Draft OAGi Interoperable Mapping Specification

OAGi Members

2022/05/25

1 Introduction

This document describes a data mapping language designed to serve as an *interoperable exchange form* for expressing the intent of many mapping and data restructuring needs. As an interoperable exchange form, it is intended that the language can be translated (by humans and machine agents) into mapping specification in other languages. For example, it should be possible to translate statements in the exchange form into mapping specification used by commercial mapping tools.

The document is a draft and as of this writing (2022/05/25) is likely to be updated often. OAGi members can find the most recent version of this document in the *OAGi Mapping Specification Working Group* Confluence pages (under "Mapping Doc").

The mapping language described (provisionally called RADmapper) borrows predominantly from the JSONata language [**?**], but also includes provisions for mapping to/from forms other than JSON. These forms include tables (e.g. Excel) XML, and networks of data such as knowledge graphs. The RADmapper language also addresses additional use cases, including in-place updating of target data sources. To support networks of data, the mapping language borrows ideas from the Object Management Group's Queries, Views and Transformation relational (QVT-r) [**?**] mapping language, and Datalog [**?**].

The document describes both the interoperable exchange form and a software tool that executes it. The tool serves as a reference model to explore the operational semantics of the language. Whether or not the tool is useful for transforming information in real-world settings remains to be seen; we'll generate an OpenAPI Specification for the tool to test that idea. At the current state of development, it isn't clear whether the language being described is the interoperable exchange form itself or something derived from the executable language being developed. For example, it is possible that abstract syntax trees (ASTs) of content in the language being described could themselves be transformed into the actual exchange form. The same process could possibly be used to produce "mapping tables", spreadsheets provided to programmers to hand-code the transformations in their system's native programming language. In this light, the RADmapper tool can be thought of as means to get the intent of the mapping correct before handing it off to programmers. But this discussion is getting ahead of itself; the next section surveys key aspects of the language.

2 Quick Start: Example Mapping Tasks

In order to give you a sense of things, the next subsections describe the basics and how some common query, data restructuring, and mapping challenges are addressed by the language.

2.1 Simple Operations

2.2 Vectors and Iteration

2.2.1 Mapping Over a Vector

In the language, iteration is typically performed implicitly in the sense that when the argument to an operation is an array, the operation is applied to each element of the collection. The result returned is an array of the result of applying the operator to each element. Thus in Figure 1, `.zipcode` is applied to the array `$ADDRS` returning an array `[``20898'' ``07010-35445'', ``10878'']`.

Figure 1: Map over an array of entities, collecting ZIP code.

```

(
  $ADDRS :=
     [{"name"     : "Peter",
       "street"   : "123 Mockingbird Lane",
       "zipcode"  : "20898"},

       {"name"    : "Bill",
        "street"  : "23 Main Street",
        "zipcode" : "07010-3545"},

       {"name"    : "Lisa",
        "street"  : "903 Forest Road",
        "zipcode" : "10878"}];

   $ADDRS.zipcode
)
```

In the case that the operator returns null for elements of the array, no value is collected. For example, if "Bill" in the above didn't have a zipcode, `$ADDRS.zipcode` would have returned `["20898", "10878"]`.

2.2.2 Filtering an Array

Suppose, for some odd reason, we didn't like 9-digit Zip codes (what the US Postal Service calls "Zip+4") and that we simply want to exclude them from the collection. There are a few simple ways to do this. One of those is to append a test expression to the end of the previous expression. This "filtering" test expression needs to be enclosed in square brackets to indicate that it is filtering. `$ADDRS.zipcode[$match(/^\d[5,5]$/)]`. Regular expression syntax is arcane but what is shown is typical of JavaScript, Java and many other regex libraries. The above matches on strings that are exactly 5 number characters, thus excluding the Zip+4 Zip Codes.

2.2.3 Index Variables

Index variables are variables used to iterate through arrays. They are common in procedural code, for example, the variable `i` in the pseudocode `for i in [1..9]` {...} is an index variable.. A goal of interoperable exchange is to allow the expression of individual mapping statements that can be defined independent of larger program context. This kind of independence is useful in conventional *mapping tables*, spreadsheet used to describe to programmers the intent of proposed mapping. The problem with index variables in this regard is that they would add meandering procedural expression where concise, independent, declarative expression is desired. But what if you needed to enumerate the things you were working with, for example, to put an index number in front of them? In cases like this, you can avoid the need for an index variable by using the language's `$map` higher-order function; it is like JavaScript's `map`. See Figure 2 below.

