Pts

An algebraic point animation language

Michael Johns (mjohns@google.com)

Reference Manual

1. Lexical conventions

There are five kinds of tokens: identifiers, keywords, constants, operators, and other separators. Spaces, tabs, and newlines will be referred to as whitespace throughout and have no impact on program semantics beyond separating tokens. In other words, each sequence of whitespace could be collapsed to a single space with no effect on a programs behavior. “;” is used as a statement terminator throughout.

1.1 Comments

The characters /* introduce a comment, which terminates with the characters */. C++ style comments are also supported. These comments begin with the characters // and continue until a newline is reached.

1.2 Identifiers (Names)

An identifier is a letter followed by a sequence of letters and digits. “_” is considered a letter. Two identifiers are considered the same if they have the same sequence of letters and digits. Identifiers are case sensitive. In other words ID1 is considered a different identifier than id1.

1.3 Keywords

The following identifiers are reserved keywords:
const, func, matrix, point, points, for, from, to, render, sleep

1.3 Numeric Constants

Numeric constants are expressed as decimal point numbers. They consist of an integer part followed by a fractional part which consists of a decimal point followed by an integer value. The integer values consist of a sequence of the digits 0-9. At least one of the integer and fractional part must be present. This can be expressed formally with the following regular expressions:

digits = [0-9]+
digits(.digits)? | digits?.digits

Examples: 1.3, 1, .1, 0.123

2. Types

2.1 Number

Numbers are the primary data type in the language. Numbers can be defined using numeric constants as outlined in **1.3** and are represented as double precision floating point values. A numeric constant can be defined using the *const* keyword as follows:

```
const PI = 3.145;
```

2.2 Point

A point is a traditional 3 dimensional point whose coordinates are specified using numbers. The point consists of an x, y, and z coordinate. To define a point you use the keyword *point* followed by an identifier and assignment. The rvalue of the assignment must be of the form: { x_value, y_value, z_value}. Once the point is definied it is immutable.

Examples:

```
point a = { 1, 2, 3};
point b = { 1.0, 1.0, .3};
```

2.3 Points

You can define a collection of points to animate using the *points* data type. You define an instance using the keyword *points* followed by an assignment to an initial collection of points. [] produces an initially empty collection. You can create an instance with initial values using the following syntax: points my_pts = [pt1, pt2, pt3];

You can add additional points the the collection using the add method available on all *points* instances. Example: my_pts.add pt1;

2.4 Matrix

You can define 3x3 and 4x4 matrices using the keyword *matrix*. You specify the matrix rvalue using “[values]” where values is a a comma separated list of numeric expressions or function identifiers. The size of the matrix is determined by how many values appear in the list. The values are specified starting with the top row followed by the next and row and so on. All 3x3 matrices will be implicitly extended to 4x4 matrices by adding all 0s in the 4th row and column except for the value in row 3, column 4 (indexing starting at 0) which contains a value of 1.

For example, the identity 3x3 identity matrix would be represented as follows:

```
matrix my_identity = [
        1, 0, 0,
        0, 1, 0,
        0, 0, 1
];
```

Again whitespace is only meaningful in terms of making the matrix more readable.

3. Numeric Expressions

Numeric expressions are built using numeric constants and a set of binary operators including basic addition and multiplication operators +,-,*,/. Exponentiation is also supported with the ^

operator. * and / are at a lower precedence than ^. And + and - are at the lowest precedence as expected. All operators are left associative. "-" can also represent negation and is at the lowest precedence.

4. Functions

Function definitions must be placed at the top level and can not be nested inside any other blocks or within other functions. A function definition begins with the keyword *func* followed by an identifier for the function name. The function name is followed by "=" and a space separated list of argument names ending with "->". The function body follows after the "->" and must return a numeric value and end with a statement terminating ";".

    func my_name = arg1 arg2 -> *function_body*;

The function body can be an expression evaluating to a numeric value. Or it can be a series of let statements defining variables to be used in the final numeric expression.

    function_body -> numeric_expression
        | let *variable_id* = numeric_expression in *function_body*

Examples:

```
func add_five = x -> x + 5;
func foo = y -> let dbl = y * 2 in dbl ^ 2;
func bar = t ->
   let temp1 = t + 1 in
   let temp2 = t + 2 in
   temp1 * temp2;
```

A function only has access to global constants and the parameters from within the function body.

When invoking a function you specify the function identifier followed by a whitespace separated list of parameters.

5. For loops

A for loop allows you to iterate over a range of integer values. For loops use the following syntax:
for identifier *from* start *to* end { loop_body }
Identifier is the variable name in which the current index will be stored. Start is the initial value the identifier will contain on the initial iteration of the loop. The range is inclusive of start to end. Start can be less than end and in those cases the identifier will be decremented at each iteration of the loop.

Example:

```
for i from 0 to 4 {
   point p = {i, i, i};
}
```

```
    // p will be {0, 0, 0}, then {1, 1, 1} … and finally {4, 4, 4}
```
Start and end must be integers (i.e. no decimal point) but will be treated as double precision floating point numbers when passed to functions or used in numeric expressions.

6. Special Functions

There are a few special functions used for rendering the points and performing mathematical computations .

6.1 Render

The first function is render which allows you to render a set of transformed points at a specified time value. The first argument is a series of matrices multiplied together followed by a point or set of points which will be referred to as a transformation.

transformation -> *point* | *points* | matrix * *transformation*

Examples: matrix_a * matrix_b * my_point, my_points, matrix_a * {1.0, 2, 3}

The second argument, *t*, to render is the current time value to use when evaluating the transformation. Render does not return any value.

6.2 Sleep

Sleep allows you to control the frequency that you render new transformations creating animations. Sleep takes one argument which is a numeric value for the number of milliseconds to sleep. It does not return any value.
Example: sleep 50; // sleeps for 50 ms

6.3 Mathematical Functions

Basic mathematical functions are supported including sin, cos, tan. They are simple functions taking a double and returning a double and are treated in the same way as algebraic functions you define yourself using the *func* keyword.

7. Semantics

7.1 Scope

Scoping is static. Variables defined at the top level are in the global scope and are available after their definition. For loop bodies define their own scope with variables local to each iteration of the loop in the same way as in C, C++, Java. For loop bodies have access to the global scope and any other block in which they are embedded. Function bodies only have access to their parameters and local variables defined using let, with one exception. They have access to global *const* variables defined before the function definition

8. Example Program

```
const PI = 3.14;
```

```
func my_cubic = x -> 3*x^3 + 2*x^2;

func foo = y -> PI * y + my_cubic y;

func bar = t ->
   let temp = foo t in
   t * temp;

matrix a = [
   1.0, 2.0, 3,
   0,  1, foo
   my_cubic, 0, 1
];

matrix identity = [
   1.0, 0, 0,
    0, 1, 0,
    0, 0, 1
];

matrix bar = a * identity;

points my_points = [];
for i from 10 to 0 {
  point a = { i, 1.0 * i, PI};
  my_points.add a;
}

for i from 0 to 4 {
  render (bar * my_points) (i * 2.3);
  sleep 50;
}
```