PLT Team

Adam Vartanian asv2105@columbia.edu
Boriana Ditcheva bhd2105@columbia.edu
Marinos Constantinides mc2570@columbia.edu
Yianni Alexander rypa2101@columbia.edu
Yavor Tchakalov ytt2101@columbia.edu

Project Proposal

Patternizer™
a software solution for recursive pattern creation and manipulation

Section 1 Overview

1.1 Description of the Language

The language will be used to create geometrical patterns, which would otherwise be difficult and time-consuming to create, unless one has knowledge of graphic design tools. The language works by combining one or more simple geometric figures into *one* object or entity (called a *pattern* in our language). This pattern can be a circle, an arc, a pentagon, or simply a collection of lines. The user can then manipulate this pattern further by scaling, translating, rotating, tiling, or concatenating it with another pattern; thus, arbitrarily complex geometric patterns can be created recursively.

1.2 Fundamentals: primitives

At the core of our language lies the concept of a *pattern*. A pattern is essentially a template of how to draw something. An actual copy of a pattern we will call an instance. Patterns feature three intrinsic properties: origin, angle, and scale. The values of these properties are not absolute. They are calculated in relation to the parent pattern, the pattern which was used to create the current one. Each pattern comes equipped with two default constructors, one that defaults its intrinsic properties to (0, 0), 0º, 1.0 respectively, and one that allows the user to specify them. The language also provides default accessors and mutators for its intrinsic properties.

A pattern can also have custom properties defined by the user to avoid redundant coding of new patterns. Thus we can have a generic Arc pattern whose number of degrees can be changed at will, instead of needing to define a 60DegreeArc pattern, a 90DegreeArcPattern, etc.

There exists the concept of a *native* pattern in Patternizer. That is the *Point* pattern, which represents a single point. A Point will be viewed as a vector with regard to its intrinsic properties. Thus scaling an instance of a Point (e.g. modifying its scale from 1.0 to 2.0) will apply vector scaling with respect to the origin of the Point instance.

The remaining native data types or primitives will be integer and floating point numbers.

1.3 Fundamentals: operators

Integers and real numbers will have all basic arithmetic operators defined on them: addition, subtraction, multiplication, division, and modulo.

Additionally, patterns will be able to be operated upon via several operators:

Pattern	[translate]	Point	=>	Pattern	(translation)
Pattern	[rotate]	Real	=>	Patter	(rotation about origin)
Pattern	[scale]	Real	=>	Pattern	(proportional scaling)
Pattern	[mirror vertical]		=>	Pattern	(vertical flip)
Pattern	[mirror horizontal]		=>	Pattern	(horizontal flip)
Pattern	[scaleof]		=>	Real	(get scaling factor)
Pattern	[lengthof]		=>	Real	(distance from origin)
Pattern	[angleof]		=>	Real	(angle from vertical
Pattern	[positionof]		=>	Point	(get location)
Pattern	[draw]		=>	Pattern	(the object is marked for visualization)

1.4 Miscellaneous

We'll probably have a standard library of basic patterns (like Circle, Line, Triangle, Square, etc) like the C Standard Library. Standard control structures like **for**, **while**, and **if** would also be included, with normal logical operators. **==** and **!=** would be the only valid operators on objects.

Section 2 Advantages

2.1 Why Learn Patternizer?

A user will be able to quickly learn this simple and concise language and be able to create arbitrarily complex patterns from simple geometric primitives like lines, circles, and squares. One of the strengths of the language is its recursive nature. This feature not only adds power to what a user can do with the language, but also encourages the user to think in a different way about patterns. The user thinks about new patterns in terms of combining already existing ones, thus simplifying the creative process.

Section 3 Sample Use

3.1 Sample Program #1

Patterns are declared by the keyword pattern, followed by an identifier for the pattern, a list of parameters (somehow), and a set of statements declaring how to draw the object. Thus, for example, if Line() produced a vertical line from 0,0 to 0,1, we could have:

```
pattern Rectangle() {
       y = Line() [scale] 10;               // x is a line from (0,0) to (0,10)
       [draw] y [translate] Point(0.5,-5); // draws a line from (0.5,-5) to (0.5,5)
       [draw] y [translate] Point(-0.5,-5);//draws a line from(-0.5,-5) to (-0.5,5)
       x = Line() [rotate] 90;             // x is now a line from (0,0) to (1,0)
       [draw] x [translate] Point(-0.5,5); // draws a line from(-0.5,5) to (0.5,5)
       [draw] x [translate] Point(-0.5,-5);//draws a line from(-0.5,-5) to(-0.5,-5)
}
```

3.2 Sample Program #2

We could also implement a hypothetical [line] operator by a parameterized Line pattern. For instance, if Line() gave you a vertical line from 0,0 to 0,1:

```
pattern Line(Point a, Point b) {
  x = Line() [scale] ([lengthof] (a [subtract] b));
  x = x [rotate] ([angleof] (a [subtract] b));
  [draw] x [translate] b;
}
```

Thus our Rectangle pattern would end up as:

```
Pattern Rectangle() {
       [draw] Line(Point(0.5,-5), Point(0.5,5));
       [draw] Line(Point(0.5,5), Point(-0.5,5));
       [draw] Line(Point(-0.5,5), Point(-0.5,-5));
       [draw] Line(Point(-0.5,-5), Point(0.5,-5));
}
```

3.3 Sample Program #3

If we have already created two patterns Rect1 and Rect2 by one of the methods above, we can combine them into one pattern, DoubleRect.

```
Pattern DoubleRect() {

      [draw] Rect1();

      [draw] Rect2();

}
```

Now, we can create a new pattern from this one by rotating it a different number of degrees.

```
Pattern RotatedRects(){

      x = DoubleRect();

      for( i = -2, i < 3, i++){

       [draw] x;

       x [rotate] (i*15);

     }

}
```

Section 4 Results

4.1 The Sample Drawing