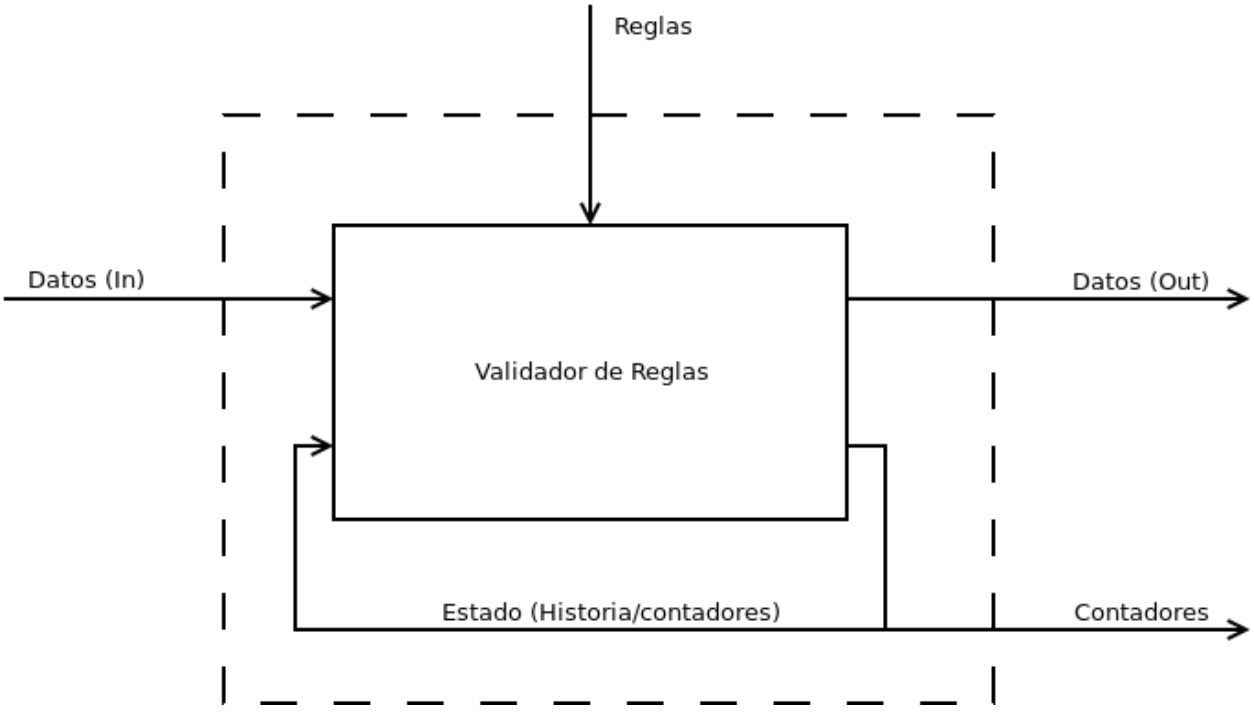
Trabajo Práctico N°1: Validador de Reglas

Enunciado

Se deberá implementar una herramienta que permita validar reglas configurables sobre una entrada de datos. Esta será utilizada para otros proyectos más adelante.



El motor recibe una secuencia de datos, y según el conjunto de reglas mantiene estadísticas y produce algún número finito de datos que genera como resultado.

Las reglas serán dadas por el cliente en un DSL (Domain Specific Language) descrito al final de este documento. Debe poder usarse cualquier regla válida en dicho lenguaje.

La operación futura del sistema implica procesar cantidades enormes de datos. No será posible almacenar o procesar la totalidad de los datos, por lo que deberá hallarse alguna forma de ignorar o resumir datos que no afecten las respuestas del sistema.

Funcionalidad a Implementar

Implementar la API provista de forma que se pueda realizar este procesamiento.

Objetivos

- Implementar una versión inicial de la herramienta.
- Aplicar las técnicas vistas en la teoría y en la práctica.
- Ganar conocimiento sobre Clojure.

Herramientas a utilizar

- Clojure
- Leiningen
- Git

Restricciones

- **Queda prohibido ejecutar una regla directamente como código.** Esto incluye el uso de eval, macros, y otros mecanismos similares.

Criterios de Corrección

- Cumplimiento de las restricciones
- Diseño del modelo
- Diseño del código
- Documentación entregada (informe, diagramas y tests)
- Correctitud
- Completitud
- Buen uso de las herramientas indicadas
- Uso de buenas prácticas en general

Entrega

- Hacer un fork del repositorio
- Crear un tag "entrega1.0" (Campo "Tag Version" en github) sobre el commit a entregar

Calendario

Jueves 22/3	Presentación del TP
Miercoles 11/4	Entrega TP, via GIT
Jueves 12/4	Validación y aceptación de la entrega en clase.

