FOGL (Figure Oriented Graphics Language)

Julian Rosenblum, Evgenia Nitishinskaya, Richard Zou

0 - Introduction

FOGL is a language for the sole purpose of creating static and interactive graphics. As the name suggests, programs are primarily made up of specialized data structures called figures. Figures are similar to classes except they are more tailored towards graphics. FOGL is dynamically typed and imperative.

0.1 - Program

A program consists of a single file containing figure definitions followed by an implicit main function. Any code that is not a figure definition is considered to be part of the main function. All figure definitions must come before the implicit main function.

0.2 - Execution

All figure instances declared in the main function are implicitly drawn (see 1.9). The program then loops. During a cycle, each figure's _update function is called if and only if its _animating property is set to true (see 1.6, 1.8). If any figure is mutated during a cycle, all figures are erased and redrawn.

1 - Figures

A figure is a class-like definition of an object that can be drawn from its components given parameters. Like classes, figures can have multiple instances. Unlike classes, all properties are public and there is no inheritance. Figures can contain any number of any of the following within it:

1.1 - Parameters

```
        param rad,
        loc = <<0,0>>,
        col = #ff0
```

Parameters are the given arguments that are automatically assigned to properties of the same identifier. If a parameter is not given a default value, then it must be declared for each instance of a figure; otherwise, it is optional (see 1.9). Parameters eliminate most of the need for constructors. For example, the code:

```
        figure PacMan { param loc }
```

is essentially equivalent to the following in Java:

```
public class PacMan {
        private Location loc;
        public PacMan (Location myLoc) {
                loc = myLoc;
        }
}
```

Constructors, in the rare event that they are necessary, can be simulated using the _update function (see 1.6).

1.2 - Components

```
        comp body = Arc(loc:loc radius:rad col:col),
        mouth = Arc(loc:loc radius:rad start:PI/4 end:-PI/4 col:#000 fill:true),
        eye = Arc(loc:loc+<<rad/2,-rad/2>> radius:3 col=#000)
```

Components are properties that are drawn whenever the figure is drawn. A component must be a figure (built-in or user-defined) or a list of figures. Components are bound to figure references and cannot be assigned directly.

1.3 - Properties

```
	var mouthOpenness = 1,
	isAlive = true
```

A property is a variable that is defined in a figure whose value is unique for each instance of the figure. Parameters and components are both types of properties, but the term "property" generally refers to a property that is neither of the two. Properties are public. All properties (including parameters and components) can be assigned directly.

1.4 - Methods

```
	var doSomething = func (arg1, arg2) {
		; statements
	}
```

Methods are routines that are defined for a figure but manipulate properties for whichever instance called it. All methods are lambdas assigned to properties. Arguments in method and function calls are referenced by numerical order, not by name.

1.5 - Recursive Figures

Figures can be recursive, meaning they have components that are instances of themselves. Recursive figures must have at least one component that isn't recursive. The base case is considered to be when all non-recursive the drawn components of the figure combined are smaller than a single pixel. For example, consider the Sierpinski triangle (assume the figure `Equilateral` has already been defined with loc referring to the top vertex):

```
	figure Sierpinski {
		param loc = <<0,0>>, sideLength
		comp triangle = Equilateral(loc:loc length:sideLength fill:false),
		s1 = Sierpinski(loc:loc sideLength:sideLength/2),
		s2 = Sierpinski(loc:loc+<<-sideLength/4,sideLength/4*RT_3>>
sideLength:sideLength/2),
		s3 = Sierpinski(loc:loc+<<sideLength/4,sideLength/4*RT_3>>
sideLength:sideLength/2)
	}
	var s = Sierpinski(sideLength:5)
```

1.6 - The _update Method

Figures may have a method called _update which is used by the compiler to perform animations. With each cycle, every figure's _update method is called (if such a method exists) with one argument: the current number of milliseconds since the program began. This function can be used to perform animations by modifying the figure based on the milliseconds parameter. The function does not return anything. The following example smoothly opens and closes PacMan's mouth where cycleLength is the number of milliseconds it takes to go from open to closed:

```
	var _update = func (ms) {
		var state = (cycleLength * 2) % ms
		if state > cycleLength { ; If the mouth is closing
			state = cycleLength * 2 - state
		}
		mouth.start = (state / cycleLength) * PI/4
		mouth.end = -mouth.start
		if ms == 3000 { ; additionally, at 3 seconds
			body.color = #00f ; change to blue
		}
	}
```

1.7 - The _visible Property

Every figure has an implicit _visible property. It is set to true by default. If explicitly set to false, the figure's components are not drawn.

