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:

Anantharam R U	**PES1201700088**
Gaurang Rao	**PES1201701103**
Shashank Prasad	**PES1201700667**

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

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]
```

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]
```

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-zA-Z][_a-zA-Z0-9]*
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

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 :

```
#import statement
import math

# integer assigns
a = 10
b = 20
c = 30
d = 5

# string assigns
s = "someString!"
t = "someOtherString!"

# print statement
print("abc", s, t)

# some operations
e = a + b * a / d
f = a + b

# check if control
if a < c:
    a = s
    b = s

g = 20

if b < a:
    f = d

f = 1
e = 0

# while loop
while e < g:
    e = e + f
    a = a + e
    b = b + a

x = 15
y = a * b + c / d - e + f

# for loop
for char in s:
    a = a

# seems like everything is okay?
z = "The-End!"
```

Symbol Table :

```
SYMBOL TABLE:
Name    Type    Scope    Line No. Value
[math            var              0              3              ]
[a               var              1              53             a]
[b               var              0              48             +]
[c               var              0              48             30]
[d               var              0              48             5]
[s               var              0              52             "someString!"]
[t               var              0              19             "someOtherString!"]
[e               var              0              48             +]
[f               var              0              48             1]
[g               var              0              42             20]
[x               var              0              47             15]
[y               var              0              48             *]
[char            var              0              52             ]
[z               var              0              57             "The-End!"]

---------------------
```

AST :

```
└──if
    ├──<
    |   ├──a
    |   └──c
    |└──=
    |   ├──a
    |   └──s
    └──=
        ├──b
        └──s
└──=
    ├──g
    └──20
└──if
    ├──<
    |   ├──b
    |   └──a
    └──=
        ├──f
        └──d
└──=
    ├──f
    └──1
└──=
    ├──e
    └──0
└──while
    ├──<
    |   ├──e
    |   └──g
    |└──=
    |   ├──e
    |   └──+
    |       ├──e
    |       └──f
    |└──=
    |   ├──a
    |   └──+
```

```
└──import
    ├──math
└──=
    ├──a
    └──10
└──=
    ├──b
    └──20
└──=
    ├──c
    └──30
└──=
    ├──d
    └──5
└──=
    ├──s
    └──"someString!"
└──=
    ├──t
    └──"someOtherString!"
└──print
    ├──"abc"
    |└──s
    └──t
└──=
    ├──e
    └──+
        ├──a
        └──*
            ├──b
            └──/
                ├──a
                └──d
└──=
    ├──f
    └──+
        ├──a
        └──b
```

```
    |       ├─a
    |       └─e
    └─=
        ├─b
        └─+
            ├─b
            └─a
└─=
    ├─x
    └─15
└─=
    ├─y
    └─*
        ├─a
        └─+
            ├─b
            └─/
                ├─c
                └─-
                    ├─d
                    └─+
                        ├─e
                        └─f
└─for
    ├─in
    |   ├─char
    |   └─s
    └─=
        ├─a
        └─a
└─=
    ├─z
    └─"The-End!"

--------------------
```

Intermediate Code:

```
Intermediate Code Generation:

a = 10
b = 20
c = 30
d = 5
s = "someString!"
t = "someOtherString!"
T0 = a / d
T1 = b * T0
T2 = a + T1
e = T2
f = a + b
ifFalse a < c GOTO L0
a = s
b = s
L0: g = 20
ifFalse b < a GOTO L1
f = d
L1: f = 1
e = 0
L2: ifFalse e < g GOTO L3
e = e + f
a = a + e
b = b + a
goto L2
L3: x = 15
T3 = e + f
T4 = d - T3
T5 = c / T4
T6 = b + T5
T7 = a * T6
y = T7
L4: ifFalse char in s GOTO L5
a = a
goto L4
L5: z = "The-End!"

End of Intermediate Code Generation
```

