MINI PYTHON COMPILER

A PROJECT REPORT

Submitted by

BL.EN.U4AIE19007	Apoorva Mani
BL.EN.U4AIE19041	Maturi Tanuj
BL.EN.U4AIE19068	Aishwarya V

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

ARTIFICIAL INTELLIGENCE AND ENGINEERING

AMRITA SCHOOL OF ENGINEERING, BANGALORE

AMRITA VISHWA VIDYAPEETHAM

BANGALORE 560 035

June – 2022

BONAFIDE CERTIFICATE

This is to certify that the project report entitled **MINI PYTHON COMPILER** submitted by

BL.EN.U4AIE19007	Apoorva Mani
BL.EN.U4AIE19041	Maturi Tanuj
BL.EN.U4AIE19068	Aishwarya V

in partial fulfillment of the requirements in the **Degree Bachelor of Technology** "Artificial Intelligence and Engineering" as part of 21AIE313 Introduction to Modern Compiler Design Project is a bonafide record of the work carried out under my guidance and supervision at Amrita School of Engineering, Bangalore.

<signature>	<signature>	
<Guide name>	<Guide name>	Dr. Sriram Devanathan
<Designation>	<Designation>	Professor and Chairperson,
Dept. of CSE	Dept. of CSE	Dept. of CSE

This project report was evaluated by us on ………

ACKNOWLEDGEMENT

The satisfaction that accompanies successful completion of any task would be incomplete without mention of people who made it possible, and whose constant encouragement and guidance have been source of inspiration throughout the course of this project work.

We offer our sincere pranams at the lotus feet of **"AMMA", MATA AMRITANANDAMAYI DEVI** who showered her blessing upon us throughout the course of this project work.

We owe our gratitude to **Dr. Manoj P.**, Director, Amrita School of Engineering, Bangalore.

We thank **Dr. Sriram Devanathan,** Principal and Chairperson-CSE, Amrita School of Engineering, Bangalore for his support and inspiration.

It is a great pleasure to express our gratitude and indebtedness to our project guide "**GUIDE NAME AND DESIGNATION"**, Department of Computer Science and Engineering, Amrita School of Engineering, Bangalore for her valuable guidance, encouragement, moral support, and affection throughout the project work.

We would like to thank express our gratitude to project panel members for their suggestions, encouragement, and moral support during the process of project work and all faculty members for their academic support. Finally, we are forever grateful to our parents, who have loved, supported, and encouraged us in all our endeavors.

ABSTRACT

The Mini-Compiler contains all phases of compiler has been made for the language Python by using C language (till intermediate code optimization phase) and we used Python language itself for target code generation as well. The constructs that have been focused on are ‘if-else’ and ‘while’ statements. The optimizations handled for the intermediate code are ‘packing temporaries’ and ‘constant propagation’. Syntax and semantic errors have been handled and syntax error recovery has been implemented using Panic Mode Recovery in the lexer.

TABLE OF CONTENTS

Page no.

CHAPTER - 1

INTRODUCTION

The Mini-Compiler contains all phases of compiler has been made for the language Python by using C language (till intermediate code optimization phase) and we used Python language itself for target code generation as well. The constructs that have been focused on are ‘if-else’ and ‘while’ statements. The optimizations handled for the intermediate code are ‘packing temporaries’ and ‘constant propagation’. Syntax and semantic errors have been handled and syntax error recovery has been implemented using Panic Mode Recovery in the lexer.

The screenshots of the sample input and target code output are as follows:

Sample Input:

```
a=10
b=9
c=a+b+100
e=10
f=8
d=e*f
if(a>=b):
    a=a+b
    g=e*f*100

u=10
j=99
```

Figure 1 Sample Input python code

Sample Output:

This the target code which is generated after ICG

```
MOV R0, #10
MOV R1, #9
MOV R2, #119
MOV R3, #8
MOV R4, #80
l0:
MOV R5, #0
BNEZ R5, l1
MOV R6, #19
MOV R7, #8000
l1:
MOV R8, #99
ST b, R1
ST c, R2
ST e, R0
ST f, R3
ST d, R4
ST a, R6
ST g, R7
ST u, R0
ST j, R8
```

