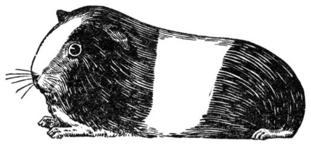
bugsy
Reference Manual

PLT Spring 2021

Ben Snyder Sofía Sánchez Zárate Evan Tilley

Michael Winitch Jason Cardinale

February 24, 2021



Contents

1 Overview

The bugsy language is a statically typed object oriented programming language focused on creating 2D graphics and animations. It is loosely inspired by p5.js and Processing. The syntax is a mixture of C and SwiftUI. bugsy has built in shape and polygon classes allowing users to create shapes, along with accompanying animation functions. The language is designed to be of assistance to those who want to work with visualization, but may not necessarily have a strong programming background. When a program is run, the language will compile an executable that will run the animation. bugsy interfaces with OpenGL, abstracting some of the library's complexities to make a simple graphical language.

2 Comments

Comments in bugsy are used to document code and are ignored by the parser, written solely for readability purposes.

2.1 Single Line Comments

Single line comments in bugsy are creating using two forward slashes.

```
// This is an example of a single line comment
```

2.2 Multi-line Comments

bugsy also supports multi-line comments. To start a multi-line comment type /* and close it with */

```
/*
   This is an example
   of a multi-line comment
*/
```

3 Reserved Words

bugsy has words that are reserved for specific uses and thus cannot be used in a typical manner. The following is a list of reserved words that can not be used for the names of variables, classes, functions:

- if
- elif
- else
- while
- for
- continue
- break
- try
- catch
- raise
- class
- True
- true
- False
- false
- not
- num
- bool
- char
- string
- array
- null
- and
- or
- void

- pt
- ellipse
- line
- shape
- triangle
- spline
- square
- polygon
- rect
- regagon
- circle
- canvas

4 Data Types

bugsy has a variety of data types, each suited for representing different types of information.

4.1 Primitive Data Types

num is used to express any number, including both integers and floating point numbers.

```
num a = 10;
num b = -12.35;
```

char is used to store a single ASCII character. The type char is declared using single quotes.

```
char c = ‘f’;
```

bool is used for boolean truth values (either true or false). As a note the word true or false can be capitalized, but it is not required.

```
bool x = True;
bool y = true;
bool z = false;
```

The string type is used to represent a concatenation of characters. Strings are made from ASCII characters and are declared using double quotes.

```
string greet = "Hello world!";
string code = "abc 123";
```

4.2 Built in Shapes

bugsy is an animation language and has pre-defined shape objects. These include typical shapes like squares and circles, but also user defined polygons as well. Below is a table of all built in shapes. The most basic unit is the point, denoted as `pt`, which consists of an x and y coordinate. The default placement for a shape is `pt(0,0)`, which corresponds to the bottom-left corner of the view. Points are especially useful when creating shape objects. See examples below for how each shape is created.

Type	**Description**	**Parameters**	**Example**
`pt`	Object that represents a point. Takes two num parameters as input that specify the origin of the point.	`pt(x,y)`	`pt x = pt(4,3);`
`shape`	Default object parent of all builtin shapes that defines thickness which extends to all other objects. Takes one num parameter specifying thickness.	`shape(thickness)`	`shape(3);`
`square`	Defined from an origin point and given a num size.	`square(pt(x,y), size)`	`square(pt(0,0), 3);`
`rect`	Defined from an origin point and takes an input width num and height num.	`rect(center, width, height);`	`rect(pt(0,0), 2, 3);`
`triangle`	3-pointed polygon defined from an origin point, with num height and width inputs.	`triangle(center, width, height);`	`triangle(pt(4,5), 5, 8);`
`circle`	Infinite-sided polygon defined from an origin point, with a num radius input.	`circle(center, radius);`	`circle(pt(2,3), 5);`

`ellipse`	Infinite-sided polygon defined from an origin point, and a num a and b which are the which are the major and minor radii inputs, respectively.	`ellipse(center,`
		`a, b);`
			`ellipse(pt(0,0),`
			`4, 5);`

`line`	A straight line defined by two points, start and end.	`line(start, end);`	`line(pt(x,y),`
			`pt(x,y));`

`canvas`	Defines a 2D plane whose boundaries constrain the locations of shapes. Has width and height parameters of type num.	`canvas(width,`
		`height);`
			`canvas(80, 100);`

