

The Platform

Columnar Time-Series Database

Documentation

Version 0.7.0

theplatform.technology

Table of Contents

Introduction	7
Platform	
General	9
Installation	12
Usage	13
Tutorial	17
Language reference	
Overview	40
Types	
Overview	42
Scalars	
Overview	45
GUIDs	46
Symbols	47
Temporal types	48
Vectors	50
Strings	52
Dictionaries	54
Tables	
Creation	56
Indexing	57
Meta information	58
Arithmetics	59
Modification	60
Inserts	62
On disk	64
Partitioned tables	67
Signals	71
Reagents	
Overview	73
Async	76
Bus	77
Deq	78
File	79
Ipc	80
Kdb	81
Kdblistener	82
Listener	83
Log	84
Null	85
State	86
Sync	87
Timer	88
Tcp	89
Tty	90
Tls	91
Udp	92

Ws	93
Attributes	94
Pattern matching	98
Lambdas	99
Shadow	104
Assign	
Assign 0	106
Assign 1	108
Destructuring	109
Verbs	
Amends/Dmends	
Triadic @ (amend)	110
Triadic . (dmend)	112
Tetradic @ (amend)	114
Tetradic . (dmend)	115
Bitwise verbs	
band	116
bnot	117
bor	118
bxor	119
Casts	
cast	120
repr	122
flip	124
sv	125
vs	127
serialization	129
compression	130
Conditional verbs	
\$ (cond)	131
? (vector conditional)	133
& (and-cond)	134
(or-cond)	135
Database input/output	
Reading/writing concept	136
Projecting files concept	137
Execution flow	
load	139
defn	140
eval	141
reagent	142
react	143
spawn	145
top	146
get	147
close	148
sleep	149
yield	150
return	151
exit	152
kill	153
panic	154

I/O verbs

set	155
get	156
write	157
read	158
readln	159
print	160
println	161
show	162
ON! (display)	163

Indexing verbs

@ (at)	164
. (dot-apply)	165

List verbs

count	166
distinct	167
enlist	168
pick	169
fill	170
filter	171
first	172
last	173
reverse	174
rotate	175
shift	176
take	177
drop, cut	179
.o.cut	181
?	182
! (til)	183

Logical verbs

~ (not)	184
(or)	185
& (and)	186
& (where)	187

Math verbs

- (negate)	188
% (monadic division)	189
+	190
-	191
*	192
% (division)	193
mod (division remainder)	194
abs (absolute value)	195
ceil	196
floor	197
frac	198
& (min)	186
(max)	185
round	201
More math functions	202

Relational verbs

<, <=	203
>, >=	204

= (equal)	205
~ (match)	206
<> (not equal)	207
Null handling ^	208
Set verbs	
Intersection	209
Difference	210
Union	211
in (contains)	212
String manipulation	
like	213
ss (str search)	214
ssr (str search and replace)	215
parse	216
format	217
.o.lower	218
.o.upper	219
System	
argv	220
getenv	221
setenv	222
system	223
os	224
Type verbs	
@ (internal type id)	225
type (typespec)	226
! (internal type id)	227
.o.typedesc	228
Other verbs	
key	229
meta	230
quote	231
rc (reference count)	232
mv (move)	233
timestamp	234
value	235
Iterators	237
Queries	
SELECT	243
WHERE	248
DISTINCT	252
GROUP	253
LIMIT	256
ORDER	258
JOIN	259
UNION	261
UPSERT	262
UPDATE	264
DELETE	265
SQL-like syntax	269
Ptables support	270
Dynamic parsers	271

Language idioms	274
Standard library	
Overview	275
core.o	
Core functions	276
For tables	280
For partition tables	282
htreq.o	288
http.o	290
json.o	293
lit.o	295
markdown.o	298
prolog.o	301
repl.o	306
sql.o	309
urllib.o	320
xml.o	322
Plugins	
Overview	325
Customers plugins	326
Plugin crypto	332
Plugin ctrlc	333
Plugin fs	334
Plugin kdb	335
Plugin serde	336
For Python developers	337
For KDB+ developers	343
Application examples	350
FAQ	352
Contributors	355
Contacts	356

Introduction

Column-based and time-series databases are the most intensively developed kind of databases, especially during the last decade. Many products of that sort emerged to meet opportunities created by explosive growth of amounts of data generated by humankind. The Platform, among the others, is an effort to create a compact, convenient and high-performant environment to process, analyze and store both real-time and historical data. It reflects a vision of its developers of how proven and widely used products should develop to fit modern requirements.

What is The Platform?

The Platform is a columnar database engine designed from the ground up to handle both real-time streaming data and historical time-series workloads. Unlike traditional row-oriented databases, The Platform stores data in columns, enabling highly efficient compression, SIMD-accelerated scans, and cache-friendly access patterns. It is built with Rust and the O language, a modern vector programming language inspired by the array-processing tradition of APL, k, and q.

The Platform serves as a kdb+ alternative for teams that need real-time analytics without the licensing costs and complexity of legacy systems. It is designed to run on commodity hardware while delivering performance competitive with specialized in-memory databases.

Key Features

Columnar Storage and Compression. Data is stored column-by-column, enabling type-specific compression algorithms that dramatically reduce storage requirements and improve query throughput. Sorted, partitioned, and indexed columns allow sub-millisecond lookups even on datasets with billions of rows.

O Language. The Platform ships with O, a concise vector programming language. O provides first-class support for tables, time-series operations, SQL-like queries, and functional programming. Its terse syntax allows analysts and engineers to express complex transformations in a few lines of code.

SIMD and Multithreaded Execution. Queries are automatically parallelized across available CPU cores. Inner loops use SIMD instructions for operations such as filtering, aggregation, and arithmetic on large arrays. This ensures that The Platform fully utilizes modern multicore processors.

Real-Time and Historical Data. The Platform supports both in-memory real-time tables and memory-mapped historical partitions. Streaming inserts, pub/sub messaging, and on-disk persistence work together seamlessly, making it suitable for applications that require both low-latency ingestion and deep historical queries.

Built with Rust. Core components are implemented in Rust, providing memory safety, zero-cost abstractions, and predictable performance without garbage collection pauses.

Use Cases

- **Financial market data.** Tick-by-tick storage and analysis of quotes, trades, and order book snapshots with nanosecond-precision timestamps.
- **IoT and sensor networks.** Ingesting millions of data points per second from distributed sensors, with real-time dashboards and anomaly detection.
- **Log and event analytics.** Storing and querying structured log data for observability, performance monitoring, and security analysis.
- **Scientific and quantitative research.** Interactive exploration of large numerical datasets using vector operations and ad-hoc queries.

Quick Start

 [Tutorial](#)

 [O Language Reference](#)

 [Queries](#)

Platform

Platform uses work stealing as a task scheduling strategy and consists of three main components: `Runtime`, `Scheduler`, `Task`.

In a work stealing runtime, each scheduler has a queue of tasks to perform. Each task consists of a series of instructions to be executed sequentially, but in the course of its execution, a task may also spawn new task that can feasibly be executed in parallel with the rest of its work. These new tasks are initially put on the queue of the scheduler executing the current (parent) task. When a scheduler runs out of work, it looks at the queues of other schedulers and "steals" their tasks. In effect, work stealing distributes the scheduling work over idle schedulers, and as long as all schedulers have work to do, no scheduling overhead occurs.

Runtime

The purpose of Runtime is correct initialization and shutdown of workers' threads and schedulers. Also, it manages bootstrap or injector scheduler for creating root task.

```
extern crate rt;
use rt::runtime::{Runtime, spawn};

fn main() {
    let num_cores = 2;
    let stack_size = 1024;
    Runtime::new(num_cores, stack_size).run(|| {
        let hello = spawn(|| {
            print!("Hello ");
        });
        hello.join().unwrap();
        println!("world!");
    });
}
```

Usually, bootstrap scheduler is used as a dedicated processor to block operations (terminal, files or network processing).

Scheduler

Scheduler implements work stealing algorithm, manages tasks execution and rescheduling them if needed (more on this later). Under the hood uses Chase-Lev deque as a tasks queue. Usually, not directly accessible for an end-user.

Task

Tasks are implemented via stackful coroutine. They can be rescheduled after explicit `this::yield_task()` call or implicitly on bounded channel operations:

```
let (s, r) = channel(2); // two element channel

s.push(41);
s.push(42);
s.push(43); //wait until r.pop() 43

runtime::spawn(|| {
  let v = r.pop();
  assert_eq!(41, v);
  let v = r.pop();
  assert_eq!(42, v);
  let v = r.pop();
  assert_eq!(43, v);

  let v = r.pop(); // yield from task until next s.push(..)
});
```

Fork-Join Parallelism

Using `Runtime`, `Scheduler` and `Task` is enough to implement fork-join parallelism. We can build acyclic DAG of computation using `spawn` (fork) and `join` (join) primitives.

Channels and Pi-Calculus

Channels are a typed conduit through which you can send and receive values to/from other tasks. Values can be channels themselves as well (Pi-Calculus feature).

We have implemented several types of channel:

Bounded single producer-single consumer

```
use sync::channel::channel;

let (s, r) = channel(8);
s.push(42);

runtime::spawn(|| {
  let v = r.pop();
  assert_eq!(42, v);
});
```

Unbounded multiple producers-single consumer

```
use sync::mpsc::channel;

let (s, r) = channel(); // no capacity. unbounded!

let s1 = s.clone();
runtime::spawn(|| {
    s1.push(41);
});

let s2 = s.clone();
runtime::spawn(|| {
    s2.push(42);
});

let v1 = r.pop();
let v2 = r.pop();

println!("{}", v1, v2); // "[41, 42] or [42, 41]"
```

Publisher-Consumers

```
use queues::publisher::Publisher;

let mut publisher: Publisher<u32> = Publisher::with_capacity(8);

//write by 4 events at the same time
match publisher.next_n(4) {
    Some(vs) => {
        vs[0] = 0;
        vs[1] = 1;
        vs[2] = 2;
        vs[3] = 3;
        publisher.commit();
    }
    None => {} // not enough space for write
}

let subscriber = publisher.subscribe();
runtime::spawn(|| {
    loop {
        match subscriber.recv_n(4) {
            Some(vs) => {
                subscriber.commit();
                assert_eq!([0, 1, 2, 4], vs);
            }
            None => {this::yield_task();} // no events
        }
    }
});
```

Installation

Will be available soon. For now, please refer to a [contacts](#) page to request a binary.

Usage

Command-line usage

To control internal system logging `OLOG` environment variable is to be used. See supported values for this variables below.

Values are organized in levels. That is top level `error` is the least level of verbosity. `trace` level is the highest including all lower levels.

OLOG value	Meaning
error	Serious errors
warn	Warnings, things to pay attention to
info	Any useful information
debug	Debugging information
trace	Verbose logging used for debugging only

Usage example:

```
$ OLOG="info" tachyon
```

```
Tachyon platform 0.1.0
Build hash 529a35cf1baeb5f89f6d296687e0b8e3fd19d5b8
Build date 2022-02-14 14:26:50
Built with rustc 1.54.0-nightly (657bc0188 2021-05-31)
Build opts profile:debug dbg_asserts:on dbg_info:off
```

```
-----
CPU cores: 12
MEMORY: total: 16777216, free: 3793848, avail: 1423292
HOSTNAME: Serhiis-MBP
OS: Darwin
OS version: 21.2.0
Created pool with 1 cores
Thread stack size: 2048 kb
Lambda stack size: 4096 kb
Allocator pool size: 64 mb
Started with : []
```

```
-----
o){x+1}"123"
** runtime error: `+`:
"123"
1
** stack backtrace:
[0]: "REPL":1
>
  {x+1}
<
**
o)
```

`debug` level also controls emitting native stack dumps when platform is built with debug info. This is especially useful for debugging complex multithreaded system issues which is often next to impossible to reproduce.

```
$ OLOG="debug" tachyon
```

```

o){x+1}"123"
DEBUG rt      >
  0: backtrace::backtrace::trace_unsynchronized
  1: backtrace::backtrace::trace
  2: backtrace::capture::Backtrace::create
  3: backtrace::capture::Backtrace::new
  4: rt::backtrace
  5: base::interpreter::Interpreter::o_error
  6: base::interpreter::Interpreter::runtime_error
  7: dyad1::dyad::plus::{{closure}}
  8: dyad1::dyad::arith_temporal_pred
  9: dyad1::dyad::arith_temporal_xor
 10: _plus
 11: apply_dyad@@32
 12: _eval
 13: base::eval_sequence_guarded::{{closure}}
 14: rt::try_std::do_call
 15: ___rust_try
 16: rt::try_std
 17: base::eval_sequence_guarded
 18: base::eval_sequence
 19: monad::apply::lambda_in_frame
 20: monad::apply::lambda_call_iter
 21: monad::apply::lambda_in_frame
 22: apply_lambda@@32
 23: apply@@32
 24: _eval
 25: tachyon::repl::{{closure}}
 26: rt::try_std::do_call
 27: ___rust_try
 28: rt::try_std
 29: tachyon::repl
 30: _console
 31: tachyon::single
 32: tachyon::run
 33: tachyon::main
 34: core::ops::function::FnOnce::call_once
 35: std::sys_common::backtrace::__rust_begin_short_backtrace
 36: std::rt::lang_start::{{closure}}
 37: std::rt::lang_start_internal
 38: std::rt::lang_start
 39: _main

** runtime error: `+`:
"123"
1
** stack backtrace:
 [0]: "REPL":1
>
  {x+1}
<
**
o)

```

Query optimization level

Query optimization level can be set at start time using `--q0` parameter. It is implemented since 0.6.0 version.

It accepts 0..3 value like below. Default optimization level is 1. For details on query optimization levels - see [Select/Compilation options](#)

```
$ OLOG="info" tachyon --q0=3
```

Query parallel execution

Query parallel execution can be enabled at start time using `--qP` parameter. It is implemented since 0.6.0 version.

It accepts value "on" and "off". Value by default is "off". For details on query parallel execution - see [Select/Compilation options](#)

```
$ OLOG="info" tachyon --qP=on
```

Introduction

This tutorial is intended to help people who lack vector languages background and want to learn basic concepts step by step.

First start

Starting language interpreter is the first thing you will need. So just execute the following command in your shell terminal:

```
$ OLOG=info cargo run --release --bin tachyon
```

It will run the interpreter and present a greeting with its version and wait for your command in prompt.

```
Tachyonic platform 0.1.0
-----
CPU cores: 8
MEMORY: total: 32869648, free: 15155004, avail: 22692744
HOSTNAME: denis-devuan
OS: Linux
OS version: 4.16.0-2-amd64
Created pool with 1 cores
Thread stack size: 2048 kb
Started with file: REPL
-----
o)
```

Whenever you see `o)` line, you can safely assume the interpreter is waiting for your input.



Small hint - it's better to run the interpreter using "rlwrap" utility. It allows using arrow keys to recall history ("up" key) and editing it with "left" and "right" arrows.

```
$ rlwrap cargo run --release --bin tachyon
```

Simple expressions

Giving expressions to an interactive interpreter is like asking questions in order to get some answers. You give an expression to calculate - interpreter gives you answers or errors (if you are unlucky :))

Let's say, you are particularly interested in learning the result of "2+2" arithmetic expression. You can ask the interpreter! Type "2+2" in its prompt and hit Enter. It will show the following text:

```
o)2+2
4
o)
```

So, your expression got correctly calculated (or evaluated) and printed in session. In way, an interpreter is nothing more than an advanced calculator.


Let's reformulate ourselves a little to summarize - we enter expressions in the interpreter prompt and get answers in the session. Any answer in O language is called a "value". A value is something that gets created when calculation/evaluation happens. In fact, any syntax construction in the language is an expression, thus has a value.

Each expression having a value has important implication - it allows to chain and combine expressions to form more complex expressions. See the one below:

```
o)1+2+2
5
o)
```

All arithmetic expressions can be entered in O.

```
o)1+1
2
o)1-1
0
o)2*2
4
o)2%1
2
```

Division (the last expression) looks a bit different from  standard notation. It's done to free the "/" symbol for other expressions (we will discuss this later).

Value bindings aka "variables"

Having expressions and evaluating them into values is fun, but let's go one step further. Let's assign (or bind) names to values. It greatly increases comprehensibility of complex expressions.

For example, let's calculate circle area for given radius. Firstly, we will name "pi" constant value (not precisely though):

```
o)pi: 3.1415
3.1415
o)
```

Yes, creating a binding is done via "[name] : [value]" expression.



Do not use `_` at the beginning of names and avoid special characters in symbols altogether.

Ok. Thus, interpreter knows that name "pi" references to value 3.1415 Let's name radius as well.

```
o)radius: 3.0
3f
o)
```

Now, we will calculate the circle area:

```
o)pi*radius*radius
28.273500000000002
o)
```

One things to note here - numeric values with fractions (called floats) should be given explicitly. Using "3" will indicate another value type - the one without fractions called "integer".

Vector processing

Even though single values (scalars) are really useful for calculation, we might need something to keep lots of scalars in one value. A sequence of values placed in one value is called a vector. This value can be indexed by integer type values, so you can access vector per element, ask its length, etc.

Simple vectors

By simple vectors we mean that it consists of elements of the same type:

```
o)intvec:(1;2;3)
1 2 3
o)
```

This is a simple vector of integers. Parenthesis with elements separated by semicolons are a generic vector notation.

```
o)intvec:1 2 3
1 2 3
o)
```

... another simple vector of integers. Giving values separated by spaces is allowed as simplified notation for simple vectors only.

```
o)fvec: 1.0 2.1 3.2
1 2.1 3.2
o)
```

... simple vector of floats.

Simple vectors have a restriction though. Once you've created them, it's not possible to change types of their elements:

```
o)a:1 2 3
1 2 3
o)a[1]:1.0
** eval error: `amend vec`:
invalid type: [s`float]
```

Simple vector with only one element is represented using "enlist" verb notation.

```
o),1
,1
o)enlist 1
,1
o)
```

Lists

List or generic vectors is a vector that can keep values of different types in one structure.

```
o)l:(1;2.0;3;3.0)
1
2f
3
3f
o)
```

Lists are printed by placing each element on new line. Another peculiarity is that float values have special suffixes defining their "float" type.



Important: lists can be nested. It means that they can include other vectors and lists or any other types.

```
o)l: (1;1 2 3;(1;2.0))
1
1 2 3
(1;2f)
o)
```

O language does not have multidimensional vectors/matrices but they can be implemented using nested lists.

```
o)a:(1 2 3; 4 5 6; 7 8 9)
1 2 3
4 5 6
7 8 9
```

Lists are one of the main ways of grouping values of different types in one value. Nested lists also have "shape". Roughly speaking, it's a definition of their nesting structure.

For example:

```
o)a:(1; 2; 1 2 3)
1
2
1 2 3
o)b:(1; 1.0; 2)
1
1f
2
o)
```

... `a` and `b` are both lists, but they have different shape as their structure contain different value types in second elements.

Now let's see something that clearly separates vector languages from "ordinary" languages. It's an ability to process vectors/lists with compatible shapes as easily as simple scalars.

```
o)a:1 2 3;
o)b:4 5 6;
o)a+b
5 7 9
o)
```

We have just added two shape-compatible vectors using the same expression as for simple scalars!

Summing vector and scalar also works fine:

```
o)a: 1 2 3;
o)b: 1;
o)a+b
2 3 4
```

Scalars are implicitly "expanded" to be compatible with its "partner" vector.

Notably, nested vectors can also be summed:

```
o)a:(1;2;1 2 3)
1
2
1 2 3
o)b:(10;20;30)
10 20 30
o)a+b
11
22
31 32 33
```

That ability to go through nested lists' contents is defined by function that we apply to values. The `+` function is fully-atomic which means that it descends into both left and right arguments if they are nested lists.

User-defined functions

Another way to improve your expression quality is defining your own functions. Functions are meant to capture several expressions under one value. In this way, you can reference an expression by its name to use it multiple times. Applying arguments to a function causes its evaluation and thus function gets its value.

For example, following expressions define a function with one expression `x+y` and apply two values to the function. Curly brackets define nameless function ("lambda" expression if you will). `x`, `y` are implicit arguments which bind function arguments to names. Binding nameless function to a name allows to reference it later. Argument names are temporary name bindings and thus cannot be used outside function body. To apply a function, use square brackets and separate arguments with semicolons:

```
o)f:{x+y};
o)f[1;2]
3
o)
```

Full syntax for defining functions is:

```
o)f:[{a;b} a+b];
o)f[1;2]
3
o)
```

To recap, let's rewrite our circle area expression with a user-defined function:

```
o)area:[{radius} pi:3.1415; pi*radius*radius];
o)area[3.0]
28.273500000000002
o)
```

Several things are worth noting here:

- Last expression automatically becomes function value.
- Expressions are separated by semicolons and evaluated one after another. However, sub-expressions are evaluated from right to left (will be discussed in details later).
- Maximum number of arguments for any function is 8.
- Both "radius" and "pi" are temporary names (values bindings) and are called "local variables". Any value binding to a name causes local variable creation. Thus, it's not necessary to declare locals explicitly.

Contrary to the rule "last expression defines function value", you can force explicit function return using `:` expression.

```
o)f:{a:x+y; :a}
{a:x+y; :a}
o)f[1;2]
3
o)
```

A function is also a value and can be an argument and a result of another function:

```
o)f:{ {x+y} };
o)f2:f[]
{x+y}
o)f2[1;2]
3
o)
```

... functions as values are fully supported.

Terms and conventions

Let's improve our current knowledge of language terms and notions. We will use them later to define new concepts.

O language likes to separate functions according to several criterias. For example, by arity (number of arguments):

- Monads or monadic functions - those which accept just a single argument.
- Dyads or dyadic functions - those which expect 2.
- Triads and tetrads - those which expect 3 and 4, respectively.
- Polyads - those which accept different number of arguments. That separation is important to grasp "adverbs" discussed later.

By analogy with natural languages, O language calls built-in functions/primitives "verbs". Functions/verbs that accept other verbs/ functions as arguments are called "adverbs".

Types

O language is said to be dynamically strict typed language. It means that interpreter checks for types of values for compatibility before evaluating expressions. Incompatible types result in errors which happen at run-time. In this case, you get an error when interpreter gets right to expression evaluation.

Most of built-in verbs/functions in O are polymorphic. That is, the same verb might accept different combinations of types and shapes as its arguments. That greatly increases language expressiveness and terseness at the cost of being somewhat cryptic for newcomers.

Types themselves characterize values but not their bindings/names/variables. To denote that value has a specific type, O uses suffixes. Like `1i` - for 32bit integer values, `1f` - for floats with double precision, etc.

Simple arrays allow using shorter syntax:

```
o)1 2 3f
1 2 3f
o)
```

... creates float number vector.

For full reference see [Scalars](#).

Symbols and dictionaries

Using vectors and lists for organizing complex values is perfectly fine, but humans tend to prefer symbolic names instead of numbers as indices. Symbols in O serve this purpose. They are a special type of values which represent abstract names.

Remember binding value to a name using `:` verb? Name (left argument) was symbol in fact.

Symbols are created in O using backtick character and evaluated to themselves.

```
o)`red
`red
o)
```

Simple symbol vectors allow a bit easier syntax skipping spaces:

```
o)`red`blue`green
`red`blue`green
o)
```

Dictionaries represent a structure of key-value pairs. These are vectors indexed by non-integer key. Key is usually a symbol.

```
o)`red`blue`green!1 2 3
red | 1
blue | 2
green | 3
o)
```

To extract a value from a dictionary, use indexing:

```
o)d:`red`blue`green!1 2 3;
o)d[`blue]
2
o)
```

As you probably guessed, dicts in O are actually represented using two vectors under the hood. So indexing a dict is in fact searching for a given key in dict "key" vector and applying its index to the second vector.

Order of evaluation

Expressions in O are evaluated in a strict and unambiguous order - right to left. Arithmetic verbs are of no exception to that rule.

```
o)4*2+3
20
o)
```

If you want to ensure some expressions are evaluated in a particular order - use parentheses.

```
o)(4*2)+3
11
o)
```

Note that evaluation order of separate expression is still left to right and top to bottom.

```
o)f:{a:1+x;a*2};  
o)f[10]  
22  
o)
```

Verbs/adverbs overloading concepts

Vector languages are famous for their terseness and high expressiveness. Main concept behind that claim is their ability to "overload" names with meanings. E.g., most monadic verbs can be used in dyadic context as well but with different meaning. Different type combinations result in different meaning as well.

Let's see some examples. `#` verb in monadic context returns the length of the argument.

```
o)#1 2 3  
3  
o)
```

When used in dyadic context with left argument being boolean vector, it's a filter:

```
o)101b#1 2 3  
1 3
```

... or "take" if the left arg is an integer scalar:

```
o)2#1 2 3  
1 2  
o)
```

... or "reshape" if both left and right args are scalars:

```
o)4#2  
2 2 2 2  
o)
```

The overloading concept allows "packing" a lot of functionality into quite limited ASCII charset.

Of course, the same idea can be applied to user-defined functions. However, some restrictions exist here. Using fewer arguments than expected will result in an error in the current version. Check for argument types and/or shapes instead.

Indexing

To extract separate elements from vectors/lists, you need to index them. To do so, use square brackets (which is a common practice in programming). Index of first element of any vector is zero.

```
o)a:1 2 3;
o)a[1]
2
o)
```

Just as any other verb in O, it allows using vectors as indices (even nested ones).

```
o)a:1 2 3;
o)a[0 1]
1 2
o)a[[1; 0 2]]
2
1 3
o)
```

Now, let's imagine that you need to extract first 3 elements from a vector. You can easily enter 3 subsequent indices by hand but what if you need 500 indices? You can use `!` monadic verb to create 0...N-1 integers.

```
o)a:!100;
o)a[!10]
0 1 2 3 4 5 6 7 8 9
o)
```

Dict indexing works exactly the same way:

```
o)d:`a`b`c!1 2 3;
o)d[`c`b`a]
3 2 1
o)
```

Reduction aka folding

To get some statistics for a vector, let's calculate "average" function. More formally, it is a sum of vector divided by the vector length. Summing of vector is done by using "fold" (or "over") adverb.

```
o)a: 10 4 2 1 3 4;
o)+/a
24
o)
```

Next, divide the sum by vector length. Vector length is called `#` monadic verb in O parlance.

```
o)(+/a)%#a
4
o)
```

Now, let's create a separate function to calculate the average:

```
o)avrg: {(+/x)%#x}
{(+/x)%#x}
o)avrg[a]
4
o)
```

So far so good! However, let's try calculating the average of a vector with a fractional average.

```
o)a:10 11 11;
o)avrg[a]
10
o)
```

Hm, it does not look good. On the other hand, we entered integer vector and thus got integer average. What if we want to get float value instead? Let's force float division by "casting" integer sums and element count as doubles. It is done with the `$` dyad. Left argument is a destination type, right argument is a source argument.

```
o)avrg: {(`float$+/x)%`float$#x};
o)avrg[a]
10.666666666666666
o)
```

For full specification on `$` dyad see [Types](#).

Boolean values and vectors

Boolean values are often produced by relational verbs. 0 and 1's correspond to false and true values, respectively. O has following relational verbs: `<`, `>`, `>=`, `<=`, `=`, `<>`. All of them are fully-atomic, that is, both left and right arguments go through nested structures.

```
o)(1;2 3 4;5)>(1;2;4)
0b
011b
1b
o)
```

Well-known boolean algebra functions like "and", "or", "xor" are fully supported. Those are `&`, `|` and `<>`, respectively.

```
o)a:1 2 3; b:2 2 1;
o)(a>b)&a>1
001b
o)
```

Counting 1's in boolean vectors is easy, just use previously mentioned "fold"/"over" verb.

```
o)a:1 2 3;
o)+/a>=2
2
o)
```

Filtering is another useful verb. In the following example, it only returns values that are greater than or equal to 2.

```
o)a:1 2 3
1 2 3
o)(a>=2)#a
2 3
o)
```

Nulls and infinities

It's time to expand our knowledge about built-in types a bit. All scalar types in O have special value meaning absence of value. It is somewhat similar to zero in arithmetics. Each type has its own null value. E.g., long integer type null value is represented with `0N`, float double precision - `0f`, etc.

For full specification on nulls and infinities - see [Scalars](#). When a verb evaluates to the absence of something, it returns a null. For example, if you look for a non-existing element in a simple array, you will get a null of the array type.

```
o)a:1 2 3;
o)a[3]
0N
o)
```

If you look for a non-existing element in a general list, you will get a generic null:

```
o)a:(1;1 2)
1
1 2
o)a[2]
o)
```

... pay attention to the interpreter output. It is literally nothing! By the way, this is a way to create a generic null (since it lacks textual representation) - to index through an empty list.

```
o)()0
o)
```

Positive and negative infinities are related to numeric types only. They have specific notation - `0w` with a corresponding type suffix, like `0wf` (float with double precision), `0wj` (long integer), short notation is `0w` and `0W`, respectively. Again, for full specification on infinities - see [Scalars](#).

Infinities are a result of calculations:

```
o)1%0
0W
o)`short$100000
0Wh
o)
```

... casts also signal overflows with infinities.

Find verb

Search is a fundamental operation and `O` has a specific verb to perform it. It's `dyadic` `?`. Left argument is a value to be searched for, right argument is a value to look for. The verb evaluates to an index of the value being searched or null if not found. If several same values reside in the left arg, only index of the first occurrence is returned.

```
o)a:1 2 2 3
1 2 2 3
o)a?2
1
o)a?4 2 0
0N 1 0N
o)
```

"Find" is quite a versatile verb. It behaves differently depending on the shape of the left argument. The verb is right-atomic when the left argument is a simple vector.

```
o)a:1 2 3
1 2 3
o)a?(4;2 0)
0N
1 0N
```

If the left argument is a list and the right argument is a simple vector, the whole left argument is looked for.

```
o)a:(0;1 2;1 2 3)
0
1 2
1 2 3
o)a?1 2 3
2
o)
```

Both nested arguments result in matching each element of the right argument against each element of the left one:

```
o)a:(0;1 2;1 2 3);
o)a?(1 2 3;1 2;1)
2 1 0N
```

"Take", "drop" and "reshape" verbs

Although indexing in theory is enough to use all operations with vectors, it's often useful to manipulate vectors with simpler and more specialized verbs.

The "take" verb cuts off a part of a vector. Left arg defines how many elements to "take" and right arg is vector to cut.

```
o)2#1 2 3
1 2
o)
```

Negative left arguments mean "taking" from the end.

```
o)-2#1 2 3
2 3
o)
```

"Drop" does the reverse operation - it evaluates to remaining part of a vector.

```
o)2_1 2 3
,3
o)-2_1 2 3
,1
```

"Reshape" is used to literally transform the shape of its right arg into new one defined by the left arg.

```
o)10#0
0 0 0 0 0 0 0 0 0 0
o)
```

... creates a simple vector of 10 integer zeroes.

```
o)3 2#0
0 0
0 0
0 0
o)
```

... creates a nested vector of simple vectors.

```
o)4 2 3#1 2 3 4
(1 2 3;4 1 2)
(3 4 1;2 3 4)
(1 2 3;4 1 2)
(3 4 1;2 3 4)
o)
```

... and a nested vector made of a single simple vector elements. See how this verb cycles through the right argument elements?

Equality

Another important trait of operating on values is ability to check for equality. Conceptually, the O language supports two kinds of equality tests.

Fully atomic dyad "equal" (=) and "match" dyad (~). The first one requires compatible shape of the arguments and explores value equality deep into structure.

```
o)(1;2 3;4)=(1;3;4)
1b
01b
1b
o)
```

"Match" dyad returns only one boolean answer - if two values are exactly the same.

```
o)(1;2 3;4)~(1;3;4)
0b
o)
```

"Each" adverb

Functional programming pays much attention to ease of functions evaluation over value sequences. O language is of no exception.

Function evaluation over every vector element is sometimes called "mapping". That is, each vector element becomes an argument of a given function and all resulting values again form a vector. Adverb responsible for mapping is "each" (`'`). It takes monadic function plus vector and produces another vector.

```
o)a:1 2 3;
o){x*x}'a
1 4 9
o)
```

...calculates squares over vector elements.

"Each dyad" adverb is quite similar to "each" but it takes two input vectors and applies pair of values to a dyad function.

```
o)a:1 2 3; b:3 2 1;
o)a*'b
3 4 3
o)
```

"Each right" adverb is another variation of mapping. It takes two values and a dyadic function. The result of its evaluation is again an application of dyad to pairs of values. Pair of values are formed a bit differently though. Left argument is not iterated but provided as a whole. Only right argument elements are "mapped". Thus its name - "each right".

```
o)a:1 2 3; b:3 2 1;
o)a*/:b
3 6 9
2 4 6
1 2 3
o)
```

"Each left" adverb has almost the same meaning. Except this time left arg is being iterated and the right one is provided as a whole.

```
o)a:1 2 3; b:3 2 1;
o)a*\:b
3 2 1
6 4 2
9 6 3
o)
```

Exercise. Table of multiplication

Now, let's try out a simple exercise and ask the interpreter to create the table of multiplication for us. We simply need to take all integer numbers between 1 and 9 and multiply all possible pairs. Firstly, we generate a vector of 1 to 9.

```
o)n:1+!9
1 2 3 4 5 6 7 8 9
o)
```

... done.

And iterate over `n` using the "each right" adverb:

```
o)n*/:n
1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
o)
```

... or "each left". It produces exactly the same result:

```
o)n*\:n
1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
o)
```

So the overall solution is:

```
n:1+!9; n*/:n
```

12 symbols. Can we do better? Remember - the shortest solution is usually the fastest.

Yes, in fact we can solve it using just one complex expression.

```
n*/:n:1+!9
```

... 10 symbols. We used the rule that each expression in O produces a value, even a variable binding! So we compressed our solution by whole 2 symbols! Not bad, not bad...

Now, what about memory consumption? This is another important factor in assessing solution. Our current solution has one unpleasant side-effect - it leaves "n" name bind to a temporary 1,2,... vector and thus this temporary vector still occupies precious memory. Let's fix that:

```
{x*/:x}1+!9
```

... back to 11 symbols, but now no temporary vectors hanging around!

Flow control verbs

Changing flow of control is often used together with destructive verbs (see later) and with side-effect computations in general. It means that the evaluation expression does not only produce a value but also does some changes in overall environment (change bindings, vector/dict contents, etc).

One consequence of having side-effects is a necessity of introducing so-called "short-circuit" expressions. These are special expressions that evaluate their arguments on demand vs. before evaluating expression itself.

```
o)a:10;b:10;
o)$[a>5;a:11;b:12];
o)a,b
11 10
o)
```

You can see that the triadic "cond" verb does not behave as other ordinary (functional) verbs. "Functional" (with side-effects) verb would evaluate all its arguments and change "b" binding as well.

"True" value is anything except boolean false, integer zero and empty vector/list.

The extended form of "cond" verb might remind you the chained "if-else" from C, Pascal and other imperative languages. More formally it is `$(cond1; true_val1; cond2; true_val2; ... ; false_val)`. That is, "cond" evaluates each condition until boolean "true" or scalar value > 0 and evaluate corresponding "trueval" expression, otherwise "falseval" expression becomes the verb result.

```
o)$[0b;`first;0b;`second;1b;`third;`fifth]
`third
o)
```

Another flow control verb is `return` verb. It interrupts current function evaluation and returns a given value as function result.

```
o){a:x+1;:a;x}1
2
o)
```

... here `return` returns `a` binding value and does not evaluate the last `x` expression at all.

Destructive verbs

Destructive (imperative, non-functional) expressions include binding changes and content changes.

Dyadic `:` verb and `set` verb bind changes. `:` verb has more complex behaviour depending on scoping. It assigns values to local variables inside functions and to globals otherwise.

```
o){oo:1; oo}[]
1
o)oo
** runtime error: `undefined symbol`:
`oo
o)oo:2
2
o)oo
2
o)
```

`Set` verb does only global bindings. It expects binding name as a symbol in its first argument and its value in the second one.

```
o){set[`a;1]}[];
o)a
1
o)
```

Verbs that change contents are much more versatile. It is related to a possibility of using more complex value shapes.

But first let's discuss some conceptions behind changing the content (or mutation as it's often called). O being a functional language protects value integrity by employing copy-on-write technique. What does it mean in practice? Let's start with scalars:

```
o)a:10; b:a;
o)b+:1;
o)a,b
10 11
o)
```

... so far, nothing exceptional. `+:` is in fact a shortcut for `.[`a; ();++;1]` (see later) and means adding 1 to an existing `a` value and update the `a` binding. Scalars in O work exactly the same way as in other imperial languages.

Now, let's try vectors:

```
o)a:3#10;b:a
10 10 10
o)b+:1;
o)a,b
10 10 10 11 11 11
o)
```

... see the idea?

Destructive changes happen only when vector internal reference counter is equal to 1. It means that only one binding to vector value exists at a time. Otherwise, a copy is first made and the changes become local to that copy only. Even though the results on expression evaluation will be the same, it's always beneficial to apply destructive verbs to values that are not shared to avoid expansive copying.

In general, there are two kinds of content-modifying verbs: amend and dmend. Both amend and dmend can be triad and tetradic. Both of them first index value to get to its required element, apply user-provided function to that element and save the evaluation result back to value element.

When monadic function is applied to a value element, triad forms are used. Dyadic functions require an additional argument, that's why a tetradic form appear.

Indexing also divides into "horizontal" and "in-depth" indexing. "Horizontal" is denoted by the verb symbol `@` and "in-depth" one - by the dot symbol `(.)`.

Destructive application is expected when first argument is a binding name represented with a symbol, otherwise - changes are applied to a copy.

Let's try to play with them:

```
o)a:1 2 3;
o)@[a; 1; -:]
1 -2 3
o)a
1 2 3
o)
```

... read it like this - apply monadic verb "negate" to an element with index 1 of the `a` copy.

```
o)a:1 2 3;
o)@[`a; 1; -:]
`a
o)a
1 -2 3
o)
```

... this time - destructive. Read it like this - apply destructively monadic verb "negate" to an element with index 1 of `a`.

Now, the same exercise but with indexing "in-depth" using "dmend":

```
o)a:(1; 1 2 3);
o).[a; 1 1; -:]
1
1 -2 3
o)
```

And destructively:

```
o)a:(1; 1 2 3)
1
1 2 3
o).[`a; 1 1; -:]
`a
o)a
1
1 -2 3
o)
```

Applying dyadic is easy after grasping the concept:

```
o)a:1 2 3;
o)@[a; 1; +; 1]
1 3 3
o)
```

... and its "physical meaning" - apply dyadic verb "plus" to 1 and an element with index 1 of `a` copy.

```
o)a:1 2 3;
o)@[`a; 1; +; 1]
`a
o)a
1 3 3
o)
```

... and its "physical meaning" - apply destructively dyadic verb "plus" to 1 and an element with index 1 of `a`. And a short form for this is:

```
o)a:1 2 3;
o)a[1]+:1;
o)a
1 3 3
o)
```

"Dmend" with dyadic:

```
o)a:(1; 1 2 3);
o).[a; 1 1; -; 1]
1
1 1 3
o)
```

... and it's destructive form:

```
o)a:(1; 1 2 3)
1
1 2 3
o).[`a; 1 1; -; 1]
`a
o) a
1
1 1 3
o)
```

One special form here is applying dyadic to the entire value:

```
o)a: 1 2 3;
o).[`a; (); -; 1]
`a
o)a
Ø 1 2
o)
```

Actually, we've already seen its short form earlier:

```
o) a: 1 2 3;
o)a-:1
`a
o)a
Ø 1 2
o)
```

Signalling and error handling

Signalling is similar to exception raising in other languages. It interrupts normal evaluation order and "throws" a value across all nested functions until someone catches it or until REPL when it is just shown. The `'` (tick) monadic verb is responsible for "throwing" its argument.

```
o){'x}`err
** signal error: `panic`:
err
o)
```

Error handling in O is done again with `@` and `.` triad. Yes, those are the most overloaded verbs in language. First argument defines a function to evaluate, the second one - its argument(s), the third one - a result in case of signalling or function catching the signal value. If no signalling occurred, the verb returns only the result of function evaluation.

```
o)@[x];1;`err2]
1
o)@[x];`err1;`err2]
`err2
o)@[x];`err1;{x}]
kind | `signal
call | "panic"
message| `err1
mark | `linum`offset`length!1 0 2
o)
```

... the `@` form expects a monadic as first argument, thus, the second argument is passed as a whole.

