TUTORIAL DE CLIPS

VERSION 0.8 / 9 de marzo de 2000

Este documento está destinado a los alumnos de tercer curso de las Ingenierías Técnicas de Informática de la Universidad de Murcia.

El objetivo del documento es que el alumno conozca una de las numerosas *shells* que existen para la construcción de Sistemas Expertos. Más concretamente el introducirlo en una de las más conocidas herramientas basadas en reglas: CLIPS.

NOTA: No se pretende con este documento realizar un manual completo y detallado sobre esta herramienta. Para una documentación completa recurra a los manuales existentes de CLIPS.

Luis Daniel Hernández Molinero
Dpto. de Informática, Inteligencia Artificial y Electronica
Fac. Informatica Universidad de Murcia.
Campus de Espinardo. 30071 Murcia. España
e-mail: ldaniel@dif.um.es

Índice General

Lección 1

INTRODUCCIÓN A CLIPS

CLIPS son las iniciales de *C Language Integrated Production System* y es una herramienta para la construcción de sistemas expertos. Es decir, es una herramienta diseñada para el desarrollo de software que requiere de conocimiento humano. Los creadores de CLIPS es la NASA, y hoy en día está siendo utilizado en la industria, gobierno y educación. La versión más reciente es la 6.0 que soporta los paradigmas de programación procedural y orientado a objetos. Se puede obtener la última información referente a CLIPS en la 'CLIPS home page' cuya dirección es http://krakatoa.jsc.nasa.gov/~clips/CLIPS.htm.

El conocimiento humano se implementa en CLIPS mediante:

- Reglas, que se formulan a partir del conocimiento heurístico basado en la experiencia.
- Deffunction, o funciones generalizadas, que se formulan a partir de conocimiento procedural.
- Programación orientada a objetos, que también se formula por conocimiento procedural pero formulado en términos de las 5 características (generalmente aceptadas) de la programación orientada a objetos: clases, paso de mensajes (en ingles, *message-handlers*), abstracción, encapsulamiento, herencia y polimorfismo.

Con CLIPS se puede desarrollar software formado sólo por reglas, sólo por objetos, o mezcla de reglas y objetos. Además

- CLIPS se ha diseñado para poder ser integrado con otros lenguajes como C y Ada,
- puede llamarse desde otros lenguajes para que CLIPS desarrolle la función y retorne la salida y el control al programa que lo llamó,
- el conocimiento procedural puede definirse como funciones externas, ser llamadas por CLIPS y retornar la salida y el control a CLIPS.

Lección 2

ELEMENTOS BÁSICOS DE CLIPS

CLIPS proporciona tres elementos básicos para escribir programas: algunos tipos de datos primitivos, funciones para manipularlos y constructores para añadirlos a la base de conocimiento.

2.1 Tipos de Datos

Los tipos de datos que proporciona CLIPS son: reales (*float*), enteros (*integer*), símbolos (*symbols*), cadenas (*strings*), direcciones externas (*external-address*), direcciones de hechos (*fact-address*), nombres de instancias (*instance-name*) y direcciones de instancias (*instance-address*).

➠ Un **número** consta de los siguientes elementos: los dígitos (0-9), un punto decimal (.), un signo (+ o -), y, opcionalmente una notación exponencial (e) con su correspondiente signo. Los números se almacenan en CLIPS como reales (*float*) o enteros (*integer*). Un número se interpreta que es entero (*integer*) si consta de un signo (opcional) seguido de sólo dígitos – internamente se representa como un *long integer* de C. Cualquier otro número se interpreta como un real (*float*) – internamente se representa como un *double float* de C. El número de dígitos significativos y los errores de redondeo que puedan producirse dependerá del ordenador que se esté utilizando.

Ejemplos de Enteros	Ejemplos de Reales
2341234	837e7
93	121.43
+8233482	+2e10
-283	-3.14

➠ Un **símbolo** en CLIPS es cualquier secuencia de caracteres que no sigue exactamente el formato de un número. Más concretamente empieza con cualquier caracter ASCII imprimible y finaliza con un delimitador (caracteres ASCII no imprimibles). Los caracteres no imprimibles son: espacios, tabulaciones, retornos de carro, "line feeds", doble comillas, '(', ')', '&', '|', '<' y '~'. El carácter ';' también actua como delimitador y es entendido por CLIPS que lo que se encuentre a partir de ahí hasta el final de linea es un comentario. CLIPS también distingue entre mayúsculas y minúsculas.

Ejemplos de Símbolos
Hola
DNI4453
Otro_simbolo
988AB

➠ Un **string** es un símbolo que empieza y termina por dobles comillas. En el caso de que se desee que el string contenga las dobles comillas, antes se colocará el backslash (\). Notar que no es lo mismo abc que "abc" ya que aunque ambos contiene los mismos caracteres imprimibles, son tipos diferentes: el primero es un símbolo y el segundo es un string.

Ejemplos de Strings
"Un string"
"Este_Tiene aqui \" una comilla"

➠ Una **dirección externa** es la dirección de una estructura de datos externa devuelta por una función escrita en C o Ada y que ha sido integrada con CLIPS. Este tipo de datos sólo puede crearse mediante la llamada a la función. La imprisión de una dirección externa es de la forma *<Pointer-XXX>* donde *XXX* es la dirección externa.

➠ Un **hecho** es una lista de uno o más valores que o bien son referenciados por su posición (para hechos ordenados) o por un nombre (para hechos no ordenados). La impresión de la **dirección de un hecho** es de la forma *<Fact-XXX>* donde *XXX* es el índice que ocupa el hecho en la memoria de CLIPS.

➠ Una **instancia** es un caso particular de un objeto de CLIPS. Los objetos de CLIPS son números, símbolos, strings, valores multicampo, direcciones externas, direcciones de hechos e instancias de una clase definida por el usuario. Una clase definida por el usuario se crea mediante la instrucción ***defclass***. Y una instancia de la clase construida se hace con la función ***make-instance***. Un **nombre de instancia** (*instance-name*) se construye encerrando entre corchetes un símbolo. Así, los símbolos puros no pueden empezar con corchetes. Es importante indicar que los corchetes no forman parte de la instancia, sino que los corchetes sólo se utilizan para indicar que el símbolo que se encuentra en su interior es la instancia.

Ejemplos de Instancias
[UnaInstancia]
[Otro_Instancia]
[9348-232]
[++otro++]

Las **direcciones de las instancias** (*instance-address*) se presentan en el formato *<Instance-XXX>* donde *XXX* es el nombre de la instancia.

➠ Por **campo** o **casilla** se entiende cualquier lugar que puede tomar un valor (de los tipos de datos primitivos) en una sentencia. De esta forma, atendiendo al número de campos que aparece en una sentencia se distinguen dos tipos de valores:

- ➪ *Valores uni-campo*: los formados por tipos de datos primitivos. En particular una constante es un valor uni-campo que no varía y está expresado como una serie de caracteres.
- ➪ *Valores multi-campo*: los formados por una secuencia de cero o más valores uni-campo. Cuando CLIPS muestra los valores multicampo, éstos se muestran entre paréntesis.

Ejemplos de Valores Multi-campo
( )
(x)
(hola)
(relaciona "rojo" 23 1e10)

Notar que no es lo mismos el valor uni-campo *hola* que el valor multicampo *(hola)*.

2.2 Funciones

Una función es la codificación de un algoritmo identificado con un nombre que puede o no devolver valores útiles a otras partes del programa. Existen dos grandes grupos de funciones:

- Funciones definidas por el sistema. Son aquellas que están implementadas en CLIPS.
- Funciones definidas por el usuario. Aquellas que se han escrito en un lenguaje distinto a CLIPS (p.e. C o ADA).

Los comandos que permiten construir funciones en CLIPS son:

deffunction Permite construir funciones en el ambiente de CLIPS usando funciones de CLIPS. Las llamadas a las funciones en CLIPS se hacen en notación prefija. Es decir, en primer lugar se escribe el nombre de la función y a continuación los argumentos de dicha función separados por uno o más espacios, todo ello encerrado entre paréntesis. La siguiente tabla muestra distintas llamadas usando las funciones suma y multiplicación:

Ejemplos de llamadas a las funciones + y *
(+ 2 3.5 9)
(* 4.454 1.34)
(+ 2 (* 3 4) 5)

defgeneric y **defmethod** Permiten definir funciones genéricas. Éstas permiten diferente tipo de código dependiendo de los argumentos pasados a la función genérica.

2.3 Constructores

Los constructores son aquellas sentencias de CLIPS que permiten crear objetos. La llamada a un constructor se realiza siempre entre paréntesis y suele comenzar con la palabra **def**. Los constructores, que se estudiarán con detenimiento más adelante, son: **defmodule**, **defrule**, **deffacts**, **deftemplate**, **defglobal**, **deffunction**, **defclass**, **definstances**, **defmessage-handler**, **defgeneric** y **defmethod**.

Lección 3

ABSTRACCIÓN DE DATOS

Existen tres formatos para representar información en CLIPS: hechos, objetos y variables globales.

3.1 Hechos

Un **hecho** (*fact*) representa un trozo de información que se almacena en la llamada **lista de hechos** (*fact-list*). A cada hecho en la lista se le asocia un identificador (*fact identifier*) que no es más que un índice asociado a ese hecho en la lista. Cuando a CLIPS se le pide que muestre el identificador de un hecho, éste se muestra de la forma *f-XXX*, donde *XXX* denota al índice asociado. Por ejemplo, f-3 se refiere al hecho que tiene el índice 3.

Se ditinguen dos tipos de hechos: los hechos ordenados y los hechos no ordenados.

- **Hechos Ordenados:** Son los formados por un símbolo seguido de cero o más campos separados por espacios y todo ello delimitado por paréntesis.
  Un modo fácil de entender este tipo de hechos es interpretando el primer campo, que es un símbolo, como una relación y el resto de los campos como los términos que se relacionan (vía esa relación).
  Como es conocido, en una relación el orden de los términos que interviene es importante. No es lo mismo decir que *Luis es hijo de Daniel*, que decir *Daniel es hijo de Luis*. Un modo de representar estos dos hechos es como sigue:
  (hijo "Luis" "Daniel") (hijo "Daniel" "Luis")
  Es importante remarcar que los hechos ordenados codifican la información según la posición, por lo que el usuario, cuando accede a la información, no sólo debe saber qué datos están almacenados, sino también qué campos contienen esos datos. Para evitar este problema, se recurre a los hechos no ordenados.

- **Hechos No Ordenados** (o *deftemplate*). Son aquellos que asignan un nombre a cada campo del hecho. La diferencia con los hechos ordenados es que ahora cada campo contiene un nombre (el del campo) y el valor que toma ese campo, y todo ello entre paréntesis.
  Por ejemplo, la situación de parentesco *Daniel es hijo de Luis* puede representarse como:
  (parentesco (padre "Luis") (hijo "Daniel"))
  (parentesco (hijo "Daniel") (padre "Luis"))
  Notar que ahora el orden de los campos en un *deftemplate* no es importante. Los dos hechos anteriores son equivalentes.

El constructor **deffacts** permite establecer un conocimiento inicial o *'a priori'* , mediante la especificación de una lista de hechos. Estos hechos no se pierden al ejecutar el comando reset. Es decir, cuando se limpia el ambiente de CLIPS, cada hecho especificado dentro de un constructor deffacts se añada en la lista de hechos (*fact-list*).

3.2 Objetos

Un **objeto** en CLIPS es un símbolo, string, número (entero o real), un campo multivaluado, una dirección externa o una instancia de una clase definida por el usuario. Los objetos son instancias de **clases**, y una **clase** es un modelo que recoge las propiedades de objetos. Algunos ejemplos de objetos y sus clases son:

Objeto (Representación Impresa)	Clase
Coche	Símbolo (Symbol)
"Coche"	String
3	Entero (Integer)
3.0	Real (Float)
(Coche 3 [Coche] 3.0)	Multicampo (Multifield)
<Pointer-0023FG2A>	Dirección Externa (External Address)
[Coche]	Coche (Definido por el usuario)

Al igual que los hechos, pueden definirse un conjunto de objetos como conocimiento inicial o *'a priori'* . En este caso se utiliza el constructor **definstances**. Nuevamente, cuando se utiliza el comando reset, cada instancia especificada dentro de un constructor **definstances** es añadido a la lista de instancias (instance-list).

3.3 Variables Globales

CLIPS permite utilizar variables globales mediante el constructor **defglobal**. Coceptualmente, son similares a las variables globales que pueden encontrarse en lenguajes procedurales como C o Pascal. Es decir, son variables cuyo valor es independiente de los constructores y pueden accederse desde cualquier ambiente de CLIPS.

Lección 4

REPRESENTACIÓN DEL CONOCIMIENTO

CLIPS permite definir el concimiento mediante dos paradigmas: heurístico y procedimental. Además permite utilizar la programación orientada a objetos. Veamos cada una de estas representaciones.

4.1 Representación Heurística: Reglas

El modo de representar conocimiento heurístico en CLIPS es mediante reglas. Una regla se compone de antecedente (o parte izquierda de la regla) y consecuente (o parte derecha de la regla).

El antecedente de la regla es el conjunto de condiciones que deben satisfacerse para que la regla se active. La condición de una regla se satisface cuando los hechos o instancias especificadas en el antecedente son ciertas para los hechos conocidos. Un tipo de condición es lo que se conoce como un patrón. Los patrones son un conjunto de restricciones que se utilizan para determinar que hechos u objetos satisfacen la condición especificada en el patrón. El proceso de contrastar los hechos y objetos con los patrones se conocen como reconocimiento de patrones.

El mecanismo por el que una regla se activa, porque sus antecedentes se satisfacen, se conoce como motor de inferencia, que es el encargado de realizar el reconocimiento de patrones y aplicar las reglas. Si hay más de una regla que puede aplicarse, el motor de inferencia utiliza una estrategía de resolución de conflictos que selecciona qué regla deber de activarse primero.

4.2 Representación Procedural

CLIPS permite conocimiento procedural como se hace en lengajes convencionales (p.e. C o Pascal). Las funciones generales y el constructor deffunctions permite al usuario definir componentes ejecutables que o bien desarrollan un conjunto de actividades útilies o retornan un valor.

El constructor *deffunction* permite definir una nueva función en CLIPS. La llamada a una *deffunction* es exactamente igual que a cualquier otra función de CLIPS. El valore que devuelve es el valor de la última expresión evaluada en *deffunction*.

Las funciones genéricas permiten definir nuevo código procedural directamente en CLIPS. Sin embargo, su mayor utilidad se encuentra en sobrecargar operadores convencionales. P.e. el operador + suele utilizarse para sumar números. Dicho operador puede sobrecargarse para que pueda utilizarse para strings.

El conocimiento en CLIPS pude particionarse en módulos. Cada módulo se construye mediante el comando **defmodule**. Cada constructor que se defina pude colocarse en un módulo. El programador puede controlar explicitamente qué constructores en un módulo pueden ser visibles a otros módulos.

Lección 5

EJECUTANDO CLIPS

5.1 Notación

Todos los comandos en CLIPS comienzan y terminan en paréntesis:`(comando)`. Esta sintaxis significa que para poder ejecutar el comando `comando`, deberá de introducirse `(comando)` tal y como se muestra: En primer lugar se introduce el caracter '`(`', seguido de los caracteres '`c`','`o`','`m`','`a`','`n`', '`d`','`o`' y finalmente el carácter '`)`'.

Sin embargo, cada comando presenta una sintaxis propia y una serie de opciones. Para poder utilizar una notación general para todos ellos se utilizará el siguiente convenio.

- Los corchetes, '`[]`' indicarán que lo que se encuentra en su interior es opcional.

  P.e. `(comando [opcion])` indica que `opcion` puede o no especificarse junto con el comando `comando`. Así, se las entradas siguientes son válidas: `(comando)` , `(comando opcion)`.

- Lo que se encuentre entre los símbolos `<` y `>`, indica que debe de sustituirse necesariamente, incluidos los símbolos `<` y `>`, por algún valor del tipo especificado.

  P.e. `<entero>` indica que debe sustituirse por un valor entero. De esta forma, entradas válidas para la sintaxis `(comando <entero>)` serían: `(comando 1)` , `(comando 10)` o `(comando -5)`.

- El símbolo `*` se asocia a un tipo, e indica que la descripción puede reemplazarse por cero o más ocurrencias del tipo especificado. En general se presenta de la forma `<tipo>*`

  P.e. `<entero>*` indica que debe sustituirse por cero o más valores enteros. Entradas válidas serían `0`, `0 1`, `0 1 -1` o simplemente ningún entero.

- El símbolo `+` se asocia a un tipo, e indica que la descripción puede reemplazarse por uno o más ocurrencias del tipo especificado. En general se presenta de la forma `<tipo>+`. Es equivelente a `<tipo> <tipo>*`.

  P.e. `<entero>+` indica que debe sustituirse por uno o más valores enteros. Entradas válidas serían `0`, `0 1`, `0 1 -1`. Necesariamente debe sustituirse por al menos un entero.

- La barra vertical `|` indica que debe hacerse una elección entre los elementos que se encuentran separador por la barra. En general se presenta de la forma `opcion-1 | opcion-2 | ... | opcion-n`.

  P.e. `yo | tu | el` debe reemplazarse por `yo`, o `tu` o `el`.

- El símbolo `::=` se utiliza para definir los términos que aparecen en una expresión. En general presenta la forma `<Termino-a-definir> ::= <Definicion-del-termino>`.

  P.e. Los enteros se definen en CLIPS como sigue:
  `<integer> ::= [+ | -] <digit>+`
  `<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

5.2 Entrando y saliendo de CLIPS

CLIPS puede ejecutarse de modo interactivo o en modo batch. El primero consiste en que tras arrancar CLIPS nos saldrá un cursos parpadeando indicando que el programa espera que le introduzcamos comandos. El segundo modo consiste en que CLIPS ejecutará los comandos que encuentre en un fichero. El modo más intuitivo para distinguir ambos modos de ejecución es verlos desde el punto de vista de la definición de la entrada de datos. Mientra que en modo interactivo la entrada de comandos es la consola (pantalla), y por tanto deben ser introducidos por el usuario, en modo batch la entrada se realiza mediante un fichero.

Un modo intermedio de ejecución consiste en cargar los constructores a partir de un fichero y seguir trabajando en modo interactivo. Esto se hace con el comando `load`.

La sintaxis de la linea de comandos para entrar en CLIPS es:

```
clips <opción>*
<opción> ::= -f <fichero-comandos> | -l <fichero-contructores>
```

- Si no se introduce ninguna opción CLIPS se ejecutará en modo interactivo.
- Para la opción -f, `<fichero-comandos>` es el fichero que contien los comandos CLIPS. Al utilizar esta opción, CLIPS se ejecuta en modo batch.
- Para la opción -l, `<fichero-contructores>` es el fichero que contien los constructores que se utilizarán. Tras cargarlos CLIPS se ejecutará en modo interactivo. Actual como el comando `load`.

Para salir de CLIPS se utiliza el comando `(exit)`.

