Report on

Compiler for the if else and for statement in python language

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Arun Kumar Kashinath Agasar	**PES1201701537**
Vikas Kalagi	**PES1201701654**
Rakesh Devani	**PES1201701602**

Under the guidance of

Prof. Preet Kanwal
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

Our mini-compiler is built for a subset of the Python language (i.e only For and IF ELSE statements). We have used tools such as yacc/bison, lex/flex and Python scripts to build the complete compiler. An example of what our compiler produces -

Sample input -

```
A=10
P=10
T=20
c=A + A+A
wet=A+A
y=1

#kjhffdjasdhfjafnj

t=10*8
y=t+1
y=10*9
y=10*9*t
y=t+1
pt=10*8



for i in range(10):
      A=A+20

#y=1
for x in range(1,10):
      T=10+T+A
      for j in range(5,25):
            y=0

y=0
g=0

if(true):
      u=0
      if(False):
```

```
            y=y+9
      f=1000


if(A>=0 and p<10 and c<25):
      print("saf")
      A=20
else:
      ty=0
```

Our compiler's output -

```
main:
MOV $R0, #10
ST A, $R0
MOV $R0, #10
ST P, $R0
MOV $R0, #20
ST T, $R0
MOV $R0, #30
ST c, $R0
MOV $R0, #20
ST wet, $R0
MOV $R0, #1
ST y, $R0
MOV $R0, #80
ST t, $R0
MOV $R0, #81
ST y, $R0
MOV $R0, #90
ST y, $R0
MOV $R0, #7200
ST y, $R0
MOV $R0, #81
ST y, $R0
MOV $R0, #80
ST pt, $R0
MOV $R0, #0
```

```
ST i, $R0

L1:
LD $R0, i
LD $R1, #10
SUBS $R0, $R0, $R1
BGEZ $R0, L2
LD $R0, A
MOV $R1, #20
ADD $R2, $R0, $R1
ST r11, $R2
LD $R0, r11
MOV $R1, $R0
ST A, $R1
LD $R0, i
MOV $R1, #1
ADD $R2, $R0, $R1
ST i, $R2
BR L1

L2:
MOV $R0, #1
ST x, $R0

L3:
LD $R0, x
LD $R1, #10
SUBS $R0, $R0, $R1
BGEZ $R0, L4
MOV $R0, #10
LD $R1, T
ADD $R2, $R1, $R0
ST r13, $R2
LD $R0, r13
LD $R1, A
ADD $R2, $R0, $R1
ST r14, $R2
LD $R0, r14
```

```
MOV $R1, $R0
ST T, $R1
MOV $R0, #5
ST j, $R0

L5:
LD $R0, j
LD $R1, #25
SUBS $R0, $R0, $R1
BGEZ $R0, L6
MOV $R0, #0
ST y, $R0
LD $R0, j
MOV $R1, #1
ADD $R2, $R0, $R1
ST j, $R2
BR L5

L6:
LD $R0, x
MOV $R1, #1
ADD $R2, $R0, $R1
ST x, $R2
BR L3

L4:
MOV $R0, #0
ST y, $R0
MOV $R0, #0
ST g, $R0
LD $R0, #1
ST r16, $R0
LD $R0, r16
CMP $R0 , 0
BEZ L9
MOV $R0, #0
ST u, $R0
LD $R0, #0
```

```
ST r17, $R0
LD $R0, r17
CMP $R0 , 0
BEZ L10
LD $R0, y
MOV $R1, #9
ADD $R2, $R0, $R1
ST r18, $R2
LD $R0, r18
MOV $R1, $R0
ST y, $R1

L10:
MOV $R0, #1000
ST f, $R0

L9:
LD $R0, A
MOV $R1, #0
CMP $R0, $R1
MOV $R2, #1
MOVLZ $R2, #0
ST r19, $R2
LD $R0, p
MOV $R1, #10
CMP $R0, $R1
MOV $R2, #1
MOVGEZ $R2, #0
ST r20, $R2
LD $R0, r19
LD $R1, r20
AND $R2, $R0, $R1
ST r21, $R2
LD $R0, c
MOV $R1, #25
CMP $R0, $R1
MOV $R2, #1
MOVGEZ $R2, #0
```

```
ST r22, $R2
LD $R0, r21
LD $R1, r22
AND $R2, $R0, $R1
ST r23, $R2
LD $R0, r23
CMP $R0 , 0
BEZ L11
MOV $R0, #20
ST A, $R0
BR L12

L11:
MOV $R0, #0
ST ty, $R0

L12:
```

