QDL Reference Manual

Version 1.4

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.

Origin of the name

“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 claims 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 designed to prevent bad code. 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. 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

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

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.

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 *require* a leading number. So `0.3` is acceptable but `.3` is not. This prevents ambiguity with stem variables in parsing (and saves you from having a ton of parentheses to disambiguate cases). Basically if you write `.23` QDL cannot tell if you intended a number of forgot the stem and will tell you so in no uncertain terms.

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)`.

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.

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 ***compound*** variable is embodied in what is termed ***stem variables***. These are of the form

```
head.tail
```

(Geeky stuff, the period is actually called either *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 scalars 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. The index is any string and we usually refer to them as keys. 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.

Example

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

In

```
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.

```
   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.

Example

You can set the indices of stems either as constants:

```
  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.

There are two common use patterns. The first pattern is to invoke a function on a stem variable which alters every element and returns a stem variable with the same keys, each element having been transformed. Here is an example. This uses integer indices and this makes it a list. To populate it you would just set the elements:

```
myList.0 := 'the';
myList.1 := 'quick';
myList.2 := 'brown';
```

or perhaps, use the handy list notation:

```
myList. := ['the', 'quick', 'brown']
```

This effectively is an array (which is just a map whose keys are integers). The second use case comes from having indices that are not indices, which allows you to make maps on the fly, with the index being the key:

```
myMap.idp := 'https://idp.bigstate.edu/saml';
myMap.port := 636;
myMap.eppn := claims.sub + '@bigstate.edu';
```

or again in compact notation,

```
myMap. := {'idp':'https://idp.bigstate.edu/saml' , 'port':  636  , 'eppn':
claims.sub + '@bigstate.edu'}
```

In this case, there is now a map with keys, idp, port and eppn whose values are as above. Stem variables have their own section later with more details. There are several operations supported on them.

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.

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.

Compact Notation

You may create lists and stems with the so-called *compact notation*. The basic syntax is for lists:

```
[x0, x1, x2, …]
```

which would make a stem with elements **x0, x1, …** and indices **0,1,2,…** For stems

```
{key0:value0, key1:value1, …}
```

where the key are strings or integers and the values are arbitrary, including stems and lists. The most important thing to remember is that you may populate these with variable and functions, so something like

```
  [abs(-2), is_defined(arf)]
[2,false]
```

(and **arf** is not a defined function is this workspace).

Further examples

It is easy and convenient to use this notation. For instance this is fine

```
[2,4] + [3,5]
[5,9]
```

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]
```

or 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

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]

  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([;1000]/100)
```

This takes 3 traversals of the list. One to create it, one to divide everything by 100 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;1/10])
```

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

In general, function composition is you 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. := [;1000]/500 - 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.
```

which applies your function to each element of the list once. There are tradeoffs for speed. If your lists are short, then evaluating f(a.) vs using for_each is probably not essential and there is more overhead using for_each. If you are evaluating a million entries of a very complex expression, then for_each is the way to go.

Benchmarking on the cheap can be done with date_ms like so

```
  start:=date_ms();for_each(@f, x.) =: b.;date_ms()-start
  219
```

which gives the number of ms this took to execute. The reason that b. is assigned is because the entire result will print otherwise, which takes some time since an enormous list has to be formatted. Different computer systems will have different speeds.

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 create a special entry for the stem

```
   a.:= {*:2}
   a.0
2
```

Note that the key here is a ***** (not the character **'*'** which represents any key it does not recognize) or you issue a command

```
set_default(stem., scalar);
```

where the *scalar* is any scalar. Then from that point forward, any time a value is accessed, if it has not been explicitly set, the default is returned. If the stem does not exist, it will be created. Note that subsequent operations on the stem do ***not*** alter the default value.

E.g.

```
   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
```

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:

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

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

```
    say(!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.

```
    say(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:

```
   say(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.

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 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.

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.

Related functions

raise_error

Description

Raises an error conditions and passes control to the catch block. Note that you may raise errors outside of a **`try[. . . ]catch[. . .]`** block, but while it will interrupt processing, that's about it. This is useful in function definitions where you *e.g.*, check the arguments and raise an error if there is a bad one. You don't want to catch that inside the function because you are communicating it to whatever called your function.

Usage

```
raise_error(message, [code, [state.]);
```

Arguments

`message` a human readable message that describes this error

`code` a user-defined integer that tells what this code is. The value of -1 is reserved by the system for system errors. If you raise an error but do not set it, it is by default 0. You may use any other value you like, though as a convention stick to non-zero integers. This reserved value is available in **`constants()`** under **`error_codes.system_error`** and **`error_codes.default_user_error`** resp.

`state.` A user-defined stem that holds useful information. Nothing is returned by default and this is wholly user created. If you wish to pass along state, you must specify a code as well.

Output

None directly. The values are set to **`error_message`** and **`error_code`** (if present) and are accessible in the catch block. Typically, you define the error code and look for it in the catch block.

Examples

```
try[
    if[ 3 == 4]then[raise_error('oops', 2);];
]catch[
   say(error_message);
```

```
   switch[
      if[error_code == 2]then[...];
       // maybe a bunch of other errors
  ]; // end switch
]
oops
```

Use in a function

```
define[
   my_func(x)
  ][
   if[x <= 0][raise_error('my_func: the argument must be positive', 1);];
   // ..rock on
];
```

Since there is no catch block *per se* you can use any return code you like. The use of this is that if you were to make a call like

```
my_func(-2)^.5
```

an error would get raised in **my_func** rather than perhaps someplace else. This lets you control where exceptions were raised.

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 ∉ 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.

Related Functions

break

Description

Interrupt loop processing by exiting the loop.

Usage

```
break();
```

Arguments

None.

Output

None.

Examples

In this example, the loop terminates if the variable equals 3.

```
while[
     for_next(j,5)
   ][
     if[
```

```
        j==3
     ]then[
        break();
     ]else[
        say('j='+j);
   ]; // end if
]; // end loop
j=0
j=1
j=2
```

check_after

Description

Sometimes only a post-positional loop will do – this means that the loop executes at least once. This is not often the case, but is very hard to replicate. Invoking this function will do just that. Your condition will be checked post-loop.

Usage

```
check_after(condition);
```

Arguments

The argument is a logically -valued expression.

Output

None. This exists only in looping statements

```
a := 0;
while[
  check_after(a != 0)
 ][
  say(a);
];
0
```

continue

Description

During loop execution, skip to the next iteration.

Usage

```
continue();
```

Arguments

None.

Output

None.

Examples

In this example, the loop skips to the next iteration is the variable is 3.

```
while[
     for_next(j,5)
   ][
     if[
        j==3
     ][
        continue();
     ]else[
        say('j='+j);
   ]; // end if
]; // end loop
j=0
j=1
j=2
j=4
```

for_keys

Description

This is a non-deterministic loop over the keys in a stem variable. All the keys will be visited, but there is no guarantee of the order. *Also* the keys are strings unlike `for_next` where they are integers.

Usage

```
for_keys(var, stem.);
```

Arguments

`var` is a simple variable and will contain the current key during the loop. If it has already been defined, its values will be over-written.

`stem` is a stem variable. The keys of this stem will be assigned to the **`var`** and may be accessed.

Output

Nothing. This is only used in looping constructions.

Example

```
    my.foo := 'bar';
    my.a   := 32;
    my.b   := 'hi';
    my.c   := -0.432;
    while[for_keys(j,my.)]do[say('key=' + j + ', value=' + my.j);];
```

```
key=a, value=32
key=b, value=hi
key=c, value=-0.432
key=foo, value=bar
```

for_lines

Description

This allows for reading a text file one line at a time. It only works in a loop.

Usage

for_lines(*var*, file_path)

Arguments

var - the name of the variable

file_path - the path to the file

Output

This only exists in loops.

Example

If you had a file name `/tmp/mystery.txt`

```
   while[for_lines(x, /tmp/mystery.txt)][say(x);]
I can't help thinking:
Why is "abbreviation"
A very long word?
```

This example just prints out each line. The intent of this is for processing very large files, since the other option is to read the file in as a string or stem which may be slow or unwieldy.

for_next

Description

This allows for a deterministic loop and will run through a set of integers.

Usage

```
for_next(var, stop_value, [start_value, increment]);
for_next(var, arg.)
```

Arguments

Only the first two are required. The two cases are

First

var the variable to be used. As the loop is executed, this value will change.

stop_value the final value for the loop. When the variable acquires this value, the loop is terminated (so the loop body does **not** execute with this value!)

start_value (optional, default is 0). The first value assigned to *var*.

increment (optional, default is 1). How much the loop variable should be incremented on each iteration.

Second

var - the variable to be used. As the loop is executed, this value will change.

arg. - A list which will be iterated over, returning each value in the list.

Output

None. This only is used in loops.

Examples

A simple loop

```
while[
  for_next(j,5)
 ]do[
  say(j);
];
0
1
2
3
4
```

Another common way to use a loop is to decrement. Here it ends at zero, starts at 5 and the increment is negative, hence it counts down:

```
while[
  for_next(k, 0, 5, -1)
 ]do[
  say(k);
];
5
4
3
2
1
```

A list example.

```
  while[for_next(i,2*[;5])][say(i);]
0
2
4
6
8
```

Each value of the list is set to **i** in turn.

And here is an example of looping through the elements of a stem variable.

```
// Set the values initially
my_stem.0 := 'mairzy';
my_stem.1 := 'doats';
my_stem.2 := 'and';
my_stem.3 := 'dozey';
my_stem.4 := 'doats';

  while[for_next(x,my_stem.)][say(x);]
mairzy
doats
and
dozey
doats
```

has_value or ∈

Description

Loop over the values of a stem or set

Usage

has_value(*var*, stem.|set) *or var* ∈ stem. | set

Arguments

var - the variable to be referenced in the body of the loop

stem. or set - either a stem or a set.

Output

In loops, nothing. This may also be used generally, see below.

Example

Use the **`fibonacci`** function in the math extensions module to generate the first 25 Fibonacci numbers, then pick the even ones and print those out. Here **`pick`** returns a set and the values of that are iterated over.

```
   while[j∈pick((x)->mod(x,2)==0, fibonacci([;25]))][say(j);]
0
2
8
34
144
610
2584
10946
46368
```

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
```

Related functions

return

Description

Return a value or none. Note that this really only makes sense within a script or function. Issuing it in, *e.g.* the workspace will not have the intended effect you want since you are asking the system to hand off the value to another process. Only use this as indicated.

Usage

```
return([value]);
```

Arguments

One (optional). The value to be returned. No value means so simply exit at that point. Note that if a function normally ends and does not return a value, you do not need a *return();*

Output

The value to be returned.

Examples

Here is a cheerful little program that ignores everything and just returns “hello world”.

```
define[
   hello_world(x)
]body[
  return('hello world!');
];

     say(hello_world(42));
hello world!
```

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 monaic 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
```

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 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.

Related functions

debugger, logger

Description

Set the debugging level or issue debugging messages.

Usage

```
debugger()
debugger(cfg.)
debugger(level)
debugger(level , message)

logger()
logger(cfg.)
logger(level)
logger(level, message)
```

Arguments

(no arguments) - Inquire about current level

cfg. - a stem of configuration values (see previous section).

level - (only) set the current level for all subsequent operations. Use either integer or moniker

level, message = output the message at the given level

message - (only) output the message at the INFO (default) level.

Output

If there is no argument, the result is current configuration.

If the argument is a new level, the result is the previous level

Otherwise, a true or false is returned if the operation succeeded.

Examples

Note that you **must** set the highest level you want first – this sets the global logging/debug level. The way this operates is that you set the debugging/logging level to the maximum you want, then tag each message with the appropriate level. Logging levels are in the **`constants().sys_log`** stem. So for instance if you set the debug level to 3, then

```
debugger(2, 'foo')
```

would display nothing, since 2 < 3. Debugging is printed to standard error and ends up on the console, (GUI, ASCII or ANSI mode) while log entries are written to the log file (current one is **`info().boot.log_file`**.

See above for an example. Logging and debugging is not hard and is very, very useful. In particular, in cases where QDL is used for scripting on a server, it may be the only way to get feedback in real time about how processing is happening inside the QDL runtime.

Built-in function reference

There are several built in functions in various categories. All of these can take simple or compound variables. Functions have optional arguments denoted with curly braces, {}, so

```
my_func(arg.{, a | a.})
```

Means that my_func requires the first argument, a stem, but that the second argument is optional and is either a stem or a scalar, so all of these are valid invocations:

```
my_func(my_arg.);
my_func(my_arg., my_scalar);
my_func(my_arg., my_other_stem.);
```

There may be multiple optional arguments

String functions

contains

Description

To find if a string contains another string

Usage

```
contains(source, snippets{, case_sensitive})
```

Arguments

`source` – a string or stem of strings that is the target of the search.

`snippets` – a string or stem of strings that are what are being search for

`case_sensitive` – (optional) if this is **true** (default) then the check is done respecting case. If it is **false** then the matching is done after converting the arguments to lower case. (Note that the original values are never altered.)

Output

A scalar or stem (if the arguments were stems) if the snippet(s) was (were) found. Note that

- source a scalar, snippet a scalar → result is a simple boolean
- source a scalar, snippet a stem → result is a stem with identical keys to the snippet and with boolean entries

- source a stem, snippet a stem, → result is a stem with identical keys to the source and boolean entries.
- Both stems, the result is conformable to the left argument and the right. In other words, to be in the result, only entries with matching keys are tested.

Examples

Example 1.

```
a := 'What light through yon window breaks?';
contains(a , 'Juliette');
```

returns `false`, since there is no string `'Juliette'` in the first string.

Example 2.

```
source := 'the rain in Spain';
snippet.article := 'the';
snippet.1 := 'in';
snippet.2 := 'Portugal';
output. := contains(source, snippet.);

output.article == true;
output.1 == true;
output.2 == false;
```

Example 3.

```
source.foo := 'bar';
source.fnord := 'baz';
source.woof := 'arf';

snippet.foo := 'ar';
snippet.fnord := 'y';

output. := contains(source., snippet.);

output.foo == true;
output.fnord == false;
```

In this case, only the corresponding keys are checked if ***both*** arguments are stem variables.

detokenize

Description

Converts list of tokens into a string using the delimiter between entries. Note that each element of the list will be converted to a string. See also `tokenize`.

Usage

```
detokenize(arg, delimiter{,options})
```

Arguments

`arg` - stem of tokens to detokenize

`delimiter` - the delimiter to put between tokens

`options` - sum of 0 or 1 for append or prepend, 2 for omit dangling delimiter

So options = 0 means append, have a trailing delimiter

options = 1 means prepend, delimiter

options = 2 means append, omit trailing delimiter

options = 3 means prepend, omit first delimiter

Output

The detokenized string.

Example

```
   detokenize([;4],'|')
0|1|2|3|
```

Note the trailing | added at the end.

```
   detokenize([;4], '|', 0)
0|1|2|3|
   detokenize([;4], '|', 1)
|0|1|2|3
   detokenize([;4]], '|',2)
0|1|2|3
   detokenize([;4], '|', 3)
0|1|2|3
```

omits the trailing |. To make a blank delimited list:

```
  caps.
[
 foo,
 compute.create:/,
 storage.read:/store
]
    detokenize(caps., ' ')
foo compute.create:/ storage.read:/store
```

differ_at

Description

Find first index where two strings differ. If the strings are equal then a value of -1 is returned. If one string is a substring of another, then the index is the length (i.e. this is the index in the longer string). You may also apply this to stems of strings.

Usage

```
differ_at(s0, s1)
```

Arguments

`s0, s1` can be either strings or stems of strings.

Output

The first index where the strings fail to match or -1 if they are identical.

Example

```
   differ_at('abcde', 'ab'); // first index they are different is 2 in 1st arg
2
   differ_at('abcd','abcd'); // -1 means they are equal
-1
   differ_at(['abcd','efgp'],['abq','efghij'])
[2,3]
  differ_at(['abcde','abed'], 'abcq')
[3,2]
```

from_uri

Description

Take the output from the to_uri call and turn it into a single valid URI.

Usage

from_uri(uri.)

Arguments

A stem with the correct components for a uri.

Output

A string that is the uri.

Examples

```
  u := 'https://www.google.com/search?
channel=fs&client=ubuntu&q=URI+specification#my_fragment';
  uri. := to_uri(u);
  u == from_uri(uri.)
true
```

head

Description

Find the starting part of a string up to a given marker

Usage

```
head(target, stopChars{, is_regex})
```

Arguments

`target` - a string or stem of strings

`stopChars` - a string or stem of string of places to stop, or a regex that determines that.

`is_regex` - (optional) if true then all matching is done with the regular expression. Default is `false`.

Output

The part of target up to the stop character, *i.e.*, the head of each string. If the **`stopChar`** is not found, the empty string is returned.

Example

Here is taking the head of each element in a list:

```
    head(['bob@foo', 'todd@foo', 'ralf!baz'], '@')
[bob,todd,]
```

This returns everything in each string up to the @. Since there is no @ in the last string, the enpty string is returned for the last element. If you specify that the expression is a regex, then the effect is to split at the regex and return everything up to it. Note that this means that non-matches are returned as empty strings too:

```
   a.bob := 'bob@foo.bar'
  a.todd := 'todd@fnord.baz'
     a.x := 'TOM@Foo.bzz'
  head(a., '@f', true)
{
 bob:bob,
 x:,
 todd:todd
```

```
}
```

index_of

Description

This finds the position of the target in the source. If the target is not in the source, then the result is -1.

Usage

```
index_of(source, snippet{,caseSensitive})
```

Arguments

`source` – a scalar or stem variable.

`snippet` – a scalar or stem variable.

`case_sensitive` – (Optional) a boolean that if **true** will check for case and if false will check against the arguments as all lower case.

Output

if both are strings, the result is the first index of where the snippet starts in the source. If one is a stem and the other a scalar, the result is conformable to the stem and the operation is applied to each element of the stem. If both are stems, then only corresponding keys are checked.

