Anaphors or Scriptlets

In computer languages a snippet of code that refers to something else is called an *anaphoric macro* (from Greek ἀναφορά for "carrying back")[1]. QDL may be used as a server-side scripting language (*e.g.* OA4MP uses it extensively). A single instance is called an ***anaphor*** or ***scriptlet***.

What problem does this solve?

A typical case is that a server has been extended with QDL, so that QDL scripts may be called and QDL then may be used to extend the server. QDL has a runtime engine (in Java) that can be invoked and a well-defined Java API for passing anaphors in and getting results back. Since we do not want to require someone to rewrite an existing server to be QDL-aware, this is a translation layer that either has a reference (or more) to a QDL script and the arguments. This is all done in JSON, which is reasonably enough widely used that most servers can at least read it and pass it along.

Since QDL should not be aware of the server, there is a neutral element called the execution metadata (or xmd – see below) that is simply a JSON object the server can manage that contains the information it needs to know when to invoke its QDL runtime engine.

Structure

This is the format for an anaphor used by QDL . This is an element in JSON and may be put any place. This is the grammar:

```
{"qdl" :
   BLOCK | [BLOCK+], // Either a block or array of blocks
   }
```

```
BLOCK :
{
  CODE
   [, ARGS]

}
```

1The pointing back reference is called an ***anaphor*** and the entity to which it refers is its ***antecedent***. E.g., “Beware the Jabberwock, my son! The jaws that bite, the claws that catch, *he* has eyes of flame!” In this case, *he* is the anaphor, since it refers back to the antecedent, the Jabberwock. Cf. ***cataphors*** which refer to things ahead, *e.g.*, “If you want *some*, there's coffee in the pot.” where *some* is the cataphor. QDL does not use cataphors. Then there are *metaphors*, which QDL also does not use, but the author, ever interested in getting his ducks on the same page, mixes badly.

Ergo., the JSON blobs in this article are references to scripts or QDL code outright. They are the anaphors which reference antecedents, existing QDL scirpts.

```
CODE:
  ("load" : LINE) | ("code"  : LINE+)

ARGS:
  "args" : ARG | [ARG+] // Either a single arg or an array of them

ARG:
   STRING | INTEGER | DECIMAL | BOOLEAN | JSON

JSON:
  Any JSON object
```

load

The name of a QDL script. The QDL runtime is aware of any configured virtual file systems and the script path set in QDL will be scanned.

code

Actual lines of QDL code. These are passed to the runtime engine and executed. Generally you should load a script and use that, but sometimes a single line of QDL is all that is needed. If you need more than a few lines in a code block, consider moving it to a proper script.

args

A JSON object of arguments. These will be turned into a stem and passed to the script. A list will have each element passed as an argument to the script. Anything else will be passed as a single argument. You may pass JSON elements in a list and these will be converted to stems and passed as arguments.

Example 1

```
{"qdl":
   {
    "load" : "scripts/utils/db_setup.qdl"
    "args" : ["bob", "my_password", 421]
   }
}
```

In this case, a script named **`db_setup.qdl`** will be run and the given arguments will be passed in, each as a separate argument. So in the script, **`args().0 == 'bob'`**, etc.

Example 2

```
{
 "qdl":
   {
    "load" : "scripts/utils/db_setup.qdl"
    "args" : {"username":"bob", "password":"my_password", "port":421}
   }
}
```

In this case, it is identical to Example 1, except for the **args** element, which has a single stem passed, so
args().0 == {'username':'bob',...}

Example 3, with additional elements

```
{
 "qdl":
   {
    "load" : "fnal/fnal-idtoken.qdl",
     "xmd" : {"exec_phase":"post_token"}
   }
}
```

Implementors are free to add more elements to anaphors to help them manage the state. In this case, there is an **xmd** element (from e**X**ecution **M**eta**D**ata). This has a given script to run, but no **args**. There are no arguments and the server would read the **xmd** and decide whether or not to call its QDL runtime engine on this. Again, QDL does not specify extra elements, the contract is that it ignores them but will faithfully transmit them if needed so they are not lost.

The Anaphors module

This is a module with tools that make running scripts easy. It is in the standard distribution.

```
   a := import(load('anaphors'))
   )funcs a
exec([1])  json_in([1])  simple_exec([1])  to_code([1])  to_load([1,2])
5 total functions
```

This loads the anaphors module and lists the functions. You can list help for the module with

```
   )help -m a
 This module defines several functions for working with scriptlets
 aka anaphora aka anaphoric macros.
 An anaphor or scriptlet is a JSON representation
 . . .lots more
```

and of course help for each function by *e.g.*,

```
   )help a#exec 1
 exec(x) - execute a generic anaphor (includes lists of them).
 x is one of
   * the JSON string for this anaphor
   * The full anaphor,e.g. {'qdl':{'load':....}}
   * The executable part of the anaphor, e.g. {'load:'...}
     or a list of these
 The result is the result of running the anaphor.
```

Example of to_load

in this example, we will run the included QDL script echo2.qdl (in the examples directory of the standard distribution). This simply returns the arguments so you can see how it all works.

First, make an anaphor (if your script_path is set to include the example directory, you do not need to include it. Here we construct the path in toto just because, from the system info

```
   echo:=info().'boot'.'qdl_home' + '/examples/echo2.qdl'
   echo
/home/ncsa/apps/qdl/examples/echo2.qdl
```

Now, create an anaphor. The **`to_load`** takes the script from the first argument and packages all the remaining ones into the **`args`** element.

```
   x := a#to_load(echo, ['hello','world'])
   x
{"qdl":{"args":["hello","world"],"load":"/home/ncsa/apps/qdl/examples/echo2.qdl"}}
```

Now execute it

```
   a#exec(x)
[hello,world]
```

Each element of the list was sent as an argument to the script, which just returned it. Contrast this with sending just a JSON object (line breaks added for readability).

```
   y := a#to_load(echo, {'first':'hello','second':'world'})
   y
{"qdl":
  {"args":{"first":"hello","second":"world"},
  "load":"/home/ncsa/apps/qdl/examples/echo2.qdl"}
}
   a#exec(y)
[
  {
  first:hello,
  second:world
 }
 ]
```

Here, the entire JSON object was turned into a single argument and passed.

Example for to_code

```
   a#to_code(['AAA:=2;', 'say(AAA+3);'])
{"qdl":{"code":["AAA:=2;","say(AAA+3);"]}}
```

which turns the list of QDL statements into a code anaphor. Since **`exec`** and **`to_code`** (or **`to_load`**) are inverses, you can issue

```
   a#exec(a#to_code(['AAA:=2;', 'say(AAA+3);']))
5
```

Do remember that this snippet is run in is own scope, so which it inherits the current scope, it would not leave AAA defined in your workspace.