KIDL Specification

Version 0.1 Author Robert Olson

Type Description Language

We define a small language that we will use to specify the types of data objects passed through the KBase API.
The type language is built from a small number of basic types. These include three scalar types:

- *string*, a sequence of ASCII characters
- *integer*, a signed integer
- *float*, a floating point number

and four compound types:

- *list*, a homogeneous ordered sequence of values
- *mapping*, a type mapping from a scalar or scalar-derived type to a value of an arbitrary type. All values in a mapping have the same type.
- *structure*, a record type that contains one or more values which can be of differing types. Each value is accessed by a name that is defined by the type.
- *tuple,* a record type that contains one or more values. The values are accessed by index (think of this as a fixed-length list where the values may be of differing types).

If we wish to define a list of integer values, we would use the type string
list<int>

A type string that denotes a mapping from a string to another string is
mapping<string, string>

A struct that contains a string id and an integer value is
structure { string id; int value; }

A tuple containing a string and an integer is
tuple<string, int>

Types can be nested in a natural way. A type that maps an integer to a list of strings is
mapping<int, list<string> >

New type names may be defined using the *typedef* construct:
typedef type-description new-type-name;

For example, to create a new type that defines a list of integers, and a record type that includes the list of integers, we write

```
typedef list<int> IntList;
typedef structure { string key; IntList value_list; } TimeSeries;
```

A type definition may optionally impose a validation constraint on the objects of that type. Syntax for this constraint is not yet proposed.

Function Definitions

The API specification language uses the type definitions defined above to specify the functions that comprise the API:

```
funcdef function-name(parameters) returns (return-parameters);
```

A function may have zero or more parameters. Each parameter is specified as a type and an optional parameter name.
For instance, a function that takes as input a list of feature identifiers and returns the current functions assigned to these features as a hash mapping from the feature identifier to the function may be defined as follows:

```
typedef FeatureId string;
funcdef get_feature_functions(list<FeatureId>)
        returns (mapping<FeatureId, string>)
```