The QDL Workspace

Version 1.4

Introduction

The QDL interpreter may be run from the command line but there is also another supplied tool called the ***workspace***. This runs a QDL interpreter and lets you run commands interactively as well as stores state (so you can create a bunch of variables and save them for future use). Sessions may be saved and loaded.

Workspaces are one organizational unit. A typical use scenario is having a workspace for a specific task. The workspace stores any functions, variables, modules etc. needed for the task and essentially allows you to use QDL as a custom “calculatdor” for that task. Workspaces are portable so you can make one and send it along to others to use.

Getting and starting the workspace

The current release of QDL is on git hub at https://github.com/ncsa/security-lib/releases and will be in qdl.jar. Download that and you should be set.

To start QDL, invoke it using java

java -jar qdl.jar

And you should get the splash screen:

```
****************************************
Welcome to the QDL Workspace
Version 1.0
Type )help for help.
****************************************
```

The prompt is simply an indent of 4 spaces. So try it out by typing the famous program

```
    say('Hello world');
Hello world
```

Congrats, you have just run your first complete QDL program. Since this is an interpreter, you issue commands in the QDL language. There is a whole other reference manual/tutorial for that. This blurb is about how to interact with the workspace.

Since it says to type)help, do it and you will see

```
   )help
This is the QDL (pronounced 'quiddle') workspace.
You may enter commands and execute them much like any other interpreter.
There are several commands available to help you manage this workspace.
```

```
Generally these start with a right parenthesis, e.g., ')off' (no quotes) exits this
program.
Here is a quick summary of what they are and do.
  )buffer -- commands relating to using buffers.
  )clear -- clear the state of the workspace. All variables, functions etc. will be
lost.
  )edit -- commands relating to running the current editor.
  )env -- commands relating to environment variables in this workspace.
  )     -- short hand to execute whatever is in the current buffer.
  ))    -- resume a suspended process
  )funcs -- list all of the imported and user defined functions this workspace
knows about.
  )help -- this message.
  )modules -- lists all the imported modules this workspace knows about.
  )off -- exit the workspace.
  )load file -- Load a file of QDL commands and execute it immediately in the
current workspace.
  )vars -- lists all of the variables this workspace knows about.
  )ws -- commands relating to this workspace.
```

If you type)off, you will exit the workspace.

```
    )off y
exiting...
```

Running a single script

You may run a single script with the *-run + script_path* argument if invoking via java:

```
java -jar qdl.jar -cfg config_file -name config_name -run script_path
```

or from the command line with the shell script as

```
qdl script_path
```

in which case your configuration will be loaded completely (no banner or splash screen) the script executed and the interpreter will exit. Note that environment variables and substitutions will be available. This lets you embed QDL in shell scripts, for instance with your required runtime. Note that this will load the system completely, so any boot scripts of virtual file system mounts will happen before your script is called.

Commands generally

Since the workspace is charged with managing the QDL environment for us, its commands are not QDL commands. All of the start with a right parenthesis, e.g.)help.

The basic structure of a command is

command action arguments

which we may write to save space as as **`command → action → arguments`**

So this is the command to save the current workspace to the file my_space.ws

```
    )ws save my_space.ws
workspace saved at Thu Jan 30 16:32:28 CST 2020
```

-)ws is the command
- *save* is the action
- *my_space.ws* is the only argument, in this case.

So on one line of text here this would written as **`)ws → save → my_space.ws`** Arguments may be simple, such as a single file name here, or there may be several of them. This varies per command, hence this reference.

Some commands have a short form because they are used so often. In this case **`)ws save`** may be replaced by **`)save`**.

In the command reference that follows, the heading of the section is the command and the subsections are the actions. An argument of – means that this may be omitted and is the default.

Geeky note If this seems a little unfamiliar you have, in point of fact been doing this sort of thing your whole computing life, at least if you have ever used a GUI. Since this is to be a text mode only application (written in Java, where there is no concept of cursor addressing) there are no things like menus. Effectively think of **`)command`** as being a menu command (like File or Edit) that has a drop down menu. You are manually navigating a menu, rather than clicking on a menu sequence like

ws → save → file dialog → select file_name

vs.

```
)ws save file_name
```

Templates, variables and script preprocessing

There are *two* ways to set values on the command line. The first allows you to set values directly in the interpreter environment and access them via the ***templating syntax*** of `${…}`. This is the analog to setting environment variables in a script.

Note: This is a feature of the workspace and is *not* part of QDL! It is to help harried coders reduce their typing workload and keep useful snippets of text at hand. If you develop code using templates, they should be removed if you are, e.g. going to save your code in a file and run it as a script later.

An example of how to use this: Let's set two environment variables:

```
)env set vo voPersonExternalID
)env set a @accounts.google.com
```

The template are done ***before*** any parsing or other processing. In computer languages a lot of use can be made of pre-processing or doing things to the code before actual evaluation, for instance, look up the Obfuscated C competition which is a humorous jab at abuses of the C pre-processor. This is a very useful facility to have and can make your work vastly easier at times, but it should be an aid, not a lifestyle choice.

So you could use this to, for instance, keep typos to a minimum but issuing statements like (assuming `claims.oidc` has the value 349765)

```
    ${vo} := claims.oidc + '${a}';
    say(${vo});
349765@accounts.google.com
    say(voPersonExternalID);
349765@accounts.google.com
```

What is the advantage of this? You can embed things like code snippets (especially the one you keep mistyping) or use it to prevent spelling issues or to future proof. In this case, if there were several references to `voPersonExternalID,` then the template allows you to set this one place in your script. Note that this lets you template for things like control structures, names of functions or really anything – it is wholly outside the language. If the name of ever changes (to say `voPersonExternalID_v2` then you would need to only change where it is set in the environment and the changes would take place automatically. While fairly trivial to use, it can make a lot of coding/testing considerably easier.

Accessing QDL variables in) commands

Another way to get information is to reference QDL variables. This is done with the prefix **>.** Remember that this is done after templates (if any) are processed. What will happen is that the variable is looked up and if found, its string representation is substituted for the argument. If not found, the argument is passed back unchanged.

```
  x := '/tmp/foo.qdl'
  )b create >x
0| |/tmp/foo.qdl
```

In this case, a variable is set and then used to create a buffer. This is intended to be very simple-minded, but since templates are done first, you could do something like

```
  )env set s src_file
  )env set t target_file
  )file copy >${s} >${t}
// resolves to
  )file copy >src_file >target_file
```

Assuming you had environment variables for s (source) and t (target) set appropriately.

Note especially that > access occurs in *all* commands, so something like this

```
)env set q >messy_variable
```

is fine. Why prefix variables with a > rather than just using variable names directly? Because you don't want to deal with quotations for arguments to system commands. It just makes it easier that everything is a string unless you tell it to resolve a reference.

Repeating the last command and history

If you enter a single **)r** then the very last command you entered is repeated.

```
   date_iso()
2020-03-16T22:01:29.258Z
  )r
2020-03-16T22:07:31.626Z
```

raise_In this case, the current date and time are created and then repeated.

This is a special case of the history command, **)h** which operates in two modes

)h - print entire history

)h *n* - execute command *n* in the history.

```
   2+3
5
   4+5
9
   )h
0: 4+5
1: 2+3
   )h 1
5
```

Note that **)r =)h 0**.

Listing things

For the following commands

-)funcs
-)vars
-)modules

Each allows for various formatting switches when listing. These are

- -r *regex* = apply the regex to each element of the output for matching
- -cols = list in columnar formatting
- -w *int* set the max display width. Note that -cols overrides this
- -compact = show namespaces as lists, so rather than a#q and b#q you'd see [a,b]#q.
- -fq (variables and functions) show fully qualified names.

