CMPE 230 Systems Programming
Homework 1 Documentation

Volkan Öztürk & Arda Arslan
2019400033 2020400078

Bogazici University, Engineering Faculty
Computer Engineering Department

In this project we have implemented a translator for a language called MatLang that translates MatLang code to C language code. Mainly five tasks are accomplished within the program:

1) Parsing
2) Syntax Check
3) BNF Implementation
4) Attribute Grammar Check
5) C Recognizable Code Generation

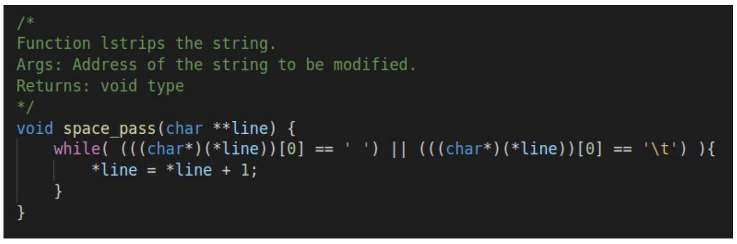
IMPORTANT REMARKS

- **-lm command is required to link <math.h>**
- **Line counter variable also counts empty lines.**

Parsing

a)Frequently used code segments and functions:

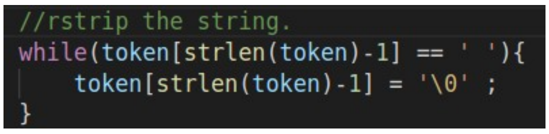
1.) space_pass(char **line) function:



```c
/*
Function lstrips the string.
Args: Address of the string to be modified.
Returns: void type
*/
void space_pass(char **line) {
    while( (((char*)(*line))[0] == ' ') || (((char*)(*line))[0] == '\t') ){
        *line = *line + 1;
    }
}
```

Pass by reference function that implements the functionality of lstrip with continuously incrementing pointer's value by one if the dereferenced value is whitespace.

2.) Code segment used for rstrip functionality:



```c
//rstrip the string.
while(token[strlen(token)-1] == ' '){
    token[strlen(token)-1] = '\0' ;
}
```

Continuously puts empty string characters to the rightmost character of the string if it's a whitespace.

3.) strsep() function*:

For separating string with the specified delimiter.

4.) strchr() function*:

For searching a character in a string.

5.) strstr() function*:

For searching a substring in a string.

6.) strscmp() function*:

To compare two strings by value.

*functions included with <string.h>

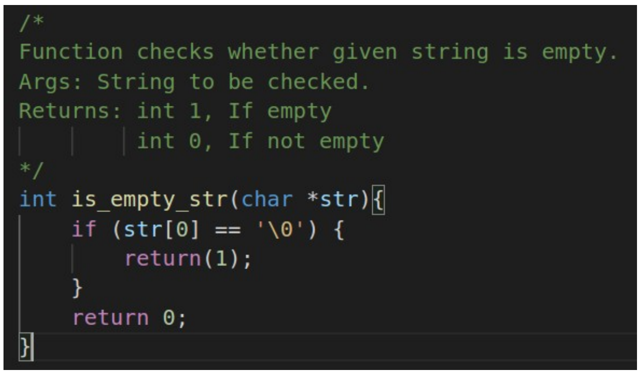
Syntax Check

a)General Approach:

After reading and parsing the line, at first, "printsep" or "print" keywords are searched. Then assignment type statements are checked with the help of special character "=". After that, scalar, vector and matrix definition(declaration) statements are handled with related keyword. Lastly for block headers are checked. These checks comply with the syntax of MatLang language.

b)Frequently used code segments and functions:

1.) is_empty_str(char *str):



