# 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("QQQQQQQQQQQQQQQQQQQQ 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 XXXXXXXXXXXXXXXX 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\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\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);
}
```

# A) BASIC BLOCKS

**Basic Block** is a straight line code sequence that has no branches in and out branches except to the entry and at the end respectively. Basic Block is a set of statements that always executes one after other, in a sequence.

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

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

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

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

3. Instruction right conditional branch instruction

```
for(int i=0;i<10000;i++){
    laddercounts[i]=0;
    isleader[i]=0;
    insNumOfLabel[i]=-1;
}
```

// initialise with known dummy values

```
103     // Function to search for an identifier in the symbol table and return its index if found
104     int findIdentifierIndex(char *id_name) {
105         for (int i = 0; i < 10000; i++) {
106             if (symbol_table[i].id_name != NULL && strcmp(symbol_table[i].id_name, id_name) == 0) {
107                 return i; // Return the index if found
108             }
109         }
110         return -1; // Return -1 if not found
111     }
```

```
343 |   input {insNumOfLabel[labelsused]=ic_idx; sprintf(icg[ic_idx++], "LABEL L%d:\n", labelsused++);} function_declaration input {
344       //add('F');
345       printf("Parser found input-functionDec-input\n");
346       int num_children = 3;  // Number of children
347       struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
348       children[0] = $1.nd;
349       children[1] = $3.nd;
350       children[2] = $4.nd;
351       $$.nd = mknode(num_children, children, "input-funDec-input");
352   }
```

```
764 if_statement:
765     telugu_if telugu_open_curly_bracket condition {
766         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++;}
767         printf("Parser found if(cond){bunch}\n");
```

INPUT 1:

```
1   okavela(9 leda 10){
2       sankhya d=9;
3       okavela(3 chinnadi 4){
4           sankhya c=3;
5       }
6       lekaokavela(5 samanam 6){
7           sankhya b=5;
8       }
9       sankhya abc=999;
10  }
11  lekaokavela(11 samanam 12){
12      sankhya e=11;
13  }
14  lekapothe{
15      sankhya f=12;
16  }
17
```

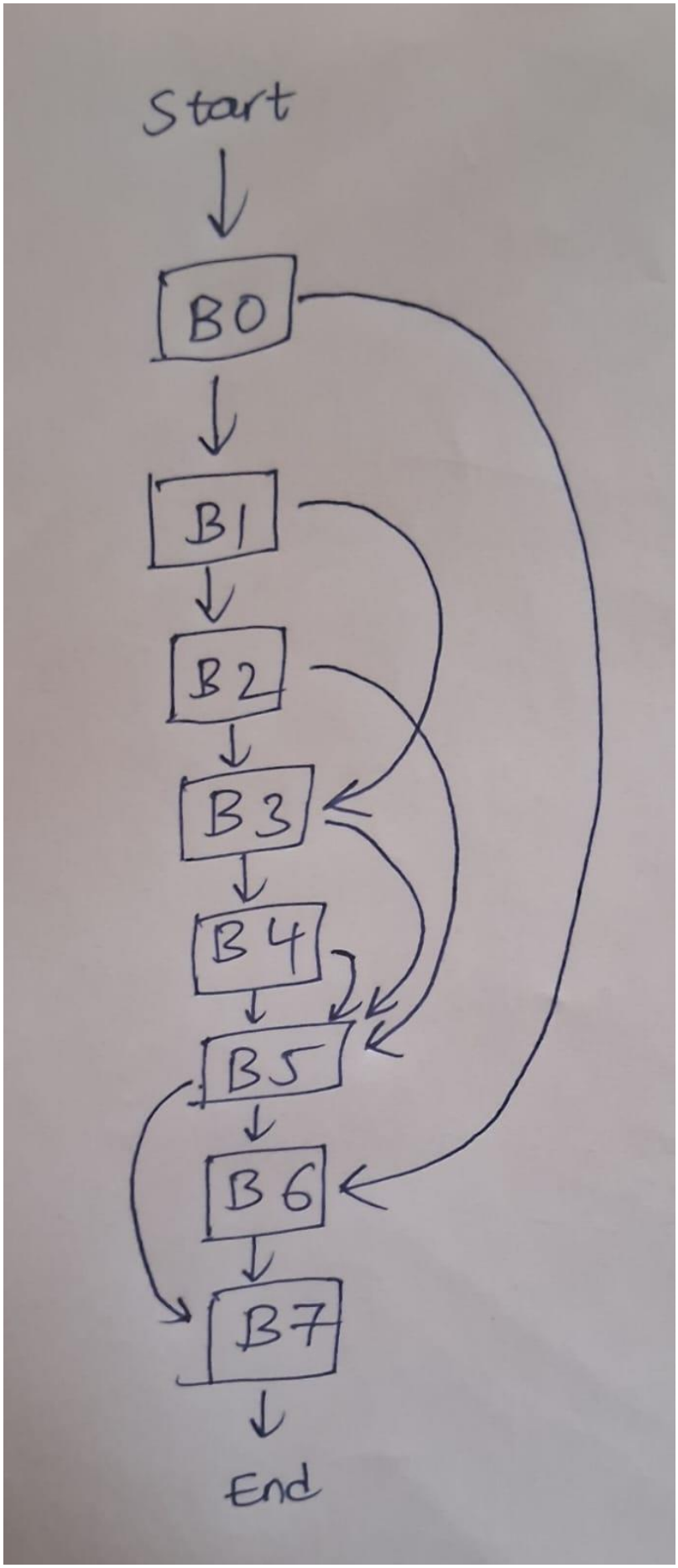
OUTPUT 1:

```
0  MOV R0 , 9
1  MOV R1 , 10
2  OR R1 , R0
3  if NOT (R1) GOTO L0
MOV R2 , 9
4  MOV R2 , 9
5  MOV d , R2
6  MOV R3 , 3
7  MOV R4 , 4
8  LT R4 , R3
9  if NOT (R4) GOTO L1
MOV R5 , 3
10  MOV R5 , 3
11  MOV c , R5
12  JUMPx L3
13  LABEL L1:
14  MOV R6 , 5
15  MOV R7 , 6
16  EQ R7 , R6
17  if NOT (R7) GOTO L2
MOV R8 , 5
18  MOV R8 , 5
19  MOV b , R8
20  JUMPx L3
21  LABEL L2:
22  LABEL L3:
23  MOV R9 , 999
24  MOV abc , R9
25  JUMPx L5
26  LABEL L0:
27  MOV R10 , 11
28  MOV R11 , 12
29  EQ R11 , R10
30  if NOT (R11) GOTO L4MOV R12 , 11
31  MOV R12 , 11
32  MOV e , R12
33  JUMPx L5
34  LABEL L4:
35  MOV R13 , 12
36  MOV f , R13
37  LABEL L5:
```

```
                        BLOCKS:

  block 0: 0 to 3
  block 1: 4 to 9
  block 2: 10 to 12
  block 3: 13 to 17
  block 4: 18 to 20
  block 5: 21 to 25
  block 6: 26 to 30
  block 7: 31 to 33
```

**FLOW GRAPH** for the above INPUT:


Start
B0
B1
B2
B3
B4
B5
B6
B7
End

INTPUT 2:

input2.txt

```
aithaunte(3 chinnadi 4){
    sankhya a=3;
    aithaunte(5 leda 6){
        sankhya b=15;
    }
    aithaunte(7 peddadi 8){
        sankhya c=7;
    }
}

