QDL Reference Manual

Version 1.6

Introduction

This document is the reference manual for the QDL (pronounced “quiddle”) jokingly referred to as the ***Q***uick and ***D***irty ***L***anguage, which was specific to the OA4MP system and is used for all manner of server side-scripting. The aim of it is to have a reasonable language that is as minimal as possible and allows for a wide range of operations on claims and server-side management of clients.

Concise synopsis of what QDL is

- An aggregate processing language with rank polymorphism
- The basic aggregate is a stem which is a generalized map.
- Operations are extended to aggregates via “freshman Algebra” i.e., linearized.
- Offers one solution to Backus' “von Neumann bottleneck”
- Many control structures are implicit in the data structures.
- Solely aggregate operations are sub-Turing, hence explicit control structures are present and Church's λ-calculus is supported.

Origins

“Quiddle” by itself has two meaning.

1. (*noun*) A small or trifling thing
2. (*verb*)Treating a serious topic in a trifling way.

QDL (as a moniker) comes from aviation navigation (“Q-codes”) which refers not to something trifling at all but a (usually essential) set of navigation bearings taken at regular intervals. Usually these are critical and needed for course corrections when a pilot cannot use their instruments (e.g. the aircraft is in severe weather, has no visibility and cross winds make judging actual speed and bearing impossible, aka it's a fix for flying blind.) In the original scripting environment, this ran on a OAuth server for certain types of additional processing. Scripting was called at regular intervals to do implementation specific tasks (such as acquiring metadata and monitoring the control flow.) As it were, scripting keeps the OAuth flow on course. (The logo has the Morse code for Q-D-L in it, by the way.)

The second meaning of quiddle is a bit more lighthearted. *E.g.* Monty Python quiddled the Middle Ages in their movie “Monty Python and the Holy Grail.” This is not a trivial language – far from it – but rather than take the approach of having an exhaustively complete language this is purposefully as

minimal as possible. The specification is barely a page and it should take nobody with a smattering of computer background more than 15 - 20 minutes to be writing productively in it. Life is too short to learn another C++…

The way this is done is that the data types for the language (stems and scalars) have embedded control structures and additionally there are major constructs for the language (looping, conditionals, encapsulation &c.) which are very bare-bones. This covers the dictum of aiming at the probabilities (what you are most likely to do) not the possibilities (what your wildest dreams envision). So you may need a loop and there is one – just one – rather than having several with niggling syntax that covers every possible variation. If you need something more, than there are tools to cobble it together.

(A much more complete language was designed and a specific subset implemented, so if there is a hue and cry for something more, it would actually be pretty easy to add it. In other words, backwards compatibility is built in for future extensions. Just saying it so we are clear.)

A bit of motivation about aggregate variables since this strikes many people as offbeat. QDL works very well as a server-side policy language. A very common situation is having to configure many rules for doing fairly simple tasks, *e.g.*, if a user has logged in through an institution, belongs to any of a few groups, has an affiliation with a third institution, then some subset of information should be returned. In large blocks

huge rats nest of conditions

==> *do something conceptually easy but very repetitive to a large set of data*

So this language needs to have a lot of horsepower for decision making and very simple ways to work on large data sets. On top of this, since these scripts frequently run on server, speed is essential and implicit looping (discussed later with stem variables) makes this quite a snap. Think of it as a notation that happens to have some control structures.

> How did this start? Basically I would write out high level algorithms using this sort of notation, then implement them in whatever language I needed. It dawned on me to cut out the middle step. However, a notation is not a programming language, so QDL was created that had the bare bones constructs for actually running it on a computer. This minimalist approach is everywhere, since creating a new computer language normally leads to lots of bloat to cover all sorts of general purpose cases. If you need something, write it, but the language itself should be the toolkit for such things, not (like C++ or Java) an ever growing welter of constructs and arcane libraries that make programming sometimes quite a chore. Nobody knows all of Java or C++ because it is not humanly possible.

But but but… what about structured programming, object oriented programming and all of the other paradigms? Those exist so programmers don't shoot themselves in the foot. QDL is, as I have stated, a notation with some control structures so it fits on a computer. The fact you can write inefficient code or some such is no more a criticism of it than complaining that if the alphabet lets you write gibberish, it should be banned. People write terrible code in C/C++, Java, Perl (kinda built in, actually), etc. *ad*

nauseam, all the time which can get hidden in all of the things[1] designed to prevent bad code. In QDL the onus is on the programmer to write elegantly.

Cheat Sheet

Constant types: null, boolean, number, string

Variable types: scalar, compound (aka stem variables), sets

Control structures: if..then...else, while loop, switch statement, try...catch.

Encapsulation: module

Getting QDL

The instructions for getting the latest distribution of the installer are here.

Running QDL

Invoking QDL is discussed under User Interfaces to QDL

Using QDL as a calculator

One great use of QDL is as a "programmer's calculator," meaning that it has any number of interactive tools that programmers often need, such as encoding/decoding strings in various formats, parsing messy JSON blobs, creating configurations in YAML or JSON, repaid prototyping. It frequently happens that having a QDL instance running someplace is just too convenient.

You generally want the interpreter to simply echo back results. The Swing GUI does this, but if you are runnign it from the command line issue

```
)ws set echo on
)ws set pp on
```

which sets echo mode on and enables pretty printing of output.

Basic syntactic concepts.

1. Weakly typed
2. Simple and stem variables
3. A very full set of logic/algebraic operations
4. A full but minimal set of control structures

---

1 Such as Design Patterns from the Gang of Four. The original was a very insightful foray and rumination on oft seen patterns and is invaluable. I can't count the number of times I've seen a welter of objects in Java trying to implement a design pattern (for the sake of using patterns since we all know they solve all bad programming issues, right?) when a few lines of well aimed code would be more understandable, perform better and be far more maintainable. Design patterns in such cases are more like talismans with magic incantations.

Constants and expressions

There are boolean, number (integers are (64 bit), decimal are unlimited, if you exceed 64 bits, it will get converted to a large decimal (using scientific notation)), string and stems. Valid identifiers a-z, upper and lower case, $_ and digits. Variables may not start with a digit. There is not limit to the length of a variable name. These are all fine:

```
a___$
$holyCow
__internal_function
```

Note that in some cases, the dollar sign may be used for escaping disallowed characters. See *encode* and *decode* for type 0 and the particulars as relates to working with, *e.g.*, JSON.

QDL supports the following standard algebraic operations:

```
+ - * / % ^ ⌈ ⌊
```

as well as several operations on sets (see below).

Note that the operator % is the integer part of division. So

```
   42/9
4.666666667
   42%9
4
   14.2%7.5
1
```

Raising a number to a power works with all exponents, so

```
   1.2^1.3
1.2674639621271
```

So to get the square root of a number, raise it to the 0.5 power. As a limitation of computers, the base must be non-negative. For integer n^{th} roots, see the function reference for **`nroot(x,n)`**.

The floor and ceiling operators, resp. ⌊, ⌈ (unicode 230a, 2308) are supported. Note that there are standard functions named **`floor()`** and **`ceiling()`** if you do not want to use the special symbols for them:

`⌊x == floor(x)` and **`⌈x == ceiling(x)`** are always true.

Note: Decimals do not require a leading number. So **`.3`** is acceptable. However, it is good practice to have one since this prevents ambiguity with stem variables when reading code.

Note: Decimal exponents require the base be non-negative. You may wonder why ^ will fail for something like **`(-2)^0.333`**, isn't that basically the cube root? Note that **`0.333 = 333/1000`** hence this actually means **`((-2)^(1/1000))^333`** said more plainly, ***every*** decimal exponent is an even root on

a computer(!) There is, of course, a way to do this in QDL, see **nroot** which allows you to take integer nth roots, in this example, **nroot(-2,3)** (or **3√-2** is you prefer operator notation)

Other notations for numbers

It is easy to write *scientific notation* in QDL. This is of the form **m*10^k**. The so-called *standard form* has m restricted to a single digit decimal. So **350 = 3.5*10^2**. QDL does no specific parsing since this is just arithmetic.

QDL also supports *engineering notation* for numbers. These are of the form

*decimal***E***exponent* or *decimal***e***exponent*

Note the the exponent must be an integer but may preceded by a + or - sign. You may have any decimal for the left side, and QDL will normalize it to a single digit, adjusting the exponent as needed.

For instance

```
  1234.567E5; // show how numbers are normalized.
1.234567E+8

  2.34E5/5.67e3; // == 234000/5670
41.269841269841269

    2.34E5*5.67E-3; // == 234000*.00567
1326.78

  2^1.2E2; // == 2^120
1.32922799578492E+36
```

But you must use this format. Entering something like this won't work

```
  2E-3
syntax error:line 1:1 missing ';' at 'E'
  2.0E-3
0.002
```

Note: the exponential function for base *e* is supported and is `exp(r)`.

QDL also supports the following logical operators

```
< <= > >= && || ! =~ <<
```

and the following increment and decrement operators, which may be used before or after variables.

```
-- ++
```

The usual convention is in force, meaning that postfix returns the current value, then carries out the operation. Prefixing means the value is updated then returned. Doing a simple example in the workspace:

```
  i := 2
  i++
```

```
2
  i
3
  ++i
4
  i
4
```

The limited character set for *symbols* is intentional since this sidesteps running it on systems that may have character encoding issues or more usually, working on a system where the supported terminal types are very limited. Generally however, the contents of a string may be UTF-8 with no issues.

Inline conditional expressions

There is a control structure of **`if[ … ]then[ … ]else[ … ]`** but this is a pretty heavy weight solution. There are blocks with local state inside the []. See below. Anything with square brackets in QDL is referred to as a **statement** and is a syntactical unit. Statements do not return values. **Expressions**, however do return values. If you just need to check a conditional, QDL also supports two different (ternary) syntax expression of

```
boolean |boolean. ?|⇒ expression0 {: expression1}
boolean. ?!|¿ expression0. {: expression1}
```

For the first, this means that the boolean is evaluated and if ***true*** then **`expression0`** is returned and if ***false*** then **`expression1`** is returned. (**`expression1`** may be omitted in which case it is assumed to be null.) The boolean is a scalar, or a simple stem (so a list or stem with no nesting). More to the point, the inline conditional is “just another expression”, so you can nest these or use them pretty much any place you want as if they were algebraic quantities. The major difference is scope – the conditional allows for variables in the bodies of the blocks and you can write even whole programs there. The inline conditional is for all the much more simple cases.

For the second this is an exclusive or construction (called a switch or select statement in other languages) and the left argument is a boolean stem with *at most* a single true value. Again, if the else clause is omitted (which is the default if all booleans are false), a null is returned.

E.g.s

```
   x := mod(date_ms(), 2); // equals 0 or 1 randomly
   say(3*(x==0?4:5));
15
// guess x == 1 this time...
```

This shows that the inline conditional is just another expression.

```
    pi()^exp() < exp()^pi() ? 'left is bigger' : 'right' + ' is' + ' bigger'
left is bigger
    3 < 2 ? 4>3 ?'a':'b':'c'
c
```

Note that the last one could be written with parentheses, **`(3 < 2) ? ((4 > 3) ?'a':'b'):'c'`** but the aim was to show that it is resolvable as written. The precedence of inline conditionals is generally

about the lowest, so everything else gets evaluated first. Finally, the else clause (after the **:**) is not optional.

In place of ? you may also use logical implies (unicode 21d2, ⇒):

```
    pi()^exp() < exp()^pi() ⇒ 'left is bigger' : 'right' + ' is' + ' bigger'
```

Conditional example with a stem as the left argument.

If the first argument is a stem or list, then a conformable result is returned. In this example, a list, p., contains strings and we need to make sure all of them are terminated with a /. We check with a regex and use the ternary operator to return a slash if needed or an empty string if not.

```
   p. := ['a:/x/y','a:/x/z/','b:/p/q']
   p.+('.*/' =~ p.?'':'/');// regex to test if an element ends with a slash
[a:/x/y/,a:/x/z/,b:/p/q/]
```

The stem does not have to be a list. There is no subsetting that occurs with the expressions and they are returned as is:

```
   a.0 := false; a.'foo' := true; a.3 := false;
   a.?[-2;2]:{0,1}
{foo: [-2,-1,0,1], 0:{0,1}, 3:{0,1}}
```

Switch/Select example

```
 rc:= script_load(
 ['post_auth','post_token','post_refresh']==exec_phase
¿['auth.qdl','token.qdl','rtx.qdl']:'init.qdl');

  if[rc != 0]
then[ //… other stuff
```

In this case, one of four scripts will execute, setting a **rc** (return code) variable for further processing.

Comparison with select, conditional and mask.

The mask function at first glance seems like the same as ?! or perhaps ?:

```
   mask([false,true],[1,3])
3
   [false,true]?![1,3]
3
```

So what is the difference?

> In mask, the argument is evaluated. In select, only the true value is evaluated.

So you can do this

```
   a:=4;
   ∃[a,b]?![a++,b++]; // Asaumption is that a or b exists, increment which exists
4
   a
5
```

where using mask would fail since it would attempt to evaluate b++ when it does not exist. In the same way, the argument to ? is only evaulated if it is true:

```
   a :=4;
   ∃a?a++:0; // Assumption is that a or b exists, increment which exists
4
```

Monadic operators, or, one gotcha.

You should be aware that the negation operator, **!** or **¬** is a monadic operator, hence it affects everything to its right, it is not part of the value. This goes for the negative of a number as well. -3 means take three and apply the negative operator to it and will result in the *number* negative 3. Continuing with logical negation, these are equivalent

```
   !a<2 && b < 3 <==> !(a<2 && b<3)
```

If you wanted ! to apply to the left hand expression, you would write

```
   (!a<2) && b < 3
```

If you intended the latter, and wrote the former, you would get the opposite value than you expect. The same goes for monaic minus and plus – they affect the result to their right. Usually this is what you want, just be aware. Here is about the simplest example of its operation

```
  ! true && false
true
   (! true) && false
false
```

Again, in the first example, everything to the right is evaluated then fed to the ! (which is a monadic operator). Why does this work differently? Because QDL allows for many operators (like ++) which do not normally live on the order of operations chart. It is simpler conceptually to treat monadic and dyadic operators in a uniform way. Standard operators like multiplication and division do have standard order of operations, but anything else does not have a canonical place there, so parentheses should be used.

The type operator, <<

The basic types in QDL are Null, Boolean, String, Integer, Decimal, Number, Stem, List and Set. You may check a value against these with the type operator, **<<**, which returns a true or false.

```
   [;5] << Stem
true
   [;5] << Integer
false
   5 << Boolean
false
```

A little difference between the type operator and other operators is that it works on the entire left hand argument rather than being extended to each element.

Notes

1. You should make it a point of either enclosing leading negative numbers in parentheses to cut down on ambiguity or just use alternate lead minus and plus signs (¯, unicode 00af, or ⁺, unicode 207a). Also be careful of spaces, since "++" and others are properly digraphs and will be interpreted differently than "+ +". How should QDL interpret

```
a---b
```

Should this mean (a- -) - b or perhaps a-(- - b)? (QDL will actually do the first, but that is not obvious at all.) Such error can be very hard to track down.

2. QDL does "short-circuit" conditionals, so *e.g.*, in `A && B && C` each of A, B, C will be evaluated only if the previous element evaluates to **`true`** . E.g. `false && true && true` will stop evaluation after the left argument is found to be **`false`** , since the rest of the conditional cannot be **`true`**.

Similarly for `true || false || false` since this must evaluate to true since the first term is **`true`**.

3. QDL allows for chained comparisons, so a statement like `a≤x<b` is perfectly fine and is equivalent to `(a≤x)∧(x<b)`. Similar for `a!=x<b` which is the same as `(a!=x)∧(x<b)`. This includes: Note well that something like `x<a<b` is most emphatically not `(x<a) && (x<b)`!

Assigning values

There is a specific ***operator***, denotes := which is used when setting a variable. This is a fine example:

```
c := 4;
```

As we will see later, you may also use this to assign a value to a stem variable by including the final period:

```
a. := [;3];
```

You may chain these

```
a := b := c := 1;
```

will assign each variable a, b, and c to the value of 1.

You may also use the reverse assignment **`=:`** (the colon goes next to the thing being defined) to do

```
5+3^4 =: x
```

Assignments are just dyadic operators, so they return their assigned value which lets you do assignments in expressions and do things like

```
  d := (false =: c) || true
  d
true
  c
false
```

However, there is a caveat and that is the reverse assignment cannot be chained easily, since it is far to easy to write ambiguous assignment statements (which may end up assigning unexpected values). Generally it is best to use parentheses if you want assignments going in different directions in a single expression.

More assignment operators.

There are also shorthand assignment operators for each basic operation of `+ - * / % ^`. So if *op* is one of these operators (referred to as *overloaded assignments*) then

A *op*= B

is identical to issuing

A := A *op B*

these only exist for left-hand assignments, by the way.

Example:

```
  a := 3;
  a ^= 2
  a
9
```

This takes *a*, which is assigned the value of 3, squares it and assigns the new value of 9 to *a*.

Another example.

```
  A := 'a'
  B := 'b'
  q := A += B += 'c'
  q
abc
  A
abc
  B
bc
```

So in summary, here are all the supported assignment operators

```
:= += -= *= /= %= ^=
```

And to make it clear, the basic assignment operator or := does not require the left-hand side exist before use, but the others do.

List assignments

A special case (read “syntactic sugar”) is to allow for multiple variables to be set in list form. Each element in the list on the left is assigned the corresponding value from this list on the right. No complex stem structures are allowed and mostly this is just you can make a visually simple set of assignments.

```
  [a, b., c] := [3, [;5], 6]
  a
3
  [a, b., c]
[3,
  [0,1,2,3,4],
 6]
```

This works for overloaded and left assignments too.

```
  [a, b., c] +=  [3, [;5], 6]
  [a, b., c]
[6,
  [0,2,4,6,8],
 12]
```

Weak typing

As we have said, there are 4 primitive or ***scalar*** types: null, boolean, number and string. Booleans are either **true** or **false,** e.g.

```
a := true;
```

Numbers are of two sorts which are used seamlessly as needed. Integers are 64 bits and require no special handling. Decimals are more or less arbitrary precision. We say more or less because if you give the decimal, it will be exact, but in operations (well, division only) where the decimal can't be exact, it is kept to a fixed decimal precision. The default is 15 digits. See numeric_digits() for how to change this, or set it in the configuration.

```
    numeric_digits(50)
15
    say(234234234.234234234*987987349857349);
231420460326946612381192.152285666
   say(1/3);
0.33333333333333333333333333333333333333333333333333
```

The first result is exact because we specified the number of digits. In the second case, there is no exact decimal representation of 1/3, so it is truncated.

Strings are single quote delimited and you may embed single quotes by escaping with a \'. So here is a string:

```
   my_string := 'abcd\'efg';
   say(my_string);
abcd'efg
```

While QDL has very a limited character set for variables, all string are fully UTF-8, *except* control characters are not allowed, though a few of the more common ones may be escaped as per this table:

Sequence	Name	Description
\b	backspace	move cursor back one space
\t	tab	insert tab character
\r	return	return cursor to start of the line
\n	new line	return cursor to start of line and advance to next line
\'	single quote	a single quote
\\	slash	a slash
\uxxxx	unicode	Any non-control unicode character. xxxx is a 4 digit hex value

```
       '\u00f7\u2234\n\u2235'
÷∴
∵
```

Here there are 3 unicode characters and a new line.

Unicode and alternate characters

QDL uses ASCII 7 characters, but a few alternates are also allowed (mostly for replacing digraphs) f you prefer – this is a matter of taste more than anything else.

Standard	Unicode	ALT	unicode escape	What is it
!	¬	!	\u00ac	logical not
-	¯	_	\u00af	unary minus, the negative sign
===	»	'	\u00bb	function/module documentation
*	×	*	\u00d7	multiplication
/	÷	/	\u00f7	division
- >	→	d	\u2192	lambda function
?	⇒	?	\u21d2	alternate for ternary conditional expression
has_value	∈	e	\u2208	set is member of
!has_value	∉	E	\u2209	set is **not** member of
/\	∩	i	\u2229	set intersection
{}	∅	n	\u2205	the empty set
\/	∪	u	\u222a	set union
&&	∧	&	\u2227	logical and
||	∨	|	\u2228	logical or
:=	≔	:	\u2254	left assignment
=:	≕	"	\u2255	right assignment
=~	≈	-	\u2248	regex matches
`	·	.	\u00b7	raised dot
!=	≠	+	\u2260	not equal to
==	≡	=	\u2261	logical equality
<=	≤	<	\u2264	less than or equals
>=	≥	>	\u2265	greater than or equals
ceiling	⌈	k	\u2308	ceiling operator
floor	⌊	l	\u230a	floor operator

|^	⊢	s	\u22a2	set conversion
[|	⟦	{	\u27e6	left closed slice bracket
|]	⟧	}	\u27e7	right closed slice bracket
assert[][]	⊨	a	\u22a8	assert
for_each	∀	A	\u2200	for_each as an operator
is_defined	∃	i	\u2203	is_defined for variables, is_function
!is_defined	∄	I	\u2204	negation of is_defined, is_function
has_key	∋	h	\u2203	has_key
!has_key	∌	H	\u220c	negation of has_key
transpose	⦸	t	\u29b0	transpose operator
expand	⊕	X	\u2295	expand operator
@	⊗	@	\u2297	function reference
reduce	⊙	x	\u2299	reduce operator
mask	∓	∓	\u2306	mask operator
pi	π	p	\u03c0	Greek letter pi.

(If you are running QDL with the **`-ansi`** option, then the ALT characters are available. See the appropriate blurb for more.) You can always get this list in the workspace with **`)help unicode`**. Note that there are some differences in function vs. operator notation. The usual pattern is

function_name(object, args…)

i.e., that the main object the function works on is the first argument. In the operator version, the main object is the left argument,

```
reduce(@f, a.) iff @f⊙a,
```

That said,

```
   f(x) → (0 ≤ x) ∧ (0.7 ≥ x ÷ 11)
   f(2)
true
```

is also a perfectly fine function definition. Greek letters (upper and lower case) are also allowed for function and variable names, but that is again a matter of taste (and keyboard availability). A full table of Greek letters is available in the workspace with **`)help greek`**. Remember that while the escape sequence can be used inside of strings, they do not work at the command line, so

```
\u03a9 \u2254 \u2205; // Fails!
Ω ≔ ∅ ;     // Works!
```

the first will fail. You must use the character (the reason for adding in the alternates is to increase readability).

Since there are many times external programs use double quotes and one of the aims of QDL is to make it interoperate nicely with other languages, using single quotes saves a lot of time dealing with niggling issues about where an extra double quote crept in.

You may also concatenate strings easily using the + operator, so

```
    say('abc'+ '123');
abc123
```

Similarly, the “-” works on strings too and removes the right elements from the left:

```
       say('abcdeababghabijab' - 'ab');
cdeghij
```

Extending + to *, you may create multiple copies of strings

```
    3*'a'
aaa
```

Multiplication for strings is defined for non-negative integers. Multiplying a string by zero returns the empty string. Division is defined as the number of times the left side is found in the right.

```
  'asdasdasd'/'as'
3
  5*'bar'/'arb'
4
```

Exponentiation of strings is not defined.

Strings may be compared as substrings using <, <=, >, >=, ==, != and =~ so

```
    'foo' == 'foo'
true
   'abc' < 'abcd'; // so abc is a proper substring of abcd
true
   'abc'<'abc'; // abc is not a proper substring of itself
false
   'abc'<='abc'; // abc is equal to itself (yup!) or is a proper substring
true
   'foo'<'bar'
false
   'arba'<3*'bar'
true
```

Inequalities test for substring. Cf. `starts_with` which tests if the substring starts on the first character.

Regular expressions

There is support in QDL for regular expressions aka regexes.

Matching

The special comparison of =~ (or ≈) compares the value on the right with a regular expression on the left:

```
   '[a-zA-Z]{3}' =~ 'aBc'; // Checks if the argument has 3 letters
true
    '[Yy][Ee][Ss]' =~ 'yEs'; //Checks that the argument is case insensitive 'yes'
true
  '[0-9]{5}' =~ [234,34567,5432345]; // Check which are 5 digit numbers
[false,true,false]
```

Note that the right hand argument is *always* converted to a string before the regular expression is matched to it.

Unlike many languages, there is no explicit type set forth for most languages and indeed, you may even change the type on the fly without penalty. For instance, this causes no error:

```
my_var := 'Avast ye scurvy dogs!';
my_var := size(my_var);
```

Where in many languages this would raise an exception. This is the “dirty” part of the name: the onus is on the programmer to keep this straight. Variables may contain the letters (upper or lower case), digits, underscore and dollar sign. Variables are case sensitive, so do be careful.

Splitting

Splitting is of the form

```
tokenize(arg, 'regex', true)
```

There is a separate section below about the tokenize function, but this section is to consolidate information about regexes in QDL.

Replacing

Replace using regular expressions is of the form

```
replace(arg, 'regex',  'replacement', true)
```

Note that *replacement* is just a string, not any sort of regular expression. Every place that the *regex* matches in **`arg`** will be replaced with *replacement*. See the section below on **`replace`**.

Reserved keywords

There are a few reserved key words in QDL:

`true`	**`if`**	**`while`**	**`try`**	**`module`**	**`define`**	**`block`**
`false`	**`then`**	**`do`**	**`catch`**	**`body`**		**`local`**

```
null        else        assert

Boolean     String      Null        Integer     Decimal     Number      Stem
List        Set
```

(Capitalized keywords are types, so **`Null`** is the type but **`null`** is a value.) The first two, **`true`** and **`false`** are boolean values. The third, **`null`** is the null value for variables. So these are fine variables in QDL one and all:

```
integer := .5;
boolean := 3.3^11;
decimal := true;
scalar. := random(1000);
```

But

```
if :=2
```

causes a syntax error..

Now as to whether you really *want* to set those variables to those values is your issue. The point is that there are very few such reserved words and they are actually constants. The aim was to keep structures as cleanly separated as possible from code.

Basic Data types. Scalars, Sets and Stems

Scalars

A ***simple*** variable, also called a ***scalar*** consists of primitive types, which are boolean, number (both integer and decimal) and strings. These look just like any other variable from most programming languages (the ":=" is the assignment operator). So for instance

```
a := 'foo';
my_boolean := true;
my_integer := 123;
my_decimal := 432.3454;
b := 'Trăm năm trong cõi người ta, Chữ tài chữ mệnh khéo là ghét nhau.';
```

are all valid simple variables. A fuller discussion of the various basic types follows.

String

String are enclosed between single quotes.:

```
   'Twas brillig and the slithy toves'
Twas brillig and the slithy toves
```

Basic operations

There are many operations on strings. To joing two strings, use the + (catenation) operator:

```
   'mairzy doats' + ' and ' + 'does eat stoats'
mairzy doats and does eat stoats
```

Similarly, to remove a substring form a string, use the – operator:

```
  'One Ring to rule them all, One Ring to find them, One Ring to bring them all' - 'One Ring to '
rule them all, find them, bring them all
```

To replicate a string, use the * (multiplication) operator

```
   3*'w00f '
w00f w00f w00f
```

The division operator tells you how many times a substring occurs

```
  'One Ring to rule them all, One Ring to find them, One Ring to bring them all' / 'One Ring to '
3
```

Comparisons

Equality (==) tells you if two strings match exactly, and is case sensitive

```
   'ragg mopp' == 'Ragg mopp'
false
   'ragg mopp' != 'Ragg mopp'
true
```

Similarly, inequality (!=) tells you if they do not match.

The comparison operators <, <=, > and >= refer to substrings.

```
   'ragg' < 'ragg mopp'
true
   'Ragg' < 'ragg mopp'
false
```

and is case-sensitive.

Finally, you may use regular expressions with the =~ operator to check if a string matches. In this example, we check if a string that consists of letters or numbers,

```
   '[a-zA-Z_]\\w*'=~'_florid3'
true
```

Other useful operators on strings

QDL has a really nice set of string operators.

`differ_at` takes two strings and returns the first index where they differ or -1 if they are equal:

```
   differ_at('abcde', 'abqw')
2
   differ_at('abcd','abcd')
-1
```

`head` returns everything up to a stop character:

```
   head('bob@bigstate.edu','@')
bob
```

Numbers

QDL supports numbers, meaning that it makes no difference between integers and decimals per se. It also support scientific notation. QDL normally works at 15 digits precision, but yo may set the precision to be arbitrary – the limit is your hardware. So to compute pi to a hundred places:

```
   numeric_digits(100);
```

```
   pi();
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706 8
```

There is a full set of mathematical functions that QDL supports, including trigonometric, hyperbolic, exponential, logarithmic and gcd, lcm, plus many others. Remember as a practical matter that convergence can be slow, so if you planning on running everything to 1,000 places, you might have to wait. That said, the default uses nicely optimized algorithms and should work acceptably fast.

Booleans

There are *two* and only two reserved words for booleans, **true** and **false**. So yes, they are case sensitive. The negation operator is !

```
   !true
false
```

null

The null is a special, reserved keyword indicating no definite value. It is equal to itself and that is that. It is very useful in programming QDL though.

Sets

QDL also allows for sets. A set is an immutable, unordered and every element is unique. These are normally written as {element0, element1,…}

```
    a := {4,1,-11,17}
{17,-11,1,4}
```

Note that the order is not specified. If you need order and the ability to access individual elements) consider using lists or stems. Sets are treated as elements in their own right (hence no trailing period which is an index operator). Operations on sets are

operator	name	example	Result
|^, ⊢	convert list to set	|^[;5]	{0,1,2,3,4}
∧ ,∩	intersection	{1,2,3} ∧ {2,4,6}	{2}
∨ , ∪	union	{1,2,3} ∨ {2,4,6}	{1,2,3,4,6}
/	difference	{1,2,3}/{2,4,6}	{1.3}
%, Δ	symmetric difference	{1,2,3}%{2,4,6}	{1,3,4,6}
==	equality	{1,2,3} == {3,1,2}	true
<, >, <=, >=	Set inclusion.	{1,2} < {1,2,3}	true
∈ , ∉	set membership	1 ∈ a, [1,5] ∈ a	true, [true, false]
~	convert set to list	~{1,2,3}	[1,2,3]

Notes

- `|^` converts a list or scalar to a set. ~ (monadic tilde) applied to a set turns it into a list. If the set is nested, the resulting list will be too. You can use the operator ⊢ (\u22a2) or its digraph `|>` for this interchangeably. Note that `⊢~A ≠ ~⊢A` in general since the order of sets is not guaranteed (and sets have unique elements only). Of course, you can just type in a the elements of the set between `{}`, *e.g.* `{2,4,6}`
- Union and intersection are either the unicode symbols, ∩ (\u2229) or ∪ (\u222a), or may be ascii digraphs made of \ and /, viz /\ is intersection and \/ is union.
- Symmetric difference is either done with the % sign or the operator Δ (\u2206).
- Order of operations is that union and intersection are at the same level, so will be interpreted from right to left in order when encountered. These are higher in precedence than symmetric difference. There is no actual universally agreed on order of operations. This is chosen because under symmetric difference and intersection, sets form a Boolean ring and symmetric difference is the analog to addition, union to multiplication.
- Note about the empty set. This is denoted by `{}`. Note that the empty *stem* is an empty list, denoted by []. You may also use the symbol ∅ (\u2205) for this.
- Sets are generally quite fast in their operation and have minimal structure. While you can have sets of sets, the onus is on you to make sense of them. Generally sets of scalars and sets of sets of these are never a problem. Sets of stems have the issue that comparing stems (which can be recursive) is at best dicey.
- Scalar operations on sets apply to each element (similar to stems).
- Operations on sets such as intersection, ordering, membership apply to whole sets.
- Selecting elements is done with the *subset* function.
- Comparisons are done with the standard <, > etc. be advised that these are strict, so `A < A` will always fail, but `A == A` or `A <= A` will work.
- `a∈b` can be fully replaced with the function `has_value(a,b),` however, `a∉b` must be replaced with `!has_value(a,b)`.

Example. Scalar operations on sets

Scalar operations on sets return a set with the operator applied to each element:

```
   3+{2,4,6}
{9,5,7}
     4<{1,2,3,4,5,6,7,8,9}
{true,false}
```

In the last case, the less than operator is applied to each element of the set and the result is added to the

answer. Since true and false are repeated, the final answer has at most two elements. Again, there is no order possible with a set. If you convert one to being a list (with the monadic tilde operator) then you should check on the order of the resulting list.

Stems

A ***stem*** is the term used in QDL for what is called a map, symbol table, dictionary or associative array in other languages. It is a data structure that of pairs (*key, value*) in which each key is unique. In QDL, keys are either integers or strings. The notation QDL uses is of the form

```
head.tail
```

(Geeky stuff, the period is the dot operator (some other languages might call it either the *reference* - or the *child-of operator*.) Remember that the variable is **head.** (note the trailing period!!) and the tail – which may be complicated -- is just indexing. The tail consists of keys separated by periods, but see the section below on tail resolution for the full story. Pretty much anything but a dot can be used as part of the name in the tail. This effectively means that a stem can be something as simple as a list or quite a complex data structure indeed. As a matter of fact, any data structure an be modeled with a stem. Some definitions:

- A ***list*** is a stem whose keys are non-negative integers
- Two stems are ***conformable*** if they have the same keys
- ***Tail resolution*** means that if a stem has many indices, like a.b.c.d, then it is resolved from right to left, with the system checking each index to see if that variable has been defined, then substituting. You may have stems embedded.
- ***Subsetting*** is in effect for most stem operations. This means that if the result is a stem, it contains only the keys common to its arguments. If two stems are conformable, no subsetting is needed.
- The ***dimension*** of a stem is the actual number of independent indices *The* **rank** of a stem is the number of dimensions. Scalars have rank 0, stems have 0 < rank. The **axis** of a stem refers to which dimension. The axis starts at 0 (as in, every stem has a 0 or first axis). The **size** of an axis is the number of elements in that axis.
- ***Wrap around*** for list indices is supported. This means that negative indices count from the end of a list. `x.(-1)` is the last element in the list, `x.(-2)` is the next to last, etc. Wraparound is not extended indefinitely, so `x.(-2)` has to exist.
- Unknown variables are replaced with the identical constant. So if `foo` is undefined, then `x.foo` and `x.'foo'` are identical. However, if you set `foo` to be a value, that will be used.

Lists

Since lists are a frequent occurrence, there is a specific notation for them,

```
[i0, i1, i2, … ]
```

Example

In `[1,2,3]`, the rank is 1 and there is one axis, zero and the size is 3

Lists may be nested to form higher rank lists:

```
x.:=[ // axis 0 are the rows
```

```
     [1,2], // axis 1 are the columns
     [3,4],
     [5,6],
   ]
```

The rank is 2. **`size(x.)`** is 3, **`size(x.0)`** is 2. And for names

```
x.:=[ // axis 0 are the rows
     [ // axis 1 are the columns
     // data
```

```
y.:=[ // axis 0 is box (1)
     [ // axis 1 are the columns
       [ // axis 2 are the rows
         // data
```

and you can nest boxes as you like. The function **`n(i,j,…)`** creates a nested list of dimensions i, j, ...

```
   y. := n(3,4,5,6); // a rank 4 stem with 360 elements in it
   dim(y.)
[3,4,5,6]
   rank(y.)
4
```

Note that this is one of the very few strict pattern enforced in QDL – a stem must end in a period *i.e*, so the period is an assertion that this refers to an aggregate. Issuing something like this (to make a list of integers)

```
a := [1,2,3];
```

fails with a message like "`Error: You cannot set a scalar variable to a stem value`". This is because the item on the right is an aggregate, aka a stem and on the one on the left is a simple scalar.

The general notation

For a general stem, the notation is

```
{k₀:v₀, k₁ :v₁, …}
```

Yes, the values may be stems. You may set the stem in this fashion directly

```
a. := {'time':'midnight', 'manner':'candlestick', 'place':'library'}
```

or individually

```
  a.'time' := 'midnight';
a.'manner' := 'candlestick';
 a.'place' := 'library';
```

or ***if*** the variables have not been defined, their name is used as the constant, so

```
  a.time := 'midnight';
a.manner := 'candlestick';
```

```
 a.place := 'library';
```

is equivalent. Generally it is a good idea to stick with constants for things that are constant and reserve variables for things that vary, but there are times where this is quite convenient.

The ~ (tilde) append operator

You may add elements to a stem individually using the ~ operator

```
   [0]~'foo'
[0, foo]
```

simply appends 'foo' to the list. This also works with general stems

```
   {'a':'b'}~{'p':'q'}
{a:b, p:q}
```

It is certainly possible to have mixed data, for instance

```
   my_stem.help := 'this is my stem'
   my_stem.~[;5]
[0,1,2,3,4]~{help:this is my stem}
```

which shows that this stem includes a list and has another entry called *help*.

Note: list indices are signed in QDL. This means that for index 0 <= *k*, the index is exactly the index. For *k* < 0, the index is relative and will start from the other end of the list, effectively being *length* + *k*.

```
    a. := n(5)
    j : -1;
    a.j
4
```

This shows the first entry from the right, 5 + (-1) = 4.

```
     remove(a.j)
    a.
[0,1,2,3]
```

This removed the last entry and the list now has 4 entries, not 5.

Removing values with !~

In a similar way, you may remove a value or values with the *excise operator*, **!~**.

```
   [2,4,5,6] !~ 5
[2,4,6]
```

The contract is that at the end of the operation, no values on the right-hand side remain. In the next example, we make a repeating 3x5 array for the numbers from 0 through 6

```
   a. := n(3,5,[;7])
[
  [0,1,2,3,4],
  [5,6,0,1,2],
  [3,4,5,6,0]
 ]
```

So we can remove, say, all the values of 1 and 4 from this

```
   a.!~[1,4]

[
  [0,2,3],
  [5,6,0,2],
  [3,5,6,0]
 ]
```

Note that !~ operates *in situ* so you can do surgery on parts of stems. Let's remove 1 and 4 from only the first elements. It returns the exact result, but the operation is on the stem:

```
   a.0 !~ [1,4]
[0,2,3]

   a.
[
  [0,2,3],
  [5,6,0,1,2],
  [3,4,5,6,0]
 ]
```

Reading the printed output

In the last example, how to read the result printed? The general form is

```
[list] ~ {map}
```

where the list is an ordered set (hence no need to write down the indices, since they are 0,1,2…) and the map has entries of the form

```
key : value
```

The tilde, ~, is a union operator and means these are a single entity. Note that this is a printed version for human readability. So here, the key is ***help*** and the value is ***this is my stem***. Note that if you create a list with sparse or missing entries, (so sparse data means only creating the entries you need, not some, vast empty array) then printing it will have the keys just written in map notation. Let us say you wanted to keep a listing of your favorite places in a stem by zip code. Your first entry might be

```
  zip.99950 := 'Ketchikan'
  zip.
{99950:Ketchikan}
```

If we were to use [] notation, how would we represent the missing 99950 elements?

Example: sparse matrix

You can define a sparse matrix using a default value and just setting what you need.

```
   a.:={*:0}
   a.3.14 := 11;
   a.2.7 := -3
   a.1.1; // check default value
0
   a.^3
```

```
{
 2:{7:-27},
 3:{14:1331}
}
```

So e.g., `a.2.7^3 == -27`.

Tip: Renumbering lists

A common idiom to re-order all elements in a list from **`0`** and this is monadic ~:

```
~list.
```

Would take the elements of **`list.`** and restart the indices from **`0`**. Since there is always subsetting involved in operations and QDL preserves indices, some operation (like a **`reduce`**) that leaves you with possible random indices may or may not need this.

Handling strange keys

Note that the index for a stem may be pretty much anything because you may pass around sets of indices, but all references (*e.g.* what you type in) must be either integers or variables or literals. The character set for variables is much smaller than for languages, so for tail resolution to work there are two options. The easiest is to use a variable to avoid ambiguity. Let's say you wanted to make a stem whose keys were your favorite functions, the first of which is `'f(x,y)'` (which is a string, of course). You would do something like

```
    p:='f(x,y)'
    q.p := 'cos(x)*sin(y)'
  q.
{
 f(x,y):cos(x)*sin(y)
}
```

but issuing `q.f(x,y)` is going to cause an error (since this is a function) vs.

```
q.'f(x,y)' := 'cos(x)*sin(y)'
```

Alternately, you may use encode/decode for type 0 which allows you to convert all unknown symbols into an escape sequence, which is a valid variable. If you really need to have something you can type in without variables consider using encode() to change a string to something that is a legitimate variable:

```
  encode('f(x,y)', 0)
f$28x$2Cy
```

Normally you only have to think about such things if you have to deal with exchanging information between external programs,e specially if unicode characters are not supported.

Applying functions to stems

It is easy and convenient to use this notation. The basic notion is that “freshmen Algebra” is right, *viz.*, that every operator is linear. So if we have a stem with entries a.i := b_i then f(a.i) ==f(b_i) For instance, a quick compendium of examples:

```
   1 + [2,3,4,5];  // add a number to a list
[3,4,5,6]
   [2,4] + [3,5]; // add two lists
[5,9]
    'a' + ['a','b','c']; // concatenate one string to a list of them
[aa, ab, ac]
```

You can also transform one stem to another. In this case we pick out all words that contain the letter n:

```
   pick((v)→'n' < v, // lambda function to check value
        ['a', 'man', 'a', 'plan', 'a', 'canal', 'Panama'])
{1:man, 3:plan, 5:canal, 6:Panama}
```

See function reference for **pick**.

Note especially that you can populate these lists with any valid QDL expression, so here is a nested array of random integers:

```
   mod([abs(random(2)), random()], 100)
[[5,20],39]
```

Writing a function

```
   f(a) → size(a)
   f(3); // scalars have size 0
0
   f([2,4,6,8]); // size of a list is just the elements
4
```

The point of this example is that the variable, a, is not qualified. You may pass in either a stem or scalar. Had we defined

```
   g(a.) -< size(a.)
   x. := [;5];
   g(x.); // list of 5 elements
5
   g(2)
illegal argument:error: a stem was expected at (1, 7)
```

If you write a function with the intention of passing in a stem or scalar, you may have to convert the argument to a stem by setting it to one.

The use of **apply** to stems as arguments.

In the previous example, wehave a function named g. To evaluate that with the apply function requires a note that this fails:

```
   apply(@g, x.)
```

```
 the function 'g' with 5 arguments was not found.
```

What happens is that the second argument is a list *of arguments*. Hence in this case, 5 arguments, each being an integer 0,1,..,4 are sent. To invoke it corrects, enclose x. in a list.

```
   apply(@g, [x.])
5
```

An example of a ragged stem.

And even more baroque (and to show that these are “ragged” arrays, unlike many other languages):

```
  mod([random(5), [random(4), [random(3), random(2)], random()]], 1000)
[[-629,-531,575,-911,222],[[867,-91,-891,-196],[[-167,468,920],[573,-162]],987]]
```

Slice operators

A *slice* is a specific notation that generates a list. There are two slice operators available.

Open slices

The open slice gives back elements from **`start`** to **`stop`** incremented by **`step`**.

```
[{start} ; stop {; step}]
```

Meaning that the result will be a list of elements that are constructed as

```
[start, start+step, start + 2*step, … ]
```

and will continue until

- **`stop < start + n*step`** if **`0 < step`**
- **`start + n*step  < stop`** if **`step < 0`**

Notes

1. this is inclusive of **`start`** and exclusive of **`stop`**.
2. **`0 == step`** will cause an error
3. omitting the first argument is the same as setting it to zero
4. omitting the last argument uses a default step of 1.

So in summary

```
[;5] == [0;5] == [0;5;1] == [;5;1]
```

E.g.s

```
  [0;5]
[0,1,2,3,4]
  [-2 ; 3 ; .75]
```

```
[-2,-1.25,-0.5,0.25,1,1.75,2.5]
   [5;0]
[5,4,3,2,1]
```

The major takeaway point is that you do not know how many elements there will be before evaluation:

```
  x. := [ -π() ; π() ; sinh(.8)];
  size(x.)
8
   x.
[-3.14159265358979,-2.25348667140217,-1.36538068921455,-.477274707026927,
0.410831275160696,1.29893725734832,2.18704323953594,3.07514922172356]
```

So a small increment yeilds a lot of values:

```
  size([ -pi() ; pi() ; sinh(.1) ])
63
```

So note that in the first case there were 8 elements required, while the second took 63. Also, while the zeroth element is guaranteed to be the first argument, the final one will be less than the second argument. So here adding **`sinh(.8)`** to the last element would be larger than π, so it is not returned.

it is also possible to omit the step, in which case it is assumed to be 1:

```
  [2 ; 11]
[2,3,4,5,6,7,8,9,10,10]
```

Closed slices

The ***closed slice*** gives back *n* evenly distributed elements over an interval, including both endpoints.

```
[| {start} ; stop {; count} |]
```

Note that the digraphs of [| and |] are made to look like double brackets in unicode ⟦ and ⟧. If the start value is omitted, it defaults to 0. If 0 < `count` is omitted, it defaults to 2 (returning the endpoints). Note that there must always be at least a stop argument.

```
  [| -1;2;6 |]
[-1,-0.4,0.2,0.8,1.4,2]
```

This gives 6 numbers distributed over the interval -1 to 2. Note that the first argument and second argument always are in the result. Just to emphasize how many elements you get back:

```
   size(⟦ -π() ; π() ; 9 ⟧)
9
```

As expected, 9 elements were requested and 9 were returned. A comparison is

```
  ⟦;5;5⟧
[0,1.25,2.5,3.75,5]
  [;5;5]
[0]
```

In the first case, 5 elements are requested. In the second case, a step of 5 is requested and that leaves a single element in the list.

```
   ⟦;5⟧
[0,5]
```

This is because the default start is 0, and the default count is 2, so single element closed slices are always 2 points.

Slice Math

Let's say you wanted to evaluate

```
sin([;10000]/1000)
```

This takes 3 traversals of the list. One to create it, one to divide everything by 1000 and one to evaluate it. This scales poorly, so be sure to use the facts that

```
 [a;b;c]×n^±1 == [a×n^±1;b×n^±1;c×n^±1]
[a;b;c] ± x == [a ± x;b ± x;c]

 ⟦a;b;c⟧×n^±1 == ⟦a×n^±1;b×n^±1;c⟧
⟦a;b;c⟧ ± x == ⟦a ± x;b ± x;c⟧
```

(*Ahem* remember that `b×n^±1` is another way to write `b×n` or `b÷n`. So rather than write the expression for the sine above, this is better.

```
sin([;10;1/1000])
```

Benchmarking note and rumination: On my system, the first expression takes 129 ms. (77.52 kHz) to evaluate the sine of 10,000 numbers and the second is 113 ms. (88.5 kHz) Which are both still excellent, but the point of this is a very complex expression might take a lot longer than you think. Does this mean there is an issue with QDL? No. QDL is properly a notation and the fact that you can write clumsy English or Swahili -- if not even outright gibberish -- does not mean those languages are invalid either.

The different slice operators exist so you can optimize creating different types of lists.

In general, function composition is your friend, so another option if you wanted to evaluate something complex is to create a function to operate on each element.. E.g. let's say you want to evaluate a polynomial at 1000 points over [-1,1]. You could write

```
   a. := [;2;1/5000]- 2
   b. := a.^3 + 4×a.^2 - a.÷7 +3
```

which takes 8 iterations through the loop, or define

```
   x. := [|-1;1;1000|]
   f(x)-> x^3 + 4×x^2 - x÷7 +3
   for_each(@f, x.) =: b.
```

Default values for stems

You may set a default value for stems – this is a very nice thing indeed so you don't have to initialize every element in one before using it. The way it works is either you set the special value:

```
   a.* := 2; // set as a special value
   a.star() := 2; // set with the special star-function
   a.:= {*:2}; // set the element directly
  set_default(a., 2); // use the command
```

All of these are equivalent. Viewing the value of a. we get

```
   a.
{*:2}~[]
```

the way to read this is that there is a default value of 2 and no other elements. So if you access a non-existent element you get the default:

```
   a.37
2
```

Note that the key here is a ***** (not the character `'*'`) If the stem does not exist, it will be created. Note that subsequent operations on the stem do ***not*** alter the default value.

Example

```
   set_default(stem., 2)
   stem.woof
2
   stem.'woof' := 4;
   stem.'woof'
4
   2 == stem.'arf'
true
```

The advantage of using the ***** entry is that it may be treated exactly like any other stem entry. If there is a default entry it will be listed when you print the stem.

```
   a.:={'p':'q', 'r':'s'}
   set_default(a., 't')
   a.
{*:t, p:q,r:s}
   a.0 == 't' && a.p == q
true
```

Subsetting and why you need to use default values at times.

Remember that there is always subsetting on for QDL so it does not simply generate data. The way you supply defaults (for potentially very complex stems) is with a default value. For instance, if we have a matrix A. and want to add

$$A = \begin{pmatrix} 9 & 0 & -8 \\ -6 & 1 & -4 \\ 6 & 7 & 9 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

You could do this in QDL as

```
   A. := [[9,0,-8],[-6,1,-4],[6,7,9]];
   B.0.0 := 1; B.1.1 := 2; B.2.2 := 3; B.* := 0;
   A. + B.
[
 [10,0,-8],
 [-6,3,-4],
 [6,7,12]
]
```

Tips

- If you have very complex index expressions that access default values, do use parentheses to make clear which is which. The right to left rule for index resolution cannot resolve * necessarily. So `a.*.2.*.3.5` gives an error whereas something like `((a.*).2.*).3.5` for instance works fine.
- Setting the default value for a stem also sets the same default value for every embedded stem.

```
   a.1 := [;5];
   a.* := 11;
   a.1
{*:11}~[0,1,2,3,4]
```

which shows that `a.1` now has the same default value as the stem.

Applying scalars to stems

A scalar is a simple value. All of the basic operations in QDL work on stems as aggregates. (So called “freshman algebra.”) So for instance, if you needed to make a list counting by 3's, you could issue

```
    3*n(5);
[0,3,6,9,12]
```

So lets say we have the following:

```
    ring.find := 'One Ring to find them';
    ring.rule := 'One Ring to rule them all';
    ring.bring := 'One Ring to bring them all';
    ring.bind := 'and in the darkness bind them';
```

Let's find every element that contains the word 'One”, respecting case:

```
    contains(ring., 'One')
{bind:false, find:true, bring:true, rule:true}
```

Or for that matter, doesn't contain this word:

```
    !contains(ring., 'One')
{bind:true, find:false, bring:false, rule:false}
```

The output of these functions is a boolean-valued stem and there is a very useful function called *mask* which simply will return the elements that have corresponding true values.

```
    mask(ring., contains(ring., 'One'))
{find:One Ring to find them,
 bring:One Ring to bring them all,
 rule:One Ring to rule them all}
```

And of course you could just find the ones that don't have the word “One” in them:

```
   mask(ring., !contains(ring., 'One'))
{bind:and in the darkness bind them}
```

This points out that using stems can do a tremendous amount of work for you. Since QDL is interpreted (each line is read, then parsed and executed) having as much happen as possible with a command improves both performance and efficiency. Besides, working with aggregates is often much ore intuitive than slogging through each element.

Tail substitutions.

If you have defined a variable, say

```
k := 3;
my_var. := n(5);
```

and issue the following

```
my_var.k := 'foo';
```

Then this will result in `my_var.3` being equal to the string `'foo'`. In short if the tail has a value at that point, this is used first. If not, then the tail itself as a string is used. This lets you do things like

```
i := 0;
while[
   i < 5
 ]do[
  say('the value = ' + my_var.i);
  i++;
];
```

which prints out

```
the value = 0
the value = 1
the value = 2
the value = foo
the value = 4
```

Moreover, substitutions happen from right to left (!! backwards from reading order), so if you set

```
  x := 0;
y.0 := 1;
z.1 := 2;
w.2 := 3;
```

Then you could reference a value like

```
w.z.y.x
```

which would resolve to

```
w.2 == w.z.y.x == 3
```

This permits more readable values, e.g. **`time.manner.place`** is a perfectly fine reference. ***HOWEVER*** the stem is always the leftmost symbol. The rest are just a very compact way to index it.

So why do this? Because it allows for something very powerful: implicit looping. You can have stems do a tremendous amount of work without ever really having to access the elements.

A Small Example.

```
  a. := abs(mod(random(1000000),1000000));
  a.42
769942
```

That's 1 million random numbers in the range of 0 to 1000000 and we showed the 42nd value just because. Here's a polynomial:

```
  a. := a.^2 + 3*a. -4;
  a.42
592812993186
```

and if you insist, here is a check

```
  769942^2 + 3*769942 - 4
592812993186
```

You may also have indices with embedded periods, so in the above example

```
    foo:= 'z.y.x';
    w.foo;
3
```

You can then take a list of indices and simply iterate over them or whatever you need to do. This again is why keys for stems generally do not allow for embedded periods.

Although you can do something like this:

```
  a := 2.3
  q.a := 4
  q.2.3 := 5
  q.
{2.3:4,2:{3:5}}
  q.a
```

```
4
  q.2.3
5
```

where there is a decimal index, it can get confusing.

More about that trailing period.

To refer to a stem as an aggregate (everything) you must include the period. This is a perfectly fine example

```
  a.0 := 'foo';
  a.1 := 'bar';
  a.2 := 'baz';

  a := 2; // so this is a scalar - no period at the end
  say(a.a);
baz
```

Here the scalar a has value of 2 which is substituted as the tail of the stem, so the answer printed is the value of a.2. This just points out that a and a. are considered to be wholly unrelated.

Note that this is *exactly* like most other programming languages (e.g. C, C++. Java). For instance this a perfectly fine program in C:

```
#include <stdio.h>
int main()
{
    float  a[3];
    float  a;
    // bunch of stuff

    a[0] = a;

    return 0;
}
```

in which an array and a variable may have the same name and are differentiated by indexing, *e.g.*, *a* vs. *a[]*. QDL simply has a very flexible approach to indices. Another way to think of stems is as “ragged arrays”, where entries may be of differing lengths per index.

You may nest stems as well, so

```
  a. := 3 * n(5); // count by 3's, so 0,3,6, … ,12
  b. := 10 * n(5);// count by 10's so 0,10, … ,40

a.b. := b.;
```

works just fine. Note that unless *b* has been assigned a value, the index of it in *a.* is *b.* You can access the value as

```
c. := a.b.; // note the trailing . on the variable c to show it is stem
```

but you cannot issue `a.b.3` and expect to get anything back (`b.3 == 30` which is not an index in `a.`) unless you have set it explicitly. This is because, again, it is included in the stem variable `a.` as an entry and you must refer to it by its proper index. TL;DR: Tail resolution happens for scalars.

Cavet on name collisions

Since tail resolution is always in effect, *do* take care with your variable names. For instance, if you decide everything in your **`X`** workspace is named **`x, x.,`** etc. then you may unwittingly re-use the same name, so trying to set your stem to have a key of **`'x'`** by setting **`x.x.0 := ...`** to something is going to give you an index error, since the middle **`x.`** introduces a recursive structure (which you can do in QDL fine). There are a couple of ways around this.

1. Use literals: **`x.'x'.0 := …`**

2. Use another variable: **`my_x := 'x'; x.my_x.0 :=…`**

3. Use better names. Is it really descriptive to have everything named **`x`** or some variant of that? This is a good point since you do want things to be self-documenting so when you pass off the workspace to someone else or come back to it after a hiatus you don't just have a mass of variables and functions that have no clear purpose or meaning.

Generally if you are having such name collisions that is a symptom that things are not named clearly or are not structured clearly. Stems are an extremely powerful paradigm, essentially turning aggregates into miniature databases (including doing queries on them with the **`query()`** command) and should be viewed in that light.

Also remember that parentheses can be (and should at times) be used to direct the order of tail resolution: **`x.(x).(x).(x)`** would tell QDL to use the value of **`x`** for tail resolutions, not **`x.`** as is the contract (when resolving tails, look for stems first, and only look for scalars if there is no

The ~ and union operator

There is a specific operator in QDL for stems (sets have their own operators), the ***tilde*** or ~ operator. This concatenates stems. There is a function version of it too called *union* (see below for more documentation). What does it do? It sticks together two stems.

```
  [1,2]~[3,4]
[1,2,3,4]
```

In this case, the two lists `[1,2]` and `[3,4]` are assembled into a single list `[1,2,3,4]`. It will also turn scalars into a list:

```
  1~'a'~true
[1,a,true]
```

This works with stems generally:

```
  a.b := 'foo';a.c:='bar';
  b.q:='baz';b.c := 'quxx';
```

```
  a.~b.
{
 q:baz,
 b:foo,
 c:quxx
}
```

Caveat in the case of lists (integer indices), the list is extended. In the case of stems with non-integer values, they cannot be extrapolated, so if the same key is encountered, the value is overwritten.

Also, the result from this operation will be a regular list, so indices will be adjusted.

```
  q.17 := 3
  q. ~[1,2]
{
 17:3,
 18:1,
 19:2
}
  union(q., [1,2])
[3,1,2]
```

One difference between this and the *union* function is that the union function reorders everything.

named stem.)

Other stem expressions

1. You can embed expressions in the indices, but have to be very clear about how it should be grouped. This means parentheses are your friend.

```
   k:=1;
   [i(5),i(4)].k
```

would return

```
[0,1,2,3]
```

You must, however, be careful. Since the . the child-of operator, so things between dots are arguments to this operator. (Note well that “child of” works in English if you read stems from right to left.)

The full contract for stem resolution assumes that all elements are resolved to stems first. So in

```
a.b.c.d
```

b and **c** will be tried as stems and if there are no such stems, then their scalar values will be used. If there are no such values, the value of the argument is simply returned. Let us say that you had both variables **b.** and **b** in the workspace and you really wanted to force that the scalar be used. Simply put it in parentheses (so it is evaluated first):

```
a.(b).c.d
```

You can also set expressions to a value. So you can do something like this

```
    a. := [-n(5), n(6)^2];
    f()-> a.;
    f().i(1).i(2) := 100;
    f();
[[0,-1,-2,-3,-4],[0,1,100,4,9,25]]
```

(Of course, **i(x)** is the identity function which just hands back **x.** Of course, this is slightly contrived to have **f()** return a stem, but the point is that as long as the expression on the left-hand side of the assignment evaluates to something reasonable, you may set it. One last caveat is an expression like

```
    to_uri('a:/b').path := 'foo'
```

is certainly legal and works, but read what it did: It parsed a uri and in that result set a single value to **foo**. If the intent was to replace the **path** with a new value, this won't work either

```
  a. :=   to_uri('a:/b').path := 'foo'
```

because the right hand side is a scalar. generally, variables exist to stash things we want later, so the right way to do it is

```
   a. :=   to_uri('a:/b');
   a.path := 'foo';
```

Now **a.** has in it what you want.

2. Recursion works but don't try to print.

You may certainly make a stem refer to itself:

```
    a. := [;5];
    a.b. := a.;
    say(a.b.2);
2
```

You can access elements directly as you wish though avoid tail resolution unless you – like any other recursive structure – have a well thought out plan to access things. In the above example, `a.b.2` is defined so there is no tail resolution needed. ***But*** do not try to print it because if you do you will get

```
    say(a.);
Error: recursive overflow at index 'b.'
```

Other ways to access stem elements.

Stems as indices: Index lists

You may use stems as indices too. An example should suffice

```
  a. := {'p':'x', 'q':'y', 'r':5, 's':[2,4,6], 't':{'m':true,'n':345.345}};
  a.s.0 ==   a.['s',0]
true
```

If a list is used as an index on a stem, then it is referred to as an ***index*** list or a ***multi-index***. In other words **`a.x.y. … z == a.[x,y, … ,z].`** Why? Because you can then create index lists dynamically. Using the . notation is great if you know the structure of the stem already, but if, for instance you have done a **`query()`** command to an unknown stem and now have a collection of multi-indices, you can access the elements. A great utility is `m_indices` in the extensions, if you have loaded them.

Caveat. In normal stem resolution, every stem is resolved to indices on its right. Let's say you wanted to access

```
a.4.3.2.1
```

using a list index. If you enter

```
a.[4,3,2].1
```

this resolves as

```
a.3
```

since

```
[4,3,2].1 == 3
```

Other ways to write this are

```
(a.[4,3,2]).1 == a.([4,3,2]~1)
```

Various functions either take multi-indices as their arguments (such as `remap`) or produce them as their results (such as `indices`).

The extraction operator

A common scenario is to have a large and very complex stem (for instance, writing a web application and getting some monstrous JSON object that you have turned in to a stem). The ***extraction operators***, **`\, \!, \>, \!>`** allow you to specify which elements to take from a given dimension to form another stem. You may also select (filter, as it is called in some other languages) elements using functions, following the syntax and usage of the **`pick`** function.

Operator	Effect
\	Integers keys will be reordered. String keys are never altered
\!	Integer keys will be preserved exactly.
>	Take stem arguments as stem indices
*	Take all of the indices

Quick examples.

- a\[2,3] means extract a.2, a.3. result will be a list [a.2, a.3]
- a\>[2,3] means extract a.2.3
- a\![2,3] means that the result is a stem with indices 2 and 3.
- a* means to take everything – the same as writing a.
- a\[2,3]*\[1,4] returns a rank 3 array list a.2.k.1, a.3.k.1, a.2.k.4, a.3.k.4 as k ranges over every index for axis 2.
- a\[2,3]\5\[1,4] return a rank 2 result (the scalar for the axis means to suppress it) so the elements are
  [a.2.5.1, a.2.5.4, a.3.5.1, a.3.5.4]
- a\((k,v)→0<v<4)\[1,5] filters the first axis for elements whose values are between 0 and 4 and then restricts the second axis to 1 and 5. See the **`pick`** function help for the semantics of filter functions.

More concrete examples are below.

Motivational Example

Let us say that you got that JSON object and that it's structure looks like this:

`web.content.server.clients.i.transactions.j`

So that in reality, *i* and *j* are the really interesting bits. In particular. there are many transactions but they all contain an identifier like uuid. Rather than slogging through the whole thing repeatedly, QDL lets you write this.

```
   a. := web\'content'\'server'\'clients'\*\'transactions'\0\'uuid'
   a.
[bc5c7110-f932-11ec-b939-0242ac120002
bc5c72fa-f932-11ec-b939-0242ac120002
bc5c7624-f932-11ec-b939-0242ac120002
bc5c7750-f932-11ec-b939-0242ac120002
bc5c785e-f932-11ec-b939-0242ac120002]
```

The result is a simple list of the uuids, one per client. The constants are omitted from the final result. * means to take everything in that dimension (here clients has 5 elements). The 0 means to take only the zero-th transaction.

You may use \ or \! the difference is that ! indicated preserving the indices as they are, otherwise for integers, they are automatically renumbered. This allows you to extract a substem with the indices intact if that is needed.

```
   b. := n(4,5,[;20])
   b\![1,3]\![2, 0]
{1:{0:5, 2:7}, 3:{0:15, 2:17}}
```

Note that the the result has the same indices as a., it is just a substem. So `a.1.2 == b.1.2`, etc.

```
   b\[1,3]\[2, 0]
[[7,5],[17,15]]
```

Do note one other thing: The order of the last dimension was swapped, which in this case means that the substem has re-ordered the elements. In the strict case no re-ordering can occur since the indices are not altered.

```
  b\[1]\[1,3]
[[6,8]]
```

Since the value of 1 is a list, the result has that dimension, so it is a 1x2 stem.

Creating extractions on the fly with \>, \>! and star()

You may also create a list of indices and tell QDL to interpret each element separately using \>. The > in effect tells QDL to distribute the \ to each element. So our above example could be written

```
a. := web\>['content','server','clients',star(),'transactions',0,'uuid']
```

Note that * may be replaced in extraction with the function star(). You may also use it in expressions like

```
   a\*
   a\star()
```

Notes

- **`\!*`** is legal, but has no effect since all indices are taken
- **`\!`*`scalar`*** has no effect since scalars are never returned in the final result
- **`\![`*`scalar`*`]`** is trivial and equivalent to **`\[`*`scalar`*`]`** since there is only one element.
- **`\!>, \>`** applies to every member of the list. Note that elements of lists must all be defined, since lists are evaluated first, then handed over to the operator.
- substems are extracted from the original, meaning the values are copied, so changing values in the substem does not effect the original stem.
- Sets may also be used as arguments. However, since there is no guarantee of order set elements, using strict mode may still not be strict. If you want order, impose it.

- Just like the reference operator . (dot) you may use variable names which will helpfully be replace the name as its value ***if*** that variable has not been defined. So **`a\p\* == a\'p'\*`** if and only if `p` has not been defined, otherwise the value of the variable `p` will be used. As with stems generally, it is certainly convenient, but one should stick with constants where there are constant.
- You may also use the full reference to a stem with or without the dot, `a.\p` and `a\p` are the same
- Missing indices are fine – they just won't have values associated with them. So if you enter
  `b\[1,37]\*`
  and there is no index 37, there will not be an error. This allows you to work with very sparse/ragged stems without having to check for conformability. Ask for everything, see what is actually there.
- Responses are stems, scalars (if all the indices are scalars) or null (if no elements of any sort were found for scalars, empty stem if some of the indices are lists).

The environment and the lifecycle of variables.

Every script has both global and local variables associated with it. Variables defined in a block (such as in the body of a loop or in the body of a conditional statement) are local to that block. Variables defined outside that block are global to the program. Modules are completely self-contained.

So what if you need to have variable defined and accessible outside a block but need to set the value inside one, like

```
  if [ /* nasty conditions */]
then [a := 'foo';]
else [a := 'bar';];
 // trying to use a will result in errors
```

What you can do is set the value to `null` outside the block. So do this.

```
a := null;
// same code as above
```

Now attempts to use the variable will work properly. You may also check if the value has been set by checking if it is null:

```
  if [a != null]
then [ /* lots of stuff */ ];
```

Visibility during function evaluation

Generally functions inherit the values of their parent state, but they do not inherit imported symbols. Following Leibnitz"lead, a function is to relate *cause* (the inputs) to *effect* (the output). Therefore, functions should have their values passed to them and not draw them in willy-nilly *e.g.* as global

values. You may, of course, just import modules in the body of the function or pass the values in as arguments. Note that the arguments to the function are executed in the function state, so

```
    f(x)->a*x^2
  f(a:=3)
27
  is_defined(a)
false
```

The value of **a** is passed in and set inside the function where it is used. It is not set in outside the function. This allows further fine control over the state

Control structures

There are 6 basic control structures. All of them are delimited with brackets [].

1. The basic conditional of
   if[*condition* **]then[** . . . **]else[** . . . **];**
   Note that the else clause is optional.
2. Switch statements (which are lists of conditionals) of
   switch[... **]**
3. **try[** . . .**]catch[** . . .**];** for error handling
4. Looping construct **while[** *condition* **]do[** . . . **];**
5. Assertion construct **assert[** ***boolean][expression];*** or its alternate
   ⊨ **boolean : expression;**
6. Block: **block[** ... **]**

The if..then..else statement

The basic format is

```
  if [condition]
then [/*statememts*/]
else [/* more statements*/];
```

Note that **then** is optional, but **else** is not. And of course, a simple example is in order:

```
j :=5;
   if [j <5 || 5 < j]
 then [
      say(j + ' is not 5');
      ]
 else [
      say('j is ' + j);
      ];
```

```
j is 5
```

Note that you can format these any which way you want. I just think it is more readable this way.

The conditional expression

Statements imply there is state managed. Expressions run in the current scope. There is an analog for the if … then … else statement which is the *ternary operator*:

```
 boolean ? A {: B}
```

in which if *boolean* is true, A is executed and if false B is executed. Note that rather than ? You may use the implies arrow ⇒ (\u21d2). B may be omitted in which case the default of **`null`** is returned.

Example

```
   a := b < 0 ? abs(b) : b;
```

in which a is assigned a value based on b. Note that doing this with an if statement requires managing some state, since in the next example, **`a`** must exist outside of the statement scope to be assigned:

```
   a:= 0;
  if [b < 0]
then [a := abs(b);]
else [a := b;];
```

A very common QDL idiom is to test existence before assigning a value

```
   a  := ∃x ⇒ f(x/2) : 0
```

Here if x exists, then a is assigned a value using a function, otherwise it is assigned a default value. An example without using the optional B is

```
   1 < size(args())  ? init(args().1);
```

So if the arguments to a script include multiple arguments, run an initialization function on it. In this case we really don't care if there is output or not.

