REPRESENTACIÓN DEL CONOCIMIENTO - CLIPS

Práctica 1

Objetivos:

1. Familiarización con el entorno de CLIPS. Manejo de Hechos y Reglas. Acciones de E/S
2. Diseño estructurado de hechos.
3. Expresiones y restricciones de campos. Evaluación de expresiones lógicas
4. Variables locales y globales
5. Definición de funciones
6. Prioridades en las reglas

CLIPS

(C Language Integrated Production System) : Lenguaje de especificación de sistemas basados en el conocimiento. El motor de inferencia de CLIPS es una implementación de un algoritmo de encadenamiento hacia delante. El sistema dispone de un intérprete que permite escribir comandos de manera interactiva, opciones de ejecución y depuración de programas, y puede mostrar información sobre el estado actual de la memoria de trabajo y las reglas activadas en cada momento.
En CLIPS se puede representar el conocimiento de tres formas distintas:

- Con Reglas que se prueban con objetos y hechos.
- Con Funciones y funciones genéricas, para conocimiento procedural.
- Utilizando Programación orientada a objetos, permitiendo la herencia y el polimorfismo.

Algunas características generales de CLIPS:

- Los comandos utilizan la notación tipo Lisp, por lo tanto hay que escribir las instrucciones **entre paréntesis**.
- Notación prefija (/ ((* 4 3) 2).
- Distinción entre mayúsculas y minúsculas.
- Funciones aritméticas: + - / * mod div sqrt round integer

1. HECHOS

En un SBR los hechos representan la información del mundo real que se conoce y la que se va obteniendo mediante la aplicación de reglas.

Un hecho puede constar de uno o varios campos. El primer campo suele representar una relación entre los restantes:

```
(<símbolo>      <datos>*)

(nombre         Alicia Mata Gonzalez)
(color          Especie1 Rosa)
(patas          Especie1 Largas)
```

También existen hechos estructurados que permiten agrupar en un mismo hecho distintas características o propiedades de una misma entidad.
Las principales operaciones que se pueden realizar con los hechos se resumen en la siguiente tabla con su orden correspondiente:

Tabla 1. Operaciones con hechos:

Definir plantillas para trabajar con hechos estructurados	**deftemplate**
Definir los hechos iniciales	**deffacts**
Insertar nuevos hechos	**assert**
Borrar un hecho	**retract (reset, clear)**
Modificar un hecho	**modify**
Duplicar un hecho	**duplicate**

A cada hecho se le asigna automáticamente un identificador único (fact-index): f-0, f-1, etc.
** No está permitido el anidamiento de hechos, lo siguiente no es correcto:

```
((persona (gaditana Maria))
```

a) Visualización estática:
- En una ventana aparte: en el menú principal Window - 1 Facts (MAIN)
- En la línea de comandos con **(facts)**.

b) Visualización dinámica: (**watch facts)** / **(unwatch facts)**
La doble flecha hacia la derecha ==> indica que un hecho se añade a la lista de hechos mientras que si está dirigida a la izquierda <== el hecho se está eliminando de la lista.

1.1 Definición de plantillas para disponer de hechos estructurados: DEFTEMPLATE

A través de las plantillas (*templates*) podremos definir hechos de forma estructurada. Las plantillas de definición son análogas a las estructuras de C (*struct*). El formato:

```
(deftemplate <nombre-entidad> [<comentario-opcional>]
             <definición-atributos>* )
```

La definición de los atributos se realiza con **slot (ó field)** o **multislot (ó multifield),** dependiendo de si el atributo va a tener uno o más argumentos.
Un slot puede tener tres características:

- **type** tipo**:** tipo de datos del slot { SYMBOL, STRING, INTEGER, }
- **allowed-values** lista_valores**:** limita los valores que se pueden introducir a los de la lista
- **default:** valor inicial o por defecto

Abre el editor de clips y guarda esta práctica como un archivo bat, que podrás ir probando de forma incremental cargando el archivo en el entorno con load o batch

