Tutorial de CLIPS, parte 1: representación de un problema con hechos y reglas (sin clases ni instancias)

1. El problema

Vamos a intentar representar por medio de un sistema de producción el siguiente problema: Queremos poner 3 litros en una jarra inicialmente vacía, sabiendo que solo podemos añadir 1 litro o añadir 2 litros a la jarra, no hay más acciones disponibles.
Como vemos, nuestro problema involucra representar por un lado con hechos el estado de la jarra (el número de litros que contiene) y por otro las acciones como reglas (añadir 1 litro, añadir 2 litros).

2. Representación de los hechos

Podríamos representar la situación inicial de la jarra con un hecho no estructurado en nuestro fichero de hechos de la forma:

```
(deffacts jarra-simple-en-0
(litros-en-jarra 0)  )
```

O bien utilizar un hecho estructurado con una plantilla jarra y un atributo o campo (slot) litros de tipo entero con un valor por omisión (default) de 0 (aunque en este caso como la jarra solo tiene un atributo no estaría justificado pero lo vamos a hacer igualmente como ejercicio):

```
(deftemplate jarra
(slot litros
(type INTEGER)
(default 0)))
```

Una deftemplate como éste solo define la plantilla (la estructura) de los hechos de un tipo dado (la jarra), es como una declaración de variables en programación. Una plantilla por si sola no crea ningún hecho en CLIPS. Por tanto es necesario crear el hecho que representa el estado inicial (la jarra vacía) incluyendo en nuestro fichero de hechos tras el deftemplate :

```
(deffacts jarra-estrucurada-en-0
(jarra litros 0)  )
```

O bien:

```
(deffacts jarra-estrucurada-en-0
(jarra)  )
```

3. Representación de las transiciones como reglas

Una regla CLIPS consta de dos partes:
La parte izquierda, o condición (o LHS, “Left hand side”), que va antes del =>.
La parte derecha, o acción (o RHS, “Right hand side”), que va después del =>.
La parte izquierda de una regla comprueba si ciertos hechos están en la base de hechos.
En caso de que se cumpla la parte izquierda de la regla, se ejecuta la parte derecha de la regla.
Genéricamente, la parte derecha de una regla puede añadir, quitar o modificar hechos
.
Nuestro problema podía experimentar dos tipos de transiciones: añadir 1 litro a la jarra, o añadir 2 litros. Estas transiciones se pueden representar de una manera informal con nuestras propias palabras:
Si tenemos una jarra con x litros, entonces añadir 1 litro.
Si tenemos una jarra con x litros, entonces añadir 2 litros.
Y en la sintaxis de CLIPS a añadir en nuestro fichero de reglas correspondiente quedaría:

```
(defrule un-litro-mas
?jarra <- (jarra (litros ?l))
=>
(modify ?jarra (litros (+ ?l 1))))

(defrule dos-litros-mas
?jarra <- (jarra (litros ?l))
=>
(modify ?jarra (litros (+ ?l 2))))
```

Ambas reglas son muy similares, así que solo analizaremos la primera. En la parte izquierda de la regla llamada un-litro-mas hemos especificado una única condición:
(jarra (litros ?l))
Cuando CLIPS se encuentra esta condición, lo que hace es intentar equipararlas con los hechos en ese instante de ejecución. Si tuviéramos los hechos (hipotéticos):
(jarra (litros 1))
(jarra (litros 101))
(mesa (peso 30))
Solo los dos primeros se equipararían con la condición de la regla. El caracter ? en CLIPS equipara con cualquier cosa: un número, un literal, una cadena, etc.
Pero el elemento de condición que estábamos analizando no era (jarra (litros ?)), sino:
(jarra (litros ?l)). ?l actúa también como comodín equiparándose con cualquier cosa que aparezca en ese lugar, pero es que además es una variable CLIPS. Esta variable tomará el valor de lo que hubiera equiparado se podrá usar (referenciar) más adelante en la propia regla. Por ejemplo, con los hechos anteriores, la condición de nuestra regla equipararía con:
(jarra (litros 1))
(jarra (litros 101))
En el primer caso, la variable ?l contendría el número 1, y en el segundo caso contendría el número 101. El contenido de dichas variables solo tiene sentido dentro de la regla en que se equiparan.
Estábamos analizando la regla:

```
(defrule un-litro-mas
?jarra <- (jarra (litros ?l))
=>
(modify ?jarra (litros (+ ?l 1))))
```

