UPDATES IN PARSER CODE:

t2parser.y

```
52  // Intermediate code generation
53  int ic_idx=0;    // used to index the intermediate 3 address codes to show them together later in output
54  int label[100];     // label stack to store the order of labels in the intermediate code
55  |   |   |   |   |   // label number in the intermediate code -> GOTO L4
56  |   |   |   |   |   // LABEL L4: ....
57  int ifelsetracker=-1;  // used to store the ending label for an if-elseLadder
58  int jumpcorrection[100]; // jumpcorrection[instruction number] = label number after a if-else Ladder
59  int lastjumps[100];
60  int lastjumpstackpointer=0;
61  int laddercounts[100];
62  int laddercountstackpointer=0;
63
64  int stackpointer=0; // used to index the label stack
65  int labelsused=0;   // used to keep track of the number of labels used in the intermediate code
66  int looplabel[100]; // another stack
67  int looplabelstackpointer=0; // another stack pointer
68
69  int gotolabel[100]; // another stack
70  int gotolabelstackpointer=0; // another stack pointer
71
72
73  char icg[100][100];  // stores the intermediate code instructions themselves as strings
74  int registerIndex=0;  // used to index the registers used in the intermediate code
75  int registers[100];   // stores the registers used in the intermediate code
76  int regstackpointer=0; // used to index the register stack
77  int firstreg=-1,secondreg=-1,thirdreg=-1; // used to track regIndices in exp*exp
78
```

```
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 {sprintf(icg[ic_idx++], "\nLABEL 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");
        }
    ;

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);
            }
    | 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);
```

```
420     }
421 |   exp {firstreg = registerIndex-1;registers[registerIndex-1]=firstreg;} telugu_arithmetic_operator exp
422     {secondreg = registerIndex-1;registers[registerIndex-1]=secondreg;} {
423         printf("Parser found exp-arithmeticOp-exp\n");
424         int num_children = 3;  // Number of children
425         struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
426 
427         // Assigning children nodes
428         children[0] = $1.nd;  // Assuming $1 represents the parse tree node for symbol1
429         children[1] = $3.nd;  // Assuming $2 represents the parse tree node for symbol2
430         children[2] = $4.nd;
431         // Assign more children if needed
432 
433         // Create the parse tree node for the production rule
434         $$.nd = mknode(num_children, children, "AthematicOp");
435 
436         // Free the memory allocated for the array of children
437         //free(children);
438         //regstackpointer--;
439         if(($3.name)[0] == '+')
440             sprintf(icg[ic_idx++], "ADD R%d , R%d\n", secondreg , registers[--regstackpointer]-1);
441         else if(($3.name)[0] == '-')
442             sprintf(icg[ic_idx++], "SUB R%d , R%d\n", secondreg , registers[--regstackpointer]-1);
443         else if(($3.name)[0] == '*')
444             sprintf(icg[ic_idx++], "MUL R%d , R%d\n", secondreg , registers[--regstackpointer]-1);
445         else if(($3.name)[0] == '/')
446             sprintf(icg[ic_idx++], "DIV R%d , R%d\n", secondreg , registers[--regstackpointer]-1);
447         else if(($3.name)[0] == '%')
448             sprintf(icg[ic_idx++], "MOD R%d , R%d\n", secondreg , registers[--regstackpointer]-1);
449         else{
450             sprintf(icg[ic_idx++], "R%d = R%d %c R%d\n", secondreg , registers[--regstackpointer]-1, ($3.name)[0],secondreg);
451         }
452         //secondreg = firstreg;
453         //first = registers[regstackpointer];
454 
455     }
456 |   exp {firstreg = registerIndex-1;} telugu_logical_operator exp {secondreg = registerIndex-1;} {
