The Fnord Manual

Scott Draves

March 25, 1991

1 Introduction

This document serves as a description of fnord as seen by a user. It's primary goal is to be comprehensive rather than tutorial. If you are trying to learn the fnord language, your best hope is probably to read Section 2, then study some real code, and to read the rest only after you understand the basics well.

This document does not describe the internals of the interpreter or visualization system. We present very little motivation of or exposition on the ideas behind fnord. This is not a scholarly work; it has no references.

We start out with an overview that includes highlights from the syntax, and some examples. The syntax is described in Section 3. Then we cover values and types. Section 5 is the largest section; it lists all the operators and what they do. Next, the operation of the interpreter is described. Section 7 describes the visualization system. Finally the standard library is explained in Section 8.

1.1 What Fnord is Not

Fnord is not a traditional symbolic algebra program like MACSYMA or Mathematica. Fnord does not factor polynomials, solve equations, or integrate expressions. Fnord is not meant to be an algebraic assistant.

1.2 What Fnord Is

Fnord has two primary components. It contains a programming language (named fnorse?). It contains a user interface and visualization system. A user describes mathematical objects, usually curves, surfaces, or other geometry, in the fnorse language. An interpreter computes the objects from the descriptions, and passes them to the visualization system where they are displayed.

A typical fnorse program may create a parametric function and a domain, and then display the surface that results from applying the function to the domain.

Fnorse programs may contain constants whose values are generated by interactive widgets. When a widget is manipulated its value changes and the objects affected by that widget are recomputed and displayed.

1.3 Typographical Conventions

We use the `typewriter` typeface for all input and output from the interpreter. We use an arrow to show evaluation `1 + 1` $\Rightarrow$ `2`. `x + -y` $\equiv$ `x - y` means the two expressions will always have the same value.

2 Overview

Before going into detail, we state the major features of the fnorse language, and give a few

examples.

- Fnorse is purely functional. That means that no operator has any side effects; there is no assignment operator. A function does nothing but compute its return value.
- All user functions take one argument and return one value. This is usually hidden from the user because the lambda operator automatically deconstructs its argument and binds its components to separate symbols.
- Functions are first-class objects; they may be created, passed to other functions, and evaluated. This is also called support for "higher order functions".
- Fnorse is optionally typed, and a limited form of type inferencing is performed. This means that functions and objects may be declared either with or without a type. Types may contain type variables that match anything; they may be *type schemas*.

2.1 Syntax

A few notes about the syntax (parentheical comparisons are to Scheme):

- Parentheses are used only to indicate grouping. They do not indicate function calls.
- Functions are invoked by juxtaposition, e.g. `f x`. Since parentheses only indicate grouping, `f(x)` also is a function call; this is the usual syntax. If the left object is not a function, then it is multiplied instead; e.g. `2x`.
- Symbols are defined with `:=` (like define), functions created with `->` (like lambda). Definitions are terminated with `;`. Thus `f := x -> x + 1;` binds `f` to a function which adds one, after which `f(2)` ⇒ `3`.
- Comma creates pairs (infix cons). It is an infix operator like any other. `left` and `right` extract parts of pairs (like car and cdr). Pairs are usually parenthesized because comma has low precendence. Thus `(1, 2)` is a pair of integers.
- Vectors and matrices are created with square brackets. `[1, 2, 3]` is a vector of three integers; `[[1, 2],[3, 4]]` is a two by two matrix. Sets can be created with curly braces. `{1, 2, 3}` is a set containing three integers.
- Entries are extracted from vectors with hat, so `[a, b]^1` ⇒ `a`. The same goes for matrices, so if `a := [[1, 2], [3, 4]]`, then `a^1` ⇒ `[1, 2]` and `a^2^1` ⇒ `3`.

2.2 Simple Examples

Figure 1 generates and displays a sphere. `X` is a typed function from the plane to space. `U` is a rectangular subset of the plane; it is sampled in a regular grid, forty samples by twenty. The type of `U` is {`R^2`} ($U \subset R^2$, or $U \in 2^{R^2}$). When `X` is applied to `U`, the interpreter sees that the domain of `X` does not match the type of `U`. So it applies `X` to each point of `U` instead.

The functions `Patch` and `Show` (and even the symbol `R`) are part of the standard library, and are defined in terms of language primitives.

The code in Figure 3 shows how functions and sets interact. `F` is the curried multiply function. `F A` is the set of functions which multiply by 1, 2, and 3. Just as a function applied to a set can map over the elements of the set, a set of functions applied to an object applies each of the functions to the object. Thus `F A 7` makes sense and is `7, 14, 21`. Lastly, a set of functions applied to a set performs an

```
X := R^2    -> R^3
  :  [u, v] -> [sin(u) cos(v),
                cos(u) cos(v),
                sin(v)]
  ;
U := Patch((-pi, pi, 40),
           (-pi/2, pi/2, 20));
widget Show(X(U));
```

Figure 1: Create and display a sphere.

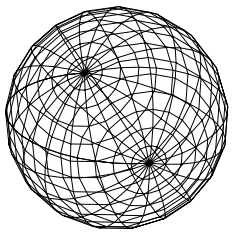


Figure 2: The Sphere

```
int := Z;
F := int -> int -> int
  :  x   -> y   -> x y
  ;
A := {1, 2, 3};
print F 7 A;
=> {Z} : {7, 14, 21}
print F A 7;
=> {Z} : {7, 14, 21}
print F A A;
=> {Z} : {1, 2, 3,
          2, 4, 6,
          3, 6, 9}
```

Figure 3: Functions and sets.

outer-product-like operation, resulting in all combinations of products.

Here we end our quick overview and begin the comprehensive description, starting with the syntax of fnorse.

3 Syntax

The syntax is based upon common mathematical notation. A fnorse program consists of a sequence of statements. There are four kinds of statements: declarations, widget declarations, commands, and interpreter directives. All statements are terminated by a semicolon.

- A declaration has the form *identifier* `:=` *expression* `;`.
- A widget declaration either has the form *identifier* `:= widget` *expression* `;`, or just `widget` *expression* `;`. See Section 7.
- A command has the form *command expression* `;` The commands and their meanings are described in Section 6.5
- An interpreter directive has the form *directive*`;`. The directives and their meanings are described in section 6.6.

3.1 Comments

Comments are C-style, but they may be nested. Comments break up other tokens; removing a comment is more akin to replacing it with whitespace than to removing exactly its characters. See Figure 4.

3.2 Literals

There are four sorts of literals recognized by the lexer: identifiers, integers, reals, and strings.

```
MakeCylinders  /* this is the name /* of */ the function */
   := R^3    -> R^2    -> R^3   /* this is the type */
   :  center -> [u, v] ->  center + [sin(u), cos(u), v]
   ;  /* we end the definition with a semi-colon */
```

Figure 4: Comments Example.

and	debug
else	eval
if	print
mod	widget
not	
or	
union	

Table 1: Reserved words.

reals	not reals
1.1	1
.1	1.
1e5	1 e5
1.1E-5	
.1e+5	

Table 2: Valid and invalid real numbers.

3.2.1 Identifiers

Fnorse identifiers are case sensitive. They consist of digits and letters, and must start with a letter. By convention, the first letter of each word of multi-word identifiers is in upper case. Thus `Append`, `GaussianCurvature`, and `a1` are identifiers while `my_func`, `my-func`, and `1a` are not.

Identifiers also may not be one of the reserved words (see Table 1), or one of the built-in functions (see Section 3.4).

3.2.2 Integers

An integer is a sequence of digits. All integers are interpreted in base ten. A sign may be placed in front of an integer, but it is not part of it. Such a sign is really an operator applied to the integer; thus all literal integers are non-negative.

The value of an integer is the integer.

3.2.3 Reals

Real numbers have a more complicated syntax; one best described by a few examples. See Table 2.

The value of a real is the real.

3.2.4 Strings

Strings are started and terminated by double quotes `"`. No escapes are supported; everything between the quotes is taken verbatim.

The value of a string is a vector of integers, where the integers are the ASCII codes of the characters in the string. Thus `"hi mom"` ⇒ `Z^6:  [104, 105, 32, 109, 111, 109]`. They are used primarily as parameters to widgets.