`polygon`	Defines a figure with num number lines, of potentially varying lengths, according to the array points.	`polygon(number,`
		`points)`
			`polygon(5,`
			`[pt(1,1),`
			`pt(2,2,),`
			`pt(3,3),`
			`pt(5,4),`
			`pt(4,5)]);`

`regagon`	Creates a regular x sided polygon of radius r at origin point pt, where x is a num input and pt is the origin point.	`regagon(pt, x,`
		`r);`
			`regagon(pt(4,3),`
			`5, 3);`

`spline`	A curve with several anchor points, centered at point center, and specified by an array of nums, points	`spline(number,`
		`points)`
			`spline(3,`
			`[1,7,9]);`

5 Variables

In bugsy variables are declared using a type and a name. The type comes first followed by a variable name and then an equals sign. The right hand side of the expression is returned as a value. If the variable is only being declared, then a semicolon follows the name instead of an equals sign.
The grammar is as follows:

type ID equals expression
type ID

```
num a;
a = 10;
```

```
num b = 20;
num a = b = 4 * 5;
square x = square(pt(0,4), 4);
string hello = "hello sirs.";
```

5.1 Variable Naming

Variables names follow the following regex `[a-zA-z_][a-zA-Z0-9_]*` and as stated in the Reserved Words section, cannot be a reserved word.

6 Arrays

An array is a collection of variables of the same type that can be accessed under one variable name and a numerical index.

6.1 Declaring Arrays

Arrays in bugsy are declared by describing the type followed by square brackets containing the size of the array (`[10]`) followed by a variable name.
The grammar goes:
type LeftBracket number RightBracket ID
type LeftBracket number RightBracket ID equals LeftBracket li_contents Right Bracket
With `li_contents` being the contents of the array (name coming from list contents). These contents can be literals, booleans, shape objects, or other expressions.

```
num[10] myArray;
```

6.2 Initializing Arrays

Arrays in bugsy must be declared but can be later initialized; alternatively, arrays can be initialized at the time of declaration. Once declared, the value at a certain index in the array can be modified by using the name of the array, followed by square brackets containing the index that you want to reference or change.

```
num[3] myFirstArray = [1,2,3]; //declared and initialized
string[4] mySecondArray = ["bugsy", "is", "a", "guinea pig"]; //declared
    and initialized, size omitted
num[2] myThirdArray; //declared without initialization (defaults all
    values to 0)

//here we initialize the values inside the array
myThirdArr[0] = 7; //this notation allows you to directly assign numbers
```

```
myThirdArr[1] = 8; //to the index between the square brackets
```

6.3 Arrays in Arrays

bugsy supports declaring arrays inside of arrays. Declaration is very similar to that of normal arrays, but includes an additional pair of square brackets. An array type which is a number followed by square brackets (eg. `num[]`), is followed by an additional set of square brackets with the size of the new array.

```
num[][] myNestedArr = [myFirstArray, MySecondArray, myThirdArray];
print(myNestedArr);
```

Compiling and running the above would output the following:

```
[[1, 2, 3], [4, 5, 6], [7, 8]]
```

7 Classes

bugsy is object oriented in the sense that it supports custom object declaration. This is done through custom classes, or user-defined types. The class declaration informs the compiler what variables belong to the newly declared object and what functions can be used on objects of this type.

7.1 Class Declarations

A class is declared by using the keyword `class`, as defined in the reserved word section, followed by the name of the class in lowercase followed by a body surrounded by curly braces that defines the behavior of that object.

Classes fall under the general declaration grammar, being one of *vdecls* (variable declarations), *fdecls* (function declarations), and *cdecls* (class declarations) since programs can either be variables and functions, or a class followed by other declarations.

Under *cdecls*, a record is created that categorizes *cdvars* (the class variables), *cdconst* (the constructor), and *cdfuncs* (class functions).

