Pts

An algebraic point animation language

Michael Johns (mjohns@google.com)

Language Proposal

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

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 or 4 x4 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;
}
```

Introduction

Pts is a language to animate a set of points using transformations matrices whose values are a function of time. To rotate a point in a circle you would simply provide a single rotation matrix where the rotational values were simply a function of t that controls the rate of rotation.

A simple program would look like:

```
file: simple.pts
func dbl = t -> 2 * t;
const PI = 3.14;
func quad = t -> t ** 2 + PI * t;
```

```
points pts = [ {1, 0, 0}];
matrix transform = [
  1, 0, 0,
  0, dbl, 0,
  0, 0, quad
];

for i from 0 to 100 {
  render transform pts i;
  sleep 10;
}
```

To compile and run you would do the following:
./pts.sh simple.pts
java Pts

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 += operator.
Example: my_pts += 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 transformation matrix or a chain of matrix multiplication. The second argument is the set of points to animate and the third argument to render, *t*, 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

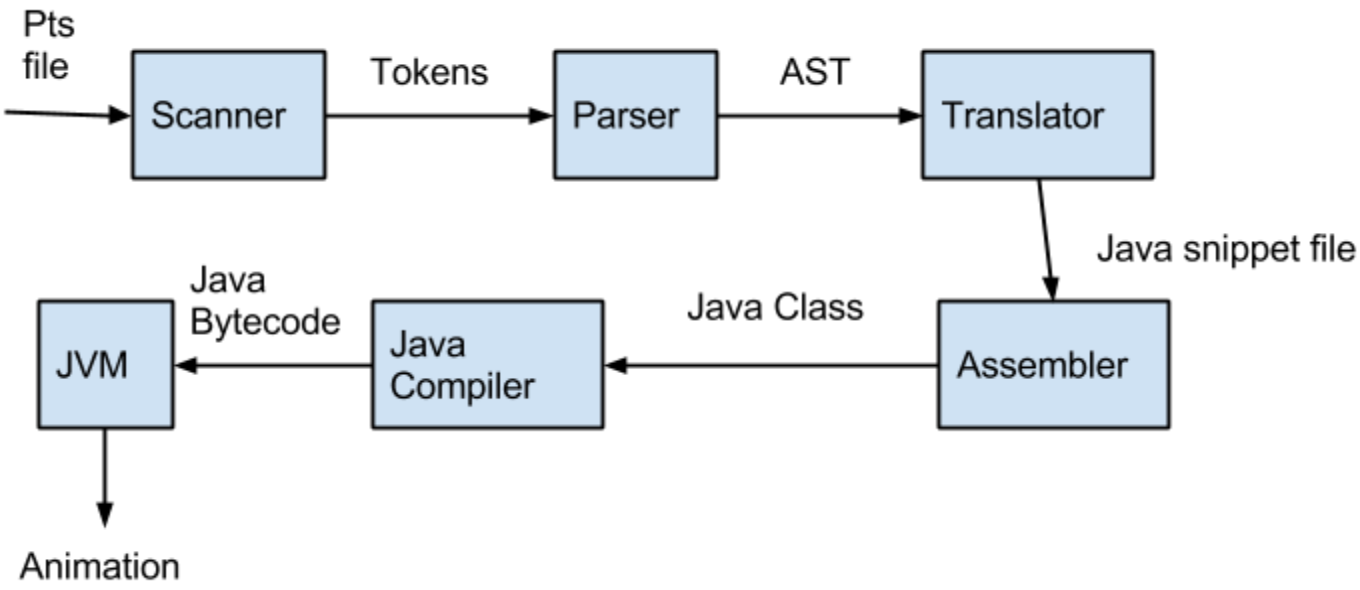
Scoping is static. Variables defined at the top level are in the global scope and are available after their definition. For loop bodies define their own scope with variables local to each iteration of the loop in the same way as in C, C++, Java. For loop bodies have access to the global scope and

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 += a;
}

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

Architecture



Scanner
The scanner takes the pts program and translates it into a series of normalized tokens. Tokens include things such as keywords, semicolons, set braces, operators, etc. This allows all subsequent steps of compilation to not worry about formatting issues like whitespace and line breaks. The scanner is built declaratively using ocamllex which produces the sequence of tokens for a given input. Token types are specified using regex patterns.

Parser
The parse take the sequence of tokens and identifies groupings of tokens that represent valid constructs in the language. These groups of tokens are turned into corresponding ocaml types which form the abstract syntax tree (AST) for the input. The parser also recognizes invalid programs that do not satisfy the grammar of the language. There is no semantic analysis or type checking done during this phase though.

Translator
The role of the translator is to take the AST and to translate it into a valid snippet of Java code. Java classes are defined outside the scope of the ocaml translator which handle the basic types needed to render the animations. The java classes are roughly analogous to the types that can be represented in the pts language and will be discussed in more detail later.

The first step of translation is creating a symbol map of id name to id type. For example, if the program defines a function as follows: func x = y -> y * 2; then an entry will be stored in the map for x with type function. This is necessary in order to determine how ids should be translated in

certain contexts. The primary case where this comes up is determining how to translate a value in a matrix definition. If you come across an id in this context you don't know if that id is for a function or if it is a constant. These two types must be handled differently since every matrix value within java must be a function. If the id represents a constant than it must be translated to a function that always returns that constant value.

After the symbol map is built, the next pass over the AST translates the different constructs into the appropriate valid java code. The output of this pass is a string of Java code which is written to an output file. The output file is not itself a valid java program.

Assembler

The assembler takes this java snippet and joins it with the framework Java class to produce a valid Java class that can be compiled and run. The framework Java code provides a run method hook where the Java snippet is supplied. When main executes it executes this method producing the animation. The joining of the framework java code and snippet is done within a base script which also handles the Java compilation.

Java Framework Classes

The framework defines classes for the basic type needed to render the animation. The most basic type is an interface for functions.

Function

A function has a method call wich takes a double and returns a double. This maps 1-1 with the pts func type which has a single numeric argument and returns a number. Utility methods for performing basic operations on functions is also supplied. Multiplying two functions produces a new function which when evaluated, evaluates the two original functions and multiplies the results. Similar functionality is also provided for other simple operations such as addition.

