

TUTORIAL DE CLIPS

VERSION 1.01 / December 22, 2006

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

Este documento NO es de libre distribución. No puede distribuirse por ningún medio de ningún soporte. Su copia para su distribución no está permitida. Usted puede tener sólo una única copia para su lectura, no para su distribución. Este documento sólo puede obtenerse por correo electrónico a ldaniel en um.es. Cualquier otro lugar de descarga o medio de distribución no está permitido salvo por permiso expreso del autor.

NOTA: No se pretende con este documento realizar un manual completo y detallado sobre esta herramienta. De hecho, todo este tutorial está realizado a partir de sus manuales. Para una documentación completa recurra a los manuales existentes y actualizados de CLIPS.

Luis Daniel Hernández Molinero
Dpto. de Ingeniería de la Información y las Comunicaciones.
Fac. Informática Universidad de Murcia.
Campus de Espinardo. 30071 Murcia. España
e-mail: ldaniel en um.es

Contenidos

1	INTRODUCCIÓN A CLIPS	6
2	ELEMENTOS BÁSICOS DE CLIPS	7
2.1	Tipos de Datos	7
2.2	Funciones	9
2.3	Constructores	9
3	ABSTRACCIÓN DE DATOS	10
3.1	Hechos	10
3.2	Objetos	11
3.3	Variables Globales	11
4	REPRESENTACIÓN DEL CONOCIMIENTO	12
4.1	Representación Heurística: Reglas	12
4.2	Representación Procedural	12
4.2.1	COOL	13
5	SINTAXIS	14
6	EJECUTANDO CLIPS	16
6.1	Modos de Ejecución	16
6.2	Entrando en CLIPS: <code>clips</code> , <code>xclips</code> ,	16
6.3	Saliendo de CLIPS: <code>exit</code>	17
6.4	La Primera Sesión	17
6.5	Limpiando la Memoria de Trabajo: <code>reset</code> y <code>clear</code>	18
6.5.1	¿Qué hace exactamente <code>reset</code> ?	19
6.5.2	¿Qué hace exactamente <code>clear</code> ?	19
7	PLANTILLAS Y HECHOS	20
7.1	¿Qué es un Hecho?	20
7.2	Plantillas	20
7.2.1	Construcción: <code>deftemplate</code>	20
7.2.2	Casillas simples y múltiples	21
7.2.3	Visualización: <code>ppdeftemplate</code> , <code>list-deftemplates</code>	22
7.2.4	Destrucción de Plantillas: <code>undeftemplate</code>	23
7.2.5	Tipos de Hechos	23
7.3	Creación, modificación, visualización y borrado de hechos	25
7.3.1	Afirmación de hechos: <code>assert</code>	25
7.3.2	Visualización de hechos: <code>facts</code>	26
7.3.3	Eliminación de hechos: <code>retract</code>	29
7.3.4	Modificación de hechos: <code>modify</code>	29
7.3.5	Duplicación de hechos: <code>duplicate</code>	31
7.4	Comandos de Depuración sobre Hechos: <code>[un]watch facts</code>	31

7.5	Creación, Visualización y Destrucción de Hechos Iniciales	31
7.5.1	Creación de hechos iniciales: <code>deffacts</code>	32
7.5.2	Visualización de Hechos Iniciales: <code>ppdfeffacts</code> , <code>list-deffacts</code>	32
7.5.3	Borrado de Hechos Iniciales: <code>undeffacts</code>	33
7.5.4	Una sesión con hechos iniciales	34
7.6	Ejercicios Propuestos sobre Hechos	36
8	RESTRICCIONES DE LOS ATRIBUTOS	38
8.1	Restricciones sobre Tipo, Rango y Cardinalidad de los Datos en una Casilla	38
8.1.1	Tipos de Atributos	38
8.1.2	Tipos de Atributos Constantes Permitidos	39
8.1.3	Rango de los Atributos	40
8.1.4	Cardinalidad de los Atributos	41
8.2	Restricciones sobre los Valores por Defecto en una Casilla	41
8.2.1	Valores por Defecto de un Atributo	42
8.3	Algunas Sesiones de Ejemplo	43
9	INTRODUCCIÓN AL MANEJO DE REGLAS	45
9.1	Definición de Reglas: <code>defrule</code>	45
9.2	Destrucción y Visualización de Reglas: <code>undefrule</code> , <code>ppdefrule</code> , <code>list-defrules</code>	46
9.3	Ejecución de Reglas en CLIPS	46
9.3.1	Iniciar la Ejecución de Reglas: <code>run</code>	47
9.3.2	Parar la Ejecución de Reglas: <code>halt</code>	48
9.3.3	Puntos de Corte: <code>set-break</code> , <code>remove-break</code> , <code>show-break</code>	50
9.4	Visualización de Reglas Activas: <code>agenda</code> , <code>[un]watch</code> <code>activations</code>	51
9.5	Visualización de Reglas Disparadas: <code>[un]watch</code> <code>rules</code>	52
9.6	Usando la Instancia de Una Regla más de una vez	54
10	ELEMENTOS CONDICIONALES BASADOS EN PATRONES	56
10.1	Elementos Condicionales	56
10.2	Literales	58
10.3	Variables Unicampo	58
10.4	Comodines	59
10.5	Variables Multicampo	60
10.6	Captura de Direcciones de Hechos	60
10.7	Variables Globales	65
10.7.1	Constructor <code>defglobal</code> y Comandos Relacionados	65
10.7.2	Función <code>bind</code>	67
10.8	Restricciones Conectivas: <code>~</code> , <code>&</code> , <code> </code>	67
10.9	Restricciones de Valores de Retorno: <code>=</code>	68
10.10	Restricción Predicado: <code>:</code>	71
11	OTROS ELEMENTOS CONDICIONALES	72
11.1	Conectivas entre Elementos Condicionales	72
11.2	Elemento Condicional <code>test</code>	73
11.3	Elemento Condicional <code>exists</code>	74

11.4	Elemento Condicional <code>forall</code>	74
11.5	Elemento Condicional <code>logical</code>	76
11.6	Ejercicios Propuestos sobre Reglas	78
12	ESTRATEGIAS DE RESOLUCIÓN DE CONFLICTOS	80
12.1	Prioridad de una Regla: <code>salience</code>	80
12.1.1	Funciones relacionadas con <code>salience</code>	81
12.2	El uso de <code>salience</code> puede ser problemático	82
12.3	Estrategias Implementadas en CLIPS: <code>set-strategy</code> y <code>get-strategy</code>	85
12.3.1	Estrategia en Profundidad: <code>depth</code>	86
12.3.2	Estrategia en Anchura: <code>breadth</code>	86
12.3.3	Estrategia Aleatoria: <code>random</code>	87
12.3.4	Estrategia basada en la Simplicidad: <code>simplicity</code>	88
12.3.5	Estrategia basada en la Complejidad: <code>complexity</code>	89
12.3.6	Estrategias temporales: <code>lex</code> y <code>mea</code>	89
13	DEPURACIÓN DE PROGRAMAS	90
13.1	Eficiencia: <code>matches</code>	90
13.2	Comentarios	93
13.2.1	Puntos de Corte: <code>set-break</code> , <code>remove-break</code> , <code>show-break</code>	93
13.3	Mostrando Toda la Memoria: <code>[un]watch</code> y <code>list-watch-items</code>	95
13.4	Traza de los programas: <code>dribble-on</code> y <code>dribble-off</code>	95
14	LENGUAJE IMPERATIVO	97
14.1	Definición de Funciones Propias	97
14.2	Asignación de Variables: <code>bind</code>	98
14.3	Función <code>if...then...else</code>	98
14.4	Función <code>return</code>	101
14.5	Función <code>while</code>	101
14.6	Función <code>break</code>	102
14.7	Función <code>loop-for-count</code>	102
14.8	Función <code>switch</code>	103
14.9	Algunos Ejemplos con Solución	104
15	FICHEROS	107
15.1	Cargar y Salvar Hechos	107
15.2	Cargar y Salvar Constructores: <code>load</code> y <code>save</code>	107
15.3	Ejecución de Comandos Desde un Fichero	108
15.4	Abriendo y Cerrando Ficheros Generales	108
15.4.1	Abriendo Fichero Lógicos: <code>open.</code>	109
15.4.2	Escritura sobre Ficheros Lógicos: <code>printout</code> y <code>format.</code>	109
15.4.3	Lectura desde Ficheros Lógicos: <code>read</code> y <code>readline.</code>	111
15.4.4	Cerrando Ficheros Lógicos: <code>close.</code>	112
15.5	Borrado y Renombrado de Ficheros: <code>remove</code> y <code>rename</code>	112

16 ALGUNAS OPERACIONES CON ALGUNAS CLASES DE CLIPS	113
16.1 Operaciones Matemáticas	113
16.1.1 Funciones Estandares	113
16.1.2 Funciones Extendidas	115
16.1.3 Funciones Trigonométricas	115
16.1.4 Funciones de Conversión	116
16.2 Operaciones con Lexemas	116
16.3 Operaciones Booleanas	117
16.3.1 Funciones Booleanas	117
16.3.2 Comprobación de Tipos	117
16.3.3 Comparación de valores numéricos	118
16.3.4 Comparación de Strings	118
16.3.5 Comparación de valores	119
16.4 Operaciones con Multicampos	119

Copia para la impartición de la asignatura
Idaniel en um.es
Ingeniería del conocimiento y S.I.

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.

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 carácter 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 actúa como delimitador y es entendido por CLIPS que lo que se encuentre a partir de ahí hasta el final de línea 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 impresión 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 mismo 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. Estas 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**, **defacts**, **deftemplate**, **defglobal**, **deffunction**, **defclass**, **definstances**, **defmessage-handler**, **defgeneric** y **defmethod**.

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 distinguen 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.

Cada uno de los campos de un hecho recibirá el nombre de **casilla**, que puede simple o múltiple. Es **simple** cuando la casilla sólo tiene un valor y es **múltiple** cuando puede tener más de un valor.

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 añade 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, una dirección de un hecho o una instancia de una clase definida por el usuario. Cada objeto se describe en dos partes: propiedades y comportamiento.

Si una serie de objetos presentan propiedades y comportamientos iguales, todos ellos se agrupan en una **clase**. O en otras palabras, una **clase** es un modelo (o plantilla) que recoge las propiedades y comportamientos comunes de una serie de objetos. Dada una clase, cada uno de los objetos que la componen recibe el nombre de **instancia**. Para referirnos a una instancia utilizaremos su nombre o su dirección.

Los objetos se describen con dos componentes: propiedades y comportamiento. Las **propiedades** de los objetos se determinan especificando valores en las *casillas* de la clase del objeto, de forma similar a las casillas de los hechos. El **comportamiento** se determina mediante código procedural que está asociado a la clase del objeto y recibe el nombre de *manipuladores-de-mensajes* (message-handlers).

Existen diferencias claras entre los objetos y los hechos. Una de ellas, quizás la más clara, es que las propiedades y comportamiento de los objetos pueden heredarse. Es decir, a una clase, y en particular a sus objetos, se le puede asignar directamente casillas y manipuladores-de-mensajes de otras clases.

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**. Conceptualmente, 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.

REPRESENTACIÓN DEL CONOCIMIENTO

CLIPS permite definir el conocimiento 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 consta de un conjunto de **condiciones** que deben satisfacerse para que la regla se aplique (para que se realicen las acciones que dice el consecuente). 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 se aplica una regla, 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 **estrategia de resolución de conflictos** que selecciona qué regla deber de aplicarse primero.

4.2 Representación Procedural

CLIPS permite conocimiento procedural como se hace en lenguajes 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 útiles 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 valor que devuelve es el valor de la última expresión evaluada en **deffunction**.

Las **funciones genéricas**, similares a **deffunction**, 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 puede *partirse* en **módulos**. Intuitivamente, cada módulo engloba conocimiento similar y se construye mediante el comando **defmodule**. En cada módulo puede tener sus propios constructores y puede especificarse cuáles serán visibles por otros módulos y cuáles no.

Como se ha dicho, los objetos se describen con dos componentes: propiedades (casillas de la clase del objeto) y comportamiento (manejadores-de-mensajes). Un manejador-de-mensajes es código procedural asociado al objeto. Pero también lo son los pasos-de-mensajes (message-passing) que son los que se utilizan para manipular los objetos (construcción, destrucción, etc).

