FRAC: Recursive Art Compiler

COMS W4115 - Language Proposal

September 30, 2015

Team

- **Anne Zhang** `az2350` - Manager
- **Kunal Kamath** `kak2211` - Language Guru
- **Justin Chiang** `jc4127` - Systems Architect
- **Calvin Li** `ctl2124` - Tester

Description

FRAC enables a programmer to generate fractals in the Graphics Interchange Format, commonly abbreviated as GIF. Specifically, the user will be able to program a fractal using a formal grammar, as well as incorporate algorithms into their fractal generation.

We define our fractal-generating grammars as:

```
G = (init, rules)
```

where,

- **init:** a string that defines the initial state of the system
- **rules:** a set of rules that defines the way variables are replaced with combinations of variables and movements during the recursion. Each rule has two halves to it, separated by an arrow (→). There are two types of rules: recursive rules, which point to a string, and terminal rules, which point to a function

Syntax

Declaration/Assignment

Variables are declared and assigned in the same syntax as C.

```
int a;
a = 0;
int b = 1;
```

Data Types

- **int:** an integer of 32 bits
- **double:** a floating point number of 64 bits
- **bool:** a true or false value
- **string:** a collection of characters/symbols
- **rule:** a standard object composed of 2 fields
  - a *predecessor* string
  - a *successor* string or function
- **gram:**: a standard object composed of 2 fields
  - an *init* string
  - a set called *rules*

e.g. Consider the following grammar declaration:

```
gram G = {
    init: 'X',
    rules: { 'X' → 'X up X down X',
             'up' → turn(90),
             'down' → turn(-90),
             'X' → move(1)
    }
};
```

Structures

- **primitive:** fundamentally built into the language
- **object:** Javascript-like object with (key, value)
- **set:** unordered collection of objects or other data types

Operators

Mathematical and logical operators are the same as in C (e.g. + - * /, && ||, < > ==).

Control Flow

If/else statements, while loops, and for loops are the same as in Java.

Functions

Functions have return values and can take parameters. They are defined with the keyword "func", followed by the function name, parameters contained in parentheses and separated by commas, and brackets.

```
func add(a, b) {
    return a + b;
}
```

Built-in Functions

- **main** () - entry point upon which program executes
- **move** (int len) - draws a line of length len. Implements turtle graphics, which graphs line segments based on a given orientation and relative position
- **turn** (int theta) - Turns the turtle's orientation by an angle of theta (degrees in the polar coordinate system)
- **draw** (gram G, int num) - generates a static fractal image. 'num' corresponds to the recursion depth of the grammar, and will indicate how many times the grammar's ruleset should be applied in the drawing
- **grow** (gram G, int num) - generates a GIF of the fractal being created at each recursive step
- **zoom** (gram G, int num) - generates a (seemingly infinite) zooming GIF of the fractal

Comments

Just like in Java, // for single line and /* */ for multi line comments.

Proposed Applications

Since FRAC makes visualizing fractals and other iterative shapes simple, we see it being quite useful to mathematicians or curious students who wish to explore the world of recursive images with minimal programming experience. Furthermore, the ability to create GIFs with FRAC will result in not only the spread of fractal art, but will also help visual learners better understand the concept of the self-similar pattern.

Additionally, because FRAC also supports Java-like control flow in addition to fractal-specific methods, users have the opportunity to algorithmically manipulate fractals. See sample program #2.

Sample Programs

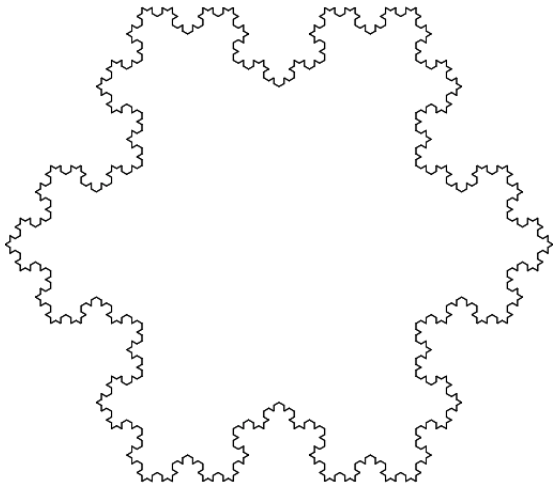
Generate a static image of the Koch snowflake

```
/*
 * See https://en.wikipedia.org/wiki/Koch_snowflake for reference
 */

gram koch = {
    init: 'F r r F r r F',
    rules: { 'F' → 'F l F r r F l F',
             'r' → turn(60),
             'l' → turn(-60),
             'F' → move(1)
    }
};

main() {
    draw(koch, 4);
}
```

Expected output:



Polyzygotic snowflakes

```
/*
 * Uses the ability to access object properties to create
 * conjoined twins, triplets, etc. of your desired fractal!
 */

gram koch = {
  init: 'F r r F r r F',
  rules: { 'F' → 'F l F r r F l F',
    'r' → turn(60),
    'l' → turn(-60),
    'F' → move(1)
  }
};

func twin(gram g, int n) {
    string new_init = '';
    for(int i = 0; i < n; i++) {
      new_init = new_init + g.init;
    }
    g.init = new_init;
    return g;
}

main() {
  draw(twin(koch, 3), 4);
}
```

Expected output: