# CompA - Complex Analyzer

Xiping Liu(xl2639), Jianshuo Qiu(jq2253), Tianwu Wang(tw2576), Yingshuang Zheng(yz3083), Zhanpeng Su(zs2329)

Septembee 25, 2017

## 1 Introduction

The motivation for writing this language is that complex numbers are an indispensable part of mathematics and they are widely used to solve many real world engineering and physics problems. Moreover, it also possesses computational rules and properties that are consistent with real numbers. Hence, computer's great power in computing real numbers can also be used to solve complex number related problems. In most cases, computations involving complex numbers tend to require more time and techniques. Thus, a language that can support complex computations will be handy. Furthermore, it also supports matrix operations which can work with complex numbers to solve problems in areas like quantum mechanics. Our language, CompA stands for Complex Analyzer and we plan to compile it to LLVM. The following proposal details some of the specifics of our language.

## 2 Language Overview

Our language is designed to conveniently and concisely support complicated scientific and engineering projects that are built upon functions and primitive operations in Linear Algebra and Complex Analysis. The syntax of our language resembles C language (without allowing pointers), except that the mathematical representations are absorbed from MATLAB.

## 2.1 Datatypes in the Language

| Types | Examples |
|---|---|
| integer | int x = 1<br>int y = 0 |
| float | float x = 3.56<br>float y = 70.0 |
| complex number | cx x = (5, 7)<br>cx y = (3, 8.0)<br>cx z = (2.0, 9) |
| boolean | bool x = true<br>bool y = false |
| string | string x = "hello world" |
| matrix | mx x = [2, 3]<br>mx y = [2, 3.0; 5.8, 7]<br>mx z = [2, (5, 3); (7, 6), 8.8] |

The 6 data types in the above table are all the built-in data types in our language. Our language is statically-typed. Namely, you must declare the data types of your variables before you use them.

According to the examples in the above table, declaring variables in our language is fairly simple. Declaration of integer, float, boolean, and string is similar to mainstream programming languages such as C and Java. The specification of declaring complex number and matrix is shown as follows.

We use a 2-tuple surrounded by parentheses to declare complex number, where the first tuple is real part, the second tuple is imaginary part. Both parts can be either integer or float. For matrix, we include all entries in square brackets. Inside the square brackets, rows are separated by semicolon, while entries of the same row are separated by comma. Integers, float numbers, and complex numbers can be put into the same matrix.

There are also some built-in constants in our language.

| Constant Name | Mathematical Meaning |
|---|---|
| PI | $\pi$ approximately 3.14159 |
| INF | $\infty$ |
| E | Euler's number approximately equals to 2.71828 |

## 2.2 Operators

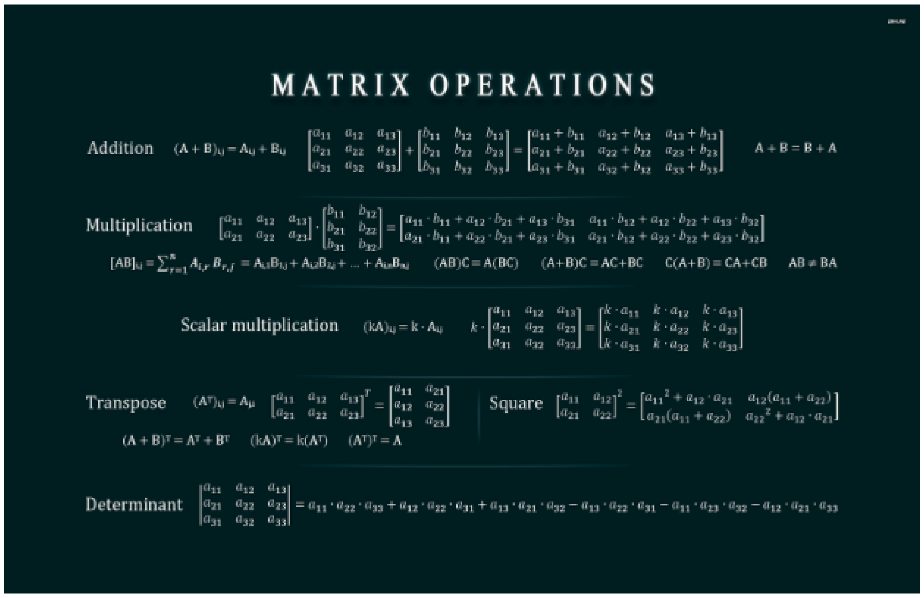
| Logical Operators | Operation Type |
|---|---|
| == (is equal to), < (is less than), > (is larger than) , <= (is less than or equal to ), >= (is more than or equal to) | numeric relational |
| && (Logical AND), \|\| (Logical OR), ! (Logical NOT) | logical |

Assume all numbers written in the form z = a + ib which we call it regular form. For all real numbers, b is simply 0.

| Complex Number Operators | Operations | Examples |
|---|---|---|
| + | addition | z1 + z2 = (a + c) + i(b + d) |
| - | substraction | z1 - z2 = (a - c) + i(b - d) |
| * | multiplication | z1 * z2 = (ac - bd) + i(ad + bc) |
| / | division | z1 / z2 =((ac - bd) + i(bc - ad)) / (c^2 + d^2) |
| ^ | power | z^n = (a + ib)^n |
| exp() | exp power | exp(z) = e^a(cos(b) + i(sin(b))) |
| conj() | conjugate | conj(z)= a - ib |
| \| \| | absolute value | \|z\| = (a^2 + b^2)^(1/2) |
| e | scientific notation | 5.12e-31 =5.12*10^(-31) |

Our language also supports matrix operations:

| Operators | Operations |
|---|---|
| + | addition |
| * | multiplication |
| tp() | transpose |
| dt() | determinant |
| tr() | trace |
| <x \| y> | inner product |



MATRIX OPERATIONS

Addition $(A+B)_{ij} = A_{ij} + B_{ij}$ $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{12} & a_{33}+b_{33} \end{bmatrix}$ $A+B=B+A$

Multiplication $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} a_{11}\cdot b_{11} + a_{12}\cdot b_{21} + a_{13}\cdot b_{31} & a_{11}\cdot b_{12} + a_{12}\cdot b_{22} + a_{13}\cdot b_{32} \\ a_{21}\cdot b_{11} + a_{22}\cdot b_{21} + a_{23}\cdot b_{31} & a_{21}\cdot b_{12} + a_{22}\cdot b_{22} + a_{23}\cdot b_{32} \end{bmatrix}$

$[AB]_{ij} = \sum_{r=1}^{n} A_{i,r} B_{r,j} = A_{i1}B_{1j} + A_{i2}B_{2j} + \dots + A_{in}B_{nj}$ $(AB)C = A(BC)$ $(A+B)C = AC+BC$ $C(A+B) = CA+CB$ $AB \neq BA$

Scalar multiplication $(kA)_{ij} = k \cdot A_{ij}$ $k \cdot \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} k\cdot a_{11} & k\cdot a_{12} & k\cdot a_{13} \\ k\cdot a_{21} & k\cdot a_{22} & k\cdot a_{23} \\ k\cdot a_{31} & k\cdot a_{32} & k\cdot a_{33} \end{bmatrix}$

Transpose $(A^T)_{ij} = A_{ji}$ $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}^T = \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \\ a_{13} & a_{23} \end{bmatrix}$

$(A+B)^T = A^T + B^T$ $(kA)^T = k(A^T)$ $(A^T)^T = A$

Square $\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^2 = \begin{bmatrix} a_{11}^2 + a_{12}\cdot a_{21} & a_{12}(a_{11}+a_{22}) \\ a_{21}(a_{11}+a_{22}) & a_{22}^2 + a_{12}\cdot a_{21} \end{bmatrix}$

Determinant $\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11}\cdot a_{22}\cdot a_{33} + a_{12}\cdot a_{22}\cdot a_{31} + a_{13}\cdot a_{21}\cdot a_{32} - a_{13}\cdot a_{22}\cdot a_{31} - a_{11}\cdot a_{23}\cdot a_{32} - a_{12}\cdot a_{21}\cdot a_{33}$

## 2.3 Control Flow

**a. if statement**

Handles conditional statements

```
if (<condition>) {
    <statements>
    } else if (<condition>) {
        <statements>
    } else {
            <statements>
}
```

**b. for loop**

Handles loop operations

```
for (int i = 0; i < 20; i++) {
    <statements>
}
```

**c. while loop**

Another way to handle loop operations

```
int i = 0;
while (i < 10) {
        <statements>;
        i ++;
}
```

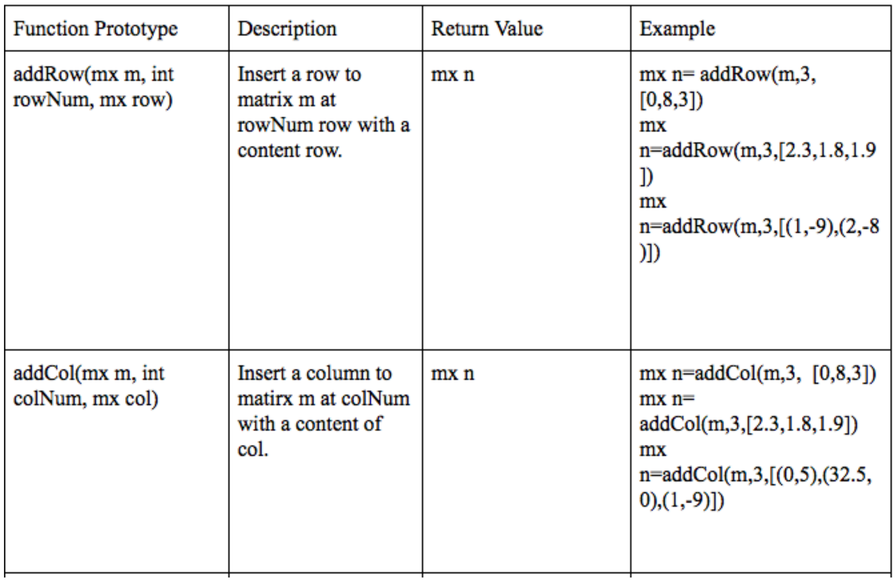
**d. break**
Terminate a loop(usually with a condition), and the program resumes at the next statement following the loop

```
int i = 0;
while (i < 10) {
          <statements>;
          i++;
     if (i > 6) {
     break;
     }
}
```

**e. continue**
When a continue statement is encountered inside a loop, control jumps to the beginning of the loop for next iteration, skipping the execution of statements inside the body of loop for the current iteration

```
for (i = 0; i < 10; i++) {
          if (i == 6) {
          continue; // when i = 6, <statements> will be skipped
          and the control will return to the loop with i = 7
          }
          <statements>;
     }
}
```

## 2.4 Built-in function



| Function Prototype | Description | Return Value | Example |
|---|---|---|---|
| addRow(mx m, int rowNum, mx row) | Insert a row to matrix m at rowNum row with a content row. | mx n | mx n= addRow(m,3, [0,8,3])<br>mx n=addRow(m,3,[2.3,1.8,1.9])<br>mx n=addRow(m,3,[(1,-9),(2,-8)]) |
| addCol(mx m, int colNum, mx col) | Insert a column to matirx m at colNum with a content of col. | mx n | mx n=addCol(m,3, [0,8,3])<br>mx n= addCol(m,3,[2.3,1.8,1.9])<br>mx n=addCol(m,3,[(0,5),(32.5, 0),(1,-9)]) |

| | | | |
|---|---|---|---|
| colLen(mx m) | Shows how many columns are there in the matrix m. | int | int colNum= rowLen(m) |
| rowLen(mx m) | Shows how many rows are there in the matrix m. | int | int rowNum= rowLen(m) |
| delRow(mx m, int rowNum) | Delete a certain row at rowNum. | mx n | mx n= delRow(m,2) |
| delCol(mx m, int colNum) | Delete colNum column from matrix m. | mx n | mx n= delCol(m,2) |
| addEle(mx m, int rowNum, int colNum, cx complexnumber) | Insert a specific entry to matrix m at the place of rowNum row, colNum column. | mx n | addEle(m,3,2,3)<br>addEle(m,3,2,3.6)<br>addEle(m,3,2,(3,2)) |
| getRow(mx m , int rowN) | Get all the elements in rowN row from matrix m. | mx row | mx row = getRow(m,3) |
| getCol(mx m , int colN) | Get all the elements colN column from matrix m. | mx col | mx row = getCol(m,3) |
| eigenValue(mx m) | Get the eigen value of a matrix m. | cx ev | cx ev = eigenValue(mx m) |
| eigenVector(mx m, cx eigenvalue) | Get the eigen vector of a matrix m with a given eigen value eigenvalue. | mx e | mx e = eigenVector(mx m, (6,0) ) |
| euler(cx a) | Change the form of a complex number into a exponential form. | mx result [magnitude,angle] | mx a = euler((1,3)) |
| exp(int/float/cx a, int/float/cx b) | Calculate the exponential of a to the b. | int/float/cx result | int result = exp(1,2)<br>float result = exp(1.9,2)<br>cx result = exp((1,2),2) |
| print() | print the result. | | print("hello world")<br>print(m)//print out a matrix<br>print( getRow(m,3))//print out a specific row of a matrix |

### 2.5 User Defined Function

Users can create their own functions by using primitive data types and built-in functions. The syntax is C-like.

**Example1**

```
int get_zero(){
    return 0;
}
```

**Example2**

```
bool age_compare(int age1, int age2){
        if (age1 >= age2){
        return true;
    } else {
            return false;
    }
}
```

**Example3**

```
mx add_matrix(mx matrix1, mx matrix2){
        return matrix1 + matrix2;
}
```

**Example4**

```
void printTrace(mx m){
        int trace = tr(m);
        print(trace);
}
```

## 3 A Sample Program

Below is an example program in our language CompA, which solves a problem in Quantum Mechanics. It uses a user defined function spinXExpectation(int t) to calculate the expectation value of the spin angular momentum in x direction of a wave function at time t. In the main() function, spinXExpectation(int t) is calculated at t = 0 and 5 and the results are printed out to the console window using the built-in function print().

```
/* start of the program */
/* global variables */
static float h_bar = 1.05457e-34;
static float B_0 = 1e-5;
static float alpha = PI/6;
static float gamma = -1.6e-19/9.11e-31
```

```
/* main functoin */
int main(mx arg) {
    print("Spin angular momentum in x direction at time t = 0");
    mx expectationValue = spinXExpectation(0);
    print(expectationValue);

    print("Spin angular momentum in x direction at time t = 5");
    expectationValue = spinXExpectation(5);
    print(expectationValue);

    return 0;
}

 /* user defined function */
mx spinXExpectation(float t) {
    mx waveFunction = [cos(alpha/2)exp((0,1)*gamma*B_0*t/2);
    sin(alpha/2)exp(-(0,1)*gamma*B_0*t/2)]; /* complex
    matrix declaration */

    mx S_x = [0, 1; 1, 0];
    return tp(conj(waveFunction))*S_x*waveFunction;/* complex
    matrix multiplication */
}

/* end of the program */
```