Figure 2: Using an index variable.

```
$map($ADDRS.zipcode, function($v, $i) {'zip ' & ($i+1) & ' ' & $v})
```

returns

```
[ 'zip 1 20898', 'zip 2 07010-3545', 'zip 3 10878'].
```

In the above, `function` introduces a function, and the following code in brackets is the body of the function, a single expression is returned. In `$map` (and `$filter`, and `$reduce` described later), the function is called once each with each member of the first argument (`$ADDRS.zipcode` here) in sequence as the value bound to `$v`, and `$i` is bound in turn to successive integers in the sequence `[1,2,3]`. This effectively eliminates the need for user-defined index variables.

2.3 Working with Code Lists

A mapping tasks commonly needed is translating a code list, or subsetting one to a set of code values that actually is used in your business processes. For example, there is a very large set of codes for transportation events in the code list [**don't recall the spec**]. Suppose you are receiving messages with lots of these values but your order management application only wants to know whether the shipment is likely to be late. There is no getting around the fact that you need to explicitly describe a functional relationship to do this; the functional relationship can be implemented as a lookup table. The language defines a switch statement to do this kind of thing. An example is shown in Figure 3.

Figure 3: Mapping code values.

```
1   $Codes2OurCodes := function($theirs)
2    {switch $theirs {
3        #{2, 5, 52, 66} : 'late';
4        #{3, 6, 77, 85} : 'on time';
5        * : 'unknown'}
6    };
```

The code in Lines 3 and 4 of Figure 3 implement a table. The #{. . . } syntax defines sets. On Line 3 the source defines set of code values (2, 5, 52, and 66) and its use indicates that elements of this set are mapped to the target value "late".

Were the table to be larger, writing it could get tedious. For this reason, it might be reasonable to provide an alternative, whereby a lookup table provided elsewhere is referenced.[1]

2.4 Working with Tabular Data

Being able to read and write spreadsheet information is a very handy capability in mapping. Of course, Excel-like spreadsheets can have multiple sheets and the content can be non-uniform, including merged cells and formula. The language currently does not provide means to deal with these complexities. However simple tables with no surprises and a header row naming columns (and common-separated value (CSV) files that conform to these requirements) can easily be viewed as an array of map structures. For example, Table 1 can be viewed as the structure shown in Figure 4.

[1] The code in Figure 3 also shows how names are associated with functions. Functions in the language do not have names. If you need to use a function in multiple places, assign it to a variable as shown in the figure, (use of `$Codes2OurCodes`).

Table 1: A simple table oriented such that columns name properties.

ShipDate	**Item**	**Qty**	**UnitPrice**
6/15/21	Widget 123	1	$10.50
6/15/21	Gadget 234	2	$12.80
6/15/21	Foobar 344	1	$100.00

Figure 4: Table 1 viewed as an array of map structures.

```

[{"ShipDate":  "2021-06-15", "Item":  "Widget 123", "Qty": 1.0, "UnitPrice": 10.5},
 {:ShipDate":  "2021-06-15", "Item":  "Gadget 234", "Qty": 2.0, "UnitPrice": 12.8},
 {:ShipDate":  "2021-06-15", "Item":  "Foobar 344", "Qty": 1.0, "UnitPrice": 100.0}]
```

The built-in function `$readSpreadsheet` reads spreadsheets. An example usage is:

```
$readSpreadsheet("data/spreadsheets/ExampleInvoiceInfo.xlsx", "Sales Info")
```

where the first argument names an Excel file and the second names a sheet in that spreadsheet. If the table is transposed (so that all the properties are in its first column, and each row concerns a different object), a third argument with value `true` can be specified to access the data in the more useful orientation.