Listing functions

System (i.e. built in) functions may be listed by passing in the system argument:

```
  )funcs system
abs([1])         exclude_keys([2])   log([1])                    set_default([2])
```

```
acos([1])       execute([1])         log_entry([0, 1, 2])      shuffle([2])
acosh([1])      exp([0, 1])          mask([2])                 sin([1])
```

(partial listing!)

Each entry is the name of the function and the number of arguments it can take, so exp (the exponential function) takes either zero or 1 argument. Note that these are formatted strings so if you want to search for functions using the regex flag, -r, it is easy.

To get help on any of these, use the help command:

```
  )help exp

 exp() - the value of  e, the base of the natural logarithm
exp(x) - the value of e^x

x can be any number.
See also: ln(), log()
```

The workspace (and QDL) tends to be extremely helpful.

Show the system functions that start with *to_* and list as a single column

```
  )funcs system -r to_.* -cols
to_boolean([1])
to_json([1, 2, 3])
to_lower([1])
to_number([1])
to_set([1])
to_string([1, 2])
to_upper([1])
to_uri([1])
8 total functions
```

Remember that for regexes the idiom **.*** means to match any character, so **to_.*** means just match anything that starts with **to_**. Also, you do not need a start of line character because each entry is not a line.

Just list the user defined functions:

```
  )funcs
a#f([1])     arf#f([1])   b#f([1])     c#f([1])     d#f([1])     woof#f([1])
```

Now list them in compact notation

```
  )funcs -compact
[a, d]#f([1])             [b, c, arf, woof]#f([1])
```

And now, list all of the system (built-in) functions with their namespaces (-fq flag is specific here) that are in the ***io*** namespace and restrict the output to 60 characters in width:

```
  )funcs system -fq -r io#.* -w 60
```

```
io#dir([1])             io#scan([0, 1])
io#file_read([1, 2])    io#vfs_mount([1])
io#file_write([2, 3])   io#vfs_unmount([1])
6 total functions
```

External Libraries

An *external library* is a collection of QDL scripts, all ending in .qdl. There is also a library path and flag in the configuration (both settable inside the workspace with the **)ws get|set** command). Note that the **lib_path** is not the same as the **script_path**. The reason is that libraries are integrated into QDL seamlessly vs. the library path. Let us say that we were using the example **my_sqrt.qdl** script in the examples directory. If **qdl_ex** is a variable that points to this directory

```
   )ws set lib_path >qdl_ex
   )ws set external_library_support on
   my_sqrt(5)
2.236067977499789
    my_sqrt(5)^2
5
```

This is entirely equivalent to issuing

```
   script_run(qdl_ex + 'my_sqrt.qdl', 5)
```

Note that you can indeed just treat it like any other function (hence the quick check that you can square the output and get back 5). The use case is some library of great command line scripts that you can simply access as an extension. All you'd need is a pointer to that directory and your workspace is extended without any imports or other calls, so QDL can behave much like shell scripts.

The one great caveat is that scripts on disk are on disk which is vastly slower than in memory. Invoking scripts this way is convenient, but if you were writing an application that, say, required looping and hundreds of invocations, loading the script instead and accessing the function that way would be a much better way to do things.

Buffers

A *buffer* is just a bunch of characters. It may reside in memory, on some storage device or represent a link between, say, a virtual file and a local file. In an interpreted language, you may want to have several statements that can be executed together. You may buffer them, then have them execute at once.

Buffers are created and have an *alias* or short name and a unique (assigned) integer, called a *handle*. Most operations on buffers take either of these. Operations on buffers are

- editing in the currently active editor
- CRUD (create, read, update and delete)

- check syntax (QDL syntax that is)
- linking to another buffer.

Major uses of buffers

- Editing code in an external editor. It’s easy to use your favorite editor with QDL.
- Running blocks and snippets of code for debugging or just convenience
- Editing code that lives in a virtual file system. Most off the shelf editors have zero concept of QDL’s VFS, so something has to be done to allow working with them.
- Checking syntax of the buffer.

Basic types

In-memory.

These reside wholly inside of QDL. You create these in the **`)buffer → create`** using the **-m** option. They are part of the workspace and will be saved when the workspace is saved. You may also write them to disk if you like, which converts them to regular buffers. When you edit an in memory buffer, a temporary file is created (usually in $QDL_HOME/temp) and that is passed to your editor. When you save inside the editor, that file is updated. When you exit the editor, QDL gets a message and reads this temp file, replacing the current in memory buffer. That’s how it works at a nuts and bolts level, but for you, you just edit however you want and the workspace does all the work.

Regular

This is simply a buffer that points to a file on disk. Note that while this can refer to a virtual file system entry, external editors have no concept of this, so a *link* (see next) is created. The buffer number will persists across workspace saves, but the file itself is not under QDL’s control. (So if you create a buffer, save the workspace, exit QDL, delete the file then restart QDL, buffer operations will fail.) If you edit the buffer from QDL, QDL shells out to the editor and control is returned on editor exit

Caveat: If you configure an external editor, QDL pauses while it is run. This makes perfectly good sense for a text mode application (since many of them run as full screen so you can't do anything in QDL anyway), but the same contract holds if you start a GUI editor from QDL – QDL will hang until you exit your GUI editor. If you want to use a GUI editor, run it in parallel, do not use the **`)buffer → edit`** command, just run it, check the syntax etc.

Link

A link couples two files the *source* and the *target,* so that operations on the target, can be used to update the source. It is used for external operations, such as editing it in your favorite editor. See the **`)buffer → link`** example in the command reference below.

Links also allow you to simply run your editor in parallel to a QDL session (again, another way to interact with a GUI editor). You work on the file, run it etc. and can update the link.

Lifecycle of a buffer.

All buffer information (the file, type of link, alias and integer) will be saved with the workspace. In memory buffers only reside in the workspace and will be saved. Regular buffers are outside of the purview of the workspace, so while the number and alias will be saved, these may or may not relate to a file.

Reasons for using buffers

At first glance it seems that this is just an extra hoop to jump through to edit files. Why not just invoke **`)file → edit`**? The main reasons are

1. Buffers integrate with the runtime system. It is much easier to debug code using buffers using the **`)`** command. They are assigned integers to help keep typing to a minimum.
2. You can create many buffers for work and cut/paste/run them as needed. This way you do not end up with bunches of little debug files. There is no reason that a buffer needs to have an existence outside of the workspace, especially if you are using it to write something that lives in the workspace (like a set of functions). **`)b → create → test0`** is perfectly fine and works for all editing as long as you don't **`)b → write`** it.
3. The workspace is a TUI (text user interface). External editors that are run from QDL must all be text mode as well if they are to integrate with the workspace. So, if your terminal is not strictly a text mode application (there are several Swing-based command lines) or you really want to use your favorite GUI editor, create a buffer (or a link if you need to seamlessly put it in a virtual file system) and manage it that way. Basically you have to manage when updates happen, but this allows for integration with your favorite editor.

Editing files and objects in QDL

There are several ways to do this. Read the section on

1. Do everything external. Just use **`script_run`** or **`script_load`** as needed. This works fine, but gets very clunky fast if you need to have a lot of little snippets of code to do things.
2. Use buffers. You may either use the built-in line editor (primitive but always works) or you make specify, within certain limits, a text mode editor (such as vi/vim, nano, Andy's editor). In memory buffers are saved as part of the workspace.
3. To edit a file, issue `)file → edit →` *`path`*
4. To edit a variable `)vars → edit →` *`var_name`*
5. To edit a function, `)funcs → edit →` *`func_name arg_count`*

6. To edit a buffer **)b → edit *alias|handle*** *or* **)edit → *alias | handle***

See the entries for each of the specific types of editing. Some of them have extra parameters (such as editing variables as text blobs).

