QDL Reference Manual

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 critical) set of navigation bearings taken at regular intervals. 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 second meaning of quiddle is the one we are after. 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 major constructs for the language (looping, conditionals, encapsulation &c.) 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.

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`. It also supports the following logical operators

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

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

```
-- ++
```

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.

Assigning values

There is a specific ***assignment 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.

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 50 decimals precision.

```
     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. Since there are many times external programs use double quotes and one of the aims of QDL is to make it interoperate nicely with this 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**.

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 := 42;
```

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.

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;
```

are all valid simple variables.

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

```
head.tail
```

(Geeky stuff, the period is actually called the *aggregation operator*.) Remember that the variable is **head.** (note the trailing period!!) and the tail – which may be complicated -- is just indexing. 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 which contains indices 0,1,...n
- Two stems are ***conformable*** if they have the same keys
- ***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.
- ***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.

Note that this is one of the very few strict pattern enforced in QDL – a stem must end in a period, so issuing something like this (to make a list of integers)

```
a := indices(5);
```

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’;
```

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 := 'http://idp.bigstate.edu/saml’;
myMap.port := 636;
myMap.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=0, help=this is my stem, 1=1, 2=2, 3=3, 4=4}
```

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

Tail substitutions.

If you have defined a variable, say

```
k := 3;
my_var. := indices(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
```

Stems can indeed do a tremendous amount of implicit looping.

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
```

This just points out that a and a. are considered to be wholly unrelated. 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.

You may nest stems as well, so

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

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

works just fine. 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 version is that tail resolution happens for scalars.

Applying scalars to stems

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

```
    a. := 3*indices(5);
    say(a.)
{0=0, 1=3, 2=6, 3=9, 4=12}
```

(The output is of the form `{key=value, key=value,…}`. 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 you issue

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 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.

Control structures

There are 4 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[`** `. . .` **`];`**

The if..then..else statement

The basic format is

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

Note that **`if[`**, **`]then[`** and **`]else[`** are treated as statements, so cannot be split. The else clause is optional. And of course, a simple example is in order:

```
    )buffer on
Buffer mode on.
j :=5;
if[
   j <5 || 5 < j
 ]then[
    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[
  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.

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]then[say('A remainder of 0 is fine.');];
      if[remainder == 4]then[say('A remainder of 4 is fine.');];
      if[remainder == 1]then[raise_error(j + ' not divisible by 5, R==1', 1);];
      if[remainder == 2]then[raise_error(j + ' not divisible by 5, R==2', 2);];
      if[remainder == 3]then[raise_error(j + ' not divisible by 5, R==3', 3);];
 ]catch[
     if[error_code == 1]then[say(error_message);];
     if[error_code == 2]then[say(error_message);];
     if[error_code == 3]then[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.

Related functions

raise_error

Description

Raises an error conditions and passes control to the catch block. Note that raising this outside of a try … catch block will have no effect, since there is nothing to catch it.

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.

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
```

Looping.

The basic structure of a loop is

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

For example, to print out the numbers from 0 to 5:

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

```
0
1
2
3
4
```

This is known as 'yer basic loop'.

There are 3 functions that can help loop.

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)
   ]do[
      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)
 ]do[
  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)
   ]do[
      if[
         j==3
      ]then[
         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.

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, it's 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`*`]);`

Arguments

Only the first two are required.

`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.

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
```

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(j,5)
```

```
 ]do[
  say(my_stem.j);
]
mairzy
doats
and
dozey
doats
```

Defining functions

The syntax for defining a function is very simple:

```
define[
    signature
  ]body[
    >> 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.
    c:= a+b;
    say('the sum is ' + c);
    return(c); // 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.

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)
  ]body[
     >> 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
```

Related functions

return

Description

Return a value or none.

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 ignore everything and just returns “hello world”.

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

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

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. Each is printed as

```
    )help list
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 that this is only for functions. A common convention is workspaces is to have a stem variable named help. each of whose keys contains a description of something useful.

```
    )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.
```

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

```
load_module('my-module.qdl');
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’);
    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.

Related Functions

import

Description

Import a module with a given alias.

Usage

```
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

```
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;];
  import('a:/a')
true
  import('a:/a','b')
true
  )modules
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.

Usage

```
load_module(file);
```

Arguments

file – the full path to the file that contains the module. This is loaded then run in its own environment which is then added to your session.

Output

Nothing

Examples

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

Would load the given module.

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.

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’;
result. = index_of(sourceStem., targetStem.);
```

returns the following stem variable, which has one entry since two stem variable are only checked against corresponding keys:

```
result.bind == 11;
```

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

```
    say(insert('abcd’, 'foo’, 2));
abfoocd
```

This also works for stem variables

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
```

replace

Description

Replace every occurrence of a string by another

Usage

```
replace(source, old, new)
```

Arguments

`source` – the original string or stem of strings

`old` – the current string

`new` – the new string.