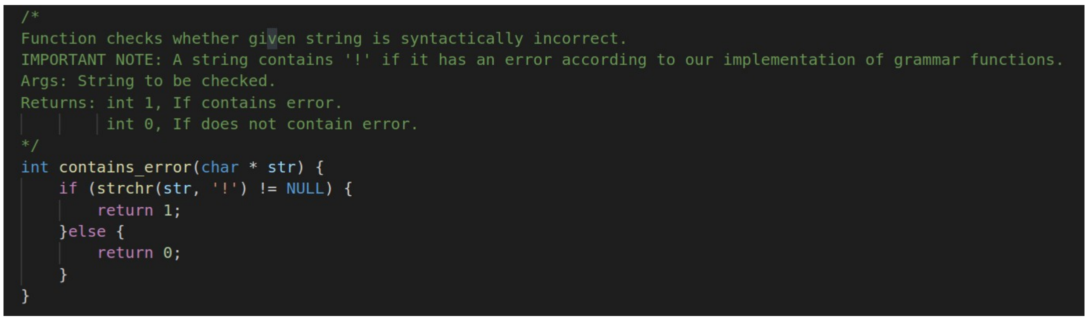
```c
/*
Function checks whether given string is empty.
Args: String to be checked.
Returns: int 1, If empty
         int 0, If not empty
*/
int is_empty_str(char *str){
    if (str[0] == '\0') {
        return(1);
    }
    return 0;
}
```

Checks whether specified string is empty.

2.) is_not_empty_str(char *str):

Checks whether specified string is not empty. Similar to functionality of is_empty_str().

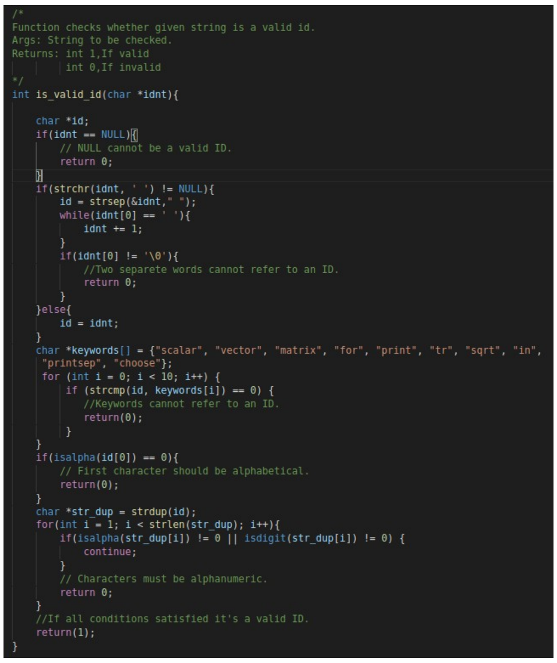
3.) contains_error(char *str):



```c
/*
Function checks whether given string is syntactically incorrect.
IMPORTANT NOTE: A string contains '!' if it has an error according to our implementation of grammar functions.
Args: String to be checked.
Returns: int 1, If contains error.
         int 0, If does not contain error.
*/
int contains_error(char * str) {
    if (strchr(str, '!') != NULL) {
        return 1;
    }else {
        return 0;
    }
}
```

In our implementation of BNF we put '!' character to return value of BNF functions to depict specified string is syntactically does not comply MatLang language rules.

4.) is_valid_id(char *idnt):



```c
/*
Function checks whether given string is a valid id.
Args: String to be checked.
Returns: int 1,If valid
         int 0,If invalid
*/
int is_valid_id(char *idnt){

    char *id;
    if(idnt == NULL){
        // NULL cannot be a valid ID.
        return 0;
    }
    if(strchr(idnt, ' ') != NULL){
        id = strsep(&idnt," ");
        while(idnt[0] == ' '){
            idnt += 1;
        }
        if(idnt[0] != '\0'){
            //Two separete words cannot refer to an ID.
            return 0;
        }
    }else{
        id = idnt;
    }
    char *keywords[] = {"scalar", "vector", "matrix", "for", "print", "tr", "sqrt", "in",
     "printsep", "choose"};
     for (int i = 0; i < 10; i++) {
         if (strcmp(id, keywords[i]) == 0) {
            //Keywords cannot refer to an ID.
            return(0);
         }
    }
    if(isalpha(id[0]) == 0){
        // First character should be alphabetical.
        return(0);
    }
    char *str_dup = strdup(id);
    for(int i = 1; i < strlen(str_dup); i++){
        if(isalpha(str_dup[i]) != 0 || isdigit(str_dup[i]) != 0) {
            continue;
        }
        // Characters must be alphanumeric.
        return 0;
    }
    //If all conditions satisfied it's a valid ID.
    return(1);
}
```

Checks whether given string can be kept as ID of a variable. Possible errors are explained in as comments embedded in code.