Command reference

)buffer

Commands relating to buffer. At creation, you assign an alias and an integer aka an *index* or *handle* is assigned. In the sequel, # refers to this handle. The buffer command is fully VFS aware, so you can simply refer to VFS entries. Note that if the VFS is read-only, you cannot update it, so writes will fail. Consider using a link to such a file instead.

check # | alias

Check the QDL syntax of the file and report any errors. Nothing is run, it is simply parsed.

create alias {*path* | -m}

Create a new buffer. You must supply the alias. If you supply a (fully qualified) path, then that will be used. If you supply the flag -m then the buffer resides in memory only.

Examples

1. Most basic creation of a buffer

```
  )b create foo /home/jeff/dev/my_file.qdl
0| |foo: /home/jeff/dev/my_file.qdl
```

2. Create an in memory only buffer

```
  )b create init -m
1|m|init: init
```

This creates a buffer named init. Note that the handle here, 1 was created. The response is read

handle | save status | alias

save status is one of

- m – in memory only
- * – file that has pending changes in the ws. If you write the buffer, this goes away
- ? – link whose status (of the target) cannot be determined because QDL does not manage it.

3. Create a buffer in the default path using the alias as the name. Set the path first so that the file can be created

```
  )b path /tmp
new default buffer save path is "/tmp", was null
  )b create foo3
2| |foo3
```

Note that this creates a new file and just uses the relative path.

4. List the buffers we just created

```
  )b
0| |foo: /home/jeff/dev/my_file.qdl
1|m|init: init
2| |foo3: /tmp/foo3
there are 3 active buffers.
```

Note that in the listing, any path is also shown.

delete, rm *handle | alias*

Deletes the given buffer. Note that the handle of a buffer will not change once it is created, so if you want to recreate a deleted buffer, it will have a different handle. This has zero effect on physical files, but in memory buffers will be deleted.

edit *handle | alias*

Use the line editor (see below) to edit the buffer. This will also edit the link if you give one, although there is no need to do that. Edits are kept in the workspace until written to the file (if any) and are persisted through workspace saves. Note that

```
    )edit handle|alias
```

is an alias for this operation. See **)edit** below for more details.

How does editing work? The buffer is retained in the workspace. When you call an editor, a temporary file is created and the editor uses that (so your editor may give you a strange name for the file – ignore it.) The built-in line editor, however, does not use temporary files but can manage the buffer directly.

When you exit the editor, the temporary file contents are read and used to update the workspace. Note that the original file has not been touched. If you want to update that, **`)b → write handle`** will update the underlying file. Note also that the buffer is part of the workspace and persists through saves, which can provide a poor man's backup of the file.

link *alias source target* [-copy]

Creates a link from the *source* to the *target*. The edit and run commands will operate on the *target*. Issuing a save command will copy the contents of the *target* to the *source* overwriting it. Effectively this tells the system that rather than having an in-memory buffer, to use the *target* for that purpose.

If the **`-copy`** switch is used, then the source is copied to the link, over writing it. If you need to provision this otherwise, you might want to use the

```
    )file copy  source target
```

command. Please refer to that below. Links can be useful for many things, including running a GUI editor for a file. You may create a link then edit the link with the GUI in another window. This is practically how you would edit a file in a VFS with a program that knows nothing of QDL. The program only “sees” the target file.

Example

```
    )b ligo link vfs#/mysql/scripts/ligo/init.qdl  /tmp/init.qdl
6|?|ligo: vfs#/mysql/scripts/ligo/init.qdl --> /tmp/init.qdl
```

This creates a link to a local file. You edit **`/tmp/init.qdl`** with your favorite editor. QDL is still running just fine. To check the file,

```
   )b check 6
ok
```

to run the file (save it first, though no need to exit the editor), issue

```
   )b run 6
```

to save the target in to the source file (which lives in a VFS), issue

```
    )b save 6
```

Now **`vfs#/mysql/scripts/ligo/init.qdl`** has been replaced with **`/tmp/init.qdl.`**

list, ls

List all buffers, along with information about them.

Example

This is just taken from my current system

```
  )b list
1|m|bar:
2|  |json: /tmp/dump.json
4|m|scratch:
5|*|hw: /tmp/hw.qdl
6|?|lig: vfs#/mysql/scripts/ligo/init.qdl --> /tmp/init.qdl
there are 5 active buffers.
```

In this case, there are 5 buffers (a gap means a buffer was deleted). The second column contains an asterisk (“*”) if that buffer has not been saved. Buffers 1 and 4 are for in-memory use. Buffers 2 and 6 are links (these never have an asterisk in the second column because that cannot be reliably determined.) And finally buffer number 5 is just a file on the local system.

path [new_path]

Buffers may be created as simple (relative) names, *e.g.*, **`temp`**. If it is not in-memory, then saving it will save it to this directory. No arguments queries the current buffer default save path, supplying a new path will reset it. Note that the default is QDL's temp directory.

reload index | name

Reload the buffer from storage. No effect if it is memory only.

reset

This will *completely* delete all the buffers and new buffers will start at 0 again. Only use this if you really need to. Anything not written will be lost.

run *handle* | *alias {& | !}*

Runs, *i.e.* executes the buffer in the current workspace as if you had typed it in. A synonym is just a **)**.

An argument of & will clone the current workspace and run the buffer in that. This is quite useful if you are trying to debug something and do not want your current state to change.

An argument of ! will create a completely clean workspace and run the buffer in that.

Note that this will allow you to just specify a file and not a buffer if you choose.

Example

Run buffer number 1 (with name *bar*). All of these are the same

```
    )b run 1
    )b run foo.qdl
    ) 1
    ) foo.qdl
    )buffer run foo.qdl
    )buffer run 1
```

Why so many aliases? Because. That way you have flexibility in how you name things and refer to them.

show *handle* | *alias* [-src]

Shows the actual content of the buffer. The **`-src`** switch will force reading the source, rather than the buffer (if not saved) or the link (the default is the link).

```
    )show 3 -src
say('Hello world');
```

Shows the content of buffer 3, which in this case is a famous program.

write, save *handle | alias {path}*

Saves the buffer to *src*. If there is a link, that is written instead. Note that if you want to write an in memory buffer, you can specify the path. The buffer then is converted to a regular file buffer from that point forward.

)clear

See **)ws clear**. A typical invocation looks like

```
    )clear
workspace cleared
```

This is identical to issuing

```
)ws clear
```

) *index|name*

This is the “run the buffer” command. It is very useful shorthand. One nice application is to use this on whatever you are editing externally. This lets you (in conjunction with the % = repeat last command) quickly test a script. Here is a sample session

```
    )b link vfs#/mysql/scripts/ligo.qdl /tmp/ligo.qdl
7| |vfs#/mysql/scripts/ligo.qdl --> /tmp/ligo.qdl
```

Go do a bunch of stuff. To run it:

```
    ) 7
// output...
```

Go do more editing. When ready to test, the very next thing you can type to run it is the “repeat last command”

```
    %
// ouput…
```

)edit

Manage editors or (as a short hand) edit a buffer by handle.

(To edit a buffer by alias, use **)b edit → *alias|handle***

A note about the line editor

The editor supplied with QDL is a very basic, no bones about it *line editor* meaning that you issue commands *about* the lines. You will not see what you are editing unless you tell it to show you (the **p** command). If you invoke the editor, you may see help by issuing the **?** At the **edit**> prompt. Here is an example of starting the editor and getting the help to print out.

```
  )edit
CLI editor.
edit >?
This is the line editor. It operates per line. Each command is of the form
command [start,stop,target] arg0 arg1 arg2...
where
command is a command for the editor (list below). There are long and short forms
[start,stop,target] are line numbers hence integers.
    start = the starting index, always 0 or greater
    stop = the ending index. All operations are inclusive of this line number
    target = where to apply the [start,stop] interval. E.g. the insertion point for
copying text.
arg0, arg1,... = list of strings, possibly in quotes if needed, that are arguments
to the command.

To see help on a specific command, type ? after the command, e.g. to get help on
the insert command type
i ?
List of commands
a (append) append text either before a given line or to the end of the buffer.
b (view) view the contents of the clipboard. Contents not editable.
c (copy) copies a range of lines to the clipboard.
d (delete) deletes a range of lines
e (edit) edit a range of lines. Note this will effectively replace those lines.
f (find) search lines that match a given regular expression, printing each line
found.
i (insert) insert lines starting at a give index or append lines to the end of the
buffer.
l (clear) clear the buffer (but not the clipboard)
m (move) move a block of text using the clipboard
p (print) print the buffer or a subset of it
q (quit) quit the editor
r (read) read a file into the buffer
s (replace) substitute over a range of lines using a regex
t (paste) paste the contents of the clipboard into the buffer
u (cut) cut a range of lines from the buffer and leave in the clipboard
v (verbose) turn on verbosity, i.e. print more about the functioning of the editor
w (write) write the buffer to a file
z (size) query the buffer for its current size
? print this help, or in context, print the help for a command.
edit >
```

Note that the end of this you are returned to the prompt. **p** prints the buffer, **i** lets you insert text until you enter a single period. **q** quits. It is actually quite nice – it has a clipboard, supports regexs for certain operations and the list goes on.

Why have such an editor in this great day and age? Because not all keyboards are created equal (*e.g.*, does your cellphone let you hit the escape key? You can't use the estimable old unix editor *vi* without that.) Also, it seems that invariably if I need to log in to a server (which never have GUIs, since they are servers) remotely during an emergency, the terminal type gets screwed up and it all defaults back to 7 bit ASCII – pretty much the exact character set QDL recognizes. (Quick reminder that variables and functions use a small set of characters, but strings are fully UTF-8.) This editor is *not* intended to be the go to. The intent is that you set an external file, use whatever you like and when you need to run your

code, just hit **)** in the workspace. The editor, however, is pretty darned useful if you need it and since I had it lying around, why not?

One last little quirk is that the editor is indeed a completely stand alone program invoked by the workspace, so you can save your work in it to a file. If you are using it to edit the local buffer, all you need to do is exit (enter **q** at the prompt) and don't choose save (unless you want another copy of the buffer in an external file), the workspace knows how to get the resulting changes back all on its own.

An aside

If you like the line editor and need to use it as a standalone program, it is included in the qdl.jar, so you can issue

```
java -cp ~/apps/qdl/lib/qdl.jar edu.uiuc.ncsa.security.util.cli.editing.LineEditor
```

since sometimes being able to run an editor for a quick fix (e.g. on a newly configured server that has no editor available) is useful.

handle

If you just supply a buffer handle (like 0 or 2 – just an integer), then the buffer editor will be invoked on that handle. You cannot use an alias here.

```
   )b create temp -m
0| |m|temp
   ) edit 0
```

And en editor should start at this point.

add *name* [-clear_screen] path

Add a new editor. This will add the editor at **path** and give it a local alias of *name*.

```
   )edit add nano /bin/nano
```

The flag - clear_screen is sometimes needed if an editor does not actually clear the screen before starting. Most modern editors do and the symptom is that the editor starts running and whatever was on the screen before that stays in place. If you get gibbersih, try setting this true.

Configuring editors can be complex. This facility just lets you do some quick on the fly setup (which is usually more than sufficient). If you need more, consider adding an editor configuration to you configuration file.

list

List the currently active editors.

rm *name*

Remove the named editor. You cannot remove the currently active editor.

use *name*

Use the named editor as the default now. If need be, the use external editor flag will be set to true and the default editor will be set to *name*.

)env

The workspace allows for environmental variables The environment

drop *variable*

remove the named variable from the environmental

get *variable*

display the current value of the variable.

list, –

show all currently defined variables.

load *filename*

load the given variables file, adding it to the current environment. Note that this may overwrite currently defined values. The format of this file is a standard java properties file. This file becomes the active one and save will go to it unless a different file is specified.

save [*filename*]

save the current environment to the file

set *variable value*

Set the *variable* to have the given *value*. Note that everything to the right of the variable is considered the value, so embedded spaces are possible. If you need to, you may surround the value in single quotes to ensure that all spaces are faithfully preserved.

Examples

Set a variable

```
  )env set www 'once upon   a   midnight'
  )env get www
once upon   a   midnight
```

Show all the defined variables. We could supply the value of *list* but don't since it is optional.

```
  )env
Current environment variables:
{
 qdl_root=/home/bob/apps/qdl
 arf=abc d goldfish
 a=armstrong(600);
 d=load_module('/tmp/boot.qdl');
 foo=12345
}
```

)file

Operations on the file system***s***. These are all VFS (virtual file system) aware and mostly use the QDL functions for these.

copy *source target*

Copy the file *source* to the file *target*, replacing it. **Note** this is VFS aware, so you may seamlessly transfer files between VFS's or between the local disk and a VFS.

```
    )copy /tmp/ligo.qdl mysql#/scripts/ligo.qdl
done.
```

This copies a local file into a virtual file system. Make sure the target directories exist (see mkdir) before issuing this.

Warning: Because of differences between underlying file systems, you should *always* give the full path in the command or “predictable but unexpected results” may occur.

delete, rm *filename*

Delete the file from either the VFS or the local system.

dir, ls *file|path*

List either the directory contents (if it is a directory) or nothing (if it is a simple file).

Example

```
  )file dir vfs#/mysql/
a
/a/
/init2/
/mysql/
4 entries
```

In this case, there are 3 directories (these end with a /) and a single file, named *a*. Note that this just passes the argument to the QDL *dir* function. It is simply supplied as a convenience.

edit filename

This will bring up the file in the currently active editor. If you do not supply the full path, QDL will assume it is under the QDL directory and attempt to resolve it there. Note that this does **not** work on VFS files, since external editors know nothing of them. For that you need buffers and links.

mkdir *path*

Create a path in the VFS. Note that the contract allows for multiple path creation, so if you had a store with the directory vfs#/scripts/ and issued

```
   )file mkdir vfs#/scripts/init/ncsa/lsst/
```

The entire set of paths `init, init/ncsa init/ncsa/lsst` , would be created.

rmdir *path*

Removes the given file. Note that rmdir will only operate on directories. To remove a file, you must issue rm/delete. This passes the argument to QDL for processing.

vfs

List all information about currently installed VFS's.

Example

```
  )file vfs
Installed virtual file systems
  type:mysql        access:rw  scheme: vfs   mount point:/mysql/    current dir:(none)
  type:zip          access:r   scheme: vfs   mount point:/zip/      current dir:(none)
  type:memory       access:rw  scheme: vfs   mount point:/ramdisk/  current dir:(none)
  type:pass_through access:rw  scheme: vfs   mount point:/pt/       current dir:(none)
```

In this case there are 4 such systems. The underlying type of each is given, the access type (r = read, w = write, note that that the zip file system is always read-only. This is because of limitations of that (this fronts a zip file and lets you pull information out of it). All of these happen to have the same scheme but different mount points. It is possible to specify a default directory in a VFS, though none of them have that. All)file operations are VFS aware.

If you want to use VFSs you should consult either the reference for the *vmount* and *vunmount* commands to do it in QDL or the configuration reference to do it at startup.

Caveat: One mistake in listing a vfs is to forget that, *e.g.* in the above `vfs#/mysql` could be a file *vs.* `vfs#/mysql/` which is properly the mount point. When in doubt, directories always end with a slash. So the former might have nothing in it, but the latter is probably what you want.

)funcs

edit func_name arg_count

This will load the definition of a function into the editor and let you edit it. When you exit the editor (save any changes there you want) the function will be updated. While it will start with the function definition, you can put anything into the file and it will simply get sucked back in to the workspace.

```
    f(x) -> x^2 +1
    )funcs edit f 1
edit>
```

The editor shows up (in this case, the default line editor). If you updated the function, that would be reflected as soon as you exit. Note that formatting does not get preserved at times from one session to the next. This is because QDL is re-assembling the function each time. If you need specific formatting, such as comments, you should probably have the definition in a separate file and just edit that.

list, – [*switches*]

list all of the current functions in your workspace See the note above for command line switches that apply to this action.

Note that if you want to list specific functions, you *must* use the list command plus a regex. So let us say you had

```
    define[f(x)]body[return(x+1);];
    define[f(x,y)]body[return(x+y);];
    define[foo(x)]body[return(x-2);];
```

To to list any functions named exactly *g* you would issue

```
  )funcs list -r g.*

```

(No such functions exist.) So now let us list the ones called *f*:

```
  )funcs list -r f.*
f(1), f(2)
```

To list every function that starts with *f* we use the regex *f.** the '.*' means to take any character (the period) and match any. So can be read as *f* + anything:

```
    )funcs list -r f.*
f(1), f(2), foo(1)
```

By the same token, to just get the functions that start with *fo* you would use *fo.** like so

```
    )funcs list fo.*
foo(1)
```

drop *function*

remove a function. Note that this only applies to functions that have been defined (but not imported) in the workspace.

help [*name arg_count*]

If no arguments, print out every function and its simple description.

```
  )funcs
a#f(1), armstrong(1), b#f(1), check_parser(1), f(1), f2(1), f3(1), fac(1)
```

Note that they are qualified as needed and have their argument counts included. To list a specific function, you would issue something like this:

```
  )funcs help armstrong 1
An Armstrong number is a 3 digit number that is equal to the sum of its cubed
digits. This computes them for 100 < n < 1000.
So for example 407 is an Armstrong number since 407 = 4^3 + 0^3 + 7^3
```

In this case, the request was for anything about armstrong functions the workspace knows so the complete description from the function definition is returned.

system [*-fq*] [switches]

This will list the built in system functions in tabular format. Note that none of these has argument counts listed, because most of these have variable length arguments. This is intended to let you look up a name. You may also show the fully qualified names with the *-fq* switch. There is also no help (at this point) for each of these available in the workspace – mostly because keeping it in sync with the reference manual would be a huge ongoing task. Consult the reference manual for all the details. If you need to have it display for a certain width, pass that in. The default is 120 characters.

```
  )funcs system -w 80
abs()              from_json()        log()              set_default()      vfs_unmount()
box()              from_uri()         mask()             shuffle()
break()            halt()             mod()              size()
check_after()      has_keys()         module_import()    substring()
common_keys()      has_value()        module_load()      to_hex()
constants()        hash()             numeric_digits()   to_json()
contains()         include_keys()     os_env()           to_list()
continue()         index_of()         print()            to_lower()
date_iso()         indices()          raise_error()      to_number()
date_ms()          info()             random()           to_string()
debug()            insert()           random_string()    to_upper()
decode_b64()       is_defined()       remove()           to_uri()
detokenize()       is_function()      rename_keys()      tokenize()
dir()              is_list()          replace()          trim()
encode_b64()       keys()             return()           unbox()
exclude_keys()     list_append()      say()              union()
file_read()        list_copy()        scan()             unique()
file_write()       list_insert_at()   script_args()      var_type()
for_keys()         list_keys()        script_load()      vdecode()
for_next()         list_starts_with() script_path()      vencode()
from_hex()         list_subset()      script_run()       vfs_mount()
```

And just to show what all the fully qualified names look like

```
 )funcs system -fq -w 80
io#dir()                stem#is_list()           sys#from_uri()          sys#vencode()
io#file_write()         stem#keys()              sys#halt()
io#mount()              stem#list_append()       sys#index_of()
io#print()              stem#list_copy()         sys#info()
io#read()               stem#list_insert_at()    sys#insert()
io#say()                stem#list_keys()         sys#is_defined()
io#scan()               stem#list_starts_with()  sys#is_function()
io#unmount()            stem#list_subset()       sys#log()
math#abs()              stem#mask()              sys#module_import()
math#date_iso()         stem#remove()            sys#module_load()
math#date_ms()          stem#rename_keys()       sys#os_env()
math#decode_b64()       stem#set_default()       sys#raise_error()
math#encode_b64()       stem#shuffle()           sys#replace()
math#from_hex()         stem#size()              sys#return()
math#hash()             stem#to_json()           sys#script_args()
math#mod()              stem#to_list()           sys#script_load()
math#numeric_digits()   stem#unbox()             sys#script_path()
math#random()           stem#union()             sys#script_run()
math#random_string()    stem#unique()            sys#substring()
math#to_hex()           sys#break()              sys#to_lower()
stem#box()              sys#check_after()        sys#to_number()
stem#common_keys()      sys#constants()          sys#to_string()
stem#exclude_keys()     sys#contains()           sys#to_upper()
stem#from_json()        sys#continue()           sys#to_uri()
stem#has_keys()         sys#debug()              sys#tokenize()
stem#has_value()        sys#detokenize()         sys#trim()
stem#include_keys()     sys#for_keys()           sys#var_type()
stem#indices()          sys#for_next()           sys#vdecode()
```

Normally you do not need to fully qualify system names, but if you really had to, for example, name a function *load_module* you could do so and just qualify which you want to use, yours or the system command.

)help

This has a few modes. No argument just prints out a generic help message, as we saw above.

```
)help *
```

Prints out the first line of help for all user defined functions, if any.

The online reference manual

A reference manual for many common topics is included. To see all topics issue

```
)help online
! (¬)     cos       intrinsic    script_args     ~| (↚)
!= (≠)    cosh      is_defined   script_load     ∈
% (∆)     date_iso  is_function  script_path     ∉
(oodles more)
```

This is alphabetized. Entries are either the name of the topic or, if it is an operator,

ASCII (Unicode)

E.g.

```
! (¬)
```

Which means that the symbol ! may be replaced with ¬. To access the help for a topic, issue

```
)help topic
```

For instance, here is help for the modulus function:

```
  )help mod
mod(a,b) - compute the modulus, i.e., the remainder after long division, of two
           integers. The arguments are either scalar or stems.
```

If you as for an operator, would get (this is the digraph (two-character glyph) for set intersection:

```
   )help /\

Ascii digraph for set intersection. The result is a set
whose elements are common to both arguments.

See also: \/ (set union), subset(@pick arg)

unicode: ∩ (\u2229), alt + i
use -ex to see examples for this topic.
```

Note that the last two lines tell you what the alternate character is plus it unicode value and the keystroke available with the ANSI terminal. The last line tells you if there are examples. In this case there are and typing the

```
)help topic -ex
```

will show for example (using the unicode symbol for intersection, just to illustrate the point).

```
  )help ∩ -ex
    {0,2,3,5} /\ {4,2,1,0}
{0,2}
```

These are supposed to be extremely simple example you can just type in (or paste) to see how it works. Some of the help can be quite extensive. Great pains are taken to keep the online reference as up to date as possible, so it is a good idea to use it as much as possible.

