**UPDATED PARSER CODE:**

%{

/\*definitions\*/

#include <stdio.h>

#include<string.h>

#include<stdlib.h>

#include<ctype.h>

#include <limits.h>

// #include"lex.yy.c" // this is creating multiple definitions

// Declaration of tree

struct node {

int num\_children; // Number of children

struct node \*\*children; // Array of pointers to child nodes

char \*token; // Token associated with the node

};

struct node \*head;

struct node\* mknode(int num\_children, struct node \*\*children, char \*token) ;

void printtree(struct node\* tree);

void printInorder(struct node \*tree);

void add(char);

void insert\_type();

int search(char \*);

void check\_declaration(char \*);

void check\_return\_type(char \*);

char \*get\_type(char \*);

char \*get\_datatype(char \*);

struct dataType {

char \* id\_name;

int used; // for optimization stage - to check if the declared variable is used anywhere else in the program

char \* data\_type;

char \* type;

int line\_no;

int thisscope;

int num\_params;

int range[10][2]; // [start index of first computation in icg,end index of assignment in icg for that chunk]

int range\_count;

} symbol\_table[10000];

int count=0;

int q;

char type[10];

extern int countn;

extern int scope;

int curr\_num\_params=0;

int curr\_num\_args=0;

extern char\* yy\_text;

char exp\_type[30]; // will be empty if no expression is there

int sem\_errors=0;

char buff[10000];

char errors[10][10000];

int oldscope=-1;

// Intermediate code generation

int ic\_idx=0; // used to index the intermediate 3 address codes to show them together later in output

int label[10000]; // label stack to store the order of labels in the intermediate code

// label number in the intermediate code -> GOTO L4

// LABEL L4: ....

int ifelsetracker=-1; // used to store the ending label for an if-elseLadder

int jumpcorrection[10000]; // jumpcorrection[instruction number] = label number after a if-else Ladder

int lastjumps[10000];

int lastjumpstackpointer=0;

int laddercounts[10000];

int laddercountstackpointer=0;

int stackpointer=0; // used to index the label stack

int labelsused=0; // used to keep track of the number of labels used in the intermediate code

int looplabel[10000]; // another stack

int looplabelstackpointer=0; // another stack pointer

int insNumOfLabel[10000]; // used to store the instruction number of each label

int gotolabel[10000]; // another stack

int gotolabelstackpointer=0; // another stack pointer

int rangestart=-1,rangeend=-1; // used to store the range of instructions for a chunk of variable declaration/assignment code temperorily

int uselessranges[10000][2]; // used to store the range of instructions for a chunk of useless variable declaration/assignment code overall

int uselessrangescount=0;

char icg[10000][20]; // stores the intermediate code instructions themselves as strings

int isleader[10000]; // stores whether the instruction is a leader or not

int registerIndex=0; // used to index the registers used in the intermediate code

int registers[10000]; // stores the registers used in the intermediate code

int regstackpointer=0; // used to index the register stack

int firstreg=-1,secondreg=-1,thirdreg=-1; // used to track regIndices in exp\*exp

//finish writing the reserved words,there are more reserved words

const int reserved\_count = 13; // why is this not working for reserved[reserved\_count][20]???

char reserved[13][20] = {"sankhya", "thelu", "aksharam", "pani", "okavela", "lekaokavela",

"lekapothe", "aithaunte", "ivvu", "thechko","theesko","chupi","theega"};

// Function to mark a variable as used if found in the symbol table

void markVariableAsUsed(const char \*id\_name) {

for (int i = 0; i < 10000; ++i) {

if (symbol\_table[i].id\_name != NULL && strcmp(symbol\_table[i].id\_name, id\_name) == 0) {

symbol\_table[i].used = 1;

return;

}

}

}

// Function to search for an identifier in the symbol table and return its index if found

int findIdentifierIndex(char \*id\_name) {

for (int i = 0; i < 10000; i++) {

if (symbol\_table[i].id\_name != NULL && strcmp(symbol\_table[i].id\_name, id\_name) == 0) {

return i; // Return the index if found

}

}

return -1; // Return -1 if not found

}

// Function to swap two ranges

void swapRanges(int range1[], int range2[]) {

int tempStart = range1[0];

int tempEnd = range1[1];

range1[0] = range2[0];

range1[1] = range2[1];

range2[0] = tempStart;

range2[1] = tempEnd;

}

// Function to sort the 2D array of ranges

void sortRanges(int ranges[][2], int rangeCount) {

for (int i = 0; i < rangeCount - 1; i++) {

for (int j = 0; j < rangeCount - i - 1; j++) {

if (ranges[j][0] > ranges[j + 1][0]) {

swapRanges(ranges[j], ranges[j + 1]);

}

}

}

}

%}

%error-verbose

%union {

struct var\_name {

char name[10000];

struct node\* nd;

} nd\_obj;

}

%token<nd\_obj> EOL TELUGU\_INT TELUGU\_FLOAT TELUGU\_ARITHMETIC\_OPERATOR TELUGU\_COMPARISON\_OPERATOR TELUGU\_ASSIGNMENT\_OPERATOR TELUGU\_LOGICAL\_OPERATOR

%token<nd\_obj> TELUGU\_DATATYPE TELUGU\_IF TELUGU\_ELIF TELUGU\_ELSE TELUGU\_WHILE TELUGU\_OPEN\_FLOOR\_BRACKET TELUGU\_CLOSED\_FLOOR\_BRACKET

%token<nd\_obj> TELUGU\_IDENTIFIER TELUGU\_STRING TELUGU\_OPEN\_CURLY\_BRACKET TELUGU\_CLOSED\_CURLY\_BRACKET TELUGU\_OPEN\_SQUARE\_BRACKET TELUGU\_CLOSED\_SQUARE\_BRACKET

%token<nd\_obj> TELUGU\_PUNCTUATION\_COMMA TELUGU\_NEWLINE TELUGU\_FINISH TELUGU\_FUNCTION TELUGU\_RETURN TELUGU\_CHARACTER TELUGU\_PRINT TELUGU\_IMPORT TELUGU\_INPUT

%type<nd\_obj> program,input,exp,condition,if\_statement,while\_loop,variable\_declaration,parameters\_repeat,equation,parameters\_line,function\_declaration,function\_content,bunch\_of\_statements,elif\_repeat,

else\_statement,if\_else\_ladder,empty\_lines,function\_call,identifiers\_line,identifiers\_repeat,telugu\_print,print\_content,print\_statement,telugu\_constant

%type<nd\_obj> telugu\_identifier\_declaring,eol,telugu\_int, telugu\_float, telugu\_arithmetic\_operator, telugu\_comparison\_operator, telugu\_assignment\_operator, telugu\_logical\_operator, telugu\_datatype, telugu\_if, telugu\_elif, telugu\_else,

telugu\_while, telugu\_identifier, telugu\_string, telugu\_open\_curly\_bracket, telugu\_closed\_curly\_bracket, telugu\_open\_square\_bracket, telugu\_closed\_square\_bracket, telugu\_open\_floor\_bracket,

telugu\_function\_name,telugu\_function\_name\_call,telugu\_closed\_floor\_bracket, telugu\_punctuation\_comma, telugu\_newline, telugu\_finish, telugu\_function, telugu\_return, telugu\_character,telugu\_import,telugu\_input,telugu\_imported\_library

%%

program:

input {int num\_children = 1; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

// Assigning children nodes

children[0] = $1.nd; // Assuming $1 represents the parse tree node for symbol1

// Assign more children if needed

// Create the parse tree node for the production rule

$$.nd = mknode(num\_children, children, "program");

head = $$.nd;}

eol:

EOL {$$.nd = mknode(NULL, NULL, "newline");}

telugu\_identifier:

TELUGU\_IDENTIFIER {printf("CHECKING FOR %s\n",$1.name);check\_declaration($1.name); printf("saw pure id2");$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_function\_name: // only for functions being declared

TELUGU\_IDENTIFIER {printf("parser saw teluguFuncNamex");

$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_function\_name\_call: // only for functions being called

TELUGU\_IDENTIFIER {printf("parser saw teluguFuncNameCall");

$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_identifier\_declaring: // only for identifiers being declared

TELUGU\_IDENTIFIER { printf("saw varDeclareid");add('V');$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_imported\_library: // only for identifiers being declared

TELUGU\_IDENTIFIER { add('L');$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_print:

TELUGU\_PRINT {add('K');$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_int:

TELUGU\_INT {$$.nd = mknode(NULL, NULL, $1.name);add('i');}

telugu\_input: // cin>> , scanf()

TELUGU\_INPUT {add('K');$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_float:

TELUGU\_FLOAT {add('f');$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_import:

TELUGU\_IMPORT {$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_constant:

TELUGU\_INT {$$.nd = mknode(NULL, NULL, $1.name);add('i');}

| TELUGU\_FLOAT {$$.nd = mknode(NULL, NULL, $1.name);add('f');}

| TELUGU\_STRING {$$.nd = mknode(NULL, NULL, $1.name);add('s');}

| TELUGU\_CHARACTER {$$.nd = mknode(NULL, NULL, $1.name);add('c');}

telugu\_arithmetic\_operator:

TELUGU\_ARITHMETIC\_OPERATOR {$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_comparison\_operator:

TELUGU\_COMPARISON\_OPERATOR {$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_assignment\_operator:

TELUGU\_ASSIGNMENT\_OPERATOR {$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_logical\_operator:

TELUGU\_LOGICAL\_OPERATOR {$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_datatype:

TELUGU\_DATATYPE {insert\_type();$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_if:

TELUGU\_IF {add('K');$$.nd = mknode(NULL, NULL, "if");}

telugu\_elif:

TELUGU\_ELIF {add('K');$$.nd = mknode(NULL, NULL, "elif");}

telugu\_else:

TELUGU\_ELSE {add('K');$$.nd = mknode(NULL, NULL, "else");}

telugu\_while:

TELUGU\_WHILE {add('K');$$.nd = mknode(NULL, NULL,"while");}

telugu\_string:

TELUGU\_STRING {add('s');$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_open\_curly\_bracket:

TELUGU\_OPEN\_CURLY\_BRACKET {$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_closed\_curly\_bracket:

TELUGU\_CLOSED\_CURLY\_BRACKET {$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_open\_square\_bracket:

TELUGU\_OPEN\_SQUARE\_BRACKET {$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_closed\_square\_bracket:

TELUGU\_CLOSED\_SQUARE\_BRACKET {$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_open\_floor\_bracket:

TELUGU\_OPEN\_FLOOR\_BRACKET {$$.nd = mknode(NULL, NULL, $1.name);scope++;} // increase scope for variables

telugu\_closed\_floor\_bracket:

TELUGU\_CLOSED\_FLOOR\_BRACKET {$$.nd = mknode(NULL, NULL, $1.name);

//here we need to remove all the variables declared in this scope

// change all of their scope to INT\_MAX

int i;

for(i=count-1; i>=0; i--) {

if(symbol\_table[i].thisscope == scope &&

strcmp(symbol\_table[i].type, "Variable")==0) {

symbol\_table[i].thisscope = INT\_MAX;

printf("\nERASING %s from symbol table as its CURRENT SCOPE is FINISHED\n", symbol\_table[i].id\_name);

}

}

scope--;

} // decrease scope for variables

telugu\_punctuation\_comma:

TELUGU\_PUNCTUATION\_COMMA {$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_newline:

TELUGU\_NEWLINE {$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_finish:

TELUGU\_FINISH {$$.nd = mknode(NULL, NULL, $1.name);

strcpy(exp\_type," ");

} // resetting exp\_type string

telugu\_function:

TELUGU\_FUNCTION {add('K');$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_return:

TELUGU\_RETURN {add('K');$$.nd = mknode(NULL, NULL, $1.name);}

telugu\_character:

TELUGU\_CHARACTER {add('c');$$.nd = mknode(NULL, NULL, $1.name);}

input: // input can be empty also

{ $$.nd = mknode(NULL, NULL, "empty"); }

| input eol {

printf("Parser found input-eol\n");

int num\_children = 2; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

$$.nd = mknode(num\_children, children, "input-eol");

}

| eol input {

printf("Parser found eol-input\n");

int num\_children = 2; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

$$.nd = mknode(num\_children, children, "eol-input");

}

| input telugu\_import telugu\_imported\_library telugu\_finish {

//add('H');

printf("Parser found input-import-lib-;\n");

int num\_children = 4; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

$$.nd = mknode(num\_children, children, "input-import-lib-;");

}

| telugu\_import telugu\_imported\_library telugu\_finish input {

//add('H');

printf("Parser found import-lib-;-input\n");

int num\_children = 4; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

$$.nd = mknode(num\_children, children, "import-lib-;-input");

}

| input bunch\_of\_statements input {

printf("Parser found input-bunch\_of\_stmts-input\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

$$.nd = mknode(num\_children, children, "input-bunch-input");

}

| input {insNumOfLabel[labelsused]=ic\_idx; sprintf(icg[ic\_idx++], "LABEL L%d:\n", labelsused++);} function\_declaration input {

//add('F');

printf("Parser found input-functionDec-input\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $3.nd;

children[2] = $4.nd;

$$.nd = mknode(num\_children, children, "input-funDec-input");

}

;

empty\_lines:

EOL

| empty\_lines EOL

exp: // empty not allowed

telugu\_int {

if(strcmp(exp\_type," ")==0) {

strcpy(exp\_type, "sankhya");

}

else if(strcmp(exp\_type, "theega")==0) {

sprintf(errors[sem\_errors], "Line %d: operation among int and string in expression not allowed\n", countn+1);

sem\_errors++;

}

printf("Parser found int\n");

int num\_children = 1; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

$$.nd = mknode(num\_children, children, "INT");

// if(firstreg == -1){

// firstreg = registerIndex++;

// sprintf(icg[ic\_idx++], "R%d = %s\n", firstreg, $1.name);

// }

// else{

// secondreg = registerIndex++;

// sprintf(icg[ic\_idx++], "R%d = %s\n", secondreg, $1.name);

// }

registers[regstackpointer++]=registerIndex;

sprintf(icg[ic\_idx++], "MOV R%d , %s\n", registerIndex++, $1.name);

}

| telugu\_float {

if(strcmp(exp\_type," ")==0) {

strcpy(exp\_type, "thelu");

}

else if(strcmp(exp\_type, "theega")==0) {

sprintf(errors[sem\_errors], "Line %d: operation among float and string in expression not allowed\n", countn+1);

sem\_errors++;

}

printf("Parser found float\n");

int num\_children = 1; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

$$.nd = mknode(num\_children, children, "FLOAT");

registers[regstackpointer++]=registerIndex;

sprintf(icg[ic\_idx++], "MOV R%d , %s\n", registerIndex++, $1.name);

}

| telugu\_character {

printf("Parser found character\n");

int num\_children = 1; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

$$.nd = mknode(num\_children, children, "CHAR");

registers[regstackpointer++]=registerIndex;

sprintf(icg[ic\_idx++], "MOV R%d , %s\n", registerIndex++, $1.name);

}

| telugu\_string {

if(strcmp(exp\_type," ")==0) {

strcpy(exp\_type, "theega");

}

else if(strcmp(exp\_type, "sankhya")==0 || strcmp(exp\_type, "thelu")==0) {

sprintf(errors[sem\_errors], "Line %d: operation among string and int/float in expression not allowed\n", countn+1);

sem\_errors++;

}

printf("Parser found string\n");

int num\_children = 1; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

$$.nd = mknode(num\_children, children, "STRING");

registers[regstackpointer++]=registerIndex;

sprintf(icg[ic\_idx++], "MOV R%d , %s\n", registerIndex++, $1.name);

}

| telugu\_identifier {

printf("Parser found identifier\n");

int num\_children = 1; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

$$.nd = mknode(num\_children, children, "ID");

registers[regstackpointer++]=registerIndex;

sprintf(icg[ic\_idx++], "MOV R%d , %s\n", registerIndex++, $1.name);

markVariableAsUsed($1.name); // optimization stage

}

| function\_call {

printf("Parser found funcCall\n");

int num\_children = 1; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

$$.nd = mknode(num\_children, children, "funcCall");

}

| telugu\_identifier telugu\_open\_square\_bracket exp telugu\_closed\_square\_bracket {

printf("Parser found id[exp]\n");

int num\_children = 4; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

$$.nd = mknode(num\_children, children, "ID[exp]");

//registers[regstackpointer++]=registerIndex;

sprintf(icg[ic\_idx++], "MOV R%d , %s+R%d\n", registerIndex-1 , $1.name,registerIndex-1);

}

| telugu\_open\_curly\_bracket exp telugu\_closed\_curly\_bracket {

printf("Parser found (exp)\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

// Assigning children nodes

children[0] = $1.nd; // Assuming $1 represents the parse tree node for symbol1

children[1] = $2.nd; // Assuming $2 represents the parse tree node for symbol2

children[2] = $3.nd;

$$.nd = mknode(num\_children, children, "(exp)");

// Free the memory allocated for the array of children

//free(children);

}

| exp {firstreg = registerIndex-1;registers[registerIndex-1]=firstreg;} telugu\_arithmetic\_operator exp

{secondreg = registerIndex-1;registers[registerIndex-1]=secondreg;} {

printf("Parser found exp-arithmeticOp-exp\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

// Assigning children nodes

children[0] = $1.nd; // Assuming $1 represents the parse tree node for symbol1

children[1] = $3.nd; // Assuming $2 represents the parse tree node for symbol2

children[2] = $4.nd;

// Assign more children if needed

// Create the parse tree node for the production rule

$$.nd = mknode(num\_children, children, "AthematicOp");

// Free the memory allocated for the array of children

//free(children);

//regstackpointer--;

if(($3.name)[0] == '+')

sprintf(icg[ic\_idx++], "ADD R%d , R%d\n", secondreg , registers[--regstackpointer]-1);

else if(($3.name)[0] == '-')

sprintf(icg[ic\_idx++], "SUB R%d , R%d\n", secondreg , registers[--regstackpointer]-1);

else if(($3.name)[0] == '\*')

sprintf(icg[ic\_idx++], "MUL R%d , R%d\n", secondreg , registers[--regstackpointer]-1);

else if(($3.name)[0] == '/')

sprintf(icg[ic\_idx++], "DIV R%d , R%d\n", secondreg , registers[--regstackpointer]-1);

else if(($3.name)[0] == '%')

sprintf(icg[ic\_idx++], "MOD R%d , R%d\n", secondreg , registers[--regstackpointer]-1);

else{

sprintf(icg[ic\_idx++], "R%d = R%d %c R%d\n", secondreg , registers[--regstackpointer]-1, ($3.name)[0],secondreg);

}

//secondreg = firstreg;

//first = registers[regstackpointer];

}

| exp {firstreg = registerIndex-1;} telugu\_logical\_operator exp {secondreg = registerIndex-1;} {

printf("Parser found exp-logicalOp-exp\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $3.nd;

children[2] = $4.nd;

$$.nd = mknode(num\_children, children, "LogicalOp");