3.3 Expressions

Being a functional language, nearly all of the syntax (and semantics) is connected to expressions. All expression syntax is listed in Table 3 and Table 4. The operators are classified as

unary prefix, unary postfix, binary infix, and ternary infix, and atomic. Each operator has precedence and associativity. There are some special cases, which are described below. The table lists all the operators with syntax, their class, syntax, and associativity (for infix binary). The table is from highest to lowest precedence, horizontal lines seperate groups of equal precedence.

Notice that apply appears twice. The higher precedence apply, sometimes called p-apply, exists because without it `sin(x) cos(x)` would associate `((sin x) cos) x` rather than `(sin x)(cos x)`.

The relational operators are a special case. When they are chained together, `and` nodes are added between the pairs. Thus `(x < y < z)` $\equiv$ `(x < y and y < z)` and `(x = y > z)` $\equiv$ `(x = y and y > z)`.

The if-else operator is meant to be chained like this:

```
F := x -> -1 if  x < -1 else
          x if  x <  1 else
          1
  ;
```

3.4 Built-In Functions

Besides the operators with special syntax there are a number of built-in functions. These all have the same syntax: *name arguments*. For example, `sin x`, and `binomial(x, y)`. The built-in functions are listed in Table 5 and Table 6.

Caveat 1 *The built-in functions differ from user functions in that they do not all take one argument. Those that take two (or more) arguments must be supplied with exacly two (or more) arguments. For example,* `a := (3, 4); b := binomial(a)` *is an error, as is* `f := sin; b := f(1);`.

name	section
`acos`	5.7
`acosh`	5.7
`asin`	5.7
`asinh`	5.7
`atan`	5.7
`atanh`	5.7
`assoc`	5.10
`binomial`	
`bitand`	5.14
`bitnot`	5.14
`bitor`	5.14
`bitxor`	5.14
`complex`	5.2.4
`cos`	5.7
`cosh`	5.7
`cxfrag`	5.2.4
`exp`	5.7
`fragment`	5.2.3
`identity`	
`interval`	5.9
`left`	5.2.1
`log`	5.7
`min`	5.6
`max`	5.6
`nop`	
`polynomial`	5.15
`realtoint`	
`reduce`	5.11
`right`	5.2.1
`sample`	5.9
`sin`	5.7
`sinh`	5.7
`spew`	

Table 5: The Built-In Functions. This is the first half of the table.

name	class	syntax	associates	section
parentheses	atomic	`(x)`		
set	atomic	`{x}`		5.2.2
null set	atomic	`{ }`		5.2.2
vector	atomic	`[x]`		5.2.3
blank	atomic	`?`		5.16
quote	prefix	`'x`		5.12
prime	postfix	`x'`		5.8
factorial	postfix	`x!`		??
row extract	infix	`x_y`	left	5.2.3
column extract	infix	`x^y`	left	5.2.3, 5.6
power	infix	`x**y`	right	5.6
apply	infix	`x(y)`	left	5.1.2
apply	infix	`x y`	left	5.1.2
opposite	prefix	`-x`		??
	prefix	`+x`		
not	prefix	`not x`		5.3

Table 3: Operators with Special Syntax. This is the top half the the table.

name	section
`sqrt`	5.7
`square`	
`tan`	5.7
`tanh`	5.7
`triangle`	5.9
`type`	5.16.1
`typematch`	5.16.2
`typematchx`	5.16.2
`zero`	

Table 6: The Built-In Functions. This is the second half of the table.

4 Values and Types

Values fall into four classes, independent of the type system. The four classes are bare, set, symbol, and type.

4.1 Bare

Bare values in fnord correspond to the values of most other programming languages. Here are such common constructions as integers, vectors, and tuples.

A bare value may be a member of one of the *base domains*, or it may be a composite value. The base domains are booleans, integers, reals, complexes, maps. The first four are straightforward. A map is a function expressed in fnorse. Maps are created by the lambda operator (Section 5.1.1). Other languages refer to maps as closures.

A composite value is either a pair of two bare values (see section 5.2.1), or a vector of bare values (see Section 5.2.3).

4.2 Sets

Where bare value are elements of the base domains, sets are subsets of the base and com-

name	class	syntax	associates	section
modulo	infix	`x mod y`	left	??
cross	infix	`x >< y`	left	5.6, 5.9
dot	infix	`x . y`	left	5.6
times	infix	`x * y`	left	5.6
over	infix	`x / y`	left	5.6
cat	infix	`x :: y`	left	5.2.3
plus	infix	`x + y`	left	5.6
minus	infix	`x - y`	left	5.6
equal	infix	`x = y`	left	5.4
exactly equal	infix	`x === y`	left	5.4
not equal	infix	`x != y`	left	5.4
less than	infix	`x < y`	left	5.4
greater than	infix	`x > y`	left	5.4
less or equal	infix	`x <= y`	left	5.4
greater or equal	infix	`x >= y`	left	5.4
and	infix	`x and y`	left	5.3
or	infix	`x or y`	left	5.3
union	infix	`x union y`	left	5.9
enumerate	infix	`x..y`	left	5.13
if	infix	`x if y`	left	5.5
if else	ternary	`x if y else z`	left	5.5
lambda	infix	`x -> y`	right	5.1.1
cast	infix	`x : y`	left	5.1.3
	infix	`x <- y`	left	5.10
comma	infix	`x, y`	left	5.2.1

Table 4: Operators with Special Syntax. This is the bottom half of the table.

posite domains.

A Set is an unordered, non-homogeneous collection of bare values. "unordered" means there is no first element of a set, and there is no natural correspondence between the elements of two sets, even if they have the same number of elements. "Non-homogeneous" means the elements of a set may be of different types. Fnord sets may be continuous, that is, a set might be all the real numbers between 0 and 1, inclusive. It is even possible to have continuous sets in function spaces.

A set has two parts: samples and connectivity. A continuous set is represented with a piece-wise linear approximation, or PLA. The elements of the set are always barycentric linear combinations of the samples. The connectivity controls which samples may be combined. For finite sets the samples are the elements of the set and the connectivity is discrete; no samples may be combined.

Because all the elements of a set are either enumerated, or are between two other elements, all sets are closed, bounded subsets in the mathematical sense.

Sets may contain the same element more than once.

4.2.1 Blocks

The samples of a set are divided into blocks. Each block has homogeneous type and a regular pattern of connectivity. Samples from different blocks are never connected.

4.2.2 Connectivity

The samples of each block are connected in a regular pattern. These patterns are based on n-dimensional meshes. The number of dimensions is the *rank* of the block. If you think of the block as a n-manifold enbedded in m-space, then the rank of the block is usually (unless some of its coordinates are discrete or triangular) n. m depends on the type of the set.

Each coordinate of the rank has the following information: an integer, the discrete/connected flag, and the linear/triangular flag. For now, ignore the triangular flag, that will be explained later.

The integer is the numbers of samples along that coordinate of the mesh. The discrete/connected flags tell whether or not the samples of the mesh are connected in that coordinate. Some examples are in order, see Figure 5

Note that examples 2 and 5 represent identical connectivities. Both are discrete.

The triangular flag is rather strange. The number of samples for this coordinate is now $n(n+1)/2$ if n is the integer. The samples are connected in a triangular mesh; see Figure 6.

The total number of samples in a block is the product of the number of samples for each coordinate. The total dimension (the n in n-manifold) is the sum of the contributions of the coordinates. Discrete coordinates contribute zero, connected linear coordinates contribute one, and connected triangular coordinates contribute two. Thus a block with connectivity [(4, connected, triangular), (3, discrete, linear), (2, connected, linear)] has 60 samples and is a 3-manifold.

4.3 Symbols

A value may be a symbol. These can only be created with the quote operator, see Section 5.12.

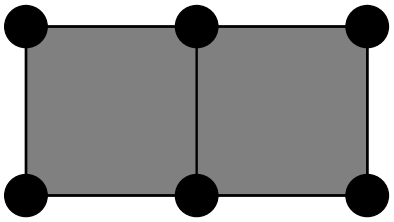
4.4 Type Values

Types are first-class objects, and therefore may be the values of expressions. Some operators work differently when their operands are types, see Section 5.16.

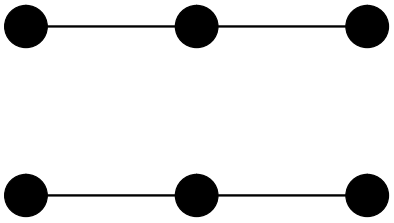
1. rank 1, [(3, connected, linear)]

2. rank 1, [(6, discrete, linear)]

3. rank 2, [(3, connected, linear), (2, connected, linear)]



4. rank 2, [(3, connected, linear), (2, discrete, linear)]



5. rank 2, [(3, discrete, linear), (2, discrete, linear)]

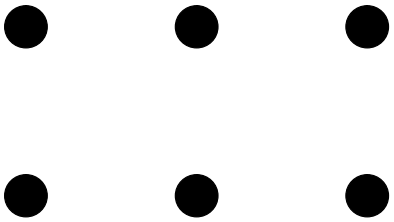


Figure 5: Linear connectivity. For each example the rank is given, followed by the samples and flags for each coordinate.

rank 1, [(3, connected, triangular)]

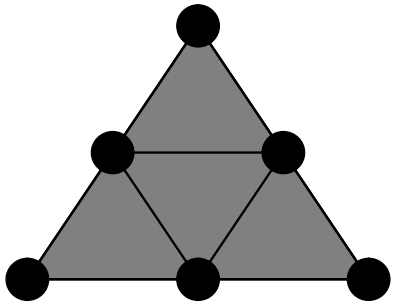


Figure 6: Triangular connectivity.

4.5 Types of Values

The type system is odd. Talk about it in general. List the constructors they correspond to. Make a ref to Section 5.16.

Non-homogenious sets get assigned the type `{?}`. There is no way to construct a type that corresponds to "sets containing some reals and some integers", for example.

Types can have *blanks* in them. Such a type is really a type schema, but fnord doesn't treat such things any differently. A blank has syntax `?`. The length of a vector may also be a blank, and has the same syntax. For example `(Z, R^? -> {?})` represents the type of a pair of an integer and a function from vectors of reals of any length to sets of anything.

Blanks are like type variables, with each blank standing for a different variable. This means that pattern matching rather than unification can be used to type check and cast values.

5 The Operators

This section is a catalogue of all the operators in the fnord language.

5.1 Fundamental Operators

5.1.1 Lambda

The lambda operator creates functions. It does not evaluate its arguments. Its syntax is `x -> y`.

Both arguments to lambda are combined to create a function. A function is represented by a DAG (directed acyclic graph) with one source and one sink. When a function is evaluated, its parameter is placed in the source of the graph; the graph is evaluated; and the value of the function is taken from the sink. See section 6 for more detail on how functions are evaluated.

The DAG for the new function is derived from the abstract syntax trees (ASTs) of the two arguments to lambda. Consider the ASTs to be DAGs with arrows (directed edges) pointing from the leaves to the root (the direction of evaluation). The leaves of ASTs are symbols and constants, the nodes are operations. The left AST is inverted. Then merge each of the inverted tree's ex-leaves with the leaf of the right AST that has the same symbol. Thus the root of the left AST becomes the source of the function DAG the the root of the right AST becomes the sink. See Figure 7 for an example.

This description of lambda ignores scoping. Scoping is done lexically; leaves are not merged if the right leaf is nested inside another lambda and is one of that lambda's parameters. Thus `(x -> x -> x)(y)(z)` ≡ `z`.

The type of the resulting function is `? -> ?` unless there was an expected type. In that case, that type is used.

Lambda is also a type operator, see section 5.16.

`([x, y, z], f) -> f [x, y + z]` becomes

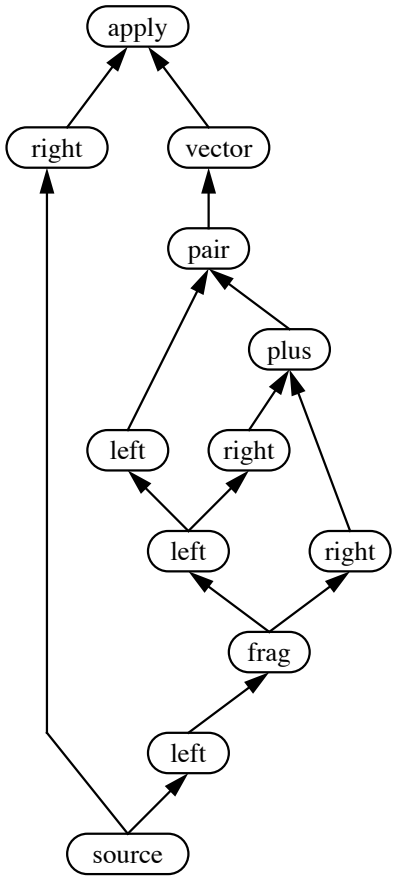


Figure 7: Lambda and argument deconstruction.

5.1.2 Apply

The apply operator evaluates functions on points. It can also represent multiplication.

Its syntax is juxtaposition: `x y`. Apply is one of the most complex operators; it has some special functionality.

Apply evaluates its arguments as usual. If the expected type of the apply is `a` then the expected type of the for the left argument is `? -> a`, and the expected type of the right argument is `?`.

Currently, the way apply works isn't entirely pretty, but it does "do the right thing." When a set of functions is applied to an object, the result may depend on how the set was created. Its behavior is described in Table 7.

In summery, functions map over sets, and a set of functions maps over a value. When a set of functions is applied to a set, there are two possibilities:

- Each function is applied to each point. The resulting points are placed in a set. This is what happens when the set of functions was created with a curried function.
- Each function is applied to the set. The results of these applications are placed in a set. This happens when the set of functions was created explicitly.

See Figure 8 for an example of using the outer product apply.

After substituting and evaluating, the result is cast to the codomain part of the function's type. Thus `(R -> R : x -> x)(Z : 1)` ⇒ `R : 1.0`. Note that the right argument to apply is *not* cast to the domain type before it is substituted. Thus `(R -> ? : x -> type x)(1)` ⇒ `type Z`. This is becuase many operations are faster on integers than on real numbers (ditto for reals and complexes), so it is better to cast as late as possible.

Bug 1 *If a constant valued function is applied to a set of objects, then the result is a bogus value. This value will behave properly most of the time, but not always. Try, for example,*

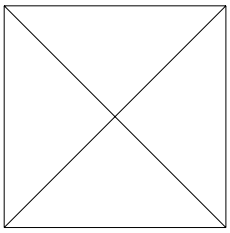


Figure 9: The Tetrahedron.

```
F := R -> Z : x -> 1;
A := Interval(0, 1, 10);
f := (x, y) -> x + y;
print reduce(F(A), f, 0);
```

5.1.3 Cast

The cast operator makes sure an expression has a particular type. It has syntax `x : y`. The left argument is expected to be a type. The right argument is evaluated with this type as its expected type. It is then cast to the type.

It has two main uses. First, when declaring a function it is used to declare the function's type. Thus evaluating `R -> R : x -> 2x` ⇒ `R -> R : fn`, whereas just `x -> 2x` ⇒ `? -> ? : fn`. A specific type is required if a function is to be differentiated (see Section 5.8), or to map over sets (see Section 5.1.2).

The other main use is to cast a scalar up the domain chain. Thus `R : 1` ⇒ `R : 1.0` and `C : 1` ⇒ `C : 1.0 + 0.0i`.

The cast operator accepts types with blanks in them. Thus `(R, ?) : (1, 2)` ⇒ `(R, Z) : (1.0, 2)` and `R^? : [1, 2]` ⇒ `R^2 : [1.0, 2.0]`.

