Project Proposal for EZMath

Group Information:

Manager:	*Piaoyang Cui*	*pc2618*
Language Guru:	*Yi Wang*	*yw2580*
System Architect:	*Shangjin Zhang*	*sz2425*
Verification & Validation:	*Zhejiao Chen*	*zc2291*

1 Introduction

1.1 Motivation

Complex mathematical operations and representations is always highly demanded for scientific programming. When creating top-notch academic papers, LaTex, a markup language to typeset document, is often used to prettify mathematical expressions and the overall layout. By adopting syntax from LaTex, user can easily type complicated mathematical equations for calculation purpose. Thus, we propose a new but familiar language called *EZMath*, including math expression syntax from LaTex, modifying unpleasant features in C, hopefully it could make programming in C with heavy mathematical manipulation more smoothly.

1.2 Description

EZMath is a LaTex syntax supported modified C language. It adopts the exact syntax for describing complex mathematical expressions from LaTex. With the ease of expressing complex mathematical expressions, programmers can focus on the bigger picture. The compiler for this project will translate EZMath program into an equivalent program written in another high-level language -- C++/C.

2 Language Definition

2.1 Data Types

Type	Definition	Example
int	integer value	int i = 0;
double	double-precision floating-point numbers	double d = 2.0;
bool	boolean value	bool flag = true;

formula	a mathematical formula using Latex syntax	formula f = $$ \sum_{i=1}^{n}{i} $$;
matrix	a matrix using Latex syntax	matrix m = $$ \begin {bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end {bmatrix} $$

2.2 Basic Arithmetic Operators

We support basic arithmetic operators for int, double, including, “+”, “-”, “*”, “/” and “^” (power). Also, we support logical operators, such as “<”, “>”, “==”, “>=”, “<=”, “&&” (and), “||” (or) and “!” (not).

2.3 Advanced Arithmetic Operators

Matrix Operators:

Operator	Definition	Example
+	matrix add	A + B
-	matrix minus	A - B
*	matrix multiply	A * B
.*	multiply the corresponding elements	A .* B
/ and \	matrix divide	A * B =C B = A \ C A = C / B
./	divide the corresponding elements	A ./ B
’	matrix transpose	A’

Formula Operators:

Operator	Definition	Example
( )	solve the formula with parameters	formula f = $$ \sum_{i=1}^{n}{i} $$; int result = f(%n, 4); // “result” has value 10 now

2.4 Control Flow

Control flows exactly follow C convention.

```
if(condition1){
}
else if(condition2){
}
else{
}

while(condition){
}

for( ; ; ){
}
```

2.5 Comment

Comments exactly follow C convention.

// for one single line

/***/ for multiple lines

3 Sample Codes

```
func main() int
{
        matrix A = $$
                        \begin{bmatrix}
                        1 & 0 & 2\\
                        -1 & 3 & 1
```

```
                    \end{bmatrix}
                $$;
        matrix B = $$
                    \begin{bmatrix}
                    3 & 1\\
                    2 & 1\\
                    1 & 0
                    \end{bmatrix}
                $$;
        matrix C = A * B;
        /* now C is $$
                    \begin{bmatrix}
                    5 & 1\\
                    4 & 2
                    \end{bmatrix}
                $$;
        */

        // Introduce a math equation with parameters
        formula f = $$
                        \frac{ \log_{x}{y} } {\cos theta}
                $$;
        double result = f(%x%y%theta, 2, 16, 60);
        // or
        double result = $$
                                \frac{ \log_{x}{y} } {\cos theta}
                    $$(%x%y%theta, 2, 16, 60);

        return result;
}
```

4 Test Plan

What we plan to do for testing is to build up a suite of tests gradually and continuously during the construction of the compiler. As the functionalities of the compiler are gradually implemented, the tests for unit functionality are as well added to the test suite library. Additionally, we will seek ways to make the testing part as automatic and fast as possible. The ways to achieve this goal may be to run the test suite in a framework, and build a one command to run the test suite. Since the rules for our language are very likely to be changed during the process, it is expected that we will make changes to the compiler very often, which will introduce bugs in the older code with high possibility. Thus regression tests will be needed and the test suite just comes in handy, because all the older tests will run to make sure no bugs are introduced when adding changes.