```
interface Function {
  double call(double arg);
}
```

Point

The point object is a simple wrapper around a 3d point. The main advantage of defining a class for this as opposed to simply using a double array is that you can construct point objects the same way in all contexts using the constructor which takes x, y, and z coordinates. The array would require more complicated java to construct in contexts where the literal {x, y, z} syntax is not valid (outside of variable definition). The poiint object also automatically extends the point to have a 4th h dimension that allows for easy multiplication with 4x4 matrices and normalization in 3d perspective transformations.

Matrix

The matrix object is a 4x4 matrix whose values are functions. The constructor supports taking the 9 values from a 3x3 matrix and extending it to 4x4 behind the scenes. Multiplication methods are supplier for matrix * matrix and matrix * point. When multiplying two matrices the functions are multiplied using the utility method defined for the Function interface. This allows the matrix functions to only be evaluated when multiplying by an actual point. Multiplying a matrix by a point also takes the time t as an argument. When multiplying the matrix and point, the matrix's functions are evaluated with t and then multiplied by the point's coordinate.

Render

The render function takes a matrix transformation, a list of points, and the time t and applies the matrix to each point at the give time producing the transformed point. The current implementation only prints out the transformed values but feeding these into an existing rendering framework would be a simple process.

Testing

Given time constraints only manual testing was done although thought was put into the desired testing patterns. Ideally standard junit tests would exist for all of the framework classes that ensure matrix transformations produce the desired results and different inputs are handled correctly.

The compiler would support a flag that writes out the series of transformed points to a file in a normalized format. Input programs would be written testing the various program cases and the output file containing the time series of transformed points would be compared against the expected results. This would remove the dependency on the actual rendering which would be more difficult to test since it requires some level of visual verification.

Desired Test Cases

- Defining types with identifiers that conflict with Java keywords. For example: func double = x -> x * 2; (This is currently not handled by the compiler! A simple solution was to add a prefix to all ids in the parser although defining an id type produced shift/reduce conflicts related to determining if the id is an id in and expression or an function id in a function application expression)
- Creating nested for loops and ensuring scoping of variables is handled correctly.
- Single matrix transformations using all literal.
- Multiplying matrices containing both literals and function ids.
- Exercising all valid syntactic structures to ensure they parse correctly.
- Animating multiple points

Lessons Learned

You should try and write actual programs in your language from the start. You will come up

against problems where you want to do something and the language does not support it. For example with pts, when you are rendering the points you would want to have control over rendering options like the coordinates that are displayed. You would want easier ways to express common transformation matrices like rotations and translations as opposed to having to explicitly create the matrix.

I also learned that if you continue to simplify the language to make the scope feasible you can come out with something in the end that isn't incredibly useful. Pts would be much more powerful if it was written simply as a Java library instead of a full fledged language. You wouldn't have the same issues and limitations. For example, it is very cumbersome to write a transformation matrix for an animation that translates linearly with time. You would need to supply a function that produces some value * t. So if you wanted to have the translation occur at some rate you would need to define a specific function for that constant. func t3 = t -> t * 3; If you wanted to update the speed you would need to define/alter the function. This could be solved by allowing functions to return other functions (more similar to a full lambda calculus) but that was cut due to time constraints.

Also error reporting is extremely important for a developers productivity. The parser just dies on invalid syntax which is not very helpful. It would also have been nice to pass line numbers along to the compiler so that it could report useful errors during the compilation type checking phase. In the current implementation type checking is mainly left to the java compiler but it makes it hard to link the errors with the actual code being written

Code

scanner.mll