3 Working with Data Having Complex Interrelations

Some situations call for more complex query and restructuring functionality than what has been illustrated above. Everything to this point in the discussion could be performed easily with JSONata [**?**], a language for querying and restructuring JSON-like data. JSONata is used, for example, in the Open Integration Hub [**?**], an open software platform that received its initial funding in 2017 from the German Federal Government, Federal Ministry for Economic Affairs and Energy. JSONata is also used in Node-RED [**?**].

The JSONata viewpoint could be described as one where a tree (or equivalently, a JSON object) is navigated from the root. Decades of experience, dating back to the early days of EDI, has shown that tree-based organization such as JSON objects is a quite reasonable choice where the communication of document-like information (e.g. "messaging") is needed. The point of an interoperable exchange form such as RADmapper[2], however, includes additional requirements. Among these is the ability to describe relationships to and from data possessing complex interrelationship, for example, data described by a relational schema or knowledge graph. To argue that APIs to the back-end system already do this work is to miss the point: we are not trying to replace a back-end system function, but to describe the relationships in ways that help business analysts, programmers, and machine agents.[3]

To illustrate the challenge of using a JSONata-like engine for data having complex interrelations, suppose, for example. . . . [Discussion of the OWL example]

[2]RADmapper is my shot at a name. RAD=rapid application development. Suggest a name!

[3]Likewise we are going to try to address situations where there are multiple sources. There are tools to do this already, but again, we are trying to describe the relationships, not replace existing software.

3.1 Data Organized as Triples

Relational and graph-based data can be described by triples $[x, rel, y]$ where x is an entity identifier, y is data (string, number, object, array, another entity identifier, etc.)[4] [5] and rel is a relationship (predicate) holding between x and y. The graph-based viewpoint on such data is that x and y are vertices, and rel is an edge. rel might be implemented as a string. Thus, the translation of structured data such as JSON objects into such triples is relatively straightforward:

1. A unique entity identifier is associated with each object; this identifier serves as the first element of triples about that object. In JSON, for example, an object is a collection of name-value pairs delimited by curly brackets.

2. Each object attribute is represented by the string naming it[6]; these supply the second element of the triple, the *rel*.

3. The third element of the triple provides an atomic datum (string, number, entity identifier, etc.) associated with the subject attribute of the subject object.

For example, the first object in Figure 4, `{"ShipDate": "2021-06-15", "Item": "Widget 123", "Qty": 1.0, "UnitPrice": 10.5}` can be encoded as the following four triples, where the integer 11 is the unique entity identifier for this object.

```
[11, "ShipDate",  "2021-06-15"]
[11, "Item",  "Widget 123"]
[11, "Qty", 1.0,]
[11, "UnitPrice", 10.5]
```

Of course, an object can have multiple values for an attribute. For example, a person can have multiple phone numbers. The way you'd typically handle cardinality greater than 1 in JSON and similar schemes is to simply provide the attribute with an array value. For example,

```
{"name" : "Bob", "phoneNumbers" : ["123-456-1111", "123-456-2222"]}
```

In these cases, there would be a triple for each of phone numbers. Thus, the above might be normalized to triples as follows:

```
[12, "name" "Bob"]
[12, "phoneNumbers" "123-456-1111"]
[12, "phoneNumbers" "123-456-2222"]
```

In the following, an array of objects is used for phone numbers. Consequently, the `PhoneNumberObjs` triples on entity 13 reference additional entities, not data.

```
{"name" : "Bob", "phoneNumberObjs" : [{"cell" : "123-456-1111"}, {"work" : "123-456-2222"}]}
```

[4]What additional atomic data types shall we provide? Dates? URIs? UUIDs? etc.

[5]Triples provide 5th-normal form relational data. Recomposing a concept from 5-th normal form typically requires as many joins as there are attributes to recompose. The upside of using triples are that (1) substantial amounts of data are already in triples (e.g. RDF etc.), (2) some query engines helpfully index triples in multiple ways entity-attribute-value, attribute-entity-value, and attribute-value-entity, and (3) mapping engines can use triples effectively. The challenge in using triples with this language is in preserving the abstraction of a functional pipeline owed to JSONata-like language features.

[6]I suppose we'll only support strings, though namespaced symbols would be nice!

