# CSE 220: Systems Fundamentals I

Stony Brook University
Homework Assignment #7
FALL 2025

Due: November 14th, 2025, Friday, by 11:59 pm Eastern Time.

**READ THIS:**

- **You are being given approximately two weeks to complete this assignment. Do not squander this time.**
- **This assignment is very important to test your C understanding.**
- **The assignment will be checked for plagiarism or AI tool usage.**
- **The marks of this HW and HW6 will be compared with your marks in the Final Exam.**
    - **Any inconsistency will automatically make HW7 grade zero**

## Learning Outcomes

After completion of this homework assignment, you should be able to:

- Open, read, and close text files.
- Use functions from `stdio.h` to perform file I/O operations.
- Use functions from `string.h` to perform string operations.
- Dynamically allocate and deallocate data structures.

## Download the Template Repository

Accept the invitation to begin the assignment.

https://classroom.github.com/a/2hZouSUT

## Overview

In this assignment, you will be implementing a program to perform simple linear algebra computations: matrix addition, multiplication, and transpose. The matrices and formulas containing these operations will be stored in scripts that your program must parse and execute. You'll need to manage memory carefully, allocating and deallocating data structures as you process the file and evaluate expressions.

# Mathematical Preliminaries

## Matrices

A matrix is a rectangular array of numbers arranged in rows and columns. The size of a matrix is determined by the number of rows and columns it contains. Matrices are widely used in mathematics, physics, engineering, computer science, and many other fields.

## Matrix Addition

Matrix addition is an operation that involves adding the corresponding elements of two matrices of the same size. Mathematically, if A and B are both *m x n* matrices (with *m* rows and *n* columns), then their sum, denoted as C, is an *m x n* matrix where the entry in the i-th row and j-th column of C is given by:

$$\mathbf{C}_{ij} = \mathbf{A}_{ij} + \mathbf{B}_{ij}$$

In this formula, $\mathbf{A}_{ij}$ is the entry in the i-th row and j-th column of matrix A, and $\mathbf{B}_{ij}$ is the entry in the i-th row and j-th column of matrix B. Note that to add two matrices, they must be of the same size, i.e., they must have the same number of rows and the same number of columns.

In our scripts, matrix addition will be denoted **as C = A + B**. Any number of spaces may appear before or after symbols and matrix names in our formulas.

## Matrix Multiplication

Matrix multiplication is a way of combining two matrices to produce a new matrix. The resulting matrix has the same number of rows as the first matrix and the same number of columns as the second matrix.

Mathematically, if A is an *m x n* matrix and B is an *n x p* matrix, then the product of A and B, denoted as C, is an *m x p* matrix where the entry in the i-th row and j-th column of C is given by:

$$\mathbf{C}_{ij} = \Sigma_{(k=1 \text{ to } n)}\, \mathbf{A}_{ik} \times \mathbf{B}_{kj}$$

In this formula, $A_{ik}$ is the entry in the i-th row and k-th column of A, and $B_{kj}$ is the entry in the k-th row and j-th column of B.

In our scripts, matrix multiplication will be denoted C = A*B. Any number of spaces may appear before or after symbols and matrix names in our formulas.

### Matrix Transpose

Matrix transpose is an operation that involves flipping a matrix over its diagonal. Mathematically, if A is an *m x n* matrix, then its transpose, denoted as $A^T$, is an *n x m* matrix where the entry in the i-th row and j-th column of $A^T$ is given by:

$$(A')_{ij} = A_{ji}$$

In other words, the rows of the original matrix become columns in the transposed matrix, and the columns of the original matrix become rows in the transposed matrix.

In our scripts, the transpose of matrix A will be denoted as A'; **(Single quotation mark.)**

## Script Format

Your program will process a script containing commands to define matrices and perform operations on them. Every line of a script has one of two forms:

- a definition of a new matrix from a list of values
- a definition of a new matrix created from a formula containing at least one operator.

A new matrix is defined by providing:

- a single uppercase character
- zero or more spaces
- an equals symbol
- zero or more spaces
- a positive integer containing one or more digits that gives the number of rows of the matrix (call this number NR)
- one or more spaces
- a positive integer containing one or more digits that gives the number of columns of the matrix (call this number NC)
- zero or more spaces
- a left square bracket
- zero or more spaces
- NC integers separated by spaces and terminated by a semicolon. There might or might not be spaces surrounding the semicolon.
- NR-1 additional lists of NC integers as described in the previous item
- a right square bracket
- zero or more spaces
- a newline

Example: `A = 3 2 [4 5; 19 -34 ; 192 -9110; ] \n`

A formula is defined by providing:

- a single uppercase character that has not appeared in the script on a previous line
- zero or more spaces
- an equals symbol
- zero or more spaces
- an expression consisting of + (for addition), * (for multiplication), ' (for transpose), left and right parentheses, and matrix names. The precedence of the operators, from highest to lowest, is:
    - transpose
    - multiplication
    - addition

  Parentheses can be used to change the order of evaluation. An expression may contain spaces anywhere throughout the expression around any of the symbols.
- a newline

Example: `Z = (A +B)' * C * (D' + A)' \n`

Once a symbol has been defined, it cannot be redefined (i.e., reassigned). A symbol can appear multiple times on the right-hand side of a formula.

Scripts will always be syntactically valid. Extra spaces (or no spaces) may surround the *tokens* (e.g., matrix name, equals sign, left and right square brackets, left and right parentheses, semicolons, operators, numbers). When two integers are next to each other, they will be separated by at least one space, or in the case where we have reached the end of a row, a semicolon. There might or might not be spaces around the semicolons.

Scripts will always be semantically valid, meaning that only valid formulas will be provided. For instance, a script will not contain formulas that cannot be evaluated for matrices of the given dimensions.

To summarize: your main task in parsing the text file is NOT to worry about error cases because there will be no error cases. Focus on implementing the order of operations correctly and evaluating the expressions correctly.

# Data Structures

## matrix_sf

The following data type is defined in `hw7.h`, which you will use to implement the functions in this assignment. As its name and members suggest, it represents a matrix of integers:

```
typedef struct {
    char name;
    unsigned int num_rows;
    unsigned int num_cols;
    int values[]; // stores the matrix's contents in row-major order
} matrix_sf;
```

To allocate the memory for a matrix, make a call to `malloc`: [1]

```
matrix_sf *m = malloc(sizeof(matrix_sf)+num_rows*num_cols*sizeof(int));
```

Then you need to assign values to the `name`, `num_rows`, `num_cols`, and `values` members of the struct.

To deallocate the memory for a matrix pointed to by `m`, make a call to `free`:

```
free(m);
```

Do NOT attempt to separately `free()` the `values` array because it is contiguous with the other members of the struct.

## bst_sf

The following data type is also defined in `hw7.h` and represents a node in a binary search tree, where the tree uses the names of the matrices to sort the nodes. Recall that matrix names are unique.

```
typedef struct bst_sf {
    matrix_sf* mat;
    struct bst_sf* left_child;
    struct bst_sf* right_child;
} bst_sf;
```

[1] We discussed this kind of memory allocation when an array of undefined is allocated in a structure and at the end.

This is a simple BST and is not balanced. Don't get fancy or cute here, or else your code will fail many unit tests. Implement a basic, simple BST. `left_child`/`right_child` is `NULL` if a node has no left/right child.

We will use a BST as a symbol table to associate names of matrices appearing in formulas with `matrix_sf` objects. This is explained in more detail later.

## Functions to Implement

You'll most likely want to implement these in the order listed, but jump around if that works better for your workflow.

```
matrix_sf* add_mats_sf(const matrix_sf *mat1, const matrix_sf *mat2)
```

- Perform the matrix addition `mat1+mat2` and return the sum.

```
matrix_sf* mult_mats_sf(const matrix_sf *mat1,
                        const matrix_sf *mat2)
```

- Perform the matrix multiplication `mat1*mat2` and return the product.

```
matrix_sf* transpose_mat_sf(const matrix_sf *mat)
```

- Return the transpose of `mat`.

```
matrix_sf* create_matrix_sf(char name, const char *expr)
```