Essentially:
decls →
    vdecls
    | fdecls
    | cdecls

cdecls →
    cdvars

| cdconst
| cdfuncs

7.1.1 Constructors

Classes require a single constructor in order to initialize the instance variables. A constructor is a method called constructor and takes in a set of parameters. Constructors do not have a return type unlike functions, which are described in **section 10**.

```
class guineapig{
    num myNum;
    string myStr;

    constructor(num v, string st){
        myNum = v;
        myStr = st;
    }

    // a function that can be called on a MyObject object.
    printMyObject(){
        print("I am an object that says " + myStr + " and hold the
            value: " + myNum);
    }

}
```

7.2 Dot Notation

bugsy has no understanding of scope modifiers like private or public variables so all variables of classes are accessible using dot notation. This is done simply by referencing the object followed by the property that you want to access or mutate.

```
guineapig graynold = guineapig(20, "Hello!");
print(graynold.myNum);
graynold.printMyObject();
```

Compiling and running the above would output the following:

```
20
I am an object that says Hello! and hold the value: 20
```

8 Operators and Arithmetic

All arithmetic done in bugsy will be expressions composed of operators and values. bugsy supports both unary and binary operators.

The grammar for expressions breaks down into literals, booleans, unary operations, and binary operations. The unary and binary operations either include one or two more *expr's* in the pattern matching so that it can expand. For instance, addition is done with *expr* PLUS *expr*.

8.1 Unary Operators

bugsy utilizes three four of unary operators, increment (`++`), decrement (`--`), the unary minus sign (`-`) in front of an expression to flip the sign of a numerical value (or a variable containing a numerical value), and an exclamation mark (`!`) in front of an expression to flip the resulting boolean. Using the unary minus on a boolean expression is not valid.

Grammar:
OPERATOR expr
expr OPERATOR

Operator	**Character(s)**	**Example**

Increment	`++`	`num i = 0;`
		`i++;`
		`++i;`

Decrement	`--`	`num j = 0;`
		`j--;`
		`--j;`

Unary minus (negation operator)	`-`	`num x = -5;`
		`num y = -x;`

Not	`!`	`bool x = true;`
		`bool y = !x;`

8.2 Binary Operators

The binary operators in bugsy are outlined as follows in the table below. Addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), plus equals (`+=`), minus equals (`-=`), multiplication equals (`*=`), division equals (`/=`), assignment (`=`), modulus (`%`), and exponentiation (`^`) all result in numerical values and can only take expressions that produce numerical values. The greater than comparison (`>`), less than comparison (`<`), greater than or equal to comparison (`>=`), less than or equal to comparison (`<=`), equality check (`=?`), and not equals to (`!=`) result in booleans but only accept numeric values. And (`and`), and or (`or`) also produce booleans only accept boolean expression inputs. The exponent operator will not flip bits.

Grammar: *expr OPERATOR expr*

Note that in the following examples, x and y are initialized as

```
num x = 0;
num y = 0;
```

Operator	**Character(s)**	**Example**
Addition	`+`	`x + y;`
Subtraction	`-`	`x - y;`
Multiplication	`*`	`x * y;`
Division	`!`	`x / y;`
Plus equals	`+=`	`x += y;`
Minus equals	`-=`	`x -= y;`
Multiplication equals	`*=`	`x *= y;`
Division equals	`/=`	`x /= y;`
Assignment	`=`	`x = y;`
Modulus	`%`	`x % y;`
Exponentiation	`^`	`x ^ y;`

Equality check	`==`	`x =? y;`
Not equals (neq)	`!=`	`x != y;`
Greater than	`>`	`x > y;`
Less than	`<`	`x < y;`
Greater than or equal to	`>=`	`x >= y;`
Less than or equal to	`<=`	`x <= y;`

8.3 Logical Operators

As previously mentioned, not (`!`), and (`‘and’`), or (`‘or’`) are bugsy’s logical operators. Booleans can be expressed as either (`1`), any positive number, (`‘true’`), or (`‘True’`) for `true`. and (`0`), any negative number, (`‘false’`), or (`‘False’`) for `false`.
The grammar is the same as the previous operators, depending on if it’s unary or binary.

9 Control Flow

bugsy supports a majority of the most common aspects of control flow. As an animation oriented language, looping plays an important role in displaying the intended content and will be used within several of the standard library functions.

