Expressions

Graceful handling

You may have seen some strange uses of expressions in the docs so far, so this doc will clear up things quite a bit.
VAMP's expressions are geared towards "graceful handling" - which means it will not generate any failure unless the expression is malformed. This means your variables can have any data-type and your expression will hapily calculate it, either explicitly, or impliclitly; meaning that it will take care or things like: "devide by zero", or "negative zero", or "to the power of zero", etc, etc.

Expression Governors

Every expression in VAMP is preceded either by an "explicit" or "implicit" `governor`.
VAMP expressions have strict rules; however, an "expression governor" dictates how "the following expression" will operate.

For instance:

- "method-calls" are governed by the "caller" governor, which tests the validity of arguments, and handles synchronous returns & asynchronous call-backs
- "data-lookups" are governed by the "lookup" governor, which injects a data-source into an expression and guides the expression to skip "meta-aspects", create a result list and insert into it pointers to what the expression found
- "truth-tests" are governed by the "tester" governor, which injects data to be tested and runs the expression arguments recursively - checking the data accordingly

The above are just a few examples, but you get the idea. You can also define your own "expression governors", but we'll get to that later.

Arithmetic expressions

I'm sure you're keen to try out the raw power that VAMP offers with its data-type arithmetics.
The examples below are not complete lists of all possible cases, but merely indications of how things work in general.
Data-types other than numbers are sensibly handled as numerics and results are calculated implicitly.

Void & Zero

Void is numerically treated as `0` (absolute zero), else it is treated as "nothing".
Zero is numerically treated as `0` (absolute zero); so "negative zero" is `0`, and "positive zero" is `0`; the same applies to Void in terms of "polarity".

```
()                ::
().Kind           ::    Void

v:()              ::    "v" is "nothing" -or 0

------------------------------------------------------------

(v + v)           ::
(v - v)           ::
(v * v)           ::
(v / v)           ::
(v % v)           ::
(v ^ v)           ::

(v + 0)           ::    0
(v - 0)           ::    0
(v * 0)           ::    0
(v / 0)           ::    0
(v % 0)           ::    0
(v ^ 0)           ::    0

(0 + 0)           ::    0
(0 - 0)           ::    0
(0 * 0)           ::    0
(0 / 0)           ::    0
(0 % 0)           ::    0
(0 ^ 0)           ::    0

(v + 3)           ::    3
(v - 3)           ::    -3
(v * 3)           ::    0
(v / 3)           ::    0
(v % 3)           ::    0
(v ^ 3)           ::    0

(3 + v)           ::    3
(3 - v)           ::    3
(3 * v)           ::    0
(3 / v)           ::    3
(3 % v)           ::    3
(3 ^ v)           ::    0

(0 + 3)           ::    3
(0 - 3)           ::    -3
(0 * 3)           ::    0
(0 / 3)           ::    0
(0 % 3)           ::    0
(0 ^ 3)           ::    0
```

Spin

Spin is numerically treated as 1 of 3 values:

- `-` -1
- `?` 0
- `+` 1

The numbers to these values are "numeric barriers" of the `Spin`, from: `-1` to: `1`.
Any number less than `0` is `-`
Any number exactly `0` is `~`
Any number more than `0` is `+`

Before any calculation is done on the `Spin`, these numerics are processed as described above.

Here are some examples:

```
ns:-                 ::    NegaSpin
is:?                 ::    IffySpin
ps:+                 ::    PosiSpin

-----------------------------------------------------------

(-True)              ::    -
(?True)              ::    +
(+True)              ::    +

-----------------------------------------------------------

($ns)                 ::    -
($is)                 ::    ?
($ps)                 ::    +

-----------------------------------------------------------

($ps + $ns)            ::    ?
($ps + $is)            ::    +
($ps + $ps)            ::    +

($ps - $ns)            ::    +
($ps - $is)            ::    +
($ps - $ps)            ::    ?

($ps * $ns)            ::    -
($ps * $is)            ::    ?
($ps * $ps)            ::    +

-----------------------------------------------------------

($ns / 0)             ::    -
($is / 1)             ::    ?
($ps / 2)             ::    +

($ns % 0)             ::    ?
($is % 1)             ::    ?
($ps % 2)             ::    ?

($ns ^ 0)             ::    ?
($is ^ 1)             ::    ?
($ps ^ 2)             ::    +
```

Nume

Numerals are the easiest in VAMP expressions. Here are a couple of examples:

```
(-1)                 ::    -1
(0)                  ::    0
(1)                  ::    1

(-1)                 ::    -1
(+0)                 ::    0
(+1)                 ::    1

(-(1))               ::    -1
(?(1))               ::    +
(+(1))               ::    1

(-(-1))              ::    1
(?(-1))              ::    -
(+(-1))              ::    1

-----------------------------------------------------------

(1 + 1)              ::    2
(2 - 2)              ::    0
(3 * 3)              ::    9
(4 / 4)              ::    1
(5 % 5)              ::    0
(6 ^ 6)              ::    46656
```

