The QDL Workspace

Version 1.2.2

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.

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 line 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

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.

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.

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 access QDL variables directly. 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

If you enter a single % then the very last command you entered is repeated.

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

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

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.

A few examples:

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

```
  )funcs system -r to_.* -cols
to_hex()
to_json()
to_list()
to_lower()
to_upper()
```

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 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()      io#print()    io#say()      io#unmount()
io#mount()    io#read()     io#scan()     io#write()
```

Command reference

)buffer

A *buffer* is just a bunch of text. A buffer on disk is just a file, hence every file is a buffer. 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. There is also an editor supplied (see the **)edit** section below), though that is wholly optional.

There are two types of buffer, a regular buffer or a *link*. A link is used for external operations, such as editing it in your favorite editor. Remember that the workspace is aware of QDL virtual file systems, so a common pattern might be to create a link on the local system to a virtual file. You may then work seamlessly on the link, updating the virtual file as you see fit.

Lifecycle of a buffer.

You create a buffer or a link. You may then edit the contents. The contents are kept in the buffer until you save/write them to whatever store. If the buffer is a link, then the link is used as the source for save/write. Once they are created there is an index

reset

This will *completely* reset all the buffers and new buffers will start at 0 again. Only use this if you really need to.

create *path*

Create a new buffer.

```
)buffer create foo
0| |/tmp/hw.qdl
```

creates a buffer named foo. This is the zero-th buffer and most operations can access it using the index or the full name. Note that since buffers live in system memory (and the files associated with them are only updated when you say too), you may have several of them open at once. You should not save them since the system will attempt to find a local path and file (most likely this would get written to **`$QDL_HOME/foo`** if you wonder).

```
)b create vfs#/mysql/scripts/interociter.qdl
```

creates a link in a virtual file system (in this example is it a MySQL backed VFS).

delete, rm *index* | *name*

Deletes the given buffer. Note that the index of a buffer will not change once it is created, so if you want to recreate a deleted buffer, it will have a different index.

edit *index|name*

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. Note that

```
    )edit index|name
```

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

link *source link* [-copy]

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

Example

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

This creates a link to a local file. You may then update the local file with your favorite editor. To run the file, issue

```
   )b run 6
```

to save the link in to the source file, issue

```
    )b save 6
```

list, ls

List all buffers, along with information about them.

Example

This is just taken from my current system

```
  )b list
1|*|bar
2|  |/tmp/baz.qdl --> fnord
4|*|vfs#/ramdisk/foo
5|  |/tmp/hw.qdl
```

```
6| |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.

run *index* | *name*

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

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
    ) 1
    ) foo
    )buffer run foo
    )buffer run 1
```

Why so many aliases? Because.

show *index* | *name* [-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 in links).

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

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

write, save *index* | *name*

Saves the buffer to *src*. If there is a link, that is written instead.

)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 *index|name*

A note about the 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? 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.

)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 double 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.

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

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.

```
)help user_function | system_function
```

Will print out help about the function. If it is a system function (*e.g.* list_insert_at) it will print out a general help message. This is designed to be sufficient to get you using the function but please refer to the reference manual for more details.

Supplying a user_function will print the first line of help for each of the arg counts associated with that function.

```
)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.

```
)help --help
```

Will print out how to use the help command to get more information about functions.

)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.

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}
```

)ws

echo [*on* | *off*] [-pp on|off]

Turns *echo mode* on or off. If no argument, the current mode is displayed. If the switch **-pp** is used, then pretty print mode is turned on or off. This refers to printing all stems vertically.

Echo mode is the default on workspace start up. It will take any expression (so not control statements, like if, switch, etc.) and wrap it in a *say();* call. It will also add any implied but missing final ";".

```
  )ws echo
echo mode currently on
  2 + 2
4
  )ws echo off
echo mode 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 echo on
echo mode 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}
```

Echo mode lets you use QDL like a big calculator.

Another example. Turn on pretty print

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

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).

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*

Save the current workspace to the given file. Relative references are resolved against the qdl root directory (which you can see since it is saved in the environment).

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
```