



***Report on***

**“MiniPy3 - A mini Python3 compiler”**

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

***Compiler Design Laboratory***

**Bachelor of Technology  
in  
Computer Science & Engineering**

***Submitted by:***

<b>Anantharam R U</b>	<b>PES1201700088</b>
<b>Gaurang Rao</b>	<b>PES1201701103</b>
<b>Shashank Prasad</b>	<b>PES1201700667</b>

***Under the guidance of***

**Kiran P.  
Assistant Professor, Dept. of CSE**

**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**

## **TABLE OF CONTENTS**

<b>Chapter No.</b>	<b>Title</b>	<b>Page No.</b>
<b>1.</b>	<b>INTRODUCTION</b>	<b>3</b>
<b>2.</b>	<b>ARCHITECTURE OF LANGUAGE:</b>	<b>5</b>
<b>3.</b>	<b>LITERATURE SURVEY</b>	<b>5</b>
<b>4.</b>	<b>CONTEXT FREE GRAMMAR</b>	<b>6</b>
<b>5.</b>	<b>DESIGN STRATEGY</b> <ul style="list-style-type: none"><li>• SYMBOL TABLE CREATION</li><li>• ABSTRACT SYNTAX TREE</li><li>• INTERMEDIATE CODE GENERATION</li><li>• CODE OPTIMIZATION</li><li>• ERROR HANDLING</li><li>• TARGET CODE GENERATION</li></ul>	<b>8</b>
<b>6.</b>	<b>IMPLEMENTATION DETAILS:</b> <ul style="list-style-type: none"><li>• SYMBOL TABLE CREATION</li><li>• ABSTRACT SYNTAX TREE</li><li>• INTERMEDIATE CODE GENERATION</li><li>• CODE OPTIMIZATION</li><li>• ASSEMBLY CODE GENERATION</li><li>• ERROR HANDLING</li><li>• INSTRUCTION FOR EXECUTION.</li></ul>	<b>9</b>
<b>7.</b>	<b>RESULTS</b>	<b>11</b>
<b>8.</b>	<b>SNAPSHOTS</b>	<b>12</b>
<b>9.</b>	<b>CONCLUSIONS</b>	<b>26</b>
<b>10.</b>	<b>FURTHER ENHANCEMENTS</b>	<b>26</b>
<b>REFERENCES/BIBLIOGRAPHY</b>		

# 1. Introduction

This Mini-compiler is built for **Python3** and handles constructs like if, for and while. It also handles import and print statements. It is built using the C++ language, for Lexical Analysis, Symbol Table Generation, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Intermediate Code Optimization and Python for Target Code Generation.

## Sample Input:

```
test > testInput3.py
1  # this is a test comment, to be removed by our compiler
2  import numpy
3
4  a = 10
5  b = 20
6  c = 5
7  s = "string!"
8
9  """
10 this is a test
11 multiline comment
12 to be removed
13 """
14
15 if a < b:
16     a = a + b * c
17     c = b
18
19 d = 10
20
21 for char in s:
22     print(char)
23
24 e = a + b * c / d
```

## Sample Output:

```
test > ≡ output3.txt
84 Target Code:
85
86 MOV R0, #10
87
88 MOV R1, #20
89
90 MOV R2, #5
91
92 str: .asciiz "string!"
93 LDR R3, str
94 LDR R4, s
95 LDR R5, [R3]
96 STR R5, [R4]
97
98 LDR R0, a
99 LDR R1, b
100 CMP R0, R1
101 BGE L0
102
103 LDR R1, b
104 LDR R3, T0
105 LDR R5, [R1]
106 MUL R5, R5, c
107 STR R5, [R3]
108
109 LDR R0, a
110 LDR R5, T1
111 LDR R6, [R0]
112 ADD R6, R6, T0
113 STR R6, [R5]
114
115 LDR R5, T1
116 LDR R0, a
117 LDR R6, [R5]
118 STR R6, [R0]
119

test > ≡ output3.txt
120 LDR R1, b
121 LDR R2, c
122 LDR R5, [R1]
123 STR R5, [R2]
124
125 L0:
126 MOV R5, #10
127
128 LDR R6, #0
129 L1:
130 LDR R7, s(R6)
131 ADD R6, R6, #1
132 CMP R7, 0x00
133 BEQ L2
134
135 L2:
136 LDR R2, c
137 LDR R8, T2
138 LDR R9, [R2]
139 DIV R9, R9, d
140 STR R9, [R8]
141 B L1
142
143 LDR R1, b
144 LDR R2, T3
145 LDR R6, [R1]
146 MUL R6, R6, T2
147 STR R6, [R2]
148
149 LDR R0, a
150 LDR R1, T4
151 LDR R6, [R0]
152 ADD R6, R6, T3
153 STR R6, [R1]
154

test > ≡ output3.txt
154
155 LDR R1, T4
156 LDR R0, e
157 LDR R6, [R1]
158 STR R6, [R0]
159
160 LDR R13 s
161 STR R4, [R13]
162 LDR R13 T0
163 STR R3, [R13]
164 LDR R13 d
165 STR R5, [R13]
166 LDR R13 T2
167 STR R8, [R13]
168 LDR R13 T3
169 STR R2, [R13]
170
171 End of Target Code!
172
```

