>
<OPERATOR, ), 1, 0>
<OPERATOR, :, 1, 0>
<NEWLINE,2, 0>
<IDENTFIER, c, 2, 4>
<OPERATOR, =, 2, 4>
<IDENTFIER, [10,20], 2, 4>
<NEWLINE, 3, 0>
```

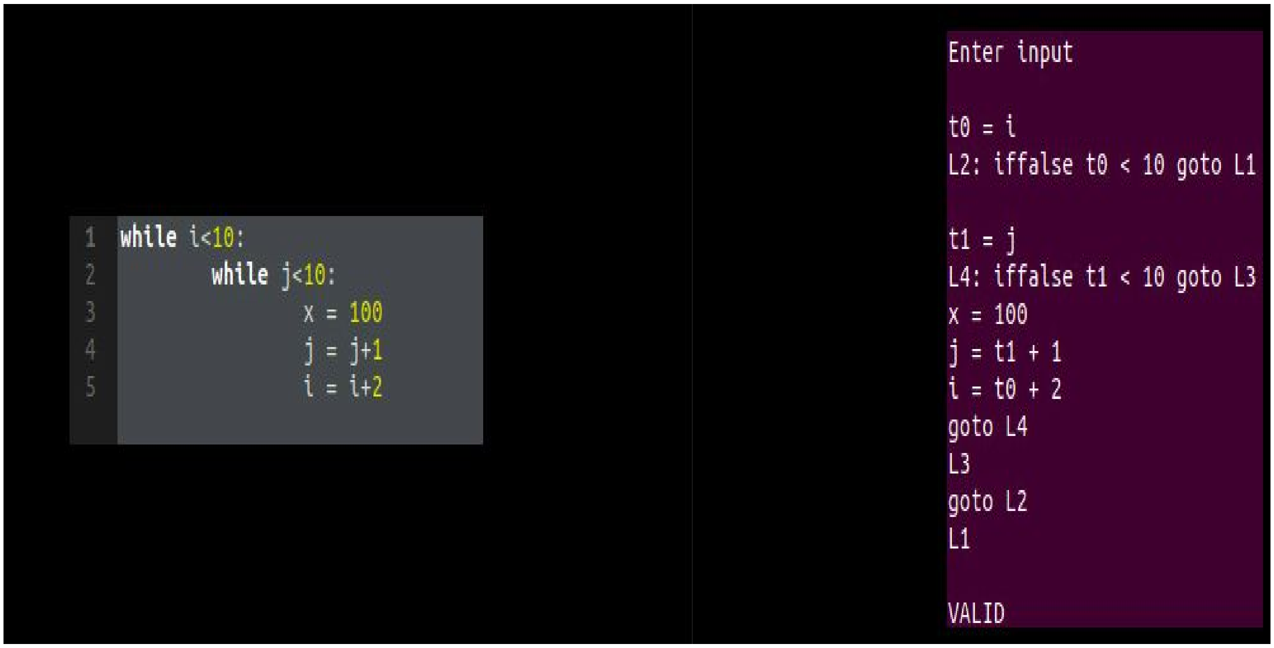
Abstract Syntax Tree :



```
for i in range(10) :
        for j in range(20) :
                b = 1
```

```
VALID
-----lEVEL oRDER------(number in parenthesis in number of children that node has)

FOR_BLOCK (1)
FOR (2)
IN (3)
empty_suite (0)
IDENTIFIER (0)
RANGE (0)
DIGIT (0)

-----END---------------
-----lEVEL oRDER------(number in parenthesis in number of children that node has)

FOR_BLOCK (1)
FOR (2)
IN (3)
suite (2)
IDENTIFIER (0)
RANGE (0)
DIGIT (0)
asign_id_digit (3)
empty_suite (0)
IDENTIFIER (0)
ASSIGN_OP (0)
DIGIT (0)

-----END---------------
```

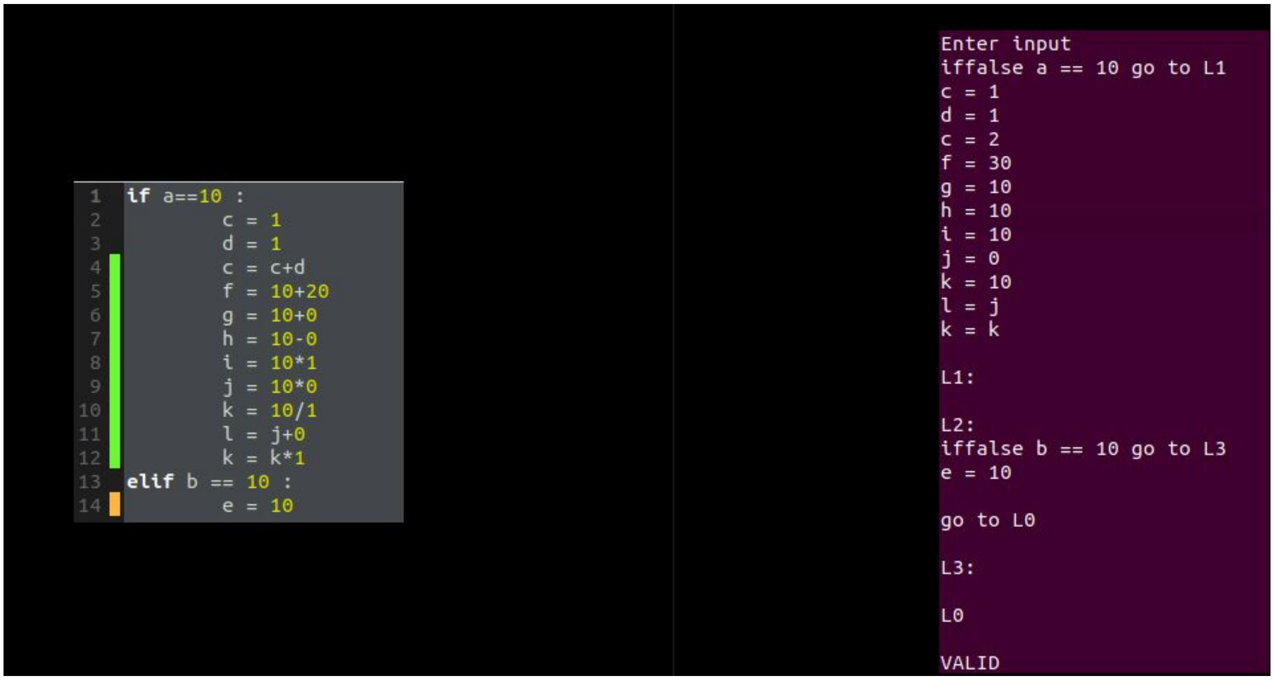
Intermediate Code :



```
while i<10:
        while j<10:
                x = 100
                j = j+1
                i = i+2
```

```
Enter input

t0 = i
L2: iffalse t0 < 10 goto L1

t1 = j
L4: iffalse t1 < 10 goto L3
x = 100
j = t1 + 1
i = t0 + 2
goto L4
L3
goto L2
L1

