Updates in parser code:

t2parser.y

```
%{
    /*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 *);
    int check_types(char *, char *);
    char *get_type(char *);

    struct dataType {
        char * id_name;
        char * data_type;
        char * type;
        int line_no;
        int thisscope;
        int num_params;
    } symbol_table[40];

    int count=0;
    int q;
    char type[10];
    extern int countn;
    extern int scope;
    int curr_num_params=0;
    int curr_num_args=0;
```

```
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 {$$.nd = mknode(NULL, NULL, $1.name);add('c');}
telugu_datatype:
    TELUGU_DATATYPE {insert_type();$$.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,exp_type_empty); //this is not working
    } // 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);}
---
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;
                    $$.nd = mknode(num_children, children, "INT");
                }
|   telugu_float {
                    if(strcmp(exp_type," ")==0) {
                        strcpy(exp_type, "thelu");
                    }
                    else if(strcmp(exp_type, "theega")==0) {
                        sprintf(errors[sem_errors], "Line %d: operation among float and string in expression not allowed\n", countn+1);
                        sem_errors++;
                    }
                    printf("Parser found float\n");
                    int num_children = 1;  // Number of children
                    struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
                    children[0] = $1.nd;
                    $$.nd = mknode(num_children, children, "FLOAT");
                }
|   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");
                }
|   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);
```

```
327                 sem_errors++;
328             }
329             printf("Parser found string\n");
330             int num_children = 1;  // Number of children
331             struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
332             children[0] = $1.nd;
333             $$.nd = mknode(num_children, children, "STRING");
334         }
```

Updates in lexical analyser code:

```
12    extern int countn;  // replaced new_line count
13    extern int scope=1;  // starts from 1
```

```
642 variable_declaration:
643     telugu_datatype telugu_identifier_declaring telugu_assignment_operator exp {
644         //add('V');   // this is taking ';' as a variable
645         printf("Parser found datatypeId=exp\n");
646         int num_children = 4;  // Number of children
647         struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
648         children[0] = $1.nd;
649         children[1] = $2.nd;
650         children[2] = $3.nd;
651         children[3] = $4.nd;
652         $$.nd = mknode(num_children, children, "datatypeId=exp");
653         if(strcmp($1.name , exp_type)){
654             sprintf(errors[sem_errors], "Line %d: Data type casting not allowed in declaration\n", countn);
655             sem_errors++;
656         }
657     }
681 parameters_repeat: // can be empty 0 or more occurences
682     { $$.nd = mknode(NULL, NULL, "empty"); }
683 |   parameters_repeat telugu_datatype telugu_identifier_declaring telugu_punctuation_comma {
684         printf("Parser found paramRepDatatypeIdComma\n");
685         curr_num_params++;
686         int num_children = 4;  // Number of children
687         struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
688         children[0] = $1.nd;
689         children[1] = $2.nd;
690         children[2] = $3.nd;
691         children[3] = $4.nd;
692         $$.nd = mknode(num_children, children, "paramRepDatatypeIdComma");
693     }
694
695 parameters_line: // can be empty
696     { $$.nd = mknode(NULL, NULL, "empty"); }
697 |   parameters_repeat telugu_datatype telugu_identifier_declaring {
698         printf("Parser found parameters_line\n");
699         curr_num_params++;
700         int num_children = 3;  // Number of children
701         struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
702         children[0] = $1.nd;
703         children[1] = $2.nd;
704         children[2] = $3.nd;
705         $$.nd = mknode(num_children, children, "paramLine");
706     }
707
708 identifiers_repeat: // abc,x,y,p  can be empty
709     { $$.nd = mknode(NULL, NULL, "empty"); }
710 |   telugu_identifier {
711         curr_num_args++;
712         printf("Parser found lastparam\n");
713         int num_children = 1;  // Number of children
714         struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
715         children[0] = $1.nd;
716         $$.nd = mknode(num_children, children, "paramEnd");
717     }
718 |   telugu_constant {
719         curr_num_args++;
720         printf("Parser found lastparam\n");
721         int num_children = 1;  // Number of children
722         struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
723         children[0] = $1.nd;
724         $$.nd = mknode(num_children, children, "paramEnd");
725     }
```

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

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
        if(strcmp(get_type($1.name) , exp_type)){
            sprintf(errors[sem_errors], "Line %d: Data type casting not allowed in equation\n", countn);
            sem_errors++;
        }
    }
```

