# A Data Language

Thant Tessman
January 17, 2009

## Overview

This document describes a data format designed to provide computer application developers a convenient, standardized, text-based, human-readable, easy-to-parse means of exchanging data between different applications, or between different invocations of a single application. DL's approach to the problem is unique in that the format avoids assuming any application-specific *semantic* interpretation of the data. Instead, DL limits itself to describing the *structure* of the data. In a sense, DL is a programming language with support for typed data structures, but without support for control structures.

# Contents

# Introduction

### *Motivation*

DL was originally conceived of as the next evolutionary step in computer graphics file formats for describing 3D geometry and associated scene data. Previous efforts have illuminated a tension between flexibility and efficiency. For example, geometric vertex position data often—but not always—takes the form of homogenous arrays of 3-dimensional coordinates. The question is: How can a format allow for the efficient parsing into memory of data structures like vertex arrays without relying on *semantic* assumptions to tell us how data is to be formatted? The answer is to allow for type declarations analogous to type declarations in a programming language like C. And like C, DL supports a small set of atomic data types, plus a small number of ways to assemble the data into larger structures.

DL's type system differs from C's in a few ways, but the most important difference is that in DL the type declarations are optional. This gives DL the feel of a dynamically-typed scripting language when type declarations are unnecessary or inappropriate.

### *How DL Compares to XML*

At a higher level, DL might best be understood by comparing it to XML (Extensible Markup Language) which purports to serve somewhat the same purpose. XML evolved as an almost backwards-compatible extension of HTML (HyperText Markup Language). Like DL, XML is intended to describe many different kinds of data (packaged as a *document*), and to allow programs to modify and validate documents without prior knowledge of their form.

The problem is that XML is, at its core, merely *tagged* text. To verify the structure of the data and to parse numerical (or other) data elements within the text, applications need meta information about a document in the form of Document Type Definitions and XML *schema*. The complex nature of this meta information (along with XML's generally inconvenient syntax) tends to poison the supposed application-neutral advantages of XML.

### *Design and Audience*

The design of DL attempts to avoid being gratuitously novel. DL is mainly an assemblage of a small number of ideas gathered from various programming languages and culled by experience. DL's syntax for identifiers, numbers and strings is lifted directly from the C programming language, for example. But it is the hope that the ability to read and edit a DL file is not restricted to C programmers in particular or programmers in general.

This document is divided into three sections. The first will informally describe all the elements of DL using examples. The second will more formally describe its syntax and type system. The third will discuss implementation details and strategies.

# Tutorial

The purpose of DL is to store and communicate values, organize them into structures, and to support optional type declarations. The type declarations make it possible to mechanically verify the data is organized as required by the needs of a given application. Type declarations also facilitate efficient parsing of the data into memory. DL also supports references. These allow one part of a data structure to refer to data or type declarations in another part of the data structure, effectively sharing that data across the data structure.

## Atomic Types

The language supports values of a small, general set of atomic types: *character*, *integer, real*, and *symbol*. (Values of these types have a syntax borrowed (with minor qualification) from the analogous types in the C programming language.)

### *Characters*

A character is an individual letter, digit, punctuation, space, etc. Characters are written using single quotes:

```
'x'
```

As in the C programming language, special characters such as tab and newline can be represented using escape sequences. An escape sequence is a backslash followed by something specifying the special character. For example, a tab can be specified like so:

```
'\t'
```

The single quote itself can be specified like so:

```
'\''
```

A complete description of escape sequences can be found in the formal specification.

The character *type signature* is specified with the keyword:

```
char
```

More on how to put type signatures to use later.

### *Numbers*

DL distinguishes between numbers of type integer, and numbers of type real. If a number contains a decimal point or an exponent (or both), it is considered a real. If it is otherwise just a string of decimal digits, it is an integer.

Exponents are specified using `e` or `E`. Here is Avogadro's number:

```
6.022e23
```

(A number of type real borrows its syntax from C's `double`. But unlike C, DL's integer syntax supports neither octal, nor hexadecimal notation.)

