PRÁCTICA NÚMERO 1
INTRODUCCIÓN A CLIPS

¿Qué es CLIPS?

CLIPS es una herramienta software para la creación de sistemas expertos. Un sistema experto es un programa de ordenador pensado para realizar las tareas que normalmente realiza un ser humano considerado "experto" en algún área. La finalidad del sistema experto es realizar la tarea del experto humano reproduciendo el mismo modelo que éste utiliza para la resolución de problemas pertenecientes a su ámbito de saber. De este modo, en todo sistema experto se pueden distinguir las siguiente partes:

- Los datos, o información base sobre el campo en el que se desarrollarán las actividades del experto.
- La base de conocimientos, que recoge las formas de utilizar la información citada en el punto anterior.
- El motor de inferencias, que modela el proceso de razonamiento mediante el cual el experto humano realiza su tarea.

CLIPS es una herramienta para construir sistemas expertos basados en reglas. Las reglas son una forma de representar los conocimientos del experto, constan normalmente de una parte condicional que, si se cumple, tiene como consecuencia la ejecución de una acción asociada. Cada vez que se aplica una de las reglas, los datos guardados en el sistema pueden variar, dando lugar a que se activen otras reglas que en la situación anterior no lo estaban. La resolución del problema planteado pasa por la aplicación sucesiva de reglas hasta que se llega a la condición de finalización. A este proceso se le denomina **conducido por los datos**, ya que las acciones sólo se producen cuando hay datos que las justifican.

En CLIPS, las acciones se ejecutan en base a los datos que existen en ese momento en el sistema, lo que supone una importante diferencia con los lenguajes procedurales (Pascal, C,

etc) que pueden ejecutar acciones si están programadas, sin tener en cuenta la existencia de datos.

La nomenclatura seguida en CLIPS para designar las principales partes del sistema experto son:

- **Hechos (facts)** son los que forman la base de información o datos del sistema.
- **Reglas (rules)** forman la base de conocimiento.

Entorno de desarrollo de CLIPS

La herramienta CLIPS incluye un entorno de desarrollo para programar sistemas de forma completa. En la ventana de comandos aparece el prompt:

CLIPS>

En este prompt se pueden introducir comandos directamente, en lo que se denomina "alto nivel".

Para salir de CLIPS se usa el comando **exit**.

CLIPS>(exit)

También se puede usar la entrada del menú desplegable File->Quit Clips o la secuencia de teclas abreviada Ctrl+Q.

En CLIPS, todos los elementos deben ir encerrados entre paréntesis, teniendo una sintaxis parecida a la de LISP.

Los hechos

Los hechos son los datos de partida del problema cuando éste se empieza a ejecutar y además recoge el estado del sistema en cualquier momento de su ejecución. Habitualmente la introducción de los hechos iniciales es la primera operación a realizar.

Para introducir hechos en el sistema se usa el comando **assert**:

CLIPS>(assert (color tomate rojo))

Cabe señalar que el espacio en blanco es un delimitador de elementos en CLIPS, de este modo, "color", "tomate" y "rojo" son símbolos diferentes para el sistema. También delimitan elementos el tabulador o el retorno de carro, aunque CLIPS convierte todos a espacios en blanco. La interpretación del hecho en sí (color tomate rojo) no está prefijada de ninguna forma, sino que queda a elección del programador, aunque es conveniente seguir una misma norma en todos los casos. En el ejemplo, para aludir a una cualidad de un objeto se ha supuesto la estructura (cualidad objeto valor-de-la-cualidad). Si se hubiese introducido el hecho (tomate color rojo), sería tratado como un hecho diferente al anterior, ya que para CLIPS el orden de los símbolos es significativo.

En este caso se ha introducido un solo hecho, aunque con un comando assert pueden introducirse varios hechos en la misma instrucción, lo único que hay que hacer es delimitar los hechos por medio de paréntesis:

CLIPS>(assert (hecho-1) (hecho-2))

En este caso se han introducido dos hechos de un **campo**, mientras que en el caso anterior se introdujo un hecho de tres campos

Con los hechos que se van introduciendo al sistema CLIPS genera una lista de hechos, numerándolos desde el número 0 en adelante.

