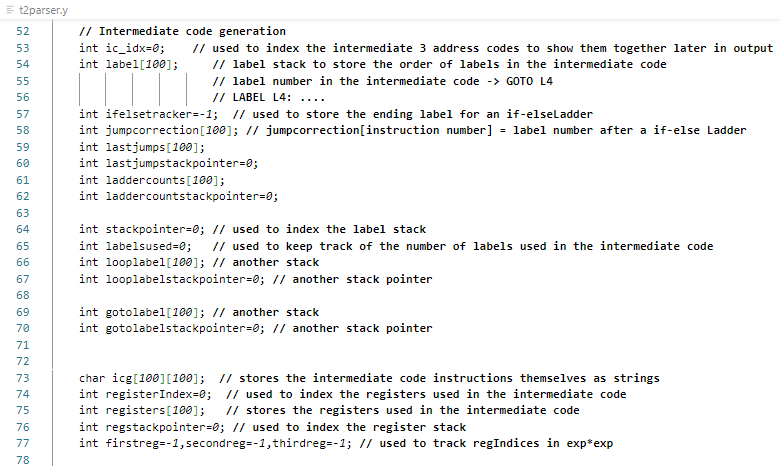
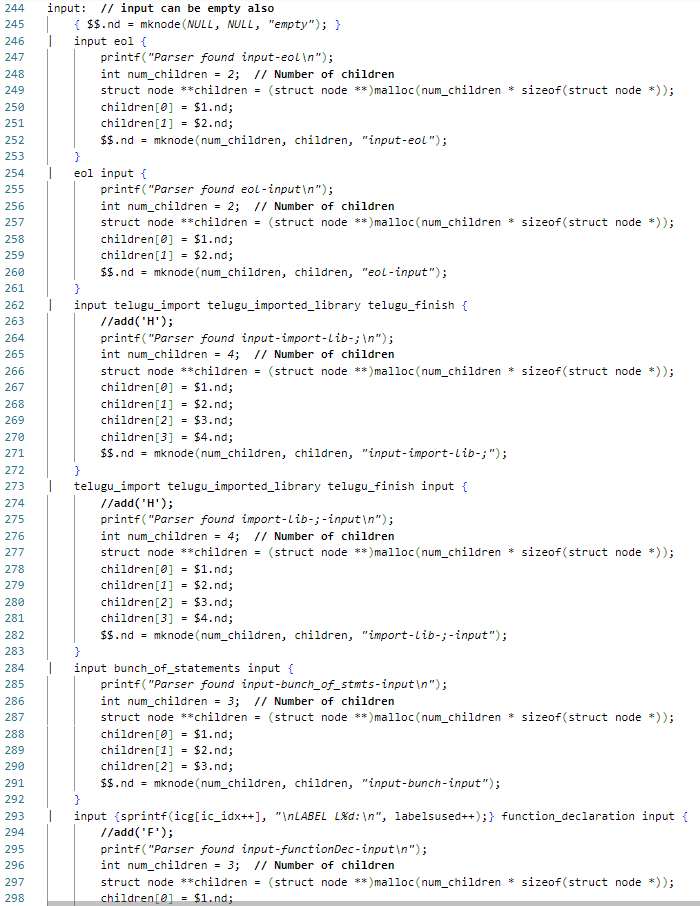
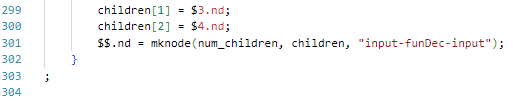
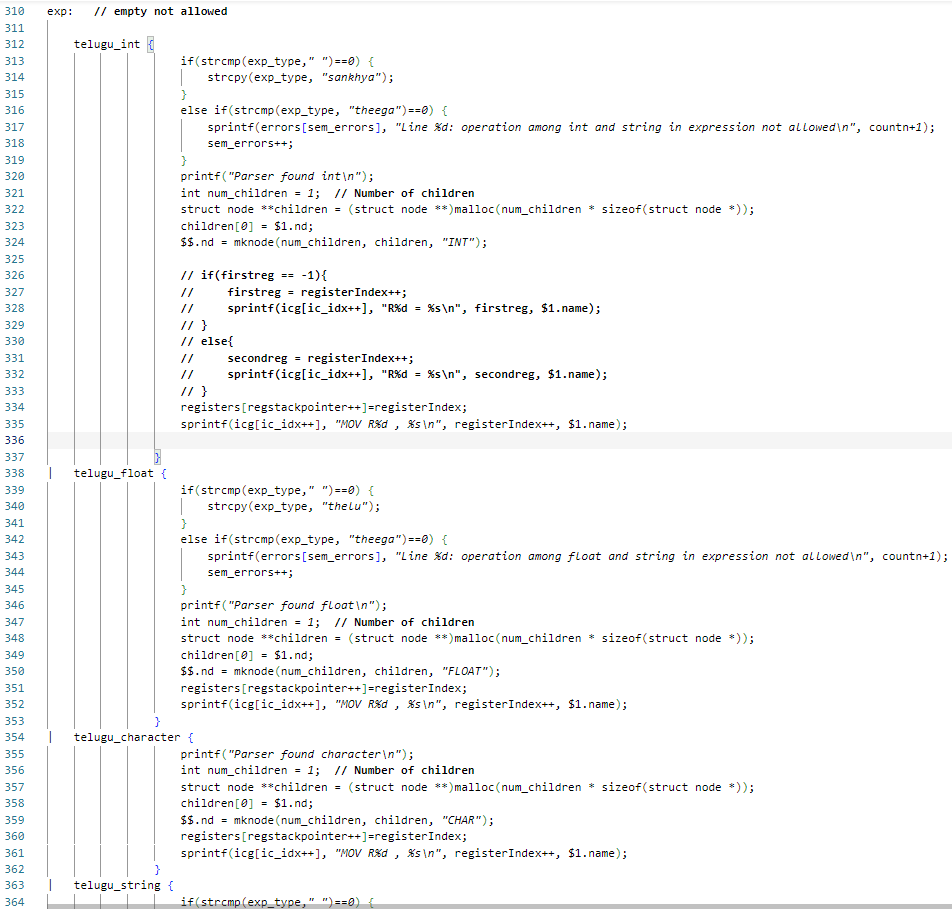
**UPDATES IN PARSER CODE:**