Data

The following shows how TextData is handled:

```
txt: 'abcde'
olo: 'Hello World!'
say: `What's the deal with "all-THIS"?`

-----------------------------------------------------------

(-$txt)                 ::    edcba
(?$txt)                 ::    ?
(+$txt)                 ::    abcde

-----------------------------------------------------------

(a @ $txt)              ::    1
(e @ $txt)              ::    5
(z @ $txt)              ::    0

(1 @ $txt)              ::    a
(5 @ $txt)              ::    e
(9 @ $txt)              ::

($olo % o)              ::    Hell Wrld!

((2 <> -2) @ $txt)      ::    bcd
((b <> d) @ $txt)       ::    bcd

((2 >< -2) @ $txt)      ::    c
((b >< d) @ $txt)       ::    c

((a <> z) @ Data)       ::    abcdefghijklmnopqrstyvwxyz
((A <> Z) @ Data)       ::    ABCDEFGHIJKLMNOPQRSTYVWXYZ
((0 <> 9) @ Nume)       ::    0123456789

($say * %CASE:UC)       ::    WHAT'S THE DEAL WITH "ALL-THIS"?
($say * %CASE:LC)       ::    what's the deal with "all-this"?
($say * %CASE:CC)       ::    WhatSTheDealWithAllThis
($say * %CASE:CB)       ::    whatStheDealWithAllThis

-----------------------------------------------------------

($txt + fg)             ::    abcdefg
($txt - de)             ::    abc
($txt * $txt)           ::
($txt / c)              ::    [ab, de]
($txt % $txt)           ::    []
($txt ^ $txt)           ::

-----------------------------------------------------------

(-0 + $txt)             ::    abcde
(-1 + $txt)             ::    bcde
(-2 + $txt)             ::    cde

(-1 - $txt)             ::
(-1 - $txt).Type        ::    Void
```

Data - continued

```
($txt + 0)               ::    abcde0
($txt + 1)               ::    abcde1
($txt + 2)               ::    abcde2

($txt - 0)               ::    abcde
($txt - 1)               ::    abcd
($txt - 2)               ::    abc

($txt * 0)               ::
($txt * 1)               ::    abcde
($txt * 2)               ::    [abcde,abcde]

($txt / 0)               ::    [abcde]
($txt / 1)               ::    [a,b,c,d,e]
($txt / 2)               ::    [ab,cd,e]

($txt % 0)               ::    [abcde]
($txt % 1)               ::    []
($txt % 2)               ::    [e]

($txt ^ 0)               ::
($txt ^ 1)               ::    abcde
($txt ^ 2)               ::    abcde
```

Time

When you call `Time()`, a Unix timestamp is returned with micro-seconds as a `FracNume`.
If you want a modified string from the Time-call instead, you can also call it with an option-string like:`%YMD`.
You can also get a formatted time-string from micro-timestamp with the `%TIME` modifier in expressions.
When you have your time-string you can modify it as you wish.

Thius also works the other way around: when you use a "correctly formatted time-string", it will produce a time-stamp.
When referring to "correctly formatted time-string", it simply means to use it as indicated below:

```
now: Time()
txt: 'Mar 8 2016 2:40 am'

-----------------------------------------------------------

$now                                ::    1457404805.999

($now * %TIME:Y)                    ::    2016
($now * %TIME:12h)                  ::    am

('2016-03-08 02:40:05' * %TIME)     ::    1457404805
(txt * %TIME)                       ::    1457404800

Time(%Y)                            ::    2016
Time(%YM)                           ::    2016-03
Time(%YMD)                          ::    2016-03-08
Time(%YMD:h)                        ::    2016-03-08 02
Time(%YMD:hm)                       ::    2016-03-08 02:40
Time(%YMD:hms)                      ::    2016-03-08 02:40:05
Time(%YMD:hmsn)                     ::    2016-03-08 02:40:05:999
Time(%YMD:hmsn:24h)                 ::    2016-03-08 02:40:05:999
Time(%YMD:hmsn:12h)                 ::    2016-03-08 2:40:05:999 am
Time(%YMD:hmsn:12h:dlz)             ::    2016-3-8 2:40:5:999 am

Time(%hmsn)                         ::    02:40:05:999
Time(%hms:wlz)                      ::    02:40:05        :: with leading zeros
Time(%hms:dlz)                      ::    2:40:5          :: drop leading zeros
Time(%hms:12h)                      ::    2:40:05 am

Time(txt)                           ::    1457404800
```

Lookup expressions

Lookup expressions are like "queries"; even so, they have ways to optimize searches.

```
ride: {color:blue, roof:none}

-----------------------------------------------------------

$ride(*e)                   ::    ['blue', 'none']

$ride(*:*e)                 ::    ['blue', 'none']

$ride(*:*e 2)               ::    ['blue', 'none']
$ride(*:*e 1)               ::    ['blue']

$ride(<< *:*e 1)            ::    ['blue']
$ride(>> *:*e 1)            ::    ['none']

$ride(1 *:*e 1)             ::    ['blue']
$ride(3 *:*e 1)             ::    ['none']
```