La sesión 5.1 muestra los comandos a ejecutar para entrar y salir de CLIPS:

```
1: bash%¿ clips ↩
2: CLIPS> (+ 2 3) ↩
   5
3: CLIPS> (exit) ↩
4: bash%¿
```

Sesión 5.1: Primera sesión con CLIPS

La enumeración inicial que aparece indica los pasos de la sesión y no aparecen nunca en una sesión real. Esta enumeración se utilizará para determinar los pasos que deben seguirse en la sesión real.

Esta primera sesión consta de tres pasos. El primer paso, 1:, cosiste en teclear `clips` en la linea de comandos del sistema operativo que se esté utilizando y a continuación pulsar la tecla return del ordenador, indicado por el símbolo ↩. Tras ejecutar el primer paso, saldrá el mensaje `CLIPS>` indicando que hemos entrado en el programa CLIPS. En el segundo paso le pedimos a CLIPS que calcule la suma de los valores enteros 2 y 3, a lo que CLIPS responde con el valor 5. En el tercer paso, 3:, se ejecuta el comando `exit` - recuerde que los comandos se ejecutan siempre entre paréntesis - lo que provoca que salgamos de nuevo al sistema (paso 4:).

Lección 6

HECHOS Y PLANTILLAS

6.1 Hechos

CLIPS distingue entre dos tipos de hechos. Ordenados y no ordenados (ver apartado 3.1). Básicamente, un hecho consta de un **nombre de la relación** (campo simbólico) seguido de cero o más **casillas**[1] (también campos simbólicos). Un ejemplo de hecho es el siguiente:

```
(persona  (nombre "Luis Daniel")
          (apellido "Hernández Molinero")
          (color-ojos marrones)
          (altura 1.90) )
```

Cada hecho, al igual que cada casilla, está delimitado por los paréntesis `( )`. El símbolo `persona` es el nombre de la relación y el hecho consta de cuatro campos: *nombre*, *apellido*, *color-ojos* y *altura*. El valor de la casilla *nombre* es *"Luis Daniel*, el de la casilla *apellido* es *"Hernández Molinero"*, el de la casilla *color-ojos* es *marrones* y el de la casilla *altura* es *1.90*.

El mismo hecho anterior puede escribirse como sigue:

```
(persona  (nombre "Luis Daniel")
          (color-ojos marrones)
          (apellido "Hernández Molinero")
          (altura 1.90) )

(persona  (apellido "Hernández Molinero")
          (nombre "Luis Daniel")
          (color-ojos marrones)
          (altura 1.90) )

(persona  (nombre "Luis Daniel")
          (color-ojos marrones)
          (altura 1.90)
          (apellido "Hernández Molinero") )

(persona  (altura 1.90)
          (nombre "Luis Daniel")
          (color-ojos marrones)
          (apellido "Hernández Molinero") )
```

Es decir, el orden de las casillas es irrelevante.

6.2 Plantillas

Antes de crear los hechos, debe informarse a CLIPS del modelo o plantilla de hechos que se consideran válidos. Es decir, informar del nombre de la relación y de la lista de casillas válidas. Los grupos de hechos que comparten el mismo nombre de la relación se describen mediante el constructor `deftemplate`. El formato general de este constructor es el siguiente:

```
(deftemplate       <nombre-deftemplate> [<comentario>]
                   <definicion-casillas>*)
```

donde

```
<definicion-casillas>   ::=  <def-casilla-simple> | <def-casilla-multiple>
<def-casilla-simple>    ::=  (slot <nombre-casilla> <attributos-plantilla>*)
<def-casilla-multiple>  ::=  (multislot <nombre-casilla> <attributos-plantilla>*)
<atributos-plantilla>   ::=  <atrib-por-defecto> | <restricciones-atributo>
<atrib-por-defecto>     ::=  (default ?DERIVE | ?NONE | <expresion>*)
```

[1] El término casilla será la traducción que se le dará en este tutorial al término inglés **slot**. Este término inglés hace referencia a la casilla, lugar o posición dentro de una tabla

A lo largo de esta sección se irán viendo las distintas opciones que presenta este constructor. Por ahora, baste ver como en la sesión 6.1 se construye la plantilla de hechos `persona` para el ejemplo anterior. Notar que se ha supuesto que ya se ha entrado en el entorno de CLIPS. En lo que sigue, supondremos que siempre nos encontramos dentro de dicho entorno.

```
1: CLIPS> (deftemplate persona "Relacion persona"
                        (slot nombre)
                        (slot apellido)
                        (slot color-ojos)
                        (slot altura) )  ↩
```

Sesión 6.1: Definiendo el hecho `persona` con casillas simples

6.3 Slots simples y múltiples

Las casillas de un hecho, definido por el constructor `deftemplate`, que contienen la palabra clave *slot* contienen sólo un valor. Estas casillas se han hecho referencia con el término `<def-casilla-simple>`.

Sin embargo, a menudo, es deseable poner cero o más valores en una casilla. Para este propósito se utiliza el término `<def-casilla-multiple>` o si se prefiere la palabra clave *multislot*.

```
1: CLIPS> (deftemplate persona "Relacion persona"
                        (multislot nombre)
                        (multislot apellido)
                        (slot color-ojos)
                        (slot altura) )  ↩
2: CLIPS>
```

Sesión 6.2: Definiendo el hecho `persona` con casillas mutiples

En la sesión 6.2 puede ver cómo se define la relación `persona` con casillas múltiples. Lo que permite definir hechos como los siguientes:

```
(persona  (nombre "LuisDaniel")
          (apellido "HernándezMolinero")
          (color-ojos marrones)
          (altura 1.90) )
(persona  (nombre Luis Daniel)
          (apellido Hernández Molinero)
          (color-ojos marrones)
          (altura 1.90) )
```

6.4 Tipos de Hechos

Los hechos que tienen asociado un **deftemplate** se llaman hechos no ordenados. Los que no tiene asociado una plantilla definida se llaman hechos ordenados. La diferencia con los primeros es que las casillas no tienen ningún nombre asociado. Cuando se crea un hecho no ordenado, CLIPS crea automáticamente una plantilla implícita (en oposición a los hechos ordenados en los que hay que definir una plantilla de forma explícita con el constructor `deftemplate`).

Por ejemplo, el hecho `(persona "Luis DanielHernández Molinero"marrones 1.90)` es equivalente al siguiente modelo `(deftemplate persona (multislot valores))` para posteriormente definir el hecho `(persona (valores "Luis DanielHernández Molinero"marrones 1.90))`.

Generalmente, los hechos ordenados NO deben de utilizarse ya que el construir plantillas hace más fácil y entendible el trabajo que se realiza. Sin embargo, hay dos claras excepciones:

1. En la utilización de *flags* (banderas o indicadores). En estos casos el nombre de la relación es útil para indicar que se han realizado un conjunto de órdenes. P.e. `(todo-hecho)` puede utilizarse para indicar que se han procesado todas las órdenes de una regla.

2. Cuando los hechos contienen una sóla casilla y el nombre de la casilla es sinónima del nombre de la relación. P.e. `(temperatura 30.5)` y `(tamaño-coches pequeño mediano grande)` indican `(temperatura (valor 30.5))` y `(tamaño-coches (valor pequeño mediano grande))`.

6.5 Añadiendo, visualizando y borrando hechos

Una vez definido una plantilla de hechos mediante el constructor `deftemplate` se puede proceder a introducir los hechos concretos. Esta operación se realiza con el comando `assert` que tiene la siguiente sintáxis.

```
(assert <facts>+)
```

En la sesión 6.3 puede verse un ejemplo de utilización de este comando. Más concretamenete, en esta sesión, se están realizando los siguientes pasos:

1: Se define la "Relacion persona", donde todas los casillas son simples
2: Se introduce un hecho concreto. Tras pulsar ↩, CLIPS devuelve el mensaje `<Fact-0>`, indicando que al hecho se le asigna el índice `0` en la lista de hechos.
3: Se introduce otro hecho concreto. Tras pulsar ↩, CLIPS devuelve el mensaje `<Fact-1>`, indicando que el hecho introducido se le ha asignado el índice `1` en la lista de hechos.
4: En este paso se introduce el mismo hecho que en el paso anterior, a lo que CLIPS responde con el mensaje `FALSE` indicando que **no** pueden duplicarse hechos.

Para visualizar los hechos que contiene CLIPS en su lista de hechos se utiliza el comando `facts`, que presenta la siguiente sintaxis:

```
(facts [<inicio> [<final> [<máximo>]]])
```

donde `<inicio>`, `<final>` y `<máximo>` son números enteros positivos (y opcionales). Las distintos formas de ejecutar este comando son:

- `(facts)`. No se especifican argumentos. Se mostrarán todos los hechos.
- `(facts inicio)`. Se muestran los hechos con índice mayor o igual a `inicio`.
- `(facts inicio final)`. Se muestran los hechos comprendidos entre lo índices `inicio` y `final` ambos inclusive.
- `(facts inicio final máximo)`. Igual que la opción anterior, pero ahora no se muestran más de `máximo`-hechos.

Observe como funciona el comando `facts` en la sesión 6.4.

Del mismo modo que un hecho puede añadirse a la lista de hechos, un hecho, ya almacenado en la lista de hechos, puede borrarse mediante el siguiente comando:

```
(retract <índice>+)
```

```
1: Repetir la sesión 6.1.
2: CLIPS> (assert ( persona
                        (nombre "Luis Daniel")
                        (apellido "Hernández")
                        (color-ojos marrones)
                        (altura 189) ) ) ↩
   <Fact-0>
3: CLIPS> (assert (persona
                        (nombre "Maria Jesús")
                        (apellido "Rubio")
                        (color-ojos marrones)
                        (altura 165) ) ) ↩
   <Fact-1>
4: CLIPS> (assert (persona
                        (nombre "Maria Jesús")
                        (apellido "Rubio")
                        (color-ojos marrones)
                        (altura 165) ) ) ↩
   FALSE
```

Sesión 6.3: Usando el comando `assert`

```
1: Repetir la sesión 6.3
2: CLIPS> (facts)↩
   f-0    (Persona (nombre "Luis Daniel") (apellido "Hernandez") (color-ojos marrones) (altura
   189))
   f-1    (Persona (nombre "Maria Jesus") (apellido "Rubio") (color-ojos marrones) (altura 165))
   For a total of 2 facts.
3: CLIPS> (facts 0 1)↩
   f-0    (Persona (nombre "Luis Daniel") (apellido "Hernandez") (color-ojos marrones) (altura
   189))
   f-1    (Persona (nombre "Maria Jesus") (apellido "Rubio") (color-ojos marrones) (altura 165))
   For a total of 2 facts.
4: CLIPS> (facts 1)↩
   f-1    (Persona (nombre "Maria Jesus") (apellido "Rubio") (color-ojos marrones) (altura 165))
   For a total of 1 facts.
5: CLIPS> (facts 1 1 0)↩
6: CLIPS> (facts 0 1 0)↩
7: CLIPS> (facts 1 1 1) ↩
   f-1    (Persona (nombre "Maria Jesus") (apellido "Rubio") (color-ojos marrones) (altura 165))
   For a total of 1 fact.
```

Sesión 6.4: Usando el comando `facts`

donde `índice` indica el índice del hecho que se le asoció cuando se introdujo en la lista de hechos. La sesión 6.5 muestra un ejemplo de uso del comando `retract`. Observarse que tanto si se introduce un hecho ya inexistente como la no introducción de índices produce un error en CLIPS.

```
1: Repetir la sesión 6.3
2: CLIPS> (retract 1) ↩
3: CLIPS> (retract 1) ↩
   [PRNTUTIL1] Unable to find fact f-1.
4: CLIPS> (facts) ↩
   f-0    (Persona (nombre "Luis Daniel") (apellido "Hernández") (color-ojos marrones) (altura
   189))
   For a total of 1 fact.
5: CLIPS> (retract) ↩
   [ARGACCES4] Function retract expected at least 1 argument(s)
6: CLIPS> (retract 0) ↩
7: CLIPS> (facts) ↩
8: CLIPS>
```

Sesión 6.5: Usando el comando `retract`

6.6 Modificación y duplicación de hechos

Los valores de un hecho pueden modificarse mediante el comando

```
(modify <índice-hecho> <casilla-a-modificar>+)
```

donde

- `<índice-hecho>` es el índice del hecho a modificar, y
- `<casilla-a-modificar> ::= (<nombre-casilla> <valor-de-la-casilla>)`

  donde `<nombre-casilla>` es el nombre que se le asignó a una casilla al usar el constructor `deftemplate` y `<valor-de-la-casilla>` es el nuevo valor que se le asigna.

Observe el funcionamiento de este comando en la sesión 6.6. Note como el comando `modify` trabaja del siguiente modo: En primer lugar borra el hecho original y posteriormente afirma un nuevo hecho con los nuevos valores de las casillas modificadas.

En ocasiones interesa introducir hechos semajantes cuyos valores se diferencian en pocas casillas. Para no tener que teclear constante la misma información, CLIPS proporciona el comando `duplicate`. Este comando funciona igual que el comando `modify` pero con la diferencia de que no borra el hecho original (sesión 6.7)

✍ **Los comandos `modify` y `duplicate` no pueden utilizarse con hechos ordenados.**

6.7 Restricciones y Valores por Defecto en una casilla

Observe que en la definición de una plantilla de hechos mediante el constructor `deftemplate`, apartado 6.2, se puede introducir varios o ningún atributo por defecto en las casillas (campo `<atributos-plantilla>`). Dichos atributos hace referencia a ciertas restricciones que tendrán las casillas (campo `<restricciones-atributo>`) y el valor por defecto que tomarán las casillas (campo `<atributos-plantilla>`). Conviene que lea el contenido del apartado 7 antes de continuar.

1: Repetir pasos 1: y 2: de la sesión 6.3.
2: `CLIPS> (modify 0 (altura 200))↩`
`<Fact-1>`
3: `CLIPS> (facts)↩`
f-1 (persona (nombre "Luis Daniel") (apellido "Hernandez") (color-ojos marrones) (altura 200))
For a total of 1 fact.
4: `CLIPS> (modify 0 (color-ojos azules))↩`
[PRNTUTIL1] Unable to find fact f-0.
FALSE
5: `CLIPS> (modify 1 (color-ojos azules))↩`
`<Fact-2>`
6: `CLIPS> (facts)↩`
f-2 (persona (nombre "Luis Daniel") (apellido "Hernandez") (color-ojos azules) (altura 200))
For a total of 1 fact.

Sesión 6.6: Usando el comando `modify`

1: Repetir la sesión 6.6.
2: `CLIPS> (duplicate 2 (altura 100))↩`
`<Fact-3>`
3: `CLIPS> (duplicate 2 (nombre "Juan"))↩`
`<Fact-4>`
4: `CLIPS> (facts)↩`
f-2 (persona (nombre "Luis Daniel") (apellido "Hernandez") (color-ojos azules) (altura 200))
f-3 (persona (nombre "Luis Daniel") (apellido "Hernandez") (color-ojos azules) (altura 100))
f-4 (persona (nombre "Juan") (apellido "Hernandez") (color-ojos azules) (altura 200))
For a total of 3 fact.

Sesión 6.7: Usando el comando `duplicate`

La sesión 6.8 muestra un ejemplo de como puede utilizarse la restricción sobre el tipo de datos que puede contener una casilla. Más ejemplos sobre restricciones puede verlos en el apartado 7.

```
1: CLIPS> (deftemplate persona "Relacion persona"
                        (slot nombre (type STRING))
                        (slot apellido (type STRING))
                        (slot color-ojos (type SYMBOL))
                        (slot altura (type FLOAT))
                        (slot edad (type INTEGER)) ) ↩
```

Sesión 6.8: Definiendo restricciones sobre el tipo de datos

Otro aspecto que establece restricciones sobre las casillas son los valores por defecto que se tomarán en cada una de ellas. Cuando esta restricción no se especifica, CLIPS presupone que se ha querido introducir el valor por defecto `(default ?DERIVE)`. Pero veamos que significa esto observando la expresión completa:

```
<atrib-por-defecto> ::= (default ?DERIVE | ?NONE | <expresion>*)
```

El campo `<atrib-por-defecto>` del constructor `deftemplate` especifica cuales son los valores por defecto que se usarán en las casillas (*slots*) de la plantilla cuando se ejecute un comando `assert`. Las posibles opciones son:

- `(default ?DERIVE)`
  En este caso se obtiene un valor por defecto que viene dado por la restricción de la casilla. De nuevo, es conviente que lea el contenido del apartado 7 antes de continuar.

- `(default ?NONE)`
  Establece que siempre es necesario que el usuario establezca un valor para la casilla a la que se le aplica esta restricción. Si no se introducen valores por parte del usuario, en las casillas con esta restricción, CLIPS no aceptará el hecho.

- `(default <expresion>*)`
  Tomará como valor por defecto la propia expresión.

En la sesión 6.9 puede ver el funcionamiento de estas tres opciones. La sesión 6.10 es semejante a la sesión 6.9 pero además se establecen restricciones sobre el tipo. Observe en la sesión 6.11 que es lo que ocurre cuando se establece un valor por defecto que no coincide con el tipo especificado.