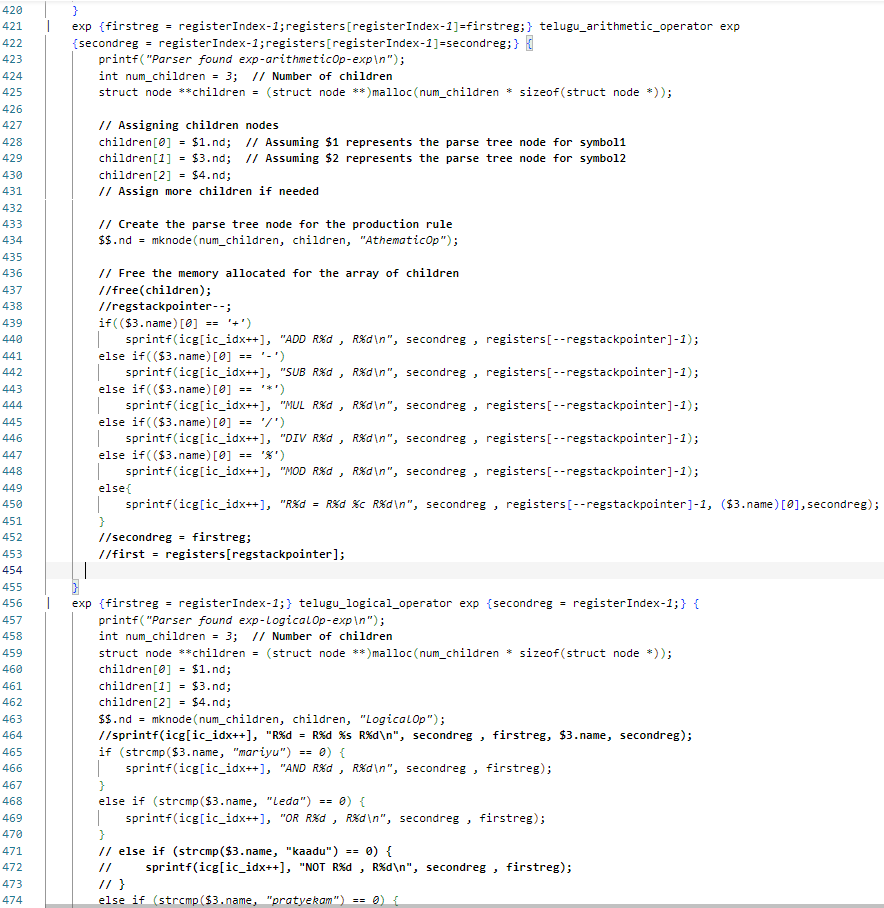


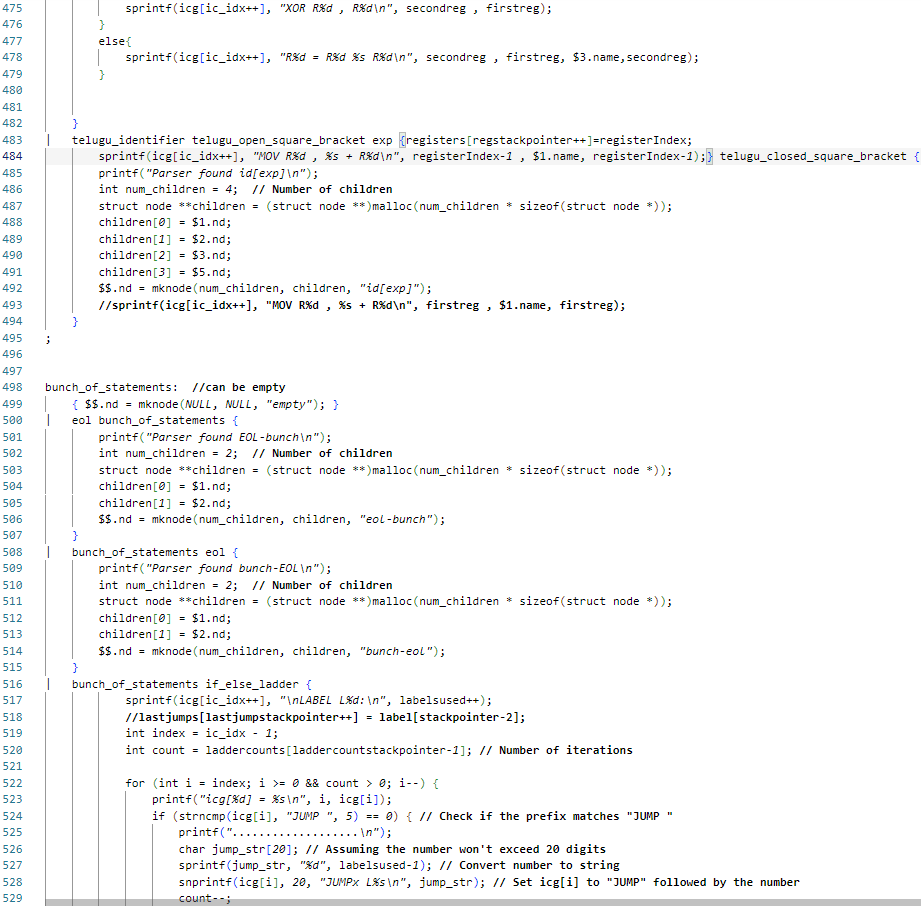


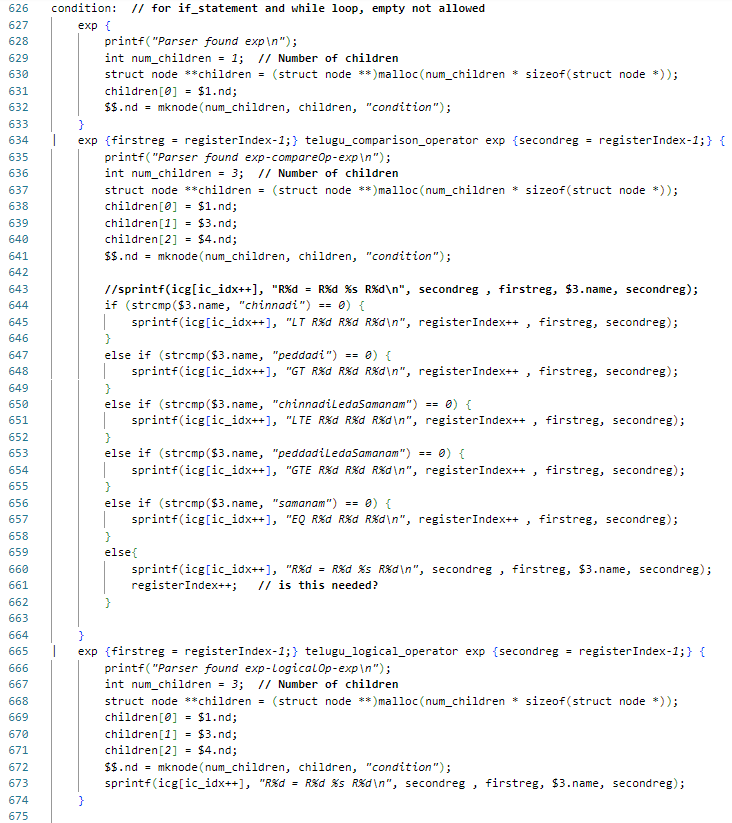


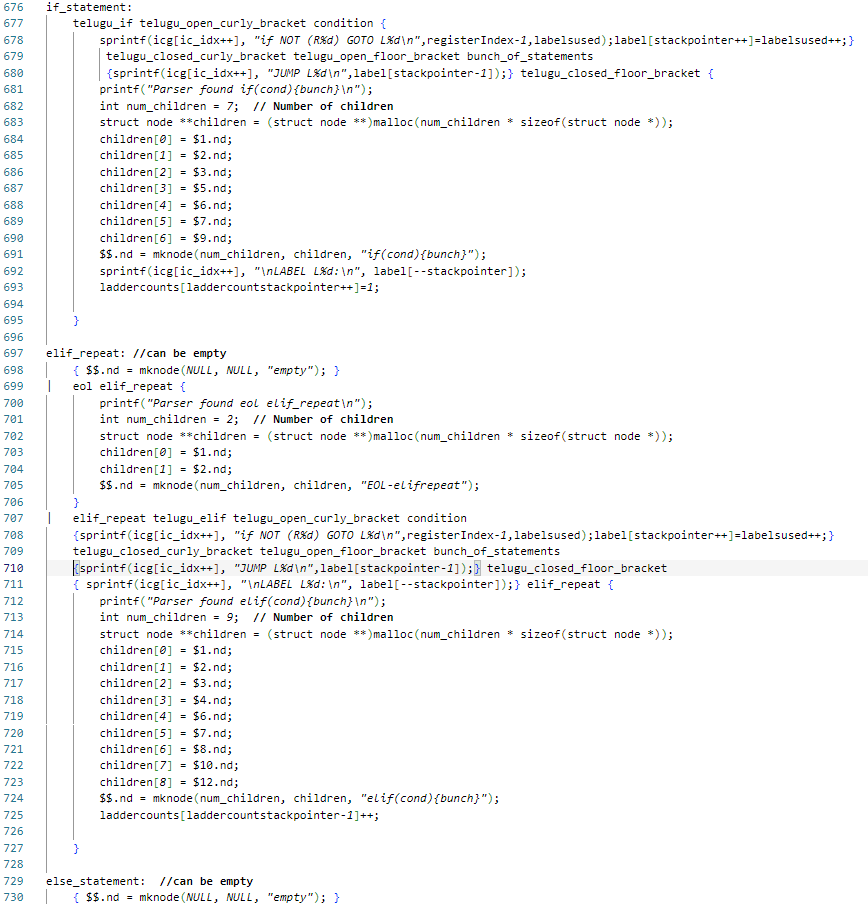


