Draft OAGi Interoperable Mapping Specification

OAGi Members

2022/12/01

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 specifications used by commercial mapping tools.

RADmapper, the mapping language, integrates JSONata expression language [1] with Datalog [2] capabilities. It currently supports mapping to/from JSON, XML, tables (e.g. Excel), and knowledge graphs (RDF). Use cases for the RADmapper language including in-place updating of target data sources, and mapping from multiple sources.

As of this writing, a strategy for interoperable exchange of mapping specifications is under development. There is also on-going exploratory work for RADmapper to support streaming data. These two topic are currently not discussed. The reference implementation of the RADmapper language can be found in the Github repository https://github.com/pdenno/RADmapper. An web-based "exerciser" tool to explore the language is available as a Docker image https://hub.docker.com/r/podenno/rm-exerciser. The document is a draft and, as of this writing (2022/12/01), is likely to be updated often, as will the language implementation and exerciser.

2 Quick Start: Example Mapping Tasks

This section uses examples to describe the basic features of the RADmapper language. RADmapper provides the complete expression language and all the built-in functions of JSONata. Examples of JSONata can be found in the JSONata specification. This section only goes into detail about the capabilities of RADmapper not found in JSONata. The principal concepts to discuss are

- the relational and graph forms of data,
- RADmapper's `query` declaration, used to query relational data,
- RADmapper's `express` declaration, used to reorganize ("map") data from its original form to the form in which it is needed, and
- RADmapper's strategy for interoperability. [Not yet!]

2.1 Data Organized as Triples and the query Construct

`query` is the principal construct of RADmapper providing Datalog-like functionality to the language. `query` declarations are used like JSONata or Javascript function declaration in the sense that the value of the declaration (a function) can be assigned to variables and used directly. For example,

```
$addOne := function(x){x + 1}
```

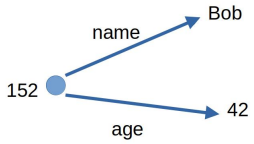
defines a function and assigns it to the variable `$addOne`. `$addOne(3)` is a call to the function with the argument 3. Similarly,

```
$myQuery := query(){[$DB1 ?person :age 42]}
```

defines a query that can be used like a function. `$myQuery($myDB)` is a call to the function with whatever "database" is assigned to `$myDB`. However, unlike ordinary functions, the body of a `query` consists of one or more *Datalog patterns* such as the pattern `[$DB1 ?person :age 42]` shown. We'll start by talking about the database argument to the query, for example, the value assigned to `$MyDB` in the expression `$myQuery($MyDB)` then get back to talking about the patterns.

Relational (table-based) and graph-based data can be described by triples $[x, rel, y]$ where x is an entity reference, y is data (string, number, entity reference, etc.) and rel is a relationship (predicate) holding between x and y. For example, in syntax similar to JSON we could describe the fact that Bob's age is 42 with an object {`'name' : 'Bob', 'age' : 42`}. As triples, we represent Bob being 42 years old with two triples, for example, `[152 'name' 'Bob']` and `[152 'age' 42]`. As a graph, this would look like the following:

Figure 1: Information about Bob in graph form



You might wonder where the 152 in the triples came from, or for that matter, why the fact that Bob being age 42 couldn't simply be represented by the single triple `['Bob' 'age' 42]`. The answer is that you could represent this fact with that one triple, but in doing so you are using `'Bob'` as a key which might not be that good if your database has more than one Bob in it. So instead, to prepare for the more general case, Datalog-like graph databases use unique integers to refer to entities. 152 here is like a primary key in relational DB, or an IRI for a particular entity defined in RDF. A complete RADmapper program for querying the Bob database for people age 42 is as follows:

Figure 2: A complete query

```
1  ( $myDB := [{'name' : 'Bob', 'age' : 42}];
2    $ageQuery := query(){[?e :age 42]};
3    $result := $ageQuery($myDB); )
```

The above needs some explanation. On Line 1 we put a JSON-like object in an Javascript-like array by wrapping it in square brackets; this is the literal form of a very small database. On Line 2 we defined the query function; here note the use of a *query variable* `?e`. Also note that whereas we used the string `'age'` for the relation, we used `:age` — syntax we call an *attribute* — to represent that same relation in the pattern. On Line 3 we apply the query function `$ageQuery` to the database `$myDB` defined on Line 1. Of course, `$myDB` isn't much of a database; it contains only the data shown in Figure 1. Alternatively, you could have defined data through other means, such as reference to a file or a GraphQL query. The result of this query, assigned to `$result` is a *binding set*, a set of objects each describing a consistent binding of the search pattern's variables to values from the database. In this example, with the data in Figure 1, the binding set consists of just one binding element and it binds one variable; the binding set is `[{?e : 152}]`. This indicates that the entity indexed at 152 has an attribute `:age` with value 42. If there were more entities in the database possessing an `:age` attribute, the query would have returned one such {`?e : <whatever>`} binding object for each of them.