Para obtener un listado con todos los hechos de la base de hechos se utiliza el comando **facts**, que para introducirlo en el sistema, debe ir rodeado de paréntesis:

CLIPS>(facts)

El resultado es un listado de hechos ordenados según su orden de introducción y numerados desde el número 0 hasta el número de hechos total menos 1. La identificación del hecho se realiza mediante la notación

f-nº de hecho

Para eliminar hechos del sistema se utiliza el comando **retract**, seguido del número de índice que tenga el hecho en la lista de hechos. La eliminación de hechos no supone la renumeración automática de los que quedan en la lista, por lo que podrá haber índices sin ocupar.

CLIPS> (retract 0)

Si se quiere eliminar toda la base de hechos de una vez, se debe utilizar el comando **reset**. Este comando elimina todos los hechos existentes y además añade uno a la lista, que será el de orden cero y tiene como nombre "initial-fact". Este hecho es de carácter auxiliar y puede ser usado para disparar reglas al comienzo del proceso.

Posibles valores para los campos de los hechos

Hay varios **tipos** de campo para los hechos: **número flotante, entero, símbolo, cadena, dirección-externa, dirección-de-hecho, nombre-de-instancia y dirección-de-instancia**.

El tipo de dato introducido en un campo puede ser determinado de forma automática según el valor introducido en el mismo o de forma explícita.

Un **símbolo** es un tipo de campo que comienza por un carácter ASCII imprimible seguido de ninguno o varios caracteres imprimibles más. CLIPS distingue entre mayúsculas y minúsculas y tiene ciertos caracteres reservados, como:

" ( ) & | < ~ ; ? $

Los caracteres "&", "|" y "~" no se deben usar como símbolos independientes ni como parte de ningún símbolo.

El punto y coma actúa como el comienzo de un comentario en CLIPS

Una **cadena** es un tipo de datos que va encerrado entre dobles comillas, que también forman parte del campo. Para introducir la doble comilla como parte de un campo se debe usar la barra invertida \.

Los **campos numéricos** pueden ser **enteros** o de **punto flotante**, siendo tratados como enteros largos o flotantes de doble precisión respectivamente. No se puede introducir un número aislado como un hecho independiente, sino que deben introducirse como hechos de dos campos, siendo el primer campo un símbolo.

CLIPS>(assert (a 233))

Hechos ordenados y no ordenados

Cuando el orden de los campos de un hecho es importante para la interpretación del mismo, se dice que el hecho es **ordenado**. Es el caso del hecho (color tomate rojo), que es diferente del hecho (color rojo tomate). La interpretación del significado del hecho queda a cargo del usuario del sistema, pudiendo no ser obvia siempre. Hasta ahora sólo se han visto hechos ordenados.

Cuando se quiere explicitar un cierto significado para los campos se les da nombre y se fija su contenido, de forma que, aparezcan en el orden en que aparezcan en el hecho, siempre se sabrá lo que deben contener.

Introducción inicial de hechos.

La instrucción **deffacts** se usa para introducir una serie de hechos en la base de hechos sin tener que hacerlo de uno en uno. Un ejemplo sería:

```
(deffacts hechos-iniciales
        (color luz-semaforo rojo)
        (edad juan 44))
```

Para introducir estos hechos por primera vez hay que reiniciar el sistema con el comando reset ya visto. Este comando limpia la base de hechos, introduce el hecho (initial-fact) y además todos los hechos que existan en las instrucciones deffacts.

El entorno de desarrollo de CLIPS

El entorno de desarrollo de clips consta de tres archivos:

- El ejecutable (clipswin.exe) es el que controla el interfaz de comandos y el mecanismo de inferencia.
- El archivo de ayuda (clips6.hlp) contiene referencias a los comandos de CLIPS.
- El editor de texto (clipsedt.exe) sirve para introducir comandos, pudiendo guardarlos en archivos para usarlos posteriormente.