que, recordemos, queríamos que representara la transición desde un estado en el que la jarra contenía un número ?l de litros a un estado en el que contendría ?l + 1 litros. Ahora podemos ver con más detalle el significado de la regla anterior:
1. Equiparamos la parte izquierda de la regla con el hecho jarra.
2. Modificamos esa jarra para que el valor de su slot litros contenga un litro más.
El segundo paso, la modificación, lo hacemos en la parte derecha de la regla con el código:
(modify ?jarra (litros (+ ?l 1)))
Pero para modificar un hecho de jarra que hemos equiparado en la parte izquierda, tendríamos que haberlo almacenado en una variable previamente en esa misma parte izquierda. Eso lo hacemos con ?jarra **<-** en:
?jarra <- (jarra (litros ?l))
De esta forma, ?jarra es también una variable CLIPS, pero a diferencia de ?l que contenía un valor, esta contiene un hecho entero, precisamente el que hemos equiparado con (jarra (litros ? l)). Ese es el que modificamos en la parte derecha con (modify ?jarra (litros (+ ?l 1))).
En resumen, si hubiéramos partido de la base de hechos:
(jarra (litros 0))
(que es la situación inicial), y se hubiera disparado la regla un-litro-mas, nos resultaría en la base de hechos:
(jarra (litros 1))

4. Representación del fin de la ejecución

Además, podemos añadir una regla a la base de reglas que nos indique cuando el sistema de producción ha alcanzado el estado deseado o final (en nuestro caso, que la jarra contenga 3 litros). Esa regla se escribiría como:

```
(defrule acabar
(jarra (litros 3))
=>
(printout t "Lo he conseguido" crlf) (halt))
```

Esta regla equipararía solo cuando en la base de hechos haya un hecho como (jarra (litros 3)) y sólo en el caso de que aparezca el valor 3 en el campo litros de algún hecho. Además vemos algo nuevo en la parte derecha: printout, que sirve para imprimir por pantalla. crlf significa que al final se imprime un retorno de carro.

5. Ejecución del programa y su traza

Nuestro fichero de hechos contiene:

```
; Define la plantilla para la jarra
(deftemplate jarra
(slot litros
(type INTEGER)
(default 0)))
(deffacts jarra-estrucurada-en-0
(jarra)  )
```

Y el de reglas:

```
;; Añade un litro
(defrule un-litro-mas
?jarra <- (jarra (litros ?l))
=>
(modify ?jarra (litros (+ ?l 1))))
;; Añade dos litros
(defrule dos-litros-mas
?jarra <- (jarra (litros ?l))
=>
(modify ?jarra (litros (+ ?l 2))))
;; Termina
(defrule acabar
(jarra (litros 3))
=>
(printout t "Lo he conseguido" crlf) )
```

Para ejecutarlo, realicemos los siguientes pasos:
Carguemos el programa, o bien desde la barra de opciones superior de la interfaz gráfica o escribiendo en la línea de comandos de CLIPS:

```
CLIPS> (load "ejemplo_jarra.clp")
```

Aparecerá entonces en pantalla

```
Defining deftemplate: jarra
Defining defrule: un-litro-mas +j
Defining defrule: dos-litros-mas +j
Defining defrule: acabar +j
TRUE
```

TRUE indica que no se encontró ningún error de sintaxis. En caso contrario aparecerá FALSE e indicará el lugar del código dónde se encontró el error sintáctico.

Digámosle a CLIPS que queremos ver que reglas se disparan (watch rules) y que hechos se ponen y se quitan de la memoria de trabajo (watch facts).
Hagamos (reset) y veamos cómo se añaden los hechos (incluyendo initial-fact, que es necesario para que las reglas que no tienen nada en la parte izquierda se puedan disparar.