Bug 2 *Casting down should be an error; instead it produces bogus objects. For example,*

left type	right type	action	result type	example	result
N	B	multiply	N	2 3	6
A -> B	A	simply-apply	B	f 2	6
A -> B	{A}	right-map	{B}	f {2, 3}	{6, 9}
{A -> B}	A	left-map	{B}	{f, g}2	{4, 6}
{A -> B}	{A}	outer-product	{B}	G {2, 3} {2, 3}	{4, 6, 6, 9}
{A -> B}	{A}	double-map	{{B}}	{f, g}{2, 3}	{{4, 6}, {6, 9}}

symbol	denotes
N	any non-functional type
A, B	any type
f	f := Z -> Z : x -> 3 * x
f	g := Z -> Z : x -> 2 * x
G	G := Z -> Z -> Z : x -> y -> x * y

Table 7: Apply's Cases.

```
Tetrahedron :=
  ([[-1, -1, -1], [ 1,  1, -1], [-1,  1,  1], [ 1, -1,  1]],
   {[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]})
  ;

makeEdges
  := R^?^? -> Z^2  -> R -> R^?
  :  verts -> edge -> t -> t verts^(edge^1) + (1-t)verts^(edge^2)
  ;

Polygon
  := (R^?^?, {Z^2}) -> {R^?}
  :  (verts, edges) -> (makeEdges verts edges)(Interval(0, 1, 2))
  ;

widget Show(Polygon(Tetrahedron));
```

Figure 8: Code that makes use of outer-product apply.

`(Z : 1.1)` $\Rightarrow$ `(Z : 1.1)`.

5.2 Constructors and Destructors

Define constructuors and destructors. What the hell is their definition anyway?

5.2.1 Pairs

Pairs the the analog of LISP's cons cells. Any two objects may form a pair. Pairs are used to form longer length tuples and lists. Pairs are constructed with the comma operator and deconstructed with the left and right operators (LISP's car and cdr).

The syntax for pair is `x, y`. Left and right are built-in functions.

The members of a tuple are its leaves. Thus the members of a tuple may not be pairs.

Tuples of the same length have multiple representations, for example a 3-tuple may be created with either `((1, 2), 3)` or `(1, (2, 3))`. Operations that pay attention to the members of a tuple treat the representations equivalently, but to other operations it is significant.

Pair, left, and right are all type operators, see section 5.16.

5.2.2 Sets

Sets are modeled after their mathematical namesake. See section 4.2 for a precise description. Sets are unusual because there are no destructors[1].

There are two seperate operators for creating sets, which appear almost the same: the set operator and the null set. The syntax for set is `{x}`. Its argument is a tuple. The elements of the created set are the members of the tuple. Such sets have discrete connectivity, see Section 4.2.2.

Caveat 2 *Because the elements are the members of a tuple, it is difficult to explicitly create a set with elements that are tuples. There are two ways to do it, however:*

- *A product of singleton sets, for example* `{1} >< {2}` *creates a set containing the pair* `(1, 2)`.
- *Use a function like*

```
TupleSet
   := (?, ?) -> {?}
   :  tuple  -> (Z -> ? :
                 n -> tuple){1}
   ;
```

The null set has syntax { }. It creates a set with no elements.

Set is a type operator, see Section 5.16. There are other ways of creating sets, see Section 5.9.

5.2.3 Vectors

Vectors are also modeled after their mathematical namesakes, but more closely. A vector has some number of entries. Each entry can be any object, but they all must be the same type. The number of entries is the *dimension* of the vector.

Vectors are created in much the way sets are, the vector build operator has syntax `[x]`. The vector created has entries that are the members of tuple `x`.

Caveat 3 *Because the members of a tuple are used, vectors cannot have entires that are themselves tuples. This causes problems (see Section 5.8).*

There are two ways to destruct vectors. The vector fragment operator converts a vector to

[1] The reduce operator could be considered a destructor of sorts, see Section 5.11.

a tuple whose members are entries of the vector. The tuple will be flat. Thus `fragment[x, y, z]` ≡ `((x, y), z)`. Users rarely use this operator directly, instead it is generated by the lambda operator, see Section 5.1.1. fragment is a built-in function.

The vector constructor is smart enough to map over sets. Thus `[{1, 2, 3}]` ⇒ {`[1]`, `[2]`, `[3]`}.

The extract entry operator has syntax `x^y`. `x` is a vector, and `y` is an integer index. Vector entries are indexed from one. Thus `[x, y]^1` ≡ `x`. Using an out-of-range index is an error.

The concatenate operator has syntax `x :: y`. It is defined by `x :: y` ≡ `[(fragment x), (fragment y)]` [2]. For example `[1, 2] :: [3, 4]` ⇒ `[1, 2, 3, 4]`. Since strings are really integer vectors, `::` is also the string concatenation operator; `"one" :: "two"` ≡ `"onetwo"`.

There are actually two kinds of vectors: row vectors and column vectors. The two types are identical except in name, and how some operators may behave. Currently there is no way to create row vectors. The row extract operator (`x_y`) is therefore of no use. This will change when the transpose operator is written.

5.2.4 Complexes and Booleans

There is no literal form for complex numbers or booleans. Therefore, they must have their own constructors. A complex number is created with the built-in function `complex`. Its arguments are the real and imaginary parts. For example, the standard definitions include `i := complex(0, 1);`. After this definition, complex numbers may be entered in more familiar forms such as `(2+3i)(4i)(1-i)`.

[2]This definition wouldn't work if vectors entries could be pairs, at some point I may add an operator that creates vectors of pairs, and then `::` will be better justified

The destructor for complex numbers is the `cxfrag` operator, which takes a complex number and returns a pair of reals. Thus `cxfrag(complex(1, 2)` ⇒ `(1.0, 2.0)`.

Booleans also have no literal form. They are defined in the standard thus: `True := 0 = 0; False := 1 = 0;`. The closest thing to a destructor for booleans is the if-else operator.

5.3 Logical

The logical operators are and, or, and not. They have syntaxes `x and y`, `x or y`, and `not x`, respectively. Their operands are expected to be booleans. Each performs the obvious operation. They do *not* short circuit. That means that when evaluating `x and y`, if `x` is false, then `y` is still evaluated. Since fnord is purely function, one usually cannot tell the difference. The exception to this is termination of recursive functions.

5.4 Relational

The relational operators are equal, exactly equal, not equal, less than, greater than, less or equal, and greater or equal, with syntaxes `= === != < > <= >=`.

As noted in Section 3.3, the relational operators chain. Operators chain in any combination, and to any length.

All of the relationals are defined in the obvious way on the integers, with equals and exactly equals being identical.

On reals, equals is up to epsilon, less (resp. greater) mean "less than and not within epsilon", and less or equal means "less than or exactly equal". epsilon is a small real number, typically about 1e-5.

On complexes, only equal, not equal, and exactly equal are defined; the first two are epsilon based.

Equals, exactly equals, and not equals are also defined on booleans and symbols. They

are *not* defined on tuples, vectors, or functions of any sort.

5.5 Conditional

There are two conditional operators, if-else and if. The syntax for if-else is `x if y else z` and `x if y` for if.

If-else first evaluates the clause (the second argument) with an expected type of `B`. If its value is true, then the first argument is evaluated and returned, otherwise the third argument if evaluated and returned.

Bug 3 *When the if-else is inside an apply that is mapping over a set, then* all *arguments are fully evaluated, and the types of the first and third arguments must be castible. Eventually it will only evaluate the what's necessary, but the castible requirement will stay.*

The if operator is similar to assert() in C. If the clause is true, then the value of the if is the value of the first argument, otherwise it is an error.

The expected type of the first and third arguments is the expected type received by the if-else or if.

5.6 Arithmetic

The arithmetic operations are addition, subtraction, min, max, multiplication, division, power, dot product, and cross product. They have syntaxes `x + y` `x - y` `x * y` `x / y` `x ** y` `x . y` `x >< y`, except min and max, which are built-in functions.

The arguments are evaluated as usual. For addition, subtraction, min, max, and cross the expected type of the arguments is the expected type received. For multiplication the expected type for both arguments is `?`. For division the expected type for the left argument is the expected type received, and the expected type for the right argument is a scalar of unknown base domain. [[Power]]. For dot the expected type for both arguments is `A^?`, where dot received type `A`.

Caveat 4 *The expected type generated by multiplication is lame.*

Addition and subtraction work on vectors and tuples of integers, reals, and complexes in any combination. Min and max are the same, except don't handle complexes.