4.2.1 COOL

COOL son las iniciales de “CLIPS Object-Oriented Language”. Es un lenguaje que incluye tanto elementos de abstracción de datos como representación del conocimiento y, como indica su nombre, está basado en la programación orientada a objetos (POO).

Un sistema basado en POO se caracteriza por las siguientes propiedades:

- ⇒ Abstracción. Permite representar conceptos complejos mediante representaciones intuitivas con un alto nivel de abstracción.
- ⇒ Encapsulamiento. Todos los detalles de implementación de un objeto están enmascarados (encapsulados) para el usuario, pero puede acceder a ellos mediante una interface externa.
- ⇒ Herencia. Una clases pueden copiar (heredar) las propiedades y comportamientos de otra clase.
- ⇒ Polimorfismo. Diferentes objetos responde al mismo mensaje en función de su naturaleza.
- ⇒ Ligadura dinámica. En tiempo de ejecución se puede posponer la selección de qué manejadores-de-mensajes se utilizarán cuando el objeto recibe un mensaje.

COOL dispone de un sistema de preguntas para manejar las instancias de las clases definidas por el usuario en función de los criterios que establezca dicha preguntas. Así, por ejemplo, se pueden determinar y/o desarrollar acciones sobre ese conjunto de instancias, comprobar si se verifican ciertas relaciones, salvar características, etc.

SINTAXIS

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 ')'.
Copia para la impartición en un curso de Ingeniería de Software y Conocimiento

Sin embargo, cada comando presenta una sintaxis propia y puede presentar una serie de opciones que pueden ser optativas u obligatorias. Para poder utilizar una notación general para todos ellos es necesario seguir unas normas básicas de escritura. Seguiremos las mismas normas que las empleadas en los manuales de CLIPS con objeto de que al lector le resulte familiar dicha sintaxis cuando recurra a esos manuales.

Seguiremos 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, 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 equivalente 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 separados 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`

*Copia para la impartición de la asignatura
Idaniel en um.es
Ingeniería del conocimiento y S.I.*

EJECUTANDO CLIPS

6.1 Modos de Ejecución

CLIPS puede ejecutarse de modo interactivo o en modo batch. El primero consiste en que tras arrancar CLIPS nos saldrá el cursor `CLIPS>` 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:

- ✧ En modo interactivo la entrada de comandos es la consola (pantalla). En este caso los comandos deben teclearse uno a uno por el usuario. Caso de que sea necesario empezar desde el principio, el usuario tendrá que volver a teclear todos los comandos de nuevo.
- ✧ En modo batch la entrada de comandos se realiza leyendo el contenido de un fichero. En este caso los comandos se escriben en el fichero leído por CLIPS. Caso de que sea necesario empezar desde el principio, el usuario no tiene que teclear nada, pues lo que debería de teclear ya se encuentra escrito en el fichero.
- ✧ Un modo intermedio de ejecución consiste en cargar de un fichero las definiciones básicas necesarias para trabajar (es decir, los constructores) y seguir trabajando en modo interactivo.

6.2 Entrando en CLIPS: `clips`, `xclips`, ...

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

Sintaxis 6.1: `clips`

La sintaxis que debe utilizar desde la línea de comandos para entrar en CLIPS es la que se muestra en la Sintaxis 6.1. Más concretamente:

- ✧ Si no se introduce ninguna opción CLIPS se ejecutará en modo interactivo.
- ✧ Para la opción `-f`, `<fichero-comandos>` es el fichero que contiene los comandos CLIPS. Al utilizar esta opción, CLIPS se ejecutará en modo batch.

Esta opción es equivalente a entrar en CLIPS en modo interactivo, y después utilizar el comando `batch` (ver apartado 15.3 para detalles.)

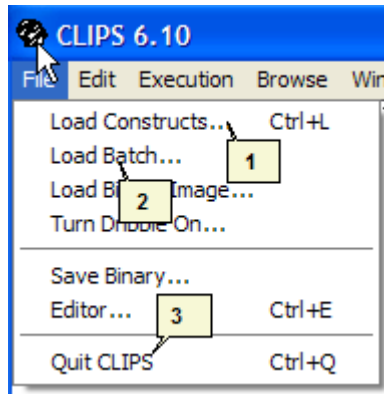


Figura 1: Menú *File* de *clipswin*

- ⇒ Para la opción `-l`, `<fichero>-constructores` es el fichero que contiene los constructores que se utilizarán. Tras cargarlos CLIPS entrará en modo interactivo a la espera de que le de órdenes.

Esta opción es equivalente a entra en CLIPS en modo interactivo, y después utilizar el comando `load` (ver apartado 15.2 para detalles.)

Si utiliza algún entorno de ventanas como *xclips*, *clipswin*, *wxclips*, etc ... consulte el manual para saber cómo utilizarlos. No obstante en la Figura 1 se muestra la opción *File* de *clipswin*. Las etiquetas 1 y 2 muestra cómo utilizar los comandos `load` y `batch`, lo que equivale a usar las opciones `-l` y `-f` respectivamente.

6.3 Saliendo de CLIPS: `exit`

Salir de CLIPS no tiene ningún problema. Basta utilizar el comando (`exit`) que presenta la Sintaxis 6.2. Si utiliza alguna versión de CLIPS con entorno de ventanas posiblemente en el menú *File* tendrá una opción para salir. Por ejemplo, la etiqueta 3 de la Figura 1 es la opción que deberá seleccionar para salir de *clipswin*.

(`exit`)

Sintaxis 6.2: `exit`

6.4 La Primera Sesión

Como primera sesión de este tutorial, se muestra en la Sesión 6.1 los comandos que debería ejecutar para entrar y salir de CLIPS. La enumeración que aparece en el margen izquierdo indica los pasos de la sesión y no aparecen nunca en una sesión real. Esta enumeración se utilizará para indicar aquellos pasos que el usuario debe seguir así como los pasos de los mensajes de salida de CLIPS que surgen en una sesión real.

Esta primera sesión consta de 6 pasos. Los comentamos:

```
1: bash%> clips ↵
2:          CLIPS (V6.10 07/01/98)
3: CLIPS> (+ 2 3) ↵
4: 5
5: CLIPS> (exit) ↵
6: bash%>
```

Sesión 6.1: Primera sesión con CLIPS

- ✧ En primer lugar tecleamos **clips** en la línea 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 un mensaje en el que se indica la versión de CLIPS que estemos utilizando, paso 2:.
- ✧ También aparecerá el indicador **CLIPS>**, paso 3:, indicando que CLIPS está esperando a que le demos órdenes. Además, en ese tercer 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 paso 4:.
- ✧ En el quinto paso, 5:, se ejecuta el comando **exit** - recuerde que los comandos se ejecutan siempre entre paréntesis - lo que provoca que salgamos de nuevo al sistema (paso 6:).

6.5 Limpiando la Memoria de Trabajo: **reset** y **clear**

Imagine que carga en la memoria de CLIPS hechos y objetos, y que posteriormente crea reglas, funciones, más hecho y más objetos. Suponga que tras varias ejecuciones desea limpiar la memoria de trabajo. CLIPS proporciona dos comandos para este fin: **reset** y **clear**, que presentan la Sintaxis 6.3

```
(reset)
(clear)
```

Sintaxis 6.3: **reset** y **clear**

Quizás se esté preguntando por qué dos comandos para limpiar la memoria de trabajo. La respuesta es sencilla. En términos generales, quédese con las siguientes ideas:

- ✧ El comando **reset** es conveniente utilizarlo en las siguientes situaciones:
 - ✓ Cuando exista una regla (sin antecedente) que se considera como la primera que debe ejecutarse.
 - ✓ Cuando se desee ejecutar varias veces el mismo programa (con los mismo datos, reglas, etc). Se realizará el **reset** antes de cada ejecución.
 - ✓ Cuando se desee probar distintas estrategias para los mismo o distintos hechos manteniendo la misma base de conocimiento (plantillas y reglas).

- ⇒ El comando `clear` es conveniente utilizarlo cuando se desea cambiar completamente de base de conocimiento.

Si es la primera vez que lee este tutorial pase a la lección siguiente, pero no olvide volver a este apartado cuando estudie el constructor `deffacts`.

6.5.1 ¿Qué hace exactamente `reset`?

Intuitivamente, el efecto del comando `reset` es limpiar la memoria de todos aquellos hechos que hayan introducido en la memoria. Sin embargo, su ejecución produce algo más. En concreto, se realicen las siguientes acciones:

1. Se borran todas las reglas activadas. Es decir, todas aquellas reglas que estaban en la memoria de trabajo dispuestas a ser utilizadas en el siguiente paso de ejecución del programa.
2. Se borran todos los hechos que el usuario haya introducido en la consola o se hayan deducido al aplicar alguna regla.
3. Se lanza el constructor `deftemplate` y crea una nueva plantilla. Es decir, se ejecuta (`deftemplate initial-fact`) (ver apartado 7.2.1 para detalles.)
4. Se lanza el constructor `deffacts` para:
 - ⇒ Crear el hecho `initial-fact` (es decir, se ejecuta (`deffacts initial-fact (initial-fact)`)), y
 - ⇒ Crea todos aquellos hechos que se ha definido como hechos iniciales antes de la ejecución de `reset`.

Vea la sección 7.5.1 para más información sobre el comando `deffacts`.

6.5.2 ¿Qué hace exactamente `clear`?

Intuitivamente, el efecto del comando `clear` es limpiar por completo toda la memoria. En concreto, la 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. Se lanza el constructor `deftemplate` y crea una nueva plantilla. Es decir, se ejecuta (`deftemplate initial-fact`) (ver apartado 7.2.1 para detalles.)

Lectura 7

PLANTILLAS Y HECHOS

7.1 ¿Qué es un Hecho?

Para representar los datos abstractos, CLIPS utiliza hechos, objetos y variables globales (si no lo recuerda repase la Lección 3). En esta lección nos centraremos en los primeros. Un hecho no es más que un dato sobre el dominio que quiere modelarse pero empleando una sintaxis particular (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 (altura 1.90)
         (nombre "Luis Daniel")
         (color-ojos marrones)
         (apellido "Hernández Molinero") )
```

Es decir, el orden de las casillas es irrelevante.

7.2 Plantillas

7.2.1 Construcción: `deftemplate`.

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 se muestra en Sintaxis 7.1.

¹El 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

```

(deftemplate <nombre-deftemplate> [<comentario>]
  <definicion-casillas>*)
donde
<nombre-deftemplate>      ::= Un símbolo
<comentario>              ::= Un string
<definicion-casillas>     ::= <def-casilla-simple> | <def-casilla-multiple>
<def-casilla-simple>      ::= (slot <nombre-casilla> <atributos-plantilla>*)
<def-casilla-multiple>    ::= (multislot <nombre-casilla> <atributos-plantilla>*)
<atributos-plantilla>    ::= <atrib-por-defecto> | <restricciones-atributo>
<atrib-por-defecto>      ::= Sintaxis 8.6
<restricciones-atributo>  ::= Sintaxis 8.1

```

Sintaxis 7.1: deftemplate

A lo largo de esta sección se irán viendo las distintas opciones que presenta este constructor. Por ahora, baste ver como en la Sesión 7.1 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.

7.2.2 Casillas simples y múltiples

En la Sintaxis 7.1, se indica que la definición de una casilla, <definicion-casillas>, puede ser

```
<definicion-casillas> ::= <def-casilla-simple> | <def-casilla-multiple>
```

¿Qué significa esto?.

Considere la definición de la plantilla **persona** de la Sesión 7.1. En esta sesión se ha optado por definir todas las casillas de la plantilla según el término sintáctico <def-casilla-simple>. Es decir, según la expresión:

```
<def-casilla-simple> ::= (slot <nombre-casilla> <atributos-plantilla>*)
```

```

1: CLIPS> (deftemplate persona "Relacion persona"
2:         (slot nombre)
3:         (slot apellido)
4:         (slot color-ojos)
5:         (slot altura) ) ←

```

Sesión 7.1: Definiendo el hecho **persona** con casillas simples

En la Sesión 7.1 no se han especificado los parámetros opcionales <atributos-plantilla>*. Este aspecto lo veremos más adelante. Por ahora notar que cada casilla se define con la palabra clave **slot**. En la práctica, esto significa que cuando se defina un hecho, estas casillas podrán contener sólo **un** valor. Así, podemos definir hechos como los siguientes:

⇒ En las casillas simples se introducen 2 strings.