The `.` form expects a list that can be nested for all function parameters.

```
o).[+;(1;2);`err2]
3
o).[x+y];(1;2);`err2]
3
o).[y];(1;`err1);{x}]
kind | `signal
call | "panic"
message| `err1
mark | `linum`offset`length!1 0 2
o)
```

Scripts

A script is an O program stored in a text file on disk. Typically O programs have ".o" file extension.

Just provide your script filename as a first argument and instead of continuously evaluating expressions from REPL, interpreter executes the script and stops after evaluation completes.

For example:

```
$ cargo run --bin tachyon -- -f etc/factorial.o
Tachyonic platform 0.1.0
-----
CPU cores: 8
MEMORY: total: 32869648, free: 14945768, avail: 23050972
HOSTNAME: denis-devuan
OS: Linux
OS version: 4.16.0-2-amd64
Created pool with 1 cores
Thread stack size: 2048 kb
Started with file: etc/factorial.o
-----
2432902008176640000
```

Overview

History

The O language is not really new, it is more of a logical evolution step for the APL-like languages. When it comes to ideological lines, O is a successor of K family (Arthur Whitney) of programming languages. However, it doesn't share a single line of code with K5 and is implemented in Rust instead of C. O's development started in Dec 2016 and the language has been redesigned 5 times so far. Current design seems to be polished enough to be considered as 'stable'.

Typographic Conventions

Example	Description
Tutorial	Links you can click to go to another page
{ distinct }	In syntax, optional items are enclosed in tortoise shell brackets. Do not type the brackets.
< expression >	In syntax, text within angle brackets represents items you should replace with information appropriate to your situation. Do not type the brackets.
<table> <join chain>	In syntax, when you must choose between items, only one.
{ <cols> <expr> }	In syntax, when you can select any combination of the items, or no item. Do not type the brackets and vertical bar.
{..}	In the syntax, it indicates the required number of elements similar to immediately before.
{; ..}	In the syntax, it indicates the required number of elements separated by a semicolon similar to immediately before.
<...>	In the syntax, the ellipsis indicates something code or text.



Section for additional important information.
All pages with such sections you can find by word `notice`.



Section for warning information.



Section for critical information.

You can find all pages with warning and critical sections by word `warning`.

Code Conventions

Example	Description
<code>o)1+1</code>	o) the prompt of REPL. It is not necessary to write it.
<code>//find sum</code>	Code comments

Types, casting, etc

Types and related verbs

Each expression in O has its type. Type in O defines value domain/set of supported values and is heavily to define the polymorphic behaviour of verbs. Internally it's coded using special undocumented integer value:

```
o)@1
320
o)
```

Monadic verb `@` returns the numeric representation of internal type id. Interpreter uses its exact value internally.



Avoid relying on internal type ids' exact values in your programs. They may change in any later version.

The only valid operation on numeric type representation is checking for equality.

```
o)(@2)=@1
1b
o)
```

Now, what's a better practice of type referencing if not using internal ids? That's what `!` monadic verb is for.

```
o)!`s`int
256
o)!`s`long
320
o)
```

That way you are referencing type id of scalar int. A single argument for `!` is called *type spec*. First symbol in type spec defines structure (scalar, vector, etc), second symbol defines scalar type name. For other scalar type names, see [Scalars](#)

Encoding structure in type spec is as follows:

Structure	Typespec	Example
Scalar	`s	`s`bool
Vector	`v	`v`int

Using the `.o.typedesc` you can find the type that corresponds to an integer type id:

```
o).o.typedesc 320
`s`long
o).o.typedesc[@1 2 3i]
`v`int
o)
```

Another useful monadic verb for getting type spec is `type`. It returns type spec for a given value:

```
o)type 10#0
`v`long
o)type 0
`s`long
o)
```



The `!` verb has one nice property - it might be evaluated at parse time if its argument is constant.

This example just assigns constant integer value at runtime:

```
o)a:!`v`int
45312
o)
```

Casting

Type casting in O is done using `$ dyadic`. Its left argument defines "destination" type, right argument is the "source".

Giving a single symbol as type spec means - *leave right argument structure intact and just change its element type*.

```
o)`int$10 20 30
10 20 30i
o)`symbol$10 20 30
`10`20`30
o)`float$10 20 30
10 20 30f
o)
```

For element type names, see [Scalars](#).

A better practice to define full type is using the `!` verb:

```
o)(!`s`int)$10
10i
o)(!`v`int)$10 20 30
10 20 30i
o)(!`s`float)$0
0f
o)
```

Here is an idiomatic way to ensure that two vectors have the same type:

```
o)a:1 2 3; b:10 20 30i; (@b)$a
1 2 3i
o)
```

And yes, an internal type id can be given as left argument but it's better to use it only in REPL.

```
o)a:1 2 3; b:10 20 30i; (@b)$a
1 2 3i
o)
```

Beware of collapsing lists in cases like:

```
o)`int$(1;2.0;3)
1 2 3i
o)
```

Casting and over/underflows

When casting, you can encounter values that are too large or too small to be held by type. Infinities are used to signal that:

```
o)`int$1000000000000 -1000000000000
0W -0Wi
o)
```

Nulls/NaN are retained between types.

```
o)`int$1.0 0n 0w
1 0N 0Wi
o)
```

Scalars

Scalars are value types. They contain their payload directly and don't need boxing.

- Characters are enclosed in double quotes `"` and can make use of the escape sequences `\n`, `\t`, `\"` or `\\` to produce a newline, tab, double quote or backslash character, respectively. If more than one unescaped character is enclosed in quotes, the noun is a list of characters (see below), also known as a string.
- Symbols start with a backtick ``` and are followed by an optional name. Names must start with a dot or a letter, and may contain letters, digits or a dot. Symbols are mainly useful as handles for variable names or keys in dictionaries.

O has following scalars:

Type	Size	Typespec	Value scalar/vector	Null	Infinity
Bool	1	<code>`bool</code>	1b or 0b / 100b		
Byte	1	<code>`byte</code>	1x / 1 1x	0Nx	0Wx
Short	2	<code>`short</code>	1h / 1 1h	0Nh	0Wh
Int	4	<code>`int</code>	1i / 1 1i	0Ni	0Wi
Long	8	<code>`long</code>	1 / 1 2 3	0N / 0Nj	0W / 0Wj
Symbol	8	<code>`symbol</code>	<code>`a</code> / <code>`a`b`c</code>	<code>`</code>	
Char	1	<code>`char</code>	NYI / "abc"		
Enum		NYI	<code>`sym\$a</code> / <code>`sym\$a`b`c</code>	<code>`sym\$`</code>	
Int128	16	NYI	NYI		
Guid	16	<code>`guid</code>	0Ng	0Ng	
Single	4	<code>`real</code>	1.0e / 1 1e	0Ne	0We
Double	8	<code>`float</code>	1.0 / 1 1f	0n / 0Nf	0w / 0Wf
Quad	16	NYI	NYI		
Timestamp	8	<code>`timestamp</code>	2020.11.05D12:30:21.123456789	0Np	0Wp
Timespan	8	<code>`timespan</code>	7614D12:30:21.12345679	0Nn	0Wn
Datetime	8	<code>`datetime</code>	2020.11.05T12:30:21.123	0Nz	0Wz
Date	4	<code>`date</code>	2021.01.01	0Nd	0Wd
Month	4	<code>`month</code>	2021.01m	0Nm	0Wm
Time	4	<code>`time</code>	12:30:21.123	0Nt	0Wt
Minute	4	<code>`minute</code>	12:30 or -12:30	0Nu	0Wu
Second	4	<code>`second</code>	12:30:21	0Nv	0Wv
Generic (used for casts)		<code>`s`</code>		0N0 - generic null	

Guids

Provide support for Universally Unique Identifiers (UUIDs). A guid type is a unique 128-bit number stored as 16 octets. It is used to assign unique identifiers to entities without requiring a central allocating authority.

Guid can be created from a string:

```
o)"G"$"61e35154-10bc-a49a-d11f-f10e1a377000"  
61e35154-10bc-a49a-d11f-f10e1a377000  
o)
```

... or generated from scratch:

```
o)-2?0Ng  
16bf5b77-9713-4061-92a3-750083b68307 5209d2c9-956a-4232-9a85-26afa5168d96  
o)
```

Null guid:

```
o)0Ng  
0Ng  
o)
```



There is no literal entry for a guid, it has no conversions and the only scalar primitives it supports are =, < and > (similar to sym).

There are no other limitations on storing, serializing guides, etc. in contrast to other O types.

strings

Symbols

Symbols are entities similar to those of Lisp. They mostly serv the same purpose - referencing variable names and representing keys in dictionaries.

They are defined by backticks followed by an optional name. Names can contain UTF8 letters, digits and dots:

```
o)`a
 `a
o)`abc
 `abc
o)
```

Symbol vectors:

```
o)`a`b`c
 `a`b`c
o)
```

A dictionary with a symbol vector as a set of keys:

```
o)`x`y`z!(1 2 3)
xl 1
yl 2
zl 3
o)
```

A symbolic file handle used to refer to a file:

```
o)`:folder/file.txt
 `:folder/file.txt
o)
```

Temporal types

There are eight temporal times in O:

Type	Type letter	Typespec	Example
Timestamp	"p"	`timestamp`	2020.12.03D11:49:00.505770835
Timespan	"n"	`timespan`	7614D12:30:21.12345679
Datetime	"z"	`datetime`	2020.12.03T11:49:00.505
Date	"d"	`date`	2020.12.03
Month	"m"	`month`	2021.01m
Time	"t"	`time`	11:49:00.505
Minute	"u"	`minute`	11:49 or -11:49
Second	"v"	`second`	11:49:00

Current timestamp

Getting current date and time is done using standard `ts` intrinsic.

```
o)ts[]
2020.01.28D11:52:53.574067653
```

Casting examples

```
o)p: 2020.12.02D15:51:57.123456789;
o)`timespan$p
7641D15:51:57.123456789
o)`datetime$p
2020.12.02T15:51:57.123
o)`date$p
2020.12.02
o)`month$p
2020.12m
o)`time$p
15:51:57.123
o)`minute$p
15:51
o)`second$p
15:51:57
```

How to get a year, month, day, hours, minutes, seconds, milliseconds and nanoseconds?

With `cast` you can get any part of temporal scalar as `int`.

```
o)p: 2022.04.22D01:02:03.123456789;  
o)`dd$p  
22i  
o)`year`mm`dd$p  
2022 4 22i  
o)`hh`uu`ss`ms`ns$p  
1 2 3 123 123456789i  
o)
```

Helpful

```
o)p: ts[]; type p-p  
`s`long  
o)n: "n"$p; type n-n  
`s`long  
o)z: "z"$p; type z-z  
`s`long
```

```
o)d: "d"$p; type d-d  
`s`int  
o)m: "m"$p; type m-m  
`s`int  
o)t: "t"$p; type t-t  
`s`int  
o)u: "u"$p; type u-u  
`s`int  
o)v: "v"$p; type v-v  
`s`int
```

 [Scalar types](#)

 [timestamp](#)

Vectors

Vectors are just sequences of same type values. They consist of typed arrays with such benefits as constant-time indexing and SIMD instructions used for parallel computing.

The only exception is generic vectors or simply Lists. Each item of a list can contain any datatype in AST.

List have ``v`1` type spec.

A vector can be created from a sequence of scalars separated by spaces:

```
o)1 2 3
1 2 3
o)`a`s`d
`a`s`d
o)1000101b
1000101b
o)(1;2 3;"foo";`bar)
1
2 3
"foo"
`bar
o)
```

Vectors can form dictionaries and tables:

```
o)`a`b`c!(1 2 3)
a| 1
b| 2
c| 3
o)([]a: 1 2 3; b: 1.1 2.2 3.3)
a b
-----
1 1.1
2 2.2
3 3.3
o)
```

Enumerated vectors

Conceptually, an enumerated vector is a subset of a symbol vector.

Each enumerated vector has additional information - domain symbol. Domain symbol defines global (usually) variable name containing a vector of symbols that form a domain. Only symbols belonging to this domain can appear in enumerated vector.

This results in an important advantage - each enumerated vector item can be represented with a fixed-size integer. Technically, each enumerated vector is i32 vector. This results in quick indexing and is especially beneficial for mapped vectors.

An example below demonstrates "sym" domain with 3 symbols and creation of enumerated vector "a" with 4 items.

```
o)sm:`a`b`c; a:`sm$a`b`c`a; a
`sm$a`b`c`a
o)
```

... if you try casting a symbol vector with symbols that are not present in domain, you will receive an error:

```
o)sm:`a`b`c; `sm$a`d
** runtime error: `cast: invalid value.`:
true
o)
```

On the contrary, null symbol is considered to be a part of each domain:

```
o)sm:`a`b`c; a:`sm$a``c; a
`sm$a``c
o)
```

Enumerated symbols require special treatment when saved to disk. An enumerated vector has no domain defined after "getting" it from file. Thus, you need to save domain values to disk as well. Casting enumerated vector "b" with domain "sym" assigns domain to vector like this:

```
o)a:`sm$c`b`a;
o)f:`:/tmp/o_enum1.dat; f set a;
o)fsym:`:/tmp/o_sm_enum1.dat; fsym set sm;
o)b:get f; sm: get fsym;
o)b:`sm$b;
o)b
`sm$c`b`a
o)
```

Strings

String is a special type in O. Technically it is an immutable vector of bytes. Strings are always stored in UTF-8 format. Thus, each symbol occupies a variable number of bytes.

Being a variable coding limits supported functions for strings to:

- getting length of a string in bytes;
- concatenation;
- packing to/unpacking from a byte vector and UTF-8 scalar vector;
- serving as scalar in special cases.

Length of string

Getting string length in bytes is done with a standard `#` monad:

```
o)#"111a6B"  
9  
o)
```

Concatenation

Since concatenation is easy in UTF-8 format, it is supported using a common `,` dyad and works as expected:

```
o)"111a6B", "222"  
"111a6B222"  
o)
```

Packing/unpacking

If you are absolutely sure that a string contains only ASCII characters, you can cast string into vector of bytes and back again.

```
o)"x"$"111"  
0x313131  
o) "c"$0x313131  
"111"  
o)
```

Another way to do generic UTF-8 string unpacking/packing is doing it into/from UTF-8 scalar values. It results in a vector of 32-bit integers. Its usefulness is questionable though.

```
o)`v`int$"aбв"  
1072 1073 1074i  
o)`v`char$1072 1073 1074i  
"aбв"  
o)
```

Strings as table initializers

Due to frequent usage of strings in tables some shortcuts have been made to ease it.








E.g., when inserting strings into tables, they behave as if they were scalars:

```
o)t:+`a`b`c!(,1;,2;, "123");  
o).[`t; (); ,; (10;20;"456")];  
o)t  
a b c  
-----  
1 2 "123"  
10 20 "456"  
o)
```

Search by strings

Another shortcut to avoid too much casting is string search inside a list of strings:

```
o)("123";"456";"789")?"456"  
1  
o)
```

-  [like](#)
-  [string search](#)
-  [string search and replace](#)
-  [parse](#)
-  [print](#)
-  [println](#)
-  [format](#)

Dicts

A dict is a mapping between two vectors of the same length: keys and values. The syntax for creating a dictionary uses an exclamation mark:

```
o)`a`b`c!1 2 3
a| 1
b| 2
c| 3
o)`a`b`c!1 2
** eval error: `!`:
arguments length mismatch: [`a`b`c;1 2]
o)
```

A dictionary can be decomposed into key and value vectors:

```
o)d:`a`b`c!(1 2 3)
a| 1
b| 2
c| 3
o)key d
`a`b`c
o)value d
1 2 3
o)
```

To look up a value in dict by key use square brackets:

```
o)d[`c]
3
o)
```

If a value is not in a dict, look up will result in a null of the present values' type:

```
o)d[`z]
0N
o)
```

You can flip a dictionary to get a table and vice versa:

```
o)flip `a`b`c!(1 2 3)
a b c
-----
1 2 3
o)flip ([a:1 2 3;b:1.1 2.2 3.3])
a| 1 2 3
b| 1.1 2.2 3.3
o)
```

Keys in dictionaries can be non-unique, but look-up will only return the first occurring value:

```
o)d: `a`b`c`a!(1 2 3 4); d[`a]
1
o)
```

Both keys and values can be nested lists but this should be taken into account while looking up value:

```
o)d:(`a`b; `c`d`e; enlist `f)!1 2 3
`a`b | 1
`c`d`e| 2
,`f | 3
o)d[enlist `f]
3
o)d[`f]
0N
o)d[`a`b]
1
o)d[`b]
0N
o)
```

 [flip](#)

 [!\(internal type id\)](#)

 [!\(til\)](#)

 [key](#)

 [value](#)

Tables

A table is similar to a dict and represents relational tables. Keys as column names correspond to values as simple vectors/sequences.

Table creation

Syntax for creating tables at parse time is:

```
o)([]a:1 2 3;b:4 5 6;c:7 8 9)
a b c
-----
1 4 7
2 5 8
3 6 9
o)
```

Only constant expressions for field vectors are allowed.

Another way to create a table is flipping a dictionary:

```
o)(+:)`a`b`c!(1 2 3;4 5 6;7 8 9)
a b c
-----
1 4 7
2 5 8
3 6 9
o)
```

For compatibility with Q, the "flip" verb is also supported.

```
o)flip `a`b`c!(1 2 3;4 5 6;7 8 9)
a b c
-----
1 4 7
2 5 8
3 6 9
o)
```



Observation here is that flipping the dictionary is a constant time operation. Technically, it's a new "table" type structure created with references to dictionary contents.

[Dictionaries](#)

[next >>> Indexing](#)

Table indexing

Indexing with an integer scalar returns a dictionary of values at corresponding "record" position:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); a[1]
a| 2
b| 2
c| 2
o)
```

... while indexing with an integer vector returns a table of records at positions:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); a[2 1 0]
a b c
-----
3 3 3
2 2 2
1 1 1
o)
```

 <<< prev Creation

 next >>> Meta information

Table meta-information

To get table fields, use monadic `!` verb:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); !a
`a`b`c
o)
```

Asking for table "values" results in a list of table vectors:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); value a
1 2 3
1 2 3
1 2 3
o)
```

Every table can be queried for its length - the amount of "records" it contains:

```
o)a:(+:)`a`b`c!(1 2 3 4;1 2 3 4;1 2 3 4); #a
4
o)
```

Designated `meta` verb returns column names, types, internal typed ids, attributes and list of fields of composite index for a table:

```
o)t:(+:)`a`b`c!(`asc#!5;`g#!5;!5); @[`t;,,`a;~[#];`g];
o)meta t
+`column`type`id`attr!(`a`b`c;`long`long`long;176448 438592 45376;`asc`g`)
(,`a)
```

[attributes](#)

[<<< prev Indexing](#)

[next >>> Arithmetics](#)

Arithmetics on tables

Applying arithmetic verbs on tables results in expected behaviour:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); a+1
a b c
-----
2 2 2
3 3 3
4 4 4
o)
```

Adding vector to a table gives per field addition.

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); a+1 2 3
a b c
-----
2 3 4
3 4 5
4 5 6
o)
```

Adding two tables keeps the record count:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); a+a
a b c
-----
2 2 2
4 4 4
6 6 6
o)
```

 [<<< prev Meta information](#)

 [next >>> Modification](#)

Table modification

Table amend/dmend work mostly the same as for dictionaries. Following example increases the "b" field by 1 for records at positions 0 and 2.

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); .[a;`b;0 2);+;1]
a b c
-----
1 2 1
2 2 2
3 4 3
o)
```

To create a new integer field of ones, do:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); @[a;`d;;1]
a b c d
-----
1 1 1 1
2 2 2 1
3 3 3 1
o)
```

You can also combine primitive function application with creating a new field:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); @[a;`a`b`d;+;1]
a b c d
-----
2 2 1 1
3 3 2 1
4 4 3 1
o)
```

Multiplication also works:

```
o)a:(+:)(,`a)!(1 2 3); @[a;`a`b;*;2]
a b
---
2 2
4 2
6 2
o)
```



You can see that the "initial" value for addition and multiplication is different. Indeed, assign/plus/minus dyads assume zero value as initial, mul/div take one for initial.

... however, the following code won't work, as interpreter does not try to analyze lambda result types.

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); @[a;`a`b`d;{x+y};1]
** runtime error: `amend`:
fields: invalid table field
```

More complex example shows creating of a new field and vector addition as single amend expression:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); @[a;`a`b`d;+(4 4 4;1;2)]
a b c d
-----
5 2 1 2
6 3 2 2
7 4 3 2
o)
```

Table modification caveats

Version 0.5.0 introduced extra logic to avoid breaking tables. Before it was possible to amend table, changing table field into non-compatible shaped scalars/vectors. That would lead to platform process crashed eventually.

That behaviour was fixed. However that means additional steps:

- Complex amends now make changes to a table copy.
- After all fields updated an extra validation is made.
- If incompatible field shapes detected - an error is raised and no updates actually applied.
- Otherwise - table is being modified, indices changes applied and on-disk changes occur.

e.g. following amend is rejected correctly in 0.5.0+

```
o) a:(+:)`a`b`c!(!3;10+!3;20+!3);
o) @[`a;`a`b`c;:(!3;!4;!4)];
** eval error: `amend`:
arguments length mismatch: [field length]
```

Extra logic for checking new fields is quite heavy on resources, thus it's applied only if verbs used in 3d amend/dmend argument are potentially shape-changing:

- user lambdas
- monads - "flip", "count", "tp", "first", "last", "enlist", "distinct", "where", "sersync", "desync", "sum", "avg", "min", "max"
- dyad or commute dyads - "assign", "take", "drop", "at", "matches", "sect", "filter", "remove", "diff", "union", "int_d"

That's means, ideally you should avoid making amends/dmends on tables for given verbs if you are for performance/RAM usage.



"Concat" verb is specifically optimized to avoid making temporary tables copies. So it's safe for performance.

[<<< prev Arithmetics](#)

[next >>> Inserts](#)

Table inserts

Table insert is a main operation for appending new records to tables. Both destructive and functional inserts are done using "concat" verb.

Single record insert/append. See yourself:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); a,(1 2 3)
a b c
-----
1 1 1
2 2 2
3 3 3
1 2 3
o)
```

Append with scalars extended:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); a,(1 2 3)
a b c
-----
1 1 1
2 2 2
3 3 3
1 2 3
o)
```

And inserting via dictionary having same fields:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); a,`c`b`a!(1 2 3)
a b c
-----
1 1 1
2 2 2
3 3 3
3 2 1
o)
```

Obviously, just as with dicts, concatenating tables works fine:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); a,a
a b c
-----
1 1 1
2 2 2
3 3 3
1 1 1
2 2 2
3 3 3
o)
```

Appending records is as simple as:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); .[`a;();,(10;20;30)]
`a
o)a
a b c
-----
1 1 1
2 2 2
3 3 3
10 20 30
o)
```

Or using shorter syntax:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); a,:(10;20;30)
`a
o)a
a b c
-----
1 1 1
2 2 2
3 3 3
10 20 30
o)
```

 <<< prev Modification

 next >>> On disk

Tables on disk

You can save the table to disk as a "serialized table" or "splayed table". It is easy to store complex data in serialized tables, but such tables are entirely `get`ting into memory, and there is no way to work with such tables larger than RAM.

```
o)t: +`simple`complex!(1 2; (`a`b`c!13; `c`d!(1 2; 3 4)));
o)`:./tmp/stbl set t;
o)tbl: get `:./tmp/stbl
simple complex
-----
1      `a`b`c!0 1 2
2      `c`d!(1 2;3 4)
o)
```

"Splayed table" in O means just the same as in Q/KDB - saving each field vector in a separate continuous file capable of mmaping it.

"Mmapping" allows to avoid `load`ing all table contents in memory in one go, but to work with a table off the disk (by using OS memory-mapped files technique).

Supported field types

Only following field types are supported in splayed tables:

- Vectors excluding symbol vectors.
- Compound lists - lists containing primitive scalars (all scalars except GUIDs) and/or vectors excluding symbols vectors. Nested lists are not supported.



Version 0.6.0+ supports scalar GUIDs in compound lists.



Convert symbol vectors into enum vectors first to be able to save them in splayed table. You can use `.o.en` function for that.

Saving on disk

Saving table values to disk is pretty simple. Just choose directory to keep your table files and execute something similar to:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); f:`:/tmp/o_table/; f set a;
o)
```



Important! Please pay attention to trailing slash at the end of the path symbol. That defines saving table as a splayed table.

Doing that creates `/tmp/o_table` directory filled with files named ".d", "a", "b" and "c". These files corresponding to fields and the ".d" file which contains symbol vector `a`b`c to keep the field order of the table.

Please pay attention that only fields mentioned in ".d" file are considered as table fields. Thus even if files are in the table directory, but not in the ".d" file, they will be ignored.

Use "get" function to load table contents:

```
o)f:`:/tmp/o_table/; get f
a b c
-----
1 1 1
2 2 2
3 3 3
o)
```



Getting table contents results in a separate/full copy in memory. That is modifying loaded table does not influence the copy on disk in any way.

On-disk table modifications

Modifying table on disk can be done in two ways:

- Amending via a path symbol.
- Opening table in workspace.

Amending via a path symbol is useful e.g. for adding a new field.

```
o)// creating test table
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); f:`:/tmp/o_table2/; f set a;
o)// append two new fields
o)@[f;`e`d;;(3#1;3#2)];
o)// modify .d file to record new fields
o).[`:/tmp/o_table2/.d;();,;`e`d];
```

Appending records via path symbol is ok.

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); f:`:/tmp/o_table3/; f set a;
o)// appending one record
o).[f;();,;(10;20;30)];
o)// appending two records
o).[f;();,;(10 10;20 20;30 30)];
```

Opening table in workspace works by binding global variable to table mapped on disk.

```
o)// create new table on disk
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); f:`:/tmp/odb1/; f set a;
o)// load it to workspace. Now odb1 global variable contains a writable reference
o)load f;
o)// append new record
o).[`odb1;();;;(10;20;30)];
o)// ... or via short syntax
o)odb1,:(10;20;30);
o)// see contents
o)odb1
a b c
-----
1 1 1
2 2 2
3 3 3
10 20 30
10 20 30
o)// close table, saving pending changes
o)odb1::0;
o)
```

 <<< prev Inserts

 Reading/writing concept

 Projecting files concept

 Relation Functions in core.o

Partitioned tables

Special way of storing tables splitting whole table by records. Criteria for splitting is duplicating values of single or several fields. In other words, it's table grouping by fields. This format naturally suits time series data.

Fields to group partition by are called partition fields. For instance, partitioning by dates/months/years/etc.

Partitioned table is a table consisting of a set of partitioned vectors. They also have keys/field names - symbol vector & values/field vectors.

Partitioned tables will be called ptables for brevity.

Ptable support

Almost all common verbs made for tables should operate on partitioned tables transparently.

Some exceptions exist though:

- All mutating verbs - amend, dmends, upserts, etc. Cast back to dict first & mutate as usual.
- Neither scalar nor composite indices are supported. That has same implications for verbs like `?` ("find"), queries, etc.
- `~` ("match") verb is not implemented for ptables.
- Queries has their own set of limitations for ptables. See corresponding section.

Ptable creation

Creating ptables consists of several stages:

- Creating non-partitioned table collecting all partitioning fields. Let's call it `gf`.
- Creating partition information table. This table keeps all persistent information on each partition. It's `info` in our example.
- Creating mount-related table. This table keeps current partition data. `mnt` field will keep it.
- Optional domain symbol in `sym` field.
- Collecting all above data under single dict (dict with meta-information) & casting it to ptable type.



Zero-sized partitioned tables are not supported. At least one partition should exist.

Our small example as follows:

```

o) gf:+`a`b!(1 2;10 20);
o) info:+`size`segId`refPath!(2 2; 0 0; (0N0;0N0));
o) p1: +`c`d!(100 100;200 200);
o) p2: +`c`d!(1000 1000;2000 2000);
o) mnt:+`mntval!(p1;p2);
o) mt: `gf`info`mnt!(gf;info;mnt);
o) pt: `ptable$mt;
o) pt
a b c d
-----
1 10 100 200
1 10 100 200
2 20 1000 2000
2 20 1000 2000

```

Reverse conversion back to dict is also supported via casting at any time.

```

o) gf:+`a`b!(1 2;10 20);
o) info:+`size`segId`refPath!(2 2; 0 0; (0N0;0N0));
o) p1: +`c`d!(100 100;200 200);
o) p2: +`c`d!(1000 1000;2000 2000);
o) mnt:+`mntval!(p1;p2);
o) mt: `gf`info`mnt!(gf;info;mnt);
o) pt: `ptable$mt;
o) d:`dict$pt;
o) d
gf | +`a`b!(1 2;10 20)
info | +`size`segId`refPath!(2 2;0 0;(0N0;0N0))
mnt | +,`mntval!((+`c`d!(100 100;200 200);+`c`d!(1000 1000;2000 2000)))

```

Now, let's make a detailed overview of required information in meta-dict.

Meta dict fields				Description / comments
<code>`gf</code>	<code>`info</code>	<code>`mnt</code>	<code>`sym</code>	Top level meta-dict fields
All partitioning fields (table)	All persistent info (table)	Mounting info (table)	Domain symbol	
<code>`gf1 `gf2` `gf3 ...</code>	<code>`size `segId `refPath</code>	<code>`mntval</code>		Meta-dict table fields
All partitioning fields table	<code>`size</code> - vector of longs keeping partition size	mounted partition data		
	<code>`segId</code> - segment id, vector of longs (reserved)			
	<code>`refPath</code> - list/vector of reference paths for each partition			
Possible combinations for partition info				
<code>val1 val2 val3</code>	<code>`size</code> - (long), <code>segId</code> - (long), <code>`refPath</code> - 0N0	Immediate partition table value		Partition table is in RAM. Saved on unmount.
<code>val1 val2 val3</code>	<code>`size</code> - (long), <code>segId</code> - (long), <code>`refPath</code> - 0N0	0N0		Special case for partition table. Only partitioning fields exist.
<code>val1 val2 val3</code>	<code>`size</code> - (long), <code>segId</code> - (long), <code>`refPath</code> - file symbol (without slash at the end)	Immediate partition table value		Partition is loaded entirely on mount. Saved on unmount in a single file.
<code>val1 val2 val3</code>	<code>`size</code> - (long), <code>segId</code> - (long), <code>`refPath</code> - file symbol (with slash at the end)	Projected table value		Partition is in splayed table on disk.
<code>val1 val2 val3</code>	<code>`size</code> - (long), <code>segId</code> - (long), <code>`refPath</code> - file or namespace symbol or namespace path list	Same value as in <code>`refPath</code>		Lazily mounted partition.



For `refPath` having value other than 0N0, it's recommended to keep non-zero corresponding `size` field. Otherwise, on turning meta-dict to ptable a possibly expensive operation of loading partition & determining its size occurs. In other words, even though lazy partition are supported, they require knowing their size in advance.

Ptable related functions

All ptable-related functions reside in the standard library `std/core.o`.

Make sure it's loaded before using them.

Example

```
t:+`created`id!(2022.10.10D10:00:01.0 2022.10.10D10:10:01.0 2022.10.10D10:10:05.0;1 2 3);
f:`:/tmp/pt/pt221010/; f set t;
t:+`created`id!(2022.10.11D10:00:01.0 2022.10.11D10:10:05.0;10 20);
f:`:/tmp/pt/pt221011/; f set t;
t:+`created`id!(2022.10.12D10:00:07.0 2022.10.12D10:10:07.0 2022.10.10D10:10:10.0;100+!3);
f:`:/tmp/pt/pt221012/; f set t;
t:+`created`id!(2022.10.13D10:00:01.0 2022.10.13D10:10:05.0;1000 2000);
f:`:/tmp/pt/pt221013/; f set t;

load "core";

ptdir:"/tmp/pt/";
dates: 2022.10.10 2022.10.11 2022.10.12;
names: { fmt["pt%%";(`year`mm`dd$x) mod 100i] }`dates;
paths: { `$format["%/" ;x] } `names;
size: {#get[0b; `$format[ptdir,"%/" ;x]]}`names; // can be optimized further to "get" specific field length...

gf: +`crDate!dates;
info: +`size`segId`refPath!(size;(# dates)#0; paths);
mnt: +`mntval!paths;
pd: `gf`info`mnt`root!(gf;info;mnt;`$ptdir);
pt: .o.pnew[0N0; pd];
.o.pset[`$ptdir; pt];
pt:.o.pget[`$ptdir];

pt:.o.pmnt[pt; 0N0]; // simulate mounting all partitions

idx: dates?2022.10.11;
pt:.o.pumnt[pt; idx]; // umount specific partition before changing

pd:`dict$mv pt;
0N!pd;

// append new record in idx partition
.[`pd; (`mnt;`mntval;idx); { ptab: `$ptdir,`v`char$1_`int$$x; .[ptab; 0]; ,; (2022.10.11D10:10:30.000000000; 30)]; x }};
// correct "size" value
.[`pd; (`info;`size;idx); +; 1];

pt:`ptable$pd;
show pt;
```

 `core.o`

Signals / Error handling

Signalling in O means throwing an error. The idea is similar to raising exceptions in other languages. That is signals are meant to break normal calculation flow and report an error.

Trapping a signal means running a function and expecting any error occurrence. Conceptually it is similar to "catch" clause in other languages.

Signals

Technically signalling is done using monadic verb `'`. Its argument is any value describing error.

```
o)a:10; { { '(`s;1); set[`a;0] }[] }[]
** signal error: `panic`:
s
1
o)a
10
o)
```

Trap

Trapping is done using `@` and `.` triads.

The `@` triad applies its first argument (a monadic function) to its second argument. When function fails or signals, the entire expression becomes equal to third argument.

```
o)@[x];1;`err2
1
o)@[x];`err1;`err2
`err2
o)
```

Another possible usage is trapping with a function as the third argument. When signal occurs, it leads to execution of the third argument function with signal value as an argument.

```
o)@[x];`err1;[x]
kind | `signal
call | "panic"
message| `err1
mark | `linum`offset`length!1 0 2
o)
```

Runtime errors are also caught using traps. Trapping runtime errors results in a signal value with the error text message as its value.

```
o)@[+ /x; "123"; {x`kind`call}]
`runtime
"+"
o)
```

The `[.]` triad is pretty much the same expression except its second argument can be a list/vector. It is used as several arguments for a trapped function. Thus, it allows using a function with arity greater than 1.

```
o).[+;(1;2);`err2]
3
o).[{'y};(1;`err1);{x}]
kind | `signal
call | "panic"
message| `err1
mark | `linum`offset`length!1 0 2
o)
```



If the last argument of the trap is not lambda, do not use print verbs. The last argument is calculated in advance and printing is always performed.

```
o).[+;(1;!);[0N!"error";0]]
"error"
0
o).[+;(1;2);[0N!"error";0]]
"error"
3
o).[+;(1;!);{0N!"error";0}]
"error"
0
o).[+;(1;2);{0N!"error";0}]
3
o)
```

[lambda](#)

[dictionaries](#)

Reagents

O language has a specific reagent type that deals with I/O.

Some reagents are built-in:

Arguments	Description
<code>reagent[`tty]</code>	Console reagent
<code>reagent[`listener;"addr:port"]</code>	TCP listener reagent
<code>reagent[`tcp;"addr:port"]</code>	TCP reagent (client side)
<code>reagent[`tcp;tcp_reagent]</code>	TCP reagent (server side)
<code>reagent[`ipc;"addr:port"]</code>	IPC reagent (client side)
<code>reagent[`ipc;tcp_reagent]</code>	IPC reagent (server side)
<code>reagent[`udp;"addr:port"]</code>	UDP reagent
<code>reagent[`ws;"addr:port"]</code>	Websocket reagent (client side)
<code>reagent[`ws;tcp_reagent]</code>	Websocket reagent (server side)
<code>reagent[`tls;tcp_reagent]</code>	Wrapper for tcp reagent that deals additional tls layer
<code>reagent[`file;`:/path/to/file]</code>	Reagent that backed up with a file
<code>reagent[`log;`:/path/to/file]</code>	Same as a file but operates by AST. Can be useful for creation journals
<code>reagent[`null]</code>	Empty reagent that produces nothing
<code>reagent[`async]</code>	MPSC reagent without blocking on sender side
<code>reagent[`sync]</code>	MPSC reagent with blocking on sender side
<code>reagent[`deq]</code>	Reagent that behaves as MPMC queue
<code>reagent[`bus]</code>	Much like an async reagent but allows to create reactions on it from different tasks
<code>reagent[`timer;timeout;repeat]</code>	TIMER reagent: timeout in ms, repeat: 0 means forever
<code>reagent[`state;other_reagent]</code>	Special reagent for tracking state of another reagent

Some reagents are shipped as [plugins](#):

Arguments	Description
<code>reagent[`kdb_listener]</code>	Reagent listener specific for kdb+ ipc protocol
<code>reagent[`kdb;"addr:port"]</code>	Reagent for kdb+ ipc protocol (client side)

Another important thing about reagents is that they can be used in declarative programming through declaring reactions:

```
o)r:reagent[`async];
o)react {[x:r] println["received: %";x]};
o)r[1];
received: 1
o)
```

As you can see, `react[..]` verb accepts lambda as a reaction body. The only difference from a regular lambda is arguments: they have "reagent-bound" definition.

Of course, reactions can (and usually do) have more than one argument. Such reactions trigger only when all the arguments are ready:

```
o)r1:reagent[`async]; r2:reagent[`async];
o)react {[x:r1;y:r2] println["X: % Y: %";x;y]};
o)r1[1];
o)// Nothing happens because there is no r2 yet.
o)r2[2];
X: 1 Y: 2
o)
```

Use the verb `meta[]` with reagents to see info about them:

```
o)meta r1
id | 4
state| `running
type | "async"
o)
```



Reactions on the same reagent are only allowed from the same task.

Reagents and imperative style

Some words of caution. All reagents can be polled for values using `get` verb. It can be called "imperative style" of work with reagents. It has some practical benefits as it makes code using reagents serial and that might be easier to comprehend.

However keep in mind, that mixing reactive and imperative/serial code could result in unexpected behaviour. Some specific reagents (like "state") are driven by reagents which they depend on and are useful only in reactive way.



Avoid mixing reactive and imperative/serial code, otherwise hard to debug issues are expected.

Some limitations

Reagents mostly behave as any other type in O (they are first class objects) but have some limitations:

- cannot be serialized;
- cannot be passed through the IPC.

Any other operations are allowed with reagents as well as with any other type in O:

```
o)r:reagent[`async];
o)meta r
id | 6
state| `running
type | "async"
o)react {[x:r] 0N!x};
o)// and it doesn't allow defining reaction on r from another task
o)spawn { react {[x:r] 0N!x} }
<Reagent#8>
  WARN base    > Task < react {[x:r] 0N!x} >
** I/O error: `react`:
-- {[x:r] 0N!x}
-- receiver has been already taken
--> REPL:1
|
1 | react {[x:r] 0N!x}
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
** stack backtrace:
[0]: "REPL":1
>
  {react {[x:r] 0N!x}..}
<
**
o)// only from the same task
o)react {[x:r] 0N!x+1};
o)
```

Specific reagent(s)

There is a specific type of reagent: taskhandle. It can not be created through calling reagent[] verb, it is a result of calling spawn[]. In all other aspects it behaves as well as any other reagent:

```
o)r: reagent[`async]
<Reagent#9>
o)h: spawn {100{x+1}/1}
<Reagent#11>
o)react {[x:r;y:h] println["X: % Y: %";x;y]};
o)r[6];
X: 6 Y: 101
o)
```

 [reagent](#)

 [react](#)

 [close](#)

 [meta](#)

Reagent async

An asynchronous multiple producer - single consumer queue. Most usable type of reagent.