Figure 5: The object for Bob as triples

```
1  [13, "name" "Bob"]
2  [13, "phoneNumberObjs" 14]
3  [13, "phoneNumberObjs" 15]
4  [14, "cell" "123-456-1111"]
5  [15, "work" "123-456-2222"].
```

It is worth taking the time to get comfortable with this idea of encoding information in triples; it is how we work with complex data in RADmapper.

3.2 Queries on Triples

You can query triples using Datalog-like notation.[7] Queries have a square-bracket notation similar to triples illustrated above, but with *query variables* substituted for some of the values. In JSONata, variables start with a `$`, in queries, query variables start with a `?`. For example, `[?e "name" "Bob"]` is a query that will bind the query variable `?e` to the entity identifier of any entity that has ”Bob” as its `name` attribute. But why would you want the entity identifier, it isn't domain data? The typical answer is that you will need it to perform a sort of join on triples. For example, if we want to get the `cell` number for `Bob`, we could use the following interrelation of triples.

```
[?e "name" "Bob"]
[?e "phoneNumberObj" ?pn]
[?pn "cell" ?cellNum]
```

If the triples queried are just the ones of Figure 5, then the triple on Line 1 matches the query pattern `[?e "name" "Bob"]`, binding `?e` to the entity id 13. The second query pattern, `[?e "phoneNumberObjs" ?pn]`, matches two triples, those on Lines 2 and 3, binding `?pn` to either entity 14 or 15. Owing to specifying `"cell"` as the attribute, however, the third query pattern, `[?pn "cell" ?cellNum]`, only matches the triple on Line 4. This result could be expressed in a JSON-like object notation as a collection of objects such as: `[{?e: 13, ?pn: 14, ?cellNum: "123-456-1111"}]`. Each such collection of bindings values that together consistently match the query variables in the data is called a *binding set*. Note that binding sets can contain the database entity IDs, for example, binding `?e` to entity 13 and `?pn` to entity 14 in the above. Though entity IDs are not part of the domain data, they are part of how data is strung together by the triples and thereby they might be useful. Figure 6 depicts another example, the generation of two binding sets from in-line data.

Figure 6: A simple query against in-line data.

```
 ( $ := [{'Person/firstname' : 'Bob'  , 'Person/lastname' : 'Clark'},
         {'Person/firstname' : 'Peter', 'Person/lastname' : 'Dee'}];
   $.$query([?person :Person/firstname ?fname]
            [?person :Person/lastname  ?lname]) )

  // Returns the following binding set:
  [{:person 4, :fname 'Peter', :lname 'Dee'}
   {:person 3, :fname 'Bob', :lname 'Clark'}]
```

There are a few thing to note about working with Datalog-like languages like this one. First, except for performance concerns, it does not matter what order you list the query triples. Second, you can put query variables in any (or all) of the three positions. For example, `[?e "cell" "123-456-1111"]`

[7] There are many implementations of Datalog, includes ones for Java and JavaScript, so what is being proposed here should not be hard to implement.

binds `?e` to 14 in the above. Third, it is possible that there is in the data more than one match; for example, `[?e ?a ?v]` returns a collection binding set matching all data that is in context. Finally, you can imagine that using attribute names like ”name” and ”qty” could get confusing. (Name of what? Quantity of what?) For this reason, where possible, best practice in Datalog-like languages uses namespaced attributes. We could use, for example, strings with a slash between the namespace name and the property name. Thus perhaps, `"Person/name"` and `"Phone/cell"`. But since these identifiers serve the special purpose of *roles*, the language uses a special syntax: a prefixed colon rather than string delimiters. Thus `:Person/name` and `:Phone/cell`. Figure 6 above illustrates use of the role syntax on Lines 3 and 4.

3.3 Query Functions

`$query` is a RADmapper language element that takes a collection of triple forms and (optional) parameters and returns a function that, when applied to data, returns a collection of binding sets. For example:

```
$.$query([?e "name" "Bob"]
         [?e "phoneNumberObjs" ?pn]
         [?pn "cell" ?cellNum])
```

returns `[{?e: 13, ?pn: 14, ?cellNum: "123-456-1111"}]` when `$` is the data of Figure 5. The example below illustrates the use of an optional parameter `$name`:

```
$queryCellByName := $query($name)([?e "name" $name]
                                  [?e "phoneNumberObjs" ?pn]
                                  [?pn "cell" ?cellNum])
```

Using the same data, `$.$queryCellByName("Bob")` would return the same result as the previous query, where ”Bob” is hard-coded.