6.8 Definición, Visualización y Destrucción de Hechos Iniciales

- Definición de Hechos Iniciales.
  En ocasiones es conveniente afirmar un conjunto de hechos en vez de ir afirmando uno a uno. Si bien el comando `assert` permite afirmar un conjunto de hechos, presenta el siguiente inconveniente. Suponga que usted realiza una docena de afirmaciones antes de que CLIPS busque soluciones a su problema. Imagine que desea comprobar de nuevo que CLIPS *no se ha equivocado*. Entonces, deberá de eliminar todos los nuevos hechos que haya podido derivar CLIPS, introducir de nuevo esa docena de afirmaciones y pedirle de nuevo a CLIPS que busque las soluciones.

```
1: CLIPS> (deftemplate Relacion Ün ejemplo de relacion"
                        (slot casilla-1 (default ?NONE))
                        (slot casilla-2 (default ?DERIVE))
                        (slot casilla-3 (default MiValor)) )  ↩
2: CLIPS> (assert (Relacion)) ↩
   [TMPLTRHS1] Slot casilla-1 requires a value because of its (default ?NONE) attribute.
3: CLIPS> (assert (Relacion (casilla-1 UnValor)))↩
   <Fact-0>
4: CLIPS> (facts)↩
   f-0    (Relacion (casilla-1 UnValor) (casilla-2 nil) (casilla-3 MiValor))
   For a total of 1 fact.
```

Sesión 6.9: Definiendo restricciones sobre valores por defecto

```
1: CLIPS> (deftemplate relacion Ün ejemplo mas "
                        (slot c-1 (type STRING) (default ?NONE))
                        (slot c-2 (type INTEGER) (default 5))
                        (slot c-3 (type FLOAT))
                        (slot c-4 (type SYMBOL) (default UnValor))
                        (slot c-5 (type SYMBOL) )) ↩
2: CLIPS> (assert (relacion (c-1 "hola")))↩
   <Fact-0>
3: CLIPS> (facts)↩
   f-0    (relacion (c-1 "hola") (c-2 5) (c-3 0.0) (c-4 UnValor) (c-5 nil))
   For a total of 1 fact.
```

Sesión 6.10: Definiendo restricciones sobre tipos y valores por defecto

```
1: CLIPS> (deftemplate Relacion "Esto no funciona "
                        (slot c-1 (type STRING) (default MiValor))) ↩
   [CSTRNCHK1] An expression found in the default attribute
   does not match the allowed types for slot c-1.

   ERROR:
   (deftemplate MAIN::Relacion (slot c-1 (type STRING) (default MiValor))
```

Sesión 6.11: Restricciones sobre el tipo y valores por defecto no coinciden

```
1: CLIPS> (deffacts EstadoCoche "El estado de mi coche"
   (coche motor a-punto) (coche bujias limpias) (coche luces funcionan)) ↩
2: CLIPS> (assert (coche intermitente no-funciona))↩
   <Fact-0>
3: CLIPS> (facts)↩
   f-0   (coche intermitente no-funciona)
   For a total of 1 fact.
4: CLIPS> (reset)↩
5: CLIPS> (facts)↩
   f-0   (initial-fact)
   f-1   (coche motor a-punto)
   f-2   (coche bujias limpias)
   f-3   (coche luces funcionan)
   For a total of 4 facts.
6: CLIPS> (deffacts MiCasa "El estado de mi casa"(casa salon grande)) ↩
7: CLIPS> (ppdeffacts)↩
    [ARGACCES4] Function ppdeffacts expected exactly 1 argument(s)
8: CLIPS> (list-deffacts)↩
   initial-fact
   EstadoCoche
   MiCasa
   For a total of 3 deffacts.
9: CLIPS> (ppdeffacts MiCasa)↩
   (deffacts MAIN::MiCasa "El estado de mi casa"
                          (casa salon grande))
10: CLIPS> (undeffacts EstadoCoche)↩
11: CLIPS> (reset)↩
12: CLIPS> (facts)↩
   f-0   (initial-fact)
   f-1   (casa salon grande)
   For a total of 2 facts.
13: CLIPS> (list-deffacts)↩
   initial-fact
   MiCasa
   For a total of 2 deffacts.
14: CLIPS> (list-deffacts *)↩
   MAIN:
      initial-fact
      MiCasa
   For a total of 2 deffacts.
15: CLIPS> (undeffacts)↩
    [ARGACCES4] Function undeffacts expected exactly 1 argument(s)
16: CLIPS> (undeffacts *)↩
17: CLIPS> (reset)↩
18: CLIPS> (facts)↩
19: CLIPS>
```

Sesión 6.12: Usando los comandos de definición de hechos

La eliminación de todos los hechos de la memoria de trabajo, sin borrar las plantillas, para posteriormente afirmar un conjunto de hechos se realiza mediante el comando `reset`.

Para que CLIPS sepa cual es el conjunto de hechos que debe de afirmar automáticamente después de limpiar la memoria de trabajo debe de utilizarse el comando `deffacts` que presenta la sintaxis:

```
(deffacts <nombre-de-la-definicion>  [<comentario>]
                                     <hechos>* )
```

- Visualización de Hechos Iniciales
  Para mostra los hechos definidos con un constructor `deffacts` se utiliza el comando

  ```
  (ppdeffacts [<nombre-de-la-definicion>]
  ```

  Éste muestra exactamente la informacion que se introdujo en el constructor `deffacts` pero del siguiente modo:

  ```
  (deffacts MODULO::<nombre-de-la-definicion>  [<comentario>]
                                               <hechos>* )
  ```

  donde MODULO indica el módulo en el cual el constructor se ha colocado. Los módulos son un macanismo que permite particionar el conocimiento. Por defecto, CLIPS sólo considera un módulo, llamado MAIN, salvo que especifique el usuario la existencia de otros módulos.

  Para mostrar todos los nombres de las listas de hechos almacenados se utiliza el comando:

  ```
  (list-deffacts [<nombre-del-modulo>]
  ```

  Si `<nombre-del-modulo>` no se especifica, se muestran todos los hechos del módulo de conocimiento actual. Si se especifica, se muestran los hechos que se definieron para ese módulo. Si se utilizan el comodín `*`, se mostraran todos los hechos definidos en todos los módulos.

- Borrado de Hechos Iniciales
  Para borrar hechos definidos previamente se utiliza el destructor

  ```
  (undeffacts <nombre-de-la-definicion>
  ```

  Utilizando el destructor no se volverán a afirmar los hechos definidos para `<nombre-de-la-definicion>`. Puede utilizar el comodín `*` para eliminar todas las definiciones existentes.

La sesión 6.12 muestra la utilización del constructor `deffacts`, el destructor `undeffacts` y los camandos de visualización de hechos. Para conocer el significado de `initial-fact` lea la sección 6.11.

6.9 Comandos de Visualización y Destrucción de Plantillas

El constructor de plantillas, `deftemplate`, tiene relacionados otros comandos de visualiazación y destrucción, de forma análoga al constructor `deffacts` de la sección 6.8. Estos comandos son los siguientes:

- Para mostra las plantillas definidas con un constructor `deftemplate` se utiliza el comando

  ```
  (ppdeftemplate [<nombre-de-la-plantilla>]
  ```

- Para mostrar todos los nombres de las plantillas almacenados en los módulos de un programa CLIPS:

  ```
  (list-deftemplates [<nombre-del-modulo>])
  ```

```
1: CLIPS> (deftemplate Persona "Relacion Persona"
         (slot nombre (type SYMBOL)) (slot edad (type INTEGER)) )↩
2: CLIPS> (deftemplate Animal "Relacion Animal"
              (slot patas (allowed-integers 0 1 2 4))
              (slot tipo (allowed-symbols terrestre acuatico volador)))↩
3: CLIPS> (ppdeftemplate)↩
   [ARGACCES4] Function ppdeftemplate expected exactly 1 argument(s)
4: CLIPS> (list-deftemplates)↩
   initial-fact
   Persona
   Animal
   For a total of 3 deftemplates.
5: CLIPS> (list-deftemplates *)↩
   MAIN:
      initial-fact
      Persona
      Animal
   For a total of 3 deftemplates.
6: CLIPS> (undeftemplate Persona)↩
7: CLIPS> (list-deftemplates)↩
   initial-fact
   Animal
   For a total of 2 deftemplates.
8: CLIPS> (assert (Animal))↩
   <Fact-0>
9: CLIPS> (facts)↩
   f-0 (Animal (patas nil) (tipo terrestre))
   For a total of 1 fact.
10: CLIPS> (assert (Animal))↩
   <Fact-0>
11: CLIPS> (facts)↩
   f-0   (Animal (patas nil) (tipo terrestre))
   For a total of 1 fact.
12: CLIPS> (retract 0)↩
13: CLIPS> (facts)↩
14: CLIPS> (undeftemplate Animal)↩
15: CLIPS> (list-deftemplates)↩
   initial-fact
   For a total of 1 deftemplate.
16: CLIPS>
```

Sesión 6.13: Usando los comandos de visualización y destrucción de plantillas

Si `<nombre-del-modulo>` no se especifica, se muestran todas las plantillas del módulo de conocimiento actual. Si se especifica, se muestran las plantillas que se definieron para ese módulo. Si se utilizan el comodín `*`, se mostrarán todas las plantillas definidas en todos los módulos.

- Para borrar plantillas definidas previamente se utiliza el destructor

  `(undeffacts <nombre-de-la-definicion>`

  El destructor no tendrá efecto si existen hechos definidos para la plantilla que se quiera destruir. Puede utilizar el comodín `*` para eliminar todas las plantillas existentes.

La sesión 6.13 muestra la utilización del destructor `undeftemplate` y los camandos de visualización de plantillas. Para conocer el significado de `initial-fact` lea la sección 6.11.

6.10 Comandos de Depuración sobre Hechos

CLIPS también proporciona comandos para depurar los programas. Uno de ellos es el comando `watch facts` que permite la impresión de mensajes que indican las modificaciones que se realizan en la lista de hechos cuando se afirman o retractan hechos. Si se desea desactivas la impresión de mensajes se utiliza el comando `unwatch facts`.

```
1: CLIPS> (deftemplate Animal "Que dice un animal"
                        (slot animal)
                        (slot dice)) ↩
2: CLIPS> (assert (Animal (animal perro) (dice guau))) ↩
   <Fact-0>
3: CLIPS> (watch facts) ↩
4: CLIPS> (assert (Animal (animal gato) (dice miau))) ↩
   ==> f-1 (Animal (animal gato) (dice miau))
   <Fact-1>
5: CLIPS> (modify 1 (animal gallo) (dice kikiriki))) ↩
   <== f-1 (Animal (animal gato) (dice miau))
   ==> f-2 (Animal (animal gallo) (dice kikiriki))
   <Fact-2>
6: CLIPS> (retract 2) ↩
   <== f-2 (Animal (animal gallo) (dice kikiriki))
7: CLIPS> (unwatch facts) ↩
8: CLIPS> (retract 0) ↩
9: CLIPS>
```

Sesión 6.14: Usando los comandos de visualización y destrucción de plantillas

La sesión 6.14 muestra el funcionamiento de estos comandos. Cuando CLIPS muestra la secuencia `<== f-xxxx`, está indicando que el hecho con índice `xxxx` se ha eliminado de la lista. Y si muestra la secuencia `==> f-xxxx`, indica que el hecho con índice `xxxx` se ha añadido a la lista.

6.11 Limpiando la Memoria de Trabajo

Existen dos comandos para limpiar la memoria de trabajo de CLIPS: `reset` y `clear`.

El comando `reset` se utiliza para limpiar la memoria de todos aquellos hechos que han sido introducidos en la memoria. Su ejecución produce que automáticamente se realicen los siguientes comandos:

1. Borra todas las reglas activadas.
2. Borra todos los hechos.
3. `(deftemplate initial-fact)`
4. `(deffacts initial-fact (initial-fact))`
5. Ejecuta todos las definiciones de hechos que hayan podido realizars antes de `reset`. Vea la sección 6.8 para más información.

El ejecución de `clear` produce automáticamente los siguientes comandos:

1. Borra todas las afirmaciones posibles (hechos, reglas, etc)
2. Borra todos los modelos (plantillas, reglas, etc)
3. `(deftemplate initial-fact)`

En términos generales, el comando `reset` es conveniente utilizarlo cuando se desea probar distintas estrategias para distintos hechos manteniendo la misma base de conocimiento (plantillas y reglas). Conviene utilizarlo antes de ejecutar un programa CLIPS si el programa ya ha sido ejecutado anteriormente. El comando `clear` es conveniente utilizarlo cuando se desea cambiar completamente de base de conocimiento.

6.12 Ejercicios Propuestos sobre Hechos

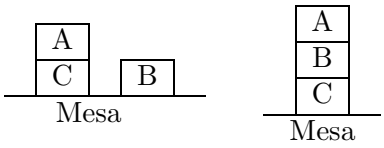
Realice los siguientes ejercicios. Muestre los enunciados de cada uno de ellos en su dirección WEB en una página llamada RCLIPS-1.html. Al final de cada ejercicio ponga dos enlaces. El primero enlazará con la solución del problema. El segundo enlazará con una sesión CLIPS que muestra como se resuelve el problema.

Ejercicio 1 Defina una plantilla que permita definir conjuntos. La plantilla debe incluir información sobre el nombre del conjunto, la lista de elementos que lo componen y quién es subconjunto.

Particularice a los siguientes conjuntos:

- Números={2,4,6,1.0,3.0}
- Enteros={2,4}
- Reales={1.0,3.0}
- Lexemas={"rojo", "amarillo", azul, verde}
- Stings={"rojo"}
- Simbolos={verde}

Ejercicio 2 Defina una plantilla para indicar que un objeto se encuentra encima de otro. Particularice a las siguientes situaciones:



Ejercicio 3 ¿Cómo implementaría un array de objetos usando plantillas?

Ejercicio 4 Defina las plantillas que permitan representar las relaciones de parentesco: padre/madre, tio/a, cuñado/a, primo/a, abuelo/a. Particularice para la siguiente familia.

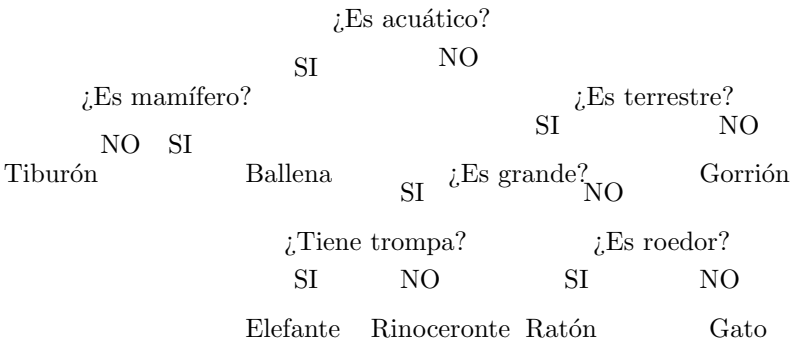
José Felisa Belén David

Pablo Miguel Susana Tomás Ana Juan

Luis Antonio Maria Jesus Angeles

El arco $A \rightarrow B$ indica que A es progenitor de B y un arco del tipo A—-B indica que A y B están casados.

Ejercicio 5 Convierta el siguiente árbol de clasificación en una definición de hechos. Muestre cómo representa los enlaces entre los nodos. ¿Cree que es necesario una representación distinta para los nodos hojas?.



Lección 7

Restricciones de los Atributos

Como ya se ha visto se pueden establecer restricciones en los atributos en la definición de plantillas (`deftemplate`) y clases (`defclasses`). Existen dos tipos de restricciones estáticas y dinámicas:

- Restricciones estáticas. En este caso la posible violación de la restricción se chequea cuando se carga un programa CLIPS.

- Restricciones dinámicas. En este caso la posible violación de la restricción se chequea cuando un programa CLIPS está ejecutandose.

Las restricciones de un atributo se presenta de la siguiente forma:

```
<restricciones-atributo>  ::=  <tipo-atributo> |
                               <atributo-cte-permitido> |
                               <rango-atributo> |
                               <cardinalidad-atributo>
```

7.1 Tipo de Atributo

El tipo de atributo especifica el tipo de valor que puede almacenarse en la casilla que se restringe.

```
 <tipo-atributo>   ::=  (type <tipo>)
     <tipo>        ::=  <tipo-permitido>+ | ?VARIABLE
<tipo-permitido>   ::=  SYMBOL | STRING | LEXEME | INTEGER |
                        FLOAT | NUMBER | INSTANCE-NAME | INSTANCE-ADDRESS |
                        INSTANCE | EXTERNAL-ADDRESS | FACT-ADDRESS
```

Usar NUMBER para un atributo es equivalente a usar tanto INTEGER como FLOAT. Usar LEXEME para un atriuto es equivalente a usar tanto SYMBOL como STRING. Usar INSTANCE para un atriuto es equivalente a usar tanto INSTANCE-NAME como INSTANCE-ADDRESS. Usar ?VARIABLE es decir que se permite cualquier tipo.

7.2 Atributos Constantes Permitidos

Los atributos constantes constantes pormitidos establece los valores constantes de un tipo específico que pueden almacenarse en una casilla restringida.

```
<atributo-cte-permitido>   ::=  (allowed-symbols <lista-simbolos>) |
                                (allowed-strings <lista-cadenas>) |
                                (allowed-lexemes <lista-lexemes>) |
                                (allowed-integers <lista-enteros>) |
                                (allowed-floats <lista-reales>) |
                                (allowed-numbers <lista-numeros>) |
                                (allowed-instance-names <lista-nombre-instancias>) |
                                (allowed-values <lista-valores>)
     <lista-simbolos>      ::=  <simbolo>+ | ?VARIABLE
      <lista-cadenas>      ::=  <cadena>+ | ?VARIABLE
      <lista-lexemes>      ::=  <lexeme>+ | ?VARIABLE
      <lista-enteros>      ::=  <entero>+ | ?VARIABLE
       <lista-reales>      ::=  <real>+ | ?VARIABLE
      <lista-numeros>      ::=  <numero>+ | ?VARIABLE
<lista-nombre-instancias>  ::=  <nombre-instacia>+ | ?VARIABLE
     <lista-valores>       ::=  <constante>+ | ?VARIABLE
```

La lista de valores puede ser o una lista de constantes del tipo especificado o la palabara clave ?VARIABLE (se admite cualquier constante del tipo permitido).

Usar `allowed-lexemes` es equivalente a usar simultaneamente `allowed-symbols` y `allowed-strings`. Usar `allowed-numbers` es equivalente a usar simultaneamente `allowed-integers` y `allowed-floats`.

También conviene hacer la siguiente matización `allowed-values` hace que una casilla se restringa a los valores especificados que pueden ser de todos los tipos. Así, por ejemplo, podría utilizarse esta restricción para permitir sólo valores simbólicos. Entonces, ¿qué diferencia hay entre `allowed-values` y `allowed-symbols`. Si por ejemplo establecemos las restricciones `(allowed-symbols hola adios)` y `(allowed-values hola adios)`, el primero establece que si los valores son de tipo símbolo, entonces su valor debe ser uno de la lista de símboloes. El segundo establece que sólo son `hola` y `adios` esos valores permitidos.

7.3 Rango de los Atributos

Cuando los atributos son numéricos conviene, en ocasiones, establcer sus valores máximo y mínimo, para lo que se utiliza la sintaxis:

```
<rango-atributo>  ::=  (range <limite> <limite>)
        <limite>  ::=  <numero> | ?VARIABLE
```

Si se utiliza la palabra ?VARIABLE como valor mínimo, entonces el límite inferior toma el valor $-\infty$, y si se utiliza como valor máximo, dicho límite toma el valor $+\infty$.

✍ **Si se utiliza la restricción de rango, `range`, no pueden utilizarse las restricciones `allowed-integers`, `allowed-floats`, `allowed-numbers` y `allowed-values`.**

7.4 Cardinalidad de los Atributos

En otras ocasiones puede que no deseemos introducir un número ilimitado de campos en una casilla multicampo.

```
<cardinalidad-atributo>  ::=  (cardinality <limite> <limite>)
              <limite>   ::=  <entero> | ?VARIABLE
```

El primer `<limite>` es el número mínimo de campos que pueden almacenarse en una casilla y el segundo `<limite>` es el número máximo de campos que pueden almacenarse en una casilla. Si se utiliza la palabra clave ?VARIABLE como primer límite, entonces la cardinalidad mínima es cero. Si se utiliza la palabra clave ?VARIABLE como segundo límite, entonces la cardinalidad máxima es $+\infty$. Si no se espcifica la cardinalidad de un atributo se supone que se optado por introducir ?VARIABLE en ambos límites.

✍ **La restricción de cardinalidad no puede utilizarse en casillas formadas por un sólo campo. Sólo se utilizan para casillas multicampo.**

7.5 Valores por Defecto de un Atributo

Las casillas en los constructores `deftemplate` y `instance` pueden tomar valores por defecto si no se especifica ningún valor por defecto. CLIPS utiliza las siguientes reglas, y en el orden que se exponen, para derivar el valor por defecto que no se ha especificado.