//sprintf(icg[ic\_idx++], "R%d = R%d %s R%d\n", secondreg , firstreg, $3.name, secondreg);

if (strcmp($3.name, "mariyu") == 0) {

sprintf(icg[ic\_idx++], "AND R%d , R%d\n", secondreg , firstreg);

}

else if (strcmp($3.name, "leda") == 0) {

sprintf(icg[ic\_idx++], "OR R%d , R%d\n", secondreg , firstreg);

}

// else if (strcmp($3.name, "kaadu") == 0) {

// sprintf(icg[ic\_idx++], "NOT R%d , R%d\n", secondreg , firstreg);

// }

else if (strcmp($3.name, "pratyekam") == 0) {

sprintf(icg[ic\_idx++], "XOR R%d , R%d\n", secondreg , firstreg);

}

else{

sprintf(icg[ic\_idx++], "R%d = R%d %s R%d\n", secondreg , firstreg, $3.name,secondreg);

}

}

| exp {firstreg = registerIndex-1;} telugu\_comparison\_operator exp {secondreg = registerIndex-1;} {

printf("Parser found exp-compOp-exp\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $3.nd;

children[2] = $4.nd;

$$.nd = mknode(num\_children, children, "CompOp");

//sprintf(icg[ic\_idx++], "R%d = R%d %s R%d\n", secondreg , firstreg, $3.name, secondreg);

if (strcmp($3.name, "chinnadi") == 0) {

sprintf(icg[ic\_idx++], "LT R%d , R%d\n", secondreg , firstreg);

}

else if (strcmp($3.name, "peddadi") == 0) {

sprintf(icg[ic\_idx++], "GT R%d , R%d\n", secondreg , firstreg);

}

// else if (strcmp($3.name, "kaadu") == 0) {

// sprintf(icg[ic\_idx++], "NOT R%d , R%d\n", secondreg , firstreg);

// }

else if (strcmp($3.name, "peddadiLedaSamanam") == 0) {

sprintf(icg[ic\_idx++], "GE R%d , R%d\n", secondreg , firstreg);

}

else if (strcmp($3.name, "chinnadiLedaSamanam") == 0) {

sprintf(icg[ic\_idx++], "LE R%d , R%d\n", secondreg , firstreg);

}

else if (strcmp($3.name, "samanam") == 0) {

sprintf(icg[ic\_idx++], "EQ R%d , R%d\n", secondreg , firstreg);

}

else if (strcmp($3.name, "bhinnam") == 0) {

sprintf(icg[ic\_idx++], "NE R%d , R%d\n", secondreg , firstreg);

}

else{

sprintf(icg[ic\_idx++], "R%d = R%d %s R%d\n", secondreg , firstreg, $3.name,secondreg);

}

}

| telugu\_identifier telugu\_open\_square\_bracket exp {registers[regstackpointer++]=registerIndex;

sprintf(icg[ic\_idx++], "MOV R%d , %s + R%d\n", registerIndex-1 , $1.name, registerIndex-1);} telugu\_closed\_square\_bracket {

printf("Parser found id[exp]\n");

int num\_children = 4; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $5.nd;

$$.nd = mknode(num\_children, children, "id[exp]");

//sprintf(icg[ic\_idx++], "MOV R%d , %s + R%d\n", firstreg , $1.name, firstreg);

}

;

bunch\_of\_statements: //can be empty

{ $$.nd = mknode(NULL, NULL, "empty"); }

| eol bunch\_of\_statements {

printf("Parser found EOL-bunch\n");

int num\_children = 2; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

$$.nd = mknode(num\_children, children, "eol-bunch");

}

| bunch\_of\_statements eol {

printf("Parser found bunch-EOL\n");

int num\_children = 2; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

$$.nd = mknode(num\_children, children, "bunch-eol");

}

| bunch\_of\_statements if\_else\_ladder {

insNumOfLabel[labelsused]=ic\_idx;

sprintf(icg[ic\_idx++], "LABEL L%d:\n", labelsused++);

//lastjumps[lastjumpstackpointer++] = label[stackpointer-2];

int index = ic\_idx - 1;

int count = laddercounts[laddercountstackpointer-1]; // Number of iterations

for (int i = index; i >= 0 && count > 0; i--) {

printf("icg[%d] = %s\n", i, icg[i]);

if (strncmp(icg[i], "JUMP ", 5) == 0) { // Check if the prefix matches "JUMP "

printf("...................\n");

char jump\_str[20]; // Assuming the number won't exceed 20 digits

sprintf(jump\_str, "%d", labelsused-1); // Convert number to string

snprintf(icg[i], 20, "JUMPx L%s\n", jump\_str); // Set icg[i] to "JUMP" followed by the number

count--;

}

}

lastjumpstackpointer--; // forgetting the current ifelseLadder's lastjump and counts

laddercountstackpointer--;

} bunch\_of\_statements {

} {

printf("Parser found bunch\_of\_statement if\_else\_ladder bunch\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $4.nd;

$$.nd = mknode(num\_children, children, "bunch-IfElse-bunch");

}

| bunch\_of\_statements telugu\_input telugu\_open\_curly\_bracket telugu\_identifier telugu\_closed\_curly\_bracket telugu\_finish bunch\_of\_statements {

printf("Parser found bunch\_of\_statement-inputscan-bunch\n");

int num\_children = 7; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

children[4] = $5.nd;

children[5] = $6.nd;

children[6] = $7.nd;

$$.nd = mknode(num\_children, children, "bunch-inputScan-bunch");

}

| bunch\_of\_statements while\_loop bunch\_of\_statements {

printf("Parser found bunch\_of\_statement while\_loop bunch\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

$$.nd = mknode(num\_children, children, "bunch-while-bunch");

}

| bunch\_of\_statements print\_statement telugu\_finish bunch\_of\_statements {

printf("Parser found bunch-printStmt-finish\n");

int num\_children = 4; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

$$.nd = mknode(num\_children, children, "bunch-printStmt-;-bunch");

}

| bunch\_of\_statements variable\_declaration telugu\_finish bunch\_of\_statements {

printf("Parser found bunch-varDeclare-finish\n");

int num\_children = 4; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

$$.nd = mknode(num\_children, children, "bunch-varDeclare-;-bunch");

}

| bunch\_of\_statements telugu\_open\_floor\_bracket bunch\_of\_statements telugu\_closed\_floor\_bracket bunch\_of\_statements {

printf("parser found bunch {bunch}\n");

int num\_children = 5; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

children[4] = $5.nd;

$$.nd = mknode(num\_children, children, "bunch-{bunch}-bunch");

}

| bunch\_of\_statements function\_call telugu\_finish bunch\_of\_statements {

printf("Parser found bunch-functionCall-;\n");

int num\_children = 4; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

$$.nd = mknode(num\_children, children, "bunch-functionCall-;-bunch");

}

| bunch\_of\_statements equation telugu\_finish bunch\_of\_statements {

printf("Parser found bunch-equation-finish\n");

int num\_children = 4; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

$$.nd = mknode(num\_children, children, "bunch-equation-;-bunch");

}

| error telugu\_finish {

printf("PARSER ERROR: syntax error \n");

int num\_children = 0; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

$$.nd = mknode(num\_children, children, "error-;");

}

condition: // for if\_statement and while loop, empty not allowed

exp {

printf("Parser found exp as condition\n");

int num\_children = 1; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

$$.nd = mknode(num\_children, children, "condition");

}

| exp {firstreg = registerIndex-1;} telugu\_comparison\_operator exp {secondreg = registerIndex-1;} {

printf("Parser found exp-compareOp-exp\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $3.nd;

children[2] = $4.nd;

$$.nd = mknode(num\_children, children, "condition");

//sprintf(icg[ic\_idx++], "R%d = R%d %s R%d\n", secondreg , firstreg, $3.name, secondreg);

if (strcmp($3.name, "chinnadi") == 0) {

sprintf(icg[ic\_idx++], "LT R%d R%d R%d\n", registerIndex++ , firstreg, secondreg);

}

else if (strcmp($3.name, "peddadi") == 0) {

sprintf(icg[ic\_idx++], "GT R%d R%d R%d\n", registerIndex++ , firstreg, secondreg);

}

else if (strcmp($3.name, "chinnadiLedaSamanam") == 0) {

sprintf(icg[ic\_idx++], "LTE R%d R%d R%d\n", registerIndex++ , firstreg, secondreg);

}

else if (strcmp($3.name, "peddadiLedaSamanam") == 0) {

sprintf(icg[ic\_idx++], "GTE R%d R%d R%d\n", registerIndex++ , firstreg, secondreg);

}

else if (strcmp($3.name, "samanam") == 0) {

sprintf(icg[ic\_idx++], "EQ R%d R%d R%d\n", registerIndex++ , firstreg, secondreg);

}

else{

sprintf(icg[ic\_idx++], "R%d = R%d %s R%d\n", secondreg , firstreg, $3.name, secondreg);

registerIndex++; // is this needed?