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

Examples

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’;
output. := replace(sourceStem., old., newValue);
```

The resulting output is a stem (because an input is) and in this case it is

```
output.bind == 'and in the two bind them’;
```

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’.

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.........
```

tokenize

Description

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

Usage

```
tokenize(arg, delimiter)
```

Arguments

`arg` – either a string or stem of strings

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,',',','));
{0=ab, 1=de, 2=ef}
```

An example tokenizing a stem variable.

```
    q.foo := 'asd fgh';
    q.bar := 'qwe rty';
    say(tokenize(q., ' '));
{bar={0=qwe, 1=rty}, foo={0=asd, 1=fgh}}
```

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

```
    say(tokenize(12345, '1'));
12345
```

In this case, the argument is returned unchanged.

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.)

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 is 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('54686520717569636b2062726f776e20666f78206a756d7073206f766572207468652
06c617a7920646f67'));
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));
{0=3, 1=0}
```

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

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 digits) used.

Usage

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

Arguments

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

Output

The current value.

Examples

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

```
  numeric_digits(15)
50
  4/19
0.210526315789473
```

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

```
  numeric_digits(100)
15
  4/19
0.2105263157894736842105263157894736842105263157894736842105263157894736842105263157
7894736842105263157
```

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);
{0=992008599211413363,
 1=-923325524698236965,
 2=-538113564003975731,
 3=4402717987856179625,
 4=-7602891879941326663}
```

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.);
{0=4, 1=2, 2=5, 3=2, 4=5, 5=10, 6=9, 7=5, 8=9, 9=1}
```

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)
{0=2viCMZn17CCBcnX4,
 1=ktA6eYqUeoSbthQA,
 2=w9dWMTFGVfGsWnSj,
 3=6fdP8wdYNXzcQFT9}
```

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. Note that this useful but cannot generally be used in, say, web traffic because the you have to preserve the exact character set used. 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'));
54686520717569636b2062726f776e20666f78206a756d7073206f766572207468652065206c617a7920646
f67
```

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 + indices(5);
  b. := 5 + indices(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.
{
 b.={0=5, 1=6, 2=7, 3=8, 4=9},
 a.={0=-5, 1=-4, 2=-3, 3=-2, 4=-1}
}
```

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}
```

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 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. Also, since stems end in a period, if an entry in a JSON object is another JSON object, that key will get a required period. If you use the to_json command, this will always return valid JSON with stem markers. The names of properties will be escaped.

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, *vencode* will be used on each key of the JSON object. In QDL, stem indices are variables but 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

- do not replicate integer indices as lists (it's fine to have stem.0. be an entry and the stem entry will be deserialized) but not both stem.0. and stem.0 and as long as this is observed
- 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)
```

Arguments

var = the string-valued variable to be converted

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=0, 1=1, 2=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"}

  b. := from_json(b)
  say(b., true)
{
 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
}

// And just to check it really is a stem
  b.foo.woof.
{a3tyu=ftf222gh, a3rty=456222, a3ghjjh=422256456}
```

A simple example of a stem them cannot be represented in JSON

In this case, there is an index clash and an index error is generated

```
  stem.0. := indices(3)
  stem.0 := 5
  // So let’s show out stem. Note the there is both a scalar and list entry:
  stem.
{0=5, 0.={0=0, 1=1, 2=2}}
  to_json(stem.)
Error: The stem contains a list element "0" and a stem entry "0.". This is not
convertible to a JSON Object
```

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]}');
  a.
```

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

A less contrived example.

```
  a. := from_json('{"Jäger-Groß":"enrolled"}')
  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.

```
while[
   for_keys(key, a.)
]do[
  if[
       key == vencode('Jäger-Groß’)
     ]then[
       // … 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.
```

list_keys

Description

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

Usage

```
list_keys(arg.)
```

Arguments

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

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.

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

```
var. := list_keys(sourceStem.);
```

The result would be

```
  var.0 == bind
  var.1 == find
  var.2 == bring
  var.3 == rule
```

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

Another example.

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.

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.));
{0=true, 1=true, 2=true, 3=true, 4=true, 5=false, 6=false, 7=false, 8=false,
9=false}
```

include_keys

Description

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

Usage

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

Arguments

var. – a stem

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

Output

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

Examples

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

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

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

indices

Description

Make a list of indices. This is very useful in conjunction with looping.

Usage

```
indices(arg);
```

Arguments

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

Output

A stem variable whose keys are the integers and whose entries are the same.

Examples

```
x. := indices(3);
```

results in

```
x.0 == 0
x.1 == 1
x.2 == 2
```

So a common pattern is

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

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=0, help=this is my stem, 1=1, 2=2, 3=3, 4=4}
  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
```

keys

Description

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

Usage

`keys(arg.)`

Arguments

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

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.