- Parse a string (`expr`) containing a valid definition of a new matrix (as described in the "Script Format" section) and return a pointer to a correctly initialized `matrix_sf` struct. For this function, `expr` does not necessarily end with a newline character. The spacing between tokens in `expr` can vary significantly inside the string.
- For example, for `name = 'G', expr = "2 5 [ 8 4 9 1 13 ;-5 0 6 22 24 ;] ",` the return value would point to a `matrix_sf` struct with the following contents:

  ```
  name = 'G'
  num_rows = 2
  num_cols = 5
  values = {8, 4, 9, 1, 13, -5, 0, 6, 22, 24}
  ```

`bst_sf* insert_bst_sf(matrix_sf *mat, bst_sf *root)`

- Given a pointer to the `bst_sf` struct `root`, which could be `NULL`, insert the provided matrix `mat` into the BST without making a copy of `mat`. The function must ensure that the sorted property of the BST is maintained. The function creates a new BST if the `root` is `NULL`.
- Return a pointer to the root of the updated (or new) BST.
- Assume that no other `matrix_sf` exists in the BST that has the same name as `mat`.
- The function should perform no validation whatsoever of the contents of `mat`. You may assume that `mat != NULL` and that no other `matrix_sf` is stored in the BST with the same value for `mat->name`. The function must insert `mat` into the BST even if any of `num_rows`, `num_cols`, or `values` is invalid.

`matrix_sf* find_bst_sf(char name, bst_sf *root);`

- Given a pointer to the `bst_sf` struct `root`, which could be `NULL`, locate and return a pointer to a matrix named `name`, if it exists.
- If no matrix of the given name is in the BST, return `NULL`.
- Assume that at most one matrix named `name` exists in the BST.
- Assume that `root` points to a valid BST or is `NULL`.
- Assume that no two BST nodes point to the same `matrix_sf` struct.

`void free_bst_sf(bst_sf *root);`

- Given a pointer to the `bst_sf` struct `root`, which could be `NULL`, free all the nodes of the tree. Also, free all the `matrix_sf` structs pointed to by the BST nodes.
- Assume that `root` points to a valid BST or is `NULL`.
- Assume that no two BST nodes point to the same `matrix_sf` struct.

`char* infix2postfix_sf(char *infix);`

- Given an infix expression `infix`, convert it to its equivalent postfix expression.
- Return the newly allocated string containing the postfix expression.
  - You are strongly encouraged to adapt the stack-based algorithm you learned in CSE 214/260 to convert an infix expression into postfix. You'll need to handle +, *, ', (, ) and matrix names. (Note that ' is a unary operator and has higher precedence than + or *.)
- Do not forget to consider the precedence of the operators for this conversion.
- Any memory allocated by the function must be deallocated before returning, except for the string that contains the postfix expression.

`matrix_sf* evaluate_expr_sf(char name, char *expr, bst_sf *root);`

- Given the name of a new matrix (`name`), an expression involving one or more matrices and one or more matrix operations (`expr`), and a pointer to the root of a BST (`root`), perform the following tasks:
  - Evaluate `expr`, storing the resulting matrix in a new matrix called `name`.
  - Return a pointer to the new matrix.
- Any memory allocated by the function must be deallocated before returning, except for the matrix that contains the value of the expression.
- Some considerations:
  - `expr` is an infix expression. Use the function `infix2postfix_sf` to convert it to postfix.
  - To help you evaluate the postfix expression, create a stack of pointers to `matrix_sf` structs. When you encounter a +, *, or ' in the expression, pop one (for ') or two (for +, *) matrices off the stack and perform the operation. As you evaluate a +, *, or ' operation, create a new `matrix_sf` and push it onto the stack. Do NOT add the new matrix to the BST.
  - Remember: the function must deallocate the postfix expression string returned by calling the `infix2postfix_sf` function.
  - Remember: the function must deallocate any memory it allocates while evaluating the expression. (Again, the returned matrix is an exception.) When you need to create a new matrix on the fly, give it a non-alphabetical name. Later on, when this matrix is popped off the stack to apply an operator (+, *, '), you can easily identify it as a matrix that `evaluate_expr_sf` needed to create. You can then safely `free()` it after applying the operator and getting the result. Once you start implementing this function, you will understand this advice better.

Example: Suppose `expr = "P'*K+A"`.