Ejecutando el archivo “clipswin.exe” aparece la ventana de comandos, con el prompt “CLIPS>”. Desde este prompt ya se pueden insertar comandos, aunque tiene la desventaja de no ofrecer la posibilidad de copiar y pegar ni de mantener un histórico de comandos. Por esto,

lo más cómodo es abrir una ventana del editor (Ctrl+E), escribir las secuencias de comandos, copiarlos y pegarlos en la ventana de comandos.

La entrada Edit->Complete.., con secuencia de teclas equivalente Ctrl+J completa un comando del que se han escrito las primeras letras. Si hay más de un comando que comience por esa secuencia de letras, se abre una ventana para poder elegir cual es el que se busca.

La entrada Execution->Reset es igual que el comando reset, pudiéndose usar la secuencia de teclas Ctrl+U.

La entrada Window permite abrir diversas ventanas para observar el contenido de la base de hechos, la de reglas, etc. Normalmente se trabajará usando Window->All Above, que abre todas las ventanas posibles.

Las reglas

Hasta ahora sólo se ha visto el primero de los elementos de los que se compone un sistema experto: los hechos. Como tales, estos hechos son estáticos, el sistema no evolucionaría si no hubiese algo capaz de cambiar la base de hechos de acuerdo al conocimiento del experto sobre la forma de usar esos datos. Este conocimiento del experto está implementado en CLIPS en forma de las denominadas **reglas**.

El funcionamiento de una regla es parecido al de una sentencia IF...THEN de un lenguaje de programación procedural.

Por ejemplo, en un sistema que se encargue de indicar si se puede cruzar una calle regulada por un semáforo, una posible regla simple sería: SI la luz del semáforo está verde, ENTONCES se puede cruzar. La base de hechos podría ser la siguiente:

(color luz-semaforo roja)
(se-puede-cruzar no)

Cuando la luz del semáforo pase a ser verde, entonces, el hecho (se-puede-cruzar no) pasaría a ser (se-puede-cruzar si). De realizar esta operación se encarga la regla que se activará cuando se encuentre en la base de hechos el hecho (color luz-semaforo verde).

La regla sería la siguiente:

```
(defrule pasar
        (color luz-semaforo verde)
=>
        (assert (se-puede-cruzar si)))
```

La regla podría haberse escrito toda en una línea, pero es conveniente hacerlo de forma que sea fácilmente comprensible.

La definición de la regla se realiza mediante la instrucción defrule, viniendo a continuación el nombre de la regla. El resto de la definición de la regla está dividido en dos partes:

- La parte de antes del símbolo "=>" se denomina "**parte izquierda de la regla**" y está compuesta por **patrones**, que son cada una de las condiciones que se deben cumplir en la base de hechos para que la regla se **active**. En este ejemplo sólo hay un patrón.
- La parte que aparece con posterioridad al símbolo "=>" se denomina "**parte derecha de la regla**" y está formada por cada una de las **acciones** que tendrán lugar cuando todos los patrones de la parte izquierda se correspondan con hechos de la base de hechos.

Para que una regla se active, cada uno de los patrones de su parte izquierda debe **corresponder** con algún hecho de la base de hechos. En el caso anterior, la correspondencia se dará si en la base de hechos existe un hecho que sea exactamente (color luz-semaforo verde), aunque no siempre será así. Puede haber ocasiones en los que se busque la correspondencia con un patrón del tipo (color luz-semaforo ?), lo que querría decir que se

busca un hecho en la base de hechos que tenga tres símbolos: el primero debe ser color, el segundo debe ser luz-semaforo y el tercero no importa cual sea.

En el caso del ejemplo, si la base de hechos es:

(color luz-semaforo roja)
(se-puede-cruzar no)

y pasa a ser:

(color luz-semaforo verde)
(se-puede-cruzar no)

entonces, la parte izquierda de la regla "pasar" corresponde con el hecho (color luz-semaforo verde), por lo que la regla "pasar" se activará. Cuando esta regla se ejecute, la situación de la base de hechos cambiará, quedando:

(color luz-semaforo verde)
(se-puede-cruzar no)
(se-puede-cruzar si)

En este momento, podría activarse otra regla que tuviese en su parte izquierda el hecho (se-puede-cruzar si) y cuya acción fuese activar un motor de un robot para que éste cruzase la calle.