## 2. Architecture of Language

Our Python Compiler takes care of the following:

### Syntax:

- Assignment operations of string and integer data types.
- Arithmetic operations.
- **if** conditional statements constructs.
- **while** and **for** loop constructs.
- Single line comments.
- Multi-line comments.
- **print** statements.
- **import** statements.

### Semantics :

- We ensure that our conditional statements for **if**, **for** and **while**, are Boolean expressions.

## 3. Literature Survey

1. Lex and Yacc: A Brief Tutorial
  - Saumya K. Debray
2. Lex and Yacc Tutorial
  - Tom Niemann
3. Mastering Regular Expressions
  - Jeffrey E. F. Friedl
4. Official Bison Documentation
5. Official ARM Documentation
6. Stackoverflow

## 4. Context Free Grammar

### LEGEND

ENDF	End-Of-File
NUMBER	[0-9] +
STRING_LIT	\ " ( [ ^ \ " \ n ] ) * \ "   \ ' ( [ ^ \ ' \ n ] ) * \ ' '
SPACE	" "
TAB	" "
FOR	"for"
WHILE	"while"
IMPORT	"import"
PRINT	"print"
IN	"in"
PLUS	"+"
MINUS	"_"
DIVIDE	"/"
MUL	"*"

GT	">"
LT	"<"
GTE	">="
LTE	"<="
EQ	"="
IDENTIFIER	[ _ a - z A - Z ] [ _ a - z A - Z 0 - 9 ] *
LBRACKET	" ("
RBRACKET	") "
SEMICOLON	";"
COLON	":"
COMMA	","
NL	"\n"
KEYWORD	"class"   "def"   "range"
RET	"return"

## GRAMMAR

S → stmt S  
| NL S  
| ENDF

cond\_lit → ID | NUMBER | STRING\_LIT

stmt → expre  
| control\_loops  
| arith\_expr  
| import\_stmt  
| print\_stmt

repeat\_stmt → %empty  
| stmt repeat\_stmt {\$\$ = \$1;}

print\_stmt → PRINT LBRACKET print\_internals RBRACKET

print\_internals → cond\_lit COMMA SPACE print\_internals  
| cond\_lit

import\_stmt → IMPORT SPACE cond\_lit

expre → ID EQ arith\_expr  
| ID SPACE EQ SPACE arith\_expr

arith\_expr → cond\_lit bin\_op arith\_expr  
| cond\_lit SPACE bin\_op SPACE arith\_expr  
| cond\_lit

control\_loops → FOR SPACE conditions COLON body  
| FOR LBRACKET conditions RBRACKET COLON body  
| WHILE SPACE conditions COLON body  
| WHILE LBRACKET conditions RBRACKET COLON body  
| IF SPACE conditions COLON body  
| IF LBRACKET conditions RBRACKET COLON body

body → NL TAB stmt repeat\_stmt body  
| NL SPACE stmt repeat\_stmt body  
| NL

conditions → cond\_lit SPACE relop SPACE cond\_lit  
| cond\_lit

rel\_op → GT | LT | GTE | LTE | IN

bin\_op → PLUS | MINUS | DIVIDE | MUL

## 5. Design Strategy

### 1. Initial:

- The first thing we implemented was removal of single and multi-line comments from the input file.

### 2. Symbol Table:

- Since this compiler was mostly built in C++, we were fortunate enough to use the simplicity of the STL vector, and its methods.
- Our **Symbol Table** was stored as a vector of entries.
- These records were customly defined structures, called as tokens, that had associated with them, a name, value, line number, scope and type.
- Each time the Lexical Analyser identified an identifier, it would be added to our **Symbol Table**.
- On reading a value associated with it (either a string, another identifier or an integer), the **Symbol Table** value associated with the record, would get updated.

### 3. Abstract Syntax Tree:

- Our **Abstract Syntax Tree (AST)** was implemented by defining a structure for nodes in the tree. Different root nodes would be stored in a single vector, for further use.
- The node structure contained a record, a pointer to the equivalent value in the **Symbol Table**, an integer indicating the number of children nodes, a vector of node pointers, containing the children nodes, a value and its associated type.
- Node creation would happen recursively if there were to be multiple layers of nodes created.
- Printing of the **AST** used a simple recursive function, that gave us the output in a pretty format.

### 4. Intermediate Code Generation:

- **Intermediate Code Generation (ICG)** happened by traversing each root node in the AST root vector, and using a recursive function to realise the needed temporary variables, and creating them on the way back.
- The **ICG** was stored in Quads, (a structure defined by us), consisting of three symbol table pointers - result, argument 1 and argument 2, and an operator.
- The **ICG** was then printed out, to be used later on.

### 5. Code Optimisation:

- For our **Code Optimisation** phase, we performed **Dead Code Removal** and **Constant Propagation**.
- Dead Code Removal, removed variables that were declared on the LHS, but never reused on the RHS.



- Constant propagation suggests variables that can be replaced by their immediate values.
6. Error Handling:
- Our compiler accurately detects invalid syntax, that does not follow the grammar, and neatly prints the line number and the error associated with it.
7. Target Code Generation:
- For our **Target Code Generation** we used a Python script that took as input the **ICG**, and acted as a one-pass assembler that handled variable liveness.
  - Variables that would not be used in the future, would be deallocated from their respective registers, and similarly, variables that would be used soon, would be stored in the registers.

## 6. Implementation Details

1. Symbol Table:
- a. The **Symbol Table**, is defined to be a vector collection of symbol-table entries, called as tokens.
  - b. Our **Symbol Table** was implemented with the following structures:

```
typedef struct token{
    std::string name;
    std::string type;
    std::string value;
    int scope;
    int lineNo;
}token;

typedef struct symbolTable{
    std::vector<token> symTab;
} symbolTable;
```

2. Abstract Syntax Tree:
- a. Our **Abstract Syntax Tree (AST)** was implemented using custom-defined structures.
  - b. The node structure contained a record, a pointer to the equivalent value in the **Symbol Table**, an integer indicating the number of children nodes, a vector of node pointers, containing the children nodes, a value and its associated type, with the following structure:

```
typedef struct node{
    token * record;

    int numNodes;

    std::string value;

    std::string type;

    std::vector<struct node *> ptrVec;
} node;
```

### 3. Intermediate Code Generation:

- a. **Intermediate Code Generation (ICG)** was stored in custom defined Quads, involving symbol table pointers and an operator:

```
/* ICG - Quads representation */
typedef struct quad{
    token * arg1;
    token * arg2;
    token * result;
    std::string op;
} quad;
```

### 4. Code Optimisation:

- a. For our **Code Optimisation** phase, we performed **Dead Code Removal** and **Constant Propagation**, by taking as input the Quads, and using the **Symbol Table**, to count the number of times a variable was used.
- b. The structures used included:

```
typedef struct varCount{
    std::string name;
    std::string value;
    int lhsCount;
    int rhsCount;
} varCount;
```

### 5. Error Handling:

- a. Our compiler used the yyerror along with the yylineno utilities of Yacc, to handle and detect the location of errors, and to print the corresponding error messages.

### 6. Target Code Generation:

- a. For our **Target Code Generation**, (ARM) we used a Python dictionary, to keep track of registers, their variables and their last-time-of-use.

## 7. Results

Our Mini Python Compiler parses Python grammar and generates optimised code for basic Python syntax as well as the target ARM assembly code for the input file.

Our Mini Python Compiler has a few shortcomings. These include:

- The compiler does not take care of functions and classes (as of now).
- `print` statements are not handled in the target code generation phase.
- The output optimized code for a sample input is not accurate at all times. This forces us to use the Intermediate Code generated as the input to our assembler.

