A Quick QDL Tutorial

Table of Contents

Introduction

QDL is a general purpose, interpreted programming language that also may be used for scripting – or even interactively as a calculator. The big draw is that it is mostly a notation rather than structured like most other languages (e.g. Java or C). Essentially you specify the things that ought to happen to your data and it gets done. You can do a lot in a just a few commands and because of this, a lot of control structures (which it does have, rest assured) are usually not needed. As it were, certain common control structures are embedded in the data.

It also has a workspace model, meaning that you start the interpreter, type in commands interactively creating variables, functions etc. and can save your current state and resume where you left off. This lets you develop specialized sets of tools to do tasks that are complex and really don't fit in with other language paradigms. QDL is very useful at taking up the slack where a lot of other languages don't quite work, or would require a large investment (such as designing and implementing a set of objects) to do anything reasonable. It is largely a functional programming language.

The language description is quite short and you should be able to be productive in it within a few minutes. The full reference QDL reference is a great next step after this tutorial.

Running QDL

QDL runs in a java virtual machine. Be sure you have at least Java 8.

Installing QDL

Get the latest installer at https://github.com/ncsa/security-lib/releases. This is the file qdl_installer.jar. Run it as

```
   java -jar qdl_installer.jar
```

and follow the instructions. Assuming you installed in a directory called $QDL_HOME. Invoke it as

```
$QDL_HOME/bin/qdl
```

You should see a splash screen and banner like this

```
- - . -  / - . .  / . - . .
(  ___  )(  __  \ ( \
| (   ) || (  \  )| (
| |   | || |   | || |
| | /\| || |   ) || |
| (_\ \ || (__/  )| (____/\
(____\/_)(______/ (_______/
- - . -  / - . .  / . - . .
**************************************
Welcome to the QDL Workspace
Version 1.3.1
Type )help for help.
**************************************
```

Of course, you may customize startup to do any number of things and the reference for that is here: qdl configuration.

Navigating the workspace

The basic unit of work is the *workspace* or *session*. This is keeps track of everything you've entered and its job is to help manage using the language. The workspace has many built in operations and generally they all start with a right parenthesis. For instance, to display all of the built in functions

```
  )funcs system
abs()    date_ms()     has_keys()
```

(Lots more of these, omitted to save space.)

Of course, there is a complete reference for the workspace here.

Help!

To get help on a workspace command, either invoke

```
  )help
This is the QDL (pronounced 'quiddle') workspace.
You may enter commands and execute them much like any other interpreter.
… lots of help
```

And a list of available commands. Generally if you are not sure what to do, invoke a command with the flag --help (note the double hyphen): E.g.,

```
  )ws --help
Workspace commands
   load file  - load a saved workspace.
  save [file] - save the current workspace to the given file. If the current
workspace
                has been loaded or saved, you may omit the file.
        clear - removes all user defined variables and functions
          get -  get a workspace value.
          lib - list the entries in a library.
       memory - give the amount of memory available to the workspace.
         name - give the file name of the currently loaded workspace.
                If no workspace has been loaded, no name is returned.
          set -  set a workspace value.
```

There is a complete set of documentation available and you should read it. The aim of the tutorial is to bootstrap your understanding of QDL. If you need to exit QDL, the way to bail on everything (rather than saving your session) is to issue

```
   )off y
```

Which tells the workspace to turn itself off and the parameter *y* tells it you really don't want to save anything.

Expressions

Basics

The prompt is simple an indent of 3 or 4 spaces. Try your first expression

```
   2+2
4
```

This means to evaluate 2+2 and the answer is printed without indent.

