PRÁCTICA NÚMERO 2
INTRODUCCIÓN A CLIPS

Las plantillas

En los **hechos ordenados**, con los que se han trabajado en la práctica anterior, la información esta almacenada posicionalmente , es decir , para acceder a esa información se debe conocer que está contenida en ese hecho y además en qué posición concreta está almacenada . Esto último puede ser evitado con los **hechos no ordenados** , donde se asignan nombres a los campos y de ese modo no es necesario conocer la posición exacta de la entidad dentro del hecho, pues basta con utilizar su nombre para poder acceder a ellos. Los hechos no ordenados se definen mediante el constructor **deftemplate**. Un ejemplo sería:

```
(deftemplate persona                ;nombre de la plantilla
      "información adicional"       ;comentarios adicionales opcionales
      (slot nombre                     ;nombre del campo
         (type STRING)              ;tipo de campo
         (default ?DERIVE))         ;valor por defecto de este campo
      (multislot apellidos
         (type STRING)
         (default ?DERIVE))
      (slot edad
         (type NUMBER)
         (default 22))
      (slot estado
         (type SYMBOL)
         (default soltero)))
```

El ejemplo tiene cuatro campos o atributos denominados, nombre, apellidos, edad y estado. Insertamos hechos que utilizan esta plantilla sin importarnos el orden en el que se han definido los atributos, ya que se acceden a ellos por nombre, por ejemplo:

```
(assert (persona (apellidos "Picazo" "Pérez") (nombre "Mario")))
(assert (persona (nombre "Jose")))
```

Los componentes de la plantilla son :

- El **nombre** de la plantilla.
- Los **slots** o atributos de la plantilla, también denominados campos, para cada uno de los cuales se pueden definir las siguientes propiedades:
    - Nombre del atributo, es la única propiedad obligatoria.
    - Tipo de datos (type) que puede tomar el atributo: *symbol, string, integer, float,* si es indiferente que sea int o float se utiliza *number*.
    - Valor por defecto (default)que toma el atributo. Si se indica ?DERIVE, CLIPS inserta un valor por defecto dependiente del tipo de dato escogido, por ejemplo para string introduce "", para int 0, para float 0.0. Si se indica ?NONE , entonces es obligatorio asignar un valor a este slot al realizar el assert de un hecho que utilice esta plantilla.
    - Valores permitidos (allowed-type: allowed-symbols, allowed-strings , allowed-numbers, ...) en esta propiedad se pueden definir una lista de valores permitidos para este atributo.
    - Rango de valores (range) en esta propiedad se puede definir el rango de valores que puede tomar este atributo.

Las propiedades allowed-type y range no pueden ser usadas simultáneamente.

Los slots pueden ser del tipo **simple(slot)** o **múltiple(multislot)** . Un slot simple contiene exactamente un campo, mientras que uno múltiple contiene cero o más campos. Dentro de una plantilla se puede definir un número indeterminado de simples y múltiples slots.

Una plantilla no puede ser redefinida o eliminada mientras esta siendo usada, por ejemplo por un hecho o un patrón en una regla.

Otro ejemplo sería:

```
(deftemplate persona
 (slot nombre
    (type SYMBOL)
    (default ?NONE))          ;obligamos a que esta propiedad sea definida al insertar un
                              ;hecho con esta plantilla
(multislot apellidos
    (type SYMBOL)
    (default ?DERIVE))
(slot edad
    (type NUMBER)
    (range 18 ?VARIABLE)
    (default 22))
(slot estado
    (type SYMBOL)
    (default soltero)
   (allowed-symbols soltero casado divorciado))
)
```

Una plantilla puede ser usada como patrón dentro de una regla de igual forma que cualquier patrón ordinario. Por ejemplo, en la siguiente regla se seleccionan personas que cumplen unos determinados requisitos:

Base de hechos:

```
(assert (persona (nombre Juan) (apellidos Diez Perez) (edad 22) (estado casado))
        (persona (nombre Jose) (apellidos Rojo Llamas)))
```

Regla:

```
(defrule seleccionar-persona
        (persona (nombre ?nombre) (apellidos $?apellidos)(estado soltero))
=>
(printout t  "Ha sido seleccionado  " ?nombre  " " ?apellidos crlf)
)
```

Hay que resaltar, como se puede ver en el ejemplo anterior, que en los patrones de las reglas utilizaremos **?** ó **?nombrevariable** para sustituir o almacenar el valor de un slot simple y **$?** ó **$?nombrevariable** para los multislot.