Target Assembly Code :

```
MOV R0, #10

MOV R1, #20

MOV R2, #30

MOV R3, #5

str: .asciiz "someString!"
LDR R4, str
LDR R5, s
LDR R6, [R4]
STR R6, [R5]

str: .asciiz "someOtherString!"
LDR R4, str
LDR R6, t
LDR R7, [R4]
STR R7, [R6]

LDR R0, a
LDR R4, T0
LDR R7, [R0]
DIV R7, R7, d
STR R7, [R4]

LDR R1, b
LDR R7, T1
LDR R8, [R1]
MUL R8, R8, T0
STR R8, [R7]

LDR R0, a
LDR R8, T2
LDR R9, [R0]
ADD R9, R9, T1
STR R9, [R8]

LDR R8, T2
LDR R9, e
LDR R10, [R8]
STR R10, [R9]

LDR R0, a
LDR R8, f
LDR R10, [R0]
ADD R10, R10, b
STR R10, [R8]

LDR R0, a
LDR R2, c
CMP R0, R2
BGE L0

LDR R5, s
LDR R0, a
LDR R10, [R5]
STR R10, [R0]

LDR R5, s
LDR R1, b
LDR R10, [R5]
STR R10, [R1]

L0:
MOV R10, #20

LDR R1, b
LDR R0, a
CMP R1, R0
BGE L1

LDR R3, d
LDR R8, f
LDR R11, [R3]
STR R11, [R8]

L1:
MOV R8, #1

MOV R9, #0

L2:
LDR R9, e
LDR R10, g
CMP R9, R10
BGE L3

LDR R9, e
LDR R9, e
LDR R11, [R9]
ADD R11, R11, f
STR R11, [R9]
B L2

LDR R0, a
LDR R0, a
LDR R11, [R0]
ADD R11, R11, e
STR R11, [R0]

LDR R1, b
LDR R1, b
LDR R11, [R1]
ADD R11, R11, a
STR R11, [R1]

L3:
MOV R11, #15

LDR R9, e
LDR R11, T3
LDR R12, [R9]
ADD R12, R12, f
STR R12, [R11]

LDR R3, d
LDR R12, T4
LDR R13 t
STR R6, [R13]
LDR R6, [R3]
SUB R6, R6, T3
STR R6, [R12]

LDR R2, c
LDR R6, T5
LDR R13 T0
STR R4, [R13]
LDR R4, [R2]
DIV R4, R4, T4
STR R4, [R6]
```

```
LDR R1, b
LDR R4, T6
LDR R13 T1
STR R7, [R13]
LDR R7, [R1]
ADD R7, R7, T5
STR R7, [R4]

LDR R0, a
LDR R1, T7
LDR R7, [R0]
MUL R7, R7, T6
STR R7, [R1]

LDR R1, T7
LDR R7, y
LDR R13 s
STR R5, [R13]
LDR R5, [R1]
STR R5, [R7]

LDR R1, #0
L4:
LDR R5, s(R1)
ADD R1, R1, #1
CMP R5, 0x00
BEQ L5

LDR R0, a
LDR R0, a
LDR R7, [R0]
STR R7, [R0]
B L4
```

```
L5:
str: .asciiz "The-End!"
LDR R0, str
LDR R1, z
LDR R5, [R0]
STR R5, [R1]

LDR R13 f
STR R8, [R13]
LDR R13 g
STR R10, [R13]
LDR R13 e
STR R9, [R13]
LDR R13 T3
STR R11, [R13]
LDR R13 d
STR R3, [R13]
LDR R13 T4
STR R12, [R13]
LDR R13 c
STR R2, [R13]
LDR R13 T5
STR R6, [R13]
LDR R13 T6
STR R4, [R13]

End of Target Code!
```

Test Input 2:

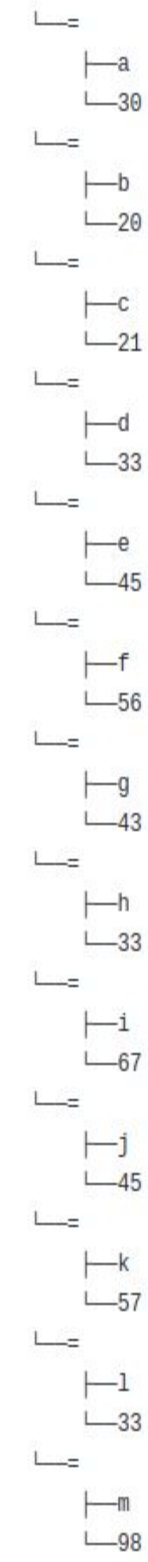
Sample Input:

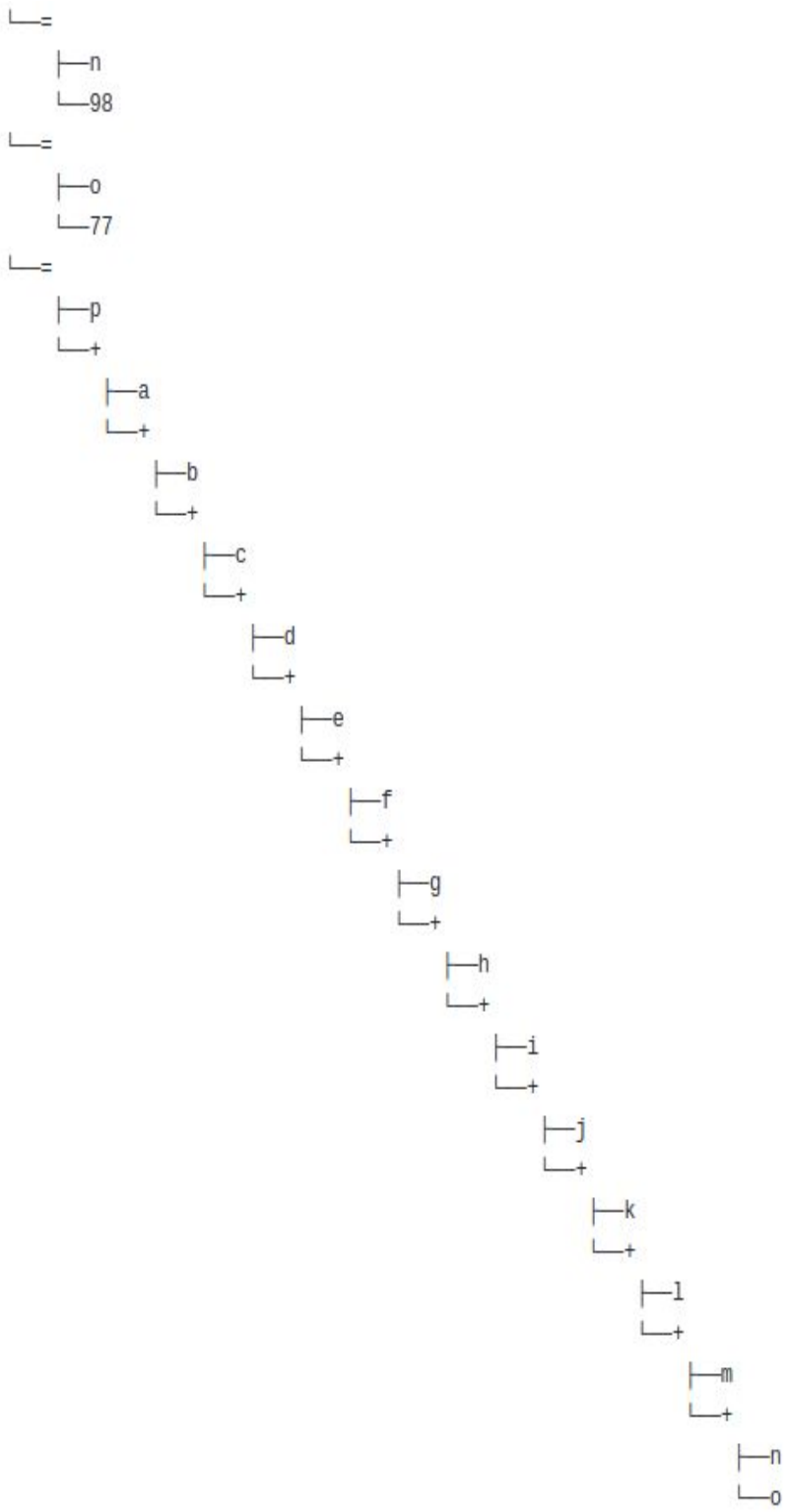
```
a = 30
b = 20
c = 21
d = 33
e = 45
f = 56
g = 43
h = 33
i = 67
j = 45
k = 57
l = 33
n = 98
o = 77

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:




=
n
98
=
o
77
=
p
+
a
+
b
+
c
+
d
+
e
+
f
+
g
+
h
+
i
+
j
+
k
+
l
+
m
+
n
o

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
```

```
LDR R5, f
LDR R6, T8
LDR R12, [R5]
ADD R12, R12, T7
STR R12, [R6]

LDR R4, e
LDR R5, T9
LDR R12, [R4]
ADD R12, R12, T8
STR R12, [R5]

LDR R3, d
LDR R4, T10
LDR R12, [R3]
ADD R12, R12, T9
STR R12, [R4]

LDR R3, c
LDR R12, T11
LDR R13 T0
STR R1, [R13]
LDR R1, [R3]
ADD R1, R1, T10
STR R1, [R12]

LDR R1, b
LDR R3, T12
LDR R13 T1
STR R0, [R13]
LDR R0, [R1]
ADD R0, R0, T11
STR R0, [R3]

LDR R0, a
LDR R1, T13
LDR R13 T2
STR R2, [R13]
LDR R2, [R0]
ADD R2, R2, T12
STR R2, [R1]

LDR R1, T13
LDR R0, o
LDR R2, [R1]
STR R2, [R0]

LDR R13 T3
STR R11, [R13]
LDR R13 T4
STR R10, [R13]
LDR R13 T5
STR R9, [R13]
LDR R13 T6
STR R8, [R13]
LDR R13 T7
STR R7, [R13]
LDR R13 T8
STR R6, [R13]
LDR R13 T9
STR R5, [R13]
LDR R13 T10
STR R4, [R13]
LDR R13 T11
STR R12, [R13]
LDR R13 T12
STR R3, [R13]

End of Target Code!
```

