Pts

A language to animate a set of pts as a function of time.

Author: Michael Johns

Functions

The language will provide the ability to define pure functions with a similar syntax to Ocaml. All values in the language will be doubles and every function must be a function of double to double.

```
sin_squared = t -> let s = sin t in s ** 2;
double = t -> 2 * t;
```

The output of one function can be piped into another using the >> operator.

```
double_then_sin_squared = double >> sin_squared;
```

Functions can be used as expressions in arithmetic operations.

```
double_sin = sin + sin;
```

double_sin 3 would produce the sin 3 + sin 3

Matrices

You will animate the points by chaining together a series of transformation matrices. All matrices are 3 x 3 and defined using square brackets. The matrix is specified as a series of 9 expressions which can either be a function of time (any single param function) or a scalar value. The expressions are space separated and complex expressions must be wrapped in a set of parenthesis. As the animation progresses the time value maintained by the program will be incremented. At each step of the rendering the functions will be evaluated with t so that an all scalar matrix can be used for the actual transformation.

```
m = [
  double,   sin_squared,  0,
  0,          0 ,             1,
  1,          2,              double + sin_squared
]
```

you can define 4 x 4 matrices to define translations. All 3 x 3 matrices will be extended to 4x4 during multiplication by adding 0
0
0
0 0 0 1

when the expression (double + sin_squared) is evaluated with t, it will produce the result of (double t) + (sin_squared_t). All mathematical operators are valid in these matrix expressions as well as the chaining operator >>.

Points
A point is a vector of 3 doubles x, y, z. The point is defined with a similar syntax to the matrix using a comma separated list of literals between square brackets.
p = [1.0, 2.1, 3.0]

Sets
Sets are used to define a set of points and can only contain these 3 value vectors. The set provides the ability to add a new point. They are used to group series of pts together who should share the same transformations.

```
mySet = {};
mySet.add [ 1.0, 2.0, 3.0];
anotherSet = { [1.0, 2.0, 3.0] };
```

Applying transformation matrices.
You define a series of transformations for each set of points and register it to be animated.

```
animation m1 -> m2 -> m3 -> mySet;
animation m1 -> m3 -> mySet;
```

Animating it all

```
animate {
  range: 0.0 -> 200;
  stepSize: 0.1;
  stepRate: 10ms;
}
```

A loop will be generated where t is incremented at the specified rate and with the specified steps. At each step the functions in the matrices will be evaluated to produce scalar matrices which will be left multiplied to transform the point. The transformed point will then be rendered.

Implementation
The code will be compiled to a single Java class with static functions. And the main function driving the animation loop. An appropriate java lib will be found to render the 3d output. That will then be compiled to bytecode using the Java compiler which will in turn be executed.

Example program

```
t = x -> x; // Function that just returns the value
```

```
scale = [
   t, 0, 0,
   0, t, 0,
   0, 0, t
]

translate_x = [
       1, 0, 0, t,
        0, 1, 0, 0,
        0, 0, 1, 0,
         0, 0, 0, 1
]

myPts = { [ 1, 2, 3], [3, 4, 5]}
animation scale -> translate_x -> myPts

animate {
   range: 0.0 -> 200;
   stepSize: 0.1;
   stepRate: 10ms;
}
```