MINI PYTHON COMPILER

Project submitted to the

SRM University – AP, Andhra Pradesh

for the partial fulfillment of the requirements to award the degree of

Bachelor of Technology

In

Computer Science and Engineering

School of Engineering and

Sciences

Submitted by

SLS Praneet | AP21110010206

Puneet Sai S | AP21110010256

Bhargav K | AP21110010204

Amey Khare | AP21110010224

Under the Guidance of

Dr. ARNAB MITRA

SRM University–AP

Neerukonda, Mangalagiri, Guntur

Andhra Pradesh – 522 240

November, 2023.

Certificate

This is to certify that the work present in this Project entitled "**MINI PYTHON COMPILER**" has been carried out by **SLS Praneet AP21110010206, Puneet Sai S AP21110010256, Bhargav K AP21110010204, Amey Khare AP21110010224** under my/our supervision. The work is genuine, original, and suitable for submission to the SRM University – AP for the award of Bachelor of Technology/Master of Technology in **School of Engineering and Sciences**.

Supervisor

(Signature)

Prof. / Dr. Arnab Mitra

Designation,

Affiliation.

Acknowledgements

Table of Contents

Introduction

The Mini-Compiler, contains all phases of compiler has been made for the language Python by using C language (till intermediate code optimisation 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.

Our aim is to develop a mini python compiler in which we will be doing following things:

1.   First we will do lexical analysis of the given input python code, from which we will generate symbol table and identify the tokens.

2.   Then we will do syntax analysis in which we will generate AST (Abstract Syntax tree).

3.   Using AST, we will generate unoptimised intermediate code. Then we will convert this into optimised intermediate code and display symbol table before and after optimisation.

4.   After converting into optimised ICG, we will use this to generate target code (Target code generation deals with assembly language to convert optimized code into machine understandable format).

5.   We also took care of error handling (like syntax error and semantic error in the given input code).

6.   Finally all of the above steps will be compiled and hence we will be executing all phases of the compiler.

Languages used to develop this project :

1. C 2. C++ 3. YACC 4. LEX 5. PYTHON

Tools used

1. Flex (to make lex program work) 2. Bison (to make yacc program work)

Architecture Of Language

For this mini-compiler, the following aspects of the Python language syntax have been covered:

1) Constructs like 'if-else' and 'while' and the required indentation for these loops.
2) Nested Loops
3) 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, when the name begins with a digit, the compiler neglects the digit and considers the rest as the identifier name.

Modules Of The Project

Folders:

1) **Token_AND_Symbol Table:** This folder contains the code that outputs the tokens and the symbol table.
2) **Abstract_Syntax_Tree:** This folder contains the code that displays the abstract syntax tree.
3) **Intermediate_Code_Generation:** This folder contains the code that generates the symbol table before optimisations and the intermediate code.
4) **Optimised_ICG:** This folder contains the code that generates the symbol table after optimisations, the quadruples table and the optimised intermediate code.
5) **Target_Code:** This folder contains the code that displays the assembly code/target code.

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-else and 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) **final.py:** It is the python file which converts the ICG to target code using regex.
4) **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.

Context-Free Grammar for the Compiler

Regular Expressions:

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 | continue

Assignment -> 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 |mat

mat -> [listnum] | [liststr]

list -> listnum | liststr | Range

listnum -> num,listnum | epsilon |num

liststr -> 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 |cond2

cond2 -> opnot cond2 | cond3

cond3 -> (cond) | relexp | bool

relexp -> relexp relop E1 | E1 | id | num

relop -> < | > | <= | >= | == | != | in | not in

bool -> 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 |epsilon

X -> Str | Arr | id | num

sep -> sep = Str

end -> end =Str

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 which hold the data, left pointer and right pointer respectively. The functions that aid 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) CODE OPTIMIZATION

A data structure known as quadruple is used to optimize the code. This data structure holds the details of each of the assignment, label and goto statements.

5) ERROR HANDLING

- Syntax Error:
  If the token returned does not satisfy the grammar, then yyerror() is used 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 is displayed.

- 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.

6) TARGET CODE GENERATION

The optimised intermediate code is read from a text file, line after line, and goes through a series of if-else loops to generate the target code. A hypothetical target machine model has been used as the target machine and the limit on the number of reusable registers has been set to 13, numbered from R0 to R12. A hypothetical machine model has been used that follows the following instruction set architecture:

```
1)  Load/Store Operations:
    ST <loc>, R
    LD R, <loc>

2)  Move Operations:
    MOV R_d, #<num>

3)  Arithmetic Operations:
    <ADD/SUB/MUL/DIV> R_d, R_1, R_2

4)  Compare Operations:
    CMP<cond> R_d, R_1, R_2

    (<cond>: E for ==, NE for !=, G for >,
    L for <, GE for >= or LE for <=)

5)  Logical Operations:
    NOT R_d, R
    <AND/OR> R_d, R_1, R_2

6)  Conditional Branch:
    BNEZ R_d, label

7)  Unconditional Branch:
    BR label
```

Implementation Details

1) SYMBOL TABLE CREATION

The following snapshot shows the structure declaration for symbol

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

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();
```

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;
```

The following functions build and display the syntax tree:

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

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 of intermediate code:

```c
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];
```

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

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

These snapshots are taken from proj1.y file in ICG folder.

4) CODE OPTIMISATION

The data structure quadruple declaration has been shown below:

```c
typedef struct quadruples
{
    char *op;
    char *arg1;
    char *arg2;
    char *res;
}quad;
```

The following functions are used to add to the quadruples table and display it onto the terminal:

```c
void displayquad();
char addquad(char*,char*,char*,char*);
```

These snapshots are taken from proj1.y file in Optimised_ICG folder.

5) TARGET CODE GENERATION

A global dictionary holds the mapping between each constant/identifier and the corresponding register that holds that constant/identifier. There also is a global list that holds the identifiers that need to be stored towards the end of the program.

There are two functions which are used for register allocation. The 'getreg()' function gets the next free/unallocated register and uses the 'fifo()' function in cases when all the registers are used up. The 'fifo()' function uses the 'First In First Out' method to free a register and return it to the 'getreg()' function.

```python
def getreg():
    for i in range(0,13):
        if reg[i]==0:
            reg[i]=1
            return 'R'+str(i)
    register = fifo()
    return register
```

```python
def fifo():
    global fifo_reg
    global fifo_return_reg
    for k,v in var.copy().items():
        if(v == 'R'+str(fifo_reg) ):
            fifo_return_reg = v
            var.pop(k)
            if(k in store_seq):
                store_seq.remove(k)
                print("ST ", k, ', ', v, sep='')
    fifo_reg = int(fifo_return_reg[1:]) + 1
    return fifo_return_reg
```

These functions are as follows:

These snapshots are taken from final.py file in Target_Code folder.

6) ERROR HANDLING

The following snapshot shows the error handling function for syntaxerrors:

```
int yyerror(){
        printf("\n-----------------SYNTAX ERROR : at line number %d -----------------------\n",yylineno-1);
        error = 1;
        v=0;
        return 0;
}
```

The following snapshot shows semantic error handling functionality:

```
t_ptr=search($1);

if(t_ptr==NULL)
{
        printf("\n---------------------ERROR : variable %s undeclared------------------\n",$1);
        error = 1;
}
```

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

The regex for panic mode recovery implemented in the lexer is

```
[0-9;!,@#]*/(({alpha}|"_")({alpha}|{digits}|"_")*)
```

as follows:

The above snapshots are taken from proj.l file in Symbol table folder.

BUILD AND RUN THE PROGRAM:

The following screenshot displays what commands need to be executed to build and 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 the project folder that contains the code for the compiler

Results

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 different phases 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 and the 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 is stopped 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 the compiler program

Test Cases (I/P & O/P)

TEST CASE 1 (Correct input):

Input:

```
 1    a=10
 2    b=9
 3    c=a+b+100
 4    e=10
 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
```

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
```

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 )))))))))
```

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
```

Symbol Table, Quadruples Table and Optimised Intermediate Code:

```
·················SYMBOL TABLE after Optimisations················
----------------------------------------------------------------
LABEL    TYPE            VALUE    SCOPE    LINENO
a        identifier      19       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       -        5
d        identifier      80       global   6
t3       identifier      1        -        6
t4       identifier      0        -        6
t5       identifier      8000     -        8
g        identifier      8000     local    9
u        identifier      10       local    11
j        identifier      99       local    12

··············QUADRUPLES···················
         op       arg1     arg2     result
         =        10                a
         =        9                 b
         +        a        b        t0
         +        t0       100      t1
         =        t1                c
         =        10                e
         =        8                 f
         *        e        f        t2
         =        t2                d
         Label                      l0
         >=       a        b        t3
         goto                       l1
         =        t0                a
         *        t2       100      t5
         =        t5                g
         Label                      l1
         =        10                u
         =        99                j
```

```
ICG with optimisations(Packing temporaries & Constant Propagation)
a = 10
b = 9
t0 = 10 + 9
t1 = 19 + 100
c = 119
e = 10
f = 8
t2 = 10 * 8
d = 80
l0:
t3 = 10 >= 9
t4 = not 1
if 0 goto l1
a = 19
t5 = 80 * 100
g = 8000
l1:
u = 10
j = 99
```

Target Code:

```
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
```

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
```

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

TEST CASE 3 (Semantic Error):

Input:

```
 1    a=10
 2    b=b+9    //error
 3    c=a+b+100
 4    e=10
 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
```

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

TEST CASE 4 (Error Recovery):

Input:

```
 1    a=10
 2    b=b+9   //error
 3    c=a+b+100
 4    e=10
 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
```

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

Conclusion

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 mini compiler 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 for optimisation is Quadruples. The optimisation techniques used are constant propagation and packing temporaries.

6. The optimised intermediate code is then converted to the Target code using a hypothetical machine model.

7. Error handling and recovery implemented take care of erroneous inputs.

Future Work

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. Data types 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 look more enhanced and beautiful. The overall efficiency and speed of the program can be improved by using some other data structures, functions or approaches.