The switch statement

Branched decision-making is a basic construction for most languages. In the case of QDL, there is the **`switch[];`** construct. The basic format is

```
switch[
     if[condition1]then[body1];
     if[condition2]then[body2];
     if[condition3]then[body3];
     //… arbitrarily many
     if[true][body]; // default case
   ];
```

The execution is each condition is checked and as soon as one returns *true* that body is executed and the construct returns.

Examples

An example of a switch statement might be

```
    i := 11;
    switch[
       if[i<5][var.foo := 'bar';];
       if[5=i][var.foo := 'fnord';];
       if[5<i][var.foo := 'blarf';];
    ];
    say(var.foo);
blarf
```

(Optional then keyword omitted.) Note that the elements of the switch statement are **`if..then`** blocks (including final semicolon). No else clauses will be accepted. Whitespace aside of strings, of course, is ignored.

Example: Setting a default case

Many languages with a switch construct have a “default” clause which should be done if no other cases apply. QDL does not since it is really easy to set one up otherwise. Since the conditionals in the switch block are executed in order, if you want the cases to fall through to a default, simply have the last one test for **`true`**:

```
    i := 11;
    switch[
       if[5<i<8][var.foo := 'bar';];
       if  [5=i][var.foo := 'fnord';];
       if[2<i<5][var.foo := 'blarf';];
       if [true][var.foo := 'woof';];
    ];
    say(var.foo);
woof
```

Since none of the other cases apply, it falls through to the last.

The switch expression

Again, statements imply state, so there is a switch expression

```
   switch. ?! case. {:default}
```

in which **`switch.`** Is a stem of booleans and **`case.`** Is a stem of selections. An optional default case is allowed. You may use the character **`¿`** (\u00bf) instead of the digraph **`?!`**.

The contract is that *at most* one of the elements of **`switch.`** is true and the corresponding element of case. Will be returned. If all elements are false, the default is returned. If the default is omitted, then the value of **`null`** is used.

Example

On a list

```
   [false,true]¿[a^2,3+3]:0
6
```

Note that the default case is not even evaluated

```
  {'p':false,1:true}?!{1:3+3,'z':a*3}:5
6
```

In the next case, we select one of the values of a list of the power of pi:

```
   mod([1;6],3)==0 ?! pi([1;6])
31.0062766802997
```

`?!` will try to avoid evaluating expressions if possible and can do so if **`case.`** is an explicit stem. However, the following will fail

```
  [false,false,true] ?! f(x, a)
```

assuming that **`a`** is not defined and even if **`f`** returns a stem. The reason is that as per the contract for functions, **`a`** must be evaluated and passed to **`f`**.

Error handling

There is a **`try … catch`** block construction. Its format is

```
try[
  // statements;
 ]catch[
  // statements;
];
```

You enclose statements in a try block and call **`raise_error`** if there is an exceptional case. Note that unlike many languages, all exception are fail-fast, so there is no way to hop back at the point of the error and resume processing. An example

```
j := 42;
try[
     remainder := mod(j, 5);
     if[remainder == 0][say('A remainder of 0 is fine.');];
     if[remainder == 4][say('A remainder of 4 is fine.');];
     if[remainder == 1][raise_error(j + ' not divisible by 5, R==1', 1);];
     if[remainder == 2][raise_error(j + ' not divisible by 5, R==2', 2);];
     if[remainder == 3][raise_error(j + ' not divisible by 5, R==3', 3);];
 ]catch[
    if[error_code == 1][say(error_message);];
    if[error_code == 2][say(error_message);];
    if[error_code == 3][say(error_message);];
 ]; // end catch block
```

```
42 not divisible by 5, R==2
```

So the way these work is if a certain condition is met, you call the **raise_error** function with a message and optional numeric value. These are available in the catch block so you can figure out which error was raised and deal with it. Not much else to them.

There are two reserved error codes. For an assertion (see the entry for the assert keyword) the code is -2. For every other system error the is value **-1**. This last value is issued by the system if there is an internal error during processing. The message is intended to be helpful at this point, but may also not be (since it is being propagated from some other component). You may either use an assertion to raise an error with code -2 or manually do it with **raise_error**.

Example. System errors.

In this example, we create an error in the course of normal processing and catch it:

```
  try[3/0;]catch[if[error_code==-1]then[say(error_message);];];
/ by zero
```

Note that this will not catch parsing errors, so if you did something like

```
  try[2+;]catch[say(error_message);];
syntax error:could not parse input
```

(the body of the try statement has a syntax error in it) it would not get caught because the parser would intercept it first.

One possible usage is to assert a stem with a specific structure rather than just a message to the user. The vastly more common case is to just assert a message.

Another example. User input

You can also use this for checking user input

```
   my_number := -1;
  try[my_number := to_number(scan('enter value>'));]catch[say('That is no
number!');];
enter value>foo
That is no number!
```

In this case if the user enters something unparseable as a number, a message is printed, but it would be easy to simply re-ask the user. (*Caveat:* The example as is does not work reliably in ANSI mode depending on the underlying system implementation. Break up the scan and to_number check into separate lines.)

Example: Assertions

Assertions are treated like exceptions since they may be thrown anywhere. These simply have a reserved code of -2:

```
 try[assert[3==4]['foo'];]catch[say(error_message);]
foo
```

If you needed to handle assertions in a script you might want to do something like this:

```
session_id := 0;
try[
  session_id :=  script_load('logon.qdl', username, password);
]catch[
   if[
     error_code == -2
   ][
     say(error_message + ': logging in as guest');
     session_id :=  script_load('logon.qdl', 'guest' , '');
    ];
];
  // bunch of other stuff
```

In this case a login is attempted and if that fails, a default is used. For assertions, the **`error_message`** is simply the message clause of the assert statement.

Looping.

The basic structure of a loop is

```
while[
   logical condition
  ]do[ or ][
    statements
];
```

Note that the **`do`** keyword is optional. For example, to print out the numbers from 0 to 5:

```
i := 0;
while[i< 5][say(i++);];

0
1
2
3
4
```

This is known as 'yer basic loop'. You may also loop over sets:

```
   a:=6
   while[j∈{1,2,3,4,5}][a:=a*j;]
   a
720
```

Note that set membership works the same with the has_value functional

```
   a:=6
   while[has_value(j,{1,2,3,4,5})][a:=a*j;]
   a
720
```

Also note that it is not possible to loop over $\notin$ since that makes no sense.

Style issues

This is an example of bad QDL:

```
y. := null
while[for_next(j, 100)][y.j := sin(j/100);]
```

This just fills up a variable, y., with values.

Good QDL:

```
y. := sin([;1;1/100]);
```

QDL does array processing just fine. Loops are usually not needed for most operations and they have a fair bit of overhead so do use them with discretion. When you write functions, opt for list processing as well.

Scope

Definitions

Scope refers to the specific context where code resides/executes. There is *lexical scope* which is the part of the source code where the item exists and there is *dynamic scope* which where the item exists as the code executes.

Example

```
f(x)→x+1;
```

The lexical scope is that x exists in the definition of the function (on the lefty hand side of the arrow) and in the single statement on the right hand side. Since this function is now available in the workspace, its dynamic scope is everywere.

Overriding scope.

QDL allows for scopes to be overridden. Consider the block command

```
local[…]
```

this means that statements inside the brackets have nothing to do with the external environment, so

```
   a := 3;
   local[say(a);];
unknown symbol 'a' At (2, 10)
```

Why do this? Because you might not want the symbol table littered with variables and functions that are just needed for a specific task. Note the following behavior especially:

```
   local[a:=11;z:=4;say(a+z);];
15
 a
3
```

i.e., The *a* inside the local block has nothing to do with the *a* outside the block

On the other hand

```
block[…];
```

will inherit the current (or *ambient*) scope but new definitions inside the block are local to it.

```
   a:=3;
   block[z:=4;say(a+z);];
7
   say(z);
unknown symbol 'z' At (1, 4)
   a
4
```

So *z* exists solely in the block, but since *a* existed before it is re-assigned.

In the sequel, functions and modules have their own scopes (discussed in detail). In telegraphic form (hey, this is a reference manual)

define[] uses local[]

→ uses block[]

module[] uses local[]

Defining functions

Functions in QDL

QDL is mostly what is termed a *functional programming language* which means that mostly you define functions and carry out tasks invoking or composing them. You may pass them as arguments to other functions, for instance. Since QDL allows *in situ* definitions of functions, they must be defined in the code before they are used, unlike some languages that let you put them any place. This gives enormous flexibility in managing them.

Example. Comparing QDL to another language.

Here we create an array of *n* numbers, double each of them, then add up the contents of the array and store it in a variable named *sum* would look like this is Java:

```
n = 10; // for instance
int[] array = new int[10];
int sum = 0;
for(int i = 0; i < n; i++){
   array[i] = 2 * (1+i);
   sum = sum + array[i];
}
```

In QDL:

```
   n :=10; array.:=null;  // define array. here so it's not local to the function.
   sum := reduce(@+, array.:=2*(1+[;n]));
```

Functions are very easy to create (especially using the **`() ->`** syntax). There are, unlike many pure functional language, constructs for loops and conditionals, but QDL uses list processing wherever possible, so by and large, you mostly need to describe what you want to happen to your data (such as above, where you **`reduce`** it in one fell swoop). It is therefore best to think of QDL as a notation for describing algorithms that happens to have some control structures. Why? Because frankly some problems are very easy to describe in the functional paradigm and some are not. QDL is designed so that if you need to make a clearly defined structure it is possible. Such things in a purely functional language can be very awkward indeed to express.

Defining a function with the full formal syntax

The formal or full syntax for defining a function is very simple:

```
define[
    signature
  ]body[ or ][
    === useful comments
    statements
  ];
```

Note that within the body, any variables defined are local to that block unless they are save, *e.g.* in the environment. The state is new so that only variables passed in are available. The variables in the signature are populated with the values (which are copied) from the ambient. To return a value, invoke the method return – which is only allowed inside function definitions, by the way. No return statement implies there is no output from the function.

The ***signature*** of a function is

```
name(arg0, arg1, arg2,…)
```

this means that the _function is defined by its name. Of course, if you define it in a module, you may have to qualify it.

An example

```
define[
    sum(a, b)
   ]body[
    » add a pair of integers and return the sum.
    return(a+b); // this terminates execution and returns the value.
   ];

say('the sum of 3 and 4 is ' + sum(3,4));
```

This prints

```
the sum of 3 and 4 is 7
```

You may use functions any place in your code once they have been defined, so it is a good practice to put them at the beginning.

Also, note that this example works on both scalars and stems: The signature does not determine the type, so

```
  sum([1,2,'abc'],[5,7,'dgoldfish'])
[6,9,abcdgoldfish]
```

is just fine.

Example

Note: If your signature contains stems, then passing it a non-stem will be caught and flagged as an error. You do this if you require a stem in a certain position. If you do not specify, then you can pass anything as an argument,

```
   define[glom(p.)][return(p.~[;5]);]
   glom(2)
illegal argument:error: a stem was expected

   define[glom2(p)][return(p~[;5]);]
   glom2(2)
[2,0,1,2,3,4]
   glom2([11,12])
[11,12,0,1,2,3,4]
```

You may access the variable as **p** (not **p.**) in the function, but indexing still works the same, so **p.0.1** would be fine.

Help

The lines that start with » (unicode \u00bb) or `===` (triple equals sign) at the beginning of the body are read by the system as help and can be accessed using `)help function_name arg_count`. So for the example above

```
   )help sum 2
add a pair of integers and return the sum.
```

This allows you to document your function and generate online help in one step. There is a much more complete section below, but this is important enough to mention twice.

Overloading

Overloading a function refers to having various forms of a function with different arguments. For instance

```
     define[sum(a,b)][…
     define[sum(a,b,c)][...
```

both define the same named function with different arguments. You just call the one you want and the interpreter figures out the one you want. Some languages do not allow overloading (Python, e.g.) where the contract is to write a function with a possibly huge argument list of everything you may need and then dispatch it internally by case. Some (such as C++, Java) are strongly typed, so these would be fine in C++

```
float add(float a, float a) { }
float add(double a, double a) { }
int   add(int a, double b) { }
int   add(int a, int b) { }
// . . . tons more of these!
```

The problem with that is it can lead to an explosion of functions. Since QDL is mostly untyped (really the only difference is scalar vs. aggregate) QDL can only differentiate functions based on the number of arguments, but it is up to you to sort them out after that. In summary, QDL allows **partial** overloading of functions.

Overloading System Functions

You *may* overload system functions with arguments that are not in use. For instance if you wanted a version of the size function that worked with two argument you could write

```
  size(x,y)->size(x)*size(y)
  size([;5],[;7])
35
```

Similarly you can override with arguments that are normally not allowed. Note that attempting to override base functions will normally throw an error unless you explicitly set the workspace variable

(also available as a configuration option) `overwrite_base_functions` is set on. To override a built in function requires then that you reference it with its module:

```
   size(x)-1+stem#size(x)
   size([;5])
6
```

However while this lets you completely rewrite QDL from the ground up, it will also adversely impact speed since built in functions have very fast access and the system must search each and every call to a function in the user defined space first. This is normally about a 20% speed decrease.

Examples

Here is a QDL program to find the *Armstrong numbers* in the range of 100 – 1000. A number, *xyz* is an Armstrong number iff $xyz = x^3 + y^3 + z^3$. This is written to contrast a procedural style (possible in QDL) with a more native example that follows.

```
define[
  armstrong(m)
  ][
     === Armstrong number: 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
     if[ m < 100]
   then[say('sorry, m must be 100 or larger'); return();];

     if[1000 < m]
   then[say('sorry, m must be less than 1000'); return();];

     sum := 0;
     while[for_next(j, m)]
        do[
           n := j;
           while[0 < n]
              do[
                  b := mod(n, 10);
                sum := sum + b^3;
                  n := n%10; // integer division means n goes to zero
               ]; //end inner while
             if[sum == j]
           then[return(sum);];
           sum := 0;
         ]; // end outer while
     return(false);
  ]; // end define
```

This requires nested loops, the outer to go over the integers to test, and an inner loop to take each number and test the place values.

Doing it the QDL way.

There are many ways to do it in QDL. Here is one that picks out all of the Armstrong numbers based on a test. This punches out each place value, cubes them, adds them then checks it is equal to the original number. Then you use this to pick out the ones < 1000.

```
    armstrong(a)->a == (a%100)^3 + (a%10-a%100*10)^3 + (a-a%10*10)^3;
```

```
    ~pick(@armstrong, [;1000]); // lead ~ turns result into a nice list
[0,1,153,370,371,407]
```

The QDL is a lot simpler, there are no (explicit) control structures and over all it is a bit more intuitive in the sense that if you handed it to someone with little explanation, there is a good chance they could figure out what it does, knowing what an Armstrong number is to start with. You read line 2 as “pick armstrong less than 1000”. Note that it returns an aggregate where as the standard procedural method returns a single value and you must iterate to find all you want.

Lambdas: The short form for functions

You may define functions quite tersely, but be advised that such things as function documentation etc. are not allowed. (Geeky stuff, this is a nod to Church's **λ calculus**). The syntax is either

```
signature -> single expression;
```

In which case, the expressions result will be returned. Two examples are

```
   f(x,y,z) -> x+y+z;
   f(3,2,1)
6
   sum(n)->(n!=0)?sum(n-1)+n : 0; // recursive function
   sum(7)
28
```

Or, to have multiple statements, put them inside a block:

```
signature -> {block}[statement 1; statement 2;…];
```

i.e. the **`block`** keyword is the default if missing.

Note that there will be no automatic return of a result, since these allow for very complex definitions (such as conditionals) and hence it is impossible to be able to predict the logic flow and right result.

```
   f(x,y,z) -> block[q:=x; q:=q+y ; q:= q+z ; return(q);];
   f(3,2,1)
6
```

Example of a quick functions that prints 1 if the argument is positive, 0 otherwise (just to show how to stick a conditional in):

```
  q(x) -> 0<x?1:0;
  q(0)
0
  q(2)
1
```

There are several built in functions in QDL and you can see them by issuing

```
)funcs system
```

in the workspace. Most of them have longish names for a reason. Partly it is to be descriptive, but mostly it is because these are to be the palette from which you draw your own functions. Since it is easy to create functions in QDL with lambdas, the easy words are left for you.

Variable visibility in lambdas and defined functions.

One major difference between lambdas and the full function definition with the `define` statement is visibility of the ambient environment: In lambdas, the ambient environment is still available where in defined functions it is not. So

```
   a := 4;
   f(x)->a*x;
   f(3)
12
   define[g(x)][return(a*x);]
   g(3);
unknown symbol 'a'
```

You may however get reduced visibility with a block statement and a lambda. This means anything defined in the block stays there, but it still inherits the ambient state:

```
  f(x)->block[y:=x^2;return(a*y);]
  f(2)
16
  y
unknown symbol 'y' at (1, 0)
```

Lambdas as arguments

If your function requires a reference, you can pass along a lambda in the call. For instance, to call reduce with the function **`x + y`**, on **`arg.`**

```
reduce((x,y)→x+y, arg.)
```

As function arguments, lambdas do not need a name.. You may certainly use one with the caveat that it will supercede any in the workspace inside the function. So for instance

```
  g(x,y)→ x-y;
  reduce(g(x,y)->x+y, n(10))
45
  g(3,4)
-1
```

So passing along a function named **g** does has that function and only that function used during the course of evaluation.

Summary

define[f(args)][…] ⇔ f(*args*) →local[…] have the exact same visibility.

f(*args*)→*expression* allows a single expression and returns its value. Very common idiom.

f(*args*)→block[*expressions*] allows a multiple expressions with no automatic return value. Note that this is how you make a multi-line body for a lambda functions.

Lambdas inside other functions.

You can nest function definitions inside function definitions these, so

```
    h(x) -> [f(x)->x^2; return(f(x));];
```

is fine.

Nesting

You may define functions within other functions, but they are only visible within the enclosing function. See visibility and lifetime below. For instance

```
define[
    f(x)
 ][
    s(x)->x^2;
    return(s(x));
];
```

In this case, the function **s(x)** is defined inside the body of the function, **f(x)**. Note that in this case x refers to dummy variables in the definition of **f** and **s**, and refers to the actual value that was passed only in the return statement, where it is evaluated.

Function visibility and lifetime

Functions are defined within the current block and you may define them pretty much anywhere, e.g.

```
if[
   x<2
][
   g(x)->x^2;
   //.. lots of stuff
]else[
   g(x)->x^3+1;
   //.. lots of other stuff
];
```

Within the **`then`** and **`else`** blocks, **`g`** exists as defined. Issuing a call to it outside of the block would cause an undefined function error.

One use is having conditionals define the function. Set the function to something trivial (much like setting a variable to **`null`** first:

```
g(x)-> null;
if[
```

```
  x < 2
][ //… etc.
```

Then subsequent calls to **g** would use whatever the definition turned out to be. It is true that you could also just have a conditional inside the function to select the expression, but a common pattern is the so-called *factory pattern* in which:

```
g(x)->null;
define[
   G_Factory(a,b,...)
][
  if[
   // condition
  ][
    g(x)->  . . .
  ];
  // lots of other logic machinery to determine various avatars of g
];

G_Factory(x,y, ...);
```

In this case, a very complex bit of logic determines what g is and sets it. You go back to the factory and use it every time you need to determine **g**. This keeps the use of **g** separate from implementation and runtime considerations.

Function references

Terminology. Algebraic operations are merely examples of functions that take one argument (monads) or two arguments (dyads). Functions do things with data (like numbers, strings, stems etc.). On the other hand, there are *operators* which do things to functions as well. References are the mechanism by which functions are passed to operators.

You may also pass along references to functions in other functions. A ***reference*** to a function is of the form

`@name` *or* **`@name()`**

where **`name`** is the name of the function and the parentheses are optional. There are no arguments. In a similar vein, a reference to an algebraic operation is of the form

@op

E.g. **`@+`** would be addition, **`@*`** would be multiplication.

This means that you can basically pass along the function as an argument to be applied. This works for most operations (such as + - * etc.)

```
   r(x)->x^2 + 1;
   f(@h, x) -> h(x)
```

The first function is defined. The second one (and this is a really simple example to show how this works) just applies the function to the argument. Note that in the definition, **h** is just a place holder. It will be replaced by the first argument. To invoke it,

```
   f(@r, 2)
5
```

So in this case, **f** takes **r** and applies it to **2** . This also works with operators too

```
   op(@h(), x, y) -> h(x,y)
   op(@*, 2, 3)
6
```

This passes along the operator ***** and applies it as a dyad (a *dyad* is a function with two arguments – all operators are simply dyadic functions) to 2 and 3. Similarly, a *monad* takes a single arguments (*e.g.* logical not or **!**). Note that operators always are realized as monadic or dyadic operators, depending on the number of arguments, since otherwise you would need some different notation for functions vs operators, which would get very cumbersome fast. And example of an operator that is both monaidc and dyadic is **-**, the minus operator. It can be monadic if it means the negative of a number, like **-4** or it can be dyadic and represent subtraction, *e.g.*, **5** or **-3**.

An alternate for the monadic signs are the raised minus sign (unicode 00af, ¯) or raised plus sign ⁺ (uncode 207a). This does have the advantage that it is never ambiguous, e.g. - - - b is unclear but -- ¯b or ¯- -b never are.

A very useful built in function that uses function references is **reduce** (and the related function, **expand**).

Example: Getting the even and odd parts of a function.

A function, **g**, even if and only if g(x) == g(-x) and odd if g(x) == -g(-x). The classic example of an even function is *x*^2 and of an odd function is *x*^3, and yes the name refers to the fact they act like even and odd powers. Generally functions are neither even nor odd, but can always be decomposed into even and odd parts (since this has a notation with fancy **o** and **e** in Math., we'll use that):

```
o(g) = (g(x) - g(-x))/2
```

```
e(g) = (g(x) + g(-x))/2
```

A common thing to do in Math is to grab the even and odd parts of **g** and work with those. How to do this in QDL is extremely easy with operators:

```
   odd(@g,x)->(g(x) - g(-x))/2;
   even(@g,x)->(g(x) + g(-x))/2;
   h(x)-> x^2 + x^3;
   h(2)
12
   odd(@h, 2);
8
```

```
   even(@h, 2)
4
```

Other Topics

Assertions

Many times you want to assert that a certain condition is met or that processing should (unrecoverably) end. This is the function of the **assert** construct. The form for it is very simple (Note that it is not an expression, but a regular control structure, so it is best to have it alone on a line.)

```
   assert[ conditional ][ feedback];
```

Note the feedback is any expression. If the value is a scalar, it is converted to being a string and set as the message. If a stem, that is set as the stem value. You can access these in a catch block just like any other error using the error_message or error_state. An alternate syntax uses unicode and is

```
⊨ conditional : feedback;
```

where ⊨ is unicode 22a8. Again, this really should be on a single line. This is a specific construction for the **`raise_error`** call which can make code vastly easier to maintain and read.

For instance.

```
  try[⊨false : {'a':'b'};]catch[say(error_code);say(error_message);say(error_state.);]
-2
assertion failed
{a:b}
```

Compare with raising the exception with the reserved system code of -2 manually:

```
   try[
     raise_error('my assert',-2,{'a':'b'});
       ]
  catch[
     say(error_code);
     say(error_message);
     say(error_state.);
    ];
-2
my assert
{a:b}
```

Caveat: This does allow for a general expression as the feedback. Bad practice is putting an entire error handling routine in the feedback. Good practice is having a simple string as the message or perhaps a structured stem. If the conditional is true, nothing happens and the expression is ignored. If it is false, then the *expression* is evaluated and the result is converted to a string and a specific error is raised (so you can catch these). These are identical

```
   assert[ 0 < size(args()) ][ 'you must supply at least one argument'];

   ⊨ 0 < size(args()) : 'you must supply at least one argument';
```

This is one of the most commonly occurring ways to control program flow, so properly speaking, it is simply a useful idiom.

E.g of returning a stem

In this example, a condition is checked and a structured stem is the feedback

```
// … code

⊨ old_lifetime == new_lifetime : { 'message':'lifetime not reset',
                                   'asserted': new_lifetime,
                                   'expected': old_lifetime};

// … more code
```

In which case some other try … catch block may write an error report based on the feedback.

Blocks

All a block does is create a local environment for its duration. This lets you create variables local to the block without cluttering your symbol table. There are two types, standard (keyword **block**) which inherits the ambient state and **local** which has no shared state. For instance, you may want to write an initialization script which sets a bunch of variables in your workspace. This must be executed with **`script_load`** to set variables, but it may need to do some work that involves variables and functions that you do not want propagated. Or if the current environment has many specialized functions, you may have to use a **`script_load`** since **`script_run`** would not inherit these. The solution is to stick them in a block. You may use blocks wherever you like, nest them, etc.

Example

If we had a script **`ldap_init`** we might want to write something like this:

```
/* global variables */
ldap. := null;
start_time = date_ms();
block[
   cfg. := file_read(script_args(0),1); // read in file as stem
   ldap.username := 'ou=people,' + cfg.0 ; // or whatever
   my_useful_function()- > ...
   // more local stuff
];
```

At the end of this script, **`ldap.`** and **`start_time`** are in now available in the workspace, but **`cfg.`** and **`my_useful_function`** are not. In this way, having a ton of helper variables and functions don't clutter up the user's workspace.

Example.

Here is an example of using a local block.

```
   a := 5
   local[ a:= 0;];
   1/a
0.2
```

A local environment allows you to use any state that you wish without having it propagate. As with a block, a local environment has its specific uses.

Help in Functions

Functions have a very specific documentation feature. At the very top of the body, you may enter documentation, each line is of the form

```
» text
```

or

```
=== text
```

and these will be taken when the function definition is read and kept available for consultation. You may get a full listing of every user-defined (meaning, not built in) function the workspace knows about. Such documentation is not available in lambdas because documentation has a specific format (so it can be found by the help system) involving per line statements.

Generic help

Each is printed as

```
    )help *
fib2(1): This will compute the n-th element of a Fibonacci sequence.
f(1): This is a comment
```

where there is the `name(number of arguments)` and the first line of the comment (which should be meaningful and explain to the user what the function does.) The first is a good comment. The second is not. **Note: `)help *`** is a “kitchen sink” option where you kind of know what you are looking for but don't really remember. It gives you a nice list to peruse and consider.

Specific help

If you want to read all of the documentation for a given function, invoke **`)help`** with the name of the function and its argument count. For instance;

```
    )help fib2 1
fib2(1):
This will compute the n-th element of a Fibonacci sequence.
A Fibonacci sequence, a_, a_1, a_2, ... is defined as
    a_n = a_n-1 + a_n-2
```

```
Acceptable inputs are any positive integer.
This function returns a stem list whose elements are the sequence.
```

There are many things that the)help command can do. Please see the workspace documentation for more.

Help for modules in the workspace

This will also work for loaded modules, so for instance to read the module help from the cli module

```
  cli := j_load('cli')
  )help -m cli
  module name : QDLCLIToolsModule
    namespace : qdl:/tools/cli
default alias : cli
… lots more
```

All of the following are equivalent, using either the namespace for the module or the variable. If you used to the older system (module_import) then this works as well. You may also access help for modules that are nested by giving their module path, e.g. `a#b#c#d`

```
  )help -m qdl:/tools/cli
  )help -m cli
```

To get list all of the functions in the module variable

```
  )funcs cli
to_stem([0,1,2,3])
1 total functions
```

To get help on a specific function, use the name and arg count.

```
  )help cli#to_stem 1
to_stem - convert a list of arguments for a script to a stem
to_stem(args. | marker) - if a list, process that with the default switch marker
         otherwise, use args() with the given switch marker
… lots more
```

or query it from the functions menu

```
   )funcs help cli#to_stem 1
```

The functions menu does allow for a more general query. To list all of the functions for each arg count, omit the argument count

```
    )funcs help cli#to_stem
to_stem(args. | marker) - if a list, process that with the default switch marker
to_stem(args., marker) - specify both the list of arguments and the switch marker
to_stem() - use args() as the argument, default switch marker of -
to_stem(args., marker, flags.) - specify the list of arguments, switch marker and
flags.
```

This also works with listing modules variables, if any, using

```
)vars module_name or path
```

There are also functions to list the functions and variables in a module. See below.

Operators or, what's up with the funny characters?

Many of the functions in QDL (e.g. `for_each`) have an operator form (e.g. ∀). The reasons for this are (1) simple readability and (2) streamlining your code once you are practiced at it. The only point is that the general form of an operator is modelled on that of monadic and dyadic operators, like -. So the general syntax is

operator *main argument*

additional argument operator *main argument*

A typical example is the transpose function,

transpose(arg.) ⇔ ⍉arg.

transpose(arg., axis) ⇔ axis ⍉arg.

Rest assured that you do ***not*** need to ever write with these and can ignore them for the most part. Others may use them (and often they really help readability). The major result of using them is reducing the number of lines and probably the number of intermediate results that need to get stored. This

```
∃a ∧ a≤2 ⇒ f(a):0
```

Just looks neater than

```
is_defined(a) && a <=2 ? f(a) : 0
```

Extending QDL with the library facility

One way to extend QDL is with the *library facility* which means to set directories whose scripts and modules are treated as if they were seamlessly part of the workspace. This is extremely useful when using QDL as a scripting language (as opposed to a general purpose programming language).

Example

Let us say you had a collection of scripts, all ending in .qdl and modules, all ending in .mdl in a directory /my/cool/scripts with content

```
setup.qdl
my_utilities.mdl
sort_dir.qdl
```

update_git.qdl

. If you enabled support and set your path:

```
 lib_support(true)
false
   lib_support(['/my/cool/scripts']);
[]
```

Then you may simply call scripts as if they were functions, e.g.

```
   setup();
   util := import(load('my_utilities'));
   test_files. := sort_dir(util#filter(dir(), '*.txt'));
   update_git(test_files., true, 42);
// …
```

Conventions

- All names must be valid QDL. Your OS might let you make a file named **`my!!util . qdl`** but that will not parse correctly in QDL. It would have to be *e.g.* **`my_util.qdl`**
- scripts *must* end in .qdl
- modules *must* end in .mdl
- Scripts etc. in subdirectories will not be resolved unless the path is added to the library path. Do take some care naming them. Having a script named **`setup.qdl`** in each of several directories will be hard to resolve right (though you can always just manually load or run it as a script if needed.)

The way it works is that scripts are run as if

```
script_load(file+'.qdl', arg0, arg1,…)
```

were called. Remember the side effect that the scope of the script is the current scope, so if you want variables and functions local to the script, enclose them in a **`block[]`** statement. This is for the default library mode of **`load`** though you may change the mode to **`run`** which will do the analogous call to **`script_run`** instead, but there will be no shared state with the workspace.

Although this looks like “syntactic sugar” it is a surprisingly natural and useful way to extend QDL, permitting a large collection of related scripts to grow. (Historical note: this was how the first incarnations of QDL operated, before the workspace was invented, which does a very similar, though vastly more flexible thing.)

See the **`lib_support`** function entry in the function reference for how to configure this in QDL. There are also workspace calls to do this as well.

Modules

Namespace control is an essential part of programming as is encapsulation. *Modules* in QDL accomplish this. to wit

A ***namespace*** is a set of names that refer to a set of objects. Using a namespace ensures that all objects have unique names with repsect to that namespace. A very common example is a file system, where every file within a directory has a unique name.

One of the most basic ideas in programming is *encapsulation* that is to say, a group of statements with their own state (so the variables and functions know about each other and their workings are independent of the rest of the environment). There is usually some mechanism to limit or control visibility to the outside. External code then does not have to be concerned with how inner workings of encapsulated code.

In QDL both of these are accomplished through modules. Each module has its own namespace and every object in the workspace is associated with a namespace. The namespace qualifier is a URN. It is profitable to think of namepsaces as a dictionary of resources. Modules may also have the visibility of the members controlled. One of the great evils of many scripting languages is that there is no encapsulation – every variable is global and the net effect is extremely hard to find bugs.

Module syntax

Modules are defined as

```
module[urn]
 {body}[
    === module level comments.
    statements];
```

The statements have local scope, so inside the body of the module you do not need to qualify variables or functions. By default elements inside the module are visible (see *intrinsic variables* below on how to make them externally invisible). Modules may be nested. Modules may use other modules *ad lib*, so there is no restriction. Modules may also reside in an external file (often with the suffix .mdl) and may be loaded. Loading a module is little more than reading it into the current workspace. You may either use the definition directly in the workspace, put it in a file and load it or write a Java extension and load that.

Supported operations

The following operations may be done on modules:

- **load** - If the module statement(s) are in a file, this has the effect of running **script_load.** The URI is registered with the workspace and you may create copies of it.
- **import** - Create an instance of the module using the ambient state. You may assign the module to a variable and pass it around like any other variable.

- **use** - this imports the given module into the ambient scope, so no namespace qualifications are needed.
- **loaded** - lists the loaded modules by their URI.
- **drop** - removes the loaded module
- **rename** - for a loaded module, changes the URI to a new URI.

The following operations interrogate a module for information

- **funcs** - list the functions in the module
- **vars** - list the variables in the module
- **docs** - list the module documentation (as a stem of strings).

An few examples

Here is a complete example of how to use a module.

```
   module['A:X'][f(x)→x;y:='foo';]; // loads it
   z := import('A:X'); // initializes it, puts it in z
   z#f(5)
5
   z#y
foo
   funcs(z)
[f([1])]
   vars(z)
[y]
```

What this does is show how to make a module available. The functions and variables are intended to be human readable lists though they can be parsed.

```
  loaded()
[A:X]
```

This returns a list of the URIs that are loaded in current scope (here, the workspace.)

```
   ww(e,s)->e#f(s)
   ww(z,3)
3
```

This starts to show the power of having modules as variables – you can pass them around and resolve them. So for instance you might have

```
   connect(db, parameters.)→db#connect(parameters.)
   my_sql := jload('db');
   postgres := jload('db');
   connect(my_sql, mysql_param.)
ok
   connect(postgres, pg_param.)
ok
```

in which multiple instances of a database module are running, each with their own state. This could even be used for another module that is a utility for working with databases.

Modules may be nested, so you may have modules within modules:

```
  module['A:Y']
        [module['A:X']
               [f(x)→x;
                y:='foo';
               ];
         z:=import('A:X');
       ];
  y := import('A:Y');
  y#z#f(3)
3
  loaded()
[A:Y]
```

Here z is a module in y. The last **loaded** command is to show that the internal z module local to y and the workspace is therefore unaware of it.

One final note on scope, when you import a module, it is created with exactly the state of the current workspace.

```
  w(z)->z^2
  module['A:Y'][f(x)->w(2*x);]
  import('A:Y') =: h
  h#f(3)
36
```

Things to do with modules

- encapsulate sets of functions to extend the workspace or QDL. E.g. a module that implements a set of specific Math functions. Typically you would want to dump these into the current scope with the **`use`** command.
- encapsulate specific functionality, e.g., a module to do database processing. You may import then various modules to talk to various databases.
- encapsulate specific utilities.

Import, use

There are two ways to get access to the module.

`import` will create a new instance and hand it back. You must assign it to a variable to use it.

`use` create a new instance and dump the contents into the current scope. You do not need to qualify the functions or variables and they may be treated like any other QDL object.

Example

Let us say that you had loaded the standard mathx module that comes with QDL. You could then either import it with its own namespace

```
   mathx := import('qdl:/ext/math')
   mathx#cosh(3)
10.0676619957778
```

Alternately, you could simply use it and make it an extension to the current workspace

```
   use('qdl:/ext/math')
   cosh(3)
10.0676619957778
```

Another use is inside of an module, where you can make it local, hence not need to use namespace qualification:

```
   module['my:/ext/math']
         [load('/path/to/modules/mathx.qdl');
          use('qdl:/ext/math');
          versinh(x)→ 2*sinh(x/2)^2; // hyperbolic versine
         ];
    h := import('my:/ext/math');
    h#versinh(1)
0.543080634815244
```

Logging and debugging

There are two additional ways to keep track of what QDL is doing. Logging consists of having a running *log* or file that has informational messages in it like

```
INFO: qdl(Fri Oct 23 10:39:53 CDT 2020): VFS MySQL mount: vfs#/mysql
```

A log then is a running accounting of how the system is running and doing things. You may write to the log. There is also a *debugging* facility included. This writes (to system error, not the console, though your output may end up there depending on how you have things set up). Debugging statements are normally put into your code and may be turned on and off as needed.

Think of logging as being part of the normal operations of more complex programs and debugging is for hard to track down issues. You do not want debugging information to end up in the log. Typically logs accumulate and are only looked at if there is an issue. Debugging statements are left in place and only turned on if there is some issue (such as log messages that something is not right, or other complaints).

These are primitive functions in the sense that you would normally do something like

```
trace(x, user)→debugger(1,'my_script.qdl at ' + date_iso() + '(' + user + '):' + x);
warn(x, user)→debugger(3,…);
```

and set up customized logging/debugging method with things like the script name, time stamps or whatever. In this case, information about the script and current user are printed.

The dividing line between what should make it in a log and debug is arbitrary. Both work the same. Which is to say there are levels of debugging/logging:

Name	Value	Description
OFF	10	Disable logging/debugging. Nothing is printed. Default for debug
SEVERE	5	an issue that prevents processing of anything. Generally issued right before raising an error.
ERROR	4	errors - the system cannot resolve the issue, but processing continues
WARN	3	warnings - things that are serious but don't require action
INFO	2	informational
TRACE	1	Print everything at all levels

Each level is more restrictive than the last. So if you set

```
   debugger(4);
3
// previous debug level was 3 so now only errors or above would get outputted.
```

Then debug commands at a lower level will not print. This lets you ramp the amount of output up and down as needed. Here is an example.

```
debugger(2); //set only warning on for debugger statements
if[!is_defined(user.id)][
 log_entry(1, 'Processing as anonymous user');
 //.. do anonymous user stuff
  debugger(1, 'checking anonymous permissions.');
]else[
 log_entry(1, 'user ' + user.id + ' found.');
 // do known user stuff. Say we can't find this user, log it:
 log_entry(3, 'user not found in database! Done.');
];
```

So as a point, in this example, it is entirely possible (and normal) that the user is not in the database and while the system can't do anything about it (and logs the fact), normal processing continues in this case.

You may reset the log and debug levels on the fly or specify them in the configuration file. The advantage to the latter is that for very complex systems, there is higher level control. A common use case is to set it in the configuration and never touch it.

Final note that if you have not configured debugging or logging then these commands do nothing. Turning debugging on in a session is a simple as issuing a command like

```
debugger(1)
```

(or any of the non-negative levels). Logs, however, involve writing to the system and are configured in the configuration file. Note that in server mode, most operations to change logging are forbidden since the server should control its log, not QDL.

Debugging configuration

The debugger can be configured and this is done by passing in a stem with various values.

Name	Default	Description
title	QDLWorkspace	This is the name associated with the entry.
ts_on	true	Whether or not to display timestamps in the entry. Generally you do want these on
level	10	The debugging level. The default is that debugging is off. You may use numbers of monikers, *e.g.* 'info', 'warn', etc.
delimiter	' | '	The delimiter between fields in the entry. This should enhance readability
host	--	If you want to display a host (really any string, since no checking is done) you may set it here. This is useful when debugging output from several different machines.

Note again that the debugger is quite configurable on the fly, but the logger is much more limited because it is configured in the configuration file.

Example. Getting the current debugger settings

```
  debugger()
{
     level : 10,
 delimiter :  | ,
     ts_on : true,
     title : QDLWorkspace
      host :
}
```

Example. Setting some values

```
   debugger({'title':'my entry','level':'info'})
{title:QDLWorkspace, level:off}
```

The response is the previous values.

Example. Making an entry

```
  debugger('info', 'test message')
true
2023-04-25T13:11:43.979Z | my entry | info: test message
```

The output of the debugger is the **`true`** which means that the debugger command executed successfully. Since this is the console, standard out gets dumped here and the next line is what the actual entry it. It is of the form

timestamp | title | level: *message*

Example. More realistic

Let us say you had a large complex set of scripts and needed to have very specific messages. The Right Way to do this is to define your own set of debug commands like

```
format_error(script, host, message)→ …; // format the message
trace(script, message)→debugger(1, format_error(script, 'localhost', message));
info(script, message)→debugger(2, format_error(script, 'localhost', message));
warn(script, message)→debugger(3, format_error(script, 'localhost', message));
```

etc. and use these in your code.

Capturing the debugging.

Normally this is done at invocation of a script like (assuming bash$ is the shell prompt) which captures it in a file as the program runs

```
bash$./my-script.qdl arg0 arg1 2> dbg.txt
```

This lets you run your program and test it at the command line like normal without having all the debug commands fall out. You can then look at the file at the end of you session.