```
function_declaration:
    telugu_function telugu_function_name { add('F'); } 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] = $2.nd;
        children[2] = $4.nd;
        children[3] = $5.nd;
        children[4] = $6.nd;
        children[5] = $7.nd;
        children[6] = $8.nd;
        children[7] = $9.nd;
        $$.nd = mknode(num_children, children, "func-id-(param){content}");
        symbol_table[count-curr_num_params-1].num_params= curr_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!=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;
    }
```

```
990  ////////////////////////////////////  SYMBOL TABLE & SEMANTIC ANALYSIS PART
991
992  int search(char *type) {
993      int i;
994      for(i=count-1; i>=0; i--) {
995          if(strcmp(symbol_table[i].id_name, type)==0) {
996              return symbol_table[i].thisscope;
997              break;
998          }
999      }
1000     return 0;
1001 }
1002
1003 void check_declaration(char *c) {
1004     q = search(c);
1005     if(!q) {
1006         sprintf(errors[sem_errors], "Line %d: Variable \"%s\" not declared before usage!\n", countn+1, c);
1007         sem_errors++;
1008     }
1009 }
1010
1011 char *get_type(char *var){
1012     for(int i=0; i<count; i++) {
1013         // Handle case of use before declaration
1014         if(!strcmp(symbol_table[i].id_name, var)) {
1015             return symbol_table[i].type;
1016         }
1017     }
1018 }
1019
1020 void add(char c) {
1021     if(c == 'V'){ // variable
1022         for(int i=0; i<reserved_count; i++){
1023             if(!strcmp(reserved[i], strdup(yy_text))){
1024                 sprintf(errors[sem_errors], "Line %d: Variable name \"%s\" is a reserved keyword!\n", countn+1, yy_text);
1025                 sem_errors++;
1026                 return;
1027             }
1028         }
1029     }
```

```
1030        q=search(yy_text);
1031        if(!q) {   // insert into symbol table only if not already present
1032            if(c == 'H') {  //header
1033                symbol_table[count].id_name=strdup(yy_text);
1034                symbol_table[count].data_type=strdup(type);
1035                symbol_table[count].line_no=countn;
1036                symbol_table[count].type=strdup("Header");
1037                symbol_table[count].thisscope=scope;
1038                symbol_table[count].num_params=0;
1039                count++;
1040            }
1041            else if(c == 'K') { //keyword
1042                symbol_table[count].id_name=strdup(yy_text);
1043                symbol_table[count].data_type=strdup("N/A");
1044                symbol_table[count].line_no=countn;
1045                symbol_table[count].type=strdup("Keyword\t");
1046                symbol_table[count].thisscope=scope;
1047                symbol_table[count].num_params=0;
1048                count++;
1049            }
1050            else if(c == 'V') { //variable
1051                printf("yytext: %s\n", yy_text);
1052                symbol_table[count].id_name=strdup(yy_text);
1053                symbol_table[count].data_type=strdup(type);
1054                symbol_table[count].line_no=countn;
1055                symbol_table[count].type=strdup("Variable");
1056                symbol_table[count].thisscope=scope;
1057                symbol_table[count].num_params=0;
1058                count++;
1059            }
1060            else if(c == 'C') {  //constant sankhya
1061                symbol_table[count].id_name=strdup(yy_text);
1062                symbol_table[count].data_type=strdup("CONST");
1063                symbol_table[count].line_no=countn;
1064                symbol_table[count].type=strdup("constantx");
1065                symbol_table[count].thisscope=scope;
1066                symbol_table[count].num_params=0;
1067                count++;
1068            }
1069            else if(c == 'i') {  //constant sankhya
1070                symbol_table[count].id_name=strdup(yy_text);
1071                symbol_table[count].data_type=strdup("CONST");
1072                symbol_table[count].line_no=countn;
1073                symbol_table[count].type=strdup("sankhya");
1074                symbol_table[count].thisscope=scope;
```

