CompA Language

Language Reference Manual

Manager: Xiping Liu (xl2639)

Language Guru: Jianshuo Qiu (jq2253)

System Architect: Zhanpeng Su (zs2329), Tianwu Wang (tw2576)

Tester: Yingshuang Zheng (yz3083)

Contents

1. Introduction

Welcome to the CompA reference manual! Our team members' shared experience in complex number related mathematical computations motivated us to create this language. Complex numbers are widely used in areas like signal processing, quantum mechanics, and multi-dimensional data analysis. These computations are usually complicated and time-consuming. Hence, CompA is targeted to solve those problems efficiently and correctly.

2. Types

2.1 Basic Data Types

Types	Examples
integer	int x = 1 int y = 0
float	float x = 3.56 float y = 70.0
complex number	cx x = (5, 7) cx y = (3, 8.0) cx z = (2.0, 9)
boolean	bool x = true bool y = false
string	string x = "hello world"
matrix	mx x = [2, 3] mx y = [2, 3.0; 5.8, 7] 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	π approximately 3.14159
INF	∞
E	Euler's number approximately equals to 2.71828

3. Lexical Conventions

3.1 Identifiers

Identifiers are tokens that used to store data of dynamic type. Identifiers must start with lowercase letter, followed by any combinations of numbers, letters and underscores.

3.2 Keywords

Keywords are reserved identifiers that cannot be modified by others. Below are the keywords used in the language:

int float cx bool string mx if else for

while break continue return print true false

3.3 Constants

Constants are pre-defined values for easier access by users. Some examples:

PI INF E

3.4 Comments

Comments are similar to C language, in which /* starts comments and */ ends comments. Anything between /* and */ will be ignored by the syntax. Comments cannot be nested.

3.5 Operators

There are three sets of operators: logical operators, complex numeric operators and matrix operators. There will be some overlap on numeric and matrix 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
m[rowNum, columnNum]	matrix reference

3.6 Precedence

The numeric calculation precedence is just like normal mathematical calculations. With introduction of matrix into the language, matrix calculations will be the highest precedence

4. Syntax Notation

4.1 Expressions

4.1.1 Primary Expressions

Primary Expressions are the most basic expressions which make up more complex expressions. These include identifiers, constants, strings, or expressions in parentheses.

4.1.2 Postfix and Prefix Expressions

Postfix expressions in CompA include the following:

expression[expression]

expression(parameter-list)

expression++

expression--

And prefix expressions include the following:

++expression

--expression

4.1.3 Matrix References

Matrix elements are referenced through postfix expressions of the form:

expression[expression]

where the first expression is the identifier of an initialized matrix. In the case of a 1-dimensional matrix, the expression in brackets is simply an integer value. For a 2-dimensional matrix, the expression in brackets is a pair of comma separated integer values.

4.1.4 Function Calls

Function calls are postfix expressions of the form:

expression(parameter-list)

where expression is an existing function identifier and the optional parameter-list consists of comma-separated expressions that are passed as the function parameters.

4.2 Declarations

4.2.1 Type Specifiers

- void
- String

  declaration of a string
- cx

declaration of a complex number

- boolean

  declaration of a boolean variable

4.2.2 Matrix Declarations

mx[# of rows, # of columns] name

Define a matrix/vector by specifying its name. We use `;` to separate rows in matrix, so if only 1 line, then it is a vector. Otherwise if it has more than one line, then it is a matrix. We specify the size of matrix by specifying the number of rows and number of columns of the matrix.

4.2.3 Function Declarations

T name (T arg, …) { statements }

The function declaration above specifies the return type of the function which is denoted as T. It also specifies the name of the function and the arguments that the function take. Curly braces are used to group statements of the function.

4.3 Initialization

4.3.1 complex number

We can assign a complex value to a complex variable using "=" while declaring it. If we do not do that, the complex variable will be (0, 0) by default.

Example:

```
cx c1 = (0, 0);
cx c2 = (1.0, 2.3);
```

4.3.2 boolean

Use "=" to assign a boolean variable true/false while declaring it. It is by default false.

