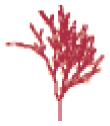
Organic Form Language

Eric Larson



Overview

Self-similar organic forms of unlimited complexity are used in computer graphics for generating organic images such as those of plants, and also generating patterns such as fractals. For this purpose, Organic Form Language, or "OrgForm", was designed. An interpretor will translate OrgForm code, into line drawings.

OrgForm was inspired by L-systems, which is a mathematical way of defining organic forms, and by functional languages such as Scheme, which simplifies the building of complicated programs. L-systems were pressed into use as a programming language, though it was originally intended as mathematic formula notation. Functional languages are an intuitive way of representing recursion, which is the predominate characteristic of L-systems. By bringing these concepts together, OrgForm makes building more sophisticated organic forms easier.

Concept

Every OrgForm program starts with a call to a rule. The program is executed in iterations. During each iteration, each call to a rule is replaced by that rule's definition.

For example, here is pseudo-code of a program which builds a plant. First it makes a branch, but in subsequent iterations, that branch grows into more branches.

```
rule: branch = branch, turn right, branch
```

0 iterations:
branch

1 iteration:
branch, turn right, branch

2 iteration:
branch, turn right, branch, turn right, branch, turn right, branch

The programmer selects the number of iterations that the program will run. When the program has executed the final iteration, the result is translated into graphics. In the above example, branch may mean "draw a line segment".

In OrgForm, the above information would be captured in the following program.

```
Start:branch
Iterations:2

Rule:branch
         Right
                branch;
                branch;
        ;
        Final: Draw
```

Notice that the function calls are each put on a new line. This is just to make the code more clear. In OrgForm, single carriage returns are ignored. The branch function could also have been defined as follows.

```
Rule:branch Right branch; branch;; Final:Draw
```

Design Concerns

Functions

To make OrgForm a powerful language, it is necessary to be able to represent as many organic shapes as possible. A possible bench-mark is to create a language which can represent all the organic shapes representable in L-systems. One way to reach this bench-mark is to design OrgForm with analogies to the two features which give L-systems its expressiveness, iteration and state frames.

State frames are groups of states which may be pushed and popped. There are two separate types, one for states which involve "time", and one for all other states.

Iteration in L-systems provides a way to increment the growth of all portions of the form simultaneously. Because the parts of organisms grow simultaneously, iteration provides a means for modeling the growth of plants.

Both of these features of L-systems provide users with abstraction. Because of state frames, users do not need to concern themselves with modifying the states for all the components of forms. Iteration means that users do not need to describe every sequence of the growth of forms. Both iteration and state frames are natural analogies to functions. The problem is finding a way to map multiple function-like behaviors onto a functional language which only has one type of function.

The solution is to have two types of functions. One is a "rule", the other is a "function". Using keywords, the programmer declares a defined method to be of either type.

To imply the inheritance of state frames, the order of execution of the functions if from left to right, ignoring spaces, tabs, and single carriage returns. Each function call takes one argument, which is whatever function is to its immediate right. The function being called is executed, then it executes the function that it took as a parameter.

For example, the following line will call my_func_1 first, which will execute, possibly changing state values, and then call my_func_2.

```
my_func_1
        my_func_2
```

The only exception is if a “:” or a “;” is between the function calls. These symbols tell the assembler to pass the call on the right to the function preceding the function call to the left. In the following example my_func_1 will take and execute both my_func_2 and my_func_3, and my_func_2 will not execute my_func_3. Any state changes caused by my_func_2 will not effect my_func_3.

```
        my_func_1
                my_func_2;
                my_func_3
```

Notice that function calls that which are are

To define a function, use the key word "Function:".

```
Function:my_func_1
                my_func2;
                my_func1
```

To express the difference between a function which has a state frame, and a iteration stage, which may or may not have a frame, OrgForm provides the keyword "Rule:".

```
Rule:my_rule_1
                my_func2;
                my_func1
```

If the rule has a frame, just define a function which is called by the rule.

Simplicity

To make the language easy to learn, complete English words are used instead of symbols when sensible.   There are only eleven keywords and four special symbols.

The only thing that a programmer may define in OrgForm are branches and functions. These names must be all lower-case letters, numbers, and under-scores ("_").

Every statement in an OrgForm program will have an effect on the programs state.

The only drawing concepts are lines and the angles between them. There is no concept of a point or location. Because the language is only concerned with forms, points are not needed.

All keywords begin with upper-case letters.

Keywords

Drawing and moving commands

“Right”, “Left”, “Draw”, “Forward”

“Iterations”, Number of times the program will resolve rules.
“Start”, The first rule or function call the program will make.
“Final”, The value that the given rule will resolve after the final iteration.

“Color”, The color code.
“Thickness”, The thickness of the line.
“Length”, How far a “Draw” or a “Forward” command will move.

“Rule”, A new rule declaration.
“Function”, A new function declaration.

Special Characters

":", Set a value.
"+", Increase a value. Only used after “Color|Thickness|Length”.
"-", Decrease a value. Only used after “Color|Thickness|Length”.
“,”, Revert state back except time.
";", Revert the entire state back.
“#”, Comment.

Example OrgForm Code

```
#Program: Alfonse

#Builds a mult-color bristled plant which leans to the right.

#Setup initial environment
Iterations:5
Angle:22.5
Color:12
Start:base
```

```
Rule:base
        color:12 branch
                base,
                Left Left

                left2_bristles,
                branch
                        right2_bristles,
                        Right


#Make a bristle at a sharp left angle
Function:left2_bristles
        Left Left
                bristles

#Make a bristle at a sharp left right
Function:right2_bristles
        Right Right
                bristles

#Make upper part of plant
Rule:upper
        Color:1
                branch
                        upper,
                        Right Right
                                left2_bristles,
                                branch
                                        right2_bristles,
                                        base

Rule:bristles
        make_bristle,


#Seperate from Rule “bristles” so bristles will always revert
# the state back to what it was before it was called.
Function:make_bristle
        Color:4
                Right Forward Left Forward Left Forward Right transpose
Right Forward Left Forward Left Forward

Rule:branch
        Forward
```

```
                left_bristles;
                right_bristle,
                Forward

Function:left_bristles
        transpose Left Left bristles


Function:right_bristles
        transpose Right Right bristles


#Needs to be refined
Function:transpose
        Left Left Left
```

The graphical result will roughly be like this.