Semantic Design Language, Sedela

To specify semantic designs generally, I've created a meta-language called SEDELA (for Semantic Design Language). First, we present the definition of SEDELA, then an example semantic design for MetaFunctions (a replacement for Microservices) in terms of SEDELA. Although I may aim to write a parser and type-checker for SEDELA, there will never be a compiler or intepreter. Thus, SEDELA will have no syntax for **if** expressions and the like. The only Meanings (SEDELA's nomenclature for functions) defined in the Prelude will be combinators such as id, const, flip, and etc. SEDELA's primitive types are all defined in terms of Axioms (types without formal definitions) with no available operations.

Sedela Language Definition

```
Axiom :=              Axiom[!] "Informal definition."                     where ! denotes intended effectfulness

Meaning Type :=       A -> ... -> Z                                       where A ... Z are Type Expressions

Meaning Defn :=       f (a : A) ... (z : Z) : R = Expression | Axiom      where f is the Meaning Identifier
                                                                          and a ... z are Parameter Identifiers
                                                                          and A ... Z, R are Type Expressions

Expression :=         Example: f a (g b)                                  where f and g are a Meaning Identifiers
                                                                          and a and b are Paremeter Identifiers

Product :=            MyProduct<...> = A | (A : A, ..., Z : Z) | Axiom    where MyProduct<...> is the Product Identifier
                                                                          and A ... Z are Field Identifiers
                                                                          and A ... Z are Type Expressions

Sum :=                MySum<...> =                                        where MySum<...> is the Sum Identifier
                        | A of (A | Axiom)                                and A ... Z are Case Identifiers
                        | ...                                             and A ... Z are Type Expressions
                        | Z of (Z | Axiom)

Type Identifier :=    Product Identifier | Sum Identifier

Type Expression :=    Meaning Type | Type Identifier

Type Parameters :=    Type Identifier<                                    where A ... Z are Type Expressions
                        A, ..., Z;                                        and A ... Z are Category Identifiers used for
                        A<A, ..., Z>; ...; Z<...>>                        constraining A ... Z

Category :=           category MyCat<...> =                               where MyCat<...> is the Category Identifier
                        | f : A                                           and f ... g are Equivilence Identifiers
                        | ...                                             and A ... Z are Types Expressions
                        | g : Z

Witness :=            witness A =                                         where A is a Category Identifier
                        | f (a : A) ... (z : Z) : R = Expression          and f ... g are Equivilence Identifiers
                        | ...                                             and a ... z are Parameter Identifiers
                        | g (a : A) ... (z : Z) : R = Expression          and A ... Z, R are Type Expressions

Categorization :=     Rule: iff type A has a witness for category A, A is allowable for type parameter categorized as A
```

Line Comment :=	**Example:** // comment text
fun *a* *b* ... *z* -> *expr* **:=**	\a (\b (... \z.*expr*))
a **->** *b* **:=**	_ = (_ : *a*) : *b*
() **:=**	**Explanation:** The unit type / value.
f . g **:=**	**Explanation:** Function composition.

Sedela Language Prelude

```
Bool = Axiom "A binary type."
Real = Axiom "A real number type."
Whole = Axiom "A whole number type."
String = Axiom "A textual type."
Maybe<a> = | Some of a | None
Either<a, b> = | Left of a | Right of b
List<a> = | Nil | Link of (a, List<a>)
Map<a, b> = | Leaf of (a, b) | Node of (Map<a, b>, Map<a, b>)

category Semigroup<a> =
  | append : a -> a -> a

category Monoid<m; Semigroup<m>> =
  | empty : m

category Pointed<p> =
  | pure<a> : a -> p<a>

category Functor<f; Pointed<f>> =
  | map<a, b> : (a -> b) -> f<a> -> f<b>

category Applicative<p; Functor<p>> =
  | apply<a, b> : p<a -> b> -> p<a> -> p<b>

category Monad<m; Applicative<m>> =
  | bind<a, b> : m<a> -> (a -> m<b>) -> m<b>

category Alternative<l; Applicative<l>> =
  | empty<a> : l<a>
  | choice : l<a> -> l<a> -> l<a>

category Comonad<c; Functor<c>> =
  | extract<a> : c<a> -> a
  | duplicate<a, b> : c<a> -> c<c<a>>
  | extend<a, b> : (c<a> -> b) -> c<a> -> c<b>

category Foldable<f> =
  | fold<a, b> : (a -> b -> b) -> b -> f<a> -> b

category Traversable<t; Functor<t>; Foldable<t>> =
  | traverse<a, b, p; Applicative<f>> : (a -> p<b>) -> t<a> -> p<t<b>>
```

```
// TODO: define the Arrow categories.

id a = a
const a _ = a
flip f a b = f b a
```

Semantic Design for Nu Script

```
witness Monoid =
  | append = +
  | empty = toEmpty -t-

witness Monad =
  | pure = toPure -t-
  | map = map
  | apply = app
  | bind = bind

witness Foldable =
  | fold = fold

// TODO: define traverse so we can make a witness for Traversable.

Property = Axiom "A property of a simulant."
Relation = Axiom "Indexes a simulant or event relative to the local simulant."
Address = Axiom "Indexes a global simulant or event."

get<a> : Property -> Relation -> a = Axiom "Retrieve a property of a simulant indexed by Relation."
set<a> : Property -> Relation -> a -> a = Axiom! "Update a property of a simulant indexed by Relation, then returns its
value."

Stream<a> = Axiom "A stream of simulant property or event values."
getAsStream<a> : Property -> Relation -> Stream<a> = Axiom "Construct a stream of values from a simulant property."
setToStream<a> property relation stream = foldStream (fun _ -> set<a> property relation) stream
eventStream<a> : Address -> Stream<a> = Axiom "Construct a stream of values from event data."

foldStream<a, b> : (b -> a -> b) -> Stream<a> -> b = Axiom "Fold over a stream."
productStream<a, b> : Stream<a> -> Stream<b> -> Stream<(a, b)> = Axiom "Combines two streams into a single product stream"
sumStream<a, b> : Stream<a> -> Stream<b> -> Stream<Either<a, b>> = Axiom "Combines two streams into a single sum stream."
mapStream<a, b> mapper stream = foldStream (fun _ -> mapper a) stream


witness Comonad =
  | map = mapStream
  | extract = fun f a -> f a
  | duplicate = fun f -> f f
  | extend = fun f -> map f . duplicate
```

Semantic Design for MetaFunctions (a replacement for micro-services - unrelated to Nu)

```
Any = Axiom "The universal base type."

Symbol =
  | Atom of String
  | Number of String
  | String of String
  | Quote of Symbol
  | Symbols of List<Symbol>
symbolToString (symbol : Symbol) : String = Axiom "Convert a symbol to string."
symbolFromString (str : String) : Symbol = Axiom "Convert a string to a symbol."

Vsync<a> = Axiom "The potentially asynchronous monad such as the one defined by Prime."
vsyncBind<a, b> (vsync : Vsync<a>) (f : a -> Vsync<b>) : Vsync<b> = Axiom "Create a potentially asynchronous operation."
vsyncReturn<a> (a : a) : Vsync<a> = Axiom "Create a potentially asynchronous operation that return the result 'a'."
witness Monad =
  | bind = vsyncBind
  | return = vsyncReturn

IPAddress = String
NetworkPort = Whole
Endpoint = (IPAddress, NetworkPort)
Intent = String // the intended meaning of a MetaFunction (indexes a MetaFunction from a Provider - see below)

Container = Intent -> Symbol -> Vsync<Symbol>
Provider = | Endpoint | Container
MetaFunction = Provider -> Intent -> Symbol -> Vsync<Symbol>

makeContainer (asynchrounous : Bool) (repositoryUrl : String) (credentials : (String, String)) (envDeps : Map<String, Any>) :
Container = Axiom "Make a container configured with its Vsync as asyncronous or not, built from source pulled from the givern
GIT url, and provided the given environmental dependencies."

attachDebugger (container : Container) = Axiom! "Attach debugger to code called inside the given container."

call (mfn : MetaFunction) provider intent args : Vsync<Symbol> = mfn provider intent args
```