Figure 2 Sample output

CHAPTER - 2

SYSTEM SPECIFICATIONS

2.1 SOFTWARE REQUIREMENTS

ARCHITECTURE OF LANGUAGE

For this mini compiler, the following aspects of the Python language syntax havebeen covered:

- Constructs like ‘if-else’ and ‘while’ and the required indentation for these loops.
- Nested loops
- Integer and float data types

Specific error messages are displayed based on the type of error. Syntax errors are handled using the yyerror() function, while the semantic errors are handled by making a call to a function that searches for a particular identifier in the symbol table. The line number is displayed as part of the error message.

As a part of error recovery, panic mode recovery has been implemented for the lexer. It recovers from errors in variable declaration. In case of identifiers, whenthe name begins with a digit, the compiler neglects the digit and considers the rest as the identifier name.

Languages used to develop this project:

- C
- YACC
- LEX
- PYTHON

2.2 DIFFERENT MODULES OF PROJECT

Different Folders:

1. **Token_And Symbol_Table**: This folder contains the code that outputs the tokens andthe symbol table.
2. **Abstract_Syntax_Tree**: This folder contains the code that displays the abstract syntaxtree.
3. **Intermediate_Code_Generation**: This folder contains the code that generates thesymbol table before optimisations and the intermediate code.

Different Files:

1. **proj.l**: It is the Lexical analyser file which defines all the terminals of the productions stated in the yacc file. It contains regular expressions.
2. **proj1.y**: Yacc file is where the productions for the conditional statements like if-elseand while and expressions are mentioned. This file also contains the semantic rules defined against every production necessary. Rules for producing three address code is also present.
3. **inp.py:** The input python code which will be parsed and checked for semantic correctness by executing the lex and yacc files along with it.

2.3 CONTEXT-FREE GRAMMAR

REGEX

digits -> [0-9]
num -> digits+(\.digits+)?([Ee][+|-]?digits+)?
id -> [a-zA-Z][a-zA-Z0-9]*
integer -> [0-9]+
string -> [a-z | A-Z | 0-9 | special]*
special -> [! " # $ % & \ ( ) * + , - . / : ; < = > ? @ [\ \ ] ^ _ ` { | } ~]

GRAMMAR

P ->S
S ->Simple S | Compound S | epsilon
Simple ->Assignment LB | Cond LB | Print LB | break | pass | continueAssignment -> id opassgn E1 | id opassgn cond | id listassgn Arr
| id strassgn Str

E1 -> E1 op1 E2 | E2
E2 -> E2 op2 E3 | E3
E3 -> E4 op3 E3 | E4
E4 -> num | id | (E1)

opassgn -> = | /= | *= | += | -=op1 -> + | -
op2 -> / | *
op3 -> **
LB -> \n
listassgn -> = strassgn -> = | += | -=

Arr -> [list] | [list] mul | [list] add | matmat -> [listnum] | [liststr]
list -> listnum | liststr | Range listnum -> num,listnum | epsilon | numliststr -> Str,liststr | epsilon | Str

mul -> * integer
add -> + Arr
Range -> range (start , stop , step)start -> integer | epsilon
stop -> integer
step -> integer | epsilon
Str -> string | string mul | string addstr
addstr -> + string

Compound -> if_else LB | while_loop LB
if_else -> if condition : LB IND else | if condition : LB IND
| if condition : S | if condition : S else
else -> else : LB IND | else : S
while_loop -> while condition : LB IND | while condition : S

condition -> cond | (cond)
cond -> cond opor cond1 | cond1 cond1 -> cond1 opand cond2 | cond2cond2 -
> opnot cond2 | cond3
cond3 -> (cond) | relexp | bool
relexp -> relexp relop E1 | E1 | id | num relop -> < | > | <= | >= | == | != | in | not
inbool -> True | False

opor -> || | or
opand -> && | and
opnot -> not | ~

IND -> indent S dedent
indent -> \t
dedent -> -\t

Print -> print (toprint) | print (toprint,sep) | print (toprint,sep,end)
| print (toprint,end) toprint -> X | X,toprint | epsilonX -> Str | Arr | id
| num
sep -> sep = Str
end -> end = St

CHAPTER - 3

3.1 DESIGN STRATEGY

1) SYMBOL TABLE CREATION

Linked list is being used to create the symbol table. The final output shows the label, value, scope, line number and type. We have created three functions to generate the symbol table. They are:

- Insert: It pushes the node onto the linked list.
- Display: It displays the symbol table.
- Search: It searches for a particular label in the linked list.