```
(persona (nombre "Luis Daniel")
```

```
(apellido "Hernández Molinero")
(color-ojos marrones)
(altura 1.90) )
```

- ⇒ En las casillas simples se introducen 2 símbolos.

```
(persona (nombre LuisDaniel)
          (apellido HernándezMolinero)
          (color-ojos marrones)
          (altura 1.90) )
```

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` según la sintaxis

```
<def-casilla-multiple> ::= (multislot <nombre-casilla> <atributos-plantilla>*)
```

En la Sesión 7.2 puede ver cómo se define la relación `persona` con casillas múltiples. Lo que permite definir hechos como los siguientes:

- ⇒ En las casillas múltiples se introducen 2 strings.

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

- ⇒ En las casillas múltiples se introducen 2 símbolos.

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

- ⇒ En las casillas múltiples se introducen símbolos y strings.

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

7.2.3 Visualización: `ppdeftemplate`, `list-deftemplates`.

Además de la construcción de plantillas, CLIPS proporciona otros comandos relacionados con el manejo de plantillas como su visualización en pantalla (o fichero) y la destrucción de plantillas. En particular, los comandos relacionados con la visualización son los siguientes:

```

1: CLIPS> (deftemplate persona "Relacion persona"
2:         (multislot nombre)
3:         (multislot apellido)
4:         (slot color-ojos)
5:         (slot altura) ) ←

```

Sesión 7.2: Definiendo el hecho **persona** con casillas múltiples

- ⇒ Para mostrar las plantillas definidas con un constructor **deftemplate** se utiliza el comando **(ppdeftemplate <nombre-de-la-plantilla>)**

Imprime exactamente toda la información que se introdujo cuando se definió la plantilla **<nombre-de-la-plantilla>**, pero presenta una información adicional: el nombre del módulo en el que se definió.

Un módulo es un conjunto de hechos, reglas, etc que se identifican con un nombre. Si no se especifican módulos, el módulo por defecto recibe el nombre de **MAIN**.

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

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

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.

7.2.4 Destrucción de Plantillas: **undeftemplate**.

Para borrar plantillas definidas previamente por el constructor **deftemplate** se utiliza el destructor **(undeftemplate <nombre-de-la-definicion>)**

Puede utilizar el comodín ***** para eliminar todas las plantillas existentes en la memoria de trabajo. Pero si define una plantilla y posteriormente introduce hechos, o si lo prefiere afirmaciones basadas en esas plantillas, el destructor no tendrá efecto. Es decir, el destructor no podrá destruir una plantilla si existen hechos definidos para esa plantilla que se quiera destruir. Los comandos para introducir y destruir hechos se verán en breve.

La sesión 7.3 muestra la utilización del destructor **undeftemplate** y los comandos de visualización de plantillas. Para conocer el significado de **initial-fact** lea la sección 6.5.

7.2.5 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 (ver apartado 3.1). La sintaxis general de un hecho se muestra en la Sintaxis 7.2.

Intuitivamente, la diferencia entre los hechos no ordenados y los ordenados es que las casillas de los segundos no tienen ningún nombre asociado. Cuando se crea un hecho no ordenado, CLIPS

```

1: CLIPS> (deftemplate Persona "Relacion Persona"
2:           (slot nombre (type SYMBOL)) (slot edad (type INTEGER)) )↵
3: CLIPS> (deftemplate Animal "Relacion Animal"
4:           (slot patas (allowed-integers 0 1 2 4))
5:           (slot tipo (allowed-symbols terrestre acuatico volador)))↵
6: CLIPS> (ppdeftemplate)↵
7: [ARGACCES4] Function ppdeftemplate expected exactly 1 argument(s)
8: CLIPS> (ppdeftemplate *)↵
9: [PRNTUTIL1] Unable to find deftemplate *.
10: CLIPS> (list-deftemplates)↵
11: initial-fact
12: Persona
13: Animal
14: For a total of 3 deftemplates.
15: CLIPS> (list-deftemplates *)↵
16: MAIN:
17:     initial-fact
18:     Persona
19:     Animal
20: For a total of 3 deftemplates.
21: CLIPS> (ppdeftemplate Persona) ↵
22: (deftemplate MAIN::Persona "Relacion Persona"
23:   (slot nombre (type SYMBOL))
24:   (slot edad (type INTEGER)))
25: CLIPS> (undeftemplate Persona)↵
26: CLIPS> (list-deftemplates)↵
27: initial-fact
28: Animal
29: For a total of 2 deftemplates.
30: CLIPS> (ppdeftemplate Animal) ↵
31: (deftemplate MAIN::Animal "Relacion Animal"
32:   (slot patas (allowed-integers 0 1 2 4))
33:   (slot tipo (allowed-symbols terrestre acuatico volador)))
34: CLIPS> (undeftemplate Animal)↵
35: CLIPS> (list-deftemplates)↵
36: initial-fact
37: For a total of 1 deftemplate.

```

Sesión 7.3: Usando los comandos de visualización y destrucción de plantillas

<code><hecho> ::= (<nombre-relación> <casillas-plantilla>* <valores>*)</code>	
donde	
<code><nombre-relación></code>	<code>::=</code> Nombre del hecho o relación. Corresponde a <code><nombre-deftemplate></code> en el caso de hechos ordenados. (Ver Sintaxis 7.1.)
<code><casillas-plantilla></code>	<code>::=</code> (<code><nombre-casilla></code> <code><valores>*</code>)
<code><nombre-casilla></code>	<code>::=</code> corresponde al nombre de una casilla definida en el constructor <code>deftemplate</code> .
<code><valores></code>	<code>::=</code> hace referencia a los datos usuales (enteros, strings, símbolos, etc)

Sintaxis 7.2: Hechos

crea automáticamente una plantilla implícita (en oposición a los hechos ordenados en los que el usuario tiene que definir una plantilla de forma explícita con el constructor `deftemplate`).

Por ejemplo, imagine que no ha definido aún ninguna plantilla, y afirma el siguiente hecho:

```
(persona "Luis Daniel" "Hernández Molinero" marrones 1.90)
```

Como no ha definido ninguna plantilla este hecho es no ordenado, pero su afirmación es equivalente a

1. Definir la plantilla:
`(deftemplate persona (multislot valores))`
2. Para, posteriormente, definir el hecho:
`(persona (valores "Luis Daniel" "Hernández 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 sólo 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))`.

7.3 Creación, modificación, visualización y borrado de hechos

7.3.1 Afirmación de hechos: `assert`.

Una vez definida 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 Sintaxis 7.3.

(assert <hecho>+)

donde

<hecho> ::= Sintaxis 7.2.

Sintaxis 7.3: assert

Cada nuevo hecho que afirme con **assert** se almacena en la memoria de trabajo. En concreto en lo que se conoce como lista de hechos. A cada hecho se le asigna un índice en dicha lista, que presenta la sintaxis <Fact-xxx> siendo xxx el lugar que ocupa en la lista. Así, el primer hecho que se introduzca tendrá la dirección <Fact-0>, el segundo la dirección <Fact-1>, el tercero <Fact-2>, y así sucesivamente.

En la sesión 7.4 puede verse un ejemplo de utilización de este comando. Más concretamente, en esta sesión, se están realizando los siguientes pasos:

- 1: Se define la "Relacion persona", donde todas las casillas son simples
- 2: - 6: Se introduce un hecho concreto. Tras pulsar ⇐, CLIPS devuelve el mensaje <Fact-0>, paso 7:, indicando que al hecho se le asigna el índice 0 en la lista de hechos.
- 8: - 12: Se introduce otro hecho concreto. Tras pulsar ⇐, CLIPS devuelve el mensaje <Fact-1>, paso 13:, indicando que el hecho introducido se le ha asignado el índice 1 en la lista de hechos.
- 14: - 18: En este paso se introduce el mismo hecho que en el paso anterior, a lo que CLIPS responde con el mensaje **FALSE**, paso 19:, indicando que **no** pueden duplicarse hechos.

Note que, en principio, según las Sintaxis 7.3 y 7.2 no se requiere introducir obligatoriamente todos los valores de todos los campos de un hecho ordenado. De hecho, salvo se especifique lo contrario, sólo es necesario introducir el nombre de la relación para afirmar un nuevo hecho (si bien esto suele ser poco útil.)

7.3.2 Visualización de hechos: facts.

Para visualizar los hechos que contiene CLIPS en su lista de hechos se utiliza el comando **facts**, que presenta la Sintaxis 7.4. Las distintas 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 los 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 7.5. ¿Por qué no hay respuesta de CLIPS en los pasos 18: y 19:?

```

1: Repetir la Sesión 7.1.
2: CLIPS> (assert (persona
3:             (nombre "Luis Daniel")
4:             (apellido "Hernández")
5:             (color-ojos marrones)
6:             (altura 189) ) ) ←
7: <Fact-0>
8: CLIPS> (assert (persona
9:             (nombre "Maria Jesús")
10:            (apellido "Rubio")
11:            (color-ojos marrones)
12:            (altura 165) ) ) ←
13: <Fact-1>
14: CLIPS> (assert (persona
15:             (nombre "Maria Jesús")
16:             (apellido "Rubio")
17:             (color-ojos marrones)
18:             (altura 165) ) ) ←
19: FALSE

```

Sesión 7.4: Uso del comando **assert**

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

donde

```

<inicio> ::= números entero positivo (opcional).
<final>  ::= números entero positivo (opcional).
<máximo> ::= números entero positivo (opcional).

```

Sintaxis 7.4: **facts**

1: Repetir 1:- 13: de la sesión 7.4
2: CLIPS> (facts) ←
3: f-0 (Persona (nombre "Luis Daniel") (apellido "Hernandez") (color-ojos marrones)
4: (altura 189))
5: f-1 (Persona (nombre "Maria Jesus") (apellido "Rubio") (color-ojos marrones)
6: (altura 165))
7: For a total of 2 facts.
8: CLIPS> (facts 0 1) ←
9: f-0 (Persona (nombre "Luis Daniel") (apellido "Hernandez") (color-ojos marrones)
10: (altura 189))
11: f-1 (Persona (nombre "Maria Jesus") (apellido "Rubio") (color-ojos marrones)
12: (altura 165))
13: For a total of 2 facts.
14: CLIPS> (facts 1) ←
15: f-1 (Persona (nombre "Maria Jesus") (apellido "Rubio") (color-ojos marrones)
16: (altura 165))
17: For a total of 1 facts.
18: CLIPS> (facts 1 1 0) ←
19: CLIPS> (facts 0 1 0) ←
20: CLIPS> (facts 1 1 1) ←
21: f-1 (Persona (nombre "Maria Jesus") (apellido "Rubio") (color-ojos marrones)
22: (altura 165))
23: For a total of 1 fact.

Sesión 7.5: Uso del comando **facts**

7.3.3 Eliminación de hechos: retract.

Del mismo modo que un hecho puede añadirse a la lista de hechos, un hecho ya almacenado en la lista puede borrarse. Para ello se utiliza el comando **retract**, con Sintaxis 7.5. Notar que <índice> indica el índice del hecho que se le asoció cuando se introdujo en la lista de hechos.

(retract <índice>+)

donde
<índice> ::= número entero no negativo (el índice del hecho a eliminar).

Sintaxis 7.5: retract

La sesión 7.6 muestra un ejemplo en el que se aclara cómo usar el comando **retract**. Observe que tanto si se introduce un hecho ya inexistente (paso 3:) como la no introducción de índices (paso 9:) produce un error en CLIPS (pasos 4: y 10: respectivamente).