3.4 Mapping Networked Data

Binding sets, in themselves, are not too useful; you might be wondering why we even discuss them. The answer is that they are crucial to mapping networked data and doing in-place updates. Once you have the binding sets, you are halfway there; what remains is to use the binding sets to produce target data. This section describes that process, beginning with definitions of some terms just used:

networked data a collection of data that contains pointers to other parts of the same data collection.

For example, we could have a data in triples that includes information about Bob. Instead of repeating everything we know about Bob each time he is referenced in the data, networked data can use references to that same data. Resolving the reference (which might be implemented as a UUID, for example) connects to the information about Bob.

in-place update the idea that the mapping task might involve updating an existing collection of data, rather than defining new data about it.

For example, on Bob’s 30th birthday, we don’t just add the fact `{'person/name' : 'Bob', 'person/age' : 30}`, we retract the fact that Bob is 29 and assert the new fact.

Before we can talk about mapping networked data and in-place updating, it is important to recognize that these activities require some additional knowledge about the (target) data. For example, how do we know (a) that an action is intended to update some referenced data rather than add new additional information about it, and (b) whether to use a pointer to reference some data rather than just put the data there? We need more information about the data. Specifically, to perform these more complex mapping tasks we have to know: (1) the cardinality of each attribute, (2) the type of each attribute (or at least the distinction between references and data types), and (3) attributes that provide keys (that is, uniquely identify an object of a given type).

Notice that condition (3) mentions the idea of object type. It is not the case that objects need to have any inherent notion of type to use RADmapper mapping. It is enough that the programmer recognizes the type by the attributes it possesses. (This is the so called *duck typing* — if it walks like a duck and quacks like a duck it is a duck.) Thus, an object that has an email address and a phone number might be recognized as a customer.

3.4.1 Complex mapping task example

The following example illustrates mapping of a network of Web Ontology Language (OWL) data to a relational form. The source OWL data used is simplified from actual OWL data to allow easier discussion. The data consists of objects of two types, one is OWL classes; these have the value `owl/Class` in their `rdf/type` attribute. The other is OWL properties (`owl/ObjectProperty`). The simplifications include using single values for `rdfs/domain`, `rdfs/range`, and `rdfs/subClassOf`. An OWL class and property is depicted in Figure 7.

Figure 7: An object (`owl/Class`) and a relation (`owl/ObjectProperty`) in the source population. These are somewhat simplified from realistic OWL data.

```
{'resource/iri'       : 'dol/endurant',
 'resource/name'      : 'endurant',
 'resource/namespace' : 'dol',
 'rdf/type'           : 'owl/Class',
 'rdfs/comment'       : ['The main characteristic of endurants is...'],
 'rdfs/subClassOf     : :dol/spatio-temporal-particular,
 'owl/disjointWith'   : ['dol/abstract', 'dol/quality', 'dol/perdurant']}

{'resource/iri'       : 'dol/participant',
 'resource/name'      : 'participant',
 'resource/namespace' : 'dol',
 'rdf/type'           : 'owl/ObjectProperty',
 'rdfs/comment'       : ['The immediate relation holding between endurants and perdurants...'],
 'owl/inverseOf'      : 'dol/participant-in',
 'rdfs/domain'        : 'dol/perdurant',
 'rdfs/range'         : 'dol/endurant'}
```

The target data we'll be creating consists of three kinds of things: schema, tables, and columns. Even with this simple data, there are a few options for designing the relational schema. The following are design choices that define the form of the mapping target:

1. Both `owl/Class` and `owl/ObjectProperty` can have `rdfs/comment` and the relationship is one-to-many. A single table with keys consisting of the resource IRI and comment text will suffice.

2. `rdf/type` is one-to-one with the class. Though the possible values are limited to just a few such as `owl/Class` and `owl/ObjectProperty`, we will represent it with a string naming the type.

3. `resource/name` and `resource/namespace` are also one-to-one and are just the two parts of `resource/iri` and could be computed, but we will store these in the class table too.