Architecture of Language

Our compiler supports the following language features -

- We only handle code present inside the main function.
- All types of arithmetic and logical expressions are handled.
- For and IF ELSE statements are also handled.
- We have taken care of indentation.
- We have also taken care of nested for and if else loop.

Literature Survey

- https://www.geeksforgeeks.org/introduction-to-yacc/
  By referring this page, we improved our yacc programming skills
- https://drive.google.com/drive/u/0/folders/1_HEqMdujON3L1ICtA059LjPY1HxhY0F5
  (Class Notes) We referred this link for conceptual to build a compiler.

Context Free Grammar

```
PHASE_START: START
             ;


START:  ASSIGN NEWLINE START
        | FOR_STMT START
        | IF_STMT START
        | SIMPLE_STMT NEWLINE START
        |error
        |NEWLINE START
        |;

START_FOR_INDENT:  INDENT_GRAMMER ASSIGN NEWLINE START_FOR_INDENT
              | INDENT_GRAMMER FOR_STMT NEWLINE START_FOR_INDENT
              | INDENT_GRAMMER INNER_IF NEWLINE START_FOR_INDENT
              | INDENT_GRAMMER SIMPLE_STMT NEWLINE START_FOR_INDENT
              | ;

INTERMEDIATE: INDENT_GRAMMER ASSIGN NEWLINE START_FOR_INDENT
             | INDENT_GRAMMER FOR_STMT NEWLINE START_FOR_INDENT
             | INDENT_GRAMMER INNER_IF NEWLINE START_FOR_INDENT
             | INDENT_GRAMMER SIMPLE_STMT NEWLINE START_FOR_INDENT
               ;

INDENT_GRAMMER: T_INDENT GOP
                ;

GOP: T_INDENT GOP
    | ;

SIMPLE_STMT : T_INT'('A')'
             | T_LIST'('B')'
             | T_LIST '('A')'
             | T_PRINT'(' UI ')'
             | A
             | B
              ;
A: T_INPUT '('GI')'
    ;
B: T_INPUT'('GI')' '.' T_SPLIT'('')'
   ;
```

```
UI: M
   |;

GI: T_STRING
   |;

ASSIGN: T_ID T_ASSIGN E
        | T_ID T_ASSIGN SIMPLE_STMT
         ;

E: E '+' T
   | E T_minus T
   | T
    ;

T: T '*' F
   | T'/'F
   | F
    ;

F: Q T_star F
  | Q
   ;

Q: L
  | '('E')'
  | VAR
   ;

VAR: T_ID
   | T_DIGIT
   | T_STRING
     ;

L: '[' M ']'
   | '[' ']'
    ;
M: Q ',' M
  | Q;
REL_OP: T_LT
       | T_GT
       | T_GE
       | T_LE
```

```
        | LIST_OP
         ;

LIST_OP: T_EEQ
         |T_NEQ
          ;


COND: COND T_OR AND_STMT
     | AND_STMT
      ;

AND_STMT: AND_STMT T_AND NOT_STMT
          | NOT_STMT
           ;

NOT_STMT: T_NOT NOT_STMT
          | N
           ;

N: '('COND')'
    | REL_EXP
    | T_TRUE
    | T_FALSE
      ;

REL_EXP: VAR REL_OP VAR
          | L LIST_OP L
          | VAR LIST_OP L
          | L LIST_OP VAR
           ;

INNER_SUITE: {k=0;indent++;} INTERMEDIATE {fun_cond(fun_indent(1));$$ =
$2;} ;

INNER_IF: T_IF '(' COND ')' ':' NEWLINE INNER_SUITE  ELIF_STMT;

SUITE: NEWLINE {k=0;indent++;} INTERMEDIATE {fun_cond(fun_indent(1));$$ =
$3;} ;
IF_STMT: T_IF '(' COND ')' ':' SUITE ELIF_STMT
        ;

ELIF_STMT: T_ELIF '(' COND ')' ':' SUITE ELIF_STMT
```

```
           | T_ELSE ':' SUITE
           |
             ;

FOR_STMT: T_FOR T_ID T_IN TARGET ':' SUITE
            ;

TARGET: T_RANGE'('C')'
        | T_RANGE'('C',' C')'
        | T_RANGE'('C','C','C')'
        | T_ID
        | L
         ;

C: T_ID
   |T_DIGIT
    ;
```

