eGrapher Language Reference Manual

Long Long: ll3078@columbia.edu
Xinli Jia: xj2191@columbia.edu
Jiefu Ying: jy2799@columbia.edu
Linnan Wang: lw2645@columbia.edu
Darren Chen: dsc2155@columbia.edu

Content

1.Introduction

eGrapher offers an innovative way of drawing art with pinpoint accuracy. The language gives the user a wide set of tools to accurately illustrate detailed drawings compared to simple and rigid pre-installed tools such as paint on Windows. The goal of eGrapher is to make drawing on digital systems easier as well as allow for users to draw more complicated paintings through mathematical functions. The project will allow us to also better understand the intersection of mathematics and art. Users will be able to use simple syntax and math expression to draw graphs, diagrams, and more complicated objects through simple objects.

The following is a reference manual for using eGrapher. It describes in detail the ideas and design thoughts behind lexical conventions, basic types scoping rules, built-in functions, and also gives a sample program with its output.

2. Types and Literals

eGrapher has a set of types and literals which are similar to those of most programming languages. Since eGrapher compiles into C, most of these specifications match those of the compiled language. Types in eGrapher can be largely classified into primitive types and nonprimitive (userdefined) types. List, named tuples and graph nodes fall into the latter. While primitive data types are passed by value, nonprimitive data types are always passed by reference

2.1 Primitive types

2.1.1 Integers

An integer literal (e.g. 0, 12, 1024, etc.) is a sequence of digits and is taken to be decimal.

In eGraph, the keyword *int* indicates an integer. And all integers are 4 bytes in size thus can represent any signed integer in the range [2147483647, +2147483647].

int **a** = 10;

2.1.2 String

String literals are defined as a sequence of symbols. They are identified as being surrounded by double quotes: “hello world”. For variable instantiation, the *string* keyword is used to state type. In order for double quote (“) or single quote (‘) symbols to be a part of the string, they must be preceded with a backslash (e.g. “he said \“Hi\” ”)

string **a** = “Hello!”;

2.1.3 Floating Point Numbers

A floating literal consists of an integer part, a decimal point and a fraction part,The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing.

float **a** = “1.1”;

2.1.4 Boolean

Boolean represents the most basic unit of logic. It takes either the value of *True* or *False*. Variables are typed using the keyword *bool*.

bool **a** = *True;*

2.2 Lists

The list data type is signified using the left and right braces “[ ]”. List types are like arrays in that they allow for manipulation using indexing, but are also variable in size. They automatically resize to the amount of data in the list.

list<*int*> **a** = [1,2,3,4,5];
list<string> **b** = [];

Lists can be mutated to include more or less values using built-in function: *add* and *del*.
The *add* keyword appends the value to the end of the list. The *del* keyword deletes the object at the index specified. The length keyword returns the number of element in a list:

a.*add*(6);
a.*del*(2);
a.*length*();

2.3 Comment
The characters /* introduce a comment, which terminates with the characters */.

3. Expressions

3.1 Primary expressions

3.1.1 **identifier**

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore ‘‘_’’ counts as alphabetic. Upper and lower case letters are considered different.

3.1.2 **literals**

A literal refers to a data type literal. The integer, floating point, boolean, string literals have a range of possible values defined in the previous section.

3.1.3 grouping expressions: (*expression*)

The set of opening and closing parenthesis can be used to group an expression into a higher precedence. The type and value of this primary expression are that of the contained expression.

3.1.4 list index access: **identifier** [*expression*]

An identifier followed by expression surrounded by opening and closing brackets, denotes element index access. The identifier is a list, and the expression must be a nonnegative integer, and the type and value of the primary expression are that of the element at index.

3.2 Unary operators

The operators in this section have right-to-left associativity

3.2.1 - *expression*

The - operator denotes arithmetic negation for integer and floating point types.

3.2.2 !*expression*

The ! operator denotes boolean negation for boolean types.

3.3 Multiplicative Operators

The operators in this section have left-to-right associativity. Both expressions can be either integers or floating point numbers.

3.3.1 *expression* * *expression*

The * denotes multiplication of integers or floating point numbers.

3.3.2 *expression* / *expression*

The / denotes division of integers or floating point numbers.

3.3.3 expression ^ expression

The ^ denotes power of integers or floating point numbers.

3.3.4 *expression* % *expression*

The % denotes module operator. The number produced by the evaluation of the operator will have the same sign as the dividend (left) operand.