```
1: Repetir 1: - 13: de la sesión 7.4
2: CLIPS> (retract 1) ↵
3: CLIPS> (retract 1) ↵
4: [PRNTUTIL1] Unable to find fact f-1.
5: CLIPS> (facts) ↵
6: f-0 (Persona (nombre "Luis Daniel") (apellido "Hernández") (color-ojos marrones)
7: (altura 189))
8: For a total of 1 fact.
9: CLIPS> (retract) ↵
10: [ARGACCES4] Function retract expected at least 1 argument(s)
11: CLIPS> (retract 0) ↵
12: CLIPS> (facts) ↵
13: CLIPS>
```

Sesión 7.6: Uso del comando retract

7.3.4 Modificación de hechos: modify

Los valores de un hecho pueden modificarse mediante el comando **modify** usando la Sintaxis 7.6.

Observe el funcionamiento de este comando en la sesión 7.7. ¿Ha observado la respuesta de CLIPS en el paso 3: cuando realiza la orden del paso 2:? ¿Y la respuesta del paso 12: cuando realiza la orden del paso 11:?. En efecto, el comando **modify** trabaja del siguiente modo:

- ⇒ CLIPS selecciona el hecho indicado por <índice-hecho>.
- ⇒ A continuación se elimina ese hecho original.
- ⇒ Posteriormente se afirma un nuevo hecho, pero con los nuevos valores de las casillas modificadas.

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

donde

<índice-hecho> ::= número entero no negativo (el índice del hecho a modificar)
<casilla-a-modificar> ::= (<nombre-casilla> <valor-de-la-casilla>)
<nombre-casilla> ::= nombre que se le asignó a una casilla al usar el constructor `deftemplate`
<valor-de-la-casilla> ::= el nuevo valor que se le asigna a la casilla.



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

Sintaxis 7.6: `modify` y `duplicate`

```
1: Repetir pasos 1: a 7: de la sesión 7.4.
2: CLIPS> (modify 0 (altura 200))↵
3: <Fact-1>
4: CLIPS> (facts)↵
5: f-1 (persona (nombre "Luis Daniel") (apellido "Hernandez") (color-ojos marrones)
6:   (altura 200))
7: For a total of 1 fact.
8: CLIPS> (modify 0 (color-ojos azules))↵
9: [PRNTUTIL1] Unable to find fact f-0.
10: FALSE
11: CLIPS> (modify 1 (color-ojos azules))↵
12: <Fact-2>
13: CLIPS> (facts)↵
14: f-2 (persona (nombre "Luis Daniel") (apellido "Hernandez") (color-ojos azules)
15:   (altura 200))
16: For a total of 1 fact.
```

Sesión 7.7: Usando el comando `modify`

7.3.5 Duplicación de hechos: `duplicate`.

En ocasiones interesa introducir hechos semajantes cuyos valores se diferencian en pocas casillas—por ejemplo, para no tener que teclear constantemente la misma información. Para estas situaciones CLIPS proporciona el comando `duplicate`. Este comando funciona igual que el comando `modify` pero con la diferencia de que no borra el hecho original. Sigue la Sintaxis 7.6 y puede verse su funcionamiento en la Sesión 7.8.

```
1: Repetir la sesión 7.7.
2: CLIPS> (duplicate 2 (altura 100))↵
3: <Fact-3>
4: CLIPS> (duplicate 2 (nombre "Juan"))↵
5: <Fact-4>
6: CLIPS> (facts)↵
7: f-2 (persona (nombre "Luis Daniel") (apellido "Hernandez") (color-ojos azules)
8:      (altura 200))
9: f-3 (persona (nombre "Luis Daniel") (apellido "Hernandez") (color-ojos azules)
10:      (altura 100))
11: f-4 (persona (nombre "Juan") (apellido "Hernandez") (color-ojos azules)
12:      (altura 200))
13: For a total of 3 fact.
```

Sesión 7.8: Uso del comando `duplicate`

7.4 Comandos de Depuración sobre Hechos: `[un]watch facts`

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 desactivar la impresión de mensajes se utiliza el comando `unwatch facts`.

La sesión 7.9 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.

7.5 Creación, Visualización y Destrucción de Hechos Iniciales

En todo problema existe un conocimiento inicial que debe introducirse en CLIPS. Ya conoce un modo de hacerlo: con `assert`. De usar este comando, debería de introducir tantos *asserts* como hechos iniciales conozca. Sin embargo este comando presenta ciertos *inconvenientes*. Por ejemplo, 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.

Sería deseable que CLIPS tuviera algún comando de tal forma que cuando se limpiara la memoria de trabajo se cargue de forma automática todo el conocimiento inicial del problema sin necesidad

```

1: CLIPS> (deftemplate Animal "Que dice un animal"
2:           (slot animal)
3:           (slot dice)) ←
4: CLIPS> (assert (Animal (animal perro) (dice guau))) ←
5: <Fact-0>
6: CLIPS> (watch facts) ←
7: CLIPS> (assert (Animal (animal gato) (dice miau))) ←
8: ==> f-1      (Animal (animal gato) (dice miau))
9: <Fact-1>
10: CLIPS> (modify 1 (animal gallo) (dice kikiriki))) ←
11: <== f-1      (Animal (animal gato) (dice miau))
12: ==> f-2      (Animal (animal gallo) (dice kikiriki))
13: <Fact-2>
14: CLIPS> (retract 2) ←
15: <== f-2      (Animal (animal gallo) (dice kikiriki))
16: CLIPS> (unwatch facts) ←
17: CLIPS> (retract 0) ←
18: CLIPS>

```

Sesión 7.9: Usando los comandos de visualización y destrucción de plantillas

de volver a introducirlo. No se preocupe, CLIPS dispone de esa instrucción que permite construir ese **conjunto de hechos que forman el conocimiento inicial**, y se llama **deffacts**. Dispone también de comandos para visualizar y destruir esos conjuntos de hechos de forma análoga a las instrucciones **facts** y **retract** para visualizar y destruir hechos individuales.

7.5.1 Creación de hechos iniciales: deffacts

Para la declaración de un conjunto de hechos iniciales se utiliza la Sintaxis 7.7. Sin embargo, la declaración no basta para que CLIPS lo almacene en la memoria. Es decir, utilizar **deffacts** sólo declara los hechos pero **no** serán almacenados en la memoria de trabajo para su inmediata utilización. Ya que **deffacts** se entiende como el conjunto de hechos iniciales que se disponen al comenzar el problema deberemos de decirle a CLIPS: *“empieza desde el principio utilizando el conocimiento (hechos) inicial del problema”*. Esto se consigue con el comando **reset** que tiene la Sintaxis 6.3. Así, el modo usual de utilizar **deffacts** es:

1. Declarar tantos hechos iniciales como deseemos con uno o varios **deffacts**.
2. Ejecutar **reset** para usar los hecho previamente declarados.
3. Ejecutar más comandos de CLIPS.

7.5.2 Visualización de Hechos Iniciales: ppdfeffects, list-deffacts

Ya conoce un comando para visualizar los hechos que se encuentran en la memoria de trabajo: **facts**. Así, si después de un **deffacts** utilizar **reset**, el comando **facts** le permitirá ver los hechos


```

                (deffacts <nombre-de-la-definicion>  [<comentario>]
                                <hecho>* )
donde
<nombre-de-la-definicion>  ::=  Un símbolo
<comentario>                ::=  Un string
<hecho>                      ::=  Sintaxis 7.2

```

Sintaxis 7.7: deffacts

de **deffacts** que se encuentran en la memoria. Sin embargo, en ocasiones no queremos saber qué tenemos en la memoria sino qué es lo que afirmamos como hechos iniciales, independientemente de si están o no en la memoria. CLIPS proporciona dos comandos para esta tarea.

El primero, **ppdeffacts** con la Sintaxis 7.8, permite mostrar exactamente la información que se introdujo al construir un grupo de hechos. La información se muestra del siguiente modo:

```

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

```

donde **MODULO** indica el módulo en el que se ha colocado el constructor **deffacts** correspondiente. Los módulos son un mecanismo que permite *partir* el conocimiento global en *trozos* más pequeños de conocimiento. Por defecto, CLIPS sólo considera un módulo, llamado **MAIN**, salvo que especifique el usuario la existencia de otros módulos.

```

                (ppdeffacts <nombre-de-la-definicion>)
donde
<nombre-de-la-definicion>  ::=  Un símbolo utilizado previamente por un deffacts

```

Sintaxis 7.8: ppdeffacts

El segundo, **list-deffacts** con la Sintaxis 7.9, se utiliza para mostrar todos los nombres de las listas de hechos que se hayan declarado. Observe que el argumento **<nombre-del-modulo>** es opcional:

- ◇ Si no se especifica, se muestran todos los hechos del módulo de conocimiento actual. Si no ha declarado ningún módulo, el no utilizar el argumento es equivalente a teclear (**list-deffacts MAIN**).
- ◇ Si se especifica, y se utiliza el comodín *****, se mostraran todos los hechos definidos en todos los módulos.
- ◇ Si se especifica, se muestran los hechos que se definieron para ese módulo.

7.5.3 Borrado de Hechos Iniciales: undeffacts

Para borrar hechos definidos previamente se utiliza el destructor **undeffacts** que tiene la Sintaxis 7.10. Observe que es obligatorio un argumento. Puede utilizar el comodín ***** para eliminar todas las definiciones existentes.

```
(list-deffacts [<nombre-del-modulo>])
donde
  <nombre-del-modulo> ::= *|MAIN| Alguno definido por el usuario
```

Sintaxis 7.9: list-deffacts

No debe confundir **retract** con **undeffacts**. El primero lo que hace es eliminar un hecho de la memoria de trabajo. El segundo **no** elimina nada de la memoria de trabajo. Sólo dice que en el caso de que reinicie (después de un reset) no se vuelvan a afirmar los hechos que se construyeron con <nombre-de-la-definicion>.

```
(undeffacts <nombre-de-la-definicion>)
donde
  <nombre-de-la-definicion> ::= *|Un símbolo utilizado previamente por un deffacts
```

Sintaxis 7.10: undeffacts

7.5.4 Una sesión con hechos iniciales

La Sesión 7.10 muestra la utilización del constructor **deffacts**, el destructor **undeffacts** y los comandos de visualización de hechos. Lo más destacado es la siguiente.

- ⇒ Se declaran dos conjuntos de hechos iniciales se realizan en los pasos 1: - 2: y 15:. El primero se identifica con el símbolo **EstadoCoche**, el segundo con el símbolo **MiCasa**.
- ⇒ En el paso 8: limpiamos la memoria de trabajo y observamos que el hecho introducido en el paso 3: se ha eliminado pero se han añadido los hechos de **EstadoCoche** (pasos 9: - 14:). Para conocer el significado de **initial-fact** lea la Sección 6.5.
- ⇒ Recuerde que **ppdeffacts** y **undeffacts** necesitan obligatoriamente un argumento. Si no pone ninguno obtendrá mensajes de error (pasos 16: y 41:).
- ⇒ Note la diferencia que hay cuando se utiliza **list-deffacts** sin y con argumentos. Sin argumento muestra los conjuntos de hechos iniciales del módulo actual (paso 32:), con el comodín aparecen clasificados por módulos (paso 36:).