Examples

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

    targetStem.all  := 'all';
    targetStem.One  := 'One';
    targetStem.bind := 'darkness';
    targetStem.7    := 'seven';
    index_of(sourceStem., targetStem.);
{bind:11}
```

i.e., it returns a stem variable, which has one entry, the common key and the index, so *darkness* is found starting at index 11 in *sourceStem*.

insert

Description

Insert a given string at a given position of another string

Usage

```
insert(source, snippet, index)
```

Arguments

`source` – the string to be updated

`snippet` – the string to insert

`index` – the position in the source string to insert the snippet

Output

The updated string.

Examples

```
    insert('abcd', 'foo', 2)
abfoocd
```

This also works for stem variables

replace

Description

Replace every occurrence of a string by another

Usage

```
replace(source, old, new{, is_regex})
```

Arguments

`source` – the original string or stem of strings

`old` – the current string

`new` – the new string.

`is_regex` - (optional) treat the second argument as a regular expression. If omitted, the default is *false*.

There is an statement about conformability. In this case if 2 or three of the arguments are stems, then only matching keys get replaced – the same key must be in all arguments or this is skipped. If exactly one of the arguments is a stem, then the replacement is made one each element with same arguments – in effect they are turned in to stem variables with constant entries. If all three are scalars it is just a standard replacement.

Output

The updated string

Example on a stem

And example with two stem variables and a simple string.

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

    old.all  := 'all';
    old.One  := 'One';
    old.bind := 'darkness';
    old.7    := 'seven';
    newValue := 'two';
    replace(sourceStem., old., newValue);
{bind:and in the two bind them}
```

The resulting output is a stem (because an input is) with the common index of *bind*

Why is this? Because the only key that the two stems have in common is 'bind' and that is applied to replace 'darkness' with the new value of 'two'.

An example using regular expressions.

In this example, all the spaces in a string are replaced with periods.

```
  replace('a b c  d e fgh', '\\s+', '.', true)
a.b.c.d.e.fgh
```

substring

Description

Return the substring of an argument beginning at the *n*th position.

Usage

```
substring(arg, n{,length} {,padding})
```

Arguments

```
arg – the string or stem of strings to be acted up
```

`n` - the start position in each string

`length` (optional) – the number of characters to return. Note that if this is omitted, the rest of the string is returned. If it is longer than the length of the string, only the rest of the string is returned *unless* the *pad* argument is given.

`padding` – (optional) a string that is used cyclically as the source for padding.

Output

The substring. Notice that this behaves somewhat differently than in some other languages in that it may be used to make results longer than the original argument.

Examples

A basic example. Remember that the first index of a string is 0, so *n* = 2 means the substring starts on the *third* character.

```
    a := 'abcd';
    say(substring(a,2));
cd
```

To use the padding feature

```
    say(substring(a,3,10,'.'));
d.........
```

And do note that the *padding* need not just be a character, but will be repeated as needed:

```
    say(substring(a,1,20,'<>'));
bcd<><><><><><><><><
```

Finally, a stem example. Note that the padding option makes all results the same length:

```
    b.0 := 'once upon';
    b.1 := 'a midnight';
    b.2 := 'dreary';
    d. := substring(b., 0, 15,'.');
    while[for_next(j,3)]do[say(d.j);];
once upon.......
a midnight......
dreary..........
```

Or you could get fancy do do something like make a table of contents:

```
    d. := d. + ' p. ' + n(3)
   while[for_next(j,3)]do[say(d.j);];
once upon....... p. 0
a midnight...... p. 1
dreary.......... p. 2
```

tail

Description

Return the right hand side of a string given a delimiter to start at.

Usage

```
tail(target, delimiter (, is_regex))
```

Arguments

`target` - the input (string or stem of strings) to be acted on

`delimiter` - the marker to be found. The *last* such marker is used

`is_regex`- if **`true`** then all matching uses **`delimiter`** as a regular expression. **`false`** is the default.

Output

The tail of of the string(s) or the empty string if there is no match. The delimiter is not returned.

Examples

```
  tail('qwe@asd@zxc', '@'); // only last occurrence is returned.
zxc

  tail('qweaAzxc', '[aA]+', true)
zxc
```

The second example uses a regex to do a case insensitive match on double a.

to_lower, to_upper

Description

This will convert the case of a string to all upper or lower case respectively.

Usage

```
to_lower(arg), to_upper(arg)
```

Arguments

`arg` – either a string or a stem variable of strings. Non-strings are ignored.

Output

A conformable argument of strings.

Examples

```
    a := 'mairzy doats';
    b := to_upper(a);
    say(b);
MAIRZY DOATS
```

to_uri

Description

Parse a string into a stem whose entries are the (RFC 3986 compliant) components

Usage

```
to_uri(string)
```

Arguments

string - any string. If the string is not a valid URI, this will fail.

Output

A stem variable of the components, each of which is a string (except the port, which is an integer.) Note that no supplied port sets the value to -1;

Examples

```
  u := 'https://www.google.com/search?
channel=fs&client=ubuntu&q=URI+specification#my_fragment'
  to_uri(u)
{
 path:/search,
 fragment:my_fragment,
 scheme_specific_part://www.google.com/search?channel=fs&client=ubuntu&q=URI+specification,
 scheme:https,
 port:-1,
 authority:www.google.com,
 query:channel=fs&client=ubuntu&q=URI+specification,
 host:www.google.com
}
```

So you can see what the host is, grab the fragment, look at the query.

```
   tokenize(to_uri(u).query, '&')
[channel=fs,client=ubuntu,q=URI+specification]
```

splits the query into its elements quite nicely.

tokenize

Description

This will take a string and and delimiter then split the string using the delimiter.

Usage

```
tokenize(arg, delimiter{,useRegex})
```

Arguments

`arg` – either a string or stem of strings

`delimiter` - either a delimiter string or a regular expression (last argument is **true**.)

`useRegex` - (optional) second argument is a regular expression. Default is **false**.

Output

If *arg* is a string, then a list of tokens. If *arg* is a stem, then a stem of stems. Remember that the keys are preserved if the argument is a stem and a simple list (keys are 0, 1,…) if a string.

Examples

A simple example

```
    say(tokenize('ab,de,ef,',','));
[ab,de,ef]
```

An example tokenizing a stem variable.

```
    q.foo := 'asd fgh';
    q.bar := 'qwe rty';
    say(tokenize(q., ' '));
{bar:[qwe,rty], foo:[asd,fgh]}
```

Tokenizer only works on strings. Here is the result of attempting to tokenize an integer.

```
    say(tokenize(123145, '1'));
123145
```

In this case, the argument is returned unchanged.

Example with a regular expression

```
   a := 'a d, m, i.n'
   r := '\\s+|,\\s*|\\.\\s*'
   tokenize(a,r,true)
[a,d,m,i,n]
```

Don’t forget that you can do regular expression matching with the =~ operator.

trim

Description

Trim trailing space from both ends of a string. One point to note is that since stem variables can contain stem variable, this only operates at the top-level and if you wish to trim included stems you must do so directly. This prevents “predictable but unwanted behavior.”

Usage

```
trim(arg)
```

Arguments

`arg` is either a string or a stem of strings. This function has no effect on non-strings.

Output

A result conformable to its argument.

Examples

An example

```
    a := '   blanks   ';
'blanks' == trim(a);
```

Another example, using a mixed stem variable.

```
    my_stem.0 := '   'foo';
    my_stem.1 := -42;
    my_stem. := trim(my_stem);
    my_stem.0 == 'foo';
    my_stem.1 == -42; // unchanged.
```

Math functions

abs

Description

Find the absolute value of a number

Usage

```
abs(arg)
```

Arguments

`arg` – a number or a stem filled with numbers.

Output

If a single number, the absolute value of that number. If a stem of numbers, the absolute value of all of them.

Examples

```
    say(abs(-123));
```

```
123
```

date_ms, date_iso

Description

Compute and convert dates between milliseconds and the ISO 8601 standard format.

Usage

```
date_ms([arg])
date_iso([arg])
```

Arguments

either none, an argument or stem of arguments.

None:

`date_ms()` – returns the current time in milliseconds

`date_iso()` – returns the current time in ISO 8601 format.

A single argument

`date_ms(arg)` -- if *arg* is in ms, return it, otherwise convert it to ISO format

`date_iso(arg)` – if *arg* is ISO format, convert it to ms. Otherwise return it.

Output

The date in the appropriate format.

Examples

```
    say(date_iso());
2020-01-18T22:10:38.250Z

    say(date_ms('2020-01-18T22:10:38.250Z'));
1579385438250

    say(date_ms(1579385438250));
1579385438250
```

decode

Description

Decode an encoded string. The result will be a simple string, so if the original is binary, you will see gibberish.

Usage

```
decode(arg{,type})
```

Arguments

`arg` – may be a string or a stem of strings. Each string will be decoded.

`type` - optional integer for the type of encoding or decoding. Default is 64 for base 64.

```
type  name   notes
 0     v     Variable encode/decode that is QDL safe. Used in boxing, some JSON
 1     url   URL encode/decode
 16    hex   RFC 4648, character set is a-zA-Z0-9-_
 32    b32   RFC 4648, character set is A-Z2-7
 64    b64   RFC 4648, character set is  A-Za-z0-9-/
```

Base 16,32 and 64 follow this specification.

Output

The decoded string.

Examples

```
    decode('VGhlIHF1aWNrIGJyb3duIGZveCBqdW1wcyBvdmVyIHRoZSBsYXp5IGRvZw')
The quick brown fox jumps over the lazy dog
```

encode

Description

Encode a string in. The returned string is URL safe.

Usage

```
encode(arg{,type})
```

Arguments

`arg` – a string or it may be a stem of strings.

`type` - integer that sets the type.

Output

If a single argument, it will be encoded. If a stem, each element will be. Non-strings are not changed.

Examples

```
    encode('The quick brown fox jumps over the lazy dog')
VGhlIHF1aWNrIGJyb3duIGZveCBqdW1wcyBvdmVyIHRoZSBsYXp5IGRvZw
```

hash

Description

Calculates the digest of the arguments and returns the value as a hex string. There are various algorithms supported:

Supported algorithms are:

```
md2
md5
sha-1
sha-2 (same as sha-256)
sha-256
sha-384
sha-512
```

The arguments are case insensitive. The default is SHA-1, which is a fixed 20 byte result (and the resulting output is a hexadecimal string exactly 40 characters long, regardless of the size of the input string).

Usage

```
hash(arg{,algorithm})
```

Arguments

`arg` – either a single string or a stem of strings.

`algorithm` – which algorithm to use. Default is SHA-1

Output

A hex string that is the hash. Note that while this is a hex string, it is most emphatically not the same as the output from the `encode` function.

Examples

```
    hash('the quick brown fox jumps over the lazy dog')
2fd4e1c67a2d28fced849ee1bb76e7391b93eb12

    hash('the quick brown fox jumps over the lazy do')
6186ce3119913cfabfe4b7952ba765b132948dd2
```

A couple of examples showing other hashes

```
   hash('the quick brown fox jumps over the lazy dog','md5')
77add1d5f41223d5582fca736a5cb335
  hash('the quick brown fox jumps over the lazy dog','sha-2')
05c6e08f1d9fdafa03147fcb8f82f124c76d2f70e3d989dc8aadb5e7d7450bec
```

A point to make about hashes is that even a tiny change in the input completely changes the output in

very hard to guess ways. This is what makes them so useful in *e.g.*, security. A very common use is to generate a password in an application and store the hash of it. This means only the user has the actual password and when presenting it, the hash is recomputed and checked.

identity, i

Description

This simply returns its argument. It is the identity function

Usage

```
identity(x)
i(x)
```

Arguments

x - any valid QDL expression or value.

Output

x (the input)

Examples

This is extremely useful in complex expressions to organize stem indices and such. It gives a way use only function notation. Consider

```
   n(3).n(4).n(5).i(0)
0
```

This evaluates from right to left, so **i(0)** returns the index of **0** and as it marches backwards, each of the indices function – which return the integers from **0** to **n-1** – does the same.

max

Description

Compute the maximum of two numbers

Usage

```
max(a,b)
```

Arguments

a - a number or stem of numbers

b - a number or stem of numbers

Output

The largest of the two arguments.

Example.

```
   max(3,2.5)
3
   max([-2,5],[2,4])
[2,5]
```

min

Description

Compute the minimum of two numbers

Usage

```
min(a,b)
```

Arguments

a - a number or stem of numbers

b - a number or stem of numbers

Output

The smallest of the two arguments.

Example.

```
   min(3,2.5)
2.5
   min([-2,5],[2,4])
[-2,4]
```

mod

Description

Compute the modulus, *i.e.*, the remainder after long division, of two integer. Since there are currently only integers as allowed numbers, this is needed in cases where the remainder is required.

Usage

```
mod(a,b)
```

Arguments

a and b may be either scalars or stems.

Output

Examples

To compute the simple remainder

```
    say(mod(27,4));
3
```

And sure enough 27/4 = 6 with a remainder of **3**.

A stem example. Computer the remainder of a bunch of values

```
    a.0 := 11;
    a.1 := 20;
    say(mod(a.,4));
[3,0]
```

In this case, since 20 is evenly divisible by 4, the modulus (aka remainder) is 0.

Another example

Here is how to make 5 random integers in the range of -18 to 20:

```
  1 + mod(random(5),20)
[-5,-12,13,-2,-14]
```

(The random numbers are signed so the smallest using the mod function could be -19, adding 1 gives us -18.)

Or if you prefer all positive numbers in the range of 1 to 20:

```
  1 + abs(mod(random(5),20))
[16,11,16,2,20]
```

numeric_digits

Description

Set or query the precision for decimal numerical operations. Since decimals do not completely represent fractions, this sets the precision (i.e., number of significant digits) used.

Note: significant figures start at the left of the number. So if we had precision of 2, then 1.20 and 1.234 would be equivalent. This is not the number digits to the right of the decimal point. Consider the following snippet for a function **h(x)** defined in terms of transcendental functions:

```
    numeric_digits(15)
50
```

```
    h([1,2.3])
[
 -10.0676619957778,
 -1.3600025400480680000000000000
]
```

Both of them have 15 significant digits, but the second value has a lot more decimals. Since **h(x)** is defined in terms of transcendental functions, the extra values are artifacts of approximation.

Usage

```
numeric_digits([new_value])
```

Arguments

`new_value` (optional) if supplied, the new value for all non-exact decimal operations.

Output

The current value.

Caveat.

Make sure your precision matches your needs. Consider this

```
    9223372036854775806 + 3
9223372036854780003
```

Which cannot be right. What gives? Since the precision is 15 and the number you gave is 18 digits long, what happened is that the number was rounded to 15 places, then 3 was added. So the value is right … to 15 places. If you want to see all of your digits, you need to set the precision correctly:

```
    numeric_digits(25)
15
    9223372036854775806 + 3
9223372036854775809
```

Examples

The default is 15 digits. Set the value to 50:

```
  numeric_digits(50)
15
  4^0.19
1.3013418554419335668321600491224611591208423214517
```

Set the number of digits to 100 and re-evaluate this expression

```
  numeric_digits(100)
15
  4^0.19
```

```
1.30134185544193356683216004912246115912084232145172743294973477331558331880613340430618283829970273
5
```

random

Description

Generate either a single signed 64 bit random number (no argument) or a list of them.

Usage

```
random([n])
```

Arguments

`number` (optional) -- the number of random values you want to generate.

Output

If there is no argument, a single random 64 bit number. If there is a number, *n*, supplied. Then a stem variable with indices 0,1,… n-1 containing 64 bit integers.

Examples

```
    say(random());
8781275837297675785

    random(5)
[-7902203766022328986,-60507163193724589,
3266880166912262770,-895740002133721315,-181676033275893516]
```

Here is an example of generating a list of 10 random numbers between 1 and 10:

```
  x. := 1+ abs(mod(random(10),11));
  say(x.);
[6,5,4,1,8,6,6,8,8,9]
```

random_string

Description

Generate a random string. There are random strings and there are random strings. I mean is that this will be pseudo-random (really best a computer can do) and it will be the correct number of bytes. The result will be base 64 encoded. See note in *encode* about length of strings. Note especially that the first argument is the number of *bytes* you want back, not the length of the string.

Usage

```
random_string([n{,count}])
```

Arguments

`n` (optional) – gives the size in bytes. The default is 16 bytes = 128 bits.

`count` (optional) – the number of strings to return. If this is larger than 1, then you will get a stem back.

No arguments returns a single random string that is the default size.

Output

A base 64 string that faithfully (url safe) encodes the bytes.

Examples

```
  random_string()
Kb5NlgFgTRDWp_qW7MyUEA
```

Returns a random string 16 bytes = 32 characters long (the default). Note that this is 22 characters long as 16*4/3 = 21.33333 rounds up to 22.

```
  random_string(32)
uf04ljhu90899QPHOMWsywxLafjsieU2nRtdeffhSvY
```

Returns a single string that is 32 bytes = 43 characters long.

```
   random_string(12,4)
[lHZhFtAYlSR-85pU,GWkbC3q5iNRFNBdT,Yp6M6Bir1JTArEuF,9CaQ2knCaiSp11-_]
```

Returns 4 strings that are 12 bytes = 16 characters long.

An example where the result needs to be a hex string.

In this example, we need a random, hex string that is 16 bytes long. Here's how to do it.

```
  encode(decode(random_string(16)),16)
efbfbdefbfbd3e7374efbfbd22efbfbdefbfbd06efbfbd10c69d2178
```

Transcendental functions

(Transcendental functions are those that cannot be represented by polynomials or rational expressions of them. They therefore "transcend" Algebra which explains the name given to them in the late 18th century, i.e., they require infinite series.)

These are the standard math functions you would expect. Rather than have separate entries, here is a list (**y** refers to the output values, **x** to the inputs):

Name	input	output	Description
`acos(x)`	`-1 <= x <= 1`	`0 <= y <= π`	arc cosine, result in radians
`acosh(x)`	`1 <= x`	`0 <= y`	inverse of the hyperbolic cosine

asin(x)	-1 <= x <= 1	-π/2 <= y <= π/2	arc sine, result in radians
asinh(x)	any	any	inverse of the hyperbolic sine
atan(x)	any	-π/2 < y < π/2	arc tangent, result in radians
atanh(x)	-1 < x < 1	any	inverse of hyperbolic tangent
ceiling(x)	any	any	the ceiling of the number.
cos(x)	any	-1 <= y <= 1	cosine of the angle x, x in radians.
cosh(x)	any	1 <= y	hyperbolic cosine
exp([x])	any	0 < y	exponential function. **exp(x) == e^x**
floor(x)	any	any	the floor of the number
gcd(x,y)	any int	int	greatest common divisor
lcm(x,y)	any int	int	least common multiple
ln(x)	0<x	any	natural (base *e*) logarithm, inverse is **e^y**
log(x)	0 < x	any	base 10 logarithm. Note inverse is **10^y**
nroot(x,n)	n odd, any x n even, 0 <= x	any	Compute the nth root of x. Note that **n != 0** must be an integer
pi([x])	--	π	the power of pi in the current precision, **π^x**
π([x])			identical to pi(), π is unicode \u03c0
sin(x)	any	-1 <= y <= 1	the sine of the angle x, x in radians
sinh(x)	any	any	hyperbolic sine
tan(x)	any	any	the tangent of the angle x, x in radians
tanh(x)	any	any	inverse of the hyperbolic tangent.