Amazing as that query might seem, running against a database with one entity in it and all, it didn't tell us *what person* is age 42. All we got was a binding set for every entity that has an age attribute equal to 42. Entity IDs are internal to the database and not of much value to RADmapper

users.[1] In order to make any use of this information, we need to join the `[?e :age 42]` pattern with a pattern that grabs the name attribute in the data, `[?e :name ?name]`. This is shown in Figure 3 below.

Figure 3: A more useful query, getting the name of the 42-year old person

```
1  ( $myDB := [{'name' : 'Bob', 'age' : 42}];
2    $ageQuery := query(){[?e :age 42]
3                         [?e :name ?name]}
4    $result := $ageQuery($myDB); )
```

The pattern on Line 3, by virtue of its reuse of `?e`, imposes an additional constraint on the graph match: the entity bound to `?e` must have a `:name` attribute. The value of the name attribute is bound to `?name`. Thus each element in the binding set will bind two variable, `?e` and `?name`. The binding set for the database in the graph depicted in Figure 1 is `[{?e : 152, ?name 'Bob'}]`.[2]

In this example, we used one variable to match on the entity and another to capture a value, however each position (entity, attribute, and value) can take a variable. Further, you can use variables in more than one of those positions. For example, `query(){[?entity ?attr ?val]}` is a query that matches on every entity, attribute, and value of the database; it represents every edge of the database's graph. `query{[_ ?attr _]}` would return the names of all the attributes in the database (without duplicates). This provides a kind of introspection that is typically more difficult to obtain in other database technology. When matching the value position you are doing a relational join, for example, we might match the social security number (SSN) in some data against a same-valued (but possibly differently named) value in other data. For example,

```
$relJoinQuery := query(){[$DB1 ?e1 :ssn ?id]
                         [$DB2 ?e2 :id  ?id]}
```

You may have noticed that the query above, and one used earlier have four elements in their pattern whereas most of the examples have only three (entity, attribute, and value). If four elements are provided, the first is the database to which the pattern is applied. If there is only one database being queried, as we've been doing earlier, you don't need to use four-place patterns. Let's look at a complete `query` example that uses two databases.

Figure 4: A query that looks into two databases

```
1  ( $DBa := [{'email' : 'bob@example.com', 'aAttr' : 'Bob-A-data',   'name' : 'Bob'},
2             {'email' : 'alice@alice.org', 'aAttr' : 'Alice-A-data', 'name' : 'Alice'}];
3    $DBb := [{'id'    : 'bob@example.com', 'bAttr' : 'Bob-B-data'},
4             {'id'    : 'alice@alice.org', 'bAttr' : 'Alice-B-data'}];
5
6    $qFn :=  query(){[$DBa ?e1 :email ?id]
7                     [$DBb ?e2 :id    ?id]
8                     [$DBa ?e1 :name  ?name]
9                     [$DBa ?e1 :aAttr ?aData]
10                    [$DBb ?e2 :bAttr ?bData]};
11
12   $bSet := $qFn($DBa, $DBb);  )
```

In Figure 4 Lines 1–4 we define two small databases and assign them to variables `$DBa` and `$DBb` respectively. On Lines 6–10 we define the query. Since both databases use email addresses for customer identification, we can use the `:email` and `:id` attributes to join together information about a customer from the two databases. That is the purpose of the patterns on Lines 6 and 7; the two patterns use different variables for the entities, `?e1` and `?e2`, because the information is coming from different databases and we don't control entity IDs, but both use `?id` to force matches on email address. The

[1] In fact, I sort of told a fib for ease of exposition; by default the binding of variables in the entity position, entity IDs, aren't provided in a binding object. Using default settings, what this example returns is `[{}]` meaning "one match was found." The binding sets depicted in most examples won't include bindings for entity IDs.

