A Quick QDL Tutorial

Introduction

QDL is a general purpose, interpreted programming language that also may be used for scripting – or 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 ca

n 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.

A common use is to start it and just use it for all manner of algorithmic calculating. For instance, I had two databases with similar sets of columns to compare. Doing it by hand would have take quite some time, but it was relatively easy to do some basic simplification of the information then compare which were substrings. This would have taken writing several pages of custom code in another language. It is also useful for exploring data structures (e.g. you get a JSON object and have to make sense of it), hashes for quick pass word management and the list goes on. It is one of the most useful pieces of code I routinely run. This is in addition to using it for its original purpose which was server-side scripting.

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 11.

Installing QDL

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

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

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

```
$QDL_HOME/bin/qdl -tty swing
```

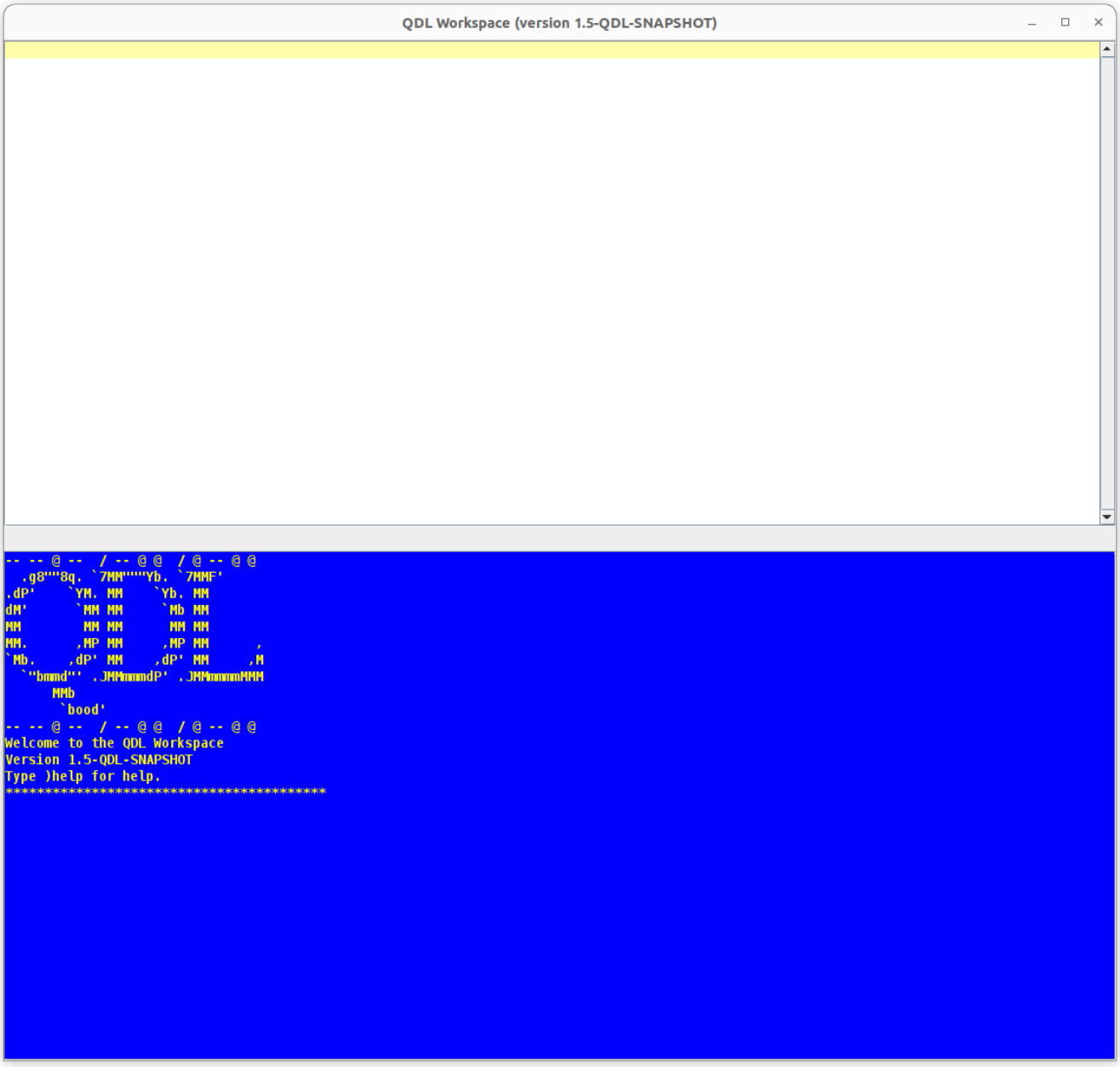
You should see a splash screen and banner like this

```
- - . -  / - . .  / . - . .
(  ___  )(  __  \ ( \
| (   ) || (  \  )| (
| |   | || |   | || |
| | /\| || |   ) || |
| (_\ \ || (__/  )| (____/\
(____\/_)(_______/ (_______/
- - . -  / - . .  / . - . .
**************************************
Welcome to the QDL Workspace
Version 1.5
Type )help for help.
**************************************
```

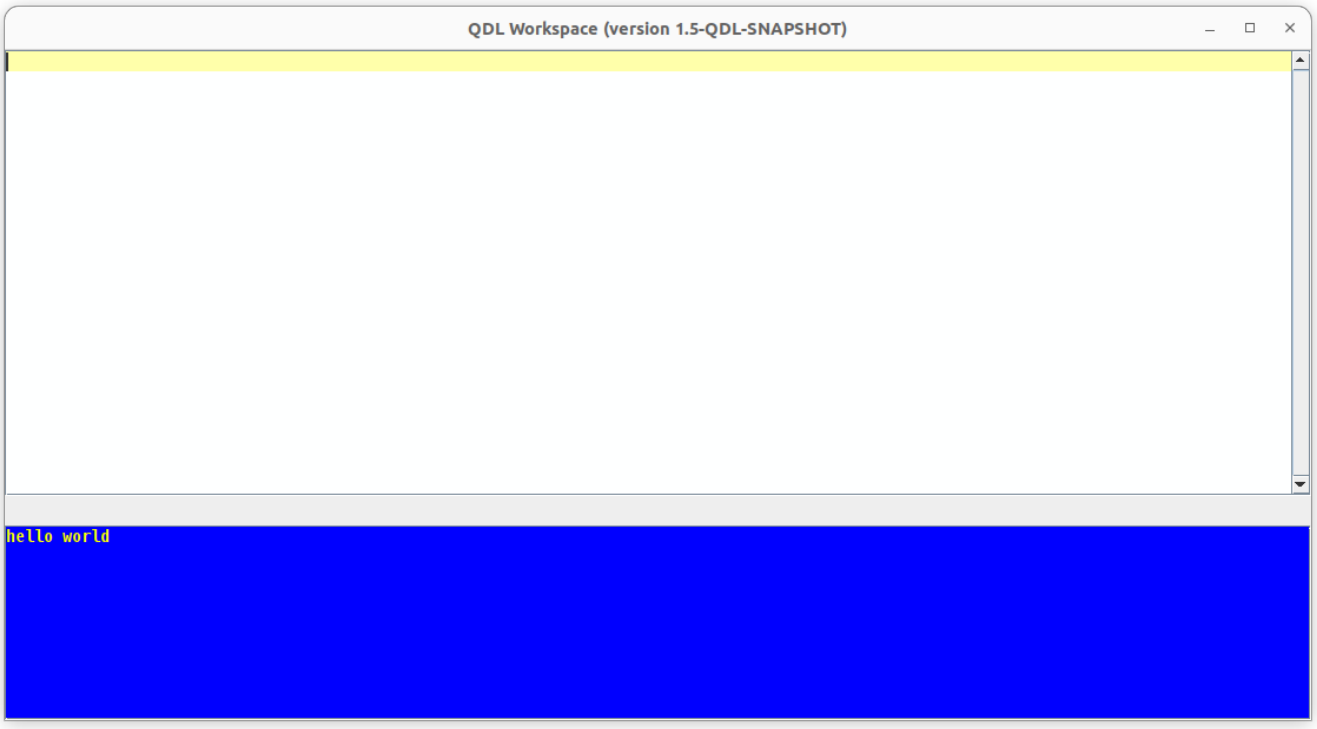
Of course, you may customize startup to do any number of things and the reference for that is here: qdl configuration.

First step: QDL as a calculator (Swing mode)

If you have started QDL as a swing app (add **-tty swing** to the invocation), you should see something like this:



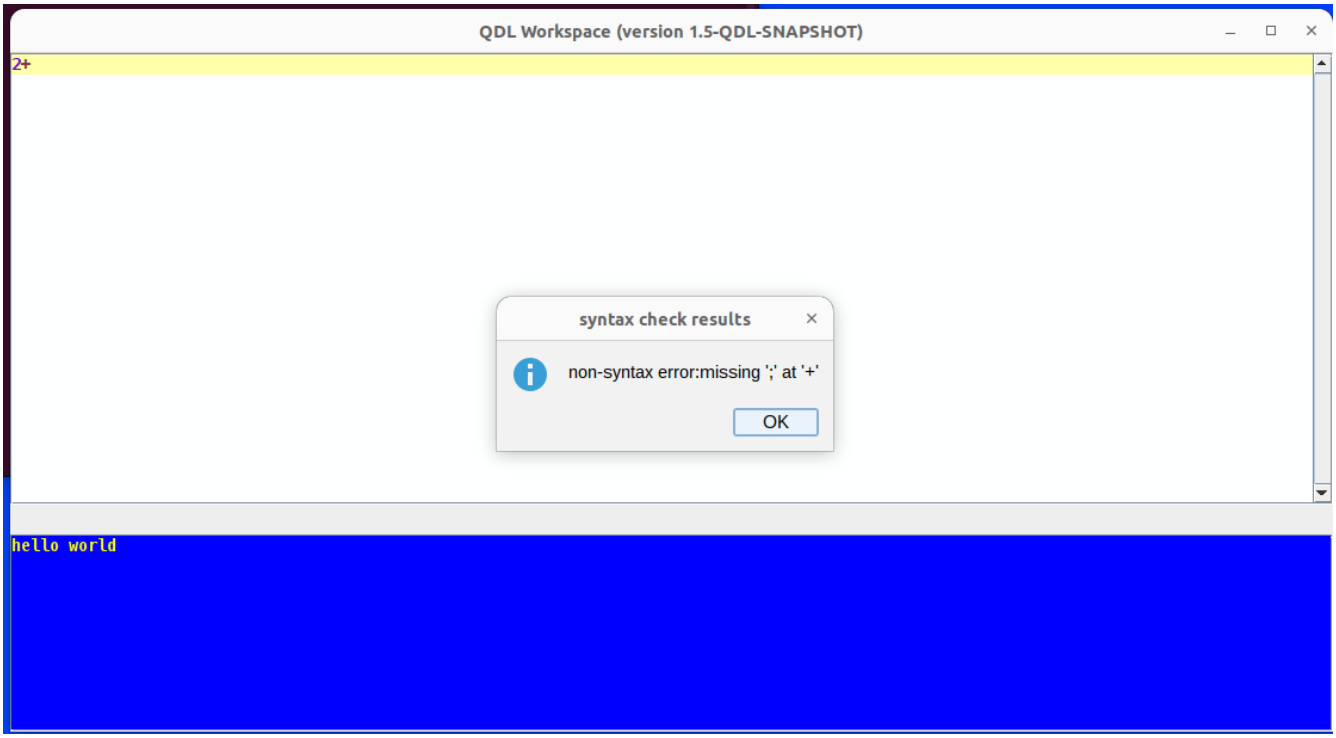
and can set to work. The top (white) window is input, the lower (blue) window is output. You execute an expression by typing it in, then pressing ctrl+alt+enter, referred to as *evaluating* the QDL expression. Every QDL expression ends with a semi-colon, but in echo mode, this is added for you if it is missing (at least for single statements, if there are multiples you have to add them) and the result is echoed to the output window. So for the ubiquitous first “Hello world” program, you type it in as a string (single quotes) and evaluate it:



Several things happened. First off, the input window cleared, ready for more and then the result was echoed to the output window. There is a command history, accessible with ctrl+alt+pg up or ctrl+alt+pg down.

Checking syntax

You may check the syntax of the input window – or a selection of it – by pressing ctrl + h. So let's make a mistake. Enter `2 +` in the input window, then hit ctrl+h:

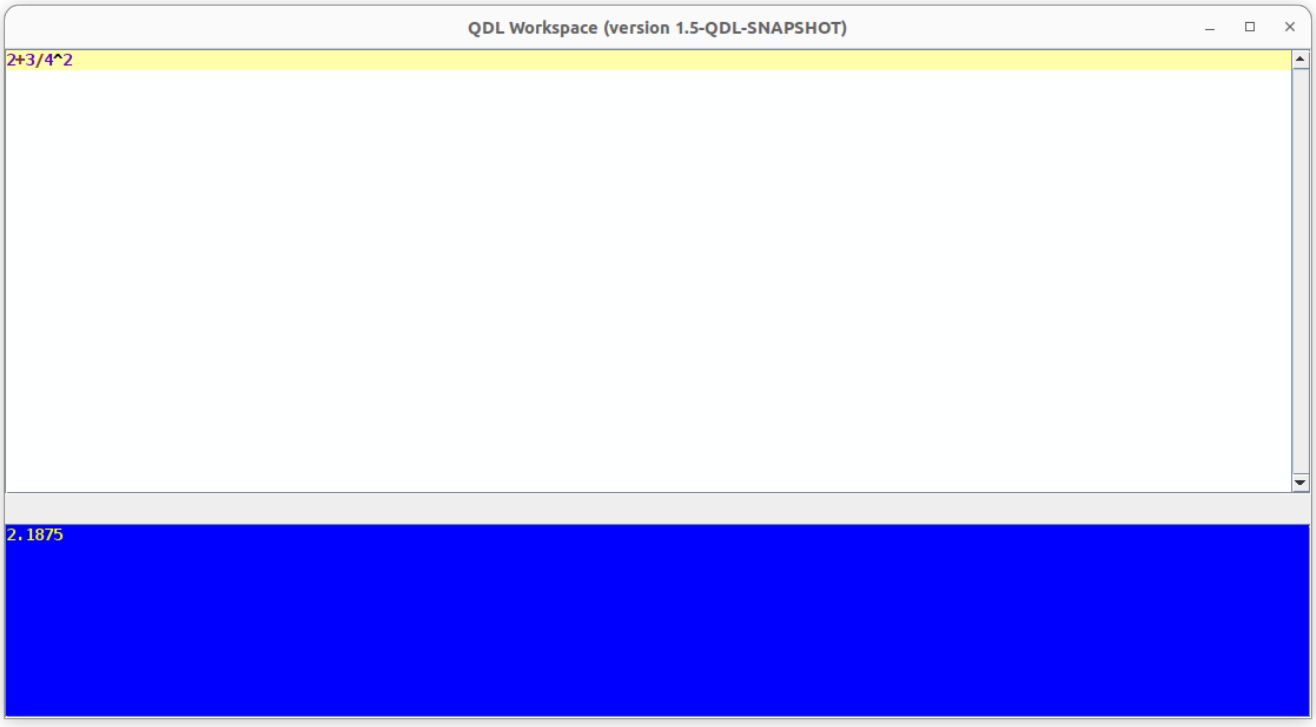


This is telling us there was an error (in this case there was no ending) ; so parsing terminated early. In short *there is never a reason to have a syntax error in QDL*. There are many other editor features which you may read by issuing ctrl+F1.

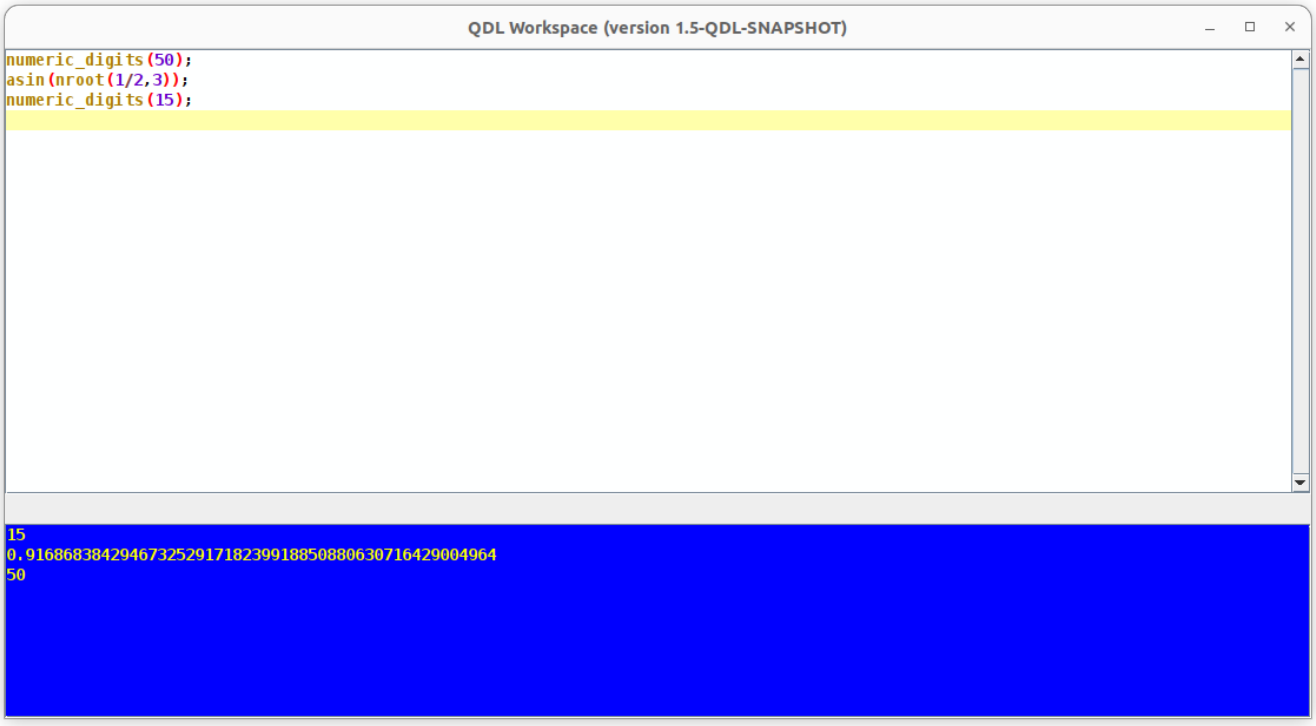
Entering expressions

QDL supports the full range of algebraic operations such as + - * / ^ and % (the last one is integer division) along with the expected standard order of operations. So to evaluate $2+3/4^2$

Note that I did ctrl+alt+pg up to show the input too!

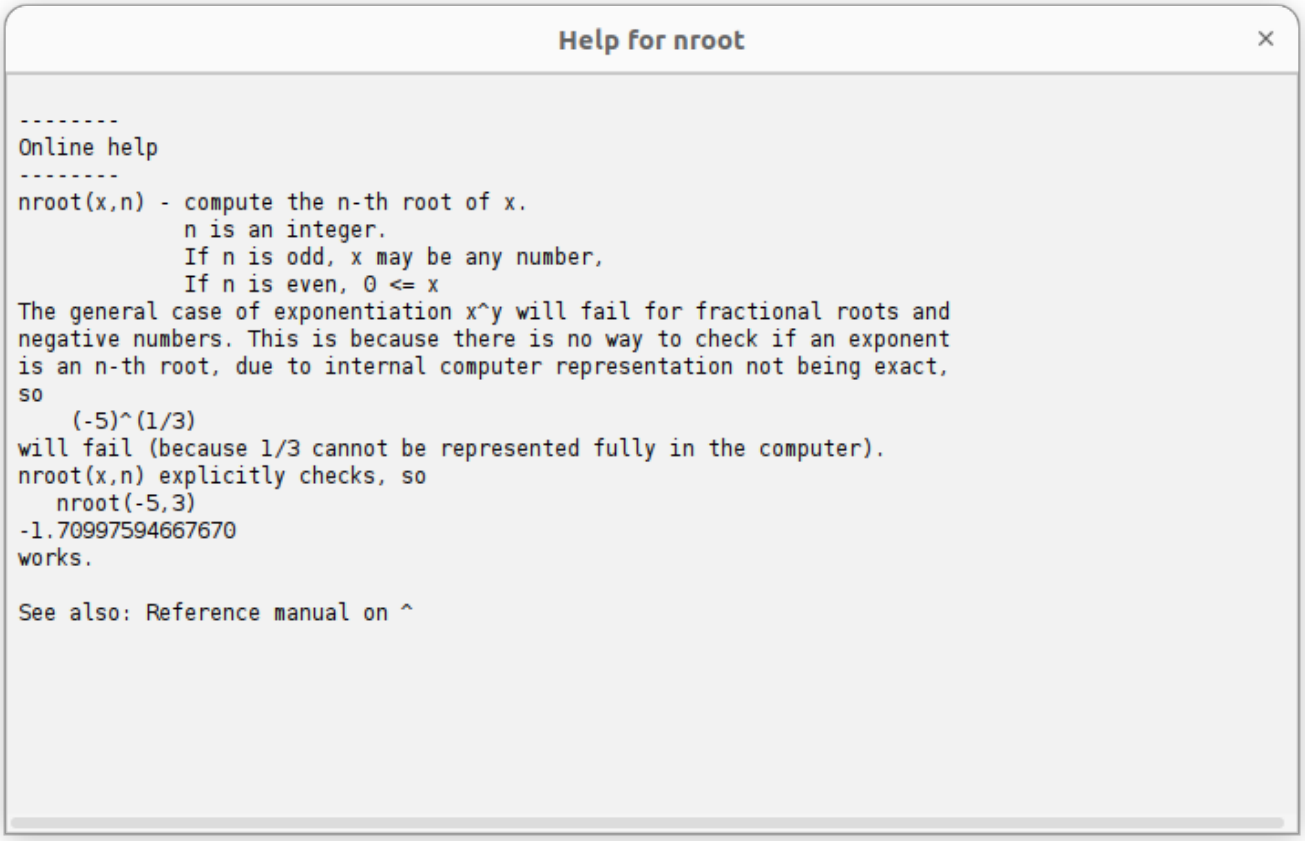


There are also all the standard math functions, such as sine, sinh, asin, log (base *e*) and so forth. If you want to get carried away, QDL even supports arbitrary precision, so you can issue set numeric_digits and compute away. Note that when you type there is command completion (which I cannot seem to capture in a screenshot). Just it ctrl+space and a list of options will appear. In the nextexample, I set the precision to 50 digits, compute the $\sin^{-1}(\sqrt[3]{}(1/2))$, then set the value back. Each line's output is echoed.

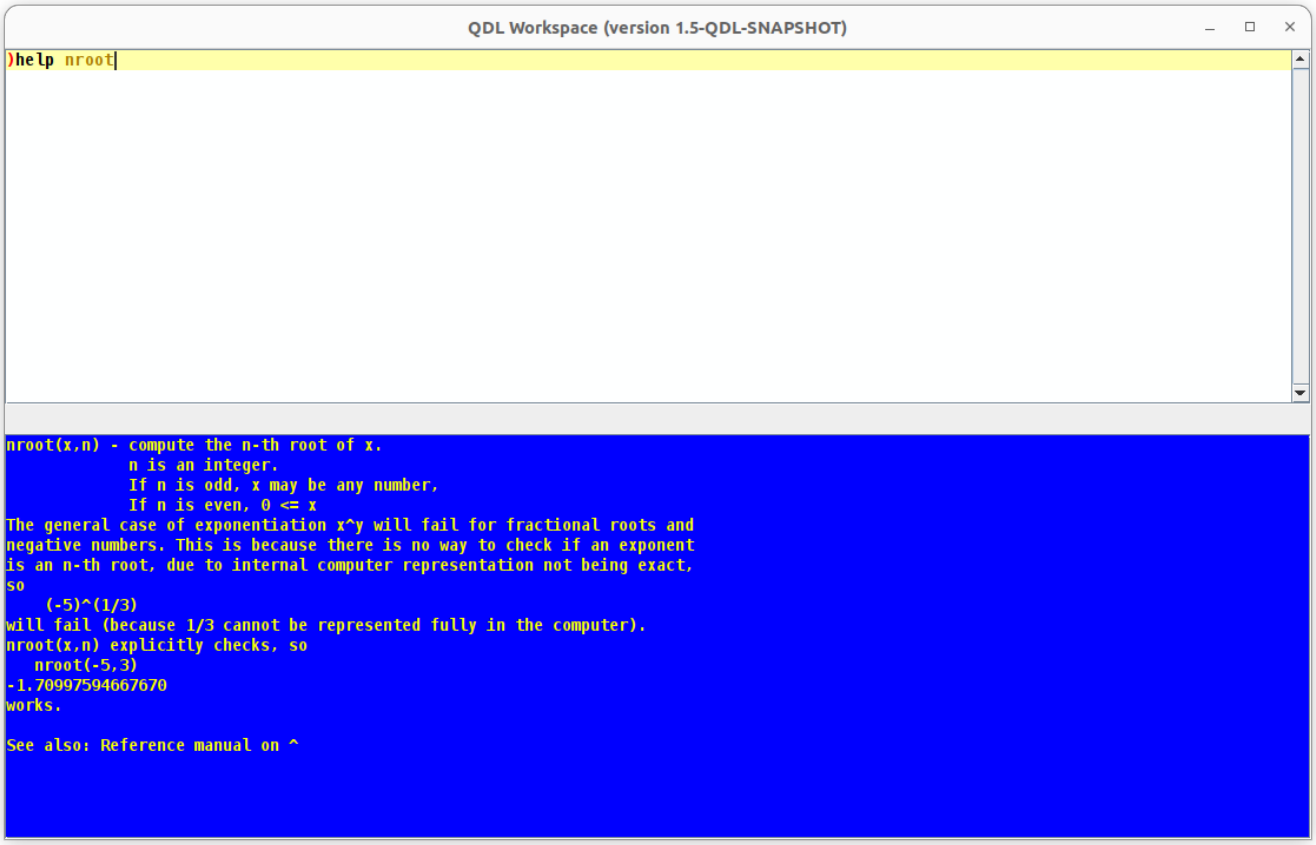


Getting help

Don’t know what the nroot function is? Highlight it and hit F1 and you will get the following



Every base function in QDL comes with help like this, including examples. Alternately, you can invoke it directly using the)help workspace command and execute it:



Second step: QDL from the command line.

If you start QDL straight out of the box, it comes up in a terminal window:

```
$bash>qdl
-- -- @ --  / -- @ @  / @ -- @ @
  .g8""8q. `7MM""""Yb. `7MMF'
.dP'    `YM. MM    `Yb. MM
dM'      `MM MM     `Mb MM
MM        MM MM      MM MM
MM.      ,MP MM     ,MP MM      ,
`Mb.    ,dP' MM    ,dP' MM     ,M
  `"bmmd"' .JMMmmmdP' .JMMmmmmMMM
      MMb
       `bood'
-- -- @ --  / -- @ @  / @ -- @ @
Welcome to the QDL Workspace
Version 1.5-QDL-SNAPSHOT
Type )help for help.
****************************************
ISO 6429 terminal
```

This tells you that cursor addressing is available (ISO 6429) and that QDL is ready. Obviously, there is not the same functionality as in the Swing version, but this does run on servers and is full featured. If you are using the OA4MP extensions, it is most likely you will only be using this on a server remotely.

The basic pattern is the prompt is 3 spaces and the output is printed immediately. So to evaluate 2+2 you would issue

```
    2+2
4
```

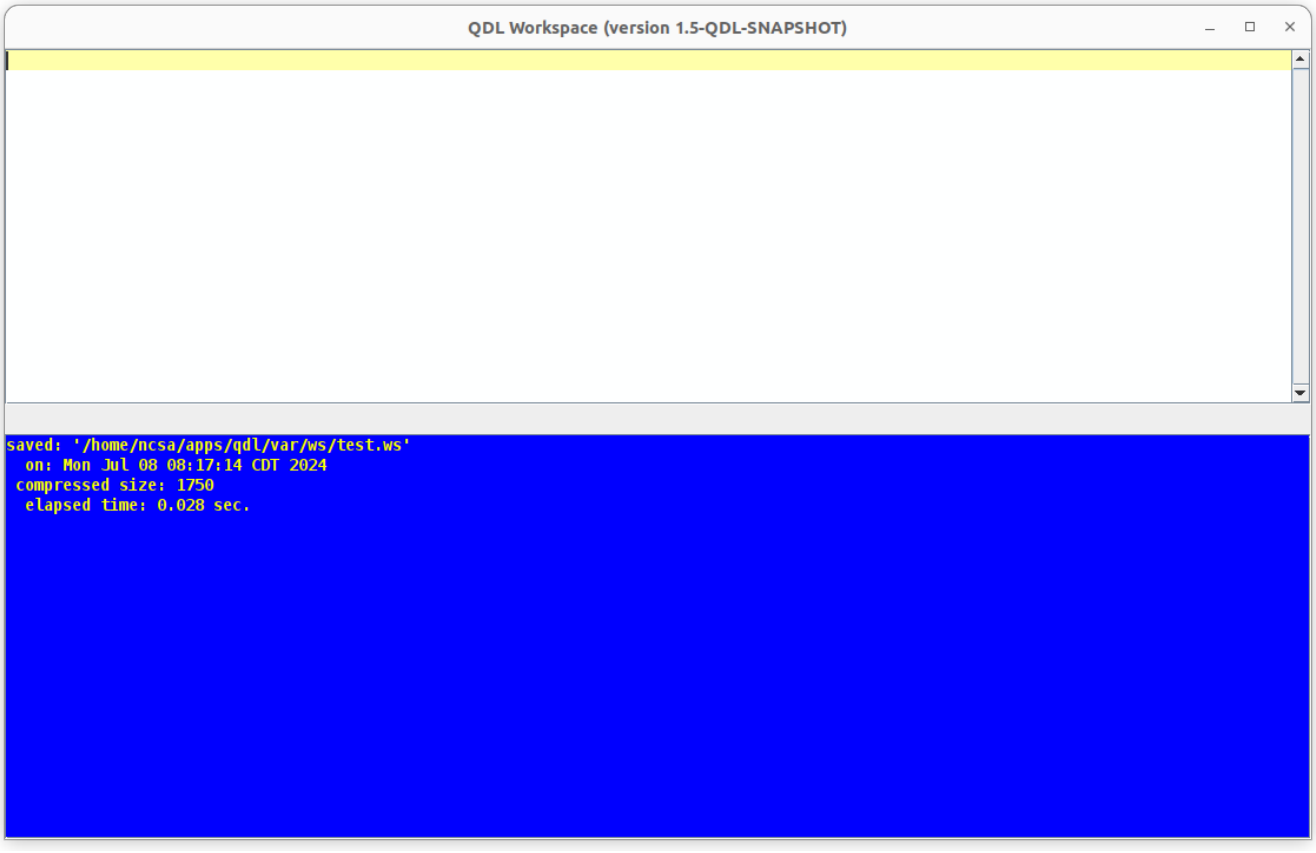
The general way of showing QDL in this document is via the text interface. The inputs are given and then the output is displayed.

Workspaces

The basic unit of organization in QDL is the *workspace* which consists of saving your entire state and command history. You interact with it by issuing workspace commands, each of which starts with a) (close parenthesis). *E.g.* type **`)help`**

Saving your work

The basic unit of work in QDL is the *workspace* You may, of course, customize a lot of it, but just using the defaults, you Just issue **`)save name`**



In the future, you can just issue **`)save`** and the workspace will be saved *in toto*. You may reload it with the **`)load name`** command. Every workspace command can have help invoked by added the **`--help`**

(note the double hyphen!) option. The next section tells you how the workspace operates from the command line and then there are more example on using expressions.

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. It is, in effect, QDL's built in IDE (integrated development environment). 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

And expression returns a result. This is different that a statement. A statement is given with square brackets and denotes local state. We won't use statements much in the sequel.

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. Here is a short list of the essentials to start off:

```
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*. There is a full compliment of functions for numbers such as the basic trigonometric functions, gcd, lcm, logarithms and such. (Look for transcendental functions in the reference manual).

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. No I have no ability or background in this language, I just like calligraphy.) 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
```

which also allows for multiplication with integers:

```
   'abcd' + 10*'.'
abcd..........
```

and you can compare strings with the equals operators

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

and the operators <. <=, >, >= which test for substrings:

```
    'cd' < 'abcd'
true
```

and finally there is even a regex test, =~

```
   regex=~ target
```

where the regex is a string. Here we test if a string is nothing but digits:

```
   '[0-9]+' =~ '321465'
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
```

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.

size(

```
   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.

Online help for functions

QDL comes with many build in functions. A full list can be seen with

```
  )funcs system
abs([1])                file_write([1,2,3])       list_keys([1])
script_args([0,1])      π([0,1])
acos([1])
… lots more!
161 total functions
```

The way to read this *e.g.* **`abs([1])`** a function named **`abs`** that takes a single argument. To get help, issue

```
  )help abs
abs(arg) - find the absolute value of a number or stem.
use -ex to see examples for this topic.
```

The function reference should be consulted. If you have defined a function, any help you have documented will be shown.

Defining your own function.

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

```
def -> expression
```

which will evaluate the statement and return the result.

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. The contract for a single expression is to simply return the value of the expression. If you want to write more complex, mutliline functions you can use the **`define[]`** statement:

```
   define[f(x,y)][y := abs(y);return(x*nroot(y,2));];
   f(3,-16)
12
```

which returns the product of x and the square root of |y|.

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.

Definitioin

The more familiar map works the same

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

or set them explicitly

```
  y.foo:='bar'; y.fnord:='baz';
  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.

Basic operations on stems

Stems as arguments to functions

Another example is to replace one string by another. For this we need a stem (see next section if needed) to make a correspondence between the old and new values

```
  replace('asdqweasdzxcasdertasdcvb', {'asd':' woof! '})
 woof! qwe woof! zxc woof! ert woof! cvb
```

Here we take our string above and replace 'asd' by ' woof! '

Remove an element

```
remove(x.0)
```

Removes the given key from this stem

Add an element

The easiest way is to use the join or tilde operator, ~.

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

And yes, you can add whole stems to each other. Note that the output is a new stem, and does not alter the arguments.

```
  y. ~ x.
{0:2, 1:4, 2:6, 3:foo, arf:woof, fnord:baz, foo:bar}
```

Change an element

Just set it with the assignment operator

```
y.'foo' := 42
```

Does an element exist?

```
   is_defined(y.'foo')
true
   is_defined(y.'my_key')
false
```

Note that this really tests if the specific key is in the stem

Count the elements in a stem

```
   size(y.)
3
```

Does the stem contain a value

```
   has_value(y., 'bar')
true
```

Example comparing Python and QDL

In python, the syntax for lists is rather similar and they have a concept of a “list comprehension” which is simply having an embedded loop to do something.