}

}

| exp {firstreg = registerIndex-1;} telugu\_logical\_operator exp {secondreg = registerIndex-1;} {

printf("Parser found exp-logicalOp-exp\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $3.nd;

children[2] = $4.nd;

$$.nd = mknode(num\_children, children, "condition");

sprintf(icg[ic\_idx++], "R%d = R%d %s R%d\n", secondreg , firstreg, $3.name, secondreg);

}

if\_statement:

telugu\_if telugu\_open\_curly\_bracket condition {

sprintf(icg[ic\_idx++], "if NOT (R%d) GOTO L%d\n",registerIndex-1,labelsused);isleader[insNumOfLabel[labelsused]]=1;isleader[ic\_idx]=1; label[stackpointer++]=labelsused++;} telugu\_closed\_curly\_bracket telugu\_open\_floor\_bracket bunch\_of\_statements {sprintf(icg[ic\_idx++], "JUMP L%d\n",label[stackpointer-1]);} telugu\_closed\_floor\_bracket {

printf("Parser found if(cond){bunch}\n");

int num\_children = 7; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $5.nd;

children[4] = $6.nd;

children[5] = $7.nd;

children[6] = $9.nd;

$$.nd = mknode(num\_children, children, "if(cond){bunch}");

insNumOfLabel[label[stackpointer-1]]=ic\_idx;

sprintf(icg[ic\_idx++], "LABEL L%d:\n", label[--stackpointer]);

laddercounts[laddercountstackpointer++]=1;

}

elif\_repeat: //can be empty

{ $$.nd = mknode(NULL, NULL, "empty"); }

| eol elif\_repeat {

printf("Parser found eol elif\_repeat\n");

int num\_children = 2; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

$$.nd = mknode(num\_children, children, "EOL-elifrepeat");

}

| elif\_repeat telugu\_elif telugu\_open\_curly\_bracket condition

{sprintf(icg[ic\_idx++], "if NOT (R%d) GOTO L%d\n",registerIndex-1,labelsused);isleader[insNumOfLabel[labelsused]]=1;isleader[ic\_idx]=1;label[stackpointer++]=labelsused++;}

telugu\_closed\_curly\_bracket telugu\_open\_floor\_bracket bunch\_of\_statements

{sprintf(icg[ic\_idx++], "JUMP L%d\n",label[stackpointer-1]);} telugu\_closed\_floor\_bracket

{insNumOfLabel[label[stackpointer-1]]=ic\_idx; sprintf(icg[ic\_idx++], "LABEL L%d:\n", label[--stackpointer]);} elif\_repeat {

printf("Parser found elif(cond){bunch}\n");

int num\_children = 9; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

children[4] = $6.nd;

children[5] = $7.nd;

children[6] = $8.nd;

children[7] = $10.nd;

children[8] = $12.nd;

$$.nd = mknode(num\_children, children, "elif(cond){bunch}");

laddercounts[laddercountstackpointer-1]++;

}

else\_statement: //can be empty

{ $$.nd = mknode(NULL, NULL, "empty"); }

| eol else\_statement {

printf("Parser found EOL-else\n");

int num\_children = 2; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

$$.nd = mknode(num\_children, children, "EOL-else");

}

| telugu\_else telugu\_open\_floor\_bracket bunch\_of\_statements telugu\_closed\_floor\_bracket {

printf("Parser found else{bunch}\n");

int num\_children = 4; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

$$.nd = mknode(num\_children, children, "else{bunch}");

}

if\_else\_ladder:

if\_statement elif\_repeat

{

lastjumps[lastjumpstackpointer++] = label[stackpointer-1];

}

else\_statement {

printf("Parser found ifElseLadder\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $4.nd;

$$.nd = mknode(num\_children, children, "ifElseLadder");

// lastjumpstackpointer--; // forgetting the current ifelseLadder's lastjump and counts

// laddercountstackpointer--;

}

| if\_statement elif\_repeat

{

lastjumps[lastjumpstackpointer++] = label[stackpointer-1];

// int index = ic\_idx - 1;

// int count = laddercounts[laddercountstackpointer-1]; // Number of iterations

// for (int i = index; i >= 0 && count > 0; i--) {

// if (strncmp(icg[i], "JUMP ", 5) == 0) { // Check if the prefix matches "JUMP "

// char jump\_str[20]; // Assuming the number won't exceed 20 digits

// sprintf(jump\_str, "%d", lastjumps[lastjumpstackpointer-1]); // Convert number to string

// snprintf(icg[i], 20, "JUMPx L%s\n", jump\_str); // Set icg[i] to "JUMP" followed by the number

// count--;

// }

// }

}

{ // without the else statement

printf("Parser found ifElseLadder\n");

int num\_children = 2; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

$$.nd = mknode(num\_children, children, "ifElseLadder");

}

while\_loop:

telugu\_while telugu\_open\_curly\_bracket condition { looplabel[looplabelstackpointer++] = labelsused; insNumOfLabel[labelsused]=ic\_idx; sprintf(icg[ic\_idx++], "LABEL L%d:\n", labelsused++);

gotolabel[gotolabelstackpointer++]=labelsused; sprintf(icg[ic\_idx++], "if NOT (R%d) GOTO L%d\n",registerIndex-1,labelsused++);isleader[insNumOfLabel[labelsused-1]]=1;isleader[ic\_idx]=1;

} telugu\_closed\_curly\_bracket telugu\_open\_floor\_bracket bunch\_of\_statements { sprintf(icg[ic\_idx++], "JUMPtoLOOP L%d\n",looplabel[--looplabelstackpointer]);

} telugu\_closed\_floor\_bracket {insNumOfLabel[gotolabel[gotolabelstackpointer-1]]=ic\_idx; sprintf(icg[ic\_idx++], "LABEL L%d:\n", gotolabel[--gotolabelstackpointer]);} {

printf("Parser found while(cond){bunch}\n");

int num\_children = 7; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $5.nd;

children[4] = $6.nd;

children[5] = $7.nd;

children[6] = $9.nd;

$$.nd = mknode(num\_children, children, "while(cond){bunch}");

}

variable\_declaration:

telugu\_datatype telugu\_identifier\_declaring telugu\_assignment\_operator {rangestart = ic\_idx;} exp {

//add('V'); // this is taking ';' as a variable

printf("Parser found datatypeId=exp\n");

int num\_children = 4; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $5.nd;

$$.nd = mknode(num\_children, children, "datatypeId=exp");

if(strcmp(exp\_type,$1.name)!=0 && strcmp(exp\_type, " ")!=0){

sprintf("$1name=%s and exp\_type=%s\n", $1.name,exp\_type);

sprintf(errors[sem\_errors], "Line %d: Data type casting not allowed in declaration\n", countn);

sem\_errors++;

}

rangeend = ic\_idx;

printf("QQQQQQQQQQQQQQQQQQQQQ rangestart=%d rangeend=%d\n",rangestart,rangeend);

int idIndexinSymbolTable = findIdentifierIndex($2.name);

symbol\_table[idIndexinSymbolTable].range[symbol\_table[idIndexinSymbolTable].range\_count][0] = rangestart;

symbol\_table[idIndexinSymbolTable].range[symbol\_table[idIndexinSymbolTable].range\_count++][1] = rangeend; //stack counter is increased

sprintf(icg[ic\_idx++], "MOV %s , R%d\n", $2.name, registerIndex-1);

}

| telugu\_datatype telugu\_identifier\_declaring {

printf("Parser found datatype Id\n");

int num\_children = 2; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

$$.nd = mknode(num\_children, children, "datatypeId");

//// not needed here

// rangestart = ic\_idx;

// rangeend = ic\_idx;

// int idIndexinSymbolTable = findIdentifierIndex($2.name);

// symbol\_table[idIndexinSymbolTable].range[symbol\_table[idIndexinSymbolTable].range\_count][0] = rangestart;

// symbol\_table[idIndexinSymbolTable].range[symbol\_table[idIndexinSymbolTable].range\_count++][1] = rangeend; //stack counter is increased

}

| telugu\_datatype telugu\_identifier\_declaring telugu\_open\_square\_bracket exp telugu\_closed\_square\_bracket {

printf("Parser found datatype Id\n");

int num\_children = 5; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

children[4] = $5.nd;

$$.nd = mknode(num\_children, children, "datatype Id[exp]");

}

parameters\_repeat: // can be empty 0 or more occurences

{ $$.nd = mknode(NULL, NULL, "empty"); }

| parameters\_repeat telugu\_datatype telugu\_identifier\_declaring telugu\_punctuation\_comma {

printf("Parser found paramRepDatatypeIdComma\n");

curr\_num\_params++;

int num\_children = 4; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

$$.nd = mknode(num\_children, children, "paramRepDatatypeIdComma");

}

parameters\_line: // can be empty

{ $$.nd = mknode(NULL, NULL, "empty"); }

| {scope++;} parameters\_repeat telugu\_datatype telugu\_identifier\_declaring {scope--;} {

printf("Parser found parameters\_line\n");

curr\_num\_params++;

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $2.nd;

children[1] = $3.nd;

children[2] = $4.nd;

$$.nd = mknode(num\_children, children, "paramLine");

}

identifiers\_repeat: // abc,x,y,p can be empty

{ $$.nd = mknode(NULL, NULL, "empty"); }

| telugu\_identifier {

curr\_num\_args++;

printf("Parser found lastparam\n");

int num\_children = 1; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

$$.nd = mknode(num\_children, children, "paramEnd");

sprintf(icg[ic\_idx++], "PARAM %s\n", $1.name);

}

| telugu\_constant {

curr\_num\_args++;

printf("Parser found lastparam\n");

int num\_children = 1; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

$$.nd = mknode(num\_children, children, "paramEnd");

sprintf(icg[ic\_idx++], "PARAM %s\n", $1.name);

}

| exp {

curr\_num\_args++;

printf("Parser found lastparam\n");

int num\_children = 1; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

$$.nd = mknode(num\_children, children, "paramEnd");

sprintf(icg[ic\_idx++], "PARAM %s\n", $1.name);