1. First transform this expression into its postfix format, which is `"P'K*A+"`, by calling the function `infix2postfix_sf`.
2. Initialize an empty stack of pointers to `matrix_sf` structs.
3. Start processing characters sequentially from `"P'K*A+"`.
4. We see `P` in the input. Push the matrix called `P` on the stack.
(Call `find_bst_sf('P', root)` to get a pointer to that matrix.)
5. We see ' in the input. Transpose is a unary operator. Pop the top element of the stack, create a new matrix that stores the transpose, and push that new matrix onto the stack. Do NOT add the new matrix to the BST.
6. We see `K` in the input. Push the matrix called `K` on the stack.
7. We see `*` in the input. Pop the top two elements off the stack, create a new matrix that stores the product, and push that new matrix onto the stack. If either or both of the popped matrices do NOT have an alphabetical name (oh look at

that, one of them doesn't), `free()` the applicable matrix/matrices.

8. We see `A` in the input. Push the matrix called `A` on the stack.
9. We see `+` in the input. Pop the top two elements off the stack, create a new matrix that stores the sum, and push that new matrix onto the stack. If either or both of the popped matrices do not have an alphabetical name, `free()` the applicable matrix/matrices.
10. We have reached the end of the input. We will return a pointer to the matrix on top of the stack. But first, as necessary, `free()` your stack data structure and any other memory you allocated but haven't deallocated yet.

```
matrix_sf* execute_script_sf(char *filename);
```

- Given the name of a file containing a script as described under "Script Format" above, execute the contents of the file and return a pointer to the final, named matrix created on the last line of the script. For example, suppose `"C = A*B + D'"` is the last line of the file. The function will return a pointer to the `matrix_sf` struct named `'C'`.
- You are encouraged to use the function `getline`, which can read a line of text from a file and will even allocate the memory to store the string. (You'll have to deallocate the memory yourself.) You would call it as follows:

```
char *str = NULL; // Why? Read the linked web page.
FILE *file = …;
size_t max_line_size = MAX_LINE_LEN; // defined in hw7.h
getline(&str, &max_line_size, file);
```

- The function will need to call some other functions as needed:
  - `create_matrix_sf` when a line in the script defines a new matrix by providing its contents (e.g., `A = 3 4 [ … ]`), followed by a call to `insert_bst_sf` to insert the new matrix into the BST
  - `evaluate_expr_sf` when a line in the script creates a new matrix by performing one or more matrix operations, followed by a call to `insert_bst_sf` to insert the new matrix into the BST

# Toy Example

## Sample Script File: example.txt

```
D = 2 2 [1 2; 3 4;]
B = 2 2 [5 6; 7 8;]
F = D + B
A = F'
C = D * A
```

We'll trace through the execution of this script line by line. Notice how the BST will be balanced (not skewed) because we're inserting nodes in the order D, B, F, A, C.

## LINE 1: D = 2 2 [1 2; 3 4;]

### Step 1: Identify Line Type

This is a matrix definition (not a formula). It contains matrix dimensions and values in square brackets.

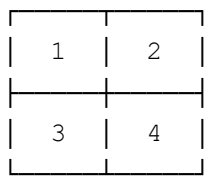
### Step 2: Parse the String

**Input string:** "2 2 [1 2; 3 4;]"

- '2' (first integer) → num_rows = 2
- '2' (second integer) → num_cols = 2
- '[' → start of values
- '1 2' → first row values
- ';' → end of first row
- '3 4' → second row values
- ';' → end of second row
- ']' → end of values

### Step 3: Visual Representation

Matrix D (2×2):




In memory (row-major order): [1, 2, 3, 4]

### Step 4: Insert into BST

Call insert_bst_sf(D, NULL). Since BST is empty, D becomes the root.

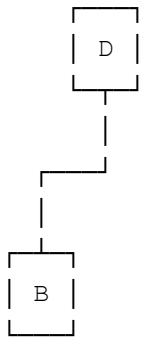
# LINE 2: B = 2 2 [5 6; 7 8;]

## Create Matrix B

Matrix B (2×2):


## Insert into BST

Compare 'B' with root 'D': 'B' < 'D' → insert as left child




# LINE 3: F = D + B

## Step 1: Convert to Postfix

Call infix2postfix_sf("D + B") → returns "D B +"

Postfix notation puts operators after operands.

## Step 2: Evaluate Postfix "D B +"

Using a stack of matrix pointers:

- Read 'D': Find matrix D in BST, push onto stack → Stack: [D]
- Read 'B': Find matrix B in BST, push onto stack → Stack: [D, B]
- Read '+': Pop B and D, compute D+B, push result → Stack: [result]

## Step 3: Matrix Addition Computation

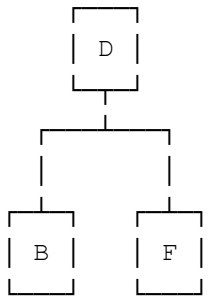
Element-wise addition:

```
D = [1  2]    B = [5  6]
    [3  4]        [7  8]

F[0][0] = 1 + 5 = 6
F[0][1] = 2 + 6 = 8
F[1][0] = 3 + 7 = 10
F[1][1] = 4 + 8 = 12


Result F = [6   8]
```

```
[10  12]
```

### Step 4: Insert into BST

Compare 'F' with root 'D': 'F' > 'D' → insert as right child




## LINE 4: A = F'

### Step 1: Convert and Evaluate

Postfix: "F'" (transpose is already postfix)

- Read 'F': Find matrix F in BST, push onto stack → Stack: [F]
- Read "'": Pop F, compute transpose, push result → Stack: [result]

### Step 2: Matrix Transpose

Flip rows and columns:

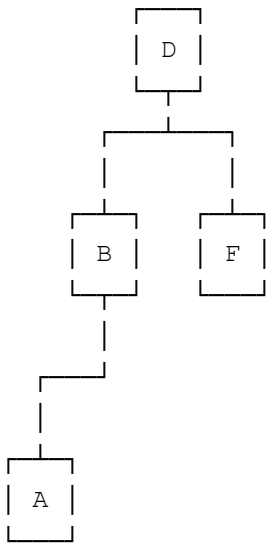
```
F = [6   8]  →  A = [6   10]
    [10  12]        [8   12]
```

Row 0 of F [6, 8] becomes Column 0 of A

### Step 3: Insert into BST

Compare 'A' with root 'D': 'A' < 'D' → go left

Compare 'A' with 'B': 'A' < 'B' → insert as left child of B




## LINE 5: C = D * A (FINAL LINE)

## Step 1: Convert and Evaluate

Postfix: "D A *"

- Read 'D': Find matrix D in BST, push → Stack: [D]
- Read 'A': Find matrix A in BST, push → Stack: [D, A]
- Read '*': Pop A and D, compute D*A, push result → Stack: [result]

## Step 2: Matrix Multiplication

```
D = [1  2] (2×2)     A = [6   10] (2×2)
    [3  4]               [8   12]
```

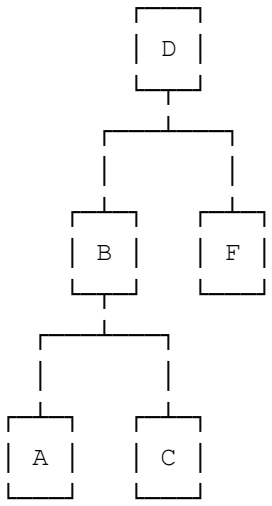
Formula: C[i][j] = Σ(D[i][k] * A[k][j])

```
C[0][0] = 1*6 + 2*8  = 6 + 16 = 22
C[0][1] = 1*10 + 2*12 = 10 + 24 = 34
C[1][0] = 3*6 + 4*8  = 18 + 32 = 50
C[1][1] = 3*10 + 4*12 = 30 + 48 = 78

Result C = [22  34]
           [50  78]
```

## Step 3: Insert into BST

Compare 'C' with root 'D': 'C' < 'D' → go left

Compare 'C' with 'B': 'C' > 'B' → insert as right child of B




## Step 4: Return

Since this is the last line of the script, execute_script_sf returns a pointer to matrix C.

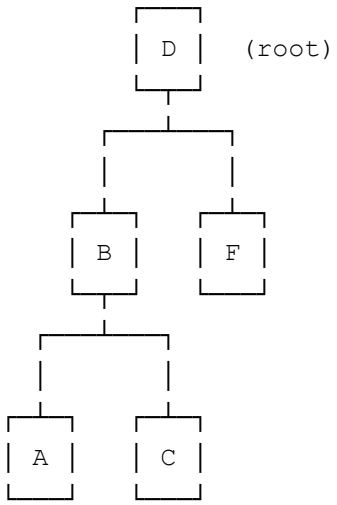
# Final Summary

## All Matrices Created

```
D = [1   2]      B = [5   6]      F = [6    8]
    [3   4]          [7   8]          [10   12]

A = [6    10]    C = [22   34]
    [8    12]        [50   78]
```

## Complete Balanced BST Structure

Notice how the tree is balanced with nodes on both left and right subtrees:




This balanced structure is more efficient than a skewed tree, with better search performance.

## BST Search Example

To find matrix 'A' in the BST:

- Start at root D: 'A' < 'D' → go left to B
- At node B: 'A' < 'B' → go left to A
- Found! Return pointer to matrix A

# Complex Expression Example

**Expression:** G = (D + B)' * F

## Step 1: Convert to Postfix

```
Infix:   (D + B)' * F
Postfix: D B + ' F *
```

Operator precedence: ' (highest) → * → + (lowest)

### Step 2: Evaluation Process

- Read 'D': Push D → Stack: [D]
- Read 'B': Push B → Stack: [D, B]
- Read '+': Pop B, D → compute temp1 = D+B = [6,8;10,12] → Stack: [temp1]
- Read "'": Pop temp1 → compute temp2 = (temp1)' = [6,10;8,12] → Stack: [temp2]
- Free temp1 (temporary matrix with non-alphabetic name)
- Read 'F': Push F → Stack: [temp2, F]
- Read '*': Pop F, temp2 → compute result = temp2*F → Stack: [result]
- Free temp2 (temporary matrix)

**Key Point:** Temporary matrices created during evaluation are given non-alphabetic names (like '!', '@') so they can be identified and freed after use.

## Memory Management

### What Gets Allocated

- Matrix structures via malloc(sizeof(matrix_sf) + ...)
- BST nodes via malloc(sizeof(bst_sf))
- Postfix strings from infix2postfix_sf()
- Temporary matrices during expression evaluation

### What Gets Freed

- Temporary matrices (non-alphabetic names) immediately after use
- Postfix strings after evaluation completes
- Entire BST and all matrices at program end via free_bst_sf()

### Memory Management Example

```
// In evaluate_expr_sf:
char *postfix = infix2postfix_sf(infix);
// ... use postfix ...
free(postfix);  // Don't forget!
// When popping from stack:
matrix_sf *m = pop_stack();
if (!isalpha(m->name)) {
    free(m);  // Temporary matrix
}
```

## Recommended Implementation Order

- 1. add_mats_sf, mult_mats_sf, transpose_mat_sf (basic operations)
- 2. create_matrix_sf (parse matrix definitions)
- 3. insert_bst_sf, find_bst_sf, free_bst_sf (BST operations)
- 4. infix2postfix_sf (expression conversion)
- 5. evaluate_expr_sf (evaluate expressions using stack)
- 6. execute_script_sf (tie everything together)

# Testing Your Code

## Test Case 1: Basic Operations

```
A = 1 1 [5;]

B = 1 1 [3;]

C = A + B
```

**Expected:** C = [8]

## Test Case 2: Transpose

```
A = 2 3 [1 2 3; 4 5 6;]

B = A'

Expected: B = [1  4]

            [2  5]

            [3  6]
```

## Test Case 3: Multiplication

```
A = 2 2 [1 2; 3 4;]

B = 2 1 [5; 6;]

C = A * B

Expected: C = [17]  (1×5+2×6)

            [39]  (3×5+4×6)
```

# Advanced Test Cases

Here are some complex test cases to thoroughly test your implementation:

## Test Case 4: Complex Expression with Parentheses

```
A = 2 2 [1 2; 3 4;]
B = 2 2 [2 0; 1 2;]
C = 2 2 [1 1; 1 1;]
D = (A + B) * C
```

**Step-by-step:**

- A + B = [3 2; 4 6]
- (A + B) * C = [5 5; 10 10]