Listing all system functions

You can also just list the system functions using

```
  )funcs system
abs([1])      exclude_keys([2])   log([1])              set_default([2])
acos([1])     execute([1])        log_entry([0,1,2])    shuffle([2])
acosh([1])    exp([0,1])          mask([2])             sin([1])
(more)
```

Which gives each system function and the number of arguments it may take. For instance, the exp function may take 0 or 1 argument. You may also search for specific arguments. To find all the string functions that can take 3 arguments you would issue

```
  )funcs system -fq -r string.*3.*
string#contains([2,3])      string#head([2,3])      string#insert([3])
string#substring([1,2,3,4]) string#tokenize([2,3])  string#detokenize([2,3])
string#index_of([2,3])      string#replace([3,4])   string#tail([2,3])
9 total functions
```

-fq means to fully qualify the functions. You may then search the resulting list with regular expressions like any other list of strings. To beat a dead horse, you can also have this display to a 60 character wide display with the -w switch:

```
    )funcs system -fq -r string.*3.* -w 60
string#contains([2,3])       string#insert([3])          string#tokenize([2,3])
string#detokenize([2,3])     string#replace([3,4])
string#head([2,3])           string#substring([1,2,3,4])
string#index_of([2,3])       string#tail([2,3])
9 total functions
```

Help for user functions

This works very similar to the reference manual. You may query help based on the argument count:

```
)help user_function arg_count
```

This will print out detailed help for the given user function for the number of arguments. Arg counts are ignored for system functions, because in many cases you need to compare the functionality. If you really get stuck, there is help for help as per usual.

```
)help --help
```

)modules

list, –

list the modules that are known to the workspace. This does not mean they have been imported.

imports

list the imports (URN and current alias) that are active.

```
  )modules
a:a  b:b  qdl:/java
```

This lists that there are three modules. To see if these are associated with aliases, use the imports actions:

```
  )modules imports -cols
a:a->[a, d]
b:b->[b, c, arf, woof]
```

These are the modules that have been explicitly imported. In this case, there is one, *qdl:/java* which has not.

Note that a synonym for this is just

```
)imports
```

)off [*y*]

This causes the system to exit. If you supply the argument **y** then the system will exit without saving. Otherwise, you will be prompted, but if you agree to exit, it will be immediate and with no save.

)si

This relates to the *state indicator* functionality. The state indicator is a powerful debugging tool. It lets you clone the current workspace and run scripts or whatever in the clone and inspect the contents. Generally, you use the standard)run command and include a command line argument of &. The effect is to take the current state of the workspace and clone it then run the buffer in that workspace. Either the program runs to completion and it exits, or, if there are breakpoints set in the code with the halt() command, the script runs until one of those is encountered. Resuming the program (with the)) command) will run until the next halt.

There is a small script that can be used a tutorial for this. That is halt_test.qdl and it included in the standard distro. Here is a sample session exploring it. The buffer is created then first attempted to be run in the workspace. This fails because halt() cannot be invoked in the main workspace (or the workspace itself halts). Then how to properly start it a clone is done.

Create a buffer

```
   )b create /home/jeff/qdl/examples/halt_test.qdl
0| |/home/jeff/qdl/examples/halt_test.qdl
```

This sample has multiple halt statement in it. trying to invoke it in the main workspace fails:

```
  ) 0
starting halt test
sorry, you cannot halt the main workspace Consider starting a separate process.
```

List the current state indicator. This is all the currently running processes in QDL.

```
  )si list
pid | active | stmt  | time                          | size  |  message
0   |   *    |       | Mon Oct 26 16:15:20 CDT 2020  | 2965  | system
```

Meaning there is exactly one process (the process id or ***pid*** identifies these. The main system process is always pid = 0). Now we start the buffer with a ***&*** added. This will completely clone the current workspace, load the buffer in to that and run it until the first halt() is invoked. Compare this with using a **!** rather than the **&**: That would start a new instance of QDL with *nothing* from the current state. Useful for other operations. Both **!** and **&** cannot be given at the same time.

```
  ) 0 &
starting halt test
101
```

What this does is start evaluating the script (in this case it just prints a message) and once a halt() is encountered, the pid is returned. Since the script is now suspended, you can just view its state in the state indicator:

```
  )si list
pid | active | stmt| time                          | size  | message
0   |   *    |     | Mon Oct 26 16:15:20 CDT 2020  | 2965  | system
101 |  ---   | 1   | Mon Oct 26 16:15:49 CDT 2020  | 2965  | Stop #1.
```

What this shows is the new entry with pid 101. It lists what the active state is (0, marked with a *) and what statement number the halt() was encountered on, here stmt 1 in the file.

Note: This does not give back the line number, but the statement. Remember that a statement in QDL ends in a ;, so while it is good practice to put statements on a line, QDL removes the whitespace before processing the statements. Upshot is that it is always a good idea to have a halt() on its own line. Furthermore, these are statements inside the active block, so if you issue a halt inside a loop, the statement number is the index inside the body of the loop.

The time this was started is shown. The size of the state (includes all variables and what not is shown). Finally, when halt(*msg*) is invoked, the message is displayed in the last column. Let's set the default for the workspace to pid = 101:

```
  )si set 101
pid set to 101
```

Now we can simply use the *resume* shorthand (a double left parentheses) without argument to run to the next halt command:

```
  ))
```

Since there was no output from the script, but a variable was set, nothing displays. Now let's show the state indicator again:

```
  )si list
pid | active | line | time                          | size  | message
0   |   *    |      | Mon Oct 26 16:15:20 CDT 2020 | 2965  | system
101 |   *    | 3    | Mon Oct 26 16:16:06 CDT 2020 | 3001  | Stop #2. a. is set
```

The message in the halt command is that a variable, *a.* was set. Let's check the variables in the workspace:

```
  )vars
a.  java#eg.
  a.
[2,4,6]
```

If we want we can swap back to the original state of the workspace by setting the pid to 0:

```
)si set 0
```

Commands.

edit var_name

Edit a variable in the current editor. Save your work in the editor and when you exit it, the definition in the workspace will be automatically updated.

Note that it is generally impossible to keep the format for a variable, so QDL does not, but will re-assemble the most current variable. This is because, e.g., variables, well, vary, so there is generally no way to get the “source code” for a variable.

list

List all current processes.

reset

Clear all pending processes, restoring the main system process as the only one.

resume [pid]

resume the evaluation of a process by the process id (id). Note that if the current process is what you want, you do not need the pid.

