Report on

"Python Compiler Using C"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Sanjay Chari	**PES1201700278**
Aditya Shankaran	**PES1201700710**
Athul Sandosh	**PES1201701110**

Under the guidance of

Mr. Kiran P
Assistant 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

TABLE OF CONTENTS

INTRODUCTION

The project showcases a mini compiler coded using lex and yacc; that compiles Python3 code. We have implemented if-elif-else constructs, while loops, and functions.

Sample Input :

```
#Basic Code

import scipy
def f():
	x = 1

x = 2
y = 1

a = 3
b = 4
c = 2
d = a+b

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

Output (Assembly Code) :

```
MOV R1, 2
STR R1, T10
MOV R2, R1
STR R2, x
MOV R3, 1
STR R3, T13
MOV R4, R3
STR R4, y
MOV R5, 3
STR R5, T16
MOV R6, R5
STR R6, a
MOV R7, 4
STR R7, T19
MOV R8, R7
STR R8, b
MOV R9, 2
```

```
STR R9, T22
MOV R10, R9
STR R10, c
MOV R11, R6
STR R11, T25
MOV R12, R8
STR R12, T26
ADD R13, R11, R12
STR R13, T27
MOV R14, R13
STR R14, d
MOV R1, R2
STR R1, T30
MOV R3, 1
STR R3, T31
CMP R1, R3
BE Label1
MOV R4, 0
B Label2
Label1 :MOV R4, 1
Label2 :STR R4, T32
```

ARCHITECTURE OF LANGUAGE

We have handled the following aspects in the syntax and semantics of the Python anguage :

1. Assignment Operations
2. Arithmetic and Relational Operators
3. If-elif-else constructs
4. while loops
5. for loops
6. Function definitions and calls
7. Single Line and Multi Line Comments
8. pass statement
9. break statement

In semantics, we implemented the following aspects :

1. Scope : We identified the scope of the variables using indent and dedent, as is followed in the Python3 Interpreter.
2. Values : We included the values of the variables in the symbol table of our compiler.
3. Type of Keyword : We identified a given keyword as function, variable, or parameter in the symbol table of our compiler.

LITERATURE SURVEY

1. http://dinosaur.compilertools.net/
2. GeeksforGeeks : https://www.geeksforgeeks.org/
3. Lex and Yacc Tutorial by Tom Niemann : https://www.isi.edu/~pedro/Teaching/CSCI565-Fall15/Materials/LexAndYaccTutorial.pdf
4. https://www.javatpoint.com/code-generation
5. https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lects/final.codegen.pdf

CONTEXT FREE GRAMMAR

```
constant : T_Number {insertRecord("Constant", $<text>1, @1.first_line,
currentScope); $$ = createID_Const("Constant", $<text>1, currentScope);}
        | T_String {insertRecord("Constant", $<text>1, @1.first_line, currentScope); $$
= createID_Const("Constant", $<text>1, currentScope);};

term : T_ID {modifyRecordID("Identifier", $<text>1, @1.first_line, currentScope); $$
= createID_Const("Identifier", $<text>1, currentScope);}
    | constant {$$ = $1;}
    | list_index {$$ = $1;};


list_index : T_ID T_OB constant T_CB {checkList($<text>1, @1.first_line,
currentScope); $$ = createOp("ListIndex", 2, createID_Const("ListTypeID",
$<text>1, currentScope), $3);};

StartParse : T_NL StartParse {$$=$2;}| finalStatements T_NL {resetDepth();}
StartParse {$$ = createOp("NewLine", 2, $1, $4);}| finalStatements T_NL {$$=$1;};

basic_stmt : pass_stmt {$$=$1;}
          | break_stmt {$$=$1;}
          | import_stmt {$$=$1;}
          | assign_stmt {$$=$1;}
          | arith_exp {$$=$1;}
          | bool_exp {$$=$1;}
          | print_stmt {$$=$1;}
          | return_stmt {$$=$1;};

arith_exp : term {$$=$1;}
         | arith_exp  T_PL  arith_exp {$$ = createOp("+", 2, $1, $3);}
         | arith_exp  T_MN  arith_exp {$$ = createOp("-", 2, $1, $3);}
         | arith_exp  T_ML  arith_exp {$$ = createOp("*", 2, $1, $3);}
         | arith_exp  T_DV  arith_exp {$$ = createOp("/", 2, $1, $3);}
         | T_MN arith_exp {$$ = createOp("-", 1, $2);}
         | T_OP arith_exp T_CP {$$ = $2;} ;


bool_exp : bool_term T_Or bool_term {$$ = createOp("or", 2, $1, $3);}
        | arith_exp T_LT arith_exp {$$ = createOp("<", 2, $1, $3);}
        | bool_term T_And bool_term {$$ = createOp("and", 2, $1, $3);}
        | arith_exp T_GT arith_exp {$$ = createOp(">", 2, $1, $3);}
```

