Developer documentation for the Shade expression language

March 10, 2012

1 Overview

Shade is an expression language embedded in Javascript. It encodes a subset of the OpenGL ES Shading Language, GLSL ES (From now on, I will say "GLSL" when I mean "GLSL ES"). The fundamental difference between full GLSL and Shade is that Shade expressions denote *values*, and are (for the vast majority) side-effect free.

Because of this, Shade has very simple semantics which allow for programmatic manipulation of Shade expressions. This enables, for example, powerful constant folding and expression manipulation, which are necessary components of a practical high-level language that aims to be efficiently compiled.

Shade does not compile GLSL itself. Instead, every Shade expression denotes some value in GLSL. The Shade compiler works by transforming Shade expressions into GLSL programs, but it is free to choose different GLSL statements, as long as they all denote the same value.

For example, consider the following snippet:

```
var a = Shade.uniform("float");
var b = Shade.vec(a, a.sqrt(), 1.0);
var c = Shade.sin(b).z();
```

The value denoted by `a` is a scalar whose value cannot be determined at compile time (it's a GLSL uniform [?]). `b` denotes a 3-dimensional vector, of which some components depend on the values of `a`. Finally, `c` denotes the z-component of the sine of `b`. The value of `c` *can* be computed at compile-time, but only if we know the semantics of vector constructors, sine functions, etc. The Shade compiler knows about these, and so can determine that `c` does not depend on `a` at all. If no other parts of the program depend on `a`, Shade will completely remove the uniform declaration from the final GLSL shader. This allows users to write high-level libraries in Shade, without paying many of the costs of abstraction.

2 Shade Objects

Every expression in Shade is a javascript object of type `Exp`, defined in `src/shade/exp.js`. The vast majority of these denote values, and are of type `ValueExp`, defined in `src/shade/value_exp.js`. Every `ValueExp` has a *type*, corresponding to one of the GLSL types, and can be compiled to a GLSL expression that when evaluated, denotes the appropriate value (the semantics will be given below).

3 Shade Types

4 Shade Program

A Shade program encodes a GLSL vertex shader and a GLSL fragment shader.

The Shade compiler will compute the necessary interfaces automatically. This reduces much of the hassle of writing maintainable, reusable shader libraries. For example, names of GLSL `varying`, `uniform` and `attribute` variables must all be consistent, and determined ahead of time. In addition, there exist restrictions of which variables can be accessed on which shaders. The Shade compiler will automatically create shader pairs which respect these restrictions.

A Shade program is created by a call to `Shade.program`, in `src/shade/program.js`. `Shade.program` takes as a parameter a Javascript object with at least two keys: `color` and `position`. The values associated to the keys denote, respectively, Shade expressions for the color of the final fragment, and the position of the vertex in homogeneous coordinates. Other key-value pairs are allowed, and are ignored by `Shade.program`.

5 GLSL Values

We use the following GLSL types and expressions:

- `float`: atomic floating-point value
- `int`: atomic integer value
- `bool`: atomic boolean value
- `vec2`, `vec3`, `vec4`: fixed-size floating point vectors
- `ivec2`, `ivec3`, `ivec4`: fixed-size integer vectors
- `bvec2`, `bvec3`, `bvec4`: fixed-size integer vectors
- `mat2`, `mat3`, `mat4`: fixed-size floating point square matrices
- `sampler2D`, opaque texture reference type

Values of type `vec[234]` and `mat[234]` are denoted as `vec3(a, b, c)`, `mat2(vec2(a, b), vec2(c, d))`, and so on. If `v` is of type `vec[234]`, then `v[i]` is of type float, and denotes the i-th component of the vector. If `m` is of type `mat[234]`, then `m[i]` is of type `vec[234]`, and denotes the i-th column of the matrix

6 Shade Grammar

Because Shade is an embedded language in Javascript, its grammar has an almost one-to-one correspondence with functions in the `Shade` object. Most of these have a direct interpretation in terms of GLSL functions, methods or expressions.

You will note that there is some inconsistency in the case conventions of the functions below. Generally speaking, GLSL conforms to camelCase conventions. Shade functions and methods use underscore_conventions, while Shade object prototypes and classes use CamelCase.