```
{ open Parser }

let digit = ['0'-'9']
let digits = (digit)+
let fractional = '.'(digits)

let letter = ['a'-'z''A'-'Z''_']
let letter_or_digit = (letter) | (digit)

rule token =
  parse [' ' '\t' '\r' '\n'] { token lexbuf }
    | '+'   { PLUS }
    | "+="  { PLUSEQUALS }
    | '-'   { MINUS }
```

```
    | '*'   { TIMES }
    | "**"  { POWER }
    | '/'   { DIVIDE }
    | '('   { LPAREN }
    | ')'   { RPAREN }
    | '{'   { LBRACE }
    | '}'   { RBRACE }
    | '['   { LBRACKET }
    | ']'   { RBRACKET }
    | ';'   { SEMI }
    | ','   { COMMA }
    | '='   { ASSIGN }
    | "->"  { ARROW }
    | "for"      { FOR }
    | "from"     { FROM }
    | "to"      { TO }
    | "const"    { CONST }
    | "matrix"   { MATRIX }
    | "point"    { POINT }
    | "points"   { POINTS }
    | "func"     { FUNC }
    | "render"   { RENDER }
    | "sleep"    { SLEEP }
    | digits as int_lit { INT_LITERAL(int_lit) }
    | (digits)(fractional)? | (digits)?(fractional) as lit { LITERAL(lit) }
    | (letter)(letter_or_digit)* as identifier { ID(identifier) }
    | eof { EOF }
```

ast.mli

```
type operator = Add | Sub | Mul | Div | Pow

type expr =
    Binop of expr * operator * expr
  | Lit of string
  | Id of string
  | FuncApplication of string * expr

type point_literal = {
  x: expr;
```

```
  y: expr;
  z: expr;
}

type point =
    PointId of string
  | PointLit of point_literal

type matrix =
    MatrixChain of string list
  | MatrixLit of expr list

type def =
    FuncDef of string * string * expr
  | ConstDef of string * expr
  | MatrixDef of string * matrix
  | PointDef of string * point
  | PointsDef of string * point list

type stmt =
    AddPoint of string * point
  | Render of string * string * expr
  | Sleep of expr
  | Def of def
  | ForLoop of string * string * string * stmt list

type program = stmt list
```

parser.mly

```
%{ open Ast %}

%token FOR CONST MATRIX POINT POINTS FUNC LITERAL
%token SLEEP RENDER PLUSEQUALS FROM TO INT_LITERAL
%token LBRACE RBRACE LBRACKET RBRACKET ARROW
%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA EOF
%token PLUS MINUS TIMES DIVIDE ASSIGN POWER