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}
```

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*

`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=0, 1=1, 2=2, 3=3, 4=4}
  stem2. := 5*indices(6);
  stem2.
{0=0, 1=5, 2=10, 3=15, 4=20, 5=25}
  list_append(stem1., stem2.)
{0=0, 1=1, 2=2, 3=3, 4=4, 5=0, 6=5, 7=10, 8=15, 9=20, 10=25}
  list_append(stem2., 'woof')
{0=0, 1=5, 2=10, 3=15, 4=20, 5=25, 6=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)
{0=-50, 1=-49, 2=-48, 3=-47, 4=12, 5=13, 6=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)
{0=-100, 1=-99, 2=-98, 3=-97, 4=22, 5=23, 6=24, 7=-96, 8=-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_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)
```

```
{0=22, 1=23, 2=24}
  // grab some stuff in the middle.
  list_subset(stem., 1, 3)
{0=21, 1=22, 2=23}
```

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
```

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.

Examples

```
  set_default(x., 1);
  say(x.);
{}
```

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

```
  x.0 := 10;
  say(x.);
{0=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=0}
```

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)));
{0=3, 1=45.34, 2=abc, 3={0=0, 1=1, 2=2}}
```

unbox

Description

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

Usage

```
unbox(stem.);
```

Arguments

stem. - the stem to unbox

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.

Examples

```
  a. := -5 + indices(10);
  b. := 5 + indices(5);
  a.arf := 'woof';
  b.woof := 'bow wow';
  c. := -20 + indices(3);
  union(a., b., c.)
{0=-20, 1=-19, 2=-18, 3=8, 4=9, arf=woof, 5=0, 6=1, 7=2, 8=3, woof=bow wow, 9=4}
```

Note that in this example these stems have some keys contained in the previous one.

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')
{0=scripts/, 1=other/, 2=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

Examples

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 o 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=0,
 1=1,
 2=2,
 3=3,
 4=4,
 5=5
}
```

Compare this with just plain printing it:

```
    say(f.);
{0=0, 1=1, 2=2, 3=3, 4=4, 5=5}
```

read_file

Description

Read a file. The result is always a string

Usage

```
read_file(file_name [, 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

`to_list` == 0 – return the result as a stem list each line separate

`is_binary` == 1 – return the result as a base 64 encoded string of bytes.

If no second argument is given, the result is simply a string of the entire contents of the file.

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 si flagged as binary.

Examples

```
cfg. := read_file('/var/lib/tomcat/conf/server.xml', 0);
```

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 := read_file('/var/lib/tomcat/conf/server.xml');
```

Which reads the same file and puts the entire thing in a single string.

```
my_b64 := read_file('/var/lib/crypto/keystore.jks' , 1);
```

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(read_file('/home/ncsa/dev/ncsa-git/security-lib/ncsa-qdl/src/test/
resources/hello_world.qdl',0));
{0=/*, 1=  The expected Hello World program.  , 2=  Jeff Gaynor, 3=  1/26/2020,
4=*/, 5=say('Hello world!');}

    // read the exact same file and turn the bytes into a base 64 string.
    say(read_file('/home/ncsa/dev/ncsa-git/security-lib/ncsa-qdl/src/test/
resources/hello_world.qdl',1));
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 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

```
write_file(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';
    write_file('/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.

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
```

load_script

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.

Usage

```
load_script(file_name)
```

Arguments

file_name – the fully qualified path to the file

Output

None

Examples

```
load_script('/home/bob/qdl/math_util.qdl');
```

run_script

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

`run_script(file_name)`

Arguments

`file_name` – the fully qualified path to the file

Output

none

Examples

`run_script('/home/bob/qdl/format_reports.qdl');`

General functions

These are functions that are generally applicable and do not fall in to the other categories.

is_defined

Description

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

Usage

`is_defined(`*`var`*`)`

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.

remove

Description

Remove a stem and its values from the symbol table

Usage

```
remove(var)
```

Arguments

var – a scalar or stem to be removed.

Output

none

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
```

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.0 := 5
  stem.
{0=5, 0.={0=0, 1=1, 2=2}}
  is_defined(stem.0)
true
```

```
  remove(stem.0)
true
  is_defined(stem.0)
false


  stem.
{0.={0=0, 1=1, 2=2}}
  is_defined(stem.0.)
true
```