EJERCICIO 1

```
(clear)
(reset)
(deftemplate persona
      (multislot nombre) ;para almacenar nombre y apellidos
```

```
      (slot dni (type INTEGER))
      (slot profesion (default estudiante))
      (slot nacionalidad (allowed-values Es Fr Po Al In It))
);persona

(deftemplate intelectual_europeo
     (multislot nombre)
     (slot dni)
      (slot idioma)
     (slot nacionalidad)
);intelectual

(list-deftemplates)  ;;para mostrar las plantillas definidas
```

1.2 Definición de los hechos iniciales: DEFFACTS

Los hechos iniciales se definen con **deffacts:**

```
(deffacts <nombre>
     <hecho>)
```

Continuando con el ejercicio 1 …

Hechos:
Juana Bodega Gallego con dni 3334444
Mario Canterio Cansino, con dni 122333 de profesión escritor y nacionalidad española.
Gertrudis es gaditana y Filoberto de Sevilla

```
(deffacts iniciales
    ;; con hechos estructurados
     (persona   (nombre Mario Cantero Cansino)
                (dni 122333)
                (profesion escritor)
                (nacionalidad Es))
      (persona (nombre Juana Bodega Gallego) (dni 3334444))
     ;; con hechos no estructurados
     (ciudad Gertrudis Cadiz)
     (ciudad Filoberto Sevilla)
);;iniciales

(facts)   ;; visualizar los hechos actuales
```

;; no verás nada porque para añadir los hechos iniciales hay que usar la orden **(reset)** que restaura las condiciones iniciales:

```
(reset)
```

```
(facts)
```

1.3 Inserción de hechos: ASSERT

La forma más simple y habitual de insertar hechos en la base de hechos se realiza mediante la orden **assert.**

```
(assert <hecho>*)
```

Continuando con el ejercicio 1 …

```
(assert (ciudad Juana Cadiz))
(assert (persona (nombre Alicia Mata Rueda) (dni 1234)))
(assert  (intelectual_europeo (nombre Victor Hugo) (idioma francés)))

(assert (persona (nombre Mateo Duran Barbera)
                 (dni 333221)))
```

1.4 Eliminación de hechos: RETRACT, RESET, …

La eliminación de un hecho concreto se realiza con **retract**: (* es un carácter comodín en retract)

```
(retract índice-hecho)
```

Continuando con el ejercicio 1 …

```
(facts)
(retract 1)
(facts) ;; comprueba que se ha eliminado el hecho 1
(retract 3 5)
(facts) ;; comprueba que se han eliminado los hechos 3 y 5
(retract *)
(facts)  ;; comprueba que se ha eliminado todo

(reset)

¿qué ha pasado después de ejecutar reset? Comprueba la lista de hechos de
la memoria de trabajo
```

Además es posible desde el entorno:

a) La eliminación de todos los hechos con **(reset),** que elimina también las activaciones de la agenda y restaura las condiciones iniciales definidas con deffacts**.**

b) La eliminación de todos los hechos y construcciones de la memoria de trabajo **(clear)**

1.5 Modificación de hechos: MODIFY

- **modify :** permite modificar los valores de los slots de una plantilla. Si no es un hecho estructurado (definido con plantilla) no funciona esta instrucción.

```
(modify  <indice-hecho>  <modificador>+)
```

Continuando con el ejercicio 1 …

```
(reset)
(modify  1 (nombre Maria Cantero Cansino)))
(facts)
```

1.6 Copiado de hechos: DUPLICATE

duplicate: duplica un hecho y lo actualiza con los valores de los slots que se especifiquen.

```
(duplicate  <indice-hecho>  <modificador >+)
```

Continuando con el ejercicio 1 …

```
(duplicate 2 (nombre Maria Bodega Gallego) (dni 11))

(facts)
```

2. REGLAS

Mediante las reglas se expresa el conocimiento para deducir nueva información (nuevos hechos).

```
(<Condición>*) => (<Acción>*)
```

Defrule, es la orden para crear una regla, indicando primero las precondiciones (todas las premisas deben cumplirse para activar una regla), a continuación los símbolos => y por último se especifican el conjunto de acciones que se derivan de las condiciones.

```
(defrule <nom_regla>
     <condición>*
```

```
    =>
        <acción>*)
```

Cuando las precondiciones de una regla se satisfacen, entonces se produce la activación de la regla, y se añade a la **agenda.** La agenda contiene el conjunto de activaciones producidas al realizar la equiparación de los hechos con las condiciones de una regla.