1. El tipo por defecto de una casilla se elige a partir de la siguiente lista con el siguiente orden de precedencia: SYMBOL, STRING, INTEGER, FLOAT, INSTANCE-NAME, INSTANCE-ADDRESS, FACT-ADDRESS, EXTERNAL-ADDRESS.

2. Si el tipo por defecto tiene una restricción de constantes permitidas, las del tipo `allowed-xxxx` - donde `xxxx` es el tipo de valores -, entonces el primer valor especificado por esta restricción será el valor por defecto de la casilla.

3. Si los valores por defecto no vienen dados por el paso 2, y el tipo por defecto es INTEGER o FLOAT, y el rango del atributo está espcificado, entonces el valor del limite inferior del rango se tomará como valor por defecto siempre y cuando no se haya especifico el limite inferior con la palabra ?VARIABLE. En otro caso, se tomará como valor por defecto el límite superior del rango siempre y cuando no se haya especifico el limite superior con la palabra ?VARIABLE.

4. Si los valores por defecto no vienen dados por el paso 2 o 3, entonces se utilizan los siguientes valores por defecto:

Tipo de valor	*Valor por defecto*
Symbol	nil
String	""
Integer	0
Float	0.0
Instance-name	nil
Instance-address	Puntero a un instancia vacia
Fact-address	Puntero a un hecho vacio
External-address	NULL

5. Si el valor por defecto debe obtenerse para una casilla con un sólo campo, entonces el valor por defecto se obtiene a partir de los cuatro pasos anteriores. El valor por defecto de para una casilla multicampo es un valor multicampo de loongitud nula. No obstante, si en la casilla multicampo se ha especificado una cardinalidad mínima, se creará un valor multicampo de esa longitud donde cada uno de las campos que lo componene tomarán un valor atendiendo a las normas de las casillas de un sólo campo.

Lección 8

INTRODUCCIÓN AL MANEJO DE REGLAS

8.1 Reglas

Una regla es una expresión del tipo **Si** *antedente,* **entonces** *consecuente.* Su lectura es la siguiente: Si el *antecedente* es cierto para los hechos almacenados en la lista de hechos, entonces pueden realizarse las acciones especificadas en el *consecuente.*

En CLIPS estas sentencias presentan la siguiente sintaxis:

```
(defrule  <nombre-de-la-regla>  [<comentario-string>]
          [<propiedades>]
          <antecedente>*
          =>
          <acciones>* )
```

Por ejemplo, imagine que usted considera la siguiente reglas: **Si** *está lloviendo y no tengo paraguas*, **entonces** *me mojaré.* Si quiere implementarla en CLIPS deberá de escribir:

```
(defrule  LLuvia              "Mi primera regla"
          (EstaLloviendo si)
          (TengoParaguas no)
          =>
          (MeMojaré si))
```

Conviene notar en la sintaxis de construcción de reglas lo siguiente:

- El término **Si**, del si...entonces, se omite.
- El término **entonces**, del si...entonces, se sustituye por la secuencia `=>`.
- Pueden no existir elementos antecedentes o pueden no existir elementos consecuentes (acciones).
- Los elementos que constituyen el antecedente están unidos implicitamente por una conjunción.
- Si no existen elementos antecedentes, entonces el hecho inicial (`initial-fact`) se utilizará automáticamente.
- Si no existen acciones en la regla, en el caso de que una regla se ejecute, no ocurrirá nada.

Al igual que la construcción de plantillas (`deftemplate`) y de hechos (`deffact`) dispone de comandos de visualización y destrucción, el constructor de reglas (`defrule`) también dispone de comandos análogos. Más concretamente:

- La destrucción de reglas se realiza mediante el comando:

  ```
  (undefrule <nombre-de-la-regla>)
  ```

  Si se utiliza el comodín *, entonces todas las reglas definidas por constructores `defrule` será destruidas.

- La visualización de reglas puede hacerse mediante los comandos:

– `(ppdefrule <nombre-de-la-regla>)`
Muestra la definición de una regla indicando el módulo al que pertenece. Por defecto el módulo es MAIN.

– `(list-defrules [<nombre-del-modulo>])`
Si `<nombre-del-modulo>` no se especifica, se mostrarán todas las reglas del módulo actual. Si se especifica `<nombre-del-modulo>`, se mostrarán los nombres de todas las reglas para el módulo especificado. Si se utiliza el comodín `*` se mostrarán los nombres de todas las reglas de todos los módulos.

```
 1: CLIPS> (defrule R1
                   => (assert (hazR2)))↩
 2: CLIPS> (defrule R2
               (hazR2) => (assert (hazR3) (hazR4)))↩
 3: CLIPS> (defrule R3
               (hazR3) => (assert (hazR2)))↩
 4: CLIPS> (defrule R4
               (hazR4) => )↩
 5: CLIPS> (reset)↩
 6: CLIPS> (facts)↩
    f-0   (initial-fact)
    For a total of 1 fact.
 7: CLIPS> (run)↩
 8: CLIPS>(facts)↩
    f-0   (initial-fact)
    f-1   (hazR2)
    f-2   (hazR3)
    f-3   (hazR4)
    For a total of 4 facts.
 9: CLIPS> (reset)↩
10: CLIPS> (run 1)↩
11: CLIPS> (facts)↩
    f-0   (initial-fact)
    f-1   (hazR2)
    For a total of 2 facts.
12: CLIPS> (run 1)↩
13: CLIPS> (facts)↩
    f-0   (initial-fact)
    f-1   (hazR2)
    f-2   (hazR3)
    f-3   (hazR4)
    For a total of 4 facts.
14: CLIPS> (run)↩
```

Sesión 8.1: Ejecución de Reglas

8.2 Ejecución de Reglas

Una vez que se ha especificado el conocimiento de un problema mediante un conjunto de reglas y una lista de hechos, CLIPS está listo para aplicar las reglas. ¿Pero cómo se aplican las reglas en CLIPS?. Para entenderlo, es necesario conocer algunos conceptos previos:

- Se dice que una regla se activa cuando las precondiciones de una reglan se satisfacen. Es decir, casan o encajan con algunos hechos de la memoria de trabajo.

  Notar que una regla puede activarse para distintos conjuntos de hechos (hay distintas razones para que la regla se active). En este caso, cada posible activación de la regla se llama **instancia de una regla**.

- Se dice que una regla está desactivada si no está activada. Es decir, no existe ninguna instancia de una regla.

- Se dice que una regla se dispara si se ha optado por realizar las acciones (consecuentes) de una regla. Notar que es prerrequisito que la regla esté activada.

- Se conoce por **agenda** a la parte del sistema que contiene una lista de las reglas activadas. Notar que una regla aparecerá tantas veces en la agenda como instancias existan de la regla.

- Una **estrategia de resolución de conflictos** es el conjunto de criterios que permiten determinar, en el caso de que haya varias reglas en la agenda (varias reglas activas), qué regla debe dispararse.

- Se llama **prioridad** de una regla a un valor numérico asociado a la regla que indica la importancia de la regla, y por tanto la prioridad con que debe ejecutarse. Cuanto mayor es el valor, mayor es la prioridad. Notar que todas las instancias de una regla tendrán la misma prioridad.

  CLIPS permite establecer valores de prioridad entre los valores -10000 y +10000. Por defecto, todas las reglas tienen el valor de prioridad 0.

Con estos conceptos, ya se está en condiciones de entender el ciclo básico de ejecución de CLIPS.

1. Para ejecutar las reglas en CLIPS se introduce el comando:

   ```
   (run [<máximo>])
   ```

   Si `<máximo>` no se especifica se ejecutan todas las reglas hasta que no haya mas reglas que aplicar. Si se especifica el parámetro, se ejecutarán como máximo tantas reglas como el valor `<máximo>`.

2. Si el límite de ejecución de reglas se ha alcanzado, la ejecución se para.

3. Se actualiza la agenda atendiendo a la lista de hechos de la memoria de trabajo.

   Es decir, a la luz de los hechos iniciales y deducidos en la aplicación de reglas previas, pueden activarse nuevas reglas o reglas que anteriormente estaban activas pueden ahora desactivarse. Las reglas activas nuevas se añadirán a la agenda y aquellas que, estando en la agenda, ahora se desactivan son eliminadas de la agenda.

4. Se selecciona la mejor regla atendiendo a la estrategia de resolución de conflictos y las prioridades de las reglas.

5. La instancia seleccionada de una regla se dispara (se ejecuta), y es eliminada de la agenda.

6. Volver al paso 2.

La sesión 8.1 muestra el uso del comando `run`.

```
1: Hacer pasos 1-4 de la sesión 8.1
2: CLIPS> (agenda)↩
3: CLIPS> (reset)↩
4: CLIPS> (agenda)↩
   0   R1: f-0
   For a total of 1 activation.
5: CLIPS> (run 1)↩
6: CLIPS> (agenda)↩0   R2: f-1
   For a total of 1 activation.
7: CLIPS> (run 1)↩
8: CLIPS> (agenda)↩
   0   R4: f-3
   0   R3: f-2
   For a total of 2 activations.
9: CLIPS> (run 1)↩
10: CLIPS> (agenda)↩0   R3: f-2
   For a total of 1 activation.
11: CLIPS> (run 1)↩
12: CLIPS> (agenda)↩
```

Sesión 8.2: Visualización de Reglas Activas

8.3 Visualización de Reglas Activas y Disparadas

- Reglas Activas. Se visualizan con los comandos

  - `(agenda)`. Muestra las reglas activas en el formato

    `0 <Nombre-de-la-regla>:  f-xxx1, f-xxx2,...,f-xxxn`

    que refleja el hecho de que la regla `<Nombre-de-la-regla>` se activa porque se verifican los hechos `f-xxx1, f-xxx2,...,f-xxxn`.

  - `(watch activations)`. Muestra las reglas activas en el formato

    `==> Activation 0 <Nombre-de-la-regla>:  f-xxx1, f-xxx2,...,f-xxxn`

    para reflejar que la regla `<Nombre-de-la-regla>` se activa porque se verifican los hechos `f-xxx1, f-xxx2,...,f-xxxn`.

    Cuando se desactiva una regla, éstas se muestran en el formato

    `<== Activation 0 <Nombre-de-la-regla>:  f-xxx1, f-xxx2,...,f-xxxn`

    En definitiva, la secuencia `==>` indica que se añade una nueva regla a la agenda, y la secuencia `<==` indica que se suprime una regla de la agenda.

    Puede desactivarse esta última opción introduciendo el comando

    `(unwatch activations)`

```
1: Hacer pasos 1-4 de la sesión 8.1
2: CLIPS> (watch activations)↩
3: CLIPS> (reset)↩
   ==> Activation 0   R1: f-0
4: CLIPS> (run)↩
   ==> Activation 0   R2: f-1
   ==> Activation 0   R3: f-2
   ==> Activation 0   R4: f-3
5: CLIPS> (reset)↩
   ==> Activation 0   R1: f-0
6: CLIPS> (run 2)↩
   ==> Activation 0   R2: f-1
   ==> Activation 0   R3: f-2
   ==> Activation 0   R4: f-3
7: CLIPS> (reset)↩
   <== Activation 0   R3: f-2
   <== Activation 0   R4: f-3
   ==> Activation 0   R1: f-0
```

Sesión 8.3: Visualización Dinámica de Reglas Activas

Las sesiones 8.2 y 8.3 muestran la activación de reglas. Notar que la diferencia básica entre `(watch activations)` y `(agenda)` es que la primera muestra la activación de reglas en tiempo de ejecución mientras que con la segunda opción es necesario obligar al sistema a que *espere* para poder realizar la consulta de la agenda.

- Reglas Disparadas. Se visualizan con el comando

  `(watch rules)`

  Muestra las reglas disparadas en el formato

  `FIRE  XXX <Nombre-de-la-regla>:  f-xxx1,...,f-xxxn`

  Donde:

  - `FIRE`, indica que se ha disparado una regla.
  - `XXX`, indica el número de disparos, incluida esta regla, desde que se realizó el último comando `run`.
  - `<Nombre-de-la-regla>`, es el nombre de la regla que acaba de dispararse.
  - `f-xxx1,...,f-xxxn`, son los hechos que justifican la activación y disparo de la regla.

  Puede desactivarse esta opción introduciendo el comando

  `(unwatch rules)`

  La sesión 8.4 muestra el uso de estos comandos.

8.4 Parando la Ejecución de Reglas

A CLIPS se le puede indicar que pare la ejecución cuando alcance una regla determinada. Los comandos asociados a esta actividad son los siguientes.

```
1: Hacer pasos 1-4 de la sesión 8.1
2: CLIPS> (watch rules)↩
3: CLIPS> (run)↩
   FIRE   1 R1: f-0
   FIRE   2 R2: f-1
   FIRE   3 R4: f-3
   FIRE   4 R3: f-2
4: CLIPS> (watch activations)↩
5: CLIPS> (run)↩
   FIRE   1 R1: f-0
   ==> Activation 0   R2: f-1
   FIRE   2 R2: f-1
   ==> Activation 0   R3: f-2
   ==> Activation 0   R4: f-3
   FIRE   3 R4: f-3
   FIRE   4 R3: f-2
```

Sesión 8.4: Visualización Dinámica de Reglas Disparadas

- Puntos de corte

```
(set-break <nombre-de-la-regla>)
```

- Borrado de los puntos de corte

```
(remove-break [<nombre-de-la-regla>])
```

- Visualización de los puntos de corte

```
(show-break)
```

A estas alturas, el uso de estos comandos resulta trivial. Se deja como actividad, que pruebe el uso de estos comandos estableciendo como puntos de corte las reglas R2 y R4 de la sesión 8.1.

8.5 Patrones: Literales, Comodines y Variables

El antecedente de una regla consta de cero o más elementos condicionales. El elemento condicional más simple es un patrón. Cada patrón consta de una o varias restricciones sobre el valor de algunos atributos (casillas de la plantilla) de un hecho. Las posibles restricciones son:

```
<restriccion>  ::=  <literal> | ?  | $?
                    <variable-de-casilla-simple>
                    <variable-de-casilla-multicampo>
<variable-de-casilla-simple>  ::=  ?<simbolo-de-la-variable>
<variable-de-casilla-multicampo>  ::=  $?<simbolo-de-la-variable>
```

Estudiemoslas más detenidamente a partir de los hechos definidos en la sesión 8.5.

8.5.1 Literales.

La restricción más básica que puede utilizarse en un elemento condicional es aquella en la que se exige un valor exacto para una casilla de un hecho. Estas restricciones reciben el nombre de restricciones literales. La sesión 8.6 muestra el uso de este tipo de restricción.

```
1: CLIPS> (deftemplate Persona
                        (slot nombre)
                        (slot apellido)
                        (slot edad)
                        (multislot amigos))↩
2: CLIPS> (deffacts Amigos
              (Persona (nombre Luis)
                        (apellido Perez)
                        (edad 30)
                        (amigos Jose Manolo Daniel))
              (Persona (nombre Daniel)
                        (apellido Rubio)
                        (edad 25)
                        (amigos Jose Daniel Maria))
              (Persona (nombre Jose)
                        (apellido Perez)
                        (edad 32)
                        (amigos Daniel Luis))
              (Persona (nombre Manolo)
                        (apellido Palma)
                        (edad 30)
                        (amigos Maria Daniel))
              (Persona (nombre Maria)
                        (apellido Rubio)
                        (edad 30)
                        (amigos Daniel Jose Manolo)))↩
3: CLIPS> (watch rules)↩
4: CLIPS> (watch activations)↩
```

Sesión 8.5: Más Hechos

```
1: Realizar la sesión 8.5.
2: CLIPS> (defrule ImprimePerez
                        (Persona (apellido Perez))
                        =>
                        (printout t "Existe al menos un Perez"crlf))↩
3: CLIPS> (reset)↩
   ==> Activation 0   ImprimePerez: f-1
   ==> Activation 0   ImprimePerez: f-3
4: CLIPS> (run)↩
   FIRE   1 ImprimePerez: f-3
   Existe al menos un Perez
   FIRE   2 ImprimePerez: f-1
   Existe al menos un Perez
```

Sesión 8.6: Restricciones Literales

```
1: Realizar la sesión 8.5.
2: CLIPS> (defrule ImprimeNombre
                        (Apellido-a-Buscar ?x)
                        (Persona (nombre ?y) (apellido ?x))
                        =>
                        (printout t ?y - se apellida -- "?x crlf))↩
3: CLIPS> (reset)↩
4: CLIPS> (assert (Apellido-a-Buscar Perez))↩
   ==> Activation 0   ImprimeNombre: f-6,f-1
   ==> Activation 0   ImprimeNombre: f-6,f-3
   <Fact-6>
5: CLIPS> (run)↩
   FIRE   1 ImprimeNombre: f-6,f-3
   Jose – se apellida – Perez
   FIRE   2 ImprimeNombre: f-6,f-1
   Luis – se apellida – Perez
```

Sesión 8.7: Restricciones Varibles Uni-Campo

```
1: Realizar la sesión 8.5.
2: CLIPS> (reset)↩
3: CLIPS> (defrule Saludo
                        (Persona (nombre ?))
                        =>
                        (printout t "hola"crlf))↩
   ==> Activation 0   Saludo: f-1
   ==> Activation 0   Saludo: f-2
   ==> Activation 0   Saludo: f-3
   ==> Activation 0   Saludo: f-4
   ==> Activation 0   Saludo: f-5
```

Sesión 8.8: Restricciones Comodin Uni-Campo en Casillas Simples

```
1: Realizar la sesión 8.5.
2: CLIPS> (defrule ImprimeAmigos-1
                        (Buscar-Amigos-de ?x)
                        (Persona (nombre ?n) (amigos ?  ?  ?x))
                        =>
                        (printout t ?n "es amigo de "?x crlf))↩
3: CLIPS> (defrule ImprimeAmigos-2
                        (Buscar-Amigos-de ?x)
                        (Persona (nombre ?n) (amigos ?  ?x ?))
                        =>
                        (printout t ?n "es amigo de "?x crlf))↩
4: CLIPS> (defrule ImprimeAmigos-3
                        (Buscar-Amigos-de ?x)
                        (Persona (nombre ?n) (amigos ?x ?  ?))
                        =>
                        (printout t ?n "es amigo de "?x crlf))↩
5: CLIPS> (defrule ImprimeAmigos-4
                        (Buscar-Amigos-de ?x)
                        (Persona (nombre ?n) (amigos ?  ?x))
                        =>
                        (printout t ?n "es amigo de "?x crlf))↩
6: CLIPS> (defrule ImprimeAmigos-5
                        (Buscar-Amigos-de ?x)
                        (Persona (nombre ?n) (amigos ?x ?))
                        =>
                        (printout t ?n "es amigo de "?x crlf))↩
7: CLIPS> (reset)↩
8: CLIPS> (assert (Buscar-Amigos-de Daniel))↩
   ==> Activation 0   ImprimeAmigos-5: f-6,f-3
   ==> Activation 0   ImprimeAmigos-4: f-6,f-4
   ==> Activation 0   ImprimeAmigos-3: f-6,f-5
   ==> Activation 0   ImprimeAmigos-2: f-6,f-2
   ==> Activation 0   ImprimeAmigos-1: f-6,f-1
   <Fact-6>
```