457         printf("Parser found exp-logicalOp-exp\n");
458         int num_children = 3;  // Number of children
459         struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
460         children[0] = $1.nd;
461         children[1] = $3.nd;
462         children[2] = $4.nd;
463         $$.nd = mknode(num_children, children, "LogicalOp");
464         //sprintf(icg[ic_idx++], "R%d = R%d %s R%d\n", secondreg , firstreg, $3.name, secondreg);
465         if (strcmp($3.name, "mariyu") == 0) {
466             sprintf(icg[ic_idx++], "AND R%d , R%d\n", secondreg , firstreg);
467         }
468         else if (strcmp($3.name, "Leda") == 0) {
469             sprintf(icg[ic_idx++], "OR R%d , R%d\n", secondreg , firstreg);
470         }
471         // else if (strcmp($3.name, "kaadu") == 0) {
472         //     sprintf(icg[ic_idx++], "NOT R%d , R%d\n", secondreg , firstreg);
473         // }
474         else if (strcmp($3.name, "pratyekam") == 0) {
```

```
475             sprintf(icg[ic_idx++], "XOR R%d , R%d\n", secondreg , firstreg);
476         }
477         else{
478             sprintf(icg[ic_idx++], "R%d = R%d %s R%d\n", secondreg , firstreg, $3.name,secondreg);
479         }
480
481
482     }
483 |   telugu_identifier telugu_open_square_bracket exp {registers[regstackpointer++]=registerIndex;
484         sprintf(icg[ic_idx++], "MOV R%d , %s + R%d\n", registerIndex-1 , $1.name, registerIndex-1);} telugu_closed_square_bracket {
485         printf("Parser found id[exp]\n");
486         int num_children = 4;  // Number of children
487         struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
488         children[0] = $1.nd;
489         children[1] = $2.nd;
490         children[2] = $3.nd;
491         children[3] = $5.nd;
492         $$.nd = mknode(num_children, children, "id[exp]");
493         //sprintf(icg[ic_idx++], "MOV R%d , %s + R%d\n", firstreg , $1.name, firstreg);
494     }
495 ;
496
497
498 bunch_of_statements:  //can be empty
499     { $$.nd = mknode(NULL, NULL, "empty"); }
500 |   eol bunch_of_statements {
501         printf("Parser found EOL-bunch\n");
502         int num_children = 2;  // Number of children
503         struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
504         children[0] = $1.nd;
505         children[1] = $2.nd;
506         $$.nd = mknode(num_children, children, "eol-bunch");
507     }
508 |   bunch_of_statements eol {
509         printf("Parser found bunch-EOL\n");
510         int num_children = 2;  // Number of children
511         struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
512         children[0] = $1.nd;
513         children[1] = $2.nd;
514         $$.nd = mknode(num_children, children, "bunch-eol");
515     }
516 |   bunch_of_statements if_else_ladder {
517             sprintf(icg[ic_idx++], "\nLABEL L%d:\n", labelsused++);
518             //lastjumps[lastjumpstackpointer++] = label[stackpointer-2];
519             int index = ic_idx - 1;
520             int count = laddercounts[laddercountstackpointer-1]; // Number of iterations
521
522             for (int i = index; i >= 0 && count > 0; i--) {
523                 printf("icg[%d] = %s\n", i, icg[i]);
524                 if (strncmp(icg[i], "JUMP ", 5) == 0) { // Check if the prefix matches "JUMP "
525                     printf("..................\n");
526                     char jump_str[20]; // Assuming the number won't exceed 20 digits
527                     sprintf(jump_str, "%d", labelsused-1); // Convert number to string
528                     snprintf(icg[i], 20, "JUMPx L%s\n", jump_str); // Set icg[i] to "JUMP" followed by the number
529                     count--;
```

```
condition:  // for if_statement and while loop, empty not allowed
    exp {
        printf("Parser found exp\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);
    }