Multiplication is either scalar-anything, anything-scalar, matrix-matrix, or matrix-vector. Integers, reals, and complexes are supported. Division may only be anything-scalar.

[[Power]]

Dot product takes two equal length vectors of scalars (integers, reals, or complexes). If the right vector is complex, it is conjugated.

Vector cross product works on length three vectors of reals or integers only. If the arguments are sets, then a cartesian set product is performed, see Section 5.9. Cross is also a type operator, see Section 5.16.

5.7 Scalar Math Functions

An assortment of scalar math functions are provided. They are all built-in functions, so they don't have any syntax. See Tables 8 and 9 for complete lists. They evaluate their argument with the same expected type as they receive. The type of the result is real, unless the argument was complex, then it is complex. This applies to square root in particular: `sqrt(-1)` ⇒ `NaN`, but `sqrt(C : -1)` ⇒ `0 + 1i`.

The scalar math functions map over sets.

5.8 Differentiation

The differentiation operator takes functions are returns their derivative. It can handle a

sin cos tan ln sqrt exp

Table 8: Functions for reals and complexes. `ln` and `exp` are base e

`sinh cosh tanh`	hyperbolic
`asin acos atan`	inverse
`asinh acosh atanh`	inverse hyperbolic

Table 9: Functions for reals only

wide variety of domains and codomains. It has syntax `x'`.

The code for the derivative is generated in the following fasion: First, the domain type of the function is converted into equivalent code. Each leaf of this tree connects to the sink of a DAG derived from the body of the function by the usual derivative rules. Where the derivative of the parameter is required in this DAG, the domain type is again translated into code, but this time the leaves are replaced with ones and zeroes rather than DAGs. A leaf is a one if it is the same leaf as we are currently working on at the top level, otherwise it is a zero.

The type of resulting function has the same domain as the argument and a codomain of `?`.

Complicated vector (and worse) functions are handled correctly[3] because of the use of the domain type. See Table 10 for examples. It is easy to predict what the type of the derivative is going to be: just substitute the type of the codomain in for each base domain in the domain.

Bug 4 *Note that because of the way the vector build operator flattens tuples, not all function types can be differentiated, and some may be done incorrectly. For example, a function of type* `(R^2, R) -> (R^2, R)` *requires* `[(R^2, R), (R^2, R)]`*, which is illegal, and* `(R^2, R) -> (R, R)` *results in* `(R^4, (R, R))` *rather than* `((R, R)^2, (R, R))`*.*

It would be nice to give an example, but anything non-trivial is really immense.

Bug 5 *Not all the operators are handled. For example, none of the exclusively integer or set-oriented functions work. The only ones that* should *work and don't are* `polynomial`*,* `min`*,* `max`*,* `complex`*, and* `cxfrag`*.*

5.9 Set Operations

Besides the set constructor operator, there are several other ops that create and manipulate sets. The interval and triangle operators create connected sets; they are built-in functions. The union and cartesian product operators both take two sets and return another set; they have syntaxes `x union y` and `x >< y`, respectively. The sample operator reduces a set to its samples; it is a built-in function.

The interval operator creates a connected subset of the reals; `interval(0, 1)` creates the set of all reals between 0 and 1, inclusive. The first two arguments have expected type `R`. An optional third argument has expected type `Z`, it specifies the number of samples taken on the interval. It is an error if the first argument is greater than the second, or if the third is less than one. The samples are spaced evenly across the interval with one at the lower bound and one at the upper bound.

The triangle operator creates a connected subset of `R^?`. If the expected type received is `A^n` then the expected type used to evaluate the first argument is xxx. The three columns of the matrix are treated as the three vertices of the triangle. The expected type for the optional second argument is `Z`. It specifies the number of samples along each edge of

[3] Almost; when converting the domain type into code, row and column vector build operations should be exchanged. Fnord doesn't do this right now because the different vector types aren't really distinguished

function	derivative
`R -> R`	`R -> R`
`R -> R^n`	`R -> R^n`
`R^n -> R`	`R^n -> R^n`
`R^n -> R^m`	`R^n -> R^m^n`
`(R^2^2, R^3) -> R^4`	`(R^2^2, R^3) -> (R^4^2^2, R^4^3)`
`(R, R) -> (R, R, R)`	`(R, R) -> ((R, R, R), (R, R, R))`

Table 10: Types of Differentiated Functions

the triangle; the total number of samples is $n(n+1)/2$.

The number of samples argument to both triangle and interval is not used directly. There is an interpreter global which controls the sampling. This parameter (a real number) is multiplied by the passed value before use. Currently there is no way to control the global, it is always `1.0`. [[the other globals should be described in appropriate places, and there should be a section that refers to them]].

The union operator evaluates both is arguments with same expected type that it receives. They must both be sets. It returns the union.

If the expected type received by the cartesian product operator is `(A, B)` then the expected type of the first argument is `A` and the expected type of the second argument is `B`. Both arguments must be sets. If the first argument is X and the second is Y, then the product is $\{(x,y)|x \in X, y \in Y\}$. It works correctly for connected sets. Note that to compute the connectivity of a product, one just concatenates the connectivities of the two sets.

The sample operator takes a set and returns a discrete set whose points are the samples of its input. Thus `sample(interval(0, 1, 5))` $\Rightarrow$ {`0.0, 0.25, 0.5, 0.75, 1.0`}.

5.10 Associations

A-lists in fnord are similar to a-lists in LISP. In fnord, the cars of the cells must be symbols, and rather than a list, you can have a tree. You can create them by hand using quote and pair like this: `(‘foo, 1), (‘bar, True)`, or you can use the a-list pair operator: `foo <- 1, bar <- True`.

The advantage of using trees rather than lists for a-lists is that two can be combined with cons, rather than append. So if `A` and `B` are a-lists, then `A, B` is also an a-list.

Values can be extracted from a-lists using `assoc`, a built-in operator. It takes a symbol to search for and an a-list, and returns the value of that symbol. It is an error if the symbol has no association. The a-list is searched left to right; the first association found is returned.

5.11 Reduce

The reduce operator takes a discrete set, an associative function, and an initial value. The function is applied "between" the elements of the set. For example, if the function is `(x, y) -> x + y` and the initial value is `0`, then reduce will sum the elements of the set. It is an error if the set is not discrete. The function should be associative because the order of its application is not defined (this is to allow log-

```
factorial
  := Z -> R
  :  n -> reduce(({R} : {3..n}),
                 (a, b) -> a*b,
                 2)
  ;
print factorial(20);
=> R : 2.4329e+18
```

Figure 10: Factorial using Reduce.

time implementation on a SIMD machine). If it is not associative, it is not an error, but the result is undefined.

Bug 6 *Currently only the following functions are supported by reduce: addition, min, and max of anything; scalar multiplication; union; and logical or and and.*

Bug 7 *The set cannot be a require casting to match the initial value.*

Reduce is a built-in function. If the expected type received is `A`, then the expected type of the first argument is `{A}`, the expected type for the second argument is `(A, A) -> A`, and the expected type for the third argument is `A`. The arguments are all evaluated in normal fasion.

See Figure 10 for an example.

5.12 Quote

The quote operator has syntax `‘x`. Its argument must be a literal symbol. The result is a symbol value. For example, the value of `‘foo` is the symbol `foo`. This operator isn't used very often, except when specifying the parent widget of a widget; see Section 7.6.

Bug 8 *Symbols are not properly handled by the interpreter. The two ways that this manifests itself are*

- *Symbols cannot appear more than once in the same expression; thus* `(‘foo, ‘foo)` *produces an error.*
- *A symbol cannot be the sole argument to a function;* `(x -> x)(‘foo)` *fails.*

5.13 Enumerate

The enumerate operator creates a tuple of consecutive integers. It has syntax `x..y`.

Its arguments must be integers. It creates a left-associated tuple whose entries are the integers starting with the left argument, and ending with the right argument. Thus `(2..5)` ⇒ `(2, 3, 4, 5)`. Enumerate is often combined with vector and set operations `{1..3}` ⇒ `{1, 2, 3}` and `[1..3]` ⇒ `[1, 2, 3]`.

It is an error if the first argument is greater than the second. If they are the same, then the result is not a tuple: `1..1` ⇒ `1`.