Also a shorthand for this is

```
)) [pid]
```

rm pid

remove the given process from the system. If it is active, then the main system process will be restored.

set pid

Set the current process to the given pid. Note that when you do that, you replace the current state of the workspace and can inspect the system in its current state. You may also issue resume commands which will update the current system's state. This allows you to debug a program ve

)vars

drop

list – [*switches*]

List all of the variables (but not their content) in the current workspace. See the switches supported by this action above.

size

List the total number of symbols in use in the workspace. This does not tell you how much memory is in use. See the workspace memory command for that.

system [switches]

List the system constants. Note that these are not kept in the symbol table and can only be accessed in fully qualified form.

```
  )vars system
sys#constants.
  sys#constants.var_types.
{boolean=1, string=3, null=0, integer=2, decimal=5, stem=4, undefined=-1}
```

edit [-x] var_name

Edit the variable. Note that the **-x** flag indicates that the variable is to be treated as text, so if it does not exist, it will be created as one. Text mode means that everything you type in will be taken as a string – so no quotes are needed. You may also edit a list of strings (each entrywill be a line), but a generic stem cannot be edited this way.

)ws

get *var*

Gets *i.e.,* displays the value of a workspace variable. Workspace variables are values that tell the workspace how to operate, such as whether to echo results to the console and such. A listing of all variables is available by issuing

```
  )ws get
autosave_interval
autosave_messages_on
autosave_on
compress_xml
debug
description
echo
enable_library_support
external_editor
id
lib_path
pp
pretty_print
root_dir
save_dir
start_ts
use_external_editor
```

To get the current value, simply supply the name. *E.g.* to see if pretty_print is enabled

```
    )ws get pretty_print
on
```

Note that **)help variable** will print some information about each variable.

lib drop [path] [-r regex] [-f]

Drops either a single workspace or those that fulfill the regex. The optional flag **-f** (for force deletes) if present means there will be no prompting before the deletes. Otherwise, you will be asked before each and every file. The reason for this is that it is easy to make a mistake and workspaces can take a lot of effort to create, so accidentally deleting one must be avoided. This is also helpful if you want to narrow down which files to delete, but a regex to only select those is impossible. Omit the **-f** flag and decide on each file as it is presented to you.

lib [path] [*-v] | [-l] | [-show_failures] | [-only_failures] | [-w width] | [-r regex]*

Shows the either the information about a workspace or all files in a directory. Information about each workspace will be shown, such as format. There are a few display modes, such as a short form that occupies a single line, or printing out detailed information about the stored workspace.

Note: This has a shorthand and **)ws lib** is equivalent to just issuing **)lib**

Example.

Print out a summary of all the files in the current save directory

```
  )ws lib
ws-test.zml     * (no id)    2021-01-09T13:42:45.278Z  (no description)
ws.zml          * sdf9634iu  2021-01-11T17:58:23.948Z  test description, which i...
zz.zml          * (no id)    2021-01-11T18:39:29.981Z  (no description)
```

These are the name, if compressed (* = yes), any set if, the creation timestamp and the first part of the description. The same thing but in *long format* is

```
  )ws lib -l
   compressed : true
    create_ts : 2021-01-09T13:42:45.278Z
    file_name : ws-test.zml
    file_path : /home/ncsa/dev/qdl/var/ws
       format : xml
last_modified : 2021-01-09T13:43:03.000Z
       length : 1544
-----
   compressed : true
    create_ts : 2021-01-11T17:58:23.948Z
  description : test description, which is a very long an completely unimaginati...
    file_name : ws.zml
    file_path : /home/ncsa/dev/qdl/var/ws
       format : xml
           id : sdf9634iudf
last_modified : 2021-01-11T18:08:16.000Z
       length : 1629
-----
   compressed : true
    create_ts : 2021-01-11T18:39:29.981Z
    file_name : zz.zml
```

```
    file_path : /home/ncsa/dev/qdl/var/ws
       format : xml
last_modified : 2021-01-11T18:39:58.000Z
       length : 1544
```

Note that each attribute is restricted to a single line. To print out a file and the complete contents of each field, use the -v or verbose switches

```
    )lib ws.zml -v
   compressed : true
    create_ts : 2021-01-11T17:58:23.948Z
  description : test description, which is a very long an completely unimaginative,
                verbose, overly loquacious and probably more than a bit too
                pedantic for most purposes.
    file_name : ws.zml
    file_path : /home/ncsa/dev/qdl/var/ws
       format : xml
           id : sdf9634iudf
last_modified : 2021-01-11T18:08:16.000Z
       length : 1629
```

Example using a regex

A common pattern is using a file filter like *.ws to get all of the similar files. The workspace supports regexes for this. So to get all the files like ws*.* you would supply the following regex (here operating on the default save directory):

```
   )ws lib -r ws.*\..*
ws-test.ws     * (no id)    2021-01-09T13:42:45.278Z  (no description)
ws.ws          * 'sdf9634iu 2021-01-11T17:58:23.948Z  test description
```

remember that * translates to .* and a period (which matches every character in a regex, hence requires special handling to just get a period itself) is represented by the \. as per above.

An example with -show_failures

The -show_failures flag will print out a report on *every* file that the workspace attempts to process. Normally a call to the lib command produces a report on recognized workspaces. If want to see what it thinks of other files, use this.

```
   )ws lib /tmp -r ws.*\..* -show_failures -w 80
ws.ws            failed:Error processing XML:Error: cannot deserialize class "edu
ws.ws          * f00fy      2021-01-11T14:06:22.372Z  my description of this wo
ws0.xml          failed:Error processing XML:Error: cannot deserialize class "edu
ws1.xml          (no id)    2021-01-07T22:12:05.612Z  (no description)
ws2.xml          failed:Error processing XML:Error: cannot deserialize class "edu
ws2.ws           failed:Error processing XML:Error: cannot deserialize class "edu
ws3.xml        * foo        2021-01-10T21:26:08.454Z  this workspace is designe
ws4.xml          foo2       2021-01-10T21:26:08.454Z  this workspace is designe
```

(note that the -w flag for display width is used here so it fits in this blurb, limiting the out put to 80 characters). This filters all the files in the /tmp directory by `ws*.*` in this case, we see that several of these failed because of deserialization issues – most likely a missing or changed java class. We also see which are compressed.

An example with -only_failures

In this case, only files that fail to be loadable workspaces will be shown.

```
   )lib /tmp -r ws.*\..* -only_failures -w 80
ws.xml            failed:Error processing XML:Error: cannot deserialize class "edu
ws0.xml           failed:Error processing XML:Error: cannot deserialize class "edu
ws2.xml           failed:Error processing XML:Error: cannot deserialize class "edu
ws2.zml           failed:Error processing XML:Error: cannot deserialize class "edu
```

This is very useful for tracking down munged workspaces and dropping them.

load [*file]*

Load a saved workspace from the given file. Relative references are resolved against the qdl root directory (which you can see since it is saved in the environment). This has short form

```
)load file [-keep_wsf] [-qdl]
```

`-keep_wsf` = keep the current ws_file as the default for saves and loads rather than resetting it to any argument.

`-qdl` = process as QDL. If the suffix is .qdl then this is implicit.

Note that if there are no arguments, the current default workspace file can be set/gotten as ws_file. If there is a default it will be used. For example

```
  )load
/home/ncsa/dev/qdl/var/ws/ws.zml loaded 12678 bytes. Last saved on 2021-01-12T04:14:44.000Z
```

Note that if you supply a file, that becomes the default ws_file.

If you load a QDL file this way, it clears the workspace then essentially runs a script load on it. QDL format, again, is not intended to reconstruct the workspace since QDL cannot actually do a lot of the state management needed to do that (how would get store all the pending in memory edit buffers?) . A dump is very useful at extracting some code to send to others. One other option is, of course, to simply issue a `script_load` command against the dump which would import everything to the current workspace.

Running a default function on load.

If you create a function named **`__init()`** (starts with a double underscore and has no arguments), then when you load the workspace, that will be invoked immediately after loading. This allows you to do things like write a generic workspace which requires some configuration. So

```
__init() → script_load('etc/config.qdl')
```

So part of the distribution of the workspace is to tell people to populate their local (in this case) **`config.qdl`** file with whatever information is needed.