Syntax: `reagent[\`async]`

Mostly used as a transport to allow tasks communicate with each other:

```
o)r: reagent[\`async];
o)spawn { react {[x:r] println["task receives: %";x]} };
o)r[123];
task receives: 123
o)
```

Another useful case is use async reagent as a sync primitive to wait some event:

```
o)barrier: reagent[\`async];
o)spawn { barrier["TASK 1"] };
o)println["task % has been spawned";get barrier];
task TASK 1 has been spawned
o)
```

[📖 All reagents](#)

[📖 reagent](#)

[📖 react](#)

[📖 close](#)

[📖 meta](#)

Reagent bus

Reagent for unite other reagents to a hub such that every value passed is being multiplexed for every participant.

Syntax: `reagent[`bus]`

```
o)// create empty bus
o)bus: reagent[`bus];
o)r1: reagent[`async];
o)spawn { react {[x:r1] println["R1: %";x]}};
o)r2: reagent[`async];
o)spawn { react {[x:r2] println["R2: %";x]}};
o)// add reagents to a bus
o)ctl[bus;(r1;r2)];
o)// send message to a bus
o)bus["Hello all!"];
R2: Hello all!
R1: Hello all!
o)// remove r1 from a bus
o)ctl[bus;(meta r1)`id];
o)// send message to a bus
o)bus["Hello all!"];
R2: Hello all!
o)
```

 [All reagents](#)

 [reagent](#)

 [react](#)

 [close](#)

 [meta](#)

Reagent deq

Reagent deq is an analog of MPMC asynchronous queue.

Syntax: `reagent[`deq]`

```
o)// create reagent deq
o)d1: reagent[`deq];
o)// spawn 3 stealers
o){spawn[{{id] d:reagent[`deq;d1]; react {[x:d] println["Task #%\nrecv: %";id;x]}};x] }'!3;
o)// send 10 items to a reagent
o){d1[x]}'!10;
Task #0
recv: 0
Task #2
recv: 2
Task #1
recv: 1
Task #0
recv: 3
Task #2
recv: 4
Task #1
recv: 5
Task #0
recv: 6
Task #2
recv: 7
Task #1
recv: 8
Task #0
recv: 9
o)
```

[All reagents](#)

[reagent](#)

[react](#)

[close](#)

[meta](#)

Reagent file

Reagent for writing/reading bytes from a file. If file exists bytes will append to file.

Syntax: `reagent[`file; <file name>]`

```
o)w:reagent[`file;`:/tmp/test.txt];
o)w "x"$"write\n"
o)
```

You can read file through the get to immediately after creating the reagent.

```
o)w:reagent[`file;`:/tmp/test.txt];
o)"c"$get w
"write\n"
o)w "x"$"append\n"
o)$get w
""
o)r:reagent[`file;`:/tmp/test.txt];
o)"c"$get r
"write\nappend\n"
o)
```

 [All reagents](#)

 [reagent](#)

 [get](#)

 [close](#)

 [meta](#)

Reagent ipc

Syntax: `reagent[`ipc;arg]`

Where arg is one of:

- string of format "host:port" to connect to
- reagent tcp to wrap to an ipc

Server example:

```
listener: reagent[`listener;"0.0.0.0:5100"];
spawn {
  react {[x:listener]
    cli: reagent[`ipc;x];
    react {[x:cli] println["\nclient request: %";x]; cli[(ts[];x)]}
  };
};
```

Client example:

```
o)cli: reagent[`ipc;"127.0.0.1:5100"];
o)react {[x:cli] println["server response in % ms : %";(ts[]-x[0])%1000000;x[1]]};
o)cli["Hello from ipc client"];
client request: Hello from ipc client
server response in 0 ms : Hello from ipc client
```



The search index is not serialized with structures. The host can build a search index on its own.

[All reagents](#)

[reagent](#)

[react](#)

[close](#)

[meta](#)

Reagent Kdb+

Reagent implements KDB+ ipc protocol. Can be created directly by call `reagent[`kdb; "host:port"]` or returns from reagent `kdb_listener` as a value.

```
o)load "kdb";
o)kdb_listener: reagent[`kdb_listener;"0.0.0.0:5100"];
o)spawn {react {[cli: kdb_listener] spawn{[cli] react {[x: cli] cli@[eval parse x];x;{`error$x}}];cli}};
```

```
o)load "kdb";
o)kdb: reagent[`kdb;"127.0.0.1:5100"];
o)react {[x:kdb] println["kdb server reply: %";x]};
o)// send request to a kdb server:
o)kdb["1+2"];
kdb server reply: 3
```

 [All reagents](#)

 [reagent](#)

 [react](#)

 [close](#)

 [meta](#)

Reagent Kdb+ listener

Allows ThePlatform behave like an KDB+ server.

Syntax: `reagent[`kdb_listener]`

```
load "kdb";
kdb_listener: reagent[`kdb_listener;"0.0.0.0:5100"];

react {[cli: kdb_listener]
  spawn[{{cli]
    // utility reagent to track stream state changes
    s: reagent[`state;cli]; react {[x:s] 0N!x};
    // receive kdb message end echos it back
    react {[x: cli] cli[@[eval parse x];x;{`error$x}]]}
  };cli];
};
```

[All reagents](#)

[reagent](#)

[react](#)

[close](#)

[meta](#)

Reagent listener

Binds to a network interface and accepts incoming tcp connections, producing reagents ``tcp` as a values.

Syntax: `reagent[`listener;"host:port"]`

```
listener: reagent[`listener;"0.0.0.0:5100"];

spawn {
  react {[x:listener]
    cli: reagent[`ipc;x];
    react {[x:cli] println["\nclient request: [%] -- %";ts[],x]; cli[x]}
  };
};
```

[All reagents](#)

[reagent](#)

[react](#)

[close](#)

[meta](#)

Reagent log

Reagent to be used to create journals.

Syntax: `reagent[`log]`

```
o)l:reagent[`log;`:/tmp/journal.log];
o){l[(`f;x;x+1)]}'!10;
o)f:{println["X: % Y: %";x;y]};
o)close l
o)l:reagent[`log;`:/tmp/journal.log];
o)react {[x:l] eval x};
X: 0 Y: 1
X: 1 Y: 2
X: 2 Y: 3
X: 3 Y: 4
X: 4 Y: 5
X: 5 Y: 6
X: 6 Y: 7
X: 7 Y: 8
X: 8 Y: 9
X: 9 Y: 10
o)
```

Of course, journal can be read at once by using verb ``get`

```
o)get `:/tmp/journal.log
0b
((`f;0;1);(`f;1;2);(`f;2;3);(`f;3;4);(`f;4;5);(`f;5;6);(`f;6;7);(`f;7;8);(`f;8;9);(`f;9;10))
```

`get` called on a journal file returns 2-element list which first element is bool indicates corrupted journal, second element is journal itself. If journal is corrupted, first element would be a 1b, second one - valid part of a journal;

[All reagents](#)

[reagent](#)

[react](#)

[close](#)

[meta](#)

Reagent null

Empty reagent never produce any value. There is no practic sence to create it explicitly.

[All reagents](#)

[reagent](#)

[react](#)

[close](#)

[meta](#)

Reagent state

Subscribes to any other reagent state changes and produce such events as items.

Syntax: `reagent[`state;other_reagent]`



Avoid mixing reactive and imperative/serial code in relation to "state" reagent. It is driven by reagents which it depends upon and is useful only in reactions.

```
o)r: reagent[`async];
o)// create reagent to receive r state notifications
o)s: reagent[`state;r];
o)react {[x:s] println["main State changed: \n%";x]};
o)close r;
main State changed:
id      | 12
message| Custom { kind: BrokenPipe, error: "closed" }
o)
```

 [All reagents](#)

 [reagent](#)

 [react](#)

 [close](#)

 [meta](#)

Reagent sync

Much like as `async reagent`, but waits ACKs on every send. Is used for ensure that message has been delivered.

Syntax: `reagent[`sync]`



Avoid using sync reagent in the same task because it causes an deadlock.

```
o)s: reagent[ `sync];
o)spawn {react {[x:s] println["value from sync reagent has been received: %";x]}};
o)s[123];
value from sync reagent has been received: 123
```

[All reagents](#)

[reagent](#)

[react](#)

[close](#)

[meta](#)

Reagent timer

Reagent to produce items repeatedly by an specified interval.

Syntax: `reagent[`timer;timeout;repeat]`

Where:

- `timeout`: an integer value (type long) in milliseconds
- `repeat`: an integer value (type long) indicates number of timer repetitions (0W, 0 means forever)

```
o)t: reagent[`timer;1000;3];
o)react {[x:t] println["timer tick: %";x]};
o)
timer tick: 1000
timer tick: 1001
timer tick: 1001
```

[All reagents](#)

[reagent](#)

[react](#)

[close](#)

[meta](#)

Reagent tcp

Justs simple tcp socket. No serialization, operates by raw bytes.

Syntax: `reagent[`tcp;arg]`

Where arg is one of:

- string of format "host:port" to connect to
- reagent tcp (does nothing since it is already an tcp)

Server example:

```
listener: reagent[`listener;"0.0.0.0:5100"];
spawn {
  react {[cli:listener]
    react {[x:cli] println["\nclient request: %";x]; cli[0x vs ts[]]}
  };
};
```

Client example:

```
o)cli: reagent[`tcp;"127.0.0.1:5100"];
o)react {[x:cli] println["server response in % ms";(ts[]-0p sv x)%1000000]};
o)cli[0x0102030405];
client request: 0x0102030405
server response in 0 ms
```

[All reagents](#)

[reagent](#)

[react](#)

[close](#)

[meta](#)

Reagent tty

Reagent for working with terminal I/O

Syntax: `reagent[`tty]`

Usually this reagent is being used for creating REPL.

For complete reference please see repl.o in a std library of ThePlatform.

[📖 All reagents](#)

[📖 reagent](#)

[📖 react](#)

[📖 close](#)

[📖 meta](#)

Reagent tls

Reagent used to wrap an raw tcp reagent to an encrypted via tls.

Syntax: `reagent[`tls;tcp_reagent]`

[All reagents](#)

[reagent](#)

[react](#)

[close](#)

[meta](#)

Reagent udp

Reagent for making udp sockets.

Syntax: `reagent[`udp;"host:port";["multicast_addr"]]`

```
o)srv: reagent[`udp;"0.0.0.0:31337"];
o)react {[x:srv] println["request from udp client: %";"c"$x[0]]; srv[x]; close[srv]};
o)cli: reagent[`udp;"0.0.0.0:45667"];
o)react {[x:cli] println["reply from udp server: %";"c"$x[0]]; close[cli]};
o)cli[("hello";"127.0.0.1:31337")];
request from udp client: hello
reply from udp server: hello
o)
```

Joining multicast group:

```
o)// join udp socket to a multicast group 224.1.1.1
o)srv: reagent[`udp;"0.0.0.0:5007";"224.1.1.1"];
o)react {[x:srv] println["-- data from udp multicast: %";"c"$x[0]]};
o)cli: reagent[`udp;"0.0.0.0:12345"];
o)// send data to a multicast address
o)cli[("hello";"224.1.1.1:5007")];
-- data from udp multicast: hello
o)
```

 [All reagents](#)

 [reagent](#)

 [react](#)

 [close](#)

 [meta](#)

Reagent ws

Reagent for making websockets.

Syntax: `reagent[`ws;arg]`

Where arg is one of:

- uri: string representing url to connect to
- tcp/tls reagent to be wrapped to a websocket

```
ws: reagent[`listener;"0.0.0.0:45101"];
react[{{[x:ws]
  spawn[{{[sock]
    h: reagent[`ws;sock];
    react[{{[x:h] h[x]}]
  }};x]
}}];
```

[All reagents](#)

[reagent](#)

[react](#)

[close](#)

[meta](#)

Attributes

Attribute is an optional piece of information describing vector property.

E.g. sorted attribute can speed up search, join and other verbs.



Once an attribute is attached to vector, it cannot be removed by amend/dmend.

Attaching an attribute is done using # dyad with left argument being a predefined symbol.

Type	Attr symbol
Ascending sort	`s or `asc
Descending sort	`desc
Search index	`g

Sorted attribute

When attached to vector, sorted attribute ensures vector is sorted. Both ascending and descending sorts are supported. When sorted attribute is present, binary search algorithm is used instead of linear scan.

```
o)a: `s#1 2 3
`asc#1 2 3
o)a: `asc#1 2 3
`asc#1 2 3
o)a: `desc#3 2 1
`desc#3 2 1
o)
```

One-element vector and empty vectors:

```
o)a: `s#`long$()
`asc#`long$()
o)a: `s#,1
`asc#,1
o)a: `desc#`long$()
`desc#`long$()
```

Attaching sorted attribute to an unsorted vector will result in an error:

```
o) a: `s#3 2 1
** runtime error: `attribute`:
not sorted
o)
```

Search index attribute `g`

When attached to vector, search index attribute creates a separate search index structure which uses additional memory. With search index attribute, search algorithm is used instead of linear scan.

There are no specific requirements for vector contents.

```
o)a: `g#til 10
`g#0 1 2 3 4 5 6 7 8 9
o)
```

Existing value modification

Destructively attaching an attribute to an existing vector:

```
o)a: 1 2 3
1 2 3
o).[`a;();{`s#x}]
`a
o)a
`asc#1 2 3
o)
```

Destructively removing an attribute from an existing vector:

```
o)a: `s#1 2 3
`asc#1 2 3
o).[`a;();{`#x}]
`a
o) a
1 2 3
o)
```

Before modifying a vector, interpreter checks it for an attribute at runtime:

```
o)a: `s#1 2 3;
o)@[a;0;;3]
** runtime error: `attributes`:
attr violation
o)
```

Multi-column attributes/indices

Attributes defined on several table fields are useful for search and query joins. Thus, attributes are defined not on fields/vectors, but on tables themselves. In line with simple attributes, all mutable verbs preserve attributes state consistency. It means that trying to mutate a table in an incompatible way with the index results in a runtime error. E.g., only append and update are supported.

Currently, a single attribute type supported is `g` based on tree index.

There are two different ways to define multi-column indices/attributes. To create an immutable attribute, use the # dyad:

```
o)a:(+:)`a`b`c!(!5;!5;`asc#!5); b:`g#(2!a);
o)b
a b c
-----
0 0 0
1 1 1
2 2 2
3 3 3
4 4 4
o)meta b
+`column`type`id`attr!(`a`b`c;`long`long`long;45376 45376 176448;```asc)
(`a`b)
o)
```



"meta" verb can be used to see which table indices are present.

Mutable attribute build is done via @ [tetrad](#) with an enclosed symbol vector in the second argument:

```
o)a:(+:)`a`b`c!(!5;!5;!5); @[`a;`,`a`b`c;~[#];`g]; meta a
+`column`type`id`attr!(`a`b`c;`long`long`long;45376 45376 45376;```)
(`a`b`c)
```

In either way, "find" verb will use an attribute/index with appropriate fields automatically.

```
o)a:(+:)`a`b`c!(!10;!10;!10);
o)@[`a;`,`a`b`c;~[#];`g];
o)(!10)~a?(+:)`a`c`b!(!10;!10;!10)
1b
o)
```

To drop an index, use a null symbol in amend:

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); @[`a;`,`a`b;~[#];`g];
o)meta a
+`column`type`id`attr!(`a`b`c;`long`long`long;45376 45376 45376;```)
(`a`b)
o)@[`a;`,`a`b;~[#];`];
o)meta a
+`column`type`id`attr!(`a`b`c;`long`long`long;45376 45376 45376;```)
()
```

Multi-column index on disk

Multi-column indices on disk are fully supported. See [index on disk with enums](#):

```
o)sym:`symbol$!5;
o)fsym:`:/tmp/midx/o_sym_midx.dat; fsym set sym;
o)a:(+:)`a`b`c!(`sym$`symbol$!5;10+!5;20+!5);
o)@[`a;`,`a`b`c;~[#];`g];
o)f:`:/tmp/midx/; f set a;
o)b:get f; sym: get fsym; // read all data from disk
o)@[`b;`a;~[$];`sym]; // attach symbol for enums
o) a~b
1b
o)
```

Example with index read on demand off the disk:

```
o)sym:`symbol$!15;
o)fsym:`:/tmp/midx/o_sym_midx.dat; fsym set sym;
o)a:(+:)`a`b`c!(`sym$`symbol$!5;`sym$`symbol$10+!5;20+!5);
o)@[`a;`,`a`b`c;~[#];`g];
o)f:`:/tmp/midx/; f set a; // save an entire table with index on disk
o)load f; // read table with index into workspace
o)@[`midx;`a`b;~[$];`sym];
o)midx?(+:)`a`b`c!(`sym$`symbol$2 3;`sym$`symbol$12 13;22 23)
2 3
o)
```

 # (count)

 # (take)

 table meta

Patterns

O has a [pattern matching](#) feature. Its principle is similar to the "switch" construction in C language, but pattern matching in O is more powerful since it allows matching against all types, including lists, vectors and their parts (e.g. destructuring). Usually, pattern matching is used to construct various types according to input constraints.

```
o)m:{match [x;y] { (1 2 3;4 5 6) -> "888"; (`a`s`d!1 2 3;_) -> `a`s`d; (12;_) -> 10#3; _ -> 777 } };
o)m[1 2 3;0]
777
o)m[(1 2 3;4 5 6);0]
777
o)m[1 2 3;4 5 6]
"888"
o)m[12;3]
3 3 3 3 3 3 3 3 3 3
o)
```

In scripts you can write `match` more readable:

```
m:{match [x;y] {
  (1 2 3;4 5 6)    -> "888";
  (`a`s`d!1 2 3;_) -> `a`s`d;
  (12;_)          -> 10#3;
  -               -> 777
} };
show m[1 2 3;4 5 6];

"888"
```



Be careful if you pass a list to a match: it may turn into a vector after evaluation. Use the `pick` to avoid problems.

```
o)a:1;
o)b:`c;
o)m:{match [x] { (1; _) -> "start with one"; _ -> type x }};
o)m (a;b)
"start with one"
o)b:2; m (a;b)
`v`long
o)m:{match [x] { pick[1; _] -> "start with one"; _ -> type x }};
o)m pick[a;b]
"start with one"
o)
```

 [pick](#)

{ } Lambda

Defines user function

Syntax: `{<...>}; {[x {; ..}] <...> }`

Function declaration consists of two phases - defining lambda expression and binding it to some name. Lambdas include one or several expressions separated by semicolons.

Lambda arguments

Lambda arguments are defined using `[]`:

```
o).n.sum: {[arr] a:+/arr; a}
{[arr] a:+/arr; a}
o)ff: {[a;b] */a+!1+b-a}
{[a;b] */a+!1+b-a}
o)
```



Currently, maximum of 8 named arguments are allowed.

Lambda arguments can be defined implicitly via using pre-defined names `x`, `y` and `z`.

```
o)f: {x+y+z}; f[1;2;3]
6
o)f: {x*y}; f[10;5]
2
o)
```

Lambda result

The last expression in the lambda is the result of that lambda. To return the result early, you can use the verb `return`.

If the lambda is terminated by `;`, the result will be a generic null.



To avoid problems, do not use comments after the last lambda expression, or use the combination:

```
return <expr>; // <comment>
}
```

Locals

Local variables are defined implicitly via assignment to bindings inside lambdas that are not defined yet.



Currently, maximum of 22 named local variables are allowed.

```
o)f:{local1:x; local2:y; local1+local2}
{local1:x; local2:y; local1+local2}
o)f[10;20]
30
o)
```

```
o).n.sum:{a:+/x; a}
{a:+/x; a}
o).n.sum [!10]
45
o)
```



Be careful with the implicit use of arguments and local variables `y` and `z`. In this case the lambda expects more than 1 argument.

```
o)fxz: {z:2; x*z};
o)fxz[3] //projection
{{z:2; x*z}[3;;]}
o)fxz: {[x] z:2; x*z};
o)fxz[3]
6
o)
```



When using nested lambda, the scope of the local variable is limited to the lambda of the first nesting. If you need to extend the scope to all nested lambdas, use the `shadow`.

```
o)a:42;
o){a:2; show a; {show a; {show a}[ ] }[ ] }[ ];
2
2
42
```

Using `::` you can create/change variables in non-local lambdas scope.

```
o)(a;b):42 24;
o){a:2; b::3; show a,b; {show a,b; {show a,b}[ ] }[ ] }[ ];
2 3
2 3
42 3
o)
```



`::` must also be used if you want to change the value of a local variable using this local variable.

```
o)(a;b):42 24;
o){a:2; b::3; show a,b; {a::a*a; show a,b; {show a,b}[ ] }[ ] }[ ];
2 3
4 3
42 3
o)
```



The presence of a local variable in the lambda overshadows the variable with the same name.

```
o)a:42;
o){ show a; }[]
42
o){ show a; a:2; show a }[]
0N0
2
o){ show a; a:2; show a; {show a; a:3; show a;}[]; show a; }[]
0N0
2
0N0
3
2
o)a
42
o)
```

Projection

When lambda or verb expects 2 or more arguments but gets less - the result is a lambda projection on the arguments provided.

```

o)add2: +[2;]
{+[2;]}
o)add2 10
12
o)f: {[a;b;c] a*a+b-c};
o)g: f[3;;4]
{[a;b;c] a*a+b-c}[3;;4]}
o)g 5
10
o)

```

Sometimes the projections help to indicate the specific arity of the verb.

```

o)r: reagent[`async];
o)r [10]
o)// get r - monad
o)get r
10
o)//to get from channel without lock use get[100;r] - dyad
o)//to catch error use trap
o)//the next trap uses the default monadic get, and we have an "invalid type" error
o).[get;(100;r);{x`message}]
"invalid type: [s`long]"
o)//the next trap uses the projection of dyadic get, and we catch an correct "timeout" error
o).[get[;];(100;r);{x`message}]
"timeout elapsed"
o)

```

Function/lambda application

Insert arguments in [] after the function name or omit brackets if there is only one argument.

```

o).n.sum:{a:+/x; a}
{a:+/x; a}
o)arr:!3
0 1 2
o).n.sum arr
3
o).n.sum[arr]
3

```

You can also apply arguments to lambdas without binding the latter to a name:

```

o){x*2}1
2

```

Recursive lambdas

binding is special in lambdas body. It defines reference to enclosing lambda itself. Thus, it allows creating recursive lambdas:

```
o)fibonacci: {[x] $[x<2;x;o[x-1]+o[x-2]]}; fibo[6]
8
```

... fibonacci with memoization:

```
o)d:0 1!0 1; fib: {$[d[x]=0N;d[x]:o[x-2]+o[x-1];()]; d[x]};
o)fib[6]
8
o)
```

However, pay attention to clashes with locals/arguments:

```
o){[o] o:1; o}[1]
1
o){[o] .[ `o;() ;+;1]; o}[1]
2
o){[o] o+:1; o}[1]
2
```

Closures

Closures are another kind of functions - they capture parent local variables. They can be used everywhere instead of simple functions:

```
o)parent: { upval: x; {upval + x} }; closure: parent[2]; closure[3]
5
```

The fibonacci example given above can be rewritten to avoid creating global state like:

```
o)fibonacci: { d:0 1!0 1; { $[d[x]=0N;d[x]:o[x-2]+o[x-1];()]; d[x] }[] };
o)fibonacci[6]
8
o)
```

[return](#)

[shadow](#)

shadow

Creates variables that are visible down the stack.

Syntax: `shadow `x; x::y`; `.x:y`

Allowed only inside lambda's body, otherwise it doesn't make sense.

```
o)f:{s::20;};
o)f2:{shadow `s; s::10;f[];s};
o)f2[]
20
o)
```

```
o)a:42;
o){shadow `a; a::2; show a;{show a; {show a}[] }[] }[]
2
2
2
```



Always use the verb `shadow` in pair with `::` otherwise you will use a local variable.

```
o)a:42;
o){shadow `a; a:2; show a;{show a; {show a}[] }[] }[]
2
2
0N0
```

For shadow variable it is more convenient to use names that start with a dot.

```
o){.a:2; show .a;{show .a; {show .a}[] }[] }[]
2
2
2
```



A shadow variable in a lambda overshadows a variable with the same name only after its initialization (unlike a local variable).

```
o).a:42;
o){ show .a; }[]
42
o){show .a; .a:2; show .a }[]
42
2
o).a
42
o)
```

 [lambdas](#)

0: FileText (assign 0)

Read and write a text file.

Syntax: `x 0: y`; `0: [x;y]`

Write list of strings to a file: for left argument use a symbol path to the file.

Read list of strings: left argument is a separator represented with a [regex](#) string, right argument is a symbol path to the file to be read.

```
o)\:f.txt 0: ("1,2,3";"4,5,6")
\:f.txt
o)"[\,\\n]" 0: \:f.txt
"1"
"2"
"3"
"4"
"5"
"6"
""
o)read[\:f.txt]
0x312c322c330a342c352c360a
```

CSV

With 0: we can work with csv files. First of all you must prepare the table columns as formatted text.

In this use, 0: has as left operand a char delimiter and as right operand a table.

```
o)t:+`a`b`c!(1 2;3 4;5 6);
o)f:"," 0: t
"a,b,c"
"1,3,5"
"2,4,6"
o)\:t.csv 0: f;
o)
```

To read csv file need define types of columns and char delimiter.

```
o)tbl:("JJJ"; enlist ",") 0: \:t.csv
a b c
-----
1 3 5
2 4 6
o)
```

Delimiter char must be enlisted when the first record of the file contain column names.

```
o)`:tt.csv 0: ("1,2,3";"4,5,6");
o)+`aa`bb`cc!("JJJ";",") 0: `:tt.csv
aa bb cc
-----
1 2 3
4 5 6
o)
```

Example with module "core" for file with column names:

```
o)load "core";
o)t:flip `aa`bb`cc!(1 2 3;`a`b`c;("r0";"r1";"r2"));
o)wcsv[t;!t;",";`:t.csv];
o)rcsv[();",";"JSc";`:t.csv]
aa bb cc
-----
1 a "r0"
2 b "r1"
3 c "r2"
o)
```

Example with module "core" for file without column names:

```
o)load "core";
o)t:flip `aa`bb`cc!(1 2 3;`a`b`c;("r0";"r1";"r2"));
o)cols:!t;
o)wcsv[t;();",";`:tt.csv];
o)rcsv[cols;",";"JSc";`:tt.csv]
aa bb cc
-----
1 a "r0"
2 b "r1"
3 c "r2"
o)
```

 [Assign 1](#)

 [read](#)

 [repr](#)

1: FileBinary (assign 1)

Reads or writes bytes

Syntax: `x 1: y`; `1:[x;y]`

Where:

- x is a two-element list (a string or a nested string representing type and a long vector or a scalar);
- y is a byte vector.

```
o)(8;"j")1:0x1000000000000000 // big endian
1152921504606846976
o)("j";8)1:0x1000000000000000 // little endian
16
o)("ij";4 8)1:0x000100001000000000000000 // little endian
256i
16
o)
```

Supports reading records as list of vectors since 0.7.0. It requires nested string as type format. Only lower case type formats are supported. Returns list of vectors suitable for quick table conversion using `flip`. See below:

```
o)(,"xs";1 2)1: 0x017072037072
0x0103
`pr`pr
o)xx:(,"xg";1 16)1: 0x0102030405060708091011121314151617
0x01
,02030405-0607-0809-1011-121314151617
o)+`a`b!xx
a b
-----
1 02030405-0607-0809-1011-121314151617
```

See `repr` format string for reference on types.

 [Assign 0](#)

 [repr](#)

Destructuring

Extracts values from complex structures such as dicts and tables:

```
o){(a;_;c):x;c}[(1;(1 2; 3);4)]
4
o)(a!b):( `a`s`d`f!(1 2 3 4));b
1 2 3 4
o)(a!b):+( `a`s`d`f!(1 2 3 4));a
 `a`s`d`f
o)(a!(b;c;d;e)):( `a`s`d`f!(1 2 3 4));e
4
o)((a;b;u)@`s`f`d):( `a`s`d`f!(1; 2; 3;( `u`l!(11 444)))));u
3
o)((a;b).`f`u):( `a`s`d`f!(1; 2; 3;( `u`l!(11 444;0 2 3 4)))));b
444
o)((a;b);c):( +`a`b!(1 2;3 4);3);
o)a
a| 1
b| 3
o)b
a| 2
b| 4
o)c
3
o)
```

Triadic @ (amend)

Applies a monadic verb to a certain value.

Syntax: @ [<x>; <y>; <z>]

where **z** is a monadic verb to be applied and **x** is a structure to be indexed with **y**.

```
o)a:1 2 3; @[a;2;{x+1}]
1 2 4
o)d:`a`b`c!(1 2;3 4;5 6);
o)@[d;`a;{x+10}]
a| 11 12
b| 3 4
c| 5 6
o)t:flip `a`b!(!3;3+!3)
a b
---
0 3
1 4
2 5
o)@[t;`b;{x*3}]
a b
----
0 9
1 12
2 15
o)
```

For inplace modify use variable symbol in the first argument:

```
o)a:1 2 3; @[`a;2;{x+1}]
`a
o)a
1 2 4
o)
```




In triadic amend the last argument should be monadic, but some verbs are treated as dyadic. Use **:** after the verb to indicate the use of monadic.

```
o)a: !10
0 1 2 3 4 5 6 7 8 9
o)@[a;2 4 6; -]
** eval error: `amend vec` :
invalid type: [``dyad]
o)@[a;2 4 6; -:]
0 1 -2 3 -4 5 -6 7 8 9
o)
```

 @ (internal type id)

 @ (tetradic amend)

 . (triadic dmend)

 . (tetradic dmend)

Triadic . (dmend)

Applies a monadic verb to a certain value.

Syntax: `. [<x>; <y>; <z>]`

where `z` is a monadic verb to be applied and `x` is a structure to be indexed in depth with a vector index `y`.

```
o)l:(1 2 3;4 5 6); .[l;0 1;{x+1}]
1 3 3
4 5 6
o)d:`a`b`c!(1 2;3 4;5 6);
o).[d;(`a;0);{x+1}]
a| 2 2
b| 3 4
c| 5 6
o)t:flip `a`b!(!3;3+!3)
a b
---
0 3
1 4
2 5
o).[t;(`b;2);{x*3}]
a b
----
0 3
1 4
2 15
o)
```

For destructive updates use variable symbol in the first argument:

```
o)a:(1 2 3;4 5 6); .[`a;0 1;{x+1}]
`a
o)a
1 3 3
4 5 6
o)
```



In triadic dmend the last argument should be monadic, but some verbs are treated as dyadic.. Use `:` after the verb to indicate the use of monadic.

```
o)a: (!5; !4)
0 1 2 3 4
0 1 2 3
o).[a;1 3; -]
** eval error: `amend vec`:
invalid type: [``dyad]
o).[a;1 3; -:]
0 1 2 3 4
0 1 2 -3
o)
```

 [. \(tetradic dmend\)](#)

 [@ \(triadic amend\)](#)

 [@ \(tetradic amend\)](#)

 [. \(apply\)](#)

 [. \(value\)](#)

Tetradic @ (amend)

Applies a dyadic verb to a certain value.

Syntax: @ [<x>; <y>; <z>; <w>]

where **z** is a dyadic verb to be applied, **x** is a structure to be indexed with **y** and **w** is the second argument of the verb.

```
o)a:1 2 3; @[a;2;+;1]
1 2 4
o)d:`a`b`c!(1 2;3 4;5 6);
o)@[d;`b;*;2]
a| 1 2
b| 6 8
c| 5 6
o)t:flip `a`b!(!3;3+!3)
a b
---
0 3
1 4
2 5
o)@[t;`a;;0]
a b
---
0 3
0 4
0 5
o)
```

For destructive updates use variable symbol:

```
o)a:1 2 3; @[`a;2;+;1]
`a
o)a
1 2 4
o)
```

-  @ (indexing)
-  @ (internal type id)
-  @ (triadic amend)
-  . (triadic dmend)
-  . (tetradic dmend)

Tetradic . (dmend)

Applies a dyadic verb to a certain value.






Syntax: `. [<x>; <y>; <z>; <w>]`

where `z` is a dyadic verb to be applied, `x` is a structure to be indexed in depth with a vector index `y` and `w` is the second argument of the verb.

```
o)a:(1 2 3;4 5 6);
o).[a;0 1;+;1]
1 3 3
4 5 6
o)d:`a`b`c!(1 2;3 4;5 6);
o).[d;(`c;1);;0]
a| 1 2
b| 3 4
c| 5 0
o)t:flip `a`b!(13;3+!3)
a b
---
0 3
1 4
2 5
o).[t;(`b;0);+;10]
a b
----
0 13
1 4
2 5
o)
```

For destructive updates use variable symbol:

```
o)a:(1 2 3;4 5 6); .[^a;0 1;+;1]
`a
o)a
1 3 3
4 5 6
o)
```

-  [. \(triadic dmend\)](#)
-  [@ \(triadic amend\)](#)
-  [@ \(tetradic amend\)](#)
-  [. \(apply\)](#)
-  [. \(value\)](#)

Dyadic bitwise AND (band)

Syntax: `<x> band <y>`; `band[<x>; <y>]`

```
o)10 band 12
8
o)
```

Explanation:

```
  1010
AND 1100
====
  1000
```

Monadic bitwise NOT (bnot)

Syntax: `bnot <x>`; `bnot [<x>]`

```
o)bnot 2h
-3h
o)
```

Explanation:

```
NOT 0000000000000010
=====
1111111111111101
```

Dyadic bitwise OR (bor)

Syntax: `<x> bor <y>`; `bor[<x>; <y>]`

```
o)10 bor 12
14
o)
```

Explanation:

```
  1010
OR 1100
====
  1110
```

Dyadic bitwise XOR (bxor)

Syntax: `<x> bxor <y>`; `bxor [<x>; <y>]`

```
o)10 bxor 12
6
o)
```

Explanation:

```
  1010
XOR 1100
====
  0110
```

Dyadic \$ (cast)

Converts right arg values according to left arg specification.

Syntax: `<x> $ <y>; $[<x>; <y>]`

Where `x` is a lower-case letter or symbol from the table below. Returns `y` cast according to `x`.

Type letter	Type symbol
"b"	`bool
"x"	`byte
"h"	`short
"i"	`int
"j"	`long
"s"	`symbol
"c"	`char
"g"	`guid
"e"	`real
"f"	`float
"p"	`timestamp
"n"	`timespan
"z"	`datetime
"d"	`date
"m"	`month
"t"	`time
"u"	`minute
"v"	`second

```
o)"i"$12
12i
o)`bool$1
1b
o)"j"$1.1
1
o)
o)"f"$10
10f
o)
o)"d"$2020.12.20D11:39:42.550501414
2020.12.20
```

Casting to a string has a shorthand form: `$x`

```
o)$123
"123"
o)$2020.12.11D11:39:42.550501414
"2020.12.11D11:39:42.550501414"
o)$!10
"0"
"1"
"2"
"3"
"4"
"5"
"6"
"7"
"8"
"9"
o)
```



You can use short form `$` cast to string only for scalars and a list of scalars. Monadic `repr` represents a value of any type.

[Types, Casting, etc](#)

[\\$ \(representation\)](#)

Monadic repr

Represent a value to string.

Syntax: `repr <x>`; `repr [<x>]`

```
o)d: `a`b!(1 2;("1"; "2"));
o)repr d
"a| 1 2\nb| (\`1\`; \`2\`)"
o)
```

Dyadic representation \$

Interprets a string as a data value.

Syntax: `<x> $ <y>`; `$(<x>; <y>]`

Where:

- y is a string;
- x is a upper-case char as below.

Type name	Repr string
bool	B
guid	G
byte	X
short	H
int	I
long	J
real	E
float	F
symbol	S
timestamp	P
month	M
date	D
datetime	Z
timespan	N
minute	U
second	V
time	T

```
o)"F"$"123"  
123f  
o)"G"$"61e35154-10bc-a49a-d11f-f10e1a377000"  
61e35154-10bc-a49a-d11f-f10e1a377000  
o)
```

📖 Types, Casting, etc

📖 \$ (cast)

Monadic + (flip)

Returns `x` transposed, where `x` may be a list of lists, a dictionary or a table.

Syntax: `flip <x>`; `flip[<x>]`; `+<x>`; `+ [<x>]`

```
o)flip (1 2 3;4 5 6;7 8 9)
1 4 7
2 5 8
3 6 9
o)+`a`s`d!(1 2 3;4 5 6;7 8 9)
a s d
-----
1 4 7
2 5 8
3 6 9
o)flip flip `a`s`d!(1 2 3;4 5 6;7 8 9)
a| 1 2 3
s| 4 5 6
d| 7 8 9
o)
```



The flip verb automatically extends non-enlisted scalars.

```
o)+`a`b!(1;(1;2))
a b
---
1 1
1 2
o)+`a`b!(1;,1 2)
a b
-----
1 1 2
o)+`a`b!(`c;(1 2))
a b
---
c 1
c 2
o)+`a`b!(`c;(,1 2))
a b
-----
c 1 2
o)
```



```
o)0x sv 0x090c2ee8ad1a7942
651947622933363010
o)0p sv 0x090c2ee8ad1a7942
2020.08.28D16:33:42.933363010
```

```
o)0x sv 0x000000000000000a
10
o)0x sv 0x4024400000000000
4621889486333149184
o)0f sv 0x4024400000000000
10.125
```

 [vs \(vector from scalar\)](#)

Dyadic vs (vector from scalar)

Separate a scalar into a vector of elements.

Syntax: `<x> vs <y>`; `vs [<x>; <y>]`

where `x` is a separator, `y` is a vector to be separated.

Examples with string:

```
o) ", " vs "a,b,c,d"
"a"
"b"
"c"
"d"
o) "" vs "a,b,c,d"
""
"a"
", "
"b"
", "
"c"
", "
"d"
""
```



If `y` is a string, then string `x` used as `regular expression`

Let's see few examples about it:

```
o) ". " vs "2022.02.22"
""
""
""
""
""
""
""
""
""
""
o) "\\." vs "2022.02.22"
"2022"
"02"
"22"
o) {"I"$x} each "[.]" vs "2022.02.22"
2022 2 22i
o) `year`mm`dd`D"$"2022.02.22"
2022 2 22i
o)
```


Monadic .o.compress

The platform has the ability to compress a byte vector using the LZ4 algorithm. LZ4 is a lossless data compression algorithm that is focused on compression and decompression speed.

Syntax: `.o.compress <vec>`; `.o.compress [<vec>]`

Monadic .o.decompress

Decompresses a byte vector.

Syntax: `.o.decompress <vec>`; `.o.decompress [<vec>]`

```
o)l:!1000;
o)v: >>1;
o)count v
8008
o)c:.o.compress v;
o)count c
4011
o)20# << .o.decompress[c]
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
o)
```

 [serialization](#)

Triadic \$ (condition)

Short-circuit conditional expression. Thus this verb is special. It does not evaluate all of its arguments immediately. Only the first argument is evaluated. If it is true, the second argument is evaluated and becomes the result of the `cond` expression. Otherwise, the third argument is evaluated.

Syntax short form: `$_[<cond>; <true res>; 0N0]` or `if [<cond>] {<true res>}`

Syntax full form: `$_[<cond>; <true res>; <else res>]` or `if [<cond>] {<true res>} else {<else res>}`

Syntax extended form without syntactic sugar: `$_[<cond1>; <true res1> (; <cond2>; <true res2> (; ..))]; <else res>]`

Syntax extended form with syntactic sugar: `if [<cond1>] {<true res1>} (elif [<cond2>] {<true res2>} (..)) else {<else res>}`

The concept of truth here is a bit complex:

- For boolean scalars, truth means 1b.
- For integer scalars, truth means any value except 0 and null.
- For other scalar types, truth means any non-null value.
- **For vectors, dicts, tables, truth means non-empty structure!!!**
- **Everything else (monads, dyads, lambdas, ...) results in truth!!!**

```
o)$_[2<3;"yes";"no"]
"yes"
o)
o)$_[0n;"yes";"no"]
"no"
o)$_[();1;2]
2
o)$_[1=2;0;3=3;1;2]
1
o)$_[1=2;0;3=4;1;2]
2
o)a:3; b:3; if [a<b] {"<"} elif [a>b] {">"} else {"="}
"="
o)t:([]a:1 2;b:1.1 2.2)
a b
-----
1 1.1
2 2.2
o)$_[t;1;0]
1
o)d:`a`b!(1 2 3;1.1 2.2 3.3)
a| 1 2 3
b| 1.1 2.2 3.3
o)$_[d;1;0]
1
o)
```

Another thing to remember is that simulation of short circuit evaluation of condition itself is done using nested conds:

```
o) a:1 2 3;
o) $[a;${1=a[0]};2;3];4]
2
o)t:([]a:1 2;b:1.1 2.2)
a b
-----
1 1.1
2 2.2
o)$[3=count t;${d[`b;1]=2.2;1;2};${d[`a;0]=0;3;4}]
3
o)
```

 [vector conditional](#)

 [filter](#)

Triadic ? (vector conditional)

Syntax: `?[<cond>; <vector for true>; <vector for false>]`

This verb returns value made of second or third argument depending on first boolean scalar or vector argument. Think about vector if-then-else expression. Boolean truth results in second argument usage, false - in third one.

First scalar boolean is the same condition expression.