```
        | arith_exp T_ELT arith_exp {$$ = createOp("<=", 2, $1, $3);}
        | arith_exp T_EGT arith_exp {$$ = createOp(">=", 2, $1, $3);}
        | arith_exp T_In T_ID {checkList($<text>3, @3.first_line, currentScope); $$ =
createOp("in", 2, $1, createID_Const("Constant", $<text>3, currentScope));}
        | bool_term {$$=$1;};

bool_term : bool_factor {$$ = $1;}
        | arith_exp T_EQ arith_exp {$$ = createOp("==", 2, $1, $3);}
        | T_True {insertRecord("Constant", "True", @1.first_line, currentScope); $$ =
createID_Const("Constant", "True", currentScope);}
        | T_False {insertRecord("Constant", "False", @1.first_line, currentScope); $$
= createID_Const("Constant", "False", currentScope);};

bool_factor : T_Not bool_factor {$$ = createOp("!", 1, $2);}
          | T_OP bool_exp T_CP {$$ = $2;};

import_stmt : T_Import T_ID {insertRecord("PackageName", $<text>2, @2.first_line,
currentScope); $$ = createOp("import", 1, createID_Const("PackageName",
$<text>2, currentScope));};
pass_stmt : T_Pass {$$ = createOp("pass", 0);};
break_stmt : T_Break {$$ = createOp("break", 0);};
return_stmt : T_Return {$$ = createOp("return", 0);};;

assign_stmt : T_ID T_EQL arith_exp {insertRecord("Identifier", $<text>1,
@1.first_line, currentScope); $$ = createOp("=", 2, createID_Const("Identifier",
$<text>1, currentScope), $3);}
          | T_ID T_EQL bool_exp {insertRecord("Identifier", $<text>1, @1.first_line,
currentScope);$$ = createOp("=", 2, createID_Const("Identifier", $<text>1,
currentScope), $3);}
          | T_ID  T_EQL func_call {insertRecord("Identifier", $<text>1, @1.first_line,
currentScope); $$ = createOp("=", 2, createID_Const("Identifier", $<text>1,
currentScope), $3);}
          | T_ID T_EQL T_OB T_CB {insertRecord("ListTypeID", $<text>1,
@1.first_line, currentScope); $$ = createID_Const("ListTypeID", $<text>1,
currentScope);} ;

print_stmt : T_Print T_OP term T_CP {$$ = createOp("Print", 1, $3);};

finalStatements : basic_stmt {$$ = $1;}
              | cmpd_stmt {$$ = $1;}
              | func_def {$$ = $1;}
              | func_call {$$ = $1;}
              | error T_NL {yyerrok; yyclearin; $$=createOp("SyntaxError", 0);};

cmpd_stmt : if_stmt {$$ = $1;}
```

```
        | while_stmt {$$ = $1;};


if_stmt : T_If bool_exp T_Cln start_suite {$$ = createOp("If", 2, $2, $4);}
      | T_If bool_exp T_Cln start_suite elif_stmts {$$ = createOp("If", 3, $2, $4, $5);};

elif_stmts : else_stmt {$$= $1;}
        | T_Elif bool_exp T_Cln start_suite elif_stmts {$$= createOp("Elif", 3, $2, $4,
$5);};

else_stmt : T_Else T_Cln start_suite {$$ = createOp("Else", 1, $3);};

while_stmt : T_While bool_exp T_Cln start_suite {$$ = createOp("While", 2, $2,
$4);};

start_suite : basic_stmt {$$ = $1;}
         | T_NL ID {initNewTable($<depth>2); updateCScope($<depth>2);}
finalStatements suite {$$ = createOp("BeginBlock", 2, $4, $5);};

