QDL Reference Manual

Version 1.4

Introduction

This document is the reference manual for the QDL (pronounced “quiddle”) ***Q***uick and ***D***irty ***L***anguage, which is 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 its 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 “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. Virtually nobody knows all of Java or C++ because it is not humanly possible.

Cheat Sheet

Scalar (aka primitive, simple) types: null, boolean, number, string

Variable types: scalar or compound (aka stem variables)

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), 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 *vencode* and *vdecode* for the particulars as relates to working with, *e.g.*, JSON.

QDL supports the following standard algebraic operations:

```
+ - * / % ^
```

Note that the operator % is integer division. So `42/9 = 4.666…` and `42%9 = 4`. 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 .5 power.

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.

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)^0.001)^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***xponent* or *decimal***e***xponent*

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; // And it works everywhere.
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 `e(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.

Notes

1. You should make it a point of enclosing leading negative numbers in parentheses to cut down on ambiguity. 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)? 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**.

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. := indices(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 then

A *op*= B

is identical to issuing

A := A *op B*

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.

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, thogh 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
*	×	*	\u00d7	multiplication
/	÷	/	\u00f7	division
- >	→	d	\u2192	lambda function
null	∅	n	\u2205	null
&&	∧	&	\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
true	⊤	t	\u22a4	logical true
false	⊥	f	\u22a5	logical false
[|	⟦	{	\u27e6	left closed slice bracket
|]	⟧	}	\u27e7	right closed slice bracket

assert[][]	⊨	a	\u22a8	assert
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.)

```
   f(x) → (0 ≤ x) ∧ (.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). 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
```

Strings may be compared using ==, != and =~ so

```
    'foo' == 'foo'
```

would be **true**.

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}' =~ 23456; // Check if this is a 5 digit number
true
```

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 exactly a few reserved key words in QDL:

true	*if*	*while*	*try*	*module*	*define*
false	*then*	*do*	*catch*	*body*	*block*
null	*else*				

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.

Variables and stem variables.

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

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

are all valid simple variables.

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

```
head.asserttail
```

(Geeky stuff, the period is actually called the *child 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, there is no subsetting.
- 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.

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 1 is box (1)
     [ // axis 2 are the columns
       [ // axis 3 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.

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'
   list_append(my_stem., indices(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?

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 **vencode/vdecode** 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 vencode() to change a string to something that is a legitimate variable:

```
  vencode('f(x,y)')
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]]
```

One little caveat is the built_in function `list_append`: The left-hand argument is that name of a variable, not a stem.

Inline conditionals

There is a control structure of **if[…]then[…]else[…]** but this is a pretty heavy weight solution if you just need to check a conditional. QDL also supports the (ternary) syntax of

```
boolean  ? expression0 : expression1
```

meaning that the boolean is evaluated and if ***true*** then *expression0* is returned and if ***false*** then *expression1* is returned. More to the point, this 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.

E.g.

```
    pi()^exp() < exp()^pi() ? 'left i'+'s 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.

One gotcha

You should be aware that the negation operator, ! or ¬ is a monadic operator, hence it affects everything to its right, so there are equivalent

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

If you wanted ! to apply to the left hand side, 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.

The ~ or union operator

There is a specific operator in QDL for stems, 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]
[3,1,2]
```

So even though the first list had a single element in it at index 17, the joined list has that as the zeroth element.

One difference between this and the *union* function is that the union function changes the argument. The ~ operator does not alter its arguments.

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 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.'
```

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.

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);
   say(stem.woof);
2
   say(has_key(stem., 'woof'));
false
```

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
```

Slice operators

There are two slice operators available. The first is the ***open slice*** and is defined as

```
[(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` i.*e.* it is inclusive of `start` and exclusive of `stop`.

Note that omitting the first argument is the same as setting it to zero, omitting the last argument uses a default step of 1. So

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

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]
```

The second type, astonishingly called the ***closed slice*** gives back *n* evenly distributed elements over an interval

```
[| (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. Note that there must always be a stop and a count 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.

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. In other words **`a.x.y. … z == a.[x,y, … ,z].`** Why? Because you can then create index liFsts 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 index lists, you can access the elements.

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)
```

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[
   // lots of stuff
  a := 'foo';
 ]else[
  // lots of stuff
  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
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:

```
    )buffer on
Buffer mode on.
j :=5;
if[
   j <5 || 5 < j
 ][
    say(j + ' is not 5');
 ]else[
    say('j is ' + j);
];
.
j is 5

)buffer off
Buffer mode off.
```

(Rather than stick this in a script, we used the interpreters)`buffer` command to keep all of the lines and then execute it when there is a single period as the first character on a line. Oh and interpreter commands which start with a ')' still work in buffer mode. Consults the work space documentation for more.)

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

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

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  [i<5]then[var.foo := 'bar';];
       if  [5=i]then[var.foo := 'fnord';];
       if[5<i%4]then[var.foo := 'blarf';];
       if [true]then[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 is *exactly* one reserved error code and the is value **`-1`**. This 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).

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.

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.

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]);
```

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, but you may use any other value you like. This reserved value is available in **`sys#constants()`** under **`error_codes.system_error`**

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'.

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([;100]/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_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
```

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. Previously defined values are still available. The variables in the signature are populated with the values (which are copied) . 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.

Help

The lines that start with **`>>`** at the beginning of the body are read by the system 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.

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 $xyz = x^3 + y^3 + z^3$.

```
define[
  armstrong(m)
  ][
     >> An Armstrong number is a 3 digit number that is equal to the sum of its cubed
digits.
      >>  This computes them for 100 < n < 1000.
     >> So for example 407 is an Armstrong number since 407 = 4^3 + 0^3 + 7^3
     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[say(sum);];
          sum := 0;
       ]; // end while
  ]; // end define
```

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.

```
   f(x,y,z) -> x+y+z;
   f(3,2,1)
6
```

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

```
signature -> [statement 1; statement 2;…];
```

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) -> [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) -> [if[0<x][return(1);]else[return(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.

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.

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!
```

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 ][ expression];
```

An alternate syntax uses unicode and is

```
⊨ conditional : expression
```

where ⊨ is unicode 22a8. Again, this really should be on a single line. If the conditional is true, nothing happens. If it is false, then the *expression* is evaluated and the result is converted to a string and is returned.

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

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

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. 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.

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
```

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.

Modules

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, of course, the concept of a ***module*** in QDL that does this. Very simply, a module is defined using a unique identifier that **must** be a URN and an alias. The alias is just a regular name like a variable. 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.

Let us say that we had the following script in the file my-module.qdl:

```
module[
   'a:a','a'
]body[
    foo := 'abar';
    define[f(n)]body[return(n+1);];
]; // end
```

The syntax is clear here:

```
module[
   uri, alias
 ]body[
  // any statements except another module definition
];
```

The URN puts this in a unique namespace. There can be no conflicts. The alias is used as a shorthand to access it. URNs many be quite complex. Generally though you should not put in query parameters and such. To access anything in a module explicitly you must ***qualify*** the name *i.e.*,

```
alias#name
```

It is easiest to see this in action, so in the interpreter (clear state)

```
    )funcs

    )vars

```

Note that there is nothing in this session so these show nothing. Let's import our module above in a file called `my-module.qdl`. We are going to import it with another alias just to show we canonical

```
module_load('my-module.qdl');
module_import('a:a', 'b');

    )vars
b#foo

    )funcs
b#f(1);
```

Loading the module makes the session aware of it, but you must also import it. You may import a module multiple times with different aliases. If we did not specify the second argument, then the default alias in the module definition, 'a', would be used. Now at this point, there is nothing else in the session and there are no name clashes possible, so we can use the unqualified name:

```
    say(foo);
abar
```

And we can invoke the function too

```
    say(f(5));
6
```

Local variables and functions vs. modules ones

If we had something like

```
    foo := 5;
    load('my-module.qdl');
    module_import('a:a', 'b');
    )vars
b#foo, foo
```

This shows us that there is a session variable with the same name. If we try to access the unqualified name we get

```
say(foo);
Error: The variable "foo" exists in multiple modules. You must qualify which one
you wish to use.
```

So this means that there is no way for the interpreter to resolve this. The solution is that you have to qualify the session variable with a simple #:

```
say(#foo);
6
```

There is no namespace in effect for a basic session. This is the most straightforward way to handle name clashes, *viz*., if there is any conflict, require the user resolve it rather than having the interpreter doing it and risking getting it wrong. Part of the philosophy with QDL is that is avoids a lot of the sorts of things that cause hard to find errors and tells you up front how to fix it.

Help for modules

Just like functions, you can create help for a module by inserting documentation commands right before the body. You can access this in the workspace with

```
)module name|alias -help
```

Note that all of the functions in a module (with their fully qualified names) are automatically found by the help system, so this command is just for the module itself.

Related Functions

module_import

Description

Import a module with a given alias.

Usage

```
module_import(urn[, alias]);
```

Arguments

`urn` = a Uniform Resource Name (as per RFC 2141 and 1737, if you track such things).

`alias` = a string that conforms to the requirements of a QDL variable name.

Note that both of these are passed in as strings and the URN is checked for form. Also, when a module is defined, it has an alias given. This is the default, so you may import the module with only the URN. If you specify another alias, that is used.

Output

A logical *true* if the import succeeded and *false* otherwise.

Examples

```
module_import('my:/thingie', 'ahab');
```

Would locate the module with URN *`my:/thingie`* and let you use *`ahab`* as the local alias. Remember that the module must be loaded first or you will get an error.

Another example of multiple imports

You may import a module multiple times with different aliases. This effectively creates new independent copies of it. In the following small session, a module is created (this effectively loads it, so you do not need the `load_module` command. It is then imported twice, first using the default name, then with a different alias. The values are set so you may see the they are independent. (This is very much like most object oriented languages which have a notion of a class, which is a user defined template from which instances are created. The instances are accessed using their alias. So if you do python or java, `import(a,b)` is the same as creating a new instance of a class.)

```
  module['a:/a','a']body[q:=1;];
  module_import('a:/a')
```

```
true
  module_import('a:/a','b')
true
  )modules imports
a:/a = [a,b]
  a#q :=5;
  b#q :=6;
  a#q
5
  b#q
6
```

load_module

Description

Load a module from an external source. Note that once a module is loaded, it is automatically available for import in any other module. Since namespaces are unique, there should *never* be a collision. Said differently, there there is a master list of all loaded modules in a given workspace.

Usage

```
module_load(arg[,type]);
```

Arguments

`arg` – Either the full path to the file that contains the module, when there is no second module *or* the fully qualified Java class name. The Java class may be either a class that extends the QDLLoader class or extends the JavaModule class. This is loaded then run in its own environment which is then added to your session. *Be sure the class is included in your class path*.

`type -` If absent this implies file. The only other supported value at this point is `'java'` which means that the first argument is a class name.

Note that you can also load modules at start up via the configuration file. You should look in the documentation there for more information.

Output

A true if it succeeds and false otherwise (or an error message).

Examples

```
   module_load('/home/bob/qdl/modules/mega_module.qdl');
```

Would load the given module.

Loading a Java module

The example loader ships with QDL, so you may always import that. To do so issue the following:

```
 module_load('edu.uiuc.ncsa.qdl.extensions.example.QDLLoaderImpl', 'java')
true
```

To check that it was imported, we ask the workspace, import it and then check again.

```
  )modules
qdl:/examples/java
  )imports
(no imports)
  module_import('qdl:/examples/java')
true
```

If it imported correctly, it should have brought in a single command.

```
  )funcs
java#concat(2)
```

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).

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	0	Disable logging/debugging. Nothing is printed
INFO	1	informational
WARN	2	warnings - things that are serious but don't require action
ERROR	3	errors - the system cannot resolve the issue, but processing continues
SEVERE	4	an issue that prevents processing of anything.
TRACE	5	Print everything at all levels

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

```
   debug(2);
3
```

```
// previous debug level was 3 so only errors 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.

```
debug(2); //set only warning on
if[!is_defined(user.id)][
 log_entry(1, 'Processing as anonymous user');
 //.. do anonymous user stuff
  debug(0, '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 them on in a session is a simple as issuing a command like

```
debug(1)
```

(or any of the non-negative levels).

Related functions

debug, log_entry

Description

Set the debugging level or issue debugging messages.

Usage

```
debug([level , message])
log_entry([level, message])
```

Arguments

(no arguments) - Inquire about current level

level - (only) set the current level for all subsequent operations 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 level.

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

```
debug(2, ‘foo’)
```

would display nothing, since 2 < 3. Debugging is printed to syserr and ends up on the screen, 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.

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.

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]
```

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.

`is_regex` - if true then all matching is case sensitive. Note this is only a scalar. 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, an 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, an empty string is returned for the last element.

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

A regex example.

```
  head('a\t\t d, \tm,\ti.n','\\s+', true);
a
```

This matches by whitespace, in this case ignoring tabs.

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

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.

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
```

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';
    ssourceStem., 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` – 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
```

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 (implies 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.
```

vdecode

Description

Decode and encoded variable name. All escaped characters are unescaped. **Note** this does support UTF-8 encodings of all characters implicitly.

Usage

```
vdecode(arg)
```

Arguments

`arg` – the String to be decoded. Note that not every string can be decoded! Since escaped characters are of the form $xy where x and y are hexadecimal numbers, it is possible to make illegal escapes, *e.g.*, $$$ will most certainly fail

Output

The decoded string.

Examples

```
  vdecode('$26$2A$28$26$25$23')
&*(&%#
```

An example of a name that cannot be decoded. This is, of course, because escaped values are of the form $xy where x and y are hexadecimal numbers:

```
   vdecode('$$foo')
Error: Could not decode string:Invalid escape sequence
```

Here is an example showing the UTF-8 implicit encodings of non-ASCII characters

```
  vencode('你浣')
$E4$BD$A0$E6$B5$A3

  vdecode('$E4$BD$A0$E6$B5$A3')
你浣

  '你浣' ==  vdecode('$E4$BD$A0$E6$B5$A3')
true
```

(This example was chosen because these two characters can get swapped if the encodings are not carefully handled, so this shows that a well-known edge case is handled correctly. And no, I have no idea what these characters represent.)

Finally, a little Vietnamese (first line of *The Tale of Kieu*.) which is very hard to represent in strings:

```
  a :='Trăm năm trong cõi người ta, Chữ tài chữ mệnh khéo là ghét nhau.'
  a == vdecode(vencode(a))
true
```

vencode

Description

Encode a string, escaping all characters that are not legal for a variable name. This is done by using the standard URL escape specification (where every escaped character is replaced with a % and s 2 digit hexadecimal code), but using a $ in place of the %. We have to do this in variable names since % represents integer division in QDL. So for instance, the blank ' ' escapes to %20 which in turn we would represent as $20.

The reason for this is interoperability, mostly with JSON. If we are going to export variables (such as stems) the stem names are, in point of fact, legal variable names and will be resolved against the symbol table. In the case of JSON, these correspond to the property names, which are simple strings and may contain illegal characters. So if we have a JSON object

```
{“p:/*/q#”:”woof”}
```

This cannot be turned in to a stem without some change to the property name of `p:/*/q#`. This function will escape everything and return `p$3A$2F$2A$2Fq$23` which can be turned back in to the original using the vdecode function.

Usage

```
vencode(arg)
```

Arguments

`arg` – the string for which all illegal variable characters will be escaped.

Output

A that is a legal variable name in QDL. Note that every string input can be encoded this way.

Examples

```
  vencode('&*(&%#')
$26$2A$28$26$25$23
```

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_b64

Description

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

Usage

```
decode_b64(arg)
```

Arguments

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

Output

The decoded string

Examples

```
    say(decode_b64('VGhlIHF1aWNrIGJyb3duIGZveCBqdW1wcyBvdmVyIHRoZSBsYXp5IGRvZw');
The quick brown fox jumps over the lazy dog
```

encode_b64

Description

Encode a string in base 64. The returned string is URL safe.

Usage

```
encode_b64(arg)
```

Arguments

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

Output

If a single argument, it will be base 64 encoded. If a stem, each element will be. Non-strings are not changed.

Examples

```
    say(encode_b64('The quick brown fox jumps over the lazy dog');
VGhlIHF1aWNrIGJyb3duIGZveCBqdW1wcyBvdmVyIHRoZSBsYXp5IGRvZw
```

from_hex

Description

Take a hexadecimal representation of a string and convert it back

Usage

```
from_hex(arg)
```

Arguments

`arg` -- a hexadecimal string or a stem of them.

Output

The string that corresponds to this argument or a stem of them (if the argument was a stem). ***Note*** that there is not a one-to-one correspondence between the number of bytes and the string length! In base 64, 3 character are cannibalized in to 24 bits, then the bits are re-grouped into 4, 6-bit characters (there are 64 such 6-bit characters, hence the name). So to represent n bytes requires 4*n*/3 characters and since this is only exact when *n* is divisible by 3, there is some padding. 8 bytes will give you 11 characters.

Examples

```
say(from_hex('54686520717569636b2062726f776e20666f78206a756d7073206f76657220746865206c617a7920646f67'));
The quick brown fox jumps over the lazy dog
```

hash

Description

Calculates the SHA-1 digest of the arguments and returns the value as a hex string. This means that the result 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). This is used in various cryptographic applications.

Usage

```
hash(arg)
```

Arguments

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

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 `to_hex` function.

Examples

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

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

A point to make about SHA-1 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.

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,
 -1.36000254004806800000000000000
]
```

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^.19
1.3013418554419335668321600491224611591208423214517
```

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

```
  numeric_digits(100)
15
  4^.19
```

```
1.301341855441933566832160049122461159120842321451727432949734773315583318806133404
306182838299702735
```

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_b64* 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.

```
  to_hex(decode_b64(random_string(16)))
efbfbdefbfbd3e7374efbfbd22efbfbdefbfbd06efbfbd10c69d2178
```

to_hex

Description

Convert a string to its hexadecimal representation (*i.e.* base 16 encoding). Note that this useful but cannot generally be used in, say, web traffic because the you have to preserve the exact character set used. (Practically, most applications that use this send along the character set, here UTF-8, just make sure you do too if you need to.) If you really need to send something faithfully, you should consider using base 64 encoding instead.

Usage

```
to_hex(arg)
```

Arguments

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

Output

A string that is the hexadecimal representation of the underlying bytes for the string.

Examples

```
    say(to_hex('The quick brown fox jumps over the lazy dog'));
54686520717569636b2062726f776e20666f78206a756d7073206f76657220746865206c617a7920646
f67
```

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
`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`**
`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([x])`	`--`	`π`	the power of pi in the current precision, **`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, `log₂ (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 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.

Stem specific 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(indices(10), 6+indices(5))
[0,1,2,3,4]
```

detokenize

Description

Inverse of `tokenize()`. Take a stem and turn it into a string using a delimeter.

Usage

```
detokenize(arg, delimiter[, options])
```

Arguments

`arg` - either a scalar or stem

`delimiter` - either a string or a stem of strings to be used as delimiters

`options` - (options) an integer that dictates how the delimiters are used.

Output

A string consisting of the elements in arg with the delimiter between them. If both arguments are scalars, this is equivalent to simple concatenation (unless the options preclude that, then only the left argument is returned)

Examples

A comma delimited list of integers

```
  detokenize(i(10), ', ', 2)
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

In this case, a ', ' is placed between each element of the stem. The final option of 2 says to omit the trailing delimiter (or it would have ended with `'...8, 9, '`)

You can make a blank delimited list as per the following example.

```
  caps.
[
 foo,
 compute.create:/,
 storage.read:/store
]
    detokenize(caps., ' ', 2)
foo compute.create:/ storage.read:/store
```

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}
```

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]
```

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

```
  n(4) == n(7)
[true,true,true,true]
```

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(@==, n(4), n(7)))
[true,true,true,true,false,false,false]
```

This compares every element in n(4) with every element in n(7), then reduces that result. 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. ⊗ 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 looks a tiny bit like the tensor product sign.

Usage

```
for_each(@f, arg_1., arg_2., ..., arg_n.)
```

Arguments

`@f` - the n-adic function. It will be fed all of the elements from all the stems

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

Output

A stem consisting of the outer (Cartesian) product of each of the arguments with the function ***f*** applied to each.

Examples

Simple example, making a multiplication table.

```
 a. := for_each(@*, 1+n(5), 1+n(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.

Example.

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

```
 z(x,y) -> x*y;
 z. := for_each(@z, [|-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. (The simple example of something common that is impossible to represent a set in JSON without structuring it somehow.) Also, how to represent certain common things like dates varies by library. If you get something very odd where you expect a date (normally some large object with a ton of entries in it) , then the suggestion is to, if at all possible, replace any dates with ISO 8601 format dates, which are the standard or perhaps integers that represent the time in milliseconds.

Stems are more general data structures than JSON, and if you really need to serialize an object to JSON (as it is properly termed) then you might want to serialize your stem in sub-units rather than send over a single massive object. Keep it simple.

Finally, can *vencode* will be used on each key of the JSON object if you specify it. In QDL, stem indices may be quite general, but since their values may be accessed via variables – which have much more limited naming – you may have to set a variable to access a stem value. In JSON they are arbitrary string that are keys.

Note: It should always work turning a JSON object in to a stem. Turning it back might not but the former is all we can guarantee. Now, if the stem follows a few guidelines of

- be careful of integer indices in arrays and the JSONObject, since these can get overwritten.
- No odd characters in the names of JSON properties

then there should be no problems mapping stems to JSON and back and forth.

Usage

```
from_json(var[, convertAll])
```

Arguments

`var` = the string-valued variable to be converted

`convertAll` = if *true* apply *vencode* to each key.

Output

A stem that contains the information in the original.

Generally a stem list corresponds to a JSON array, do if you need to convert back and forth, it is best not to overload the stem with non-list values. Just keep everything as simple as possible.

Note that JSON properties will be turned in to valid QDL variable names by escaping them as needed. 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}
```

Escaping of JSON keys.

In this example, a simple JSON list is given and the key for this list is not even remotely a valid variable in QDL. In this case, the name is escaped.

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

A less contrived example.

```
  a. := from_json('{"Jäger-Groß":"enrolled"}', true)
  a.
{J$C3$A4ger$2DGro$C3$9F:enrolled}
```

This may seem a bit hard to deal with, but do remember that there are many functions such as *for_keys* that allow you to simply run over all stem values without having to know how the escaping turned out and if there is a question, you can use *vencode* or *vdecode* plus the fact that variables are resolved.

Mostly this machinery exists when trying to send data to other systems in a reliable way. When you need it, you really need it but otherwise it is quite specialized.

```
while[
   for_keys(key, a.)
][
  if[
       key == vencode('Jäger-Groß')
     ][
       // … do stuff with this
   ];
];

// or just grab a values directly

x := vencode('Jäger-Groß');

attributes. := a.x; // grabs this by key and the values are now accessible.
```

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)
```

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. := indices(3);
    b. := indices(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
}
```

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.

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` = 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.

has_keys

Description

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

Usage

```
has_keys(target., list.)
```

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. := indices(10);
    say(has_keys(var., w.));
[true,true,true,true,true,false,false,false,false,false]
```

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.

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, n

Description

Make a list of indices. This is very useful in conjunction with looping. indices() and i() are equivalent

Usage

```
indices(arg0[,arg1,arg2,... fill.]); n(arg[,arg1,arg2,..., fill.])
```

Arguments

`arg_k` is a number. This will be the size of the resulting index set.

`fill.` is an optional stem of scalars that will be used as values cyclically

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]
 ]
```

In the last 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. :=  indices(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, -8452824104954706854]
XkiEEZ1g297TfZY3ItjL5w
[5095335425002685458, 380662481390939243, -2302353563657105155]
FUKe27vv56w8-lahY2InNA
```

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'
  list_append(my_stem., indices(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(indices(10))
true
```

join

Description

Join two stems along a given 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. So in

`a.p.q.r`

`a.` means axis 0

`a.p` is axis 1

`a.p.q` is axis 2

`a.p.q.r` is axis 3.

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

Output

The joined stem.

Examples

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.
    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]]
]
```

```
  // also q.~w.
 z. := join0(q.,w.)
 //  *   <--- 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)
 //   *  <--- 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)
 //     *  <--- 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.k) == size(q.k) + size(w.k)

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

This takes and interval [start, stop] and creates n total points on it, then evaluates whatever the function is. The result is a join of the x. and y., 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.

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` = (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` = 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 keys() command gives the following

```
  keys(b.)
{
 OA2_arf:OA2_arf,
 fnord:fnord,
 OA2_foo:OA2_foo,
 OA2_woof:OA2_woof
}
```

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

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

list_append

Description

Append a list stem another list stem.

Usage

```
list_append(stem1., stem2. | value)
```

Arguments

`stem1.` = the stem to which the values will be appended

`stem2.` = the stem from which values will be taken *or* increasing

`value` = the scalar that will be appended.

Output

A stem, the same as the first argument but with the appended values.

Examples

```
  stem1. := indices(5);
  stem1.
 [0,1,2,3,4]
  stem2. := 5*indices(6);
  stem2.
[0,5,10,15,20,25]
  list_append(stem1., stem2.)
[0,1,2,3,4,0,5,10,15,20,25]
  list_append(stem2., 'woof')
[0,5,10,15,20,25,woof]
```

And to be clear, the last command is identical to

```
  stem2.~'woof'
```

list_copy

Description

Copy from one list stem to another.

Usage

```
list_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

`length` = how many elements to copy

`target.` = the target stem of the copy

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

Output

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

Examples

```
  source. := indices(5)+10
  target. := indices(6) - 50
  list_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.

list_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

```
list_insert_at(source., start_index, length, target., target_index);
```

Arguments

Examples

```
  source. := indices(5) + 20
  target. := indices(6) - 100
  list_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.

list_reverse

Description

Reverse the order of the elements in a list

Usage

```
list_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

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

list_subset

Description

Grab a subset of a given list.

Usage

```
list_subset(source., start_index [, length]);
```

Arguments

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

`start_index` = where to start in the source stem

`length` (option) = how many elements to take. Omitting this means take the rest of the stem

Output

The altered stem. Note that this does nothing to the original stem and the indices are adjusted accordingly, so that the first index of the output is 0.

Examples

```
  stem. := indices(5) + 20;
  // Just grab the tail of this list
  list_subset(stem., 2)
[22,21,24]
  // grab some stuff in the middle.
  list_subset(stem., 1, 3)
[21,22,23]
```

list_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

```
list_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

```
 list_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 < list_starts_with(X.,Y.))
```

So

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

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
```

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
```

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
```

reduce

Description

Apply a dyadic function pairwise to each member of a list, returning the final output only.

Usage

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

Arguments

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

`list.` - the list to be operated upon

Output

A scalar.

Examples

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.

rename_keys

Description

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

Usage

```
rename_keys(target., new_keys.);
```

Arguments

`target.` - the stem to be altered

`new_keys.` - a stem of keys. The keys in it are the ones in target. To be altered to their values.

Output

A new stem whose keys have been altered as per the second list.

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.)
  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 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}
```

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);
```

Arguments

`target.` – the stem

`scalar` – the default value

Output

This returns the default value set.

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*indices(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.
```

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.

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

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(indices(10))
10
```

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.

Usage

```
to_json(stem. [,convert? | indent, indent])
```

Arguments

`stem.` = the stem to represent in JSON notation

`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.

So these are valid calls

to_json(stem., false) – do not convert the names

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

to_json(stem., false, 2) – do not convert the stem variable names and indent with a spacing of 2.

Output

A string in JSON which represents the argument.

Examples

```
  a. := indices(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 *vdecode:*

```
  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.

to_list

Description

Take an arbitrary list of things and put them all in to a stem list.

Usage

```
to_list(arg0,arg1,arg2,…)
```

Arguments

Any number of arguments of any type are allowed, including stems.

Output

A stem list

Examples

In this example, a few values are given, including a stem that is made with the *indices* command. Note that each element of the new list is one of the arguments.

```
    say(to_list(3,45.34,'abc',indices(3)));
[3,45.34,abc,[0,1,2]]
```

Again, this is equivalent to

```
3~45.34~'abc'~[0,1,2]
```

unique

Description

Return the elements of a list that are unique.

Usages

unique(list.)

Arguments

list. = A list of scalars. (If the elements are lists or stems then it may not be able to determine which are truly unique).

Output

A list of exactly the unique values.

Examples

In this example, a couple of lists are

```
  unique(['a',2,4,true]~['a','b',0,3,true])
[0,a,2,b,3,4,true]
```

Another example, showing that this only applies to lists, not whole stems:

```
    unique({'p':'q'}~{'p':'r'}~(2*indices(3))~(2+indices(4)))
[0,2,3,4,5]
```

unbox

Description

Takes a stem variable and splits it up, turning each key in to a variable.

Usage

```
unbox(stem. [, safeModeOn]);
```

Arguments

stem. - the stem to unbox

safeModeOn - 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: safeModeOn = *true* means that nothing will get processed if there is a clash.

Ouput

A *true* if the result worked.

Examples

```
  a. := -5 + indices(5);
  b. := 5 + indices(5);
  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 + indices(10);
  b. := 5 + indices(5);
  a.arf := 'woof';
  b.woof := 'bow wow';
  c. := -20 + indices(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.

```
  p.6 := 100;
  p.~[1,2,3]
{
 6:100,
 7:1,
 8:2,
 9:3
}
```

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.

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])
print(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)

```
    print(42);
42
    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. := indices(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(file_name [, as_string || to_list || is_binary])
```

Arguments

`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

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 **`sys#constants()`** as `file_types.`

Output

Either a simple string (only file name is given), a stem if it is flagged as a list 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_b64('LyoKICBUaGUgZXhwZWN0ZWQgSGVsbG8gV29ybGQgcHJvZ3JhbS4gIAogIEplZmYgR2F
5bm9yCiAgMS8yNi8yMDIwCiovCnNheSgnSGVsbG8gd29ybGQhJyk7Cg'));
/*
  The expected Hello World program.
  Jeff Gaynor
  1/26/2020
*/
say('Hello world!');
```

(This is the sample hello world program for qdl).

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.)

write_file

Description

write contents to a file

Usage

```
file_write(file_name, contents [,is_base64])
```

Arguments

`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.

`is_base64` (optional) – if **true** then try to decode this into binary and write it as a binary object.

Output

Returns **true** if this succeeded.

Examples

```
           hello_world :=
'LyoKICBUaGUgZXhwZWN0ZWQgSGVsbG8gV29ybGQgcHJvZ3JhbS4gIAogIEplZmYgR2F5bm9yCiAgMS8yNi
8yMDIwCiovCnNheSgnSGVsbG8gd29ybGQhJyk7Cg';
    file _write('/tmp/test.qdl', hello_world, true);
```

This is just the base64 encoded hello_world.qdl script from the read_file example.

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.

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.

execute

Description

Send a string to the interpreter and evaluate it.

Usage

```
execute(string)
```

Arguments

`string` – the string to be executed. This must be a valid QDL statement or it will fail

Output

Whatever the executes statement outputs.

Examples

```
    execute('say(2 + 2);');
4

    execute('say(\'abc\' + \'def\');');
abcdef
```

Usage

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 or function

Usage

```
input_form(module_name)
input_form(variable[,pretty_print])
input_form(function, arg_count)
```

Arguments

For variables, this is the name, *e.g.* **foo** or **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.

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 con 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.).

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.

So how to use this with, say, **execute**? Since this is exactly the content of the variable **a.** you need to turn it into a statement:

```
  execute('b.:='+input_form(a.) + ';')
true
   b.
[2,4,6,a,234,-1.23,false]
```

note that this printed output is not input form, for instance, there are no quotes around the string *a* . Part of converting to input form is escaping characters (like quotes) in strings and such. To use this with **execute** you must turn it into a statement. As another check, compare the two variables

```
   a. == b.
[true,true,true,true,true,true,true]
```

Note that for variables this returns the essential parts of the statement. If QDL returned a whole statement, then most of the time you would be picking it apart to use elsewhere. It is vastly easier to add things (like ending semicolons) than picking apart strings trying to get it right.

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');
    input_form('a: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.

script_args

Description

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.

constants, sys#constants

Description

Get constants that QDL defines

Usage

```
sys#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.

Examples

```
  print(sys#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.

Table of current constants

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, sys#info

Description

Get various bits of system information in a stem.

Usage

`sys#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

```
  print(sys#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.

Getting a single property.

```
  sys#info('os.name')
Linux
```

Table of constant names

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?

is_defined

Description

A scalar-only function that will return if a given variais_definedble is defined, *i.e.*, has been assigned a value.

Usage

```
is_defined(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';
    say(is_defined(a));
true

    say(is_defined(b));
false

    b. := make_index(4);
    say(is_defined(b.));
true

    say(is_defined(b.1));
true

    say(is_defined(b.woof));
false
```

This last example shows 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 [, argCount]*)

Arguments

`var` is a string that contains the name of the function.

`argCount` (optional) – This is the number of arguments that the function may accept. If you omit it than the result will reflect if there is ***any*** such named function, regardless of argument count.

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*

```
    say(is_defined('f'));
false
    import('mytest:functions');
    say(is_defined('f'));
true
```

This also works with stem elements, so

is_defined(t.x)

would return true if the stem t. contained the element x.

os_env, sys#os_env

Description

Get or list the environment variscript_pathables 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. := indices(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. := indices(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. := indices(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, sys#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` - 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(indices(3))
args = [0, 1, 2]
```

Example print vs. to_string

This will contrast the output of `print` *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.

```
  print(4 + print(3 + 4));
7
11
  print(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 `sys#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 local on
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
    )
integer:42
{0=-6087687479374980224, 1=-6728256611667942117, 2=5319763765663058324}
string:foo
boolean:true
decimal:-34555.554345
null
```

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
```