```

```
676 if_statement:
677     telugu_if telugu_open_curly_bracket condition {
678         sprintf(icg[ic_idx++], "if NOT (R%d) GOTO L%d\n",registerIndex-1,labelsused);label[stackpointer++]=labelsused++;}
679         telugu_closed_curly_bracket telugu_open_floor_bracket bunch_of_statements
680         {sprintf(icg[ic_idx++], "JUMP L%d\n",label[stackpointer-1]);} telugu_closed_floor_bracket {
681         printf("Parser found if(cond){bunch}\n");
682         int num_children = 7;  // Number of children
683         struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
684         children[0] = $1.nd;
685         children[1] = $2.nd;
686         children[2] = $3.nd;
687         children[3] = $5.nd;
688         children[4] = $6.nd;
689         children[5] = $7.nd;
690         children[6] = $9.nd;
691         $$.nd = mknode(num_children, children, "if(cond){bunch}");
692         sprintf(icg[ic_idx++], "\nLABEL L%d:\n", label[--stackpointer]);
693         laddercounts[laddercountstackpointer++]=1;
694 
695     }
696 
697 elif_repeat: //can be empty
698     { $$.nd = mknode(NULL, NULL, "empty"); }
699     eol elif_repeat {
700         printf("Parser found eol elif_repeat\n");
701         int num_children = 2;  // Number of children
702         struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
703         children[0] = $1.nd;
704         children[1] = $2.nd;
705         $$.nd = mknode(num_children, children, "EOL-elifrepeat");
706     }
707     elif_repeat telugu_elif telugu_open_curly_bracket condition
708     {sprintf(icg[ic_idx++], "if NOT (R%d) GOTO L%d\n",registerIndex-1,labelsused);label[stackpointer++]=labelsused++;}
709     telugu_closed_curly_bracket telugu_open_floor_bracket bunch_of_statements
710     {sprintf(icg[ic_idx++], "JUMP L%d\n",label[stackpointer-1]);} telugu_closed_floor_bracket
711     { sprintf(icg[ic_idx++], "\nLABEL L%d:\n", label[--stackpointer]);} elif_repeat {
712         printf("Parser found elif(cond){bunch}\n");
713         int num_children = 9;  // Number of children
714         struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
715         children[0] = $1.nd;
716         children[1] = $2.nd;
717         children[2] = $3.nd;
718         children[3] = $4.nd;
719         children[4] = $6.nd;
720         children[5] = $7.nd;
721         children[6] = $8.nd;
722         children[7] = $10.nd;
723         children[8] = $12.nd;
724         $$.nd = mknode(num_children, children, "elif(cond){bunch}");
725         laddercounts[laddercountstackpointer-1]++;
726 
727     }
728 
729 else_statement:  //can be empty
730     { $$.nd = mknode(NULL, NULL, "empty"); }
```

```
750 if_else_ladder:
751     if_statement elif_repeat
752     {
753         lastjumps[lastjumpstackpointer++] = label[stackpointer-1];
754 
755     }
756     else_statement {
757     printf("Parser found ifElseLadder\n");
758     int num_children = 3;  // Number of children
759     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
760     children[0] = $1.nd;
761     children[1] = $2.nd;
762     children[2] = $4.nd;
763     $$.nd = mknode(num_children, children, "ifElseLadder");
764     // lastjumpstackpointer--;  // forgetting the current ifelseLadder's lastjump and counts
765     // laddercountstackpointer--;
766     }
767 |   if_statement elif_repeat
768     {
769         lastjumps[lastjumpstackpointer++] = label[stackpointer-1];
770         // int index = ic_idx - 1;
771         // int count = laddercounts[laddercountstackpointer-1]; // Number of iterations
772 
773         // for (int i = index; i >= 0 && count > 0; i--) {
774         //     if (strncmp(icg[i], "JUMP ", 5) == 0) { // Check if the prefix matches "JUMP "
