Sistemas Basados en Reglas. Introducción a CLIPS.

Una vez que hemos lanzado la aplicación CLIPS, deberás ver un cuadro de diálogo (prompt de un buffer) en el que puedes comunicarte con el intérprete.

Adición, eliminación y modificación de hechos

- (assert <hecho>+).
- (retract <índice>+).
- (modify <índice> (<nombre-atributo> <valor>)+).
- (duplicate <índice> (<nombre-atributo> <valor>)+).

Como introducir hechos:
(assert (Hoy es domingo))

Podemos ir completando la base de hechos, que podemos listar usando:
(facts)
(assert (Tiempo es soleado))
(facts)

Podemos eliminar hechos
(retract 1)

Comprobamos la base de hechos otra vez
(facts)

Aunque podemos destruir todos los hechos
(clear)

Podemos definir un conjunto de hechos de una vez
(deffacts hoy
	(Hoy es domingo)
	(Tiempo es soleado)
)

Cargar las condiciones iniciales:
(reset)
(facts)

El comando deffats es una expresión de tipo Lisp, primero el comando, después el nombre de la lista y después se definen los hechos.
Si queremos borrar la lista de hechos:
(undeffacts hoy)

Podemos ejecutar el comando deffacts en el buffer, sin embargo lo normal es cargarlo desde un archivo creado con algún editor.

(load "miarchivo.clp")

Una vez que se ha cargado hay que iniciar los hechos
(reset)

El comando (reset) quita cualquier hecho que exista en la base de hechos e inserta cualquier hecho asociado con todos los conjuntos de hechos definidos, además de añadir un hecho simple definido por el sistema.
<inicial-fact>
Esto se hace por conveniencia ya que comúnmente se suele empezar con una regla que reconoce este hecho y comienza ciclo de reconocimiento y disparo de reglas. Sin embargo no es obligatorio utilizarlo.

PLANTILLAS o hechos ordenados: (las especificaciones sobre los hechos estructurados las puedes mirar en el manual).

```
(deftemplate estudiante "definición de estudiante"
        (slot nombre (type STRING))
        (slot edad (type NUMBER) (default 22))
)

(deffacts estudiantes
        (estudiante (nombre "carlos") (edad 23))
        (estudiante (nombre "ana"))
)

(deftemplate persona (multislot nombre) (slot dni))

(deffacts ejemplo
       (persona (nombre Jose L. Perez) (dni 22454322))
        (persona (nombre Juan Gomez) (dni 23443325))
)
```

Reglas.

Para definir reglas en CLIPS se utiliza la siguiente sintaxis.

```
(defrule <nombre_regla>
        <comentario opcional>
        <declaración opcional>
        <primera premisa>
        <segunda premisa>
        ....
        <ultima premisa>
        =>
        <acción 1>
        ...
        <acción n>
)
```

Por ejemplo:

```
(defrule R1
        "Cosas que hacer en domingo"
        (Hoy es domingo)
        (Tiempo es soleado)
        =>
        (assert (Lavar coche))
        (assert (Ir al monte))
)
```

En esta regla las premisas son: (Hoy es domingo) y (Tiempo es soleado). Cuando los hechos casen con las premisas, la regla se lanzará y se insertarán dos nuevos hechos en la base de hechos (Lavar coche) e (Ir al monte).

```
(assert (Hoy es domingo) (Tiempo es soleado))
(watch rules)
(run)
```

EL termino (salience 10) indica lo importante que es la regla, que se utiliza en el caso de que varias reglas casen con los hechos, por lo tanto se provee de un mecanismo para la elección. A *salience* se le pueden asignar valores en el rango [-10000 10000]. Por defecto se asigna 0.

Regla para comenzar el proceso de inferencia:

```
(defrule start
        (initial-fact)
        =>
        (printout t "Hola mundo!" crlf)
)
(reset)
(run)
```

Variables.

Como en otros lenguajes, CLIPS tiene variables para almacenar valores. Por convenio, todo identificador de una variable es precedido por el carácter ?, como por ejemplo ?color, ?nombre, ?valor.
Un primer uso de las variables es ligar una variable a un valor en la parte izquierda de una regla y usar el valor posteriormente en la parte derecha.

```
(deftemplate personaje  (slot nombre) (slot ojos)(slot pelo))

(defrule busca-ojos-azules
                (personaje (nombre ?nombre) (ojos azules))
                    =>
                (printout t ?nombre " tiene los ojos azules." crlf))

(deffacts gente
(personaje (nombre Juan) (ojos azules) (pelo castagno))
(personaje (nombre Luis) (ojos verdes) (pelo rojo))
```

```
(personaje (nombre Pedro) (ojos azules) (pelo rubio))
(personaje (nombre Maria) (ojos castagnos) (pelo negro)))

(reset)
(run)
```

Si queremos eliminar reglas (undefrule *).

Algunas veces es útil comprobar la existencia de un valor en un atributo sin necesidad de ligar el valor a ninguna variable. Para ello utilizaremos el símbolo "?", que puede ser visto como un comodín que reconoce cualquier valor. Cuando se quiere un comodín que reconozca cero o más valores de un atributo multi-valor, se puede utilizar "$?". Por ejemplo:

```
(deftemplate persona (multislot nombre) (slot dni))

(deffacts ejemplo
  (persona (nombre Jose L. Perez) (dni 22454322))
  (persona (nombre Juan Gomez) (dni 23443325)))

(defrule imprime-dni1
   (persona (nombre ? ? ?Apellido) (dni ?dni))
  =>
  (printout t ?dni " " ?Apellido crlf))

(defrule imprime-dni2
   (persona (nombre $?nombre ?Apellido) (dni ?dni))
  =>
  (printout t ?dni " " ?nombre " " ?Apellido crlf))
```

Ejercicio 1.

Consideremos un conjunto de hechos que almacena información sobre los nombres de un conjunto de alumnos y las notas que tienen en cierta asignatura:

```
(deftemplate alumno
    (slot nombre)
     (slot nota))

(deffacts notas-alumnos
  (alumno (nombre Ana) (nota 7))
  (alumno (nombre Eva) (nota 3))
  (alumno (nombre Ivan) (nota 4))
  (alumno (nombre Iker) (nota 6))
  (alumno (nombre Jesus) (nota 1))
  (alumno (nombre Ana) (nota 7))
  (alumno (nombre Jon) (nota 5))
  (alumno (nombre Chesko) (nota no-presentado)))
```

Escribir las reglas necesarias para presentar en pantalla el número total de alumnos no presentados, el número total de alumnos aprobados, el número total de alumnos suspensos y la nota media calculada sobre los alumnos presentados a examen.
Puede ser necesario incluir en la base de conocimientos, hechos que sirvan para almacenar los datos que se quieren ir contabilizando.
Construir una regla por cada situación que se pueda presentar (es decir, una para los alumnos no presentados, otra para los alumnos aprobados y otra para los alumnos suspensos) de manera que se modifiquen de forma adecuada los hechos que sirven para almacenar los datos que se están contabilizando.
Finalmente construir una regla que presente en pantalla los datos contabilizados cuando ya no queden por procesar datos sobre los alumnos.

Restricciones.

Podemos representar patrones que reconozcan hechos que no tengan un determinado valor en un atributo, o que tengan alguno de los valores que se especifican. Los operadores operador ~, | y & cumplen este propósito. Por ejemplo
(persona (nombre ?nombre) (pelo ~rubio))

reconocerá a toda persona con pelo que no sea rubio, y

(persona (nombre ?nombre) (pelo castagno | pelirrojo))

reconocerá a toda persona con el pelo castaño o pelirrojo.

El operador & se utiliza en realidad en combinación con los anteriores para ligar valor a una variable. Por ejemplo:

```
(defrule pelo-castagno-o-rubio
  (persona (nombre ?nombre) (pelo  ?color & rubio|castagno))
   =>
  (printout t ?nombre " tiene el pelo " ?color crlf))
```

También es posible expresar predicados sobre el valor de un atributo directamente en el patrón. Por ejemplo:

```
(defrule mayor-de-edad
  (persona (nombre $?nombre) (edad ?edad &:(> ?edad 18))
   =>
  (printout t ?nombre " es mayor de edad." crlf))
```

Para indicar que el valor de un atributo debe ser igual al resultado de la evaluación de una función se indica mediante el operador =. Por ejemplo, para reconocer las personas con 2 años más que la mayoría de edad se puede expresar mediante el patrón:

(persona (edad =(+ 18 2)))

Función test.