```
o)?[1b;1;2]
1
o)?[0b;0 1 2;2 2 3]
2 2 3
o)
```

First vector boolean is much more useful. Second and third argument should have compatible types. All arguments must have the same shape.

```
o)?[011b;4 2 3;1 1 1]
1 2 3
o)?[001b;1;(2;"123";3)]
2
"123"
1
o)
```

 [conditional](#)

 [? \(distinct\)](#)

 [? \(search\)](#)

Polyadic & (and condition)

Polyadic short-circuit "and" condition. Triadic form behaves exactly as `$[<cond>; <>true res>; <>false res>]` verb.

Syntax: `&[<cond1> (; <cond2> (; ..)) ; <>true res>; <>false res>]`

```
o)&[1b; 1b; 1; 2]
1
o)&[1b; 0b; 1b; 1b; 1; 2]
2
o)x:1 2 3
1 2 3
o)i:"s"
"s"
o)&[(!`v`long)=@x; (!`s`long)=@i; x+i; "error"]
"error"
```

📖 \$

Polyadic | (or condition)

Polyadic short-circuit "or" condition. Triadic form behaves exactly as `$(<cond>; <>true res>; <>false res>)` verb.

Syntax: `| [<cond1> (; <cond2> (; ..)) ; <>true res>; <>false res>]`

```
o)|[1b; 1b; 1; 2]
1
o)|[1b; 0b; 0b; 0b; 1; 2]
1
o)f:{|[(!`v`long)=@x; (!``table)=@x; x y; "error"]}
{|[(!`v`long)=@x; (!``table)=@x; x y; "error"]}
o)f[1 2 3; 1]
2
o)f[+`a!1 2 3; 1]
a| 2
o)f["123"; 1]
"error"
```

 \$

Reading/writing concept

Database persistence is an important O feature. Conceptually, there are two kinds of persistence in O language - reading/writing entire files from/to disk and projecting vectors/tables directly from disk.

The first kind is easier and more powerful as it supports more O types. The second kind is often faster and more memory-efficient, but supports only a subset of O structures - vectors of simple/fixed types, dictionaries and tables.

Reading and writing are done using `get` and `set` verbs.

Syntax: `<x> set <y>`; `set[<x>; <y>]`

where `x` is a symbolic file handle (a symbol starting with ":", followed by directory and ending with filename + extension) and `y` is an item to be written.

Syntax: `get <x>`; `get[<x>]`

where `x` is a symbolic file handle.

The simplest example is generating a vector and saving it to disk via `set` dyad. Later we can read it.

```
o)a:!10; f:`:./tmp/test.dat; f set a;
o)b:get f;
o)b
0 1 2 3 4 5 6 7 8 9
o)
```



Remember - `set` verb changes its behaviour based on format of its left argument.

The same idea goes for complex/nested list.

```
o) a:(!10; "123"; `symbol; `a`b`c!1 2 3); f:`:./tmp/test.dat; f set a;
o) b:get f;
o) b
0 1 2 3 4 5 6 7 8 9
"123"
`symbol
`a`b`c!1 2 3
o)
```

Projecting files concept

In order to use projecting into memory, structure must fit into flat vectors of fixed-sized elements.

This concept is served via another pair of verb - `set` and `load`. It works only for vectors, dictionaries and tables.

So the first part is the same - saving a vector/table to disk.

Syntax: `<x> set <y>`; `set[<x>; <y>]`

where `x` is a symbolic file handle (a symbol starting with ":", followed by directory and ending with filename + extension) and `y` is an item to be written.

The projecting part is done via `load` verb. Data will be binding with the symbol in the global namespace named the same as the file.

After projecting, one can apply destructive [amend verbs](#) to change vector contents right on disk.

Assigning anything through `::` to a projected vector variable completes projection and flushes changes to disk.

Syntax: `load <x>`; `load[<x>]`

where `x` is a symbolic file handle.

```
o)a:!10; f:`./tmp/vec.dat; f set a;
o)load f;
o)@[`vec;!10;;10#1]; vec::0;
o)b:get f; b
1 1 1 1 1 1 1 1 1 1
o)
```

Another way of projecting files is working via amends/dmends. Basically, it's just the same projecting/flushing under the hood, but done automatically.



Note trailing slash!

```
o)a:`a`b`c!(1 2 3;1 2 3;1 2 3); f:`./tmp/o_dict/; f set a;
o)@[f;`d`e;;(10#1;20#2)];b:get f; b
a| 1 2 3
b| 1 2 3
c| 1 2 3
d| 1 1 1 1 1 1 1 1 1 1
e| 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
o)
```

Quite similar to dicts is projecting tables.




Please pay attention to trailing slash at the end of the path symbol. That defines saving table as a splayed table. Currently only splayed tables can be saved on disk. For detail see [table on disk](#).

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); f:`../tmp/o_table/; f set a;
o)@[f;`e`d;:(3#1;3#2)]; b:get f; b
a b c e d
-----
1 1 1 1 2
2 2 2 1 2
3 3 3 1 2
o)
```

Of course, projecting via `load` verb works for tables as well.

Conceptually only updating and concatenation at the end is allowed while working with tables on disk.

```
o)a:(+:)`a`b`c!(1 2 3;1 2 3;1 2 3); f:`../tmp/o_table/; f set a;
o).[f;()];;(10 10;20 20;30 30)];
o)b:get f; b
a b c
-----
1 1 1
2 2 2
3 3 3
10 20 30
10 20 30
o)
```

 [load \(scripts\)](#)

 [get](#)

Monadic/Dyadic load

Loads a source file or a plugin library.

Syntax: `load <x>`; `\l <x>`; `load[<x>]`; `load[<x>; <y>]`

Use either a full path (with platform dependent prefixes, suffixes, file extensions) or just a filename. In the last case, extension, prefix, and suffix will be added automatically and kernel will recursively search for a specified file. Optionally, pass a search path in the left argument.

```
o)\l "serde"
"./plugins/serde/libserde.dylib"
o)load "core"
"./std/core.o"
o)load "std/core"
"std/core.o"
o)load["std";"core.o"]
"std/core.o"
```



`load` work fast without recursively search if file extension is dependent.

```
o)load["."; "core"]
"./std/core.o"
o)load["std"; "core.o"]
"std/core.o"
o)load["."; "core.o"]
** I/O error: `load`:
-- ["/core.o"]: Os { code: 2, kind: NotFound, message: "No such file or directory" }
o)
```

[Plugins](#)

[load \(projecting files\)](#)

[read](#)

Dyadic defn

Dynamically defines a function.

Syntax: `<x> defn <y>`; `defn [<x>; <y>]`

where `x` is a symbol vector with argument names, and `y` is a list of expressions.

```
o)f: defn[`a`b; ,(+;`a;`b)]
{[`a`b](+;a;b)}
o)f[1;2]
3
o)
```

Monadic eval

Evaluates parse trees.

Syntax: `eval <x>`; `eval [<x>]`

```
o)eval(+;1;1)
2
o)d:parse "`a`b!(1 2)"
!
`a`b
1 2
o)eval d
a| 1
b| 2
o)d1:parse "3*`a`b!(1 2)"
*
3
(!;`a`b;1 2)
o)eval d1
a| 3
b| 6
o)
```

 [parse](#)

Polyadic reagent

Creates a reagent - an async participant of reactions.

Syntax: `reagent [<...>]`

It's a polyadic function. Argument types and count depend on the reagent type. Reagents can be built-in or plugin extensions.

```
o)r:reagent[`timer;1000;3];
o)react {[x:r] 0N!x};
o)
1000
1001
1001
```

[All reagents](#)

[react](#)

[close](#)

[meta](#)

Polyadic react

Creates reaction on reagents set.

Syntax: `react { [<x>:<r1> (; <y>:<r2> (; ..))] <...> } ; react[[[<x>:<r1> (; <y>:<r2> (; ..))] <...>]]`

Where:

- x, y, .. are arguments of a lambda to be evaluated on reaction triggering;
- r1, r2, .. are reagents involved into reaction.

```
o)r:reagent[`async];
o)react{[x:r] 0N!x};
o)r[123];
123
o)
```

Reactions can be redefined. To do this, just create reaction on the set of reagents already defined.

All `react` with `match` at the end can peek at the content of reagents and have different signatures, so they are not redefined, but new ones are created.



When a reaction is defined, each reagent involved receives a Descriptor. This means "pinning" that reagent onto the Task that has this reaction.

```
o)r1:reagent[`async]; r2:reagent[`async]; react {[x:r1;y:r2] println["// r1: %; r2: %";x;y]};
o)r1[1];r2[2]
// r1: 1; r2: 2
o)// Now redefine reaction:
o)react {[x:r1;y:r2] println["// redefined reaction: r1: %; r2: %";x;y]};
o)r1[1];r2[2]
// redefined reaction: r1: 1; r2: 2
o)
```

If a reagent is dropped, all reactions defined on this reagent will be dropped too since there is no need in them anymore.



If a task has at least one reaction, it's called an IO task. Such task will wait on IO resources involved into reaction(s) and won't be terminated until signal or exception is received or all reactions are dropped.

To drop a reaction, use `react` with the reaction id argument.

```
o)r: reagent[`async];
o)rid: react {[x:r] println["received %";x]};
o)r 4
received 4
o)r 2
received 2
o)react[rid]
o)r 42
o)get r
42
o)
```

Let's see dynamic creation of reagents/reactions in a wide practical example of a ipc server:

```
srv: reagent[`listener;"0.0.0.0:5100"];
react {[x:srv]
  // create new IO task to handle client's session
  spawn[{{cli}
    client: reagent[`ipc;cli]; // create ipc client on accepted sock
    react {[msg:client] // dynamically define reaction on newly created reagent
      client[msg] // echo back to a client
    }
  }];x]
};
```

That's all! Session-based asynchronous server is done. Simple, yes? To see which tasks exist and their state, use [top](#):

```
o)top[]
tid handle      name                state  created      run          iowait      total      load
-----
0  <Reagent#7> "main"             IOWait 19:13:46.141 00:00:00.333 00:00:00.393 00:00:00.726 0
```

[All reagents](#)

[reagent](#)

[react with `match` at the end](#)

[spawn](#)

[top](#)

[Typographic Conventions](#)

Polyadic spawn

Schedules lambda to run on some free scheduler. Returns a join handle.

Syntax: `spawn [<x> (; <y> (; ..))]`

Where:

- x is a lambda to be run;
- y, .. are optional arguments.

```
o)spawn[!0N!(,':x)];!10]
<Reagent#4>
0 0N
1 0
2 1
3 2
4 3
5 4
6 5
7 6
8 7
9 8
o)
```



Returned handle is the reagent. You can create reactions with it, apply a get to it for synchronization or to get results. You can send a signal to the reagent that will stop lambdas and be its result.

```
o)h: spawn {100000{x+1}/1; "done"}
<Reagent#4>
o)get h
"done"
o)
```

```
o)h: spawn {{x+1}/1; "done"};
o)h["killed"]
o)get h
"killed"
```

[reagent](#)

[react](#)

[get](#)

[Typographic Conventions](#)

Monadic top

Shows information about running tasks.

Syntax: `top[]`

```
o)top[]
tid handle      name                state  created      run          iowait      total      load
-----
0  <Reagent#3>  "main"                IOWait 19:13:46.141 00:00:00.333 00:00:00.393 00:00:00.726 0
o)
```

Monadic/Dyadic get

Get data from reagent.

Syntax: `get <x>`; `get [<x>]`; `get [<t>; <x>]`

Where `x` - reagent, `t` - timeout in milliseconds.



`get` will waiting for data and without `t` can lock the task.

```
o)r:reagent[`async];
o)//for example, another task sends data to reagent r
o)spawn {r[1]; r[2]};
o)get r
1
o)get[100;r]
2
o)get[100;r]
** runtime error: `get`:
timeout elapsed
o)//Using trap you can catch timeout error
o)@[get[100;];r;{x`message}]
"timeout elapsed"
o)
```



Do not use `get` after `react` for the same reagent. `react` will capture all data and `get` will lock the task.

[All reagents](#)

[reagent](#)

[react](#)

[trap](#)

[spawn](#)

[top](#)

Monadic close

Deactivates a reagent. If `x` is an IPC or TCP reagent, `close` closes the connection.

Syntax: `close <x>;` `close [<x>]`

where `x` is a reagent:

```
o)r:reagent[`async];
o)react {[x:r] 0N!x};
o)r[1]
1
o)meta r
id | 3
statal `running
type | "async"
o)close[r]
o)meta r
id | 3
statal `closed
type | "async"
o)r[2]
** I/O error: `reagent send`:
-- meta:
id | 3
statal `closed
type | "async"
-- channel is closed
```

[All reagents](#)

[reagent](#)

[react](#)

[meta](#)

Monadic sleep

Schedule current task to sleep `x` milliseconds.

Syntax: `sleep <x>`; `sleep[<x>]`

```
o)sleep[1]
o)
```

Monadic yield

Interrupts the task, falls to a scheduler, continues evaluation later.

Syntax: `yield[]`

```
o)yield[]  
o)
```

Monadic return

Early return from a function

Syntax: `return <x>;` `return[<x>];` `:<x>`

```
o){return 1;2}[]  
1  
o)
```

Monadic exit

Terminates current process with the specified exit code.

Syntax: `exit <x>`; `exit[<x>]`

```
OLOG=warn tachyon -c 4 -f repl
```

```
o)exit 10  
ERROR tachyon > tachyon didn't exit successfully: UserError(10)
```

 [kill](#)

Monadic/Dyadic kill

Sends a kill signal to the task by a task handle or to itself.

Syntax: `kill <c>`; `kill[<c>]`; `kill[<h>; <c>]`

Where:

- `c` is a kill code number;
- `h` is a task handle to be killed.

```
o)h: spawn { r:reagent[`async]; react {[x:r] 0N!x}};
o)top[]
tid handle      name                state  created      run          iowait      total      load
-----
0  "main"         Running 11:54:05.200 00:00:00.398 00:00:08.559 00:00:08.957 0
3  "r:reagent[`async]; r.." IOWait 11:54:10.850 00:00:00.001 00:00:03.306 00:00:03.307 0
o)kill[h;101]
o)h: spawn { r:reagent[`async]; react {[x:r] 0N!x}};
o)top[]
tid handle      name                state  created      run          iowait      total      load
-----
0  "main"         Running 11:54:05.200 00:00:00.410 00:01:27.216 00:01:27.626 0
8  "r:reagent[`async]; r.." IOWait 11:55:26.500 00:00:00.001 00:00:06.326 00:00:06.327 0
o)kill[h;0]
o)top[]
tid handle      name  state  created      run          iowait      total      load
-----
0  "main" Running 11:54:05.200 00:00:00.414 00:01:51.654 00:01:52.068 0
o)spawn {kill[102]}

o)kill 0
```

[exit](#)

[spawn](#)

[top](#)

[react](#)

[reagent](#)

[All reagents](#)

Monadic panic

Raises panic exception

Syntax: `panic <x>;` `panic [<x>]`

```
o)panic 1
** runtime error: `(apply_dyad;put;#1;(apply_monad;eval;(apply_dyad;parse;"REPL";#0)))`:
Undefined panic
o)panic "panic"
** runtime error: `(apply_dyad;put;#1;(apply_monad;eval;(apply_dyad;parse;"REPL";#0)))`:
panic
o@[{{panic["test panic"]};();{key x}}]
`kind`call`message`stack`mark
o@[{{panic["test panic"]};();{x`kind`message}}]
`runtime
"test panic"
o)
```

Dyadic set

Sets global variable or persists in a file.

Syntax: `<x> set <y>`; `set [<x>; <y>]`

```
o)`x set 100
100
o){set[`a;1]}[];
o)a
1
o)t:+`a`s!(1 2;3 4)
a s
---
1 3
2 4
o)`:/tmp/db/ set t
`:/tmp/db/
o)
```



Please pay attention to trailing slash at the end of the path symbol. That defines saving table as a splayed table. Currently only splayed tables can be saved on disk. For detail see [table on disk](#).

 get

Monadic get

Reads AST value from a file specified in `x` as a symbol.

Syntax: `get <x>`; `get [<x>]`

```
o)t:+'a`s`d!(1 2 3;4 5 6;7 8 9)
a s d
-----
1 4 7
2 5 8
3 6 9
o)`:db/ set t
`:db/
o)get `:db/
a s d
-----
1 4 7
2 5 8
3 6 9
o)
```



Please pay attention to trailing slash at the end of the path symbol. That defines saving table as a splayed table. Currently only splayed tables can be saved on disk. For detail see [table on disk](#).

 [set](#)

Dyadic write

Writes a string or a byte vector to a file.

Syntax: `<x> write <y>;` `write[<x>; <y>]`

```
o)"aaa" write `:/tmp/t.txt
`:/tmp/t.txt
o)0x010203010203aff write `:/tmp/t.txt
`:/tmp/t.txt
o)
```

 read

Monadic read

Reads file contents.

Syntax: `read <x>`; `read[<x>]`

```
o)OHOME:getenv[`OHOME];
o)read `$:",OHOME,"/std/core.o"
0x2f2f20436f72652066756e6374696f6e730a2f2f2072756e20636f72655f746573742e6f20746f20636865636b0a2f2f202d2d0a0a2f2f2046756e..
o)"c"$ read `$:",OHOME,"/std/core.o"
"// Core functions\n// run core_test.o to check\n// --\n\n// Function for generating tests\ntest:{{[tnm;lhs;rhs]\n    $[l..
o)
```

 write

 load

 cast

Monadic readln

Reads a line from `x` fd.

Syntax: `readln <x>`; `readln [<x>]`

```
o)s:readln[0];  
1 2  
o)s  
"1 2"  
o)
```

Polyadic print

Prints down a string representation of `x` without quotes. Format string contains `%` at each position to be replaced by formatted value.

Syntax: `print <string>`; `print[<string>]`; `print[" (<...>) (% (<...>) (..)) " (:x (; ..))]`

```
o)print["The final result is %\n";42]
The final result is 42
o)print["a is %, b is %\n";1;2 3 4 5]
a is 1, b is 2 3 4 5
o)
```

[println](#)

[format](#)

[Typographic Conventions](#)

Polyadic println

Prints down a string line representation of `x`. Format string contains `%` at each position to be replaced by formatted value.

Syntax: `println <string>; println[<string>]; println[" (<...>) (% (<...>) (..)) " (;x (; ..))]`

```
o)println["The final result is %";42]
The final result is 42
o)println["a is %, b is %";1;2 3 4 5]
a is 1, b is 2 3 4 5
o)
```

[print](#)

[format](#)

[Typographic Conventions](#)

Monadic show

Prints down full contents of a value.

Syntax: `show <x>`; `show [<x>]`

Be careful using `show` with large values cause it can take much time to print all of the contents.

```
o)l:!1000;
o)l
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 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
o)show l
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 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
o)
```

`show` can also be used to see intermediate lambda results:

```
o)f:{a:1 2 3 4;+/a}
{a:1 2 3 4;+/a}
o)f[]
10
o)f:{show a:1 2 3 4;+/a}
{show a:1 2 3 4;+/a}
o)f[]
1 2 3 4
10
o)
```

Monadic ON!

Write to console and return.

Syntax: `ON!<x>`

Returns x after printing its unformatted representation to the console.

```
o)ON!"first\nsecond"
"first\nsecond"
"first\nsecond"
o)println "first\nsecond"
first
second
```

With ON! you can see the current value of any part of the expression. It useful for debugging.

```
o)*/ON!1+ON!!5
0 1 2 3 4
1 2 3 4 5
120
o)
```



Results of print, println, show and `ON!` can be redirected. `-1!` works like `ON!` but non-redirectable. You can redirect STDERR with `.o.err`. `-2!` write to STDERR and return result without redirection.

```
r:reagent[`async];
oldout:@[.o.out];ON;ON];
//redirect stdout to `async reagent
.o.out::r;
ON!"test ON!";
-1!"test -1!";
//stop redirecting
.o.out::oldout;
println "";
println get r;

"test -1!"

"test ON!"
```

[println](#)

[show](#)

Dyadic @ (at)

Indexes left argument (vector/list/dict) by right argument.

Syntax: `<x> @ <y>`; `@ [<x>; <y>]`

```
o)1 2 3@0 1
1 2
o)@[til 10;2 8]
2 8
o)(`a`b!1 2)@`b`a`c
2 1 0N
o)t:([]a:1 2 3;b:`a`b`c)
a b
---
1 a
2 b
3 c
o)t@2
a| 3
b| `c
o)
```

 . (indexing in depth)

 @ (internal type id)

 @ (triadic amend)

 @ (tetradic amend)

Dyadic . (dot-apply)

Indexes left argument (vector/list/dict) in depth by right argument.

Syntax: `<x> . <y>;` `.[<x>; <y>]`

```
o)(1 2;3 4).(0 1)
2
o)a:((1 2 3;4 5);(6 7 8 9; 10)).(1 0 2)
8
o)(1.1 1.2;1.3 1.3 1.3).(0 2)
0n
o)
```

If the left argument is a verb or a lambda, then it is applied to the right argument.

```
o){x+y} . 1 2
3
o)format . ("test % %";1;2)
"test 1 2"
o)
```

Works like a monadic value for a list with a verb or lambda at the beginning.

```
o). ({x+y};1;2)
3
o). (format;"test % %";1;2)
"test 1 2"
o)value (format;"test % %";1;2)
"test 1 2"
o)
```

 [. \(value\)](#)

 [@ \(indexing\)](#)

 [triadic dmend](#)

 [tetradic dmend](#)

Monadic count

Returns the number of items in a list, dictionary or table.

Syntax: `count <x>`; `count [<x>]`; `#<x>`; `# [<x>]`

```
o)count 1 2 3
3
o)count til 10
10
o)count "one"
3
o)count `a`b`c!1 2 3
3
o)count flip `a`b`c!1 2 3
1
o)count([a:`a`b`c;b:1 2 3)
3
o)
```

Returns 1 for everything else:

```
o)count 0
1
o)count `ten
1
o)count({x+1})
1
o)
```

To count elements of nested structures, use `each`:

```
o)count(1 2;3;45 67 89)
3
o)count each(1 2;3;45 67 89)
2 1 3
o)count `x`y`z!(1 2 3;`a`b`c;1.1 2.2 3.3)
3
o)count each `x`y`z!(1 2 3;`a`b`c;1.1 2.2 3.3)
x| 3
y| 3
z| 3
o)
```

`#` (attaching attribute)

`rc` (reference count)

`#` (take)

Monadic distinct

Returns unique set of elements from it's argument.

Syntax: `?<x>`; `? [<x>]`

```
o)?1 2 3 1 2 1 2
1 2 3
o)
```

Returns unique rows for tables:

```
o)t:([[]a:1 2 3 3;b:1.1 2.2 1.5 1.5)
a b
-----
1 1.1
2 2.2
3 1.5
3 1.5
o)distinct t
a b
-----
1 1.1
2 2.2
3 1.5
o)
```

 ? (vector conditional)

 ? (search)

Polyadic enlist

Takes an arbitrary number of arguments and produces a list as a result. If all elements are of the same type, the result will be collapsed to a vector.

Syntax: `enlist <x>`; `enlist[<x>]`; `enlist[<x> (: <y> (: ..))]`

```
o)enlist[1;2 3;"asd"]
1
2 3
"asd"
o)enlist[1;2;3;4]
1 2 3 4
o)a:0
0
o)b:enlist a
,0
o)c:enlist b
,,0
o)type each(a;b;c)
`s`long
`v`long
`v`l
o)
```

 [first](#)

 [pick](#)

 [Typographic Conventions](#)

Polyadic pick

Takes an arbitrary number of arguments and produces a list as a result. Unlike [enlist](#) it doesn't collapse to a vector if all elements are of the same type. Sometimes it is useful for generic list creation;

Syntax: `pick[<x> (; <y> (; ..))] ; ([<x> (; <y> (; ..)))`

```
o)pick[1;2;3;4]
1
2
3
4
o)([] 1;2;3;4)
1
2
3
4
o)
```



To convert a vector to a list use `pick . x`

```
o)v:!5;
o)pick . v
0
1
2
3
4
o)
```

[first](#)

[enlist](#)

[Typographic Conventions](#)

Dyadic ^ (fill)

Replaces nulls in it's right argument by value provided in a left argument

Syntax: `<x> ^ <y>;` `^ [<x>; <y>]`

```
o)10^1 2 3 0N 4 0N 6
1 2 3 10 4 10 6
o)
```

Fill with scan replace nulls with last non-null item

```
o)^\1 2 3 0N 4 0N 6
1 2 3 3 4 4 6
o)^\0N 0N 1 2 3 0N 4 0N 6
0N 0N 1 2 3 3 4 4 6
o)5 ^\ 0N 0N 1 2 3 0N 4 0N 6
5 5 1 2 3 3 4 4 6
o)
```

 [^ \(null\)](#)

 [iterators](#)

Dyadic filter

Returns elements from `y` for which `x` expression is true.

Syntax: `<x> filter <y>`; `filter[<x>; <y>]`

```
o)(0=x mod 2)filter x:1 5 6 8 11 17 20 21
6 8 20
o)filter[0=x mod 2;x:1 5 6 8 11 17 20 21]
6 8 20
o)(x>10)filter x:1 5 6 8 11 17 20 21
11 17 20 21
o)
```

 \$ (conditional)

Monadic first

Returns the first element from `x` if `x` is a list, else returns `x`.

Syntax: `first <x>`; `first[<x>]`; `*<x>`

```
o)first 1 2 3 4
1
o)first 1
1
o)* 1 2 3 4
1
o)* 1
1
o)first ("asd";1 2 3 4)
"asd"
o)first each ("asd";1 2 3 4)
"asd"
1
o)
```

 [last](#)

 [enlist](#)

Monadic last

Returns the last element from `x` if `x` is list, else returns `x`.

Syntax: `last <x>`; `last[<x>]`; `_<x>`

```
o)last 1 2 3 4
4
o)last 1
1
o)_ 1 2 3 4
4
o)_ 1
1
o)last ("asd";1 2 3 4)
1 2 3 4
o)last each (1; 2 3 4; `five `six)
1
4
`six
o)
```



Do not use `_` at the beginning of names. You get last after binding.

```
o)_t:1 2 3
3
o)
```

 [first](#)

Monadic | (reverse)

Reverses the order of elements in a list.

Syntax: | <x>; | [<x>]

```
o)| 1 2 3
3 2 1
o)a:(1 2 3;4 5;enlist 6)
1 2 3
4 5
,6
o)|a
,6
4 5
1 2 3
o)|a 1
5 4
o)|'a
3 2 1
5 4
,6
o)
```

Dyadic rotate

Rotates elements in a list y. Positive x means rotate to the left, negative - to the right.

Syntax: `<x> rotate <y>`; `rotate[<x>; <y>]`

```
o)1 rotate 1 2 3
2 3 1
o)v:("123";1 2 3;1f);
o).[`v;();~[rotate];-4];
o)v
1f
"123"
1 2 3
```

Dyadic shift

Shifts elements in a list y. Positive x means shift to the left, negative - to the right. Null are shifted in from the other end.

Syntax: `<x> shift <y>`; `shift[<x>; <y>]`

```
o)1 shift 1 2 3
2 3 0N
o)v:("123";1 2 3;1f);
o).[`v;();~[shift];-1];
o)v
0N0
"123"
1 2 3
```

Dyadic take

Takes first `x` elements of `y`.

Syntax: `<x> # <y>`; `# [<x>; <y>]`

where `x` is an integer atom or vector, `y` is an atom, list, dictionary, or table:

```
o)2#1 2 3 4 5
1 2
o)1#(`a`b!(1;2))
a| 1
o)3#(⎕a:1 2 3 4;b:1.1 2.2 3.3 4.4)
a| b
-----
1| 1.1
2| 2.2
3| 3.3
o)
```

If `x` is an integer vector, a matrix with `count x` dimensions is created:

```
o)3 4#1
1 1 1 1
1 1 1 1
1 1 1 1
o)
```

If the size of `y` is less than `x`, the verb goes over `y` again:

```
o)2#1
1 1
o)5#1 2
1 2 1 2 1
o)4#(`a`b!(1;2))
a| 1
b| 2
a| 1
b| 2
o)
```

For negative `x`, the verb takes elements from the reversed `y`:

```
o)-3# til 10
7 8 9
o)-1#(⎕a:1 2 3 4;b:1.1 2.2 3.3 4.4)
a| b
-----
4| 4.4
o)
```

For 0 in the first argument, `take` returns an empty list:

```
o)0#1 2 3
`long$()
o)0#`a`b`c
`symbol$()
o)0#(`a`b!(1;2))
`symbol$()| `long$()
o)
```

 # (attaching attribute)

 # (count)

Dyadic _ (cut, drop)

Cut

It cuts a list or a vector into parts and drops the first part. Only positive indices supported.

Syntax: `<x> _ <y>`; `_[<x>; <y>]`

```
o)2 4 6 _ til 10
2 3
4 5
6 7 8 9
o)
```

Drop

Returns subset of items from a list, vector, dict and table starting with the item under index `x` (starting from the end if `x` is a negative number)

Syntax: `<x> _ <y>`; `_[<x>; <y>]`

```
o)1 _ 1 2 3
2 3
o)2 _ 1 2 3
,3
o)-1 _ 1 2 3
1 2
o)d:`a`a`b!1 1 2
a| 1
a| 1
b| 2
o)-1 _ d
a| 1
a| 1
o)t:([]a:1 2 3 3;b:1.1 2.2 1.5 1.5)
a b
-----
1 1.1
2 2.2
3 1.5
3 1.5
o)2 _ t
a b
-----
3 1.5
3 1.5
o)
```

You can use the key for dicts and tables in drop.

```
o)d: `a`b`c!(1 2;3 4;5 6)
a| 1 2
b| 3 4
c| 5 6
o)`b _ d
a| 1 2
c| 5 6
o)t: +d
a b c
-----
1 3 5
2 4 6
o)`b _ t
a c
---
1 5
2 6
o)
```



Be careful. Drop with a key may not work for a dictionary with numeric keys.

And for string keys, the drop will not work at all. Instead of strings, it is better to use symbols.

```
o)d: (!5)!(`a`b;1 3;"qwerty";3 5i;`c`d)
0| `a`b
1| 1 3
2| "qwerty"
3| 3 5i
4| `c`d
o)2 _ d
2| "qwerty"
3| 3 5i
4| `c`d
```



Drop for big data works slowly! Do not use drop with big data!

Dyadic .o.cut

Cuts a list or vector into pieces of the specified size.

Syntax: `<size> .o.cut <v>`; `.o.cut[<size>; <v>]`

```
o).o.cut[3; til 14]
0 1 2
3 4 5
6 7 8
9 10 11
12 13
o).o.cut[,3;!14]
0 1 2 3 4
5 6 7 8 9
10 11 12 13
o)
```

Dyadic ? (search)

Searches for the first occurrence of its right argument in the left one. Returns position index as a number or null if not found.

Syntax: `<x> ? <y>`; `? [<x>; <y>]`

```
o)1 2 3 4 5 6?3
2
o)1 2 3 4 5 6?9
0N
o)
```

If `x` is a vector, `?` searches for `y` in the left argument.

```
o)1 2 3?2 3 4
1 2 0N
o)
o)1 2 3?(1;2;3 4)
0
1
2 0N
o)
```

If `x` is a list of lists and `y` is a simple list, `?` searches for `y` in items of `x`:

```
o)(1 2 3;(4 5 6))?4 5 6
1
o)
```

[?](#) (distinct)

[?](#) (vector conditional)

[?](#) Queries

Monadic ! (til)

Returns a range of natural numbers from 0 til the argument (excluding the argument).

Syntax: `!<x>`; `![<x>]`; `til <x>`; `til [<x>]`

```
o)til 1b
,0
o)!10
0 1 2 3 4 5 6 7 8 9
o)til 0
`long$( )
o)10+til 5
10 11 12 13 14
o)(til 2)*5
0 5
o)!2*5
0 1 2 3 4 5 6 7 8 9
o)
```

The argument must be a non-negative integer atom:

```
o)til 10f
** eval error: `key`:
invalid type: [s`float]
```

 [! \(internal type id\)](#)

 [! in dicts](#)

Monadic ~ (not)

Inverts boolean vector. Right atomic.

Syntax: `~<x>`; `not <x>`; `~[<x>]`; `not [<x>]`

```
o)~1010b
0101b
o)~1<2
0b
o)~(1>2 0 -1)
100b
o)
```

... or checks for zeroes if `x` is a number:

```
o)~1 0 3f
010b
o)~(100 - 10 100 1000)
010b
o)
```

 - (negate)

Dyadic | (or/max)

Applies boolean "or" for bool arguments. Fully atomic.

Syntax: `<x> | <y>`; `<x> or <y>`; `| [<x>; <y>]`; `or [<x>; <y>]`

```
o)1b|010b
111b
o)(1>2)|2<1
0b
o)a:1010b
1010b
o)a|~a
1111b
o)
```

For number vectors, it results in calculating "max".

```
o)1 2 3|0
1 2 3
o)1 2 3|3 2 1
3 2 3
o)
```

 [& \(and/min\)](#)

Dyadic & (and/min)

Applies boolean "and" for bool arguments. Fully atomic.

Syntax: `<x> & <y>`; `<x> and <y>`; `&[<x>; <y>]`; `and[<x>; <y>]`

```
o)10b&1b
10b
o)a:1010b
1010b
o)a&~a
0000b
o)
```

For number vectors, it results in calculating "min".

```
o)1 2 3&0 2 1
0 2 1
o)0&(-2 + til 5)
-2 -1 0 0 0
o)
```

 | (or/max)

 & (where)

Monadic & (where)

Returns integer vector of true positions in boolean vectors.

Syntax: `&<x>`; `where <x>`; `&[<x>]`; `where [<x>]`

```
o)&0101b
1 3
o)&(1>-1 0 1)
0 1
o)
```

When applied to integer vector, generates N indices of corresponding elements.

```
o)&1 2 3
0 1 1 2 2 2
o)where 2 3 0 1
0 0 1 1 1 3
o)
```

 [& \(and/minimum\)](#)

Monadic - (negate)


Inverts signs of numbers. Right atomic.

Syntax: `-<x>`; `- [<x>]`; `neg <x>`; `neg [<x>]`

```
o)-(1 2 3)
-1 -2 -3
o)neg(1 2 3)
-1 -2 -3
o)neg(1 2 3; -4 -5)
-1 -2 -3
4 5
o)neg 01001b
10110b
o)
o)-(2<3)
0b
o)
```

Null does not have a sign:

```
o)neg (0W;-0w;0N)
-0W
0w
0N
o)
```


 `~` (not)

Monadic % (reciprocal)

Calculates reciprocal (1/x) in vector/list. Fully atomic. Only floats are supported.

Syntax: %<x>; % [<x>]

```
o)%1 2 3f
1 0.5 0.3333333333333333
o)%10.12
0.09881422924901187
o)%-2f
-0.5
o)
```

 % (dyadic division)

Dyadic + (plus)

Adds scalar/vector elements. Fully atomic.

Syntax: `<x> + <y>`; `+ [<x>; <y>]`

Where `x` and `y` are numeric scalars or vectors, returns their sum.

```
o)1+2
3
o)2+3 4 5
5 6 7
o)1 2 3+4
5 6 7
o)1 2 3+4 5 6
5 7 9
o)1 2 3+4 5 6 7
** eval error: `+`:
arguments length mismatch: [1 2 3;4 5 6 7]
o)
```

Where numeric value is added to a dictionary or a table, the verb adds up the numeric value to numeric values in the dict/table:

```
o)10+`a`b`c!1 2 3
a| 11
b| 12
c| 13
o)([sym:`a`b`c]x:1.5 2.2 0.3; y:20.4 13.0 2.5)+0.5
x  y
-----
2  20.9
2.7 13.5
0.8 3
o)
```

Dyadic - (minus)

Subtracts scalar/vector elements. Fully atomic.

Syntax: `<x>-<y>`; `-[<x>; <y>]`

```
o)2-1
1
o)(til 5)-10
-10 -9 -8 -7 -6
o)1 2 3 - 0 5 7
1 -3 -4
o)
```



Spaces around minus are important!

Let's see few examples about it:

```
o)5 - 2
3
o)5 -2
5 -2
o)- 2 3
-2 -3
o)-2 3
-2 3
```

Where numeric value is subtracted from a dictionary or a table, the verb subtracts the numeric value from numeric values in the dict/table:

```
o)`a`b`c!100 200 300 - 10
a| 90
b| 190
c| 290
o)([sym:`a`b`c]x:1.5 2.2 0.3; y:2021 13.0 2.5)-10.34
x      y
-----
-8.84 2010.66
-8.14 2.66
-10.04 -7.84
o)
```

Dyadic * (multiplication)

Multiplies scalar/vector elements. Fully atomic.

Syntax: `<x> * <y>`; `*[<x>; <y>]`

```
o)2*2
4
o)2*1 2 3
2 4 6
o)1 2 3*2
2 4 6
o)1 2 3f * 1.1
1.1 2.2 3.30000000000000003
o)1 2 3*4 5 6
4 10 18
o)1 2 3*4 5 6 7
** eval error: `*`:
arguments length mismatch: [1 2 3;4 5 6 7]
```

Where `x` is a dictionary and `y` is a numeric value, the numeric values of `x` are multiplied by `y`:

```
o)`a`b`c!5 10 20f*1.1
a| 5.5
b| 11
c| 22
o)
```

Dyadic % (division)

Divides scalar/vector elements. Fully atomic.

Syntax: `<x> % <y>;` `% [<x>; <y>]`

Where `x` and `y` are scalars or vectors:

```
o)1%2
0
o)2%2
1
o)2%1
2
o)-7 % 2
-3
o)2f%2f
1f
o)2f%3f
0.6666666666666666
o)2f%3 5 4 3 2f
0.6666666666666666 0.4 0.5 0.6666666666666666 1
o)
o)1 -2 3f % 10 20 30f
0.1 -0.1 0.1
o)
```

 % (monadic division)

 mod (division remainder)

Dyadic mod

Returns the `<x> % <y>` division remainder.

Syntax: `<x> mod <y>`; `mod[<x>; <y>]`

```
o)1 mod 2
1
o)2 mod 2
0
o)-10 -17 -103 mod 5
0 -2 -3
o)10 20 30 mod 3 2 7
1 0 2
o)3 2 7 mod 10 20 30
3 2 7
o)
```

```
o)4 mod 6 5 4 3 2
4 4 0 1 0
o)(11 15;18 19 20) mod 5
1 0
3 4 0
o)10 19 mod 3 4
1 3
o)
```

```
o)-7 % 3
-2
o)(-2*3)+(-7 mod 3)
-7
o)
```

 [% \(dyadic division\)](#)

Monadic absolute value

Returns the absolute value of a numeral or temporal, returns null for null.

Syntax: `abs <x>`; `abs[<x>]`

```
o)abs 1.0 -10.5 0
1 10.5 0
o)abs -11:08
11:08
```

`abs` is fully atomic:

```
o)abs (1;-2 3;-4 5 6.0)
1
2 3
4 5 6f
```

Monadic ceil

`ceil` returns the least integer greater or equal to the argument.