775         //         char jump_str[20]; // Assuming the number won't exceed 20 digits
776         //         sprintf(jump_str, "%d", lastjumps[lastjumpstackpointer-1]); // Convert number to string
777         //         snprintf(icg[i], 20, "JUMPx L%s\n", jump_str); // Set icg[i] to "JUMP" followed by the number
778         //         count--;
779         //     }
780         // }
781 
782     }
783     {    // without the else statement
784     printf("Parser found ifElseLadder\n");
785     int num_children = 2;  // Number of children
786     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
787     children[0] = $1.nd;
788     children[1] = $2.nd;
789     $$.nd = mknode(num_children, children, "ifElseLadder");
790     }
791 
792 while_loop:
793     telugu_while telugu_open_curly_bracket condition
794     { looplabel[looplabelstackpointer++] = labelsused; sprintf(icg[ic_idx++], "\nLABEL L%d:\n", labelsused++);
795     gotolabel[gotolabelstackpointer++]=labelsused;  sprintf(icg[ic_idx++], "if NOT (R%d) GOTO L%d\n",registerIndex-1,labelsused++);}
796     telugu_closed_curly_bracket telugu_open_floor_bracket bunch_of_statements { sprintf(icg[ic_idx++], "JUMPtoLOOP L%d\n",looplabel[--looplabelstackpoi
797     telugu_closed_floor_bracket { sprintf(icg[ic_idx++], "\nLABEL L%d:\n", gotolabel[--gotolabelstackpointer]);} {
798     printf("Parser found while(cond){bunch}\n");
799     int num_children = 7;  // Number of children
800     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
801     children[0] = $1.nd;
802     children[1] = $2.nd;
803     children[2] = $3.nd;
804     children[3] = $5.nd;
```

```
805         children[4] = $6.nd;
806         children[5] = $7.nd;
807         children[6] = $9.nd;
808         $$.nd = mknode(num_children, children, "while(cond){bunch}");
809     }
810
811 variable_declaration:
812     telugu_datatype telugu_identifier_declaring telugu_assignment_operator exp {
813         //add('V');   // this is taking ';' as a variable
814         printf("Parser found datatypeId=exp\n");
815         int num_children = 4;  // Number of children
816         struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
817         children[0] = $1.nd;
818         children[1] = $2.nd;
819         children[2] = $3.nd;
820         children[3] = $4.nd;
821         $$.nd = mknode(num_children, children, "datatypeId=exp");
822         if(strcmp(exp_type,$1.name)!=0 && strcmp(exp_type, " ")!=0){
823             sprintf("$1name=%s and exp_type=%s\n", $1.name,exp_type);
824             sprintf(errors[sem_errors], "Line %d: Data type casting not allowed in declaration\n", countn);
825             sem_errors++;
826         }
827
828         sprintf(icg[ic_idx++], "MOV %s , R%d\n", $2.name, registerIndex-1);
829     }
```

```
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;
        $$.nd = mknode(num_children, children, "paramEnd");
        sprintf(icg[ic_idx++], "PARAM %s\n", $1.name);
    }
|   telugu_identifier telugu_punctuation_comma identifiers_repeat {
        curr_num_args++;
        printf("Parser found id-comma-prep\n");
        int num_children = 3;  // Number of children
        struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
        children[0] = $1.nd;
        children[1] = $2.nd;
        children[2] = $3.nd;
        $$.nd = mknode(num_children, children, "paramRep");
        sprintf(icg[ic_idx++], "PARAM %s\n", $1.name);
    }
|   telugu_constant telugu_punctuation_comma identifiers_repeat {
        curr_num_args++;
        printf("Parser found const-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);
    }