Un elemento condicional puede evaluar una expresión, en lugar de representar un patrón que debe ser reconocido. Por ejemplo, para comprobar que la variable ?edad es mayor que 18 se puede representar mediante

(test (> ?edad 18)).

Función bind.

Si queremos ligar un valor a una variable para utilizarlo en varios sitios sin tener que recalcular el valor podemos hacerlo con la función bind. Por ejemplo:

(bind ?suma (+ ?a ?b))

Combinación de elementos condicionales con And, Or, y Not.

Para que una regla esté activada todos los elementos condicionales de la precondición deben reconocer hechos de la memoria de trabajo. Por lo tanto, hasta ahora todas las reglas vistas tienen un **and** implícito entre los elementos condicionales. CLIPS ofrece la posibilidad de representar **and** y **or** explícitos, así como la posibilidad de indicar mediante **not** que una regla será activada si no existe ningún hecho reconocido por un patrón.
En el siguiente ejemplo, en lugar de tener dos reglas que tienen la misma acción, se puede representar mediante una única regla que se activará si existe una emergencia de tipo inundación o si se han activado los aspersores del sistema de extinción de incendios.

```
(defrule desconectar-sistema-eléctrico
  (or (emergencia (tipo inundacion))
      (sistema-extincion-incendios (tipo aspersor-agua) (estado activado)))
  =>
  (printout t  "Desconectado sistema electrico." crlf))
```

La siguiente regla se activa en el caso de que no haya dos personas que cumplan los años el mismo día:

```
(defrule no-fecha-identica
  (not (and (persona (nombre ?nombre)
                     (cumpleagnos ?fecha)
            (persona (nombre ~?nombre)
                     (cumpleagnos ?fecha)))
  =>
  (printout t  "No hay dos personas con el mismo cumpleagnos." crlf))
```

EJERCICIO 2.

Las reglas que están activadas son incluidas en la agenda. En principio todas las reglas de la agenda serán ejecutadas, pero la ejecución de una regla puede desactivar reglas de la agenda al modificar hechos de la memoria de trabajo. Además, si hacemos la comparación con un problema de búsqueda, los nodos a expandir son las reglas en la agenda. Desde este punto de vista, la ejecución de una regla da lugar a nuevas ramas (nuevas instancias de reglas en la agenda). Por todo lo dicho anteriormente será importante el orden en el que se ejecuten las reglas de la agenda. Los criterios de ordenación de la agenda constituyen la estrategia de control.
CLIPS ofrece varias estrategias para ordenar la agenda:

- depth.
- breadht.
- LEX.
- MEA.
- complexity.
- simplicity.
- random.

Por defecto CLIPS utiliza la estrategia de búsqueda en profundidad.
El resto de estrategias, salvo random que produce una ordenación aleatoria, se basan en una combinación de los siguientes criterios:

- **Refracción**: Una regla instanciada no puede ser disparada más de una vez. De esta forma se evitan bucles.
- **Novedad (Recency):** Tienen prioridad las instancias de reglas con hechos más recientes.
- **Especificidad**: Tienen prioridad las reglas que tienen mayor número de patrones.
- **Prioridad (*Salience*):** Asignación explícita de una prioridad a las reglas.

En CLIPS la prioridad por defecto es 0. Se puede definir un valor de prioridad entre -10000 y +10000 declarándola dentro de la regla. Por ejemplo:
(declare (salience 100)).

Usando el conjunto de reglas utilizado en clase de teoría:
R1: IF B AND C THEN F.
R2: IF D AND G THEN A.
.
.
R9: IF X AND B THEN D.

Siendo el OBJETIVO = H y la base de hechos: B, C.

Escribir el programa y ejecutarlo con las diferentes estrategias de ordenación de la agenda. Ejecutar el programa paso a paso (Ctrl + T) visualizando la (agenda).

FUNCIONES.

Una función en CLIPS tiene una apariencia Lisp, con la importante diferencia que las variables deben empezar con un prefijo ?, como vemos en el ejemplo:

```
(deffunction hipotenusa (?a ?b)
  (sqrt (+ (* ?a ?a) (* ?b ?b)))
)
```

El valor de la última expresión es el devuelto por la función. Podemos definir además funciones que sirvan para ejecutar acciones.

```
(deffuntion iniciar (?dia)
   (reset)
   (assert (Hoy es ?dia)
)
```

Las estructuras de control que se pueden usar en las funciones son las típicas, loop, while, if. Puedes encontrar más información en el manual de CLIPS.

