Implementation of Matlang Transpiler to C

### Files

* main.c → takes the input as argument and prints the output file, calls the other files when needed.
* split.c →reads the input file character by character and splits it into the lexemes.
* scalar.c
* vector.c → checks the syntax and prints the a definiton statement if there is no error.
* matrix.c
* printsep.c
* print.c → checks the syntax and prints the executable statement if there is no error.
* f4.c
* assignment.c
* expr.c → takes an expression and returns the expression’s result’s dimension and the expression’s equivalent syntax in C code(like a\*b -> product(a,b)).
* searcher.c → takes a variable name and returns the variable’s dimension.
* stacksolver.c → solves the postfix expressions.
* stack.h → header file for the stack data type. The last four files hold the code for parsing and solving the expressions.

### Taking Input From File

Input is stored in a triple dimension array called lexemes which the first dimension corresponds to the lines, second corresponds to the lexemes, and the third corresponds to a single character.

In the split.c the input file is being taken character by character in each iteration and written into the lexemes array. Whenever the taken character is a special one, (paranthesis, operators, commas and columns) it continues from the next lexeme array. Whenever the taken character is a comment character, it skips the rest of the characters until it is a newline character. Whenever the taken character is a newline character it continues from the next line array.

 Figure 1: The way the input is being taken

### Converting Expression to Dimension and Equivalent Code in C(Syntax Check)

Example:

Matrix dimension: [3,3]

Vector dimension: [3]

Scalar dimension: 3

We started to implement our parser by using your code, however we changed %80-85 percent of the code and added a lot of different stuff. When the code progresses recursively to make prefix to postfix, we solve the postfix for every recursion.Think that we have terms -> terms moreterms, when code goes back to terms in the left side, code solves the postfix and code generates a equivalent code in C for this operation.

Example: a \* b -> a b \*(it solves it actually and returns a dimension) and product(a,b)

Note: We have different functions for +, - , \*; but for simplicity we use product, sum and sub.

Other than generating the equivalent code, code uses dimensions to check syntax errors. Therefore for every recursion step, we solve the postfix and return a dimension instead of postfix(This is for syntax, not generation). We have two strings which are returned by these operations. As I mentioned, one of them is the dimension of the all expression, other one is the equivalent code in C.

To have equivalent code in C, for every recursion step we send the current expression in the current recursion step to have the dimension and by this dimension, we decide which function we should use(for matrices, vectors or scalars etc.).

**Transpose:**

tr(<expr>). To have transpose, we first return the dimension inside of the expression and after that we transpose it. Also, for equivalent code in C, code puts “tr(“ and “)" outside of expression. For matrix and vector, transpose we have “trm”.

**Sqrt:**

Sqrt(<expr>). To have sqrt, we first return the dimension inside of the expression and after that we do nothing for dimension because it doesn’t affect our syntax check. Also, for equivalent code in C, code puts “sqrt(“ and “)” outside of expression. For matrix and vector, transpose we have “sqrtm”.

**Choose:**

Choose(<expr>, <expr>, <expr>, <expr>). To have choose, we first return the dimension inside of the expressions and check whether they are scalars or not. After that we return random scalar for dimension. Also, for equivalent code in C, code puts “choose(“ and “)” outside of expression. Also we handle the commas, too.

There is one more important function to mention that we use it for variables with indices.

Like upper functions, first we make the inside expressions as dimensions. Later, we check that whether they are scalar or not. In C code we used stack to store the variables, so for generating C code, we do this: A.values[][]. For both vectors and matrices, because we assumed that they are same in only our C code.

**Printing to the Output File**

Since the matrix operations on c are not defined the functions were defined at the start of the c code the functions are implemented. Matrix and vectors are stored the same way in the output, as a matrix struct, the dimensions are stored as integers and the values are stored in a double dimension array that is a part of the struct.

Note: Some of functions that are used in the output c code are implemented through the use of math library so in the compilation -lm flag must be used.