```

1: CLIPS> (deffacts EstadoCoche "El estado de mi coche"
2: (coche motor a-punto) (coche bujias limpias) (coche luces funcionan)) ←
3: CLIPS> (assert (coche intermitente no-funciona))←
4: <Fact-0>
5: CLIPS> (facts)←
6: f-0 (coche intermitente no-funciona)
7: For a total of 1 fact.
8: CLIPS> (reset)←
9: CLIPS> (facts)←
10: f-0 (initial-fact)
11: f-1 (coche motor a-punto)
12: f-2 (coche bujias limpias)
13: f-3 (coche luces funcionan)
14: For a total of 4 facts.
15: CLIPS> (deffacts MiCasa "El estado de mi casa" (casa salon grande))←
16: CLIPS> (ppdeffacts)←
17: [ARGACCES4] Function ppdeffacts expected exactly 1 argument(s)
18: CLIPS> (list-deffacts)←
19: initial-fact
20: EstadoCoche
21: MiCasa
22: For a total of 3 deffacts.
23: CLIPS> (ppdeffacts MiCasa)←
24: (deffacts MAIN::MiCasa "El estado de mi casa"
25: (casa salon grande))
26: CLIPS> (undeffacts EstadoCoche)←
27: CLIPS> (reset)←
28: CLIPS> (facts)←
29: f-0 (initial-fact)
30: f-1 (casa salon grande)
31: For a total of 2 facts.
32: CLIPS> (list-deffacts)←
33: initial-fact
34: MiCasa
35: For a total of 2 deffacts.
36: CLIPS> (list-deffacts *)←
37: MAIN:
38:   initial-fact
39:   MiCasa
40: For a total of 2 deffacts.
41: CLIPS> (undeffacts)←
42: [ARGACCES4] Function undeffacts expected exactly 1 argument(s)
43: CLIPS> (undeffacts *)←
44: CLIPS> (reset)←
45: CLIPS> (facts)←
46: CLIPS>

```

7.6 Ejercicios Propuestos sobre Hechos

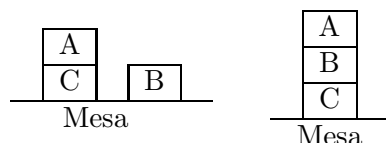
Realice los siguientes ejercicios. Muestre los enunciados de cada uno de ellos en su dirección WEB en una página llamada RCLIPS-1.html. Al final de cada ejercicio ponga dos enlaces. El primero enlazará con la solución del problema. El segundo enlazará con una sesión CLIPS que muestra como se resuelve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

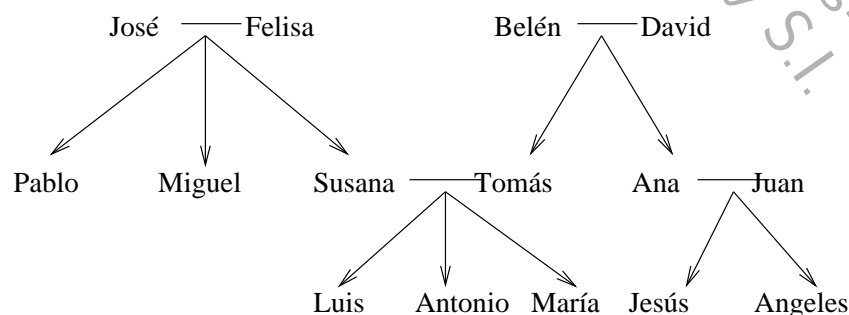
- ◇ Números={2,4,6,1.0,3.0}
- ◇ Enteros={2,4}
- ◇ Reales={1.0,3.0}
- ◇ Lexemas={"rojo", "amarillo", azul, verde}
- ◇ Stings={"rojo"}
- ◇ Simbolos={verde}

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



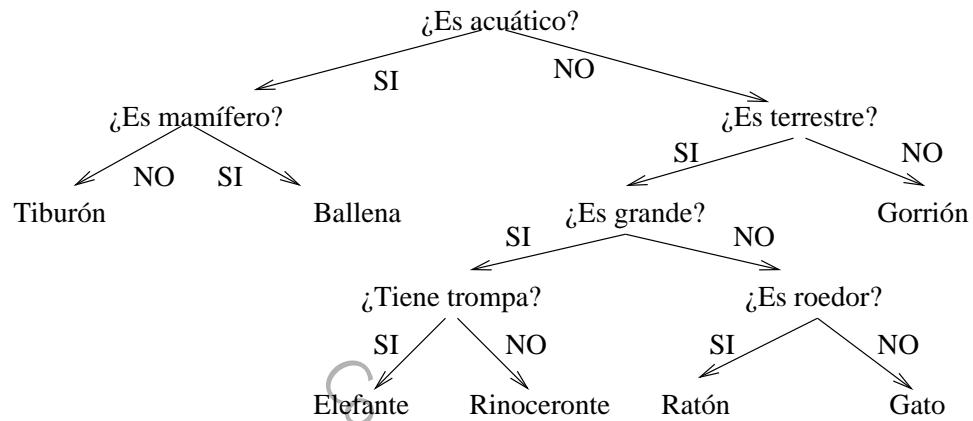
Ejercicio 3 ¿Cómo implementaría un array de objetos usando plantillas?

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



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?.



Lectura 8

RESTRICCIONES DE LOS ATRIBUTOS

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 ejecutándose.

En lección vamos a abordar el término **<atributos-plantilla>** de la Sintaxis 7.1 de **deftemplate**. Este término hace referencia a los atributos que tendrá la casilla, entendiendo como tales a restricciones referentes tanto a los valores por defecto de las casillas, **<atrib-por-defecto>**, como al tipo de datos, rango y cardinalidad que pueden almacenar, **<restricciones-atributo>**.

8.1 Restricciones sobre Tipo, Rango y Cardinalidad de los Datos en una Casilla

En este apartado vamos a comentar con un poco de más profundidad el término **<restricciones-atributo>** de la Sintaxis 7.1. Las restricciones de un atributo se presenta la expresión de la Sintaxis 8.1. Veamos cada una de sus posibles términos del lado derecho.

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

Sintaxis 8.1: Posibles Restricciones de un Atributo (ver Sintaxis 7.1)

8.1.1 Tipos de Atributos

El tipo de atributo especifica el tipo de valor que puede almacenarse en la casilla que se restringe. Lo mejor para entender este tipo de restricción es considerar un ejemplo. Analicemos la Sesión 8.1. En este caso, se establece que las casillas

- ◇ **nombre** y **apellido** sean del tipo **STRING**,
- ◇ **color-ojos** del tipo **SYMBOL**,
- ◇ **altura** del tipo número real, y
- ◇ **edad** del tipo número entero.

```

1: CLIPS> (deftemplate persona "Relacion persona"
2:          (slot nombre (type STRING))
3:          (slot apellido (type STRING))
4:          (slot color-ojos (type SYMBOL))
5:          (slot altura (type FLOAT))
6:          (slot edad (type INTEGER)) ) ↵

```

Sesión 8.1: Una definición de restricciones sobre el tipo de datos en las casillas de la relación "Relacion Persona".

```

<tipo-atributo> ::= (type <tipo>)

donde
<tipo> ::= <tipo-permitido>+ | ?VARIABLE
<tipo-permitido> ::= SYMBOL | STRING | LEXEME | INTEGER |
                  FLOAT | NUMBER | INSTANCE-NAME | INSTANCE-ADDRESS |
                  INSTANCE | EXTERNAL-ADDRESS | FACT-ADDRESS

```

Sintaxis 8.2: Posibles Restricciones sobre el Tipo de Dato de un Atributo

La definición formal de este tipo de restricciones puede verse en la Sintaxis 8.2. Conviene notar que,

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

8.1.2 Tipos de Atributos Constantes Permitidos

Los atributos constantes permitidos establece los posibles *valores constantes* de un tipo específico que pueden almacenarse en una casilla restringida. La sintaxis general puede verse en la Sintaxis 8.3

Notar que dicha sintaxis viene a decir que la lista de valores o bien puede ser una lista de constantes del tipo especificado o bien se puede utilizar la palabra clave **?VARIABLE** para indicar que se admite cualquier constante del tipo permitido. De forma similar a la declaración del tipo de dato admitido en una casilla, en un tipo de dato constante

- ⇒ usar **allowed-lexemes** es equivalente a usar simultáneamente **allowed-symbols** y **allowed-strings**, y

```

<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-istan.>) |
                             (allowed-values <lista-valores>)


donde
<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-istan.> ::= <nombre-instancia>+ | ?VARIABLE
<lista-valores>      ::= <constante>+ | ?VARIABLE

```

Sintaxis 8.3: Posibles Restricciones sobre el Tipo Constante de Dato de un Atributo

- ◇ usar `allowed-numbers` es equivalente a usar simultaneamente `allowed-integers` y `allowed-floats`.

También conviene hacer la siguiente observación


 `allowed-values` restringe los valores de una casilla a los valores especificados `<constante>+`, y 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 símbolos. El segundo establece que sólo son `hola` y `adios` esos valores permitidos.

8.1.3 Rango de los Atributos

Cuando los atributos son numéricos conviene, en ocasiones, establecer sus valores máximo y mínimo, para lo que se utiliza la Sintaxis 8.4.

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`.


```
<rango-atributo> ::= (range <limite> <limite>)
```

donde

```
<limite> ::= <numero> | ?VARIABLE
```

Sintaxis 8.4: Restricciones sobre el Rango de los datos de un Atributo

8.1.4 Cardinalidad de los Atributos

En otras ocasiones puede que no deseemos que el usuario introduzca un número ilimitado de valores en una casilla multicampo o, por contra, que deba introducir una cantidad mínima de valores. Para ello se utiliza la restricción de cardinalidad, que presenta la Sintaxis 8.5.

```
<cardinalidad-atributo> ::= (cardinality <limite> <limite>)
```

donde

```
<limite> ::= <entero> | ?VARIABLE
```

Sintaxis 8.5: Restricciones sobre la Cardinalidad de los datos de un Atributo

- ◇ 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 especifica la cardinalidad de un atributo se supone que se optó 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.

8.2 Restricciones sobre los Valores por Defecto en una Casilla

Otro aspecto que establece restricciones sobre las casillas son los valores por defecto que se tomarán en cada una de ellas, <atrib-por-defecto>. 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, Sintaxis 8.6.

Las posibles opciones son:

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

Sintaxis 8.6: Valores por defecto de una casilla

◇ (default ?DERIVE)

Este es el caso que considera por defecto CLIPS. Con ?DERIVE se considerará un valor por defecto que viene dado por el tipo de dato de la casilla. Por ejemplo, si es del tipo símbolo el valor por defecto será nil. Lea la Sección 8.2.1 para más detalles.

◇ (default ?NONE)

Establece que no se tomará ningún valor por defecto. Será necesario que el usuario establezca un valor para la casilla a la que se le aplica esta restricción. O, en otras palabras, el usuario tendrá que dar obligatoriamente un valor a la casilla que tenga el atributo (default ?NONE). Si no se introducen valores por parte del usuario, en las casillas con esta restricción, CLIPS no aceptará el hecho y dará un mensaje de error.

◇ (default <expresión>*)

El valor por defecto que tomará una casilla será la que se indique en **expresión**.

Así, el campo **<atrib-por-defecto>** del constructor **deftemplate** especifica cuáles serán los valores por defecto que se usarán en las casillas (*slots*) de la plantilla cuando se ejecute un comando **assert**.

8.2.1 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 especificado, entonces el valor del límite inferior del rango se tomará como valor por defecto siempre y cuando no se haya especifico el límite 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 límite 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:

<i>Tipo de valor</i>	<i>Valor por defecto</i>
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 unicampo** (de un valor), entonces el valor por defecto se obtiene a partir de los cuatro pasos anteriores.

El valor por defecto para una **casilla multicampo** (varios valores) es un valor multicampo de longitud 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 compone tomarán un valor atendiendo a las normas de las casillas de un sólo campo.

8.3 Algunas Sesiones de Ejemplo

Algunas sesiones que le ayudarán a entender los aspectos más básicos de esta lección son las siguientes:

- ⇒ En la Sesión 8.2 puede ver el funcionamiento de estas tres opciones. Observe el error que muestra CLIPS en el paso 6:.
- ⇒ La Sesión 8.3 es semejante a la Sesión 8.2 pero en este caso se establecen además restricciones sobre el tipo de dato.
- ⇒ Por último en la Sesión 8.4 se muestra qué ocurre cuando se establece un valor por defecto que no coincide con el tipo especificado.

```

1: CLIPS> (deftemplate Relacion "Ejemplo con valores por defecto"
2:         (slot casilla-1 (default ?NONE))
3:         (slot casilla-2 (default ?DERIVE))
4:         (slot casilla-3 (default MiValor)) ) ←
5: CLIPS> (assert (Relacion)) ←
6: [TMPLTRHS1] Slot casilla-1 requires a value because of its
7: (default ?NONE) attribute.
8: CLIPS> (assert (Relacion (casilla-1 UnValor)))←
9: <Fact-0>
10: CLIPS> (facts)←
11: f-0 (Relacion (casilla-1 UnValor) (casilla-2 nil) (casilla-3 MiValor))
12: For a total of 1 fact.

```

Sesión 8.2: Definición de restricciones sobre valores por defecto

```

1: CLIPS> (deftemplate relacion "Ejemplo con tipos de datos y valores
2: por defecto"
3:         (slot c-1 (type STRING) (default ?NONE))
4:         (slot c-2 (type INTEGER) (default 5))
5:         (slot c-3 (type FLOAT))
6:         (slot c-4 (type SYMBOL) (default UnValor))
7:         (slot c-5 (type SYMBOL)) ) ←
8: CLIPS> (assert (relacion (c-1 "hola")))←
9: <Fact-0>
10: CLIPS> (facts)←
11: f-0 (relacion (c-1 "hola") (c-2 5) (c-3 0.0) (c-4 UnValor) (c-5 nil))
12: For a total of 1 fact.

```

Sesión 8.3: Definición de restricciones sobre tipos y valores por defecto

```

1: CLIPS> (deftemplate Relacion "Esto no funcionará en la vida"
2:         (slot c-1 (type STRING) (default MiValor))) ←
3: [CSTRNCHK1] An expression found in the default attribute does not match the
4: allowed types for slot c-1.
5:
6: ERROR: (deftemplate MAIN::Relacion (slot c-1 (type STRING) (default MiValor))

```

Sesión 8.4: Un error: Las restricciones sobre el tipo y valores por defecto no coinciden

Lectura 9

INTRODUCCIÓN AL MANEJO DE REGLAS

9.1 Definición de Reglas: defrule

Una regla es una expresión del tipo

Si *antecedente*, **entonces** *consecuente*

Su lectura es la siguiente: Si el *antecedente* es cierto para algunos hechos almacenados en la lista de hechos, independientemente de lo que le ocurran a los demás, entonces pueden realizarse las acciones especificadas en el *consecuente*.

En CLIPS estas sentencias presentan la Sintaxis 9.1:

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

Sintaxis 9.1: Definición general de una regla con **defrule**

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 algo similar a lo que se muestra en la Sesión 9.1.