La orden **(agenda)** muestra las reglas que están preparadas para ser ejecutadas. (porque existen hechos que se equiparan con los antecedentes de las reglas, con la orden (run) se ejecutarían)

3. USO DE VARIABLES

Pueden usarse variables, entendiéndose que están cuantificadas universalmente. Las variables siempre comienzan con la interrogación final **?**:

```
?x    ?temperatura    ?altura  ?persona
```

Continuando con el ejercicio 1 …

Reglas del sistema:

```
;;Los gaditanos y los sevillanos son andaluces

(defrule R1_gaditanos
  (ciudad ?x Cadiz)
  =>(assert(comunidad ?x Andalucia))
);;R1
(agenda) ;; lista de reglas del sistema

(defrule R2_sevillanos
  (ciudad ?x Sevilla)
  =>(assert (comunidad ?x Andalucia))
);;R2

(reset)
(agenda) ;; lista de reglas del sistema

(run 1) ;; ejecuta el sistema para disparar la primera regla de la agenda
(facts) ;; comprueba si se han añadido nuevos hechos al sistema
(run 1) ;; siguiente regla de la agenda
(facts)
```

*** La ejecución de un programa finaliza cuando no hay más activaciones en la agenda.

*** El sistema se reinicia con (reset) y con (run)

***** Añade una nueva regla para inferir que los escritores españoles son intelectuales europeos**

```
(defrule R3_intelectuales
   (persona (nombre ?x ?y ?z) (profesion escritor) (nacionalidad Es))
   => (assert (intelectual_europeo (nombre ?x ?y ?z) (nacionalidad
española)))
);;R3
```

CLIPS está diseñado bajo el principio de **Refracción o Refractariedad:** cuando un conjunto de hechos satisfacen una regla, ésta sólo se dispara una vez. En caso contrario se tendría un bucle infinito de activación de las reglas hasta que los hechos desaparecieran de la lista de hechos.

*** Con la orden **refresh** se pueden recargar las reglas disparadas:

```
(agenda)
(refresh r1_gaditanos)
(agenda)
```

3.1 Comodines

Dentro de los hechos o patrones, un campo se puede sustituir por un comodín. Existen dos tipos de comodines:

- De un solo campo: **?**
- De varios campos: **$?** Representa a 0 o más casos de un campo.

Si una variable va precedida del símbolo de $ está haciendo referencia a varios valores de un hecho. Por ejemplo, para imprimir los nombres de los hermanos de Juan:

EJEMPLO

```
(assert (hermanos Juan Pedro Paula Patricia))

(defrule mostrarhermanos
      (hermanos Juan $?resto)
      =>
      (printout t "Los hermanos de Juan son " $?resto  crlf))

*** La regla del ejercicio 1 que hace referencia a los intelectuales
europeos podría haberse escrito usando comodines en las variables del
nombre y apellidos.

(defrule R3_intelectuales
   (persona (nombre $?n) (profesion escritor) (nacionalidad Es))
```

```
   => (assert (intelectual_europeo (nombre $?n) (nacionalidad española)))
);;R3
```

Tabla2. Otras órdenes del entorno útiles:

; punto y coma para comentarios que acaban con un retorno de carro

(dribble-on nombre-fichero) ; para grabar en un fichero la traza de ejecución
(dribble-off)

```
(list-defrules)
(list-deftemplates)
(list deffacts)

(undefrule <nombre_regla>)
(undeftemplate <nombre_plantilla>)
(undeffacts <nombre-hecho_inicial>)

(watch statistics) / (unwatch statistics)
(watch rules)/ (unwatch rules)
(watch activations)
(unwatch all)

(ppdefrule <nombre_regla>)
(ppdeftemplate <nombre-plantilla>)
(ppdeffacts <nombre-hecho inicial>)
```

3.2 Referencia a los hechos a través de una variable

En algunas situaciones puede resultar interesante guardar la dirección de un hecho, por ejemplo para borrarlo después. Con **<-** es posible asignar el identificador de un hecho a una variable.