4. We will assume `rdfs/domain`, `rdfs/range`, and `rdfs/subClassOf` are single-valued. Typically they are not in a real OWL ontology.

5. We will assume that we want to support storage of individuals of the types defined by OWL classes. Though our approach here is not at all reflective of description logic, where class subsumption is the primary kind of inference, mapping will produce a two-column table where one column is the individual's IRI and the other is the foreign key of a class to which it belongs.

6. We will assume that all relations are conceptually binary. Thus storing individuals means that for each `owl/ObjectProperty` mapping will produce a two-column table to represent both a relation and its inverse ("inverse pairs") where an inverse is defined. Such a table works in both directions (the relation and its inverse), so we will have to prevent creating a table for one member of each inverse pair.

With the above considerations in mind, it becomes apparent that some of the work involves nothing more than storing class and property metadata into tables we can define ahead of time. The `owl/ObjectProperty` definitions, however, entails one new table for each relation (or relation pair if an inverse is defined). The rows of these tables would be populated by instances of the classes specified by `rdfs/domain` and `rdfs/range`. This information is not in the ontology, but Figure 8 depicts the static tables in typical relational DDL. These are populated by the ontology `owl/Class` content of the Figure 9 depicts tables that would be created.

Figure 8: Static DDL for storing class and object relation metadata. The mapping will generate information equivalent to DML to populate this from the source data.

```
CREATE SCHEMA typicalOWL;

CREATE TABLE ObjectDefinition
  (resourceIRI       VARCHAR(300) primary key,
   resourceLabel     VARCHAR(300) not null,
   resourceNamespace VARCHAR(300) not null);

CREATE TABLE ClassDefinition
  (resourceIRI VARCHAR(300) primary key,
   subClassOf  VARCHAR(300) references ClassDefinition);

CREATE TABLE ObjectClass
  (resourceIRI VARCHAR(300) primary key,
   class       VARCHAR(300) references ClassDefinition);

CREATE TABLE DisjointClass
  (disjointID  INT primary key,
   disjoint1   VARCHAR(300) not null references ObjectDefinition,
   disjoint2   VARCHAR(300) not null references ObjectDefinition);

CREATE TABLE ResourceComment
  (commentID    INT primary key,
   resourceIRI  VARCHAR(300) not null references ObjectDefinition,
   commentText  VARCHAR(900) not null);

CREATE TABLE PropertyDefinition
  (resourceIRI    VARCHAR(300) primary key,
   relationDomain VARCHAR(300) references ObjectDefinition,
   relationRange  VARCHAR(300) references ObjectDefinition);
```

Figure 9 depicts the result of mapping data from Figure 7 using a mapping specification that will be described below.

Figure 9: Result of mapping the data depicted in Figure 7. This specifies content equivalent to (1) DML to capture metadata for `owl/Class` and `owl/ObjectProperty` objects in the static tables defined above, and (2) DDL to create tables for `owl/ObjectProperty` objects.

```
{'instance-of'  : 'insert-row',
 'table'        : 'ObjectDefinition',
 'content'      : [{'resourceIRI'       : 'dol/endurant'},
                   {'resourceLabel'     : 'endurant'},
                   {'resourceNamespace' : 'dol'}]}

{'instance-of'  : 'insert-row',
 'table'        : 'ClassDefinition',
 'content'      : [{'resourceIRI'    : 'dol/endurant'},
                   {'subClassOf'     : 'dol/spatio-temporal-particular'}]}

{'instance-of'  : 'insert-row',
 'table'        : 'DisjointClass',
 'content'      : [{'disjointID'   ; 1},
                   {'disjoint1'    : 'dol/endurant'},
                   {'disjoint2'    : 'dol/abstract'}]} // ... (Two more disjoints elided.)

{'instance-of'  : 'insert-row',
 'table'        : 'ResourceComment',
 'content'      : [{'commentID'   : 1},
                   {'resourceIRI' : 'dol/endurant'},
                   {'commentText' : 'The main characteristic of endurants is...'}]}

// Similar content for the ObjectPropery dol/participant is elided.

{'instance-of'  : 'insert-row',
 'table'        : 'PropertyDefinition',
 'content'      : [{'resourceIRI'    : 'dol/participant'},
                   {'relationDomain' : 'dol/perdurant'},
                   {'relationRange'  : 'dol/endurant'}]}

// The DDL for the participant table:

{'instance-of' : 'create-table',
 'table'       : 'DOLparticipant',
 'columns'     : [{'colName' : 'propertyID',
                   'dtype'   : {'type' : 'varchar', 'size' : 300, 'key' : 'primary'}},
                  {'colName' : 'role1',
                   'dtype'   : {'type' : 'varchar', 'size' : 300, 'ref' : 'ObjectDefinition'}},
                  {'colName' : 'role2',
                   'dtype'   : {'type' : 'varchar', 'size' : 300, 'ref' : 'ObjectDefinition'}}]}
```