[2] If you read the previous footnote you know it actually returns just `[{?name 'Bob'}]`.

remainder of the patterns in the query, Lines 8–10, pick up various information from the two databases. Line 12 calls the query function bound to `$bSet` to get the binding sets against the two databases. Unlike our previous calls to the query function, this one takes two databases as arguments. The order of the arguments in the call must be the same as the order in which the databases appear in the query statement; `$DBa` appears first on Line 6, `$DBb` appears first on Line 7, so `$DBa` is the first argument to the call.

The binding set that is produced, the value of `$bSet`, consists of two binding objects:

```
[{?id : "bob@example.com", ?name : "Bob",   ?aData : "Bob-A-data",   ?bData : "Bob-B-data"  },
 {?id : "alice@alice.org", ?name : "Alice", ?aData : "Alice-A-data", ?bData : "Alice-B-data"}]
```

One small but very significant point before moving on to discuss `express`: writing queries like `query(){[?e :age 42] [?e :name ?name]}` could become rather tedious in the case that you might want to get data about some other age value. For this reason, the `query` declaration can serve to produce a *higher-order function*, a function that returns (query) functions as values. Figure 5 demonstrates the idea.

Figure 5: Get the names of people ages 42 and 33.

```
1  ( $myDB := [{'name' : 'Bob',   'age' : 42},
2              'name' : 'Alice', 'age' : 33}]
3
4    $ageQueryT := query($age){[?e :age  $age]
5                              [?e :age  ?age]
6                              [?e :name ?name]}
7
8    $ageQ42    := $ageQueryT(42);
9    $ageQ33    := $ageQueryT(33);
10
11   $append($ageQ42($myDB) , $ageQ33($myDB))  )
```

The key difference is that on Line 4, the `query` construct, defines a parameter, `$age`, using ordinary JSONata-like syntax. You can define as many parameters as you'd like and they can be used to substitute into any of the pattern positions, entity, attribute, or value. By convention, if we are to assign a higher-order `query` function to a variable, we end the variable name with a `T` such as shown on Line 4, `$ageQueryT`. The T suggests that the variable denotes a *query template*. Lines 8 and 9 define query functions for querying ages 42 and 33, respectively. Line 11 uses the JSONata-like builtin `$append` to combine the two binding sets. Note that the pattern `[?e :age ?age]` on Line 5 is used get ?age into the binding sets. The result of running this example is the binding set

```
[{?name : 'Bob',   ?age : 42},
 {?name : 'Alice', ?age : 33}].
```

2.2 Constructing Target Data with express

Binding sets produced by `query` provide ordinary JSONata-like objects[3] that could be used with JSONata built-in functions and operators to produce target structures. However, the `express` construct provides capabilities beyond those of the JSONata expression language. This section describes some of those features.

Let's continue with our two-database example. Lines 1–12 of Figure 6 below are as they were in Figure 4. The binding set assigned to `$bSet` is

```
[{?id : "bob@example.com", ?name : "Bob",   ?aData : "Bob-A-data",   ?bData : "Bob-B-data"  },
 {?id : "alice@alice.org", ?name : "Alice", ?aData : "Alice-A-data", ?bData : "Alice-B-data"}].
```

Assuming that we'd just like to present the target data as nested objects indexed by the customer's email address, for example:

[3]Two differences: (1) the keys of binding sets are query variables, and (2) binding sets are flat objects; the values at the keys are not objects themselves though they may be entity IDs (integers). These differences notwithstanding, you can use binding sets as though they are ordinary JSONata-like objects.

```
{"alice@alice.org" {"name"  : "Alice",
                    "aData" : "Alice-A-data",
                    "bData" : "Alice-B-data" },

 "bob@example.com" {"name"  : "Bob",
                    "aData" : "Bob-A-data",
                    "bData" : "Bob-B-data"   }}
```

we could use the `express` declaration that begins on Line 14 of Figure 6 to do this.

Figure 6: Using express with a query that looks into two databases

```
1  ( $DBa := [{'email' : 'bob@example.com', 'aAttr' : 'Bob-A-data',   'name' : 'Bob'},
2            {'email' : 'alice@alice.org', 'aAttr' : 'Alice-A-data', 'name' : 'Alice'}];
3    $DBb := [{'id'    : 'bob@example.com', 'bAttr' : 'Bob-B-data'},
4            {'id'    : 'alice@alice.org', 'bAttr' : 'Alice-B-data'}];
5
6    $qFn := query(){[$DBa ?e1 :email ?id]
7                    [$DBb ?e2 :id    ?id]
8                    [$DBa ?e1 :name  ?name]
9                    [$DBa ?e1 :aAttr ?aData]
10                   [$DBb ?e2 :bAttr ?bData]};
11
12   $bSet := $qFn($DBa, $DBb);
13
14   $eFn  := express(){{?id : {'name'  : ?name,
15                              'aData' : ?aData,
16                              'bData' : ?bData}}};
17
18   $reduce($bSet, $eFn) )
```

`express` is a function-defining construct that, like `query`, is capable of returning express functions when supplied with parameters. In that sense it is capable of being a higher-order function. But here on Line 14 we aren't using parameter; the declaration is simply `express(){...}`; no parameters imply it isn't a template and can be used directly. The body of the `express` declaration, the text inside the outer curly brackets, defines the pattern of JSONata-like object structure. The target data is produced by iterating over the express function bound to `$eFn` using the elements of the binding set. Two of the most common ways to iterate over a function in functional languages such as JSONata and RADmapper are map and reduce. Map and reduce differ in how they process the collection of arguments: map applies a given function to each element independently; reduce "summarizes" results by allowing each element to affect a summary outcome.[4] Whether you `$map` or `$reduce` the `express` function over the binding set can have great influence on the outcome, as will be demonstrated in subsequent examples. As written above, the call to `$reduce` on Line 18 creates a nested structure for the first binding element and then inserts a second structure into it at the second `?id` key, as shown above. Were Line 18 replaced with `$map($bSet, $eFn)`, the result would be two independent nested maps:

```
[{"bob@example.com" : {"name" : "Bob",   "aData" : "Bob-A-data",  "bData" : "Bob-B-data"  }},
 {"alice@alice.org" : {"name" : "Alice", "aData" : "Alice-A-data","bData" : "Alice-B-data"}}].
```

2.3 Predicate patterns and an example from a different perspective

This example illustrates (1) simple restructuring of a data structure, (2) use of predicates as query patterns, (3) extensive joining to navigate deeply nested structures, and (4) use of query variable in the attribute position of patterns. The example is based on a discussion on the JSONata Slack channel. The goal of the example is to swap the nesting of `owners` and `systems` as shown in Figure 7.

[4] Examples to help you recall how these work:
`$map([1,2,3,4], function($x){$x * 2})` returns [2,4,6,8]; it multiplies each argument by 2.
`$reduce([1,2,3,4], function($x, $y){$x + $y})` returns 10; it applies + to 1 and 2, then to 3 and that sum, then to 4 and that sum.

Figure 7: The goal is to swap the nesting of 'owners' and 'systems' in the data on Lines 1–9 so that it looks Lines 13–21.

```
1 { "systems":
2    { "system1": { "owners": { "owner1": { "device1": { "id": 100, "status": "Ok" },
3                                           "device2": { "id": 200, "status": "Ok" }},
4                               "owner2": { "device3": { "id": 300, "status": "Ok" },
5                                           "device4": { "id": 400, "status": "Ok" }}}},
6      "system2": { "owners": { "owner1": { "device5": { "id": 500, "status": "Ok" },
7                                           "device6": { "id": 600, "status": "Ok" }},
8                               "owner2": { "device7": { "id": 700, "status": "Ok" },
9                                           "device8": { "id": 800, "status": "Ok" }}}}}
10
11  /* ...so that it looks like: */
12
13 {"owners":
14    {"owner1": {"systems": {"system1": {"device1": {"id": 100, "status": "Ok"},
15                                        "device2": {"id": 200, "status": "Ok"}},
16                            "system2": {"device5": {"id": 500, "status": "Ok"},
17                                        "device6": {"id": 600, "status": "Ok"}}},
18     "owner2": {"systems": {"system1": {"device3": {"id": 300, "status": "Ok"},
19                                        "device4": {"id": 400, "status": "Ok"}},
20                            "system2": {"device7": {"id": 700, "status": "Ok"},
21                                        "device8": {"id": 800, "status": "Ok"}}}}}}}}}
```

Though these structures, rendered as JSON, may look odd (for example, there are no arrays used despite apparent value in having them[5]) this is not relevant to the discussion. Typical of a restructuring task, the goal data on Lines 13–21 does not contradict any of the facts evident in the original data on Lines 1–9. For example, there is an owner called `owner1`, and `owner1` still has the same devices associated. The idea that mapping is about how facts are viewed is a key concept in RADmapper. What has changed in restructuring the example data is not a fact but the kind of forward navigation that is possible in the two structures. In the original structure, it is possible to navigate forward from a system to an owner (such as on Line 1, from `system1` to `owner1`). In the restructured data, it is possible only to navigate forward from owner to system (such as `owner1` to `system1` on Line 14). The restructuring task can be achieved using only JSONata functions and operators as depicted in Figure 8.

Figure 8: Using JSONata to restructure the data.

```
1 {
2     "owners": $distinct(systems.*.owners.$each(function($d, $ownerName) {$ownerName}))@$o.{
3       $o: $each($$.systems, function($sys, $sysName) {{$sysName: $lookup($$.systems, $sysName).owners ~>
        $lookup($o)
4       }}) ~> $merge()
5     } ~> $merge()
6 }
```

Though the RADmapper solution to this problems involves more lines of code (see Figure 9 below), it is arguably easier to understand. The RADmapper solution is also arguably a better candidate for *interoperable* exchange of mappings, as will be discussed later.

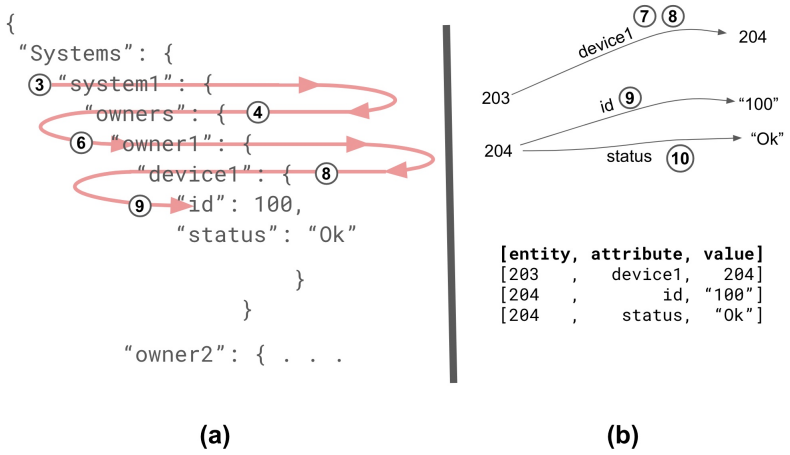
[5]For example, several devices are associated with an owner, but instead of arrays, the developer used meaningful names as keys, "device1","device2" etc. Perhaps these structures only seems unusual if your perspective involves the object/attribute/value patterns that are typical of messaging structures.

Figure 9: RADmapper `query` and `express` used to restructure data. Note that on Line 15 we use `express` in-lined rather than assigning that function to a variable and passing the variable into `$reduce`. The choice to in-line has no effect on the result.

```
1   (   $data := $read('data/testing/jsonata/sTPDRs--6.json');
2       $q := query(){ [?s ?systemName ?x]
3                      [($match(?systemName, /system\d/))]
4                      [?x :owners ?y]
5                      [?y ?ownerName ?z]
6                      [($match(?ownerName, /owner\d/))]
7                      [?z ?deviceName ?d]
8                      [($match(?deviceName, /device\d/))]
9                      [?d :id ?id]
10                     [?d :status ?status] };
11
12    $bsets := $q($data);
13
14    $reduce($bsets,
15            express() {  {'owners':
16                           {?ownerName:
17                             {'systems':
18                                {?systemName:
19                                   {?deviceName : {'id'     : ?id,
20                                                   'status' : ?status}}}}}}
21                        }
22          )
23  )
```

Where this example differs from earlier ones is the use of `$match` on Lines 3, 6 and 8. Instead of the usual 3- or 4-place pattern, we have a single *predicate* that is being applied using query variables from other patterns in the query. `$match` is a JSONata built-in Boolean function that returns true if the first argument, a string, matches the second argument, a regular expression. This example is also the first to use extensive joins to navigate into deeply nested objects. A "hairpin" pattern of joined variables is depicted in Figure 10 and reflected in Lines 2–10 of Figure 9.

Figure 10: Part (a): navigation of the source structure (Lines 1–9 of the data in Figure 7). Part (b): graph and triples representing device1. The circled numbers refer to lines in the code of Figure 9.