aithaunte(9 samanam 10){
    sankhya d=9;
}
```

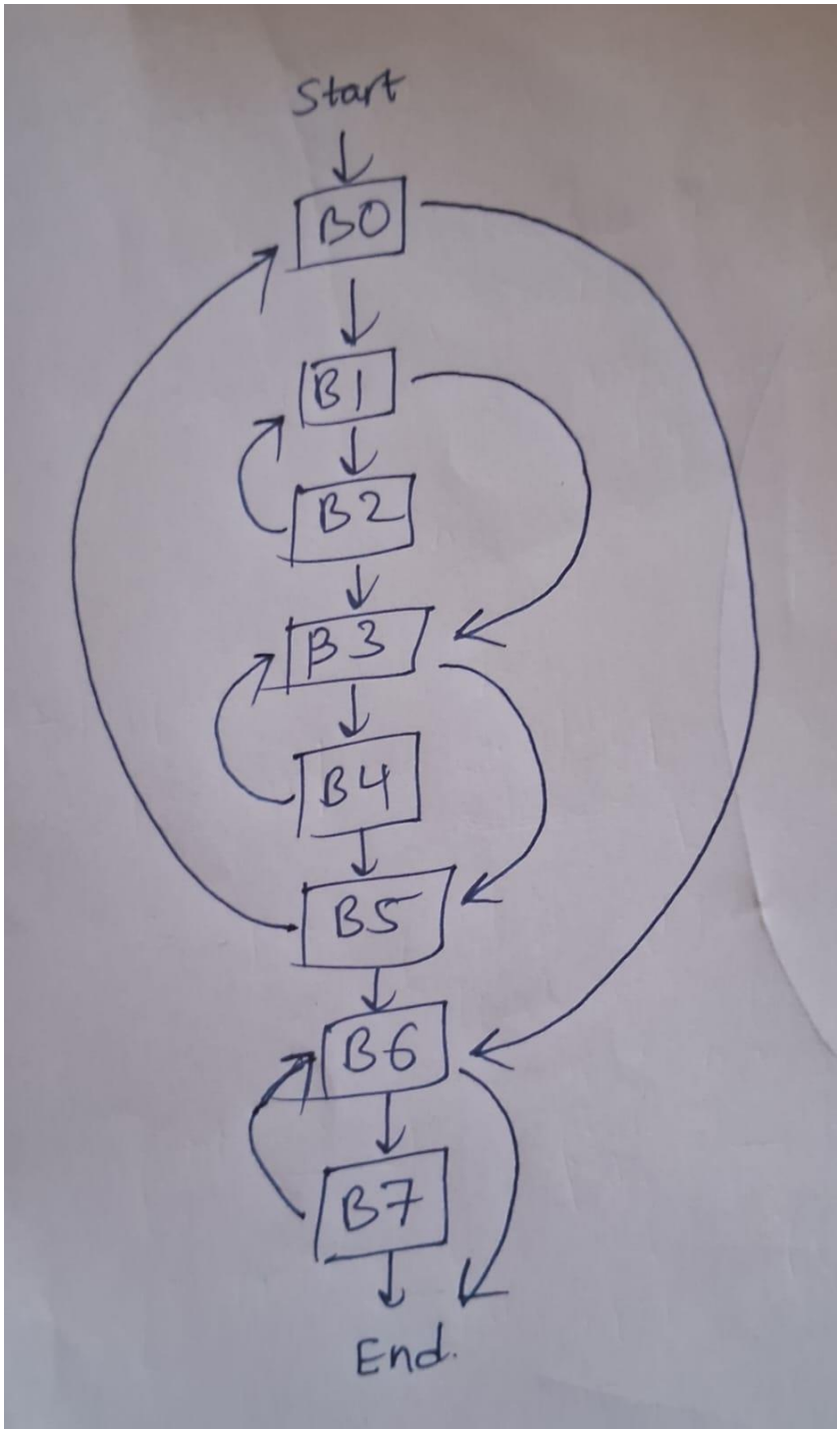
OUTPUT 2:

```
0  MOV R0 , 3
1  MOV R1 , 4
2  LT R1 , R0
3  LABEL L0:
4  if NOT (R1) GOTO L1
MOV R2 , 3
5  MOV R2 , 3
6  MOV a , R2
7  MOV R3 , 5
8  MOV R4 , 6
9  OR R4 , R3
10  LABEL L2:
11  if NOT (R4) GOTO L3
MOV R5 , 15
12  MOV R5 , 15
13  MOV b , R5
14  JUMPtoLOOP L2
15  LABEL L3:
16  MOV R6 , 7
17  MOV R7 , 8
18  GT R7 , R6
19  LABEL L4:
20  if NOT (R7) GOTO L5
MOV R8 , 7
21  MOV R8 , 7
22  MOV c , R8
23  JUMPtoLOOP L4
24  LABEL L5:
25  JUMPtoLOOP L0
26  LABEL L1:
27  MOV R9 , 9
28  MOV R10 , 10
29  EQ R10 , R9
30  LABEL L6:
31  if NOT (R10) GOTO L7MOV R11 , 9
32  MOV R11 , 9
33  MOV d , R11
34  JUMPtoLOOP L6
35  LABEL L7:
```

```
                        BLOCKS:

block 0: 0 to 4
block 1: 5 to 11
block 2: 12 to 14
block 3: 15 to 20
block 4: 21 to 23
block 5: 24 to 25
block 6: 26 to 31
block 7: 32 to 34
```

**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:

```
30    struct dataType {
31        char * id_name;
32        int used;   // for optimization stage - to check if the declared variable is used anywhere else in the program
33        char * data_type;
34        char * type;
35        int line_no;
36        int thisscope;
37        int num_params;
38        int range[10][2];  // [start index of first computation in icg,end index of assignment in icg for that chunk]
39        int range_count;
40    } symbol_table[10000];
```

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

```
93    // Function to mark a variable as used if found in the symbol table
94        void markVariableAsUsed(const char *id_name) {
95            for (int i = 0; i < 10000; ++i) {
96                if (symbol_table[i].id_name != NULL && strcmp(symbol_table[i].id_name, id_name) == 0) {
97                    symbol_table[i].used = 1;
98                    return;
99                }
100           }
101       }
102
```

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.

```
113     // Function to swap two ranges
114     void swapRanges(int range1[], int range2[]) {
115         int tempStart = range1[0];
116         int tempEnd = range1[1];
117         range1[0] = range2[0];
118         range1[1] = range2[1];
119         range2[0] = tempStart;
120         range2[1] = tempEnd;
121     }
122
123     // Function to sort the 2D array of ranges
124     void sortRanges(int ranges[][2], int rangeCount) {
125         for (int i = 0; i < rangeCount - 1; i++) {
126             for (int j = 0; j < rangeCount - i - 1; j++) {
127                 if (ranges[j][0] > ranges[j + 1][0]) {
128                     swapRanges(ranges[j], ranges[j + 1]);
129                 }
130             }
131         }
132     }
```

[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.

```
1343    // Iterate over the symbol table to print ranges for unused variables
1344    for (int i = 0; i < count; i++) {
1345        if (symbol_table[i].used <= 0 && strcmp(symbol_table[i].type, "Variable") == 0) {
1346            printf("Variable %s declared but not used\n", symbol_table[i].id_name);
1347            printf("Ranges for %s:\n", symbol_table[i].id_name);
1348            for (int j = 0; j < symbol_table[i].range_count; j++) {
1349                printf("[%d, %d]\n", symbol_table[i].range[j][0], symbol_table[i].range[j][1]);
1350                uselessranges[uselessrangescount][0] = symbol_table[i].range[j][0];
1351                uselessranges[uselessrangescount++][1] = symbol_table[i].range[j][1];
1352            }
1353            printf("\n");
1354        }
1355    }
1356    for(i=0;i<count;i++) {
1357        free(symbol_table[i].id_name);   // symbol is needed, so dont free yet
1358        free(symbol_table[i].type);
1359    }
1360    //printf("done");
1361
1362    // Sort uselessranges
1363    sortRanges(uselessranges, uselessrangescount);
1364    int uselessrangesidx = 0;
1365    printf(" uselessrangescount=%d\n", uselessrangescount);
1366    printf("\t\t\t\t\t\t\t\t   PHASE 5: OPTIMIZATION \n\n");
1367    for(int i=0; i<ic_idx; i++){
1368        if (uselessrangesidx < uselessrangescount && i == uselessranges[uselessrangesidx][0]) {
1369            uselessrangesidx++;
1370            i=uselessranges[uselessrangesidx-1][1];
1371            //printf("skipping from %d to %d\n", uselessranges[uselessrangesidx-1][0], uselessranges[uselessrangesidx-1][1]);
1372            continue;
1373        }
1374        if(icg[i][0]=='L' && icg[i][0]=='A'){
1375            printf("\n");
1376        }
1377        printf("%d %s",i, icg[i]);
1378    }
1379    printf("\n\n");
```

## 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:

input2.txt

```
1    sankhya a = 3+4;
2    sankhya b = 6-2;
3    thelu c;
4    b = a+2;
5    a = a*7;
6
```

OUTPUT:

```
                                                        PHASE 4: INTERMEDIATE CODE GENERATION

0  MOV R0 , 3
1  MOV R1 , 4
2  ADD R1 , R0
3  MOV a , R1
4  MOV R2 , 6
5  MOV R3 , 2
6  SUB R3 , R2
7  MOV b , R3
8  MOV R4 , a
9  MOV R5 , 2
10  ADD R5 , R4
11  b = R5
12  MOV R6 , a
13  MOV R7 , 7
14  MUL R7 , R6
15  a = R7


