Organic Form Language
Final Report

Eric Larson
el2264
May 1, 2006
COMS W4115

-----Overview:
Self-similiar organic forms of unlimited complexity has applications in computer graphics for generating organic images such as those of plants, and and other branching patterns. For this purpose, Organic Form Language, or OFL, was designed. An interpretor will translate OFL code, into drawings.

OFL 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 was 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, OFL makes building more sophisticated organic forms easier.

Concept:
Every OFL 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.

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.

----Design Goals:

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

--Simplicity

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

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.

-- Language Tutorial

There are two key principles to understanding how to program in OFL,
indentation, and replacement.

The indentation of a statement determines which changes of position and direction from preceding statements will effect it.
If, for example, you have a statement preceded by three white spaces, it will inherit the changes of position and direction of any statement

above it which is indented by two or one spaces. In the following code example, only changes caused by executing statemen1 will effect
where statement3 begins executing.

```
 statement1
    statement2
   statement3
```

Replacement is the way each iteration of the program changes the plant.
Rules are defined in an OFL program, and during each generation, the
definition of a rule replaces any references. The complex patterns in images generated come from rules referring to themselves.
For instance, this rule will be referred to by the string "bushy", and will generate a progressively complicated image with each new iteration of the program. Definitions begin with a name and a colon.

```
bushy:
 Draw
   bushy
  Right
   bushy
```

"Draw" causes a line segment to be painted, "Right" causes all inheriting
statements to execute in a direction moved clockwise by a few degrees.

Every OFL program must have a "Start:" to show where to begin executing. Start should call some user defined rule.

All user defined rule must begin with a newline followed immediately by
a string which begins with a lower case letter and is followed
by a colon.

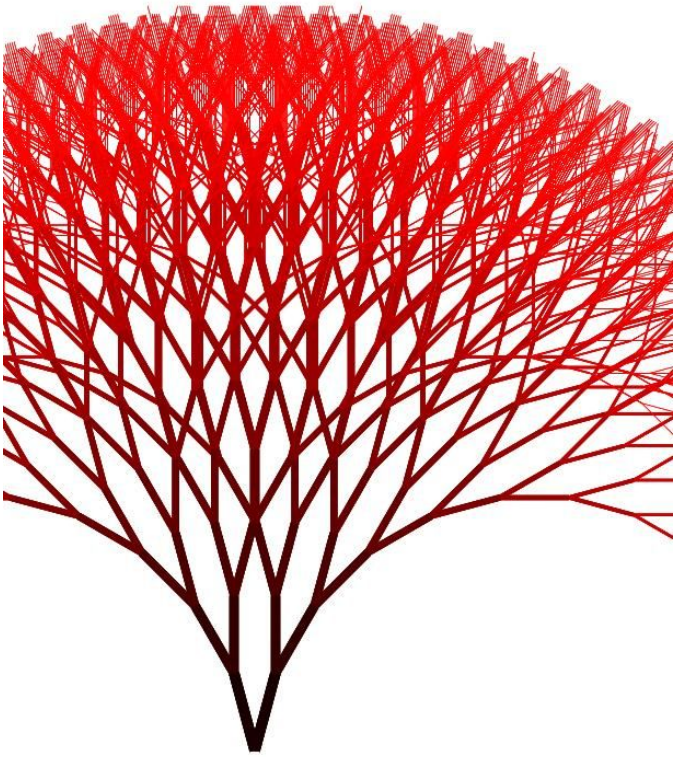
Comments begin with a pound sign and end at the end of a line.

Here is a simple program that will draw a red tree.

```
#Simple red tree
```

```
Generations=10
Red=1

#Begin execution here
Start:
 rightLine

#Every iteration will replace the string rightLine with
#all indented lines following it.

rightLine:
   Right
    Draw
     rightLine
   Left
    Draw
     rightLine
```

This image will be produced.



---Language Reference Manual

1. Language Overview

Organic Form Language, OFL, is based on L-systems. OFL adds structure through blocks and methods. Code blocks define scoping. A method is a code block with a name. Indentation is used to define code blocks.

2. Lexical Conventions

There are six types of tokens: keywords, identifiers, comments, operators, new line followed by white spaces, and integers. Operators, white spaces, and newlines are used to separate tokens.