Notes:

1. Both exp() and pi() (or π()), take arguments, raising them to the indicated power. No argument means the default argument is 1.
2. *e* is not used as a number because it conflicts with engineering notation, so *e^x* won't work. Use *exp(x)*

Note that *x* here is a scalar, but these will operate on stems and lists. This is a good collection that should cover most cases and it is easy to define others you need. E.g.

```
sec(x)->1/cos(x);
asec(x)->acos(1/x);
logn(x, n)->ln(x)/ln(n); // convert a log to another base, n.
```

Example

The first few powers of 2 are

```
2^n(5)
[1,2,4,8,16]
```

If you wanted to get the logarithm, base 2, $\log_2$ `(x) = ln(x)/ln(2)` so to do this for our list:

```
 ln(2^(n(5)))/ln(2)
[
 0E-15,
 1.000000000000000,
 2.000000000000000,
 3.000000000000007,
 4.000000000000000
]
```

Note. While QDL supports arbitrary decimal precision, remember that computing the above values often relies on algorithms that converge slowly to the answer and in bad cases the time rises as the square of the digits. QDL will dutifully compute everything to 10,000 places for your 100 digit number if you like, but you must embrace patience. Need we remind you that physical measurement stops about `10^(-11)`? For most real life problems, precision of 15, these functions converge very quickly.

List functions

expand (⊕)

Description

Apply a dyadic function pairwise to each member of a list, returning the intermediate results.

Usage

```
expand(@f, list.)
```

Arguments

`@f` - reference to the function you want to use.

`list.` - the list to be operated on

Output

A list where the (dyadic) function f is applied to each element in the list successively.

Examples

The *factorial* of a number, n! is the product of all the numbers 1 * 2 * . . . * n. Here's how to compute the factorial of 5 with all the factorials for 1, 2, 3 and 4:

```
   expand(@*, 1 + n(5))

[1, 2, 6, 24, 120]
```

It is more obvious if we show it against the arguments

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

```
   *  *   *   *
[1, 2, 6, 24, 120]
```

Compare this with **reduce** which only returns the final result.

Another example: Getting part of a list

Let's say we wanted to get only the elements of a list of integers less than or equal to 4. Here's how

```
  a. := 1+ 2*n(5)
  mask(a., expand(@&&, a. <= 5))
[1, 3, 5]
```

A final example. Computing the terms of a series

If we have a series that depends on the previous term and current index, such as $x_n = (x_{n-1}^2 - 1)/(n^2 + 1)$

then this can be done as follows:

```
  r(x,y)->(x^2 - 1)/(y^2+1)
  expand(@r, [1;10])
[1,0,-0.1,-0.058235294117647,-0.038331101943039,-0.026987316935779,
-0.019985433694492,-0.015378470499077,-0.012192237837135]
```

insert_at

Description

Insert a sublist into another list, starting at a given point. All the indices in the target list are shuffled to accommodate this.

Usage

```
insert_at(source.{, start_index, length}, target.{, target_index});
```

Arguments

`source.` = the stem that is the source of the copy.

`start_index` = the index in the source where the copy starts. Default is 0

`length` = how many elements to copy, default is from start_index to end

`target.` = the target stem of the copy

`target_index` = the index in the target that will receive the copy. default is 0. Note that any elements already in these locations will be moved. If you need to overwrite elements, consider using the *copy* command.

Examples

```
  source. := [;5] + 20
  target. := [;6] - 100
  insert_at(source., 2, 3, target., 4)
```

```
[-100,-99,-98,-97,22,23,24,-96,-95]
```

So this inserted 3 elements from the source starting at index 2 in the source and placed them at index 4 in the target, moving everything else.

copy

Description

Copy from one list to another.

Usage

```
copy(source.{, start_index, length}, target.{, target_index})
```

Arguments

`source.` = the stem that is the source of the copy.

`start_index` = the index in the source where the copy starts. Default is 0

`length` = how many elements to copy, default is from start_index to end

`target.` = the target stem of the copy

`target_index` = the index in the target that will receive the copy. default is 0. Note that any elements already in these locations will be replaced. If you need to insert elements, consider using the *insert_at* command.

N.B. That the start and target indices may be signed, so you can specify from the end of the list.

Output

The updated target stem. Note that the target *is* modified in this operation.

Examples

```
  source. := [;5]+10
  target. := [;6] - 50
  copy(source., 2, 3, target., 4)
[-50,-49,-48,-47,12,13,14]
```

So this took the 3 elements from source. starting at index 2 and copied them to target. starting at index 4 there.

pick

Description

A function that chooses elements of the argument based on a boolean valued function.

Usage

```
pick(@f, arg)
```

Arguments

`@f` - any boolean valued function that will accept the elements of the argument. Valence is 1 or 2.

`arg` - any argument, stem, set or scalar.

If the valence of `@f` is 1 then the function gets the current value of the element. If the valence is 2, then the arguments are `key, value`.

Output

A result conformable to arg whose elements satisfy the pick function. If the argument is a scalar, so is the result (and will be null if the pick function returns false). Note that this does not alter the argument. Any boolean results for `@f` that are false and not included.

Example

```
  pick((v)-> 7<v<20,[|pi(); pi(3) ; 10|])
{2:9.33374465952533, 3:12.4298206624931, 4:15.52589666546087, 5:18.62197266842864}
```

In a list of 10 numbers between π and π^3 which are between 7 and 20?

```
  pick((k,v)-> 0<(v^2/(k^2+1))<3/2,[1;10])
{0:1, 4:5, 5:6, 6:7, 7:8, 8:9}
```

Sets criteria for selecting elements from the given list. (k,v) are the key (i.e. index) and value.

An example on sets. Since there is no concept of a key or index, you can only have a pick function based on the value:

```
  pick((v)->'a'<v, |^random_string(10,10))
{dSBREElOapucxQ,mW_29aEYR_MyaQ,KEAIL-qoKm8a4g}
```

This makes some random strings and grabs only those that have a character of 'a'. Note that the result is also a set.

reverse

Description

Reverse the order of the elements in a list

Usage

```
reverse(list.)
```

Arguments

`list.` - the list to be reverse

Ouput

The elements of list. in reverse order. Note that this will only return the list part of the argument. If there are any extra stem entries, they will be omitted.

Examples

```
reverse([4,5,'a','b'])
[ b, a, 5, 4]
```

sort

Description

Sort an object

Usage

```
sort(arg{,up})
```

Arguments

`arg` - the argument. This may be any type.

`up` - (optional) a boolean that if true (default) sorts in ascending order and if false, descending order

Output

A list. Note that different types are not comparable. QDL's solution is to sort each type and return them in groups. If your data is of one type (e.g. numbers, strings) then this exactly sorts as you expect.

Example

```
   sort([5,-2/3, 4/7, cos(pi()/8)])
[-0.666666666666666,0.571428571428571,0.923879532511287,5]
  sort([5,-2/3, 4/7, cos(pi()/8)], false); //descending order
[5,0.923879532511287,0.571428571428571,-0.666666666666666]
```

A mixed example

```
  sort({'a':'SPQR','b':-2/3,'c':3/7,'d':'woof'})
[SPQR,woof,-0.666666666666666,0.428571428571428]
```

Note that strings are grouped first, then numbers. Certain elements (like sets and embedded stems) are simply grouped unordered at the end of the list. The point is that this sorting is for probabilities – the

vast majority of the time you have homogeneous data and want to sort that – vs. possibilities, where you have some very odd data structure.

starts_with

Description

Find the indices of elements in the right that start elements in the left. This is the case that you have a bunch of strings (no order and may or may not be complete or have too many elements) and need to know which ones start which. This only works on strings at present and only for lists. Read the name as "(left) list starts with..."

Usage

```
starts_with(target., caputs.)
```

Arguments

target. = a list of elements which are to be searched.

caput. = (Latin caput =head) are the starting of lines to be used.

Output

A list conformable to the left argument, *i.e.*, the list is identical in length. The values are which element in the right fulfills the requirement. If there is no match, then a value of -1 is used.

Examples

```
   starts_with(['a','qrs','pqr'],['a','p','s','t'])
[0,-1,1]
```

read this as:

left arg index 0 starts with right arg index 0
left arg index 1 starts with nothing on right
left arg index 2 starts with right arg index 1

How to get the subset of things that start? Use mask:

```
mask(X., -1 < starts_with(X.,Y.))
```

So

```
 mask(['a','qrs','pqr'], -1 <starts_with(['a','qrs','pqr'],['a','p','s','t']))
{0=a, 2=pqr}
```

sublist

Description

Grab a sublist of a given argument with elements addressed by a range of indices (lists only). This operates on lists only.

Usage

```
subset(source., start_index{, count});
```

Arguments

`source.` - the stem list to take a subset of

`start_index` - where to start in the source stem

`count` - (optional) how many elements to take. Omitting this means take the rest of the argument

Output

The altered stem. Note that this does nothing to the original stem.

Example

```
  // grab a subset of a set - all elements less than 3.
  subset((x)->x<3, {-2,0,4,5})
{0,-2}
  stem. := [;5] + 20;
  // Just grab the tail of this list
  subset(stem., 2)
[22,21,24]
  // grab some stuff in the middle.
  subset(stem., 1, 3)
[21,22,23]
```

Signed start_index is allowed

You may use a negative start_index – this simply counts from the end of the list

```
  subset([;15], -7, 4)
[8,9,10,11]
```

Using a list to get a subset.

In this case,

```
    2*[;5] +1
[1,3,5,7,9]
   subset(3*[;10], 2*[;5]+1)
[3,9,15,21,27]
```

Note that you do not need to stick with integer keys

```
   subset(3*[;15], {'foo':3,'bar':5,'baz':7})
{bar:15, foo:9, baz:21}
```

Stem functions

box

Description

Take any set of variables and turn them in to a stem, their names becoming the keys. This removes them from the symbol table so the only access afterwards is as part of the stem. See the function *unbox* for the inverse of this.

Usage

```
box(var0, var1, …);
```

Arguments

There must be at least on argument. The arguments are variables that have been defined. These will be put in to a stem and removed from the symbol table. Arguments may be scalars or stems.

Ouput

A *true* if this succeeded.

Examples

```
  a. := -5 + i(5);
  b. := 5 + i(5);
  )vars
a., b.
c. := box(a., b.);
    )vars
c.
```

So a. and b. no longer are in the symbol table, but are in the stem:

```
  c.
{a=[-5,-4,-3,-2,-1], b=[5,6,7,8,9]}
```

common_keys

Description

Find the keys common to two stems

Usage

```
common_keys(stem1., stem2.)
```

Arguments

stem1. and stem2. are any stems.

Output

A list of keys common to *both* stems. The order of the stems does not matter

Examples

```
  common_keys(n(10), 6+n(5))
[0,1,2,3,4]
```

diff

Description

Compare two stems, returning a stem of the differences only between them, element by element.

Usage

```
diff(x., y.{,subsettingOn})
```

Arguments

`x.` - the first argument

`y.` the second arguments

`subsettingOn` - (optional) a boolean that if true means that only common keys are processed. if false, then missing keys are treated as if they have a **`null`** value.

Output

A stem whose keys are the same in both (`subsettingOn == true`) or *every* key in either of the stems (`subsettingOn == false`) In the case of reading files line by line, this gives a QDL analog to the venerable unix command line utility diff.

Examples

```
   diff({'a':'p','b':'q'},{'a':'p','b':'r', 'c':'t'})
{b:[q,r]}
```

There is one difference between these stems. For the key **b** the first argument has value **q** and the second has value **r**.

```
  diff({'a':'p','b':'q'},{'a':'p','b':'r', 'c':'t'}, false)
{b:[q,r], c:[null,t]}
```

In this case, subsetting is turned off and in addition to the first result, it tells us that for the key **c** the first argument is missing this and the second has a value of **t**.

```
   diff({'a':'p','b':'r', 'c':'t'}, 'p')
{b:[r,p], c:[t,p]}
```

In the case of a scalar argument, this is extended to every element on the left, so the result says that entries **b** and **c** do not have the value of 'p'

dim

Description

Return the dimension vector associated with a stem.

Usage

```
dim(arg)
```

Arguments

`arg` - the argument to operate on. It may be a scalar (trivial case) or a stem.

Output

If `arg` is a scalar, the output is the constant 0. If it is a stem, it is the number of independent axis each with their size.

Examples

```
    dim(4)
0
```

A scalar has no dimension.

```
   dim(n(3,4,5))
[3,4,5]
```

Dimension vectors can be extremely useful to query a stem about its structure.

exclude_keys

Description

Remove a set of keys from a stem.

Usage

```
exclude_keys(stem1,, stem2.)
```

Arguments

`stem1.` = the target of this operation.

`stem2.` = a list of keys to be removed. Note that the *values* of this stem are what are to be removed from the target. There is no assumption that the keys of stem2 are integers, for instance.

Output

A new stem that contains none of the keys in stem2.

Examples

```
    a.foo := 'q';
    a.bar := 'w';
    b.w := 42;
    b.a := 17;
    say(exclude_keys(b., a.));
{a=17}

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

    list.0 := 'rule';
    list.1 := 'bring';

    exclude_keys(a., list.)
{bind:and in the darkness bind them,
 find:One Ring to find them}
```

for_each (∀)

Description

Apply an n-adic function, f, to each element of the outer (Cartesian) product of the arg_k.

Note that in QDL, there is subsetting involved in stem operations. For something like

```
   1+2*n(4) == n(7)
```

```
[false,false,false,false]
```

This is a very natural choice and the right one. However, if we needed to preserve the second argument, so 7 elements, ***for_each*** is made to order.

```
  reduce(@||, for_each(@==, 1+2*[;4], [;7]))
[false,true,false,true,false,true,false]
```

This compares every element in n(4) with every element in n(7), then reduces that result. `reduce` squishes along the zero-th axis, leaving the 7 element list. See below for another example of creating a table of values this way. You can pass in *any* function you want to do this. e.g.

```
   q(x,y) -> x*y
z. := for_each(@q, [|-1;1;10|],[0;2;0.25])
```

Note: This applies to the zero-th axis of each argument. If you really want to get geeky, dyadic **`for_each`** generalizes

$$a. \otimes b.$$

i.e., the tensor product of two vectors to arbitrary functions, not just multiplication. Geeky aside: The reason that we use @ for function references is because it kinda looks a tiny bit like the tensor product sign. Kinda. In any case, you can use the ⊗ (unicode , \u2297) in place of @ if you like, so

```
   reduce(⊗∨, ⊗≡∀[1+2×[;4], [;7]])
[false,true,false,true,false,true,false]
```

is fine QDL.

Comment. `for_each` largely replaces looping in most cases. If you are writing a `while[]` loop, do consider if `for_each` would work. Not always (or we would not have a looping construct) but it does work a very large percentage of the time, if more efficient and makes the code vastly easier to read.

Usage

```
for_each(@f, arg_1, arg_2, ..., arg_n)
@f∀[arg_1,...arg_n]
```

Arguments

`@f` - the n-adic function. It will be fed all of the elements from all the stems. You may also have lambdas.

`arg_1., arg_2., … arg_n.` - n values (stems or scalars) to process. every point of each of these will be fed in succession to ***f*** to evaluate.

Note that for the `∀` notation, the argument is a list and the left hand side must be a function reference (not a lambda).

Output

A stem consisting of the outer (Cartesian) product of each of the arguments with the function ***f*** applied to each. Dimension is `[dim(arg_1.), dim(arg_2.),. . ., dim(arg_n.)]`

Examples

Simple example, making a multiplication table.

```
 a. := for_each(@*, 1+[;5], 1+[;6])
 a.
[
 [1,2,3,4,5,6],
 [2,4,6,8,10,12],
 [3,6,9,12,15,18],
 [4,8,12,16,20,24],
 [5,10,15,20,25,30]
]
```

Things to note:

- The result is the product of the stems, so here there is a 5 x 6 array that results. a.i.j is the product of the i-th and j-th elements.
- An alternate way to do this would involve a double `while` loop over the elements, multiply them then set the value of a.i.j directly. `for_each` should be considered any place you might want to loop, since it is probably the more efficient way to do it and vastly easier to use than nested loops.

Example

Fill a 5×6 array with zeros

```
   for_each((x,y)->0,[;5],[;6])
[
  [0,0,0,0,0,0],
  . . .
 ]
```

In this way, `for_each` lets you create any sized array you want with any values. A 10×10 identity matrix:

```
   for_each((x,y)->x==y?1:0,[;10],[;10])
[
  [1,0,0,0,0,0,0,0,0,0],
  . . .
 ]
```

Example.

Create the grid points for a quadric (polynomial) surface over a region.

```
  z. := for_each((x,y)->x*y, [|-1;1;15|],[0;3;0.25])
  dim(z.)
[15,12]
```

This creates a table over the region of the plane for -1 <= x <= 1
and 0 <= y < 3. There are 15 total points in the x direction and the y direction is done in increments of 0.25, resulting in 12 values, so the result is a 15 x 12 array.

Example.

Turning a stem of values into a stem of string.

One common case is that you have a stem of values, **a.** and need to change each element to a string. Invoking the **`to_string`** method does not give you the result:

```
    z. := ['abc', true, 0.23, -1]
   to_string(z.) // A single string of characters, not a list
[abc,true,0.23,-1]

  w. := for_each(@to_string, z.) // returns a list of strings.
  w.
[abc,true,0.23,-1]
  size(w.)
4
```

from_json

Description

Take a string that represents an object in JSON (JavaScript Object Notation) and return a stem representation. JSON is quite popular, (as in, it is inescapable anymore) but do remember it is a notation for objects that live in Javascript. To be blunt, it was designed to send information in REST-ful applications but because it is easy to use, is now being used by many as a general data description format. It was intended to tightly couple in-memory Javascript data structures on a server to be processed by a browser, so using it generally is arguably a bad idea. And yet, here we are. If you stick to the intended original purpose, it works great.

Note that there are some incompatibilities. First off, there is no actual standard way for JSON to represent all data, so you have to know what the structure of a JSON object is before you get it. A URI may be a string, or it may be represented as some arcane JSON structure. Same with dates.

Stems are more general data structures than JSON, and cannot be fully represented. For instance, JSON objects do not have default values, not can they represent sets. If you have a stem of sets with a default value, it may be very hard to get a resonable JSON format of it. This is a limitation of JSON.