```
1: CLIPS> (defrule Lluvia "Mi primera regla"
2:         (EstaLloviendo si)
3:         (TengoParaguas si)
4:         =>
5:         (MeMojare si))
6: CLIPS>
```

Sesión 9.1: Ejemplo simple de definición de una regla

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 implícitamente 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.

9.2 Destrucción y Visualización de Reglas: **undefrule**, **ppdefrule**, **list-defrules**

Al igual que la construcción de plantillas (**deftemplate**) y de hechos (**defact**) 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** (Sintaxis 9.2)

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

Sintaxis 9.2: Destrucción de reglas: **undefrule**

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

- ⇨ La visualización de reglas puede hacerse con dos comandos:

```
(ppdefrule <nombre-de-la-regla>)  
(list-defrules [<nombre-del-modulo>])
```

Sintaxis 9.3: Visualización de reglas: **ppdefrule**, **list-defrules**

- ✓ Mediante **ppdefrule**, Sintaxis 9.3, se muestra la definición de una regla, mostrando también el módulo al que pertenece. Por defecto el módulo es **MAIN**.
- ✓ Con **list-defrules**, Sintaxis 9.3, se muestran todas las reglas de un módulo. 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.

9.3 Ejecución de Reglas en CLIPS

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 regla 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 prerequisite que la regla esté activada.
- ⇒ Se conoce por **agenda** a la parte del sistema que contiene una lista de las instancias de las reglas activadas. Es decir, 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.

9.3.1 Iniciar la Ejecución de Reglas: run.

1. Para ejecutar las reglas en CLIPS se introduce el comando **run** (Sintaxis 9.4)

```
(run [<máximo>])
```

donde

<máximo> ::= Entero positivo

Sintaxis 9.4: Ejecución de reglas: run

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 9.2 muestra el uso del comando **run**:

- ✧ En primer lugar definimos las cuatro reglas **R1-R4** (líneas 1: - 8:).
- ✧ Tras ejecutar **reset** (línea 9:), sólo tenemos un hecho inicial **initial-fact** (línea 11:)
- ✧ Después de ejecutar las reglas (línea 13:), tenemos en total cuatro hechos (líneas 15: - 18:).
- ✧ Volvemos a limpiar la memoria de trabajo (línea 20:).
- ✧ Ahora ejecutamos las reglas paso a paso (líneas 21: y 26:), mostrando los hechos que se van añadiendo (líneas 22: y 27:).
- ✧ Por último ejecutamos las posibles reglas que queden por dispararse (línea 33:).

Responda a las siguientes preguntas:

1. ¿Quedan reglas por dispararse?
2. Si es así, ¿qué hechos se afirmarían? ¿por qué?.

9.3.2 Parar la Ejecución de Reglas: **halt**.

Para parar la ejecución de reglas en CLIPS se puede utilizar el comando **halt** (Sintaxis 9.5).

Se suele utilizar en el lado derecho de las reglas (consecuentes) para evitar que las reglas se sigan disparando. Además, cuando se invoca a este comando la agenda se deja intacta. Para continuar con la ejecución del programa, se utilizará de nuevo el comando **run**.

En la Sesión 9.3 se muestra un ejemplo. Se definen las mismas reglas que en la Sesión 9.2, pero en el consecuente de la regla **R2** se ha añadido el comando **halt**. Como puede comprobar el consecuente de **R2** se ha ejecutado.


```

1: CLIPS> (defrule R1
2:           => (assert (hazR2))) ←
3: CLIPS> (defrule R2
4:           (hazR2) => (assert (hazR3) (hazR4))) ←
5: CLIPS> (defrule R3
6:           (hazR3) => (assert (hazR2))) ←
7: CLIPS> (defrule R4
8:           (hazR4) => ) ←
9: CLIPS> (reset) ←
10: CLIPS> (facts) ←
11: f-0 (initial-fact)
12: For a total of 1 fact.
13: CLIPS> (run) ←
14: CLIPS>(facts) ←
15: f-0 (initial-fact)
16: f-1 (hazR2)
17: f-2 (hazR3)
18: f-3 (hazR4)
19: For a total of 4 facts.
20: CLIPS> (reset) ←
21: CLIPS> (run 1) ←
22: CLIPS> (facts) ←
23: f-0 (initial-fact)
24: f-1 (hazR2)
25: For a total of 2 facts.
26: CLIPS> (run 1) ←
27: CLIPS> (facts) ←
28: f-0 (initial-fact)
29: f-1 (hazR2)
30: f-2 (hazR3)
31: f-3 (hazR4)
32: For a total of 4 facts.
33: CLIPS> (run) ←

```

Sesión 9.2: Ejecución de Reglas

(halt)

Sintaxis 9.5: Parar la ejecución de reglas: halt

```

1: CLIPS> Realizar los pasos 1: - 2: de la sesión 9.2.
2: CLIPS> (defrule R2
3:         (hazR2) => (halt) (assert (hazR3) (hazR4))) ←
4: CLIPS> Realizar los pasos 5: - 9: de la sesión 9.2.
5: CLIPS> (run) ←
6: CLIPS> (facts) ←
7: f-0      (initial-fact)
8: f-1      (hazR2)
9: f-2      (hazR3)
10: f-3     (hazR4)
11: For a total of 4 facts.

```

Sesión 9.3: Parada de Reglas: **halt**

9.3.3 Puntos de Corte: **set-break**, **remove-break**, **show-break**

Existen otros comandos en CLIPS que permiten parar la ejecución de reglas.

El comando **halt** se suele utilizar en el consecuente de las reglas y, por tanto, se trata de una acción que se realiza **después** de que la regla se haya disparado. Uno podría pensar en utilizar **halt** para depurar programas. Esto es un error, este tipo de depuración conllevaría redefinir constantemente las reglas.

CLIPS proporciona otros comandos para parar las reglas y que están más orientados para la depuración de programas. Los comandos asociados a esta actividad manejan puntos de corte. Un punto de corte no es más que una marca que se establece en una regla y que permite parar la ejecución de un programa **antes** de que la regla se dispare. Los comandos asociados son los siguientes.

- ⇒ Para definir un punto de corte se utiliza el comando **set-break** (Sintaxis 9.6).

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

Sintaxis 9.6: Definición de un punto de corte

Al igual que con el comando **halt**, si un programa CLIPS se para al encontrar un punto de corte, puede continuar con la ejecución del programa utilizando de nuevo el comando **run**.

- ⇒ Si desea borrar un puntos de corte se usa **remove-break** (Sintaxis 9.7). Si se utiliza sin argumento se borrarán todos los puntos de corte.

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

Sintaxis 9.7: Eliminación de un punto de corte

- ⇒ Si quiere visualizar los puntos de corte de un módulo se usa **show-breaks** (Sintaxis 9.8). Si no indica el nombre del módulo se mostrarán los nombres de todas las reglas con puntos de

corte en el módulo actual. Puede utilizar el comodín *, en cuyo caso se mostrarán todos los nombre de todas las reglas que tengan punto de corte de todos los módulos.

```
(show-breaks [<nombre-del-modulo>])
```

Sintaxis 9.8: Visualización de puntos de corte

La Sesión 14.7 muestra un ejemplo de estos comandos. Compare las semejanzas y diferencias entre **halt** (Sesión 9.3) y **set-break** (Sesión 9.4). En ambas sesiones se definen básicamente las mismas reglas, y se establece una parada en la regla R2. Sin embargo, cuando ejecutamos el programa y éste se para, por el **halt** o por el **set-break**, los hechos que tenemos en la memoria de trabajo son totalmente diferentes. Mientras que con **halt** se tienen 4 hechos, con **set-break** se tienen sólo 2. La razón se debe a que en el primer caso el programa se paró **después** de disparar la regla R2, pero en el segundo caso el programa se paró **antes** de disparar la regla R2.

```
1: CLIPS> Realizar los pasos 1:~ 9: de la sesión 9.2.
2: CLIPS> (set-break R2) ←
3: CLIPS> (run) ←
4: CLIPS> (facts) ←
5: f-0      (initial-fact)
6: f-1      (hazR2)
7: For a total of 2 facts.
8: CLIPS> (show-breaks)
9: R2
10: CLIPS> (show-breaks *)
11: MAIN:
12:      R2
13: CLIPS> (remove-break)
14: CLIPS> (show-breaks)
15: CLIPS>
```

Sesión 9.4: Depuración de Reglas: **set-break**, **show-breaks**, **show-breaks**

9.4 Visualización de Reglas Activas: agenda, [un]watch activations.

La visualización de las reglas que se van activando o desactivando conforme se van añadiendo o retractando hechos, puede realizarse mediante dos comandos.

El primer comando, **agenda**, sigue la Sintaxis 9.9 y muestra las reglas activas según el formato:

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

Este formato refleja que:

- ⇨ La regla <Nombre-de-la-regla> se ha activado.

- ◇ Su activación se ha producido porque se verifican los hechos $f\text{-xxx1}, f\text{-xxx2}, \dots, f\text{-xxxn}$. Es decir, se muestran las razones (hechos de la memoria de trabajo) por las que se justifica que la regla se ha activado.

(agenda)

Sintaxis 9.9: Visualización de reglas activas, mostrando su justificación: **agenda**

El otro comando, **watch activations**, muestra las reglas activas en el formato

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

¿Qué significa?

- ◇ La expresión **==> Activation 0**, indica que se ha introducido en la agenda una nueva regla.
- ◇ La regla introducida (activada) es **<Nombre-de-la-regla>**, y
- ◇ la regla se activa porque se verifican los hechos $f\text{-xxx1}, f\text{-xxx2}, \dots, f\text{-xxxn}$.

Cuando se desactiva una regla, estas se muestran con el formato

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

donde

- ◇ La expresión **<== Activation 0** indica que se ha suprimido una regla existente en la agenda.
- ◇ La regla que se suprime es **<Nombre-de-la-regla>**,
- ◇ que fue en su momento activada porque se verificaban los hechos $f\text{-xxx1}, f\text{-xxx2}, \dots, f\text{-xxxn}$.

Caso de que desee desactivar la opción de ir visualizando las reglas que se van activando y desactivando, deberá utilizar el comando **unwatch activations**.

Las Sesiones 9.5 y 9.6 muestran los dos modos para visualizar 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 que el usuario pueda realizar la consulta de la agenda.

9.5 Visualización de Reglas Disparadas: **[un]watch rules**.

Para visualizar cómo las reglas se van disparando se utiliza el comando **watch rules**. Muestra las reglas disparadas en el formato

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

donde:

```

1: CLIPS> Hacer pasos 1-4 de la sesión 9.2
2: CLIPS> (agenda) ←
3: CLIPS> (reset) ←
4: CLIPS> (agenda) ←
5: 0      R1: f-0
6: For a total of 1 activation.
7: CLIPS> (run 1) ←
8: CLIPS> (agenda) ←
9: 0      R2: f-1
10: For a total of 1 activation.
11: CLIPS> (run 1) ←
12: CLIPS> (agenda) ←
13: 0      R4: f-3
14: 0      R3: f-2
15: For a total of 2 activations.
16: CLIPS> (run 1) ←
17: CLIPS> (agenda) ←
18: 0      R3: f-2
19: For a total of 1 activation.
20: CLIPS> (run 1) ←
21: CLIPS> (agenda) ←

```

Sesión 9.5: Visualización *Estática* de Reglas Activas

```

1: CLIPS> Hacer pasos 1-4 de la sesión 9.2
2: CLIPS> (watch activations) ←
3: CLIPS> (reset) ←
4: ==> Activation 0 R1: f-0
5: CLIPS> (run) ←
6: ==> Activation 0      R2: f-1
7: ==> Activation 0      R3: f-2
8: ==> Activation 0      R4: f-3
9: CLIPS> (reset) ←
10: ==> Activation 0      R1: f-0
11: CLIPS> (run 2) ←
12: ==> Activation 0      R2: f-1
13: ==> Activation 0      R3: f-2
14: ==> Activation 0      R4: f-3
15: CLIPS> (reset) ←
16: <== Activation 0      R3: f-2
17: <== Activation 0      R4: f-3
18: ==> Activation 0      R1: f-0

```

Sesión 9.6: Visualización *Dinámica* de Reglas Activas

- ◇ 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.

También puede desactivar la visualización de las reglas que se van disparando introduciendo el comando `unwatch rules`.

La Sesión 9.7 muestra el uso de estos comandos.