Sesión 8.9: Restricciones Comodin Uni-Campo en Casillas Multiples

```
1: Realizar la sesión 8.5.
2: CLIPS> (defrule ImprimeAmigos-Dch
                        (Buscar-Amigos-de ?x)
                        (Persona (nombre ?n) (amigos $?  ?x))
                        =>
                        (printout t ?n "es amigo de "?x crlf))↩
3: CLIPS> (defrule ImprimeAmigos-Izda
                        (Buscar-Amigos-de ?x)
                        (Persona (nombre ?n) (amigos ?x $?))
                        =>
                        (printout t ?n "es amigo de "?x crlf))↩
4: CLIPS> (reset)↩
5: CLIPS> (assert (Buscar-Amigos-de Daniel))↩
   ==> Activation 0   ImprimeAmigos-Izda: f-6,f-3
   ==> Activation 0   ImprimeAmigos-Izda: f-6,f-5
   ==> Activation 0   ImprimeAmigos-Dch: f-6,f-1
   ==> Activation 0   ImprimeAmigos-Dch: f-6,f-4
   <Fact-6>
```

Sesión 8.10: Restricciones Comodin Multi-Campo (en Casillas Multiples)

```
1: Realizar la sesión 8.5.
2: CLIPS> (defrule ImprimeAmigos
          (Buscar-Amigos-de ?x)
          (Persona (nombre ?n) (amigos $?antes ?x $?despues))
          =>
          (printout t ?n "es amigo de #"?antes "# "?x "#"?despues "# "crlf))↩
3: CLIPS> (reset)↩
4: CLIPS> (assert (Buscar-Amigos-de Daniel))↩
   ==> Activation 0   ImprimeAmigos: f-6,f-1
   ==> Activation 0   ImprimeAmigos: f-6,f-2
   ==> Activation 0   ImprimeAmigos: f-6,f-3
   ==> Activation 0   ImprimeAmigos: f-6,f-4
   ==> Activation 0   ImprimeAmigos: f-6,f-5
   <Fact-6>
5: CLIPS> (run)↩
   FIRE   1 ImprimeAmigos: f-6,f-5
   Maria es amigo de #()# Daniel #(Jose Manolo)#
   FIRE   2 ImprimeAmigos: f-6,f-4
   Manolo es amigo de #(Maria)# Daniel #()#
   FIRE   3 ImprimeAmigos: f-6,f-3
   Jose es amigo de #()# Daniel #(Luis)#
   FIRE   4 ImprimeAmigos: f-6,f-2
   Daniel es amigo de #(Jose)# Daniel #(Maria)#
   FIRE   5 ImprimeAmigos: f-6,f-1
   Luis es amigo de #(Jose Manolo)# Daniel #()#
```

Sesión 8.11: Restricciones Comodín Multi-Campo (en Casillas Multiples)

8.5.2 Variables Unicampo.

El valor de un campo puede capturarse mediante una variable. En el caso de que aparezca la misma variable en más de un elemento condicional, la primera ocurrencia de la variable lo que hace es capturar el valor, en las restantes ocurrencias se sustituye el valor capturado por la variable.

Las variables en casillas uni-campo tiene la sintaxis mostrada anteriormente en `<variable-de-casilla-simple>`. La sesión 8.7 muestra el uso de este tipo de restricción.

8.5.3 Comodines.

Un comodín representa cualquier valor que se encuentre almacenado en una casilla. Si distinguen dos tipos de comodines:

- Comodín Unicampo, representado por el símbolo ?, sustituye cualquier valor de una casilla, y exactamente uno.
- Comodín Multicampo, representado por el símbolo $?, sustituye a cero o más valores de una casilla.

La sesión 8.8 muestra un ejemplo muy simple del comodín ?. En esta sesión se activan todas las reglas que contengan un valor en la casilla *nombre* de la plantilla *Persona*. Como se sabe, CLIPS siempre asigna un valor por defecto a las casillas, por lo que la utilización del comodín ? para casillas uni-campo no suele emplearse demasiado.

La sesión 8.9 resulta algo más interesante. En esta sesión se construyen las reglas `ImprimeAmigos-1`, ..., `ImprimeAmigos-5` para determinar las personas `?n` que considera como amigo a `?x`. Notar que las tres primeras permiten detectar las personas que tienen tres amigos y las dos últimas permite detectar las personas que tienen dos amigos. Notar también que sería necesario construir una sexta regla para aquellas personas que sólo tienen un amigo (¡hagase!). Estas reglas tendrían todo el sentido si se interpreta la casilla `amigos` como la secuencia de amigos de `?n` por un orden de importancia. Por ejemplo, la regla `ImprimeAmigos-1` se interpretaría que la persona `?n` considera a `?x` como su tercer mejor amigo. Si ésta fuera la interpretación de las reglas, estaríamos obligados a la construcción de las 6 reglas mencionadas. No obstante, aun suponiendo que estamos interesados en establecer este tipo de reglas, esta sesión es lo suficientemente significativa como para entender las limitaciones del comodín ?: El comodín ? siempre debe reemplazarse por exactamente un valor. Imagine si necesitar trabajar con casillas multicampo de hasta n-valores: *¡debería construir n! reglas!*.

Imagine que ahora las personas que aparecen en la casilla `amigos` son considerados por `?n` con la misma importancia. Sería necesario simplificar el número de reglas. CLIPS proporciona el comodín $? para solventar este problema. La sesión 8.10 es un primer intento de resolver el problema de la simplificación de reglas. En esta sesión, con tan sólo dos reglas se pueden imprimir aquellas personas `?n` que consideran a `?x` como amigo. La regla `ImprimeAmigos-Dch` imprime a las personas `?n` que tienen a `?x` al final de la lista, y la regla `ImprimeAmigos-Izda` imprime a las personas `?n` que tienen a `?x` al principio de la lista. Pero, ¿podría simplificarse aún más el número de reglas para este problema?. La respuesta es afirmativa. ¡Hágase!.

8.5.4 Variables Multicampo.

Como ha podido apreciar la diferencia sustancial entre el comodín `?` y una variable `?x` es que con un comodín se hace la sustitución de un valor y con una variable dicho valor se asocia a la variable.

De forma totalmente análoga se puede hacer con valores multicampo. Es decir, CLIPS permite asociar los valores de un comodín multicampo, `$?`, a una variable, llamada variable multicampo, que presenta una sintaxis de la forma `$?<simbolo-de-la-variable>`.

La sesión 8.11 muestra el uso de variables multicampo. En dicha sesión se realiza la *fusión* de las dos reglas definidas en la sesión 8.10 mediante el uso de variables de este tipo con la ventaja añadida de conocer, mediante las variables multicampo, los valores asciados.

```
 1: CLIPS> (defrule NoPuedoCruzar
           (Semaforo ~verde) => (printout t "No Puedo Cruzar"crlf))↩
 2: CLIPS> (defrule PuedoCruzar
           (Semaforo ~rojo&~amarillo) => (printout t "Puedo Cruzar"crlf))↩
 3: CLIPS> (defrule MeAtrevoACruzar
           (Semaforo amarillo|verde) => (printout t "Me Atrevo a Cruzar"crlf))↩
 4: CLIPS> (assert (Semaforo rojo))↩
    <Fact-0>
 5: CLIPS> (run)↩
    No Puedo Cruzar
 6: CLIPS> (retract 0)↩
 7: CLIPS> (assert (Semaforo amarillo))↩
    <Fact-1>
 8: CLIPS> (run)↩
    No Puedo Cruzar
    Me Atrevo a Cruzar
 9: CLIPS> (retract 1)↩
10: CLIPS> (assert (Semaforo verde))↩
    <Fact-2>
11: CLIPS> (run)↩
    Puedo Cruzar
    Me Atrevo a Cruzar
12: CLIPS> (assert (Semaforo amarillo))↩
    <Fact-3>
13: CLIPS> (run)↩
    No Puedo Cruzar
    Me Atrevo a Cruzar
14: CLIPS> (assert (Semaforo rojo))↩
    <Fact-4>
15: CLIPS> (run)↩
    No Puedo Cruzar
```

Sesión 8.12: Restricciones Conectivas

8.6 Patrones: Conectivas en una Casilla

Hasta ahora se han visto restricciones de una casilla establecidos en términos de literales, comodines y variables. CLIPS permite establecer relaciones entre estos tipos de restricciones mediante **(restricciones) conectivas**. A saber:

- Restricción **no**. Se denota por ~ y se satisface si la restricción que le sigue no se satisface.

```
1: CLIPS> (defrule NoPuedoCruzar
        (Semaforo ?x&~verde) => (printout t "("?x ") No Puedo Cruzar"crlf))↩
2: CLIPS> (defrule PuedoCruzar
        (Semaforo ?x&~rojo&~amarillo) => (printout t "("?x ") Puedo
   Cruzar"crlf))↩
3: CLIPS> (defrule MeAtrevoACruzar
        (Semaforo ?x&amarillo|verde) => (printout t "("?x ") Me Atrevo a
   Cruzar"crlf))↩
4: Realizar los pasos, a partir del cuarto, de la sesión 8.12
```

Sesión 8.13: Restricciones Conectivas con Variables

- Restricción **y**. Se denota por & y se satisface si las restricciones adyacentes se satisfacen.

- Restricción **o**. Se denota por — y se satisface si alguna de las restricciones adyacentes se satisfacen.

En el caso de existir varias conectivas en una casilla, la restricción ~ será la primera en realizarse, seguida de las restricciones &, y finalizando con las restricciones —. En caso de empate, la evaluación de las restricciones se harán en el orden de aparición (de izquierda a derecha). Tan sólo hay una excepción a esta regla: si la primera restricción es una variable sguida de la conectiva &, entonces la primera restricción (la variable) será tratada a parte. Por ejemplo, la restricción `?x&a|b` no se aplica según la regla general, que la trataría como `(?x&a)|b`, sino que su tratamiento es `?x&(a|b)`. De forma totalmente análoga la restricción `?x&~a|b` no se trata como `(?x&(~a))|b` sino como `?x&((~a)|b)`.

La sesión 8.12 aclara estos conceptos. La sesión 8.13 es completamente análoga a la sesión 8.12 pero ahora las restricciones conectivas contienen una variable. Notar como ahora a la variable `?x` se le asocia el resultado de la restricción conectiva.

8.7 Conectivas entre Elementos Condicionales

Las restricciones conectivas estrablecen restricciones del tipo **no**, **y** y **o** en los valores de una casilla. Esta idea puede extenderse dando lugar a la conectividad entre los distintos elementos condicionales.

Recuerde que por defecto, todos los elementos condicionales de una regla se encuentran enlazados por la conectiva **y**. Por ejemplo, la regla

```
(defrule  LLuvia              "Mi primera regla"
          (EstaLloviendo si)
          (TengoParaguas no)
          =>
          (MeMojaré si))
```

establece implicitamente la conectiva **y** entre los elementos condicionales `(EstaLloviendo si)` y `(TengoParaguas no)`.

Las conectivas entre elementos condicionales también reciben el nombre de elementos condicionales. Los más relevantes son:

- Elemento condicional **o**. Presenta la sintaxis:

```
(or <elementos-condicionales>+)
```

Ejemplo:

```
(defrule  Excursion
          (or (tiempo bueno) (tiempo malo))
          =>
          (printout t "Me voy de excursión"crlf)
```

- Elemento condicional **y**. Presenta la sintaxis:

```
(and <elementos-condicionales>+)
```

Ejemplo:

```
(defrule  Conducir
          (and (tengo coche) (tengo llaves))
          =>
          (printout t "Puedo conducir"crlf)
```

- Elemento condicional **no**. Presenta la sintaxis:

```
(not <elemento-condicional>)
```

Ejemplo:

```
(defrule  VoyAClase
          (not (centro cerrado))
          =>
          (printout t "Tengo clase"crlf)
```

8.8 Restricciones de Valores de Retorno

Un modo de restringir el valor de un campo es mediante un literal. P.e. `(defrule Regla (Persona (edad 30)) =>)`. CLIPS permite sustituir dicho literal por el valor retornado en la llamada a una función. Esta sustitución se realiza con la restricción de valores de retorno (en ingles, *return value constraint*), denotado por `=`. El valor devuelto por la función debe ser uno de los tipos de datos primitivos. La sintaxis de esta restricción es:

```
= <llamada-a-la-función>
```

Dos ejemplos sencillos de su uso son los siguientes:

```
(defrule  Regla1
          (dato ?x)
          (Persona (edad =(* 2 ?x) ) )
          =>)
(defrule  Regla2
          (dato ?x)
          (Persona (edad ?y&=(* 2 ?x)|=(+ 2 ?x) ) )
          =>)
```

8.9 Restricción Predicado

En ocasiones es necesario restringir un campo basado en la veracidad de una expresión booleana. Esta restricción se realiza mediante la restricción predicado que se denota por `:`. El modo de usar esta restricción es como sigue:

```
?x&:<restricción-con-?x>
```

y debe leerse del siguiente modo:

determina el valor ?x tal que ?x verifica la restricción `<restricción-con-?x>`

Imagine que desea establecer en el entecdente de una regla que el hecho `valor` tiene una cantidad positiva. Un modo de definirlo es el siguiente (ver apartado 8.11 para el significado de `test`):

```
(defrule  Regla
          (valor ?x)
          (test (> ?x 0))
          => )
```

El antecedente de la regla puede sustituirse por el siguiente: *determinar el valor ?x tal que ?x verifica la restricción ser mayor que 0.* Utilizando la restricción predicado, la regla anterior quedaría como sigue:

```
(defrule  Regla1
          (valor ?x&:(> ?x 0))
          => )
```

Otro ejemplo es el siguiente. Para determinar si el valor del campo de un hecho es un número pude usarse la regla (ver apartado 10.3.2):

```
(defrule  Regla2
          (Hecho (valor ?x&:(integerp ?x)|:(floatp ?x) ) )
          => )
```

8.10 Captura de Direcciones de Hechos

En muchas ocasiones resulta útil poder realizar modificaciones, duplicaciones o eliminaciones de hechos (e instancias) en el consecuente de una regla. Para ello resulta imprescindible poder detectar el índice (dirección) que CLIPS le asoció al hecho cuando éste se afirmó. La captura de direcciones se realiza en el antecedente de la regla mediante la sintaxis.

```
?<simbolo-variable> <- <Patrón-Elemento-Condicional>
```

La sesión 8.14 muestra un ejemplo de cómo puede utilizarse la captura de direcciones para modificar y borrar hechos.

8.11 Elemento Condicional `Test`

El elemento condicional `test` permite evaluar expresiones en el antecedente de la regla. Los elementos condicionales anteriores se basaban principalmente en determinar hechos que casaran con un cierto patrón. El elemento condicional `test` permite evaluar una expresión. La evaluación puede ser no-FALSE o FALSE. Si es no-FALSE la regla puede ser activada si también se satisfacen el resto de los elementos condicionales de la regla. Si es FALSE la regla nunca se activaría salvo existan elementos condicionales conectivos que hagan que el antecedente se satisfaga. Su sintaxis es:

```
(test <expresión>)
```

La sesión 8.15 muestra como puede utilizarse.

Este elemento condicional también puede utilizarse con elementos condicionales conectivos. Por ejemplo, los siguientes elementos condicionales sirven para determinar si el valor de una variable es entera y se encuentra en el rango 1-10.

```
(test  (and  (integerp ?valor)
             (>= ?valor 1)
             (<= valor 10)))
```

```
(test  (or  (not (integerp ?valor)
            (< ?valor 1)
            (> valor 10)))
```

```
1: Realizar la sesión 8.5
2: CLIPS> (deftemplate CumpleAgnos (slot nombre) (slot edad))↩
3: CLIPS> (defrule UnAgnoMas
              ?Hecho1 <- (CumpleAgnos (nombre ?x) (edad ?y))
              ?Hecho2 <- (Persona (nombre ?x))
              =>
              (modify ?Hecho2 (edad ?y))
              (retract ?Hecho1))↩
4: CLIPS> (reset)↩
5: CLIPS> (facts)↩
   f-0   (initial-fact)
   f-1   (Persona (nombre Luis) (apellido Perez) (edad 30) (amigos Jose Manolo Daniel))
   f-2   (Persona (nombre Daniel) (apellido Rubio) (edad 25) (amigos Jose Daniel Maria))
   f-3   (Persona (nombre Jose) (apellido Perez) (edad 32) (amigos Daniel Luis))
   f-4   (Persona (nombre Manolo) (apellido Palma) (edad 30) (amigos Maria Daniel))
   f-5   (Persona (nombre Maria) (apellido Rubio) (edad 30) (amigos Daniel Jose Manolo))
   For a total of 6 facts.
6: CLIPS> (assert (CumpleAgnos (nombre Daniel) (edad 27)))↩
   ==> Activation 0   UnAgnoMas: f-6,f-2
   <Fact-6>
7: CLIPS> (run)↩
   FIRE   1 UnAgnoMas: f-6,f-2
   ==> Activation 0   UnAgnoMas: f-6,f-7
   <== Activation 0   UnAgnoMas: f-6,f-7
8: CLIPS> (facts)↩
   f-0   (initial-fact)
   f-1   (Persona (nombre Luis) (apellido Perez) (edad 30) (amigos Jose Manolo Daniel))
   f-3   (Persona (nombre Jose) (apellido Perez) (edad 32) (amigos Daniel Luis))
   f-4   (Persona (nombre Manolo) (apellido Palma) (edad 30) (amigos Maria Daniel))
   f-5   (Persona (nombre Maria) (apellido Rubio) (edad 30) (amigos Daniel Jose Manolo))
   f-7   (Persona (nombre Daniel) (apellido Rubio) (edad 27) (amigos Jose Daniel Maria))
   For a total of 6 facts.
```

Sesión 8.14: Capturando Direcciones de Hechos

```
1: Realizar la sesión 8.5
2: CLIPS> (reset)↩
3: CLIPS> (defrule NoTanJoven
                       (Persona (nombre ?nombre) (edad ?edad))
                       (test (> ?edad 25))
                       =>
                       (printout t ?nombre "ya no es tan joven.  ¡Tiene "?edad
   "años"crlf)))↩
   ==> Activation 0   NoTanJoven: f-1
   ==> Activation 0   NoTanJoven: f-3
   ==> Activation 0   NoTanJoven: f-4
   ==> Activation 0   NoTanJoven: f-5
```

Sesión 8.15: Elemento Condicional Test

Para conocer el significado de las expresiones de comparación que se utilizan en este apartado lea la sección 10