A Datalog database can be viewed as several index tables about the triples (such as those depicted in Figure 10 (b).) One such index table will index primarily by entity, another primarily by attribute, etc. Some of Lines 2–10 in Figure 9, for example, Line 2, `[?s ?systemName ?x]`, have the form of these triples, where respectively `?s`, `?systemName` and `?x` match an entity, attribute, and value. Line 4, `[?x :owners ?y]`, is similar but the attribute position is occupied by `:owners`. This triple will match any fact that has the string `"owners"` in its attribute position. There are two such facts in the data

depicted in Figure 7, one on Line 2, and one on Line 6. In both cases the value position is occupied by another entity reference. Figure 10(b) similarly depicts a triple whose value position references another entity, `[203, device1, 204]`.

Lines 3, 6 and 8 of the query do not use the 3-place triple pattern; they consist of a single JSONata expression wrapped in parentheses. The expression should be a Boolean (should return true or false). Note that the expression uses variable bindings (e.g `?systemName`, `?ownerName`, and `?deviceName`) used in triples of the query. `/system\d/` is a JavaScript regular expression matching a string containing `"system"` followed by a single digit (the `\d` part). The following paragraph provides a line-by-line summary of how the query in Lines 2–10 works.

Line 2 `[?s ?systemName ?x]`: All positions of this triple pattern are occupied by variables, so by itself it would match every triple in the database.

Line 3 `[($match(?systemName, /system\d/))]`: This uses the query variable `?systemName` bound to the entity position on Line 2. Therefore, between this pattern and the one in Line2, the only triples matching both of these patterns have "system1" or "system2" in the attribute position. (See the data to verify this.) `?s` and `?x` from Line 2 are thus bound to the entity and value positions of triples having either "system1" or "system2" in their attribute positions.

Line 4 `[?x :owners ?y]`: Here we see `?x`, which was introduced in Line 2, reused in the entity position. We know that `?x` is bound to an entity by looking at the data. (See Line 2 in the data for example, `''owners'': {. . . .` The "`: {`" means the thing keyed by "owners" here is a JSON object (an "entity" in the database). This triple pattern ensures that `?x` refers to entities for with the "owners" occupies the attribute position.

Line 5 `[?y ?ownerName ?z]`: Like Line 2, this is used to introduce an new variable, `?ownerName` in this case, which will be used in the a `$match` predicate similar to Line 3. One difference here, however, is that the value of `?y` is constrained by the triple pattern on Line 4.

Line 6 `[($match(?ownerName, /owner\d/))]`: This serves a similar purpose Line 3, but for owner keys.

Line 7 `[?z ?deviceName ?d]`: This is like Lines 1 and 5, introducting a new variable for use in the `$match`. Like Line 5, the value of the variable in entity position is constrained by another pattern.

Line 8 `[($match(?deviceName, /device\d/))]`: This is similar in purpose to the other two uses of `$match`.

Line 9 `[?d :id ?id]`: Here we are finally ready to pick up some user data, the value of a device's "id" attribute.

Line 10 `[?d :status ?id]`: Like Line 9, but to pick up the value of a device's "status" attribute.

It was suggested earlier that the encoding of the source data was a bit unusual. For example, object key values such as `owner1`, `system1` and `device1` are being used, apparently, both to indicate a unique entity and to identify the kind of entity being described, presumably an owner, system and device respectively. We might seek target data that explicitly declares the type and identifier separated, for example using attributes `type` and `id` respectively. The target data sought, for example, might be as depicted in Figure 11.

Figure 11: An alternative organization of the target data from the example.

```
[{"type"   : "OWNER",
  "id"     : "owner1",
  "systems": [{"type"   : "SYSTEM",
               "id"     : "system1",
               "devices": [{"type"   : "DEVICE",
                            "id"     : "100",
                            "status" : "Ok"},
                           {"type"   : "DEVICE",
                            "id"     : "200",
                            "status": "Ok"}]},
              {"type"   : "SYSTEM",
               "id"     : "system2",
               "devices": [{"type"   : "DEVICE",
                            "id"     : "500",
                            "status" : "Ok"},
                           {"type"   : "DEVICE",
                            "id"     : "600",
                            "status": "Ok"}]}]},
 {"type"   : "OWNER",
  "id"     : "owner2",
  "systems": [{"type"   : "SYSTEM",
               "id"     : "system1",
               "devices": [{"type"   : "DEVICE",
                            "id"     : "300",
                            "status" : "Ok"},
                           {"type"   : "DEVICE",
                            "id"     : "400",
                            "status": "Ok"}]},
              {"type"   : "SYSTEM",
               "id"     : "system2",
               "devices": [{"type"   : "DEVICE",
                            "id"     : "700",
                            "status" : "Ok"},
                           {"type"   : "DEVICE",
                            "id"     : "800",
                            "status": "Ok"}]}]}]
```

As the figure depicts, the output is more verbose, and only in Hollywood do the colons line up in the columns like that. A guess at an `express` structure to produce this output is as follows.