3.4 Additive Operators

The operators in this section have left-to-right associativity.

3.4.1 *expression* + *expression*

The + operators denote integer and floating point addition.

3.4.2 *expression* - *expression*
The - operators denote integer and floating point subtraction.

3.5 Relational operators
The operators in this section have left-to-right associativity. The result of relational operator evaluation is a boolean type, the value of which corresponds to the truth value of the expression. Relational operators support integer or floating point operands, but not a combination of both.

3.5.1 *expression* < *expression*, *expression* > *expression*
The < and > operators denote the less than and greater than comparison.

3.5.2 *expression* <= *expression*, *expression* >= *expression*
The <= and >= operators denote the less than or equal to and greater than or equal to comparison.

3.6 Equality Operators
The operators in this section have left-to-right associativity. The result of equality operator evaluation is a boolean type, the value of which corresponds to the truth value of the expression.

3.6.1 expression == expression, expression != expression.
The == and != operators denote the equal to and not equal to comparison, which support integer and floating point operands. If an integer is compared to a floating point, the integer is automatically promoted to a floating point.

3.7 Boolean AND
3.7.1 expression && expression The && operator denotes the boolean AND operation. It has lefttoright associativity and only supports boolean operators; type conversions or demotions are not supported.

3.8 Boolean OR
3.8.1 expression | | expression The | | operator denotes the boolean OR operation. It has lefttoright associativity and only supports boolean operators; type conversions or demotions are not supported.

3.9 Assignment
3.9.1 **identifier** = *expression*
The = operator denotes assignment. It has right-to-left associativity. The identifier must have the same type as the expression; type conversions, promotions, or demotions, are not supported.
3.9.2 **identifier** (+ or - or *or /) = *Integer/float*
Self increment of identifier

identifier = *Integer/float* + **identifier**

4. Statements

4.1 Conditional

Conditional statements include if and if/else statements and have the following form:

selection-statement:

if (*expression*) **statement**

if (expression) **statement** *else* **statement**

Selection statements choose one of a set of statements to execute, based on the evaluation of the expression. The expression is referred to as the controlling expression.

The controlling expression of an if statement must have scalar type. For both forms of the if statement, the first statement is executed if the controlling expression evaluates to nonzero. For the second form, the second statement is executed if the controlling expression evaluates to zero. An else clause that follows multiple sequential else-less if statements is associated with the most recent if statement in the same block (that is, not in an enclosed block).

Example:

if (*expression*)

statement

if (*expression*) **statement**

else **statement**

if (*expression*)

{

statement

if (*expression*) **statement**

}

else **statement**

The first example indicates that later if statement is corresponding to the else statement.

The first example indicates that former if statement is corresponding to the else statement, because the later if statement is inside the scope of the first statement.

4.2 *while*

while statement execute the attached statement (called the body) repeatedly until the controlling expression evaluates to zero. In the for statement, the second expression is the controlling expression. The format is as follows:

iteration-statement:

while (expression) {**statement**}

The controlling expression of a while statement is evaluated before each execution of the body.

4.3 *for* loop

The for statement has the following form:

for (expression1; expression2; expression3) {**statement**}

The first expression specifies initialization for the loop. The second expression is the controlling expression, which is evaluated before each iteration. The third expression often specifies incrementation. It is evaluated after each iteration.

This statement is equivalent to the following:

*expression1; while(expression2) {***statement** *expression3;}*

One exception exists, however. If a continue statement is encountered, *expression3* of the for statement is executed prior to the next iteration.

Any or all of the expressions can be omitted. A missing *expression2* makes the implied while clause equivalent to while. Other missing expressions are simply dropped from the previous expansion.

4.4 *break*

The *break* statement can appear only in the body of an iteration statement. It transfers control to the statement immediately following the smallest enclosing iteration or switch statement, terminating its execution.

4.5 *continue*

The continue statement can appear only in the body of while or loop statement. It causes control to pass to the loop-continuation portion of the smallest enclosing while, or for statement; that is, to the end of the loop. Consider each of the following statements:

```
while (...)
{
..
continue;
}
for (...) {
...
continue;
}
```

4.6 *return*

A function returns to its caller by means of the return statement. The value of the expression is returned to the caller (after conversion to the declared type of the function), as the value of the function call expression. The return statement cannot have an expression if the type of the current function is void. If the end of a function is reached before the execution of an explicit

return, an implicit return (with no expression) is executed. If the value of the function call expression is used when none is returned, the behavior is undefined.