%token <string> LITERAL
%token <string> INT_LITERAL
```

```
%token <string> ID

%right ASSIGN
%left PLUS MINUS
%left TIMES DIVIDE
%left POWER
%nonassoc ID

%start program
%type <Ast.program> program

%%

program:
  stmt_list { $1 }

stmt_list:
    /* nothing */  { [] }
  | stmt stmt_list { $1 :: $2 }

stmt:
    def SEMI { Def($1) }
  | SLEEP expr SEMI { Sleep($2) }
  | RENDER ID ID expr SEMI { Render($2, $3, $4) }
  | ID PLUSEQUALS point SEMI { AddPoint($1, $3) }
  | FOR ID FROM INT_LITERAL TO INT_LITERAL LBRACE stmt_list RBRACE
        { ForLoop($2, $4, $6, $8) }

def:
    CONST ID ASSIGN expr { ConstDef($2, $4) }
  | FUNC ID ASSIGN ID ARROW expr { FuncDef($2, $4, $6) }
  | POINT ID ASSIGN point { PointDef($2, $4) }
  | POINTS ID ASSIGN LBRACKET point_list RBRACKET { PointsDef($2, List.rev $5) }
  | MATRIX ID ASSIGN matrix { MatrixDef($2, $4) }

point:
    LBRACE expr COMMA expr COMMA expr RBRACE { PointLit({x=$2; y=$4; z=$6}) }
  | ID { PointId($1) }
```

```
matrix:
    LBRACKET matrix_entries RBRACKET { MatrixLit(List.rev $2) }
  | matrix_chain { MatrixChain(List.rev $1) }

matrix_chain:
    /* nothing */  { [] }
  | ID { [$1] }
  | matrix_chain TIMES ID { $3 :: $1 }

point_list:
    /* nothing */  { [] }
  | point { [$1] }
  | point_list COMMA point { $3 :: $1 }

matrix_entries:
    /* nothing */  { [] }
  | expr { [$1] }
  | matrix_entries COMMA expr { $3 :: $1 }

expr:
    expr PLUS expr { Binop($1, Add, $3) }
  | expr MINUS expr { Binop($1, Sub, $3) }
  | expr TIMES expr { Binop($1, Mul, $3) }
  | expr DIVIDE expr { Binop($1, Div, $3) }
  | expr POWER expr { Binop($1, Pow, $3) }
  | ID { Id($1) }
  | LITERAL { Lit($1) }
  | INT_LITERAL { Lit($1) }
  | LPAREN expr RPAREN { $2 }
  | ID expr { FuncApplication($1, $2) }
```

pts.ml

```
open Ast
open Printf

module StringMap = Map.Make(String)

type id_type = IdConst | IdPoint | IdPoints | IdMatrix | IdFunc
```

```
let add_id_type map def = match def with
    ConstDef(id, expr) -> StringMap.add id IdConst map
  | PointDef(id, point) -> StringMap.add id IdPoint map
  | PointsDef(id, points) -> StringMap.add id IdPoints map
  | MatrixDef(id, m) -> StringMap.add id IdMatrix map
  | FuncDef(id, arg, expr) -> StringMap.add id IdFunc map

let rec build_id_type_map stmts map = match stmts with
    [] -> map
  | hd :: tl ->
      (match hd with
          Def(def) ->
            let id_map = build_id_type_map tl map in
            add_id_type id_map def
        | _ -> map)

let rec translate_expr = function
    Lit(x) -> x
  | Binop(e1, op, e2) ->
      let v1 = translate_expr e1 and v2 = translate_expr e2 in
      (match op with
          Add -> "(" ^ v1 ^ " + " ^ v2 ^ ")"
        | Sub -> "(" ^ v1 ^ " - " ^ v2 ^ ")"
        | Mul -> "(" ^ v1 ^ " * " ^ v2 ^ ")"
        | Div -> "(" ^ v1 ^ " / " ^ v2 ^ ")"
        | Pow -> "Math.pow(" ^ v1 ^ ", " ^ v2 ^ ")")
   | Id(x) -> x
   | FuncApplication(id, arg) -> id ^ "(" ^ (translate_expr arg) ^ ")"

let translate_point = function
    PointLit(p) -> "new Point(" ^ (translate_expr p.x) ^
        ", " ^ (translate_expr p.y) ^ ", " ^ (translate_expr p.z) ^ ")"
  | PointId(pid) -> pid

let rec add_points list_id points = match points with
    [] -> ""
  | hd :: tl -> ";\n " ^ list_id ^ ".add(" ^ (translate_point hd) ^ ")" ^ (add_points list_id tl)

let translate_function_literal expr = "Functions.literal(" ^ (translate_expr expr) ^ ")"
```

```
let translate_matrix_expr expr id_map = match expr with
    Id(name) -> let id_type = StringMap.find name id_map in
       (match id_type with
           IdFunc -> name
         | _ ->  translate_function_literal expr)
  | _ -> translate_function_literal expr

let rec translate_matrix mlist id_map = match mlist with
    [] -> ""
  | [x] -> (translate_matrix_expr x id_map)
  | hd :: tl -> (translate_matrix_expr hd id_map) ^ "," ^ (translate_matrix tl id_map)

let rec translate_matrix_chain = function
    [] -> ""
  | [x] -> x
  | hd :: tl -> "Matrix.multiply(" ^ hd ^ ", " ^ (translate_matrix_chain tl) ^ ")"

let translate_def def id_map = match def with
    ConstDef(id, expr) -> "final double " ^ id ^ " = " ^ (translate_expr expr)
  | PointDef(id, point) -> "final Point " ^ id ^ " = " ^ (translate_point point)
  | PointsDef(id, points) -> "final LinkedList<Point> " ^ id ^ " = new LinkedList<Point>()" ^
      (add_points id points)
  | MatrixDef(id, m) ->
      let var_name = "final Matrix " ^ id ^ " = " in
      (match m with
          MatrixLit(values) ->  var_name ^ "new Matrix(" ^ (translate_matrix values id_map) ^
")"
        | MatrixChain(ids) -> var_name ^ (translate_matrix_chain ids))
  | FuncDef(id, arg, expr) ->
      "final Function " ^ id ^ " = new Function() {\n" ^
      "  @Override public double call(double " ^ arg ^") {\n" ^
      "    return " ^ (translate_expr expr) ^ ";\n" ^
      "  }\n" ^
      "}"

let rec translate_prog stmt_list id_map = match stmt_list with
    [] -> ""
  | [x] -> (translate_stmt x id_map) ^ ";\n"
```

```
  | hd :: tl -> (translate_stmt hd id_map) ^ ";\n" ^ (translate_prog tl id_map)
and translate_stmt stmt id_map = match stmt with
    Def(def) -> translate_def def id_map
  | Sleep(time) -> "Thread.sleep(" ^ (translate_expr time) ^ ")"
  | Render(m, p, t) -> "render(" ^ m ^ ", " ^ p ^ ", " ^ (translate_expr t) ^ ")"
  | AddPoint(pts, p) -> pts ^ ".add(" ^ (translate_point p) ^ ")"
  | ForLoop(i, low, high, body) ->
      "for (int " ^ i ^ " = " ^ low ^ "; " ^ i ^ " <= " ^ high ^ "; ++" ^ i ^ ") {\n" ^
      (translate_prog body id_map) ^ "}"



let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  let id_map = build_id_type_map program StringMap.empty in
  let file = open_out "pts.output" in
  fprintf file "%s\n" (translate_prog program id_map);
  close_out file;
```

Java Framework

```
import java.util.LinkedList;
import java.util.List;
import java.util.ArrayList;
import java.awt.Graphics2D;

public class Pts {

  interface Function {
    double call(double arg);
  }

  static class Functions {
    static Function literal(final double val) {
      return new Function() {
        @Override
        public double call(double ignore) {
          return val;
```

```
      }
    };
  }

  static Function multiply(final Function lhs, final Function rhs) {
    return new Function() {
      @Override
      public double call(double x) {
        return lhs.call(x) * rhs.call(x);
      }
    };
  }

  static Function add(final Function lhs, final Function rhs) {
    return new Function() {
      @Override
      public double call(double x) {
        return lhs.call(x) + rhs.call(x);
      }
    };
  }
}

static <T> List<T> toList(T[] array) {
  List<T> list = new LinkedList<T>();
  for (T val : array) {
    list.add(val);
  }
  return list;
}

static class Point {
  private final double values[];

  Point(double x, double y, double z) {
    this.values = new double[] {x, y, z, 1.0};
  }

  double get(int i) {
```

```
    return values[i];
  }

  void set(double val, int i) {
    values[i] = val;
  }

  @Override
  public String toString() {
    return "{ " + values[0] + ", " + values[1] + ", " + values[2] + "}";
  }
}

static class Matrix {
  private static final int N = 4; // 4x4 matrix

  private final ArrayList<Function> values;