## 8. Snapshots

### Test Input 1:

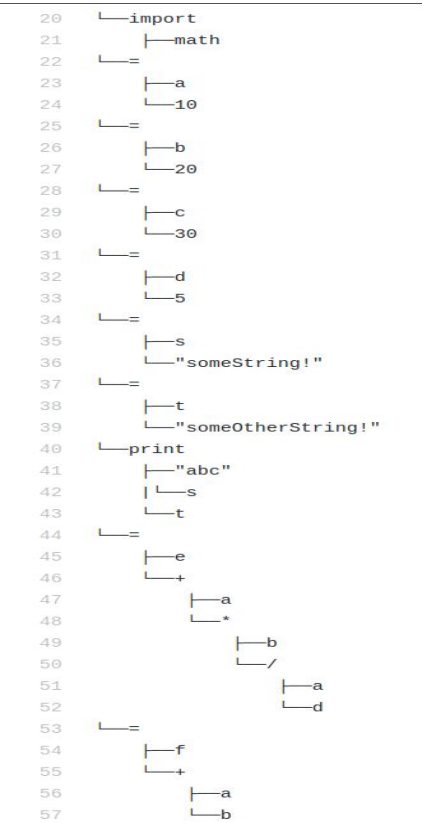
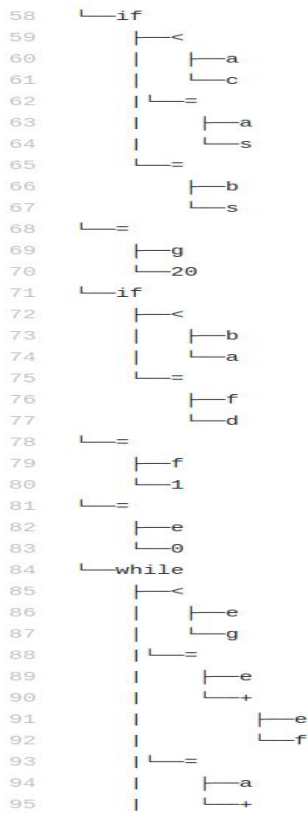
*Sample Input :*

```
1  #import statement
2  import math
3
4  # integer assigns
5  a = 10
6  b = 20
7  c = 30
8  d = 5
9
10 # string assigns
11 s = "someString!"
12 t = "someOtherString!"
13
14 # print statement
15 print("abc", s, t)
16
17 # some operations
18 e = a + b * a / d
19 f = a + b
20
21 # check if control
22 if a < c:
23     a = s
24     b = s
25
26 g = 20
27
28 if b < a:
29     f = d
30
31 f = 1
32 e = 0
33
34 # while loop
35 while e < g:
36     e = e + f
37     a = a + e
38     b = b + a
39
40 x = 15
41 y = a * b + c / d - e + f
42
43 # for loop
44 for char in s:
45     a = a
46
47 # seems like everything is okay?
48 z = "The-End!"
```

Symbol Table :

2	SYMBOL TABLE:					
3	Name	Type	Scope	Line No.	Value	
4	[math		var	0	3	]
5	[a		var	1	53	a]
6	[b		var	0	48	+]
7	[c		var	0	48	30]
8	[d		var	0	48	5]
9	[s		var	0	52	"someString!"]
10	[t		var	0	19	"someOtherString!"]
11	[e		var	0	48	+]
12	[f		var	0	48	1]
13	[g		var	0	42	20]
14	[x		var	0	47	15]
15	[y		var	0	48	*]
16	[char		var	0	52	]
17	[z		var	0	57	"The-End!"]
18						
19	-----					

AST:



```

96      |      |a
97      |      |e
98      |      |
99      |      |b
100     |      |
101     |      |b
102     |      |a
103     |      |
104     |      |x
105     |      |15
106     |      |
107     |      |y
108     |      |*
109     |      |a
110     |      |
111     |      |b
112     |      |
113     |      |c
114     |      |
115     |      |d
116     |      |
117     |      |e
118     |      |f
119     |      |
120     |      |in
121     |      |char
122     |      |s
123     |      |
124     |      |a
125     |      |a
126     |      |
127     |      |z
128     |      |"The-End!"
129     |      |
130     |      |

```

### Intermediate Code:

```

132  Intermediate Code Generation:
133
134  a = 10
135  b = 20
136  c = 30
137  d = 5
138  s = "someString!"
139  t = "someOtherString!"
140  T0 = a / d
141  T1 = b * T0
142  T2 = a + T1
143  e = T2
144  f = a + b
145  ifFalse a < c GOTO L0
146  a = s
147  b = s
148  L0: g = 20
149  ifFalse b < a GOTO L1
150  f = d
151  L1: f = 1
152  e = 0
153  L2: ifFalse e < g GOTO L3
154  e = e + f
155  a = a + e
156  b = b + a
157  goto L2
158  L3: x = 15
159  T3 = e + f
160  T4 = d - T3
161  T5 = c / T4
162  T6 = b + T5
163  T7 = a * T6
164  y = T7
165  L4: ifFalse char in s GOTO L5
166  a = a
167  goto L4
168  L5: z = "The-End!"
169
170  End of Intermediate Code Generation