Syntax: `ceil <x>`; `ceil [<x>]`

```
o)ceil 10.5
11f
o)ceil -10.5
-10f
o)ceil -2.5 0 2.5
-2 -0 -2f
o)abs ceil -1.2
1f
o)
```

The function is atomic:

```
o)ceil(2.3 4.5;6.7)
3 5f
7f
```

 [floor](#)

 [round](#)

 [frac](#)

Monadic floor

`floor` returns the greatest integer less or equal to the argument.

Syntax: `floor <x>`; `floor[<x>]`

```
o)floor[10.5]
10f
o)floor[-10.5]
-11f
o)floor[-2.5 0 2.5]
-3 0 2f
o)abs floor[-1.2]
2f
o)
```

The function is atomic:

```
o)floor[(2.3 4.5;6.7)]
2 4f
6f
o)
```

[ceil](#)

[round](#)

[frac](#)

Monadic frac

`frac` returns the fractional part of float for the argument.

Syntax: `frac <x>`; `frac[<x>]`

```
o)frac[10.5]
0.5
o)frac[-10.5]
-0.5
o)frac[(-2.5; 0.0; 2.5)]
-0.5 0 0.5
o)frac[-1.14 0.14 2.14]
-0.14 0.14 0.14
o)
```

The function is atomic:

```
o)frac(2.3 4.5;6.7)
0.3 0.5
0.7
```

[ceil](#)

[floor](#)

[round](#)

Dyadic & (and/min)

Applies boolean "and" for bool arguments. Fully atomic.

Syntax: `<x> & <y>`; `<x> and <y>`; `&[<x>; <y>]`; `and[<x>; <y>]`

```
o)10b&1b
10b
o)a:1010b
1010b
o)a&~a
0000b
o)
```

For number vectors, it results in calculating "min".

```
o)1 2 3&0 2 1
0 2 1
o)0&(-2 + til 5)
-2 -1 0 0 0
o)
```

 | (or/max)

 & (where)

Dyadic | (or/max)

Applies boolean "or" for bool arguments. Fully atomic.

Syntax: `<x> | <y>`; `<x> or <y>`; `| [<x>; <y>]`; `or [<x>; <y>]`

```
o)1b|010b
111b
o)(1>2)|2<1
0b
o)a:1010b
1010b
o)a|~a
1111b
o)
```

For number vectors, it results in calculating "max".

```
o)1 2 3|0
1 2 3
o)1 2 3|3 2 1
3 2 3
o)
```

 [& \(and/min\)](#)

Monadic round

`round` returns the rounded integer for the argument.

Syntax: `round <x>`; `round[<x>]`

```
o)round 10.5
11f
o)round -10.5
-11f
o)round -2.5 0 2.5
-3 -0 -3f
o)abs round -1.2
1f
o)
```

The function is atomic:

```
o)round(2.3 4.5;6.7)
2 5f
7f
```

 [ceil](#)

 [floor](#)

 [frac](#)

More Math Functions

Following fully atomic monadic intrinsics are supported for floats only.

Name	Function	Comment
sin	sine	argument in radians
asin	arcsine	returns radians
cos	cosine	argument in radians
acos	arccosine	returns radians
tan	tangent	argument in radians
atan	arctangent	returns radians
exp	e^x	Raise e to x power
log	$\ln x$	Natural logarithm of x
sqrt	sqrt	Square root of x

```
o)cos 0f
1f
o)acos 0.5
1.0471975511965976
o)exp 1f
2.718281828459045
o)log 10f
2.302585092994046
o)sqrt 1.5
1.224744871391589
```

Another set of dyadic fully atomic intrinsics are supported (floats only).

Name	Function	Comment
xexp	$x ^ y$	Raise x to a power y
xlog	$\log_x(y)$	Returns base-x logarithm of y

```
o)2f xexp 10f
1024f
o)10f xexp 1.2
15.848931924611133
o)5f xlog 25f
2f
o)100f xlog 1f
0f
o)1.2 xlog 2.5
5.025685102665476
```

Dyadic < (less than) and <= (up to)

Returns `1b` where `x` is less than (or up to) `y`. Fully atomic.

Syntax: `<x> < <y>`; `<x> <= <y>`; `<[<x>; <y>]`; `<=[<x>; <y>]`

```
o)1 0 3<1
010b
o)10 20 30<=20 20 20
110b
o)(0;5 1)>(2 -2;3)
01b
10b
o)0.5>0 1 2 3f
1000b
o)
```

 [>, >= \(greater than, at least\)](#)

Dyadic > (greater) and >= (greater or equal)

Returns `1b` where `x` is greater than (or at least) `y`. Fully atomic.

Syntax: `<x> > <y>`; `<x> >= <y>`; `>[<x>; <y>]`; `>=[<x>; <y>]`

```
o)1 -1 11 > 1
001b
o)111 222 333>=100 300 333
101b
o)(0;5 2)>(2 -2;3)
01b
10b
o)1.1>0 1 2f
110b
o)
```

 [<, <= \(less than, up to\)](#)

Dyadic = (equal)

Returns `1b` where `x` and `y` or their elements are equal. Fully atomic.

Syntax: `<x> = <y>`; `=[<x>; <y>]`

```
o)1 0 3 = 1 2 3
101b
o)(1;2 0)=(3 1;0)
01b
01b
o)
o)10:00:00=10:00:01
0b
o)
```

 `<>` (not equal)

 `~` (match)

Dyadic ~ (match)

Returns `1b` where `x` and `y` are identical.

Syntax: `<x> ~ <y>`; `~[<x>; <y>]`

```
o) 1 0 3 ~ 1 2 3
0b
o) 1 2 3 ~ 1 2 3
1b
o)
```

This verb compares both values and types:

```
o) 1~1.0
0b
o) 1f~10f%10f
1b
o) 0~00:00:00
0b
o) `a~"a"
0b
o)
```

Match ignores attributes:

```
o) 1:2*!5
0 2 4 6 8
o) 1~`s#1
1b
o) 1~`g#1
1b
```

`⊞` = (equal)

`⊞` match (patterns)

Dyadic <> (not equal)

Returns `1b` where `x` and `y` or their elements are not equal.

Syntax: `<x> <> <y>`; `<> [<x>; <y>]`

```
o)1<>2
1b
o)
o)1<>1
0b
o)1 0<>0 1
11b
o)1 0 <> 1
01b
o)(1; 2 3)<>(1 3;2)
01b
01b
o)
```


`⊞` = (equal)

Monadic ^ (null)

Checks if `x` is null.

Syntax: `null <x>`; `null [<x>]`; `^<x>`; `^ [<x>]`

```
o)^1
0b
o)^0101b
0000b
o)^0N 1 2
100b
o)null 0
0b
o)null 0 1 2 3
0000b
o)null 0 1 2 3 0N
00001b
o)a:!9;
o)^ a?9
1b
o)dict:`a`b!(1 2);
o)^ dict[`c]
1b
o)
```

 ~ (not)

 ^ (fill)

Dyadic intersection

A standard set functions are implemented for simple vectors. That is vectors are treated as sets of values. No attributes are expected to be attached to input vectors. No particular order is guaranteed.

Set intersection is a dyadic verb. Returns only those values present in both left and right arguments.

Syntax: `<x> sect <y>`; `sect [<x>; <y>]`

```
o)0 1 2 3 4 sect 4 0
0 4
o)`a`b`c sect `b`v
,`b
o)(0 1; 2; 3 5 6) sect (-3 -2 -1;0 1; 2 3 4)
,0 1
o)
```

Dyadic difference

Syntax: `<x> diff <y>`; `diff[<x>; <y>]`

Set difference is a dyadic verb. Returns values which appear in `x` but not in `y`.

```
o)0 1 2 3 4 diff 4 0
1 2 3
o)`a`b`c diff `b`v
`a`c
o)(0 1; 2; 3 5 6) diff (-3 -2 -1;0 1; 2 3 4)
2
3 5 6
o)
```

Dyadic union

Set union is a dyadic verb. Returns all values from both arguments once.

Syntax: `<x> union <y>`; `union[<x>; <y>]`

```
o)1 2 3 4 union 4 0
1 2 3 4 0
o)distinct 10 12 31 31 65, 12 14 65 81 // same as distinct on join
10 12 31 65 14 81
o)
```

Dyadic in (contains)

A dyadic verb that returns `1b` for each right value present in the left argument.

Syntax: `<x> in <y>`; `in[<x>; <y>]`

```
o)1 in 1
1b
o)1 in 10
0b
o)1 in 1 0
1b
o) 1 2 3 in 3 4 5
001b
o)
```

`in` is often used in queries:

```
o)t:([]a:1 2 3;b:1.1 2.2 3.3;c:`x`y`z)
a b c
-----
1 1.1 x
2 2.2 y
3 3.3 z
o)load "sql";
o)select from t where a in 1 2
a b c
-----
1 1.1 x
2 2.2 y
o)
```

 [queries](#)

 [in \(select condition\)](#)

Dyadic like

Returns boolean where `x` matches `regex` in `y`. `x` can be a symbol atom, a list of symbols, a string, or a list of strings. `y` must be a string. `like` is case-sensitive.


Syntax: `<x> like <y>`; `like[<x>; <y>]`

```
o)"asd" like "A"
0b
o)"asd" like "a"
1b
o)`asd like "a"
1b
o)("asd";"asfs";"sfh") like "a"
110b
o)("asd";"asfs";"sfh") like "asd"
100b
o)"dark" like "[bd]ark"
1b
o)"darker" like "dark*"
1b
o)
```

If `y` is a substring of `x` or matches it, `like` returns `1b`. If `x` is a substring of `y`, `like` returns `0b`:

```
o)"darker" like "dark"
1b
o)"dark" like "darker"
0b
o)
```

 [ss \(string search\)](#)

 [ssr \(string search and replace\)](#)

Dyadic ss (string search)

Returns an int vector of position(s) of first element(s) of substrings in `x` that match pattern `y`.

Syntax: `<x> ss <y>`; `ss [<x>; <y>]`

Where `y` is a `regex`, `x` is a string to be searched for matches of `y`.

```
o)ss["MSFT"; "^MS"]
,0
o)ss["MSFT"; "[A-Z]"]
0 1 2 3
o)s:"rent a tent"
"rent a tent"
o)s ss "ent"
1 8
o)s ss "[rt]ent"
0 7
o)
```

 [like](#)

 [ssr \(string search and replace\)](#)

Triadic ssr (string search and replace)

Replaces the found substring.

Syntax: `ssr[<x>; <y>; <z>]`

Where:

- `x` is a string to be searched for matches;
- `y` is a `regex` for search;
- `z` is a string to substitute matches, or lambda to be called on each matching substring.

```
o)ssr["ababa ga lamaga"; "[bg]a"; "*"]
"a** *lama*"
o)ssr["ababa ga lamaga"; "[bg]a"; "{0N!x; "*"}]
"ba"
"ba"
"ga"
"ga"
"a** *lama*"
o)ssr["rent a tent"; " "; "0"]
"rent0a0tent"
o)
```

 [like](#)

 [ss \(string search\)](#)

Monadic parse

Parses `x` to an AST (abstract syntax tree).

Syntax: `parse <x>`; `parse[<x>]`

```
o)parse "1+2"  
+  
1  
2  
o)parse "`a`b!(1;2)"  
!  
`a`b  
(,;1;2)  
o)parse "1 2 3 +neg 5 1 7"  
+  
1 2 3  
((-:);5 1 7)  
o)
```

Execute a parse tree with `eval`:

```
o)x:parse "1 2 3 + 3 4 5"  
+  
1 2 3  
3 4 5  
o)eval x  
4 6 8  
o)
```

`eval`

`quote`

Polyadic format

Returns string representation of `x` with quotes. Format string contains `%` at each position to be replaced by formatted value.

Syntax: `format["<...> (% (<...>) (..)) " (;x (; ..))]`

```
o)format["The final result is %";42]
"The final result is 42"
o)format["a is %, b is %";1;2 3 4 5]
"a is 1, b is 2 3 4 5"
o)
```

 [print](#)

 [println](#)

 [Typographic Conventions](#)

Monadic .o.lower

Returns lowercased string or symbol.

Syntax: `.o.lower <x>`; `.o.lower [<x>]`

```
o).o.lower "A"
"a"
o).o.lower["Qwerty"]
"qwerty"
o).o.lower `TEST
`test
```

If some chars requires special considerations (e.g. multiple chars) they given by [SpecialCasing.txt](#)

```
o).o.lower "SS"
"ss"
o).o.lower "É"
"é"
o)
```

This operation performs an unconditional mapping without tailoring. That is, the conversion is independent of context and language.

In the [Unicode Standard](#), Chapter 4 (Character Properties) discusses case mapping in general and Chapter 3 (Conformance) discusses the default algorithm for case conversion.

 [.o.upper](#)

Monadic .o.upper

Returns uppercased string or symbol.

Syntax: `.o.upper <x>`; `.o.upper [<x>]`

```
o).o.upper "a"  
"A"  
o).o.upper["Qwerty"]  
"QWERTY"  
o).o.upper `test  
`TEST
```

If some chars requires special considerations (e.g. multiple chars) they given by [SpecialCasing.txt](#)

```
o).o.upper "ß"  
"SS"  
o).o.upper "Русский военный корабль, иди нах*й!"  
"РУССКИЙ ВОЕННЫЙ КОРАБЛЬ, ИДИ НАХ*Й!"  
o)
```

This operation performs an unconditional mapping without tailoring. That is, the conversion is independent of context and language.

In the [Unicode Standard](#), Chapter 4 (Character Properties) discusses case mapping in general and Chapter 3 (Conformance) discusses the default algorithm for case conversion.

 [.o.lower](#)

Monadic argv

Returns a dictionary with the command line arguments.

Syntax: `argv[]`; `argv[`raw]`; `argv `raw`

```
o)argv[]
__bin__      | "./tachyon"
__scheds__   | "4"
__filenames__| "[repl]"
o)argv `raw
"./tachyon -c 4 -f repl"
o)
```

Monadic getenv

Reads environment variable value(s).

Syntax: `getenv <x>`; `getenv [<x>]`

```
o)getenv[`TERM]
"xterm-256color"
o)getenv[`TERM`SHELL]
"xterm-256color"
"/bin/bash"
o)
```

 [setenv](#)

 [set](#)

 [get](#)

Dyadic setenv

Set environment variables

Syntax: `<x> setenv <y>`; `setenv [<x>; <y>]`

```
o)setenv[`A;"AA"]
o)setenv[`B`C;("BB";"CC")]
o)getenv[`A`B`C]
"AA"
"BB"
"CC"
o)
```

 [getenv](#)

 [set](#)

 [get](#)

Monadic system

Executes a shell command.

Syntax: `system <x>`; `system [<x>]`

Returns a 3-element list: (code;stdout;stderr).

```
o)system "uname -s"  
0i  
"Linux\n"  
"  
o)
```

 os

Polyadic os

Executes same shell command.

Use `os []` to see list of available command.

Syntax: `os []`; `os <x>`; `os [<x>]`

```
o)os[]
"pwd"
"exe"
"cd"
"name"
"home"
"tmp"
"pid"
"dir"
o)os "name"
"linux"
o)
o)os["dir";"std"]
(2023.07.21D13:40:09.791455513;`file;12450;"std/core.o")
(2023.04.11D10:00:41.644733021;`file;5499;"std/http.o")
(2023.07.21D13:40:09.791913985;`file;9396;"std/core_test.o")
(2023.09.07D08:26:42.919121500;`file;16874;"std/sql.o")
(2021.10.11D14:10:41.730449203;`file;1017;"std/xml.o")
(2020.03.12D15:24:15.150369550;`file;506;"std/json.o")
(2021.10.11D14:10:41.728006589;`file;3673;"std/lit.o")
(2023.07.21D13:40:09.792253959;`file;10610;"std/repl.o")
(2023.07.21D13:40:09.793254928;`file;7544;"std/urllib.o")
(2023.04.11D10:00:41.645062312;`file;2491;"std/markdown.o")
(2023.04.11D10:00:41.644044351;`file;699;"std/htreq.o")
(2020.03.12D15:24:15.150907150;`file;4851;"std/prolog.o")
o)
```

 [system](#)

Monadic @ (type)

Returns internal type id of an argument.

Syntax: @<x>; @[<x>]



Avoid relying on these ids in production!

```
o)@1
320
o)@"s"
46400
o)@([]a:1;b:"a")
126988
o)d:`a`c!(1;2)
a| 1
c| 2
o)@
126992
o)@flip d
126988
o)
```

📖 @ (indexing)

📖 @ (triadic amend)

📖 @ (tetradic amend)

Monadic type

Returns type spec of its argument.

Syntax: `type <x>`; `type [<x>]`

```
o)type 1
`s`long
o)type 1.0
`s`float
o)type 1 2
`v`long
o)type "a"
`v`char
o)d:`a`b!(1;2)
a| 1
b| 2
o)type d
``dict
o)type flip d
``table
o)
```

See the full list of scalar types [here](#).

Monadic ! (id)

When applied to type spec returns corresponding internal type id.

Syntax: `!<x>`; `![<x>]`

```
o)!`s`int
256
o)!`v`int
45312
o)!`v`long
45376
o)!``dict
126992
o)
```

See the full list of scalar types [here](#).

 `.o.typedesc` (inverse function)

 `!` (til)

 `!` in dicts

Monadic .o.typedesc

Returns type spec by internal type id.

Syntax: `.o.typedesc <x>`; `.o.typedesc [<x>]`

```
o).o.typedesc 320
`s`long
o).o.typedesc[45312]
`v`int
o).o.typedesc[126992]
``dict
o).o.typedesc[@()]
`v`l
o)
```

 ! for type

Monadic ! (key)

Returns keys of `x` if it is a keyed value.

Syntax: `! <x>`; `! [<x>]`; `key <x>`; `key [<x>]`

It can be used to retrieve all names inside root or any namespace since environment is a dictionary itself.

```
o)a:42
42
o)key `
`a
o)key `.`
o | `cut`upper`lower`compress`decompress`typedesc!(.o.cut;.o.upper;.o.lower;.o.compress;.o.decompress;.o.typedesc)
repl | `opt`version`prompt`ps1`out`x`bt`ltrim`rtrim`trim`sig`peval`psend`enum2sym`fmt`rapi`inc`connect`takeAtMax`klen`keys..
o)! `a`s`d!(1 2;3 4;5 6)
`a`s`d
o)key +`a`s`d!(1 2;3 4;5 6)
`a`s`d
o)
```

Monadic meta

Retrieves meta-information about tables, [reagents](#) and tasks.

Syntax: `meta <x>`; `meta [<x>]`; `meta []`

```
o)meta ([a:`asc#1 2 3;b:1.1 2.0 5.3)
+`column`type`id`attr!(`a`b;`long`float;176448 45632;`asc`)
()
o)r:reagent[`async]
<Reagent#3>
o)meta[r]
id | 3
state | `running
type | "async"
o)
```

`meta []` returns meta-information about the current task. To get information about a specific task, pass the join handle as an argument.

```
o)meta []
taskid | 0
parent | <Reagent#4>
children | ()
schedid | 1
o)h:spawn {10000000{x+1}/1;}
<Reagent#6>
o)meta []
taskid | 0
parent | <Reagent#7>
children | (<Reagent#8>)
schedid | 1
o)
```

[reagent](#)

[table meta](#)

Monadic quote

Returns `x` unevaluated.

Syntax: `quote <x>`; `quote[<x>]`

```
o)quote 1+2
+
1
2
o)quote `a`b!(1;2)
!
`a`b
(,;1;2)
o)
```

Polyadic quote

Syntax: `quote ((<...>) (; ..))`; `quote[((<...>) (; ..))]`

```
o)quote (1+11;2+2;1<2)
,
(+;1;11)
(+;2;2)
(<;1;2)
o)
```

To evaluate the result of `quote`, use `eval`:

```
o)a:quote 1+2
+
1
2
o)eval a
3
o)b:quote `a`b!(1;2)
!
`a`b
(,;1;2)
o)eval b
a| 1
b| 2
o)
```

[eval](#)

[parse](#)

Monadic rc

Reference count of x

Syntax: `rc <x>`; `rc[<x>]`

```
o)a: 1 2 3
1 2 3
o)rc[a]
1
o)b:a
1 2 3
o)rc[a]
2
o)l:(a;b)
1 2 3
1 2 3
o)rc[a]
4
o)l:()
()
o)rc[a]
2
o)
```

 count

Monadic mv (move)

Moves or "consumes" local or global variable or upval (captured value in closures).

Syntax: `mv <x>`; `mv [<x>]`

By moving or consuming it means literally taking value out of its binding symbol. Moving value leaves generic null in its binding. Used for optimization mostly by helping to keep internal rc count 1. That allows to guarantee mutating values in-place greatly improving performance in some cases.

```
o) t: +`a`b!(!10000; !10000); // simulate large table
o) t:10#{ 0N!rc x; @[`x; `a; +; 1]; x } mv t
1 1 1
a b
----
1 0
2 1
3 2
4 3
5 4
6 5
7 6
8 7
9 8
10 9
```

Monadic ts

Returns current timestamp.

Syntax: `ts[]`; `ts[`utc]`; `ts[`local]`; `ts[`instant]`

`ts[`utc]` returns the current timestamp of the UTC time zone.

With the `local` or without argument - the timestamp for the current time zone.

With the `instant` argument, the result is the number of microseconds since the platform started.

```
o)ts[`utc]
2020.02.12D09:33:44.365124088
o)ts[`local]
2020.02.12D11:33:47.428896927
o)ts[]
2020.02.12D11:33:50.868108644
o)
```

 [Temporal Types](#)

Monadic . (dot-value)

Returns value of `x`.

Syntax: `value <x>;` `value[<x>];` `. <x>;` `. [<x>]`

```
o)value 1
1
o)value 1 2 3
1 2 3
```

```
o)value `
,`l`,`o`repl!(`cut`upper`lower`compress`decompress`typedesc!(.o.cut;.o.upper;.o.lower;.o.compress;.o.decompress;.o.typedesc)
o)value `
o | `cut`upper`lower`compress`decompress`typedesc!(.o.cut;.o.upper;.o.lower;.o.compress;.o.decompress;.o.typedesc)
repl! `opt`version`prompt`ps1`out`x`bt`ltrim`rtrim`trim`sig`peval`psend`enum2sym`fmt`rapi`inc`connect`takeAtMax`klen`keys..
o)
```

```
o)d:`a`s`d!(1 2;3 4;5 6)
a| 1 2
s| 3 4
d| 5 6
o)value d
1 2
3 4
5 6
o)t:+d;
o)value t
1 2
3 4
5 6
o)f:{x+1}
{x+1}
o)value f
,(+;#0;1)
o)
```



For lists, `value` works like `eval`, but `eval` "executes" lists recursively, and `value` "executes" only the first level. "Executed" means that the first item in the list is treated as a verb, and this verb is applied to the remaining items in the list.

```

o)l:(,;(4);(5))
,
(!;4)
(!;5)
o)eval l
{![4;]}
{![5;]}
o)// ! interpreted as a dyadic. So we got projections, not vectors.
o)value l
!
4
!
5
o). l
!
4
!
5
o)// : after ! indicates that monadic is used
o)l:(,;(4);(5))
,
((!);4)
((!);5)
o)eval l
0 1 2 3 0 1 2 3 4
o)value l
(!;)
4
(!;)
5
o)

```

 [eval](#)

 [. \(apply\)](#)

 [triadic dmend](#)

 [tetradic dmend](#)

Adverbs

Adverbs are high order constructions that modify the way a verb is applied to data.

Each monad

Applies monadic in the left argument to each element of the right structure.

```
o){x+1}'4 5 6
5 6 7
o)
```

Each dyad

Applies dyad to each pair of left and right arguments' elements or pair with one item.

```
o)1 2 3 +' 4 5 6
5 7 9
o)10+'1 2 3
11 12 13
o)
```

Each prior

Applies dyad to subsequent pairs of right argument structure.

```
o)+' :1 2 3
1 3 5
o)
```

... beware of edge cases though:

```
o){y}' : ,1
,0N
o){y}' :1 2
0N 1
o)
```

In lambdas, implicit an argument x - current item, y - prior item.

Each prior with initial value

Same as above, but the left argument defines initial value.

```
o)10*':1 2 2
10 2 4
o)
```

Each right

Applies dyad to each pair of the left argument and right argument elements.

```
o)10 100+/:1 2 3
11 101
12 102
13 103
o)
```

Each left

Applies dyad to each pair of the left argument elements and right argument.

```
o)10 100+\:1 2 3
11 12 13
101 102 103
o)
```

Over aka fold

Reduces right argument structure using left dyad. Initial value is different for some verbs.

```
o)+/1 2 3
6
o){0N!x,y; x+y}/10 3 2 1
10 3
13 2
15 1
16
o)
```

Boolean vectors behave as vectors of long for some verbs:

```
o)+/101b
2
o)
```

```
o) + / 0 # 0
0
o)
```

However:

```
o) * / 0 # 0
1
o) : / 0 # 0
0 N
o)
```

Over with initial value

Folds right argument structure using left dyad. Initial value is given explicitly:

```
o) 10 * / 1 2 2
40
o)
```

Over "converge"/"fixedpoint"

Repeatedly applies monadic function the left argument until result stops changing:

```
o) {x%2} / 100
0
o)
```



If `over` with a lambda is applied to a vector, and there is no implicit argument `y` in the lambda, then the entire vector is taken as one argument.

```
o) // !!! infinite loop !!!
o) {0N!x; x+1} / 1 2 3
2 3 4
3 4 5
...
```

In this line, x will change infinitely.

```
o){0N!x; (1+x*3)%4}/2 5 3 1
2 5 3 1
1 4 2 1
1 3 1 1
1 2 1 1
1 1 1 1
1 1 1 1
o)
```

Scan

Acts pretty much the same as "over", but returns a structure of the same size.

```
o)+\1 2 3
1 3 6
o)+\011b
0 1 2
o)
```

Scan with initial value

"Scan" but with explicitly given initial value.

```
o)10*\1 2 2
10 20 40
o)
```

Scan "converge"/"fixedpoint"

Repeatedly applies monadic function in the left argument to the right one, until result stops changing and accumulates all intermediate results:

```
o){x%2}\100
100 50 25 12 6 3 1 0
o)
```

If `scan` with a lambda is applied to a vector, and there is no implicit argument `y` in the lambda, then the entire vector is taken as one argument.



```
o) // !!! infinite loop !!!
o) {x+1}\1 2 3
2 3 4
3 4 5
...
```

In this line, x will change infinitely.

```
o){(1+x*3)%4}\2 5 3 1
2 5 3 1
1 4 2 1
1 3 1 1
1 2 1 1
1 1 1 1
1 1 1 1
o)
```

"For" loop

Applies monadic verb to the right argument left argument times.

```
o)5{x+1}/10
15
o)
```

"While" loop

Applies monadic verb to the right argument while the left argument returns true value.

```
o){x<100}{x*2}/1
128
o)
```

Scan "For" loop

Applies monadic verb to the right argument left argument times.

```
o)5{x+1}\10
10 11 12 13 14 15
o)
```

Scan "While" loop

Applies monadic verb to the right argument while the left argument returns true value.

```
o){x<100}{x*2}\1
1 2 4 8 16 32 64 128
o)
```

Commute

transforms dyadic to have its left and right argument swapped.

Amend/dmend also support commute dyads:

```
o)a:!5
0 1 2 3 4
o).[a;()~[-];-2]
0 1 2
o)
```

Relational queries are nice tool to have in a vector language. They form a special sub-language operating on tables somewhat similar to SQL. The term "record" referenced below means a row of fields taken from a table.



Queries support is currently limited to "select" polyad & "upsert" triad.

Select

In order to execute a query, one must first define a query expression, compile it into internal representation and apply "fetch" verb to get results. All three stages are done explicitly.

Overall syntax

Overall syntax for query stages looks like:

```
q: ?[ t; w; g; f; ot; os ]; // define query
cq: opt . q; // compile query
res: fe#cq; // fetch query results
```

where

`t` is a table-like data source. It might be a table/query/join/union expression.

`w` is a nested list of conditions. It might be an empty list to define no conditions.

`g` is a grouping dict or 0b expression for none.

`f` is a field dict.

`ot` is an optional "take" argument. It might be either a scalar or a two-element integer vector. It defines a range of record indices to limit results.

`os` is an optional "sorting" argument which must be a nested list of fields to sort with direction.

`opt` is an optional "compilation options" argument which must be a dict with specific fields. It overrides default query compile options (e.g. optimization level) locally. It is implemented since 0.6.0 version.

Compilation options

"Compilation options" dict field list consists of several possible values:

- `qO` (optimization level) and must have 0..3 integer values only.
- `qP` (enable parallel execution) and must have 0b/1b values.

Opt level	Meaning	Usage recommendations
0	All optimizations are disabled	For "debug" cases or for "perfect" hand-written queries which lack any possible improvement.
1	Quick & basic optimizations	For generated queries & large hand-written which might have some room for improvement.
2	General optimizations	For large generated queries over large datasets.
3	General & relatively slow optimizations	For large generated queries over large datasets. Query should perform as fast as possible.

Compilation option for parallel execution is ignored when the platform is started using single thread only (without `-c` option).



Parallel execution is guaranteed to provide the same results as "single" thread execution provided no side-effects (namespace mutation, chunk order/size expectations, IO - reagents,files, etc) are used in user query lambdas.

For dummy example see below.

```
o) t:(+:)`a`b`c`d!(1 2 3;3 4 5;6 7 8;("111";"222";"333"));
o) 0N#( `q0`qP!(3;1b)) . ?[t;();0b;`a`b`c`d!`a`b`c`d]
a b c d
-----
1 3 6 "111"
2 4 7 "222"
3 5 8 "333"
```

It is generally recommended to avoid overriding global optimization level besides extra-ordinary cases. It's better to use global / command-line option at start time.



"Compilation options" argument and qO,qP options are implemented since 0.6.0 version.

Compilation & fetching results

Let's have a look at a simple example first. It just creates a single table, a "q" query, compiles it into "cq" and fetches all table data into "res".

```
o)t:(+:)`a`b`c`d!(1 2 3;3 4 5;6 7 8;("111";"222";"333"));
o)q:[ t; (); 0b; `a`b`c`d!`a`b`c`d ];
o)cq:. q;
o)res:0N#cq;
o)res
a b c d
-----
1 3 6 "111"
2 4 7 "222"
3 5 8 "333"
o)
```



Note a space between "dot" monad and "q" variable: `cq:. q`

As you can see, the verb "fetch" is a dyad expecting either null to fetch all query results in one go, or an amount of records if positive number.

Fields/renaming

Fields defined in fourth argument are a dictionary where keys are expected as a symbol vector. Keys define query field names. Dictionary values define fields as symbols or expressions from data source (first argument). To rename query field, just change dictionary keys as appropriate.

```
o)t:(+:)`a`b`c`d!(1 2 3;3 4 5;6 7 8;("111";"222";"333"));
o)q:[ t; (); 0b; `ra`rb`rc`rd!`a`b`c`d ];
o)cq:. q;
o)res:0N#cq;
o)res
ra rb rc rd
-----
1 3 6 "111"
2 4 7 "222"
3 5 8 "333"
o)
```

Table aliases

While field references are unique symbols (e.g. from a single source table), underlying query engine is fine with scalar symbols as fields. However, queries might become too complex if different tables have the same field names. To disambiguate between tables, table aliases are used.

Defining table alias is done via providing a two-element list as a data source instead of a single table. The first element is a symbol defining alias symbol, the second one is a table, join or another data source to bind.

To reference a field via alias, use monadic `~` with a two-element symbol vector:

```
o)t:(+:)`a`b`c`d!(1 2 3;3 4 5;6 7 8;("111";"222";"333"));
o)q:[ ( `alias;t); (); 0b; `ra`rb`rc`rd!((~`alias`a);(~`alias`b);(~`alias`c);(~`alias`d)) ];
o)cq:. q;
o)res:0N#cq;
o)res
ra rb rc rd
-----
1 3 6 "111"
2 4 7 "222"
3 5 8 "333"
o)
```

Field expressions

Field definition in queries can also contain various expressions calculated in-flight.

Expressions are defined using functional/"lisp" syntax:

```
o)t:(+:)`a`b`c!(1 2 3;3 4 5;6 7 8);
o)0N#.(?[t;();0b;`d`e!((-:;`a);(+;`b;`c))])
d e
-----
-1 9
-2 11
-3 13
o)
```

Virtual field "i"

To identify table rows, use virtual field ``i`. It's merely an automatically generated zero-based index.

```
o)t:(+:)`a`b`c!(1 2 3;3 4 5;6 7 8);
o)0N#.(?[t;();0b;`a`b`c`i!`a`b`c`i])
a b c i
-----
1 3 6 0
2 4 7 1
3 5 8 2
o)
```

Constants

Constants can be used in place of field, conditions, etc.

```
o)t:(+:)`a`b`c!(1 2 3;3 4 5;6 7 8);
o)0N#.(?[t;()0b;`a`b`c`const!(`a;`b;`c;10)])
a b c const
-----
1 3 6 10
2 4 7 10
3 5 8 10
o)
```

To distinguish between symbol constants and fields, symbol constants must be enlisted.

```
o)t:(+:)`a`b`c!(1 2 3;3 4 5;6 7 8);
o)0N#.(?[t;()0b;`a`b`c`const!(`a;`b;`c;;`d)])
a b c const
-----
1 3 6 d
2 4 7 d
3 5 8 d
o)
```

 [Tables](#)

 [next >>> WHERE](#)

Conditions

Using conditions in queries means leaving only those records that satisfy conditions. If a corresponding field contains an attribute, it will be used to speed-up the query execution.

Here is an example with a single condition filter:

```
o)t:(+:)`a`b`c`d!(1 2 3;3 4 5;6 7 8;("111";"222";"333"));
o)c:,(<=;`b;3);
o)0N#.(?[t;c;0b;`a`b`c!`a`b`c])
a b c
-----
1 3 6
o)
```



List enclosure in the "w" definition of select is mandatory. The condition must be a proper list!

Conditions may contain field expressions:

```
o)t:(+:)`a`b`c!(1 2 3;3 4 5;6 7 8);
o)c:,(<=;(+:`b;`c);9);
o)0N#.(?[t;c;0b;`a`b`c!`a`b`c])
a b c
-----
1 3 6
o)
```



For now, constants in dyadic conditions must be the second argument.

And/or/not clauses, simplified syntax

Of course, single field conditions are not enough, several conditions are trivially created using `&` and `|` dyads in condition list:

```
o)sym:`symbol$!10;
o)t:(+:)`a`b`c`d!(1 2 3;`g#`3`4`5;6 7 8;`g#`sym$`3`4`5);
o)c:,(&;(>=;`d;`sym$`3);(<=;`b;`3));
o)0N#.(?[t;c;0b;`a`b`c!`a`b`c])
a b c
-----
1 3 6
o)
```

"not" clause is a bit awkward, as the ordinary `~` monad does not work here. Using `~:` as monadic "not" is not possible because it results in "commute" instead. You need to use a special verb `not`:

```
o)t:(+:)`a`b`c!(!10;0+!10;`symbol$!10);
o)0N#.(?[t;,(not;(=;`a;6));0b;`a`b`c!`a`b`c])
a b c
-----
0 0 0
1 1 1
2 2 2
3 3 3
4 4 4
5 5 5
7 7 7
8 8 8
9 9 9
o)
```

Quite common pattern is joining all field conditions with "and" logic, thus a simplified syntax is also supported:

```
o)t:(+:)`a`b`c!(1 2 3;3 4 5;6 7 8);
o)c:((<=;`b;4);(>=;`c;7));
o)0N#.(?[t;c;0b;`a`b`c!`a`b`c])
a b c
-----
2 4 7
o)
```

Where "subselect"

Another commonly used pattern is a "sub-select". It applies another (sub-select) query to condition as an argument. It can be used as seen below:

```
o)t1:(+:)`a`b`c!(1 2 3;3 4 5;6 7 8);
o)t2:(+:)`b`c!(,2;,4);
o)c:(=;`a;?[t2;C];0b;(,`b)!(,`b));
o)0N#.(?[t1; c; 0b; `a`c!`a`c])
a c
---
2 7
o)
```

Pay attention that just a single scalar value is expected as a result of "t2" query, otherwise an error occurs:

```
o)t1: (+:)`a`b`c!(1 2 3;3 4 5;6 7 8);
o)t2:(+:)`b`c!(!2;!2);
o)c: ,(=;`a;?[t2;C];0b;(,`b)!(,`b));
o)0N#.(?[t1; c; 0b; `a`c!`a`c])
** runtime error: `select`:
scalar expected
o)
```

Where "in"

Another predicate accepting sub-selects is `in`. It checks if the left arg is within the right argument set of values (either constant vector or a query).

```
o)t1:(+:)`a`b`c!(1 2 3;3 4 5;6 7 8);
o)t2:(+:)`b`c!(1 2;3 4);
o)c:,(in;`a;?[t2;C];0b;(,`b)!(,`b));
o)0N#.(?[t1; c; 0b; `a`c!`a`c])
a c
---
1 6
2 7
o)
```

Where "like"

"Like" predicate matches against a regular expression. Both strings and symbols are supported.

```
o)t:((+:)`a`d`e`f!(1 2 3;3 4 5;6 7 8;("111";"222";"333")));
o)c:,(like;`f;"111|222");
o)0N#.(?[t; c; 0b; `a`f!`a`f])
a f
-----
1 "111"
2 "222"
o)
```

Where "match"

"Match" checks for exact equality.

```
o)t:((+:)`a`d`e`f!(1 2 3;3 4 5;6 7 8;("111";"222";"333")));
o)c:,(~;`f;"111");
o)0N#.(?[t; c; 0b; `a`f!`a`f])
a f
-----
1 "111"
o)
```

Where with generic lambda

Any lambda monad/dyad can be used in place of predicate. However, it must satisfy the following requirements.

For monads:

- support vectors as arguments;
- return resulting boolean vector of the argument's length.

```
o)t:(+:)`a`b`c!(1 2 3;0 0N 5;6 7 8);
o)0N#.([t;,(null;`b);0b;( `a`b)!( `a`b)])
a b
----
2 0N
o)
```

For dyads:

- support vectors in the left argument;
- support either vectors or scalars in the right argument (depending on the query predicate third parameter);
- return resulting boolean vector of the left argument's length.

```
o) t:(+:)`a`b`c!(1 2 3;0 1 5;6 7 8);
o) 0N#.([t;,{x>y};`a;`b);0b;( `a`b)!( `a`b)])
a b
---
1 0
2 1
o)
```

Select from select

Selecting from inner select is also supported.

```
o)t1:((+:)`a`b`c`d!(1 2 3;3 4 5;6 7 8;8 9 10));
o)t2: ?[( `t1;t1);();0b; `a`b`c! `a`b`c];
o)c: ,( >=; `a;2);
o)0N#.([t2; c; 0b; `a`c! `a`c])
a c
---
2 7
3 8
o)
```

 <<< prev SELECT

 next >>> ORDER

Distinct

If the grouping field is 1b, you get a distinct/unique set of records. See an example:

```
o)t:((+:)`a`b`c!(1 2 2 3;3 4 4 5;6 5 5 9));
o)0N#.(?[t;(); 1b; ())
a b c
-----
1 3 6
2 4 5
3 5 9
o)
```

 <<< prev GROUP

 next >>> LIMIT

Grouping/aggregation

Grouping fields is done in the third argument. It should be a dict with keys representing resulting field names and values representing source fields. A field dict in the fourth argument plays a different role when grouping dict is present. It defines grouping expressions to apply.

