Lenguaje común para condiciones en Vortex

Este documento intenta definir las operaciones y funciones que son comunes en todas la implementaciones de vortex, de manera que la semántica de las operaciones sea equivalente en cualquier entorno donde se ejecute.   
Las operaciones que pueden mantener la semántica entre distintas implementaciones de vortex pueden ser utilizadas como lenguaje común para filtrar los mensajes en el medio, y no sólo al final. Lo que optimizaría el ruteo de mensajes.

# Niveles de implementación

Como probablemente resulte muy difícil implementar todas las operaciones en todos los lenguajes y en todos los entornos de ejecución, definimos niveles de implementación para tratar de asegurar que algunas operaciones mínimas estén disponibles como conjunto de operaciones.  
Las implementaciones podrán entonces indicar que cumplen con la totalidad de un nivel, y quizás parcialmente otro, pero un nivel siempre incluirá la totalidad de operaciones del nivel inferior.

Nivel 0 – Elementales

El nivel más básico debería ser implementable en entornos con capacidad de procesamiento reducida como un arduino. En tales dispositivos donde el lenguaje también es simple. Las operaciones más básicas deberían ser utilizablesm y el resto ignoradas. Tomando como resultado de su evaluación un valor que pase la condición. Lo que permite que el mensaje se finalmente filtrado en destino, en el peor de los casos

Nivel 1 – Estándar

El nivel 1 sería el nivel exigible en entornos tipo PC donde el lenguaje de programación tiene la mayoría de los operadores como acciones nativas. Este nivel sería el deseable en cualquier implementación de vortex

Nivel 2 – Extensiones

El nivel 2 agrega algunas operaciones que son redundantes tal vez utilizando operaciones más básicas, pero que permiten mayor expresividad. Este nivel sería deseable en implementaciones que busquen ofrecer toda la potencialidad al momento de evaluar las condiciones

# Semántica adaptable

Debido a que en el camino de un mensaje pueden existir distintos niveles de implementación de los operadores y funciones, cada implementor debe fallar de manera que el mensaje llegue a destino.

Cualquier nodo que evalúe condiciones por otro nodo debe asegurarse de entregar el mensaje si no “entiende” una parte, o no tiene definidos los operadores necesarios para interpretarlo.  
Dicho de otra manera, a menos que sea posible determinar la falsedad de la condición sobre un mensaje, se debe adoptar true como respuesta de toda la condición.

Ejemplos

Tomemos como ejemplo un nodo muy simple que sólo puede evaluar el equals “=”.  
A continuación se muestran distintas condiciones que el nodo podrá evaluar o no, y qué acción debe tomar

**False determinable**

Si es posible determinar que una de las ramas es falsa e invalida la condición, entonces se puede definir el valor de verdad para el conjunto como false

* **AND**
  + **=**
    - **2**
    - **3**
  + contains
    - value\_of
      * “propiedad1”
    - “a”

Sabiendo que 2 != 3, se puede determinar que el AND es false, y que no importa el valor de la otra rama del AND, la condición evaluará **false**.

* **AND = false**
  + **false**
  + **Indecidible**

**True determinable**

Si una de las ramas es decidible y no requiere la evaluación de la otra, entonces puede tomarse su valor de verdad

* **OR**
  + **=**
    - **2**
    - **2**
  + contains
    - value\_of
      * “propiedad1”
    - “a”

Como 2 = 2, y está dentro de un OR, ya podemos determinar que la condición evaluará a true

* **OR = true**
  + **true**
  + **indecidible**

**Valor no determinable**

Si la evaluación de la condición depende de una rama no evaluable, entonces se deberá considerar como mensaje aceptable

* **OR**
  + **=**
    - **2**
    - **3**
  + contains
    - value\_of
      * “propiedad1”
    - “a”

Sabiendo que 2 != 3, no podemos determinar el valor de verdad para el OR. Como la otra rama del or es indecidible, Adoptamos true como valor de verdad de la condición

* **OR = indecidible => true**
  + **false**
  + **indecidible**

**Error de evaluación**

Si existe un error en la formulación de la condición, o se produce un error al aplicar una función indebida, la condición deberá ser tomada como true para que el mensaje llegue a destino.  
Probablemente en destino el mensaje debería producir un error (pero debe llegar de todas formas).