In the list of value constructors below, we use `f(t1, t2, t3)` to mean that there exists a Shade constructor of the object `f` which takes three parameters of types `t1`, `t2` and `t3` respectively. To give a particular semantics for Shade, all we have to do is provide functions for each of these types.

6.1 Shade value constructors

- abs
  - `abs(float)`
  - `abs(vecX)`, `X` $\in [2, 3, 4]$
- acos
  - `acos(float)`
  - `acos(vecX)`, `X` $\in [2, 3, 4]$
- add
  - `add(float, float)`
  - `add(vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `add(matX, matX)`, `X` $\in [2, 3, 4]$
  - `add(vecX, float)`, `X` $\in [2, 3, 4]$
  - `add(matX, float)`, `X` $\in [2, 3, 4]$
  - `add(float, vecX)`, `X` $\in [2, 3, 4]$
  - `add(float, matX)`, `X` $\in [2, 3, 4]$
- all
  - `all(bool)`

- `all(bvecX)`, $X \in [2,3,4]$

- and
  - `and(bool, bool)`
- any
  - `any(bool)`
  - `any(bvecX)`, $X \in [2,3,4]$
- array
  - `array([list of float])`
  - `array([list of bool])`
  - `array([list of vecX])`, $X \in [2,3,4]$
  - `array([list of bvecX])`, $X \in [2,3,4]$
  - `array([list of ivecX])`, $X \in [2,3,4]$
  - `array([list of matX])`, $X \in [2,3,4]$
- asin
  - `asin(float)`
  - `asin(vecX)`, $X \in [2,3,4]$
- at
  - `at(vecX, int)`, $X \in [2,3,4]$
  - `at(matX, int)`, $X \in [2,3,4]$
- atan
  - `atan(float, float)`
  - `atan(vecX, vecX)`, $X \in [2,3,4]$
- attribute
  - FIXME
- ceil
  - `ceil(float)`
  - `ceil(vecX)`, $X \in [2,3,4]$
- clamp
  - `clamp(float, float, float)`
  - `clamp(vecX, vecX, float)`, $X \in [2,3,4]$

  - `clamp(matX, matX, float)`, `X` $\in [2, 3, 4]$

- cos
  - `cos(float)`
  - `cos(vecX)`, `X` $\in [2, 3, 4]$
- cross
  - `cross(vec3, vec3)`
- degrees
  - `degrees(float)`
- discard_if
  - `discard_if(ANY, bool)`
- distance
  - `distance(float, float)`
  - `distance(vecX, vecX)`, `X` $\in [2, 3, 4]$
- div
  - `div(float, float)`
  - `div(vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `div(matX, matX)`, `X` $\in [2, 3, 4]$
  - `div(vecX, float)`, `X` $\in [2, 3, 4]$
  - `div(matX, float)`, `X` $\in [2, 3, 4]$
  - `div(float, vecX)`, `X` $\in [2, 3, 4]$
  - `div(float, matX)`, `X` $\in [2, 3, 4]$
- dot
  - `dot(float, float)`
  - `dot(vecX, vecX)`, `X` $\in [2, 3, 4]$
- eq
  - `eq(float, float)`
  - `eq(int, int)`
  - `eq(bool, bool)`
  - `eq(vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `eq(bvecX, bvecX)`, `X` $\in [2, 3, 4]$
  - `eq(ivecX, ivecX)`, `X` $\in [2, 3, 4]$

  - `eq(matX, matX)`, $\texttt{X} \in [2, 3, 4]$

- equal
  - `equal(vecX, vecX)`
  - `equal(bvecX, bvecX)`
  - `equal(ivecX, ivecX)`
- exp
  - `exp(float, float)`
  - `exp(vecX, vecX)`, $\texttt{X} \in [2, 3, 4]$
- exp2
  - `exp2(float, float)`
  - `exp2(vecX, vecX)`, $\texttt{X} \in [2, 3, 4]$
- faceforward
  - `faceforward(float, float, float)`
  - `faceforward(vecX, vecX, vecX)`, $\texttt{X} \in [2, 3, 4]$
- floor
  - `floor(float, float)`
  - `floor(vecX, vecX)`, $\texttt{X} \in [2, 3, 4]$
- fract
  - `fract(float, float)`
  - `fract(vecX, vecX)`, $\texttt{X} \in [2, 3, 4]$
- fragCoord
  - `fragCoord`
- ge
  - `ge(float, float)`
  - `ge(int, int)`
  - `ge(bool, bool)`
  - `ge(vecX, vecX)`, $\texttt{X} \in [2, 3, 4]$
  - `ge(bvecX, bvecX)`, $\texttt{X} \in [2, 3, 4]$
  - `ge(ivecX, ivecX)`, $\texttt{X} \in [2, 3, 4]$
  - `ge(matX, matX)`, $\texttt{X} \in [2, 3, 4]$
- greaterThan

  - `greaterThan(vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `greaterThan(ivecX, ivecX)`, `X` $\in [2, 3, 4]$

- greaterThanEqual
  - `greaterThanEqual(vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `greaterThanEqual(ivecX, ivecX)`, `X` $\in [2, 3, 4]$

- gt
  - `gt(float, float)`
  - `gt(int, int)`
  - `gt(bool, bool)`
  - `gt(vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `gt(bvecX, bvecX)`, `X` $\in [2, 3, 4]$
  - `gt(ivecX, ivecX)`, `X` $\in [2, 3, 4]$
  - `gt(matX, matX)`, `X` $\in [2, 3, 4]$

- inversesqrt
  - `inversesqrt(float, float)`
  - `inversesqrt(vecX, vecX)`, `X` $\in [2, 3, 4]$

- le
  - `le(float, float)`
  - `le(int, int)`
  - `le(bool, bool)`
  - `le(vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `le(bvecX, bvecX)`, `X` $\in [2, 3, 4]$
  - `le(ivecX, ivecX)`, `X` $\in [2, 3, 4]$
  - `le(matX, matX)`, `X` $\in [2, 3, 4]$

- length
  - `length(float)`
  - `length(vecX)`, `X` $\in [2, 3, 4]$

- lessThan
  - `lessThan(vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `lessThan(ivecX, ivecX)`, `X` $\in [2, 3, 4]$

- lessThanEqual

  - `lessThanEqual(vecX, vecX)`, `X` $\in [2,3,4]$
  - `lessThanEqual(ivecX, ivecX)`, `X` $\in [2,3,4]$

- log
  - `log(float)`
  - `log(vecX)`, `X` $\in [2,3,4]$
- log2
  - `log2(float)`
  - `log2(vecX)`, `X` $\in [2,3,4]$
- look_at
  - `look_at(vec3, vec3, vec3)`
- lt
  - `lt(float, float)`
  - `lt(int, int)`
  - `lt(bool, bool)`
  - `lt(vecX, vecX)`, `X` $\in [2,3,4]$
  - `lt(bvecX, bvecX)`, `X` $\in [2,3,4]$
  - `lt(ivecX, ivecX)`, `X` $\in [2,3,4]$
  - `lt(matX, matX)`, `X` $\in [2,3,4]$
- mat
  - `mat(float, float, float, float)`
  - `mat(float, ..., float)` (9 parameters for a mat3)
  - `mat(float, ..., float)` (16 parameters for a mat4)
  - `mat(vec2, vec2)`
  - `mat(vec3, vec3, vec3)`
  - `mat(vec4, vec4, vec4, vec4)`
- mat3
  - `mat3(vec3, vec3, vec3)` FIXME WHY DO WE HAVE THIS?
- matrixCompMult
  - `matrixCompMult(matX, matX)`, `X` $\in [2,3,4]$
- max
  - `max(int, int)`

  - `max(float, float)`
  - `max(vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `max(vecX, float)`, `X` $\in [2, 3, 4]$

- min
  - `min(int, int)`
  - `min(float, float)`
  - `min(vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `min(vecX, float)`, `X` $\in [2, 3, 4]$

- mix
  - `mix(float, float, float)`
  - `mix(vecX, vecX, float)`, `X` $\in [2, 3, 4]$
  - `mix(vecX, vecX, vecX)`, `X` $\in [2, 3, 4]$

- mod
  - `mod(int, int)`
  - `mod(float, float)`
  - `mod(vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `mod(vecX, float)`, `X` $\in [2, 3, 4]$

- mul
  - `mul(float, float)`
  - `mul(int, int)`
  - `mul(vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `mul(matX, matX)`, `X` $\in [2, 3, 4]$
  - `mul(matX, vecX)`, `X` $\in [2, 3, 4]$
  - `mul(vecX, matX)`, `X` $\in [2, 3, 4]$

- ne
  - `ne(float, float)`
  - `ne(int, int)`
  - `ne(bool, bool)`
  - `ne(vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `ne(bvecX, bvecX)`, `X` $\in [2, 3, 4]$
  - `ne(ivecX, ivecX)`, `X` $\in [2, 3, 4]$
  - `ne(matX, matX)`, `X` $\in [2, 3, 4]$

- neg
  - `neg(float)`
  - `neg(vec2)`
  - `neg(vec3)`
  - `neg(vec4)`
  - `neg(int)`
  - `neg(ivec2)`
  - `neg(ivec3)`
  - `neg(ivec4)`
- normalize
  - `normalize(float)`
  - `normalize(vecX)`, `X` $\in [2, 3, 4]$
- not
  - `not(bool)`
- notEqual
  - `notEqual(vecX, vecX)`
  - `notEqual(ivecX, ivecX)`
  - `notEqual(bvecX, bvecX)`
- or
  - `or(bool, bool)`
- per_vertex
  - `per_vertex(ANY)`
- pointCoord
  - `pointCoord`
- pow
  - `pow(float, float)`
  - `pow(vecX, vecX)`, `X` $\in [2, 3, 4]$
- radians
  - `radians(float)`
- reflect

  - `reflect(float, float, float)`
  - `reflect(vecX, vecX, vecX)`, `X` $\in [2, 3, 4]$

- refract
  - `refract(float, float, float)`
  - `refract(vecX, vecX, vecX)`, `X` $\in [2, 3, 4]$

- selection
  - `selection(bool, ANY, ANY)`

- sign
  - `sign(float)`
  - `sign(vecX)`, `X` $\in [2, 3, 4]$

- sin
  - `sin(float)`
  - `sin(vecX)`, `X` $\in [2, 3, 4]$

- smoothstep
  - `smoothstep(float, float, float)`
  - `smoothstep(vecX, vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `smoothstep(float, float, vecX)`, `X` $\in [2, 3, 4]$

- sqrt
  - `sqrt(float)`
  - `sqrt(vecX)`, `X` $\in [2, 3, 4]$

- step
  - `step(float,float)`
  - `step(vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `step(float,vecX)`, `X` $\in [2, 3, 4]$

- sub
  - `sub(float, float)`
  - `sub(vecX, vecX)`, `X` $\in [2, 3, 4]$
  - `sub(matX, matX)`, `X` $\in [2, 3, 4]$
  - `sub(vecX, float)`, `X` $\in [2, 3, 4]$
  - `sub(matX, float)`, `X` $\in [2, 3, 4]$
  - `sub(float, vecX)`, `X` $\in [2, 3, 4]$

  - `sub(float, matX)`, `X` $\in [2, 3, 4]$

- swizzle
  - `swizzle(string, vec2)`
  - `swizzle(string, vec3)`
  - `swizzle(string, vec4)`
- tan
  - `tan(float)`
  - `tan(vecX)`, `X` $\in [2, 3, 4]$
- texture2D
  - `texture2D(sampler2D, vec4)`
- uniform

  This one is ugly: in the code, it takes as a parameter the string corresponding to the appropriate type. Here we will use different uniform constructors to denote uniforms of different types.
  - `uniform_float()`
  - `uniform_vecX()`, `X` $\in [2, 3, 4]$
  - `uniform_matX()`, `X` $\in [2, 3, 4]$
- vec
  - `vec(float)`
  - `vec(vec2)`
  - `vec(vec3)`
  - `vec(vec4)`
  - `vec(float, float)`
  - `vec(float, float, float)`
  - `vec(float, float, float, float)`
  - `vec(vec2, float, float)`
  - `vec(float, vec2, float)`
  - `vec(float, float, vec2)`
  - `vec(vec2, vec2)`
  - `vec(vec3, float)`
  - `vec(float, vec3)`
- xor
  - `xor(bool, bool)`

6.2 Internal use only

7 Shade Semantics

Most of Shade has very simple denotational semantics. Simply put, we will model the semantics of Shade as functions from the set of (well-formed) Shade expressions, S, to sets of values which we care about. (In case the phrase "denotational semantics" scares you: GLSL does not allow recursive functions, so all the scary math of lambda-calculus denotational semantics is not needed). More concretely, we will define two different semantic functions: *constant semantics* and *value semantics*. The set corresponding to the range of the semantics function will be known as the *target* of that semantics. Our target for the constant semantics will be values in boolean algebra. For the value semantics, it will be values in an algebra of scalars, vectors and matrices. The semantic functions will make extensive use of an auxiliary function Element$[x, i,]$, which extracts the ith element from the compound value x.

Semantic function application will be denoted with square brackets to avoid confusion with the constructor notation of Shade values, which uses regular parentheses.

We give here denotational semantics for the value-expression portion of the Shade language. This is a *purely-functional* subset, and expressions of this subset are referentially transparent: they denote the same value regardless of where they appear in the full program. This lets us write simple optimizers and analyzers for this portion of the Shade language. When embedded in a high-level language such as Javascript, this purely functional subsect is already sufficient enough to create a rich EDSL.

Even within this restricted subset of GLSL, the actual value denoted by some Shade expressions cannot be determined entirely statically: they will depend on the WebGL state machine (for example, the values of uniform parameters and the results of texture fetches). In that case, we will use a special value `Unknown` to denote the result. We assume that the target algebras have been enriched such that, *unless otherwise specified*, any operation involving `Unknown` returns `Unknown`. In that way, `Unknown` values are propagated through the semantics. For cases whose values are only left undefined in the GLSL spec, we will also use the `Unknown` value.

Values in the Shade language are typeset `in monospace`. Values in the target set are typeset normally. Some functions on the target set are left implicitly defined (such as sines, cosines, etc). When there's overlap in the structural matching rules described below, the rule described first in the list takes precedence.

7.1 Constant semantics

Const[`expression`] $: S \to \{\text{true}, \text{false}\}$

The constant semantics function is conservative: it is possible to write more specific rules which would let us determine with a finer granularity values that are computable at compile-time. As a very simple example, the following rule would be valid:

Const[`sub(floor(x), sub(x, fract(x)))`] = true

However, with the specification below, the constant semantics would evaluate `sub(floor(x), sub(x, fract(x)))` to Const[x], which could be false. This is unavoidable: there is no computable function PerfectConst that perfectly determines whether every well-formed expression is constant. (big hammer: Rice's theorem; small hammer, consider trying to compute PerfectConst[(quintic_polynomial over a uniform)^2 = 0])

- Basic values
  - Const[`float(x)`] = true
  - Const[`bool(x)`] = true
- abs
  - Const[`abs(x)`] = Const[x]
- acos
  - Const[`acos(x)`] = Const[x]
- add
  - Const[`add(x, y)`] = Const[x] $\wedge$ Const[y]
- all
  - Const[`all(bool(x))`] = Const[`bool(x)`]
  - Const[`all(bvec2(x, y))`] = Const[`and(x, y)`]
  - Const[`all(bvec3(x, y, z))`] = Const[`and(x, and(y, z))`]
  - Const[`all(bvec4(x, y, z, w))`] = Const[`and(x, and(y, and(z, w)))`]
- and
  - Const[`and(x, y)`] = (Const[x] $\wedge \neg$Value[x]) $\vee$ (Const[y] $\wedge \neg$Value[y])
- any
  - Const[`any(bool(x))`] = Const[`bool(x)`]
  - Const[`any(bvec2(x, y))`] = Const[`or(x, y)`]
  - Const[`any(bvec3(x, y, z))`] = Const[`or(x, or(y, z))`]
  - Const[`any(bvec4(x, y, z, w))`] = Const[`or(x, or(y, or(z, w)))`]
- array
  - Const[`array(x)`] = false