Modificar hechos

Las plantillas facilitan el acceso a un campo específico en un patrón dado que los campos se identifican por su nombre. La acción **modify** se usa para eliminar e insertar un nuevo hecho en una única acción, especificando uno o más atributos de la plantilla para que sean modificados.
Un ejemplo, en el que la propiedad estado de la persona Jose pasa de estar soltero a casado sería:

Base de hechos:

```
(assert (persona (nombre Juan) (apellidos Diez Perez) (edad 22) (estado casado))
        (persona (nombre Jose) (apellidos Rojo Llamas)))
```

Regla:

```
(defrule pillado
        ?novio<-(persona (nombre ?nombre) (estado soltero))
=>
(printout t   ?nombre “  se ha casado “  crlf)
(modify ?novio (estado casado))
)
```

Como se puede comprobar desde la ventana de hechos, el hecho Jose estado soltero ha sido eliminado y se ha insertado un nuevo hecho que mantiene todas las propiedades de Jose, excepto el estado, que ha pasado como se indica en la instrucción modify a casado.

Asignación de un valor a una variable en el consecuente de una regla: BIND

En el antecedente de una regla se le asigna un valor a una variable mediante la correspondencia del patrón con un hecho de la base de hechos, mientras que en el consecuente de la regla se hace mediante la función **bind.**
Por ejemplo, suponer que hoy es el cumpleaños de Jose:

```
(defrule cumpleagnos
?indice<-(persona (nombre Jose) (edad ?agnos))
=>
(bind ?edad (+ 1 ?agnos))
(printout t   "Jose cumple " ?edad  "años "crlf)
(modify ?indice (edad ?edad))
)
```

Es conveniente asignar valor a las variables en la parte derecha de la regla cuando los mismos valores van a ser usados reiteradamente.

Define tus propias funciones: deffunction

CLIPS permite definir al usuario funciones mediante **deffunction.** Las funciones son globales, lo que permite ahorrar el esfuerzo de meter las mismas acciones una y otra vez, así como ganar en comprensión del programa.

Por ejemplo, crear una función que calcule la hipotenusa y después usarlo en una regla.

```
(deffunction hipotenusa          ; nombre de la función
        ( ?a  ?b)                    ;argumentos
        (sqrt (+ (* ?a  ?a) (*  ?b  ?b))))      ;acción

(assert (dimensiones 3 4))
(defrule calcular-hipotenusa
        (dimensiones ?base ?altura)
=>
(bind  ?hipotenusa  (hipotenusa ?base ?altura))
 (printout t "la hipotenusa es "  ?hipotenusa crlf)
(assert (hipotenusa ?hipotenusa)))
```

Deffunction sólo devuelve el valor evaluado en la última acción definida dentro de la función. Si no tiene acciones devuelve el símbolo FALSE.

Cuando el número de argumentos que se le pueden pasar a la función no es fijo se indica utilizando la combinación **$?,** por ejemplo, una función que nos devuelva la longitud de una lista de elementos:

```
CLIPS>(deffunction cuenta-elementos
        ($?argumentos)
        (length $?argumentos))
CLIPS>( cuenta-elementos 1 2 3 4 5 6)
```

Nota: La función length, devuelve el número de entidades, o el número de caracteres en una cadena o símbolo.

Entrada desde el teclado

En una regla, además de obtenerse información mediante la correspondencia de patrones, también puede obtenerse información que el usuario introduzca desde el teclado. Para este fin se usa la función **read**:

```
CLIPS>(defrule leer-nombre
        (initial-fact)
=>
        (printout t “Escriba su nombre “ crlf)
        (assert (nombre-usuario (read)))
)
```

La instrucción **read** sólo lee un símbolo. Para leer más de una palabra, se debe encerrar la frase entera entre comillas. Lo mismo puede hacerse sin necesidad de usar las comillas mediante la sentencia **readline**, que lee valores del teclado hasta que se introduce un retorno de carro.