**Expected:** D = [5 5; 10 10]

## Test Case 5: Transpose in Complex Expression

```
A = 2 3 [1 2 3; 4 5 6;]
B = 3 2 [1 0; 0 1; 1 1;]
C = A * B'
```

**Step-by-step:**

- B' = [1 0 1; 0 1 1] (2×3)
- C[0][0] = 1×1 + 2×0 + 3×1 = 4
- C[0][1] = 1×0 + 2×1 + 3×1 = 5
- C[1][0] = 4×1 + 5×0 + 6×1 = 10
- C[1][1] = 4×0 + 5×1 + 6×1 = 11

**Expected:** C = [4 5; 10 11]

## Test Case 6: Multiple Transposes

```
A = 3 2 [1 2; 3 4; 5 6;]
```

`B = (A')'`

**Step-by-step:**

- `A' = [1 3 5; 2 4 6] (2×3)`
- `(A')' = [1 2; 3 4; 5 6] (3×2) - same as A`

**Expected:** `B = [1 2; 3 4; 5 6]`

## Test Case 7: Chained Operations with Three Matrices

`A = 2 2 [1 0; 0 1;]`

`B = 2 2 [2 3; 4 5;]`

`C = 2 2 [1 1; 1 1;]`

`D = A * B + C`

**Step-by-step:**

- `A * B = [2 3; 4 5] (identity times B equals B)`
- `(A * B) + C = [3 4; 5 6]`

**Expected:** `D = [3 4; 5 6]`

## Test Case 8: Large Matrix Multiplication

`A = 3 4 [1 2 3 4; 5 6 7 8; 9 10 11 12;]`

`B = 4 2 [1 0; 0 1; 1 0; 0 1;]`

`C = A * B`

**Step-by-step:**

- `A (3×4) * B (4×2) = C (3×2)`
- `C[0][0] = 1×1 + 2×0 + 3×1 + 4×0 = 4`
- `C[0][1] = 1×0 + 2×1 + 3×0 + 4×1 = 6`
- `C[1][0] = 5×1 + 6×0 + 7×1 + 8×0 = 12`
- `C[1][1] = 5×0 + 6×1 + 7×0 + 8×1 = 14`
- `C[2][0] = 9×1 + 10×0 + 11×1 + 12×0 = 20`
- `C[2][1] = 9×0 + 10×1 + 11×0 + 12×1 = 22`

**Expected:** `C = [4 6; 12 14; 20 22]`

## Test Case 9: Complex Nested Parentheses

```
A = 2 2 [1 2; 3 4;]
B = 2 2 [5 6; 7 8;]
C = 2 2 [1 0; 0 1;]
D = ((A + B)' * C)'
```

**Step-by-step:**

- A + B = [6 8; 10 12]
- (A + B)' = [6 10; 8 12]
- (A + B)' * C = [6 10; 8 12] (identity matrix preserves)
- ((A + B)' * C)' = [6 8; 10 12]

**Expected:** D = [6 8; 10 12]

## Test Case 10: All Operations Combined

```
A = 2 3 [1 2 3; 4 5 6;]
B = 3 2 [1 0; 0 1; 1 1;]
C = 2 2 [2 2; 2 2;]
D = A * B + C
```

**Step-by-step:**

- A * B = [4 5; 10 11]
- A * B + C = [6 7; 12 13]

**Expected:** D = [6 7; 12 13]

## Test Case 11: Matrix Chain with Transpose

```
A = 2 3 [1 0 1; 0 1 0;]
B = 2 3 [1 1 1; 2 2 2;]
C = 3 2 [1 0; 0 1; 1 0;]
D = (A + B) * C
E = D'
```

**Step-by-step:**

- A + B = [2 1 2; 2 3 2]

- (A + B) * C = [4 1; 6 3]
- D' = [4 6; 1 3]

**Expected:** D = [4 1; 6 3], E = [4 6; 1 3]

## Test Case 12: Identity Matrix Properties

I = 3 3 [1 0 0; 0 1 0; 0 0 1;]

A = 3 3 [2 3 4; 5 6 7; 8 9 10;]