EJERCICIO 2

```
(deffacts inicio
    (soltero Jose)
    (soltera Maria)
);;deffacts

(defrule casamiento
     (comprometidos ?x ?y)
     ?dir1 <- (soltero ?x)
```

```
    ?dir2<- (soltera ?y)
  => (assert (casados ?x ?y))
     (retract ?dir1)
     (retract ?dir2)
);;casamiento
```

Desde el entorno:

```
(reset)
(assert (comprometidos Jose Maria))

(facts)
(run)
```

*** Comprueba que ya no aparecen los hechos que afirman que Jose y Maria están solteros (pero aparecería que siguen comprometidos y casados) ya que comprometidos no se ha borrado

3.3 VARIABLES TEMPORALES

Para almacenar resultados temporales se utiliza la función **bind**, que asigna un valor a una determinada variable:

(bind <variable> valor)

```
(defrule areayperime
  (circulo ?radio)
=>(bind ?pi 3.14159)
   (printout t "Area del circulo " (* ?pi ?radio ?radio) crlf)
   (printout t "Perimetro " (* 2 ?pi ?radio) crlf ))

(assert (circulo 3))
```

3.4 VARIABLES GLOBALES

Las variables globales en CLIPS se denotan comenzando por interrogación y asterisco, y acaban en un asterisco: ?*nomvar*

```
(defglobal <var_global> = <valor> )
(defglobal  ?*saludo* = "hola"   ;; cadena de texto
            ?*verbo1*  = (create$ soy eres es somos sois son ser)) ;; valor multicampo
```

⇒ *Ver lista de órdenes para cadenas de texto y valores multicampo*

4. EXPRESIONES

4.1 ARITMÉTICAS

Las expresiones aritméticas siguen la notación prefija. Las operaciones se calculan de izquierda a derecha, no hay orden de precedencia. La precedencia se establece por la anidación de paréntesis:

- (+ 2 3 6 7 7)
- (/ ?Y ?X)
- (> (/ 5 2) 3)
- (* 3 4 5)

Calcular área de un rectángulo.

```
(deftemplate rectangulo
        (slot altura)
        (slot base))

(defrule calcular_area
    (rectangulo (altura ?altura) (base ?base))
=>(assert (area (* ?base ?altura))))
```

4.2 RESTRICCIONES DE CAMPO

Las restricciones de campo son expresiones lógicas que permiten establecer un filtro en los atributos de los hechos que se satisfacen en el antecedente de una regla:

Suponiendo la siguiente plantilla:

```
(deftemplate persona
    (slot nombre)
    (slot cabello (allowed-values rubio moreno castagno blanco negro))
    (slot ojos (allowed-values negros verdes azules)))
```

Podemos realizar los siguientes asertos para probar reglas sobre las restricciones de campo:

```
(assert (persona (nombre Claudia) (cabello rubio) (ojos azules)))
(assert (persona (nombre Juan) (cabello moreno) (ojos verdes)))
(assert (persona (nombre Elena) (cabello blanco) (ojos negros)))
(assert (persona (nombre Jorge) (cabello castagno) (ojos negros)))
(assert (persona (nombre Pascual) (cabello moreno) (ojos negros)))
```

Restricción Not : ~ Niega el resultado de la siguiente restricción

```
(defrule persona-no-moreno
        (persona  (nombre ?nombre)
```

```
        (cabello ~moreno))   ;;; restricción NOT
        =>
        (printout t ?nombre " no tiene el cabello moreno" crlf))
```

Restricción Or : | permite que varias opciones puedan coincidir como campo de un patrón

```
(defrule persona-morena-o-castagno
        (persona  (nombre ?nombre)
         (cabello castagno| moreno))  ;;; Castaño o Moreno
        =>
        (printout t ?nombre "  tiene el cabello oscuro" crlf))
```