Figure 10 depicts the complete specification of a transformation of the source. `$transform` is a side-effecting function that takes three arguments, a data context, a binding set, and an enforce function; it returns a connection to resulting data. (Returning the data itself might not be reasonable in the case that it is very large and managed by a database.) When the binding set argument is a literal call to `$query` it is possible for the parser to do syntax checking between the `$query` and `$enforce`.

The `$query` produces a binding set for the source data. The `$enforce` defines how values from the binding set are used in the target population.

Figure 10: Mapping the example OWL to the relational database schema

```
 1 ( $data := $readFile('data/testing/owl-example.edn');
 2
 3   $qtype  := query($rdfType, $extraTrips)
 4               { [?class :rdf/type            $rdfType]
 5                 [?class :resource/iri        ?class-iri]
 6                 [?class :resource/namespace  ?class-ns]
 7                 [?class :resource/name       ?class-name]
 8                 // ToDo: $extraTrips
 9               };  // Defines a higher-order function, a template of sorts.
10
11   $etype  := enforce($tableType)
12              {  {'instance-of'  : 'insert-row',
13                  'table'        : $tableType,
14                  'content'      : {'resourceIRI'       : ?class-iri,
15                                    'resourceNamespace' : ?class-ns,
16                                    'resourceLabel'     : ?class-name}}
17                          }; // Likewise, for an enforce template.
18                             // The target tables for objects and relations a very similar.
19
20   $quClass := $qtype('owl/Class');     // Use the template, here and the next three assignments.
21
22   // This one doesn't just specify a value for $rdfType, but for $extraTrips.
23   $quProp       := $qtype('owl/ObjectProperty'); // ToDo: ,queryTriples{[?class :rdfs/domain ?domain] [?
       class :rdfs/range ?range]});
24   $enClassTable := $etype('ClassDefinition');
25   $enPropTable  := $etype('PropertyDefinition');
26
27   // Run the class query; return a collection of binding sets about classes.
28   $clasBsets := $quClass($data);
29
30   // We start enforcing with no data, thus the third argument is [].
31   $tar_data := $reduce($clasBsets, $enClassTable, []);
32
33   // Get bindings sets for the ObjectProperties and make similar tables.
34   $propBsets := $quProp($data);
35
36   // We pass in the target data created so far.
37   $reduce($propBsets, $enPropTable, $tar_data) // The code block returns the target data.
38 )
```

The example dataset from DOLCE is fairly large. Let's suppose it contains 50 classes, each involved in 10 relations. That suggests a collection of 500 binding sets. That does not necessarily entail 500 structures like Lines 10 through 14, however. In contrast to the JSONata-like operations, which maps physical structures to physical structures, the mapping engine here is mapping between logical structures. Specifically, a triple represents a fact and the database of triples need only represent a fact once. It is the knowledge of keys and references provided by the schema that allows the mapping engine to construct physical structure from the logical relationship defined by the mapping specification.

3.4.2 Summary of the complex mapping work process

A good approach to complex mapping tasks might start with performing `$query` on the data.

3.4.3 In-place Updates

[This part needs to be updated.] Used alone in code, `$query` does not fit the pattern of a JSONata-like capability. JSONata is effective and concise because it allow one to thread data through a pipeline of primitive transformations. If you place a `$query` call anywhere in that pipeline, you lose all the data except what is specifically captured by `$query`. There are certainly instances where that is useful, but is `$query` any easier than pure JSONata in these circumstances?[8] Perhaps the unique value of `$query`

[8]Stakeholders, do you have examples where you think `$query` might be better? Nothing comes to me, at the moment. There is also the matter of getting back an object keyed by binding variables. Thus far we haven't defined ways to manipulate that.

is as an argument to higher-level functions to do tasks such as in-place and bi-directional updating. Therefore, discussed next is (1) a new construct called `$enforce` and (2) how `$query` and `$enforce` can be used as arguments to a higher-order function `$transform` to do in-place updating.