```
1075            symbol_table[count].num_params=0;
1076            count++;
1077        }
1078        else if(c == 'f') {  //constant float thelu
1079            symbol_table[count].id_name=strdup(yy_text);
1080            symbol_table[count].data_type=strdup("CONST");
1081            symbol_table[count].line_no=countn;
1082            symbol_table[count].type=strdup("thelu");
1083            symbol_table[count].thisscope=scope;
1084            symbol_table[count].num_params=0;
1085            count++;
1086        }
1087        else if(c == 'c') {  //constant character aksharam
1088            symbol_table[count].id_name=strdup(yy_text);
1089            symbol_table[count].data_type=strdup("CONST");
1090            symbol_table[count].line_no=countn;
1091            symbol_table[count].type=strdup("aksharam");
1092            symbol_table[count].thisscope=scope;
1093            symbol_table[count].num_params=0;
1094            count++;
1095        }
1096        else if(c == 's') {  //constant string theega
1097            symbol_table[count].id_name=strdup(yy_text);
1098            symbol_table[count].data_type=strdup("CONST");
1099            symbol_table[count].line_no=countn;
1100            symbol_table[count].type=strdup("theega");
1101            symbol_table[count].thisscope=scope;
1102            symbol_table[count].num_params=0;
1103            count++;
1104        }
1105        else if(c == 'F') {
1106            symbol_table[count].id_name=strdup(yy_text);
1107            symbol_table[count].data_type=strdup(type);
1108            symbol_table[count].line_no=countn;
1109            symbol_table[count].type=strdup("Function");
1110            symbol_table[count].thisscope=scope;
1111            printf("\nSETTING %s's params to %d\n", symbol_table[count-curr_num_params].id_name, curr_num_params);
1112            symbol_table[count-curr_num_params].num_params=curr_num_params;
1113            curr_num_params=0;
1114            count++;
1115        }
1116        else if(c == 'L') {
1117            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);
}
```

SAMPLE INPUT:

input3.txt

```
thechko numpy;
sankhya a=10;
a=a-5;
theega y="sdf";
y=y+"scf";
okavela(a peddadi 2){
    sankhya x=5;
    x=x+2;
}
```

SAMPLE OUTPUT:

```
PS C:\Users\Karthik Chittoor\Desktop\compiler design lab> Get-Content input3.txt | my_program.exe

                                                                PHASE 1: LEXICAL ANALYSIS

thechko is import keyword (line: 1, column: 1)
numpy is an identifier (line: 1, column: 9)
; is the end of a statement (line: 1, column: 14)

 is a new line (line: 1)
sankhya is a data type (line: 2, column: 1)
Parser found eol-input
Parser found import-lib-;-input
a is an identifier (line: 2, column: 9)
saw varDeclareidyytext: a
= is an assignment operator (line: 2, column: 10)
10 is a  integer3 (line: 2, column: 11)
Parser found int
; is the end of a statement (line: 2, column: 11)
Parser found datatypeId=exp

 is a new line (line: 2)
a is an identifier (line: 3, column: 1)
CHECKING FOR a
saw pure id2= is an assignment operator (line: 3, column: 2)
a is an identifier (line: 3, column: 3)
CHECKING FOR a
saw pure id2 - is an arithmetic operator(line: 3, column: 5)
Parser found identifier
5 is an  integer1 (line: 3, column: 4)
Parser found int
; is the end of a statement (line: 3, column: 4)
Parser found exp-arithmeticOp-exp
Parser found equation
type of identifier: Variable XXXXXXXXXXXXXXXX exp_type=sankhya


 is a new line (line: 3)
theega is a data type (line: 4, column: 1)
y is an identifier (line: 4, column: 8)
saw varDeclareidyytext: y
= is an assignment operator (line: 4, column: 9)
"sdf" is a string (line: 4, column: 10)
Parser found string
; is the end of a statement (line: 4, column: 15)
```

```
Parser found datatypeId=exp

 is a new line (line: 4)
y is an identifier (line: 5, column: 1)
CHECKING FOR y
saw pure id2= is an assignment operator (line: 5, column: 2)
y is an identifier (line: 5, column: 3)
CHECKING FOR y
saw pure id2+ is an arithmatic operator (line: 5, column: 4)
Parser found identifier
"scf" is a string (line: 5, column: 5)
Parser found string
; is the end of a statement (line: 5, column: 10)
Parser found exp-arithmeticOp-exp
Parser found equation
type of identifier: Variable XXXXXXXXXXXXXXXX exp_type=theega


 is a new line (line: 5)
okavela is a telugu_if statement (line: 6, column: 1)
( is an open curly bracket (line: 6, column: 8)
a is an identifier (line: 6, column: 9)
CHECKING FOR a
saw pure id2peddadi is a comparison operator (line: 6, column: 11)
Parser found identifier
2 is an  integer1 (line: 6, column: 19)
Parser found int
) is an closed curly bracket (line: 6, column: 19)
Parser found exp-compareOp-exp
{ is an open flower bracket (line: 6, column: 20)

 is a new line (line: 6)
sankhya is a data type (line: 7, column: 5)
x is an identifier (line: 7, column: 13)
saw varDeclareidyytext: x
= is an assignment operator (line: 7, column: 14)
5 is a  integer3 (line: 7, column: 15)
Parser found int
; is the end of a statement (line: 7, column: 15)
```