Hay que tener en cuenta que en la base de hechos se tiene ahora dos hechos que según el sentido común serían contradictorios:

(se-puede-cruzar no)
(se-puede-cruzar si)

por lo que lo normal será eliminar el hecho (se-puede-cruzar no) cuando se introduzca el (se-puede-cruzar si) y viceversa.

Que una regla se active no quiere decir que las acciones de su parte derecha se vayan a ejecutar. La activación consiste en el paso de la regla a una especie de memoria de trabajo denominada **agenda**, donde queda almacenada para ser ejecutada. Sin embargo, esto no presupone que se vaya a ejecutar, ya que puede activarse otra regla con mayor prioridad que ésta y que elimine las condiciones que hacían que la primera regla estuviese activada.

De todas las reglas que hay en la agenda, CLIPS ejecuta las acciones de la parte derecha de la regla que tenga mayor prioridad. Mientras haya reglas activadas en la agenda CLIPS irá ejecutando en cada momento la de mayor prioridad hasta que no queden reglas activas.

La **prioridad** de una regla se puede establecer manualmente mediante la propiedad **salience** que se puede asociar a una regla cuando se define. El valor de la propiedad salience puede variar entre –10.000 y 10.000, valiendo cero por defecto, es decir, cuando no se especifique ningún valor explícitamente.

```
(defrule pasar
        (declare (salience 1))
        (color luz-semaforo verde)
=>
        (assert (se-puede-cruzar si)))
```

Para observar las reglas que hay activas en la agenda en cada momento se puede usar el comando **agenda** o ver las reglas activas en la ventana agenda.

El comando reset no borra las reglas, actúa sobre los hechos. El comando clear borra tanto hechos como reglas.

Ejecución de un programa

Las reglas se activan cuando el sistema se ejecuta. Una vez que se tiene completa la base de hechos y las reglas del sistema, se puede comprobar su funcionamiento ejecutando el programa. El sistema se ejecuta mediante la sentencia **run** o usando la combinación Ctrl+r.

```
CLIPS>(run)
```

Con esta sentencia, la ejecución se realiza de forma ininterrumpida, hasta que no hay más reglas activas en la agenda. Se puede indicar que no se ejecute el programa hasta el final, sino sólo un número fijo de pasos o reglas que se ejecutarán. Esto se hace indicando este número en la sentencia run:

```
CLIPS>(run 3)
```

Si se quiere ejecutar el sistema regla a regla se pondría (run 1) (o la secuencia de teclas Ctrl+t).

En el ejemplo anterior, si se tiene:

Base de hechos:

```
(color luz-semaforo verde)
(se-puede-cruzar no)
```

Reglas:

```
(defrule pasar
        (color luz-semaforo verde)
=>
        (assert (se-puede-cruzar si)))
```

Si se ejecuta el comando (run), la situación final será:

Base de hechos:

(color luz-semaforo verde)
(se-puede-cruzar no)
(se-puede-cruzar si)

Cabe pensar que, puesto que el patrón del lado izquierdo de la regla "pasar" todavía se corresponde con el hecho (color luz-semaforo verde) de la base de hechos, ésta regla seguiría activándose de forma indefinida. Esto no es así porque CLIPS implementa un mecanismo denominado **refracción** que consiste en que una regla nunca se activa más de una vez para la misma combinación de hechos. Sí se activaría de nuevo si se elimina el hecho (color luz-semaforo verde) y se vuelve introducir de nuevo, ya que CLIPS trataría a estos dos hechos como hechos diferentes.

La cantidad de condiciones que se deben cumplir para que una regla se active depende de las especificaciones del problema a resolver. Si hay más de una condición, todas deberán cumplirse. La condición de activación es una operación AND lógica sobre todos los patrones de la parte izquierda de la regla.