Figure 12: Draft express structure for target data as depicted in Figure 11

```
$ex1 := express{{'type'   : 'OWNER',
                 'id'     : ?ownerName,
                 'systems': [{'type'   : 'SYSTEM',
                              'id'     : ?systemName,
                              'devices': [{'type'  : 'DEVICE',
                                           'id'    : ?deviceName,
                                           'status': ?status}]}]}
              }
```

If we were to map over that `express` body, that is `$map($bsets,$ex1)`, the result would be (logically correct, of course but) even more verbose and also repetitive because there will be one full structure like the `express` body for each of the 8 binding sets. A better alternative might be to `$reduce` over the `express`, that is, `$reduce($bsets,$ex1)`; `$reduce` can have the effect of "summarizing" data, in this case by squeezing out the repetition. However, there is a problem with reducing over the `express` body as written; it would produce the following:

Figure 13: Reducing over the `express` body here results in iteratively overwriting data, so that the result is the same as applying `express` body to just the last binding set.

```
1  {"type"    : "OWNER",
2   "id"      : "owner1",
3   "systems" : {"type"    : "SYSTEM",
4                "id"      : "system1",
5                "devices" : {"type"   : "DEVICE",
6                             "id"     : "device2",
7                             "status" : "Ok"}}}
8
9     /* This result indicates that the last binding set processed was the following */
10    {?ownerName : "owner1", ?systemName : "system1", ?deviceName : "device2", ?status : "Ok", ?id : 200}
```

What happened to the rest of the data? Why were data not lost similarly when we reduced over the `express` body on Lines 14–20 of Figure 9? The answer is that the `express` body of Figure 9 uses unique values for the object keys. The effect of reducing over the `express` body in that example is to "drop in" new pathways for each binding set. Specifically, if we consider the Lines 13–21 of Figure 7, the result of applying the `express` body on Lines 14–20 of Figure 9 to the key values in each corresponding binding set, it is apparent that:

- the Line 14 object was created by the keys `"owners"`, `"owners1"`, `"system1"`, `"device1"`,
- the Line 15 object was created by the keys `"owners"`, `"owners1"`, `"system1"`, `"device2"`,
- the Line 16 object was created by the keys `"owners"`, `"owners1"`, `"system2"`, `"device5"`, and
- et cetera, to Line 21, created by the keys `"owners"`, `"owners2"`, `"system2"`, `"device8"`.

There are then 3 solutions to the problem just highlighted: (1) use unique keys to "drop in" new pathways like in the original example, (2) use `$map` instead of `$reduce`, which entails accepting the fact that the result will be verbose with lots of repetition, or (3) wrap keys in the `key` construct as described below.

2.3.1 The key construct of express

We'll continue with the running example. In order to prevent the overwriting that was depicted in Figure 13 we will adapt the code from Figure 12 by adding the `key` construct as shown.

Figure 14: The express structure from Figure 12 revised to identify keys

```
1  $ex1 := express{{'owners' : {'type'    : 'OWNER',
2                               'id'      : key(?ownerName),
3                               'systems': [{'type'    : 'SYSTEM',
4                                            'id'      : key(?systemName),
5                                            'devices': [{'type'  : 'DEVICE',
6                                                         'id'    : key(?deviceName),
7                                                         'status': ?status}]}]
8                           }
9                    }
10                  }
```

There are two significant modification to the orginal `express` body. First, Lines 2, 4, and 6 wrap the respective binding variables `?ownerName`, `?systemName`, and `?deviceName` in the `key` construct to declare identity conditions for the corresponding objects. This will ensure that binding sets on paths with those keys insert new data, not overwrite existing data. Second, compared to the orginal code, the whole body is now wrapped in {`'owners ...`}. This ensures that calls to reduce have a common root on which to grow the structure.

Incidentally, the situation that we just handled is similar to the choice between updating and inserting data. When we declare a field a key, we are saying that other values for that field necessarily

refer to other conceptual objects, and therefore the data should be inserted, not used to update the existing object.

2.4 Summary so far

2.5 Don't bother to read past here.

2.6 Interoperability?

[Discuss simple examples of using ASTs and mapping to "mapping tables". 90 percent of use cases don't need this.]