* OR
  + >
    - “hola”
    - 2
  + 5

Como la rama de la comparación “>” no es aplicable a un string, debería producirse un error que hace indecidible el resultado. Por otro lado la rama del 5 no es evaluable a un booleano por lo que también queda como indecidible

* OR = indecidible => true
  + indecidible
  + indecidible

# Tipos de datos

En Vortex existen los mismos tipos de datos que en JSON. De hecho vortex toma los tipos de datos de JSON para ser completamente compatibles. Por lo tanto las operaciones que pueden realizarse sobre los datos dependerán de los tipos de datos.

Tipos de vortex

1. string: cadena de caracteres en formato Unicode. Representa un texto
2. number: número que puede tener fracción y exponente. Representa un número
3. object: una agrupación de valores con forma de mapa o dictionary. Es la manera de representar un objeto
4. array: Lista ordenada de cualquier valor. Representa una lista de objetos
5. true: constante booleana que representa el valor de verdad true
6. false: constante booleana que representa el valor de verdad false
7. null: constante que representa el valor nulo para propiedades

# Operadores y funciones por tipo de dato

Cada tipo de datos de Vortex tiene operaciones asociadas que son aplicables en ese tipo y puede ser un error utilizarlos en otro

## String

Operadores

* “=”: Indica si dos cadenas son iguales (case-sensitive)
* “~”: Indica si dos cadenas son iguales (case-insensitive)
* “starts\_with”: Indica si la cadena empieza con la porción indicada (case-sensitive)
* “ends\_with”: Indica si la cadena termina con la porción indicada (case-sensitive)
* “contains”: Indica si la cadena contiene la porción indicada (case-sensitive)
* “!=”: Indica si la cadena es distinta de la pasada
* “matches”: Indica si cadena es aceptadad por una regex
* “in”: Indica si la cadena se encuentra en alguno de los valores pasados (case-sensitive)
* “in~”: indica si la cadena se encuentra en alguno de los valores pasados (case-insensitive)

Funciones

* “size”: Devuelve la longitud de la cadena en caracteres
* “char\_at”: Devuelve el carácter de la cadena en el índice indicado como otra cadena
* “type”: Devuelve “string” para indicar el tipo
* “as\_number”: Intenta convertir la cadena a número (si falla el resultado es null)

## Number

Operadores

* “=”: Indica si este número es igual a otro
* “<”: Indica si el número es menor que otro
* “<=”: Indica si el número es menor o igual que otro
* “>=”: indica si el número es mayor o igual que otro
* “<”: indica si el número es mayor que otro
* “between”: Indica si el número se encuentra entre dos valores (inclusivos)
* “in”: Indica si este número es parte de un array pasado

Funciones

* “as\_string”: Genera un texto como representación del número (usando el formato decimal de USA)
* “type”: devuelve “number” para identificar el tipo

## Object

Operadores

* “has\_property”: Indica si tiene alguna propiedad con el nombre indicado (case-insensitive)
* “is\_null”: indica si la propiedad indicada tiene el valor null asociado
* “is\_true”: Indica si la propiedad tiene el valor true
* “is\_false”: Indica si la propiedad tiene el valor false
* “is\_empty”: Indica si el objeto no tiene propiedades internas

Funciones

* “value\_of”: Devulve el valor de una propiedad del objeto por nombre (case-insensitive)
* “keys”: Devuelve un array con los nombres de las propiedades
* “type”: Devuelve “object” para indicar que es un objeto

## Array

Operadores

* “is\_empty”: Indica si el array no tiene elementos
* “contains”: Indica si el array contiene un valor indicado

Funciones

* “value\_at”: Devuelve el valor del índice indicado
* “size”: Devuelve la cantidad de elementos de este array

## True/False

Operadores

* “=”: Indica si este valor es igual al otro booleano
* “and”: Indica si dos booleanos son true
* “or”: Indica si alguno de dos booleanos son true
* “and\*”: Indica si todos de varios booleanos son true
* “or\*”: Indica si alguno de varios booleanos es true

Funciones

* “!”: Devuelve el valor negado

## Null

Operadores

* “=”: Indica si este valor null es igual a otro

# Operaciones disponibles:

Esta sección lista todas las operaciones disponibles, indicando el nivel de implementación, y los tipos a los que es aplicable, además de describir los detalles de implementación de cada operación