E.g.:

```
o)t:(+:)`a`b`c`d!(1 2 2;3 4 4;6 7 7;8 9 10));
o)gf: `a`b`c!`a`b`c; // grouping fields
o)ge: (,`d)!(,(sum;`d)); //grouping expressions
o)0N#.(?[t;(); gf; ge])
a b c d
-----
1 3 6 8
2 4 7 19
o)
```

Expressions for group dict values are supported. Conventional functional syntax is used:

```
o)t:(+:)`a`b`c`d!(1 2 2;3 4 4;6 7 7;8 9 10);
o)gf: `a`b!`a`b; // grouping fields
o)ge: (,`e)!(,(sum;(+:`c;`d))); //grouping expressions
o)0N#.(?[t;(); gf; ge])
a b e
-----
1 3 14
2 4 33
o)
```

As for grouping expressions, the following list of monads is currently supported and optimized for performance:

Exp	Meaning	Description
#:	count	number of non-null values in group
*:	first	first non-null value in group
last	last	last non-null value in group
min	minimum	minimum non-null value in group
max	maximum	maximum non-null value in group
sum	sum	sum of non-null values in group
avg	arithmetic average	average of non-null values in group

User-defined monads are accepted too but they may lead to slower performance as all intermediate group vectors must be preserved during query processing and a more general approach is used:

```
o)t:((+:)`a`b`c`d!(1 2 3;3 4 5;6 7 8;8 9 10));
o)gf:{+/x};
o)0N#.(?[t;C);`a`b`c!`a`b`c;(,`d)!(C,(gf;`d))])
a b c d
-----
1 3 6 8
2 4 7 9
3 5 8 10
o)
```



See "Grouping using incremental lambdas" to speed-up custom user-defined aggregations.

Generic aggregation only (without explicit grouping fields) is done like this:

```
o)t1:((+:)`a`b`c!(1 2 3 2;3 4 5 6;6 7 8 8));
o)0N#.(?[C` t1;t1);C);C);`d`e!(({/x};~`t1`c);({+/x};~`t1`b))])
d e
-----
29 18
o)
```

Pay attention to an empty list in the third argument that enables aggregation only.

Aggregation only (without explicit grouping fields) using special monads is done like:

```
o)t:((+:)`a`b`c`d!(1 2 3;3 4 5;6 7 8;8 9 10));
o)0N#.(?[C` t;t);C);0b;`a`b`c`d!((sum;~`t`a);(avg;~`t`b);(max;~`t`c);(min;~`t`d))])
a b c d
-----
6 4 8 8
o)
```

Grouping using incremental lambdas

As it has been noted before, grouping using custom/user monads is supported, but it can result in excessive memory usage. It happens due to necessity of accumulating all intermediate data in grouping buckets until all data is collected. Another way of grouping is supported specifically to make it work incrementally. Incremental grouping does its job by splitting entire data in chunks and processing each data chunk as comes.

Essentially, incremental grouping requires three user-defined lambdas. First lambda - to initialize grouping state, preparing data structures. Second - doing incremental aggregation itself. And third - to finalize grouping on completion.

Key	Meaning	Input parameters	Expected result
<code>`init</code>	Initialize state	<code>[cg; data]</code> , where <code>`cg`</code> - number of groups found so far, <code>`data`</code> - current data chunk	Any shaped initialized aggregation state
<code>`aggr</code>	Incremental aggregation	<code>[s; cg; ix; data]</code> , where <code>`s`</code> - current aggregation state, <code>`cg`</code> - number of groups found so far, <code>`ix`</code> - vector of group indices for each element in <code>`data`</code> , <code>`data`</code> - current data chunk	Aggregation state
<code>`fin</code>	Finalize aggregation state	<code>[s]</code> , where <code>`s`</code> - current aggregation state	Finalized aggregation state



Make sure ``aggr`` lambda resizes state data for coming data & handle nulls as needed.

And the simple example of aggregation via summing:

```
o) t:+`a`b!(1 2 2;3 4 0N);
o) d:`init`aggr`fin!({[cg;data] cg#0}; {[s;cg;ix;data] .[`s;();,(cg-#s)#0]; .[`data;();~[^\];0]; @[`s;ix;+;data]; s};{[s] s
o) 0N#(.[t;();`a!`a;`b!(d;`b)])
a b
---
1 3
2 4
```

[<<< prev ORDER](#)

[next >>> DISTINCT](#)

Take range (first, last, between)

Take range is given as the fifth argument for select to limit records fetch by the record index.

To grab the first N records from query results, use a positive scalar:

```
o) t:(+:)`a`b`c!(1 2 3;3 4 5;6 7 8);
o) 0N#.([t; (); 0b; `a`c!`a`c; 2])
a c
---
1 6
2 7
o)
```

To grab the last N records from query results, use a negative scalar:

```
o) t:(+:)`a`b`c!(1 2 3;3 4 5;6 7 8);
o) 0N#.([t; (); 0b; `a`c!`a`c; -2])
a c
---
2 7
3 8
o)
```

And to grab records within some range, use a two-element vector (from;length):

```
o)t:(+:)`a`b`c!(1 2 3;3 4 5;6 7 8);
o)0N#.([t; (); 0b; `a`c!`a`c; 1 2])
a c
---
2 7
3 8
o)
```

To grab all records, use a generic null `0N0` or an empty list or a cardinal vector.

```
o)t:(+:)`a`b`c!(1 2 3;3 4 5;6 7 8);
o)0N#.([t; (); 0b; `a`c!`a`c; 0N0])
a c
---
1 6
2 7
3 8
o)0N#.([t; (); 0b; `a`c!`a`c; ()])
a c
---
1 6
2 7
3 8
o)
```


 <<< prev DISTINCT

 next >>> JOIN

Sorting

Sorting query results is done with the sixth argument in a select. The argument must contain a nested list of fields to sort with direction:

```
o)t:((+:)`a`b`c!(1 2 3;3 4 5;6 7 8));
o)0N#.(?[t;O;0b;`a`c!`a`c;O;((>;`a);(<;`b))])
a c
---
3 8
2 7
1 6
o)
```

Here the monadic  means ascending order,  - the descending one.

 <<< prev WHERE

 next >>> GROUP

Joins

Joining a pair of tables (or table-likes) is a base operation in queries. Both joining on a single field and on several fields are supported. Usually, table aliases are required to disambiguate fields.

Inner join is defined using the verb `ij`:

```
o)t1:((+:)`a`b`c!(1 2 3;3 4 5;6 7 8));
o)t2:((+:)`a`d`e`f!(1 2 3;3 4 5;6 7 8;("111";"222";"333")));
o)j: ij[(`t1;t1);(`t2;t2);(~`t1`a;~`t2`a)];
o)0N#.(?[j; () ; 0b; `a`b`e`f!(~`t1`a;~`t1`b;~`t2`e;~`t2`f)])
a b e f
-----
1 3 6 "111"
2 4 7 "222"
3 5 8 "333"
o)
```

Inner join is different from the one in SQL - it does not produce Cartesian product for duplicating ids, it emits the first found for second relation.

```
o)t1:((+:)`a`b`c!(`g#3 2 0;3 4 5;6 7 8));
o)t2:((+:)`a`d`e`f!(`g#2 2 5;13 14 15;16 17 18;("111";"222";"333")));
o)j: ij[(`t1;t1);(`t2;t2);(~`t1`a;~`t2`a)];
o)0N#.(?[j; () ; 0b; `a`b`e`f!(~`t1`a;~`t1`b;~`t2`e;~`t2`f)])
a b e f
-----
2 4 16 "111"
o)
```

It's strongly recommended to attach attributes to a field before the join. If there is no appropriate attribute, query engine will build a temporary one every time the query is executed.

Multi-column joins are also supported. The same logic of building indices beforehand/in-time is applied here.

```
o)t1:((+:)`a`b`c!(3 1 2;3 4 5;6 7 8));
o)t2:((+:)`a`d`e`f!(1 2 3;4 5 3;16 17 18;("111";"222";"333")));
o)@[`t1; ,`a`b;~[#];`g];
o)@[`t2; ,`a`d;~[#];`g];
o)j: ij[(`t1;t1);(`t2;t2);((~`t1`a;~`t1`b);(~`t2`a;~`t2`d))];
o)0N#.(?[j; () ; 0b; `a`b`e`f!(~`t1`a;~`t1`b;~`t2`e;~`t2`f)])
a b e f
-----
1 4 16 "111"
2 5 17 "222"
3 3 18 "333"
o)
```

Left join is defined using the verb `lj`:

```
o)t1:((+:)`a`b`c!(3 1 2;3 4 5;6 7 8));
o)t2:((+:)`a`d`e`f!(`g#1 2;13 14;16 17;("111";"222")));
o)j:l_j[(`t1;t1);(`t2;t2);(~`t1`a;~`t2`a)];
o)0N#.(?[j; (); 0b; `a`b`e`f!(~`t1`a;~`t1`b;~`t2`e;~`t2`f)])
a b e f
-----
3 3 0N 0N0
1 4 16 "111"
2 5 17 "222"
o)
```

Left join differs from the one in SQL as well - it does not produce Cartesian product for duplicating ids, it emits first found for second relation.

```
o)t1:((+:)`a`b`c!(3 2 0;3 4 5;6 7 8));
o)t2:((+:)`a`d`e`f!(`g#2 2 5;13 14 15;16 17 18;("111";"222";"333")));
o)j:l_j[(`t1;t1);(`t2;t2);(~`t1`a;~`t2`a)];
o)0N#.(?[j; (); 0b; `a`b`e`f!(~`t1`a;~`t1`b;~`t2`e;~`t2`f)])
a b e f
-----
3 3 0N 0N0
2 4 16 "111"
0 5 0N 0N0
o)
```

Just like with [i_j](#), it's strongly recommended to attach attributes to a field before the join. If there is no appropriate attribute, query engine will build a temporary one every time the query is executed.

Multi-column left joins are also supported. The same logic of building indices beforehand/in-time is applied here.

```
o)t1:((+:)`a`b`c!(3 2 0;3 4 5;6 7 8));
o)t2:((+:)`a`b`e`f!(3 2 3;3 4 3;16 17 18;("111";"222";"333")));
o)@[ `t2; , `a`b;~[#]; `g];
o)j:l_j[(`t1;t1);(`t2;t2);((~`t1`a;~`t1`b);(~`t2`a;~`t2`b))];
o)0N#.(?[j; (); 0b; `a`b`e`f!(~`t1`a;~`t1`b;~`t2`e;~`t2`f)])
a b e f
-----
3 3 16 "111"
2 4 17 "222"
0 5 0N 0N0
o)
```

[<<< prev LIMIT](#)

[next >>> UNION](#)

Unions

Union of a pair of tables (or table-likes) is another base operation in queries. Conceptually, it's just lazy "concatating" table-likes to get a combined view.

"Union all" union is defined using the verb `ua`:

```
o) t1:+`a`b`c!(1 2;3 4;5 6);
o) t2:+`a`b`c!(10 20;30 40;50 60);
o) u:ua[t1;t2];
o) 0N#.(?[u; () ; 0b; `a`b`c!`a`b`c])
a b c
-----
1 3 5
2 4 6
10 30 50
20 40 60
```

Of course, unioning several table-likes is supported as well:

```
o) t1:+`a`b`c!(1 2;3 4;5 6);
o) t2:+`a`b`c!(10 20;30 40;50 60);
o) t3:+`a`b`c!(100 200;300 400;500 600);
o) s3:?[t3;() ;0b; `a`b`c!`a`b`c];
o) u:ua/(t1;t2;s3);
o) 0N#.(?[u; () ; 0b; `a`b`c!`a`b`c])
a b c
-----
1 3 5
2 4 6
10 30 50
20 40 60
100 300 500
200 400 600
```

Perhaps unsurprisingly, any other expression accepting table-like works for unions as well. e.g. joining (don't forget aliases):

```
o) t1:+`a`b`c!(1 2 3;4 5 6;7 8 9);
o) t2:+`a`b`c!(10 20;30 40;50 60);
o) t3:+`a`b`c!(1 10 20;100 300 400;200 500 600);
o) u1:ua[t1;t2];
o) j:ij[(~`u1;u1);(~`t3;t3);(~`u1`a;~`t3`a)];
o) 0N#.(?[j; () ; 0b; `a`b`c!(~`u1`a;~`u1`b;~`t3`c)])
a b c
-----
1 4 200
10 30 500
20 40 600
```

Upsert

Triad acts similarly to SQL "UPDATE OR INSERT ..." expression. It also supports creating new fields in addition to updating and inserting existing fields.

First argument is expected to be either a table or a symbol of a table for in-place modification.

Second argument is expected to be a table.

Third argument must be a two-element list of key symbol vectors. Key field vectors must match in length. The first field vector corresponds to the first table, the second one - to the second table.



It is recommended to have (composite) an index for the keys of the left table. It will speed-up processing.

Let's have a look at some examples:

```
o)a:(+:)`a`b`c`d!(!10;!10;!10;!10);
o)b:(+:)`a`b`c!(8+!5;8+!5;10+!5);
o)upsert[a;b;(`a`b;`a`b)]
a b c d
-----
0 0 0 0
1 1 1 1
2 2 2 2
3 3 3 3
4 4 4 4
5 5 5 5
6 6 6 6
7 7 7 7
8 8 10 8
9 9 11 9
10 10 12 0N
11 11 13 0N
12 12 14 0N
o)
```

And in-place modification example:

```

o)a:(+:)`a`b`c`d!(!10;!10;!10;!10);
o)b:(+:)`a`b`c!(8+!5;8+!5;10+!5);
o)upsert[`a;b;( `a`b; `a`b)]
 `a
o)a
a b c d
-----
0 0 0 0
1 1 1 1
2 2 2 2
3 3 3 3
4 4 4 4
5 5 5 5
6 6 6 6
7 7 7 7
8 8 10 8
9 9 11 9
10 10 12 0N
11 11 13 0N
12 12 14 0N
o)

```

See creating new fields:

```

o)a:(+:)`a`b`c!(!10;!10;!10); b:(+:)`a`b`c`d`e!(2+!5;2+!5;10+!5;20+!5;5#,"123"); upsert[a;b;( `a`b; `a`b)]
a b c d e
-----
0 0 0 0N 0N0
1 1 1 0N 0N0
2 2 10 20 "123"
3 3 11 21 "123"
4 4 12 22 "123"
5 5 13 23 "123"
6 6 14 24 "123"
7 7 7 0N 0N0
8 8 8 0N 0N0
9 9 9 0N 0N0
o)

```

 <<< prev UNION

 next >>> UPDATE

Update

For updating tables in memory, see [Tables modification](#) and [Table inserts](#).

For on-disk table modification, see [Tables on disk](#) and [Projectiong files concept](#).

 <<< prev UPSERT

 next >>> DELETE

Delete

Currently, there is no special verb `DELETE` for deleting information in tables. This can be done in different ways. Let's consider examples.

Delete columns in an in-memory table:

```
o)t: +`a`b`c`d!(!10; 10+!10; 100+!10; 1000+!10);
o)nt: `c _ t
a b d
-----
0 10 1000
1 11 1001
2 12 1002
3 13 1003
4 14 1004
5 15 1005
6 16 1006
7 17 1007
8 18 1008
9 19 1009
o)nt: +`a`d!t`a`d
a d
-----
0 1000
1 1001
2 1002
3 1003
4 1004
5 1005
6 1006
7 1007
8 1008
9 1009
o)
```

```
o)load "sql";
o)nt:select a, c from t
a c
-----
0 100
1 101
2 102
3 103
4 104
5 105
6 106
7 107
8 108
9 109
o)
```

Delete columns in an on-disk table:

```
o)t: +'a`b`c`d!(!10; 10+!10; 100+!10; 1000+!10); f:`:/tmp/t/; f set t;
o)`:/tmp/t/.d set `a`d
`:/tmp/t/.d
o)load `:/tmp/t/;
o)t
a d
-----
0 1000
1 1001
2 1002
3 1003
4 1004
5 1005
6 1006
7 1007
8 1008
9 1009
```



We draw your attention to the fact that deletion from the table is a relatively slow operation. It should not happen often, as it is the same `select` from the whole table.

Delete rows in an in-memory table:

```
o)t: +`a`b!(!10; 10+!10);
o)nt: 4 # t
a b
----
0 10
1 11
2 12
3 13
o)nt: -4 # t
a b
----
6 16
7 17
8 18
9 19
o)nt: (t[`a] in 3 5) # t
a b
----
3 13
5 15
o)nt: (~t[`a] in 3 5) # t
a b
----
0 10
1 11
2 12
4 14
6 16
7 17
8 18
9 19
o)nt: 5 _ t
a b
----
5 15
6 16
7 17
8 18
9 19
o)nt: -5 _ t
a b
----
0 10
1 11
2 12
3 13
4 14
```

```

o)t: +`a`b!(!10; 10+!10);
o)remove_by_i: {i:(#y)#1b; i[x]:0b; i#y};
o)nt: remove_by_i[1 3 5 7; t]
a b
----
0 10
2 12
4 14
6 16
8 18
9 19

```

```

o)load "sql";
o)t: +`a`b!(!10; 10+!10);
o)nt: select [4] from t
a b
----
0 10
1 11
2 12
3 13
o)nt: select [-4] from t
a b
----
6 16
7 17
8 18
9 19
o)nt: select [4 3] from t
a b
4 14
5 15
6 16
o)select from t where a>4, a<7
a b
----
5 15
6 16
o)// using virtual column i
o)select from t where i>4, i<7
a b
----
5 15
6 16
o)

```

Delete rows in an on-disk table:

Don't do that. Seriously. Just select records you want to retain and replace original table.

[⏪ <<< prev UPDATE](#)

[⏩ next >>> SQL-like syntax](#)

SQL-like syntax

After loading the `sql` script, you can use SQL-like syntax.

Syntax: `select (distinct) [[<range-sort>]] (<cols>|<expr>) from <table>|<join chain> (where <filters>) (by <groups>)`

In more detail about the SQL-like syntax in the description of the [sql.o](#) library.

```
o)\l "sql";
o)t:(+:)`a`b`c`d!(!10;!10;!10;!10);
o)select [8] M:min a,max b,last d from t where c<100, c>1 by c
c M f1 f2
-----
2 2 2 2
3 3 3 3
4 4 4 4
5 5 5 5
6 6 6 6
7 7 7 7
8 8 8 8
9 9 9 9
o)
```

[<<< prev DELETE](#)

[next >>> Ptable support](#)

[sql.o](#)

[Typographic Conventions](#)

Ptables support

Partitioned tables are supported transparently in queries. However, due to their special nature some operations are not supported.

Namely:

- Inner joins
- Left joins where ptable is right table.

 [<<< prev SQL-like syntax](#)

 [Partitioned tables](#)

 [core.o](#)

Dynamic parsers

Text parsers can be dynamically created as well as binary parsers for user-defined grammars using PEG parser combinator right out-of-the-box. See [Parsing Expression Grammar](#).

For example, let's create a simple parser that eats only strings consisting of "p" chars:

```
o)p:<- "p"+
<Parser["p"+]>
o)p "ppppp"
"ppppp"
o)p "pppppa"
** runtime error: `peg`:
Input: <a>
o)
```

Main syntax is `<-` followed by a parsing rule. A parser, like any other type in O language, can be assigned to a variable (symbol) and used later. Using parser with a string or a byte array just calls it like a function.

Parsing expression rules

- An atomic parsing expression consists of:
 - any terminal symbol,
 - any nonterminal symbol, or
 - the empty string ϵ .
- Given any existing parsing expressions e , e_1 , and e_2 , a new parsing expression can be constructed using such operators:
 - Sequence: $e_1 e_2$
 - Ordered choice: $e_1 | e_2$
 - Zero-or-more: e^*
 - One-or-more: e^+
 - Optional: $e?$
 - And-predicate: $\&e$
 - Not-predicate: $!e$

Syntax reference

Term	Description
"text"	String literal. Escapes are supported as well.
[abctrn]	One of symbols.
[a-z]	Range.
[^ab0-9]	Not one of/range.
"a" "b" [f-k]	Sequence of terms.
thing	Reference to another parser bound to symbol <code>`thing`</code>
thing?	An optional expression. This is greedy, always consuming thing if it exists.
&thing	A lookahead assertion. Ensures thing matches at the current position but does not consume it.
!thing	A negative lookahead assertion. Matches if thing isn't found here. Doesn't consume any text.
thing*	Zero or more thing. This is greedy, always consuming as many repetitions as it can.
thing+	One or more thing. This is greedy, always consuming as many repetitions as it can.
thing{2,3}	Minimum 2 or maximum 3 times thing. This is greedy, always consuming as many repetitions as it can.
\x01	Binary parser matches byte 0x01.
thing#	Drops matched input, parsed by thing.
thing/{"I"\$x}	Maps result of parser <code>`thing`</code> to lambda followed by <code>**/**</code> .
thing1 thing2	Tries thing1 then thing2 if first was not successful.
thing->nm	Maps result of thing to dict with key <code>`nm`</code> .
\d	Matches any digit.
\w	Matches any alphabetical symbol.
\W	Matches any digit or alphabetial symbol.
\s	Matches SPACE
\S	Matches any of: SPACE, TAB, CARRIAGE_RETURN, LINE_FEED
\.	Matches any symbol.
\\$	End of input.
@(..)	Any expression returns parser
(thing)	Grouping.

FIX parsing example

```
o)string: <- [^\\t\\r\\n]+;
o)header: <- "8=# string "|"#"9=# \\d+ "|"# /{`fixver`size!(x 0;"I"$x 1)};
o)header "8=FIX.4.4|9=126|"
fixver| "FIX.4.4"
size | 126i
o)
```

Language idioms

Language idiom is special combinations of verbs, which has better performance and memory optimizations. Idioms are recognized and optimized in lambda bodies only at parse time.

"Type id" idiom

Syntax: `! [symvec]`

where `symvec` is a constant symbol vector defining type. Idiom is calculated at parse time and replaced with actual type id as long integer. It's recommended to use as the fastest way of checking type for strict equality.

```
o) { (!`s`long)=@x }1
1b
```

"Find from" idiom

Quite often it's necessary to find index of some value in list starting with some position from the left.

Syntax: `pos+(pos _ vec)?val`

where `pos` is a position to start search from, `vec` - vector to look in, `val` - value to look for.

```
o) {[vec;pos;val] pos+(pos _ vec)?val }[2 1 2 3; 1; 2]
2
```

"Find to" idiom

Similar to "find from" idiom, this idiom searches until some position in vector.

Syntax: `(pos#vec)?val`

where `pos` is a position to stop search, `vec` - vector to look in, `val` - value to look for.

```
o) {[vec;pos;val] (pos#vec)?val }[1 2 2 3; 2; 3]
0N
```

"Where equal" idiom

Optimized search of indices in `g` indexed vector. Implementation uses optimized index search if `vec` vector is long enough.

Syntax: `&vec=val` or `&val=vec`

where `vec` - vector to look in, `val` - value to look for.

Standard library

The standard library scripts are located in the `std/` folder. Using these scripts makes writing code easier.

Current set of scripts

File name	Description
<code>core.o</code>	General functions and specific functions for working with data
<code>htreq.o</code>	Functions for query platform over http
<code>http.o</code>	Functions for creating an HTTP server
<code>json.o</code>	Functions for presenting data in JSON format
<code>lit.o</code>	Functions and parsers for processing self-documenting code
<code>markdown.o</code>	Functions and parsers for processing MD data
<code>prolog.o</code>	Functions and parsers to implement the Prolog language
<code>repl.o</code>	Functions that implement the Read-Eval-Print Loop
<code>sql.o</code>	Functions and parsers that implement the SQL-like syntax
<code>urllib.o</code>	Functions and parsers for processing URL strings
<code>xml.o</code>	Functions and parsers for processing XML data

Before using a particular script, you need to load it into the namespace.

```
o)load "core"  
"./std/core.o"
```

To not download the script again, you can check the presence of any of its elements in the namespace.

```
o)if [not `urllib in !(.`)] {load "urllib"};  
o)if [not `repl in !(.`)] {load "repl"};
```

 [load](#)

Additional general functions

Name	Description / comments
<code>assert_or[<cond>;<err or fn>]</code>	Assert <code>cond~1b</code> , otherwise throw signal with <code>err</code> string or call <code>fn[]</code>
<code>assert[<cond>;<err>]</code>	Assert <code>cond~1b</code> , otherwise throw signal with message "--- Assert [err] Failed"
<code>assert_eq[<expected>;<got>;<err>]</code>	Assert <code>lhs~rhs</code> , otherwise throw signal with <code>err</code> message and differences
<code>test[<name>;<expected>;<got>]</code>	Function for generating tests
<code>fc[<vec>;<size>]</code>	Fill or cut
<code>fmt[<string with %>; <list>]</code>	Dynamic formatting
<code>ltrim[<string>]</code>	Trim leading spaces
<code>rtrim[<string>]</code>	Trim trailing spaces
<code>trim[<string>]</code>	Trim leading and trailing spaces
<code>rcsv[<cols>;<sep>;<types>;<file>]</code>	CSV file loading
<code>wcsv[<tbl>;<cols>;<sep>;<file>]</code>	CSV file write
<code>peach[<fn>;<vec>]</code>	Call <code>fn</code> for each from <code>vec</code> in parallel
<code>xpeach[<fn>;<vec>]</code>	Parallel each splits <code>vec</code> onto <code>__cores__ - 1</code> pieces
<code>con[&ldict1>;<dict2>]</code>	Concatenates two dicts, duplicate keys concatenates their values
<code>differ[<list>]</code>	Returns a boolean list indicating where consecutive pairs of items in <code>x</code> differ
<code>lshift[<number>;<cnt>]</code>	Left shift number by <code>cnt</code> bits
<code>rshift[<number>;<cnt>]</code>	Right shift number by <code>cnt</code> bits
<code>lrot[<number>;<cnt>]</code>	Left rotate number by <code>cnt</code> bits
<code>.o.cell[<globvar>]</code>	Create cell to be used for synchronized access to a global variable from any task

Examples

Testing and assertions:

```
o)load[getenv[`OHOME`];"core"];
o)
o)// Run test - passes
o)test["addition";1+1;2]
--- Test [addition] Passed
o)
o)// Run test - fails
o)test["wrong";1+1;3]
--- Test [wrong] Failed
> Expected: 3
> Got: 2
---
o)
o)// Assert condition
o)assert[5>3;"five should be greater than three"]
o)
o)// Assert equality
o)assert_eq[10;5+5;"sum should equal 10"]
```

String manipulation:

```
o)// Trim whitespace
o)trim[" hello "]
"hello"
o)ltrim[" hello "]
"hello "
o)rtrim[" hello "]
" hello"
o)
o)// Dynamic formatting
o)fmt["Hello %, you are % years old";("Alice";"30")]
"Hello Alice, you are 30 years old"
```

Vector operations:

```
o)// Fill or cut vector to size
o)fc[1 2 3;5;0] // fill with 0
1 2 3 0 0
o)fc[1 2 3 4 5;3;0] // cut to size 3
1 2 3
o)
o)// Check where consecutive items differ
o)differ[1 1 2 2 3 3]
101010b
o)differ["AAABBBC"]
"0001001b"
```

Bit operations:

```
o)// Left shift
o)lshift[5;2] // 5 << 2 = 20
20
o)
o)// Right shift
o)rshift[20;2] // 20 >> 2 = 5
5
o)
o)// Left rotate
o)lrot[5;2]
20
```

CSV operations:

```
o)// Create table
o)t: +`name`age`city!(("Alice";"Bob");25 30;("NYC";"LA"));
o)
o)// Write CSV to file
o)wcsv[t;C];",",";`$`:/tmp/data.csv";
o)
o)// Read CSV from file (auto-detect columns)
o)t2: rcsv[C];",",";`symbol`long`symbol;`$`:/tmp/data.csv";
o)
o)// Read CSV with specific column names
o)t3: rcsv[`name`age`city;",";`symbol`long`symbol;`$`:/tmp/data.csv];
```

Parallel execution:

```
o)// Parallel each - run function in parallel
o)peach[{{x*x}};1 2 3 4 5]
1 4 9 16 25
o)
o)// Extended parallel each - splits work across cores
o)xpeach[{{x*x}};!1000] // Process 1000 items in parallel
```

Dictionary operations:

```
o)// Concatenate dictionaries
o)d1: `a`b!(1 2;3 4);
o)d2: `b`c!(5 6;7 8);
o)con[d1;d2]
a| 1 2
b| 3 4 5 6
c| 7 8
```

Thread-safe cell:

```
o)// Create cell for synchronized access
o)counter: 0;
o)cell: .o.cell[`counter];
o)
o)// Multiple threads can safely update counter through cell
o)spawn[{{cell[{{x+1}}];O}] // Increment in thread
```



The `peach` and `xpeach` functions require starting tachyon with `-c N` flag where N is the number of scheduler threads.

[Parallel execution](#)

[Tables](#)

Additional functions for tables

Name	Description / comments
<code>ej[<tbl1>;<tbl2>]</code>	Each join
<code>except[<tbl1>;<tbl2>]</code>	Returns table1 entries which are not in table2
<code>xcol[<names>;<tbl>]</code>	Rename table columns
<code>xcols[<names>;<tbl>]</code>	Reorder table columns
<code>.o.en[<tbl>;<sym>]</code>	Returns table with columns enumerated
<code>.o.sym[<tbl>]</code>	Returns table with enum columns converted into symbols

.o.en

As you probably know, symbols cannot be part of splayed tables due to technical reasons. In order to permit symbol-like behaviour in tables, enums are used. The usual task of converting symbols into enums can be done using `.o.en` function.

Syntax: `.o.en[t;sym]`

where `t` is a table or table symbol for in-place modification;

`sym` is a global domain symbol, i.e. ``sym`.

All symbol fields in the table will be converted into enum fields. Global domain vector will contain new symbols (if necessary) added to its back.

```
o)load "core";
o)t:+`a`b`c!(1 2;3 4;`c`d);
o)sym:`symbol$(0);
o)r:.o.en[t;`sym];
o)r`c
`sym$`c`d
o)
```

When a table has just been loaded from disk, all its enum fields are not bind to any domain vector. One can use the same `.o.en` function to rebind them back.

```

o)load "core";
o)sym:`c`d;
o)t:+`a`b`c!(1 2;3 4;`sym$`c`d);
o)`:/tmp/tbl/ set t; t:0;
o)load `:/tmp/tbl/;
o)tbl
a b c
-----
1 3 0
2 4 1i
o).o.en[`tbl;`sym];
o)tbl
a b c
-----
1 3 `c
2 4 `d

```

.o.sym

A less frequent, but still useful procedure is turning enums into ordinary symbols. It's done using `.o.sym` function.

Syntax: `.o.sym[t]`

where `t` is a table or table symbol for in-place modification.

```

o)load "core";
o)sym:`c`d;
o)t:+`a`b`c!(1 2;3 4;`sym$`c`d);
o).o.sym[`t];           // destructive enums -> symbols
o)t
a b c
-----
1 3 c
2 4 d
o)t`c
`c`d

```

Additional functions for partitioned tables

Name	Description / comments
<code>.o.pnew[<mt>;<pd>]</code>	Creates new partitioned table or appends new partition
<code>.o.pget[<dir>]</code>	Loads partition metadict from disk & converts into partitioned table
<code>.o.pset[<dir>;<mt>]</code>	Stores partitions and metadict on disk
<code>.o.pstore[<dir>;<mt>;<ids>]</code>	Stores partitions and metadict on disk
<code>.o.psym[<mt>;<ids>]</code>	Re-assign domain symbol
<code>.o.pdel[<mt>;<ids>]</code>	Deletes partitions from partitioned table
<code>.o.pmd[<dir>]</code>	Return partition metainfo dict filename relative to "dir"
<code>.o.pmnt[<mt>;<ids>]</code>	Mounts specific partitions
<code>.o.pumnt[<mt>;<ids>]</code>	Un-mounts specific partitions
<code>.o.pmtvalid[<mt>]</code>	Checks if mt is proper metainfo dict
<code>.o.pvalid[<mt>]</code>	Checks if mt is ptable or proper metainfo dict
<code>.o.pidvalid[<ids>]</code>	Checks if partition ids are valid
<code>.o.pid2ix[<mt>;<ids>]</code>	Translate partition ids into simple indices

where

`mt` - ptable or ptable metainfo dict or ONO (for new ptable creation)

`pd` - dict with ``gf`info` fields describing new partition

`dir` - ptable directory symbol

`ids` - partition identifiers

Creating ptable or adding new partition

Passing ONO to `.o.pnew` as existing partitioned table makes a new table, otherwise adds a new partition.

```

o) load "core";
o) gf:+`a`b!(1 2;10 20);
o) info:+`size`segId`refPath!(2 2; 0 0; (0N0;0N0));
o) p1: +`c`d!(100 100;200 200);
o) p2: +`c`d!(1000 1000;2000 2000);
o) mnt:+`mntval!(p1;p2);
o) pd: `gf`info`mnt!(gf;info;mnt);
o) pt:.o.pnew[0N0; pd];
o) pt
a b c d
-----
1 10 100 200
1 10 100 200
2 20 1000 2000
2 20 1000 2000

```

And creating new ptable with enum fields is done like:

```

o) load "core";
o) sym1::`1`2`3`4`100`1000;
o) sym2::sym1;
o) gf:+`a`b!(`sym1$`1`2;10 20);
o) info:+`size`segId`refPath!(2 2; 0 0; (0N0;0N0));
o) p1: +`c`d!(`sym1$`100`100;200 200);
o) p2: +`c`d!(`sym1$`1000`1000;2000 2000);
o) mnt:+`mntval!(p1;p2);
o) pd: `gf`info`mnt`sym!(gf;info;mnt;`sym2);
o) gf2:+`a`b!(`sym1$`3`4;10 20);
o) pd2: `gf`info`mnt!(gf2;info;mnt);
o) pt:.o.pnew[`ptable$pd; pd2];
o) pt`a
`sym2$`1`1`2`2`3`3`4`4

```

Note `sym2` becomes domain for the entire ptable.

See following for adding new partition.

```

o) load "core";
o) gf:+`a`b!(1 2;10 20);
o) info:+`size`segId`refPath!(2 2; 0 0; (0N0;0N0));
o) p1: +`c`d!(100 100;200 200);
o) p2: +`c`d!(1000 1000;2000 2000);
o) mnt:+`mntval!(p1;p2);
o) pd: `gf`info`mnt!(gf;info;mnt);
o) gf2:+`a`b!(3 4;10 20);
o) pd2:`gf`info`mnt!(gf2;info;mnt);
o) pt:.o.pnew[`ptable$pd; pd2];
o) pt
a b c d
-----
1 10 100 200
1 10 100 200
2 20 1000 2000
2 20 1000 2000
3 10 100 200
3 10 100 200
4 20 1000 2000
4 20 1000 2000

```

Saving / loading ptable to/from disk

Ptables are saved into one directory by default. Sub-directories are named using random GUID values unless manually named (use `refPath`).

`.o.pset` function is used for saving to disk, `.o.pget` - to get saved ptable meta from disk.

```

o) load "core";
o) gf:+`a`b!(1 2;10 20);
o) info:+`size`segId`refPath!(2 2; 0 0; (0N0;0N0));
o) p1: +`c`d!(100 100;200 200);
o) p2: +`c`d!(1000 1000;2000 2000);
o) mnt:+`mntval!(p1;p2);
o) pd: `gf`info`mnt!(gf;info;mnt);
o) pt:.o.pnew[0N0; pd];
o) .o.pset[`:tmp/pset1/; pt];
o) pt2:.o.pget[`:tmp/pset1/];

```



Important! Please pay attention to trailing slash at the end of the dir symbol.

Just as with ordinary table, it's user responsibility to assign domains to enums. Use `.o.psym` function to re-assign domain symbol after loading ptable.

```

o) load "core";
o) sym1::`1`2`100`1000;
o) sym2::sym1;
o) gf:+`a`b!(`sym1$`1`2;10 20);
o) info:+`size`segId`refPath!(2 2; 0 0; (0N0;0N0));
o) p1: +`c`d!(`sym1$`100`100;(200;"TEST"));
o) p2: +`c`d!(`sym1$`1000`1000;(2000;"TEST"));
o) mnt:+`mntval!(p1;p2);
o) pd: `gf`info`mnt!(gf;info;mnt);
o) pt:.o.pnew[0N0; pd];
o) .o.pset[`:/tmp/pset3/; pt];
o) pt2:.o.pget[`:/tmp/pset3/];
o) pt2:.o.psym[pt2; `sym2];
o) pt2`a
`sym2$`1`1`2`2

```

Partition ids

Several following functions accept partition ids as argument. Partition ids can be defined as:

Id value	Example	Meaning
0N0	0N0	all partitions
vector of longs	,0	only partitions with given indices starting from zero
table of partitioning fields	+`x`y!(1 2; 2000.09.11 2022.02.24)	only partitions with corresponding partitioning fields

Saving individual partitions to disk

`.o.pstore` function is just more general alternative to `.o.pset`. It can be used to save all or one or several partitions at a time along with meta-dict. It expects target directory as first arg, `ptable` - as second one, and partition ids - as third one.

```

o) load "core";
o) gf:+`a`b!(1 2;10 20);
o) info:+`size`segId`refPath!(2 2; 0 0; (0N0;0N0));
o) p1: +`c`d!(100 100;200 200);
o) p2: +`c`d!(1000 1000;2000 2000);
o) mnt:+`mntval!(p1;p2);
o) pd: `gf`info`mnt!(gf;info;mnt);
o) pt:.o.pnew[0N0; pd];
o) .o.pstore[`:/tmp/pset1/; pt; ,0]; // just first partition + meta-dict will be saved
o) .o.pstore[`:/tmp/pset1/; pt; ,( #gf)-1]; // just last partition + meta-dict will be saved
o) .o.pstore[`:/tmp/pset1/; pt; 0N0]; // all partitions + meta-dict will be saved
o) .o.pstore[`:/tmp/pset1/; pt; gf]; // all partitions defined by "gf" + meta-dict will be saved

```

Deleting partitions

Removing partition is made using `.o.pdel` function. It returns partitioned table without deleted partition. Partition are specified by their partition ids.

```
o) load "core";
o) gf:+`a`b!(1 2;10 20);
o) info:+`size`segId`refPath!(2 2; 0 0; (0N0;0N0));
o) p1: +`c`d!(100 100;200 200);
o) p2: +`c`d!(1000 1000;2000 2000);
o) mnt:+`mntval!(p1;p2);
o) pd: `gf`info`mnt!(gf;info;mnt);
o) pt:.o.pnew[0N0; pd];
o) pt2:.o.pdel[pt; 0];
o) pt2
a b c d
-----
2 20 1000 2000
2 20 1000 2000
```

Mounting / unmounting partitions

To avoid quite heavy lazy mounting and immediately unmounting partitions, manual mounting is supported.

See example below. Note compound lists partitions.

```
o) // test with compound list in partitions
o) load "core";
o) gf:+`a`b!(1 2;10 20);
o) info:+`size`segId`refPath!(2 2; 0 0; (0N0;0N0));
o) .p1: +`c`d!(100 100;(200;"TEST"));
o) .p2: +`c`d!(1000 1000;(2000;"TEST"));
o) mnt:+`mntval!(.p1;.p2);
o) pd: `gf`info`mnt!(gf;info;mnt);
o) pt:.o.pnew[0N0; pd];
o)
o) // save to disk and load back
o) .o.pset[`:~/tmp/pmnt_umnt1/; pt];
o) pt2:.o.pget[`:~/tmp/pmnt_umnt1/];
o)
o) pt3:.o.pmnt[pt2; gf];
o) d3:`dict$pt3;
o) pt4:.o.pumnt[pt3; gf];
o) d4:`dict$pt4;
o) d3
gf | +`a`b!(1 2;10 20)
info| +`size`segId`refPath!(2 2;0 0;(C` :1c516dea-89b1-4182-9f1f-feeceb3420efe/;`:3848a412-1c17-4935-92ae-fd6ce2dc4a40/))
mnt | +,`mntval!((+`c`d!(100 100;(200;"TEST"));+`c`d!(1000 1000;(2000;"TEST"))))
root| `~/tmp/pmnt_umnt1/
o) d4
gf | +`a`b!(1 2;10 20)
info| +`size`segId`refPath!(2 2;0 0;(C` :1c516dea-89b1-4182-9f1f-feeceb3420efe/;`:3848a412-1c17-4935-92ae-fd6ce2dc4a40/))
mnt | +,`mntval!(C` :1c516dea-89b1-4182-9f1f-feeceb3420efe/;`:3848a412-1c17-4935-92ae-fd6ce2dc4a40/))
root| `~/tmp/pmnt_umnt1/
```

Note how `mntval` contents change in each stage.

 [Partitioned tables](#)

htreq.o

Library for query platform over HTTP.

To properly work, must be defined the system environment variable OHOME with a path to the platform.

To simply start the server need call `.http.run[]`.