3.5 Working with multiple sources

3.6 Working with knowledge graphs

4 Use Cases

4.1 Interoperable Mapping Tables

4.2 Integration Flows

4.3 Data Catalogs and Knowledge Graphs

4.4 Data Validation

4.5 Joint Cognitive Work / Lo-Code Tools

5 Specification

[This section will include a feature-by-feature description of the language similar to a language reference manual. Right now I've just listed the functions we think are needed.]

5.1 Language Syntax

This section is apt to be rewritten several times as the features of the language emerge.

5.1.1 JSONata Subset

[I haven't found a formal definition (e.g. BNF) for JSONata, but the following *seems to work*. Not yet complete.]

```
<content> ::= <code-block> | <exp>

<code-block> ::= '('  (<exp-or-assignment> ';')* ')'

<exp-or-assignment> ::= <exp> | <assignment>

<exp> ::= ( <filter-exp> | <binary-exp> | (<builtin-un-op> <exp>) | <paren-delimited-exp> |
            <square-delimited-exp> |<fn-call> | <literal> | <field> | <id> )
          <conditional-tail>?

<binary-exp> ::= <operand-exp> <binary-op> <exp>

<operand-exp> ::= <field> | <id> | literal | <fn-call> | <delimited> | <unary-op-exp>

<filter-exp> ::=  <operand-exp> '[' <exp> ']'

<conditional-tail> ::= '?'  <exp> ':' <exp>

<range-exp> ::= '[' <exp> '..' <exp> ']'

<fn-call> ::=  <id> '(' (<exp> (, <exp>)*)? ')'

<fn-def> ::= 'function' '(' (<id> (',' <id>)*)? ')' '{' <exp> '}'
```

```
<js-map> ::= "{" (<map-pair> (’,’ <map-pair>)*)? "}"

<map-pair> ::=  <string>" ":" <json-data>

<json-data> ::= STRING | NUMBER | <json-array>

<json-array> :: = ’[’ (<json-data> (’,’ <json-data>)*)? ’]’

<literal> ::= STRING | NUMBER | ’true’ | ’false’ | <regex> | <js-map>

<regex> (conforms to JavaScript’s implementation)

<id> (a string starting with a \$... More on this later.)
```

5.2 Table of elementary language features

Task	**Example**
Access (or navigate)	Address.PhoneNumber
Select from array (simple)	[2]
Select from array (from end)	[-2]
Select all from array	(done implicitly)
Select a subset of array	[[0..2]]
Entire input document	$
Simple query	Phone[type=’mobile’]
Select values of all fields	Address.*
Select value from any child	*.Postcode
Select arbitrarily deep	**.Postcode
Concat	’a’ & ’b’
Usual numeric operators	+,-,*,/
Array constructors	
Object Constructors	Phone{type: number}
Object Constructors (force array)	Phone{type: number[]}
Code Block	(exp1; exp2; exp3)
Value as key in result	Account.Order.Product{‘Product Name‘: Price}
Map	seq.exp
Filter	seq[exp]
Reduce (group and aggregate)	seq{exp:exp, exp:exp. . . }
Sort	seq ˆ(exp)
Index	seq # $var
Join	seq @ $var
Functions (in-line)	function($x) : {(. . .)};
Naming functions	$myfunc = function$(x). . .
Context variable	$
Root context variable	
Conditional expression	pred ? exp : exp
Variable binding	$my_var := ”value”
Function chaining	value ∼> $f2-$ >f3
Regular expressions (like JS)	/ab123/
Time	now(),millis()
Transform	head ∼>— location — update [,delete] —
Example variable binding	[needs investigation]
Type coersion	asType(obj, typespec)
Type checking	isTypeOf(obj, typespec)
Coersion to integer	toInteger(x)
Coersion to real	toReal(x)
Returning a substring	substring()