- asin
  - Const[`asin(x)`] = Const[`x`]
- at
  - FIXME
- atan
  - Const[`atan(x)`] = Const[`x`]
  - Const[`atan(x, y)`] = Const[`x`] $\wedge$ Const[`y`]
- attribute
  - Const[`attribute_float`] = false
  - Const[`attribute_vecX`] = false, `X` $\in [2, 3, 4]$
  - Const[`attribute_matX`] = false, `X` $\in [2, 3, 4]$
- ceil
  - Const[`ceil(x)`] = Const(`x`)

7.2 FIXME FINISH

7.3 Value semantics

$$\text{Value}[\texttt{expression}] : S \to \text{GLSL\_value}$$

To show that `Val` uniquely defines a value for every finite expression, all we have to show is that there every equation for a term is is defined in terms of "simpler" terms. In other words, we can prove that the structural recursion that `Val` gives terminates by structural induction. This is tedious, but easy enough to do (although it's not in here yet).

`Val[]` behaves like you expect with respect to constants.

Some operators are extended to behave well with respect to unknown values, to allow short-circuit optimizations:

- unknown $\vee$ true = true
- unknown $\wedge$ false = false
- unknown $* 0 = 0$

The full list follows below.

- basic types
  - Value[`float(3)`] = 3

  - ...

- abs
  - Val[`abs(x)`] = case Val[`x`] of: - float(a) -¿ —a— - vec2(a, b) -¿ vec2(—a—, —b—) - vec3(a, b, c) -¿ vec3(—a—, —b—, —c—) - vec4(a, b, c, d) -¿ vec4(—a—, —b—, —c—, —d—)
- acos
  - Val[`acos(x)`] = case Val[`x`] of: - float(a) -¿ acos(a) - vec2(a, b) -¿ vec2(acos(a), acos(b)) - vec3(a, b, c) -¿ vec3(acos(a), acos(b), acos(c)) - vec4(a, b, c, d) -¿ vec4(acos(a), acos(b), acos(c), acos(d))
- add
  - Val[`add(x, y)`] = case Val[`x`], Val[`y`] of: - float(a), float(b) -¿ a + b - vec2(a, b), float(c) -¿ vec2(a+c, b+c) - vec3(a, b, c), float(d) -¿ vec2(a+d, b+d, c+d) - vec4(a, b, c, d), float(e) -¿ vec4(a+e, b+e, c+e, d+e) - float(c), vec2(a, b) -¿ vec2(a+c, b+c) - float(d), vec3(a, b, c) -¿ vec2(a+d, b+d, c+d) - float(e), vec4(a, b, c, d) -¿ vec4(a+e, b+e, c+e, d+e) - vec2(a, b), vec2(c, d) -¿ vec2(a+c, b+d) - ... FIXME FINISH
- all
  - Val[`all(x)`] = case Val[`x`] of: - bool(a) -¿ x - bvec2(a, b) -¿ $a \wedge b$ - bvec2(a, b, c) -¿ $a \wedge b \wedge c$ - bvec2(a, b, c, d) -¿ $a \wedge b \wedge c \wedge d$
- and
  - Value[`and(x, y)`] = Value[`x`] $\wedge$ Value[`y`]
- any
  - Val[`any(x)`] = case Val[`x`] of: - bool(a) -¿ x - bvec2(a, b) -¿ $a \vee b$ - bvec2(a, b, c) -¿ $a \vee b \vee c$ - bvec2(a, b, c, d) -¿ $a \vee b \vee c \vee d$
- array
  - Val[`array(ANY)`] = `Unknown` More precisely, `array` does not have a GLSL value associated with it. However, it can be combined with `at()` to produce a GLSL value.
- asin
  - Val[`asin(x)`] = case Val[`x`] of: - float(a) -¿ asin(a) - vec2(a, b) -¿ vec2(asin(a), asin(b)) - vec3(a, b, c) -¿ vec3(asin(a), asin(b), asin(c)) - vec4(a, b, c, d) -¿ vec4(asin(a), asin(b), asin(c), asin(d))
- at