Bug 9 *Enumerate does not work inside of an apply that is mapping over a set. This is because the type of its value is not entirely determined by the type of its arguments.*

5.14 Bitwise Operators

The bitwise operators are `bitand`, `bitxor`, `bitor`, and `bitnot`. They operate on integers. They are built-in functions. They evaluate their opperands with an expected type of `Z`. Each performs the obvious operation.

5.15 Polynomial

The `polynomial` operator evaluates multivariate polynomials using Horner's rule. It is a built-in function with two arguments. The

first argument is a scalar. Its expected type is the same as the expected type received by the polynomial operator. The expected type for the second argument is a vector of any length. Integers, reals, and complexes are all handled.

The vector contains the coefficients of the polynomial, and the scalar is the value of the variable. Thus `polynomial(2, [1, 2, 3])` $\Rightarrow$ `17`.

The vector can really be a tensor of any order. The rank is reduced by one by performing a polynomial calculation on each row of the (for example) matrix. The result is a vector; this can then be fed into polynomial again to evaluate multi-variable polynomials. For example `polynomial(x, polynomial(y, [[1, 0, 3], [4, 5, 0]]))` $\Rightarrow 1 + 3y^2 + 4x + 5xy$.

5.16 Type Operators

A number of operators work on type values as well as normal values. These operators include: pair, lambda, cross, left, right, set, and extract entry. There are also a few operators that operate exclusively on types; they are described in the following subsections.

Mostly, their operation as type operators is pretty simple: if their arguments are types, then they function as a type operator, otherwise they behave normally. There are a few complications, these are dealt with later.

The actions of all the type operators is described in Table 11.

The type operators can also deal with arguments are that are not types. If an operand is a normal value with type `{X}`, then the type operator uses `X`. This is to facilitate "types by example". See Figure 11.

Note that ambiguity is introduced in two cases. Both lambda and pair can be interpreted either as type expressions or as regular expressions. For example, `(R, R)` might mean `(type R, type R)` or `type (R, R)`. Also `R -> R` might mean the identity function, or `type (R -> R)`. These ambiguities are resolved using the expected type received by the operator.

```
U := Patch((0, 1, 10), (0, 1, 20));
X := U -> R^3 : u -> [u, v, sin(u v)];
```

Figure 11: Type by Example. The type of `X` is `R^2 -> R^3`; the type of `U` is `{R^2}`.

5.16.1 The Type Operator

The `type` operator is used to extract the type of an object. It evaluates its argument, and returns the type of it. Thus `type 1` $\Rightarrow$ `type Z` and `type[{1}]` $\Rightarrow$ `type {(Z^1)}`. If you pass it a type, it just returns that type, thus `type(type x)` $\equiv$ `type x`.

The symbols `B`, `Z`, `R`, and `C` are defined in the standard library with the type operator:

```
B := type (0=0);
Z := type 0;
R := type 1.1;
C := type complex(0,0);
```

5.16.2 Type Matching Operators

There are two operators which can be used to determine if two types are the same. `typematchx` makes sure they are exactly the same, `typematch` will allow casting. Both return a boolean value; true if they are the same.

5.17 Casting

All the operators cast their arguments up the base domain hierarchy as needed. Does there anything more than that need to be said? Examples?

name	input	output
pair	`(type X), (type Y)`	`type (X, Y)`
lambda	`(type X) -> (type Y)`	`type (X -> Y)`
cross	`(type X) >< (type Y)`	`type (X, Y)`
left	`type (X, Y)`	`type X`
left	`type (X -> Y)`	`type X`
left	`type (X^n)`	`type X`
right	`type (X, Y)`	`type Y`
right	`type (X -> Y)`	`type Y`
right	`type (X^n)`	`n`
set	`{type X}`	`type {X}`
extract	`(type X)^n`	`type (X^n)`
blank	`?`	`?`

Table 11: Type Operators. `X` and `Y` are any types, `n` is an integer.

6 The Interpreter

This section talks about the interpreter at a high level. It also touches on the implementation.

6.1 Type Inference

Fnord includes a very weak form of type inferencing. The normal way to evaluate an expression tree is as follows: recursively evaluate both children, and then combine with the operator for this node.

In fnord, there is always a *expected* type. Before evaluating the children, an expcted type is calculated for them. For example, if we expect the result of an add operation to be an integer, we can expect that both its operands are integers as well.

Not all cases are so easy; for example, if the expected type is `R^3^3` and the operation is multiplication, then you can't tell if the right argument should be in `R^3` or `R^3^3` (it may be a matrix-matrix or a matrix-vector multiplication). In cases like this, either a choice is made, or a expected type of `?` is used.

Computation and information flow down the tree as well as up the tree.

Say more; eg how this is used. Mention index notation.

6.2 Parallelism

When a function is mapped over a set, it must be evaluated for each point of the domain. Each of these evaluations is independent; they can be carried out in a SIMD (Single Instruction, Multple Data) fasion. Thus, fnord should be able to make good use of most sorts of parallel hardware, including shared memory multiprocessors and vector machines.

Even on a single processor machine, doing evaluation in SIMD fasion is highly profitable. The reason is that it reduces overhead. The function DAG is only traversed once rather than once for each domain point. The CPU spends its time in tight loops rather than chasing pointers.

Because fnord is interpreted, however, maps and reduces consume far more memory bandwidth than a compiled language could use.

6.3 Performance

Give numbers. Integrate with previous section? Comparison to other packages?

Fnord performs very badly on flow of control. Make more of this.

6.4 Input

The interpreter reads text from each of the file names specified on its command line. The file name "stdin" is treated specially. If no files are specified, then stdin is used.

There is also a gnu-emacs mode for editing and running fnord programs. It is quite primitive, but functional.

6.5 Commands

Commands cause the interpreter to evaluate the expression, and then do something with the result. The commands are

print writes the print-form of the value to standard out.

debug dumps internal data structures for the value to standard out.

eval does nothing with the value.

The tokens are `print`, `debug`, and `eval`, respectively.

6.6 Directives

The interpreter directives and their actions are

fix serve as a marker for clear. This may only appear once.

clear remove all definitions since the interpreter was fixed.

quit stop reading input and exit.

The tokens are `FnordFix`, `FnordClear`, and `FnordQuit`, respectively.

Fix and clear are implemented using environments. Definitions are collected into a permanent environment until the fix. Then a temporary environment is created which points the permanent environment, and definitions go into the temporary one. The clear clears the temporary environment, leaving the permanent environment untouched.

The fix/clear functionality is used by the interactive text widget, see Section 7.3.

7 Visualization System

The visualization system is based on *widgets*, so called because they map one to one with X toolkit widgets (there are many exceptions, but this is usually true). Widgets provide input and output to fnorse programs. For example, the view widget displays a graphical representation of the value of a fnord program. That fnord program may use the value of a slider widget in its computations. In general, widgets are used at the ends of the "computational pipe," where the purely functional model is less well suited.

A widget is created with a widget declaration. See Section 3 for the syntax. The expression should be an a-list. The associations of symbols in the a-list determine all characteristics of the widget: type, appearance/geometry, possible values, and behavior.

The association of the `type` symbol should be a symbol that is the name of a widget class. See Table 12 for a complete list. It determines the class of the widget. [[probably `type` should be replaced with `class`]]. This rarely actually appears in user code. Instead the widget functions from the standard library are used, see Section 8.

By default, widgets of each class are placed in the same window. They are stacked ver-

tically. View widgets are an exception; each is placed in its own window. It is possible to control exactly which widgets appear in what windows, and how they are arranged, see Section 7.6.

The association of the `name` symbol is used as the name for a widget. This name is usually displayed somewhere on the widget, or on the title-bar of its window. If the widget is bound to a symbol, then the symbol is used as a default for the name. If all else fails, "default" is used as the name.

7.1 Input Widgets

There are a variety of input widgets, including buttons, checkboxes, sliders, type-ins, and even a bitmap editor and tapedeck.

7.1.1 Checkbox Widget

The simplest widget is the checkbox widget. Its value is either true or false. Each time you click on it, it toggles from true to false, or vice versa.