<u>BNF Implementation</u>

a) BNF for MatLang language:

```
<expr>   →  <term> + <expr>
            | <term> – <expr>
            | <term>

<term>   →  <factor> * <expr>
            | <factor>

<factor> →  (<expr>)
            | id[<expr>, <expr>]
            | id[<expr>]
            | sqrt(<expr>)
            | choose(<expr>)
            | tr(<expr>)
            | id
            | num
```

b) General approach for expr(char *str), term(char *str), factor(char *str):

1.) expr(char *str):
Specified string is parsed with first occurrence of ‘+’ or ‘-’ signs. Precedence of parantheses and brackets are also implemented within the function and precedence of ‘*’ comes naturally with the implementation of BNF. Part before the operand is sent to term(char *str) and part after the operand is recursively sent to expr(char *str) as argument.

2.) term(char *str):
Specified string is parsed with first occurrence of ‘*’. Precedence of parantheses and brackets are also implemented within the function. Part before the operand is sent to factor(char *str) and part after the operand is recursively sent to term(char *str) as argument.

3.) factor(char *str):
First searches ‘(‘ character in string given as argument. If found, tries to match part of the string before ‘(‘ with possible factor types. Else if searches for ‘[‘ character. If found, tries to match part of the string before ‘[’ with a matrix or vector id. Else tries to match the string with an id of variable or a number.

Attribute Grammar Check

a) Types of Attribute Check:

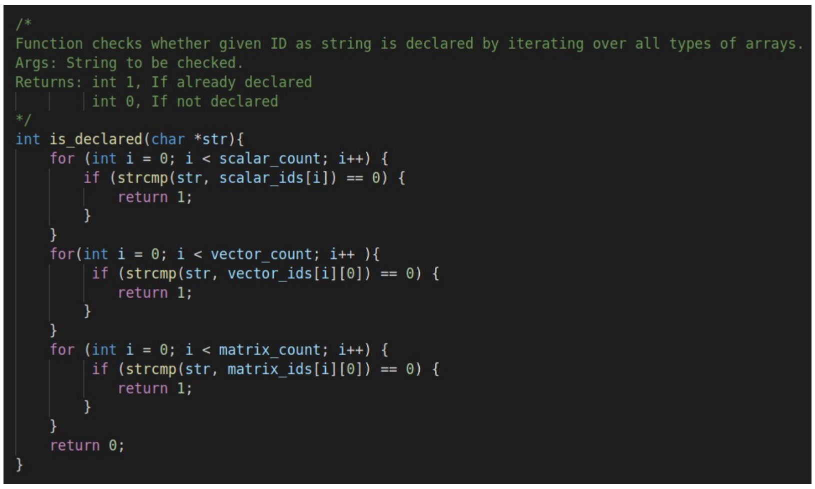
1.) Type Check:
In assignment statements, if left hand side’s type is scalar, right hand side must be of scalar type. Same applies to matrix type (which also includes vectors as one column matrices).

2.) Dimension Check:
In matrix type assignment statements (i.e. lhs and rhs are of matrix type), dimensions must also match. In matrix multiplication operation two matrices must comply the rules for multiplication of Linear Algebra.

3.) Redeclaration:
According to MatLang language specifications an id can be declared only at once. i.e. redeclaration is not allowed.

b) Frequently used functions:

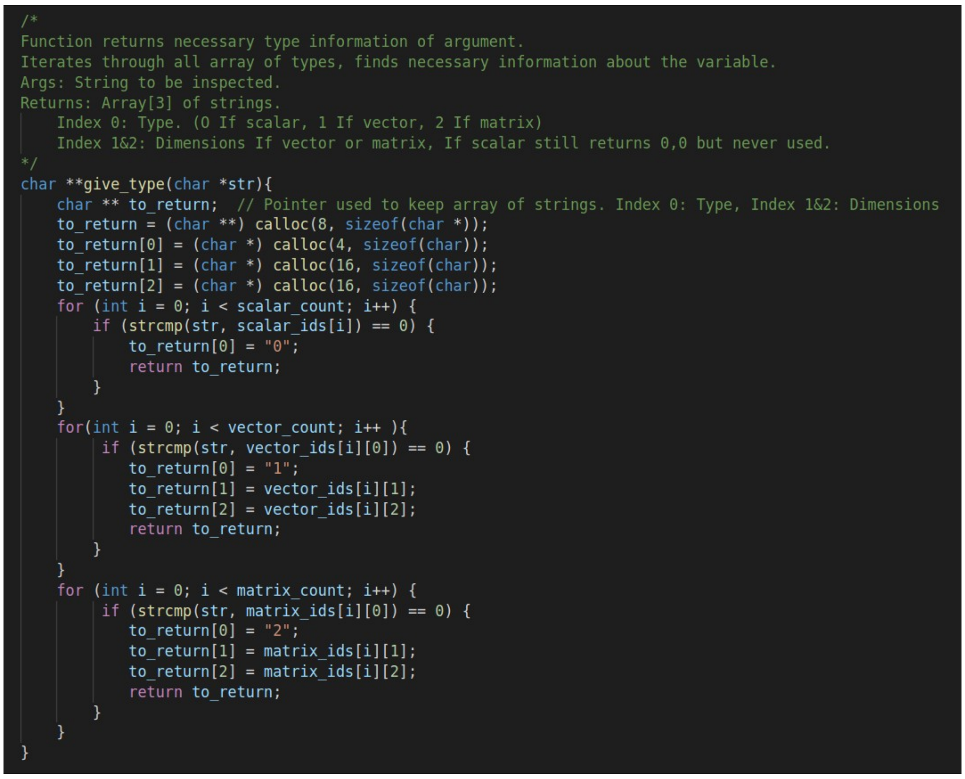
1.)is_declared(char *str):



```c
/*
Function checks whether given ID as string is declared by iterating over all types of arrays.
Args: String to be checked.
Returns: int 1, If already declared
         int 0, If not declared
*/
int is_declared(char *str){
    for (int i = 0; i < scalar_count; i++) {
        if (strcmp(str, scalar_ids[i]) == 0) {
            return 1;
        }
    }
    for(int i = 0; i < vector_count; i++ ){
         if (strcmp(str, vector_ids[i][0]) == 0) {
            return 1;
        }
    }
    for (int i = 0; i < matrix_count; i++) {
         if (strcmp(str, matrix_ids[i][0]) == 0) {
            return 1;
        }
    }
    return 0;
}
```

Checks if given string is previously set as a variable id.

2.) give_type(char *str):



```c
/*
Function returns necessary type information of argument.
Iterates through all array of types, finds necessary information about the variable.
Args: String to be inspected.
Returns: Array[3] of strings.
    Index 0: Type. (0 If scalar, 1 If vector, 2 If matrix)
    Index 1&2: Dimensions If vector or matrix, If scalar still returns 0,0 but never used.
*/
char **give_type(char *str){
    char ** to_return;  // Pointer used to keep array of strings. Index 0: Type, Index 1&2: Dimensions
    to_return = (char **) calloc(8, sizeof(char *));
    to_return[0] = (char *) calloc(4, sizeof(char));
    to_return[1] = (char *) calloc(16, sizeof(char));
    to_return[2] = (char *) calloc(16, sizeof(char));
    for (int i = 0; i < scalar_count; i++) {
        if (strcmp(str, scalar_ids[i]) == 0) {
            to_return[0] = "0";
            return to_return;
        }
    }
    for(int i = 0; i < vector_count; i++ ){
         if (strcmp(str, vector_ids[i][0]) == 0) {
            to_return[0] = "1";
            to_return[1] = vector_ids[i][1];
            to_return[2] = vector_ids[i][2];
            return to_return;
        }
    }
    for (int i = 0; i < matrix_count; i++) {
         if (strcmp(str, matrix_ids[i][0]) == 0) {
            to_return[0] = "2";
            to_return[1] = matrix_ids[i][1];
            to_return[2] = matrix_ids[i][2];
            return to_return;
        }
    }
}
```

For explanation refer to comments above.