En el ejemplo del semáforo, se puede tener la regla:

```
(defrule pasar
        (color luz-semaforo verde)
        (estado camino libre)
=>
        (assert (se-puede-cruzar si)))
```

En este caso, además de estar la luz del semáforo verde, el camino debe estar libre de obstáculos, lo cual se indica por el hecho de la base de hechos (estado camino libre). El número de condiciones a poner en la regla dependerá de las especificaciones del problema, de la calidad de la solución aportada, etc.

Mediante la instrucción **save** se guardan las reglas del sistema a un archivo:

CLIPS>(save archivo.clp)

Estas reglas se pueden volver a cargar mediante la instrucción **load**:

CLIPS>(load archivo.clp)

Si se vuelve a definir una regla con el nombre de otra ya existente, la última remplazará a la más antigua.

Los hechos se cargan y salvan con las instrucciones **load-facts** y **save-facts** respectivamente.

Imprimir mensajes

Para comprobar el funcionamiento de los programas se puede usar la sentencia **printout**. Para que la salida sea por pantalla, hay que indicarlo con la letra "t"

CLIPS>(printout t "Hola" crlf)

"crlf" es un símbolo especial que indica que se imprima un retorno de carro.

Esta instrucción se usa habitualmente en la parte de las acciones de las reglas, para comprobar el resultado de su ejecución.

Ayudas para la depuración

Mediante la instrucción **watch facts** se puede controlar la evolución de los hechos en la base de hechos:

CLIPS>(watch facts)

Se indica por pantalla cada vez que un hecho nuevo se introduce o cada vez que alguno de los existentes es borrado.

Para dejar de monitorizar los cambios en los hechos se usa el comando **unwatch**.

Además de los hechos, se puede tener control sobre los cambios en prácticamente todas las partes del sistema mediante comandos del tipo **watch elemento**. Todos estos comandos pueden introducirse también mediante el menú Execution->Watch... o mediante la secuencia de teclas Ctrl+W.

Mediante el comando **watch rules** se pueden ver las reglas que se ejecutan y con el comando **watch activations** las que pasan a la agenda.

El comando **dribble-on** guarda todo lo que se introduce en la ventana de diálogo a un archivo del disco hasta que se introduzca el comando **dribble-off**.

CLIPS>(dribble-on archivo)

Comodines

El carácter **?** es empleado como comodín dentro de un patrón en el antecedente, parte izquierda de las regla, de manera que cualquier valor en la posición referenciada por dicho carácter dentro del patrón será aceptado.

Por ejemplo, un recepcionista de hotel quiere saber si hay alguna persona inscrita que se llame "Jose", y tiene la siguiente base de hechos:

(huesped Mario Picazo)
(huesped Jose Cano)
(huesped Jose Perez Prieto)
(huesped Jose Alaiz Llamas Garcia)

y tiene definida la siguiente regla:

```
(defrule nombre -jose
        (huesped  Jose ?)
=>
        (printout  t "Si hay alguien inscrito que se llame Jose" crlf))
```

Lo que se busca es la existencia de huéspedes que se llamen Jose y q aparezcan en la base de hechos con un único apellido (independientemente de cuale sea éste), en estos casos en los que sólo una parte del patrón es especificada es cuando deben utilizarse los comodines.

Como el patrón tiene 3 campos, uno de los cuales es un comodín, solo los hechos de la base de conocimientos que consten de exactamente tres símbolos pueden corresponderlo. Con lo cual está regla sólo se activará con el hecho (huesped Jose Cano).

Si además de llamarse José se buscan huéspedes que estén inscritos con 2 apellidos, lo que hay que hacer es especificar el patrón de la siguiente manera:

```
(defrule nombre-jose-2apellidos
        (huesped  Jose  ? ?)
=>
        (printout t "Si hay alguien inscrito que se llame Jose" crlf))
```

Si se busca un huesped que este inscrito con los dos apellidos y además su primer apellido sea Perez y nos da igual como se llame o cual sea su segundo apellido ,

```
(defrule primerapellido-Perez
        (huesped  ?  Perez ?)
=>
        (printout t "Si hay alguien inscrito con primer apellido Perez" crlf))
```

? es un **comodín simple** que debe representar exclusivamente una entidad dentro del patrón. También existe el **comodín múltiple** representado por los caracteres **$?** que representa cero o más entidades. Ambos pueden ser combinados de cualquier forma dentro de un patrón.