7.1.2 Button Widget

The button widget is like the checkbox widget in that they are both boolean valued. It is different, however, because it is a *strobe* widget. Strobe widgets interact in special ways with evolution widgets, see Section 7.4. The value of the button is always false, except for after it is clicked it will be true for exactly one iteration of any running evolution widgets.

7.1.3 Slider Widget

A slider widget appears as a scrollbar on the screen. Its value is a number between the values of the min and max attributes. Min and max default to 0 and 1. If just min is set, then max defaults to min + 1. When the widget is created, its value is taken from the `init` attribute. The value of the widget can be either `B`, `Z`, `R`, or `C`, depending on the value of the `field` attribute (it is truncated for integers, and injected to the real line for complexes). The `drag` attribute controls when the value of the slider changes. If it is false, then it will change only when the mouse button is released. If it is true, then it will change as the mouse is moved.

7.1.4 Type-in Widget

Type-in widgets are similar to slider widgets, but the user specifies an exact value by typing it in. It doesn't need the `min` and `max` attributes; it does recognize the `field` attribute. The value is updated when the Return key is pressed in the widget.

7.1.5 Tapedeck Widget

The tapedeck widget is also similar to the slider widget, but it can change its value on its own at regular time intervals. The widget appears as three buttons below a label. The buttons are are forward, reverse, and stop. The label displays the current value of the tapedeck. When the tapedeck is playing (either forward or reverse), its value is repeatedly increased (decreased for reverse) by the value of the `speed` attribute. The tapedeck widget recognizes the `init`, and `field` attriutes; they have the same meaning as with slider widgets.

7.1.6 Bitmap Widget

The bitmap widget is the most complex input widget. Its value is an array of integers, which are either zero or one (yes, they should be booleans). The size of the bitmap is determined by the `width` and `height` attributes. The size of one bit in the editor is determined by the `zoom` attribute; each bit is a square that

widget	usage	active	strobe	types
bitmap	input	no	no	`Z^?^?`
button	input	no	yes	`B`
checkbox	input	no	no	`B`
editor	neither			
evolve	input	yes	no	any
object	neither			
output	neither			
point	input	no	maybe	`R^n` or `(B, R^n)`
printer	output			any
slider	input	no	no	`F`
tapedeck	input	yes	no	`F`
typein	input	no	no	`F`
view	output	yes	no	$\{$`R^n`$\}, 2 \leq n \leq 4$
window	neither			

Table 12: Widget Classes and Types.

category	attributes
class	type
appearence	top bottom left right width height parent name
output	set
input	min max init field next
particular	numUnits drag color append zoom speed modifiers dims strobe

Table 13: Complete Attribute List. [[This is a terrible table]]

many pixels on a side. Use the left mouse button to set bits, the right button to clear bits, and the middle button either sets or clears, whichever makes a difference (this is how a standard Macintosh "pencil" tool works).

Caveat 5 *Because the* `width` *and* `height` *are used for the size of the bitmap, they cannot be used to place a bitmap widget inside a window widget. Use the* `bottom` *and* `right` *attributes instead.*

7.1.7 Point Widget

The point widget rivals the bitmap widget in complexity; it certainly has more options. The purpose of the point widget is to translate clicks of the mouse (button two always) into 2d or 3d points. The type of the value of the strobe widget depends on the `strobe` and `dims` attributes. If strobe is on, then its value is `(B, R^dims)`, otherwise just `R^dims`. `dims` must be either 2 or 3. The boolean part of the value is true when the mouse is clicked, or, additionally, when it is moved if `drag` is true.

The point widget's `parent` is usually set to a view widget. In this case, the click point is transformed to account for the view's camera so that the widget's value corresponds to the location of the mouse (See Figure 12). Otherwise the value is the coordinates of the click in pixels.

Multiple point widgets may coexist in the same view widget. The `modifiers` attribute is used to decide which view widget gets a particular mouse click. Its value is a standard Xt modifiers string, for example `None`, `Shift`, `Ctrl`, or `cs` (which is an abbreviation for control and shift).

Bug 10 *There is a terrible memory leak when the point widget is strobed.*

```
N := D -> (D union {right pt})
          if (left pt) else D;

E := widget Evolve({}, N);

v := widget Show(E union
                 Axes union
                 Labels);

pt := widget Point,
              strobe <- True,
              parent <- `v,
              dims <- 3;
```

Figure 12: Example Use of Point Widget.

7.2 Output Widgets

Output widgets provide one of two ways to get results out of the interpreter (the other is the print command and its ilk). They have the advantage of being *dynamically updated.* In general, an output widget is given a fnord value to display. The value is given as the association of the `set` attribute.

We say output widgets are dynamic because if the value of an input widget is changed by the user (or even changes on its own accord), and the value displayed by an output widget depends on (used in the course of computation) the input widget, then the output value will be recomputed and redisplayed in the output widget. See Figure 13 for an example

7.2.1 Print Widget

The print widget pretty-prints the value and displays the text in a text editor widget. If the `append` attribute associates to true, then when new values are displayed, they are appended to the text (rather than replacing it).

```
X := R^2 -> R^3
  :  [u, v] -> [u, v, (u v) ** n]
  ;
U := Patch((-1, 1, 30), (-1, 1, 30));
n := widget Slider(0, 10), field <- Z;
widget Print(n);
widget Show(X(U));
```

Figure 13: Example of dynamic output widget. When the slider for `n` is moved, the value in the print widget, and the surface in the view widget both change to reflect the new value.

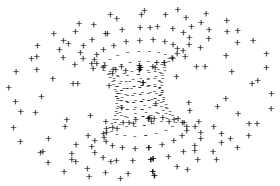


Figure 15: The Text-Torus.

7.2.2 View Widget

The view widget is by far the most used output widget. It displays three-dimensional geometry; that is, elements of {`R^3`}. It will actually accept several sorts of sets, and convert them as needed. The conversions follow:

- Sets in {`R^2`} are injected into the XY plane.
- Sets in {`R^3`} are preserved.
- Sets in {`R^4`}, a coordinate is dropped and used as a color index.
- Sets in {`(R^n, Z^?)`} each point is a pair of a geometric point (like one of the above), and a string. The string is printed as two-dimensional text starting at the point. See Figure 14.

Objects in {`R^2`} and {`R^3`} may also be assigned a color (here a single color is used for the entire set). How to do this, as well as how to select coordinates and color maps for the other cases is explained in Section 7.7. In short, the set attribute accepts object widgets which attach color information to normal sets.

The operation of the view widget is relatively complicated. The object or objects are initially displayed in wire-frame. They are (collectively) scaled and translated to fill the window nicely. The camera projection is orthographic (parallel), and starts out looking towards -Z, with +X pointing up. Use the left mouse button to rotate the objects. The interaction technique is called "virtual sphere" rotation; one is supposed to imagine that the obects are imbedded in a transparent, frictionless trackball; dragging the mouse rotates the trackball. The simulation is so accurate that if you release the mouse button while moving the mouse, the sphere (and hence the objects) will continue to spin indefinitely. To stop the rotation, click the left button (this is called *inertial* rotation). Typing the 'r' key will reset camera to its initial state.

If the objects being viewed change due to user interaction with other widgets, the scale and translate initially chosen by the view widget may become obsolete. To rescale and recenter the camera on the object, type 's'.

The initial viewing mode is wire-frame mode. In it, points appear as points, lines appear as lines, and surfaces appear as rectangular meshes edges only. In hidden-surface mode, points appear as points, lines appear as lines, and surfaces appear as opaque, edged

```
U := Patch((-pi, pi, 20), (-pi, pi, 20));

F := R^2    -> R^4
  :  [x, y] -> [sin x, cos x, sin y, cos y] / sqrt(2);

project
   := R^4 -> (R^3, Z^?)
   :  [x, y, z, w]   -> ([x, y, z] / (w - 1), "-" if w < 0 else "+")
   ;

widget Show(project(F(U)));
```

Figure 14: Two Dimensional Text. Text is also useful for attaching labels to various objects.

triangles. The space bar toggles between wire-frame and hidden-surface modes.