DL does not specify the precision with which numbers are represented. This is entirely implementation-dependent. (Typically, DL's integer type is expected to correspond to C's native `int` and a real to `float`. But an implementation of DL exists now that uses a fixed-point number type to represent reals.)

The integer number type signature is specified with the keyword:

```
int
```

The real number type signature is specified with the keyword:

```
real
```

### *Symbols*

A *symbol* is a named, globally-unique enumerant. Symbols are like read-only strings that store and compare in memory as integers. Lexically, a symbol starts with an octothorpe (#) and is followed by a C-style identifier. A C identifier is made up of letters and digits. The first character must be a letter. The underscore "_" counts as a letter. Upper and lower case are different.

Symbols are borrowed from the Scheme programming language, but they bear some resemblance to C's notion of an enumerant. A DL file might use symbols to denote boolean values, for example:

```
#false
#true
```

The type signature for symbols is specified with the keyword:

```
sym
```

# Structures

The format also supports two structures to assemble values into larger conglomerations: *record*, and *vector*. Records are unordered collections of values indexed by name. Vectors are ordered collections of values indexed by a nonnegative integer. Records and vectors can be arbitrarily nested.

(Records are roughly analogous to structures in the C programming language, and vectors are roughly analogous to C's arrays. Following the convention in C, the index is an offset from the beginning of the vector. That is, an index of zero refers to the first item in the vector. More on indexing later.)

### *Strings*

Vectors have their own specific syntax which is described below. However, for convenience, vectors of characters (i.e. *strings*) are supported directly with their own C-like double-quote syntax. Like characters, strings include support for the translation of backslashed special characters like *tab* (`\t`) and *newline* (`\n`). Also, consecutive strings separated only by whitespace are concatenated when parsed to form a single string in memory.

```
"This is a string."
```

### *Records and Vectors*

As mentioned, records are collections of values indexed by name. Names borrow their lexical form from C's identifiers (described earlier in the section on symbols). A single DL file corresponds to a single record and is, at the top level, merely a sequence of named values (i.e. bindings). Example:

```
pi = 3.14159265
number_of_feet = 2
fortune = "A witty saying proves nothing. --Voltaire"
true_or_false = #true
```

When read in, this will create a record containing four bindings. A binding consists of three parts: a name (implemented as a symbol), a value, and a type constraint. Type constraints will be described more fully later. For now, all that's important to know is that the above example specifies no type constraints. The effect is that the type constraints for each of the four bindings default to the type signature:

```
any
```

The following example illustrates the syntax of records and vectors:

```
colors = {
  red = [1, 0, 0]
  orange = [1, 0.5, 0]
  yellow = [0.9, 0.8, 0.1]
}
```

This creates a single binding named *colors* which is a record containing three vectors named *red*, *orange*, and *yellow*. Vectors are a sequence of values within a pair of square brackets and separated by commas. (It's not an error if the last element is followed by a comma.) Records are a sequence of bindings (name/value pairs) within a pair of curly brackets. Although a single DL file itself corresponds to a single record, the curly brackets for this top-level record are omitted since they are implied.

# References

A value can refer to a previously-bound value by name. More than that, a value can refer to values within a previously defined record or vector using a C-like notation. Record bindings are accessed using dot-member notation, and vectors are indexed using square-bracket notation. References must begin with a dollar sign. Given the previous definition for *colors*, the following is valid:

```
paint_color = $colors.orange
green_part_of_yellow = $colors.yellow[1]
```

Bindings are lexically scoped. That is, a reference doesn't have to refer only to a binding at the top level. It can start with any binding defined in any outer record as long as it precedes the binding currently being defined. The following associates the binding name `president` inside the `info` record with the string value `"Moe"`.

```
info = {
  names = ["Larry", "Curly", "Moe"]
  president = $names[2]
}
```

This, however, will not work:

```
info = {
  names = ["Larry", "Curly", "Moe"]
  president = $info.names[2]
}
```

This because `info` is the binding currently being defined. This restriction has the effect of precluding circular references.