```

### Target Assembly Code :

```
175  MOV R0, #10
176
177  MOV R1, #20
178
179  MOV R2, #30
180
181  MOV R3, #5
182
183  str: .ascii "someString!"
184  LDR R4, str
185  LDR R5, s
186  LDR R6, [R4]
187  STR R6, [R5]
188
189  str: .ascii "someOtherString!"
190  LDR R4, str
191  LDR R6, t
192  LDR R7, [R4]
193  STR R7, [R6]
194
195  LDR R0, a
196  LDR R4, T0
197  LDR R7, [R0]
198  DIV R7, R7, d
199  STR R7, [R4]

229  LDR R5, s
230  LDR R0, a
231  LDR R10, [R5]
232  STR R10, [R0]
233
234  LDR R5, s
235  LDR R1, b
236  LDR R10, [R5]
237  STR R10, [R1]
238
239  L0:
240  MOV R10, #20
241
242  LDR R1, b
243  LDR R0, a
244  CMP R1, R0
245  BGE L1
246
247  LDR R3, d
248  LDR R8, f
249  LDR R11, [R3]
250  STR R11, [R8]

201  LDR R1, b
202  LDR R7, T1
203  LDR R8, [R1]
204  MUL R8, R8, T0
205  STR R8, [R7]
206
207  LDR R0, a
208  LDR R8, T2
209  LDR R9, [R0]
210  ADD R9, R9, T1
211  STR R9, [R8]
212
213  LDR R8, T2
214  LDR R9, e
215  LDR R10, [R8]
216  STR R10, [R9]
217
218  LDR R0, a
219  LDR R8, f
220  LDR R10, [R0]
221  ADD R10, R10, b
222  STR R10, [R8]
223
224  LDR R0, a
225  LDR R2, c
226  CMP R0, R2
227  BGE L0

252  L1:
253  MOV R8, #1
254
255  MOV R9, #0
256
257  L2:
258  LDR R9, e
259  LDR R10, g
260  CMP R9, R10
261  BGE L3
262
263  LDR R9, e
264  LDR R9, e
265  LDR R11, [R9]
266  ADD R11, R11, f
267  STR R11, [R9]
268  B L2
269
270  LDR R0, a
271  LDR R0, a
272  LDR R11, [R0]
273  ADD R11, R11, e
274  STR R11, [R0]
275
276  LDR R1, b
277  LDR R1, b
278  LDR R11, [R1]
279  ADD R11, R11, a
280  STR R11, [R1]

282  L3:
283  MOV R11, #15
284
285  LDR R9, e
286  LDR R11, T3
287  LDR R12, [R9]
288  ADD R12, R12, f
289  STR R12, [R11]
290
291  LDR R3, d
292  LDR R12, T4
293  LDR R13 t
294  STR R6, [R13]
295  LDR R6, [R3]
296  SUB R6, R6, T3
297  STR R6, [R12]
298
299  LDR R2, c
300  LDR R6, T5
301  LDR R13 T0
302  STR R4, [R13]
303  LDR R4, [R2]
304  DIV R4, R4, T4
305  STR R4, [R6]
```

307	LDR R1, b	341	L5:
308	LDR R4, T6	342	str: .asciiz "The-End!"
309	LDR R13 T1	343	LDR R0, str
310	STR R7, [R13]	344	LDR R1, z
311	LDR R7, [R1]	345	LDR R5, [R0]
312	ADD R7, R7, T5	346	STR R5, [R1]
313	STR R7, [R4]	347	
314		348	LDR R13 f
315	LDR R0, a	349	STR R8, [R13]
316	LDR R1, T7	350	LDR R13 g
317	LDR R7, [R0]	351	STR R10, [R13]
318	MUL R7, R7, T6	352	LDR R13 e
319	STR R7, [R1]	353	STR R9, [R13]
320		354	LDR R13 T3
321	LDR R1, T7	355	STR R11, [R13]
322	LDR R7, y	356	LDR R13 d
323	LDR R13 s	357	STR R3, [R13]
324	STR R5, [R13]	358	LDR R13 T4
325	LDR R5, [R1]	359	STR R12, [R13]
326	STR R5, [R7]	360	LDR R13 c
327		361	STR R2, [R13]
328	LDR R1, #0	362	LDR R13 T5
329	L4:	363	STR R6, [R13]
330	LDR R5, s(R1)	364	LDR R13 T6
331	ADD R1, R1, #1	365	STR R4, [R13]
332	CMP R5, 0x00	366	
333	BEQ L5	367	End of Target Code!
334			
335	LDR R0, a		
336	LDR R0, a		
337	LDR R7, [R0]		
338	STR R7, [R0]		
339	B L4		



Test Input 2:

Sample Input:

1	a = 30
2	b = 20
3	c = 21
4	d = 33
5	e = 45
6	f = 56
7	g = 43
8	h = 33
9	i = 67
10	j = 45
11	k = 57
12	l = 33
13	n = 98
14	o = 77
15	
16	p = a + b + c + d + e + f + g + h + i + j + k + l + m + n + o

Symbol Table:

SYMBOL TABLE:					
Name	Type	Scope	Line No.	Value	
[a		var	0	16	30]
[b		var	0	16	20]
[c		var	0	16	21]
[d		var	0	16	33]
[e		var	0	16	45]
[f		var	0	16	56]
[g		var	0	16	43]
[h		var	0	16	33]
[i		var	0	16	67]
[j		var	0	16	45]
[k		var	0	16	57]
[l		var	0	16	33]
[m		var	0	16	98]
[n		var	0	16	77]
[o		var	0	16	+] ]

AST:

$\perp =$   
     $\vdash a$   
     $\perp_{30}$

$\perp =$   
     $\vdash b$   
     $\perp_{20}$

$\perp =$   
     $\vdash c$   
     $\perp_{21}$

$\perp =$   
     $\vdash d$   
     $\perp_{33}$

$\perp =$   
     $\vdash e$   
     $\perp_{45}$

$\perp =$   
     $\vdash f$   
     $\perp_{56}$

$\perp =$   
     $\vdash g$   
     $\perp_{43}$

$\perp =$   
     $\vdash h$   
     $\perp_{33}$

$\perp =$   
     $\vdash i$   
     $\perp_{67}$

$\perp =$   
     $\vdash j$   
     $\perp_{45}$

$\perp =$   
     $\vdash k$   
     $\perp_{57}$

$\perp =$   
     $\vdash l$   
     $\perp_{33}$

$\perp =$   
     $\vdash m$   
     $\perp_{98}$

$$\begin{array}{r} \text{L} = \\ | \\ \text{L} - n \\ | \\ \text{L} - 98 \end{array}$$
$$\begin{array}{r} \text{L} = \\ | - 0 \\ \text{L} - 77 \end{array}$$
$$\begin{array}{c} \vdash \\ \vdash \\ \vdash \end{array}$$
$$\begin{array}{l} \vdash a \\ \vdash + \end{array}$$
$$\begin{array}{c} | - b \\ | - + \end{array}$$
$$\begin{array}{c} \text{---C} \\ | \\ \text{---+} \end{array}$$
$$\begin{array}{c} | \\ \text{---} d \\ | \\ \text{---} + \end{array}$$
$$\begin{array}{c} | - e \\ | - + \end{array}$$
$$\begin{array}{c} \vdash f \\ \vdash + \end{array}$$
$$\begin{array}{c} \vdash g \\ \vdash + \end{array}$$
$$\begin{array}{c} | - h \\ | + \end{array}$$
$$\begin{array}{c} \text{---}i \\ \text{---}+ \end{array}$$

Hj  
L+

$$\begin{array}{c} | \text{---} k \\ | \text{---} + \end{array}$$
$$\begin{array}{c} \text{H} \\ | \\ \text{H} \end{array}$$

A diagram showing a mass  $m$  suspended from a spring with spring constant  $k$ . The mass is represented by a horizontal line with a vertical line extending downwards from its center, ending in a small circle. The spring is represented by a coiled line extending upwards from the mass, ending in a horizontal line. The spring constant  $k$  is written next to the coiled line.

$$\begin{array}{l} \vdash n \\ \vdash 0 \end{array}$$

### *Intermediate Code:*

Intermediate Code Generation:

```
a = 30
b = 20
c = 21
d = 33
e = 45
f = 56
g = 43
h = 33
i = 67
j = 45
k = 57
l = 33
m = 98
n = 77
T0 = n + o
T1 = m + T0
T2 = l + T1
T3 = k + T2
T4 = j + T3
T5 = i + T4
T6 = h + T5
T7 = g + T6
T8 = f + T7
T9 = e + T8
T10 = d + T9
T11 = c + T10
T12 = b + T11
T13 = a + T12
o = T13
```