  Matrix(Function... newValues) {
    this(toList(newValues));
  }

  Matrix() {
    this.values = new ArrayList<Function>(N * N);
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
        values.add(Functions.literal(i == j ? 1 : 0));
      }
    }
  }

  Matrix(List<Function> newValues) {
    if (newValues.size() == 16) {
      values = new ArrayList<Function>(newValues);
    } else if (newValues.size() == 9) {
      // Extend to 4x4 matrix

      // Row 1
      values = new ArrayList<Function>(newValues.subList(0, 3));
```

```
    values.add(Functions.literal(0));

    // Row 2
    values.addAll(newValues.subList(3, 6));
    values.add(Functions.literal(0));

    // Row 3
    values.addAll(newValues.subList(6, 9));
    values.add(Functions.literal(0));

    // Row 4
    values.add(Functions.literal(0));
    values.add(Functions.literal(0));
    values.add(Functions.literal(0));
    values.add(Functions.literal(1));
  } else {
    throw new AssertionError("Unsupported array size: " + newValues.size());
  }
}

Function get(int col, int row) {
  return values.get(4 * row + col);
}

ArrayList<Function> getRow(int row) {
  return new ArrayList<Function>(values.subList(4 * row, 4 * row + N));
}

ArrayList<Function> getCol(int col) {
  ArrayList<Function> result = new ArrayList<Function>();
  result.add(get(col, 0));
  result.add(get(col, 1));
  result.add(get(col, 2));
  result.add(get(col, 3));
  return result;
}

void set(int col, int row, Function value) {
  values.set(4 * row + col, value);
```

```
}

static Matrix multiply(Matrix lhs, Matrix rhs) {
  Matrix result = new Matrix();
  for (int col = 0; col < N; col++) {
    for (int row = 0; row < N; row++) {
      ArrayList<Function> lValues = lhs.getRow(row);
      ArrayList<Function> rValues = rhs.getCol(col);
      Function total = Functions.literal(0);
      for (int i = 0; i < N; i++) {
        total = Functions.add(total,
            Functions.multiply(lValues.get(i), rValues.get(i)));
      }
      result.set(col, row, total);
    }
  }
  return result;
}

static Point multiply(Matrix m, Point point, double t) {
  Point result = new Point(0, 0, 0);
  for (int row = 0; row < N; row++) {
    ArrayList<Function> values = m.getRow(row);
    double total = 0.0;
    for (int i = 0; i < N; i++) {
      total += values.get(i).call(t) * point.get(i);
    }
    result.set(total, row);
  }
  return result;
}

@Override
public String toString() {
  StringBuilder b = new StringBuilder("[\n");
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      b.append(get(j, i).call(1)).append(",  ");
    }
```

```
      b.append("\n");
    }
    b.append("]");
    return b.toString();
  }
 }

 static void render(Matrix transform, List<Point> points, double t) {
  System.out.println("Rendering t=" + t);
  for (Point p : points) {
   System.out.println(Matrix.multiply(transform, p, t));
  }
 }

 public static void main(String[] args) {
  try {
   run();
  } catch (InterruptedException e) {
   System.out.println("Interrupted");
  }
 }

 private static void run() throws InterruptedException {
```

pts.sh

```
#!/bin/bash

./compiler/pts < $1 # produce pts.output with java code to insert

cat PtsTop.java > Pts.java
cat pts.output >> Pts.java
echo "}}" >> Pts.java

javac Pts.java
```

build_pts.sh

```
#!/bin/bash

function compile() {
  echo "COMPILING: $1"
  $1
  if [ $? -ne 0 ]; then
    echo "FAILED"
    exit 1
  fi
  echo "DONE"
}

compile "ocamllex scanner.mll" # create scanner.ml

compile "ocamlyacc parser.mly" # create parser.ml and parser.mli

compile "ocamlc -c ast.mli"

compile "ocamlc -c parser.mli"

compile "ocamlc -c scanner.ml"

compile "ocamlc -c parser.ml"

compile "ocamlc -c pts.ml"

compile "ocamlc -o pts parser.cmo scanner.cmo pts.cmo"
```