CLIPS> (reset)
==> f-0 (initial-fact)
==> f-1 (jarra (litros 0))
La flecha ==> quiere decir que se ha añadido el hecho
Ejecutémos las reglas, pero sólo 4 veces ((run 4)).
CLIPS> (run 4)
FIRE 1 un-litro-mas: f-1
<== f-1 (jarra (litros 0))
==> f-2 (jarra (litros 1))
FIRE 2 un-litro-mas: f-2
<== f-2 (jarra (litros 1))
==> f-3 (jarra (litros 2))
FIRE 3 un-litro-mas: f-3
<== f-3 (jarra (litros 2))
==> f-4 (jarra (litros 3))
FIRE 4 un-litro-mas: f-4
<== f-4 (jarra (litros 3))
==> f-5 (jarra (litros 4))
En esta traza de ejecución podemos ver varias cosas:
1. Que la regla un-litro-mas equipara con el hecho f-1, la cual se dispara, quita el hecho (jarra (litros 0)) y añade el hecho (jarra (litros 1)) (observar que para CLIPS, modificar un hecho equivale internamente a quitar el hecho antiguo y a añadir el modificado).
2. Que se vuelve a disparar la regla un-litro-mas
3. Y asi sucesivamente. La regla que comprueba si hemos llegado al hecho deseado (acabar) no se dispara nunca.
Algo ha ido mal: se ha disparado siempre la misma regla y la jarra se ha llenado demasiado.
Vayamos paso por paso para ver que ocurre:
Primero reset. Después veamos que reglas pertenecen al conjunto conflicto (las que podrían dispararse en un momento dado) CLIPS lo denomina agenda y con esa orden las muestra. Solo una regla puede dispararse en cada ciclo.
CLIPS> (reset)
CLIPS> (agenda)
0 un-litro-mas: f-1
0 dos-litros-mas: f-1
For a total of 2 activations.
Inicialmente hay dos reglas activas: un-litro-mas y dos-litros-mas (ambas equiparan con el mismo hecho: f-1).
Demos un paso:
CLIPS> (run 1)
FIRE 1 un-litro-mas: f-1
<== f-1 (jarra (litros 0))
==> f-2 (jarra (litros 1))
CLIPS> (agenda)
0 un-litro-mas: f-2
0 dos-litros-mas: f-2
For a total of 2 activations.
Vemos que se ha disparado un-litro-mas y que un-litro-mas y dos-litros-mas vuelven a estar disponibles para dispararse (ahora equiparan con f-2, la jarra resultante de añadir 1 litro a la jarra vacía).
Un par de pasos más:
CLIPS> (run 1)
FIRE 1 un-litro-mas: f-2
<== f-2 (jarra (litros 1))
==> f-3 (jarra (litros 2))
CLIPS> (agenda)

0 un-litro-mas: f-3
0 dos-litros-mas: f-3
For a total of 2 activations.
CLIPS> (run 1)
FIRE 1 un-litro-mas: f-3
<== f-3 (jarra (litros 2))
==> f-4 (jarra (litros 3))
CLIPS> (agenda)
0 un-litro-mas: f-4
0 dos-litros-mas: f-4
0 acabar: f-4
For a total of 3 activations.
CLIPS> (run 1)
FIRE 1 un-litro-mas: f-4
<== f-4 (jarra (litros 3))
==> f-5 (jarra (litros 4))
CLIPS> (agenda)
0 un-litro-mas: f-5
0 dos-litros-mas: f-5
For a total of 2 activations.

Ya vemos cual es nuestro problema 1: de todas las reglas activas, en cada ciclo se dispara siempre la primera de la agenda. A la manera en la que un sistema de producción ordena las activaciones se la denomina estrategia de control o estrategia de resolución del conjunto conflicto. En el caso de CLIPS, las reglas de la agenda se ordenan de la siguiente manera:

- Una regla recién activada se coloca en la agenda por delante (en ejecución) de todas las reglas de menor prioridad y por detrás de las reglas de mayor prioridad. Después se verá que es la prioridad de una regla, pero por el momento será suficiente saber que es un número que se asigna a una regla y que determina el orden en el que se colocan esas reglas en la agenda.
- Se usa alguna estrategia de resolución de conflictos para ordenar las reglas con el mismo valor numérico de prioridad. Por omisión, la estrategia de resolución de conflictos es la profundidad. Esta estrategia consiste en que las reglas recién activadas se colocan encima de las reglas con la misma prioridad. Existen otras estrategias de control, como simplicidad, complejidad, LEX, MEA y random. Para poner una estrategia de control distinta a la de profundidad, usar el comando set-strategy. Ej: (set-strategy random).
- Si dos (o más) reglas se activan a la vez por la misma aserción o borrado de un hecho, y los criterios anteriores no bastan para ordenarlas, entonces esas reglas se ordenan de manera aleatoria.