```

1: CLIPS> Hacer pasos 1-4 de la sesión 9.2
2: CLIPS> (watch rules) ←
3: CLIPS> (run) ←
4: FIRE      1 R1: f-0
5: FIRE      2 R2: f-1
6: FIRE      3 R4: f-3
7: FIRE      4 R3: f-2
8: CLIPS> (watch activations) ←
9: CLIPS> (run) ←
10: FIRE      1 R1: f-0
11: ==> Activation 0 R2: f-1
12: FIRE      2 R2: f-1
13: ==> Activation 0 R3: f-2
14: ==> Activation 0 R4: f-3
15: FIRE      3 R4: f-3
16: FIRE      4 R3: f-2

```

Sesión 9.7: Visualización Dinámica de Reglas Disparadas

9.6 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 se dispara, la instancia deja de ser útil. Es decir, 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` que sigue la Sintaxis 9.10.

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

Sintaxis 9.10: Activar una regla más de una vez

La Sesión 9.8 muestra cómo afecta el comando `refresh` a la agenda.

```

1: CLIPS> (deffacts ValoresIniciales
2:         (valor 1) (valor 2) (valor 3) (valor 4)) ←
3: CLIPS> (reset) ←
4: CLIPS> (defrule ImprimeValor
5:         (valor ?x)
6:         =>
7:         (printout t "Valor= " ?x crlf)) ←
8: CLIPS> (watch rule) ←
9: CLIPS> (agenda) ←
10: 0   ImprimeValor: f-4
11: 0   ImprimeValor: f-3
12: 0   ImprimeValor: f-2
13: 0   ImprimeValor: f-1
14: For a total of 4 activations.
15: CLIPS> (run 2) ←
16: FIRE   1 ImprimeValor: f-4
17: Valor= 4
18: FIRE   2 ImprimeValor: f-3
19: Valor= 3
20: CLIPS> (agenda) ←
21: 0   ImprimeValor: f-2
22: 0   ImprimeValor: f-1
23: For a total of 2 activations.
24: CLIPS> (refresh ImprimeValor) ←
25: CLIPS> (agenda) ←
26: 0   ImprimeValor: f-3
27: 0   ImprimeValor: f-4
28: 0   ImprimeValor: f-2
29: 0   ImprimeValor: f-1
30: For a total of 4 activations.

```

Sesión 9.8: Usando una regla más de una vez

Lectura 10

ELEMENTOS CONDICIONALES BASADOS EN PATRONES

10.1 Elementos Condicionales

Para que una regla se active es necesario que el antecedente *case* con algunos hechos de la memoria de trabajo. Cuando se produce ese *casamiento*, o coincidencia, se dice que los hechos se ajustan a los elementos condicionales (EC) del antecedente. En otras palabras, una regla se activará cuando se verifiquen todos los EC que compongan el antecedente de la regla. El antecedente de una regla consta de cero o más elementos condicionales.

CLIPS distingue varios tipos de EC: los basados en patrones y los demás (que pueden estar basados en combinación de patrones, en funciones lógicas, etc).



El elemento condicional más simple es un patrón. Un patrón es una restricción que debe verificar sobre los valores de algunos atributos (casillas o campos de la plantilla) de una entidad (hecho o instancia). Cuando una entidad verifica la restricción se dice que se ajusta al patrón.

En esta lección veremos las restricciones dadas en la Sintaxis 10.1, así como el manejo de variables globales y variables que almacenan direcciones de hechos.

```
donde
    <restriccion> ::= <conectivas> | ? | $?
    <conectivas> ::= <termino> |
                    ~<termino> |
                    <termino> & <conectivas> |
                    <termino> | <conectivas>
    <termino> ::= <literal> |
                 <variable-de-casilla-simple> |
                 <variable-de-casilla-multicampo> |
                 <conectivas> |
                 <valor-de-función>
                 <función-predicado>
    <literal> ::= real | entero | símbolo | string |
                 nombre de instancia
    <variable-de-casilla-simple> ::= ?<simbolo-de-la-variable>
    <variable-de-casilla-multicampo> ::= $?<simbolo-de-la-variable>
    <valor-de-función> ::= =<llamada-a-la-función>
    <función-predicado> ::= :<función-predicado>
```

Sintaxis 10.1: Patrones


```

1: CLIPS> (deftemplate Persona
2:         (slot nombre)
3:         (slot apellido)
4:         (slot edad)
5:         (multislot amigos)) ←
6: CLIPS> (deffacts Amigos
7:         (Persona(nombre Luis)
8:                  (apellido Perez)
9:                  (edad 30)
10:                 (amigos Jose Manolo Daniel))
11:         (Persona(nombre Daniel)
12:                  (apellido Rubio)
13:                  (edad 25)
14:                 (amigos Jose Daniel Maria))
15:         (Persona(nombre Jose)
16:                  (apellido Perez)
17:                  (edad 32)
18:                 (amigos Daniel Luis))
19:         (Persona(nombre Manolo)
20:                  (apellido Palma)
21:                  (edad 30)
22:                 (amigos Maria Daniel))
23:         (Persona(nombre Maria)
24:                  (apellido Rubio)
25:                  (edad 30)
26:                 (amigos Daniel Jose Manolo))) ←
27: CLIPS> (watch rules) ←
28: CLIPS> (watch activations) ←

```

Sesión 10.1: Más Hechos

10.2 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** y responden a la expresión `<literal>` de la Sintaxis 10.1.

En la Sesión 10.2, basada en los hechos definidos en la Sesión 10.1, se muestra el uso de este tipo de restricción. En la sesión se define la regla `ImprimePerez` que se activará para todas aquellas entidades (hechos) que tenga exactamente el valor `Perez` en la casilla `apellido` de la relación `Persona`. Si además la regla se dispara, se mostrará un mensaje en el dispositivo de salida estándar. Para conocer con detenimiento el significado de la instrucción `printout` lea la lección 15.

```
1: CLIPS> Realizar la Sesión 10.1.
2: CLIPS> (defrule ImprimePerez
3:         (Persona (apellido Perez))
4:         =>
5:         (printout t "Existe al menos un Perez" crlf)) ←
6: CLIPS> (reset) ←
7: ==> Activation 0      ImprimePerez: f-1
8: ==> Activation 0      ImprimePerez: f-3
9: CLIPS> (run) ←
10: FIRE    1 ImprimePerez: f-3
11: Existe al menos un Perez
12: FIRE    2 ImprimePerez: f-1
13: Existe al menos un Perez
```

Sesión 10.2: Restricciones Literales

10.3 Variables Unicampo

Por tales se entienden a aquellas variables que almacenan un sólo valor. Las variables en casillas uni-campo tiene la sintaxis que se muestra en `<variable-de-casilla-simple>` (Sintaxis 10.1), como por ejemplo:

`?x, ?y, ?unaVariable, ?otraVariable, ?PonLoQueQuieras, etc`



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 (primero se asigna y después se sustituye.)

Estas variables se utilizan para capturar/comprobar el valor de casillas de un valor (de un *slot*), o capturar/comprobar un sólo valor de un casilla con varios valores (de un *multislot*).

La sesión 10.3 muestra el uso de este tipo de restricción: En ésta se define una regla cuyo objetivo es buscar los nombres de las personas que tengan un apellido concreto que viene dado por `Apellido-a-Buscar`, e imprimirlos.

```

1: CLIPS> Realizar la Sesión 10.1.
2: CLIPS> (defrule ImprimeNombre
3:         (Apellido-a-Buscar ?x)
4:         (Persona (nombre ?y) (apellido ?x))
5:         =>
6:         (printout t ?y " -- se apellida -- " ?x crlf)) ←
7: CLIPS> (reset) ←
8: CLIPS> (assert (Apellido-a-Buscar Perez)) ←
9: ==> Activation 0      ImprimeNombre: f-6,f-1
10: ==> Activation 0      ImprimeNombre: f-6,f-3
11: <Fact-6>
12: CLIPS> (run) ←
13: FIRE      1 ImprimeNombre: f-6,f-3
14: Jose -- se apellida -- Perez
15: FIRE      2 ImprimeNombre: f-6,f-1
16: Luis -- se apellida -- Perez

```

Sesión 10.3: Restricciones Variables Uni-Campo

10.4 Comodines

Un comodín representa cualquier valor que se encuentre almacenado en una casilla. Si distinguen dos tipos de comodines (Sintaxis 10.1):

- ◇ 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 10.4 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**. ¿Qué ocurrirá cuando realice el paso 12?

Como ya sabe, CLIPS siempre asigna un valor por defecto a las casillas (ver Lección 8), por lo que la utilización del comodín `?` para casillas uni-campo no suele emplearse demasiado.

La sesión 10.5 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.
- ◇ Observe 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

```

1: CLIPS> Realizar la Sesión 10.1.
2: CLIPS> (reset) ←
3: CLIPS> (defrule Saludo
4:           (Persona (nombre ?))
5:           =>
6:           (printout t "hola" crlf)) ←
7: ==> Activation 0    Saludo: f-1
8: ==> Activation 0    Saludo: f-2
9: ==> Activation 0    Saludo: f-3
10: ==> Activation 0   Saludo: f-4
11: ==> Activation 0   Saludo: f-5
12: CLIPS> (run)

```

Sesión 10.4: Restricciones Comodín Uni-Campo en Casillas Simples

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, aún 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 necesitara 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 10.6 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!.

10.5 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 10.7 muestra el uso de variables multicampo. En dicha sesión se realiza la *fusión* de las dos reglas definidas en la Sesión 10.6 mediante el uso de variables de este tipo con la ventaja añadida de conocer, mediante las variables multicampo, los valores asociados.

10.6 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

```

1: CLIPS> Realizar la Sesión 10.1.
2: CLIPS> (defrule ImprimeAmigos-1
3:         (Buscar-Amigos-de ?x)
4:         (Persona (nombre ?n) (amigos ? ? ?x))
5:         =>
6:         (printout t ?n " es amigo de " ?x crlf)) ←
7:
8: CLIPS> (defrule ImprimeAmigos-2
9:         (Buscar-Amigos-de ?x)
10:        (Persona (nombre ?n) (amigos ? ?x ?))
11:        =>
12:        (printout t ?n " es amigo de " ?x crlf)) ←
13:
14: CLIPS> (defrule ImprimeAmigos-3
15:         (Buscar-Amigos-de ?x)
16:         (Persona (nombre ?n) (amigos ?x ? ?))
17:         =>
18:         (printout t ?n " es amigo de " ?x crlf)) ←
19:
20: CLIPS> (defrule ImprimeAmigos-4
21:         (Buscar-Amigos-de ?x)
22:         (Persona (nombre ?n) (amigos ? ?x))
23:         =>
24:         (printout t ?n " es amigo de " ?x crlf)) ←
25:
26: CLIPS> (defrule ImprimeAmigos-5
27:         (Buscar-Amigos-de ?x)
28:         (Persona (nombre ?n) (amigos ?x ?))
29:         =>
30:         (printout t ?n " es amigo de " ?x crlf)) ←
31: CLIPS> (reset) ←
32: CLIPS> (assert (Buscar-Amigos-de Daniel)) ←
33: ==> Activation 0    ImprimeAmigos-5: f-6,f-3
34: ==> Activation 0    ImprimeAmigos-4: f-6,f-4
35: ==> Activation 0    ImprimeAmigos-3: f-6,f-5
36: ==> Activation 0    ImprimeAmigos-2: f-6,f-2
37: ==> Activation 0    ImprimeAmigos-1: f-6,f-1
38: <Fact-6>

```

Sesión 10.5: Restricciones Comodín Uni-Campo en Casillas Múltiples

