SISTEMAS BASADOS EN EL CONOCIMIENTO

Oscar García-Olalla

ogaro@unileon.es

CLIPS

Introducción

Hechos

Atajos de teclado

Reglas

Cómo ejecutar un programa

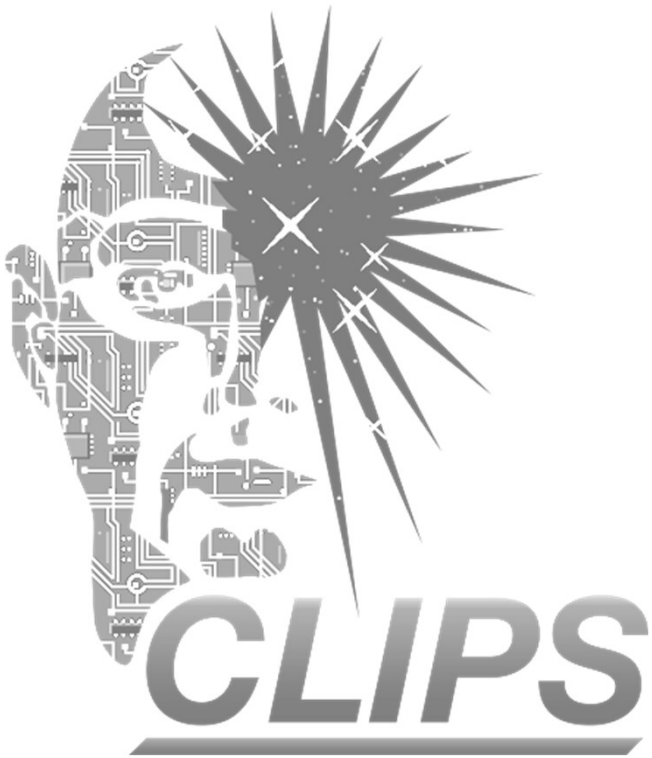
Cómo salvar y cargar programas

Comodines

Variables

Ejercicios prácticos

INTRODUCCIÓN A CLIPS



- ¿Qué es CLIPS?
- ¿Qué es un Sistema Experto?
- Entorno de desarrollo CLIPS.
- Creado en 1980 por la NASA.

HECHOS

- Datos del problema, tanto en el momento inicial como en cualquier momento de su ejecución.
- Para introducir hechos se utiliza el comando **assert**

  CLIPS>(assert (color tomate rojo))
- Se pueden introducir varios hechos en una misma función.

  CLIPS>(assert (color pera verde)(color manzana amarillo))
- Se pueden mostrar los hechos presentes actualmente con el comando **facts**

  CLIPS>(facts)

HECHOS

- Para eliminar hechos se utiliza el comando **retract** seguido del número de hecho a eliminar.

  CLIPS>(retract 1)

- Si queremos reiniciar la base de conocimiento de hechos a su estado por defecto, usaremos el comando **reset**.

  CLIPS>(reset)

HECHOS

- Los hechos aceptan diferentes tipos de datos: integers, floats, symbols, strings…
  - Integers: 23, 42, 5, -9, 146…
  - Floats: 2.54, -5,467, 3,1415…
  - Symbols: teclado, pizza, cliente…
  - String: "Nuevo pedido", "Cadena"…

- El primer campo de un hecho, debe ser un símbolo.

HECHOS

- Hechos ordenados y hechos no ordenados.
- Para fijar el estado por defecto de los hechos utilizamos el comando **deffacts**.

```
CLIPS>(deffacts hechoIniciales
                (color luz rojo)
                (longitud carretera 200))
```

- Como ya hemos visto, el comando **reset** reinicia el estado de los hechos a su estado por defecto.

ATAJOS DE TECLADO DE CLIPS

- CTRL+N: Abre nueva ventana de editor.
- CTRL+J: Autocompleta un comando.
- CTRL+E: Ejecuta un comando reset.
- Otros que veremos más adelante.

REGLAS

- Los hechos son datos estáticos. Las reglas representan el conocimiento del experto que permite alterar los hechos.
- El funcionamiento de una regla es sencillo: Si se cumplen ciertas condiciones, se ejecutan ciertos comandos.
- Para crear una regla se utiliza el comando **defrule**
- Una regla está formada por un elemento separador =>. A la izquierda del separador estarán las condiciones, y a la derecha las acciones a realizar si dichas condiciones se cumplen.


```
(color luz rojo)
(puedePasar no)

(defrule comprobarPaso
        (color luz verde)
        =>
        (assert(puedePasar si))
)
```

REGLAS

- Las reglas se activan cuando las condiciones de la izquierda coinciden con hechos presentes.
- En el caso anterior, al introducir un hecho (color luz verde) la regla se activaría y automáticamente introduciría otro hecho (puedePasar sí)
- Nótese que en este caso no se están eliminando los hechos anteriores, por lo que tras la ejecución de la regla nuestra base de hechos quedaría así:

  (color luz rojo)
  (puedePasar no)
  (color luz verde)
  (puedePasar si)

REGLAS

- Como nuestra regla anterior añade un nuevo hecho, esto podría desencadenar la activación de otra regla cuya condición de ejecución fuese (puedePasar si).
- Dado que las reglas modifican hechos y esto desencadena la ejecución de otras reglas, podemos controlar el orden en el que las reglas simultáneas se van a ejecutar asignándoles una prioridad. Para ello se utiliza el comando **salience**.
- La prioridad varía entre -10.000 y 10.000.

```
(defrule comprobarPaso
        (declare (salience 1))
        (color luz verde)

        =>
        (assert(puedePasar si)))
```

- Si se declara una regla con el mismo nombre que otra, la nueva regla sustituirá a la anterior.

EJECUCIÓN

- Para ejecutar el programa utilizaremos el comando run.

  CLIPS>(run)

- Podemos limitar el número de reglas que queremos ejecutar indicándoselo al comando. Por ejemplo, para limitar el número de reglas a ejecutar a 3:

  CLIPS>(run 3)

- Para ir ejecutando el programa regla a regla se utiliza el atajo CTRL+T. Esta es la manera recomendada de hacerlo durante las prácticas.

EJECUCIÓN

- Para mostrar texto de salida por pantalla se utiliza el comando printout.

  CLIPS>(printout t "Hola" crlf)

- La letra t, indica que la salida ha de ser por pantalla. El símbolo crlf representa un retorno de carro.
- La instrucción printout se suele usar como parte de las reglas para comprobar visualmente el resultado de su ejecución.

SALVAR Y CARGAR PROGRAMAS

- Los programas constarán de una parte inicial **deffacts** donde se indicarán los hechos por defecto en el momento de partida.
- A continuación aparecerán definidas todas las reglas que se hayan establecido.
- Los programas se salvarán con extensión .clp
- Antes de cargar un programa, utilizaremos el comando (clear) para limpiar la memoria previa de CLIPS.
- Cargaremos nuestro programa usando el comando load del menú.
- IMPORTANTE: Si hay algún error de sintaxis al cargar, CLIPS lo indicará mostrando FALSE en pantalla.
- Ejecutaremos el comando (reset) para inicializar la base de conocimiento de hechos a los indicados por el deffacts de nuestro programa.

EJEMPLO DE PROGRAMA

```
(deffacts hechos-iniciales
                        (color luz rojo)
                        (puedePasar no))

(defrule comprobarPaso
        (color luz verde)

        =>
        (assert(puedePasar si))
        (printout t "La luz está verde, ahora se puede pasar" crlf))
```

COMODINES

- El carácter ? se puede utilizar como comodín simple dentro de un patrón, de forma que represente cualquier valor dentro del patrón.

```
(huesped Benedict Cumberbatch)
(huesped Tom Cruise)
(huesped Tom William Hidelson)
(huesped Tom Stanley Holand)

(defrule existeTom
                (huesped Tom ?)
                =>
                (printout t "Existe algún huésped
llamado Tom" crlf))
```

- Cabe destacar que el patrón que activa esta regla solo tiene tres campos: "huésped", "Tom" y "?", por tanto solo se activaría con el hecho (huésped Tom Cruise), ya que los otros dos Toms, son hechos de 4 campos.