```

```
equation:
    telugu_identifier telugu_assignment_operator { strcpy(exp_type," "); } 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] = $4.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

        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;
        $$.nd = mknode(num_children, children, "id[exp]=exp");
        sprintf(icg[ic_idx++], "MOV %s+R%d , R%d\n", $1.name,thirdreg ,registerIndex-1);
    }
function_call:
    telugu_identifier { check_declaration($1.name); } telugu_open_curly_bracket identifiers_line telugu_closed_curly_bracket {
        printf("Parser found id(idLine)Finish\n");
        int num_children = 4;  // Number of children
        struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
        children[0] = $1.nd;
        children[1] = $3.nd;
        children[2] = $4.nd;
        children[3] = $5.nd;
        $$.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);
    }
%%
```

```
1122  int main(){
1123      for(int i=0;i<100;i++){
1124          laddercounts[i]=0;
1125      }

1154      printf("\t\t\t\t\t\t\t   PHASE 4: INTERMEDIATE CODE GENERATION \n\n");
1155      for(int i=0; i<ic_idx; i++){
1156          printf("%s", icg[i]);
1157      }
1158      printf("\n\n");
1159      return 0;
1160  }
```

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

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

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

1.Variable declaration:

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

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

2.expressions and equations

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

MOV R3,9

MOV R4,2

```
DIV R3,R4          // put R3/R4 in R3
MUL R2,R3
SUB p,R2
```

3. Array indexing

```
sankhya b=arr[3];  ->  MOV R1 , 3
                       MOV R1 , arr+R1          // move arr[R1] into R1
                       MOV b , R1

arr[2] = 3+4;      ->  MOV R2 , 2
                       MOV R3 , 3
                       MOV R4 , 4
                       ADD R4 , R3
                       MOV arr+R2 , R4          // move R4 into arr[R2]
```

4. Conditional Branching:

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

5. Un-Conditional Branching:

```
JUMP L4                  // go to label L4 and proceed execution
```

6. If-Else Ladders:

```
okavela(condition1){
   // do something1
}
lekapothe{
   // do something2
```

}

Corresponding Intermediate Code for the above example:

```
if NOT (condition1) GOTO L0
JUMP L1

LABEL L0:
    // something1
LABEL L1:
    //something2
```

6.Loops:

```
aithaunte(condition){
  //something1
}
//something2
```

Corresponding Intermediate Code for the above example:

```
LABEL L0:
If NOT (condition) GOTO L1
    //something1
    JUMP L0

LABEL L1:
    //something2
```

7.Function Call:

```
pani sum(sankhya a, sankhya b){        // function declaration
        chupi("addition is ",a+b);
          ivvu;
}
sum(15,24);                            // calling the function
```

Intermediate Code:

```
PARAM 15
PARAM 24
CALL sum
```

```
52  // Intermediate code generation
53  int ic_idx=0;    // used to index the intermediate 3 address codes to show them together later in output
54  int label[100];     // label stack to store the order of labels in the intermediate code
55                      // label number in the intermediate code -> GOTO L4
56                      // LABEL L4: ....
57  int ifelsetracker=-1;  // used to store the ending label for an if-elseLadder
58  int jumpcorrection[100]; // jumpcorrection[instruction number] = label number after a if-else Ladder
59  int lastjumps[100];
60  int lastjumpstackpointer=0;
61  int laddercounts[100];
62  int laddercountstackpointer=0;
63
64  int stackpointer=0; // used to index the label stack
65  int labelsused=0;   // used to keep track of the number of labels used in the intermediate code
66  int looplabel[100]; // another stack
67  int looplabelstackpointer=0; // another stack pointer
68
69  int gotolabel[100]; // another stack
70  int gotolabelstackpointer=0; // another stack pointer
71
72
73  char icg[100][100];  // stores the intermediate code instructions themselves as strings
74  int registerIndex=0;  // used to index the registers used in the intermediate code
75  int registers[100];   // stores the registers used in the intermediate code
76  int regstackpointer=0; // used to index the register stack
77  int firstreg=-1,secondreg=-1,thirdreg=-1; // used to track regIndices in exp*exp
78
```

Matching the GOTO and LABELS

Stacks have been used to handle matching GOTO statements with LABELS

On seeing a branch statement:

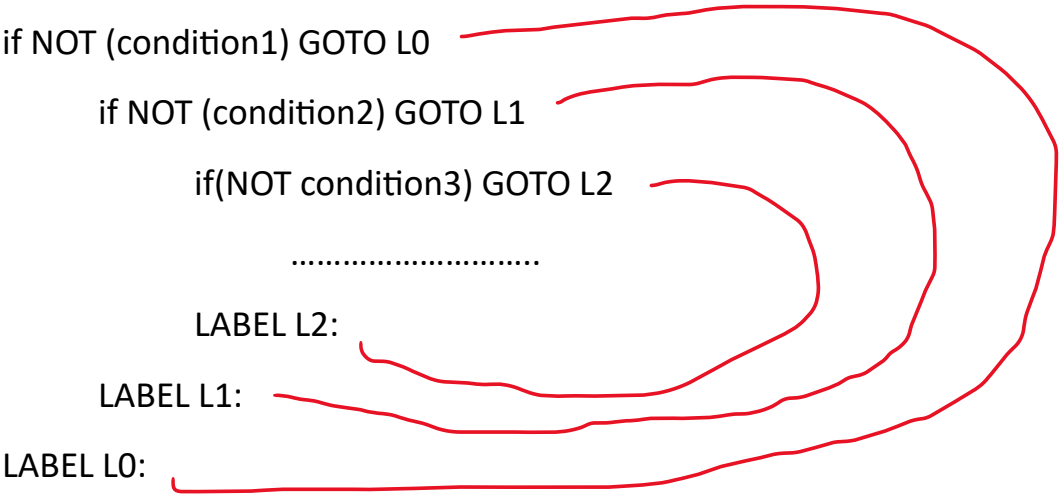
If NOT (condition) GOTO L0 // push “GOTO L0” on to the labelsused stack

……

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

// pop the topmost Label from the labelsused stack

In case of Nested If-else:



On seeing a Loop:

LABEL L0:

If NOT (loop condition) GOTO L1

……..

JUMP L0

LABEL L1: // out of the previous loop

In case of Nested Loops:

LABEL L0:

if NOT (condition1) GOTO L1

LABEL L2:

if NOT (condition2) GOTO L3

LABEL L4:

if NOT (condition3) GOTO L5

JUMPtoLOOP L4

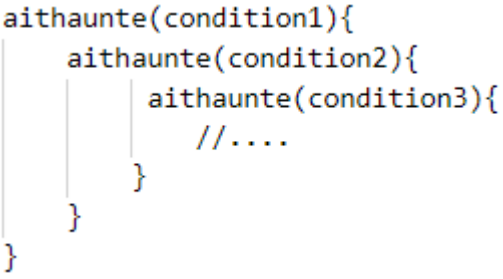
LABEL L5:

JUMPtoLOOP L2

LABEL L3:

JUMPtoLOOP L0

LABEL L1:



BACKPATCHING

```
okavela (condition1){                // IF
        …….
}
lekaokavela (condition2){          // ELSE IF
        // LABEL L0
        ……
}
lekaokavela (condition3){          // ELSE IF
        // LABEL L1
        ……..
}
Lekapothe{                              // ELSE
        // LABEL L2
        ……
}
// LABEL L3
……..
```

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

```
if NOT (condition1) GOTO L0
JUMP L3                          // How do we know L3? (it could be any label)
```

LABEL L0:

if NOT (condition2) GOTO L1

JUMP L3

LABEL L1:

if NOT (condition3) GOTO L2

JUMP L3

LABEL L2:

LABEL L3:

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

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

```
516 | bunch_of_statements if_else_ladder {
517         sprintf(icg[ic_idx++], "\nLABEL L%d:\n", labelsused++);
518         //lastjumps[lastjumpstackpointer++] = label[stackpointer-2];
519         int index = ic_idx - 1;
520         int count = laddercounts[laddercountstackpointer-1]; // Number of iterations
521 
522         for (int i = index; i >= 0 && count > 0; i--) {
523             printf("icg[%d] = %s\n", i, icg[i]);
524             if (strncmp(icg[i], "JUMP ", 5) == 0) { // Check if the prefix matches "JUMP "
525                 printf("...................\n");
526                 char jump_str[20]; // Assuming the number won't exceed 20 digits
527                 sprintf(jump_str, "%d", labelsused-1); // Convert number to string
528                 snprintf(icg[i], 20, "JUMPx L%s\n", jump_str); // Set icg[i] to "JUMP" followed by the number
529                 count--;
530             }
531         }
532         lastjumpstackpointer--;  // forgetting the current ifelseLadder's lastjump and counts
533         laddercountstackpointer--;
534 
```

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

INPUT 1:

Sub-Expressions

```
1    sankhya c=3+4*(6/2);
```

OUTPUT 1:

PHASE 4: INTERMEDIATE CODE GENERATION

```
MOV R0 , 3
MOV R1 , 4
MOV R2 , 6
MOV R3 , 2
DIV R3 , R2
MUL R3 , R1
ADD R3 , R0
MOV c , R3
```

INPUT 2:

If-else Ladders

```
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 2:

```
MOV R0 , 9
MOV R1 , 10
OR R1 , R0
if NOT (R1) GOTO L0
MOV R2 , 9
MOV d , R2
MOV R3 , 3
MOV R4 , 4
LT R5 R3 R4
if NOT (R5) GOTO L1
MOV R6 , 3
MOV c , R6
JUMPx L3

LABEL L1:
MOV R7 , 5
MOV R8 , 6
EQ R9 R7 R8
if NOT (R9) GOTO L2
MOV R10 , 5
MOV b , R10
JUMPx L3

LABEL L2:

LABEL L3:
MOV R11 , 999
MOV abc , R11
JUMPx L5

LABEL L0:
MOV R12 , 11
MOV R13 , 12
EQ R14 R12 R13
if NOT (R14) GOTO L4
MOV R15 , 11
MOV e , R15
JUMPx L5

LABEL L4:
MOV R16 , 12
MOV f , R16

LABEL L5:
```

INPUT 3:

Loops

```
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 3:

```
MOV R0 , 3
MOV R1 , 4
LT R2 R0 R1

LABEL L0:
if NOT (R2) GOTO L1
MOV R3 , 3
MOV a , R3
MOV R4 , 5
MOV R5 , 6
OR R5 , R4

LABEL L2:
if NOT (R5) GOTO L3
MOV R6 , 15
MOV b , R6
JUMPtoLOOP L2

LABEL L3:
MOV R7 , 7
MOV R8 , 8
GT R9 R7 R8

LABEL L4:
if NOT (R9) GOTO L5
MOV R10 , 7
MOV c , R10
JUMPtoLOOP L4

LABEL L5:
JUMPtoLOOP L0

LABEL L1:
MOV R11 , 9
MOV R12 , 10
EQ R13 R11 R12

LABEL L6:
if NOT (R13) GOTO L7
MOV R14 , 9
MOV d , R14
JUMPtoLOOP L6

LABEL L7:
```

INPUT 4:

Function Calls

```
pani sayhello(sankhya a){
    chupi("hello ",a);
    ivvu;
}
pani successor(sankhya x){
    chupi("successor of ", x, " is ", x+1);
    sankhya b;
    x=b*x+b;
    ivvu;
}

sayhello(1);
successor(2);
successor(6);
sayhello(4);

```

OUTPUT 4:

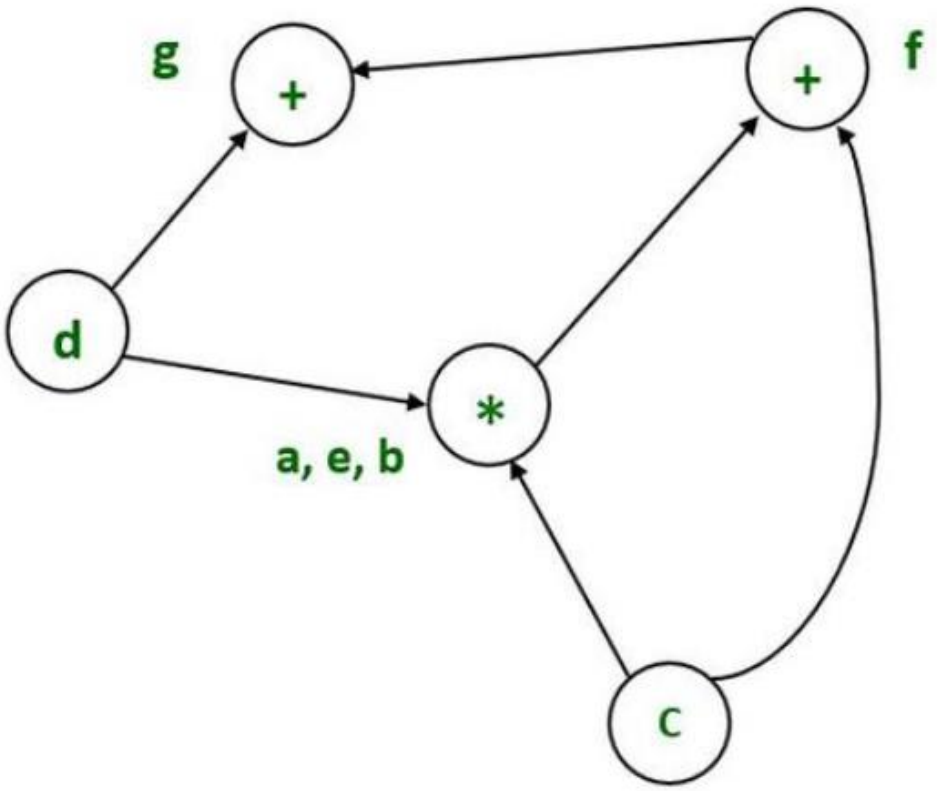
```
LABEL L0:
MOV R0 , "hello "
MOV R1 , a

LABEL L1:
MOV R2 , "successor of "
MOV R3 , x
MOV R4 , " is "
MOV R5 , x
MOV R6 , 1
ADD R6 , R5
MOV R7 , b
MOV R8 , x
MOV R9 , b
ADD R9 , R8
MUL R9 , R7
x = R9
PARAM 1
CALL sayhello
PARAM 2
CALL successor
PARAM 6
CALL successor
PARAM 4
CALL sayhello
```

DIRECTED ACYCLIC GRAPHS for Sample TELUGU programs

Input 1:

```
a = b x c
d = b
e = d x c
b = e
f = b + c
g = f + d
```

Output 1:



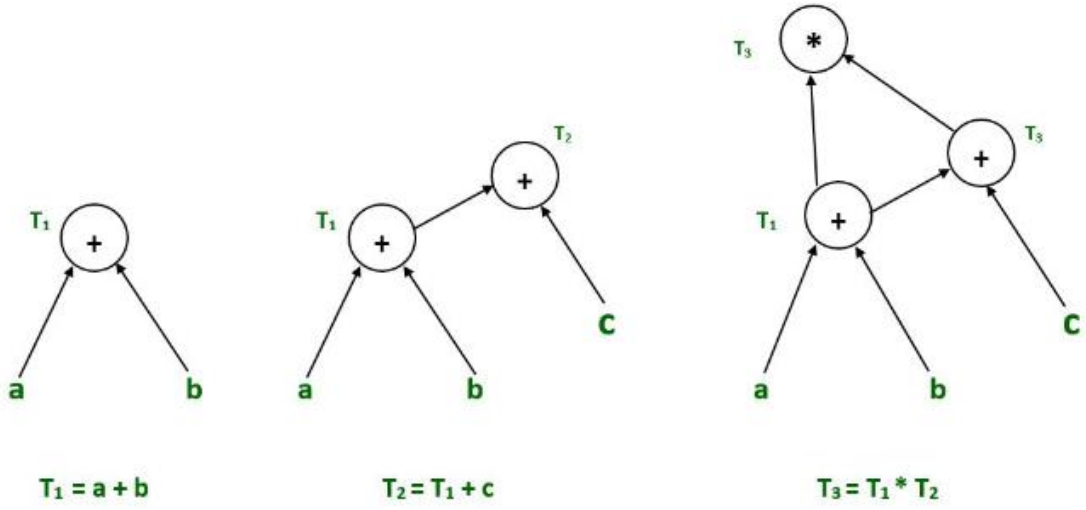
Input 2:

$T_1 = a + b$

$T_2 = T1 + c$

$T_3 = T1 \times T2$

Output 2:



Input 3:

$T_1 = a + b$

$T_2 = a - b$

$T_3 = T_1 * T_2$

$T_4 = T_1 - T_3$

$T_5 = T_4 + T_3$

Output 3: