CMPE230 – PROJECT 1 DOCUMENTATION

Ali Başaran – Ozan Oytun Karakaya

Abstract:

This program takes an matlang code as argument and produces a code written in C which is the exact translation of matlang code. Program uses a simple grammar consists of declarations, statements (print statements and assignment statements only), for loops (up to 2 dimensional nested loops). Matlang includes 3 built-in functions , choose, tr and sqrt. Implementation details are discussed below. Schematic of implemented program is depicted below.

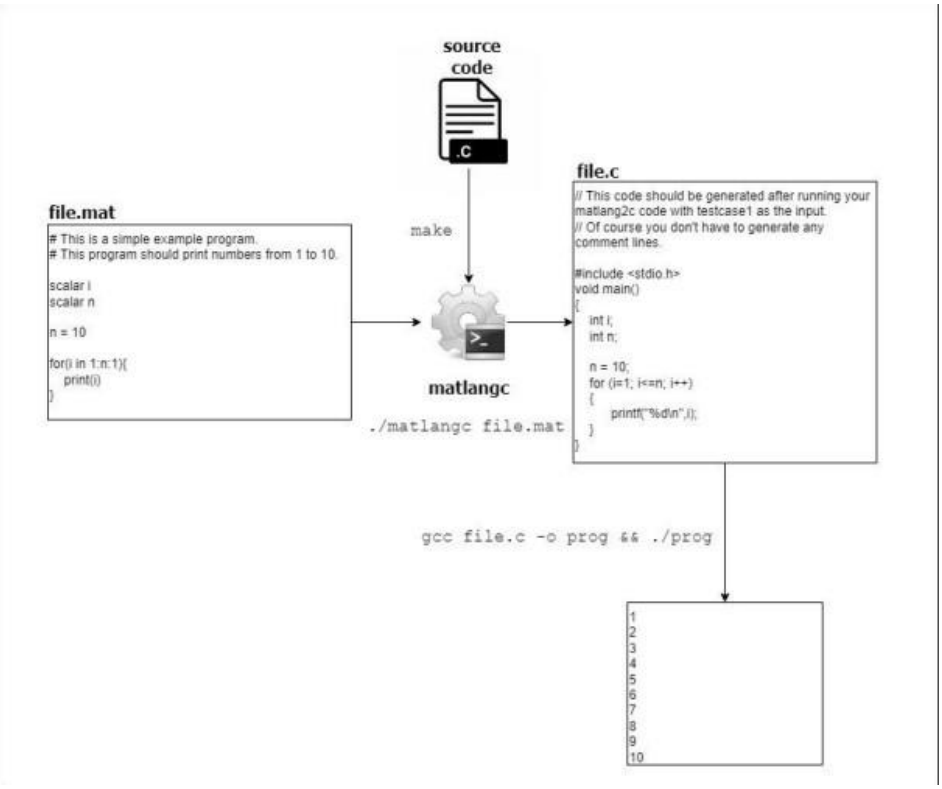


TABLE OF CONTENTS

1. ANALYZE OF INPUT

Input file taken as argument are treated with a line by line approach. Maximum line for matlang code is 256 and maximum length for a single line is 256. Input file is read with **fgets** function of C and every line stored as sentence variable. Every sentence variable is firstly arranged with white spaces between the reserved tokens of matlang code, such as parenthesis, variables, **print**, **tr**, **sqrt**, **choose**, **for**, **in** etc. These whitespaces are placed between every token to be read in order to separate them with appropriate form. After every line created, they are stored in a big 256x256 char** type pointer (it is also called array documentation) called **splittedLines.** In this array, every line consists of tokens that are separated from the lines with **strtok(newSentence, " ")** function call.

Whole input is processed in a big while loop which is the action to be taken is decided according to **splittedLines**'s current array with necessary if-else blocks.

- splittedLines[0] == "<variable-type>" for declarations
- splittedLines[0] == "<print>" for **print** statements
- splittedLines[0] == " <for>" for **for** statements
- splittedLines[1] == "=" for assignment statements

After taking the necessary key tokens for specifying statements, rest of specified tokens are taken from the **splittedLines** array, they are processed through the line. If that line does not fit the specified format, then while loop is broken and program stops with the corresponding error line number.

After constituting all necessary output codes, these are stated in the **resultArr** in order to produce a compiled language rather than an interpreted language. By this, whole matlang code is analyzed according to rules and output code is stored in a result array and whole output file printing processes are done with that array.

2. DECLARATIONS

Matlang code is in two partitions, declarations and statements. In the declaration part, there are three types of variables, scalar, vector and matrix. Scalars stand for the numerous variables such as integers and floating point numbers. Vectors are actually matrices with the dimension of column is 1. Matrices are typical 2D arrays which have size of row dimension number and every row hold another array with column dimension number. Another note about declarations is declaration of a variable can be made just once which is checked with necessary arrays explained below.

2.1 SCALAR DECLARATIONS

Scalars stand for integers and floating point numbers and these type of variable is held as double in the source code. It is also held in double type in output file, however, for integer inputs; if there is a difference that smaller than epsilon (epsilon used in source code is: 1/1024576) than it is rounded to floor and that variable is printed if necessary.

In matlang code, scalar type declaration for variable name **id** is done with line:

scalar <id>

This line above has its corresponding line in source and output C code as:

double <id>;

and when this type of line comes as an input, a variable with name <**id**> is created and put into **varNames** array and has its type is held in **varTypes** array with the corresponding **varNumber** variable which is incremented for every variable created. Also adjustments in the **row_dimensions** and **col_dimensions** arrays with the NULL value are made, which are stated furthermore.

varNames is an array 512 of pointer to pointer to char type. This array holds all variables created.

varTypes is an array 512 of pointer to pointer to char type. This array holds only two type of content; scalar or matrix since vectors are assessed as matrices with column dimension number 1. Necessary adjustments for vector indexing are done in expression parsing part and the printing part.

2.2 VECTOR DECLARATIONS

Vectors are held as matrices as stated above, so that there are no major differences between matrix declarations and vector declarations. In vector declaration statement for vector **id**, vector and the size of it is given by the line:

vector <id>[<size>]

Corresponding line in the C code firstly creates a double**<id> array with memory allocation and fills every index with the pointer to double which holds the value 1 for vectors only in the assignment part.

Same type of variable is created in the source code for error checks and it is put in the **varNames** and **varTypes** with the corresponding **varNumber** index. In the **varTypes** array vectors are stored with "**matrix**" value. Besides them, there two arrays created for dimension checks in the code, **row_dimensions** and **col_dimensions**. For vector declaration, **row_dimensions** is filled with corresponding size value and **col_dimensions** is filled with value 1 with **varNumber** index.

2.3 MATRIX DECLARATIONS

For matrices same type of double** pointer is created for storing them in the C code. Their declaration statement in matlang code is similar to vectors with the given sizes of rows and columns separated by a comma.

matrix <id>[<row_size>,<column_size>]

varTypes array takes the value "**matrix**" for current **varNumber** index and for the **varNames** array is filled with the id in the corresponding index. Since **row_dimensions** and **col_dimensions** are synchronized in the indexes just as **varNames** and **varTypes** according to **varNumber** variable their same index is filled with **row_size** and **column_size**.

These row numbers and column numbers are used in the code frequently since every multiplication, addition, assignments and so on are made with them. They will be explained below.

3. ASSIGNMENT STATEMENTS

Assignment statements constitutes the second part of the code with the print statements and assignment statements have a common grammar such:

<**id**> = <**expression**>

This expression is sent to sequential functions which will be explained below, for now it is enough to know that these functions return the true infix notation of any input given to them. Also the information of the type that the expression evaluates is kept in **varTypes** array so that comparison of the type of **id** on the left and the type of expression can be made and non-compatibility can give error for unmatched types. Note that, **id** can have the form of a specified index of a declared vector or matrix above in the matlang code. However, it is not allowed to assign a whole row of a matrix with specifying the row index of matrix, it is just allowed for single entries.

At this block of code, the declaration situation of the **id** on the left is also controlled. If **id** is not declared at the first part of the matlang code (which is the declaration part) then program breaks due to unresolved reference of **id**.

If any assignment of declared variable is not made, then it is assigned to zero value as default. Thus, indexes of vectors and matrices that are not assigned explicitly are assigned to zero.

3.1 EXPRESSION PARSING

Expressions are parsed, checked whether if there are any syntax errors and re-constituted according to grammar specified below

```
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)
```

General behavior for expressions in source code is the following:

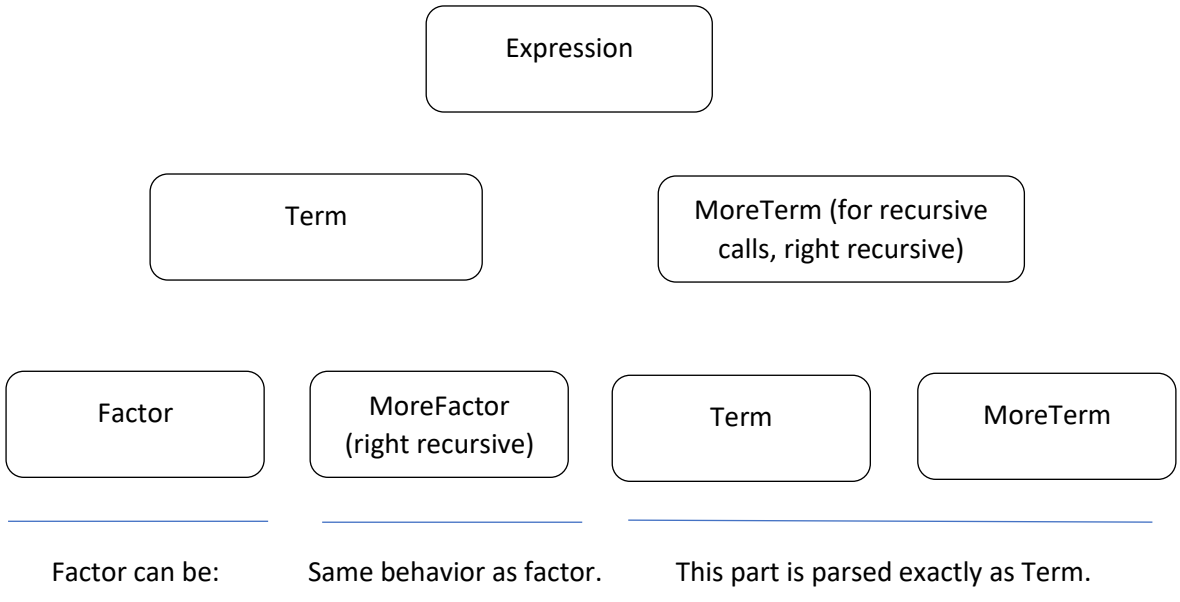
expression read from matlang → postfix notation of expression → infix notation of expression

Here two functions from two distinct files are used, **int expr(char *str, char *tokens[], char *varNames[])** from **expressions2.c** and **void postFixToInfix(char *sentence[], char *result, char *varNames[], char *varTypes, int lineId, int *varNum, char *coldm[], char *rowdm[], char *vectors)** from **expression.c** file.

3.1.1 expr FUNCTION

This function takes three parameters, str for concatenating all of the parsed sub-expressions, **tokens** array for reading the expression input and the **varNames** array used in the **main.c** file for checking whether **id**s used in sub-expressions is in declared in the variable list or not.

In this function, the grammar above is implemented with the recursive calls of sub-functions. Purpose of this function is parsing the whole expression and creating expression's postfix form while processing the syntax checks. General implementation of expr and sub-functions such as term, moreterm, factor and morefactor are the same, creating the empty strings to be filled at first, then giving them as arguments of sub-function calls. So, the minimal meaningful piece of expression is parsed till the factor function and tokens are first checked there. Then their concatenation is filled into empty string which is the parameter, then their form goes up to term, moreterm and the expr itself last. The diagram is depicted below for this process.



- (expr)
- id[expr,expr]
- id[expr]
- sqrt(expr)
- choose(expr1, expr2, expr3, expr4)
- id
- num
- tr(expr)

Recursive calls of this functions are made as following:

```
int expr( char*str, char*tokens, char*varNames[]){

    str1[0] = str2[0] = '\0';
    if (!term(str1, tokens, varNames)){
        return (0);
    }
    If (!moreterms(str2, tokens, varNames)){
        return (0);
    }
    strcat(str1,str2);
    strcpy(str,str1);
    return (1);
    }
```

Empty strings are created and they are used in calls of sub-functions, so that they can fill themselves up recursively through the other sun-functions, such as factor, morefactor, etc. Note that type of functions are integers, due to this property, if functions returns 1, as the true value, they can go on with continuing if statement and call the other sub-functions inside if statements. When they return 1, it means that expression is parsed and concatenated.

For syntax checking, there are various types of functions implemented in the source code. **isChoose** returns 1 if token read is choose, **is_integer** returns 1 if given input is an integer, i**sFunction** is additional to **isChoose** which checks for **tr** and **sqrt** functions. **Isvariable** takes an extra parameter as **varNames** and checks whether if given variable id is in variable list.

All other syntax checks are done in the factor function. This function gives error in case of matrix usages with inappropriate indexing such as not giving comma between indexes, deficiency of brackets when using and declaring matrices and also expressions are checked in case of deficit parenthesis number.

It is important to note that in order to distinguish matrices' commas, choose function's commas. In order to accomplish this, between the choose expressions the pipe character "|" is used. For matrices, if there is any matrix indexing with comma separation, then this comma is replaced with "@" character and these distinguish characters are processed in the sequence function which is **infixToPostFix**.

3.1.2 infixToPostFix FUNCTION

This function basically takes the postfix notation came from the expr function and turns that into infix notation with necessary type checks and dimension checks. It takes parameters, char *sentence[] for reading the postfix notation token by token, char *result for printing the last version of infix notation, char* varNames[] and char *varTypes[] in order to create new variables and assigning their types (this part is explained further below.), int lineID for giving errors with their line numbers, int *varNum for recursively called functions, char *rowdm[] and char *coldm[] for assigning dimension numbers and lastly char *vectors[] in order to distinguish vectors from matrices in the code.

This function is implemented with the following design:

When an expression in the postfix form given inside, it pushes the operand variables and numbers to the stack written at the beginning of the file. When addition, multiplication, subtraction operators are read from the sentence array, since the sentence is in the postfix form it pops the two operands that came earlier in the stack along with the operator. Infix notation is created with the **strcat** function of C and it is embraced with parenthesis and created as a new variable with the operand types and it is assigned in the **varNames** with infix notation name itself. Also **varType** array is modified with the operand type so, **coldm** and **rowdm** arrays. All these operations are made with the **varNum** pointer incrementation. After this variable creation, it is pushed to stack and note that the last form of expression in the infix form is the last element popped from the stack.

For all of the functions, a recursive manner is used to handle them so that calls inside another function can be handled. For all of the functions necessary expression inside the function is separated from the other tokens and it is stored in a new sentence array so that the infixToPostFix function can be recursively called with the new sentence. After new sentence is handled by the function, then its infix notation is concatenated with the necessary capsulation such as **tr(<infix_expression>)** for the **tr** function.

For dimension checks with multiplying the vectors and matrices, the new variable created have the appropriate row and column number according to operands. It is same for the **choose**, transpose and sqrt functions where choose must have the scalar or number type expressions inside and returns the same type, **tr** can have both because of convenience and the **sqrt** have a numerous expression inside and returns a numerous output.

4. PRINT STATEMENTS

Print statements can have an expression inside since it is handled in infix part so the important part for print statement is the functions in the output executable file which prints the input which is given to them.

Additionally **printsep** function is implemented in the output file along with the other functions for utility purposes.

Print processes are done when **splittedlLines[0]** token is equal to “print” token. Print statements have the type:

print(<expression>)

In the **main.c** file this is interpreted and processed as **fprintf(<infix-notation>)** where infix notation is the form which is firstly taken as input for **expr** function and then **postFixToInfix** function.

Since every numerous input type is held with double, in the printing process there is an epsilon used, a very small accurate value. If the index of any array or a scalar type variable has a smaller difference with its floored value then its integer version is printed to the screen, otherwise the double itself is printed.

For id checks, it is already done in the **expr** and **postFixToInfix** functions.

5. FOR LOOPS

For loops are easy to parse since the reserved tokens separate the expressions, so the number of loop variables is counted and if it is 2, a nested loop structure is created and please note that there are no more than 2 dimensional for loops in the matlang code.

Statements inside the for loops are distinguished with the curly braces and they are treated as any other statement read from the input file in the big while loop. Actually, after handling the expression part, there are no major process going on the **main.c** file for the loops.

6. FUNCTIONS FOR SOURCE CODE

Other than the functions explained above, there are frequently used functions in the source code which is better to mention of: isReserved, error, find.

bool isReserved(int c, int arr[]);

This function takes an integer array as parameter which is synchronized in the index with reserved token arrays, and the int c parameter it takes refers to index of that array. For any type of token, before splitting all of the tokens in a line the check process of tokens are made with the reserved ones and this function is used there.

int find(char *varName, char *varNames[])

This function is frequently used for getting the **varNumber** value since all of the arrays initialized in the **declaration** and **assignment** parts are synchronized with the **varNumber** index. Simply it takes the variable name as parameter and looks for it inside the **varNames** array with a simple for loop.

void error(int lineId)

This function is used with the break; statement since if there is any error, the compilation must stop. It takes the lineId as parameter which is held throughout whole program. It prints a simple error message to the console.

7. CREATION OF OUTPUT FILE

Before creation of output file, all of the output statements are stored in the **resultArr** as stated above. While storing results in this array, necessary capsulations of strings are made such as placing semi-colons at the end of lines or curly braces for for loops.

After the main while loop terminates, then output file is created and **resultArr** is printed line by line into that file.