6. Prioridad para las reglas

Para solucionar el problema 1 (que la regla que comprueba el final jamás se dispara), lo ideal sería poderle decirle a CLIPS que la regla acabar debería dispararse siempre antes que ninguna otra. Esto se puede hacer dándoles una prioridad (o en la terminología de CLIPS, una "salience") a las reglas. Por ejemplo, la siguiente declaración le da una prioridad de 1000 (por omisión es 0) a la regla acabar:

```
;; Termian con salience
(defrule acabar
(declare (salience 100))
(jarra (litros 3))
=>
(printout t "Lo he conseguido" crlf) )
```

Si lo hacemos, veremos que al ejecutar el sistema, la traza es correcta y termina bien:

CLIPS> (reset)
<== f-0 (initial-fact)

<== f-1 (jarra (litros 0))
CLIPS> (run 4)
FIRE 1 un-litro-mas: f-1
<== f-1 (jarra (litros 0))
==> f-2 (jarra (litros 1))
FIRE 2 un-litro-mas: f-2
<== f-2 (jarra (litros 1))
==> f-3 (jarra (litros 2))
FIRE 3 un-litro-mas: f-3
<== f-3 (jarra (litros 2))
==> f-4 (jarra (litros 3))
FIRE 4 acabar: f-4
Lo he conseguido
Desgraciadamente, esto no acaba con nuestros problemas, porque si lo ejecutamos un paso mas:
CLIPS> (run 1)
FIRE 1 un-litro-mas: f-4
<== f-4 (jarra (litros 3))
==> f-5 (jarra (litros 4))
vemos que el sistema sigue ejecutándose. Esto es así porque aunque la regla acabar se ha ejecutado, todavía quedan dos reglas activas en la agenda: un-litro-mas y dos-litros-mas. Podemos comprobarlo:
CLIPS> (agenda)
0 un-litro-mas: f-5
0 dos-litros-mas: f-5
For a total of 2 activations.
Y están activas porque hay un hecho de jarra en la memoria. Por tanto, para que el sistema termine realmente, será necesario bloquear (cerrar el paso a su ejecución) la activación de estas reglas borrando el hecho de jarra. Para conseguirlo, la regla de acabar tendrá que retirar también el hecho de jarra que haya en la memoria. Nuestra nueva regla acabar será:

```
; Terminar con Quita la jarra
(defrule acabar
(declare (salience 100))
?jarra <- (jarra (litros 3))
=>
(printout t "Lo he conseguido" crlf)
(retract ?jarra))
```

Lo único nuevo que hemos añadido es (retract ?jarra), que sirve para quitar el hecho jarra (contenido en la variable ?jarra). Notar que hemos usado **<-** para asignar a ?jarra el hecho que equipara con (jarra (litros 3)), para poder borrarla después.
Todavía queda un problema 2 por resolver: la regla dos-litros-mas no se dispara nunca. Podemos intentar resolver el problema 2 como antes, dándole una prioridad a dos-litros-mas (por ejemplo 50) mayor que a un-litro-mas (por ejemplo 25) pero menor que a la regla acabar, pero en ese caso descubriremos que dos-litros-mas se disparará siempre y ni un-litro-mas se disparará nunca (tiene menos prioridad) ni acabar se disparará tampoco (va añadiendo de dos en dos, saltándose todos los contenidos impares de la jarra).
Esto nos lleva a un par de conclusiones importantes:
- El que un problema representado por medio de un sistema de producción encuentre solución o no, depende en gran medida de la estrategia de control (señalado en el problema 1).
- Una estrategia de control basada en la numeración de prioridades puede parecer atractiva pero el control de la ejecución de las reglas suele conllevar estructuras de decisión complejas no fácilmente solubles con asignaciones numéricas de priorirades a las reglas (señalado en el problema 2).
- Incluso aunque con asignaciones numéricas pudiera resolverse, el número de dichas asignaciones crecerá con el número de reglas y su significado/razón-de-ser no es explícito, con lo que complicamos la extensión del código (problema de escalabilidad) y hacemos más difícil

mantenerlo a largo plazo y por otras personas (qué significaban el 25, 50 y 100 de las prioridades de nuestro ejemplo? Con qué otras reglas estaban relacionados esos valores?)