Python:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]
print(newlist)
['apple', 'banana', 'mango']
```

in QDL, you do this with a lambda function that is passed to the pick function

```
  fruits. := ['apple','banana','cherry','kiwi','mango']
  pick((v)->'a'<v, fruits.)
{0:apple, 1:banana, 4:mango}
```

Note that this picks them and tells you the indices where they were found, if you want to just re-order the list, use the ~ (tilde) operator:

```
  ~pick((v)->'a'<v, fruits.)
[apple,banana,mango]
```

Extension

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.

The major paradigm shift for most people

When starting to use QDL, the one paradigm shift that is hardest to wrap their heads around is the implicit looping and aggregate nature. So if a naive programmer wanted to fill up a stem with the values of the polynomial x^2 - 3x +5 over the interval [-1,1] in increments of 0.2, they would quite reasonably write

```
a. := null;
while[
    for_next(i, 11)
  ][
    x := i*0.2 - 1; // increment to the next element starting at -1
    a.i := x^2 - 3*x -+ 5;
  ];
```

à la C or Java, which certainly does it. The native QDL way is just

```
   x. := [|-1;1;11|]; // fills up stem with correct values
   a. := x.^2 - 3*x. + 5;
```

which is a lot more readable and can be digested at a glance. It is possible to simply and clearly write extremely complex expression in QDL that would be close to nightmarish in other languages on this account. If you can write a single term of the expression, then all the bookkeeping, looping etc is just done for you.

---

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.

Moreover, QDL is almost a picture language for working with data. Let us say that you had a pair of stems to compare using a function, *f*. So `f(x,y)` returns true or false for the arguments. In QDL, this would mean that `f(x.,y.)` operating on stems would return another stem of boolean values. How would you check that every element of `f(x.,y.)` is true? Well, since it may look like

```
   f(x.,y.)
[true, true, false, true, true,true,true,true]
```

the most intuitive way to deal with it would be to slap logical and && between each element. QDL has an operator (plus many more of other sorts) to do that:

```
   reduce(@&&, f(x.,y.))
false
```

All `reduce` does is take the first argument (@&& means to take the operator && and use that as a thing itself) and distribute it to everything so conceptually it is

```
     &&    &&     &&    &&    &&    &&    &&
[true, true, false, true, true, true, true, true]
```

There are many more such operators in QDL that can make your programming quite simple and intuitive.

A few more quick examples for stems

```
    a := 3;          // assigns 3   to  a
    b := 4;          //    "    4   to  b
    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. This uses a stem in an unexpected way, as an accumulator for values.

```
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 lambsie divey
{*:0, doats:2, lambsie: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 with default 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. Convenient in formatting business reports, a nightmare in scientific computing. 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. If you are “missing” some of the answers you expected, this means point blank you were missing data to start with and now know where to track down the problem.

Lists

Lists are stems with integer keys. You can create them directly:

```
   x. := [2,4,6,'foo']
```

or just assign the values explicitly

```
   x.0 := 2; x.1:= 4; x.2 := 6; x.3 := 'foo';
   x.
[2,3,5, foo]
```

Many functions return lists. For instance, the **`index_of(haystack ,needle)`** function returns the indices of needle in haystack

```
  index_of('asdqweasdzxcasdertasdcvb','asd')
[0,6,12,18]
```

That is to say, asd is found at locations 0, 6, 12 and 18 in the left argument.

Sets

A ***set*** is a collection of unique, unordered objects. In QDL we write these between curly braces:

```
  {'a', 1, 2, true, 44/3}
{a,1,2,14.666666666666666,true}
```

Sets may include other sets. There is one empty set, written either **`{}`** or ∅. Standard operations are supported such as intersection (/\), union (\/), symmetric difference (%), subsets (<. <=) and equality(==, !=). Of particular note is that operations on sets are extensible, so

```
  3+ {3, 4, '4'}
{34,6,7}
```

(Note that + is overloaded for strings, so 3 + '4' results in '34') as do functions

```
   f(x)→x^2
   f({1,2,3,4})
{16,1,9,4}
```

Do note again that sets are unordered.

Here is a bunch of basic set operations for reference. QDL has a full set of operations available which can be remarkably handy and make code much more concise.

```
   ⊢[-2;5]; // (alt+s or |^) change the values of a stem to a set
{-1,0,-2,1,2,3,4}
   |^{'a':'b','foo':'bar'}; // another example of converting the values of a stem
{b,bar}
   {}<{1}; // null set is a subset of every set
true
   1∈{1,2}; // (∈ is alt+e) test element
true
   has_value(1,{1,2}); // same as previous, ASCII only
true
   {1}∈{1,2}; // elements are not subset
false
   {1}∈{{1},{2}}; // unless you actually make them so
true
   [;5]∈{1,2,4,5}; // has element is left conformable
[false,false,true,false,true];
    // This asks if the single set on the LHS is an element on the RHS
   {0,1,2,3,4}∈{1,2,4,5};
false
   {1,2,5,-2} ∪ {2,8,-1} ; ∪ (alt + u) or \/
{-1,1,-2,2,5,8}
   {1,2,5,-2} ∩ {2,8,-1}; ∩ (alt+U) or /\
{2}
   {1,2,5,-2} Δ {2,8,-1}; Δ (alt+D) or %, everything outside of the intersection
{-1,1,-2,5,8}
```

```
   {1,2,5,-2} / {2,8,-1}; // remove the RHS elements from the LHS
{1,-2,5}
   // how to loop over the common elements of two sets
   while[j∈{-1,0,1,2,3}∩{0,2,3,-4}][say(j);]
0
2
```