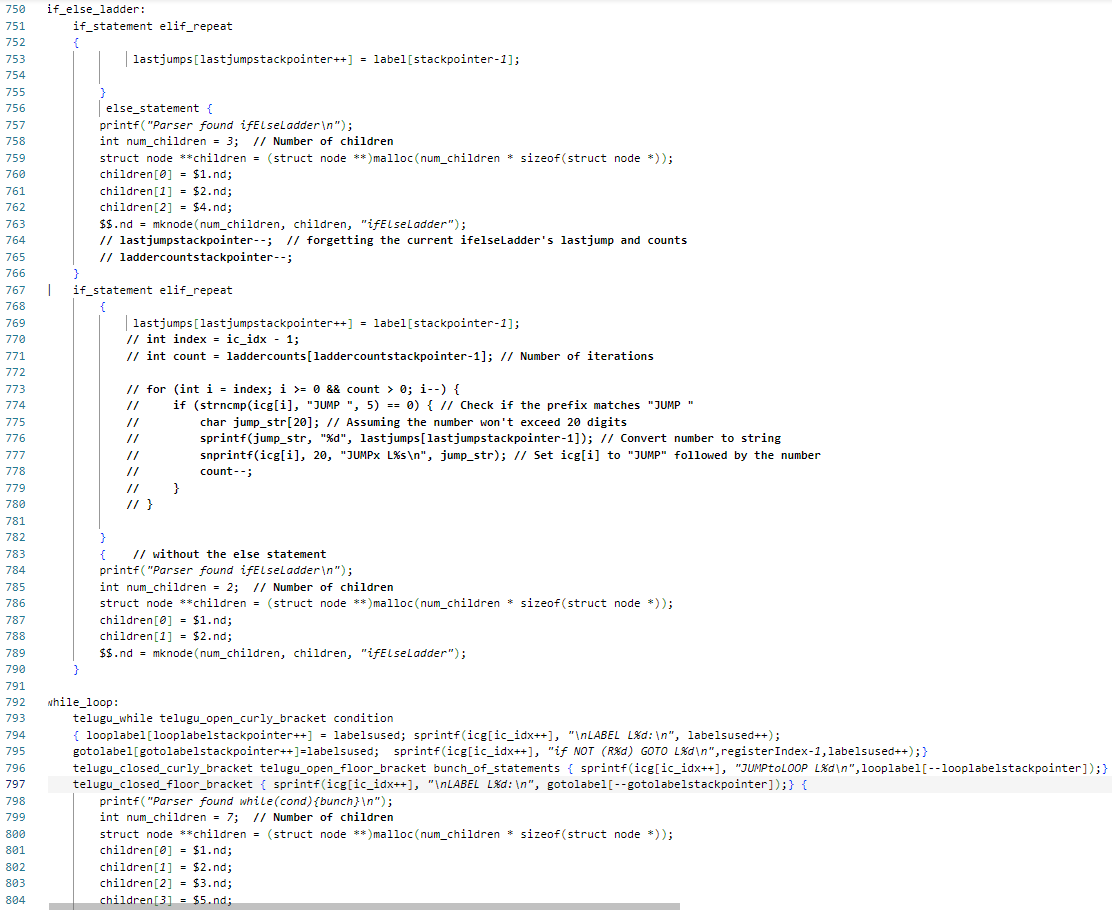


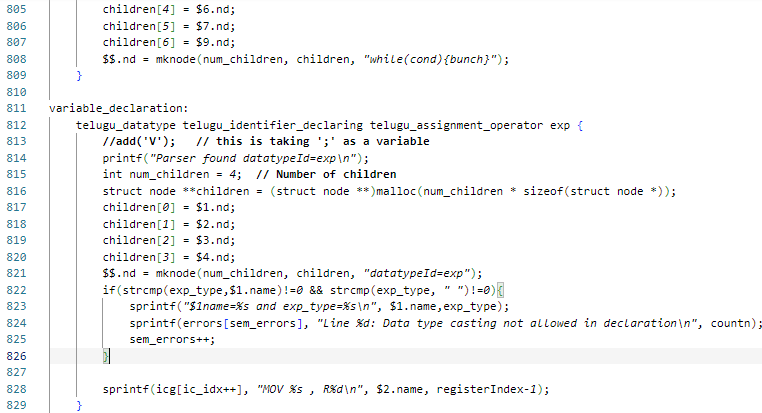


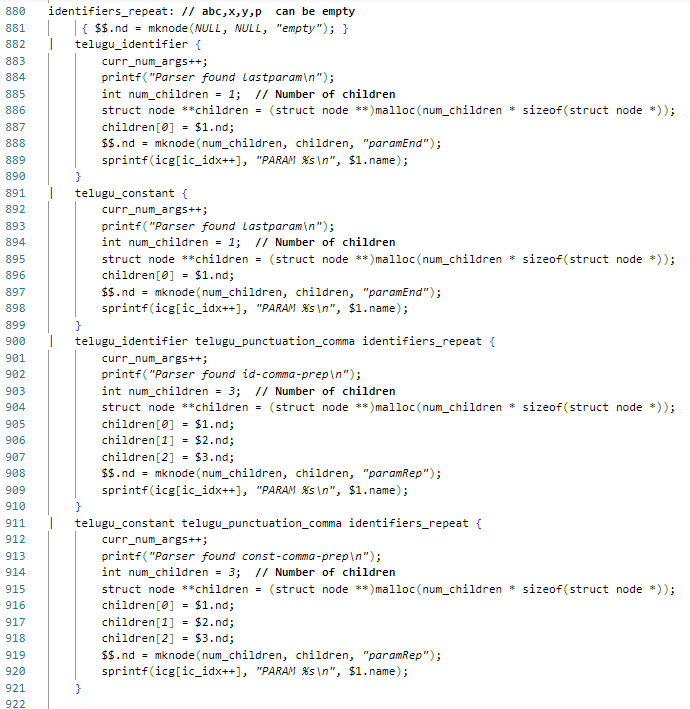


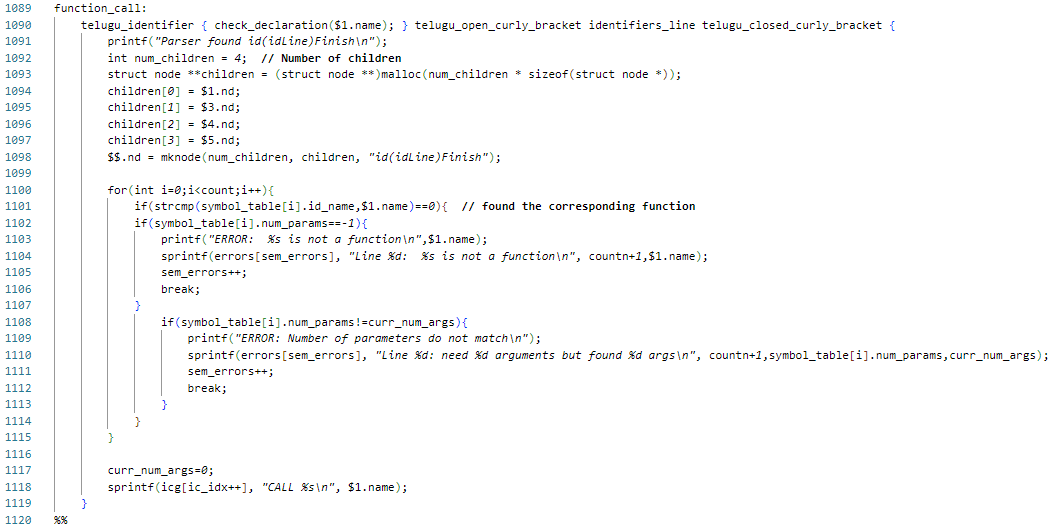
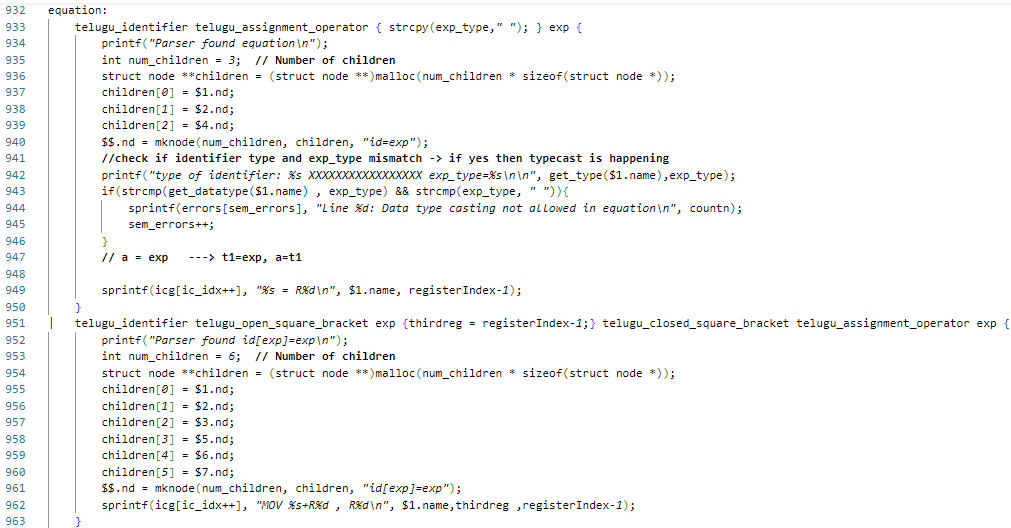


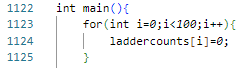


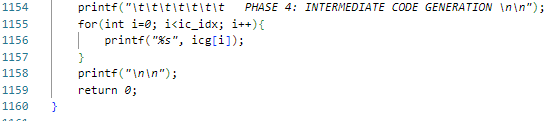






a





**Syntax directed translations for generating proper Three-Address Codes for the grammar:**

Note: Op Rx Ry === Rx <- Rx Op Ry

It is assumed that the execution starts from the first command to last command in order after the Intermediate Code has been generated.