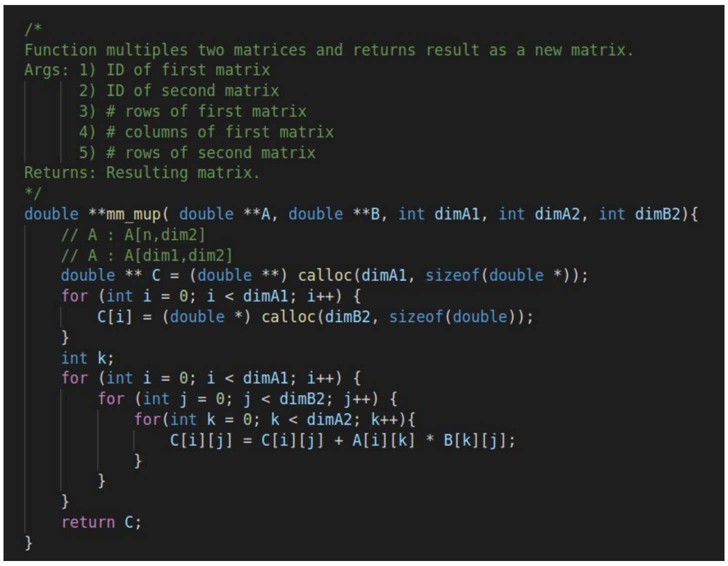
C Recognizable Code Generation

a) General Approach:

BNF functions of expr(char *str), term(char *str) and factor(char *str) returns the given string such a way that it’s tranformed to C language code. Special functions are also added to the line and implemented in the “functions.c” file.

b) Some functions implemented:

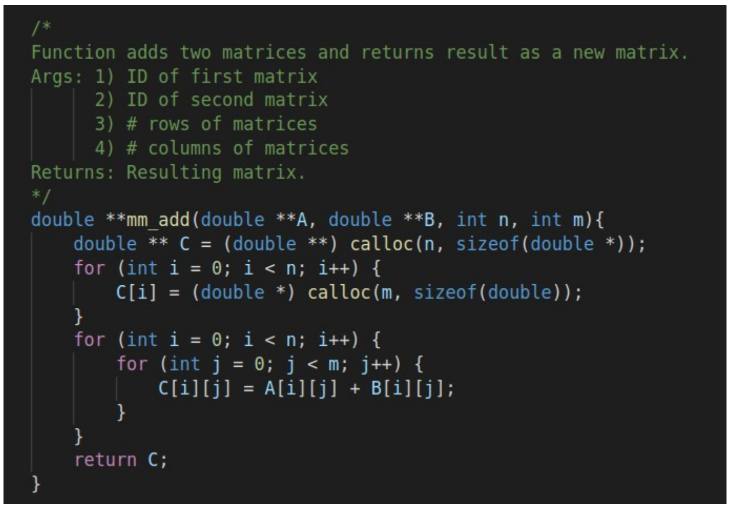
1.) mm_mup:



```c
/*
Function multiples two matrices and returns result as a new matrix.
Args: 1) ID of first matrix
      2) ID of second matrix
      3) # rows of first matrix
      4) # columns of first matrix
      5) # rows of second matrix
Returns: Resulting matrix.
*/
double **mm_mup( double **A, double **B, int dimA1, int dimA2, int dimB2){
    // A : A[n,dim2]
    // A : A[dim1,dim2]
    double ** C = (double **) calloc(dimA1, sizeof(double *));
    for (int i = 0; i < dimA1; i++) {
        C[i] = (double *) calloc(dimB2, sizeof(double));
    }
    int k;
    for (int i = 0; i < dimA1; i++) {
        for (int j = 0; j < dimB2; j++) {
            for(int k = 0; k < dimA2; k++){
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
            }
        }
    }
    return C;
}
```

Array of array of doubles is allocated with calloc and designated to be the resultant matrix. Then filled by iterating over its dimensions.

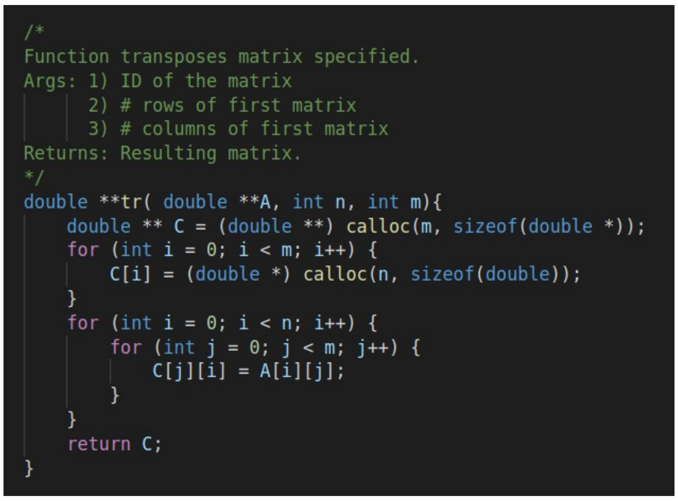
2.) mm_add:



```c
/*
Function adds two matrices and returns result as a new matrix.
Args: 1) ID of first matrix
      2) ID of second matrix
      3) # rows of matrices
      4) # columns of matrices
Returns: Resulting matrix.
*/
double **mm_add(double **A, double **B, int n, int m){
    double ** C = (double **) calloc(n, sizeof(double *));
    for (int i = 0; i < n; i++) {
        C[i] = (double *) calloc(m, sizeof(double));
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
    return C;
}
```