```
 is a new line (line: 7)
x is an identifier (line: 8, column: 5)
CHECKING FOR x
saw pure id2= is an assignment operator (line: 8, column: 6)
x is an identifier (line: 8, column: 7)
CHECKING FOR x
saw pure id2 + is an arithmetic operator(line: 8, column: 9)
Parser found identifier
2 is an  integer1 (line: 8, column: 8)
Parser found int
; is the end of a statement (line: 8, column: 8)
Parser found exp-arithmeticOp-exp
Parser found equation
type of identifier: Variable XXXXXXXXXXXXXXXX exp_type=sankhya


 is a new line (line: 8)
} is an closed flower bracket (line: 9, column: 1)
Parser found EOL-bunch
Parser found bunch-equation-finish
Parser found EOL-bunch
Parser found bunch-varDeclare-finish
Parser found EOL-bunch

ERASING x from symbol table as its CURRENT SCOPE is FINISHED
Parser found if(cond){bunch}

 is a new line (line: 9)

 is a new line (line: 10)
Parser found eol elif_repeat
Parser found eol elif_repeat
Parser found ifElseLadder
Parser found bunch_of_statement if_else_ladder bunch
Parser found EOL-bunch
Parser found bunch-equation-finish
Parser found EOL-bunch
Parser found bunch-varDeclare-finish
Parser found EOL-bunch
Parser found bunch-equation-finish
Parser found EOL-bunch
Parser found bunch-varDeclare-finish
Parser found input-bunch_of_stmts-input
```

SYMBOL TABLE:

```
SYMBOL    DATATYPE    TYPE    LineNUMBER  SCOPE  numParams
_____________________________________________

numpy           Library 1       1       0
a       sankhya Variable        2       1       0
10      CONST   sankhya 2        1       0
5       CONST   sankhya 3        1       0
y       theega  Variable        4       1       0
"sdf"   CONST   theega  4       1       0
"scf"   CONST   theega  5       1       0
okavela N/A     Keyword         6       1       0
2       CONST   sankhya 6       1       0
x       sankhya Variable        7       2147483647      0
```

```
                                PHASE 2: SYNTAX ANALYSIS


 Inorder traversal of the Parse Tree:

program, input-bunch-input, import-lib-;-input, thechko, numpy, ;, eol-input, newline, empty, bunch-varDeclare-;-bunch, em
pty, datatypeId=exp, sankhya, a, =, INT, 10, ;, eol-bunch, newline, bunch-equation-;-bunch, empty, id=exp, a, =, Athematic
Op, ID, a, -5;
theega y="sdf";
y=y+"scf";
okavela(a peddadi 2){
    sankhya x=5;
    x=x+2;
}

, INT, 5, ;, eol-bunch, newline, bunch-varDeclare-;-bunch, empty, datatypeId=exp, theega, y, =, STRING, "sdf", ;, eol-bunc
h, newline, bunch-equation-;-bunch, empty, id=exp, y, =, AthematicOp, ID, y, +, STRING, "scf", ;, eol-bunch, newline, bunc
h-IfElse-bunch, empty, ifElseLadder, if(cond){bunch}, if, (, condition, ID, a, peddadi, INT, 2, ), {, eol-bunch, newline,
bunch-varDeclare-;-bunch, empty, datatypeId=exp, sankhya, x, =, INT, 5, ;, eol-bunch, newline, bunch-equation-;-bunch, emp
ty, id=exp, x, =, AthematicOp, ID, x, +2;
}

, INT, 2, ;, eol-bunch, newline, empty, }, EOL-elifrepeat, newline, EOL-elifrepeat, newline, empty, empty, empty, empty,


                                                PHASE 3: SEMANTIC ANALYSIS

Semantic analysis completed with no errors
```

SEMANTIC ERRORS:

1.Multiple Declarations of Variables/Functions:

On encountering an identifier in **variable_declaration** production, the semantic analyser looks up in the symbol table entries to check if any symbol is already declared with **symbol_table[i].id_name = $$.name**

Input:

```
1    sankhya a=3;
2    sankhya a=15;
```

Output:

```
                                                    PHASE 1: LEXICAL ANALYSIS

sankhya is a data type (line: 1, column: 1)
a is an identifier (line: 1, column: 9)
saw varDeclareidyytext: a
= is an assignment operator (line: 1, column: 10)
3 is a  integer3 (line: 1, column: 11)
Parser found int
; is the end of a statement (line: 1, column: 11)
Parser found datatypeId=exp

 is a new line (line: 1)
sankhya is a data type (line: 2, column: 1)
a is an identifier (line: 2, column: 9)
saw varDeclareid= is an assignment operator (line: 2, column: 10)
15 is a  integer3 (line: 2, column: 11)
Parser found int
; is the end of a statement (line: 2, column: 11)
Parser found datatypeId=exp

 is a new line (line: 2)

 is a new line (line: 3)
Parser found EOL-bunch
Parser found EOL-bunch
Parser found bunch-varDeclare-finish
Parser found EOL-bunch
Parser found bunch-varDeclare-finish
Parser found input-bunch_of_stmts-input

SYMBOL   DATATYPE   TYPE   LineNUMBER  SCOPE  numParams
_____________________________________

a       sankhya Variable        1       1       0
3       CONST   sankhya 1       1       0

                                                    PHASE 2: SYNTAX ANALYSIS


 Inorder traversal of the Parse Tree:

program, input-bunch-input, empty, bunch-varDeclare-;-bunch, empty, datatypeId=exp, sankhya, a, =, INT, 3, ;, eol-bunch, n
ewline, bunch-varDeclare-;-bunch, empty, datatypeId=exp, sankhya, a, =, INT, 15, ;, eol-bunch, newline, eol-bunch, newline
, empty, empty,


                                                    PHASE 3: SEMANTIC ANALYSIS

Semantic analysis completed with 1 errors
        - Line 3: Multiple declarations of "a" not allowed!
```

2.UNDECLARED VARIABLES:

On encountering an identifier in **exp** or **equation** or **bunch_of_statements** productions, the semantic analyser looks up in the symbol table entries to check if this symbol is already declared with **symbol_table[i].id_name = $$.name**

Input:

```
1    sankhya b=a;
```

Output:

```
                                                    PHASE 3: SEMANTIC ANALYSIS

Semantic analysis completed with 1 errors
        - Line 2: Variable "a" not declared before usage!
```

3. TYPECASTING:

When parsing **equation/variable_declaration** , the semantic analyser evaluates the data type of the resulting expression in RHS of the equation. If RHS's datatypes does not match the datatype of the LHS (looks up in the symbol table for this), then semantic error is reported.

Input:

```
1    sankhya a;
2    a="hello trichy!";
```

Output:

```
SYMBOL    DATATYPE    TYPE    LineNUMBER  SCOPE  numParams
_____________________________________

a       sankhya Variable        1       1       0
"hello trichy!" CONST    theega  2       1       0


                                                                PHASE 2: SYNTAX ANALYSIS


 Inorder traversal of the Parse Tree:

program, input-bunch-input, empty, bunch-varDeclare-;-bunch, empty, datatypeId, sankhya, a, ;, eol-bunch, newline, bunch-e
quation-;-bunch, empty, id=exp, a, =, STRING, "hello trichy!", ;, eol-bunch, newline, empty, empty,


                                                                PHASE 3: SEMANTIC ANALYSIS

Semantic analysis completed with 1 errors
        - Line 2: Data type casting not allowed in equation
```

4.INVALID OPERATIONS ON DATATYPES:

The semantic analyser is designed to report invalid operations on certain datatypes associated with typecasting.

Examples: string + string //should be allowed because its concatenation

string – string // not allowed as '-' is not defined for strings

Input:

```
1    theega s="hello"+54;
```

Output:

```
SYMBOL    DATATYPE    TYPE    LineNUMBER  SCOPE   numParams
_____________________________________________

s       theega  Variable        1       1       0
"hello" CONST   theega  1       1       0
54      CONST   sankhya 1       1       0
```

```
                                                    PHASE 2: SYNTAX ANALYSIS


 Inorder traversal of the Parse Tree:

program, input-bunch-input, empty, bunch-varDeclare-;-bunch, empty, datatypeId=exp, theega, s, =, AthematicOp, STRING, "he
llo", +54;
, INT, 54, ;, eol-bunch, newline, empty, empty,
```