9.1 Loops

9.1.1 While Loops

while loops in bugsy work identical to a while loop in other imperative languages such as Python, C, or Java. A while loop is uniquely defined by its two major components which are the conditional statement which bugsy requires be enclosed in parentheses following the `while` keyword. After the close parentheses the code intended to be performed each iteration of the loop will be enclosed in curly braces.
A while loop will continue to execute the code encapsulated within the opening and closing curly braces until the condition defined within the parentheses

evaluates to false (Refer to Section 4.1 **Primitive Data Types** and Section 8.3 **Logical Operators**).
Grammar:

while LeftParen expr RightParen LeftBracket stmt_list RightBracket

```
num val = 0;
while(true) {
  // execute code
  val = val + 1;
}
```

9.1.2 For Loops

for loops, similar to while loops will execute a block of code enclosed in curly braces until the end condition is met. The for loop is defined by three statements enclosed in parentheses following the keyword `for`.
The first statement declares a num variable and initializes to some starting value. The second statement, separated by a semicolon, defines the comparison condition involving the previously declared num and another num. The third statement provides the increment or decrement step sizes for the num declared in the first statement.
Grammar:

for LeftParen Num ID SEMI ID CMP_OPERATOR SEMI ID INC_OPERATOR RightParen LeftBracket stmt_list RightBracket

```
num val = 0;
for(num i = 0; i < 10; i++) {
  val = val + i;
}
```

9.1.3 Break Statement

The keyword `break` on a line of its own will cause the loop to abruptly exit meaning further iterations of the loop will not be executed.
Grammar:

break SEMI

```
num val = 0
for(num i = 0; i < 10; i++) {
  val = val + i;
  break;
}
```

9.1.4 Continue Statement

When a `continue` statement is placed on a line of its own within a loop it will force the next iteration of the loop to execute immediately, disregarding code that would have occurred sequentially after the continue statement within the block of executing code.
Grammar:
continue SEMI

```
\begin{lstlisting}
num val = 0
for(num i = 0; i < 10; i++) {
  continue;
  val = val + i;
}
```

9.2 Conditional Statements

The `if` statement will execute a block of code enclosed in curly braces if the provided condition is satisfied. The condition follows the keyword if and is enclosed in parentheses. The `elif` statement cannot stand on its own and must follow a previous declared if statement. The condition of the elif statement is only checked if the condition of the previous if statement does not evaluate to true. elif statements can be chained with the earlier declared conditions having the highest priority. The `else` statement is used as a catch-all which will execute if none of the previously defined if or elif conditions are met. The else statement consists of the keyword else followed by curly braces as it does not have its own condition.
It is important to note that `if` statements cannot be chained together like `elif` statements. It is possible for both `if` statements to evaluate to `true` and execute their respective code blocks.
Grammar: if LeftParen expr RightParen LeftBracket stmt_list RightBracket
if LeftParen expr RightParen LeftBracket stmt_list RightBracket else LeftBracket stmt_list RightBracket
if LeftParen expr RightParen LeftBracket stmt_list RightBracket elif LeftParen expr RightParen LeftBracket stmt_list RightBracket
if LeftParen expr RightParen LeftBracket stmt_list RightBracket elif LeftParen expr RightParen LeftBracket stmt_list RightBracket else LeftBracket stmt_list RightBracket

9.2.1 If . . . elif . . . else

```
num val = 5;
num res = 0;
if(val =? 5) {
  res = 1;
} elif(val > 5) {
  res = 2;
} else {
     res = 3;
}
```

9.3 Exceptions

9.3.1 Try ... Catch Statement

A try statement is used in order to prevent the program from crashing unexpectedly if a specific block of code returns some sort of error. It is defined as a block of code surrounded by curly braces following the keyword `try`.
The try statement should be followed by a catch statement which is passed an argument which specifies the type of error the code in the try block is expected to return. The exception follows the keyword `catch` and is surrounded by parentheses. Code that will execute if an exception is caught can be placed between curly braces following the exception.
Grammar:
try LeftBracket stmt_list RightBracket catch LeftParen Exception ID RightParen LeftBracket stmt_list RightBracket

```
try {
  //run code
}
catch(Exception e) {
  //error occurred
}
```

9.3.2 Raise

The `raise` keyword is used at the start of a line followed by an Exception declaration. If the program encounters a raise call it will stop executing and return with the error message described in the Exception.
Grammar:
raise Exception LeftParen string RightParen SEMI

```
raise Exception("error occurred");
```

9.4 Pass

The `pass` keyword is used as a temporary placeholder for a function that has yet to be filled in with executable code. It will not produce any noticeable outcome and will be simply read and skipped over.
Grammar:
pass SEMI

```
num random() {
  pass;
}
```