Elements are used to start the server

Name	Description / comments
<code>.http.default.bind</code>	Get default / set string with binding ip:port
<code>.http.default.host</code>	Get default / set string with user name
<code>.http.default.home</code>	Get default / set string with current dir
<code>.http.default.handle[<sock>;<headers>]</code>	Request handler function (automatically set by htreq module)
<code>.http.run[<creds>]</code>	Start http server

Examples

Start HTTP query server:

```
o)// Save this as query_server.o and run with:
o)// tachyon -c 0 -f query_server.o
o)
o)load[getenv[`OHOME`; "htreq"];
o)// Server starts automatically and listens for HTTP requests
o)// Query format: GET /request.FORMAT?O_EXPRESSION
```

Query server from command line:

```
# Simple calculation (JSON format)
$ curl "http://localhost:8080/request.json?1+2+3"
6

# Create and query table (JSON format)
$ curl "http://localhost:8080/request.json?t:+`a`b!(1 2 3;4 5 6)"
{"a": [1,2,3], "b": [4,5,6]}

# Query table (CSV format)
$ curl "http://localhost:8080/request.csv?+`name`age!(`Alice` `Bob`;25 30)"
name,age
Alice,25
Bob,30

# Execute expression
$ curl "http://localhost:8080/request.json?2 xexp 10"
1024
```



The hreq.o module executes arbitrary O code from HTTP requests. Only use on trusted networks or implement authentication. Never expose to public internet without proper security measures.

Auxiliary elements

Name	Description / comments
.ht.json[<x>]	Convert result to JSON format
.ht.csv[<x>]	Convert result to CSV format
.ht.proc[<sock>;<fmt>;<req>]	Process HTTP request and execute O expression
.ht.catch[<sock>;<req>;<msg>]	Handle errors during request processing

http.o

Helps to create a simple HTTP server.

Elements are used to start the server

Name	Description / comments
<code>.http.default.bind</code>	Get default / set string with binding ip:port
<code>.http.default.host</code>	Get default / set string with user name
<code>.http.default.home</code>	Get default / set string with current dir
<code>.http.default.handle[<sock>;<headers>]</code>	Request handler function - receives socket and headers dictionary, must send response
<code>.http.run[<creds>]</code>	Start http server

Example:

Basic HTTP server:

```
o)load[getenv[`OHOME];"http"];
o)
o)// Configure server
o).http.default.bind: "127.0.0.1:8080";
o).http.default.host: "MyServer";
o).http.default.home: getenv[`HOME];
o)
o)// Define request handler
o).http.default.handle: {[sock;headers]
  path: "c"$headers[`Path];

  if [path~/hello"] {
    body: "
```

Hello from The Platform!

```
";
  .http.resp.ok[sock;`html;body];
} elif [path~/"] {
  body: "
```

Welcome

Server is running

```
";
  .http.resp.ok[sock;`html;body];
} else {
  .http.resp.error[sock;404;"Not found";"Page does not exist"];
}
};
o)
o)// Start server (requires -c 0 flag: tachyon -c 0 -f script.o)
o).http.run[];
---
HTTP server version: 1.0.4
Started: 2026.03.26D01:30:00.000000000
Bind: 127.0.0.1:8080
Hostname: MyServer
```

Parse URL:

```
o).http.url.parse["http://localhost:8080/test.html?id=123"]
Path | http://localhost:8080/test.html
Query| id=123
```

Send responses:

```
o)// In request handler:
o)// Success with HTML
o).http.resp.ok[sock;`html;"]
```

OK

```
"];
o)
o)// Success with JSON
o).http.resp.ok[sock;`json;{"status\":"ok\"}"];
o)
o)// Error response
o).http.resp.error[sock;404;"Not Found";"The requested page does not exist"];
o)
o)// Custom response
o).http.resp.send[sock; .http.const.code`ok; .http.const.mime`html;"]
```

Custom

```
"];
```

Auxiliary elements

Name	Description / comments
<code>.http.const.version[]</code>	Version string
<code>.http.const.error</code>	Template string of error with 3 %
<code>.http.const.code</code>	Dictionary with supported error codes
<code>.http.const.mime</code>	Dictionary with supported mime
<code>.http.const.head</code>	Template string of head page with 4 %
<code>.http.const.temp</code>	Template string of html page with 2 % for title and body
<code>.http.buffer.handle</code>	Return the lambdas which processing socket
<code>.http.url.parse[<url>]</code>	Parse URL to dictionary
<code>.http.req.parse[<req>]</code>	Request parse to dictionary
<code>.http.resp.error[<sock>;<code>;<payload>;<explanation>]</code>	Response error
<code>.http.resp.ok[<sock>;<mime>;<payload>]</code>	Response OK
<code>.http.resp.send[<sock>;<code>;<mime>;<payload>]</code>	Send response
<code>.http.serve[<bind>;<handle>;<creds>]</code>	Start http server

json.o

Library for JSON serialization. Converts O language data structures into JSON format strings.

Main element

Name	Description / comments
json[<x>]	Serialize any O value to JSON string

Examples

Numbers:

```
o)load[getenv[`OHOME];"json"];
o)json[42]
"42"
o)json[3.14]
"3.14"
```

Strings:

```
o)json["hello world"]
"\hello world\""
```

Arrays:

```
o)json[1 2 3 4 5]
"[1,2,3,4,5]"
o)json["red" "green" "blue"]
"[\red\,\green\,\blue\"]"
```

Boolean values:

```
o)json[1b]
"true"
o)json[0b]
"false"
```

Null:

```
o)json[0N0]
""
```

Dictionaries:

```
o)json[`name`age`city]("Alice";25;"Paris")]  
"{name:\"Alice\",age:25,city:\"Paris\"}"
```

Nested dictionaries:

```
o)d: `user`data!("John";`age`city!(30;"NYC"));  
o)json[d]  
"{user:\"John\",data:{age:30,city:\"NYC\"}}"
```

Tables:

```
o)t: +`a`b!(1 2 3;4 5 6);  
o)json[t]  
"{a:[1,2,3],b:[4,5,6]}"
```

Lists:

```
o)json[(1;2;3)]  
"[1,2,3]"  
o)json[(42;"hello";1b)]  
"[42,\"hello\",true]"
```

Auxiliary elements

Name	Description / comments
array[<x>]	Convert 0 array to JSON array format
jsmap[<x>]	Convert 0 dictionary/table to JSON object format

lit.o

Literate programming parser for `.lit` files. Allows mixing documentation and code in a single file following the [literate programming](#) paradigm invented by Donald Knuth.

Main elements

Name	Description / comments
<code>lparse[<file>]</code>	Parse <code>.lit</code> file and return structured data
<code>src[<parsed>]</code>	Extract source code from parsed <code>.lit</code> data
<code>html[<parsed>]</code>	Generate HTML documentation from parsed <code>.lit</code> data

Examples

Literate program structure:

```
@title My Program

@h1 Introduction

This section explains what the program does.

---program.o
// Main function
main: {
    println["Hello from literate program"];
};
---

@h2 Usage

Call main[] to run the program.
```

Parse `.lit` file:

```
o)load[getenv[`OHOME];"lit"];
o)
o)// Parse literate program file
o)parsed: lparse[`${}:path/to/program.lit"];
o)
o)// Extract source code
o)(code;filename): src[parsed];
o)code // Source code as string
o)filename // Target filename (e.g., :program.o)
o)
o)// Generate HTML documentation
o)doc: html[parsed];
o)write[`${}:path/to/output.html"; "x"$doc];
```

.lit file format

Documentation directives:

Directive	Description	Example
<code>`@title TEXT`</code>	Document title	<code>`@title My Library`</code>
<code>`@h1 TEXT`</code>	Heading level 1	<code>`@h1 Introduction`</code>
<code>`@h2 TEXT`</code>	Heading level 2	<code>`@h2 Implementation`</code>
<code>`@h3 TEXT`</code>	Heading level 3	<code>`@h3 Details`</code>
<code>`@h4 TEXT`</code>	Heading level 4	<code>`@h4 Notes`</code>
<code>`@h5 TEXT`</code>	Heading level 5	<code>`@h5 Subsection`</code>
<code>`@table TEXT`</code>	Table marker	<code>`@table Results`</code>

Code blocks:

```
---filename.o
code here
---
```

Code blocks must have `.o` extension in filename.

Include blocks:

```
---identifier
content
---
```

Include blocks can be referenced and inserted into code using `@{identifier}` syntax.

Ignore blocks:

```
---[identifier]
content to ignore
---
```

Content in ignore blocks is not included in the output code but may be used for examples or documentation.

Auxiliary elements

Name	Description / comments
htempl	HTML template for generated documentation
.space	PEG parser: parse spaces
.title	PEG parser: parse @title directive
.h1-.h5	PEG parser: parse heading directives
.table	PEG parser: parse @table directive
.sep	PEG parser: parse separator `---`
.code	PEG parser: parse code blocks
.incl	PEG parser: parse include blocks
.igno	PEG parser: parse ignore blocks
.mark	PEG parser: parse any markup
.text	PEG parser: parse plain text
.entit	PEG parser: parse entities
.lit	PEG parser: top-level entry point

Use case

Literate programming is useful when:

- Writing libraries with extensive documentation
- Creating educational code examples
- Maintaining complex algorithms where explanation is as important as implementation
- Generating both executable code and readable documentation from single source

The `.lit` format allows you to write prose explanations and code together, then extract clean source code for execution and beautiful HTML documentation for reading.

[Literate Programming on Wikipedia](#)

[Dynamic parsers](#)

markdown.o

Markdown parser based on PEG (Parsing Expression Grammar). Converts Markdown text into structured O data.

Main element

Name	Description / comments
<code>.md.pars[<markdown text>]</code>	Parse markdown string and return list of dictionaries

Examples

Parse simple markdown:

```
o)load[getenv[`OHOME`];"markdown"];
o)md: "c"$read[`${`:/path/to/file.md`}];
o)parsed: .md.pars md;
o)parsed
,`h1!(Hello World)
,`para!((,`text!(This is );,`bold!(bold);,`text!( text.)))
```

Supported markdown elements:

Element	Syntax	Parsed as
Heading 1	<code>`# Text`</code>	<code>`` `h1!(Text) ``</code>
Heading 2	<code>`## Text`</code>	<code>`` `h2!(Text) ``</code>
Heading 3-6	<code>`### Text` ...</code>	<code>`` `h3!(Text) `` ... `` `h6!(Text) ``</code>
Horizontal rule	<code>`---`</code>	<code>`` `hr!... ``</code>
Bold	<code>`**text**`</code>	<code>`` `bold!(text) ``</code>
Italic	<code>`_text_`</code>	<code>`` `curs!(text) ``</code>
Inline code	<code>`` `code` ``</code>	<code>`` `esc1!(code) ``</code>
Code block	<code>` `` `langncoden` `` `</code>	<code>`` `code!(`lang`body!(lang;code)) ``</code>
Link	<code>`[text](url)`</code>	<code>`` `link!(text:url) ``</code>
Image	<code>`![alt](url)`</code>	<code>`` `image!(alt:url) ``</code>
List	<code>`- item` or `1. item`</code>	<code>`` `list!(...) ``</code>
Table	<code>` col1 col2 `</code>	<code>`` `table!(...) ``</code>
Blockquote	<code>`> text`</code>	<code>`` `block!(text) ``</code>
Alert	<code>`::: notentextn:::`</code>	<code>`` `note!(text) ``</code>
Paragraph	Plain text	<code>`` `para!(text) ``</code>

Complete example:

```
o)load[getenv[`OHOME];"markdown"];
o)
o)md: "# Title\n\n",
      "This is bold and italic.\n\n",
      "
```

```
on", "a: 1 + 2;n", "\n\n",
```

```
"- Item 1\n",
```

```
"- Item 2\n";
```

```
o)
o)parsed: .md.pars md;
o)#parsed // Number of top-level elements
4
o)
o)parsed[0] // First element (heading)
, `h1!(Title)
o)
o)parsed[1] // Second element (paragraph)
, `para!(, `text!(This is );, `bold!(bold);, `text!( and );, `curs!(italic);, `text!(.))
```

PEG Grammar elements

The parser uses PEG (Parsing Expression Grammar) with the following building blocks:

Name	Description
<code>.md.concat</code>	Concatenate parsed results
<code>.md.indent</code>	Calculate list indentation level
<code>.md.sp</code>	Parse spaces
<code>.md.lf</code>	Parse line feeds (<code>\r\n</code> or <code>\n</code>)
<code>.md.h1</code> - <code>.md.h6</code>	Parse headings level 1-6
<code>.md.hr</code>	Parse horizontal rule
<code>.md.esc1-4</code>	Parse escaped/code elements
<code>.md.code</code>	Parse fenced code blocks
<code>.md.link</code>	Parse links
<code>.md.img</code>	Parse images
<code>.md.embd</code>	Parse embedded content
<code>.md.bold</code>	Parse bold text
<code>.md.curs</code>	Parse italic (cursive) text
<code>.md.list</code>	Parse lists (ordered and unordered)
<code>.md.alrt</code>	Parse alert blocks (<code>::: note</code> , <code>::: warn</code> , etc.)
<code>.md.blck</code>	Parse blockquotes
<code>.md.tble</code>	Parse tables
<code>.md.para</code>	Parse paragraphs
<code>.md.pars</code>	Top-level parser entry point

 [Dynamic parsers](#)

 [PEG on Wikipedia](#)

prolog.o

Prolog interpreter implemented in O language. Supports logic programming with facts, rules, queries, and unification.

Usage

Load Prolog module:

```
o)load[getenv[`\$HOME];"prolog"];
```

Switch to Prolog REPL mode:

```
o)olog[];  
?) // Prolog prompt
```

Load Prolog facts from file:

```
?)file["path/to/facts.pl"]
```

Main elements

Name	Description / comments
be[<fact or rule>]	Add fact or rule to knowledge base
search[<term>]	Query the knowledge base
olog[]	Switch to Prolog REPL mode
orpl[]	Switch back to O REPL mode
file[<path>]	Load Prolog facts/rules from file
term[<x>]	Create term structure from expression
rule[<x>]	Create rule structure
goal[<r>;<p>;<e>]	Create goal for search

Prolog syntax

Facts:

```
?)parent(tom, bob).
?)parent(tom, liz).
?)parent(bob, ann).
?)parent(bob, pat).
?)parent(pat, jim).
```

Rules:

```
?)grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
?)ancestor(X, Z) :- parent(X, Z).
?)ancestor(X, Z) :- parent(X, Y), ancestor(Y, Z).
```

Queries:

```
?)parent(tom, bob)?
Yes
?)parent(tom, X)?
X=bob
X=liz
?)grandparent(tom, Who)?
Who=ann
Who=pat
?)ancestor(tom, jim)?
Yes
```

Examples

Family relationships:

```
o)load[getenv[`\$HOME];"prolog"];
o)o!og[];
?)
?)// Define facts
?)parent(tom, bob).
?)parent(bob, ann).
?)
?)// Define rule
?)grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
?)
?)// Query
?)grandparent(tom, Who)?
Who=ann
Yes
?)
?)// Switch back to 0 mode
?)\\
o)
```

Logic puzzles:

```

?)// Who likes what?
?)likes(mary, food).
?)likes(mary, wine).
?)likes(john, wine).
?)likes(john, mary).
?)
?)// Query: What does Mary like?
?)likes(mary, X)?
X=food
X=wine
?)
?)// Query: Who likes wine?
?)likes(Who, wine)?
Who=mary
Who=john

```

List operations:

```

?)// List membership
?)member(X, [X]).
?)member(X, [H, T]) :- member(X, T).
?)
?)// Query
?)member(2, [1, 2, 3])?
Yes

```

Global state

Variable	Description
FACTS	Knowledge base (list of facts and rules)
QUEUE	Search queue for goal processing
TRACE	Enable/disable trace mode (boolean)

Debugging commands

Command	Description
`trace(1)`	Enable trace mode
`trace(0)`	Disable trace mode
`dump()`	Print all facts and rules
`\`	Exit Prolog mode (return to 0 REPL)

Auxiliary elements

Name	Description
env[]	Create empty environment
push[<x>]	Push goal onto search queue
pop[]	Pop goal from search queue
isVar[<x>]	Check if term is a variable (starts with uppercase)
isConst[<x>]	Check if term is a constant
unify[<src>;<srcEnv>;<dst>]	Unify two terms
unifyTerm[<term>;<env>]	Resolve term in environment
fmt[<x>]	Format term/rule for display
prin[<x>]	Print formatted term
dump[<x>]	Print environment bindings

PEG parser elements

Name	Description
ospce	Parse whitespace
oiden	Parse identifier
oargs	Parse argument list
ofact	Parse fact
orule	Parse rule (fact :- goals)
oterm	Parse term (fact or rule)
oquit	Parse quit command (\)
otrce	Parse trace command
odump	Parse dump command
oqry	Parse query (ends with ?)
oexpr	Parse any Prolog expression
oxprs	Parse multiple expressions

How it works

The Prolog interpreter uses:

- **Unification:** Pattern matching algorithm that binds variables to values
- **Search:** Depth-first search through the knowledge base

- **Backtracking:** Automatic exploration of alternative solutions
- **Environment:** Maps variables to their bindings during search

Example session

```
o)load[getenv[`OHOME];"prolog"];
o)olog[];
?)
?)// Family tree
?)parent(alice, bob).
?)parent(alice, charlie).
?)parent(bob, david).
?)
?)child(X, Y) :- parent(Y, X).
?)sibling(X, Y) :- parent(Z, X), parent(Z, Y).
?)
?)// Queries
?)parent(alice, Who)?
Who=bob
Who=charlie
?)
?)child(david, Who)?
Who=bob
?)
?)sibling(bob, Who)?
Who=bob
Who=charlie
?)
?)dump()
parent(alice, bob)
parent(alice, charlie)
parent(bob, david)
child(X, Y) :- parent(Y, X)
sibling(X, Y) :- parent(Z, X), parent(Z, Y)
?)
?)\\
o)
```



Variables in Prolog must start with uppercase letters (X, Y, Who, etc.). Constants and predicates start with lowercase letters.

- [Dynamic parsers](#)
- [Pattern matching](#)
- [Prolog on Wikipedia](#)

repl.o

Read-Eval-Print Loop (REPL) with support for interactive sessions and remote connections.

Usage

Start interactive REPL:

```
$ tachyon -c 0 -f std/repl.o
```

Start REPL server with remote access:

```
$ tachyon -c 0 -f std/repl.o -- -p 2222
```

Start as daemon (no interactive REPL):

```
$ tachyon -c 0 -f std/repl.o -- -p 2222 -d
```

Command-line arguments

Argument	Description
<code>`-p PORT`</code>	Open port for remote connections (e.g., <code>`-p 2222`</code>)
<code>`-d`</code>	Work as daemon (do not create interactive REPL)
<code>`-desc TEXT`</code>	Override ODESC description

Main elements

Name	Description / comments
<code>.repl.version</code>	REPL version number
<code>.repl.prompt</code>	REPL prompt string (default: "o")
<code>.repl.connect[<address>]</code>	Connect to remote REPL server
<code>.repl.peval[<expr>;<esc>;<>nullv>]</code>	Protected eval with error handling
<code>.repl.fmt[<x>]</code>	Format value using current display settings

Configuration options

Name	Description / Default
<code>.repl.opt.fetchMaxLen</code>	Maximum vector/table length to send to client (default: 100000)

Examples

Connect to remote REPL:

```
o)load[getenv[`OHOME];"repl"];
o)
o)// Connect to REPL server running on localhost:2222
o).repl.connect["127.0.0.1:2222"]
Connected to remote REPL
o)
o)// Execute commands on remote server
o)a: 1 + 2;
o)a
3
```

Start REPL server:

```
# Terminal 1: Start REPL server
$ tachyon -c 0 -f std/repl.o -- -p 2222
---
  REPL server listening on port 2222
---
o)

# Terminal 2: Connect from another tachyon instance
$ tachyon
o)load[getenv[`OHOME];"repl"];
o).repl.connect["127.0.0.1:2222"]
```

Features

Error handling:

- Catches and formats runtime errors
- Shows stack traces with line numbers
- Highlights error location in code
- Displays native stack backtrace when available

Output formatting:

- Respects `\c` console width settings
- Formats tables and dictionaries for readability
- Handles special values (null, infinity)
- Color-coded error messages (when terminal supports it)

Remote REPL:

- Execute code on remote tachyon instance
- Serializes results back to client
- Supports all O data types
- Configurable data transfer limits

Auxiliary elements

Name	Description
<code>.repl.ps1</code>	Print prompt with ANSI colors
<code>.repl.out</code>	Output result and show prompt
<code>.repl.sig[<esc>;<err>]</code>	Format error message with stack trace
<code>.repl.ltrim[<x>]</code>	Trim whitespace from left
<code>.repl.rtrim[<x>]</code>	Trim whitespace from right
<code>.repl.trim[<x>]</code>	Trim whitespace from both sides
<code>.repl.xbt</code>	Format stack backtrace entry
<code>.repl.enum2sym[<x>]</code>	Convert enums to symbols
<code>.repl.psend[<ipc>;<okerr>]</code>	Protected IPC send
<code>.repl.rapi.eval[<s>]</code>	Remote API: evaluate expression

Use cases

- **Interactive development:** - Test functions and expressions interactively - Debug code with immediate feedback - Explore data structures
- **Remote execution:** - Connect to production systems for monitoring - Execute maintenance scripts remotely - Inspect running processes
- **Educational:** - Learn O language interactively - Demonstrate language features - Experiment with code snippets



When using remote REPL, ensure proper network security. Remote REPL executes arbitrary code, so only allow connections from trusted sources.

[Language reference](#)

[Tutorial](#)

sql.o

Before using the SQL-like syntax in o, you need to 'load' it as it is unavailable out-of-the-box.



The `std/sql.o` script contains an injection for the O parser, so it must be loaded separately from the script that uses it.

You can load on platform startup or with the `load "sql"` command in REPL.

SELECT and QDEF

The SQL-like syntax allows both 'query definitions' and executes queries.

The familiar `select` expressions execute immediately.

Query definitions use `qdef` instead of `select`. Otherwise, their syntax is the same.

Query definitions are needed for two reasons.

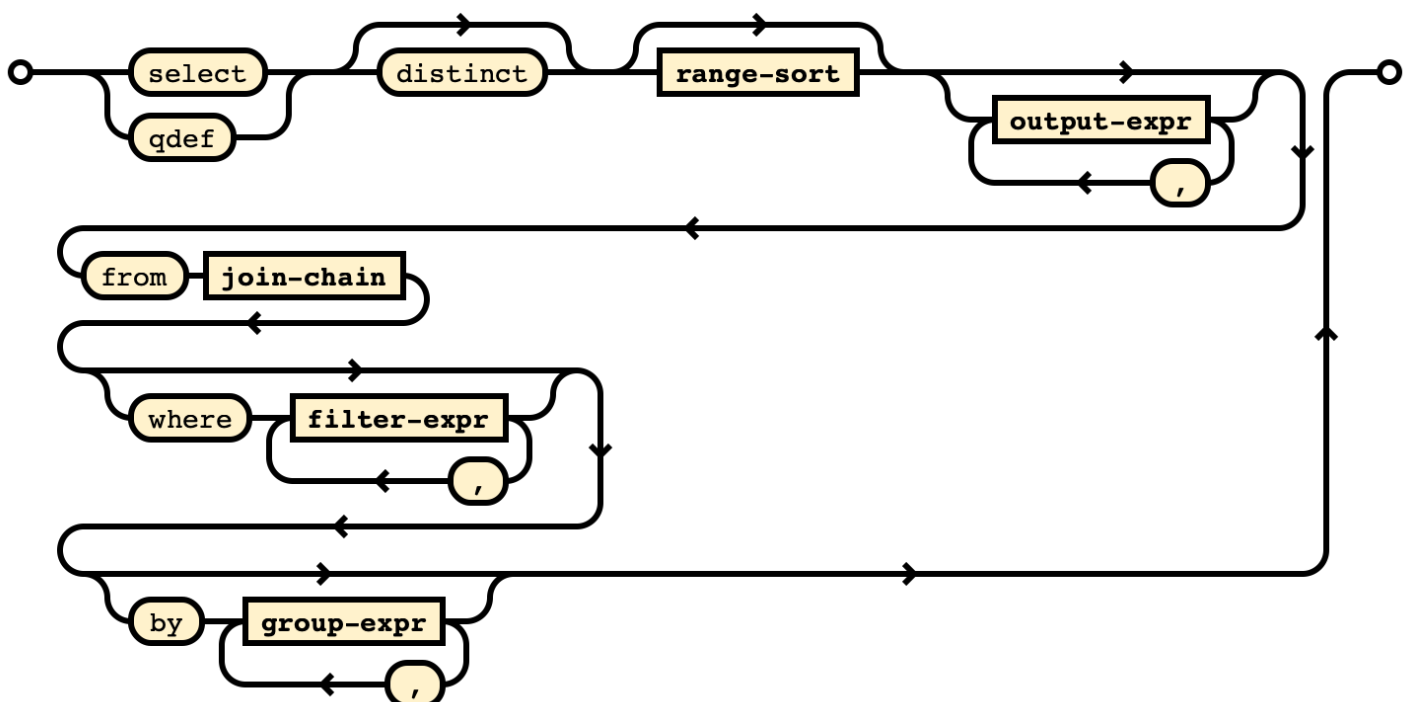
First, they allow us to introduce subqueries naturally.

```
a: qdef ... from ... ; // subquery, not executed
select ... from ... a
```

Second, they give us control over the laziness of the query execution. Given a query definition, we can 'compile' it and pull the desired number of records.

```
a:qdef ... from ...;
...
10#.a
```

The top-level diagram of the SQL-like syntax:



DISTINCT

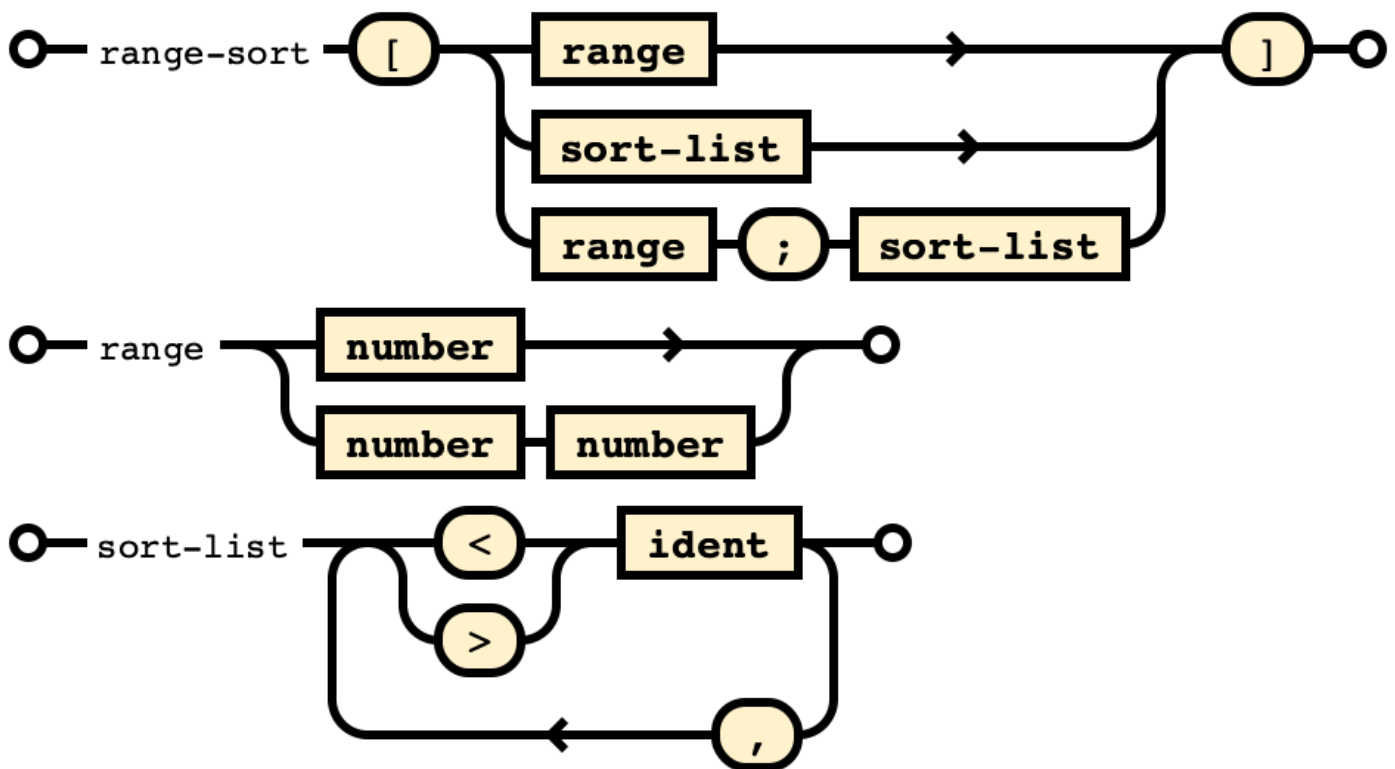


If `select` or `qdef` is followed by `distinct` the query returns only unique records.

If the query has a `by` clause it should not use `distinct` keyword.

LIMIT and ORDER

It is possible to limit the number of records returned by the query by adding a `range-sort` clause right after `select/qdef`. The grammar for the clause is as follows:



The `range-sort` clause corresponds to the 5th and 6th arguments of the functional query.

```

o)load "sql";
o)t: +`a`b`c!(1!10;1!10;10+!10)
a b c
-----
0 9 10
1 8 11
2 7 12
3 6 13
4 5 14
5 4 15
6 3 16
7 2 17
8 1 18
9 0 19
o)select [3] a, c from t
a c
----
0 10
1 11
2 12
o)select [3 3] a, c from t
a c
----
3 13
4 14
5 15
o)select [3 3;<b] a, c from t
a c
----
6 16
5 15
4 14
o)

```

Output fields

The comma-separated list of output fields follows the `select` (or `qdef`) keyword.

The list can be empty. An empty list of output fields means to return all of them.

```

o)t:+:`a`b`c!(1 2 3 4;2 3 5 7;1 4 9 16);
o)select from t
a b c
-----
1 2 1
2 3 4
3 5 9
4 7 16
o)select a, b from t
a b
---
1 2
2 3
3 5
4 7
o)

```

A output field can be renamed by prefixing it with the new name and a colon.

```
o)select x:a,y:b from t
x y
---
1 2
2 3
3 5
4 7
o)
```

For calculated fields the parser automatically chooses unique names, such as `f1`, `f2`, etc.

```
o)select a+1,b*b from t
f1 f2
-----
2 4
3 9
4 25
5 49
o)select a,a,a from t
a a1 a2
-----
1 1 1
2 2 2
3 3 3
4 4 4
o)
```

Note that aliases cannot contain dots. `kdb/q` has a similar constraint. It has a utility function to remove dots from field names.

A comma may follow the last output field just before the `from` keyword as a convenience feature. This helps a little when you put each output field on a separate line as follows:

```
select OrigTicker,
       LastDate:last date,
       OtherCustomerRef, // <- comma is OK
from mytable
```

Field references and query parameters

The SQL-like syntax supports fully qualified field references. You can refer to a field using the standard SQL syntax — `table1.field1`.

The SQL-like parser automatically captures the name of the variable containing data as a table alias.

```
select t.a from t

select t1.a + 100 from t1:longTableName
```

Any unqualified identifier, such as `a`, is a field reference. It should be unambiguous and should not be found in multiple input tables.

An identifier prefixed with `@` is a query parameter, a reference to a local (or global) variable. For example:

```
o)c:10;
o)t:+:`a`b`c!(1 2 3 4;2 3 5 7;1 4 9 16);
o)select a, b from t where c<@c
a b
---
1 2
2 3
3 5
o)
```



The values of query parameters are 'captured' upon query instantiation.

The query may run for a long time, by changes in the variables referred to by query parameters will not affect the query.

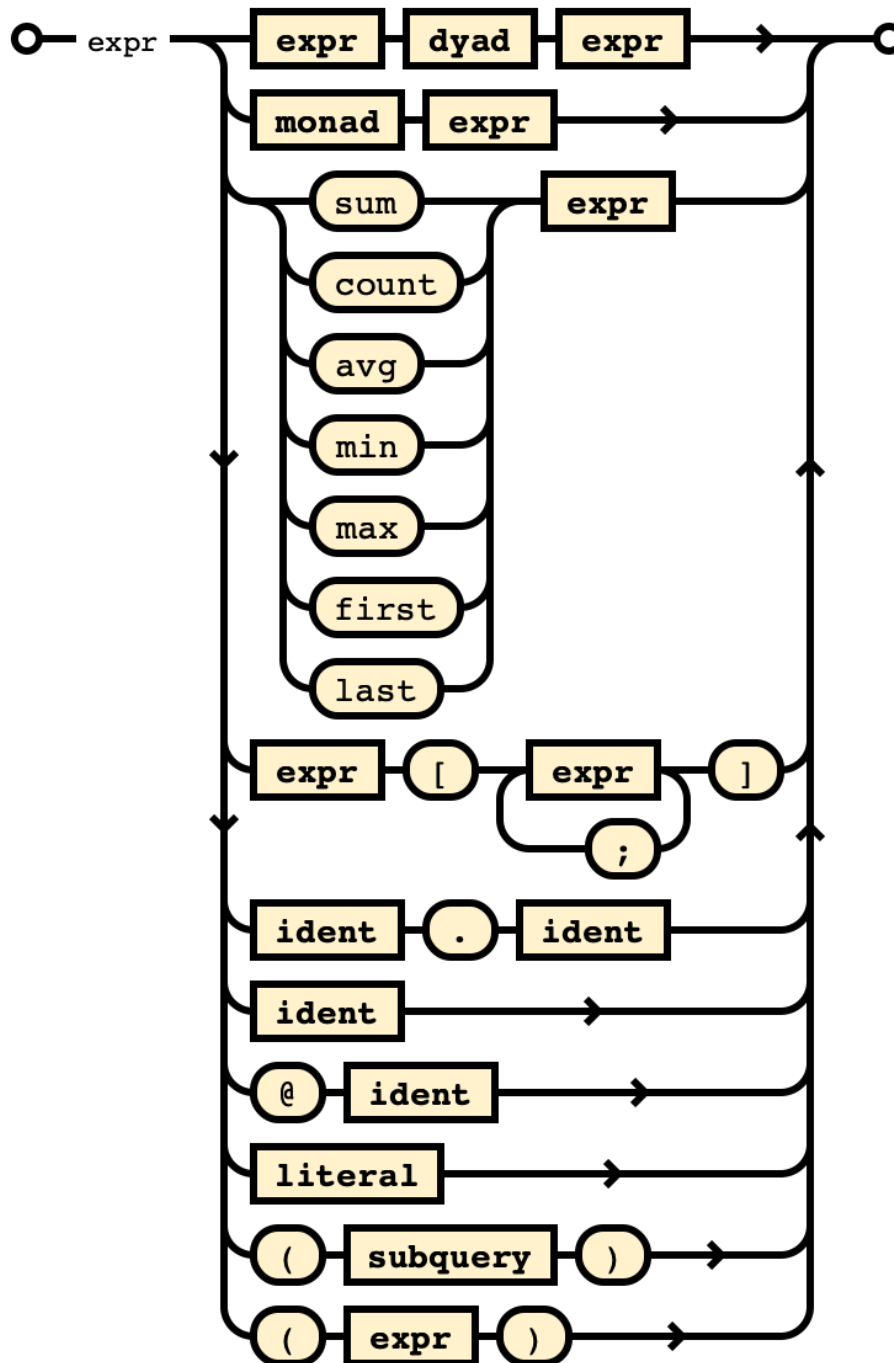
Query expressions

The syntax does not support arbitrary o-expression in the calculated fields and elsewhere. Yet the range of allowed expressions is quite rich. It is expected to expand as people start to use the syntax.

The rationale for handling the expressions within the SQL-like parser (and not in the o-parser) is to:

- Allow natural field references, such as `table1.field1`;
- Stop side effects, such as modification of a global variable;
- Stop unnecessary quoting/unquoting.

The current grammar of query expressions:



Dyads and monads allowed in the SQL-like expression must transparently work with vectors.

Thus the following verbs are allowed:

• Dyads:

- + - * % mod
- ^ (fill)
- = <> >= <= > < like in
- ~
- | or & and
- shift rotate
- xexp xlog
- bor band bxor
- ': ' each / \

• Monads:

- ^ null

- `~` `not` `bnot`
- `-` `neg`
- `%` (reciprocate)
- `|` (reverse)
- `$` (tostring)
- `sin` `cos` `tan`
- `asin` `acos` `atan`
- `exp` `log` `sqrt`
- `floor` `ceil` `round`
- `trunc` `frac`
- `abs`

```
o)t:+:`a`b`c!(1 2 3 4;2 3 5 7;1 4 9 16)
a b c
-----
1 2 1
2 3 4
3 5 9
4 7 16
o)select ab: a*b, c from t
ab c
-----
2 1
6 4
15 9
28 16
```

More complex expression can be introduced using a lambda imported into an SQL expression using a parameter. Inline lambdas are not supported.

```
median: {x[>x][#x%2]};
select @median[price] from orders
```

FROM

The `from` clause is mandatory. The simplest form of the `from` clause is `from t`, where `t` is the table's name. Instead of the table, you may use a subquery as follows.

```
o)t:+:`a`b`c!(1 2 3 4;2 3 5 7;1 4 9 16);
o)select a,b from (qdef from t where c<10)
a b
---
1 2
2 3
3 5
o)
```

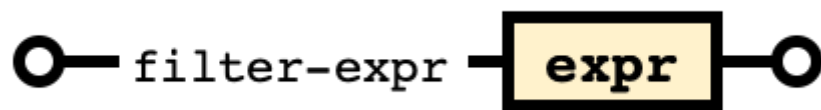
A subquery must be put in parentheses. You may use `select` in the subquery just as well.

Similarly to the fields, an input table (or a subquery) can have an alias. As with field aliases, table alias cannot contain dots.

```
o)longTableName:::`a`b`c!(1 2 3 4;2 3 5 7;1 4 9 16)
o)select from t:longTableName
a b c
-----
1 2 1
2 3 4
3 5 9
4 7 16
o)
```

WHERE

The `where` clause contains a list of arbitrary expressions separated by `,`.

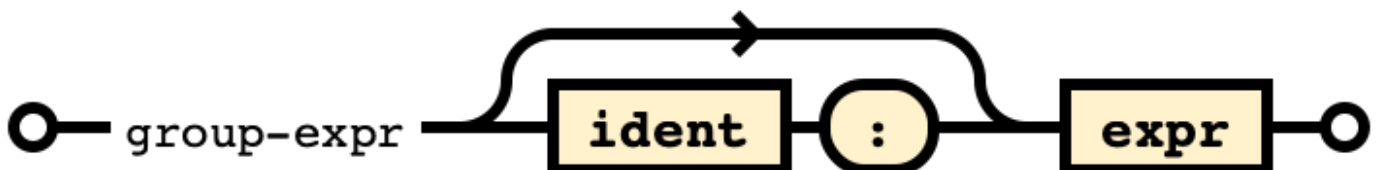


```
o)select a,b from t where c<10
a b
---
1 2
2 3
3 5
o)
```

GROUP

The `by` clause contains the list of fields (or expressions) to group the records.

The complete grammar of the element of the group list:



Similarly to the output fields the grouping fields can be renamed.

```

o)t: +`a`b`c!(10#!3;!!10;10+!10)
a b c
-----
0 9 10
1 8 11
2 7 12
0 6 13
1 5 14
2 4 15
0 3 16
1 2 17
2 1 18
0 0 19
o)select b, c from t by a
a b      c
-----
0 9 6 3 0 10 13 16 19
1 8 5 2  11 14 17
2 7 4 1  12 15 18
o)select su:sum b, av: avg c from t by a
a su av
-----
0 18 14.5
1 15 14
2 12 15
o)

```

JOIN

Any number of tables can be combined using `ij` (inner join), `lj` (left join).

```

o)t1:+:`a`b`c!(1 2 3 4;2 3 5 7;1 4 9 16);
o)t2:+:`a`f!(1 2 3;("one";"two";"three"));
o)select t2.f from t1 ij t2 on t1.a = t2.a
f
-----
"one"
"two"
"three"
o)

```

Multiple tables can be joined together, forming a join chain. The tables can be joined on multiple fields.