2.1. Keywords

The keywords begin with an upper case ASCII letter, which is followed by only upper and lower ASCII letters.

The keywords are:

Right, Left, Draw, Thickness, Length, Angle, Red, Green Blue, Generations, Start, and Final.

2.2.identifiers

Identifiers begin with a lower-case ASCII letter, and may have only upper and lower-case ASCII letters, integers, and underscores, '_'.

2.3.Comments

Comments begin with a pound-sign, '#', and continue until the first newline.

2.4. Integers

Integers are one or more decimal digits between '0' and '9'.

2.5. White Spaces following a Newline

A white space immediately after a newline indicates a code block. Otherwise white spaces are ignored.

2.6. Operators

Non-letter ASCII symbols are used as operators. They are one character long.

They are:

':','=','-','+'

3. Grammar

3.1 Indentation

Like Python, the structure of the code in OFL is indicated with indentation. Only code blocks are indented. Initialization of environment members,3.2.1.2, and Generations,3.2.2.2, always begin on a new line and are not

indented. Any method declaration will not be indented, and will immediately precede one code block.

Indentation expresses environment inheritance in which instructions will be executed. Each instruction inherits the environment from the last instruction more shallowly indented. The execution environment is not effected by those instructions more deeply indented.
Only an instruction which is more shallowly indented than the immediately subsequent instruction will have any effect on the execution environment of another instruction.

For example, here are a series of commands in OFL.

```
a1
b2
      c3
d4
```

In this case, a1 will not effect c3 because b2 is not more deeply indented than a1. The only execution environment being modified is c3 because it will inherit b2's environment, and any changes to the environment caused by b2.

3.2. Keywords

In OFL, a keyword is a directive, environment member, or a flow control statement.

3.2.1Directives and Environment Members

Both directive and environment members change the runtime environment of an OFL program, but directives also direct the display program.

3.2.1.1.Directives

Directives either rotate the direction in which future drawing is done, or direct the display program to draw.

They do not take a value, so they only can share a line with white spaces.

3.2.1.1.1Turning Commands

Right and Left rotates the direction of drawing clockwise or counter clockwise by the value of Angle, 3.2.1.2.4.

3.2.1.1.1.1.Right

Turns direction of drawing clockwise.

3.2.1.1.1.2. Left

Turns direction of drawing counter-clockwise.

3.2.1.1.1.3. Draw
Directs the display program to create a visible mark of Length, 3.2.1.2.2, long from the current position in the direction of drawing with a width of Thickness,3.2.1.2.1, pixels in of the Color,3.2.1.2.3 . This is the only means in OFL to paint on the screen.

3.2.1.2. Environment Members
Environment members only change the state of the runtime of an OFL program.

They always take an integer value, so
they are always followed by an equals sign, '=', and either an integer or a plus or minus sign, '-', '+'.

3.2.1.2.1. Thickness
The value of Thickness dictates the thickness of lines painted when Draw is called.
3.2.1.2.2. Length
The value of length dictates the length of a line painted when Draw is called.
3.2.1.2.3. Red, Blue, Green
The value of these colors dictates the color of a line painted when Draw is called.
3.2.1.2.4 Angle
The value of Angle dictates how much direction of drawing is changed when Right or Left is called.

3.2.2.Flow Control Statements
Flow control is based around the concept of a generation in recursive calls. A generation is a unit of growth of the drawn form. Methods may be declared to be “per generation”, which tells the system that they may only be invoked once for each generation in each call path.

For example, if “twig” is a per generation method and recursively calls itself twice, and is called by “branch” twice, twig will execute twice during the first generation, then four times the second generation, then eight times during the third generation. If twig is not per generation, then it will run indefinitely.

3.2.2.1.Generation

Sets the total number of times rules will be replaced in each call path. This always takes an integer value, so it is followed by an equals sign '=' and an integer. It is only set once in any program, and it is always on its own line with no preceding spaces but may have trailing white spaces before the subsequent newline.

3.2.2.3. Start

The value of Start is the first method call made during the start of any run of the OFL program.

This always takes an identifier value, so it is followed by an equals sign ':' and an identifier. It is only set once in any program, and it is always on its own line with no preceding spaces but may have trailing white spaces before the subsequent newline.