                        BLOCKS:



Variable b declared but not used
Ranges for b:
[4, 7]
[8, 11]

Variable c declared but not used
Ranges for c:

 uselessrangescount=2
                                                        PHASE 5: OPTIMIZATION

0 MOV R0 , 3
1 MOV R1 , 4
2 ADD R1 , R0
3 MOV a , R1
12 MOV R6 , a
13 MOV R7 , 7
14 MUL R7 , R6
15 a = R7
```

# C) QUICK SORT

INPUT:

input4.txt

```
pani partition(sankhya arr, sankhya start, sankhya end){
    sankhya ind = start+1;
    aithaunte(ind chinnadiLedaSamanam end){
        okavela(arr[ind] chinnadiLedaSamanam pivot){
            count=count+1;
        }
        ind = ind+1;
    }
    sankhya pivotIndex = start + count;
    sankhya dummy = arr[pivotIndex];
    arr[pivotIndex] = arr[start];
    arr[start] = dummy;
    sankhya i = start;
    sankhya j = end;
    aithaunte((i chinnadi pivotIndex) mariyu (j peddadi pivotIndex)){
        aithaunte(arr[i] chinnadiLedaSamanam pivot){
            i=i+1;
        }
        aithaunte(arr[j] peddadi pivot){
            j=j-1;
        }
        okavela((i chinnadi pivotIndex) mariyu (j peddadi pivotIndex)){
            sankhya temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            i=i+1;
            j=j-1;
        }
    }
    ivvu pivotIndex;
}
pani quickSort(sankhya arr, sankhya start, sankhya end){
    okavela(start chinnadi end){
        sankhya p = partition(arr, start, end);
        quickSort(arr,start,p-1);
        quickSort(arr,p+1,end);
    }
}
sankhya arr[6];
arr[0]=9;
arr[1]=3;
arr[2]=4;
arr[3]=2;
arr[4]=1;
arr[5]=8;
sankhya n = 6;
quickSort(arr, 0, n - 1);
sankhya i=0;
aithaunte(i chinnadi n){
    chupi(arr[i]," ");
    i=i+1;
}
```

OUTPUT:

BLOCKS:

```
block 0: 0 to 12
block 1: 13 to 14
block 2: 15 to 19
block 3: 20 to 24
block 4: 25 to 31
block 5: 32 to 59
block 6: 60 to 65
block 7: 66 to 70
block 8: 71 to 77
block 9: 78 to 91
block 10: 92 to 113
block 11: 114 to 120


 uselessrangescount=0
```

PHASE 5: OPTIMIZATION

```
0 LABEL L0:
1 MOV R0 , start
2 MOV R0 , arr+R0
3 MOV pivot , R0
4 MOV R1 , 0
5 MOV count , R1
6 MOV R2 , start
7 MOV R3 , 1
8 ADD R3 , R2
9 MOV ind , R3
10 MOV R4 , ind
11 MOV R5 , end
12 LE R5 , R4
13 LABEL L1:
14 if NOT (R5) GOTO L2
MOV R6 , ind
15 MOV R6 , ind
16 MOV R6 , arr+R6
17 MOV R7 , pivot
18 LE R7 , R6
19 if NOT (R7) GOTO L3
MOV R8 , count
20 MOV R8 , count
21 MOV R9 , 1
22 ADD R9 , R8
23 count = R9
24 JUMPx L4
25 LABEL L3:
26 LABEL L4:
27 MOV R10 , ind
28 MOV R11 , 1
29 ADD R11 , R10
30 ind = R11
31 JUMPtoLOOP L1
32 LABEL L2:
33 MOV R12 , start
34 MOV R13 , count
35 ADD R13 , R12
36 MOV pivotIndex , R13MOV R14 , pivotIndexMOV R14 , arr+R14
37 MOV R14 , pivotIndexMOV R14 , arr+R14
38 MOV R14 , arr+R14
39 MOV dummy , R14
40 MOV R15 , pivotIndexMOV R16 , start
41 MOV R16 , start
42 MOV R16 , arr+R16
43 MOV arr+R15 , R16
```

```
44 MOV R17 , start
45 MOV R18 , dummy
46 MOV arr+R17 , R18
47 MOV R19 , start
48 MOV i , R19
49 MOV R20 , end
50 MOV j , R20
51 MOV R21 , i
52 MOV R22 , pivotIndexLT R22 , R21
53 LT R22 , R21
54 MOV R23 , j
55 MOV R24 , pivotIndexGT R24 , R23
56 GT R24 , R23
57 AND R24 , R23
58 LABEL L5:
59 if NOT (R24) GOTO L6MOV R25 , i
60 MOV R25 , i
61 MOV R25 , arr+R25
62 MOV R26 , pivot
63 LE R26 , R25
64 LABEL L7:
65 if NOT (R26) GOTO L8MOV R27 , i
66 MOV R27 , i
67 MOV R28 , 1
68 ADD R28 , R27
69 i = R28
70 JUMPtoLOOP L7
71 LABEL L8:
72 MOV R29 , j
73 MOV R29 , arr+R29
74 MOV R30 , pivot
75 GT R30 , R29
76 LABEL L9:
77 if NOT (R30) GOTO L1MOV R31 , j
78 MOV R31 , j
79 MOV R32 , 1
80 SUB R32 , R31
81 j = R32
82 JUMPtoLOOP L9
83 LABEL L10:
84 MOV R33 , i
85 MOV R34 , pivotIndexLT R34 , R33
86 LT R34 , R33
87 MOV R35 , j
88 MOV R36 , pivotIndexGT R36 , R35
89 GT R36 , R35
90 AND R36 , R35
91 if NOT (R36) GOTO L1MOV R37 , i
92 MOV R37 , i
93 MOV R37 , arr+R37
```

```
94 MOV temp , R37
95 MOV R38 , i
96 MOV R39 , j
97 MOV R39 , arr+R39
98 MOV arr+R38 , R39
99 MOV R40 , j
100 MOV R41 , temp
101 MOV arr+R40 , R41
102 MOV R42 , i
103 MOV R43 , 1
104 ADD R43 , R42
105 i = R43
106 MOV R44 , j
107 MOV R45 , 1
108 SUB R45 , R44
109 j = R45
110 JUMPx L12
111 LABEL L11:
112 LABEL L12:
113 JUMPtoLOOP L5
114 LABEL L6:
115 MOV R46 , pivotIndexLABEL L13:
116 LABEL L13:
117 MOV R47 , start
118 MOV R48 , end
119 LT R48 , R47
120 if NOT (R48) GOTO L1MOV R49 , partition
MOV R50 , arr
121 MOV R49 , partition
MOV R50 , arr
122 MOV R50 , arr
123 PARAM arr
124 MOV R51 , start
125 PARAM start
126 MOV R52 , p
127 MOV R53 , 1
128 SUB R53 , R52
129 PARAM p
130 PARAM start
131 PARAM arr
132 CALL quickSort
133 MOV R54 , arr
134 PARAM arr
135 MOV R55 , p
136 MOV R56 , 1
137 ADD R56 , R55
138 PARAM p
139 MOV R57 , end
140 PARAM end
141 PARAM p
142 PARAM arr
143 CALL quickSort
144 JUMPx L15
145 LABEL L14:
```

```
MOV R0 , 6
MOV R1 , 0
MOV R2 , 9
MOV arr+R1 , R2
MOV R3 , 1
MOV R4 , 3
MOV arr+R3 , R4
MOV R5 , 2
MOV R6 , 4
MOV arr+R5 , R6
MOV R7 , 3
MOV R8 , 2
MOV arr+R7 , R8
MOV R9 , 4
MOV R10 , 1
MOV arr+R9 , R10
MOV R11 , 5
MOV R12 , 8
MOV arr+R11 , R12
MOV R13 , 6
MOV n , R13
MOV R14 , arr
PARAM arr
MOV R15 , 0
PARAM 0
MOV R16 , n
MOV R17 , 1
SUB R17 , R16
PARAM n
PARAM 0
PARAM arr
CALL quickSort
MOV R18 , 0
MOV i , R18
MOV R19 , i
MOV R20 , n
LT R20 , R19
LABEL L0:
if NOT (R20) GOTO L1MOV R21 , i
MOV R21 , i
MOV R21 , arr+R21
MOV R22 , " "
MOV R23 , i
MOV R24 , 1
ADD R24 , R23
i = R24
JUMPtoLOOP L0
LABEL L1:
```