2.7 Section Summary and Additional Thoughts

RADmapper's `query` is based on Datalog, a language which was studied extensively in the 1980s [2] and has influenced languages such as SPARQL and the Shapes Constraint Language (SHACL) [**?**]. `query` has the power of SQL's query but does operates fact-at-a-time versus SQL's entity-at-a-time. That distinction entails, for example concerning the previous example, that in Datalog the facts (1) "`device1 id` is `100`." and (2) "`device1 status` is `Ok`." are independent assertions that happen to be about the same entity, `device1`. In contrast, SQL's entity-at-a-time design entails that there is a table about devices and the table row at primary key `device1` has information about both `id` and `status`. There are advantages and disadvantages to both Datalog and SQL. For example, it should be apparent that pulling together all the information about an entity with Datalog queries requires relational joins, one each for each fact. On the other hand, RADmapper can learn Datalog-like schema by studying the data, it can create databases in milliseconds, and adding new attributes (columns in SQL) does not require data migration in Datalog.

The order of the patterns in a `query` statement does not matter at all; it does not matter with respect to the execution speed nor the result produced. However, in order to illustrate the chaining of joins being performed, it is useful to the readers of your code to keep joined things together. This is illustrated, for example, by the curvy red line in Figure 10(a).

2.8 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 15.

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 15: Table 1 viewed as an array of map structures.

```
1
2 [{"ShipDate":  "2021-06-15", "Item":  "Widget 123", "Qty": 1.0, "UnitPrice": 10.5},
3  {"ShipDate":  "2021-06-15", "Item":  "Gadget 234", "Qty": 2.0, "UnitPrice": 12.8},
4  {"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 Datalog features of RADmapper

RADmapper provides a superset of the functionality of JSONata, but more importantly, it supports a second paradigm for navigating data and some very different usage scenarios. 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, 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. RADmapper, and particularly its *AST strategy* described in Section **??** is targeted towards new integration scenarios that involve higher levels of automation and joint (human/AI) cognitive work.

3.1 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.

3.1.1 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 16.

Figure 16: 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 17 depicts the static tables in typical relational DDL. These are populated by the ontology `owl/Class` content of the Figure 18 depicts tables that would be created.

Figure 17: Static DDL for storing class and object relation metadata. The mapping will generate information equivalent to DML to populate this from the source data.

```
1   CREATE SCHEMA typicalOWL;
2
3   CREATE TABLE ObjectDefinition
4     (resourceIRI       VARCHAR(300) primary key,
5      resourceLabel     VARCHAR(300) not null,
6      resourceNamespace VARCHAR(300) not null);
7
8   CREATE TABLE ClassDefinition
9     (resourceIRI VARCHAR(300) primary key,
10     subClassOf  VARCHAR(300) references ClassDefinition);
11
12  CREATE TABLE ObjectClass
13    (resourceIRI VARCHAR(300) primary key,
14     class       VARCHAR(300) references ClassDefinition);
15
16  CREATE TABLE DisjointClass
17    (disjointID  INT primary key,
18     disjoint1   VARCHAR(300) not null references ObjectDefinition,
19     disjoint2   VARCHAR(300) not null references ObjectDefinition);
20
21  CREATE TABLE ResourceComment
22    (commentID    INT primary key,
23     resourceIRI  VARCHAR(300) not null references ObjectDefinition,
24     commentText  VARCHAR(900) not null);
25
26  CREATE TABLE PropertyDefinition
27    (resourceIRI    VARCHAR(300) primary key,
28     relationDomain VARCHAR(300) references ObjectDefinition,
29     relationRange  VARCHAR(300) references ObjectDefinition);
```

Figure 18 depicts the result of mapping data from Figure 16 using a mapping specification that will be described below.

Figure 18: Result of mapping the data depicted in Figure 16. 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 19 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 express 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 `express`.

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

Figure 19: Mapping the example OWL to the relational database schema

```
1  (   $data := $read('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.

3.1.2 Summary of the complex mapping work process

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

3.1.3 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?[6] Perhaps the unique value of `query`

[6]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 `express` and (2) how `query` and `express` can be used as arguments to a higher-order function `$transform` to do in-place updating.

3.2 Working with multiple sources

3.3 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.)
```

References

[1] Jsonata.org. JSONata: query and transformation language, 2021.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, Reading, MA, 1995.