5. Functions

Functions are defined with the *fun* keyword:
fun **Function-name** (*argument-type arg1, argument-type arg2,...*) { **statement** }
They should be defined before being used. Arguments are passed by reference, and are passed by value for other data types, and passing arguments requires the specification of their types. Types include integer, string, list and other build-in types. There can be multiple arguments given to a function. Each argument's type should be defined prior to its name when defining a function. The type of the argument should not be specified when calling a function. The program checks the types of arguments as the function is being called. The return keyword returns value. The function will terminate when it sees the return keyword, and the program will return to where the function was called. Return type can be any types that are supported in eGrapher, but it needs to match the predefined type of the function. If the return value is type void, the function returns nothing.

6. Built-in Function

6.1 *print*(**identifier**) or *print*(**literals**)

Prints an integer, a floating point number or a string.

print(**s**);

6.2 Plotting

6.2.1 *plot*(**identifier1**, **identifier2**, **identifier3**, **identifier4**)

Show the graph plotted by given two lists **identifier1** and **identifier2**.
Identifier3 and **identifier4** are String indicating style and color of dot and lines separately. If styles are not set, the default setting of marker is black dot and black line.
plot(**X**,**Y**, "o blue", "- blue");
plot(**X**,**Y**, "+ red", ": red");
plot(**X**,**Y**, "", ""); is equal to *plot*(**X**,**Y**, ". black", "- black");

Identifier3 and identifier4 style guide

Line Style Specifier	Description

`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker Specifier	**Description**
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond

6.2.2 *hold*()

If *hold*() is used, next plot will be add on the previous graph instead of creating a new one.

hold();

6.3 Type Conversions

6.3.1 *float2int*(**identifier**)

If the floating point number has a fractional portion, this will be truncated. If the floating point number is too large to represent as an integer

6.3.2 *int2float*(**identifier**)

Takes an identifier of type integer as an argument and returns the floating point representation.

6.3.3 *string2int*(**identifier**)

Takes an identifier of a string, and converts it to an integer.

6.3.4 *string2float*(**identifier**)

Takes an identifier of a string, and converts it to a float point number.

7. Program Structure

Every statement in eGrapher must belong to either the main function or defined external functions outside of it. The external functions are used to write simple programs that can be executed in the main function. As a result, only functions, variables, and objects called in the main function will be executed and stored. In addition, the name main in the global name space is reserved and does not need a return statement.

Any program will consist of a sequence of external definitions. This is designed to give the user more flexibility. External definitions may be given for function, for simple variables, and for lists. They are used to help declare and allocate storage for objects when called upon.

8. Sample Program

```
/* Bubble sort code */
int main()
{
 list<int> array = [3,5,2,1,4];
 int c, d, swap;
 int n = array.length();
 list<int> array_x;
 for (d = 1 ; d <= array.length(); d+=1){
  array_x.add(d);
 }
 plot (array_x,array, “o red”, “- red”);
 hold;
 for (c = 0 ; c < ( n - 1 ); c+=1)
 {
  for (d = 0 ; d < n - c - 1; d+=1)
```

```
  {
   if (array[d] > array[d+1]){
     swap = array[d];
     array[d] = array[d+1];
     array[d+1] = swap;
   }
  }
}
print("Sorted list in ascending order:\n");
for ( c = 0 ; c < n ; c+=1 ){
  print(array[c]);
  print(" ");
}
plot( array_x, array, "x blue", "- blue");
```

9. Complete Table of Keywords

Keywords	**Description**
int	signed integer value
string	list of characters
float	floating point number
bool	boolean value
list	Sequence of numeric, Boolean, or other types.
Null	Empty address
fun fun_name (*type* **arg1**, *type* **arg2**, …)	defines an external function
main(argv,argc)	the main script for execution (with command line arguments)
l.*add(x)*	adds an element to the end of list
l.*del(x)*	deletes the element in the list
l.length()	Return length of a list

[fun(n) for x in l]	iterative definition of a list
plot(X,Y,"arg1" , "arg2")	Plot graph
return val	returns value
for(type name; conditional; postloop) {}	c-style for loop
for variable in list {}	python-style for loop
while(conditional) {}	while loop
if(condition) {} *elif* (*condition*){} *else* {}	conditional statement
break	Stop iteration
continue	Continue iteration
True	Boolean value is true
False	Boolean value is false