Heading in the other direction, JSON can be represented as a QDL stem, *except* that QDL does not allow quite stem keys that end in a period. This JSON object:

```
{".":"a"}
```

cannot be directly ingested as a QDL object since it would be interpreted as a trivial stem. Compare:

```
  from_json('{"a.":"a"}')
{a:a}
```

So we see that the extra "." on the key is consumed converting it to a stem. This can be just fine in most cases, but does prevent round-tripping. This is why there are various conversion options that use encode.

```
    q := '{".":"a","..":"b","...c..":"c"}'
    j. := from_json(q,0)
    j.;
{
 $2E:a,
 $2E$2E:b,
 $2E$2E$2Ec$2E$2E:c
}
  to_json(j., 0,0)
{".":"a","..":"b","...c..":"c"}
```

Usage

```
from_json(var{, convert_type})
```

Arguments

`var` = the string or stem of strings to be converted

`convert_type` = (optional) if *true* apply *encode* with the given type to each key.

Output

A stem that contains the information in the original.

Note that JSON properties will be turned in to valid QDL variable names, escaping them if requested. This permits good interoperability.

Examples

A simple example.

```
  from_json('{"woof":"arf","0":0,"1":1,"2":2}')
[0,1,2]~{woof:arf}
```

Reading a file

```
// here is a large, messy example
  b := read_file('/tmp/my_json.json');

// (some indenting was done manually to keep this vaguely readable)
  b
{"a":"b","s":"n","d":"m",
 "foo":{"tyu":"ftfgh","rty":"456","ghjjh":"456456",
  "woof":{"a3tyu":"ftf222gh","a3rty":"456222","a3ghjjh":"422256456"},
 "0":"qwe","1":"eee","2":"rrr"},
"0":"foo","1":"bar"}
   // make a stem
  b. := from_json(b)
  say(b., true)
[foo,bar]~{
 a:b,
 s:n,
 d:m,
 foo:[qwe,eee,rrr]~ {
  tyu:ftfgh,
  rty:456,
  woof:  {
   a3tyu:ftf222gh,
   a3rty:456222,
   a3ghjjh:422256456
  },
  ghjjh:456456
 }
}
// And just to check it really is a stem
  b.foo.woof.
{a3tyu=ftf222gh, a3rty=456222, a3ghjjh=422256456}
```

A stem example.

Here is an example to read an encoded JWT (JSON Web Token) from the clipboard, decode it and turn it into a stem. A JWT is of the form X.Y.Z where X, Y and Z are base 64 encoded. Z (if present) is a binary signature, so cannot be read.

```
  jwt()->from_json(decode(tokenize(cb_read(), '.')\[0,1]));
```

Typical JWT. Copy it to the clipboard:

```
eyJraWQiOiJCRUZGRjU4NzZEMTAiLCJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.
eyJ3bGNnLnZlciI6IjEuMCIsImF1ZCI6Imh0dHBzOi8vd2xjZy5jZXJuLmNoL
2p3dC92MS9hbnkiLCJzdWIiOiJqZ2F5bm9yIiwibmJmIjoxNjU2MDIwMjMxLCJz
Y29wZSI6Ii9ob21lL2plZmYifQ.
bYPQkk7VDPVF4EYM4KpPRtzdIEyaPraEc7Tg
-xr6FBXzg5gDUdEWyscAvBbGm77Pj0Hn0OJrKZ1lux8SgB_DkBo
```

Do note that base 64 decode ignores any embedded blanks and linefeeds, as per the spec.

```
  jwt()
[
 {
 kid:BEFFF5876D10,
 typ:JWT,
  alg:RS256
 },
  {
  wlcg.ver:1.0,
  aud:https://wlcg.cern.ch/jwt/v1/any,
  sub:jgaynor,
  nbf:1656020231,
  scope:/home/jeff
 }
```

Note that the we extract the first two fields since the last field of Z here is binary hence not valid JSON and therefore and cannot be turned into a stem.

Escaping of JSON keys.

In this example, a simple JSON list is given and the key for this list is not a valid QDL key (ends in a period). In this case, the name is escaped.

```
  a. := from_json('{"#$rt.":[0,1,2]}', 0);
  a.
  from_json('{"#$rt.":[0,1,2]}', 0)
{$23$24rt$2E: [0,1,2]}
  a.$23$24rt.2 := 5
  a.
{$23$24rt.:[0,1,5]}
  to_json(a., 0,0)
{"#$rt":[0,1,5]}
```

Since you can loop over all indices in a stem easily, odd names can be handled:

```
if[is_defined(a.encode('#$rt.',0))]then[/*do stuff*/]
```

has_key (∋, ∌)

Description

Check is a list of keys is in a target stem.

Usage

```
has_keys(list., target.) or list.∋ target. or list.∌ target.
```

Arguments

`target.` – the target of this operation

`list.` – a list of keys.

Output

A boolean list with *true* as the value if the target contains the key and *false* if it does not.

Examples

Just because it is easy to do, I am going to make a stem filled with 5 random integers, then a list of 10 indices. Obviously only the first 5 indices in *w.* will be in *var.*:

```
    var. := random(5);
    w. := n(10);
    has_keys(w., var., w.)
[true,true,true,true,true,false,false,false,false,false]
   w. ∋ var.
[true,true,true,true,true,false,false,false,false,false]
    w. ∌ var. ; // does NOT have these keys
[false,false,false,false,false,true,true,true,true,true]
```

has_value (∈, ∉)

Description

Check if an argument contains a given value or set of values. Note that this effectively is a search function for its arguments.

Usage

```
has_value(left_arg, right_arg) or left_arg ∈ right_arg
```

Arguments

`left_arg` - A stem or scalar.

`right_arg` - a stem or scalar.

Output

The result is a boolean or stem of booleans. The result is always conformable to the `left_arg`, so if that is a scalar, the result is a scalar. If it is a stem, the result is a stem with identical keys. If both are scalars, this is the same as invoking equality (==). Extremely useful in conjunction with the `mask` function.

Examples

```
    a. := n(3);
    b. := n(5) * 2;
    c.foo := 1; c.bar := 'arf';
    has_value(a., b.)
[true,false,true]
    has_value(b., a.)
[true,true,false,false,false]
    has_value(a., c.)
[false,true,false]
```

```
    has_value(c., a.)
{
 bar:false,
 foo:true
}
   has_value('arf', c.)
true
```

A useful construct is to pair this with the mask function to grab exactly the bits of a stem you want

```
  mask(a., has_value(a., 2))
{2:2}
  mask(a., a. ∈ 2); // Identical, just shorter syntax
{2:2}
```

The effect then is that a. is searched to see if it contains the value of 2. It does (at index 2) and mask strips off everything else. You now have the right index and the right value.

include_keys

Description

Take a stem, *a.* and a list of indices, *list.* and return the values of *a.* that have the same indices as *list.*

Usage

```
include_keys(var., list.)
```

Arguments

var. – a stem

list. - a list of keys. These will be the keys of the result, *var.* will supply the values.

Output

A stem with the keys from the *list.* and the corresponding values from the *var.*

Examples

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

    list.0 := 'rule';
    list.1 := 'bring';

    say(include_keys(a., list.));
{
 bring:One Ring to bring them all,
 rule:One Ring to rule them all}
```

indices

Description

Return all the indices or keys in a stem *or* restrict to the keys for a given axis.

Usage

```
indices(arg.{,axis})
```

Arguments

`arg.` - the stem whose keys will be returned

`axis` - (optional) the axis for the keys. If it is missing, the zero-th axis will be assumed, making this identical in function to `list_keys(arg.)`

Output

A stem whose elements are the keys. If axis 0 is requested, the stem will be just a list of indices. If other axes are requested, the stem will be stem indices. This is also a great way to get a table of contents for a generic stem, though it is often simply too verbose for things like a rectangular array (e.g. a matrix) where are the indices are predictable.

Examples

Remember that

`a.p.q . . . r == a.[p,q, . . . r]`

And the list on the right hand side is called a stem index. In this example we create a list and show what the set of indices looks like.

```
    a. := [;5]~n(2,3, n(6))
    a.
[0,1,2,3,4,[0,1,2],[3,4,5]]
    indices(a., 0); // get the first axis
[0,1,2,3,4]
    indices(a., -1); // get the last axis
[[5,0],[5,1],[5,2],[6,0],[6,1],[6,2]]
    a.[6,2]
5
```

is_list

Description

Determine if the argument is precisely a list. That means, that it is a stem with only integer indices.

Usage

is_list(stem.)

Arguments

`stem.` The stem to check

Output

A boolean that tells if this is a list.

Examples

```
     my_stem.help := 'this is my stem'
     my_stem.~ n(5)
[0,1,2,3,4]~{
 help:this is my stem
}
  is_list(my_stem.)
false
```

Why is this false? Because it has a non-integer index. The function tells you if the object is a list and only a list. Compare with

```
  is_list([;10])
true
```

join

Description

Join two stems along a given axis. This increases the number of elements on the axis.

Usage

```
join(x., y., axis)
```

Arguments

`x., y.` are stems and should be conformable.

`axis` - an integer stating which axis to use.

By *axis* we mean which index of the stem's dimension. So in

`a.p.q.r`

`a.` means axis 0

`a.p` is axis 1

`a.p.q` is axis 2

See the examples below. The standard ~ operator is just a join along the default axis of 0, and operator, **~|** (unicode 2241, ⍤) that will do the join along the last axis.

Output

The joined stem. Note that the dimension does not change, but the axis (which refers to the index of the dimension) will. Easiest to look at the example below rather than describe.

Examples

Simplest is best.

```
   [;5]~[;5]
[0,1,2,3,4,0,1,2,3,4]
```

This joins the two stems along their zero-th axis and the number of elements on that axis is now 10.

A larger example

It is best to have a large example so you can see what is going on.

```
    q. := [[n(4), 4+n(4)],[8+n(4),12+n(4)], [16+n(5),21+n(5)]]
    w. := 100 + q.
    dim(q.)
[3,2,4]

    q.
[
 [[0,1,2,3],[4,5,6,7]],
 [[8,9,10,11],[12,13,14,15]],
 [[16,17,18,19,20],[21,22,23,24,25]]
]
    w.
[
 [[100,101,102,103],[104,105,106,107]],
 [[108,109,110,111],[112,113,114,115]],
 [[116,117,118,119,120],[121,122,123,124,125]]
]
```

So w. and q. have the same dimension. Invoking `join` without an axis means the join along the zeroth axis. This means that the zeroth dimension will change:

```
  // also q.~w.
   z. := join(q.,w.)
   dim(z.)
[6,2,4]

 //  *   <--- You are here
 //  z.i.j.k
 join(q.,w.,0)
[
 [[0,1,2,3],[4,5,6,7]],
 [[8,9,10,11],[12,13,14,15]],
 [[16,17,18,19,20],[21,22,23,24,25]],
 [[100,101,102,103],[104,105,106,107]],
 [[108,109,110,111],[112,113,114,115]],
```

```
 [[116,117,118,119,120],[121,122,123,124,125]]
]
```

`result` is list of combined lengths size(z.) == size(q.) + size(w.)

the second argument is treated as a list and just tacked on to the first.

```
   z. :=  join(q., w., 1)
   dim(z.)
[3,4,4]
 //   *  <--- You are here
 // z.i.j.k
[
 [[0,1,2,3],[4,5,6,7],[100,101,102,103],[104,105,106,107]],
 [[8,9,10,11],[12,13,14,15],[108,109,110,111],[112,113,114,115]],
 [[16,17,18,19,20],[21,22,23,24,25],[116,117,118,119,120],[121,122,123,124,125]]
]
```

`z.` has same shape, but `z.k == q.k ~ w.k`

This tacks all the first entries together

```
   z. := join(q.,w., 2)
   dim(z.)
[3,2,8]
 //     *  <--- You are here
 // z.i.j.k
[
 [[0,1,2,3,100,101,102,103],[4,5,6,7,104,105,106,107]],
 [[8,9,10,11,108,109,110,111],[12,13,14,15,112,113,114,115]],
 [[16,17,18,19,20,116,117,118,119,120],[21,22,23,24,25,121,122,123,124,125]]
]
```

`z.` now has size(`z.i.k`) == `size(q.i.k) + size(w.i.k)`

```
   z. :=  join(q.,w., 3)
rank error
```

A rank error happens if you exceed the actual entries of the stem.

A more concrete example

Let us say you wanted to create functions that produce pairs of values for a plotting program (such as gnuplot). In QDL you could just create a function:

```
define[
   plot(@f(), x.)
][
   x. := n(size(x.),1,x.); // resize x to nx1 column vector.
   return(x.~|f(x.)); // note that x.~|f(x.) == join(x., f(x.), -1)
];
```

This evaluates whatever the function is on the column vector. The result is a join of the x. and f(x.) along their last axis, so that inputs and outputs are together. This can be written very easily to a comma delimited file (e.g.) or perhaps just dumped as a JSON string and the plotting program can then read it. In QDL you generally describe what you need the data to do, such as here, make some values and glom them together. See also the laminate function in the extension module.

keys

Description

Return a stem of the keys for a given stem variable.

Usage

```
keys(arg.{,scalars_only | var_type})
```

Arguments

arg. = A stem variable. Remember that the entire stem is referenced by just the head + “.”

`scalars_only` = (optional boolean) if true lists only the keys that are for scalars. If false, only the keys for stems are listed. If this is omitted, all keys are returned.

`var_type` = (optional) The variable type as an integer. If this is specified then only indices with a value of that type are returned.

Output

A stem of keys where every key has itself as the value.

Examples

Example. Let's say you have the following stem variable”

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

and you issue

```
  keys(sourceStem.)
{bind:bind,
 find:find,
 bring:bring,
 rule:rule}
```

Note that there is no canonical order to keys, so the keys to `sourceStem.` Can appear in any order in the result.

This is extremely useful with *e.g.* the rename function. So you can get all the keys, change their values and rename them.

Examples of filtering

Let us take the following example of a stem with various types of entries

```
   a. := ['a',null,['x','y'],2]~{'p':123.34, 'q': -321, 'r':false}
    keys(a.)
[ 0, 1, 2, 3]~{ p:p, q:q, r:r}
```

Returns every key. remember that the values for the variables types are in

```
  constants('var_type')
{
 boolean:1,
 string:3,
 null:0,
 integer:2,
 decimal:5,
 stem:4,
 undefined:-1
}
```

To get the filter keys for the boolean entries only

```
  keys(a., 1)
{r:r}
```

To get the null entries:

```
  keys(a., 0)
{1:1}
```

To get only the integers

```
  keys(a., 2)
{q:q,3:3}
```

To get only the stem entries (vs. scalar)

```
  keys(a., false)
{2:2}
  a.2 // just checking
[x,y]
```

How to get only the entries that are scalar valued

```
  keys(a., true)
{
 p:p,
 q:q,
```

```
 r:r,
 0:0,
 1:1,
 3:3
}
```

Again, part of the contract for this call is that there is **no** canonical ordering, since there cannot be for stem keys generally.

An example to rename keys

Let us say we had the following stem with these keys (which were generated someplace else and we imported, *e.g.*, from JSON):

```
  b.OA2_foo := 'a';
  b.OA2_woof := 'b';
  b.OA2_arf := 'c';
  b.fnord := 'd';
  b.
{
 OA2_arf:c,
 OA2_foo:a,
 OA2_woof:b,
 fnord:d
}
```

The list_keys() command gives the following

```
  list_keys(b.)
[OA2_arf, fnord, OA2_foo, OA2_woof]
```

To rename all the keys so that any with the OA2_ prefix are changed, issue

```
    remap(b., list_keys(b.), list_keys(b.) - 'OA2_')
{
 arf:c,
 foo:a,
 fnord:d,
 woof:b
}
```

Example contrasting shuffle with rename_keys

This creates a list `[2,9,16,23,30,37,44]` of 7 elements. First we simple reorder them. Note that the length of the left argument is 7 and that every index is represented:

```
 shuffle(2+n(7)*7, [6,4,2,5,3,1,0])
[44,30,16,37,23,9,2]
```

In the next example, we just rename key 0 (only index on right) to 15. The display is no longer with square brackets because it is, properly speaking, no longer a list.

```
  rename_keys(2+n(7)*7, [15])
{
 1:9,
 2:16,
 3:23,
 4:30,
 5:37,
 6:44,
 15:2
}
```

To be exhaustive you can also use stem notation for the left side in this case, even though it is also a list:

```
  rename_keys(2+n(7)*7, {0:15})
{
 1:9,
 2:16,
 3:23,
 4:30,
 5:37,
 6:44,
 15:2
}
```

list_keys

Description

Return a list of the keys for a given stem variable. It allows masking by value type.

Usage

```
list_keys(arg.{,scalars_only | var_type})
```

Arguments

`arg.` = A stem variable. Remember that the entire stem is referenced by just the head + “.”

`scalars_only` = (a boolean) if true lists only the keys that are for scalars. If false, only the keys for stems are listed. If this is omitted, all keys are returned.

`var_type` = (optional) The variable type as an integer. If this is specified then only indices with a value of that type are returned.

Output

A list of keys where the new keys are cardinals and the values are the keys in the original stem. See also the *keys()* function. The difference is that this is list but *keys()* returns the set of keys. This is useful for reshuffling indices.

Examples

Example. Let's say you have the following stem variable”

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

and you issue

```
  list_keys(sourceStem.)
[bind,find,bring,rule]
```

And to just list the scalar indices (note that strings are treated as scalars):

```
  list_keys(sourceStem., true)
[bind,find,bring,rule]
```

If you tried to list just the keys for the stems, you would get an empty list back:

Note that there is no canonical order to keys, so the keys to `sourceStem.` Can appear in any order in the result.

Example:Masking

Let us say you issued a complex statement using *mask()*, like

```
  ww. := random(4, 8)
  ww.
[
 5652543194030156086,
 6244984374016755256,
 3047862711518522798,
 2011719346505809871
]
```

we need to grab the one element that has remainder 11 after division by 17 (this generates an example where one of the ones in the middle is what we want) so we use *mask()*:

```
  mask(ww., mod(ww., 17)==11)
{
 3:2011719346505809871
}
```

which dutifully informs us that the 3rd element is the one we want. To actually grab this, you can use *list_keys()* and stem resolution:

```
   k. := list_keys(mask(ww., mod(ww., 17)==11));
   ww.k.0
2011719346505809871
```

Another example: looping

Let us say that you got the above key set. How might you use it? In a loop:

```
while[
   for_keys(j, var.)
]do[
   sourceStem.var.j // … do stuff with this
];
```

which loops through all the values in source stem.

Another example: getting only values of a certain type.