7.3 Text Widget

The text widget is entirely different from the other widgets; it is neither input nor output. It provides a mouse-based editor for editing fnorse programs. It can send its text directly to the interpreter. Before the text is sent to the interpeter, the interpreter is cleared, as if by the clear directive (See Section 6.6). Thus some large quantity of fnorse code may be loaded into the interpeter and fixed. Then each time the contents of the text widget is executed the fixed environment is unsullied by any previous runs.

The operation of the text widget is quite simple. The actual text editing commands are the standard Motif key bindings (?). Buttons exist to save and load files into the editor, run the current contents, and quit (the program, not the widget).

Multiple text editors can coexist, but only one can be running at a time.

7.4 Evolution Widget

The evolution widget is quite different from the other input widgets; it maintains state. You give it an initial state, and a next-state function. The function is iterated on the state to give the sequence of values for the widget.

An evolution widget appears on the screen as a reset button and a toggle switch. When the toggle switch is on, the state of the widget will be updated with its next function. If the reset button is pressed, the state reverts to its initial value, and evolution starts over.

The value of the widget only changes when it is evolving (the toggle switch is on), even if other widgets which contribute to its value change. For example, if the initial value of the evolution widget is determined by a slider, then only when the evolution widget is reset will the value of the slider be read.

Many evolution widgets can coexist; they run synchronously. Thus if multiple evolution widgets use the value of a strobe widget, then the strobe widget will be true for all the evolution widgets for one iteration. In fact, all "active" widgets run synchronously, so, for example, if the value of a tapedeck is used in a

evolver, each value from the tapedeck will be seen once by the evolver. This also applies to a view widget that is spinning intertially. This applies to inertially spinning view widgets as well.

See Figure 16 for an example.

7.5 Output Widget

The output widget is a temporary hack. Its functionality will eventually be subsumed by the view widget.

The output widget generates a PostScript description of an image that appears in a view widgets, and writes it to a file. Type in the name of the view widget (it must have a name to be printed!) and the file to write to, and click on "print" button. The view is printed exactly as it appears, camera, hidden surface, and even size are all maintained.

The generated code includes a bounding box, for easy inclusion into TEX documents with Psfig.

7.6 Window Widget

Window widgets exist solely as containers for other widgets. With them you can arrange your other widgets arbitrarily into windows, and place them where you want within a window.

The size of the window widget is given in pixels by the width and height associations. Widgets are positioned inside of it with respect to an imaginary grid placed over the widget. The (integral) number of divisions in the grid is given by the `numUnits` association. Its default is three. The grid always has the same number of divisions horizontally and vertically, regardless of the aspect ratio of the window.

Another widget is placed inside of a window widget by setting its `parent` association to the *quoted* name of the window widget. The top, bottom, left, and right associations specify a bounding box (using the parent's imaginary grid coordinate system, numbering from zero) exactly containing the child widget. You can also use top, bottom, width, and height. Again, if none (or an incomplete set of) of these attributes are specified, the widgets are stacked vertically. The `Location` function is usually used instead of using these associations directly.

See Figure 17 for an example.

7.7 Object Widget

The object widget is used to attach object-specific (as opposed to view-specific) viewing information to a normal set. Currently, the only such information is color. In the future other surface properties and the lighting may be controlled from here. Object widgets are not associated with X toolkit widgets, they have no manifestation on the screen of their own.

The object widget recognizes only two attributes, set and color. The value associated with set is the fnord value in `{R^2}`, `{R^3}`, or `{R^4}`. The color attribute must be associated with a string. The string has several possible formats. They are described in Table 14. An object with solid color appears in that color (or the best approximation that the window system can make). If the color is indexed, then the specified coordinate is used to interpolate between the two colors given. Zero results in the first color; one in the other color, and values in between are linearly interpolated in RGB space. Other values are clamped.

8 The Standard Library

explain contents std.fnorse, and any other such files I write.

```
n := widget type <- ‘evolve, init <- 0, next <- (x -> x + 1);
widget Print(n);
```

Figure 16: Example of Evolution Widget. The value printed enumerates the integers from 0.

```
g := R -> R^2
  :  t -> [sin(a t), cos(b t)]
  ;
h := R -> R^2
  :  t -> [sin(a t), sin(b t)]
  ;
I := Interval(0, 6pi, 100);

myWin := widget Window(6), name<-"Lissajous Figures";
a := widget Slider(0, 1), parent <- ‘myWin, Location(3, 0, 4, 3);
b := widget Slider(0, 1), parent <- ‘myWin, Location(4, 0, 5, 3);
widget Show(g(I)),   parent <- ‘myWin, Location(0, 0, 3, 3);
widget Show(h(I)),   parent <- ‘myWin, Location(0, 3, 3, 6);
widget Print(a, b),  parent <- ‘myWin, Location(3, 3, 6, 6);
```

Figure 17: Example of Window Widget. Several objects are placed inside a window.

format	meaning	example
"*color*"	solid	"green"
"*coord*:*color*->*color*"	indexed	"4:green->#2ff"

Table 14: Object Widget Color String Formats.

9 Conclusion

[[the usual]]

10 Acknowledgments

I spent many months talking to and designing with Nick Thompson (nix@sgi.com). His influence is strong, and indeed, many of these ideas are his. Nick Thompson also designed and implemented the parser.

Matthew Stone (mas@cs.brown.edu) implemented the Motif interface, hidden surface removal, and PostScript output.

This work would not be possible without the support and funding of Thomas Banchoff, Andy Witkin, and Dana Scott, or the hospitality and equipment of the Computer Graphics Group at Brown.

The Xt part of the bitmap widget was written by Brian Totty (totty@cs.uiuc.edu). His code's copyright requires that the following notice be included in this manual:

Contents

1 Introduction **1**
- 1.1 What Fnord is Not 1
- 1.2 What Fnord Is 1
- 1.3 Typographical Conventions . . 1

2 Overview **1**
- 2.1 Syntax 2
- 2.2 Simple Examples 2

3 Syntax **3**
- 3.1 Comments 3
- 3.2 Literals 3
  - 3.2.1 Identifiers 4
  - 3.2.2 Integers 4
  - 3.2.3 Reals 4
  - 3.2.4 Strings 4
- 3.3 Expressions 4
- 3.4 Built-In Functions 5

4 Values and Types **6**
- 4.1 Bare 6
- 4.2 Sets 6
  - 4.2.1 Blocks 8
  - 4.2.2 Connectivity 8
- 4.3 Symbols 8
- 4.4 Type Values 8

4.5 Types of Values 9

5 The Operators 9
5.1 Fundamental Operators 10
5.1.1 Lambda 10
5.1.2 Apply 10
5.1.3 Cast 11
5.2 Constructors and Destructors . 13
5.2.1 Pairs 13
5.2.2 Sets 13
5.2.3 Vectors 13
5.2.4 Complexes and Booleans 14
5.3 Logical 14
5.4 Relational 14
5.5 Conditional 15
5.6 Arithmetic 15
5.7 Scalar Math Functions 15
5.8 Differentiation 15
5.9 Set Operations 16
5.10 Associations 17
5.11 Reduce 17
5.12 Quote 18
5.13 Enumerate 18
5.14 Bitwise Operators 18
5.15 Polynomial 18
5.16 Type Operators 19
5.16.1 The Type Operator . . 19
5.16.2 Type Matching Operators 19
5.17 Casting 19

6 The Interpreter 20
6.1 Type Inference 20
6.2 Parallelism 20
6.3 Performance 21
6.4 Input 21
6.5 Commands 21
6.6 Directives 21

7 Visualization System 21
7.1 Input Widgets 22
7.1.1 Checkbox Widget . . . 22
7.1.2 Button Widget 22
7.1.3 Slider Widget 22
7.1.4 Type-in Widget 22
7.1.5 Tapedeck Widget 22
7.1.6 Bitmap Widget 22
7.1.7 Point Widget 24
7.2 Output Widgets 24
7.2.1 Print Widget 24
7.2.2 View Widget 25
7.3 Text Widget 26
7.4 Evolution Widget 26
7.5 Output Widget 27
7.6 Window Widget 27
7.7 Object Widget 27

8 The Standard Library 27

9 Conclusion 29

10 Acknowledgments 29