✍ **Se aconseja no usar `test` como el primer elemento condicional de la regla (para más información lea el manual de CLIPS)**.

```
1: CLIPS> (deffacts ValoresIniciales
              (valor 1) (valor 2) (valor 3) (valor 4) (suma 0))↩
2: CLIPS> (watch activations)↩
3: CLIPS> (watch rules)↩
4: CLIPS> (reset)
5: CLIPS> (defrule Sumatoria
                        (valor ?x)
                        ?suma <- (suma ?total)
                        =>
                        (retract ?suma)
                        (assert (suma (+ ?total ?x))))↩
   ==> Activation 0   Sumatoria: f-4,f-5
   ==> Activation 0   Sumatoria: f-3,f-5
   ==> Activation 0   Sumatoria: f-2,f-5
   ==> Activation 0   Sumatoria: f-1,f-5
6: CLIPS> (run 2)↩
   FIRE 1 Sumatoria: f-1,f-5
   <== Activation 0   Sumatoria: f-2,f-5
   <== Activation 0   Sumatoria: f-3,f-5
   <== Activation 0   Sumatoria: f-4,f-5
   ==> Activation 0   Sumatoria: f-4,f-6
   ==> Activation 0   Sumatoria: f-3,f-6
   ==> Activation 0   Sumatoria: f-2,f-6
   ==> Activation 0   Sumatoria: f-1,f-6
   FIRE 2 Sumatoria: f-1,f-6
   <== Activation 0   Sumatoria: f-2,f-6
   <== Activation 0   Sumatoria: f-3,f-6
   <== Activation 0   Sumatoria: f-4,f-6
   ==> Activation 0   Sumatoria: f-4,f-7
   ==> Activation 0   Sumatoria: f-3,f-7
   ==> Activation 0   Sumatoria: f-2,f-7
   ==> Activation 0   Sumatoria: f-1,f-7
```

Sesión 8.16: Primera Sumatoria

8.12 Función Bind

Hasta ahora se han almacenado valores de campos en variables. A menudo es útil asociar un valor a una variable para utilizarlo posteriormente y/o evitar repetir ciertos cálculos. La función `bind` se utiliza para este propósito:

```
(bind <variable> <valor-o-expresión>)
```

Como ejemplo, nos planteamos como resolver el problema de realizar una sumatoria usando un sistema de reglas. Una primera alternativa es la sesión 8.16. Es fácil comprobar que esta primera

```
1: CLIPS> (deffacts ValoresIniciales
              (valor 1) (valor 2) (valor 3) (valor 4) (suma 0))↩
2: CLIPS> (watch rules)↩
3: CLIPS> (reset)
4: CLIPS> (defrule Sumatoria
                        ?valor <- (valor ?x)
                        ?suma <- (suma ?total)
                        =>
                        (retract ?suma ?valor)
                        (assert (suma (+ ?total ?x)))
                        (printout t "Suma="(+ ?total ?x) crlf))↩
5: CLIPS> (run)↩
   FIRE   1 Sumatoria: f-1,f-5
   Suma=1
   FIRE   1 Sumatoria: f-2,f-6
   Suma=3
   FIRE   2 Sumatoria: f-3,f-7
   Suma=6
   FIRE   3 Sumatoria: f-4,f-8
   Suma=10
```

Sesión 8.17: Segunda Sumatoria

```
1: CLIPS> (deffacts ValoresIniciales
              (valor 1) (valor 2) (valor 3) (valor 4) (suma 0))↩
2: CLIPS> (watch rules)↩
3: CLIPS> (reset)
4: CLIPS> (defrule Sumatoria
                        ?valor <- (valor ?x)
                        ?suma <- (suma ?total)
                        =>
                        (retract ?suma ?valor)
                        (bind ?sub-total (+ ?x ?valor))
                        (assert (suma ?sub-total))
                        (printout t "Suma Parcial="?sub-total crlf))↩
5: CLIPS> (run)↩
   FIRE   1 Sumatoria: f-1,f-5
   Suma Parcial = 1
   FIRE   2 Sumatoria: f-2,f-6
   Suma Parcial = 3
   FIRE   3 Sumatoria: f-3,f-7
   Suma Parcial = 6
   FIRE   4 Sumatoria: f-4,f-8
   Suma Parcial = 10
```

Sesión 8.18: Tercera Sumatoria

alternativa no permite solucionar el problema. Así, se opta por diseñar una regla más depurada, la de la sesión 8.17. Aunque esta regla sirve para nuestros presenta como incoveniente que el cálculo `(+ ?total ?x)` ha tenido que hacerse varias veces. Una alternativa más es la de la sesión 8.18 que utiliza la función `bind`.

```
1: CLIPS> (deffacts ValoresIniciales
              (valor 1) (valor 2) (valor 3) (valor 4))↩
2: CLIPS> (reset)↩
3: CLIPS> (defrule ImprimeValor
                       (valor ?x)
                       =>
                       (printout t "Valor= "?x crlf))↩
4: CLIPS> (watch rule)↩
5: CLIPS> (agenda)↩
   0   ImprimeValor: f-4
   0   ImprimeValor: f-3
   0   ImprimeValor: f-2
   0   ImprimeValor: f-1
   For a total of 4 activations.
6: CLIPS> (run 2)↩
   FIRE   1 ImprimeValor: f-4
   Valor= 4
   FIRE   2 ImprimeValor: f-3
   Valor= 3
7: CLIPS> (agenda)↩
   0   ImprimeValor: f-2
   0   ImprimeValor: f-1
   For a total of 2 activations.
8: CLIPS> (refresh ImprimeValor)↩
9: CLIPS> (agenda)↩
   0   ImprimeValor: f-3
   0   ImprimeValor: f-4
   0   ImprimeValor: f-2
   0   ImprimeValor: f-1
   For a total of 4 activations.
```

Sesión 8.19: Usando una regla más de una vez

8.13 Usando la Instancia de Una Regla más de una vez

Todas las instancias de una regla se almacenan en la agenda. Cuando una instancia de una regla es disparada, la regla se suprime de la agenda y no vuelve a activarse. Si se desea que las reglas disparadas se vuelvan a almacenar en la agenda, siempre que se siga satisfaciendo el antecedente para los hechos nuevos que se han derivado en la aplicación previa de reglas, se puede usar el comando:

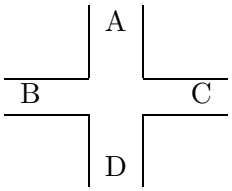
```
(refresh <nombre-de-la-regla>)
```

La sesión 8.19 muestra cómo afecta el comando `refresh` a la agenda.

8.14 Ejercicios Propuestos sobre Reglas

Realice los siguientes ejercicios. Muestre los enunciados de cada uno de ellos en su dirección WEB en una página llamada RCLIPS-2.html. Al final de cada ejercicio ponga dos enlaces. El primero enlazará con la solución del problema. El segundo enlazará con una sesión CLIPS que muestra como se resuelve el problema.

Ejercicio 1 Suponga que encuentra un cruce de calles como el de la figura:



Se pide:

- Defina el número de semáforos necesarios.
- Defina las reglas y hechos que considere adecuados para controlar los semáforos que regulan el tráfico en las siguientes situaciones:
  - Los coches de A (en lo que sigue simplemente A) pueden pasar a la calle D (en lo que sigue simplemente D) y los de D pueden pasar a A.
  - A puede pasar a B y a D; y, simultaneamente, los de D pueden pasar a A y a C.
  - Si los de A están pasando a B, a C y a D, entonces los de B, C y D no pueden circular.

Ejercicio 2 Implemente las reglas que permitan a partir de la relación *x es hijo de y*, determinar el parentesco con otros miembros familiares. Es decir, determinar las relaciones esposo, padre, hermano, tio, abuelo, bisabuelo, cuñado, etc. Suponga, por simplificar el problema, que no hay distinción de sexo y que dos personas con hijo común están casadas.

Particularice a la base de conocimiento del ejercicio 4 de la sección 6.12. Y determine los parientes (e.d. padres, abuelos, etc) de Tomás y María.

Ejercicio 3 Implemente las reglas y hechos que considere adecuados para que el ordenador determine en qué animal está pensando usted a partir del conocimiento del ejercicio 5 de la sección 6.12.

El apartado 9.4.3 puede serle útil.

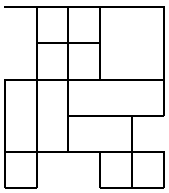
Ejercicio 4 Implemente en CLIPS el tipo de dato abstracto pila con los operadores:

- **push (A)**. Introduce un elemento **A** en la pila.
- **pop**. Elimina el último elemento introducido en la pila.

Después de aplicar cada operador, su programa debe mostrar el estado de la pila. Particularice a la siguiente situación: [1] push(A), [2] push(C), [3] pop, [4] push(B), [5] push(C).

Ejercicio 5 Diseñe un programa en CLIPS que permita al aventurero perdido, que se encuentra en el punto de salida, poder alcanzar el punto de llegada.

Llegada



Salida

Especifique claramente cómo representa el laberinto.

Lección 9

FICHEROS

Como toda aplicación que se precie, es necesario poder disponer de funciones que permitan el manejo de ficheros. CLIPS proporciona las acciones típicas de entrada y salida de datos y algunas específicas pensadas para el tipo de datos que es capaz de representar. En este tutorial clasificaremos las funciones según el tipo de acción que realizan.

9.1 Cargar y Salvar Hechos

CLIPS proporciona dos comandos para manejar ficheros de hechos:

- `(load-facts <nombre-del-fichero>)`.

  Permite hacer una afirmación de los hechos que se encuentran almacenados en un fichero.

- `(save-facts <nombre-del-fichero>)`

  Permite salvar los hechos que se encuentran en la memoria de trabajo en un fichero.

9.2 Cargar y Salvar Constructores

Hasta ahora ha visto la necesidad de teclear todos los constructores (deftemplate, deffacts o defrule) cada vez que inicia una sesión. Puede ahorrarse trabajo y sobre todo tiempo usando los comandos load y save.

Puede, en vez de teclear en todas las sesiones, crear un fichero de texto que contenga los constructores que necesita para, posteriormente, cargarlos en el entorno de CLIPS con el comando

```
(load <nombre-fichero>)
```

donde `<nombre-fichero>` será un string que contendrá el trayecto donde se encuentra ubicado el fichero de constructores. Si no ocurre ningún error, el comando retornará el valor TRUE y en otro caso el valor FALSE.

Conforme se van cargando los constructores, CLIPS mostrará en pantalla mensajes con el siguiente formato:

```
Defining defxxx :  <nombre-defxxxx>
```

donde `defxxx` indica el constructor que se está cargando y `<nombre-defxxx>` es el nombre que le dió al constructor `defxxx` en el fichero.

Si no desea ver los mensajes, puede utilizar el comando

```
(unwatch compilations)
```

en cuyo caso mostrará para cada tipo de constructor un símbolo. Por ejemplo, se mostrará el símbolo * para las reglas, las plantillas con %, los hechos con $, etc.

Por supuesto, puede volver a visualizar todos los mensajes usando el comando

```
(watch compilations)
```

Si no desea visualizar ningún tipo de información sobre el progreso de la carga de constructores puede utilizar el comando

```
(load* <nombre-fichero>)
```

Por otro lado, puede almacenar todos los constructores que haya definido en una sesión interactiva con CLIPS en un fichero con el comando

```
(save <nombre-fichero>)
```

Los constructores se almacenarán en el formato *pretty-printing* y no hay posibilidad de salvar constructores concretos en un fichero.

9.3 Ejecución de Comandos Desde un Fichero

En el apartado 2.5.2 se vio que CLIPS puede trabajar en modo *batch*, interactivo o los dos simultanemente. Un proceso batch no es mas que una secuencia de constructores o comandos que reemplaza la entrada estandar (teclado) por el contenido de un fichero. En este sentido el comando `load` realiza un proceso batch limitado a una secuencia de constructores. CLIPS también permite realizar un proceso *batch completo* desde el propio entorno con el comando

```
(batch <nombre-fichero>)
```

Es decir, `<nombre-fichero>` es una secuencia de constructores y/o comandos. En definitiva, una sesión completa de CLIPS.

Si no existe ningún error en el procesamiento del fichero, se retornará el valor TRUE y FALSE en otro caso.

Importante.**Si se realiza un comando batch en el consecuente de una regla, los comandos del fichero no serán procesados hasta qeu se devuelva al control al mensaje (i.e. prompt) de más alto nivel.**

9.4 Abriendo y Cerrando Ficheros Generales

Imagine ahora que desea volcar en un fichero la información que genera CLIPS en pantalla, o que desea cargar en CLIPS la información generada por otro programa. Para resolver este tipo de problemas, CLIPS dispone las típicas funciones de entrada y salida, mas comunmente conocidas como funciones E/S (en ingles functions I/O).

En general, el proceso usual de E/S es el siguiente:

1. Asignar la localización física del fichero a un fichero lógico (e.d. a una variable del tipo fichero).
2. Abrir el fichero lógico. La apertura se hará según el tipo de operaciones que se quieran realizar: lectura, escritura o ambas.
3. Leer o escribir en el fichero.
4. Cerrar el fichero lógico.

Veamos cuales son los comandos que pueden utilizarse en CLISP para el preceso anterior.

9.4.1 Abriendo Fichero Lógicos

La asignación de la localización física del fichero a un fichero lógico y la apertura del fichero lógico se realiza en CLIPS con un único comando:

```
(open <nombre-fichero> <nombre-logico> [<modo>])
```

Donde los modos de apertura son los siguientes:

- ”r”, se accederá al fichero para leer sólo los datos que contiene.
- ”w”, se accederá al fichero para escribir sólo datos.

- "r+", se accederá al fichero para leer y escribir.
- "a", se accederá al fichero para añadir sólo datos (a partir del último dato del fichero).
- "wb", se escribirá en el fichero en formato binario.

Ejemplo:
`CLIPS> (open "datos.dat"datos "w")`
Permite abrir un fichero para escritura.

9.4.2 Escritura sobre Ficheros Lógicos

La función

`(printout <nombre-logico> <expresion>*)`

muestra la `<expresion>` en el dispositivo que se asoció al `<nombre-lógico>`. El dispositivo debe de haberse abierto previamente con el comando `open` salvo que deseemos volcar la información sobre la salida estandar (pantalla).

La salida estandar usa una **t** para su nombre lógico.

La `<expresion>` puede contener secuencias de caracteres *especiales* cuya salida en el dispositivo se traduce en códigos de control. Se utilizan principalmente para ayudar a leer más facilmente la información volcada sobre el dispositivo. Estas secuencias son:

- crlf, provoca un retorno de carro / nueva linea.
- tab, provoca una tabulación horizontal.
- vtab, provoca una tabulación vertical.
- ff, form feed.

La apariencia de los caracteres de control dependerá del sistema operativo que utilice.

El formato de salida de la función `printout` está limitado a los caracteres de control indicados. Si deseáramos un formato de salida algo más complejo, es conveniente utilizar la función `format` en lugar de la función `printout`.

`(format <nombre-logico> <expresión-string> <expresion>*)`

`<expresión-string>` es un string usual pero que contiene secuencias de control. Esta expresión expecifica la salida exacta que tendrá el texto que vaya a escribirse. Una secuencia de control es una expresión de la forma `%-M.Nz`. La primera secuencia de control será sustituida por la primera expresión del parámetro `<expresion>*`, la segunda secuencia de control será sustituida por la segunda expresión del parámetro `<expresion>*`, etc.

El significado de la expresión `%-M.Nz` es la siguiente:

- El símbolo `-` es opcional y se usa para justificar el texto a la izquierda.
- El símbolo `M` es un número indicando que el parámetro se mostrará con al menos tantos caracteres como el el valor de `M`. Este símbolo es opcional.
- El símbolo `N` es un número que indica el número de dígitos decimales que se mostrará en su lugar. Este símbolo es opcional.
- El valor de `z` en la secuencia de control es alguno de estos caracteres:

- c, muestra el parámetro como un sólo caracter.
- d, muestra el parámetro como un número entero. `N` es ignorado.
- f, muestra el parámetro como un número real.
- e, muestra el parámetro como un número real en formato exponencial.
- g, muestra el parámetro en el formato más general (el más corto).
- o, muestra el parámetro como un número octal sin signo. `N` es ignorado.
- x, muestra el parámetro como un número hexadecimal.
- s, muestra el parámetro como un string. `N` indica el número máximo de caracteres que se imprimirán.
- n, imprime una nueva linea.
- r, imprime un retorno de carro.
- %, imprime el carácter %.

EJEMPLO 1:

```
CLIPS> (open "datos.dat"datos "w")
CLIPS> (printout datos "hola"crlf)
CLIPS> (printout datos 10.5 crlf)
```

Escribe los valores "hola"y 10.5 en el dispositivo datos.

EJEMPLO 2:

```
CLIPS> (format nil "Nombre:  %-15s Edad:  %3dJose Antonio"25)
```

Imprime "Jose Antonio"en un espacio reservado de 15 caracteres y justificado a la izquierda, y el valor 25 en un espacio reservado de 3 caracteres.

9.4.3 Lectura desde Ficheros Lógicos

La función

```
(read [<nombre-logico>])
```

permite leer información desde el disponsitivo indicado en `<nombre-logico>`.

Si no se especifica el dispositivo o se utiliza **t**, la información se leerá desde el dispositivo de entrada estandar (teclado).

La información leida es al equivalente a una palabra. Es decir, lee una secuencia de caracteres hasta que encuentra un delimitador (espacio, retorno de carro, o tabulador).

El final de fichero se identifica con la *palabra* EOF. Cualquier lectura que se haga posterior a la lectura de la palabra EOF producirá un error.

Si en vez de leer una palabra, desea leer *conjuntos de palabras* puede sustituir la función `read` por la función:

```
(readline [<nombre-logico>])
```

En este caso se leerá una secuencia de caracteres hasta encontrar un retorno de carro, una coma o la palabra EOF. En este caso ni los espacios ni las tabulaciones paran la lectura. La función retorna un string.

De nuevo, si no se especifica el dispositivo o se utiliza **t**, la información se leerá del el dispositivo de entrada estandar (teclado).

EJEMPLO 1: Introduciendo un n´umero
(defrule lectura-numero

=¿
(printout t "¿cantidad? ")
(bind ?numero (read))
(assert (valor ?numero)))

EJEMPLO 2: Introduciendo el nombre de una calle
(defrule lectura-calle
=¿
(printout t "¿nombre de la calle? ")
(bind ?nombre (readline))
(assert (nombrecalle ?nombre)))

9.4.4 Cerrando Ficheros Lógicos

Los fichero lógicos abiertos con un comando `open` deben cerrarse antes de salir de CLIPS para garantizar que los datos no se perderán. Para ello se usa el comando:

```
(close [<nombre-logico>])
```

Si no se utilizan argumentos, CLIPS cerrará todos los ficheros que estuviesen abiertos.

9.5 Borrado y Renombrado de Ficheros

Fichero ya existentes sobre el disco pueden borrarse o cambiar su nombre desde CLIPS sin necesidad de *salir* al sistema operativo. Las funciones correspondientes son:

- Borrado de ficheros.

  ```
  (remove <nombre-fichero>)
  ```

  - `<nombre-fichero>` debe de ser un string indicando la localización física del fichero que desea borrar.

