RULES

* <var\_name> -> alphanumeric (c rules)

<var\_name>[<num>]

<var\_name>[<num>,<num>]

* <operator> -> (\* + -)
* <num\_list> -> <num> | <num> <num\_list>

DEFINITION RULES

* scalar <var\_name>
* vector <var\_name>[<vector\_length>]
* matrix <var\_name>[<col\_length>,<row\_length>]

ASSIGNMENT RULES

* <assign-scalar> -> scalar <var\_name> = (<num>|<expression>)
* <assign-vector> -> vector <var\_name> = (<num\_list>|<expression>)
* <assign-matrix> -> matrix <var\_name> = (<num\_list>|<expression)

EXPRESSION RULES

* expression -> <var\_name>

<num>

<func <expression>>

<expression> <operator> <expression>

(<expression>)

FUNCTION RULES

* <func> -> <tr> | <sqrt> | <choose> | <print> | <print\_sep>
* <tr> -> tr(<expression>)
* <sqrt> -> sqrt(<expression>)
* <choose> -> choose(<expression>,<expression>,<expression>,<expression>)
* <print> -> print(<var\_name>)
* <print\_sep> -> print\_sep()

FOR RULES

* <for> ->
  + for(<var\_name> in <expression: <expression>: <expression>) -> from exp1 to exp2 by adding exp3
  + for(<var\_name>, <var\_name> in <expression>: <expression>: <expression>, <expression>: <expression>: <expression>)

We started by implementing the vector data type since something like an ArrayList from Java would make developing much easier. As we would use the vector type to hold different types of data, of which strings are the most frequently used, we decided to implement it as a double void pointer.

The program holds vectors of variable names and dimensions in a struct named Variables.

Workflow of the program:

* Start reading lines from the input file.
* Tokenize the line.
* Match the line with the rules above.
* Send the line to the convert\_line function to first check for syntax errors and then translate it into C code.
* Definition -> we have 3 custom functions: make\_scalar, make\_vector, make\_matrix.
  + All of them check if the variable names are legal, add variables and their sizes to the corresponding field in the Variables struct and construct the C equivalent.
* Assignment -> check the right hand side (expression) & construct C equivalent
* Expressions -> convert\_complex\_expr is the main function to handle expressions.
  + Expressions are categorized like binary operation, matrix operation, vector operation, in-parentheses expression etc.
  + As we define expressions recursively, handling after parsing them is not a complex task.
  + In every match the expression is trimmed and the rest is used as a parameter to the same function again.
  + There are matchers for expressions. These functions check the syntax and if it matches the rules it returns true.
  + After matching, trimming and constructing C equivalent is made in the convert\_complex\_expr function.
* Functions -> functions except print and print\_sep are handled as expressions.
  + Like expressions, after matching the function with the rules, trimming and syntax checking is applied.
  + After that, the C equivalent is constructed by translating the inside of the function with the convert\_complex\_expr function.
  + A custom header containing the C equivalents of the matlang functions is included in the output.c file.
* For -> For and double for are handled inside the make\_for function.
  + Similar to other types of expressions, first, the line is matched, then the variable names and expressions are extracted from the loop head, which are then translated into C.
  + If everything is ok it constructs the c equivalent of the loop.
* Print -> print and print\_sep functions are handled inside make\_print\_func and make\_print\_sep\_func. For the latter, “print\_sep();” is added into the output.c file. And for the former, the convert\_complex\_expr is called for the inside of the function and the corresponding print function in C is chosen according to the return type of the expression.