}

| identifiers\_repeat telugu\_punctuation\_comma identifiers\_repeat {

curr\_num\_args++;

printf("Parser found id-comma-prep\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

$$.nd = mknode(num\_children, children, "paramRep");

sprintf(icg[ic\_idx++], "PARAM %s\n", $1.name);

}

identifiers\_line: // for function call,can be empty

identifiers\_repeat {

printf("Parser found idLine\n");

int num\_children = 1; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

$$.nd = mknode(num\_children, children, "idline");

}

equation:

telugu\_identifier telugu\_assignment\_operator { strcpy(exp\_type," "); } {rangestart = ic\_idx;} exp {

printf("Parser found equation\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $5.nd;

$$.nd = mknode(num\_children, children, "id=exp");

//check if identifier type and exp\_type mismatch -> if yes then typecast is happening

printf("type of identifier: %s XXXXXXXXXXXXXXXXX exp\_type=%s\n\n", get\_type($1.name),exp\_type);

if(strcmp(get\_datatype($1.name) , exp\_type) && strcmp(exp\_type, " ")){

sprintf(errors[sem\_errors], "Line %d: Data type casting not allowed in equation\n", countn);

sem\_errors++;

}

// a = exp ---> t1=exp, a=t1

rangeend = ic\_idx;

printf("ZZZZZZZZZZ rangestart=%d rangeend=%d\n",rangestart,rangeend);

int idIndexinSymbolTable = findIdentifierIndex($1.name);

symbol\_table[idIndexinSymbolTable].range[symbol\_table[idIndexinSymbolTable].range\_count][0] = rangestart;

symbol\_table[idIndexinSymbolTable].range[symbol\_table[idIndexinSymbolTable].range\_count++][1] = rangeend; //stack counter is increased

sprintf(icg[ic\_idx++], "%s = R%d\n", $1.name, registerIndex-1);

}

| telugu\_identifier telugu\_open\_square\_bracket exp {thirdreg = registerIndex-1;} telugu\_closed\_square\_bracket telugu\_assignment\_operator exp {

printf("Parser found id[exp]=exp\n");

int num\_children = 6; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $5.nd;

children[4] = $6.nd;

children[5] = $7.nd;

$$.nd = mknode(num\_children, children, "id[exp]=exp");

sprintf(icg[ic\_idx++], "MOV %s+R%d , R%d\n", $1.name,thirdreg ,registerIndex-1);

}

function\_content: // can be empty also, return not needed

function\_content eol {

printf("Parser found funContentEOL\n");

int num\_children = 2; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

$$.nd = mknode(num\_children, children, "funContentEOL");

}

| eol function\_content {

printf("Parser found EOL-funContent\n");

int num\_children = 2; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

$$.nd = mknode(num\_children, children, "EOL-funContent");

}

| bunch\_of\_statements function\_content bunch\_of\_statements {

printf("Parser found bunch\_function\_content\_bunch\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

$$.nd = mknode(num\_children, children, "bunch-content-bunch");

}

| bunch\_of\_statements {

printf("Parser found bunch\_function\_content\_bunch\n");

int num\_children = 1; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

$$.nd = mknode(num\_children, children, "bunch-content-bunch");

}

| bunch\_of\_statements telugu\_return telugu\_finish bunch\_of\_statements {

printf("Parser found bunchReturnFinish\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

$$.nd = mknode(num\_children, children, "bunchReturnFinish");

}

| bunch\_of\_statements telugu\_return exp telugu\_finish bunch\_of\_statements {

printf("Parser found bunchReturnExpFinish\n");

int num\_children = 5; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

children[4] = $5.nd;

$$.nd = mknode(num\_children, children, "bunchReturnExpFinish");

}

| bunch\_of\_statements function\_call telugu\_finish bunch\_of\_statements {

printf("Parser found bunchReturnExpFinish\n");

int num\_children = 4; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

$$.nd = mknode(num\_children, children, "bunchFunCallFinish");

}

print\_content: // can be empty also

| print\_content eol {

printf("Parser found print\_contentEOL\n");

int num\_children = 2; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

$$.nd = mknode(num\_children, children, "print\_content-EOL");

}

| eol print\_content { // take care of infinite loop

printf("Parser found EOL-print\_content\n");

int num\_children = 2; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

$$.nd = mknode(num\_children, children, "EOL-printContent");

}

| print\_content telugu\_string {

printf("Parser found print\_content-String\n");

int num\_children = 2; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

$$.nd = mknode(num\_children, children, "printContent-String");

}

| print\_content exp {

printf("Parser found print\_content-exp\n");

int num\_children = 2; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

$$.nd = mknode(num\_children, children, "printContent-exp");

}

| print\_content telugu\_punctuation\_comma telugu\_string {

printf("Parser found print\_content-comma-String\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

$$.nd = mknode(num\_children, children, "print\_content-comma-String");

}

| print\_content telugu\_punctuation\_comma exp {

printf("Parser found print\_content-comma-exp\n");

int num\_children = 3; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

$$.nd = mknode(num\_children, children, "print\_content-comma-exp");

}

print\_statement:

telugu\_print telugu\_open\_curly\_bracket print\_content telugu\_closed\_curly\_bracket {

printf("Parser found printStatement\n");

int num\_children = 4; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $2.nd;

children[2] = $3.nd;

children[3] = $4.nd;

$$.nd = mknode(num\_children, children, "printStatement");

}

function\_declaration:

telugu\_function {oldscope=scope;scope=0;} telugu\_function\_name {add('F');scope=oldscope;} telugu\_open\_curly\_bracket parameters\_line telugu\_closed\_curly\_bracket telugu\_open\_floor\_bracket function\_content telugu\_closed\_floor\_bracket {

printf("Parser found equation\n");

int num\_children = 8; // Number of childrenfunction\_call

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $3.nd;

children[2] = $5.nd;

children[3] = $6.nd;

children[4] = $7.nd;

children[5] = $8.nd;

children[6] = $9.nd;

children[7] = $10.nd;

$$.nd = mknode(num\_children, children, "func-id-(param){content}");

symbol\_table[count-curr\_num\_params-3].num\_params= curr\_num\_params;

if(symbol\_table[count-curr\_num\_params-3].num\_params>=0){

printf("XXXX changed num\_params of %s to %d\n",symbol\_table[count-curr\_num\_params-3].id\_name,symbol\_table[count-curr\_num\_params-3].num\_params);

curr\_num\_params=0;

}

}

function\_call:

telugu\_identifier { check\_declaration($1.name); } telugu\_open\_curly\_bracket identifiers\_line telugu\_closed\_curly\_bracket {

printf("Parser found id(idLine)Finish\n");

int num\_children = 4; // Number of children

struct node \*\*children = (struct node \*\*)malloc(num\_children \* sizeof(struct node \*));

children[0] = $1.nd;

children[1] = $3.nd;

children[2] = $4.nd;

children[3] = $5.nd;

$$.nd = mknode(num\_children, children, "id(idLine)Finish");

for(int i=0;i<count;i++){

if(strcmp(symbol\_table[i].id\_name,$1.name)==0){ // found the corresponding function

if(symbol\_table[i].num\_params==-1){

printf("ERROR: %s is not a function\n",$1.name);

sprintf(errors[sem\_errors], "Line %d: %s is not a function\n", countn+1,$1.name);

sem\_errors++;

break;

}

// if(symbol\_table[i].num\_params!=curr\_num\_args){

// printf("ERROR: Number of parameters do not match\n");

// sprintf(errors[sem\_errors], "Line %d: need %d arguments but found %d args\n", countn+1,symbol\_table[i].num\_params,curr\_num\_args);

// sem\_errors++;

// break;

// }

}

}

curr\_num\_args=0;

sprintf(icg[ic\_idx++], "CALL %s\n", $1.name);

}

%%

int main(){

for(int i=0;i<10000;i++){

laddercounts[i]=0;

isleader[i]=0;

insNumOfLabel[i]=-1;

}

isleader[0]=1;

strcpy(exp\_type," ");

printf("\n\n");

printf("\t\t\t\t\t\t\t\t PHASE 1: LEXICAL ANALYSIS \n\n");

for(int i=0;i<10000;i++){

symbol\_table[i].used = 0;

symbol\_table[i].range\_count = 0;

// for(int j=0;j<10000;j++){

// symbol\_table[i].range[j][0]=-1;

// symbol\_table[i].range[j][1]=-1; // dummy values

// }

}

yyparse();

printf("\nSYMBOL DATATYPE TYPE LineNUMBER SCOPE numParams\n");

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n\n");

int i=0;

// for(i=0; i<count; i++) {

// printf("%s\t%s\t%s\t%d\t%d\t%d\n", symbol\_table[i].id\_name, symbol\_table[i].data\_type, symbol\_table[i].type, symbol\_table[i].line\_no,symbol\_table[i].thisscope,symbol\_table[i].num\_params);

// }

for (i = 0; i < count; i++) {

printf("%s\t%s\t%s\t%d\t%d\t%d\t%s\n", symbol\_table[i].id\_name, symbol\_table[i].data\_type, symbol\_table[i].type, symbol\_table[i].line\_no, symbol\_table[i].thisscope, symbol\_table[i].num\_params, symbol\_table[i].used ? "Used" : "unUsed");

}

printf("\n\n");

printf("\t\t\t\t\t\t\t\t PHASE 2: SYNTAX ANALYSIS \n\n");

printtree(head);

printf("\n\n\n\n");

printf("\t\t\t\t\t\t\t\t PHASE 3: SEMANTIC ANALYSIS \n\n");

if(sem\_errors>0) {

printf("Semantic analysis completed with %d errors\n", sem\_errors);

for(int i=0; i<sem\_errors; i++){

printf("\t - %s", errors[i]);