If you do not want this to happen, then set the WS variable **`run_init_on_load`** to **`off`** before saving the workspace.

memory

Reports the amount of used memory, remaining memory and number of processors.

```
  )ws memory
memory used = 479MB, free = 461MB, processors = 8
  a. := indices(100000);
  )ws memory
memory used = 479MB, free = 441MB, processors = 8
  size(a.);
100000
```

It should be noted that the amount of memory is not fixed: if you decide to write

```
a. := random(1000000)^5
```

You may get something like this

```
  )ws memory
memory used = 761MB, free = 408MB, processors = 8
```

Which simply means that the system allocated more memory. The standard limit is 2 GB on 32 bit systems and unlimited on 64 bit system. You can change the initial amount at startup by configuring the Java virtual machine at system startup.

save [*file] [-show] [-compress on|off] [-java] [-qdl]*

Save the current workspace to the given file or default ws_file if none is specified. Relative references are resolved against the save_dir or root_dir (qdl root directory) .

This has shorthand formatting

```
)save file [-show] [-compress on | off] [-java] [-keep_wsf] [-qdl]
```

The various options are

(no args) = save to the current ws_file.

-java = use java serialization. This is always compressed.

-show = use XML format and simply print the (uncompressed) contents to the screen. Nothing is saved

-compress = (XML only) compress the result.

-qdl = do a dump into QDL source code.

-keep_wsf = keep the current ws_file as the default for saves and loads rather than resetting it to any argument.

Generally you should use XML format because it is future proof: Changes to Java may render java serialization unreadable at some point (this is a famous issue with this). On the other hand, XML is slow to read and write and very, very large, on the order of 10 times the size of java serialization.

Note that there is the compress_xml flag in the configuration that can set this for the workspace.

Best practice is to use suffix of

- .xml = uncompressed XML
- .ws = compressed (gzip) XML
- .wsj = serialized java
- .qdl = QDL dump

Example

```
)save my_ws.ws -compress on
Saved 2573 bytes to /home/ncsa/dev/qdl/var/ws/my_ws.zml on
Tue Jan 12 07:15:54 CST 2021, uncompressed size = 11611
```

(line break added for readability). Note that the relative file is resolved against the current save_dir. This saves the works in XML format to the file, compressing it.

Redistributing your workspace

If you write a workspace and want to send it to others, you should opt for the default. The extension of **ws** simply means that it is gzipped xml.

Java can be a good choice since it can be much smaller than the default. However, because it relies on java serialization, it is entirely possible that changes to QDL or Java may break it. It is therefore not a long-term storage solution.

A QDL dump is also useful but it is not a full save of the workspace, since many things in the workspace (editor choice, buffers, state indicator) have no analog in QDL – they help you manage QDL. It is very useful if you want to grab some functions, variables, &c and send them to someone without all the state.

There is also a special reserved function named `__init().` (Note that it starts with a double underscore, hence is intrinsic to the workspace. See the note on intrinsic elements in the reference manual if needed.). This will be run when the workspace is loaded. It takes no arguments. This is normally not defined, so if you write such a function, it will be called. This is perfect place to, say, do an initial setup (first time the workspace is loaded, see the `ws_macro` function too.) then later do any other housekeeping before the user gets access to the workspace. You might want to, *e.g.*, read in an

environment variable (in `os_env()`) that tells you where an ini file resides to set up the workspace. See the documentation for ini files for more.

The only caveat that goes with this is if you have extensions written in java – they must be available in the classpath when the workspace is loaded or there will be errors (or perhaps a partial loading, since QDL tries to be forgiving of loading errors).

set

Set values for the workspace. There is a list of these in the **get** section. Note that there is help available in the workspace, e.g.

```
  )help compress_xml

(workspace variable)
compress_xml - toggle compression of XML on WS save. Options are (on | true) or (off | false).
               Note that on loading workspaces, this not used -- the workspace will figure out
               if it was saved with compression and decompress it, so you don't have to worry about
               the compression of stored workspaces.
```

autosave_interval

Set the interval between automatic saves of the workspace. This only has an effect if **autosave_on** is set to true. E.g. if autosave is set to 900 sec., then every 900 sec. (that's 15 minutes) the workspace will be saved.

autosave_messages_on

Toggles whether to print the normal save messages whenever an autosave happens. If **autosave_on** is *false* then this has no effect.

autosave_on

Toggle the automatic save feature of the workspace. What this does is save the workspace ever autosave_interval ms. if enabled.

compress_xml

Toggle compression (gzip) when saving the workspace to XML. This results in usually a vastly smaller file, on the order of 10% of raw XML.

```
)ws set compress_xml on
xml compression on
```

description

A human readable description of this workspace.

```
    d := 'a very striking and complete description of this workspace'
    )ws set description >d
description set
```

This will be, e.g., displayed on loading a workspace or when listing workspaces with the lib command. There is also an id.

debug

Toggle deep debug mode. Note that this is really only for low-level things, like you are writing your own Java extension and really need to see stack traces and such.

echo

Echo prints the result (if any) of each statement you enter to the console. This lets you use QDL like a big calculator.

```
  )ws get echo
off
  2 + 2
4
  )ws set echo off
echo off
// Without echo mode on, you would have to fully write out every expressions
// So this will fail:
  2 + 2
syntax error:

// now turn echo back on so it works
  )ws set echo on
echo on

  mod(3*4*5*6,11)
8
// same as
// print(mod(360, 11));

  a. := indices(6)+10
  a.
{0=10, 1=11, 2=12, 3=13, 4=14, 5=15}
```

ee, external_editor

Set the external editor. The default is line which is built in. Another good choice is nano and QDL comes with syntax highlighting files.

id

A user define identifier for this workspace. This allows you to, for instance, set something meaningful. This is very useful as an internal identifier if a serialized workspace is being sent someplace and file naming conventions require that it be called something counter intuitive.

```
    )ws set id continuous_functions
workspace id set to continuous_functions
```

pretty_print, pp

pretty print mode allows for stems to be printed horizontally rather than in one long line, making it much easier to read them. pp is short for pretty_print.

```
  )ws set pp off
pretty print off
  sys#constants()
{var_types.={boolean=1, string=3, null=0, integer=2, decimal=5, stem=4, undefined=-
1}}
)ws set pp on
pretty print on
  sys#constants()
{
 var_types.= {
  boolean=1,
  string=3,
  null=0,
  integer=2,
  decimal=5,
  stem=4,
  undefined=-1
 }
}
```

root_dir

The root directory for the workspace. This means that all relative file paths will be resolved against this for various operations (relating to the workspace, not QDL generally). Such editing files and such.

save_dir

The default directory for saving relative files, so

```
   )ws set save_dir /tmp/ws
   )save ws.zml
saved to / tmp/ws/ws.zml
```

start_ts

The time the workspace was first started. generally this is set and managed for you, but sometimes people like to set it to, *e.g.*, a release date.

ws_file

This is the file corresponding to the current workspace. All saves and loads will use this unless a file argument is given to them. Note that specifying a file at either load or save will cause this to be updated to that file, providing the operation is successful.

clear

This command will remove all state and return it to the exact condition of starting a clean workspace, except that environment variables are not affected. All variables, functions, imports and any other state will be lost. This is useful if, for instance, you need to start a different project or if, e.g.something went wrong and you really need to just get rid of everything. A typical invocation looks like

```
    )ws clear
workspace cleared
```