Operators

Operator categories

The operators in VAMP are categorized in terms of usage. These are:

- comment
- context
- delimit
- reference
- logical
- arithmetic

This document is for referenece purposes and will be covered in more detail later.

Comment Operators

Comment in VAMP is used both for documentation and developer reference. The porpose of all these comments is to have code look well explained, and has the ability to make `Markdown` -or HTML documentation from VAMP files. It is recommended to "minify" your code when publishing to a live environment as this speeds up loading & parsing, not only for VAMP, but with any other language or library commonly used.

These are the comment operators:

```
--       double-dash(+space)  block-comment, at line start, ends with `--` or `::(+newline)`
#        hashtag(+space)      line-comment, used anywhere
##       double-tag           line-comment, used anywhere
#!       shebang              line-comment, used at doc start
!#       bangshe              line-comment, used anywhere, or in docs as highligted tags
.:       secton-begin         line-comment, used for: `section start`, or: `thus`
::       double-colon         line-comment, used anywhere, or section end
//       double-FWD-slash     line comment, used anywhere, or section start
\\       double-BCK-slash     line comment, used anywhere, or section end
---      triple-dash          line-comment, used for code -or comment-block separation
/* */    slash-star pair      block-comment, used anywhere
```

"Line-comment" is typically ended with a `newline` character, but "block-comment" is typically ended with a pairing sequence, as shown above.

The `bangshe` (or "banshee") line-comment can be used in both coding & documentation as it has the same effect, wich is to highlight certain key-words and let the accompanying text stand out from the rest. The "accompanying text" can be whatever the developer types in there; the "quoted text" below is only an indication of what the key-word could mean; however, if no text is typed next to these key-words, then the quoted text below will be used instead.

The words & colours of the "banshee comment" can be configured, but the defaults are:

- `TODO` - dark grey - "to be done in the future"
- `TEST` - green - "not thouroghly tested yet"
- `HELP` - pink - "i need help with this"
- `NEED` - purple - "the specification -or technology needs to be updated"
- `DONE` - blue - "completed and tested"
- `BUSY` - orange - "work in progress"
- `WARN` - orange - "it could work but may fail under specific circumstances"
- `HACK` - orange - "working but not properly done and may have side effects"
- `FAIL` - red - "broken, fix this -or remove it to prevent issues"
- `VOID` - black - "deprecated - do not use in the future"

Below is an example of how to use these comments resulting in well documented code.

```
#!/usr/bin/vamp


----------------------
::      v0.1.0      ::
----------------------



## Heading
--
   Documentation paragraph with some **bold** text
--
!# TODO : not implemented yet



.: bgn section :.
-- ------------------------------------------------------------
   documentaion stuff
-- ------------------------------------------------------------
   "Hello world!"
-- ------------------------------------------------------------
:: end section ::



// begin section -- quick, but not so pretty as above
------------------------------------------------------------
"Hello world!"
------------------------------------------------------------
\\ end section
```

Context Operators

The importance of "context" in VAMP is paramount as it defines what any statement -or expression means or implies and to what it applies - and in which scope it can be valid. Comment operators also define a "context" but the contents inside the "comment context" is ignored during runtime.
Your project structure in folders and files define context (scope), as well as the following operators:

```
( )   ::    wrap, express, calculate, call
[ ]   ::    wrap, list, contain
{ }   ::    wrap, describe
" "   ::    text, quoted - unformatted plain text - single-line
' '   ::    text, quoted - unformatted plain text - single-line
` `   ::    text, quoted - formatted text, multi-line, embedded values & expressions
```

Delimit Operators

As with any (normal) language, "spaces" devide "randomtext" into "some words". The same applies here, but just with different punctuation. As with some languages, like "Python", VAMP does not completely ignore "white-space". White-space delimit text into lists & statements where appropriate.
List and statement delimitation does not apply to "quoted text", except when this text is to be parsed.

From the VAMP interpreter's perspective: any comment is ignored and the white-space at the beginning and ending of any context (except quoted text) is removed. Any white-space remaining is reduced to a single white-space character. This is important because in VAMP a white-space character is a delimiter.