suite : T_NL ND finalStatements suite {$$ = createOp("Next", 2, $3, $4);}
     | T_NL end_suite {$$ = $2;};

end_suite : DD {updateCScope($<depth>1);} finalStatements {$$ =
createOp("EndBlock", 1, $3);}
        | DD {updateCScope($<depth>1);} {$$ = createOp("EndBlock", 0);}
        | {$$ = createOp("EndBlock", 0); resetDepth();};

args : T_ID {addToList($<text>1, 1);} args_list {$$ = createOp(argsList, 0);
clearArgsList();}
    | {$$ = createOp("Void", 0);};

args_list : T_Comma T_ID {addToList($<text>2, 0);} args_list | ;


call_list : T_Comma term {addToList($<text>1, 0);} call_list | ;

call_args : T_ID {addToList($<text>1, 1);} call_list {$$ = createOp(argsList, 0);
clearArgsList();}
                                    | T_Number {addToList($<text>1, 1);} call_list {$$
= createOp(argsList, 0); clearArgsList();}
                                    | T_String {addToList($<text>1, 1);} call_list {$$ =
createOp(argsList, 0); clearArgsList();}
                                    | {$$ = createOp("Void", 0);};
```

func_def : T_Def T_ID {insertRecord("Func_Name", $<text>2, @2.first_line, currentScope);} T_OP args T_CP T_Cln start_suite {$$ = createOp("Func_Name", 3, createID_Const("Func_Name", $<text>2, currentScope), $5, $8);};

func_call : T_ID T_OP call_args T_CP {$$ = createOp("Func_Call", 2, createID_Const("Func_Name", $<text>1, currentScope), $3);};

DESIGN STRATEGY

1. Symbol Table Creation : In our lex code, we create tokens for each of the characters encountered in the code. According to the regular expressions, tokens are created. Identifiers are added to the symbol table with additional information like their type, scope, first occurrence line number, last occurrence line number, and value. We find the value of the variable using yacc.
To find the type of the keyword, we analysed the context in which it occurs.

- If an identifier is preceded by "def", it is tagged as a function.
- If an identifier is found within the scope of a function's declaration statement, it is tagged as a parameter.
- If an identifier is found otherwise, it is a variable.

To find the scope of a variable, we analysed the INDENT and DEDENT tokens that we generate in the lex file, using yacc.
To find first occurrence and last occurrence line number, we utilised a yylineno variable. The first occurrence line is set the first time a variable is encountered; and the last occurrence line number is reset everytime it is encountered again.

2. 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. 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.

3. Intermediate Code Generation : The intermediate code is generated by recursively traversing through the AST. We generate a three address code in this manner.

4. Code Optimisation :

- Dead Code Elimination : Any fragment of code that is not used anywhere else, like a function that is never called; is removed from the intermediate code.
- Reordering Statements : For example, in the case of a constant assigned to a variable inside a loop; the assignment can be brought outside the loop in order to improve the efficiency of the code.

5. Error Handling : We implement panic mode of error handling in our compiler. When an error is encountered in the code, parsing is stopped and the error is reported to the user.

6. Assembly Code Generation :
- We utilise lex and yacc, with a logic akin to LRU cache for assignment of registers, to convert our optimised code to assembly code.
- We replace assignment statements with MOV/STR instructions depending on the context; and replace arithmetic operators with the appropriate instructions(ADD for +, SUB for -, MUL for * etc.).
- We replace ifFalse.. goto… statements with CMP and B/BLE/BGE/BLT/BGT; depending on the context.

IMPLEMENTATION DETAILS

1. Symbol Table Creation :
    - Tools Used : Lex, Yacc
    - Data Structure Used : Array
2. Abstract Syntax Tree :
    - Tools Used : Lex, Yacc
    - Data Structure Used : Tree
3. Intermediate Code Generation :
    - Tools Used : Lex, Yacc
    - Data Structure Used : Array
4. Code Optimization :
    - Tools Used : Lex, Yacc
    - Data Structure Used : Array
5. Error Handling :
    - Tools Used : Lex, Yacc
    - Data Structure Used : -
    - Algorithm Used : Panic Mode
6. Assembly Code Generation :
    - Tools Used : Lex, Yacc
    - Data Structure Used : Array

Instructions to build and run the program :

1. Run
```
$git clone https://github.com/sanjaychari/CD_Project
$cd CD_Project
```

2. To view the symbol table, run
```
$cd Symbol_table
$lex phase1_finals.l
$gcc lex.yy.c -ll
$./a.out > symbol_table.txt
```

The generated tokens can be found in tokens.txt, and the symbol table is in symbol_table.txt.

3. To view only the AST, run