}

} else {

printf("Semantic analysis completed with no errors");

}

printf("\n\n");

printf("\t\t\t\t\t\t\t PHASE 4: INTERMEDIATE CODE GENERATION \n\n");

for(int i=0; i<ic\_idx; i++){

if(icg[i][0]=='L' && icg[i][0]=='A'){

printf("\n");

}

printf("%d %s", i,icg[i]);

}

printf("\n\n");

// Assuming icg[] contains the strings "LABEL L15", "LABEL L20", etc.

for (int i = 0; i < ic\_idx; i++) {

// Check if the string starts with "LABEL L"

if (strncmp(icg[i], "LABEL L", 7) == 0) {

// Extract the label number from the string

int labelNumber = atoi(icg[i] + 7); // Skip "LABEL L" and convert the rest to integer

// Use the label number to index into insNumOfLabel array

insNumOfLabel[labelNumber] = i;

}

}

for(int i=0;i<ic\_idx;i++){

if (strncmp(icg[i], "if NOT (", 8) == 0) {

char \*ptr = strstr(icg[i], "L"); // Find the first occurrence of "L" in the string

int labelNumber = atoi(ptr + 1); // Convert the substring after "L" to integer

//printf("Extracted label number: %d\n", labelNumber);

isleader[insNumOfLabel[labelNumber]] = 1;

if(i+1<10000)

isleader[i+1]=1;

}

}

printf("\t\t\tBLOCKS:\n\n");

int prev=-1,blockcount=0;

for(int i=0;i<10000;i++){

// if(insNumOfLabel[i]!=-1){

// printf("Label %d is at %d\n",i,insNumOfLabel[i]);

// }

if(isleader[i]){

if(prev!=-1)

printf("block %d: %d to %d\n",blockcount++,prev,i-1);

//printf("Leader %d\n",i);

prev=i;

}

}

printf("\n\n");

// Iterate over the symbol table to print ranges for unused variables

for (int i = 0; i < count; i++) {

if (symbol\_table[i].used <= 0 && strcmp(symbol\_table[i].type, "Variable") == 0) {

printf("Variable %s declared but not used\n", symbol\_table[i].id\_name);

printf("Ranges for %s:\n", symbol\_table[i].id\_name);

for (int j = 0; j < symbol\_table[i].range\_count; j++) {

printf("[%d, %d]\n", symbol\_table[i].range[j][0], symbol\_table[i].range[j][1]);

uselessranges[uselessrangescount][0] = symbol\_table[i].range[j][0];

uselessranges[uselessrangescount++][1] = symbol\_table[i].range[j][1];

}

printf("\n");

}

}

for(i=0;i<count;i++) {

free(symbol\_table[i].id\_name); // symbol is needed, so dont free yet

free(symbol\_table[i].type);

}

//printf("done");

// Sort uselessranges

sortRanges(uselessranges, uselessrangescount);

int uselessrangesidx = 0;

printf(" uselessrangescount=%d\n", uselessrangescount);

printf("\t\t\t\t\t\t\t PHASE 5: OPTIMIZATION \n\n");

for(int i=0; i<ic\_idx; i++){

if (uselessrangesidx < uselessrangescount && i == uselessranges[uselessrangesidx][0]) {

uselessrangesidx++;

i=uselessranges[uselessrangesidx-1][1];

//printf("skipping from %d to %d\n", uselessranges[uselessrangesidx-1][0], uselessranges[uselessrangesidx-1][1]);

continue;

}

if(icg[i][0]=='L' && icg[i][0]=='A'){

printf("\n");

}

printf("%d %s",i, icg[i]);

}

printf("\n\n");

return 0;

}

int yyerror(char \*s){

printf("PARSER ERROR: %s\n",s);

//return 0;

}

struct node\* mknode(int num\_children, struct node \*\*children, char \*token) {

struct node \*newnode = (struct node \*)malloc(sizeof(struct node));

newnode->num\_children = num\_children;

newnode->children = children;

newnode->token = strdup(token);

return newnode;

}

void printtree(struct node\* tree) {

printf("\n\n Inorder traversal of the Parse Tree: \n\n");

printInorder(tree);

printf("\n\n");

}

void printInorder(struct node \*tree) {

if (tree) {

printf("%s, ", tree->token);

for (int i = 0; i < tree->num\_children; i++) {

printInorder(tree->children[i]);

}

}

}

///////////////////////////////////// SYMBOL TABLE & SEMANTIC ANALYSIS PART

int search(char \*type) {

int i;

for(i=count-1; i>=0; i--) {

if(strcmp(symbol\_table[i].id\_name, type)==0) {

return symbol\_table[i].thisscope;

break;

}

}

return 0;

}

void check\_declaration(char \*c) {

q = search(c);

// if(!q) {

// sprintf(errors[sem\_errors], "Line %d: Variable \"%s\" not declared before usage!\n", countn+1, c);

// sem\_errors++;

// }

}

char \*get\_type(char \*var){

for(int i=0; i<count; i++) {

// Handle case of use before declaration

if(!strcmp(symbol\_table[i].id\_name, var)) {

return symbol\_table[i].type;

}

}

}

char \*get\_datatype(char \*var){

for(int i=0; i<count; i++) {

// Handle case of use before declaration

if(!strcmp(symbol\_table[i].id\_name, var)) {

return symbol\_table[i].data\_type;

}

}

}

void add(char c) {

if(c == 'V'){ // variable

for(int i=0; i<reserved\_count; i++){

if(!strcmp(reserved[i], strdup(yy\_text))){

sprintf(errors[sem\_errors], "Line %d: Variable name \"%s\" is a reserved keyword!\n", countn+1, yy\_text);

sem\_errors++;

return;

}

}

}

q=search(yy\_text);

if(!q) { // insert into symbol table only if not already present

if(c == 'H') { //header

symbol\_table[count].id\_name=strdup(yy\_text);

symbol\_table[count].data\_type=strdup(type);

symbol\_table[count].line\_no=countn;

symbol\_table[count].type=strdup("Header");

symbol\_table[count].thisscope=scope;

symbol\_table[count].num\_params=-1;

count++;

}

else if(c == 'K') { //keyword

symbol\_table[count].id\_name=strdup(yy\_text);

symbol\_table[count].data\_type=strdup("N/A");

symbol\_table[count].line\_no=countn;

symbol\_table[count].type=strdup("Keyword\t");

symbol\_table[count].thisscope=scope;

symbol\_table[count].num\_params=-1;

count++;

}

else if(c == 'V') { //variable

printf("yytext: %s\n", yy\_text);

symbol\_table[count].id\_name=strdup(yy\_text);

symbol\_table[count].data\_type=strdup(type);

symbol\_table[count].line\_no=countn;

symbol\_table[count].type=strdup("Variable");

symbol\_table[count].thisscope=scope;

symbol\_table[count].num\_params=-1;

count++;

}

else if(c == 'C') { //constant sankhya

symbol\_table[count].id\_name=strdup(yy\_text);

symbol\_table[count].data\_type=strdup("CONST");

symbol\_table[count].line\_no=countn;

symbol\_table[count].type=strdup("constantx");

symbol\_table[count].thisscope=scope;

symbol\_table[count].num\_params=-1;

count++;

}

else if(c == 'i') { //constant sankhya

symbol\_table[count].id\_name=strdup(yy\_text);

symbol\_table[count].data\_type=strdup("CONST");

symbol\_table[count].line\_no=countn;

symbol\_table[count].type=strdup("sankhya");

symbol\_table[count].thisscope=scope;

symbol\_table[count].num\_params=-1;

count++;

}

else if(c == 'f') { //constant float thelu

symbol\_table[count].id\_name=strdup(yy\_text);

symbol\_table[count].data\_type=strdup("CONST");

symbol\_table[count].line\_no=countn;

symbol\_table[count].type=strdup("thelu");

symbol\_table[count].thisscope=scope;

symbol\_table[count].num\_params=-1;

count++;

}

else if(c == 'c') { //constant character aksharam

symbol\_table[count].id\_name=strdup(yy\_text);

symbol\_table[count].data\_type=strdup("CONST");

symbol\_table[count].line\_no=countn;

symbol\_table[count].type=strdup("aksharam");

symbol\_table[count].thisscope=scope;

symbol\_table[count].num\_params=-1;

count++;

}

else if(c == 's') { //constant string theega

symbol\_table[count].id\_name=strdup(yy\_text);

symbol\_table[count].data\_type=strdup("CONST");

symbol\_table[count].line\_no=countn;

symbol\_table[count].type=strdup("theega");

symbol\_table[count].thisscope=scope;

symbol\_table[count].num\_params=-1;

count++;

}

else if(c == 'F') {

symbol\_table[count].id\_name=strdup(yy\_text);

symbol\_table[count].data\_type=strdup(type);

symbol\_table[count].line\_no=countn;

symbol\_table[count].type=strdup("Function");

symbol\_table[count].thisscope=scope;

printf("\nSETTING %s's params to %d\n", symbol\_table[count-curr\_num\_params].id\_name, curr\_num\_params);

symbol\_table[count-curr\_num\_params].num\_params=curr\_num\_params;

curr\_num\_params=0;

count++;

}

else if(c == 'L') {

symbol\_table[count].id\_name=strdup(yy\_text);

symbol\_table[count].data\_type=strdup(type);

symbol\_table[count].line\_no=countn;

symbol\_table[count].type=strdup("Library");

symbol\_table[count].thisscope=scope;

symbol\_table[count].num\_params=0;

count++;

}

}

else if(c == 'V' && q) {

if(q != INT\_MAX){

sprintf(errors[sem\_errors], "Line %d: Multiple declarations of \"%s\" not allowed!\n", countn+1, yy\_text);

sem\_errors++;

}

else{ // its scope is already destroyed, now it can be redeclared again into the symbol table with current scope

// search again for that symbol table value

int i;

for(i=count-1; i>=0; i--) {

if(strcmp(symbol\_table[i].id\_name, type)==0) {

symbol\_table[i].thisscope = scope;

symbol\_table[count].line\_no=countn;

symbol\_table[count].num\_params=0;

printf("\nReinserted %s because its previous scope is finished\n", type);

break;

}

}

}

}

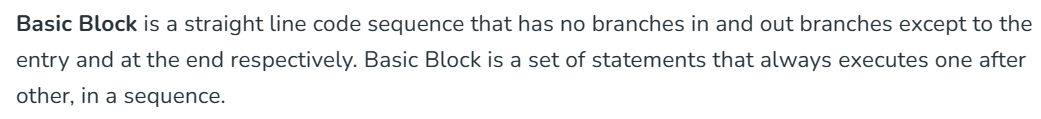
}

void insert\_type() {

strcpy(type, yy\_text);

}

1. **BASIC BLOCKS**





For a given label, we should remember which instruction it is mentioned at.