VALID
```

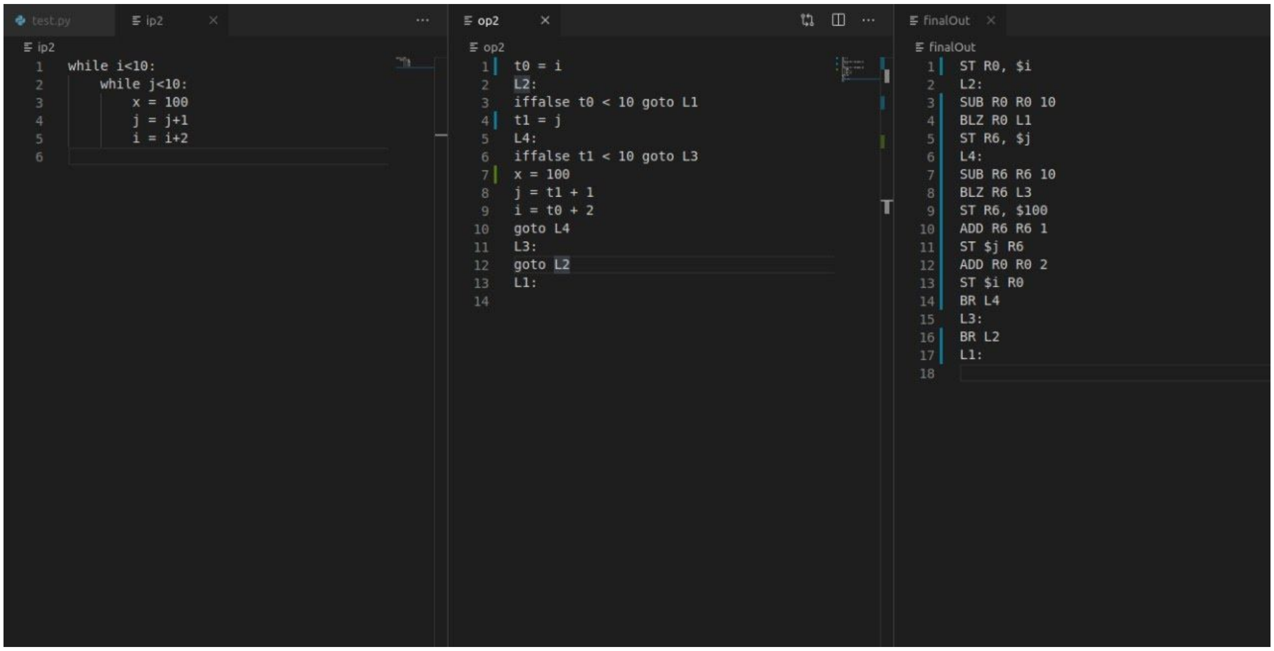
Optimizing Code :



```
if a==10 :
        c = 1
        d = 1
        c = c+d
        f = 10+20
        g = 10+0
        h = 10-0
        i = 10*1
        j = 10*0
        k = 10/1
        l = j+0
        k = k*1
elif b == 10 :
        e = 10
```

```
Enter input
iffalse a == 10 go to L1
c = 1
d = 1
c = 2
f = 30
g = 10
h = 10
i = 10
j = 0
k = 10
l = j
k = k

L1:

L2:
iffalse b == 10 go to L3
e = 10

go to L0

L3:

L0

VALID
```

Machine Code Generation:



```
while i<10:
	while j<10:
		x = 100
		j = j+1
		i = i+2
```

```
t0 = i
L2:
iffalse t0 < 10 goto L1
t1 = j
L4:
iffalse t1 < 10 goto L3
x = 100
j = t1 + 1
i = t0 + 2
goto L4
L3:
goto L2
L1:
```

```
ST R0, $i
L2:
SUB R0 R0 10
BLZ R0 L1
ST R6, $j
L4:
SUB R6 R6 10
BLZ R6 L3
ST R6, $100
ADD R6 R6 1
ST $j R6
ADD R0 R0 2
ST $i R0
BR L4
L3:
BR L2
L1:
```

RESULTS AND CONCLUSION:

The program prints valid if there are no errors in the grammar else it prints error where an error ridden input is received. The symbol table, ast and icg are printed on the terminal when generated. FInally the icg must be passed to another program which generates the target code based on MIPS architecture.

SHORTCOMINGS:

- The parser only accepts tabs as an indent; a space is not allowed.

FUTURE ENHANCEMENTS:

- Handling both spaces and tabs for indents.
- Handling more complex data structures inbuilt in python.
- Modifying symbol table to hold n children and not just a fixed number.

REFERENCES:

- Compilers: Principles, Techniques, and Tools:Alfred Aho, Ravi Sethi, Jeffrey Ullman, Monica S. Lam
- https://www.ibm.com/support/knowledgecenter/ssw_aix_72/generalprogramming/yacc_prg_handling.html
- https://avinashsuryawanshi.files.wordpress.com/2016/10/9.pdf
- https://www.geeksforgeeks.org/code-optimization-in-compiler-design/