Note that statements in QDL end in a semi-colon, **;**. In the workspace, the usual mode, however, is called *echo mode* and this just means the semi-colon is added and if there is a result (echoed to the screen), it is printed. If this is not working, tell the workspace to turn it on

```
   )ws set echo on
echo is now on
```

Parentheses are used to group things together. There are a few reserved keywords in QDL

```
true     if       define     assert     try      module
false    then     while      switch     catch
```

```
null     else     body       block      do
```

Numbers

There are basic operators, such as **`+ - * / % ^.`** There are many other built in functions, such as computing the modulus, standard trigonometric functions and such. Standard order of operations is in effect, so **`5+4*3^2`** evaluates to **`41`** as expected. Unlike many languages, QDL has no effective no limits on the size of numbers. For instances

```
   5^0.5
2.23606797749979
  2^3^4^5^7^8^9^10^11^12
1.16267946647039E6008077
```

The first computes the square root of 5 and displays it. The second does multiple exponentiations and note that engineering notation is supported too.

Simple values of a string, boolean or number are called *scalars*. There are also *aggregates* aka *stems*.

Strings

Strings are anything between single quotes. Since QDL is UTF-8 aware:

s := 'ნიანგის ცრემლები'

is a perfectly fine String in QDL. (Georgian for “tears of the crocodile” which looks really cool.) You may do various operations such as as get the size

```
   size(s)
16
```

Strings may be concatenated with the + and have elements removed with the - operators:

```
  'abcd' + 'efgh'
abcdefgh
  'abcabcdabce' - 'abc'
de
```

and you can compare strings with the equals operators

```
   'abcd' == 'abcd'
true
```

Not surprising, there are many other operations:

```
contains()    head()        replace()     to_upper()    trim()
detokenize()  index_of()    substring()   to_uri()      vdecode()
from_uri()    insert()      to_lower()    tokenize()    vencode()
```

You can access information with examples using **`)help`**, e.g.

```
  )help replace
```

```
replace(source, old, new) - replaces all occurrences of old with new in the
                            source. These may be various combinations of stems
                            and strings.
E.g
   replace('abcde', 'cd', '23');
ab23e
```

Booleans

The final scalar type is a *boolean* which has reserved values of either **`true`** or **`false`**. There are the operations you expect such as **`==`**, **`&&`** and **`||`** (resp. equality, logical and and logical or). Boolean expressions do *short-circuit* evaluation, so if the result is returned once it can be determined rather than evaluating everything first (which would be closer to algebraic use, but does not work well on computers).

```
false && (true || true)
```

for instance stops evaluation after the first `false` is evaluated, since the rest of the expression can never alter the value.

Variables

A *variable* holds a value. To assign a value, you must use **`:=`**

```
   x := 2+3
   x
5
```

When variables are assigned, their values are not shown. To see what is in a variable, type it and hit return. The names of variables are restricted to upper and lower case letters, numbers, $ and _ (the underscore). Note that while you can use UTF-8 in most places, QDL is restrictive in its allowed set of characters for variable names.

If a variable is not assigned, it can be tested. For instance, if `foo` is not defined

```
   is_defined(foo)
false
```

Scalars vs. stems

Scalars are simple values, such as a number, boolean or string (the three major type of scalar). This is in contrast to *stem* or aggregate variables. The basic idea is that there is the variable name + one or more periods followed by an index reference. For instance, to create a variable, *x*, and set three values

```
   x.0 := 2; x.1:= 3; x.2 := 5;
   x.
[2,3,5]
```

A list is just a stem whose keys are integers. As I said, QDL statements end in a semicolon – always, even if the workspace is adding it for you at times – and the first line above just shows using that to stack multiple assignments on a single line. More generally, you can use pretty much any keys you like:

```
  y.foo:='bar'; y.fnord:='baz';
  y.
{
 foo:bar,
 fnord:baz
}
```

This shows what the keys are (left-hand elements) and their values. It is also possible to enter a stem directly:

```
y. := {'foo':'bar', 'fnord':'baz'}
```

Stems may also refer to other stems. So adding another key of *x* that has a value leads to the following

```
  y.x.0 := 2
  y.
{
 foo:bar,
 fnord:baz,
 x: [2]
}
```

Stems are actually extremely powerful data structures (known technically as *associative arrays*) with a very convenient syntax for accessing elements. the operations available make them actually more akin to mini-databases.