```
$cd ../AST
$lex grammar.l
$yacc -d grammar.y
$gcc lex.yy.c y.tab.c -ll
$./a.out<TestInput1.txt > output.txt
```

4. To view AST with ICG, run

```
$cd ../AST_With_ICG
$lex grammar.l
$yacc -d grammar.y
$gcc lex.yy.c y.tab.c -ll
$./a.out<TestInput1.txt > output.txt
```

5. To view AST with ICG and code optimisation, run

```
$cd ../Code_Opt
$lex grammar.l
$yacc -d grammar.y
$gcc lex.yy.c y.tab.c -ll
$./a.out<input2.py > output.txt
```

6. To view assembly code,

```
$lex assembly.l && yacc -dv assembly.y
$gcc lex.yy.c y.tab.c
$./a.out < ICG1.txt
```

RESULTS AND SHORTCOMINGS

The result achieved is that we have a mini compiler which parses grammar corresponding to basic python syntax and finally generates python code.

Some of the shortcomings of our software is :

1. Segmentation Faults can occur sometimes, depending on the input given to the program.

2. Array has been used as the data structure for symbol table. A hash table would've given better time efficiency.

3. The output of the optimised code has to be manually copied into a text file to be passed as input to the assembly code generation program.

SNAPSHOTS

Tokens

Token	Line Number	Type
import	3	Keyword
	3	Whitespace
scipy	3	Identifier
def	4	Keyword
	4	Whitespace
f	4	Identifier
(	4	LBracket
)	4	RBracket
:	4	Colon
	5	Tab
x	5	Identifier
	5	Whitespace
=	5	Equals
	5	Whitespace
1	5	Literal
x	7	Identifier
	7	Whitespace
=	7	Equals
	7	Whitespace
2	7	Literal
y	8	Identifier
	8	Whitespace
=	8	Equals
	8	Whitespace
1	8	Literal
a	10	Identifier
	10	Whitespace
=	10	Equals
	10	Whitespace
3	10	Literal
b	11	Identifier
	11	Whitespace
=	11	Equals
	11	Whitespace
4	11	Literal
c	12	Identifier
	12	Whitespace
=	12	Equals
	12	Whitespace
2	12	Literal
d	13	Identifier
	13	Whitespace
=	13	Equals
	13	Whitespace
a	13	Identifier
+	13	Plus
b	13	Identifier
if	15	Keyword
(	15	LBracket
x	15	Identifier
=	15	Equals
=	15	Equals
1	15	Literal
)	15	RBracket
:	15	Colon
	16	Tab
c	16	Identifier
=	16	Equals
1	16	Literal
elif	17	Keyword
(	17	LBracket
y	17	Identifier
=	17	Equals
=	17	Equals
1	17	Literal
)	17	RBracket
:	17	Colon
	18	Tab

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

Symbol Table

Name	Class	Scope	Declared Line Number	Latest Occurence Line Number	Value
scipy	Variable	0	3	3	
f	Function	0	4	4	
x	Variable	1	5	15	1
y	Variable	0	8	17	1
a	Variable	0	10	13	3
b	Variable	0	11	13	4
c	Variable	0	12	20	1
d	Variable	0	13	13	

Abstract Syntax Tree

```
NewLine(2)
import(1) NewLine(2)
scipy  Func_Name(3) NewLine(2)
     f  Void  BeginBlock(2) =(2) NewLine(2)
          =(2) EndBlock  x  2  =(2) NewLine(2)
                    x  1  y  1  =(2) NewLine(2)
                        a  3  =(2) NewLine(2)
                          b  4  =(2) NewLine(2)
                            c  2  =(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
```

Optimised Intermediate Code