Instead of `on t1.a = t2.a and t1.b = t2.b and t1.c = t2.c` you may write `on (a,b,c)` to the same effect.

```

o)select t2.f from t1 ij t2 on (a)
f
-----
"one"
"two"
"three"
o)

```

Note that the list of fields must be put in parentheses. When you join more than two tables, the shortcut notation assumes that the join uses the leftmost table.

In the following example — `select ... from t1 ij t2 on (a) ij t3 on (b) ...` — the second join is a join between table `t1` and `t3` on the field `b`.

UNION

The input tables can be combined using a `ua` (union all) operation.

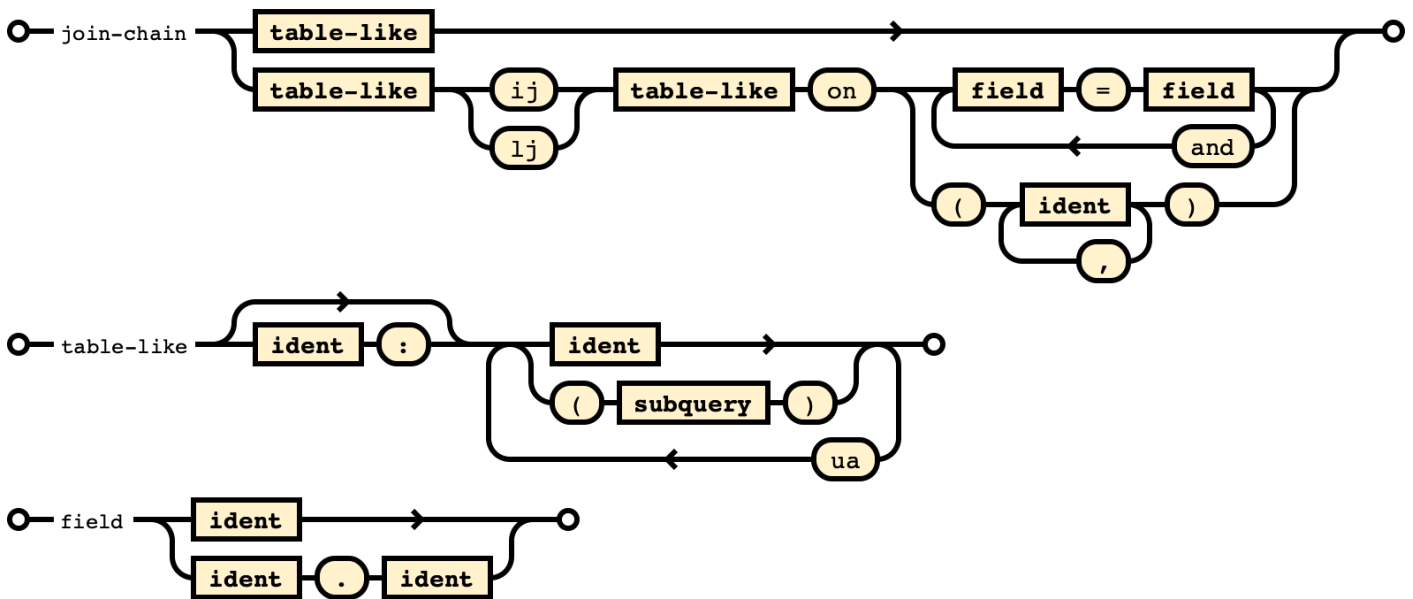
```
select ... from t1 ua t2
```

A `ua` (union all) operation has higher precedence than a join. In the example below, `t2 ua t3` is performed first, followed by a join with between `t1` and the result of the union.

```
select ... from t1 ij a:t2 ua t3 on (id)
```

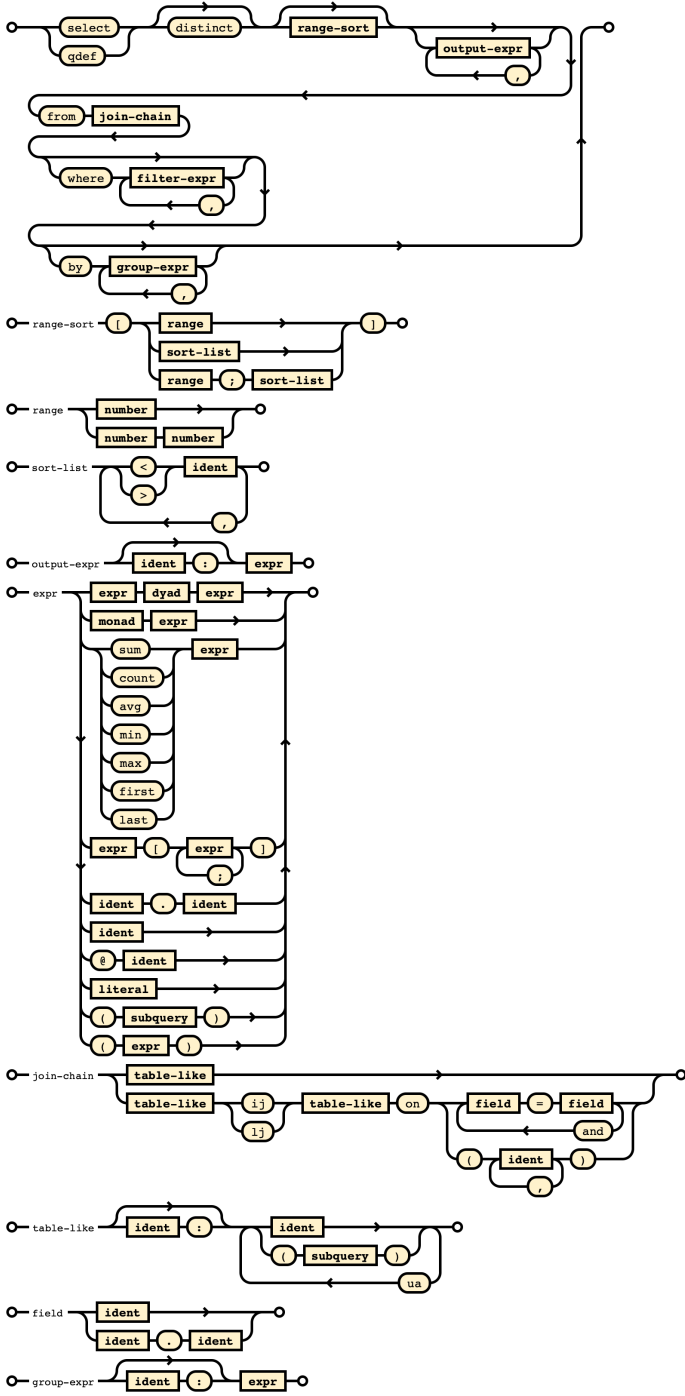
Note that an alias before the `ua` expression cannot be omitted. In general, both sides of any join operation must have aliases. In the case of a table referred to by its name, the alias is implicit.

The complete grammar of the join chain:



Arbitrary parentheses are not allowed in the `from` clause. The parentheses can only be used with a subquery or a shortcut field lists of a join.

Resume: complete SQL-like syntax grammar



queries

urllib.o

Helps in processing URL strings

Main elements

Name	Description / comments
<code>urllib.parse[<url string>]</code>	Return a dictionary with parsed URL
<code>urllib.href[<url dict>]</code>	Return URL string
<code>urllib.encode[<x>]</code>	Encode into a format that can be transmitted over the internet
<code>urllib.decode[<x>]</code>	Decode from format with "%" and whitespace as "+"

Examples:

```
o)load "urllib";
o)urllib.parse["https://tester@dazzle.com:8080"]
protocol| "https"
user    | "tester"
password| 0N0
host    | "dazzle.com"
port    | 8080
path    | 0N0
query   | 0N0
fragment| 0N0
o)urllib.href[`protocol`host`port!("http";"127.0.0.1";8080)]
"http://127.0.0.1:8080"
o)urllib.decode "http://localhost:8080/?submitted-name=adf+qe+sdfg+%2B+h"
"http://localhost:8080/?submitted-name=adf qe sdfg + h"
```

Auxiliary elements

Name	Description / comments
<code>urllib.protocol[<x>]</code>	Return field <code>`protocol</code>
<code>urllib.user[<x>]</code>	Return field <code>`user</code>
<code>urllib.password[<x>]</code>	Return field <code>`password</code>
<code>urllib.host[<x>]</code>	Return field <code>`host</code>
<code>urllib.port[<x>]</code>	Return field <code>`port</code>
<code>urllib.path[<x>]</code>	Return field <code>`path</code>
<code>urllib.query[<x>]</code>	Return field <code>`query</code>
<code>urllib.fragment[<x>]</code>	Return field <code>`fragment</code>
<code>urllib.update[<x>]</code>	Update URL dictionary with dictionary <code>x</code>
<code>urllib.test[]</code>	Run tests for all elements of the library
<code>urllib.examples[]</code>	Show use cases

xml.o

PEG-based XML parser that processes XML documents and converts them into nested dictionary structures. The parser handles XML elements, attributes, text content, and comments.

Usage

```
o)load[getenv[`OHOME`];"xml"];
```

Main elements

Name	Description / comments
<code>.xml.parsed[<xml string>]</code>	Parse XML string and return nested dictionary with element structure

Parsed structure

The parser returns a dictionary where each XML element contains:

- Element name as dictionary key (symbol)
- `children` - list of child elements or text nodes
- `attrs` - dictionary of element attributes (if present)
- `text` - text content (for text nodes)

Examples

Basic XML parsing:

```
o)load[getenv[`OHOME`];"xml"];  
o)  
o)// Simple XML document  
o)xml: "AndrewfriendHello";  
o)d: .xml.parsed[xml];  
o)  
o)// Navigate to specific element  
o)d . (`note; `children; 0; `to; `children)  
textl "Andrew"  
o)  
o)// Access body text  
o)d . (`note; `children; 2; `body; `children)  
textl "Hello"
```

XML with attributes:

```
o)// XML with element attributes
o)xml: "30";
o)d: .xml.parsed[xml];
o)
o)// Access attributes
o)d . (`person; `attrs)
id | "123"
name | "Alice"
o)
o)// Access child element
o)d . (`person; `children; 0; `age; `children)
text | "30"
```

Nested elements:

```
o)// Nested XML structure
o)xml: "
Important
Hello World";
o)d: .xml.parsed[xml];
o)
o)// Navigate to header
o)d . (`message; `children; 0; `header; `children)
text | "Important"
o)
o)// Navigate to body
o)d . (`message; `children; 1; `body; `children)
text | "Hello World"
```

Auxiliary elements

PEG parser grammar elements:

Name	Description / comments
<code>.xml.concat[<x>]</code>	Concatenate parsed results
<code>.xml.quotes[<x>]</code>	Parse quoted strings in attributes
<code>.xml.spaces[<x>]</code>	Parse and skip whitespace
<code>.xml.keyend[<x>]</code>	Parse end of element name
<code>.xml.nident[<x>]</code>	Parse XML identifier (element/attribute name)
<code>.xml.commnt[<x>]</code>	Parse XML comments (<code><!-- ... --></code>)
<code>.xml.keyval[<x>]</code>	Parse attribute key-value pairs
<code>.xml.attrs[<x>]</code>	Parse element attributes and return <code>`attrs`</code> dictionary
<code>.xml.pltext[<x>]</code>	Parse plain text content and return <code>`text`</code> field
<code>.xml.prolog[<x>]</code>	Parse XML prolog/declaration (<code><?xml ... ?></code>)
<code>.xml.lngtag[<x>]</code>	Parse long-form XML tags (<code><element>...</element></code>)
<code>.xml.inltag[<x>]</code>	Parse self-closing inline tags (<code><element /></code>)
<code>.xml.contnt[<x>]</code>	Parse element content (text or nested elements)

Plugins

In cases where new low-level functionality is needed, plugins can be used. Some plugins are included in the standard package, while others can be created independently.

The standard plugins are located in the `plugins/` folder.

Current set of standard plugins

Plugin	Description
<code>crypto</code>	Function for calculating cyclic redundancy check (CRC)
<code>ctrlc</code>	Reagent for SIGINT
<code>fs</code>	Functions for work with file system
<code>kdb</code>	Reagents with kdb+ IPC protocol
<code>serde</code>	Functions for serialization and deserialization

To use a particular plugin, you need to load it.

```
o)load "serde"  
"./plugins/serde/libserde.dylib"
```



To avoid problems, make sure that plugins are not downloaded multiple times.

 [load](#)

 [Customers plugins](#)

Custom plugins

The platform allows creating two types of plugins:

- stateless functions (monads, dyads, triads, tetrads, polyads);
- stateful reagents (async streams).

Both types are written in Rust, compiled as dynamic libraries (`.dylib` on macOS, `.so` on Linux), and loaded at runtime.

Prerequisites

Building plugins requires:

- **Rust nightly** toolchain
- **Platform source tree** — the `ext` crate and its transitive dependencies are needed at compile time

The `ext` crate depends on several internal crates, so the full kernel source tree is required:

```
kernel/  
  kernel/  
    ext/           # plugin interface  
    ops/          # operations  
    vectorize/    # proc-macros ([monad], [dyad])  
    o/ffi/        # FFI types (AST, Interpreter API)  
    rt/           # runtime (basert, utilsrt, sync)
```



Without access to the platform source code, plugins cannot be compiled.

Stateless function plugin

Let's create a plugin that registers a custom `add` dyad function.

Project setup

Create a new Rust library:

```
cargo new --lib my_plugin
```

Configure `Cargo.toml`:

```
[package]
name = "my_plugin"
version = "0.1.0"
edition = "2021"

[dependencies]
ext = { path = "/kernel/ext" }

[lib]
crate-type = ["cdylib"]
```



The `ext` crate path should point to `kernel/ext` directory inside the platform source tree. For example, if the platform source is at `/home/user/theplatform`, then use `ext = { path = "/home/user/theplatform/kernel/ext" }`.

Plugin code

Create `src/lib.rs`:

```
#![feature(abi_vectorcall)]
#![feature(allocator_api)]
#![feature(c_unwind)]

extern crate ext;
use ext::*;

#[dyad]
fn add_fn(lhs: &AST, rhs: &AST, i: &Interpreter) -> AST {
    match (lhs.tp(), rhs.tp()) {
        (SC_LONG, SC_LONG) => {
            i.new_scalar_long(lhs.long() + rhs.long())
        }
        _ => panic!("nyi")
    }
}

declare_plugin!(
    // constructor: called when plugin is loaded
    |i: &Interpreter| {
        let sym = i.intern_string("add");
        i.insert_entity(sym, i.new_verb_dyad(add_fn));
        i.new_string("add registered")
    },
    // destructor: called when plugin is unloaded
    || {}
);
```

Key elements:

- `#[dyad]` macro marks the function as a dyadic verb (2 arguments). Use `#[monad]` for monadic (1 argument).
- `declare_plugin!` macro generates the plugin entry points. The constructor receives an `Interpreter` reference and registers functions.
- `i.intern_string("add")` creates a symbol for the function name.
- `i.insert_entity(sym, i.new_verb_dyad(add_fn))` binds the symbol to the function.

Building

Requires Rust nightly toolchain:

```
cargo build --release
```

The compiled plugin will be at [target/release/libmy_plugin.dylib](#) (macOS) or [target/release/libmy_plugin.so](#) (Linux).

Usage

```
o)load "/path/to/libmy_plugin.dylib";
"add registered"
o)add[1;2]
3
o)add[10;20]
30
o)add[100;200]
300
```

Available function arities

Macro	Arity	Registration	Example call
<code>`#[monad]`</code>	1	<code>`new_verb_monad(f)`</code>	<code>`f[x]`</code>
<code>`#[dyad]`</code>	2	<code>`new_verb_dyad(f)`</code>	<code>`f[x;y]`</code>

Interpreter API

The [Interpreter](#) reference provides methods for creating and inspecting values:

Constructors:

Method	Description
<code>`new_scalar_long(v)`</code>	Create long scalar
<code>`new_string(s)`</code>	Create string
<code>`new_vector_long(iter)`</code>	Create long vector from iterator
<code>`new_dict(keys, vals)`</code>	Create dictionary
<code>`new_signal(msg)`</code>	Create error signal
<code>`new_verb_monad(f)`</code>	Wrap function as monad
<code>`new_verb_dyad(f)`</code>	Wrap function as dyad

Accessors:

Method	Description
<code>`tp()`</code>	Get value type id
<code>`long()`</code>	Extract long value
<code>`float()`</code>	Extract float value
<code>`bool()`</code>	Extract bool value
<code>`string()`</code>	Extract string slice

Type constants:

Constant	Type
<code>`SC_LONG`</code>	Scalar long
<code>`SC_FLOAT`</code>	Scalar float
<code>`SC_BOOL`</code>	Scalar bool
<code>`VEC_LONG`</code>	Vector of longs
<code>`VEC_CHAR`</code>	String (vector of chars)

Using built-in verbs

Plugins can call existing platform verbs:

```
#[dyad]
fn add_fn(lhs: &AST, rhs: &AST, i: &Interpreter) -> AST {
  match (lhs.tp(), rhs.tp()) {
    (SC_LONG, SC_LONG) => {
      let dyad = i.get_verb_dyad("+").unwrap().dyad();
      unsafe { dyad(lhs, rhs, i) }
    }
    _ => panic!("nyi")
  }
}
```

Stateful reagent plugin

Reagent plugins implement async I/O streams. They consist of a `Sender` (sink) and `Receiver` (stream).

Project setup

Same as function plugin, but also depends on `rt` crate:

```
[dependencies]
ext = { path = "/kernel/ext" }
rt = { path = "/kernel/o/rt" }
```

Plugin code

```

#![feature(allocator_api)]
#![feature(abi_vectorcall)]
#![feature(c_unwind)]

extern crate ext;
extern crate rt;

use core::pin::Pin;
use ext::*;
use std::io;
use std::task::{Context, Poll};

pub struct Receiver;

#[derive(Clone)]
pub struct Sender;

impl Sink for Sender {
    type Error = io::Error;
    fn poll_ready(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll { self.poll_flush(cx) }
    fn start_send(self: Pin<&mut Self>, item: AST)
        -> Result<(), Self::Error> {
        let interp = rt::task::local::interpreter::().unwrap();
        let mut s = String::new();
        interp.format(&mut s, &item);
        println!("reagent send: {}", s);
        Ok(())
    }
    fn poll_flush(self: Pin<&mut Self>, _cx: &mut Context<'_>)
        -> Poll { Poll::Ready(Ok(())) }
    fn poll_close(self: Pin<&mut Self>, _cx: &mut Context<'_>)
        -> Poll { Poll::Ready(Ok(())) }
}

impl Meta for Sender {
    fn meta(&self) -> Option {
        MetaInfo::builder().entry("type", "my-reagent").finish()
    }
}

impl SinkReagentExt for Sender {
    fn boxed(&self) -> Box + Send + 'static {
        Box::new(self.clone())
    }
}

impl Stream for Receiver {
    type Item = Result;
    fn poll_next(self: Pin<&mut Self>, _cx: &mut Context<'_>)
        -> Poll {
        let interp = rt::task::local::interpreter::().unwrap();
        Poll::Ready(Some(Ok(interp.new_string("hello from reagent"))))
    }
}

impl StreamReagentExt for Receiver {}

pub extern "C-unwind" fn new_reagent(
    _: &[AST], _: &Interpreter

```

```
) -> io::Result<(SinkReagent, StreamReagent)> {
    let rx = Box::new(Receiver);
    let tx = Box::new(Sender);
    Ok((tx, rx))
}

declare_plugin!(
    li: &InterpreterI {
        i.register_reagent("myreagent", new_reagent);
        i.new_string("reagent registered")
    },
    || {}
);
```

Traits to implement

Trait	Purpose
<code>`Sink`</code>	Sending data to the reagent
<code>`SinkReagentExt`</code>	Cloning for the sink
<code>`Stream`</code>	Receiving data from the reagent
<code>`StreamReagentExt`</code>	Extension trait for the stream
<code>`Meta`</code>	Metadata for introspection

Usage

```
o)load "/path/to/libmy_reagent.dylib";
"reagent registered"
o)r: reagent[`myreagent];
o)r["test"];
reagent send: "test"
o)react {[x:r] println[x]};
```

 [load](#)

 [reagent](#)

 [react](#)

Plugin crypto

At the moment, only the function for calculating cyclic redundancy check (CRC)

Content

Function	Description / comments
<code>crc32[<string> <vector bytes>]</code>	Return a CRC32 of byte vector or string

Examples:

```
o)load ["crypto"];
o)crc32 0x010203040506
-2114554076i
o)crc32 "Hello, World!"
-330644528i
```

Plugin ctrlc

SIGINT processing reagent.

Content

Reagent	Description / comments
reagent[`ctrlc]	Reagent for catching SIGINT

Examples:

```
o)load "ctrlc";
o)rINT: reagent[`ctrlc];
o)// reaction for Ctrl-C
o)react {[x:rINT] 0N!"I'll be back" };
o)
```

Plugin fs

Functions for work with file system

Content

Function	Description / comments
fs_create[<symbol name>]	Create file or dir
fs_remove[<symbol name>]	Remove file or dir
fs_rename[<symbol name>; <symbol name>]	Rename file or dir
fs_copy[<symbol name>; <symbol name>]	Copy file or dir
fs_exists[<symbol name>]	Return 1b if file exists return

Examples:

```
o)load "fs";
o)fs_create `:tmp/;
o)fs_exists `:tmp/
1b
o)fs_rename[`:tmp/;`tm/];
o)fs_exists `:tm/
1b
o)fs_remove `:tm/;
o)fs_exists `:tm/
0b
o)
```

Plugin kdb

Reagents with kdb+ IPC protocol

Content

Reagent	Description / comments
<code>reagent[`kdb_listener]</code>	Reagent listener specific for kdb+ ipc protocol
<code>reagent[`kdb;"addr:port"]</code>	Reagent for kdb+ ipc protocol (client side)

For examples see:

 [kdb listener](#)

 [kdb \(client\)](#)

Plugin serde

Functions for serialization and deserialization

Content

Function	Description / comments
<code>ser[<format>;<ATS>]</code>	Return serialized AST
<code>de[<format>;<string> <vector bytes>]</code>	Return deserialized data

Examples:

```
o)load "serde";
o)d: `a`b!(1 2; 3 4);
o)sd: ser[`json; d]
"{\"a\": [1,2], \"b\": [3,4]}"
o)de[`json; sd]
a| 1 2
b| 3 4
o)de[`yaml; ser[`yaml; d]]
a| 1 2
b| 3 4
o)de[`toml; ser[`toml; d]]
a| 1 2
b| 3 4
o)
o)de[`fix; "8=FIX.4.4\\x019=\\x00\\x01"]
8| "FIX.4.4"
9| "\\x00"
o)de[`fix; 0x383d4649582e342e3201393d303330390133353d3801]
8 | "FIX.4.2"
9 | "0309"
35| "8"
o)
```

ThePlatform for Python developers

Here you can find Platform solutions to some basic problems along with their Python solutions.

Factorial of a number

```
>>> def factorial(n): return 1 if (n==1 or n==0) else n * factorial(n - 1)
...
>>> factorial(6)
720
```

```
o)factorial:{$[x<2;1;x*o x-1]}
{$[x<2;1;x*o x-1]}
o)factorial 6
720
o)
```



binding is special in lambdas body. It defines reference to enclosing lambda itself.

To define factorial of non-recursively, multiply natural numbers from 1 to .

```
o)fact:{* / 1+til x}
{* / 1+til x}
o)fact 6
720
o)
```

Simple interest

```
>>> p=1000           # principal
>>> r=3              # rate
>>> t=5              # time periods
>>> (p*r*t)/100      # simple interest
150.0
```

```
o)p:1000          // principal
1000
o)r:3            // rate
3
o)t:5           // time periods
5
o)(p*r*t)%100    // simple interest
150
o)
```

A better practice is using vectors:

```
o)(*/(1000 3 5))%100
150
o)
```

The same applies to multiple values since most operators in O are of implicit iteration:

```
o)p:1000 2500 3000 // principal
1000 2500 3000
o)r:3 4 5          // rate
3 4 5
o)t:10 5 10       // time periods
10 5 10
o)(*/(r;t;p))%100 // simple interest
300 500 1500
o)
```

Compound interest

```
>>> p = 1300      # principal
>>> r = 4.8       # rate
>>> t = 3         # time periods
>>> p*(pow((1+r/100),t)) # compound interest
1496.3293696
```

```
o)p:1300f
1300f
o)r:4.8
4.8
o)t:3f
3f
o)p*(1f+r%100.0) xexp t
1496.3293696
o)
```



All numbers in expressions must be of the same type.

Works with lists as well:

```
o)p:1300 1500 2000f
1300 1500 2000f
o)r:4.8 5.0 5.5
4.8 5 5.5
o)t:3 4 5f
3 4 5f
o)p*(1f+r%100.0) xexp t
1496.3293696 1823.2593750000003 2613.9200128187495
o)
```

Area of a circle

The area of circle is equal to πr^2 , where r is a circle radius and π is the arc-cosine of -1.

```
>>> import numpy as np
>>> np.arccos(-1)*10*10      # area of circle of radius 10
314.1592653589793
```

```
o)r:10f                // radius
10f
o)area:(acos -1f)*r*r   // area of circle of radius 10
314.1592653589793
o)
```

Prime numbers in an interval

```
>>> from sympy import sieve
>>> list(sieve.primerange(20, 40))
[23, 29, 31, 37]
```

There is no built-in function for identifying prime numbers in O:

```
range:{x+til y-x-1};

primeinrange:{                // in REPL this code should be written in one line
  l:range[x;y];                // list of potential prime numbers
  lmt:"j"$sqrt "f"$last l;     // highest divisor to test
  l where (&/(<0<l mod/:range[2;lmt])) };
```



```
# next Fibonacci pair
def nfp(x): return [x[1], sum(x)]

# Nth Fibonacci pair
def fibp(n):
    if n<2: return [0, 1]
    return nfp(fibp(n-1))

def fib(n): return fibp(n)[0]
```

```
>>> fib(10)
34
```

```
o) nfp: {(x 1), +/x};
o) fib: {first(x-1)nfp/0 1};
```

```
o) fib 10
34
o)
```

Whether a Fibonacci number

```
import math
def is_fibonacci(n):
    phi = 0.5 + 0.5 * math.sqrt(5.0)
    a = phi * n
    return n == 0 or abs(round(a) - a) < 1.0 / n
```

```
>>> [is_fibonacci(x) for x in (5, 13, 20)]
[True, True, False]
```

x is a Fibonacci number if $5x^2 + or - 4$ is a perfect square:

```
o) is_ps: {x={x*x}"j"$sqrt "f"$x} // checks if number is a perfect square
{x={x*x}"j"$sqrt "f"$x}
o) is_fibonacci: {l/is_ps (5*x*x)+ '4 -4}
{l/is_ps (5*x*x)+ '4 -4}
o) is_fibonacci each 5 13 20
110b
o)
```

Sum of squares of first N numbers

```
def squaresum(n): return (n * (n + 1) / 2) * (2 * n + 1) / 3
```

```
>>> [squaresum(x) for x in (1,2,3,4,5,6,7,8,9,10)]  
[1.0 5.0 14.0 30.0 55.0 91.0 140.0 204.0 285.0 385.0]
```

```
o)squaresum:{x:"f"$x;(x*(x+1.0)%2.0)*(1.0+x*2.0)%3.0}  
{x:"f"$x;(x*(x+1.0)%2.0)*(1.0+x*2.0)%3.0}  
o)squaresum 1+til 10  
1 5 14 30 55 91 140 204 285 385f  
o)
```

Cube sum of first N natural numbers

```
def sum_cubes(x): return (x * (x + 1) // 2) ** 2
```

```
>>> [sum_cubes(x) for x in (5, 7)]  
[225, 784]
```

```
o)sum_cubes:{("f"$x*(x+1)%2)xexp 2f}  
{("f"$x*(x+1)%2)xexp 2f}  
o)sum_cubes 8 19  
1024 36100f  
o)
```

ThePlatform for KDB+ developers

This section contains both Platform and KDB+ solutions to some basic problems.

Factorial of a number

```
q)factorial:{$[x<2;1;x*.z.s x-1]}
q)factorial 6
720
q)
```

```
o)factorial:{$[x<2;1;x*o x-1]}
{$[x<2;1;x*o x-1]}
o)factorial 6
720
o)
```



binding is special in lambdas body. It defines reference to enclosing lambda itself.

Non-recursive solution:

```
q)prd 1+til 6
720
q)
```

```
o)*/ 1+til 6
720
o)
```

Simple interest

```
q)p:1000           // principal
q)r:3             // rate
q)t:5             // time periods
q)(p*r*t)%100     // simple interest
150f
q)
```

```

o)p:1000          // principal
1000
o)r:3            // rate
3
o)t:5           // time periods
5
o)(p*r*t)%100    // simple interest
150
o)

```

Better practice is using vectors:

```

q)(prd 1000 3 5)%100
150f
q)

```

```

o)(*/(1000 3 5))%100
150
o)

```

The same applies to multiple values due to implicit iteration:

```

o)p:1000 2500 3000 // principal
1000 2500 3000
o)r:3 4 5          // rate
3 4 5
o)t:10 5 10        // time periods
10 5 10
o)(*/(r;t;p))%100  // simple interest
300 500 1500
o)

```

Compound interest

```

q)p:1300          // principal
q)r:4.8           // rate
q)t:3             // time periods
q)p*(1+r%100)^exp t // compound interest
1496.3293696
q)

```

```
o)p:1300f
1300f
o)r:4.8
4.8
o)t:3f
3f
o)p*(1f+r%100.0)xexp t
1496.3293696
o)
```



All numbers in expressions in O must be of the same type.

Again, the same applies to lists:

```
o)p:1300 1500 2000f
1300 1500 2000f
o)r:4.8 5.0 5.5
4.8 5 5.5
o)t:3 4 5f
3 4 5f
o)p*(1f+r%100.0)xexp t
1496.3293696 1823.2593750000003 2613.9200128187495
o)
```

Area of a circle

The area of circle is equal to πr^2 , where r is a circle radius and π is the arc-cosine of -1.

```
q)r = 10
q)(acos -1)*r*r           // area of circle of radius 10
314.1592653589793
q)
```

```
o)r:10f                   // radius
10f
o)area:(acos -1f)*r*r     // area of circle of radius 10
314.1592653589793
o)
```

Prime numbers in an interval

```

q)show c:range[11;25] // candidates
11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
q)"j"$sqrt last c // need test modulo only to here
5
q)range[2;]"j"$sqrt last c
2 3 4 5

q)c mod/:2 3 4 5 // modulo each c against all of them
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
2 0 1 2 0 1 2 0 1 2 0 1 2 0 1
3 0 1 2 3 0 1 2 3 0 1 2 3 0 1
1 2 3 4 0 1 2 3 4 0 1 2 3 4 0

q)show f:0<c mod/:2 3 4 5 // flag remainders
101010101010101b
101101101101101b
101110111011101b
111101111011110b
q)all f // AND the flag vectors
101000101000100b
q)where all f // index the hits
0 2 6 8 12
q)c where all f // select from range
11 13 17 19 23
q)

```

```
range:{x+til y-x-1}
```

```

primeinrange:{
  l:range[x;y]; // list of potential prime numbers
  lmt:"j"$sqrt "f"$last l; // highest divisor to test
  l where (&/(0<l mod/:range[2;lmt])) }

```

```

q)primeinrange[20;40]
23 29 31 37

```



```
q)fib 10
34
q)
```

```
o)nfp:{(x 1),+/x};
o)fib:{first(x-1)nfp/0 1};
o)fib 10
34
o)
```

Whether a Fibonacci number

x is a Fibonacci number if $5x^2 + or - 4$ is a perfect square:

```
is_ps:{x={x*x}"j"$sqrt x}; // is perfect square?
is_fibonacci:{.[or]is_ps flip 4 -4+/:5*x*x};
```

```
q)is_fibonacci 5 13 20
110b
q)
```

```
o)is_ps:{x={x*x}"j"$sqrt "f"$x} // checks if number is a perfect square
{x={x*x}"j"$sqrt "f"$x}
o)is_fibonacci:{|/is_ps (5*x*x)+'4 -4}
{|/is_ps (5*x*x)+'4 -4}
o)is_fibonacci each 5 13 20
110b
o)
```

Sum of squares of first N numbers

```
q)squaresum:{(x*(x+1)%2)*(1+x*2)%3}
q)squaresum 1+til 10
1 5 14 30 55 91 140 204 285 385f
q)
```

```
o)squaresum:{x:"f"$x;(x*(x+1.0)%2.0)*(1.0+x*2.0)%3.0}
{x:"f"$x;(x*(x+1.0)%2.0)*(1.0+x*2.0)%3.0}
o)squaresum 1+til 10
1 5 14 30 55 91 140 204 285 385f
o)
```



All numbers in expressions in O must be of the same type.

Cube sum of first N natural numbers

```
q)sum_cubes:{(x*(x+1)div 2)xexp 2}  
q)sum_cubes 8 19  
1024 36100f  
q)
```

```
o)sum_cubes:{"f"$x*(x+1)%2)xexp 2f}  
{"f"$x*(x+1)%2)xexp 2f}  
o)sum_cubes 8 19  
1024 36100f  
o)
```

Application examples

[Dining philosophers problem](#)

```
// The dining philosophers problem implementation in Join Calculus

// define channels (async molecules) for tasks communication
thinking1: reagent[`async]; hungry1: reagent[`async]; fork1: reagent[`async];
thinking2: reagent[`async]; hungry2: reagent[`async]; fork2: reagent[`async];
thinking3: reagent[`async]; hungry3: reagent[`async]; fork3: reagent[`async];
thinking4: reagent[`async]; hungry4: reagent[`async]; fork4: reagent[`async];
thinking5: reagent[`async]; hungry5: reagent[`async]; fork5: reagent[`async];

// define reactions
react[{{x:thinking1} spawn[{{println "Socrates is thinking"; hungry1[x]};x}}];
react[{{x:thinking2} spawn[{{println "Confucius is thinking"; hungry2[x]};x}}];
react[{{x:thinking3} spawn[{{println "Plato is thinking"; hungry3[x]};x}}];
react[{{x:thinking4} spawn[{{println "Descartes is thinking"; hungry4[x]};x}}];
react[{{x:thinking5} spawn[{{println "Voltaire is thinking"; hungry5[x]};x}}];

react[{{x:hungry1;y:fork1;z:fork5} spawn[{{println["Socrates is eating #%";cnt-x]; fork1[];fork5[]; if [x] {thinking1[x]}}];
react[{{x:hungry2;y:fork2;z:fork1} spawn[{{println["Confucius is eating #%";cnt-x]; fork2[];fork1[]; if [x] {thinking2[x]}}];
react[{{x:hungry3;y:fork3;z:fork2} spawn[{{println["Plato is eating #%";cnt-x]; fork3[];fork2[]; if [x] {thinking3[x]}}];
react[{{x:hungry4;y:fork4;z:fork3} spawn[{{println["Descartes is eating #%";cnt-x]; fork4[];fork3[]; if [x] {thinking4[x]}}];
react[{{x:hungry5;y:fork5;z:fork4} spawn[{{println["Voltaire is eating #%";cnt-x]; fork5[];fork4[]; if [x] {thinking5[x]}}];

// spawn initial data
cnt:3; //count times
thinking1[cnt];thinking2[cnt];thinking3[cnt];thinking4[cnt];thinking5[cnt];
fork1[];fork2[];fork3[];fork4[];fork5[];

Socrates is thinking
Confucius is thinking
Plato is thinking
Descartes is thinking
Voltaire is thinking
Socrates is eating #1
Plato is eating #1
Socrates is thinking
Plato is thinking
Socrates is eating #2
Plato is eating #2
Socrates is thinking
Voltaire is eating #1
Confucius is eating #1
Plato is thinking
Descartes is eating #1
Voltaire is thinking
Socrates is eating #3
Confucius is thinking
Plato is eating #3
Descartes is thinking
Voltaire is eating #2
Confucius is eating #2
Descartes is eating #2
Voltaire is thinking
Confucius is thinking
Descartes is thinking
Voltaire is eating #3
Confucius is eating #3
Descartes is eating #3
```

Frequently asked questions

How do I delete records from a table?

Don't do that. Seriously. Just select records you want to retain and replace original table.

I want to make my O script both executable and loadable and pass command-line parameters to it.

Use the following shebang under Linux. You need to use coreutils 8.30+ though.

```
#!/usr/bin/env -S bash -c 'tachyon -c 0 -f <(cat "$0" | sed "1,1d") -- $@'
```

Note, previously we used shebang `#!/usr/bin/env -S bash -c 'cat "$0" | sed "1,1d" | tachyon -- $@'`. However, it captured stdin forever and prevented e.g. our REPL from properly working.

Now it's possible to drop into working REPL with newer shebang like:

```
#!/usr/bin/env -S bash -c 'tachyon -c 0 -f <(cat "$0" | sed "1,1d") -- $@'

HOME: getenv[`OPATH`];
load HOME,"repl";
```

Debugging

How can I find memory leaks in tachyon binary?

Run tachyon using `$ RUSTFLAGS="-Z sanitizer=leak -Cforce-frame-pointers=yes" cargo run --bin tachyon`.

To get meaningful backtraces, you must have `llvm-symbolizer` accessible in `$PATH` (LLVM 8+).

If your Linux distribution installs it prepending its version, just make a symlink like:

```
$ mkdir -p ~/bin; ln -s /usr/bin/llvm-symbolizer-8 ~/bin/llvm-symbolizer; export PATH=$PATH:~/bin
```

Profiling

How to profile / find places to improve performance?

- For Linux, first install `perf` utility. For Debian-based distros (e.g. Ubuntu) execute:

```
$ sudo apt-get install linux-perf
```

- Build tachyon with debug information by editing `Cargo.toml` in root of source path. Uncomment `debug = true` under `[profile.release]` section and run:

```
$ cargo build --release --bin tachyon
```

- Prepare your test as script and run:

```
$ sudo perf record -g --call-graph=lbr target/release/tachyon -f script_to_profile.o
```

... by doing that you will record profile information which can be navigated using following:

```
$ sudo perf report --no-children
```

P.S. You should omit `--call-graph=lbr` in case your CPU is not Intel. That would lower profiling quality though.

Performance/network tuning

I am getting errors like "too many open files" and like that when opening many network connections. Any help?

For Linux, following limits should be increased:

- `/etc/sysctl.conf`

```
fs.file-max = 12000000
fs.nr_open = 12000000
```

For non-systemd based Linux systems, it is enough to change only limits.conf file.

- `/etc/security/limits.conf`

```
* hard nofile 12000000
* soft nofile 12000000
```

- current shell

```
$ ulimit -n 10000000
```

For systemd based Linux system, it is necessary to use systemd specific configuration files.

e.g. to increase maximum file limit for all services, use following & restart:

```
# mkdir -p /etc/systemd/system.conf.d/  
# cat >/etc/systemd/system.conf.d/10-filelimit.conf <<EOF  
[Manager]  
DefaultLimitNOFILE=12000000  
EOF
```

See <https://access.redhat.com/solutions/1257953> for details.

I am getting "allocation failed ..." errors and like that when spawning lots of lambdas. What should I do?

For Linux, the following limit should be increased:

```
/etc/sysctl.conf
```

```
vm.max_map_count = 12000000
```

I am getting errors like "cannot assign requested address (os error 99)" when making many connections to a single IP. Any help?

For Linux, the following limit should be increased:

```
/etc/sysctl.conf
```

```
net.ipv4.ip_local_port_range = 8192 65535
```

Contributors

- [Anton Kundenko](#) - Language architecture, internal functions, reagents.
- [Denis Golovan](#) - Query engine, optimisations, allocator, internal functions.
- [Ievhenii Lysichenko](#) - Allocator, sheduler, GUI expertise.
- [Serhii Savchuk](#) - Algorithms, lock-free structures, hashes, optimisations.
- [Viktor Sovietov](#) - Ideas bringer, motivator.

Contacts

Viktor Sovietov - viktor@lynxtrading.com

Anton Kundenko - anton@lynxtrading.com