1.8 - The _animating Property

The _animating property is implicitly set to true if an _update function is defined and false otherwise. The _update function is only called if the _animating property is true, and if the _animating property is true and no _update function is defined, the program throws an error.

1.9 - Initializing Figures

Instances of figures are initialized using the syntax: `PacMan(rad:4 loc:<<6,6>>)`. Within the parentheses is a list of whitespace/newline separated parameter bindings, with the name of the parameter (as defined by the figure) on the left of a colon and an expression on the right. Parameters are bound to the *value* of the expression. All parameters of the figure that do not have default values *must* be bound for each instance of the figure (otherwise an error is thrown). Parameters that do have default values are not required but may be bound for any figure instance. "Empty" figure initializations, if all parameters have default values, are initialized using the syntax: `MyFigure(:)`. The colon allows the program to distinguish between figure initializations and function calls at compile time.

2 - Data Types

2.1 - Int

An int is a 32-bit signed integer. An int literal is formally defined as 1 or more digits.

2.2 - Float

A float is a 64-bit signed floating-point number. A float literal is formally defined as `((digit+ '.') | digit* '.' digit+) exp? | digit+ exp` where exp is defined as `['e' 'E'] ['+' '-']? digit+`.

2.3 - Boolean

A boolean value defined using the keywords `true` and `false`.

2.4 - String

A string is a string of text. String literals are placed between either single or double quotes.

2.5 - Ordered Pair

An ordered pair is a pair of x and y floats which can be expressed as a literal `<<x,y>>`. Note: the origin is the top left corner of the canvas and the y-value increases in a downward direction. Arithmetic operators may be used on two ordered pairs or one ordered pair and one integer or float. Assuming *op* is a binary operator, these operations are defined as follows: `<<x,y>>` *op* `<<z,w>>` evaluates to `<<x` *op* `z,y` *op* `w>>`. `<<x,y>>` *op* `n` is equal to `<<x` *op* `n,y` *op* `n>>`. `n` *op* `<<x,y>>` evaluates to `<<n` *op* `x,n` *op* `y>>`. The individual x and y values can be retrieved using the built-in `x()` and `y()` functions (see 5.2.3).

2.6 - List

A list is a Linked List which can be expressed as comma-separated items inside brackets. Lists can only contains elements of the same type. A list cannot contain two different types of figures.

2.7 - Color

A color is a hex colorvalue which can be expressed as the literal `#rrggbb`, `#rgb`, `#rrggbb@a`, or `#rgb@a` where `r`, `g`, and `b` are hex digits and `a` is a float literal representing alpha (opacity) on a decimal scale from 0 to 1. Note: if single digit hex values are used, the digits are duplicated to produce double-digit values. For example, `#f03@.5` evaluates as `#ff0033@.5`. If alpha is not declared, it is assumed to be 1.0.

2.8 - Figure

See Section 1.

2.9 - Type Casting

Integers are automatically cast to floats when necessary. If an arithmetic operator is used on two integers, the result is an integer. Otherwise, the result is a float. The only exceptions are the `/` and `^` operators, in which case the result is always a float, and the `//` operator, in which case the result is always an integer. In instances when explicit type casting is necessary, there are built-in functions that take convert an argument to the indicated type if such a conversion is possible and throw an error otherwise (see 5.2.1).

3 - Operators

3.1 - Arithmetic Operators

- + (addition, string concatenation)
- - (subtraction, unary negation)
- * (multiplication)
- / (floating-point division, both operands cast to float)
- // (integer division, both operands cast to int)
- % (modulo)
- ^ (exponentiation)

3.2 - Assignment Operators

- = (assigns RHS to LHS or passes RHS to argument with the identifier LHS)
- An arithmetic operator followed by = performs the operation with LHS and RHS and then assigns the value to LHS.

3.2 - Comparison Operators

- == (equal to)
- != (not equal to)
- > (greater than)
- < (less than)
- >= (greater than or equal to)
- <= (less than or equal to)

3.3 - Logical Operators

- ! (unary, logical not, casts operand to boolean)
- & (logical and, casts operands to boolean)

- | (logical or, casts operands to boolean)
- ? (unary, casts operand to boolean)

3.4 - Misc. Operators

- . (dot operator) - Selects property or method of a figure.
- : (param-bind operator) - Binds a parameter (lhs) to value (rhs)
- [n] (list indexing operator) - Returns the nth element of the list (0-indexed)

4 - Statements, Comments, etc.