2) ABSTRACT SYNTAX TREE

This is being implemented using a structure that has three members whichhold the data, left pointer and right pointer respectively. The functions thataid in creating and displaying this tree are:

- BuildTree: It is used to create a node of this structure and add it to the existing tree.
- printTree: This function displays the abstract syntax tree using pre-order traversal.

3) INTERMEDIATE CODE GENERATION

We have used the stack data structure to generate the intermediate code that uses some functions, which are called based on some conditions.

4) ERROR HANDLING

- Syntax Error:
  If the token returned does not satisfy the grammar, then yyerror() isused to display the syntax error along with the line number.

- Semantic Error:
  If there is an identifier in the RHS of an assignment statement, the symbol table is searched for that variable. If the variable does not exist in the symbol table, this is identified as a semantic error and isdisplayed.

- Error Recovery:
  Panic Mode Recovery is used as the error recovery technique, where if the variable declaration has been done with a number at the start,it ignores the number and considers the rest as the variable name. This has been implemented using regex.

CHAPTER – 4
SYSTEM IMPLEMENTATION

4.1 Modules used with description:

1) SYMBOL TABLE CREATION

The following snapshot shows the structure declaration for symbol table:

```
struct symbtab
{
        char label[20];
        char type[20];
        int value;
        char scope[20];
        int lineno;
        struct symbtab *next;
};
```

Figure 3 structure declaration for symbol table

These are the functions used to generate the symbol table:

```
void insert(char* l,char* t,int v,char* s,int ln);
struct symbtab* search(char lab[]);
void display();
```

Figure 4 generate the symbol table

These snapshots are taken from proj1.y file in Token and Symbol table folder.

2) ABSTRACT SYNTAX TREE

The following data structure is used to represent the abstract syntax tree:

```
typedef struct Abstract_syntax_tree
{
        char *name;
        struct Abstract_syntax_tree *left;
        struct Abstract_syntax_tree *right;
}node;
```

Figure 5 abstract syntax tree

The following functions build and display the syntax tree:

```
node* buildTree(char *,node *,node *);
void printTree(node *);
```

Figure 6 display the syntax tree

These snapshots are taken from proj1.y file in Abstract syntax tree folder.

3) INTERMEDIATE CODE GENERATION

The following arrays act as stacks and are used for the generation ofintermediate code:

```
char label[2]="l"; // labels
int l_ = 0;        //count of labels(l1,l2,....)
char l__[100] = {'\0'}; //labels
char st[100][10];  //stack used in icg generation
int top=0;         //top of stack
int i_=0;          //count of temp variables in icg
char i__[100] = {'\0'}; //temp variables (t1,t2,...)
char temp[2]="t";
char ICG[10000]=""; //icg
char try1[5][10];
char try[5][10];
```

Figure 7 intermediate code

The following functions push onto the stack and generate the intermediatecode, when called based on various conditions:

```
void push(char*);
void codegen(int val,char* aeval_);
void codegen_assign();
void codegen2();
void codegen3();
```

Figure 8 intermediatecode, when called based on various conditions

4.2 BUILD AND RUN THE PROGRAM:

The following screenshot displays what commands need to be executed to buildand run the program:

```
lex proj.l
yacc -d -v proj1.y
gcc lex.yy.c y.tab.c  -lm -w
a.exe
```

The above commands need to be executed on the terminal which is inside theproject folder that contains the code for the compiler.

CHAPTER – 5
SYSTEM TESTING AND RESULTS ANALYSIS

RESULTS AND SHORTCOMINGS

The mini-compiler built in this project works perfectly for the ‘if-else’ and ‘while’ constructs of Python language. Our compiler can be executed in differentphases by building and running the code separated in the various folders. The final code displays the output of all the phases on the terminal, one after the other. First, the tokens are displayed, followed by a ‘PARSE SUCCESSFUL’ message. Then abstract syntax tree is printed. Next, the symbol table along with the intermediate code is printed without optimisation. Finally, the symbol table andthe intermediate code after optimisation is displayed after the quadruples table. The final output is the target code, written in the instruction set architecture followed by the hypothetical machine model introduced in this project. This is for inputs with no errors. But in case of erroneous inputs, the token generation isstopped on error encounter and the corresponding error message is displayed.

This mini-compiler has the following shortcomings:

- User defined functions are not handled.
- Importing libraries and calling library functions is not taken care of.
- Datatypes other than integer and float, example strings, lists, tuples,dictionaries, etc have not been considered.
- Constructs other than ‘while’ and ‘if-else’ have not been added in thecompiler program.

SNAPSHOTS

TEST CASE 1 (Correct input):

Input:

```
a=10
b=9
c=a+b+100
e=10
f=8
d=e*f
if(a>=b):
    a=a+b
    g=e*f*100

u=10
j=99
```

Figure 9 input code

Tokens and Symbol Table:

```
C:\Users\Ashish\Downloads\Python-Compiler-master\Python-Compiler-master\Project Code\1-Token_and_Symbol_Table>a.exe
ID equal int
ID equal int
ID equal ID plus ID plus int
ID equal int
ID equal int
ID equal ID mul ID
if special_start ID greaterthanequal ID special_end colon
indent ID equal ID plus ID
indent ID equal ID mul ID mul int

ID equal int
ID equal int
-----------------------------PARSE SUCCESSFUL--------------------------

-----------------SYMBOL TABLE---------------
--------------------------------------------
LABEL   TYPE          VALUE   SCOPE   LINENO
a       IDENTIFIER    19      local   8
b       IDENTIFIER    9       global  2
c       IDENTIFIER    119     global  3
e       IDENTIFIER    10      global  4
f       IDENTIFIER    8       global  5
d       IDENTIFIER    80      global  6
g       IDENTIFIER    8000    local   9
u       IDENTIFIER    10      global  11
j       IDENTIFIER    99      global  12
```

Figure 10 case output symbol table

Abstract Syntax Tree:

```
C:\Users\Ashish\Downloads\Python-Compiler-master\Python-Compiler-master\Project Code\2-Abstract_Syntax_Tree>a.exe

-----------Abstract Syntax Tree-------------
( SEQ ( = a 10 )( SEQ ( = b 9 )( SEQ ( = c ( + ( + a b ) 100 ))( SEQ ( = e 10 )( SEQ ( = f 8 )( SEQ ( = d
( * e f ))( SEQ ( IF ( >= a b )( SEQ ( = a ( + a b ))( SEQ ( = g ( * ( * e f ) 100 )) NULL )))( SEQ ( = u
10 )( SEQ ( = j 99 ) NULL )))))))))
```

Figure 11 abstract syntax tree

Symbol Table and Unoptimized Intermediate Code

```
C:\Users\Ashish\Downloads\Python-Compiler-master\Python-Compiler-master\Project Code\3-Intermediate_Code_Generation>a.exe

-----------------SYMBOL TABLE before Optimisations---------------
-----------------------------------------------------------------
LABEL   TYPE            VALUE   SCOPE   LINENO
a       identifier      9       local   8
b       identifier      9       global  2
t0      identifier      19      -       2
t1      identifier      119     -       3
c       identifier      119     global  3
e       identifier      10      global  4
f       identifier      8       global  5
t2      identifier      80      -       6
d       identifier      80      global  6
t3      identifier      0       -       6
t4      identifier      9       -       8
t5      identifier      80      -       8
t6      identifier      8000    -       9
g       identifier      8000    local   9
u       identifier      10      local   11
j       identifier      99      local   12

------------ICG without optimisation------------
a=10
b=9
t0=a+b
t1=t0+100
c=t1
e=10
f=8
t2=e*f
d=t2
l0 : t3=a>=b
if not t3 goto l1
t4=a+b
a=t4
t5=e*f
t6=t5*100
g=t6
l1 : u=10
j=99
```