Example:

```
boolean a = true;
```

4.3.3 string

Use "=" to assign a string enclosed by double quotes "" while declaring it. It is by

default an empty string “”.

Example:

string str = “CompA”;

4.3.4 matrix

Use “=” to assign a matrix representation enclosed by [] while declaring it. Its default initialization depends on the type of its contents. If it is of type complex numbers, then by default it will contain complex number (0, 0) for each of its entry.

Examples:

mx<cx>[1,4] m1 = [(1, 1), (1.1, 0), (0, 0), (-1, -4)];

mx<cx>[2,1] m2 = [(1, 0); (2, 9.2)];

4.4 Statements

4.4.1 Expression Statement

Each expression statement consists of only one expression (introduced in 4.1, usually an assignment or function call). In CompA, each expression statement is followed by a semicolon:

expression;

The compiler will report error if there is no semicolon after an expression statement. However, it is OK if the expression statement before a semicolon is an empty string.

4.4.2 Compound Statement

A compound statement is a sequence of statements (which may be a compound statement) enclosed in braces. A compound statement is of the form:

{

expression;

expression;

{ *expression*; *expression*;}

}

is a compound statement. There is no semicolon following a compound statement. If a variable is declared within a compound statement, the scope of that variable is only limited within the corresponding compound statement.

4.4.3 Selection Statement

The selection statement controls the program flow according to the boolean value of certain conditions. The keywords for selection statement in CompA are *if*, and *else*. A selection statement is of the form:

if(*condition*){
  then-statements
}else{
  else-statements
}

Here, condition must be a boolean expression. If *condition* is evaluated to be true, the *then-statements* run. If *condition* is evaluated to be false, the *else-statements* run. In an if statement that is not followed by else, if *condition* is true, the *then-statements* run. Otherwise, the program control flows to next statement after the if statement. Both *then-statements* and *else-statements* are a compound statement.

4.4.4 Iteration Statement

Iteration statements are used to create loops and execute embedded statements multiple times. The keywords used in iteration statements in CompA are *for*, *while*, *break*, and *continue*. The form of a for-loop statement is in this way:

for(*initializer*; *condition*; *iterator*){
  embeded statement;
}

Here, the *initializer* sets the initial condition, which is an assignment. The statement for the *initializer* runs only once. The *condition* provides a boolean value to decide if the loop continues or finishes. The *iterator* updates what changes after each iteration. The *iterator* can be an assignment (including prefix/postfix increment/decrement), or an invocation for a function. The form of a while-loop statement is in this way:

initializer;
while(*condition*){
  embedded statement;
  iterator;
}

where *initializer*, *condition*, and *iterator* are defined the same as in for-loop statement. *Break* is used to stop the iteration and make the program flow to next statement. *Continue* is used to jump to next iteration without executing the rest of the statements in the embedded statement after continue. *Break* and *continue* are usually placed inside a selection statement in the loop.

5. Standard Library Functions

5.1 Math

Function Prototype	Description	Return Value	Example
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) float result = exp(1.9,2) cx result = exp((1,2),2)

cos(cx z)	Calculate the cosine of z.	cx	-cos(zi)=cosh(z)= A
sin(cx z)	Calculate the sine of z.	cx	-sin(zi)=sinh(z)= A

5.2 Complex Number

Function Prototype	Description	Return Value	Example
euler(cx a)	Change the form of a complex number into a exponential form.	mx result [magnitude,angle]	mx a = euler((1,3))

5.3 Matrix

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]) mx n=addRow(m,3,[2.3,1.8, 1.9]) 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]) mx n= addCol(m,3,[2.3,1.8,1.9]) 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	Insert a specific entry to matrix m at the place of	mx n	addEle(m,3,2,3) addEle(m,3,2,3.6) addEle(m,3,2,(3,2))

colNum, cx complexnumber)	rowNum row, colNum column.		
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))

5.4 I/O

Function Prototype	Description	Return Value	Example

print()	print the result,with a special character “\n” to indicate a newline.		print(“hello world”)
			print(m)//print out a matrix
			print(getRow(m,3))//print out a specific row of a matrix