API del sistema

La API del sistema se basa en tres funciones que manipulan un valor que representa el estado actual del sistema:

- `initialize-processor [rules]`
  - Recibe una lista de reglas, cada una de las cuales se representa como una lista.
  - Retorna el estado de un sistema recién creado.

- `process-data [state new-data]`
  - Recibe el estado y un dato.
  - Retorna un vector `[new-state generated-data]` que contiene el nuevo estado y un vector generated-data. generated-data, a su vez, contiene todos los datos generados por reglas de tipo signal durante la evaluación de este dato en un orden arbitrario.

- `query-counter [state counter-name counter-args]`
  - Recibe el estado, el nombre de un contador y los argumentos a dicho contador.
  - Retorna el valor numérico almacenado en ese contador para esos argumentos.

El estado del sistema es un valor opaco e inmutable:

- El consumidor se limita a almacenar estados generados por esta API y usarlos como argumento para posteriores llamadas a esta API.
- No debe haber modificaciones a un estado una vez creado.

Lenguaje de reglas

Tipos básicos de valores

El sistema usa tres tipos básicos:

- String
- Bool
- Number

Los valores de tipo Number pueden incluir parte decimal. Ninguno de los tipos admite null (o similares) como valor válido.

Formato de datos

Cada dato es un diccionario con claves de tipo String y valores de tipo arbitrario.

```
{"nombre" "Pedro", "edad" 25, "suscripto" false}
```

Reglas

Cada regla define una posible reacción del sistema a los datos entrantes. Salvo indicación de lo contrario, todos los elementos en cada regla son requeridos, expresiones, y evalúan a un tipo específico. Para cada dato entrante, todas las reglas se verifican simultáneamente, antes de aplicar sus efectos.

Contador

```
(define-counter <nombre: String> [<parametro>*] <condicion: Bool>)
```

Define un contador que almacena un número para cada combinación diferente de parámetros. El contador inicia en 0, aumenta en 1 si se cumple la condición, mantiene su valor en caso contrario. En caso de hacer referencia a dato pasado, es a cualquiera que haga verdadera la condición, si lo hay.

`nombre` siempre es un String literal. Los parámetros son de cualquier tipo y puede haber cualquier cantidad de ellos.

Señal

```
(define-signal <resultado: Dato> <condicion: Bool>)
```

Define una salida de datos. En caso que se cumpla la condición, el sistema emite el dato dado en resultado. En caso de hacer referencia a dato pasado, es a cualquiera que haga verdadera la condición, si lo hay.

Expresiones

Una expresión puede ser:

- Un valor literal de un tipo básico
- Una función aplicada a otras expresiones
- Una referencia a un campo
- Una referencia a un contador

Para las dos primeras, se usa notación normal de clojure. Para la referencia a campos, se usan:

```
(current <campo>)
(past <campo>)
```

En ambos casos, `campo` es un String literal. `current` referencia al dato que se recibió para causar esta evaluación, `past` referencia algún dato pasado específico, seleccionado según el tipo de regla.

La referencia a contador retorna el valor almacenado para los argumentos indicados:

```
(counter-value <nombre: String> [<argumento>*])
```

Las funciones que deben implementarse son:

Para todo valor:

```
=, !=
```

Para Bool:

```
or, and, not
```

Para Number:

```
+, -, *, /, mod
<, >, <=, >=
```

Para String:

```
concat
includes?, starts-with?, ends-with?
```

Estas funciones se comportan como las funciones de Clojure de su mismo nombre. La excepción es `concat`, que se comporta como la función Clojure `str`.

Errores

En caso de ocurrir un error durante la evaluación de una regla con una combinación de dato actual y pasado, se cancela esa evaluación, dejándola sin efecto. Esto no impacta sobre la evaluación de otras combinaciones de regla/dato actual/dato pasado.

Son errores:

- Referir a un dato pasado cuando no hay tal dato.
- Acceder a un campo que no existe en el dato referenciado.
- Aplicar una función a un tipo de dato incorrecto.
- Dividir por cero.

Ejemplos de reglas

Contar la cantidad de mail vistos:

```
(define-counter "email-count" []
      true)
```

Hallar cuántos mensajes son spam:

```
(define-counter "spam-count" []
      (current "spam"))
```

Mantener un indicador actualizado de qué fracción de mail es spam:

```
(define-signal {"spam-fraction" (/ (counter-value "spam-count" [])
                                   (counter-value "email-count" []))}
      true)
```

Contar la cantidad de emails reenviados por cada usuario:

```
(define-counter "resent-by" [(current "sender")]
      (and (= (current "sender")
              (past "receiver"))
           (includes? (past "subject")
                      (current "subject"))))
```

Obtener la tabla de contingencias de emails con adjuntos, marcados como spam, o importante:

```
(define-counter "attachment-spam-important-table" [
             (current "has-attachment")
             (current "spam")
             (current "important")]
      true)
```