- Cambiando de nombre de un fichero.

  ```
  (rename <nombre-fichero-viejo> <nombre-fichero-nuevo>)
  ```

  - `<nombre-fichero-viejo>` debe de ser un string indicando la localización física del fichero que desea cambiarle el nombre.
  - `<nombre-fichero-nuevo>` debe de ser un string indicando la localización física del fichero con el nombre cambiado.

Lección 10

OPERACIONES CON LAS CLASES DE CLIPS

10.1 Operaciones Matemáticas

10.1.1 Funciones Estandares

- Adición. Retorna la suma de sus argumentos

  ```
  (+ <expresión-numérica> <expresión-numérica>+)
  ```

  Ejemplos:

  ```
  CLIPS> (+ 1 2 3)
  6
  CLIPS> (+ 1 2.0 3.0)
  6.0
  CLIPS> (+ 1 2.0 -3.0)
  0.0
  ```

- Sustracción. Retorna la diferencia entre el primer argumento y la suma de todos los demás argumentos.

  ```
  (- <expresión-numérica> <expresión-numérica>+)
  ```

  Ejemplos:

  ```
  CLIPS> (- 10 2 3)
  5
  CLIPS> (- 10 2.0 3)
  5.0
  CLIPS> (- 10.0 2.0 -3.0)
  11.0
  ```

- Multiplicación. Retorna el producto de todos sus argumentos.

  ```
  (* <expresión-numérica> <expresión-numérica>+)
  ```

  Ejemplos:

  ```
  CLIPS> (* 1 2 3)
  6
  CLIPS> (* 1 2.0 -3)
  -6.0
  CLIPS> (* 1.5 2.5 -3.5)
  -13.125
  ```

- División. Retorna el valor del primer argumento dividido por cada uno de los argumentos siguientes. Convierte el primer argumento en real antes de realizar las divisiones. Convierte el primer argumento en real.

  ```
  (/ <expresión-numérica> <expresión-numérica>+)
  ```

  Ejemplos:

  ```
  CLIPS> (/ 24 2)
  ```

12.0
CLIPS> (/ 24 6)
4.0
CLIPS> (/ 24 2 3)
4.0

- División Entera. Retorna el valor del primer argumento dividido por cada uno de los argumentos siguientes. Convierte el primer argumento en real antes de realizar las divisiones. Los argumentos se convierten a enteros para poder realizar la división entera.
  (div <expresión-numérica> <expresión-numérica>+)

- Máximo. Retorna el máximo valor de todos los argumentos.
  (max <expresión-numérica>+)

- Mínimo. Retorna el mínimo valor de todos los argumentos.
  (min <expresión-numérica>+)

- Valor Absoluto. Retorna el valor absoluto de su único argumento.
  (abs <expresión-numérica>)

10.1.2 Funciones Extendidas

- Función Pi. Retorna el valor del número pi (3.141592653589793...).
  (pi)

- Raíz Cuadrada.
  (sqrt <expresión-numérica>)

- Potencial
  (** <expresión-numérica> <expresión-numérica>)

- Exponencial en base e.
  (exp <expresión-numérica>)

- Logaritmo Neperiano
  (log <expresión-numérica>)

- Logaritmo en base 10
  (log10 <expresión-numérica>)

- Redondeo. Devuelve el entero más cercano.
  (round <expresión-numérica>)

- Módulo. Devuelve el resto de la división entera de sus dos argumentos
  (mod <expresión-numérica> <expresión-numérica>)

10.1.3 Funciones Trigonométricas

Presentan la sintaxis:

(<funcion-trigonométrica> <expresión-numérica>)

Retornan un número real a partir de una expresión numérica (expresada en radianes). El valor de <funcion-trigonométrica> es alguno de los siguientes:

asin	arco seno	**asinh**	arco seno hiperbólico
acos	arco coseno	**acosh**	arco coseno hiperbólico
atan	arco tangente	**atanh**	arco tangente hiperbólico
acot	arco cotangente	**acoth**	arco cotangente hiperbólico
acsc	arco cosecante	**acsch**	arco cosecante hiperbólico
asec	arco secante	**asech**	arco secante hiperbólico
sin	seno	**sinh**	seno hiperbólico
cos	coseno	**cosh**	coseno hiperbólico
tan	tangente	**tanh**	tangente hiperbólica
cot	cotangente	**coth**	cotangente hiperbólico
csc	cotangente	**csch**	cotangente hiperbólico
sec	secante	**sech**	secante hiperbólico

10.1.4 Funciones de Conversión

- (float <expresión-numérica>). Convierte su único argumento a tipo real y retorna ese valor.

- (integer <expresión-numérica>). Convierte su único argumento a tipo entero y retorna ese valor.

10.2 Operaciones con Lexemas

- (str-cat <expresión>*).
  Concatena todos sus argumentos en un soólo string. Los argumentos pueden ser alguno de los siguientes tipos: lexemas, números o nombres de instacnias.

- (sym-cat <expresión>*).
  Concatena todos sus argumentos en un soólo símbolo. Los argumentos pueden ser alguno de los siguientes tipos: lexemas, números o nombres de instacnias.

- (sub-string <entero-1> <entero-2> <string>).
  Devuelve la porción del <string> que se encuentra entre las posiciones <entero-1> y <entero-2> (ambos inclusive).

- (str-index <string-1> <string-2>).
  Devuelve la primera posición donde aparece el <string-1> dentro del <string-2>. Si el <string-1> no está incluido en el <string-2>, devuelve FALSE.

- (eval <expresión-lexema>).
  Evalua el argumento, el cual puede ser un comando (p.e. una operación matemática), una constante o una variable global. Devuelve FALSE si se produce un error en la evaluación.

- `(upcase <expresión>)`.
  Retorna un string o símbolo donde los caracteres en minúsculas son pasados a mayúsculas.

- `(lowcase <expresión>)`.
  Retorna un string o símbolo donde los caracteres en mayúsculas son pasados a minúsculas.

- `(str-length <expresión>)`.
  Retorna la longitud de un lexema como un entero.

10.3 Operaciones Booleanas

10.3.1 Funciones Booleanas

- Función **o**. `(or <expresión>+)`.
  Retorna el valor TRUE si alguno de sus argumentos es TRUE.

- Función **y**. `(and <expresión>+)`.
  Retorna el valor TRUE si todos sus argumentos son TRUE.

- Función **no**. `(not <expresión>)`.
  Retorna el valor TRUE si alguno de su argumento es FALSE.

10.3.2 Comprobación de Tipos

Son funciones con un argumento de la forma:

`(<función> <expresión>)`

donde `<expresión>` es cualquier expresión admisible por CLIPS, que usualmente viene dada por una variable, y `<función>` es alguna de las siguientes:

- `numberp`.
  Retorna el valor TRUE si su argumento es un número (entero o real).

- `floatp`.
  Retorna el valor TRUE si su argumento es un número real.

- `integerp`.
  Retorna el valor TRUE si su argumento es un número entero.

- `evenp`.
  Retorna el valor TRUE si su argumento es un número entero par.

- `oddp`.
  Retorna el valor TRUE si su argumento es un número entero impar.

- `lexemep`.
  Retorna el valor TRUE si su argumento es un lexema (string o símbolo).

- `stringp`.
  Retorna el valor TRUE si su argumento es un string.

- `symbolp`.
  Retorna el valor TRUE si su argumento es un símbolo.

- `multifieldp`.
  Retorna el valor TRUE si su argumento es un valor multicampo.

10.3.3 Comparación de valores numéricos

- Igualdad. `(= <expresión-numérica> <expresión-numérica>+)`.
  Retorna el valor TRUE si el primer argumento es igual en valor a todos los demás argumentos.

  ✍ **Pueden aparecer problemas de precisión en la comparación de números reales. Es decir, números que pueden ser distintos pueden ser iguales, en la comparación, por problemas de redondeo.**

- Desigualdad. `(<> <expresión-numérica> <expresión-numérica>+)`.
  Retorna el valor TRUE si el primer argumento no es igual en valor a todos los demás argumentos.

- Mayor que. `(> <expresión-numérica> <expresión-numérica>+)`.
  Retorna el valor TRUE si el argumento i es mayor que el argumento $i + 1$.

- Mayor o igual que. `(>= <expresión-numérica> <expresión-numérica>+)`.
  Retorna el valor TRUE si el argumento i es mayor o igual que el argumento $i + 1$.

- Menor que. `(< <expresión-numérica> <expresión-numérica>+)`.
  Retorna el valor TRUE si el argumento i es menor que el argumento $i + 1$.

- Menor o igual que. `(<= <expresión-numérica> <expresión-numérica>+)`.
  Retorna el valor TRUE si el argumento i es menor o igual que el argumento $i + 1$.

10.3.4 Comparación de Strings

- `(str-compare <expresión-1> <expresión-2>)`.
  Retorna un entero según el resultado de la comparación. Los posibles resultados son:

  - Valor nulo (0). Si los strings son iguales.
  - Valor positivo (¿0). Si el primer string es menor que el segundo.
  - Valor negativo (¡0). Si el primer string es mayor que el segundo.

  La relación de orden considerada viene dada por la longitud de los strings y por la ordenación en el conjunto de caracteres ASCII.

10.3.5 Comparación de valores

Son funciones con un argumento de la forma:

(<función> <expresión> <expresión>+)

donde <expresión> es cualquier expresión admisible por CLIPS, y <función> es una función que compara los tipos y los valores de los argumentos. Es decir, son funciones que permiten comparar valores de cualquier tipo. Los posibles valores de <función> son:

- eq.
  Retorna el valor TRUE si el primer argumento es igual en valor a todos los demás argumentos.

- neq.
  Retorna el valor TRUE si el primer argumento no es igual en valor a todos los demás argumentos.

Lección 11

ESTRETEGIAS DE RESOLUCIÓN DE CONFLICTOS

Como se sabe, la agenda contiene las reglas activadas candidatas a ser disparadas. Cada módulo tiene su propia agenda y cada una de ellas funciona como una pila: la regla que se encuentra en el tope de la pila es la primera en ser ejecutada. Cuando una regla se añada a la agenda, se realiza atendiendo a estos factores:

1. La regla nueva, que tendrá un valor de prioridad, se coloca por debajo de todas aquellas reglas que tiene una mayor prioridad y por encima de las que tiene una prioridad menor a la nueva regla.

2. Entre las reglas con igual prioridad, se utiliza la estrategia de resolución de conflictos para reordanarlas.

3. En caso de empate, tras aplicar los pasos anteriores, CLIPS pondrá de forma arbitraria las reglas (normalmente por el orden de definición de las reglas).

En este apartado se verá como definir valores de prioridad en las reglas y se comentarán las siete estrategias que implementa CLIPS.

11.1 Prioridad de una Regla

La prioridad (en ingles, *salience*) de una regla no es má que un valor numérico. Por defecto es 0, y está comprendido entre -10000 y +10000. Sin embargo, el usuario puede asignar directamente dicho valor mediante una constante o una expresión numérica. La asignación se realiza poniendo la siguiente expresión entre el nombre de la regla y el primer elemento condicional de la regla:

```
(declare (salience <expresión-numérica>))
```

Por ejemplo,

```
(defrule R1                          (defrule test-2
(declare (salience 100))             (declare (salience (+ ?*variable* 10)))
(Elemento-Condicional-1)             (Elemento-Condicional-2)
=>                                   =>
(printout t "R1 disparada"crlf))     (printout t "R2 disparada"crlf))
```

Donde `?*variable*` es una variable global (apartado 13.2).
Otras funciones relacionadas con la prioridad de las reglas son:

- `(set-salience-evaluation <valor>)`
  donde `<valor>` es o *when-defined*, *when-activated*, o *every-cycle*. Es decir, los valores de prioridad pueden evaluarse en alguno de los siguientes momentos:
  - Cuando se define la regla: La prioridad se establece al realizar el `defrule` de la regla.
  - Cuando la regla se activa: La prioridad se establece al realizar el `defrule` de la regla y posteriormente cuando la regla se activa.
  - En cada ciclo de ejecución (ver apartado 8.2): La prioridad se establece al realizar el `defrule` de la regla, posteriormente cuando la regla se activa, y después de cada disparo.

Por defecto, la prioridad se evalua cuando se define la regla (opción when-define).

- `(get-salience-evaluation)`
  Esta función devuele el actual comportamiento de CLIPS sobre la evaluación de prioridades. Los valores que puede devolver son: when-defined, when-activated, o every-cycle.

- `(refresh-agenda [<nombre-del-modulo>])`
  Esta función fuerza a la reevaluación de las prioridades de las reglas de la agenda atendiendo a cómo se establezca dicha evaluación.

  Si no se especifica el parámetro, entonces la agenda del módulo actual es actualizado. Si se especifica el nombre del módulo se actualizará la agenda de dicho módulo. Si se utiliza el comodín, *, se actualizarán las agendas de todos los módulos.

Se propone que defina las siguientes reglas:

```
(defrule R1                    (defrule R2                    (defrule R3
(declare (salience <xx>))      (declare (salience <yy>))      (declare (salience <zz>))
=>(printout t "R1"crlf))       =>(printout t "R2"crlf))       =>(printout t "R3"crlf))
```

donde `<xx>`, `<yy>` y `<zz>` son tres valores distintos que pueden tomar los valores 10, 20 y 30. Asigne los valores a estas reglas y ejecute el programa. Podrá comprobar que la primera regla en disparse es aquella a la que haya asignado el valor de prioridad 30, seguida de la que tiene valor de prioridad 20, y por último se disparará la regla con prioridad 10.

Ya que `salience` es una herramienta muy potente para controlar la ejecución, podría pensar en usarla para establecer el orden de disparo de las reglas. De hecho, nos atreveríamos a decir a que si está empezando a conocer la programación basada en reglas, está viendo en el establecimiento de prioridades como su tabla de salvación. DESNGAÑESE. Si opta por abusar de esta potencia debe de ser consciente que el control explícito del control:

1. Está convirtiendo esta técnica de programación en una programación imperativa (o precedural) ya que estará indicando la secuencia de ejecución. Entonces, ¿para qué utilizar CLIPS?. Conformese con otro lenguaje más adecuado para estos propósitos.

2. Estará desarrollando un código muy pobre. La principal ventaja de la programación basada en reglas es que el programador no se preocupa de la ejecución del programa. Tan sólo se limita a codificar el conocimiento que se tiene de un problema para que, posteriormente, atendiendo a *criterios naturales* de ejecución, se disparen las reglas de forma óptima.

Tengo esto siempre presente: un experto no establece, en general, una jerarquía de prioridades *ad hoc* en las reglas. Aún en el caso de que fuese necesario establecer valores, se estima que no más de 3 o 4 valores son necesarios para sistemas expertos bien codificados. Es más, es casi seguro que si profundiza un poco más en el problema podrá eliminar los valores de prioridad de aquellas reglas en las que, inicialmente, considera que son imprescindibles.

Suponga que desea jugar al tic-tac-toe con la siguiente estrategia:

1. Si la casilla es ganadora y está libre, entonces poner una ficha en la casilla.

2. Si la casilla puede bloquear al contrario y está libre, entonces poner una ficha en la casilla que bloquea.

3. Si al casilla está libre, entonces poner una ficha en la casilla.

Esta estrategia puede implementarse del siguiente modo:

```
(defrule Casilla-Ganadora ; R1
(declare (salience 100))
?mov <- (muevoYo) ; Le toca al ordenador
(casilla ganadora)
=>
(retract ?mov)
(assert (mover-A ganadora)) )

(defrule Casilla-Bloqueo ; R2
(declare (salience 50))
?mov <- (muevoYo) ; Le toca al ordenador
(casilla bloquea)
=>
(retract ?mov)
(assert (mover-A bloquea)) )

(defrule Casilla-Ganadora ; R3
?mov <- (muevoYo) ; Le toca al ordenador
(casilla ?x&esquina|medio|lateral)
=>
(retract ?mov)
(assert (mover-A ?x)) )
```

Notar que si está disponible más de un tipo de casilla, se activarán las 3 reglas y se dispararán en el orden R1, R2 y R3. Notar también que cuando se dispare una, las demás se suprimirán de la agenda. Intuitivamente es claro que esto no resulta eficiente. Es más, esta forma de expresar las reglas viola un concepto básico en la programación basada en reglas: en la medida de lo posible debe eliminarse la interacción entre las reglas con objeto de que actuen oportunísticamente (y no condicionadamente). En este caso, salience expresa una relación implicita y condicionada entre las reglas. Si tenemos en cuenta que otro concepto básico en la programación basada en reglas es que una regla debe de representar lo más completamente posible una heurística, puede eliminarse relaciones condicionadas y/o implicitas añadiendo mas patrones (conocimiento) en las reglas. Así, las reglas anteriores puede expresarse como sigue:

```
(defrule Casilla-Ganadora ; RA
?mov <- (muevoYo) ; Le toca al ordenador
(casilla ganadora)
=>
(retract ?mov)
(assert (mover-A ganadora)) )
```

```
(defrule Casilla-Bloqueo ; R2
?mov <- (muevoYo) ; Le toca al ordenador
(casilla bloquea)
(not (casilla ganadora))
=>
(retract ?mov)
(assert (mover-A bloquea)) )
(defrule Casilla-Ganadora ; R3
?mov <- (muevoYo) ; Le toca al ordenador
(casilla ?x&esquina|medio|lateral)
(not (casilla ganadora))
(not (casilla bloquea))
=>
(retract ?mov)
(assert (mover-A ?x)) )
```

Lo que viene a reflejar que la auténtica estrategia es:

1. Si la casilla es ganadora y está libre, entonces poner una ficha en la casilla.
2. Si la casilla puede bloquear al contrario, no es ganadora y está libre, entonces poner una ficha en la casilla que bloquea.
3. Si al casilla está libre, no es ganadora y no bloquea al contrario, entonces poner una ficha en la casilla.

11.2 Estrategias Implementadas en CLIPS

11.2.1 Estrategia en Profundidad

Las nueva reglas, con la misma preferencia, se colocarán en el cima de la pila. Es decir, si R1 se activa para f-1 y R2 se activa para f-2, y f-1 se afirma antes que f-2, R2 se colocará por encima de R1.

Esta estrategia se activa mediante el comando `(set-strategy depth)`.

11.2.2 Estrategia en Anchura

Las nueva reglas, con la misma preferencia, se colocarán en el fondo de la pila. Es decir, si R1 se activa para f-1 y R2 se activa para f-2, y f-1 se afirma antes que f-2, R1 se mantendrá por encima de R2.

Esta estrategia se activa mediante el comando `(set-strategy breadth)`.

11.2.3 Estrategia basada en la Simplicidad

Las nueva reglas, con la misma preferencia, se colocan por encima de aquellas que tengan igual o mayor especificidad. La especificidad de una regla se determina por el número de comparaciones que

deben de realizarse en el antecedente de una regla. La especificidad es una cantidad que incremente en una unidad cada vez que:

- Se realiza una comparación con una constante o variable con valor asignado.
- La llamada a una función que involucre a los elementos condicionales `:`, `=` o `test`.