3.2.2.4. Final

Final is an optional control statement after the name of a new rule, but before the colon. It is the alternative value of the method during the final generation. It is similar to an identifier definition in that it has a declaration line followed by a code block which is more deeply indented than the declaration.

3.3. Identifiers

An identifier names a user defined method.

3.3.1.Identifier Definition

The Identifier definition is an identifier declaration line followed by a code block.

3.3.2.Identifier Declaration

An identifier declaration names the method and includes any modifiers.

The identifier is followed by a modifier and colon, or just a colon. Following the colon is a newline.

3.3.3. Identifier Method Block

A method block contains only indented lines. Only method blocks contain indented lines.

3.4.Operators

3.4.1.Assignment Operators

The assignment operators give a value to a keyword or identifier, which are the “lvalue”.

An equals sign is used if, and only if, an integer value being set.

3.4.1.1.Colon

Colons set the lvalue to code block.

A colon may possibly have a modifier between it and the lvalue.

A colon is the end of a statement on a line, so it is only followed by white space before the newline.

3.4.1.2 Numeric Assignment

Numerical assignment maybe abbreviated by using an increment or decrement value instead of an integer.

3.4.1.2.1 Equals

An equals sign is required to set an lvalue to an integer. It sits between an lvalue, and either an integer value or a minus or plus sign.

3.4.1.2.2 Minus

Minus may be used instead of a integer in a numeric assignment to indicate decrement by one the value of lvalue.

3.4.1.2.3 Plus

Plus may be used instead of a integer in a numeric assignment to indicate increment by one the value of lvalue.

3.4.2 Pound

Pound indicates the beginning of a comment. Everything between a pound sign and the first subsequent newline is a comment and is ignored. A comment may not share a line with any other statement.

---- Project Plan

The project was developed serially with the goals established before the design, which was completed before the coding. Small corrections in the preceding steps were necessary along the way.

The coding of the compiler was analogous to the steps of execution of the compiler; the front end was developed first, then the first execution phase of the back-end, and finally the PostScript output generator.

Originally CVS was going to be used to manage the history of all code files, but was unnecessary because nearly all code was developed
in one Antlr file. Using only a single file for code also greatly

simplified the build system.

The schedule was based on the class schedule deliverables, with the completion of the front end coinciding with the due date of the Language Reference Manual. The back-end was not divided into separate tasks.

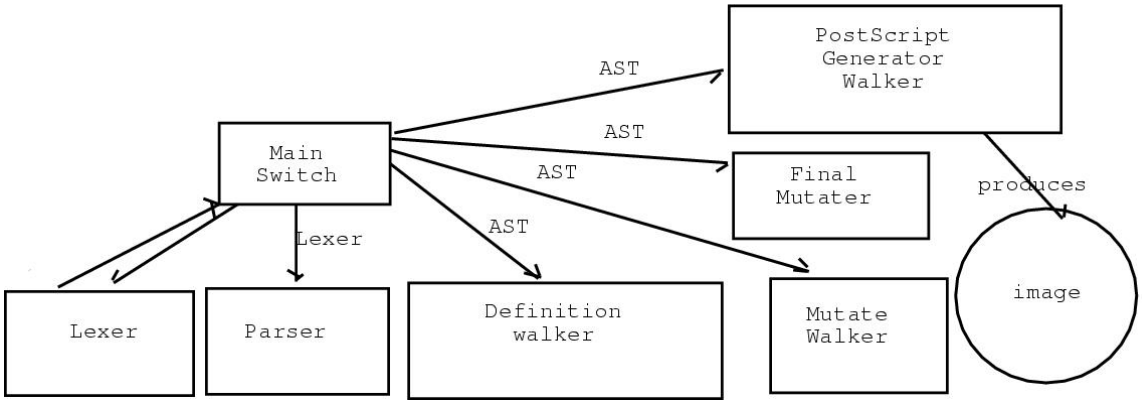
Testing was frequent because the time needed to execute a build/run cycle was never more than one minute. The testing begin with simple programs as soon as the lexer was ready. I maintained a group of test input files which for regression testing and ran them with a script whenever significant changes where made to the compiler.
The output was manually inspected.

Development was done on a Linux machine using Antlr and IntelliJ Idea, which became critical for debugging and development of the complicated tree walker in my compiler.