Design Strategy

INPUT:

A=10
if(A>10):
print("df")
for i in range(10):
y=A+20

- **Symbol Table Creation -**
  - We used linked list to store the values in the symbol table.
  - In the symbol table we store token name, token type, line number, the value of the token.
  - We use symbol table to use the value which is stored in the variable for expression evaluation.

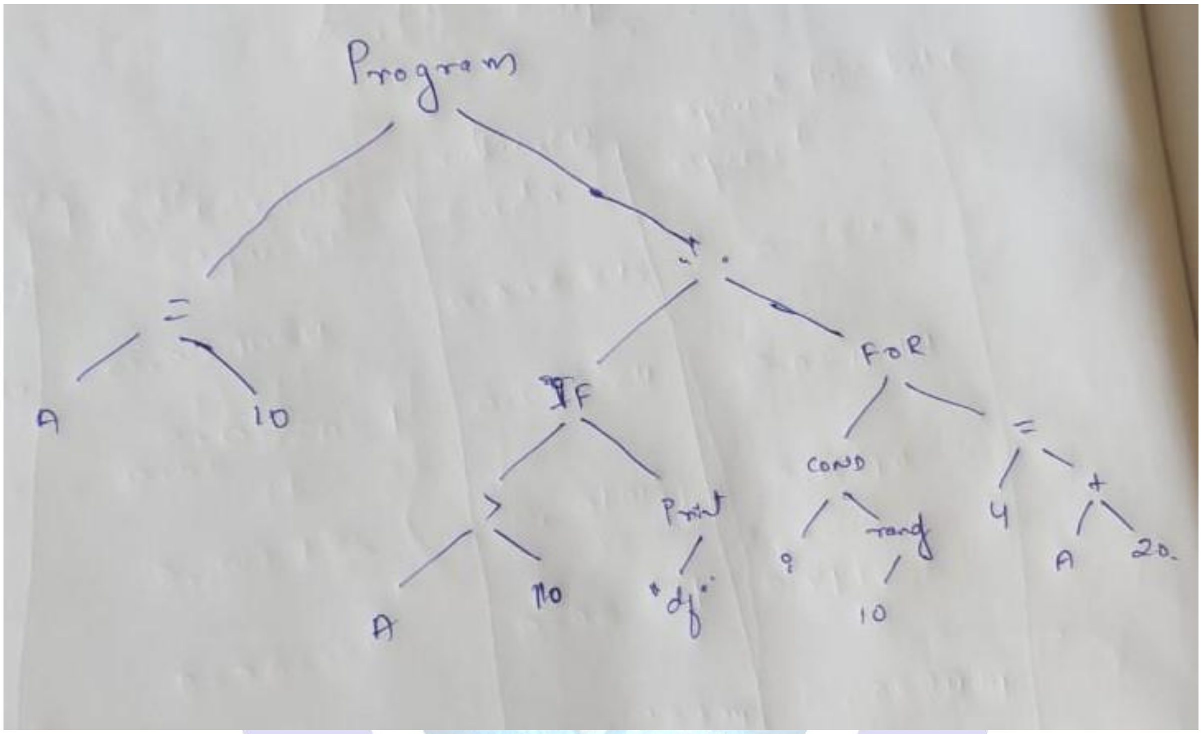
```
struct symtable{
    int line;
    //char type[10];
    char name[30];
    char value[50];
    int scope;
    int type_flag;
    struct symtable* next;};
typedef struct symtable Node;
```

SYMBOL TABLE

LINE	ID	VALUE	Type
5	y	30	numeric
1	A	10	numeric

- **Abstract Syntax Tree -**
  - The abstract syntax tree is generated as we parse the program.
  - A tree node is created based on the type of tokens parsed.
  - We handle basic types of nodes, that is – constant, IF, FOR, ELSE, identifier and operation.
  - When the control reaches the end of input if parsed successfully the tree is passed to a function that traverses the tree in preorder, postorder, inorder and prints it.

```
typedef struct node{
    char value[50];
    char expr_result[15];
    char reg_name[15];
    struct node *left;
    struct node *right;
```




```
Preorder Traversal
PROGRAM  = A 10  IF > A 10   PRINT "df"  FOR COND i range 10  = y + A 20
Inorder Traversal
A = 10  A > 10 IF "df" PRINT    i COND 10 range FOR y = A + 20    PROGRAM
Postorder Traversal
A 10 = A 10 > "df" PRINT   IF i 10 range COND y A 20 + =  FOR     PROGRAM
```

- **Intermediate Code Generation -**
    - For intermediate code generation we make use stack to define which expression comes under which indentation and loop.
    - We use structure to store the value and expression and inside structure we store the block to which the expression belongs to.
    - when it reaches the end of the expression, a function is called to print the contents of the structure according to intermediate code format including temporaries, Labels, and loops.

```
struct quad
{
    char op[5];
    char arg1[10];
    char arg2[10];
    char result[10];
    int scope;
    char block[10];
    int indi;
    char actual_result[10];
}QUAD[500];
```

```
node *make_node(char *value, node *left, node *right)
{
    node *new_node = malloc(sizeof(node));
    strcpy(new_node->value, value);
    new_node -> left = left;
    new_node -> right = right;
    return new_node;
}
```

- **Code Optimization -**
    - We performed three optimizations namely, constant folding, constant propagation, and common subexpression elimination.
    - For constant folding we check if the two arguments are numbers, if they are we replace the expression with its value.
    - Constant propagation is the process of substituting the values of known constants in expressions at compile time. For this we find a variable with a constant value and traverse ahead to find the spots where the variable has been used next and replace the variable with its value until the variable's value has been changed.
    - Common subexpression elimination is a process where if the same expression is used more than once in code, we only calculate the value of the expression once and then use that values in the future

- **Error Handling - strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator) -**
    - The scanner doesn't crash when it comes across unknown symbols.
    - The parser doesn't stop parsing on encountering error and prints a syntax error at the corresponding line number.
    - We inform the invalid indentation in the code and parsing starts from next line.
    - The code generator expects error free code to be passed to it.

- **Target Code Generation**
  - Target code is generated using a simple load-use-store mechanism.
  - This is done by looking at three address code line by line.
- **Instructions to run our project**
  - make
  - ./icg
  - for target code generation use : “python3 codegen.py < output_file.txt” in addition to the above command

Results and Shortcomings

- Our compiler is a very minimal and basic compiler, and handles programs which purely perform mathematical computations.
- Error printing of our compiler is not exhaustive and too simple to handle complicated errors.
- Assembly code outputted will be correct for any type of program that our grammar parses. However, the cost of the program is high due to a simple assembly generation algorithm.