It is an error for a set of binding names within a single record to contain duplicates, but if one record contains another record, each is allowed to have its own binding of a given name. A reference is resolved by first looking in the innermost record that contains the reference, and continuing the search in each subsequently encompassing record.

# Type Constraints

Bindings within a record can optionally be typed. In other words, the value bound to a given name can be checked for conformance to a specific form. (It is also possible to check an entire DL file against an externally-specified type constraint.) As mentioned previously, the type declarations make it possible to mechanically verify the data is organized as required by the needs of a given application, as well as facilitate efficient parsing of the data into memory.

The syntax for type constraints is:

```
name : type = value
```

As hinted at previously, DL uses a few keywords for describing basic types: `char`, `int`, `real`, `sym`, and `any`. Record and vector types can also be described. DL also provides a way to describe a type that restricts a value to a specific set of symbols. The last is called an *enumeration*.

### *Vector Type*

The keywords `vec1`, `vec2`, `vec3` and so on can be used to specify vectors of fixed size. The keyword `vec` (without the number) is used to specify a vector of unspecified size. A `vec`, `vec1`, etc. keyword must be followed by the type of the objects to be stored in the vector. For example, the following specifies a vector of unspecified length containing length-3 vectors of reals:

```
vec vec3 real
```

### *Record Type*

A record type declaration specifies that a value must be a record. It also specifies what bindings of what types a record must contain. The syntax is:

```
rec {
  name1 : type1
  name2 : type2
  ...
}
```

A record type only specifies the bindings that must be present. A record is allowed to have more bindings than the record type specifies. This means for example that

```
rec {}
```

will successfully type check against any record. The bindings are not required to be in the same order as specified in the record type. These two aspects of records are intended to make it possible to extend them in a way that doesn't break them for applications that aren't expecting the extended versions. It allows an application to find the bindings it is looking for without the need to know about the bindings it isn't looking for.

This is the colors example rewritten with type declarations:

```
colors = {
  red : vec3 real = [1, 0, 0]
  orange : vec3 real = [1, 0.5, 0]
  yellow : vec3 real = [0.9, 0.8, 0.1]
}
```

With the above type constraints, the type checker will verify that the values bound to `red`, `orange`, and `yellow` will be length 3 vectors of reals. (More than that, an implementation of DL now has the information it needs to store those values as packed arrays instead of as untyped indirections to type-tagged data.)

Note that an integer will successfully type check as a value of type real.

The colors binding can also be written as such:

```
colors : rec {
  red : vec3 real
  orange : vec3 real
  yellow : vec3 real
} = {
  red = [1, 0, 0]
  orange = [1, 0.5, 0]
  yellow = [0.9, 0.8, 0.1]
}
```

The type declaration will serve to guarantee that the record bound to `colors` contains bindings named `red`, `orange`, and `yellow`, and that the values bound to them are length-3 vectors containing reals (or integers).

### *Type Declarations*

A type declaration gives a type a name. A type declaration does not introduce a new type. It merely provides an alias for a type that would otherwise be described using the various type

keywords. A type declaration is introduced within a record (including the top-level record) using the `type` keyword.

An example using a record type:

```
type shape = rec {
  points : vec vec3 real
  normals : vec vec3 real
  size : real
}

my_obj : $shape = {
  points = [[2, 3, 4], [5, 6, 7]]
  normals = [[8, 9, 10], [11, 12, 13]]
  size = 432.1
}
```

(A real DL file would of course specify many many more points and normals, along with polygon index information, and so on.)

Type declarations are lexically scoped in the same manner that binding names are. They can be referenced using the same dot and square-bracket notation, and they share the same namespace. Type declarations can be arbitrarily intermixed with binding declarations within a record (as long as type declarations occur before they are referenced).