1.Variable declaration:

sankhya p=15; -> MOV R1, 15 // moves the value 15 into register R1

MOV p, R1 // moves value in R1 to register ‘p’

2.expressions and equations

p = p-8\*(9/2); -> MOV R2,8

MOV R3,9

MOV R4,2

DIV R3,R4 // put R3/R4 in R3

MUL R2,R3

SUB p,R2

3. Array indexing

sankhya b=arr[3]; -> MOV R1 , 3

MOV R1 , arr+R1 // move arr[R1] into R1

MOV b , R1

arr[2] = 3+4; -> MOV R2 , 2

MOV R3 , 3

MOV R4 , 4

ADD R4 , R3

MOV arr+R2 , R4 // move R4 into arr[R2]

4. Conditional Branching:

If NOT (R1) GOTO L4 // is value in R1 returns false then goto Label L4

5. Un-Conditional Branching:

JUMP L4 // go to label L4 and proceed execution

6. If-Else Ladders:

okavela(condition1){

// do something1

}

lekapothe{

// do something2

}

Corresponding Intermediate Code for the above example:

if NOT (condition1) GOTO L0

JUMP L1

LABEL L0:

// something1

LABEL L1:

//something2

6.Loops:

aithaunte(condition){

//something1

}

//something2

Corresponding Intermediate Code for the above example:

LABEL L0:

If NOT (condition) GOTO L1

//something1

JUMP L0

LABEL L1:

//something2

7.Function Call:

pani sum(sankhya a, sankhya b){ // function declaration

chupi("addition is ",a+b);

ivvu;

}

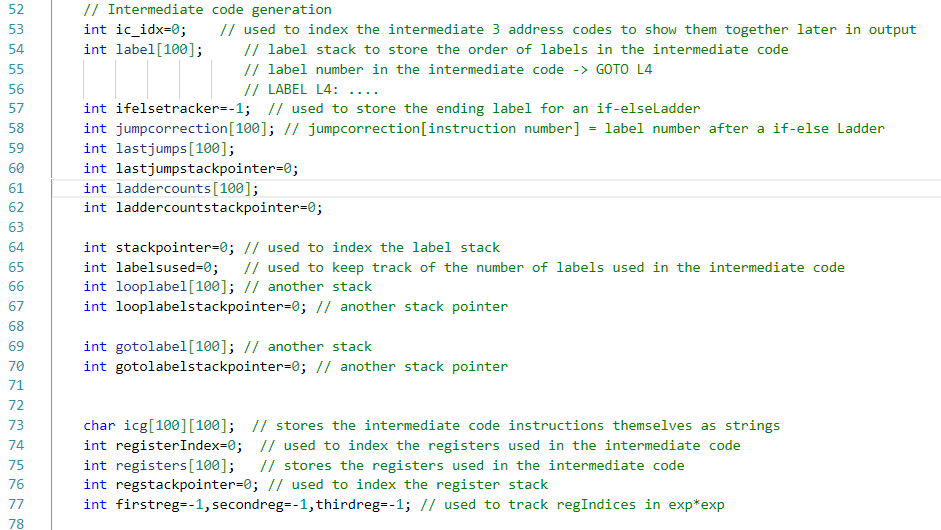
sum(15,24); // calling the function

Intermediate Code:

PARAM 15

PARAM 24

CALL sum



**Matching the GOTO and LABELS**

Stacks have been used to handle matching GOTO statements with LABELS

**On seeing a branch statement:**

If NOT (condition) GOTO L0 // push “GOTO L0” on to the labelsused stack

………………………

LABEL L0: // when the time comes to display the label

// pop the topmost Label from the labelsused stack

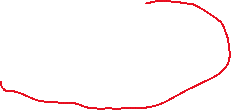
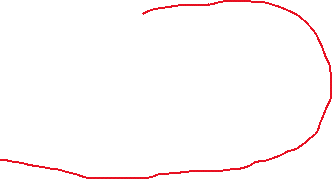
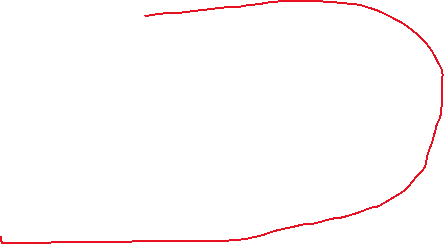
**In case of Nested If-else:**

if NOT (condition1) GOTO L0

if NOT (condition2) GOTO L1

if(NOT condition3) GOTO L2

………………………..



LABEL L2:

LABEL L1:

LABEL L0:

**On seeing a Loop:**

LABEL L0:

If NOT (loop condition) GOTO L1

……………..

JUMP L0

LABEL L1: // out of the previous loop

**In case of Nested Loops:**

LABEL L0:



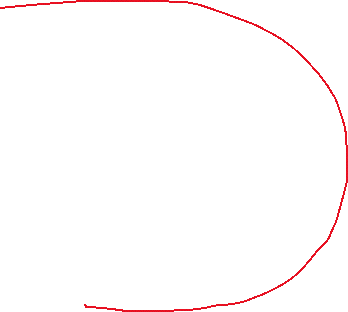
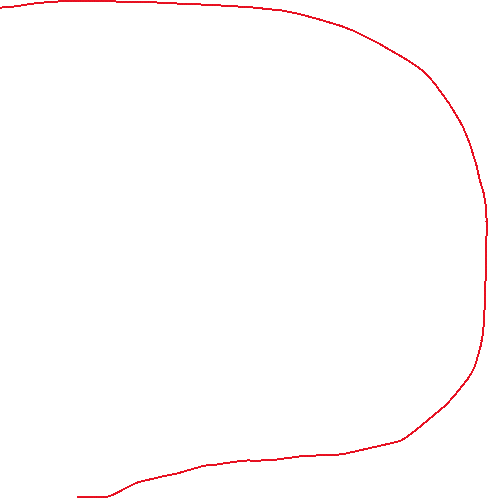
if NOT (condition1) GOTO L1

LABEL L2:

if NOT (condition2) GOTO L3

LABEL L4:

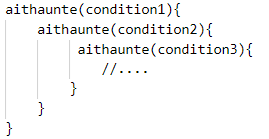
if NOT (condition3) GOTO L5



JUMPtoLOOP L4

LABEL L5:

JUMPtoLOOP L2

LABEL L3:

JUMPtoLOOP L0

LABEL L1:

**BACKPATCHING**

okavela (condition1){ // IF

…………….