Para introducir en la base de hechos símbolos y no cadenas de caracteres se debe usar la función **assert-string**, en vez de assert:

```
CLIPS>(defrule leer-nombre
        (initial-fact)
=>
        (printout t "Escriba su nombre " crlf)
        (assert-string (readline))
)
```

En este caso, para introducir un hecho desde el teclado se introducirá con los correspondientes paréntesis:

```
(nombre-usuario pepe)
```

CLIPS lo tratará como la cadena "(nombre-usuario pepe)" y, mediante la sentencia assert-string será introducido en la base de hechos como (nombre-usuario pepe).

Se propone realizar la introducción del hecho (nombre-usuario pepe) sin tener que introducir por teclado los paréntesis. Recordar que mediante la función **str-cat** se concatenan cadenas en CLIPS.

Operadores booleanos y de comparación.
Restricciones en los campos

Igual que la condición para cruzar una calle se podía escribir:

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

se podría indicar la regla para que cuando la luz no sea verde se active el hecho (se-puede-cruzar no):

```
(defrule no-pasar
        (color luz-semaforo ~verde)
=>
        (assert (se-puede-cruzar no)))
```

De este modo, independientemente del número y tipo de colores que tenga el semáforo en cuestión, si la luz no es verde, se introducirá el hecho (se-puede-cruzar no) en la base de hechos.

El carácter ~ (Alt+126) es un tipo de **restricción** que actúa sobre el valor que le sigue de tal modo que no permite este valor.

Otro tipo de restricción es la conectiva |. Mediante esta conectiva, para un mismo campo de un patrón, se producirá correspondencia cuando se de uno de los valores que aparecen relacionados mediante la conectiva.

Por ejemplo, una regla para sacar el paraguas podría ser condicionada por un tiempo lluvioso o muy nublado:

```
(defrule sacar-paraguas
        (tiempo lluvioso|muy-nublado)
=>
        (assert (sacar-paraguas))
)
```

El tercer tipo de restricción es la &. Esta restricción fuerza a que, para que se produzca la correspondencia del patrón, se cumplan todas las condiciones unidas mediante este símbolo:

Por ejemplo, si se quiere dar la razón por la cual se debe sacar el paraguas, la regla anterior quedaría:

```
(defrule sacar-paraguas
        (tiempo ?tiempo&lluvioso|muy-nublado)
=>
        (printout t "Saca el paraguas porque el tiempo esta " ?tiempo crlf)
        (assert (sacar-paraguas))
)
```

Comprobación de valores

Para comprobar el valor de números, variables y cadenas en la parte de la condición de las reglas se puede usar el elemento condicional **test**. Este elemento se usa **como un patrón** en la parte condicional de la regla. Este patrón corresponderá si el condicional que plantea se cumple.

Por ejemplo, en el ejercicio de las jarras, se puede poner

```
(defrule agregar-dos-litros
        (agregar 2)
        ?jarra<-(contenido jarra ?litros)
        (test (< ?litros 2))
=>
        (retract ?jarra)
        (assert (contenido jarra (+ ?litros 2)))
)
```

Se pueden usar los siguientes operadores:

Operador	Significado
eq	Igual (cualquier tipo)
neq	No igual (cualquier tipo)
=	Igual (tipo numérico)
<>	No igual (tipo numérico)
>=	Mayor o igual que
>	Mayor que
<=	Menor o igual que
<	Menor que

“eq” y “neq” se deben usar cuando se desconoce a priori el tipo de los datos que se comparan, ya que todos los demás operadores darán error si se compara un valor numérico con otro que no lo es.

Existen también operadores lógicos:

Operador	Significado
not	Negación lógica
and	Y lógico
or	O lógico

En ocasiones se pueden usar para el mismo contenido restricciones de campo u operadores lógicos. Por ejemplo, las dos reglas siguientes son equivalentes:

```
(defrule dos-o-tres
(or (contenido jarra 2) (contenido jarra 3))
=>
(printout t "Hay dos o tres litros" crlf)
)
```

```
(defrule dos-o-tres-bis
(contenido jarra 2|3)
=>
(printout t "Hay dos o tres litros" crlf)
)
```

Un uso típico del operador not es el indicar que se debe producir correspondencia cuando un cierto hecho no esté presente:

```
(defrule no-tiene-tres-litros
        (not (contenido jarra 3))
=>
        (printout t “La jarra no tiene tres litros” crlf)
)
```

Las restricciones en los campos se pueden usar en conjunción con la comprobación de valores, por ejemplo:

```
(defrule agregar-dos-litros
        (agregar 2)
        ?jarra<-(contenido jarra ?litros)
        (test (< ?litros 2))
=>
        (retract ?jarra)
        (assert (contenido jarra (+ ?litros 2)))
)
```

se podría haber escrito también:

```
(defrule agregar-dos-litros
        (agregar 2)
```

```
        ?jarra<-(contenido jarra ?litros&:(< ?litros 2))
=>
        (retract ?jarra)
        (assert (contenido jarra (+ ?litros 2)))
)
```

Cuando se va a poner una comprobación de valores (< ?litros 2) en un campo se debe usar el operador ":".