ENTRADA DE DATOS.

Puedes leer las entradas de teclado:

```
(bind ?resp (read))
```

Hasta que se pulsa la tecla *enter*.

```
(bind ?resp (readline))
```

```
(deffunction comienza ()
   (printout t "Comienza el sistema identificador de animales." crlf)
   (printout t "¿ Es muy grande ?")
   (bind ?resp (read))
   (assert (pregunta Es-grande ?resp))
)

(defrule start
        =>
        (comienza)
)
```

IDENTIFICADORES DE HECHOS.

Se puede guardar el identificador de un hecho en una variable local usando el operador <- . De esta manera podremos obtenerlos de la lista de hechos y modificarlos.

```
(deffacts solteros
        (soltero Juan Sanchez)
        (soltero Roberto Gomez)
)
```

```
(defrule inicio
        (intial-fact)
        ¿solt <- (soltero $?nombre)
        =>
        (retract ¿sol)
        (assert (casado ?nombre))
        (printout t ¿nombre “se ha casado”)
)
```

VARIABLES GLOBALES.

Las variables globales permiten almacenar valores que son accesibles por todas las reglas y funciones de la base de conocimientos.

Necesitan ser declaradas.

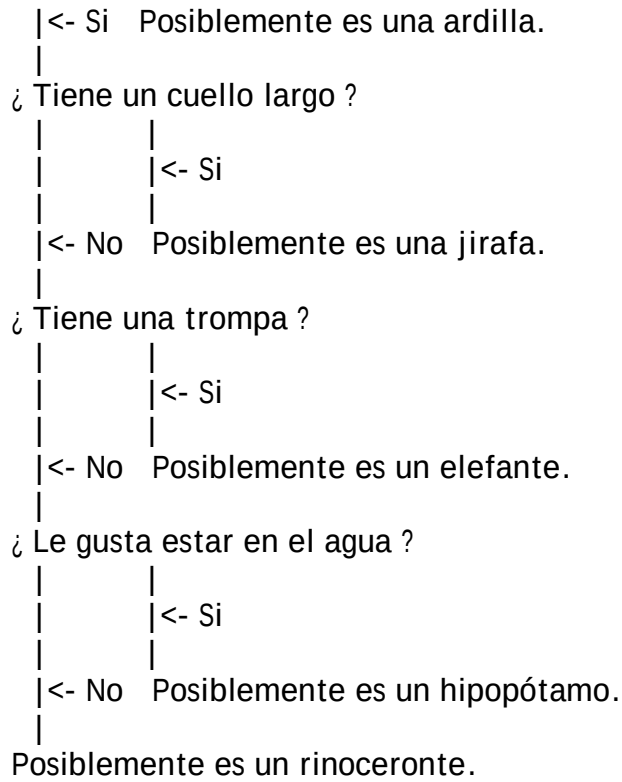
```
(defglobal
        ?*nombre* = Miguel
)

(deffacts profesores
        (profesor Miguel Pagola)
        (profesor Humberto Bustince)
)

(defrule inicio
        (initial-fact)
        (profesor ?nom ?ape)
        (test (eq ?*nombre* ?nom))
        =>
        (printout t “El profesor de practicas es:” ?nom “ “?ape crlf)
)
```

EJERCICIO 3.

Convertir el siguiente árbol de decisión para la clasificación de animales en un conjunto de reglas CLIPS. Crear condiciones que equiparen con hechos de la forma (pregunta <pregunta> <respuesta>).

Árbol de decisión:

```
¿ Es muy grande ?
  |         |
  |         |<- No
  |         |
  |       ¿ Emite chillidos ?
  |         |         |
  |         |         |<- Si
  |         |         |
  |         |<- No   Posiblemente es un ratón.
  |         |
```



Cargar el programa en CLIPS y utilizar el conjunto de reglas construido para averiguar qué tipo de animal puede ser aquel que es grande, no tiene el cuello largo y tiene trompa. Examinar y explicar las activaciones de las reglas.