Statements are terminated with newlines.

4.1 - Reserved words

```
param comp var func figure if elseif else do while true false return
```

4.2 - Comments

Single-line comments are declared using a `;` (semicolon) and terminate at the end of a line.

4.3 - If-elseif-else Statements

```
if x < 3 {
	; statements
}
elseif x > 7 {
	; statements
}
else {
	; statements
}
```

4.4 - While Statements

```
while i < 5 {
	; statements
	i += 1
}
```

4.5 - Do-while Statements

```
do {
	; statements
	i += 1
} while i < 5
```

4.6 - Global Variables

```
var foo = 5, bar = "hello"
```

Global variables can be accessed within any scope.

4.7 - Global Functions

```
var doSomething = func (arg1, arg2) {
	; statements
}
```

All functions are lambdas assigned to variables. Arguments in method and function calls are referenced by numerical order, not by name.

5 - Standard Library

5.1 - Built-in figures

- `figure Rect { param loc=<<0,0>>, width, height, col=#000, fill=true, lineWidth=1 }`
- `figure Arc { param loc=<<0,0>>, radius, start=0, end=2*PI, col=#000, fill=true, lineWidth=1, clockwise=true }`
- `figure Poly { param loc=<<0,0>>, points, col=#000, fill=true, lineWidth=1 }`
- `figure Text { param loc=<<0,0>>, text, col=#000, size=12, font="Arial", align="left" }`
  Note: lineWidth only applies if fill=false

5.2 - Built-in functions

- **5.2.1 - General**
  - `print(str)` - Prints a string to the console. Argument is cast to string.
  - `int(foo)`, `float(foo)`, `string(foo)`, `bool(foo)` - Converts argument to the indicated type if possible, throws an error otherwise. Note: `?foo` is semantically equivalent to `bool(foo)`.
- **5.2.2 - Color Operations**
  - `rgb(r, g, b)` - Returns a color literal given rgba values.
  - `rgba(r, g, b, a)` - Returns a color literal given rgba values.
- **5.2.3 - Pair Operations**
  - `x(pair)`, `y(pair)` - Returns the x or y value of an ordered pair.
  - `dist(pair1, pair2)` - Returns the Euclidian distance between two pairs.
- **5.2.4 - Vector Operations**
  - `mag(pair)` - Returns the magnitude of the vector from the origin to pair.
  - `ang(pair)` - Returns the angle (in radians) of the vector from the origin to pair. An error is thrown if `pair` is `<<0,0>>`.
  - `dot(pair1, pair2)` - Returns the dot product between two vectors v1 and v2, where v1 is a vector formed from the origin to `pair1`, and v2 is a vector formed from the origin to `pair2`.
  - `cross(pair1, pair2)` - Returns the magnitude of the cross product between two vectors v1 and v2, where v1 is a vector formed from the origin to `pair1`, and v2 is a vector from the origin to `pair2`.
  - `bet(pair1, pair2)` - Returns the angle (in radians) between two vectors v1 and v2, where v1 is a vector formed from the origin to `pair1`, and v2 is a vector from the origin to `pair2`. An error is thrown either ordered pair is `<<0,0>>`.
- **5.2.5 - Mathematical Operations**
  - `toDeg(theta)` - Converts radians to degrees.
  - `toRad(theta)` - Converts degrees to radians.
  - `sin(x)`, `cos(x)`, `tan(x) -` where `x` is in radians
  - `asin(x)`, `acos(x)`, `atan(x) -` returns the angle as a radian value
  - `log(b, e)` - where `e`, `b` > 0, `b` ≠ 1
  - `ln(x)` - where `x` > 0
  - `ceil(x)`, `floor(x)`
  - `sqrt(x)`

    ○ `round(x)`
    ○ `min(x, y)`, `max(x, y)`
    ○ `abs(x)`
    ○ `random()`

5.3 - Built-in constants

- `PI`
- `E`
- `RT_2` (Square root of 2)
- `RT_3` (Square root of 3)

Appendix A - Syntax and Grammar

A.1 - A Note on Whitespace

Since newlines are semantically significant in FOGL, they are handled by a combination of the scanner and parser. A newline is formally defined as `([' ' '\r' '\t']* '\n' | [' ' '\r' '\t']* ';' [^ '\n'] * '\n')+`, meaning a newline character or single-line comment followed by any combination of empty lines or lines consisting of only single-line comments. Grouping punctuation (parentheses, brackets, braces, double angle brackets) are considered to "absorb" newlines at the beginning and end of the group. Opening punctuation is defined as the punctuation followed by 0 or more newlines. Closing punctuation is defined as 0 or more newlines followed by the punctuation. Additionally, the keywords `else` and `elseif` are defined as 0 or more newlines followed by the keyword. For additional instances of optional whitespace, the `ws` grammar rule is used, which is defined as 0 or more newlines.