```
    q. := {'a':['p','q'],'b':'r', 'c':false,'d':123.345,'e':42}
    list_keys(q., 2); // 2 is the variable type for integers
[e]
```

Meaning, that this is a list for the keys (there is one here) of integer-valued entries in the stem.

Note:The following are equivalent

```
    list_keys(q., false)
[a]
    list_keys(q., 4)
[a]
```

So it is possible to, *e.g.* loop over only the decimal elements in a stem.

mask (⥣)

Description

Take a boolean mask of a stem.

Usage

```
mask(target., bit_stem.)
```

Arguments

`target` – the stem variable to acted upon.

bit_mask – a stem variable with the same keys as target and boolean values. If the value is **true** then the entry is kept in the result and if **false** it is not. Note that if there are missing keys then these will not be returned either (so subsetting is still in effect), essentially making them equivalent to **false** entries.

Output

A subset of the target.

Examples

```
    header.transport := 'ssl';
    header.iss := 'OA4MP_agent';
    header.idp := 'http://oa4mp.org/idp/secure';
    header.login_allowed := 'true';
    // case insensitive match
    header. := mask(header., !contains(header., 'oa4mp', false));
    // This removes every entry containing 'oa4mp'
    say(size(header.);
2
```

n

Description

Make a list of indices. This is very useful in conjunction with looping. Note that the one simple case

```
n(p) == [;p]
```

for 0 < p an integer. `n`, however, allows you to make higher dimension stems or reshape them.

Usage

```
n(arg0{,arg1,arg2,...}{,fill.});
n(dim.{,fill.});
```

Arguments

`arg_k` (first form) are numbers. This will be the size of the resulting index set.

`dim.` - (second form) a list of dimensions.

`fill.` - (optional)stem of scalars that will be used as values cyclically. Note that even if this has a single entry, it must be a stem.

Output

A list, *i.e.*, a stem variable whose keys are the integers, and whose entries are either the same or the elements of fill. re-used cyclically.

Examples

Simple examples.

```
   n(5)
[0,1,2,3,4]
   n(5,[2,3])
[2,3,2,3,2]
   n(2,3)
[
  [0,1,2],
  [0,1,2]
 ]
```

```
  n([2,3],[;6])
[
  [0,1,2],
  [3,4,5]
]
```

In the second example, the result is a 2 rank array aka a 2 x 3 matrix of integers. Since no fill was specified, the default of the last argument extended holds. Here is an example of a 2x3x6 array filled with zeros.

```
  n([2,3,6],[0])
[
  [
    [0,0,0,0,0,0],
    [0,0,0,0,0,0],
    [0,0,0,0,0,0]
  ],
  [
    [0,0,0,0,0,0],
    [0,0,0,0,0,0],
    [0,0,0,0,0,0]
  ]
 ]
```

Vector valued function.

```
  f(x) -> (x^2+1)/(x^2+2);
  f(1+n(5)/5)
[
 0.666666666666666,
 0.709302325581395,
 0.747474747474747,
 0.780701754385964,
 0.809160305343511
]
```

In this case, a function is defined and evaluated at 1, 1.2, 1.4, 1.6, 1.8.

How this relates to looping.

```
x. :=  n(size(myStem.));
```

results in

```
[0,1,2,...]
```

So a common pattern is

```
while[
    for_keys(j, x.)
  ]do[
    myStem.j := // other stuff
    // myStem.x.j is an equivalent reference.
];
```

Example: Looping over scalars and stems

A common issue is the you may have a stem some of whose elements are scalars and some are stems. writing a loop seems to require that you have knowledge about every index. This is not needed with `for_keys` since the keys are retrieved. This is especially useful in lists. IN this example, there is a stem, `a.` some of whose entries are scalars (strings) and some are 3 element random integers. Here is how to loop over the elements:

```
  while[for_keys(j,a.)]do[say(a.j);]
UiVih_S-Act_0mclp7i0CQ
[8528928954721892791, -9103201823511602727, -84528241049547068 54]
XkiEEZ1g297TfZY3ItjL5w
[5095335425002685458, 380662481390939243, -2302353563657105155]
FUKe27vv56w8-lahY2InNA
```

print

Description

This will take the stem (or list) and format it in columns according to

various parameters which may be specified in the format. argument.

Usage

display(arg.{,format. | width})

Arguments

`arg.` - stem to be formatted

`format.` - stem of formatting parameters

`width` - the total width of the display

Table of values for format.

Name	Default	Description
width	-11	The total width the output should fit in. The values may be wrapped as needed. A value of -1 means the line length is infinite.
keys	--	list of keys that are a subset to be printed
sort	true	Display has keys sorted
short	false	Restrict the value displayed to a single line within the given width.This is useful for large values to get a quick overview
string_output	true	If true, then the output is a single formatted string. If false, the output is a stem of formatted lines.
indent	0	How much to indent the entire output from the left

Output

Output is columnar and of the form

key0 : value0

key1 : value1...

Where the values may span multiple lines. The short option will truncate output to fit on a single line, with possible trailing … to show truncation.

Caveat: This is intended for displaying information in tabular format for reports, e.g. and is not a general purpose stem formatting function. You would format individual stems with this. The major reason is that a truly complex stem might end up being simply enormous. Making if format means controlling space and nested stems quickly exhaust every display device. This function, however, gives a great building block to make your own specific stem viewer.

Examples

```
  display({'a':'woof', 'fnord':'warf'})
    a : woof
fnord : warf
```

```
  print(pi()^[;7])
0 : 1
1 : 3.14159265358979
2 : 9.86960440108934
3 : 31.0062766802997
4 : 97.4090910340020
5 : 306.019684785280
6 : 961.389193575298

   print({'first':random_string(64), 'second':random_string(64)}, {'width':50})
 first : WIzEQR3-Og0i9c-3mh-LuQAfx2docUkTF3MOHVwXi
         v-QcASqN-YOrRAdEbpEErAfFy9rPEeruPEPK1zIW0
         F4vA
second : zLO5uMm2Vqt9vXIekmEw-_3BzSxDFz5Tbh8dPrBx2
         5I55ncS8rcvfSzYObymSB-tiAJ9gLjHCN6QRCJVdn
         OSBg
```

Here long strings are formatted with a total display width of 50 characters. To show a

short version (restricting the value to a single line and truncating it if needed),

set the short flag to true:

```
  print({'first':random_string(64), 'second':random_string(64)}, {'short':true,
'width':50})
    first : FLYj-L8HA2PUfuyw9Z09qdQOaE7x7w3BjzSPQTFiilF2...
   second : ydViwpcAO2mNCkAjvepnOlYAzvavAtVn2LVp3ofDAs4t...
```

Note that this only works at the top level and embedded stems are

printed in their `to_string` format.

```
  print({'p':{'a':'x','b':'y'},'q':{'c':'z','d':'w'}})
p : {a:x, b:y}
q : {c:z, d:w}
```

query

Description

Query a stem using the JSON Path language. This is found in the JSON Path specification. In large and very complex stems, it is sometimes necessary to search the stem. JSON Path is a very clean way to do this and works extremely well with QDL.

Usage

```
query(arg., query_string{, return_indices})
```

Arguments

`arg.` - the stem that is the object of the search.

`query_string` - A JSON Path query. The specification is fully supported

`return_indices` - (optional) boolean to return the indices only, no results. Default is **false**.

Output

A stem either of the results themselves or, optionally, the indices where the results reside.

Examples

A very, very simple minded example is here.

```
   a. := {'p':'x', 'q':'y', 'r':5, 's':[2,4,6], 't':{'m':true,'n':345.345}}
   query(a., '$..m')
[true]
   ndx. := query(a., '$..m',true)
  ndx.
[[t,m]]
   a.ndx.0; // same as a.t.m
true
// Alternately, to just change the found indices into a simple list, use remap
   remap(a., ndx.)
[true]
```

So in this case we have a simple example. The query uses **`$..`** which tells it to simply start searching until it hits a key of **`m`** someplace. The first query just returns the result there – always a list with a single value. The second query with the optional flag set to true returns the index for the value, here **`[t,m]`**. An example of accessing the value of the stem using the index is shown. More compactly,

```
   a.query(a., '$..m',true).0
true
```

rank

Description

Get the rank of a stem, i.e., the number of independent axes.

Usage

```
rank(arg.)
```

Arguments

`arg` - a list stem or scalar.

Output

The rank, which is equal to the size of the dimension vector. If this is a `arg` scalar, the result is 0.

Example

```
     rank([;5])
1
```

Since there is one independent axis for a vector.

remap

Description

Usage

```
remap(source., indices.);
remap(source., old_indices., new_indices);
```

Arguments

Dyadic case:

`source.` = the stem list to take a subset of

`indices.` = stem that determines which elements to take

Note that the contract is

```
out. := remap(source., indices.);
```

and indices. is of the form

```
indices. := {k_0:v_0, k_1:v_1, . . ., k_n:v_n}
```

so that the result fulfills

```
out.k_j := source.v_j
```

A very useful function to learn about is `m_indices` in the extensions module, which makes it extremely easy to create multi-indices for `remap`.

Triadic case:

`source.` = the stem list to take a subset of

`old_indices.` = stem that determines which elements to take

`new_indices.` = stem that determines the indices of the result

Note that the contract is

```
out. := remap(source., old_indices., new_indices.);
```

```
old_indices. := {k_0:v_0,. . . , k_n:v_n}
new_indices. := {k_0:r_0,. . . , k_n:r_n}
out.r_j == source.v_j
```

Output

A stem with values remapped by the argument(s). This does not alter `source.`

Examples

Subset can also be used with higher rank stems. Remember that for a stem a.,

```
   a.[p,q] == a.p.q
```

This lets you select like so:

```
   a. := n(3,4,n(12))
 remap(a., [[0,1],[1,1],[1,1],[1,1],[2,3]])
[1,5,5,5,11]
```

Example of creating another array from a given array.

This function returns a more or less linear set. Rather than have some extremely complex way to specify what the resulting shape is going to be, you should take the result of this and use it with, other functions to get what you want.

```
   b. := n(4,5, 3*n(20)-7)
   b.
[
 [-7,-4,-1,2,5],
 [8,11,14,17,20],
```

```
 [23,26,29,32,35],
 [38,41,44,47,50]
]
   n(2,2, remap(b., [[1,1],[3,2],[1,-1],[3,4]]))
[
 [11,44],
 [20,50]]
```

Finally, remember that this works on the zeroth axis unless otherwise specified.

```
  remap(b., [1,2])
[
  [8,11,14,17,20],
  [23,26,29,32,35]
 ]
```

Examples comparing integer and stem addressing

```
  a. := [;16]*3 -8
  a.
[-8,-5,-2,1,4,7,10,13,16,19,22,25,28,31,34,37]

  remap(a., [4;8])
[4,7,10,13]

  remap(a., 4, 8)
[4,7,10,13,16,19,22,25]
  remap(a., [-3;4])

[31,34,37,-8,-5,-2,1]

  b. :=   (3*[;7]-5)~{'a':42, 'b':43, 'woof':44}
  remap(b.,'woof'~[2,5,6]~'a')
[44,1,10,13,42]

  remap(b., list_keys(b.)); // linearize a stem to a list
[-5,-2,1,4,7,10,13,42,43,44]
```

Final example: Completely remapping the elements

Let us say we wanted to create the transpose of an n x m stem, `a.` if the transpose is `t.` then

`t.i.j := a.j.i`

This just means to swap (i.e. reverse) the order of the indices.

```
    a. := n(3,5,n(15))
    a.
[
  [0,1,2,3,4],
  [5,6,7,8,9],
  [10,11,12,13,14]
 ]

    old. := indices(a.-1)
    new. := for_each(@reverse,  old.)
    b. := remap(a., old., new.)
    b.
[
```

```
  [0,5,10],
  [1,6,11],
  [2,7,12],
  [3,8,13],
  [4,9,14]
 ]
```

Example. Higher dimension re-ordering of the axes

List us say we wanted to change the a 3 x 4 x 5 stem to a 5 x 3 x 4, so that the permutation of the ***axes*** is [2,0,1]. This can be done very simply as follows.

```
  w. := n(3,4,5, n(60)); // original stem
  old. := indices(w., -1); // last axis is always complete set of indices
  new. := for_each(@shuffle, old., [[2,0,1]])
  z. := remap(w.,old., new.)
```

To compare, the first bits of w. and z. are

```
  w.
[
  [
    [0,1,2,3,4],
    [5,6,7,8,9],
    [10,11,12,13,14],
    [15,16,17,18,19]
  ], . . .
```

and

```
   z.
[
  [
    [0,5,10,15],
    [20,25,30,35],
    [40,45,50,55]
  ], . . .
```

Note again that

```
z.i.j.k == w.k.i.j
```

reduce (⊙)

Description

Apply a dyadic function pairwise to each member of a list, returning the final output only. This operates on the zero-th axis by default. This may be applied to sets and generic stems

Usage

```
reduce(@f, arg)
reduce(@f(), list.{,axis})
```

Arguments

`@f()` - reference to a function or operators

First form:

`arg -` A generic stem (so has non-list entries), a set or trivially a scalar.

Second form:

`list.` - the list to be operated upon. Since list reduce implies ordering, a list is needed rather than a general stem (which has no canonical ordering).

`axis -` (optional) which (signed) axis. Default is 0. If added to the first form, it is ignored.

Output

A scalar in all cases.

Examples

```
   reduce(@*, {1,2,3,4,5}
120
```

This applies multiplication to every element in a set. Similarly, we can apply a dyadic operator to every member of a generic stem:

```
   reduce(@*, {'a':2, 'b':4, 'c':6})
48
```

Once more, this will apply the operator to *every* entry in a set or stem. Do be careful of non-commutative functions since there is no control of the order in which entries are processed.

Is a list equal to itself?

```
   reduce(@&&, n(5) == n(5))
true
```

This is equivalent to applying the and operator, **&&** between each element of the argument. In this case, the result is **true** if and only if each element of the list is **true**. See also the **expand** function which returns the list of intermediate results.

In higher dimension stems, you can use the axis function to specify a different axis. For instance, if

```
   say(x. :=[[2,3],[-1,-1]], true)
[
  [2,3],
  [-1,-1]
]
```

if you apply reduce, this adds to elements together:

```
   reduce(@+, x.)
```

```
[1,2]
```

which is the same as issuing

```
   reduce(@+, transpose(x.. 0))
```

If you specify the last axis (which are the columns) then the addition is column is added:

```
   reduce(@+, transpose(x., -1))
[5,-2]
```

Example on a generic stem.

If we have the following stem

```
a.'foo' := 'abctest0';
a.'bar' := 'abctest1';
a.'baz' := 'deftest2';
a.'fnord' := 'abdtest3';
```

How can we check if any of these start with abc? You should find out which satisfy your requirement like this:

```
  -1 < index_of(a., 'abc')
{bar:[true], foo:[true], fnord:[false], baz:[false]}
```

Are any true?

```
   reduce(@||, -1 < index_of(a., 'abc'))
true
```

rename_keys

Description

Rename the keys in a stem. See also the *keys* command.

Usage

```
rename_keys(target., new_keys.{, overwrite});
```

Arguments

`target.` - the stem to be altered

`new_keys.` - a **stem** of keys which are a subset of those in target. The values are the new keys.

`overwrite` - (optional) a flag to force overwriting existing entries. Default is `false`.

Output

This alters the target. and returns it.

Examples

```
  a.foo := 42;
  a.bar := 43;
  a.baz := 44;
  key_list.foo := 'a';
  key_list.bar := 'b';
  key_list.baz := 'woof';
  rename_keys(a., key_list.)
{a:42, b:43, woof:44}
  say(a.);
{a:42, b:43, woof:44}
```

Note that since this changes the keys in the target., the key_list. Unrecognized keys are skipped. A subset of keys to rename is fine too:

```
  a.foo := 42;
  a.bar := 43;
  a.baz := 44;
  key_list.foo := 'a';
  key_list.bar := 'b';
  rename_keys(a., key_list.)
  say(a.);
{a:42, b:43, baz:44}
```

Another example. A common case is that all of the keys in a stem need some transformation. The `keys` function will get a stem of the form `{key0:key0, key1:key1, . . . }` and you make then easily apply changes to that. QDL works easily on the *values* of a stem (e.g. `a. + 42` only alters the values, not the keys), so `rename_keys` allows you to modify the keys. The `keys` function promotes the keys to something you can operate on like any other value.

```
    a.SAML_foo := 'a';
    a.SAML_bar := 'b';
    a.SAML_baz := 'c';
    a.fnord := 'd';
    rename_keys(a., keys(a.)-'SAML_');
{
 bar:b,
 foo:a,
 fnord:d,
 baz:c
}
```

Note that `a.fnord` is not effected by this change.

Example. Overwriting values on the rename

This requires a flag to do this.

```
    c.'x':='X'; c.x_y := 'Y';
    ndx. := {'x_y':'x'};
    rename_keys(c., ndx.);
{x:Y}
```

Here the value of `c.x == c.'x':='X'` is overwritten on the rename.

set_default

Description

Set the default value for a stem. If a key is requested but has not been set, the default value is returned. This allows you initialize a stem without having to explicitly fill in every value. Note especially that the default value is not figured in to other calculations, such as listing keys.

Usage

```
set_default(target., scalar | stem.);
```

Arguments

`target.` – the stem

`scalar | stem.` – the default value

Output

This returns the previous default value set for `target.` or `null`.

Example. Turning off subsetting

Let us say that we have a stem, a. and wish to naively add another stem to each element:

```
a. := [[0,0],[0,1],[0,2],[2,0],[2,1],[2,2]]
b. := [2,3]
```

Simply adding a. + b. yields

```
   a. + b.
[[2,2],[3,4]]
```

(One issue is conformability of stems, since this is `[a.0 + b.0, a.1+b.1]`). Rather than fill up an entire stem with copies of b., just set it as a default:

```
b.:={*:[2,3]}
  a.+b.
[[2,3],[2,4],[2,5],[4,3],[4,4],[4,5]]
```

Default values means you do not have to know anything about the structure of the other stem.

Example. Setting the default

There are equivalent ways of setting the default

```
   set_default(x., 42)
   x. := x. ~ {*:42}
```

Example. Setting the default does not alter the keys

```
  set_default(x., 1);
  say(x.);
[]
```

So no values have been defined. Let's set one and check it:

```
  x.0 := 10;
  say(x.);
[10]
```

And if we needed to access a value of x. that has not been set

```
  say(x.1);
1
```

Just to emphasize, default values are not used in most stem operations.

```
  say(get_keys(x.));
[0]
```

shuffle

Description

Permute, i.*e.*shuffle a stem, given a complete list of its keys. Note especially that an incomplete list of keys will fail.

Usage

```
shuffle([int] | [source., permutation.])
```

Arguments

`int` = a positive integer

`source.` = the stem to be shuffled

`permutation.` = a *list* of keys for source. These give the new value of the indices.

Output

A stem consisting of shuffled elements. If the argument is an integer, then the returned output is a list [0,1,…,*n*-1] that has been randomly permuted. If the arguments are a pair of stems, the result is the first argument permuted according to the second.

Example: Making a permutation

```
   shuffle(5)
[2,4,3,0,1]
```

This creates a list of integers and then arranges them in random order.

Example: Permuting the elements directly

In this example, we permute the elements of a vector

```
  q.:= 10+3*[;5]
  q.
[10,13,16,19,22]
  shuffle(q., [4,2,3,1,0])
[22,16,19,13,10]
```

How to read this[1]?

