Conversions in QDL

Synopsis

QDL's convert module allows you to import or export to various standard serialization formats. The three supported are QDL, YAML, XML and HOCON (a simplified version of JSON).

Loading the module

You can import the module (listed in the default library as **`info().lib.xml`**) with

```
   module_import(module_load(info().lib.convert, 'java'));
convert
```

Imports generally.

Each of the import functions is of the form X_in where X is the format. They all operate the same:

`X_in(string)` - outputs in format X

`X_in({'file':'/path/to/file'})` - reads the file. Other options may be supported.

Exports generally.

Stems do not really match up exactly, (except for QDL format). For one thing, sets are converted to lists. The basic format for all of these is

`X_out(arg.{, file})` - convert `arg.` to format X. If there is a file path specified, use that and return true, otherwise, return a string representation of arg.

QDL

Super easy – this is just a wrapper around QDL's input form. It lets you chain together various other format seamlessly with a consistent syntax/behavior.

YAML

There are two main functions for YAML, `yaml_in` and `yaml_out`. No other import options except file reads are supported

HOCON

The two functions associated with this format are hocon_in and hocon_out.

XML

XML, on a good day, is very messy to work with. The approach here is to try and preserve the structure on import and allow for simplifying it to something much more manageable (see **`snarf`** below). The main import command is **`xml_in`** and it has various invocations possible.

`xml_in(`*`string`* `{,level})` which takes a string representation of the XML and imports at the given level (level 0 means to snarf it up, level 5 means to be completely faithful). The default is to be faithful.

```
   import('<a p="x"><b q="y">Z</b></a>')
{
 >declaration: {
  @version:1.0,
  @encoding:UTF-8,
  @standalone:false
 },
 a: [
    {@p:x,
      b:[[Z]~{@q:y}]
    }
  ]
}
```

This shows the imported structure (with declaration) of the document. The root element, *a*, has its property annotated with an initial **@** and an ordered list of its elements. In this case there is a single element. The text for the node is in the list (here Z) and other nodes/properties are in the map `{@q:y}`

Let us take a somewhat more complex example.

```
<a p="x">
  <b q="y">Z</b>
  <c>A</c>
  <b>Y</b>
</a>
```

and import it

```
   xml_in('<a p="x"><b q="y">Z</b><c>A</c><b>Y</b></a>') =: x.
{
 >declaration: {
  @version:1.0,
  @encoding:UTF-8,
  @standalone:false
 },
 a: [
    {@p : x,
      b : [[Z]~{@q:y},[Y]],
      c :[[A]]
    }
  ]
}
```

Here we see that the b nodes end up grouped (in order).

Structure of the stem

When an XML document is imported faithfully, then the structures transform directly to structure in the stem.

XML

```
<?xml version="1.0"?>



<!-- top level comment -->
<a q="T">

    <b p="S">W</b>
    <b p="U">X</b>
    <!-- Here is a comment -->
    <b p="V"><![CDATA[ Y ]]></b>
    Z
</a>
```

Stem

```
{
 >declaration : {
      @version : 1.0,
     @encoding : UTF-8,
   @standalone : false
 },
    >comment : top level comment,
 a: [[Z]~{
        @q : T,
         b : [[W]~{@p:S},
              [X]~{@p:U},
              [ Y ]~{@p:V,>cdata:true}
             ],
  >comment : Here is a comment
     }
  ]
}
```

Points to note

- The declaration, comments, entities and other fixed XML structures have keys starting with >.
- Properties of an element starting with an **@** sign.
- Elements have lists of nodes, even if there is only a single node.
- Each element turns in to a stem with a specific form of a list (of strings, the content) and a map of properties and elements.

We prefix certain stem keys with characters that are illegal names in XML, so there cannot be a clash.

Same XML document, snarfed.

```
{
 a: [Z]~ {
  @q : T,
   b : [[W]~{@p:S},
        [X]~{@p:U},
        [ Y ]~{@p:V}
       ]
    }
}
```

Mostly the aim of snarfing the XML is to have a compact stem with reasonable accessor patterns. A common complaint of using XML is having to navigate through all the structure to find the content. Snarfing the document cuts out a lot of that and then the usual stem operations (such as extraction and subsetting) allow rapid and pretty painless access, such as searching for elements with specific property values. See the Appendix below for a real life example of snarfing my Tomcat configuration file.

Snarf examples

```
  snarf(import('<a p="x"><b q="y">Z</b></a>')) =: r.
{a: {
```

```
  @p:x,
   b:[Z]~{@q:y}
 }
}
```

Which allows much more idiomatic stem access, e.g. `r.a.b.0` is the text for b.

```
  snarf(import('<a p="x"><b q="y">Z</b><c>A</c><b>Y</b></a>')) =: s.
{
 a: {
  @p:x,
   b:[[Z]~{@q:y},Y],
  c:A
 }
}
```

and, *e.g.* `s.a.c` gives the value of that node (vs. `x.s.a.c.0.0`)

Appendix, my Tomcat XML configuration file

```
{
 Server: {
   Listener:[
    {@className : org.apache.catalina.startup.VersionLoggerListener},
    {@className : org.apache.catalina.core.AprLifecycleListener,
     @SSLEngine : on},
    {@className : org.apache.catalina.core.JreMemoryLeakPreventionListener},
    {@className  :org.apache.catalina.mbeans.GlobalResourcesLifecycleListener},
    {@className : org.apache.catalina.core.ThreadLocalLeakPreventionListener}
    ],
  @shutdown : SHUTDOWN,
  GlobalNamingResources:  {
     Resource:   {
       @name:UserDatabase,
       @description:User database that can be updated and saved,
       @type:org.apache.catalina.UserDatabase,
       @auth:Container,
       @pathname:conf/tomcat-users.xml,
       @factory:org.apache.catalina.users.MemoryUserDatabaseFactory
     }
    },
  Service : {
   Connector :[{
     @redirectPort:9443,
     @connectionTimeout:20000,
     @port:44444,
     @protocol:HTTP/1.1
    },
    {
     @truststoreType:JKS,
     @URIEncoding:UTF-8,
     @sslProtocol:TLS,
     @maxThreads:150,
     @truststorePass:XXXXX,
     @clientAuth:false,
     @keystoreType:JKS,
     @scheme:https,
     @SSLEnabled:true,
     @keystorePass:XXXXX,
     @port:9443,
     @keystoreFile:${user.home}/certs/localhost-2020.jks,
     @truststoreFile:${user.home}/dev/csd/config/ncsa-cacerts,
     @secure:true,
     @protocol:HTTP/1.1
    }
    ],
```

```
   @name:Catalina,
   Engine:   {
    @defaultHost:localhost,
    @name:Catalina,
      Host : {
        @appBase:webapps,
        @name:localhost,
        @autoDeploy:true,
       Valve:     {
           @prefix:localhost_access_log,
           @directory:logs,
           @pattern:%h %l %u %t "%r" %s %b,
           @className:org.apache.catalina.valves.AccessLogValve,
           @suffix:.txt
        },
      @unpackWARs:true
     },
    Realm: {@className:org.apache.catalina.realm.LockOutRealm,
      Realm :{@className:org.apache.catalina.realm.UserDatabaseRealm,
              @resourceName:UserDatabase}
    }
   }
  },
  @port:8005
 }
}
```

So a typical access might be (if the value is in the variable `tomcat.`):

```
  tomcat.'Server'.'Service'.'Engine'.'@defaulthost'
localhost
```