VAMP does not require commas -or semi-colons between list items and statements - ONLY if these are separated by white-space. Examples will follow later, but for now, remember that VAMP compiles (or transpiles) into other forms where the "implied" items & statements are realized and treated accordingly. When the compiler minifies VAMP code it fixes the delimiters accordingly.

Speaking of lists; "text" is a list of characters (a character "string"), hence list operations are natural to text.
This brings us to: `indexing`.

Indexing in VAMP starts with 1 (NOT 0) and applies to reverse lookup also. This solves a lot of common problems - especially in boolean truth comparisson where "something found" appears first in the list, -or the first item from the end of a list (or text).
Having to do extra calculations when coding just to compensate for that zero'th key, or to calculate the length of a list just to get to the last item - is insane, and, negative zero is a bit crazy, so, it's important to remember as it will save you a lot of headache.

The following delimiters are valid:

```
,     ::    comma
;     ::    semi-colon
      ::    white-space
```

Reference Operators

VAMP's reference operators together with the context-operators and inheritable meta-attributes are very powerful; you can do a lot more with less code. This does not nescessarily mean it is slow, because VAMP does not (normally) pass whole data structures around, but rather creates pointers to these structures in memory and passes the pointers around instead.

The following operators are used as reference operators:

```
:     ::    is   - name is value, where "value" is in the context of "name"
$     ::    self - aspects of an entity
@     ::    at   - point "at", or "in", to find or retrieve
.     ::    's   - as in: "Dog's balls"  in code:  Dog.balls
```

Here are a few examples of how to use the operators above:

```
foo:bar                   ::    {foo:'bar'}
bar:foo                   ::    {bar:'foo'}

bar:$foo                  ::    {bar:'bar'}
bar:$.foo                 ::    {bar:'bar'}

('foo'):('bar')           ::    {foo:'bar'}

(foo.Name + 'Var'):bar    ::    {fooVar:'bar'}
```

Logical Operators

In VAMP, "logical operators" & "arithmetic operators" must be delimited from text by white-space; else they are just plain text.
Remember that VAMP expressions are always evaluated from left to right, no exceptions (not even multiply). If you need part of an expression to take precedence you need to enclose it inside (any) context operators.

Truth in VAMP is calculated logically. Since we need not worry about "zero'th" keys it is safe to assume that the following is logically false: `undefined`,`null`,`0`,`empty`,`error`,`false` - anything else is logically true.
We will get to the "data types" in a bit; the "logically false" references above are not part of the data-types, so you don't need to remember them, just that those kinds of things are "logically false" and anything else is logically true".

Here are the logical operators:

```
?     ::    IF    - boolean truth confirm
|     ::    OR    - choose the first boolean true comparable value, "this OR that OR bark"
&     ::    AND   - boolean comparisson used to affirm & compound expressions

!     ::    NOT   - boolean negation
=     ::    same  - strict boolean comparisson
~     ::    like  - loose boolean comparisson
<     ::    less  - is true if "left of" is logically less than "right of", else false
>     ::    more  - is true if "left of" is logically more than "right of", else false
```

Some logical operators can be used together as compound operators.
The logical compound operators are:

- `!=` not same
- `!~` not like
- `>=` more same
- `<=` less same

Arithmetic Operators

In addition to "normal" arithmetic operation, VAMP also uses arithmetic to do data manipulation on various data-types. This saves a lot of time & duplication of function names. More on this later.

These are the arithmetic operators:

```
+     ::    add
-     ::    subtract
*     ::    multiply
/     ::    divide
%     ::    modulus
^     ::    exponent
```

VAMP is extremely versatile as a language, hence it has some rules of what belongs where.
The logical and arithmetic operators do not have any calculation value outside of "expression context", so, they MUST be inside expression context in order to be "calculated" (evaluated) accordingly.

When logcal -or arithmetic operators are used outside of expression context, then it is plain text.

Take note that VAMP requires white-space between operators in "expression context" -with only 4 exceptions to this rule, the following can be used directly in front of anything, -except logical, reference & arithmetic operators:

- `!` boolean negative-truth test
- `?` negative number
- `-` negative number
- `+` positive umber