```
/                       \
|0    1     2     3     4|
|4    2     3     1     0|
\                       /
```

So the top row are the indices in the vector, the bottom row is the new value.

so old index 0 → new index 4, old index 1 → new index 2, etc. This works generally with stems too.

```
    a.p:='foo';a.q:='bar';a.r:='baz';a.0:=10;a.1:=15;
    b.q :='r';b.0:='q';b.1:=0;b.p:=1;b.r:='p';
  a.
{0:10, 1:15, p:foo, q:bar, r:baz}
  b.
{0:q, 1:0, p:1, q:r, r:p}
  shuffle(a., b.);
{0:bar, 1:10, p:15, q:baz, r:foo}
```

1 OK, I'll confess. This is from Abstract Algebra and referred to as cycle notation. QDL generalizes the indices as it is wont to do.

size

Description

Return the size of the argument

Usage

```
size(var)
```

Arguments

`var` – any variable or argument

Output

This varies.

- stem – the number of keys (this does not check if there are stems as values)
- string – the length of the string
- boolean, integer, decimal – zero, since these are scalars.

Examples

```
    size(42)
0
    size('abcd')
4
    size([;10])
10
```

star (* in extractions)

Description

A functional analog of the wildcard, *, used with extraction operator. This allows for creating expressions on the fly as regular lists.

Usage

`star()`

Arguments

None

Output

None

Examples

in extractions. That assumes the full set of indices in context.

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

This is most useful if you are constructing an argument for extraction, e.g.

```
  a.:=n(3,4,n(12))
  a\>(1~(size(a.)<4?star():[2,4]))
[4,5,6,7]
```

Note that this creates (in this case) the expression a\>[1,*] meaning go to the first element, return everything. Compare with say

```
   a\>[2,[1,3]]
[9,11]
```

to_json

Description

Convert a stem to a JSON string. Note that JSON = JavaScript Object Notation is a common way to represent objects and is treated as a notation, not a data structure. See the extended note in the `from_json` section. Not every stem can be converted to a JSON object. For instance, stems allow for cycles which JSON cannot resolve. Also, nulls in QDL are rendered as a reserved string since there is no analog of them.

Usage

```
to_json(stem.{,indent, convert_type})
```

Arguments

`stem.` = the stem to represent in JSON notation. Unlike **`from_json`**, this will convert the entire stem to JSON, not just the elements of the stem. If you really need to convert elements either loop or use **`for_each.`**

`indent` (optional) = whether or not to indent the resulting string to make it more readable. This controls how much whitespace is added. The higher the number, the more space in the result. Usually a value of 1 or 2 is sufficient for most cases.

`convert_type` - the type of decode to use on the keys,

So these are valid calls

`to_json(stem.)` – do not convert the names

`to_json(stem. 2)` – indent the output with a spacing of 2

`to_json(stem., 2, 32)` – convert so that the spacing is 2 and the keys, which were encoded in base 32, are properly decoded.

Output

A string in JSON which represents the argument.

Examples

```
  a. := [;3]
  a.woof := 'arf'
  to_json(a.)
{"woof":"arf","0":0,"1":1,"2":2}
  // and just to show how to indent the result
  to_json(a.,1)
{
 "woof": "arf",
 "0": 0,
 "1": 1,
 "2": 2
}
```

Large JSON objects are often best handled through files or other means rather than directly.

```
  claims. := from_json(read_file('/tmp/claims.json'));
  size(claims.)
137
```

An example where you convert a stem to a JSON object but do not want the variables converted with *decode:*

```
  a.$a := 2;
  a.$b := 3;
  to_json(a., false)
{"$a":2,"$b":3}
```

So the names of the variables are turned in to JSON unaltered.

Again, JSON is a notation for an object and you must know what the structure of the object is and all the particulars about it to do anything useful with it.

transpose, (⍉)

Description

Take a stem (of higher dimension) and transpose the dimensions. This permits you to restructure stems as needed. A simple case yeilds the transpose of a matrix and the operator ⍉ is chosen to show the axis about which the transpose happens.

Usage

```
transpose(arg.{, a | p.})
{a|p.}⍉arg.
```

Arguments

`arg.` - the stem to operate upon

`a` - (optional, integer) the axis. This may be signed, so `a := -1` would operate on the last axis of the `arg.`, whatever that is. Signed axes means you don't need to know the structure of the argument ahead of time.

`p.` - (optional, simple list) a permutation of the dimensions. A partial list of indices is interpreted as

```
p. ~ ~exclude_keys([0,1,…, rank-1], p.)
```

For instance, if you have a massive stem, x. with

```
dim(x.) == [4,2,6,5,8,7,9,11,15,6]

           0 1 2 3 4 5 6  7  8 9  << -- indices of the dimensions
```

and

```
p. := [3,7,1]
```

then the resulting permutation of x. would be

```
[3,7,1,0,2,4,5,6,8,9]
```

and

```
   dim(x., p.)
[5,11,2,4,6,8,7,9,15,6]
```

Finally, remember that this works on the zeroth axis unless otherwise specified so if `a.` is an n×m array, `transpose(a.)` is the standard matrix transpose m×n array, hence the name.

Examples comparing integer and stem addressing

```
  a. := [;16]*3 -8
  a.
[-8,-5,-2,1,4,7,10,13,16,19,22,25,28,31,34,37]

  remap(a., [4;8])
[4,7,10,13]
  // compare with the subset function (since a. is a simple list)
  sublist(a., 4, 8)
[4,7,10,13,16,19,22,25]

  remap(a., [-3;4])
[31,34,37,-8,-5,-2,1]

  b. :=   (3*[;7]-5)~{'a':42, 'b':43, 'woof':44}
  remap(b.,'woof'~[2,5,6]~'a')
[44,1,10,13,42]
```

```
  remap(b., list_keys(b.)); // linearize a stem to a list
[-5,-2,1,4,7,10,13,42,43,44]
```

Final example: Completely remapping the elements

Let us say we wanted to create the transpose of an n x m stem, `a.` if the transpose is `t.` then

`t.i.j := a.j.i`

This just means to swap (i.e. reverse) the order of the indices.

```
    a. := n(3,5,n(15))
    a.
[
  [0,1,2,3,4],
  [5,6,7,8,9],
  [10,11,12,13,14]
 ]

    old. := indices(a.-1)
    new. := for_each(@reverse,  old.)
    b. := remap(a., old., new.)
    b.
[
  [0,5,10],
  [1,6,11],
  [2,7,12],
  [3,8,13],
  [4,9,14]
 ]
```

(And, yes, `transpose(a.)` would do this.)

Example. Higher dimension re-ordering of the axes

List us say we wanted to change the a 3 x 4 x 5 stem to a 5 x 3 x 4, so that the permutation of the ***axes*** is [2,0,1]. This can be done very simply as follows.

```
  w. := n(3,4,5, n(60)); // original stem
  old. := indices(w., -1); // last axis is always complete set of indices
  new. := for_each(@shuffle, old., [[2,0,1]])
  z. := remap(w.,new., old.)
```

To compare, the first bits of w. and z. are

```
   z.
[
  [
    [0,1,2,3,4],
    [5,6,7,8,9], . . .
```

and

```
w.
[
  [
    [0,5,10,15],
    [20,25,30,35],
```

```
    [40,45,50,55], . . .
```

and

z.i.j.k == w.k.i.j

Output

The restructured stem. Of course, `arg.` is never altered.

Note: You really don't need to obsess over what the result is. The usual usage is that you are simply describing what part of the data should be acted upon and you need never need to gaze upon the output. There is an excellent argument that this should be made into an operator. However, it is much more flexible to have it as a function. If you wish to be terse, use the operator ⦸ (\u29b0) for this, e.g.

```
-1⦸arg.
```

Examples

The next example is shorter than it looks. Just notice the patterns of how the data moves

```
  a. := n(2,3,4, 10+n(24))
  a.
[
  [
    [10,11,12,13],
    [14,15,16,17],
    [18,19,20,21]
  ],
  [
    [22,23,24,25],
    [26,27,28,29],
    [30,31,32,33]
  ]
 ]
  // The original is always the same as transpose(a., 0) == a.
   transpose(a., 1); // glom the rows together, dim is 3×2×4
[
  [
    [10,11,12,13],
    [22,23,24,25]
  ],
  [
    [14,15,16,17],
    [26,27,28,29]
  ],
  [
    [18,19,20,21],
    [30,31,32,33]
  ]
 ]
  transpose(a., 2); // glom the columns together, dim is 4×2×3
[
  [
    [10,14,18],
    [22,26,30]
```

```
  ],
  [
    [11,15,19],
    [23,27,31]
  ],
  [
    [12,16,20],
    [24,28,32]
  ],
  [
    [13,17,21],
    [25,29,33]
  ]
 ]
```

Now for an example of a complete remapping of the indices. The stem has indices i, j, k and the right argument says that the output, call it `output`. satisfies

```
       output.j.k.i := a.i.j.k
```

This looks like

```
     transpose(a., [1,2,0]) // dim is 3×4×2
[
  [
    [10,22],
    [11,23],
    [12,24],
    [13,25]
  ],
  [
    [14,26],
    [15,27],
    [16,28],
    [17,29]
  ],
  [
    [18,30],
    [19,31],
    [20,32],
    [21,33]
  ]
 ]
```

Reducing things along axes.

So axis 0 are boxes, axis 1 is rows and axis 2 (last axis) is the columns. If you wanted to reduce down the columns, you'd issue (here -1 for the axis has the same effect as 2 for the axis)

```
  reduce(@+, reduce(a., -1))
[
  [46,62,78],
  [94,110,126]
 ]
```

Note that the shape of a. is 2×3×4 and summing along the last axis gets rid of it, so that the final shape of the reduced answer is 2×3. If you wanted to sum the rows together, you'd issue

```
  reduce(@+, transpose(a., 1))
[
  [42,45,48,51],
  [78,81,84,87]
 ]
```

The rows are added together and the 2×3×4 stem is reduced to 2×4

The standard operation for all built in functions is to operate on the zero-th axis, so

```
  reduce(@+, a.)
[
  [32,34,36,38],
  [40,42,44,46],
  [48,50,52,54]
 ]
```

Yields a 3×4 from the original 2×3×4 stem, adding the boxes together.

Example. Matrix multiplication

In this example we are going to show how to use the axis function to multiply two matrices. In general, to multiple an n×m and m×n matrix dimensions must match. Our two matrices are

```
    say(x. := [[1,2,3],[4,5,6]], true)
[
  [1,2,3],
  [4,5,6]
 ]
    say(y. := [[10,11],[20,21],[30,31]], true)
[
  [10,11],
  [20,21],
  [30,31]
 ]
```

This done by multiplying the rows of the first matrix by the columns of the second, then summing. In QDL, you'd do this as

```
z. := for_each(@*, x., transpose(y.-1))
```

which is a 2×2×3 stem. For the summation, do it along the last axis:

```
    reduce(@+, tranpose(z., -1))
[
  [140,146],
  [320,335]
 ]
```

All together, the complete program to multiply two matrices of dim n×m and m×n is a single line:

```
mm(x., y.) → reduce(@+, transpose(for_each(@*, x., transpose(y., -1)) ,-1));
```

This function is defined in the mathx module.

unbox

Description

Takes a stem variable and splits it up, turning each key in to a variable.

Usage

```
unbox(stem.{, safe_mode_on});
```

Arguments

`stem.` - the stem to unbox

`safe_mode_on` - (optional) a boolean which when *true* (default) will not overwrite variables in the current workspace and when *false* will. Note that this is an all or nothing proposition: `safe_mode_on = true` means that nothing will get processed if there is a clash.

Output

A `true` if the result worked.

Examples

```
  a. := [-5;0];
  b. := [5;10];
  c. := box(a., b.);
    )vars
c.
  unbox(c.);
  )vars
a., b.
```

union

Description

Take a set of stems and put them all together in to a single stem

Usage

```
union(stem1., stem2., ,,,);
```

Arguments

The arguments are either stems or variables that point to stems.

Output

The output is a new stem that contains all of the keys. Note that if there are multiple keys then the *last* argument with that key is what is set. The result is guaranteed to have every key in all the arguments in it. See also `join`.

Examples

```
  a. := -5 + [;10];
  b. := 5 + [;5];
  a.arf := 'woof';
  b.woof := 'bow wow';
  c. := -20 + [;3];
  union(a., b., c.)
[-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,-20,-19,-18]~{
 arf:woof,
 woof:bow wow
}
```

Note 1: that in this example these stems have some keys contained in the previous one.

Note 2: Lists are appended to the *end* of the current list.

*Example.*Compare with ~ operator

```
  p.6 := 100;
  p.~[1,2,3]
{
 6:100,
 7:1,
 8:2,
 9:3
}
   union(p., [1,2,3])
[100,1,2,3]
```

In this case, there was one entry with an integer index (6) in p., appending the list tacks it on to the end of the current list rather than overwriting elements.

values

Description

Return the set of values in a stem.

Usages

values(arg)

Arguments

arg = The argument. This may be a stem or scalar. If a scalar, then the value itself wrapped in a set is returned.

Output

A set of the unique values. Note that this will look through every set for a value.

Examples

In this example, a couple of lists are

```
  values(['a',2,4,true]~['a','b',0,3,true])
{0,a,2,b,3,4,true}
```

Another example, showing that this onlvaly applies to lists, not whole stems:

```
    values({'p':'q'}~{'p':'r'}~(2*[;3])~(2+[;4]))
{0,2,3,4,5}
```

Input and output

dir

Description

List a directory content

Usage

```
dir(arg)
```

Arguments

`arg` - the path to a directory. It may also be in a virtual file system

Output

A stem list of the elements of the directory

Examples

```
  dir('qdl-vfs#/zip/root')
[scripts/, other/, readme.txt]
```

This lists the given directory in the mounted VFS. Note that in this case it so happens to be a zip archive of a file, mounted at `qdl-vfs#/zip/`.

mkdir

Description

Make a set of directories in a file system

Usage

mkdir(arg)

Arguments

`arg` -- a path. All of the intermediate paths will be created as needed.

Output

A boolean, true if the operation succeeded and false otherwise.

Examples

An example of trying to make a directory in a read-only VFS will fail:

```
  mkdir('qdl-vfs#/zip/foo')
Error: You do not have permissions make directories in the virtual file system
```

Making a directory in a system that is writeable works fine:

```
  mkdir('qdl-vfs#/pt/woof-123')
true
```

rmdir

Description

Remove an empty directory from a file system.

Usage

```
rmdir(arg)
```

Arguments

`arg` – a path in a file system to an empty directory. You must remove all files and sub-directories for this to work. Also, unlike *mkdir*, this will only remove the last component.

Output

A true if this succeeded and a false otherwise.

Examples

rm

Description

Remove a single file from a directory.

Usage

```
rm(arg)
```

Arguments

`arg` -- the full path to the file

Output

A true if this succeeded, false otherwise

print, say

Description

Print out the argument to the console. The two functions *say* and *print* are synonyms. Use whichever you prefer for readability.

Usage

```
say(arg {,prettyPrintForStems})
```

Arguments

`arg` – anything.

`prettyPrintForStems` (optional) -***IF*** arg is a stem, try to print a pretty version of it, defined as being more vertical.

Output

The printed representation of the argument will be put to the console **and** the value returned is whatever was printed (so you can embed it in other statements – a very useful debugging trick.)

Examples

The momentous entire “Hello World” program in QDL:

```
   say('Hello World');
Hello World
```

And since 42 is the answer to all Life's questions (as per the Hitchhiker's Guide To The Galaxy)

```
    say(42);
42
```

Here is an example of how to use this to intercept and print out an intermediate result,

```
    a := say(432 + 15);
447
```

Pretty print only applies to stems. It attempts to make a somewhat more human readable version

```
  f. := [;6];
  say(f., true);
[0,1,2,3,4,5]
```

file_read

Description

Read a file. The result is always a string

Usage

```
file_read(files.{, types.})
file _read(file_name {, as_string | to_list | is_binary | is_ini})
```

Arguments

If the first (stem) form,

`files.` = stem of full file paths to read

`types.` = stem with corresponding entries of files with their type. Missing entries use default type.

If the second (scalar) form

`file_name` – the full path to the file. This may be in a virtual file system too.

The next argument is optional and is an integer

`as_string` = -1 – return the contents of the file as one long string (default).

`is_binary` = 0 – return the result as a base 64 encoded string of bytes.

`to_list` = 1 – return the result as a stem list each line separate

`is_ini` = 2 - parse the file as a QDL initialization file.

If no second argument is given, the result is simply a string of the entire contents of the file. Note that these constants are available via **`constants()`** as `file_types.`

Output

Stem form: A stem where each entry has a the content of the file a its value.

Scalar form: Either a simple string (only file name is given), a stem if it is flagged as a list or ini file (see separate documentation for how ini files work) or a base64 string if it is flagged as binary.

Examples

```
cfg. := file_read('/var/lib/tomcat/conf/server.xml', 1);
```

Would read in the file /var/lib/tomcat/conf/server.xml and return a stem. Each line in the file is in order in *cfg.0*, *cfg.1*, … Compare this with

```
big_string := file_read('/var/lib/tomcat/conf/server.xml');
```

Which reads the same file and puts the entire thing in a single string.