Test Input 3:

```
# this is a test comment, to be removed by our compiler
import numpy

a = 10
b = 20
c = 5
s = "string!"

"""
this is a test
multiline comment
to be removed
"""

if a < b:
    a = a + b * c
    c = b

d = 10

for char in s:
    print(char)

e = a + b * c / d
```

Symbol Table:

```
SYMBOL TABLE:
Name   Type   Scope   Line No. Value
[numpy          var             0               3               ]
[a              var             0               21              +]
[b              var             0               21              20]
[c              var             0               21              b]
[s              var             0               18              "string!"]
[d              var             0               21              10]
[char   var             1               19              ]
[e              var             0               21              +]
```

AST:

```
└──import
    ├──numpy
└──=
    ├──a
    └──10
└──=
    ├──b
    └──20
└──=
    ├──c
    └──5
└──=
    ├──s
    └──"string!"
└──if
    ├──<
    |   ├──a
    |   └──b
    |└──=
    |    ├──a
    |    └──+
    |        ├──a
    |        └──*
    |            ├──b
    |            └──c
    └──=
        ├──c
        └──b
└──=
    ├──d
    └──10
└──for
    ├──in
    |   ├──char
    |   └──s
    └──print
└──=
    ├──e
    └──+
        ├──a
        └──*
            ├──b
            └──/
                ├──c
                └──d

----------------------
```

Intermediate Code:

```
Intermediate Code Generation:

a = 10
b = 20
c = 5
s = "string!"
ifFalse a < b GOTO L0
T0 = b * c
T1 = a + T0
a = T1
c = b
L0: d = 10
L1: ifFalse char in s GOTO L2
goto L1
L2: T2 = c / d
T3 = b * T2
T4 = a + T3
e = T4

End of Intermediate Code Generation
```

Target/Assembly code:

```
MOV R0, #10

MOV R1, #20

MOV R2, #5

str: .asciiz "string!"
LDR R3, str
LDR R4, s
LDR R5, [R3]
STR R5, [R4]

LDR R0, a
LDR R1, b
CMP R0, R1
BGE L0

LDR R1, b
LDR R3, T0
LDR R5, [R1]
MUL R5, R5, c
STR R5, [R3]

LDR R0, a
LDR R5, T1
LDR R6, [R0]
ADD R6, R6, T0
STR R6, [R5]

LDR R5, T1
LDR R0, a
LDR R6, [R5]
STR R6, [R0]

LDR R1, b
LDR R2, c
LDR R5, [R1]
STR R5, [R2]

L0:
MOV R5, #10

LDR R6, #0
L1:
LDR R7, s(R6)
ADD R6, R6, #1
CMP R7, 0x00
BEQ L2

L2:
LDR R2, c
LDR R8, T2
LDR R9, [R2]
DIV R9, R9, d
STR R9, [R8]
B L1

LDR R1, b
LDR R2, T3
LDR R6, [R1]
MUL R6, R6, T2
STR R6, [R2]

LDR R0, a
LDR R1, T4
LDR R6, [R0]
ADD R6, R6, T3
STR R6, [R1]

LDR R1, T4
LDR R0, e
LDR R6, [R1]
STR R6, [R0]

LDR R13 s
STR R4, [R13]
LDR R13 T0
STR R3, [R13]
LDR R13 d
STR R5, [R13]
LDR R13 T2
STR R8, [R13]
LDR R13 T3
STR R2, [R13]

End of Target Code!
```

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.