### *Enumeration*

An enumeration is a type that restricts a value to one of a finite set of symbols. The prototypical example of an enumerated type is the boolean:

```
type bool = enum {#true #false}
true_or_false : $bool = #true
```

### *The* `any` *Type*

The `any` type is available within DL via the `any` keyword. If no type is specified at a record binding, it defaults to the `any` type. Consequently, the `any` keyword is redundant on its own, but it is useful within the description of compound types. For example, the type of a vector that should be allowed to contain items of any type can be described with:

```
vec any
```

And a record that should contain a binding of a given name, say `fu`, but whose value is otherwise unspecified can be described with:

```
rec {fu: any}
```

# Formal Specification

Tokens used in the DL grammar are described here using traditional Unix regular expressions. Only the tokens that require regular expressions are included in this section. For clarity, literal tokens are left as literal tokens in the section that describes the grammar. The regular expressions *escseq* and *validchar* (not in bold) are not tokens, but merely supplementary regular expressions. Note also that naturals and reals don't allow for a leading minus sign because this is implemented as a unary negative operator in the parser.

## Tokens

***identifier***:

```
[a-zA-Z_][a-zA-Z0-9_]*
```

***symbol***:

```
#{identifier}
```

***natural***:

```
[0-9]+
```

***real***:

```
(([0-9]+(\.[0-9]+)?)|(\.[0-9]+))(eE][+-]?[0-9]+)
```

***vecN***:

```
vec{natural}
```

***path***:

```
${identifier}((\.{identifier})|(\[{natural}\]))*
```

*escseq*:

```
\\([abtnvfr\\\"\']|[0-7]{3,3}|(x[0-9a-fA-F]{2,2}))
```

*validchar*:

```
[^\n\\\"]|{escseq}
```

***char***:

```
\'({validchar}|\")\'
```

***string***:

```
\"({validchar}|\')*\"
```

### *Escape Sequences*

| | |
|---|---|
| **\n** | newline |
| **\t** | tab |
| **\b** | backspace |
| **\r** | carriage return |
| **\f** | form feed |
| **\a** | alert (bell) |
| **** | backslash |
| **\'** | single quote |
| **\"** | double quote |
| *****ddd* | octal character number |
| **\x***hh* | hexadecimal character number |

The escape *****ddd* consists of a backslash followed by exactly 3 octal digits which are taken to specify the value of the desired character. The escape **\x***hh* consists of a backslash, a literal **x** and exactly two hexadecimal digits taken to specify the value of the desired character.

# Grammar

Tokens described in the previous section are in ***italicized bold***. Literal tokens are in **`courier bold`**. Literal tokens used in the grammar include the following set of symbols:

**`[ ] { } : = , -`**

The grammar for a regular DL file uses the *bindings* production as its *start* symbol and produces a record, but a DL type specification against which one would type check a DL file uses *bindingtypes* as its *start* symbol and produces a record type. Beyond that, the latter grammar is a strict subset of the former.

*start:*
    *bindings*

*bindings:*
    *bindings binding*
    *binding*

*binding:*
    ***identifier*** **`:`** *type* **`=`** *data*
    ***identifier*** **`=`** *data*
    **`type`** ***identifier*** **`=`** *type*

*elems:*
    *elems* **`,`** *data*
    *data*

*bindingtypes:*
    *bindingtypes bindingtype*
    *bindingtype*

*bindingtype:*
    ***identifier*** **`:`** *type*

*enum:*
    *enum* ***symbol***
    ***symbol***

*strings:*
    *strings* ***string***
    ***string***

*type:*

**any**
**char**
**int**
**real**
**sym**
**vec** *type*
***vecN*** *type*
**rec {** *bindingtypes* **}**
**enum {** *enum* **}**
***path***
***symbol***

*data:*

***char***
***natural***
**-** ***natural***
***real***
**-** ***real***
*strings*
***symbol***
**[ ]**
**[** *elems* **]**
**[** *elems* **, ]**
**{ }**
**{** *bindings* **}**
***path***

# Type Checking

Every record binding includes a type constraint (which, if unspecified, defaults to the `any` type). It is also practical to check an entire DL file against an externally-specified type constraint. This type constraint could itself be a text file. Its grammar would be identical to that of a DL file with two exceptions. The first is that the grammar's *start* symbol would be *bindingtypes* instead of *bindings*. The second is that it wouldn't contain any references because there are no bindings to which to refer.