A.2 - Grammar

```
program:
    figure_list main
ws:
    /* nothing */
  | NL
figure_list:
    ws
  | figure_list figure ws
figure:
    FIGURE ID ws { figure_body figure_decl ws }
  | FIGURE ID ws { ws }
figure_body:
    /* nothing */
  | figure_body figure_decl NL
figure_decl:
    PARAM decl_list
  | COMP decl_list
  | VAR decl_list
decl_list:
    decl
  | decl_list , ws decl
decl:
    ID
  | ID = expr
main:
    /* nothing */
  | main stmt NL
  | main stmt EOF
block:
    { }
  | { stmt_list stmt ws }
stmt_list:
    /* nothing */
  | stmt_list stmt NL
arg_list:
    ID
  | arg_list , ws ID
arg_list_opt:
    /* nothing */
  | arg_list
expr_list:
    expr
  | expr_list , ws expr
expr_list_opt:
    /* nothing */
  | expr_list
param:
    ID : expr
param_list:
    param
  | param_list ws param
param_list_opt:
    :
  | param_list
elseif:
    /* nothing */
  | ELSE ws block
  | ELSEIF expr ws block elseif
stmt:
    expr
  | VAR decl_list
  | expr = expr
  | expr += expr
  | expr -= expr
  | expr *= expr
  | expr /= expr
  | expr //= expr
  | expr ^= expr
  | expr %= expr
  | RETURN expr
```

```
    | WHILE expr ws block
    | DO ws block ws WHILE expr
    | IF expr ws block elseif
expr:
      ( expr )
    | INT
    | STRING
    | FLOAT
    | TRUE
    | FALSE
    | COLOR
    | << expr , ws expr >>
    | ID
    | FUNC ( arg_list_opt ) block
    | [ expr_list_opt ]
    | expr [ expr ]
    | expr . ID
    | expr + expr
    | - expr
    | expr - expr
    | expr * expr
    | expr / expr
    | expr // expr
    | expr ^ expr
    | expr % expr
    | expr == expr
    | expr != expr
    | expr > expr
    | expr < expr
    | expr >= expr
    | expr <= expr
    | expr & expr
    | expr | expr /* The second "|" refers to the character "|" */
    | ! expr
    | ? expr
    | ID ( expr_list_opt )
    | ID ( param_list_opt )
```

Appendix B - Sample Code

```
figure PacMan { ; define a new figure
        ; arguments which are automatically stored under the same name
        param rad, ; arguments without default values are required
        loc = <0,0>, ; arguments with default values are optional
        col = #ff0

        ; components to be drawn
        comp body = Arc(loc:loc radius:rad col:col),
        mouth = Arc(loc:loc radius:rad start:PI/4 end:-PI/4 col:#000 fill:true),
        eye = Arc(loc:loc+<<rad/2,-rad/2>> radius:3 col:#000)

        ; misc. properties and methods
        var mouthOpenness = 1,
        isAlive = true,
        cycleLength = 1000, ; Amount of time to go from open to closed
        _update = func (ms) {
                var state = (cycleLength * 2) % ms
                if state > cycleLength { ; If the mouth is closing
                        state = cycleLength * 2 - state
                }
                mouth.start = (state / cycleLength) * PI/4
                mouth.end = -mouth.start
                if ms == 3000 { ; additionally, at 3 seconds
                        body.color = #00f ; change to blue
                }
        }
}

figure Sierpinski {
        param loc = <<0,0>>, sideLength
        comp triangle = Equilateral(loc:loc length:sideLength fill:false),
        s1 = Sierpinski(loc:loc sideLength:sideLength/2),
        s2 = Sierpinski(loc:loc+<<-sideLength/4,sideLength/4*RT_3>>
sideLength:sideLength/2),
        s3 = Sierpinski(loc:loc+<<sideLength/4,sideLength/4*RT_3>>
sideLength:sideLength/2)
}

var p1 = PacMan(rad:10),
p2 = PacMan(rad:4 loc:<<6,6>>),
s = Sierpinski(loc:<<100,100>> sideLength:5)
```