Same logic used in mm_mup applies to construct resultant matrix.

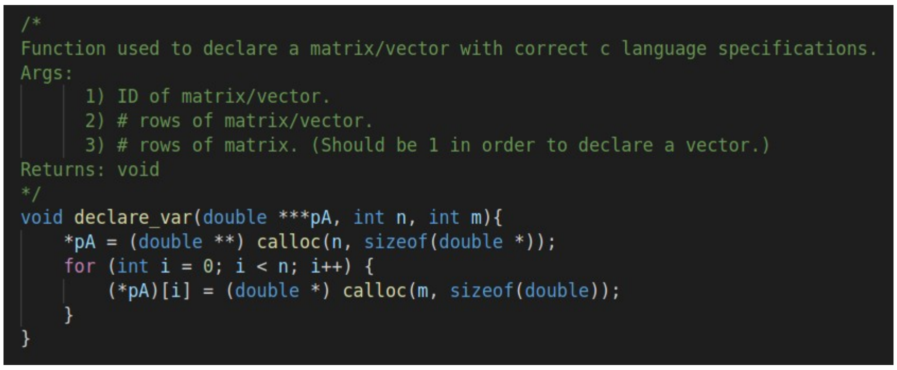
3.) tr:



```c
/*
Function transposes matrix specified.
Args: 1) ID of the matrix
      2) # rows of first matrix
      3) # columns of first matrix
Returns: Resulting matrix.
*/
double **tr( double **A, int n, int m){
    double ** C = (double **) calloc(m, sizeof(double *));
    for (int i = 0; i < m; i++) {
        C[i] = (double *) calloc(n, sizeof(double));
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            C[j][i] = A[i][j];
        }
    }
    return C;
}
```

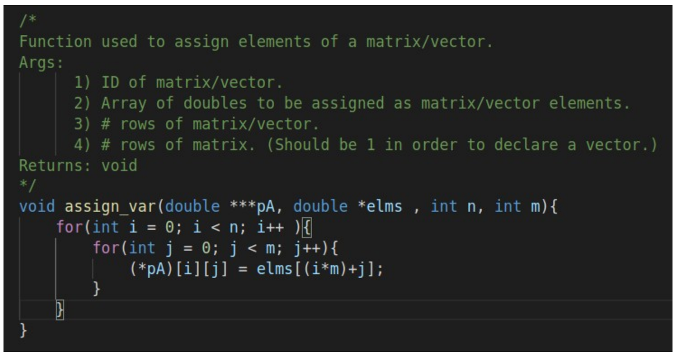
Same logic used in mm_mup applies to construct resultant matrix.

4.) declare_var:



```c
/*
Function used to declare a matrix/vector with correct c language specifications.
Args:
        1) ID of matrix/vector.
        2) # rows of matrix/vector.
        3) # rows of matrix. (Should be 1 in order to declare a vector.)
Returns: void
*/
void declare_var(double ***pA, int n, int m){
    *pA = (double **) calloc(n, sizeof(double *));
    for (int i = 0; i < n; i++) {
        (*pA)[i] = (double *) calloc(m, sizeof(double));
    }
}
```

5.) assign_var:



```c
/*
Function used to assign elements of a matrix/vector.
Args:
        1) ID of matrix/vector.
        2) Array of doubles to be assigned as matrix/vector elements.
        3) # rows of matrix/vector.
        4) # rows of matrix. (Should be 1 in order to declare a vector.)
Returns: void
*/
void assign_var(double ***pA, double *elms , int n, int m){
    for(int i = 0; i < n; i++ ){
        for(int j = 0; j < m; j++){
            (*pA)[i][j] = elms[(i*m)+j];
        }
    }
}
```