A *list* is a special case of a stem whose indices are all integers and are enclosed in square brackets. A very, very useful and important idea in QDL is *extension* meaning that all basic operations are extended to every element of a stem. This means that control structure like loops to access elements are not needed. For instance to add 10 to all elements of a list

```
   [2,4,6,8] + 10
[12,14,16,18]
```

To raise every number in a list to its square then would be

```
   [1,2,3,4,5]^2
[1,4,9,16,25]
```

A much more complex example is computing the hyperbolic sine of several values

```
   sinh([1,2,3,4,5]/12)
[
 0.083429817445953,
 0.167439343987515,
 0.252612316808168,
 0.33954055725615,
```

```
 0.428828083093884
]
```

Note especially: Stems are accessed with a period (the index operator) and the rule is that

1. The left most term is the stem, everything to its right is some form of index.
2. If an index is undefined, the string value of the index is used.
3. If an index refers to a variable, that value is substituted
4. Multiple indices are substituted from the right.
5. If a list is used as the index (after every other way to resolve it), then it is interpreted as follows:

`a.p.q. … r == a.[p,q, … ,r]`

So for instance using the above value of **`x.`**

```
   j:= 0
   x.j
2
```

Here, **`x.`** is the stem, **`j`** is the index. Since **`j`** is 0, **`x.j`** resolves to **`x.0`** which is the value 2. If you are used to other programming languages, you are probably used to an expression like (e.g. in C)

```
x[0] = 2
```

The index operator is much more flexible and allows for a great simplification. You could have a stem like

```
orders.time.manner.place
```

That tracks an entire inventory system. You can populate it for various values of time, manner and place and then apply operations to see, *e.g.* the latest date, or all things from a specific place. See the reference manual for more details on stems and their uses. They are surprisingly addictive and you'll probably start wishing other languages supported them the way QDL does[1].

Note that expressions can be used for indexing, quick example is the function **`i(k)`** that simply returns **`k`**. Then this is a fine way to access the stem **`[;5]`**

```
   [;5].i(2)
2
```

(meaning, create a list from 0 to 5 (exclusive), grab the second element.) This makes stems enormously flexible and powerful.

A few more quick examples for stems

```
    a := 3;          // assigns 3   to  a
    b := 4;          //    "    4   to  b
```

1 They are called *associative arrays*, *dictionaries*, or *maps* in other languages, but none of these other languages really have such convenient ways of accessing them. Check out the QDL function **`query`** which lets you search a stem.

```
   c := 'last';     //    " 'last' to  c
 a.b := 2;          //    "    2   to  a.4
 a.c := 5;          //    "    5   to  a.'last'
x.a.b := 'cv3d';    //    " 'cv3d' to  x.3.4
```

Note that there are two variables, **a** (a scalar) and **a.** (a stem).

Example of counting words

In this example, we will prompt the user for a sentence and count the words in it.

```
count. := {*:0};
words. := tokenize(scan('enter sentence: '), ' ');
while[for_next(j, words.)][count.j := count.j+1;];
say(count.);
```

You can put this in a script and run it. How it works. By each line

1. Set the default for the stem count. to 0. This effectively sets every possible value to 0.
2. Prompts the user for a sentence from the command line, read it and tokenizes it by blank. The result is a list of words stored in `words.`
3. loops over elements in `words.` Note that the index in `count.` is just the word itself. adds one to the current word count
4. Prints the resulting stem.

E.g. of running this (in the script /tmp/wc.qdl)

```
   script_run('/tmp/wc.qdl');
enter sentence: mairzy doats and dozey doats and liddle lambie divey
{*:0, doats:2, lambie:1, mairzy:1, liddle:1, and:2, divey:1, dozey:1}
```

So the words “doats” and “and” are repeated twice each.