```
import scipy
T10 = 2
x = T10
T13 = 1
y = T13
T16 = 3
a = T16
T19 = 4
b = T19
T22 = 2
c = T22
T25 = a
T26 = b
T27 = T25 + T26
d = T27
T30 = x
T31 = 1
T32 = T30 == T31
If False T32 goto L0
T33 = 1
c = T33
goto L1
L0: T38 = y
T39 = 1
T40 = T38 == T39
If False T40 goto L0
T41 = 2
c = T41
goto L1
L0: T46 = 1
c = T46
L1: L1:
```

Updated Symbol Table

Scope	Name	Type	Declaration	Last Used Line
(0, 1)	scipy	PackageName	3	3
(0, 1)	f	Func_Name	4	4
(0, 1)	T3	ICGTempVar	-1	-1
(0, 1)	T10	ICGTempVar	-1	-1
(0, 1)	T13	ICGTempVar	-1	-1
(0, 1)	T16	ICGTempVar	-1	-1
(0, 1)	T19	ICGTempVar	-1	-1
(0, 1)	T22	ICGTempVar	-1	-1
(0, 1)	T25	ICGTempVar	-1	-1
(0, 1)	T26	ICGTempVar	-1	-1
(0, 1)	T27	ICGTempVar	-1	-1
(0, 1)	T30	ICGTempVar	-1	-1
(0, 1)	T31	ICGTempVar	-1	-1
(0, 1)	T32	ICGTempVar	-1	-1
(0, 1)	L0	ICGTempLabel	-1	-1
(0, 1)	T33	ICGTempVar	-1	-1
(0, 1)	L1	ICGTempLabel	-1	-1
(0, 1)	T38	ICGTempVar	-1	-1
(0, 1)	T39	ICGTempVar	-1	-1
(0, 1)	T40	ICGTempVar	-1	-1
(0, 1)	T41	ICGTempVar	-1	-1
(0, 1)	T46	ICGTempVar	-1	-1
(0, 2)	1	Constant	5	15
(0, 2)	x	Identifier	5	15
(0, 2)	2	Constant	7	12
(0, 2)	y	Identifier	8	17
(0, 2)	3	Constant	10	10
(0, 2)	a	Identifier	10	13
(0, 2)	4	Constant	11	11
(0, 2)	b	Identifier	11	13
(0, 2)	c	Identifier	12	12
(0, 2)	d	Identifier	13	13
(1, 4)	1	Constant	16	17
(1, 4)	c	Identifier	16	16
(1, 8)	2	Constant	18	18
(1, 8)	c	Identifier	18	18
(1, 16)	1	Constant	20	20
(1, 16)	c	Identifier	20	20

Assembly Code

```
MOV R1, 2
STR R1, T10
MOV R2, R1
STR R2, x
MOV R3, 1
STR R3, T13
MOV R4, R3
STR R4, y
MOV R5, 3
STR R5, T16
MOV R6, R5
STR R6, a
MOV R7, 4
STR R7, T19
MOV R8, R7
STR R8, b
MOV R9, 2
STR R9, T22
MOV R10, R9
STR R10, c
MOV R11, R6
STR R11, T25
MOV R12, R8
STR R12, T26
ADD R13, R11, R12
STR R13, T27
MOV R14, R13
STR R14, d
MOV R1, R2
STR R1, T30
MOV R3, 1
STR R3, T31
CMP R1, R3
BE Label1
MOV R4, 0
B Label2
Label1 :MOV R4, 1
Label2 :STR R4, T32
```

CONCLUSION

Thus, we were able to construct a Python3 mini compiler that supports assignment statements, pass and break statements, arithmetic operations, relational operators, if-elif-else construct, while loop, and function definition and calls; using the lex and yacc tools in the C programming language.

FURTHER ENHANCEMENTS

1. Segmentation Faults can occur sometimes, depending on the input given to the program.

2. Array has been used as the data structure for symbol table. A hash table would've given better time efficiency.

3. The output of the optimised code has to be manually copied into a text file to be passed as input to the assembly code generation program.

REFERENCES

1. http://dinosaur.compilertools.net/
2. GeeksforGeeks : https://www.geeksforgeeks.org/
3. Lex and Yacc Tutorial by Tom Niemann : https://www.isi.edu/~pedro/Teaching/CSCI565-Fall15/Materials/LexAndYaccTutorial.pdf
4. https://www.javatpoint.com/code-generation
5. https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lects/final.codegen.pdf