Restricción And : & permite especificar más de una restricción a un atributo

```
(defrule persona-ni-morena-ni-castagna
        (persona  (nombre ?nombre) (cabello ?color&~castagno&~moreno))
        =>
        (printout t ?nombre "  tiene el cabello " ?color crlf))
```

Ejemplo: combinación de las restricciones de campo

```
(defrule comparacion-compleja-ojos-cabello
        (persona (nombre ?nomA)
                 (ojos ?ojosA &azules | verdes)
                 (cabello ?cabelloA &~negro))
        (persona (nombre ?nomB &~?nomA)
                 (ojos ?ojosB &~?ojosA)
                 (cabello ?cabelloB&blanco | ?cabelloA))
=>
(printout t ?nomA " tiene ojos " ?ojosA " y pelo " ?cabelloA crlf)
(printout t ?nomB " tiene ojos " ?ojosB " y pelo " ?cabelloB crlf))
```

4.3 TEST

Se usa en los antecedentes de las reglas para realizar comprobaciones lógicas: **TEST** junto con:

- Operadores de comparación: = , <>, <, >, <=, <= eq neq
- Funciones lógicas: **or not and**
- Funciones de predicado:

(**evenp** <arg>) número par
(**oddp** <arg>) número impar
(**floatp** <arg>)
(**integerp** <arg>)
(**lexemep** <arg>) símbolo o string
(**numberp** <arg>) float o entero
(**sequencep** (oddp <arg>)) valor de un campo múltiple
(**stringp** <arg>)
(**symbolp** <arg>)

(**defrule** testeando (dato ?x) (test (= ?x 8)) => (**printout t** ?x " es igual a 8 " **crlf**)) (assert (dato 8)) (assert (dato 18)) (run)	(**defrule** comprobando (respuesta ?res) (**lexemep** ?res) =>(**printout "**la respuesta es un símbolo o una cadena");

4.4 Otros operadores lógicos

Por defecto en el antecedente de las reglas, existe un AND implícito, que indica que para disparar una regla deben satisfacerse TODAS las condiciones del antecedente (incluyendo las restricciones de campo anteriormente estudiadas).

CLIPS permite además, incluir otros operadores para combinar la lista de hechos que una regla ha de satisfacer, tales como OR, NOT, LOGICAL, EXISTS, FORALL.

⇒ De entre ellos **evitaremos el uso del operador OR** que puede ser sustituido por 2 o más reglas independientes de forma que se controlen mejor los efectos laterales.

Por ejemplo:

(**defrule** abuelo (padre-de (padre ?a) (hijo ?x)) (**or** (padre-de(padre ?x)(hijo ?n)) (madre-de(madre ?x)(hijo ?n))) => (**printout t** ?a " es el abuelo de " ?n crlf)) ;; Esta regla se sustituye por 2 reglas independientes	(**defrule** abuelo_paterno (padre-de (padre ?a)(hijo ?x)) (padre-de (padre ?x)(hijo ?n)) => (**printout t** ?a " es el abuelo paterno de " ?n crlf)) (**defrule** abuelo_materno (padre-de (padre ?a)(hijo ?x)) (madre-de (madre ?x)(hijo ?n)) => (**printout t** ?a " es el abuelo materno de " ?n crlf))

5. DEFINICIÓN DE FUNCIONES

(**deffunction** <function-name> [comentario opcional]
(?arg1 ?arg2 ... [$?argN] ; lista de argumentos el último puede ser un argumento multicampo
(<accción 1>
<accción 2>
...
<acción K>) ; la única que puede devolver un valor, ninguna de las anteriores puede hacerlo

```
(deffunction hipotenusa  (?a  ?b)                          ; lista de argumentos
        (sqrt (+ (* ?a ?a)  (* ?b  ?b)) ) )

(defrule calculahipotenusa
        (dimensiones ?base ?altura)
=>      (printout t "   El valor de la hipotenusa es  "  (hipotenusa ?base ?altura)  crlf) )

(assert (dimensiones 3 4))
```

5.1 Return

(return [<expression>])