  - Val[`at(array(`$[a_0, a_1, \ldots\ldots, a_n)$`, index)`] = (Val[$a_i$], where i = Val[`index`])

- atan
  - atan(x) = arc tangent of x, in radians
  - atan2(y, x) = arc tangent of y/x, in radians, using signs of y and x to determine quadrant.
  - Val[`atan(x)`] = case Val[`x`] of: - float(a) -¿ atan(a) - vec2(a, b) -¿ vec2(atan(a), atan(b)) - vec3(a, b, c) -¿ vec3(atan(a), atan(b), atan(c)) - vec4(a, b, c, d) -¿ vec4(atan(a), atan(b), atan(c), atan(d))
  - Val[`atan(x, y)`] = case Val[`x`], Val[`y`] of: - float(a), float(b) -¿ atan2(a, b) - vec2(a, b), vec2(c, d) -¿ vec2(atan2(a,c), atan2(b,d)) - vec3(a, b, c), vec3(d, e, f) -¿ vec3(atan2(a, d), atan2(b, e), atan2(c, f)) - vec4(a, b, c, d), vec4(e, f, g, h) -¿ vec4(atan2(a, e), atan2(b, f), atan2(c, g), atan2(d, h))

- attribute
  - FIXME

- ceil
  - ceil(x) = value of x rounded towards +infinity
  - Val[`ceil(x)`] = case Val[`x`] of: - float(a) -¿ ceil(a) - vec2(a, b) -¿ vec2(ceil(a), ceil(b)) - vec3(a, b, c) -¿ vec3(ceil(a), ceil(b), ceil(c)) - vec4(a, b, c, d) -¿ vec4(ceil(a), ceil(b), ceil(c), ceil(d))

- clamp
  - max(a, b) = if a ¿ b then a else b
  - min(a, b) = if a ¡ b then a else b
  - Val[`clamp(v, mn, mx)` = case Val[`v`], Val[`mn`], Val[`mx`] of: - float(v), float(mn), float(mx) -¿ max(mn, min(mx, v)) - vec2(v1, v2), vec2(mn1, mn2), vec2(mx1, mx2) -¿ vec2(max(mn1, min(mx1, v1)), max(mn2, min(mx2, v2))) - vec3(v1, v2, v3), vec2(mn1, mn2, mn3), vec2(mx1, mx2, mx3) -¿ vec2(max(mn1, min(mx1, v1)), max(mn2, min(mx2, v2)), max(mn3, min(mx3, v3))) - FIXME FINISH

- cos
  - cos(x) = cosine of x given in radians
  - Val[`cos(x)`] = case Val[`x`] of: - float(a) -¿ cos(a) - vec2(a, b) -¿ vec2(cos(a), cos(b)) - vec3(a, b, c) -¿ vec3(cos(a), cos(b), cos(c)) - vec4(a, b, c, d) -¿ vec4(cos(a), cos(b), cos(c), cos(d))

- cross

    - Val[`cross(v1, v2)`] = Val[`vec3(sub(mul(at(v1, 1), at(v2, 2)), mul(at(v1, 2), at(v2, 1))), sub(mul(at(v1, 2), at(v2, 0)), mul(at(v1, 0), at(v2, 2))), sub(mul(at(v1, 0), at(v2, 1)), mul(at(v1, 1), at(v2, 0))))`]

- degrees
    - Val[`degrees(x)`] = Val[`x`]
    - (180/pi)

- discard_if
    - Val[`discard_if(x, bool(`true`))`] = `Unknown`
    - Val[`discard_if(x, bool(`false`))`] = x

      The semantics of `discard_if` involve a limited side-effect in the pipeline. If the condition is true, the evaluation of the fragment program is terminated and that fragment is not processed any further. That is the full extent of the side-effect created by `discard_if`: most importantly, it does not create or change any mutable state. When used judiciously, `discard_if` is a powerful way to avoid unnecessary generation of large numbers of polygons.

7.4 FIXME FINISH

7.5 Relationship between semantic functions

The relationship between the value semantics and the constant semantics is the following:

$$(\text{Value}[s] = \text{unknown}) \Rightarrow \text{Const}[s] = \text{false}$$