}

lekaokavela (condition2){ // ELSE IF

// LABEL L0

………………

}

lekaokavela (condition3){ // ELSE IF

// LABEL L1

……………….

}

Lekapothe{ // ELSE

// LABEL L2

………………

}

// LABEL L3

………………..

In the Above case, ideally, our 3-address code should be:

if NOT (condition1) GOTO L0

JUMP L3 // How do we know L3? (it could be any label)

LABEL L0:

if NOT (condition2) GOTO L1

JUMP L3

LABEL L1:

if NOT (condition3) GOTO L2

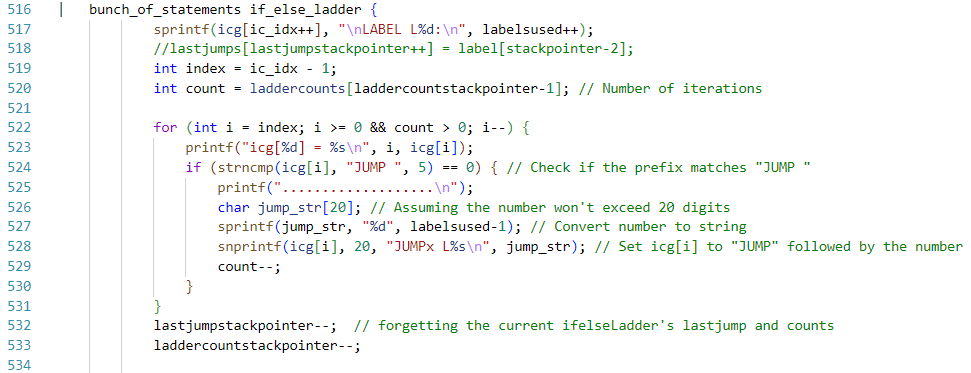
JUMP L3

LABEL L2:

LABEL L3:

In the above example, after executing ‘if’ content, we have to start execution after the entire if-else ladder. Initially in the jump statement we assign some dummy label. We maintain a stack to count the number of if and else ifs in each if-else ladder.

Suppose our current label used is L15 and the current if-else ladder is finished parsing and we found out that there are **3 else-Ifs** in it, then we can derive that the Jump statement on all of these ifs and elseifs should be corrected to L15. **NOW WE START ITERATING AMONG THE PREVIOUS 1+3 JUMP STATEMENTS BACKWARDS AND CORRECT THEM BY SETTING NEW JUMP = JUMP L15**

****

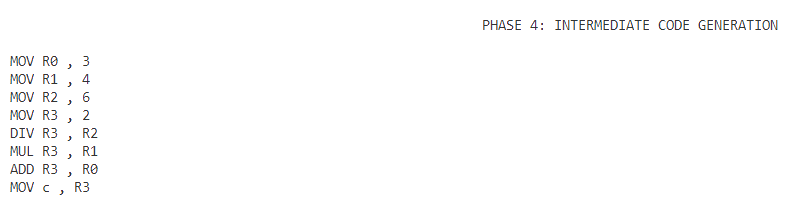
Similarly, for LOOPs, we maintain stacks indicating which LABEL our current jump statement corresponds to. Similar to “balanced parenthesis”, we will assign the jump statements accordingly. And after the scope of a loop is finished, we iterate the previous jump statements for previous loops in the same scope and assign change their JUMP dummy to JUMP Lnew.