Manejo de archivos

En CLIPS se pueden realizar operaciones con ficheros. Antes de acceder a un archivo para lectura o escritura, éste debe ser abierto. La sentencia para abrir un archivo es **open**, mientras que para cerrarlo se usa **close**. CLIPS identifica al archivo mediante un nombre lógico, que es una variable global para todas las reglas.

(Las variables globales en CLIPS se definen mediante la instrucción **defglobal**)

Para leer la información del archivo se usan las sentencias read o readline, usando el nombre lógico del archivo del que tienen que leer.

Para leer varios símbolos con read o varias líneas con readline se debe usar un bucle, mediante la activación sucesiva de reglas o usando la sentencia while. Cuando se trata de leer más allá del final del fichero, CLIPS devuelve el símbolo **EOF** como resultado.

Por ejemplo, si se tiene un fichero llamado "nombres.txt" y se desea leer la primera línea del mismo e insertarla como hecho en la base de hechos, se pondría:

```
(defrule leer-nombres
=>
(open nombres.txt fichero)
```

```
(assert (nombre (readline fichero)))
(close fichero))
```

EJERCICIOS

1.- Ordenación de una lista de números

Se trata de ordenar una lista de números de mayor a menor; inicialmente se tendrá una lista con 8 números naturales colocados de forma aleatoria. El sistema debe hacer la ordenación de forma que al final quede en la base de hechos una lista con esos 8 números ordenados de mayor a menor, presentándose por pantalla el resultado.

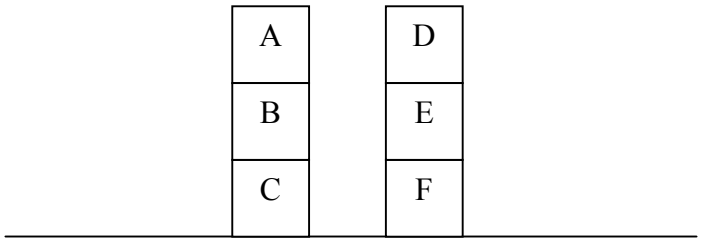
2.- El granjero, el lobo, la cabra y la col

En la orilla de un río se encuentran un granjero, un lobo, una cabra y una col. Hay una barca para cruzar al otro lado en la que como máximo pueden viajar dos de estos elementos, siendo obligatorio además que uno de ellos sea el granjero (el granjero puede viajar solo también).Dado el carácter depredador que tiene el lobo frente a la cabra y la cabra frente a la col, estas parejas no pueden estar a la vez en una orilla sin la presencia del granjero (el lobo comería a la cabra y la cabra a la col). Se pretende construir un sistema para comprobar cual es la secuencia válida de viajes que han de darse para que todos crucen al otro lado del río.

Una vez construída la base de hechos y las reglas del sistema, los movimientos que se produzcan en cada instante serán introducidos al sistema en forma de sentencias assert, debiendo cambiarse la base de hechos de acuerdo a las instrucciones recibidas. Si en algún momento lobo y cabra o cabra y col quedan en una misma orilla en ausencia del granjero, el programa debe indicar quién come a quién e impedir nuevos movimientos. Si los cuatro elementos cruzan a la orilla opuesta, se deberá indicar también con un mensaje.

3.- Mundo de bloques

Se tienen seis bloques apilados de tres en tres en dos columnas:



Los bloques pueden ser movidos por un robot que puede:

1. Poner un bloque de los que estén arriba del todo en el suelo.
2. Poner un bloque encima de otro.

Sólo se puede mover un solo bloque en cada operación.

Se desea un sistema que, partiendo de la posición inicial especificada, consiga una situación final en la que el bloque C esté encima del bloque E.