End of Intermediate Code Generation

### Target Assembly Code:

```
173
174 LDR R12, m
175 LDR R0, T1
176 LDR R2, [R12]
177 ADD R2, R2, T0
178 STR R2, [R0]
179
180 LDR R11, l
181 LDR R2, T2
182 LDR R12, [R11]
183 ADD R12, R12,
184 STR R12, [R2]
185
186 LDR R10, k
187 LDR R11, T3
188 LDR R12, [R10]
189 ADD R12, R12,
190 STR R12, [R11]
191
192 LDR R9, j
193 LDR R10, T4
194 LDR R12, [R9]
195 ADD R12, R12,
196 STR R12, [R10]
197
198 LDR R8, i
199 LDR R9, T5
200 LDR R12, [R8]
201 ADD R12, R12,
202 STR R12, [R9]
203
204 LDR R7, h
205 LDR R8, T6
206 LDR R12, [R7]
207 ADD R12, R12,
```

```
131 -----
132 Target Code:
133
134 MOV R0, #30
135
136 MOV R1, #20
137
138 MOV R2, #21
139
140 MOV R3, #33
141
142 MOV R4, #45
143
144 MOV R5, #56
145
146 MOV R6, #43
147
148 MOV R7, #33
149
150 MOV R8, #67
151
152 MOV R9, #45
153
154 MOV R10, #57
155
156 MOV R11, #33
157
158 MOV R12, #98
159
160 LDR R13 a
161 STR R0, [R13]
162 MOV R0, #77
163
164 LDR R13 b
165 STR R1, [R13]
166 LDR R0, n
167 LDR R1, T0
168 LDR R13 c
169 STR R2, [R13]
170 LDR R2, [R0]
171 ADD R2, R2, o
172 STR R2, [R1]
173
```

```

216 LDR R5, f
217 LDR R6, T8
218 LDR R12, [R5]
219 ADD R12, R12, T7
220 STR R12, [R6]
221
222 LDR R4, e
223 LDR R5, T9
224 LDR R12, [R4]
225 ADD R12, R12, T8
226 STR R12, [R5]
227
228 LDR R3, d
229 LDR R4, T10
230 LDR R12, [R3]
231 ADD R12, R12, T9
232 STR R12, [R4]
233
234 LDR R3, c
235 LDR R12, T11
236 LDR R13 T0
237 STR R1, [R13]
238 LDR R1, [R3]
239 ADD R1, R1, T10
240 STR R1, [R12]
241
242 LDR R1, b
243 LDR R3, T12
244 LDR R13 T1
245 STR R0, [R13]
246 LDR R0, [R1]
247 ADD R0, R0, T11
248 STR R0, [R3]
249

```

```

249
250 LDR R0, a
251 LDR R1, T13
252 LDR R13 T2
253 STR R2, [R13]
254 LDR R2, [R0]
255 ADD R2, R2, T12
256 STR R2, [R1]
257
258 LDR R1, T13
259 LDR R0, o
260 LDR R2, [R1]
261 STR R2, [R0]
262
263 LDR R13 T3
264 STR R11, [R13]
265 LDR R13 T4
266 STR R10, [R13]
267 LDR R13 T5
268 STR R9, [R13]
269 LDR R13 T6
270 STR R8, [R13]
271 LDR R13 T7
272 STR R7, [R13]
273 LDR R13 T8
274 STR R6, [R13]
275 LDR R13 T9
276 STR R5, [R13]
277 LDR R13 T10
278 STR R4, [R13]
279 LDR R13 T11
280 STR R12, [R13]
281 LDR R13 T12
282 STR R3, [R13]
283
284 End of Target Code!

```

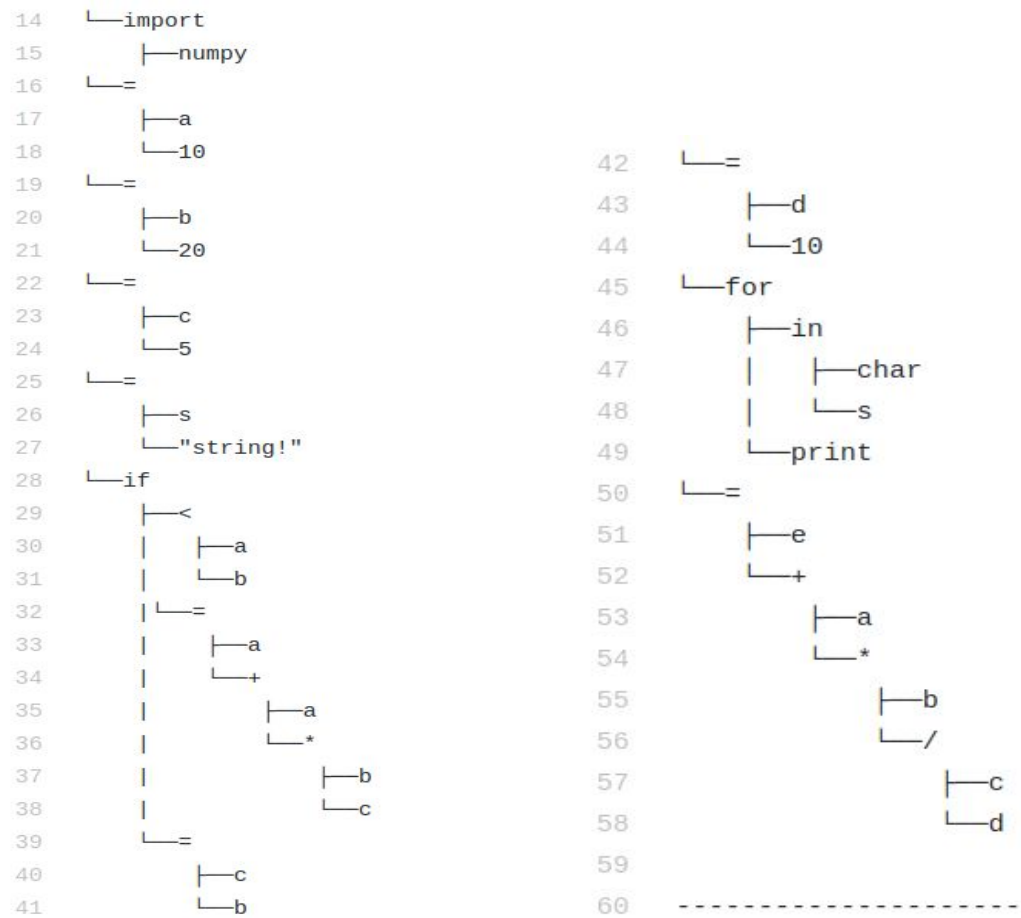
### Test Input 3:

```
1  # this is a test comment, to be removed by our compiler
2  import numpy
3
4  a = 10
5  b = 20
6  c = 5
7  s = "string!"
8
9  """
10 this is a test
11 multiline comment
12 to be removed
13 """
14
15 if a < b:
16     a = a + b * c
17     c = b
18
19 d = 10
20
21 for char in s:
22     print(char)
23
24 e = a + b * c / d
```