**INPUT 1:**

Sub-Expressions

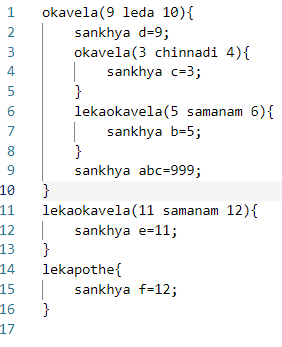


**OUTPUT 1:**

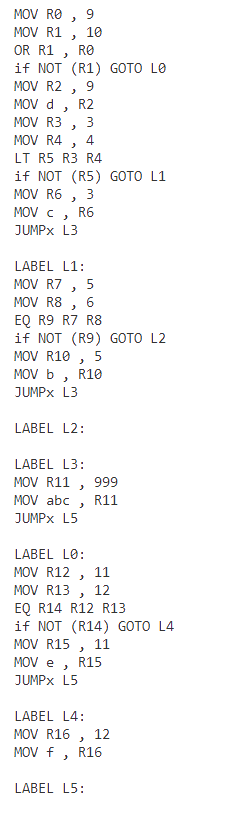


**INPUT 2:**

If-else Ladders

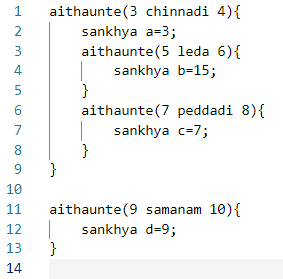


**OUTPUT 2:**

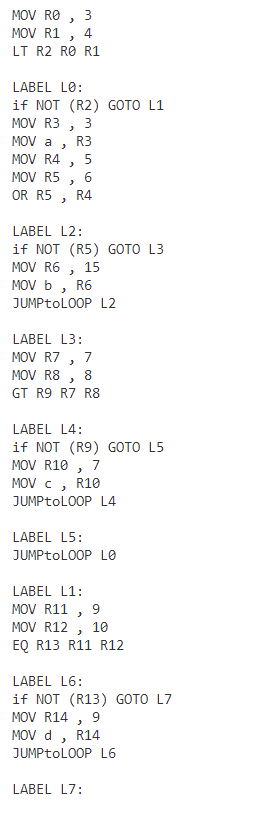


**INPUT 3:**

Loops

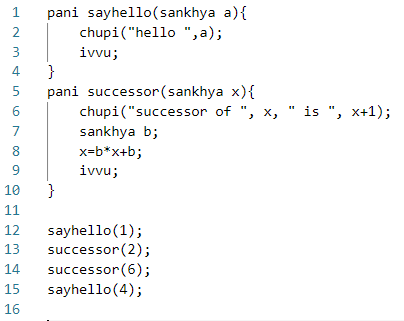


**OUTPUT 3:**

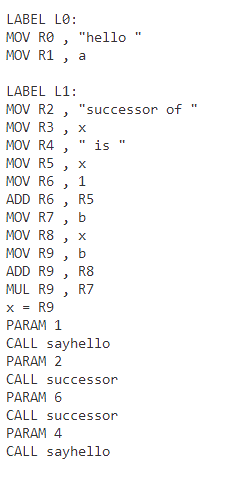


**INPUT 4:**

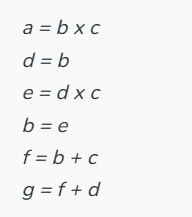
Function Calls



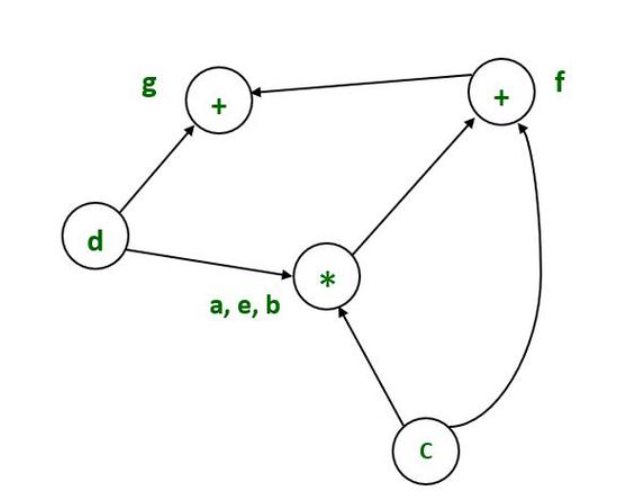
**OUTPUT 4:**



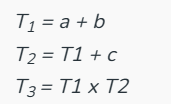
**DIRECTED ACYCLIC GRAPHS for Sample TELUGU programs**

**Input 1:**

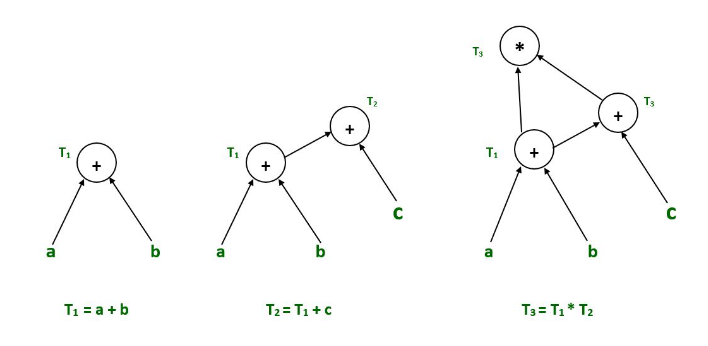
**Output 1:**

****

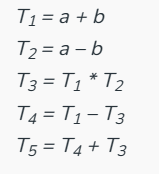
**Input 2:**

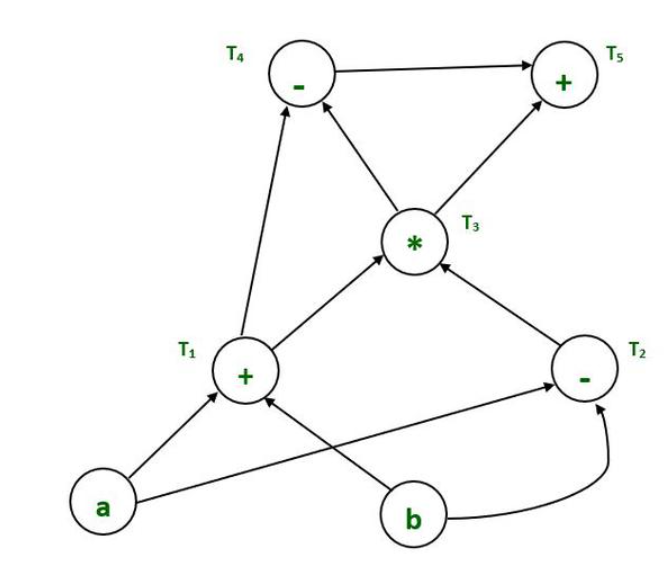
****

**Output 2:**

****

**Input 3:**

****

**Output 3:**