---- Design

The compiler is unique in that the program which it compiles is executed in the AST tree walker. The AST is altered by the execution of the statements of the OFL program. There are actually four tree walkers in the OFL compiler. The first finds the rule definitions and keyword values. The subsequent walkers only walks the tree which represents the “Start” rule. The second walker is the Mutator and it is called repeatedly. It replaces all references to user rules with their values found in the preceding tree walk. The third walker replaces rule references with their “Final” values. The last walker is the PostScript Generator. It prints the ASCII PostScript code of the image.

The Main Switch component acts as a switching station between the components generated by Antlr: Lexer, Parser, Definition Walker, Mutator, Final Mutator, and PostScript Generator. The Main passes the Lexer to the Parser, then passed the AST to rest of the components. It is the only code defined outside of the Antlr input file.



The Parser is also interesting in that it has to compare the values in the indentation tokens. Each indentation token has a value representing how deeply the subsequent statement is indented. That indentation needs to be compared with previous indentations to find where the subsequent statement belongs on the AST.

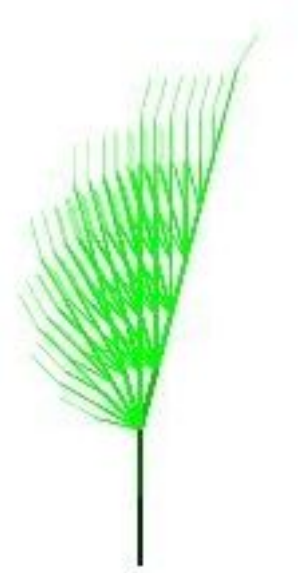
--Lessons Learned
Design your language with an eye toward the tools you are using. The way my program used indentation consumed about 30% of the time I spent on development. Most of the compiler is easy to design with Antlr because it is a good tool, but working against it is very difficult.

Likewise, using the AST tree walker was not a good way to execute the final OFL program. That consumed about 60% percent of my development time, and the majority of that was debugging.

--Code Examples
Each program listing is followed by the figure is generates.

```
#Long green grass

Generations=7
Green=2
Length=23
Thickness=4
```

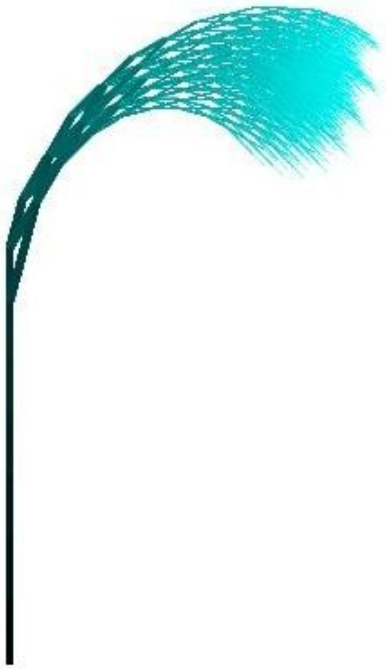
```
Angle=15

#straight line
stalk:
 Draw
  Draw

#Program begins execution here
Start:
 stalk
  stalk
  Right
   wash

#Definition of the user word wash
wash:
 Draw
  wash
 Left
  wash

#Definition of what to do when printing wash
wash Final:
  Right
   Draw
```



```
#Tall aqua stalk with a wisp on top

Generations=9
Red=0
Blue=1
Green=1
Length=37

Thickness=4
Angle=15

#just a long poll
stalk:
 Draw
  Draw
   Draw
    Draw
     Draw
      Draw

Start:
 stalk
      rl

#Which direction the plant leans
lean:
   Right

#bushiness
rl:
 Draw
   lean
```

rl
lean
Draw
Draw
rl

rl Final:
lean
Draw



#Thick green plant

Generations=6

#greenish yellow

```
Red=1
Blue=0
Green=1
Length=40
Thickness=4

#Make a bush
#Add tall branch leaning to the right
Start:
    Right
        rl
    rightlong

rightdraw:
     Right
       Draw

#Right leaning branch
rightlong:
    Right
      rightdraw
+--  5 lines:
rightdraw----------------------------------------------------------------------------------------

#Branch a lot
rl:
 branchr
  Draw
   rl
    rl
 branchl
  Draw
   rl

rl Final:
  Right
   Draw

branchr:
 Right

branchl:
 Left
```