JEoMC Language Reference Manual

Melody Hsu, Jeremy Lu, Emma Schwartz, Connie Zhou

Programming Conventions (i.e. indentation, whitespace, comments)

Identifiers

An identifier in JEoMC is a name given to a variable, method, interface, package, or class by the user. Each identifier must be unique and cannot be used more than once. An identifier must begin with a letter, followed by either numbers, letters, or underscores. There is no preference in using uppercase or lowercase letters. There are specific naming conventions associated with identifiers, represented by the regular expression below.

```
['a' - 'z' 'A' - 'Z']['a' - 'z' 'A' - 'Z' '0' - '9' '_']*
```

Whitespace

In JEoMC, whitespace is not syntactically important if the rest of the statement is semantically valid. Whitespace is defined as any space character, including the tab character. However, it is good practice for statements to not be written on a single line and instead be written on multiple lines, as it is easier to comprehend. A good example of whitespace use can be found below.

```
<part_of_method>
      <rest_of_method>
```

Comments

JEoMC supports comments that are in-line and block comments as well. The syntax can be found below.

```
#^# In-Line Comment

#^ Block Comment ^#

#^
  ^ Multi-line Block Comment
^#
```

Data Types/Structures

- JEoMC supports fixed-point numerical quantities only - these are represented as integers, or `int`. Fixed-point numerical quantities can be declared using any sequence of numerical characters, although leading 0s will be discarded by the parser (i.e., the quantities 00123 and 123 are mathematically equivalent in JEoMC).

- JEoMC supports scalar and vector quantities. Any object type can be stored in a list, as long as all objects are of the same type. For example, to declare a vector of colors, the user would define:
    - `color[] colorVec = ['R','B','Y']`
    - where color is the type of the object, and 'R', 'B', and 'Y' are strings that correspond to the built-in color tuples for red, blue and yellow respectively.

Reserved Keywords

JEoMC has a specific set of reserved keywords that may not be used as identifiers.

Statements and Blocks

The following keywords specify types of statements or blocks.

```
fun val (?) fix depending on below
```

Control Flow

The following keywords specify types used for control flow.

```
if then else for while continue break return
```

Types

Primitive types are specified by specific names used for declaration.

```
int double float bool char String rgb shape
```

Built-in Functions

The following keywords specify built in functions.

```
draw size thickness gradation color border_color print
printf
```

Standard Operations (operators?)

- JEoMC supports the following operators:
    - +, -, *, /, %, |, =, @, .
- The assignment operator (=) is a binary, right-associative operator that assigns the value on the right hand side to the identifier on the left hand side. All types can be assigned - fractal objects, integers, colors, etc.
- The attribute operator (.) <object>.<attribute> returns the attribute of that object. This will be specified for fractal objects only, such as size, color, iteration, and portion of the fractal associated with a particular iteration. Integers and String objects will not have any associated attributes.
- All standard mathematical operators are available to the user to compute simple mathematical quantities between two integers, like 5 + 4, 3 / 2, etc. All mathematical operators are left-associative, and multiplication and division have a higher precedence than addition and subtraction.
- Between two integers, we have
    - (+) integer division
    - (-) integer subtraction; negative numbers are not supported
    - (*) integer multiplication
    - (/) integer division; the result will be the floor of the resulting quotient

- The user also has the option to use operators between fractals and integers. Below, we define operators that can act on fractals. Some operators, such as the plus sign, are overloaded; others are newly defined and only valid on fractals. Operators are commutative.
    - (+) <fractal object> + <integer> allows the user to increase the number of iterations from the current fractal's configuration. For example, if the fractal object in the expression has already been generated for three iterations, and the number 5 is added to this fractal object, the final configuration would be the fractal object drawn for 8 iterations.
    - (-) <fractal object> - <integer> allows the user to decrease the number of iterations from the current fractal's configuration, similarly to the above. This operation does not support reducing the number of iterations below 1.
    - (%) <fractal object>% is a unary operator that mirrors the fractal object about the x-axis.
    - (|) <fractal object>| is a unary operator that adds decorative lines to critical vertices of the fractal object, increasing the complexity and aesthetic quality of the fractal.
    - (@) <fractal object>@ is a unary operator that increases the ornamentation level of the fractal by adding additional polynomial factors to the base curve.

Functions Available to User

General function declaration is defined by the following syntax:

```
fun name(<type1> arg1, <type2> arg2...):
...
    return image or vector
```

The user has the following built-in functions available to them. Most of them have to do with the visual aspects of the fractal images:

- draw(shape triangle):
  Starts drawing a shape. In this example, it's a triangle.
- size(int scale):
  Adjusts the size of the initial fractal that's created
- thickness(int width):
  Changes the thickness of the fractals lines
- gradation(rgb a, color b):
  Chooses colors that a gradation can be done with. The type "rgb" is an RGB tuple.
- color(rgb a):
  Makes the fractal a specific color.
- border_color(rgb a):
  Specifies the border color.
- print(String a):
  Prints a string.
- printf(int scale):
  Prints a portion of the fractal with a specific size.

Program Structure (loop syntax, if/else statements)
JEoMC supports if else conditionals as well as looping functionality, as below:

`if and else` statements
The if and else statements test a condition and based on its result, executes one part or the other of the program. In any given time, only one branch of the if/else statement should be executed.

```
bool wantCircle = true;
while (wantCircle)
  draw(circle);
else
  draw(triangle);
```

If desired, you may chain ifs together with elses to test for multiple conditions at one time.

`while` Statement
The while statement is a looping statement that evaluates the conditional surrounded in parentheses after it. If the conditional evaluates to true, the statements following will be evaluated. After the final statement within the loop has executed, the initial conditional is reevaluated, and this will go on until the conditional evaluates to false.

```
int count = 0;
while (count < 5)
  draw(triangle);
  count = count + 1;
```

`for` Statement
The for statement is another looping statement that allows for easier syntax when there is a need to keep track of a counter.
For simplicity, we will only allow a for loop if all three steps (initialize, test, and step) are filled out. This means you can't leave out for example the test part, to start in an infinite loop. If there is need for that, please consider the while statement.

```
int i;
for (i = 0; i < 5; i = i+1)
  draw(triangle);
```

`break`, `continue`, and `return` Statements
These keywords play a big role in looping logic in that they control the flow of the program. Here's an overview of what each keyword does:

- **`break`**

A break statement will cause any loop to terminate immediately without reevaluating the conditional

- **`return`**

Return functions similarly to break in that it causes a loop to terminate, but it simultaneously returns a value for the function and then quits the function

- **`continue`**

A continue statement will allow a user to skip any statements following the current one, and immediately reevaluate the conditional at the current state

Exceptions

- To be discussed. We expect these to typically consist of type errors.