Aplicado al ejemplo en el que se busca la existencia de huéspedes que se llamen Jose, si se utiliza el comodín múltiple, el resultado será independiente de que la persona se haya inscrito con uno o dos apellidos.

```
(defrule algun-jose
        (huesped  Jose $?)
=>
        (printout  t "Si hay alguien inscrito que se llame Jose" crlf))
```

Variables

Si queremos almacenar el valor por el que ha sido sustituido un comodín dentro de un patrón al ser emparejado con un hecho de la base de conocimientos, utilizamos las variables dentro de los patrones, de esta manera podremos *trabajar con el valor almacenado* en la *parte derecha de la regla* (visualizarlo, compararlo, insertarlo en la base de hechos, etc.)

La notación empleada para las variables consta de un signo de interrogación seguido de un símbolo. **?nombredelavariable**

Las variables son *locales* a las reglas donde se utilizan, también se pueden definir variables globales como se verá más adelante.
Antes de que una variable pueda ser usada, tiene que asignársele un valor.

Siguiendo con el ejemplo del recepcionista, utilizaremos la variable ?primerapellido para asignarle un valor en la parte izquierda de la regla y trabajaremos con el en la parte derecha.

Base de hechos:
(huesped Mario Picazo)
(huesped Jose Cano)
(huesped Jose Perez Prieto)
(huesped Jose Alaiz Llamas Garcia)

Regla:

```
(defrule imprimo-apellido
        (huesped  Jose  ?apellido)
=>
        (printout t "Si hay alguien inscrito que se llame Jose y se apellida  "?apellido crlf))
```

Ejemplo para insertar un hecho en la base de conocimientos mediante una variable. El recepcionista ha recordado que Mario le pidió que le despertara a las 9, como no recuerda su apellido escribe la siguiente regla:

```
(defrule despertar-Mario
        (huesped Mario ?apellido)
=>
        (assert (depertar  9 huesped Mario ?apellido  ))
)
```

Al igual que existe el comodín múltiple existe la **variable multiatributo, $?nombrevariable** que produce correspondencia con cero o más entidades dentro de un patrón.

Siguiendo con el ejemplo

Regla:

```
(defrule imprimo-apellido
        (huesped  Jose  $?apellido)
=>
        (printout t “En el hotel está inscrito un huesped con nombre Jose y se
        apellida  ”?apellido crlf))
```

Hay que destacar que una vez que a la variable se le asigna un valor por primera vez en un patrón del antecedente de una regla, éste valor permanece fijo aunque la misma variable vuelva a aparecer en otros patrones de la misma regla.

Base de hechos:

```
(pelo rubio Jose)
(pelo rubio Mario)
(pelo moreno Oscar)
(ojos azules Jose)
(ojos verdes Oscar)
(ojos verdes Mario)
```

Regla:

```
(defrule hombre-atractivo
        (pelo rubio ?nombre)
        (ojos azules ?nombre)
=>
(printout t “El hombre atractivo es “ ?nombre crlf)
)
```

Al ejecutarlo en clips se comprueba que la regla sólo la activa el conjunto de hechos siguiente:

(pelo rubio Jose)
(ojos azules Jose)
mientras que el conjunto de hechos:
(pelo rubio Mario)
(ojos azules Jose)

no activa la regla, porque en la primera aparición de la variable ?nombre se le asigna el valor Mario y esto se mantiene en toda la regla.

Otro uso muy común de las **variables** en CLIPS es para *eliminar o modificar hechos* en la parte derecha de las reglas, para ello primero hay que asignarle el índice del hecho a una variable en el antecedente de la regla. Hasta ahora a las variables se les había asignado el valor de una entidad de un hecho.
El **índice del hecho** se específica en CLIPS utilizando los símbolos <- .
Siguiendo con el ejemplo anterior, ahora el recepcionista quiere desinscribir al huesped Jose Cano.