Ejemplo

```
(deffunction sign (?num)
        (if (> ?num 0)
           then (bind ?res 1)
           else (if (< ?num 0)
                   then (bind ?res -1)
                   else (bind ?res 0)))
        (return ?res))
```

```
CLIPS> (sign 5)
1
CLIPS> (sign -10)
-1
CLIPS> (sign 0)
0
CLIPS>
```

6. DECLARACIÓN DE LA PRIORIDAD EN REGLAS

Para establecer prioridades entre reglas que permitan resolver conflictos a la hora de determinar qué regla se ejecutará primero, se realiza mediante la orden:

(declare (salience <valor>))

El valor puede oscilar entre -10000 y 10000, por defecto las reglas tienen prioridad 0.
Así en el siguiente ejemplo, ¿Qué regla se ejecutará primero?

```
(defrule R1
⇨  (printout t "R1 ejecutándose"))

(defrule R2
    (declare (salience 10))
⇨  (printout t "R2 ejecutándose"))
```

EJERCICIOS PROPUESTOS

1. El Mapa

Diseña un SBR que permita contestar preguntas sobre la posición relativa de dos ciudades, con las siguientes características:

- Se introducirán exclusivamente hechos correspondientes a las relaciones “estar al norte de” y “estar al oeste de” y sólo entre las ciudades más próximas entre sí. Por ejemplo, si suponemos 9 ciudades distribuidas en una cuadrícula:

  A B C
  D E F
  G H I

  sólo se establecerán como hechos: “A está al norte de D”, “A está al oeste de B”,etc.

  ⇒ PISTA: utiliza hechos que podrían ser por ejemplo:

  (ubicación A Norte D)
  (ubicación A oeste B)

- El sistema de representación será capaz de inferir todas las relaciones inversas de las dadas directamente, es decir, las relaciones “estar al sur de” y “estar al este de”.

- Se inferirán nuevas relaciones por transitividad. Por ejemplo, sabiendo que “A está al norte de D” y que “D está al norte de G” se inferirá que “A está al norte de G”.

- Se inferirán las relaciones noroeste, noreste, suroeste y sureste a partir de los hechos iniciales. Por ejemplo, se podrá inferir que “C está al noreste de G”.

- El hecho que se utilizará para consultar al sistema será (**situación** <ciudad_1> <ciudad_2>). Cuando este hecho se inserta en el sistema, el mismo debe responder mostrando por pantalla la situación de la ciudad 1 con respecto a la ciudad 2. Esta podría ser la regla que activa el sistema y devuelve la ubicación:

```
(defrule inicio
        ?f1 <-(situacion ?x ?y)
        (ubicacion ?x ?u ?y)
=>
        (printout t ?x " esta al " ?u " de " ?y crlf)
        (retract ?f1)
);; inicio
```

2. El Concesionario

Una tienda de venta de automóviles tiene un portal que aconseja a sus clientes qué coche comprar en función de sus preferencias. La información sobre los modelos de coches que se pueden comprar se muestra en la siguiente tabla.

Modelo	Precio	Tamaño del Maletero	Número de Caballos	ABS	Consumo en Litros
Modelo1	12000	Pequeño	65	No	4,7
Modelo2	12500	Pequeño	80	Sí	4,9
Modelo3	13000	Mediano	100	Sí	7,8
Modelo4	14000	Grande	125	Sí	6,0
Modelo5	15000	Pequeño	147	Sí	8,5

El portal proporciona a los clientes un formulario con las siguientes preguntas:

1. Cantidad de dinero que desea gastarse
2. Maletero pequeño, mediano o grande
3. Mínimo número de caballos del motor
4. Sistema ABS
5. Consumo máximo de combustible a los 100 km

Si el usuario deja algún campo en blanco se asume lo siguiente:

- El precio del coche no debe superar los 13000 euros
- Maletero grande
- Mínimo 80 caballos
- Sistema ABS
- Consumo máximo de 8 litros

1. Diseña un SBR para recomendar un modelo de acuerdo a las preferencias del usuario, dichas preferencias serán introducidas desde el teclado usando instrucciones de tipo assert.