Subsetting

There are many languages over time that have used the sorts of operations QDL supports. One problem they have is conformability of the arguments. Extending every operation to stems is great, but what if one has 5 elements and the other has 4? A common fix is to have some agreement on supplying missing values (like zero for numbers or blanks for strings). The problem with this is that if you are missing data, the system is creating information. While often extremely convenient in some cases, for things like scientific computing that is very bad. QDL's solution is minimalist: ***subsetting*** This means that only the common indices are processed and you will get back a subset generally of your arguments. For instance, multiplying two lists together where one has 6 elements and the other has 3, returns the three element list where the operation can be reasonably applied.

```
  [1,2,3,4,5,6]*[3,2,1]
```

```
[3,4,3]
```

This also tells you up front that if you are missing results, you are missing data and it is much, much better in practice to know this as early in any processing as possible, especially if you are doing numeric processing, you do not want the system to introduce systematic errors as a matter of course. Sometimes it confuses people (“where did my answer go?”) but is a useful idea.

Functions

A *function* is a self-contained unit of code that does a specific thing. Generally they relate inputs to outputs. There are several of them that are built in. This includes most standard Math functions (logarithms, sines and such) as well as a wealth of string and other commands.

```
   sin(pi()/4)
0.707106781186547
```

computes the trigonometric sine of pi divided by 4. Note that in QDL pi() is a function. No argument means compute pi at the current precision, otherwise, compute pi raised to the power of the argument:

```
  pi(0.5)
1.77245385090552
```

computed the square root of pi.

Defining your own function.

The easiest way to define a function is using the so-called lambda notation

```
def -> statement
```

which will evaluate the statement and return the result. This works for single statement functions. You can create much more complex functions using the define[] statement.

Example

```
   f(x)-> x^2
   f(3)
9
```

QDL is what is termed a *functional language* which means it is very, very easy to define and use functions.

Help!

All functions have help available in the workspace. For instance, to get help for the function substring issue

```
  )help random
```

```
random() - return a random number
random(n) - return n (an integer) random numbers as a stem list.
```

This tells you the syntax and has a basic description.

```
)help *
```

prints out a quick summary of all user defined and loaded functions.

Writing your own function

There are a couple of ways to do this. For instance, the most basic program is of the form

`name(args)-> expression`

where the `args` is a list of arguments and the expression is a single expression which is returned by the function. The standard Hello World would be

```
   hw()->'Hello World!'
```

```
   hw()
Hello World!
```

There is a much more comprehensive way to define functions (which includes making them self-documenting) which is in the reference manual.

You may also pass anonymous functions as arguments for instance to the built-in **`for_each`** function. Another example is the plot function in the standard extensions. This takes a function and list of arguments and returns a set of x and y coordinates suitable for plotting with, e.g., gnuplot.

```
    plot((x)-> sinh(x)*cosh(x), [|-1;1;10|])
[
  [-1,-1.813430203923503],
  [-0.777777777777778,-1.131661442963],
  [-0.555555555555556,-0.677134697427398],
  [-0.333333333333334,-0.358579230505522],
  [-0.111111111111112,-0.11202786652122],
  [0.111111111111111,0.112027866521218],
  [0.333333333333332,0.35857923050552],
  [0.555555555555554,0.677134697427389],
  [0.777777777777776,1.131661442962996],
  [0.999999999999998,1.813430203923503]
 ]
```

(So the input values are is the 10 element list (called a closed slice) of decimals from -1 to 1 and the output values is the product of the hyperbolic sine and cosine.)

Because of this, functions are processed in the order they are found and you must define a function before you use it. Some other programming languages are “Platonic” in the sense that functions just exist. For instance, in Java functions are defined in a class and you make instances of the class. Java

tries to support lambda functions (as these are called) but the result is a very clunky mix of interfaces. In QDL (which is a functional programming language) they may be used and created any place and even passed around as arguments to functions. This makes them behave more or less like variables.