```
my_b64 := file_read('/var/lib/crypto/keystore.jks' , 0);
```

this reads the keystore.jks file (which is binary) and base64 encodes it, storing it in the *my_b64* variable. QDL does not have the capacity to do low-level operations on binary data, but it can move them where they need to go faithfully.

A couple of more examples:

```
      // read a file as a stem
      say(file_read('/home/ncsa/dev/ncsa-git/security-lib/ncsa-qdl/src/test/
resources/hello_world.qdl',1));
{/*, The expected Hello World program.  ,  Jeff Gaynor,   1/26/2020, */, say('Hello
world!');}

    // read the exact same file and turn the bytes into a base 64 string.
    say(file_read('/home/ncsa/dev/ncsa-git/security-lib/ncsa-qdl/src/test/
resources/hello_world.qdl',0));
LyoKICBUaGUgZXhwZWN0ZWQgSGVsbG8gV29ybGQgcHJvZ3JhbS4gIAogIEplZmYgR2F5bm9yCiAgMS8yNi8
yMDIwCiovCnNheSgnSGVsbG8gd29ybGQhJyk7Cg


say(decode('LyoKICBUaGUgZXhwZWN0ZWQgSGVsbG8gV29ybGQgcHJvZ3JhbS4gIAogIEplZmYgR2F5bm9
yCiAgMS8yNi8yMDIwCiovCnNheSgnSGVsbG8gd29ybGQhJyk7Cg'));
/*
  The expected Hello World program.
  Jeff Gaynor
  1/26/2020
*/
say('Hello world!');
```

(This is the sample hello world program for qdl).

file_write

Description

write contents to a file

Usage

```
file_write(files.)
file_write(file_name, contents{,type})
```

Arguments

If the first (stem) form:

`files.` = a stem of entries to process. Each entry has

key	value
path	The full path to the file
content	Either a string or stem of strings.
type	(optional) either the integer file type or if a boolean, **true** means to base 64 encode it.

Any entry that is not of this form is ignored.

If the second (scalar) form:

`file_name` – the name of the file.

`contents` = A string or a stem list. If the stem is not a list (so indices 0, 1, …) then this will fail.

`type` (optional) – an integer of one of the following

`as_string` = -1 – treat the contents of the file as one long string (default).

`is_binary` = 0 – treat the contents as a base 64 encoded string of bytes and decode to binary.

`to_list` = 1 – treat the contents as a stem list each line separate

`is_ini` = 2 - treat the contents as an ini file and write it in that format.

If you omit the type, it is assumed that the contents should be treated as text.

Output

Returns **true** if this succeeded.

Examples

```
          hello_world :=
'LyoKICBUaGUgZXhwZWN0ZWQgSGVsbG8gV29ybGQgcHJvZ3JhbS4gIAogIEplZmYgR2F5bm9yCiAgMS8yNi
8yMDIwCiovCnNheSgnSGVsbG8gd29ybGQhJyk7Cg';
    file_write('/tmp/test.qdl', hello_world, 0);
```

This is just the base64 encoded hello_world.qdl script from the read_file example. The argument of **0** says it is base 64 encoded and to decode it to the file. In this example, you can go check it just decodes to the Hello world program.

The same example in stem form

```
   file_write([{'path':'/tmp/test.qdl', 'content':hello_world, 'type':0}]);
[true]
```

This writes a list of files (with a single entry) and returns a stem with the same shape, *i.e.* a list that contains if the operation worked.

scan

Description

Prompt a user for input

Usage

```
scan([prompt])
```

Arguments

`prompt` (optional) – something to print out, probably cuing the user.

Output

Whatever the user types in as a string. There is no end of line marker returned.

Examples

```
    response := scan('do you want to continue?(y/n):');
do you want to continue?(y/n):y
    say(response);
y
```

So the user sees the prompt (in this case “do you want to continue?(y/n):”) and types in the response of “y”, which is stored in the variable *response*. In this next example we input a loop in buffer mode, then execute it. (This assumes that local buffering is on in the workspace so you can use the)edit command).

```
)edit
edit> i
stop_looping := 'n';
while[
      stop_looping != 'y'
   ]do[
      stop_looping := scan('stop looping? (y/n):');
 ]; //end do
.
edit>q
stop looping? (y/n):foo
stop looping? (y/n):bar
stop looping? (y/n):y
```

Only when we enter the expected response of “y” does it stop.

vfs_mount

Description

Mount a virtual file system

Usage

```
vfs_mount(cfg.)
```

Arguments

`cfg.` = a stem that contains the configuration for this type.

`permissions` (optional) = the permissions the VFS has. These are 'r' for read and 'w' for write. If omitted, the VFS is mounted in read-only mode.

Required entries for the following types

`type` = the type of virtual file system. Allowed values are

```
  pass_through
  mysql
  memory
  zip
```

`scheme` = the scheme (label) for this system

`mount_point` = the internal path (starts with a /) for programs to refer to.

`access =` (optional) the permissions, 'r' for readable, 'w' for writeable or 'rw' for both. Omitting this mounts the VFS in read-only mode.

Here are the supported other parameters by type.

memory

No other parameters are required.

Example

```
cfg.type :='memory';
cfg.scheme := 'ram-disk';
cfg.mount_point := '/vfs/cache';
cfg.access := 'rw';
vfs_mount(cfg.);
```

This would create a memory store mounted at /vfs/cache and accessible with the prefix ram-disk, e.g.

```
read_file('ram-disk#/vfs/cache/bigfile.txt);
```

pass_through

`root_dir` = The directory that servers as the root for this VFS. All files and directories will be created under this

zip

`zip_file` = the absolute path to the zip file that will be mounted. All zip-based VFS are read only.

mysql

This has a lot of parameters for connecting to a database

Output

A 0 if there was no problem.

Examples

In this example, we will mount a local file system and read a file. We mount the VFS for both reads and writes. You refer to a file in the vfs seamlessly using the scheme to prefix it.

```
  cfg.type :='pass_through';
  cfg.root := '/home/ncsa/dev/qdl/scripting';
  cfg.mount_point := '/';
  cfg.scheme := 'qdl-vfs';
  cfg.access:= 'rw';
  vfs_mount(cfg.);
0
   read_file('qdl-vfs#/client.xml')
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>OA4MP stream store</comment>
… lots more
```

Another way of doing the previous example.

You can also just set all the parameters and box them up. This will, or course, remove these from the symbol table.

```
  type :='pass_through';
  root := '/home/ncsa/dev/qdl/scripting';
  mount_point := '/';
  scheme := 'qdl-vfs';
  permissions := 'rw';
  cfg. := box(type,root,mount_point,scheme, permissions);
  vfs_mount(cfg.);
```

So the given file is loaded and read. All file operations behave normally. The reason for virtual file systems is two-fold. First off, if QDL is running in server mode, directories may be mounted in read

only fashion to provide access to libraries, modules and such in a completely installation independent way. Secondly, when QDL is running in server mode, all standard file operations are prohibited but you may still have virtual ones. This allows a server to, for instance, mount a jar file with libraries in it.

(The reason for this on a server is security: Often servers run with enhanced privileges which may be inherited by applications. QDL always seeks to be a good citizen and only allows what is specifically granted to it.)

vfs_unmount

Description

Unmount a virtual file system. Once unmounted, no operations on that file system can be performed. This does nothing to the underlying file system, it just removes the mount point in the current session.

Usage

```
vfs_unmount(mount_point)
```

Arguments

`mount_point` - the scheme delimited mount point. This must be exact or the operation will fail.

Output

A boolean that is true if the operation succeeded. Otherwise an error is raised if, for instance, the mount point is invalid.

Examples

Listing the vfs's in the workspace yields

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

and to unmount `vfs#/ramdisk/` you would issue

```
    vfs_unmount('vfs#/ramdisk/')
true
```

Checking the listed VFS's yields

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

Any operations on `vfs#/ramdisk/` will now fail.

Scripts

A script is simply a sequence of QDL commands in a file. You may run scripts in a variety of ways and these commands let you do it from in the workspace.

args

Description

Get the arguments to the current script as a list. If there is no script, then this is empty.

Usage

```
args({index})
```

Arguments

`script_args([index]none -` return the whole list of arguments.

`index -` (optional) integer for the index desired. Note `args(n) == args().n`

Output

The list of arguments (no index given) or the specific value. Signed indices are allowed since it is exactly a stem.

Examples

A table relating this with the deprecated script_args:

```
Old                     New                     Description
script_args(-1)         args()                  get the arg list
script_args()           size(args());           get the number of args
script_args(n)          args(n) OR args().n     get arg with index n
```

check_syntax

Description

Check a string of QDL for syntax errors. This does not actually execute anything! It will simply check that the given string is valid QDL. Note that there are *syntax errors*, such as not closing a quote vs *runtime errors*, which arise only when the system is tunning because of the state at that point. For instance

```
a := 4/b;
```

would parse fine, but if during execution it turned out that b had a value of 0 (zero) then a runtime error would happen.

Usage

```
check_syntax(string)
```

Arguments

string - a (possibly very long) string of QDL to be checked. This may, for instance, be the entire contents of a script.

Output

Either an empty string (if everything works) or the message from the parser that contains the line and position where the first parsing error happens. The parser will exit as soon as it gets any errors, so there is only one to process.

Examples.

Let us say we had the following, simply electrifying script:

```
/* A test file */
a
  :=
     3;
b = 'foo;
```

in the file `/tmp/foo.qdl` and executed

```
  check_syntax(file_read('/tmp/foo.qdl'))
line 5:2 mismatched input '=' expecting {'^', '=<', '<=', ';', '*', '/', '++', '+',
'--', '-', '<', '>', '<=', '>=', '==', '!=', '&&', '||', '%', '~'}
```

(Note that the line wraps here.) This means that on line 5 (lines are counted starting at 1 so line 5 is the very last line in the file) at position 2 (characters are counted from zero, so the single = there is where parsing stopped. We all remember that QDL only has compound assignment operators, *n'est-ce pas?*) Since QDL could not figure out what to do next, the message is everything it thought might be there. This gives you a place to start looking, but the parser is not a mind-reader. The message is not telling you to stick one of those characters at position 2, but that as near it could determine from the grammar, one of those was *probably* intended.

If you fix the assignment to := then run it (there is still a missing quote) you get

```
  check_syntax(file_read('/tmp/foo.qdl'))
missing/unparseable right-hand expression for assignment
```

Which means that the assignment operator was found and it QDL tried to determine what to assign, but failed (in this case because the file ended before a close quote was found).

Another example

In this case a file is read (line numbers are in the left hand column):

```
37: // 37 lines of stuff
38: if[
39:  exec_phase == 'post_token' && claims.idp == idp.ncsa
40: ][
41:  flow_states.accept_requests = has_value('prj_sprout', claims.isMemberOf.);
42: ];
43: // many more lines of stuff
```

and the following message is displayed:

```
line 41:32 no viable alternative at input
'if[exec_phase=='post_token'&&claims.idp==idp.ncsa][flow_states.accept_requests='
```

This means that line 41 of the file had parsing fail at character 32 (the single = sign). Note that the message has the entire statement (statements end with a ;) up to that point that was being processed so you can see it in context.

interpret

Description

Send a string to the interpreter and evaluate it.

Usage

```
interpret(string | stem.)
```

Arguments

`string` – the string to be interpreted, i.e. run. This must be a valid QDL statement or it will fail. If there is not a final semi-colon, it will be added.

`stem.` - a list stem of strings. These will be executed sequentially.

Output

If the last statement has a result, it will be returned of or a `null` if the last statement has no result.

Examples

```
    execute('2 + 2');
4

    execute('say(\'abc\' + \'def\');');
abcdef
abcdef
```

Here you get two results if the current workspace is set to print results, since you are telling it to say the value and it is returning it.

```
  execute('var:=3')

  var
3
```

Another use of this is to store things as a type of quick serialization for later use:

```
my_stem. := . . . // lots of stuff
file_write(my_file, input_form(my_stem.));
// then later
my_other_stem. := execute(file_read(my_file));
```

fork

Description

Start a given script in its own thread, inheriting the current state. Note that these are not the same as debugging sessions, but are completely separate processes.

Usage

```
fork(path, arg0, arg1, …)
```

Arguments

`path` - the path to the file. The current script path will be searched.

`argn` -arguments to the script.

Output

An integer that is the process id number for this thread. Note that you may use the `kill` function with the pid number to stop the thread if it is active.

Example

```
   fork('vsf#/path/myscript.qdl', false, [|-pi()/2, pi()/2, 11|])
53575
```

This means that the given script vsf#/path/myscript.qdl is executing on the thread with pid 53575. If you want to see all current thread, issue

```
  )si threads
53575  vsf#/path/myscript.qdl
```

And if you want to stop this thread, issue

```
   kill(53575)
1
```

halt

Description

Halt processing of a script at the given line. This is normally used only in a debugging session in the workspace. See the workspace documentation for a treatment of how this is used.

Usage

```
halt([message])
```

Arguments

message - (optional) a message to be displayed in the state indicator.

Output

An integer which is the process identifier (pid). You may use this in the workspace to restart execution, attach to the state and inspect as well as other things.

Example

```
a := 2 + 3;
halt('a was set');
// .. other stuff
```

The effect here is that the script will stop at this point and in the workspace, you might see something like

```
    ) 0 &
11
    )si list
pid | active | line | time                          | size  | message
0   |   *    |      | Mon Oct 26 16:15:20 CDT 2020 | 2965  | system
11  |        | 1    | Mon Oct 26 16:16:06 CDT 2020 | 3001  | a was set
```

This shows that this pid, 11, is active and suspended. The ampersand (&) in the run command tell the workspace to clone its state *in toto* and run the script inside that. See the workspace documentation for a full explanation.

input_form

Description

This will return the input form, *i.e.*, what you would enter at the command line, of a variable, module, function or expression. This is effectively the inverse function for `execute`.

Usage

```
input_form(fq_module_name)
input_form(variable{,pretty_print})
input_form(function, arg_count)
input_form(expression)
```

Arguments

For variables, this is the name, *e.g.* **foo.** For modules, this must be fully qualified like **my_alias#baz#fnord.** If the flag **pretty_print** is **true** then print it out with indenting, otherwise it will end up on a single, possibly very long line. Expressions will be evaluated and the result will be converted to input form.

For modules, it is a string that is either the alias or the main module. Note that you can get the input form for a module that has been imported, but you must load it to print out individual functions (since these can be redefined by you.) For Java defined modules, there is no source, you simply get the class name.

For functions, this is the name plus you must supply the number of arguments. Note that for Java-defined functions, there is no source, you simply get the class name.

Output

A string that can be interpreted to yield the original argument. Note however, that the result is what is needed, but is not identical. The examples should make this clear and why this is a good way to do it. Generally variables have their content returned (since you probably want to work with that) and modules or functions have their complete definitions returned (since you probably want to put it into an editor, e.g.).

Caveat: If you have an alias and a variable with the same name, `input_form` of a single argument will return the module. To get the variable, use the dyadic version with a boolean second argument. Generally though you should not have variables and aliases that are indistinguishable since that can lead to confusion.

Example: A variable

```
a. := [2,4,6]~'a'~234~(-1.23)~false
b := input_form(a.)
b
```

```
[2,4,6,'a',234,-1.23,false]
```

This result is a string.

```
  input_form((543/11)^10)
8.5915484651856E16
  input_form('abc' + substring('pqr',0,10,'.') + ' foo')
'abcpqr..... foo'
```

In this case, the expressions are evaluated and the input form is returned.

Example: using input_form and execute

So how to use this with, say, **execute**? Taking a. as per above:

```
   a. := [2,4,6]~'a'~234~(-1.23)~false
  execute(input_form(a.))
[2,4,6,a,234,-1.23,false]
```

Example:A function

If you define a function such as

```
  f(x)->x^3+3*x^2 +x - 1
```

You can recover the input form as

```
  input_form(f, 1)
f(x)
->
x^3+3*x^2+x-1;
```

Note that this is not exactly what was typed, but is equivalent.

Example: A module

Let us define and import a module

```
    module['a:a','a']body[foo := 'abar';define[f(n)]body[return(n+1);];];
    module_import('a:a');
a
    input_form(a)
module['a:a','a']body[foo := 'abar';define[f(n)]body[return(n+1);];];
```

You may also print out the function definition:

```
  input_form(a#f, 1)
define[
```

```
f(n)
]body[
return(n+1)
;
];
```

Note that this is not quite the same as the original. What always happens is that the source is picked up after the parser reads it (this is how it gets into QDL) and whitespace such as blanks and linefeeds are not considered essential, hence may change.

kill

Description

Stop (aka kill) a forked process.

Usage

```
kill(pid)
```

Arguments

pid - an integer that is the unique process identification number returned from the `fork` function.

Output

There are two possible values. A 1 indicates success, a 0 indicates failure.

Examples

```
   kill(42)
1
```

Stops the process with pid 42. The result indicates that process has been successfully terminated.

script_args

Description

Deprecated. Use args() instead. When a script is invoked, the arguments to it are given as a list of strings. This may be either from the command line *or* an argument to the script_run() or script_load() functions. Typically this is called *inside* a running script to access the arguments passed in.

Usage

script_args([index])

Arguments

-1 = return all args as a stem.

index - (optional) an integer in the proper range.

Output

no arguments - the number of arguments is returned.

integer - the argument for that integer.

Examples.

In the case of invoking a script from the command line,

```
qdl -run my_script.qdl arg0 arg1 arg2
```

The arguments (*e.g.* about the first call inside `my_script.qdl`) would be accessed as

```
    say(script_args()); // how many passed in?
3
    say(script_args(1)); // print out the second one (indices start at zero.)
arg1
```

Note that if the script is invoked from the command line, then only strings will result and may have to be converted to other types, *e.g.* with the `to_number()` call.

Note further that this is not a variable for a specific reason. When calling QDL scripts it is possible to pass along stem variables as part of the argument list. Therefore getting a specific argument may be done and the type of the result checked as needed.

script_load

Description

Read a script from a file and execute it in the current environment. This means that any variables it sets of functions it defines are now part of the active workspace. Caution that this will overwrite whatever you have if there is a name clash.

See also `script_run(), script_args(), script_path()`

Usage

```
script_load(file_name{, arg}*)
```

Arguments

`file_name` – the fully qualified path to the file

`arg0, arg1…` - (optional) the arguments for the script

Output

The output of the script, if any.

Examples

```
script_load('/home/bob/qdl/math_util.qdl', 3,’foo’, false);
```

Will load the given script and send it the 3 arguments listed.

script_run

Description

Read a script from a file and execute it in a completely new environment. The output of the file is piped to the current console.

Usage

```
script_run(file_name{,arg}* )
```

Arguments

`file_name` – the fully qualified path to the file

`arg0, arg1…` - (optional) the arguments for the script

If there is a single argument that is a stem list, then the components of that will be sent to the script as the arguments.

Output

The output of the script, if any.

Examples

```
script_run('/home/bob/qdl/format_reports.qdl');
```

If the script requires command line arguments, you may simply send them along:

```
script_run('/home/bob/qdl/format_reports.qdl', '-w', 120);
```

In this case, it is the same as invoking this script from the command line like so:

```
qdl -run /home/bob/qdl/format_reports.qdl -2 120
```

General functions

These are functions that are generally applicable and do not fall in to the other categories.

cb_exists

Description

Check if the system clipboard is supported. This operation is not available in server mode.

Usage

```
cb_exists()
```

Arguments

none.

Output

A true if the clipboard is readable, false otherwise.

cb_read

Description

Read the contents of the clipboard as a string. This operation is not available in server mode.

Usage

```
cb_read()
```

Arguments

none

Output

The contents of the clipboard as a string. Note that leading and trailing whitespace is removed. The reason for this is that applications can add it as they see fit, so removing it is about the only reasonable standard policy.

Example