Type checking in DL is complicated by the need to check nested types, and by the fact that data is allowed to be more specific than the type signature that constrains it.

The task of implementing the DL type system can be divided amongst three functions. The first:

getType (*value*) ⇒ *type*

converts a value to its most specific type signature. The second:

commonType (*type*, *type*) ⇒ *type*

takes two type signatures and finds the most specific common type. That is, it combines the two type signatures into a new type signature. Any data that will successfully type check against both of the originally supplied type signatures will also successfully type check against the new type signature. The commonType function is needed to deduce a vector's type signature from the type signatures of its elements.

The third function:

isa (*type*, *type*) ⇒ *bool*

checks if the first type is a subtype of the second type. The term *subtype* is used here in its original *substitutability* sense. That is, any data that successfully type checks against the first type will type check against the second, but not necessarily vice versa. An *integer* is a *real*, but a *real* is not an *integer*.

It is possible to collapse the duties of these three functions into a single function:

isa (*value*, *type*) ⇒ *bool*

which compares a value directly to a type signature. Some applications of DL will no doubt require nothing beyond this simpler approach. But some applications will want to guarantee not only that data matches the type signature explicitly stated at a given binding, but that the type signature itself matches the type signature that the application expects. Using the first three functions as the foundation of the type system makes this practical, and is not much more work to implement than the fourth function alone.

There is one more useful function on types:

$$\text{specificType}\ (type, type) \Rightarrow type$$

This returns a type that will successfully type check against the two types provided as arguments. That is, given types *a*, *b*, and *c* such that:

$$\text{specificType}\ (a, b) \Rightarrow c$$

then it is guaranteed that:

$$\text{isa}\ (c, a) \Rightarrow true$$

and

$$\text{isa}\ (c, b) \Rightarrow true$$

It is possible that there is no overlap between the types *a* and *b*. For example, there is no type that is both a character and a record. In such a case, the specificType function will return the special type **`none`**. (The special **`none`** type is not merely an error code. It is a type in its own right, and is described in more detail in the next section.)

The specificType function returns the intersection of two types. The commonType function returns the union of two types. Given types *a*, *b*, and *c* such that:

$$\text{commonType}\ (a, b) \Rightarrow c$$

then it is guaranteed that:

$$\text{isa}\ (a, c) \Rightarrow true$$

and

$$\text{isa}\ (b, c) \Rightarrow true$$

There is a useful DL data structure transformation that requires the specificType function. Recall the earlier example:

```
type shape = rec {
  points : vec vec3 real
  normals : vec vec3 real
  size : real
}

my_obj : $shape = {
  points = [[2, 3, 4], [5, 6, 7]]
  normals = [[8, 9, 10], [11, 12, 13]]
  size = 432.1
}
```

The problem with the data in this form is that the information necessary to parse the points and normals data directly into the appropriate homogenous arrays is provided in a way that makes it difficult for a traditional parser to take advantage of.

What we would like is a utility that took complex hierarchical type constraints and mapped them down the data structures to which they apply, effectively *flattening* the type constraints. So, for example, the previous data structure after flattening would look like:

```
type shape = rec {
  points : vec vec3 real
  normals : vec vec3 real
  size : real
}

my_obj : $shape = {
  points : vec vec3 real = [[2, 3, 4], [5, 6, 7]]
  normals : vec vec3 real = [[8, 9, 10], [11, 12, 13]]
  size : real = 432.1
}
```

The type constraints in the `$shape` record type get mapped to the corresponding bindings in the `my_obj` record. This is useful because its easy to build a parser that knows how to parse the points and normals data directly into the described homogenous arrays.

A binding lower down in a data structure might already have a type constraint. Consequently, to do its job in the most general way possible, a type flattening function will need to, in effect, apply more than one type constraint at a given binding. This is where the specificType function is used. It takes two types and returns the type that is effectively the same thing as applying both types. (If no such type exists, there is a type error in the data structure anyway. The flatten function would simply make the type error explicit by producing a binding with a type constraint of **`none`**.)