Base de hechos:
(huesped Mario Picazo)
(huesped Jose Cano)
(huesped Jose Perez Prieto)
(huesped Jose Alaiz Llamas Garcia)

Regla:
(defrule desinscribir-Jose Cano
?huesped **<-** (huesped Jose Cano)
=>
(retract ?huesped)
)

Si el recepcionista no recuerda como se llama, sólo que se apellida Cano aplicaría la siguiente regla:

```
(defrule desinscribir-Cano
?huesped <- (huesped  ?nombre  Cano)
=>
(retract ?huesped)
(printout t “El huesped  “ ?nombre “ Cano se ha ido” crlf )
)
```

EJERCICIOS

1.- Llenado de jarra

Simular el llenado de una jarra de tres litros mediante dos posibles operaciones:

- Agregar un litro.
- Agregar dos litros.

Indicaciones:

El contenido de la jarra en cada momento se recogerá en un hecho del tipo:

(contenido jarra 0)

Las operaciones a realizar podrán ser hechos de dos tipos:

(agregar 1)

(agregar 2)

Deberá existir una condición de terminación, cuando la jarra tenga 3 litros.

Las operaciones aritméticas en CLIPS se indican en notación infija:

(+ 2 3) suma los números 2 y 3

Si el contenido de la jarra está en la variable ?litros, para añadir 1 a esta cantidad se haría:

(assert (contenido jarra (+ ?litros 1)))

2.- Variante del llenado de jarras

Se desea ahora que la operación de llenado de la jarra se realice de forma automática, sin indicar ninguna operación en la base de hechos. Si la jarra no tiene 3 litros, se deberá ejecutar la acción de añadir 1 o 2 litros a la misma.

Indicaciones:

Las operaciones de añadir un litro y añadir dos litros serán sendas reglas que se activan siempre que exista la instancia jarra en la base de hechos, es decir, el antecedente de estas reglas será el patrón:

(contenido jarra ?litros)

¿Cómo se evita que la jarra siga llenándose una vez que alcanza los tres litros?

3.- Lista

Una lista en CLIPS se puede representar por un hecho del tipo:
(lista a b c d e f g)

donde el primer símbolo se toma como el identificador de la lista y el resto son los elementos de la misma.

Dada una lista de este tipo en la base de hechos, ¿cómo se obtiene el primer elemento de la misma?, ¿cómo se obtiene el último?, ¿cómo se intercambiaría el primero con el último?
Escribir los programas que realicen estas operaciones, sacando por pantalla los elementos pedidos y la lista resultado.

Indicaciones

La variable multiatributo $?nombre-variable produce correspondencia con cualquier número de campos. Si se usase en un antecedente de una regla para comprobar la correspondencia con la lista anterior se podría poner:

(lista $?elementos)

este patrón correspondería con hechos del tipo (lista), (lista f), (lista r t g b), etc.

Si tuviésemos:

(lista $?elementos g)

este patrón correspondería con hechos del tipo (lista a u j g d b g), (lista g), (lista u g), etc.

4.- Suma de áreas de rectángulos

Se parte de una situación inicial con los siguientes hechos:

(rectangulo A 9 6)
(rectangulo B 7 5)
(rectangulo C 6 9)
(rectangulo D 2 5)

Estos hechos representan diferentes rectángulos de nombres A, B, C y D, con unas dimensiones de base y altura que vienen dadas por los dos números siguientes.

Se quiere obtener un programa que efectúa la suma de todas las áreas de los rectángulos.

Indicaciones:

Al igual que se puede poner como condición de una regla la correspondencia entre un patrón y algún elemento de la base de hechos, se puede indicar también como condición que un patrón no exista en la base de hechos. Para esto se usa el operador booleano **not**.

Por ejemplo, si para activar una regla es necesario que no exista en la base de hechos ningún rectángulo de base 6 se indicaría:

(not (rectangulo ? 6 ?))

Si en la base de hechos no hay ningún rectángulo con valor de base igual a 6, la correspondencia con el patrón (rectangulo ? 6 ?) daría como resultado un valor lógico de FALSO, y su negación (not) daría un valor lógico VERDADERO.