```
                                                    PHASE 3: SEMANTIC ANALYSIS

Semantic analysis completed with 1 errors
        - Line 2: operation among int and string in expression not allowed
```

5. SCOPE OF VARIABLES:

The symbol table stores the scope of each variable as an integer. Starting from 1, the scope increases by 1 on seeing a '{' and decreases by 1 on seeing a '}'.

Lower scope value -> higher the globality of that variable

All the variables within {} will be marked as out of scope/destroyed by changing their scope to +infinity. Now any variable which is already defined can be defined again in another scope after verifying its scope=infinity.

Input 1:

```
1   sankhya a=3;
2   okavela(a samanam 5){
3       sankhya a;
4       okavela(a chinnadi 6){
5           sankhya a=7;
6       }
7   }
8
```

Output 1:

```
SYMBOL    DATATYPE    TYPE    LineNUMBER  SCOPE  numParams
_____________________________________________

a       sankhya Variable        1       1       0
3       CONST   sankhya 1        1       0
okavela N/A     Keyword         2       1       0
5       CONST   sankhya 2        1       0
6       CONST   sankhya 4        2       0
7       CONST   sankhya 5        3       0
```

```
                                                PHASE 2: SYNTAX ANALYSIS


 Inorder traversal of the Parse Tree:

program, input-bunch-input, empty, bunch-varDeclare-;-bunch, empty, datatypeId=exp, sankhya, a, =, INT, 3, ;, eol-bunch, n
ewline, bunch-IfElse-bunch, empty, ifElseLadder, if(cond){bunch}, if, (, condition, ID, a, samanam, INT, 5, ), {, eol-bunc
h, newline, bunch-varDeclare-;-bunch, empty, datatypeId, sankhya, a, ;, eol-bunch, newline, bunch-IfElse-bunch, empty, ifE
lseLadder, if(cond){bunch}, if, (, condition, ID, a, chinnadi, INT, 6, ), {, eol-bunch, newline, bunch-varDeclare-;-bunch,
 empty, datatypeId=exp, sankhya, a, =, INT, 7, ;, eol-bunch, newline, empty, }, EOL-elifrepeat, newline, empty, empty, emp
ty, }, EOL-elifrepeat, newline, empty, empty, empty, empty,
```

```
                                                PHASE 3: SEMANTIC ANALYSIS

Semantic analysis completed with 2 errors
        - Line 4: Multiple declarations of "a" not allowed!
        - Line 6: Multiple declarations of "a" not allowed!
```

Input 2: // in this input scope of the variable is only within the ‘{}’ . so we are allowed to declare the same sankhya multiple times without triggering semantic errors.

```
1   sankhya a=3;
2   okavela(a samanam 5){
3       sankhya b;
4   }
5   lekaokavela(a samanam 6){
6       sankhya b;
7   }
8   lekapothe{
9       sankhya b;
10  }
```

Output2:

```
SYMBOL    DATATYPE    TYPE    LineNUMBER  SCOPE  numParams
_____________________________________________

a         sankhya Variable          1         1         0
3         CONST     sankhya 1         1         0
okavela   N/A       Keyword           2         1         0
5         CONST     sankhya 2         1         0
b         sankhya Variable          3         2147483647        0
lekaokavela       N/A       Keyword           5         1         0
6         CONST     sankhya 5         1         0
lekapothe         N/A       Keyword           8         1         0
```

PHASE 2: SYNTAX ANALYSIS

Inorder traversal of the Parse Tree:

```
program, input-bunch-input, empty, bunch-varDeclare-;-bunch, empty, datatypeId=exp, sankhya, a, =, INT, 3, ;, eol-bunch, n
ewline, bunch-IfElse-bunch, empty, ifElseLadder, if(cond){bunch}, if, (, condition, ID, a, samanam, INT, 5, ), {, eol-bunc
h, newline, bunch-varDeclare-;-bunch, empty, datatypeId, sankhya, b, ;, eol-bunch, newline, empty, }, EOL-elifrepeat, newl
ine, elif(cond){bunch}, empty, elif, (, condition, ID, a, samanam, INT, 6, ), {, eol-bunch, newline, bunch-varDeclare-;-bu
nch, empty, datatypeId, sankhya, b, ;, eol-bunch, newline, empty, }, EOL-elifrepeat, newline, empty, else{bunch}, else, {,
 eol-bunch, newline, bunch-varDeclare-;-bunch, empty, datatypeId, sankhya, b, ;, eol-bunch, newline, empty, }, eol-bunch,
newline, empty, empty,
```

PHASE 3: SEMANTIC ANALYSIS

```
Semantic analysis completed with no errors
```