Figure 12 intermediate code output

TEST CASE 2 (Syntax Error):

Input:

```
1   a=10
2   b=9
3   c=a+b+100
4   e+10  // error
5   f=8
6   d=e*f
7   if(a>=b):
8       a=a+b
9       g=e*f*100
10
11  u=10
12  j=99
```

Figure 13 sample case input

Output:

```
C:\Users\Ashish\Downloads\Python-Compiler-master\Python-Compiler-master\Project Code\1-Token_and_Symbol_Table>a.exe
ID equal int
ID equal int
ID equal ID plus ID plus int
ID plus
-----------------SYNTAX ERROR : at line number 4 ------------------------
```

Figure 14 syntax error

TEST CASE 3 (Semantic Error):

Input:

```
a=10
b=b+9   //error
c=a+b+100
e=10
f=8
d=e*f
if(a>=b):
    a=a+b
    g=e*f*100

u=10
j=99
```

Figure 15 sample case 3 input

Output:

```
C:\Users\Ashish\Downloads\Python-Compiler-master\Python-Compiler-master\Project Code\1-Token_and_Symbol_Table>a.exe
ID equal int
ID equal ID
-----------------ERROR : b Undeclared at line number 2-------------------
```

Figure 16 sample case 3 output

TEST CASE 4 (Error Recovery):

Input:

```
a=10
b=b+9   //error
c=a+b+100
e=10
f=8
d=e*f
if(a>=b):
    a=a+b
    g=e*f*100

u=10
j=99
```

Figure 17 case 4 input

Output:

```
plus int
ID equal ID plus ID plus int
ID equal int
ID equal int
ID equal ID mul ID
if special_start ID greaterthanequal ID special_end colon
indent ID equal ID plus ID
indent ID equal ID mul ID mul int

ID equal int
ID equal int
------------------------------PARSE SUCCESSFUL--------------------------
```

Figure 18 sample output

CHAPTER – 6

CONCLUSION AND FUTURE SCOPE

CONCLUSIONS

- Making a full complete compiler is a very difficult task and it takes lots of time to make it. So, we have successfully made a minicompiler which performs following operations:

1. This is a mini compiler for python using lex and yacc files which takes in a python program and according to the context free grammar written, the program is validated.
2. Regular Expressions are written to generate the tokens.
3. Symbol table is created to store the information about the identifiers.
4. Abstract syntax tree is generated and displayed according to the pre-order tree traversal.
5. Intermediate code is generated, and the data structure used foroptimization is Quadruples. The optimization techniques used are constant propagation and packing temporaries.

FUTURE ENHANCEMENTS

This mini compiler can be enhanced to a complete compiler for the Python language by making a few improvements. User defined functions can be handled and the functionality of importing libraries and calling library functions can be taken care of. Datatypes other than integer, example strings, lists, tuples, dictionaries, etc. can be included and constructs other than ‘while’ and ‘if-else’, like ‘for’ can be added in the compiler program. The output can be made to lookmore enhanced and beautiful. The overall efficiency and speed of the program can be improved by using some other data structures, functions, or approaches.

REFERENCES

1) **Lex and Yacc**

   http://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf

2) **Introduction about Flex and Bison**

   http://dinosaur.compilertools.net/

3) **Full Grammar Specification**

   https://docs.python.org/3/reference/grammar.html

4) **Introduction to Yacc**

   https://www.inf.unibz.it/~artale/Compiler/intro-yacc.pdf

5) **Intermediate Code Generation**

   https://2k8618.blogspot.com/2011/06/intermediate-code-generator-for.html?m=0

6) **A Code Generator to translate Three-Address intermediate code to MIPS assembly code**

   https://www2.cs.arizona.edu/~debray/Teaching/CSc453/DOCS/3addr2spim.pdf