Snapshots (of different outputs)

For and nested For loop test case -

```
A=10
T=100
for i in range(10):
      A=A+20

#dfjshdf

for x in range(1,10):
      T=10+T+A
      for j in range(5,25):
            y=0

g=0
```

Our compiler's output -

```
main:
MOV $R0, #10
ST A, $R0
MOV $R0, #100
ST T, $R0
MOV $R0, #0
ST i, $R0

L1:
LD $R0, i
LD $R1, #10
SUBS $R0, $R0, $R1
BGEZ $R0, L2
LD $R0, A
MOV $R1, #20
ADD $R2, $R0, $R1
```

```
ST r1, $R2
LD $R0, r1
MOV $R1, $R0
ST A, $R1
LD $R0, i
MOV $R1, #1
ADD $R2, $R0, $R1
ST i, $R2
BR L1

L2:
MOV $R0, #1
ST x, $R0

L3:
LD $R0, x
LD $R1, #10
SUBS $R0, $R0, $R1
BGEZ $R0, L4
MOV $R0, #10
LD $R1, T
ADD $R2, $R1, $R0
ST r3, $R2
LD $R0, r3
LD $R1, A
ADD $R2, $R0, $R1
ST r4, $R2
LD $R0, r4
MOV $R1, $R0
ST T, $R1
MOV $R0, #5
ST j, $R0

L5:
LD $R0, j
LD $R1, #25
SUBS $R0, $R0, $R1
BGEZ $R0, L6
```

```
MOV $R0, #0
ST y, $R0
LD $R0, j
MOV $R1, #1
ADD $R2, $R0, $R1
ST j, $R2
BR L5

L6:
LD $R0, x
MOV $R1, #1
ADD $R2, $R0, $R1
ST x, $R2
BR L3

L4:
MOV $R0, #0
ST g, $R0

L8:
```

If Elif and else case code -

```
bh=10+20
y=12

if(true):
      u=10*120+bh
elif(y>10):
      z=100*y


if(bh>=10):
      y=100
      if(true):
            bh=20+y
      h=y+60
```

Our compiler’s output -

```
main:
MOV $R0, #30
ST bh, $R0
MOV $R0, #12
ST y, $R0
LD $R0, #1
ST r1, $R0
LD $R0, r1
CMP $R0 , 0
BEZ L1
MOV $R0, #10
MOV $R1, #120
MUL $R2, $R1, $R0
ST r2, $R2
LD $R0, r2
LD $R1, bh
ADD $R2, $R0, $R1
ST r3, $R2
LD $R0, r3
MOV $R1, $R0
ST u, $R1
BR L2

L1:
LD $R0, y
MOV $R1, #10
CMP $R0, $R1
MOV $R2, #1
MOVLEZ $R2, #0
ST r4, $R2
LD $R0, r4
CMP $R0 , 0
BEZ L2
```

```
MOV $R0, #100
LD $R1, y
MUL $R2, $R1, $R0
ST r5, $R2
LD $R0, r5
MOV $R1, $R0
ST z, $R1
L2:
LD $R0, bh
MOV $R1, #10
CMP $R0, $R1
MOV $R2, #1
MOVLZ $R2, #0
ST r6, $R2
LD $R0, r6
CMP $R0 , 0
BEZ L3
MOV $R0, #100
ST y, $R0
LD $R0, #1
ST r7, $R0
LD $R0, r7
CMP $R0 , 0
BEZ L4
MOV $R0, #20
LD $R1, y
ADD $R2, $R1, $R0
ST r8, $R2
LD $R0, r8
MOV $R1, $R0
ST bh, $R1

L4:
LD $R0, y
MOV $R1, #60
ADD $R2, $R0, $R1
ST r9, $R2
LD $R0, r9
```

```
MOV $R1, $R0
ST h, $R1
L3:
```

Conclusions

- It’s very easy to type a command to compile a program, but writing and understanding all phases of a compiler is challenging.
- Powerful tools like Lex and Yacc can be used in order to replicate or build a compiler.
- Working on this project has helped us grasp the internals and all the phases of a compiler.

Further Enhancements

- Handling more data types.
- Handling arrays, pointers etc.
- Function calls and argument parsing.
- More efficient assembly code generator.

References/Bibliography

- Assembly Code Generation - https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lects/final.codegen.pdf
- Course notes.