10 Functions

Functions in bugsy are designed to be simple, clean, and reusable. The syntax for defining a function is as follows.

```
return_type function_name(input1_type input1, input2_type input2...)
    {...}
```

Where the types of the return value and the parameter can be any primitive or custom type. A function that does not return anything can be specified with the return type void. The following is an example of a function which finds the greatest common denominator of two numbers.

```
num gcd(num a, num b) {
   while (a != b) {
      if (a > b) {
         a = a - b;
} else {
   b = b - a;
}
   }
   return a;
}
```

Also note that parameters are passed by value and thus the return value of a function must be set equal to a variable.
Grammar:
V type ID LeftParen formal_list RightParen LeftBracket var_list stmt_list RightBracket

11 Memory Management

In bugsy, memory is leaked by default.

12 Standard Library

The standard library in bugsy consists of a variety of functions which are primarily used for animation and graphical manipulation. Below are the functions included in the standard library.

12.1 animateTo

12.1.1 Definition

```
 animateTo(point: a_point, speed: speed_value, scaleVal: scale_value)
```

12.1.2 Description

a_point is a Point object, speed_value is a num, and scaleVal is a num. The default values for these variables are pt(0,0), speed: 1, and scaleVal: 1. pt is the destination of the object, speed controls the speed at which the object reaches pt (where a value of 1 corresponds to 1 pixel/second), and scaleVal controls the scale of the object throughout the duration of the animation.

12.1.3 Example Usage

```
square x = square(pt(0,0), 2);
// moves to 5,6 with speed of 24 and scales down while doing that
x.animateTo(pt(5,6), speed: 24, scale: 0.6);
```

12.2 scale

12.2.1 Definition

```
scale(factor: a_factor)
```

12.2.2 Description

a_factor is a num. Calling this function on a graphical object will scale the object by a_factor.

12.2.3 Example Usage

```
square x = square(pt(0,0), 2);
// scales the square down by 50%
x.scale(0.5);
```

12.3 rotate

12.3.1 Definition

```
rotate(degree: a_degree, rpm: a_rpm, repeating: rep)
```

12.3.2 Description

where a_degree is a num. Calling this function on a graphical object will rotate the object by the degree specified, with a speed corresponding to a_rpm rotations per minute. If rep is equal to true, the shape will continue rotating forever, starting from 0 degrees each time. Note that if, for instance, 30 degrees is specified with an rpm of 1, the shape will thus rotate 30 degrees in 12 seconds, as this would correspond to an rpm of 1. Also note that if a degree value larger than 360 is specified, the object will be rotated by a_degree % 360. a_degree must be greater than or equal to 0.

12.3.3 Example Usage

```
square x = square(pt(0,0), 72);
// rotates the square by 72 degrees
x.rotate(degree: 72, rpm: 1, repeating: false);
square x = square(pt(0,0), 2);
// continually rotates the square
x.rotate(degree: 360, rpm: 2, repeating: true);
```

12.4 Color

12.4.1 Definition

```
color(r: red, g: green, b: blue)
```

12.4.2 Description

where red, green, and blue correspond to RGB values specifying a color.

12.4.3 Example Usage

```
square x = square(pt(0,0), 72);
// color the square green
x.color(0, 255, 0);
```

13 Sample Code

The following samples of code provide some illustrative examples of the purpose and coding style of bugsy.

13.1 Rotating Circle And Square

```
circle circ = circle(pt(3,3), 5); //creates a circle at the point (3,3)
    of radius 5
square sq = square(pt(3,3), 5); //creates a square at the point (3,3) of
    radius 3
sq.rotate(360, 10, true); //starts the square rotating infinitely
sq.scale(0.4); //scales the square down to 40% of original height and
    width
```

13.2 bugsy Printer

```
void oddBeanPrinter(num x){
    num counter = 0;
    while (counter <= num){
        if counter % 2 == 1 {
            print("bean");
        }
    }

}
```

13.3 FizzBug

```
int n = 100;

// loop for 100 times
for (int i=1; i<=n; i++){
    //number divisible by 15
    if (i%15=?0) {
        print("FizzBug");
    }

    //number divisible by 5
    else if (i%5=?0){
        print("Bug");
    }

    //number divisible by 3
    else if (i%3=?0){
        print("Fizz");
    }

    // print the numbers
    else{
        print(i);
    }
}
```