### *Type*

Type signatures as data structures almost exactly mirror their manifestation in the grammar. The only significant difference is the addition of a **`none`** type. This is the type for which every value will fail to type check. It is useless within the DL grammar, but it is an important part of the type checking algorithm.

*type:*
- **`any`**
- **`none`**
- **`char`**
- **`int`**
- **`real`**
- **`enum`** of *symbol ...*
- **`rec`** of (*symbol, type*) ...
- **`vec`** of *type*
- **`vecN`** of (*int, type*)

The ellipses signify zero or more occurrences.

### *The* getType *Function*

***character*** ⇒ **`char`**

***integer*** ⇒ **`int`**

***real*** ⇒ **`real`**

***symbol*** ⇒ **`enum`** of ***symbol***

***record*** of ***(symbol, value)*** ... ⇒ **`rec`** of ***(symbol,*** getType ***(value))*** ...

***vector*** of ***value*** ... ⇒
*set* ***count*** *to the number of elements in the vector*
*initialize* ***type*** *to* **`none`**
*for every* ***value*** *do*
*set* ***type*** *to* commonType ***(type,*** getType ***(value))***

*return* **`vecN`** of ***(count, type)***

The ellipses indicate zero or more occurrences. A record's type has nothing to do with any type declarations or type constraints it contains. It is a function of the binding names and values only. An empty vector has an element type of **`none`**. The **`none`** type doesn't check successfully against any other type, including itself.

### *The* commonType *Function*

Each type rule listed takes precedence over the rules that are listed after it. The underscore is used as a place marker for items that are otherwise ignored.

*(***none***, type)* ⇒ *type*
*(type,* **none***)* ⇒ *type*

*(***char***,* **char***)* ⇒ **char**
*(***int***,* **int***)* ⇒ **int**
*(***real***,* **real***)* ⇒ **real**

*(***int***,* **real***)* ⇒ **real**
*(***real***,* **int***)* ⇒ **real**

*(***sym***,* **sym***)* ⇒ **sym**

*(***sym***,* **enum** of *_)* ⇒ **sym**
*(***enum** of *_,* **sym***)* ⇒ **sym**

*(***enum** of ***a**,* **enum** of ***b**)* ⇒ **enum** of *the union of the symbol sets* ***a*** *and* ***b***

*(***vec** of ***a**,* **vec** of ***b**)* ⇒ **vec** of commonType *(**a**, **b**)*

*(***vecN** of *(_, **a**),* **vec** of ***b**)* ⇒ **vec** of commonType *(**a**, **b**)*
*(***vec** of ***a**,* **vecN** of *(_, **b**))* ⇒ **vec** of commonType *(**a**, **b**)*

*(***vecN** of *(**i**, **a**),* **vecN** of *(**j**, **b**))* ⇒
    *if* ***i***=***j***
    *then return* **vecN** *of (i,* commonType *(**a**, **b**))*
    *else return* **vec** *of* commonType *(**a**, **b**)*

*(***rec** of ***a**,* **rec** of ***b**)* ⇒
    *initialize* ***bindingtypes*** *to the empty set*
    *for every (**symbol1**, **type1**) in **a** do*
        *for every (**symbol2**, **type2**) in **b** do*
            *if* ***symbol1***=***symbol2***
            *then add (**symbol1**,* commonType *(**type1**, **type2**)) to* ***bindingtypes***
    *return* **rec** of ***bindingtypes***

*(_, _)* ⇒ **any**

### *The* specificType *Function*

Each type rule listed takes precedence over the rules that are listed after it. The underscore is used as a place marker for items that are otherwise ignored. The description provided below of how to find the specific types of two record types is merely meant to be explicit about what needs to be done. Other semantically equivalent but more efficient implementations are possible.

*(***any***, type)* ⇒ *type*
*(type,* **any***)* ⇒ *type*

*(***rec** of ***a**, **rec** of ***b**)* ⇒
    *initialize* ***bindingtypes*** *to the empty set*
    *for every (**symbol1**, **type1**) in **a** do*
        *if (**symbol2**, **type2**) in **b** exists such that **symbol1**=**symbol2***
        *then add (**symbol1**,* specificType *(**type1**, **type2**)) to **bindingtypes***
        *else add (**symbol1**, **type1**) to **bindingtypes***
    *for every (**symbol2**, **type2**) in **b***
        *if a binding with **symbol2** hasn't already been added to **bindingtypes***
        *then add (**symbol2**, **type2**) to **bindingtypes***
    *return* **rec** of ***bindingtypes***

*(***vecN** of **(i, a)**, **vecN** of **(j, b))*** ⇒
    *if* ***i***=***j***
    *then return* **vecN** *of (i,* specificType *(**a**, **b**))*
    *else return* **none**

*(***vecN** of *(size, **a**)*, **vec** of ***b**)* ⇒ **vecN** of *(size,* specificType *(**a**, **b**))*
*(***vec** of ***a**, **vecN** of *(size, **b**))* ⇒ **vecN** of *(size,* specificType *(**a**, **b**))*

*(***vec** of ***a**, **vec** of ***b**)* ⇒ **vec** of specificType *(**a**, **b**)*

*(***enum** of ***a**, **enum** of ***b**)* ⇒ **enum** of *the intersection of the symbol sets **a** and **b***

*(***sym**, **enum** of *symbol ...)* ⇒ **enum** of *symbol ...*
*(***enum** of *symbol ...,* **sym***)* ⇒ **enum** of *symbol ...*

*(***sym**, **sym***)* ⇒ **sym**

*(***int**, **real***)* ⇒ **int**
*(***real**, **int***)* ⇒ **int**

*(***char**, **char***)* ⇒ **char**
*(***int**, **int***)* ⇒ **int**
*(***real**, **real***)* ⇒ **real**

*(_, _)* ⇒ **none**

### *The* isa *Function*

Each type rule listed takes precedence over the rules that are listed after it. The underscore is used as a place marker for items that are otherwise ignored.

*(_,* **`any`***)* ⇒ ***true***

*(***`char`***,* **`char`***)* ⇒ ***true***
*(***`int`***,* **`int`***)* ⇒ ***true***
*(***`int`***,* **`real`***)* ⇒ ***true***
*(***`real`***,* **`real`***)* ⇒ ***true***
*(***`sym`***,* **`sym`***)* ⇒ ***true***
*(***`enum`** of _, **`sym`***)* ⇒ ***true***

*(***`enum`** of **a**, **`enum`** of **`b`***)* ⇒
    *for every* ***symbol*** *in* ***a*** *do*
        *if* ***symbol*** *is not contained in* ***b***
        *then return* ***false***
    *return* ***true***

*(***`vec`** of **a**, **`vec`** of **b**)* ⇒ isa *(**a**, **b**)*
*(***`vecN`** of *(_, **a**)*, **`vec`** of **b**)* ⇒ isa *(**a**, **b**)*
*(***`vecN`** of *(**i**, **a**)*, **`vecN`** of *(**j**, **b**))* ⇒
    *if* ***i≠j***
    *then return* ***false***
    *else return* isa *(**a**, **b**)*

*(***`rec`** of **a**, **`rec`** of **b**)* ⇒
    *for every (**symbol2**, **type2**) in* ***b*** *do*
        *find (**symbol1**, **type1**) in* ***a*** *such that* ***symbol1***=***symbol2***
        *if no such binding is found then return* ***false***
        *if* isa *(**type1**, **type2**) returns false then return* ***false***
    *return* ***true***

*(_, _)* ⇒ *return* ***false***

# Implementation

DL was designed not only to be relatively easy to implement, but to allow for multiple implementation strategies and application-specific optimizations. The decision, for example, to disallow recursive references was made not because it necessarily breaks things semantically, but because it precludes certain optimization strategies. Ditto the decision not to put type declarations and bindings in separate namespaces. The grammar makes it possible to do this, but it is important to allow the lexer to resolve a path and return a token based on the type or value that the path refers to. The lexer doesn't know whether it is looking for a type or a value until it finds it.

Anyone who intends to implement DL must be comfortable with the notion of a variant (variant record, tagged union, disjoint union, discriminated union, algebraic data type, ...). There is plenty of literature available on the topic. (In particular, see the *variant* library provided at http://www.boost.org.) Beyond that, the main decision an implementor needs to make is how much work they wish to put into leveraging type declarations to efficiently parse data directly into application-specific in-memory data structures. This section addresses that issue, but also continues with issues that implementors aren't likely to encounter unless they are really pushing on DL.

### *Parsing Optimizations*

In the general case, an implementation must support DL values with some manifestation of a *variant* (possibly in the form of a tagged union, or algebraic data type). Unfortunately this can be unacceptably inefficient in the case of very large arrays of homogenous data.

A type constraint can provide a parser with the information it needs to parse such data structures directly. If a parser sees:

```
points : vec vec3 real =
```

it can legally assume that what follows has a grammar that looks like:

*start:*
    `[` *vec3rs* `]`
    `[` *vec3rs* `,` `]`
    ***path***

*vec3rs:*
    *vec3rs* `,` *vec3r*
    *vec3r*

*vec3r:*
    `[` *item* `,` *item* `,` *item* `,` `]`
    `[` *item* `,` *item* `,` *item* `]`
    ***path***

*item:*
    ***int***
    ***real***
    ***path***

### *Parsing and References*

The situation is complicated by the fact that the type constraint or even parts of it might be specified using a reference to a type declaration. Also, as the grammar above illustrates, the raw data itself is allowed to contain reference paths.

Paths begin with a dollar sign so that the lexer can distinguish them from identifiers. This allows (but doesn't require) a path to be resolved in the lexing stage so that the lexer can return the corresponding data element or type token directly. This way, the presence of references doesn't sabotage the attempt to build specific type signatures and data structures directly into the grammar.

In the above example grammar for `vec vec3 real`, the path tokens can be replaced entirely with tokens corresponding to values of the type expected at that point in the grammar.

### *Transcription and References*

Much of the utility of DL is derived from the ability to conveniently convert back and forth between raw text and data that has been massaged into an application-specific form to one degree or another. Here too the situation is complicated by references. Resolving reference paths is relatively straight-forward, and every application that has the ability to read a DL file should be expected to deal with a reference everywhere they are legally allowed to occur. Generating reference paths when writing a file, on the other hand, is trickier.

One can imagine (and this writer has built) a utility that converts back and forth between a DL text file and an efficiently-packed, fast-loading binary file. In this case, if one is to avoid the need for application-specific knowledge about where references will occur, the desire to efficiently pack vectors of reals as arrays of floats conflicts with the goal of preserving references across the translation boundary. It can be done, but it unreasonably complicates the implementation. Better to restrict the preservation of references to the references that point at records and vectors, which are likely to be implemented (directly or indirectly) as pointers anyway.

### *References as Explicit Structures*

To generate references when transcribing a raw, in-memory DL data structure, one approach is to keep track of a dictionary of which records and vectors have already been transcribed, along with the sequences of binding names and vector indices by which they can be reached (i.e. the reference path). Then, before transcribing a new record or vector, first check to see if it has already been transcribed, and if so, just use the associated reference path instead.

It can get expensive to compare every (pointer to a) record or vector about to be transcribed with every (pointer to a) previously transcribed record and vector. This is not as bad as keeping track of every single value that has been transcribed so far, but it is still not ideal.

Another approach is to introduce an explicit data-reference type as part of the value-variant implementation, and a type-reference type as part of the type-variant implementation. In this approach, values and types are only recognized as being shared across the data structure if they are shared through these references. This approach requires that the resolution of references be divided into several passes:

1. Parse the data, leaving references unresolved (at least the references you want to preserve).
2. Collect the paths to every value and type referenced from somewhere else.
3. Replace each referenced value and type with actual references to the associated value or type.
4. Resolve the reference paths. These will now resolve not to values or types, but references to values or types.

This is several passes, but they are all O(n) instead of $O(n^2)$. And at transcription time, only references need to be checked against the dictionary.