Basic Blocks are bunch of instructions from one leader to the next leader.

A Leader instruction is defined as:

1. First instruction is a leader by default
2. Address of Conditional and Unconditional GOTO are leaders

LABEL L32: // they have label declaration to which other GOTO statements are destined to

…….

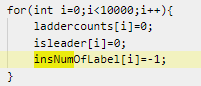
……..

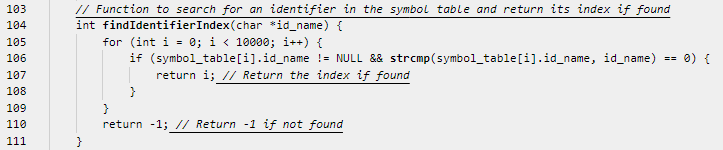
If NOT (condition) GOTO L32

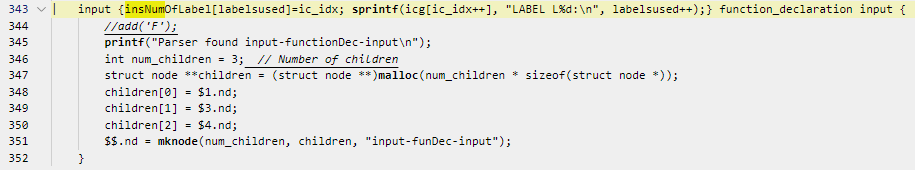
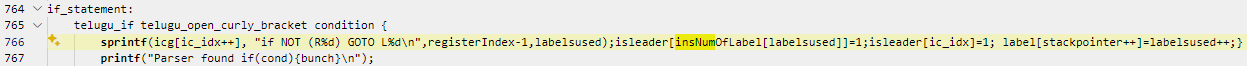
…….

JUMP L32

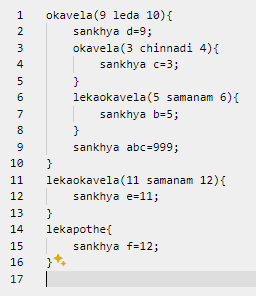
1. Instruction right conditional branch instruction

 // initialise with known dummy values



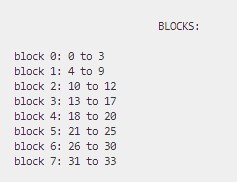


INPUT 1:

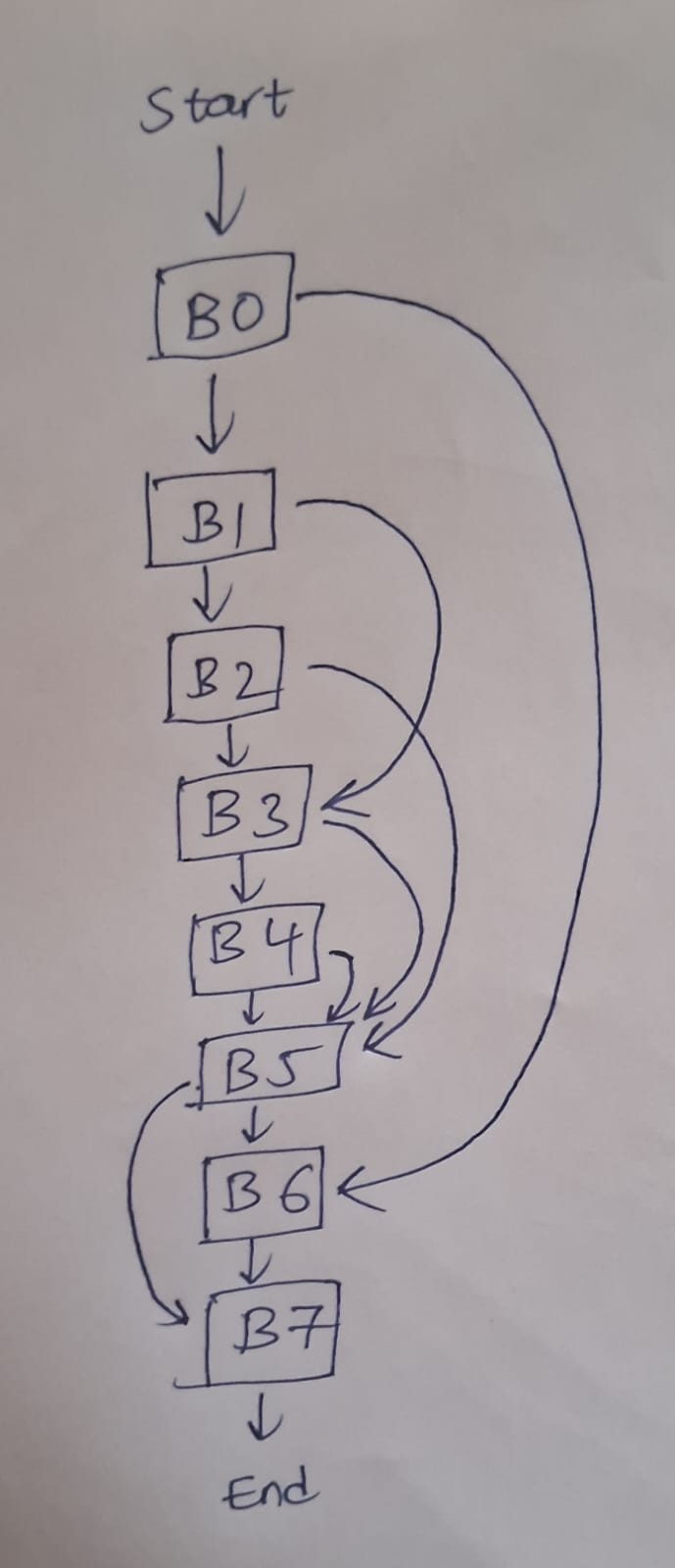


OUTPUT 1:

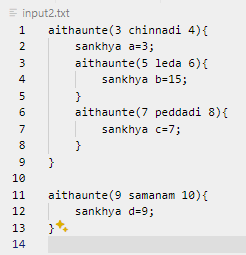




**FLOW GRAPH** for the above INPUT:

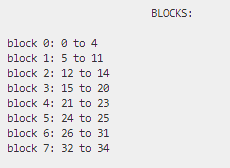


INTPUT 2:

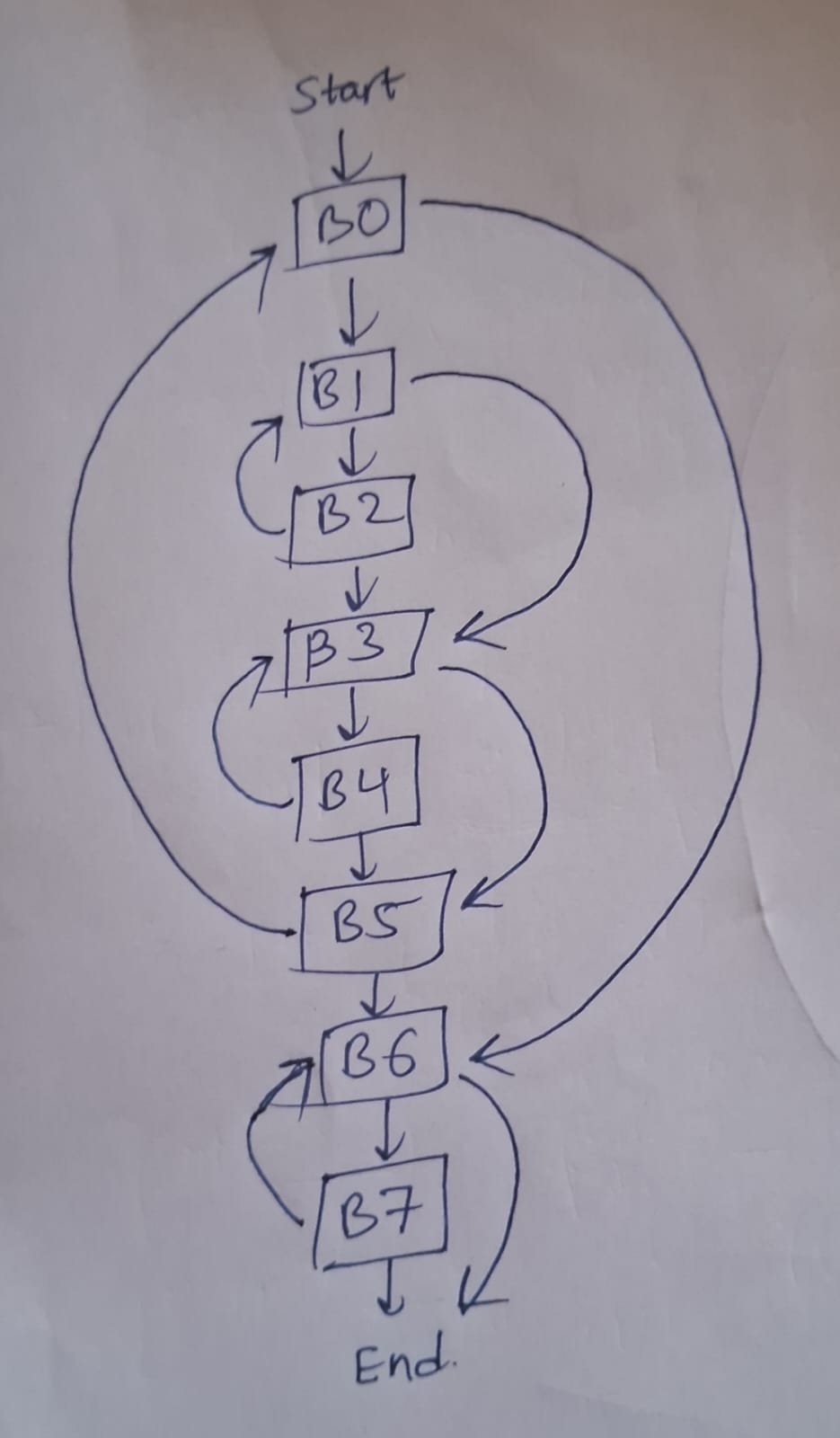


OUTPUT 2:





**FLOW GRAPH** for the above INPUT:

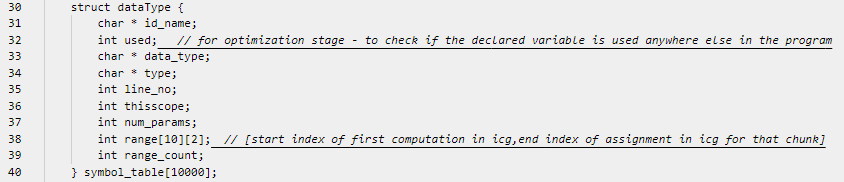


B) OPTIMIZATION

DEAD-CODE ELIMINATION:

Variables which have been declared, even if assigned some values, if not used their value anywhere else in the program is said to be a dead variables. And all of its corresponding Assignment equations need not be converted into three-address instructions.

Implementation:

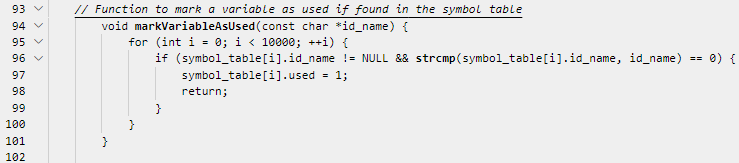


Each variable in the symbol table is marked with a Boolean used = true whenever it appears in any expression (typically in the RHS of an equation) or a print statement.

For each of these variables we track an array of ranges [start,end] where:

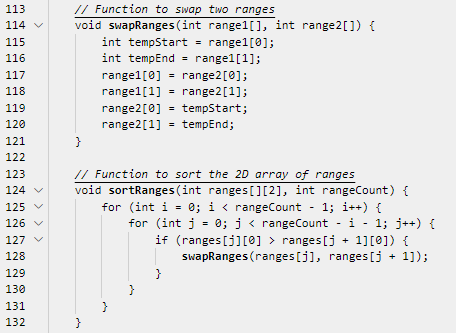
Start = index of the the first 3-address instruction in the bunch corresponding to that dead variable

End = index of the the last 3-address instruction in the bunch corresponding to that dead variable



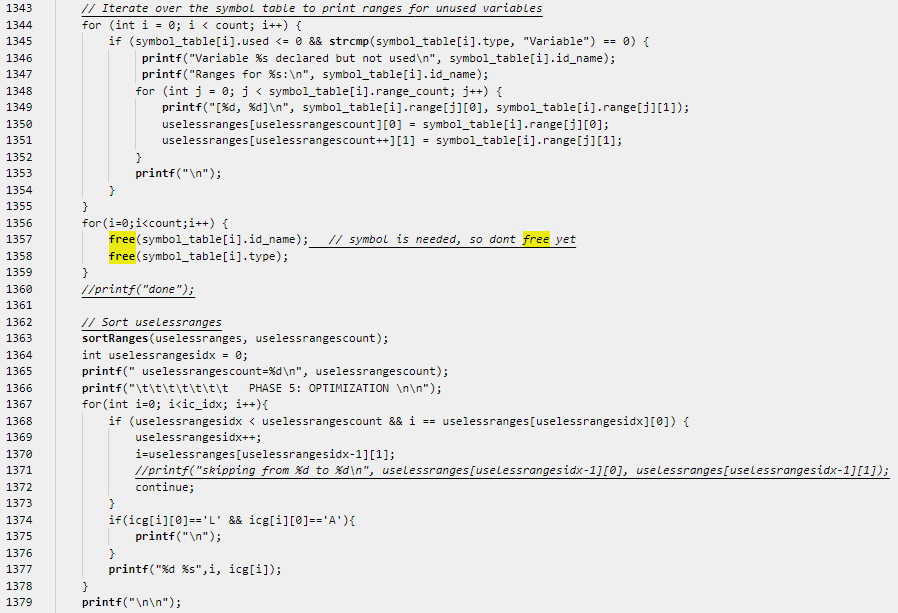
After the entire parsing is done, we iterate in the symbol table and find out which variables have used = false. We store only these dead variables’ ranges in a 2D array.

Now we sort them in ascending order.



[2,6] , [9,11] , [12,12] , [20,51] , . . . . . .

Now we don’t want to include the instructions whose indices lie in any of these ranges.



Explanation with a Sample Code:

sankhya a = 3+4;

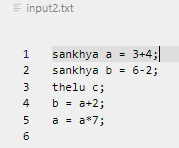
sankhya b = 6-2;

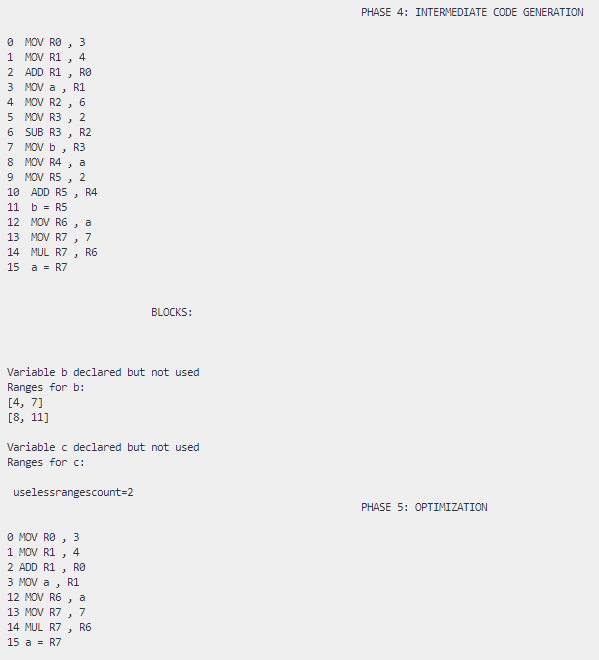
thelu c; // this declaration is useless because ‘c’ is never used

b = a+2; // a’s value has been used in computation, so ‘a’ is useful

// but b’s value has never been accessed in any print statements or RHS // of equations, so ‘b’ is also dead

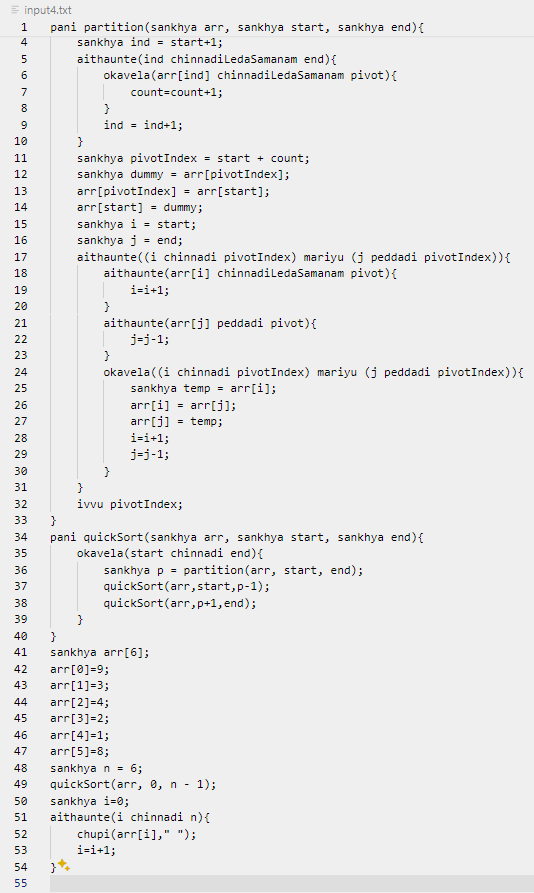
a = a\*7;

 INPUT:

OUTPUT: 

**C) QUICK SORT**

INPUT:



OUTPUT:

