* Semantic Web. Backends. DOM. Protocol.
* Java 8 / 9. JavaScript. FP.
* Reactive Programming.
* Functional Programming (Monads, Zippers : Haskell / Clojure).
* AI / ML / NLP / Parsing / IR / IT.
* ServiceMix.
* JXTA. XML. XSL. XPath.
* DIDs. Blockchain. Solid. P2P. MQ.
* Math. Category theory. ISO.
* Protege / Protege Web
* Lectures / Topics
* Guha. Mondrian. Jpivot.
* Type constructor (wrapper / type).
* Unit wrapper / value instance.
* Bind / map / flatMap. Instance argument, transform (static / functor). Instance method (applications).
* Functor: A category consists of a collection of nodes (objects) and morphisms (functions). An object could be numbers, strings, urls, customers, or any other way you wish to organize like-things. (X, Y, and Z in the graphic are the objects.).
* A map is a function to convert something from one object to another. (f, g, and fog are the maps). Google tip: A map between objects is called a Morphism.
* So an object could be simple like a Number or a String. An object could also be more abstract like a Username, A User API URL, User API HTTP Request, User API Response, User API Response JSON. Then we can create maps or morphisms between each object to get the data we want.
* Examples of morphisms:Username -> User API UrlUser API Url -> User API HTTP RequestUser API HTTP Request -> User API ResponseUser API Response -> User API Response JSON
* Google tip: Function Composition is a way to combining multiple map or morphisms to create new maps. Using Function Composition we could create a map from Username directly to User API Response JSON.
* Now that we understand what it means to be Mappable, we can finally understand what a Functor is.
* A Functor is something that is Mappable or something that can be mapped between objects in a Category.
* An Array is Mappable, so it is a Functor. In this example I am taking an Array of Numbers and morphing it into an Array of Strings.
* Note: One of the properties of a Functor is that they always stay that same type of Functor. You can morph an Arraycontaining Strings to Numbers or any other object, but the map will ensure that it will always be an Array. You cannot map an Array of Number to just a Number.
* We can extend this Mappable usefulness to other objects too! Let's take this simple example of a Thing.
* If we wanted to make Thing mappable in the same way that Array is mappable, all we have to do is give it a map(morphism) function. And that is a Functor! It really is just that simple. Google tip: The "Thing" Functor we created is known as Identity.
* Monad:
* Sometimes functions return a value already wrapped. This could be inconvenient to use with a Functor because it will re-wrap the Functor in another Functor.
* This is where flatMap comes in handy. It's similar to map, except the morphism is also expected to perform the work of wrapping the value.
* Summary:
* A Functor is something that is Mappable or something that can be mapped between objects in a Category.
* A Monad is similar to a Functor, but is Flat Mappable between Categories.
* flatMap is similar to map, but yields control of the wrapping of the return type to the mapping function.
* Monads:
* Monads are a way to compose type lifting functions: g: a => M(b), f: b => M(c). To accomplish this, monads must flatten M(b) to b before applying f(). In other words, functors are things you can map over. Monads are things you can flatMap over:
* A monad is a way of composing functions that require context in addition to the return value, such as computation, branching, or I/O. Monads type lift, flatten and map so that the types line up for lifting functions a => M(b), making them composable. It's a mapping from some type a to some type b along with some computational context, hidden in the implementation details of lift, flatten, and map:
* Functions map: a => b which lets you compose functions of type a => b
* Functors map with context: Functor(a) => Functor(b), which lets you compose functions F(a) => F(b)
* Monads flatten and map with context: Monad(Monad(a)) => Monad(b), which lets you compose lifting functions a => F(b)
* Map means, “apply a function to an a and return a b". Given some input, return some output.
* Context is the computational detail of the monad’s composition (including lift, flatten, and map). The Functor/Monad API and its workings supply the context which allows you to compose the monad with the rest of the application. The point of functors and monads is to abstract that context away so we don’t have to worry about it while we’re composing things. Mapping inside the context means that you apply a function from a => b to the value inside the context, and return a new value b wrapped inside the same kind of context. Observables on the left? Observables on the right: Observable(a) => Observable(b). Arrays on the left side? Arrays on the right side: Array(a) => Array(b).
* Type lift means to lift a type into a context, blessing the value with an API that you can use to compute from that value, trigger contextual computations, etc… a => F(a) (Monads are a kind of functor).
* Flatten means unwrap the value from the context. F(a) => a.
* Dataflow, Reactive: Function composition creates function pipelines that your data flows through. You put some input in the first stage of the pipeline, and some data pops out of the last stage of the pipeline, transformed. But for that to work, each stage of the pipeline must be expecting the data type that the previous stage returns.
* A monad is based on a simple symmetry — A way to wrap a value into a context, and a way to unwrap the value from the context:
* Lift/Unit: A type lift from some type into the monad context: a => M(a)Flatten/Join: Unwrapping the type from the context: M(a) => a
* And since monads are also functors, they can also map:
* Map: Map with context preserved: M(a) -> M(b)
* Combine flatten with map, and you get chain — function composition for lifting functions, aka Kleisli composition:
* FlatMap/Chain Flatten + map: M(M(a)) => M(b)
* Monads must satisfy three laws (axioms), collectively known as the monad laws:
* Left identity: unit(x).chain(f) ==== f(x)
* Right identity:[m.chain](about:blank)(unit) ==== m
* Associativity:[m.chain](about:blank)(f).chain(g) ==== [m.chain](http://m.chain)(x => f(x).chain(g)
* Functor: mapping between categories A / B using a function. Map, alcance, rango, dominio, imagen (infer, aggregate). Connections: same number of items (inyective, biyective). Function: de wrapper en wrapper (morphism new image). Message / Augmentation Metamodel reifications.
* Actor / Role. Dynamic Object Model (OGM / Kinds). Golden Braid: Metaclass, class, instance, occurrence relation relative to layer context levels.
* Reference Model:
* Type : Value;
* (Value, Type);
* (Type, Type);
* (Value, Value);
* Aggregation: cons cells tree traversal. Layers contexts representation. Dataflow augments updates / append corresponding tree cells: (first: (rest: nil).
* (C (S (P (O))));
* Aggregate / Augment inputs / transforms: Parsed model streams. Reactive data structures.
* Encoding: addressing, semantic graph networks. URN overlay semantic addresses encoded mappings.
* Dataflow:
* Input statements: Augment. Aggregate, build layers contexts representation quads. Parse:
* Monadic parser combinator: aggregated metamodel nested layers contexts corresponding wrapper / wrapper hierarchy types DOM / AST (from contexts quads aggregation nesting hierarchy levels).
* Augment: signatures (case meta / classes / instance / occurrences) dataflow. Apply input statements: new inputs / kind new attributes (person, employee -> position, salary). Parse.
* Protocol: Forms / Flows augmented / parsed representation / metamodel I/O. HAL / HATEOAS endpoint encoding for navigation, transforms and inputs augmentation.
* Protocol: Forms / Flows Dialog / Prompts resolution. Context roles, wrapper kinds navigation / transforms declaratively stated in encoded representations.
* Monad: context type. Metaclass.
* Monad: wrapped type. Class.
* Monad: wrapped value. Instance.
* Monad: wrappers hierarchy context type instance. Occurrence.
* Unit. Type constructor (hierarchy context / member types / values factories). flatMap / map / flatMapN.
* Monad: context type case classes hierarchy? Factory methods, hierarchy types / members values / signatures (wrapped context / values) cases (predicates):
* Optional: None | T [case context | case context / case value signature]
* Result: Error | OK [case context | case context / case value signature]
* Writer: Value, List<S> trace
* Monads: Quads Contexts wrapper of Occurrence / Subject aggregations. Root type: Resource. Kind (model / semiotic / domain meta resources) functors filter / traversal (streams: flatMap Resource set specifications).
* Wrapped resources holds references to its wrapping occurrence contexts (Resource root type interface, contexts hierarchy / reification levels: upper layer instances reify layer occurrences in contexts).
* Wrapping unit / bind: layers traversal until container wrapped category type is met. Example: Behavior(s) of an Entity. Augmentations (types, roles, attributes reified model / domains aggregations). Example: Model, SK, P; Domain, someSubjKind, somePredicate.
* Form / Flow: state navigation / browse. Dialogs / Prompts (contexts). Messages I/O: Augmentation signatures (kinds) streams.
* Messages: Augmentation according signatures. Broadcast. Key / Value, Event sourcing routes (levels). Form / Flow Dialogs / Prompts Messages  for signatures (kinds) of unknown Resources resolution (layers wrapping traversal, CSPOs layers resolution for CSPO signatures / attributes / values).
* Encoding: IDs. Embed metaclass, class, instance, occurrence metadata (context, role, attributes, values). Functional APIs. Wrappers / Transforms (augment: aggregate / classify, roles, properties "graph" rels). Polygon Vector Space Model. ANN embeddings / autoencoders.
* Forms / Flows Dialogs / Contexts. Protocol. Resources, addressing, representations, navigation / traversal: properties "graph" rels (Wrappers / Transforms). Functional APIs.
* Sets encoding: properties in axes (kinds). SortedSet (hierarchies). Metaclass, class, instance, occurrence properties in axes for CSPO IDs. Augmentations: property graph rels navigation / traversal. Dialog Forms / Flows "state" contexts.
* Contexts Wrappers kinds Transforms / Traversals functors: Augmentations declaratively stated in upper Context layers (kind classification, kind roles, kind attributes / values).
* Dialog Forms / Flows "state" Contexts browsing (upper Context SPO kinds: current context streams).
* Augmentation navigation of Transforms / Traversals as a Context (streams / filters). Levels / reification.
* Order. Iteration. Predicates (resource meta / domain / kinds). Streams filter, conditionals, jumps. Aggregation. Functional mapping / reduce, etc. Dataflow signatures (case meta / classes / instances /occurrences)
* Resource API (layer roles):
* Statement: Resource (CSPO: Statement) Property graph URL wrapper. URL occurrences aggregate. Functional occurrences properties (roles / streams) for Statement wrapped URL:
* Contexts: Metaclass URL occurrences.
* Subjects: Class URL occurrences.
* Predicates: Instance URL occurrences.
* Objects: Occurrence URL occurrences.
* Statement wrapped URL occurrences functional roles Kinds: Map<Role, Kind> (reified in Metamodel).
* CSPO Kinds (streams functors) declaratively stated in aggregated layers CSPO occurrences (kind classification, kind roles, kind attributes / values).
* Augmentation navigation of kinds Transforms / Traversals as streams / roles (filters). Levels / reification: kinds from Statement / roles layers (reification / levels axes).
* Query / stream context selectors: ID (URL), layer context type, layer context role, layers kinds. Transforms (functor kinds: augment / browse query context according kind specification with corresponding statements).
* Levels: Grammars (kinds functors dataflow signatures). Productions: Augmentations (parsed / produced in navigation contexts). Dataflow order (sets / hierarchies).
* Aggregation: inputs in contexts layers. Case matching: metaclass, class, instance, occurrence, kind grammars.
* Parsing: Monadic combinator parsers: quads contexts layers (recursion). Metaclass, class, instance, occurrence, kind aggregations parsers.
* Augmentation. Functional API: Monadic DOM / AST Parse Tree (cons cells) context layers hierarchy wrappers, Resource metaclass, class, instance, occurrence, kind hierarchy wrappers (i.e: contexts instances / parsed kinds).
* Dataflow: Aggregation, Parsing, Augmentation. Streams: reactive / event driven. Model reified Message Functors / Transforms.
* Zippers: Aggregation / Addressing: Locations / Contexts. Parsing. Monads. Augmentation (navigation / transforms) Reactive Streams (location observers / observables: paths / kind paths dataflow signatures).
* Inputs / CRUD navigation / transforms Augmentation Dataflow: parsing / zippers over core Model reified context layers meta Resources AST / DOM parsers (zippers paths in meta Resource aggregates / parses inputs in contexts). Resource metaclass, class, instance, occurrence paths / cases matching zippers.
* Blockchain: P2P (JXTA / Git) DL Distributed Ledger inter Node Backend. DIDs (traceable semantic state: Distributed IDs encoded / embedding result of transforms, labeled / property graph statements / contexts: saga / zippers). Smart contracts (signatures: Dataflow). Monads (immutable state, transactions: functor morphisms). Zippers (chain contexts: mutable chain branches, dimensional contexts / labeled property graphs). Reactive Augmentation (I/O) APIs: Resource metaclass, class, instance, occurrence paths / cases case matching (inputs quads, parsed DOM, outputs quads). Quads Forms / Flows Protocols.
* Node Protocol: Forms / Flows DAV HAL / HATEOAS Client Application Sessions (navigation contexts). JCR, Hierarchical structures (XML, XPath, XSL, XLink, XQuery, XPointer) representations of augmented reactive DOMs. Representation Levels (onto meta resources): metamodel / session / domains. Behavior encoded in (augmented) representations functional contexts traversals. JXTA / Git Backend inter Node P2P Blockchain Node quads DL IO sync. Connector Nodes: reactive dataflow (signatures: smart contracts).

Augmentation

Aggregation, Alignment, Activation:

Class (relation) and instance (relationship) being the things that could be asserted for each (domain / range for classes, pairs of "roles" for instances and attributes for both: as property graph) the difference between relation and relationship.

A naive approach of render this in pseudo RDF / RDFS:

Marriage : Relation;

Husband domain Marriage;

Husband range Male;

Wife domain Marriage;

Wife range Female;

Marriage properties (date, etc.);

aMarriage : Marriage;

aMarriage husband Pete;

aMarriage wife Mary;

Marriage attributes (domain / range). Reified Relation instances entails statements (expands links, attributes in property graphs) for Relationship roles / players attributes:

Peter marriedWith / husbandOf Mary; domain: spouse / husband; range: spouse / wife;

Mary marriedWith / wifeOf Peter; domain: spouse / wife; range: spouse / husband;

marriedWith / husbandOf / wifeOf statements in a CSPO context: aMarriage;

There should be an inference method materializing inferences of role instances attributes according the Relation class Relationship instance roles they play.

Or, if RDF Quads are not available, entailed properties schema / instances: marriedWith:\_0 / husbandOf:\_0 / wifeOf:\_0 instances of corresponding relation class attributes. Entailing relationship (aMarriage) instances context attribute.

marriedWith:\_0 rdfs:type / rdfs:subPropertyOf marriedWith (expansion property kind).

The case is that a "terminals" relation relationship resource statements "expansion" materialized view renders the Relationship "extension". Way back entailing / inferring relation / relationship class / attributes roles should be possible.

Source (higher order) relations may relate relations / relationships with other relations / relationships thus allowing a richer set of concepts into an ontology / dialect. Example: Peter / Mary Husbandhood related to their Marriage. RDFS domain / range properties provides the inference means here to parse such a relationship entailing relation context / roles.

Rules for expansion: if a Relation is a class for Relationship(s) which has Role(s) for Resource(s) in SPO statements the statements expansion is the "materialized" view of the Relation instance in SPO statements.

Having a tuple:

(Template, Relation, Relationship, Role, SPORole, Resource);

Aggregated Template SPORole Resource should enable the use of some query mechanism (SPARQL? Zippers?) for building output triples.

Reify from lower layers to expanded statements materialized views and aggregate forward (I/O, Augmentation).

Transform: apply Kind Relation. Relation defined by extension (tuples) and intension (property / attributes relations).

Layers hierarchy:

(Context, Object, Concept / Sign, Value);

(Resource, Resource, Resource, Resource);

(Role, Resource, Resource / Attribute, Resource / Value);

(Statement, Role, Resource, Resource / Attribute);

(Entity, Statement, Role, Resource);

(Template, Entity, Statement, Role);

(Mapping, Template, Entity, Statement);

(Flow / Augmentation, Mapping, Template, Entity);

(Behavior / Message, Flow, Mapping, Template);

(Measure, Behavior, Flow, Mapping);

(Unit, Measure, Behavior, Flow);

(Dimension / Axis, Unit, Measure, Behavior);

(Value, Dimension, Unit, Measure);

(Concept, Value, Dimension, Unit);

(Object, Concept, Value, Dimension);

(Context, Object, Concept, Value);

Reify from lower layers to expanded statements Resource materialized views and aggregate forward into relations / relationships: contexts (I/O, Augmentation).

(husband, role, resource, ?);

(aMarriage, husband, role, resource);

(Marriage, aMarriage, husband, role);

TBD.

<https://wiki.haskell.org/Zipper>

<https://wiki.haskell.org/Zipper_monad>

<http://learnyouahaskell.com/zippers>

<https://www.functionaljava.org/features.html>

<https://stackoverflow.com/questions/5919901/data-structure-differentiation-intuition-building>

<https://www.google.com/search?q=The+Derivative+of+a+Regular+Type+is+its+Type+of+One-Hole+Contexts&oq=The+Derivative+of+a+Regular+Type+is+its+Type+of+One-Hole+Contexts>

<https://github.com/jon-hanson/parsecj/blob/master/README.md>

<https://blogs.msdn.microsoft.com/lukeh/2007/08/19/monadic-parser-combinators-using-c-3-0/>

<https://www.w3.org/Data/events/data-ws-2019/>

<https://dzone.com/articles/functor-and-monad-examples-in-plain-java>

<https://dzone.com/articles/whats-wrong-java-8-part-iv>

<https://curiosity-driven.org/monads-in-javascript>

<https://hackernoon.com/functional-javascript-functors-monads-and-promises-679ce2ab8abe>

<https://medium.com/javascript-scene/javascript-monads-made-simple-7856be57bfe8>

<https://importantshock.wordpress.com/2009/01/18/jquery-is-a-monad/amp/>

<https://blogs.msdn.microsoft.com/lukeh/2007/08/19/monadic-parser-combinators-using-c-3-0/>

<https://github.com/jon-hanson/parsecj/blob/master/README.md>

<https://dzone.com/articles/parsing-in-java-part-3-diving-into-peg-parsers>