COMODINES

- Si quisiésemos activar la regla con los otros dos Toms, tendríamos que haber puesto algo así:

```
(defrule existeTom
        (huesped Tom ? ?)
        =>
        (printout t "Existe algún huésped llamado Tom" crlf))
```

- Y si buscamos un huesped cuyo middle-name sea William, entonces sería algo así:

```
(defrule existeWilliamMiddle
        (huesped ? William ?)
        =>
        (printout t "Existe algún huésped cuyo middle-name es
William" crlf))
```

COMODINES

- También existe el comodín múltiple $? (Colocando el símbolo $ delante de la ?).
- Mientras que el comodín simple representa un campo con cualquier valor, el comodín múltiple representa cero o más campos con cualquier valor.
- En este caso podríamos comprobar si existe algún Tom (independientemente de si contiene middle-name o no) de la siguiente manera.

```
(defrule existeTom
        (huesped Tom $?)
        =>
        (printout t "Existe algún huésped llamado Tom" crlf))
```

VARIABLES

- Sirven para almacenar el valor por el que ha sido sustituido un comodín al ser emparejado por un patrón en la parte izquierda de una regla, para poder trabajar con dicho valor en la parte derecha.
- Para utilizar una variable, basta con poner un nombre al comodín de la siguiente manera: **?nombreDeLaVariable**
- Veamos un ejemplo de uso de variables sobre el ejemplo anterior.

VARIABLES

```
(huesped Benedict Cumberbatch)
(huesped Tom Cruise)
(huesped Tom William Hidelson)
(huesped Tom Stanley Holand)
(defrule imprimirApellido
        (huesped Tom ?apellido)
        =>
        (printout t "Existe algún huésped llamado Tom, y se apellida"?apellido crlf))
```

VARIABLES

- En el ejemplo anterior hemos usado una variable simple mediante un comodín simple. También podemos usar variables múltiples de la misma manera $?nombreDeLaVariable

```
(defrule imprimirApellido
        (huesped Tom $?apellido)
        =>
        (printout t "Existe algún huésped llamado Tom, y se
apellida"?apellido crlf))
```

VARIABLES

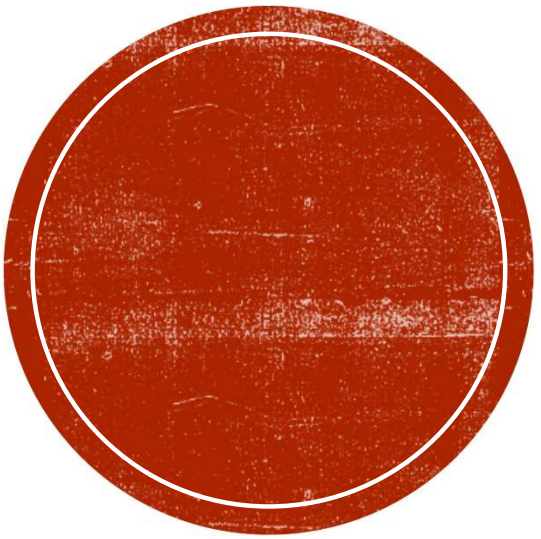
- Es importante destacar que el valor de una variable solo se asigna la primera vez que se utiliza. El resto de las veces será sustituida por el primer valor que se le haya asignado.

- Un uso muy común de variables es eliminar o modificar hechos. Ejemplo:

```
(huesped Benedict Cumberbatch)
(huesped Tom Cruise)
(huesped Tom William Hidelson)
(huesped Tom Stanley Holand)

(defrule eliminar
        ?huesped <- (huesped Tom Cruise)
        =>
        (retract ?huesped)
```

¿CÓMO AFRONTAR UNA PRÁCTICA?



¿CÓMO AFRONTAR UNA PRÁCTICA?

¿CÓMO AFRONTAR UNA PRÁCTICA?