No añaden especificidad:

- Las llamadas a las funciones booleanas `not`, `and` y `or`. Aunque si pueden añadir especificidad sus argumentos.
- La llamada a una función dentro de una función que no añade especificidad a la regla.

Ejemplo, la regla

```
(defrule example
(item ?x ?y ?x)
(test (and (numberp ?x) (> ?x (+ 10 ?y)) (< ?x 100)))
=>)
```

tiene valor de especificidad 5. En efecto:

- El término `(item ?x ?y ?x)` tiene especificidad 1, ya que la primera aparición de ?x y de ?y no añaden especificidad, pero sí la segunda aparición de ?x.
- El término `(test (and (numberp ?x) (> ?x (+ 10 ?y)) (< ?x 100)))` tiene especificidad 4, ya que incrementan las siguientes funciones: `test`, `numberp`, `>` y `<`. Notar que no añaden especificidad las funciones `and` y `+`.

Esta estrategia se activa mediante el comando `(set-strategy simplicity)`.

11.2.4 Estrategia basada en la Complejidad

Las nueva reglas, con la misma preferencia, se colocan por encima de aquellas que tengan igual o meno especificidad.

Esta estrategia se activa mediante el comando `(set-strategy complexity)`.

11.3 Estrategias lex y mea

De entre las reglas con la misma prioridad se establece una ordenación temporal.

Lea el manual de CLIPS para más detalles.

Estas estrategias se activan mediante los comandos `(set-strategy lex)` y`(set-strategy mea)` respectivamente.

11.4 Estrategia Aleatoria

Each activation is assigned a random number which is used to determine its placement among activations of equal salience. This random number is preserved when the strategy is changed so that the same ordering is reproduced when the random strategy is selected again (among activations that were on the agenda when the strategy was originally changed).

Esta estrategia se activa mediante el comando `(set-strategy random)`.

Lección 12

EFICIENCIA

El proceso de activación puede expresarse en los siguientes pasos:

- Para cada regla hacer:

    1. Buscar el conjunto de hechos particulares que permiten determinar qué patrones de la regla se satisfacen.
    2. Si el antecedente se satisface para una instancia introducir la instancia de la regla en la agenda.

Durante la ejecución pueden añadirse o eliminarse hechos, lo que provoca que en cada ciclo de ejecución deba realizarse el proceso anterior. Es claro que este proceso resulta excesivamente lento para un gran conjunto de hechos y reglas.

Un modo de resolver el problema es tener en cuenta una propiedad que manifiestan los sistemas expertos basados en reglas: La redundancia temporal. Esto significa que la lista de hechos varía muy poco a lo largo del tiempo. En otras palabras, en cada ciclo de ejecución un porcentaje mínimo de reglas pueden verse afectadas por la modificación de la lista de hechos.

CLIPS utiliza el algoritmo *Rete Pattern Matching* para actualizar la agenda. Este algoritmo está diseñado teniendo en cuenta la redundancia temporal. En vez de comprobar la (de)sactivacación de las reglas una a una, se observan los hechos que se han añadido y/o retractado y se comprueba si las reglas que dependen de éstos hechos se mantienen en la agenda. Obviamente, esto conlleva que cada regla gestione la lista de hechos que satisfacen (parcialmente) su antecedente.

Por ejemplo, imagine la siguiente situación: una regla con antecedentes A, B y C. Cuando ejecute su programa se satisfacen A y B. Pasado un número de ciclos se produce un único hecho, el C. Atendiendo al algoritmo anterior la ocurrencia de C debería producir la activación de la regla. Notar que **no** se dice que ante la ocurrencia de C se vuelva a comprobar toda la regla. S¨olo el hecho C es de interés, pues éste ha sido lo único que se ha añadido a la lista de hechos.

Es claro que esto conlleva que cada reglas dee de recordar que es lo que ya se satisfaciá. Est tipo de información es lo que se conoce como coincidencia parcial (partial match). Una coincidencia parcial para una regla es cualquier conjunto de hechos que satisfacen algunos patrones de la regla, de entre todos los que componene el antecedente. Mas concretamente, esos patrones están formados desde el primer patrón de la regla hasta un patrón posterior, incluyendo todos los intermedios, o bien los patrones individuales de la regla. Así una regla con tres patrones, A, B Y C, tiene los siguientes patrones parciales A, B, C, AB, ABC. Cuando todas las coincidencas se cumplen, la regla se activa. Esta forma de trabajar de CLIPS nos puede ayudar para mejorar la eficiencia de un programa.

Para conocer las coincidencias parciales de una regla se utiliza el comando:

```
(matches <nombre-regla>
```

que resulta muy útil para conocer qué reglas generan un número elevado de coincidencias parciales. Por ejemplo, ejecute el siguiente fichero batch y analice el resultado.

```
(defrule regla (a) (b) (c) => )
(assert (a) (b))
```

```
(matches regla)
(agenda)
(retract 1)
(matches regla)
(agenda)
(assert (c))
(matches regla)
(agenda)
(assert (b))
(matches regla)
(agenda)
```

Ya que CLIPS utiliza el algoritmo Rete Pattern Matching, es importante que no se generen un número elevado de coincidencias parciales. El siguiente ejemplo muestra la diferencia.

```
(deffacts Informacion
(datos a c)
(elemento a)
(elemento b)
(elemento c))

(defrule R1
(datos ?x ?y)
(elemento ?x)
(elemento ?y) => )

(defrule R1
(elemento ?x)
(elemento ?y)
(datos ?x ?y) => )
(reset)
(matches R1)
(matches R2)
```

A continuación se da una pequeña guía de cómo debe ordenarse los patrones.

- Los patrones más especificos, con más restricciones, deben de ir primero, ya que, generalmente, producirán el número más pequeño de coincidencia y tendrán un mayor número de variables asignadas. Lo que provocará una mayor restricción en los otros patrones.

- Los patrones temporales deben de ir los últimos. Es decir, los correspondientes a hechos que se añaden y borran frecuentemente de la lista de hechos. Esto provacará un menur número de cambios en las coincidencias parciales.

- Los patrones más *raros* deben ser los primeros. Los patrones correspondientes a hechos que se presentarán pocas veces en la lista de hechos, se se colocan al principio reducirán el número de coincidencias parciales.

Hay que indicar que estos criterios de ordenación no hay que tomarselo de un modo tajante. De hecho, pueden producirse conflictos. Por ejemplo, un hecho temporal puede ser muy específico como el caso de indicadores o banderas. Si no está presente y se coloca al principio, no se producirán coincidencias.

Realmente no existe ningún conjunto de criterios teóricos que permitan optimizar el código. Tan sólo la experiencia y la paciencia de reordenar los patrones (ensayo y error) permiten determinar los cambios que deben hacerse para que el sistema vaya más rápdido.

Otro criterio interesante es el siguiente: La función test debe ponerse lo antes posible. Por ejemplo la regla R1 puede optimizarse reescribiendola como la regla R2.

```
(defrule R1
?p1 <- (valor ?x)
?p2 <- (valor ?y)
?p3 <- (valor ?z)
(test (and  (neq ?p1 ?p2)
(neq ?p2 ?p3)
(neq ?p1 ?p3)) => )


(defrule R2
?p1 <- (valor ?x)
?p2 <- (valor ?y)
(test (neq ?p1 ?p2))
?p3 <- (valor ?z)
(test (and (neq ?p2 ?p3)
(neq ?p1 ?p3)) => )
```

Lección 13

VARIABLES

13.1 Variables Locales

Son aquellas que se crean en el ambiente de una regla. Normalmente se definen en el antecedente de una regla, es decir suelen utilizarse en el proceso de reconocimiento de patrones, para capturar algún valor concreto del campo de un hecho. Su uso lo estudiado en el apartado 8.5.

13.2 Variables Globales

El constructor `defglobal` permite definir variables globales, usándose la función `bind` para establecer su valor, el comando `reset` o el destructor `undefglobal` se utilizan para su borrado y el comando `ppdefglobal` se utiliza para visualizar el contenido del constructor.

La sintaxis del constructor de variables es:

```
(defglobal [<nombre-del-modulo>] <asignación>*)
<asignación>          ::=  <nombre-de-variable> = <valor-o-expresion>
<nombre-de-variable>  ::=  ?*<símbolo>*
```

EJEMPLO:

```
(defglobal
?*x* = 3
?*y* = ?*x*
?*z* = (+ ?*x* ?*y*)
?*q* = (create$ a b c))
```

La componente opcional `<nombre-del-modulo>` indica el módulo en el que el constructor será definido. Si no se especifica, se definirá en el módulo actual. Puede definir tantos constructores `defglobal` como considere y cada uno de ellos puede tener tantas variables como quiera. En el caso de que existan dos variables iguales en dos constructores distintos, el valor de al variable será reemplazado por el que se le asigne en el último constructor.

Pueden utilizarse en el proceso de reconocimiento de patrones como se muestra en el siguiente ejemplo:

```
(defrule example
(fact ?y&:(> ?y ?*x*))
=>)
```

y pueden usarse de forma totalmente análoga a las variables locales. Sin embargo hay dos excepciones:

1. No pueden utilizarse como un parámetro para deffunction, defmethod,o message-handler.
2. En general, no pueden utilizarse para capturar valores de casillas en patrones, del mismo modo en que se hace para variables locales.

   Por ejemplo la siguiente regla es ilegal

```
(defrule example
(fact ?*x*)
=>)
```

Lección 14

LENGUAJE IMPERATIVO

Leguajes como C, Pascal o Ada, son lenguajes imperativos o procedurales. CLIPS, sin embargo, es un lengajes simbólico. No obstante, CLIPS proporcional la posibilidad de introducir instrucciones imperativas si lo desea.

14.1 Asignación de Variables

Puede crear variables o modificar el valor de éstas mediante la función:

`(bind <variable> <expresión>*)`

donde `<variable>` puede ser local o global. Si no se especifica el valor de la `<expresión>` la variable es limpiada. Si se utiliza más de una `<expresión>`, todas las expresiones son evaluadas y agrupadas como un valor multicampo y el resultado se almacena en la `<variable>`.

Ejemplo (del manual):

```
CLIPS> (defglobal ?*x* = 3.4)
CLIPS> ?*x*
3.4
CLIPS> (bind ?*x* (+ 8 9))
17
CLIPS> ?*x*
17
CLIPS> (bind ?*x* (create$ a b c d))
(a b c d)
CLIPS> ?*x*
(a b c d)
CLIPS> (bind ?*x* d e f)
(d e f)
CLIPS> ?*x*
(d e f)
CLIPS> (bind ?*x*)
3.4
CLIPS> ?*x*
3.4
```

Otro ejemplo de la función `bind` puede verlo en la sección 8.12.

14.2 Función If...then...else

Esta función proporciona la conocida estructura de control if...then...else del lenguaje imperativo.

```
(if <expresión>
then
<acción-1>*
 [else
<acción-2>*])
```

Si la `expresión` se evalua con no FALSE se realizarán la acciones especificadas en `<acción-1>` en otro caso se realizarán, opcionalmente, las acciones indicadas en `<acción-2>`. Puede poner tantas acciones como quiera. Esta función retornará el valor de la evaluación la última expresión o acción.

Por ejemplo, suponga que tiene las siguientes dos reglas:

```
(defrule SuperNota              (defrule Nota
(calificacion 10)               (califacion ?valor& 10)
=>                              =>
(printout t                     (printout t ?valor
 "Tiene matricula"crlf))         "no es matricula"crlf))
```

Éstas puede unirla en una solo como sigue:

```
(defrule Calificacion
(calificacion ?valor)
=>
(if (= ?valor 10) then
   (printout t "Tiene matricula"crlf)
 else
   (printout t ?valor "no es matricula"crlf) )
```

Sin embargo, es recomendable que no abuse del lenguaje imperativo por lo que es aconsejable que utilice las dos reglas anteriores a ésta última regla.

14.3 Función While

Se utiliza para realizar un conjunto de acciones hasta que deje de cumplirse una condición.

```
(while <expresión> [do]
<acción>*)
```

Las acciones del bucle, `<acción>`, se realizarán hasta que `<expresión>` se evalue con el valor FALSE. Pueden utilizarse las funciones `break` y `return` para finalizar el bucle prematuramente. La función devuelve el valor FALSE salvo que la función `return` se utilice para finalizar el bucle.

```
(defrule Cuenta-Atras
(valor ?v)
=>
(while (> ?v 0)
(printout t "Valor "?v crlf)
(bind ?v (- ?v 1)) ) )
```

14.4 Función Loop-for-count

La función `loop-for-count` permite realizar un conjunto de acciones un número determinado de veces. Es el equivalente al `for` en los lenguajes imperativos.

```
(loop-for-count <rango> [do] <acción>*)
<rango>          ::=  <índice-final> |
                      (<variable> [<índice-inicio> <índice-final>])
<índice-inicio>  ::=  <expresión-entera>
<índice-final>   ::=  <expresón-entera>
```

Se desarrollan la acciones `<acción>*` tantas veces como se haya especificado en `<rango>`.

```
CLIPS> (loop-for-count 2 (printout t "Saludo al mundo"crlf))
Saludo al mundo
Saludo al mundo
FALSE
```

Si tan sólo se especifica <índice-final>, se realizarán exactamente tantas veces como se especifique en este índice. Si se especifican <índice-inicio> e <índice-final>, entonces:

- <índice-inicio> ¿ <índice-final>, entonces no se realizarán las acciones.
- <índice-inicio> ¡ <índice-final>, entonces se realizarán <índice-final>-<índice-inicio>+1 veces las acciones.

La función retorna el valor FALSE salvo que se utilice la función `return` para finalizar

EJEMPLO:

```
CLIPS>(loop-for-count (?contador1 2 4) do
     (loop-for-count (?contador2 1 3) do
         (printout t ?contador1  ?contador2 crlf)))
2 1
2 2
2 3
3 1
3 2
3 3
4 1
4 2
4 3
FALSE
```

14.5 Función Return

Se utiliza para interrumpir la ejecución funciones, ciclos, etc. Su sintaxis es:

(return [<expresión>])

Sin argumento, no retorna ningún valor. Con argumento, retoran el valor de la <expresión> espcificada.

```
CLIPS> (deffunction semaforo (?color)
(if (eq ?color rojo) then
  (return "No puedes cruzar"))
(if (eq ?color amarillo) then
  (return "Parate por seguridad"))
"Puedes Pasar")
CLIPS> (semaforo rojo)
"No puedes cruzar"
CLIPS> (semaforo amarillo)
"Parate por seguridad"
CLIPS> (semaforo verde)
"Puedes Pasar"
```

14.6 Función Break

Se utiliza para terminar la ejecución de un grupo de instrucciones que se encuentran en un flujo de control (bucles). Su sintaxis es

```
(break)
```

EJEMPLO:

```
CLIPS>(deffunction iterate (?num)
(bind ?i 0)
(while TRUE do
  (if (>= ?i ?num) then
    (break))
  (printout t ?i )
  (bind ?i (+ ?i 1)))
  (printout t crlf))
CLIPS> (iterate 1)
0
CLIPS> (iterate 10)
0 1 2 3 4 5 6 7 8 9
```

14.7 Función Switch

Permite realizar un conjunto de acciones para un valor concreto de una variable.

```
(switch <condición>
   <sentencia-caso>
   <sentencia-caso>+
   [<sentencias-por-defecto>])
<sentencia-caso>          ::=  (case <comparación> then <accción>*)
<sentencias-por-defecto>  ::=  (default <acción>*)
```

Esta función necesita como mínimo dos `<sentencia-caso>` y no deben de existir dos iguales. Su funcionamiento es el siguiente

1. La función `swtich` evalua `<condición>`, que normalmente es una variable.

2. El valor obtenido en el caso anterior es comparado con los valores `<comparación>` que hay en cada `<sentencia-caso>`. Si hay una igualdad en valor para una `<sentencia-caso>`, entonces se realizan las acciones que para este caso se especifiquen.

3. Si no existe ninguna equivalencia de valores, entonces se realizarn las acciones especificadas para `<sentencias-por-defecto>`.

```
CLISP> (deffunction dias (?no)
(bind ?resto (mod ?no 2))
(bind ?mitad (<= ?no 7))
(switch ?resto
    (case 0 then
        (switch ?mitad
        (case TRUE then
            (if (= ?no 2) then
                (printout t "28 dias"crlf)
            else
                (printout t "30 días"crlf)
            ))
        (case FALSE then (printout t "31 días"crlf))
        ))
    (case 1 then
        (switch ?mitad
        (case TRUE then (printout t "31 dias"crlf))
        (case FALSE then (printout t "30 días"crlf))
        ))
    (default algo falla) ) )

CLIPS> (dias 6)
30 días
CLIPS> (dias 7)
31 dias
CLIPS> (dias 8)
31 días
```

Lección 15

FUNCIONES

Puede definir funciones directamente en CLIPS con el constructor

```
(deffunction  <nombre> [<comentario-opcional>]
              (<parámetro>* [<parámetro-comodín>])
              <accción>*)
<parámetro>           ::=  <variable-unicampo>
<parámetro-comodín>   ::=  <variable-multicampo>
```

El constructor `deffunction` consta de cinco elementos:

1. Un nombre, `<nombre>`, único para cada función.

2. Un comentario opcional.

3. Una lista de 0 o más parámetros. La componente `<parámetro>` indica estos parámetros. Cuando se llame a la función `<nombre>`, se hará con tantos argumentos como se hayan especificado en su definición.

4. Un parámetro comodín, opcional, para manejar un número variable de argumentos. En el caso de que se especifique `<parámetro-comodín>`, la función `<nombre>` se podrá llamar a la función con más argumentos de los especificados en la componente `<parámetro>`. Esos parámetros de más, se agruparán en un valor multicampo sobre los que posteriormente podrá aplicar las funciones CLIPS sobre multicampos (p.e. length o nth).

   Ejemplo (del manual):

   ```
   CLIPS>(deffunction argumentos (?a ?b $?c)
   (printout t ?a  ?b "y "(length ?c) "extras:  "?c
   crlf))
   CLIPS> (argumentos 1 2)
   1 2 y 0 extras:  ()
   CLIPS> (argumentos a b c d)
   a b y 2 extras:  (c d)
   ```

5. Una secuencia de acciones, o expresiones, que ser realizarán en el orden especificado cuando se llame a la función.

Una vez llamada a la función, ésta devolverá el valor correspondiente a la evaluación de la última acción. Si se produce algún error, se retornará el siímbolo FALSE.

Si desea que una función llame a otra función, la segunda tendrá que haberse declarado y definido previamente. La única excepción es que la función sea recursiva.

EJEMPLO:

```
(deffunction factorial (?a)
(if (or (not (integerp ?a)) (< ?a 0)) then ; comprueba si es entero o negativo
    (printout t "Factorial Error!"crlf) ; en cuyo caso no puede calcularse (?a)!
else
    (if (= ?a 0) then ; se supone que el factorial de 0 es 1
        1
    else
        (* ?a ( factorial (- ?a 1) ) )
)))
```