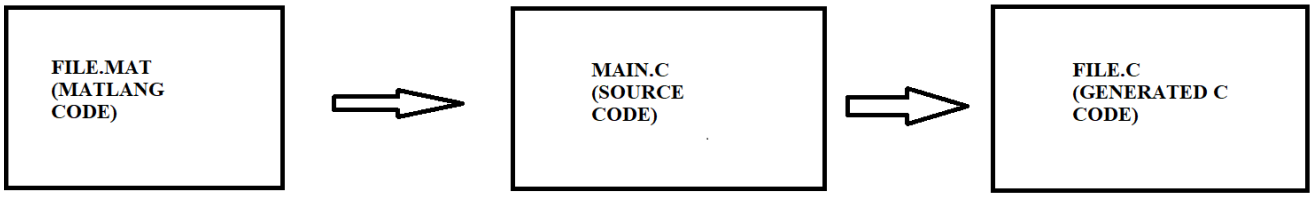
Begüm Yivli(2019400147)

Egecan Serbester(2019400231)

CmpE230 Homework Documentation-1

Diagram:



Detailed Explanation;

At first, we opened the input file with the file methods and started reading it line by line. If the file was not given in the command line, we gave an error message. Then we converted the line we read into a char array and started to analyze character by character. We opened an empty char array and if some characters like (} + come in, we add a space before and after them and add them to the array we opened. If it is not such a character, we just add itself. This made it easier for us to split the line into tokens. Then we used the strtok_r function to add a space to it. We obtained the tokens by separating them and numbered these tokens Thanks to our token numbering, we were able to check the syntax and if there was an error, we printed the line with the error to the generated file and closed the file.

We compared the obtained tokens with some expressions with the strcmp function and understood what they are. We created a number equivalent for each token in Enum and put these equivalents into an integer array called “data”. We filled this array with 0 again at the end of each line. 0 is the control variable in the enum because we never used it and made things easy. After we finished tokenizing the line, we understood what it was doing in that line according to the corresponding expression in the enum from the 0th element of the data.

If the first element of this line is #, we ignore that line with the continue command. If it is a scalar expression, we have recorded the name and type of this scalar variable in our struct arrays named variables and scalars. We did this for the vector and matrix expressions, respectively, by accepting the vector as a one-dimensional matrix and making e_cols, that is,

the number of elements in the column, the number of rows in the matrix struct 1. Thus, we have kept every property of each variable and we have made a declaration in our generated file according to the properties we have obtained.

If data[0] is the number corresponding to the print function and there is only one variable in it, we have 3 possibilities: printing the scalar value, printing the vector elements line by line, and printing the lines of the matrix in order. Here, we take the token name and check the name from the variables struct array, look for it in the matrices or scalars array according to its type, and get the value from this array and print it in our generated file.

If we find that we have come across a two-variable expression such as id[num], we have only one option, to print the value of the vector at the desired index. The point we need to pay attention to here is that array indexes start from 1 in matlang, so we reduce the desired index in our C code. Again, we find the corresponding name in the matrices array and print the variable name and value to our generated file. If we come across an expression containing 3 variables in the form of id[num,num], we print the name and value from the array to our generated file, paying attention to the index.

If data[0] is an identifier then this line is an assignment. If it contains { , then the vector or matrix is being assigned a value. Seeing this, we extract the values one by one, update the values in the array containing the structs, and make vector and matrix double array definitions in our generated file.

If data[2] is a number and then there is no element, we are assigning a number directly to an identifier. In this case, we print the code that assigns values to the generated file and update the values in our struct array, because we can quickly get the value without sending the expression to the parse function.

```
expr        →   term moreterms

moreterms   →   + term moreterms
            |   - term moreterms
            |   null

term        →   factor morefactors

morefactors →   *   factor morefactors
            |   null

factor      →   ( expr )
            |   id[ expr, expr ]
            |   id[ expr]
            |   sqrt(expr)
            |   choose(expr1,exr2,expr3,expr4)
            |   id
            |   num
            |   tr(expr)
```

Then, if there is more than one token in the expression, we parse them recursively according to the grammar rules above. We called the expr function with an empty string like expr(str) and this function pulled the tokens from the expressions array and wrote the expression as a postfix. Then, in order to evaluate this postfixed expression more easily, we divided it again into its tokens and put it in the exprarr array.

Then we check it with whatres(expr) function which returns 1 if there is no matrices in the expression (A[2,2] doesn't evaluate as matrix since it returns scalar). If it is a scalar expression then we edit it with editscalar(expr) function which returns the actually scalar values (like matrix A[2,2] or vector x[2]), it changes those charracters with their actually values. Then we search the scalar which we assigned in scalars struct array. When we reach the scalar we use resultt(expr) function which returns the result of the expression and we assign the result to the value of the scalar.

If the whatres(expr) returns 0 that means there is at least 1 matrix in expression. Then we call the resultarr(double adj[], char expr[]) function which equalizes the result of the expression which includes matrices the adjacent matrix. Then we search the matrix (or array we assume bot hare the same) which we assigned in matrices struct array. When we reached the matrix we initialize its values with assignMatrix(double adj[], struct Matrix* matrix, int size) function which initialize the values to the value of matrix one by one. After all if we didn't initialize the matrix before with the way { }, we printed the values in file.c one by one.

If we have enough time we can make the sqrt, chose and tr functions with those expressions. However we have only last 1 hour while we were writing this lines…