```
   cb_read()
the quick brown fox jumped over the lazy dog
```

This means that the given string was in the clipboard.

cb_write

Description

Write a string to the clipboard. This operation is not available in server mode.

Usage

`cb_write(arg)`

Arguments

`arg` - the argument. It may be any data type, but it will be converted to a string before being written. This also includes stems, which are turned into JSON first.

Output

true if the operation succeeded, false otherwise.

constants

Description

Get constants that QDL defines

Usage

`constants([name])`

Arguments

None – a complete stem consisting all system constants

`name` – The value associated with this property name.

Output

A stem consisting of various constants described in this document. Since this is a function, you can either access the values with an argument or a stem index.

Examples

```
  say(constants(), true)
{
 var_type: {
  boolean:1,
  string:3,
```

```
  null:0,
  integer:2,
  decimal:5,
  stem:4,
  undefined:-1
 },
 file_types: {
  string:-1,
  binary:0,
  stem:1
 },
 detokenize: {
  prepend:1,
  omit_dangling_delimiter:2
 },
 error_codes: {
  system_error:-1
 }
}
```

This consists of the types of variables that are output from the *var_type* command.

constant() values

Name	Value	Description
`var_type.boolean`	1	
`var_type.decimal`	5	
`var_type.integer`	2	
`var_type.null`	0	
`var_type.stem`	4	
`var_type.string`	3	
`var_type.undefined`	-1	
`error_codes.system_error`	-1	Used in try – catch blocks. If there is some internal error processing then this is raised and a message set.
`file_type.binary`	0	Return file contents as base 64 encoded byte stream
`file_type.stem`	1	Return file contents in a stem list, one entry per line
`file_type.string`	-1	Return file contents as single string
`detokenize.prepend`	1	See the detokenize function section
`detokenize.omit_dangling_delimiter`	2	See the detokenize function section

info

Description

Get various bits of system information in a stem.

Usage

`info([name])`

Arguments

None – a stem consisting of all properties

`name` – If a single property name is specified, that is returned or an empty string if the property is undefined.

Output

A stem with various bits of system information. This will vary from installation to installation.

Examples

```
  say(info(), true)
{
 system: {
  processors:8,
  initial_memory:479 MB,
  jvm_version:1.8.0_261
 },
 os: {
  name:Linux,
  version:5.4.0-48-generic,
  architecture:amd64
 },
 user: {
  home_dir:/home/ncsa,
  invocation_dir:/home/ncsa/dev/ncsa-git/security-lib
 }
}
```

The major bits of this are

- *qdl_version* = information about the currently running version of QDL and how it was built.
- *user* = information about the user, such ash their home directory
- *boot* = information the system used to boot itself.
- *os* = information about the current operating system QDL is running under and
- *system* = information about the computer system itself, such as the number of processors, the current java virtual machine version etc.
- *lib* = standard extension classes, such as http or database.

Getting a single property.

```
  info('os.name')
```

```
Linux
  info().'os.name' ; // Since it is a stem, you can do this too.
Linux
```

info() values

Not all of these may be available, depending on various combinations of hardware and systems.

Name	Description
`user.home_dir`	The home directory for the user in the ambient operation system
`user.invocation_dir`	The directory from which QDL was started.
`system.initial_memory`	Amount of RAM allocated to QDL at system startup. Depending on the system, more may be allocated as needed
`system.jvm_version`	The version of the Java virtual machine that is running QDL
`system.processors`	The number of CPUs that are capable of being used by QDL.
`os.architecture`	The architecture (underlying hardware info) for the current operating system
`os.name`	The name of the operating system
`os.version`	The current version of the operating system
`qdl_version.version`	The actual version of QDL you are running.
`qdl_version.created_by`	The user that compiled this version of QDL
`qdl_version.build_jdk`	The version of the JDK under which this version of QDL was compiled.
`qdl_version.build_nr`	The build number for this version of QDL
`qdl_version.build_time`	The time stamp when this version of QDL was built
`boot.qdl_home`	The home directory set for QDL. Any relative file operations are resolved against this
`boot.boot_script`	Path to any boot script that was be run on start
`boot.cfg_file`	The configuration file that was used
`boot_cfg.name`	The name of the configuration in the configuration file
`boot.log_file`	The file used for logging
`boot.log_name`	Entries within the boot file are prefixed with this so they can be searched for.
`boot.server_mode_on`	Is this running in server mode?
`lib.*`	name of a supplied Java module.

Example. Loading the HTTP module

To get a list of all the standard java modules, issue

```
   info('lib')
{
 http:edu.uiuc.ncsa.qdl.extensions.http.QDLHTTPLoader,
 db:edu.uiuc.ncsa.qdl.extensions.database.QDLDBLoader
}
```

(There may be more of these.) Here there are two modules in this distro, one for http access and one for db access. To load one of these, e.g. the http library, issue

```
  q:=module_load(info('lib').'http', 'java')
  q
qdl:/tools/db
```

You can now import this as needed.

```
   module_import(q)
http
```

is_defined (∃, ∄)

Description

A scalar-only function that will return if a given variable is defined, *i.e.*, has been assigned a value.

Usage

```
is_defined(var) or ∃var, ∄var
```

Arguments

`var` is the variable. Remember that stem variables end with a period if you are addressing the entire thing.

Output

A boolean.

Examples

```
    a := 'foo';
    is_defined(a)
true

    is_defined(b)
false

    b. := make_index(4);
    ∃b.
true

    is_defined(b.1)
true
```

```
    is_defined(b.woof)
false

    ∄b.woof; // is this undefined?
true
   ∃[a,b.,c] // check a stem of variables for existence
[true,true,false]
```

Note that if a stem is defined, then you can use this to check the elements as well.

is_function (∃, ∄)

Description

Checks if a symbol is a function.

Usage

```
is_function(var , arg_count) or f ∃ arg_count or f ∄ arg_count
```

Arguments

`var` is the name of the function or stem of them.

`argCount` – This is the number of arguments that the function may accept or a stem of them.

Output

A boolean which is **true** if the function is defined in the current scope.

Examples

In this case, a function, *f* is defined in a module called *mytest:functions*

```
    import('mytest:functions');
    is_function('f',1);
true
   f∃1; // same as previous example
true
   g∃null; // query if there are any functions, regardless of arg count, named g
false
   f∃1[;4]; // Check if f is defined for several argument counts
[true,false,false,true]
      [f,g,h]∃1; // check several functions
[true,false,true]
     [f,g]∃[1,2]// subsetting is on, check for f with 1 arg and g with 2 args.
[true, false]
```

os_env

Description

Get or list the environment variables for the system. This allow QDL to be called from a script and have access to the current system environment, such as in bash as $PATH, $HOME, etc. The difference is that you do not need to supply the leading "$". If you operating system is case sensitive, then the variables will be too, so 'path' and 'PATH' may or may not return the same value. This is OS dependent.

Usage

```
os_env([arg0, arg1,… ])
```

Arguments

No argument means to list all of the environment variables.

Arguments are the names of properties in the ambient operating system environment. If a single argument is given, then a single value is returned. If a list of them is given, then a stem of them is returned. Note that any keys are encoded.

Output

Either a stem or a single string. If a property is not found an empty string is returned (single argument). If the property is not found in a list, then that property is not returned. This is extremely useful when writing scripts and allows for seamlessly invoking them. Set any values you need in, *e.g.*, a shell script and then access them in QDL.

A final note is that in server mode, all requests to get information about the system will only return an empty string. script_path

Examples

```
    os_env('HOME')
/home/userName
```

In this case, the request is for the user's home directory and that is returned.

Another example

This parses the path on unix systems:

```
  tokenize(os_env('PATH'),':')
[/usr/local/sbin,/usr/local/bin,/usr/sbin,/usr/bin,/sbin,/bin,/usr/games,/usr/
local/games,/snap/bin]
```

So each element of the list is a path component.

remove

Description

Remove a variable and its values from the symbol table.

Usage

```
remove(var)
```

Arguments

var – a variable or string (name of object) to to be removed. If you supply the variable, then that is removed *not* its value. If you supply a string (as a constant) then that is removed.

Output

True if it was removed, false otherwise.

Examples

Here we define a stem and check is defined, then remove it.

```
    t. := [;5];
    say(is_defined(t.));
true

    remove(t.);
    say(is_defined(t.));
false
```

Here we set a variable then remove it.

```
    p := 'abc';
    say(is_defined(p));
true
    remove(p);
    say(is_defined(p));
false
```

script_path

Also, this will remove entries to stems, so

```
remove(t.b)
```

will remove the entry with index b from the stem t. Similarly

```
remove(t.x.)
```

will remove the *entire* sub-stem x. Use with care!

```
  stem.0. := [;3]
  stem.foo:= 5
  stem.
    [[0,1,2]]~{
 foo:5
```

```
}
  is_defined(stem.0)
true
  remove(stem.0)
true
  is_defined(stem.0)
false


_  stem.
{
 foo:5
}
```

Another example of passing in variables vs. a string.

Since this causes some confusion, here is an example where a stem is created and an entry is removed first using a variable and secondly as a string. The key point is that if you supply a variable then its value is not accessed.

```
  foo. := [;5]
  remove(foo.3)
truescript_path
  foo.
{
 0:0,
 1:1,
 2:2,
 4:4
}
  remove('foo.2')
true
  foo.
{
 0:0,
 1:1,
 4:4
}
```

script_path

Description

Get or set the current script path. This only affects `script_run()` and `script_load()` This is the set of all paths (including vfs paths) that will be checked when running scripts. If you run a script with an absolute path, e.g.

/ home/bob/scripts/init.qdl

Then the script is run. If the path is relative, then it will be checked against the paths in this variable. Specifying a scheme restricts resolution to that scheme. No scheme means every path will be checked.

So if

```
   script_path()
{
   0=vfs#/pt/temp/,
   1=/usr/share/qdl
}
```

Then here are the resolutions for paths

- `vfs#init.qdl ==> vfs#/pt/temp/init.qdl`
- `vfs#ncsa/reset.qdl ==> vfs#/pt/temp/ncsa/reset.qdl`
- `init.qdl ==> vfs#/pt/temp/init.qdl, /usr/share/qdl/init.qdl`
- `abc#boot.qdl ==>` *none*, because abc is not a scheme here.
- `#boot.qdl ==> /usr/share/qdl/boot.qdl` No scheme means to force resolution in the local file system, which is the default. Note that if QDL is in server mode, this will fail.

Finally, this can (and should) be set in the configuration so please consult the documentation there.

Usage

```
script_path([string | stem.])
```

Arguments

none - Return the current list of paths

`string` - a string of paths in the form path0:path1:path2… i.e., each path is separated by a colon

`stem.` - a list of paths, one per entry

Output

If no argument, a stem of the current paths. Otherwise true if the path was set from the argument.

Example

```
    script_path()
[ vfs#/mysql/,
 vfs#/pt/temp/
]
```

In this case, two paths will be checked and both are in virtual file systems.

to_boolean

Description

Convert a value to its boolean representation. This is very useful in places like scripts, where the argument may be a string (like ‘true’) and must be converted to a boolean. QDL scripts will faithfully pass along their values, but external scripts can only pass in strings.

Usage

```
to_boolean(arg)
```

Arguments

arg - any value, including stems. Conversion is as follows:

boolean - no change

string - returns logical *true* if the argument is 'true' (case sensitive)

integer - returns *true* if and only if the value equals 1

decimal - return *true* if and only if the integer part equals 1.

stems – applied to each element.

Output

A boolean value or values if applicable.

Examples

Examples of converting each type. Note that with the decimal, only the integer portion is checked and that must be equal to 1 in order to get a *true* back.

```
  to_boolean('true')
true
  to_boolean(1)
true
  to_boolean(319/47)
false
  319/47
6.787234042553191
  to_boolean(1.000003)
true
  to_boolean([0,1,0])
[false,true,false]
```

Description

Convert a scalar or simple stem to numbers.

Usage

`to_number(scalar | stem.)`

Arguments

`scalar` – any type is accepted.

`stem.` – a stem of scalars. At this point nested stems are not processed.

Output

A number or stem of numbers. The types may be mixed (so integers and decimals). Note that boolean values *true* and *false* are converted to resp. 1 and 0. Numbers are simply returned, unchanged. The value `null` cannot be converted and if found will raise an error.

Examples

Here is a stem with a few different types (including an integer as the last entry).

```
  s.0 := '123';
  s.1 := '-3.14159'
  s.2 := true
  s.3 := 365
```

To convert everything that is not already a number to a number:

```
  to_number(s.)
[
 123,
 -3.14159,
 1,
 365
]
```

Here is a check that indeed these are numbers:

```
  5 + to_number(s.)
[
 128,
 1.85841,
 6,
 370
]
```

Just as a check, adding 5 to each element will either concatenate if a string or (in the case of s.3) add it:

```
  5 + s.
[
 5123,
 5-3.14159,
 5true,
 370
]
```

to_string

Description

Convert a variable to its string representation. This creates the representation used by the `print` command but does not output it to the console. It merely returns it.

Usage

```
to_string(arg{,pretty_print})
```

Arguments

`arg` - any variable, stem or scalar-only

`pretty_print` - (optional) boolean (applies only to stems) prints in vertical format if *true*.

Output

A string that represents the argument.

Examples

This is quite useful when printing stems. Remember that if you write

```
'foo' + stem.
```

The result is to concatenate every element in the stem with 'foo' which is not wanted when printing.

```
  'args = ' + to_string([;3])
args = [0, 1, 2]
```

Example say vs. to_string

This will contrast the output of `say` *vs.* that of `to_string`. In the former case, the value of the argument is returned, in the latter, it is converted to a string.

```
  say(4 + say(3 + 4));
7
11
  say(4 + to_string(3 + 4));
47
```

In the first case, `3 + 4` is computed and the value is printed. This is added to `4` and that value, `11` is printed. In the second case, `3 + 4` is computed and turned in to a string, *7*. That is concatenated to `4`, yielding the string *47*.

var_type

Description

For a given variable, return an integer that tells what the stored type is. This is very useful in, for instance, writing switch statements to process the contents of a stem whose elements are unknown.

Usage

```
var_type(arg0, arg1, arg2, ...)
```

Arguments

`arg0`,… Each is an expression (which also means a variable or constant). Note that a list of arguments returns a stem list whose elements are the types of the arguments.

Output

The possible results are all integers and are

Value	Variable type
-1	undefined variable
0	null
1	boolean
2	long
3	string
4	stem
5	decimal

Also, these are output from the `constants()` command and may be accessed there

Examples

We will define a stem with several elements.

```
    a.0 := 42
    a.1. := random(3)
    a.2 := 'foo'
    a.4 := true
    a.5 := -34555.554345
    a.6 := null
```

Note that there is no `a.3` element – it is undefined. The entire stem can have its type checked

```
  var_type(a.)
4
```

This means it is a stem. Next, we loop through the elements and say what the type of each is. Note that the key set does not touch `a.3` since there is no such element. Note that the last

```
  while[for_keys(j, a.)]do[say(var_type(a.j));];
2
4
3
1
5
0

  var_type(a.0, a.2, a.3)
[2, 3, -1]
```

Note that the last one returns a -1, meaning that a.3 is undefined.

For example, how to use it with a switch statement:

```
    )buffer create temp
0| |temp
    )edit 0
edit>i
while[
  for_keys(j, a.)
]do[
   type := var_type(a.j);
   switch[
     if[type == -1]then[say('undefined');];
     if[type == 0]then[say('null');];
     if[type == 1]then[say('boolean:' + a.j);];
     if[type == 2]then[say('integer:' + a.j);];
     if[type == 3]then[say('string:' + a.j);];
     if[type == 4]then[say(a.j);];
     if[type == 5]then[say('decimal:' + a.j);];
    ]; //end switch
]; // end do
.
edit>q
done
    ) 0
integer:42
{0=-6087687479374980224, 1=-6728256611667942117, 2=5319763765663058324}
string:foo
boolean:true
decimal:-34555.554345
null
```

(This uses the line editor and an in-memory buffer.)

Another example

Let us say we wanted to check if the variable *foo* is undefined. If we enter

```
   var_type('foo')
3
```

We expect -1 but get 3 back. The reason is that 'foo' is a string. Make sure you don't quote things. This is right:

```
   var_type(foo)
-1
```

ws_macro

Description

Run a set of workspace commands from QDL. These will be run exactly as if you had typed them in at the console. This allows you to customize a workspace using, e.g., an ini file and the `__init()` method of the workspace. It is not intended to be used more than a wee bit for simple WS scripting, such as setting up a workspace. For instance, you can execute QDL from inside a macro, but if you are doing that, you might want to consider using a script instead. Scripts are how to get complexity in QDL, not workspace macros.

The major difference between macros and scripts is that macros allow for workspace commands and execute in exactly the workspace environment.

Usage

```
ws_macro(arg | arg.)
```

Arguments

`arg` - a string of commands. There may be multiple commands separated with line feeds.

`arg.` - a list of commands. Each line will be executed in order.

Output

This returns `true` if the command succeeds or produces an error.

Examples

Sending a single string with line feeds. This will be split and each token will be executed as a separate command.

```
    ws_macro(')ws get pp\n)ws get echo\n)ws get external_editor');
pp is on
echo is on
external_editor is line
```

Sending a stem of commands.

```
    ws_macro([')ws get pp',')ws get echo',')ws get external_editor']);
pp is on
echo is on
external_editor is line
```

You can also set this up using an ini file.

```
[workspace]
defaults:=')ws set pp on',')ws set echo on',')ws set external_editor nano'
```

would set this to a stem of commands, then you could issue

```
   ini. := file_read('/path/to/ws.ini', 2);
   ws_macro(ini.workspace.defaults);
pp is on
echo is on
external_editor is nano
```

And a __init function might look like this

```
define[
  __init()
][
    ini. := file_read('/path/to/ws.ini', 2);
    ws_macro(ini.workspace.defaults);
];
```

So on loading the workspace you might see

```
  )load my_workspace
my_workspace loaded
pp is on
echo is on
external_editor is nano
```

Another example. Loading modules

A common case is to load modules. Let’s say you wanted to write a function to load java modules. You cannot write a function like this:

```
    load_it(x)→module_import(module_load(info('lib').x, 'java'));
```

This runs fine:

```
    load_it('db')
```

(loads the database module), but then if you issue the **)modules** command, no modules are there. Why? Because the module is loaded inside the function scope and lives there only. The right way to do this is to have a function that creates a macro:

```
    load_it(x)→'module_import(module_load(info(\'lib\').' + x + ', \'java\'))'
```

and then if you run

```
    ws_macro(load_it('db'))
db
true
```

it loads the module into the current workspace.