*Symbol Table:*

2	SYMBOL TABLE:					
3	Name	Type	Scope	Line No.	Value	
4	[numpy		var	0	3	]
5	[a		var	0	21	+] ]
6	[b		var	0	21	20]
7	[c		var	0	21	b]
8	[s		var	0	18	"string!"]
9	[d		var	0	21	10]
10	[char	var		1	19	]
11	[e		var	0	21	+] ]

*AST:*





### Intermediate Code:

```
62  Intermediate Code Generation:
63
64  a = 10
65  b = 20
66  c = 5
67  s = "string!"
68  ifFalse a < b GOTO L0
69  T0 = b * c
70  T1 = a + T0
71  a = T1
72  c = b
73  L0: d = 10
74  L1: ifFalse char in s GOTO L2
75  goto L1
76  L2: T2 = c / d
77  T3 = b * T2
78  T4 = a + T3
79  e = T4
80
81  End of Intermediate Code Generation
```

### Target/Assembly code:

86	MOV R0, #10	115	LDR R5, T1	149	LDR R0, a
87		116	LDR R0, a	150	LDR R1, T4
88	MOV R1, #20	117	LDR R6, [R5]	151	LDR R6, [R0]
89		118	STR R6, [R0]	152	ADD R6, R6, T3
90	MOV R2, #5	119		153	STR R6, [R1]
91		120	LDR R1, b	154	
92	str: .ascii "string!"	121	LDR R2, c	155	LDR R1, T4
93	LDR R3, str	122	LDR R5, [R1]	156	LDR R0, e
94	LDR R4, s	123	STR R5, [R2]	157	LDR R6, [R1]
95	LDR R5, [R3]	124		158	STR R6, [R0]
96	STR R5, [R4]	125	L0:	159	
97		126	MOV R5, #10	160	LDR R13 s
98	LDR R0, a	127		161	STR R4, [R13]
99	LDR R1, b	128	LDR R6, #0	162	LDR R13 T0
100	CMP R0, R1	129	L1:	163	STR R3, [R13]
101	BGE L0	130	LDR R7, s(R6)	164	LDR R13 d
102		131	ADD R6, R6, #1	165	STR R5, [R13]
103	LDR R1, b	132	CMP R7, 0x00	166	LDR R13 T2
104	LDR R3, T0	133	BEQ L2	167	STR R8, [R13]
105	LDR R5, [R1]	134		168	LDR R13 T3
106	MUL R5, R5, c	135	L2:	169	STR R2, [R13]
107	STR R5, [R3]	136	LDR R2, c	170	
108		137	LDR R8, T2	171	End of Target Code!
109	LDR R0, a	138	LDR R9, [R2]		
110	LDR R5, T1	139	DIV R9, R9, d		
111	LDR R6, [R0]	140	STR R9, [R8]		
112	ADD R6, R6, T0	141	B L1		
113	STR R6, [R5]	142			
		143	LDR R1, b		
		144	LDR R2, T3		
		145	LDR R6, [R1]		
		146	MUL R6, R6, T2		
		147	STR R6, [R2]		

## 9. Conclusions

A mini-compiler for **Python3** was created using C++. It handles constructs like if, for and while. It handles imports and print statements as well.

Basic error handling is also taken care of and gives required information as to where the error has occurred.

## 10. Further Enhancements

Our Mini-Compiler can be further enhanced by adding

- Support for functions and classes.
- More efficient Optimization Techniques.
- Error Recovery.