7. Uso de hechos-semáforo

Para solventar los problemas de escalabilidad y mantenimiento debidos al abuso de las prioridades numéricas, se opta por el uso de hechos que actúan como semáforos, abriendo y cerrando la activación de las reglas. Estos hechos reflejan las condiciones (las razones) por las que la ejecución de unas reglas debe tener lugar antes que otras.
Aunque no lo sabíamos, cuando hemos eliminado (retract) el hecho jarra en la regla acabar, estábamos utilizándolo como un semáforo (en rojo). La regla quedaba:

```
; Terminar con Quita la jarra
(defrule acabar
(declare (salience 100))
?jarra <- (jarra (litros 3))
=>
(printout t "Lo he conseguido" crlf)
(retract ?jarra))
```

Al eliminar el hecho jarra impedimos (semáforo en rojo) que pudieran activarse en adelante las reglas un-litro-mas y dos-litros-mas que requieren de la existencia de ese hecho jarra para formar parte de la agenda y por tanto ejecutarse.
Una forma más correcta de implementar este bloqueo de ejecución (ya que jarra está teniendo un uso dual, como semáforo y como representación de la jarra) sería utilizar un nuevo hecho: (jarra-llena), que añadiríamos en la regla acabar y que exigiríamos su no existencia como condición de las reglas un-litro-mas y dos-litros-mas. El código de las reglas quedaría:

```
;; Añade un litro
(defrule un-litro-mas
?jarra <- (jarra (litros ?l))
(not (jarra-llena))
=>
(modify ?jarra (litros (+ ?l 1))))
;; Añade dos litros
(defrule dos-litros-mas
?jarra <- (jarra (litros ?l))
(not (jarra-llena))
=>
(modify ?jarra (litros (+ ?l 2))))
;; Termina
(defrule acabar
(jarra (litros 3))
=>
(printout t "Lo he conseguido" crlf) (assert (jarra-llena)) )
```

Siguiendo la analogía, se pueden añadir un hecho en una regla cuya única funcionalidad sea permitir la activación de otras reglas (semáforo en verde), en las que este hecho figurará como una condición.

8. Ejercicio a entregar

Representar el siguiente problema por medio de un sistema de producción. Tenemos dos jarras, una de 4 litros y otra de 3 litros. Ninguna de ellas tienen marcas que permitan identificar cuánta

agua hay en ellas[1]. Hay un grifo que permite llenar las jarras. El problema consiste en acabar teniendo exactamente 2 litros de agua en la jarra de 4 litros.
En concreto, especificar:
La representación del estado inicial.
Las reglas que especifican las posibles transiciones: llenar-jarra, vaciar-jarra, volcar-jarra (vuelca todo el contenido de una jarra en la otra cuando cabe quedandose vacía) y verter-jarra (vuelca todo lo que quepa de una jarra en la otra quedándose no vacía). Las reglas tienen que ser genéricas. Por ejemplo, para la regla de llenar-jarra, con una única regla se tiene que poder llenar todas las posibles jarras de cualquier capacidad que estuvieran representadas en los hechos.
La regla que detecta el estado final.
Nota: Además de lo ya visto, será necesario para el ejercicio, usar condiciones tipo test en las reglas, por ejemplo: (test (neq ?jarra1 ?jarra2)). En la parte izquierda de las reglas, además de poner patrones para que equiparen con la base de hechos, se pueden poner condiciones que deben cumplir las variables. Este test por ejemplo comprueba que dos variables no tienen el mismo valor asociado.
Esta otra condición: (test (<= (+ ?l1 ?l2) ?c2)) comprueba que la suma de dos variables ?l1 y ?l2 (que representan los litros que contienen cada jarra) es menor o igual que otra variable ?c2 (que representa la capacidad de la jarra2).

12. Entrega

Subir a AG los ficheros de hechos y reglas de CLIPS del ejercicio anterior
Al final del fichero CLIPS, como comentario, poner la respuesta a las siguientes cuestiones:
1. Cuántos ciclos de ejecución tarda en llegar a la solución?
2. Cambiar la estrategia de control a random, volver a ejecutar, cambia algo? Por qué?

1 pero las jarras en nuestro código sípueden representar los litros que contienen en cada momento