

***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**  **Aditya Shankaran**  **Athul Sandosh** | **PES1201700278**  **PES1201700710**  **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**

|  |  |  |
| --- | --- | --- |
| **Chapter No.** | **Title** | **Page No.** |
|  | **INTRODUCTION (Mini-Compiler is built for which language. Provide sample input and output of your project)** | **03** |
|  | **ARCHITECTURE OF LANGUAGE:**   * **What all have you handled in terms of syntax and semantics for the chosen language.** | **06** |
|  | **LITERATURE SURVEY (if any paper referred or link used)** | **06** |
|  | **CONTEXT FREE GRAMMAR (which you used to implement your project)** | **07** |
|  | **DESIGN STRATEGY (used to implement the following)**   * **SYMBOL TABLE CREATION** * **ABSTRACT SYNTAX TREE** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING *-* strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).** * **TARGET CODE GENERATION** | **10** |
|  | **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.** | **11** |
|  | **RESULTS AND possible shortcomings of your Mini-Compiler** | **13** |
|  | **SNAPSHOTS (of different outputs)** | **14** |
|  | **CONCLUSIONS** | **19** |
|  | **FURTHER ENHANCEMENTS** | **19** |
| **REFERENCES/BIBLIOGRAPHY** | | **19** |

**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) :

.data

x: .word 0

y: .word 0

a: .word 0

b: .word 0

c: .word 0

d: .word 0

c: .word 0

.text

.global \_start

\_start:

MOV R0,#2

LDR R1,addr\_x

LDR R2,[R1]

MOV R2,R0

MOV R3,#1

LDR R4,addr\_y

LDR R5,[R4]

MOV R5,R3

MOV R6,#3

LDR R7,addr\_a

LDR R8,[R7]

MOV R8,R6

MOV R9,#4

LDR R10,addr\_b

LDR R11,[R10]

MOV R11,R9

MOV R0,#2

LDR R1,addr\_c

LDR R3,addr\_x

STR R2,[R3]

LDR R2,[R1]

MOV R2,R0

MOV R4,R8

LDR R6,addr\_y

STR R5,[R6]

MOV R5,R11

ADD R7,R4,R5

LDR R9,addr\_a

STR R8,[R9]

LDR R8,addr\_d

LDR R10,[R8]

MOV R10,R7

LDR R0,addr\_b

STR R11,[R0]

MOV R11,R1

MOV R3,#1

CMP R11,R3

BNE L0

LDR R3,addr\_c

STR R2,[R3]

MOV R2,#1

LDR R4,addr\_c

LDR R6,[R4]

MOV R6,R2

B L1

L0: MOV R5,R7

MOV R9,#1

CMP R5,R9

BNE L0

MOV R8,#2

MOV R6,R8

B L1

LDR R11,addr\_d

STR R10,[R11]

L1: MOV R10,#1

MOV R6,R10

LDR R0,addr\_x

STR R1,[R0]

STR R6,[R3]

STR R1,[R0]

LDR R1,addr\_y

STR R7,[R1]

addr\_x: .word x

addr\_y: .word y

addr\_a: .word a

addr\_b: .word b

addr\_c: .word c

addr\_d: .word d

addr\_c: .word c

**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 python, 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 BNE/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 : Python
* Data Structure Used : List, Dictionary

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,

$cd ../Assembly

$cat ICG.txt|python3 assembly.py>assembly.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

A picture containing computer

Description automatically generated

A screenshot of a cell phone

Description automatically generated

Symbol Table

A picture containing guitar

Description automatically generated

Abstract Syntax Tree

A close up of a logo

Description automatically generated

Optimised Intermediate Code

A close up of text on a black background

Description automatically generated

Updated Symbol TableA picture containing computer

Description automatically generated

Assembly Code

A screenshot of a computer

Description automatically generated

**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>