B = I * A

C = A * I

**Property Test:**

- Identity matrix multiplied by any matrix returns the same matrix
- I * A should equal A
- A * I should also equal A

**Expected:** B = C = [2 3 4; 5 6 7; 8 9 10]

## Test Case 13: Deeply Nested Expression

A = 2 2 [1 2; 3 4;]

B = 2 2 [0 1; 1 0;]

C = 2 2 [1 1; 1 1;]

D = (A' + B) * (C + B')

**Step-by-step:**

- A' = [1 3; 2 4]
- A' + B = [1 4; 3 4]
- B' = [0 1; 1 0] (B is symmetric)
- C + B' = [1 2; 2 1]
- Final multiplication: [1 4; 3 4] * [1 2; 2 1] = [9 6; 11 10]

**Expected:** D = [9 6; 11 10]

## Test Case 14: Testing Operator Precedence

```
A = 2 2 [1 2; 3 4;]

B = 2 2 [1 0; 0 1;]

C = 2 2 [2 2; 2 2;]

D = A' * B + C
```

**Precedence Analysis:**

- Transpose has the highest precedence: A' is computed first
- Multiplication next: A' * B
- Addition last: result + C

**Step-by-step:**

- A' = [1 3; 2 4]
- A' * B = [1 3; 2 4] (B is identity)
- A' * B + C = [3 5; 4 6]

**Expected:** D = [3 5; 4 6]

# Test Case 15: Maximum Complexity

```
A = 2 3 [1 2 3; 4 5 6;]

B = 3 2 [1 0; 0 1; 1 1;]

C = 2 2 [1 1; 1 1;]

D = 2 2 [2 0; 0 2;]

E = ((A * B)' + C) * D
```

**Step-by-step:**

- A * B = [4 5; 10 11]
- (A * B)' = [4 10; 5 11]
- (A * B)' + C = [5 11; 6 12]
- Final * D = [10 22; 12 24]

**Expected:** E = [10 22; 12 24]

**Testing Tips:** Start with simple test cases and gradually increase complexity. Verify intermediate results by hand for at least one complex example. Use these test cases to ensure your implementation handles all operator precedences, parentheses nesting, and matrix dimension combinations correctly.

## Testing & Grading Notes

● During grading, only your `hw7.c` file will be copied into the grading framework's directory for processing. Make sure all of your code is self-contained in that file. ● Most test cases will be executed twice to check (a) the correctness of the return value from a function; and (b) the correctness of your memory usage by Valgrind. See `unit_tests.c` for details. As with the previous assignment, credit will be awarded for a successful Valgrind test only when its paired computational test passes.



● Remember to regularly `git commit` your work. Occasionally, `git push` your work to run the same tests on the git server ***and to submit your work for grading*** by the due date.
● It is useful and good practice to create helper functions as you are working on the assignment. These are allowed and will not interfere with grading. However, make sure to only modify `hw7.c`.

## Academic Honesty Policy

Academic honesty is taken very seriously in this course. By submitting your work for grading, you indicate your understanding of, and agreement with, the following Academic Honesty Statement:

1. I understand that representing another person's work as my own is academically dishonest.
2. I understand that copying, even with modifications, a solution from another source (such as the web or another person) as a part of my answer constitutes plagiarism. 3. I understand that sharing parts of my homework solutions (text write-up, schematics, code, electronic or hard-copy) is academic dishonesty and helps others plagiarize my work.
4. I understand that protecting my work from possible plagiarism is my responsibility. I understand the importance of saving my work such that it is visible only to me. 5. I understand that passing information that is relevant to a homework/exam to others in the course (either lecture or even in the future!) for their private use constitutes academic dishonesty. I will only discuss material that I am willing to openly post on the discussion board.
6. I understand that academic dishonesty is treated very seriously in this course. I understand that the instructor will report any incident of academic dishonesty to the University's Academic Judiciary.
7. I understand that the penalty for academic dishonesty might not be immediately administered. For instance, cheating on a homework assignment may be discovered and penalized after the grades for that homework have been recorded.
8. I understand that buying or paying another entity for any code, partial or in its entirety,

and submitting it as my own work is considered academic dishonesty.
9. I understand that there are no extenuating circumstances for academic dishonesty. 1