```

1: CLIPS> Realizar la sesión 10.1.
2: CLIPS> (defrule ImprimeAmigos-Dch
3:         (Buscar-Amigos-de ?x)
4:         (Persona (nombre ?n) (amigos $? ?x))
5:         =>
6:         (printout t ?n " es amigo de " ?x crlf)) ←
7:
8: CLIPS> (defrule ImprimeAmigos-Izda
9:         (Buscar-Amigos-de ?x)
10:        (Persona (nombre ?n) (amigos ?x $?))
11:        =>
12:        (printout t ?n " es amigo de " ?x crlf)) ←
13:
14: CLIPS> (reset) ←
15: CLIPS> (assert (Buscar-Amigos-de Daniel)) ←
16: ==> Activation 0    ImprimeAmigos-Izda: f-6,f-3
17: ==> Activation 0    ImprimeAmigos-Izda: f-6,f-5
18: ==> Activation 0    ImprimeAmigos-Dch: f-6,f-1
19: ==> Activation 0    ImprimeAmigos-Dch: f-6,f-4
20: <Fact-6>

```

Sesión 10.6: Restricciones Comodín Multi-Campo (en Casillas Múltiples)

```

1: CLIPS> Realizar la sesión 10.1.
2: CLIPS> (defrule ImprimeAmigos
3:         (Buscar-Amigos-de ?x)
4:         (Persona (nombre ?n) (amigos $?antes ?x $?despues))
5:         =>
6:         (printout t ?n " es amigo de " ?antes
7:                  "* " ?x
8:                  " *" ?despues
9:                  "* " crlf)) ←
10: CLIPS> (reset) ←
11: CLIPS> (assert (Buscar-Amigos-de Daniel)) ←
12: ==> Activation 0   ImprimeAmigos: f-6,f-1
13: ==> Activation 0   ImprimeAmigos: f-6,f-2
14: ==> Activation 0   ImprimeAmigos: f-6,f-3
15: ==> Activation 0   ImprimeAmigos: f-6,f-4
16: ==> Activation 0   ImprimeAmigos: f-6,f-5
17: <Fact-6>
18: CLIPS> (run) ←
19: FIRE    1 ImprimeAmigos: f-6,f-5
20: Maria es amigo de *()* Daniel *(Jose Manolo)*
21: FIRE    2 ImprimeAmigos: f-6,f-4
22: Manolo es amigo de *(Maria)* Daniel *()*
23: FIRE    3 ImprimeAmigos: f-6,f-3
24: Jose es amigo de *()* Daniel *(Luis)*
25: FIRE    4 ImprimeAmigos: f-6,f-2
26: Daniel es amigo de *(Jose)* Daniel *(Maria)*
27: FIRE    5 ImprimeAmigos: f-6,f-1
28: Luis es amigo de *(Jose Manolo)* Daniel *()*

```

Sesión 10.7: Restricciones Comodín Multi-Campo (en Casillas Múltiples)

```

1: CLIPS> Realizar la Sesión 10.1.
2: CLIPS> (deftemplate CumpleAgnos
3:           (slot nombre)
4:           (slot edad)) ←
5: CLIPS> (defrule UnAgoMas
6:           ?Hecho1 <- (CumpleAgnos (nombre ?x) (edad ?y))
7:           ?Hecho2 <- (Persona (nombre ?x))
8:           =>
9:           (modify ?Hecho2 (edad ?y))
10:          (retract ?Hecho1)) ←
11: CLIPS> (reset) ←
12: CLIPS> (facts) ←
13: f-0 (initial-fact)
14: f-1 (Persona (nombre Luis) (apellido Perez) (edad 30)
15:      (amigos Jose Manolo Daniel))
16: f-2 (Persona (nombre Daniel) (apellido Rubio) (edad 25)
17:      (amigos Jose Daniel Maria))
18: f-3 (Persona (nombre Jose) (apellido Perez) (edad 32)
19:      (amigos Daniel Luis))
20: f-4 (Persona (nombre Manolo) (apellido Palma) (edad 30)
21:      (amigos Maria Daniel))
22: f-5 (Persona (nombre Maria) (apellido Rubio) (edad 30)
23:      (amigos Daniel Jose Manolo))
24: For a total of 6 facts.
25: CLIPS> (assert (CumpleAgnos (nombre Daniel) (edad 27))) ←
26: ==> Activation 0    UnAgoMas: f-6,f-2
27: <Fact-6>
28: CLIPS> (run) ←
29: FIRE    1 UnAgoMas: f-6,f-2
30: ==> Activation 0    UnAgoMas: f-6,f-7
31: <== Activation 0    UnAgoMas: f-6,f-7
32: CLIPS> (facts) ←
33: f-0 (initial-fact)
34: f-1 (Persona (nombre Luis) (apellido Perez) (edad 30)
35:      (amigos Jose Manolo Daniel))
36: f-3 (Persona (nombre Jose) (apellido Perez) (edad 32)
37:      (amigos Daniel Luis))
38: f-4 (Persona (nombre Manolo) (apellido Palma) (edad 30)
39:      (amigos Maria Daniel))
40: f-5 (Persona (nombre Maria) (apellido Rubio) (edad 30)
41:      (amigos Daniel Jose Manolo))
42: f-7 (Persona (nombre Daniel) (apellido Rubio) (edad 27)
43:      (amigos Jose Daniel Maria))
44: For a total of 6 facts.

```


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 10.2.

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

Sintaxis 10.2: Captura de Direcciones

Conviene notar que un elemento condicional que se ajuste a la Sintaxis 10.2 no es un patrón. El patrón se encuentra en la componente **<Patrón-Elemento-Condicional>**.

La Sesión 10.8 muestra un ejemplo de cómo puede utilizarse la captura de direcciones para modificar y borrar hechos.

10.7 Variables Globales

Quizás se esté preguntando si es posible utilizar las variables fuera del “entorno” de una regla; es decir que sean globales para todas las reglas. La respuesta es afirmativa.

CLIPS distingue dos tipos de variables: las locales y las globales. Su distinción es importante porque la asignación de valores es diferente según el tipo de variable que maneje.

Las **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 ha estudiado en los apartados 10.3 y 10.5.

Las **variables globales** son aquellas que se construyen fueran del entorno de las reglas mediante el constructor **defglobal** según la Sintaxis 10.3. A diferencia de las variables locales, se utiliza la función **bind** para establecer su valor (Sección 10.7.2).

10.7.1 Constructor defglobal y Comandos Relacionados

Si desea utilizar variables globales debe utilizar el constructor **defglobal** según la Sintaxis 10.3. La componente opcional **<nombre-del-modulo>** de la Sintaxis 10.3 indica el módulo en el que el constructor se definirá. 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 la variable será reemplazado por el que se le asigne en el último constructor.

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

Sintaxis 10.3: Constructor de Variables Globales

En la Sesión 10.9 tiene un ejemplo de cómo definir variables globales y utilizarlas para el reconocimiento de patrones.

```

1: CLIPS> (defglobal
2:         ?*x* = 3
3:         ?*y* = ?*x*
4:         ?*z* = (+ ?*x* ?*y*))
5:         ?*q* = (create$ a b c)) % Ver Sección 16.4
6: CLIPS> (defrule ReconocimientoPatronConVarGlobal
7:         (Dato ?*x*)
8:         =>
9:         )
10: CLIPS> (defrule RestriccionPredicadoConVarGlobal
11:         (Dato ?y&:(> ?y ?*x*))
12:         =>
13:         )
14: CLIPS> (defrule ReglaIllegal
15:         (fact ?*x*))
16:         =>)

```

Sesión 10.9: Definición y uso de variables globales en el reconocimiento de patrones

Como puede observar, las variables globales se pueden utilizar de forma totalmente análoga a las variables locales. Sin embargo hay dos excepciones.



Las variables globales

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

Un ejemplo de la segunda excepción es la regla ilegal que se muestra en los pasos 14: - 16: de la Sesión 10.9. No obstante, siempre puede recurrir a una variable local para capturar el valor y, posteriormente, almacenarlo en una variable global.

Otros comandos relacionados con las variables globales son los siguientes:

- ⇒ `(list-defglobals [<modulo>])` muestra los nombres de todas las variables globales del módulo especificado.

Por ejemplo, si define sólo la variable `?*x* = 3`, `(list-defglobals)` mostrará `x`.

- ⇒ `(show-defglobals [<modulo>])` muestra los nombres y los valores que tienen en ese momento almacenados.

Por ejemplo, si define sólo la variable `?*x* = 3`, `(show-defglobals)` mostrará `?*x* = 3`.

- ⇒ `(ppdefglobal <nombre>)` se utiliza para visualizar el contenido del constructor. El `<nombre>` de la variable hace referencia sólo al nombre, es decir sin los símbolos `?` y `*`.

Por ejemplo, si define sólo la variable `?*x* = 3`, para imprimir *en bonito* la variable tendrá que usar la sintaxis `(ppdefglobals x)`.

- ◇ `(undefglobal <nombre>)`, elimina la variable global `<nombre>`. Una variable no podrá eliminarse si está en uso (por ejemplo, por una `deffunction`). Puede utilizar el comodín `*` para borrar todas las variables globales.

El `<nombre>` de la variable hace referencia sólo al nombre, es decir sin los símbolos `?` y `*`.

Recuerde que el comando `reset` no destruye constructores, por lo que este comando sólo iniciará las variables a los valores que estableciera cuando las construyó.

Existen algunos comandos más relacionados con variables globales, consulte los manuales.

10.7.2 Función bind

Hasta ahora ha almacenado valores en variables locales capturando valores de los campos de los hechos o capturando índices, y ha almacenado valores en variables globales mediante `defglobal`. Sin embargo, a menudo es útil poder asociar a una variable un valor que resulta de alguna operación para utilizarlo posteriormente o bien, simplemente, para evitar repetir ciertos cálculos. La función `bind` se utiliza para este propósito. Responde a la Sintaxis 10.4 donde `<variable>` puede ser local o global. Si no se especifica el valor de la `<valor-o-expresión>` se “limpia” la variable. Si se utiliza más de una `<expresión>`, entonces se procede a evaluar todas las expresiones y se agrupan como un valor multicampo. El resultado se almacena en la variable `<variable>`.

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

Sintaxis 10.4: Función bind

Como ejemplo de cómo utilizar la función `bind` con **variables locales** lea la Sección 14.2. Un ejemplo de cómo utilizar la función `bind` con **variables globales** puede verse en la Sesión 10.10.

10.8 Restricciones Conectivas: `~`, `&`, `|`

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 las denominadas **(restricciones) conectivas**, que responden a la expresión `<conectivas>` de la Sintaxis 10.1. Se distinguen:

- ◇ Restricción **no**. Se denota por `~` y se satisface si la restricción que le sigue no se satisface. Responde a la expresión `~<termino>` de la Sintaxis 10.1.
- ◇ Restricción **y**. Se denota por `&` y se satisface si las restricciones adyacentes se satisfacen. Responde a la expresión `<termino> & <conectivas>` de la Sintaxis 10.1.
- ◇ Restricción **o**. Se denota por `|` y se satisface si alguna de las restricciones adyacentes se satisfacen. Responde a la expresión `<termino> | <conectivas>` de la Sintaxis 10.1.

```

1: CLIPS> (defglobal ?*x* = 3.4) ←
2: CLIPS> ?*x* ←
3: 3.4
4: CLIPS> (bind ?*x* (+ 8 9)) ←
5: 17
6: CLIPS> ?*x* ←
7: 17
8: CLIPS> (bind ?*x* (create$ a b c d)) ←
9: (a b c d)
10: CLIPS> ?*x* ←
11: (a b c d)
12: CLIPS> (bind ?*x* d e f) ←
13: (d e f)
14: CLIPS> ?*x* ←
15: (d e f)
16: CLIPS> (bind ?*x*) ←
17: 3.4
18: CLIPS> ?*x* ←
19: 3.4

```

Sesión 10.10: Función bind con variables globales

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 seguida 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 10.11 aclara estos conceptos. La Sesión 10.12 es completamente análoga a la Sesión 10.11 pero ahora las restricciones conectivas contienen una variable. Notar como ahora a la variable `?x` se le asocia el resultado de la restricción conectiva.

10.9 Restricciones de Valores de Retorno: =

Ya hemos visto que un modo de restringir el valor de un campo es mediante un literal. Por ejemplo:

```
(defrule Regla (Persona (edad 30)) =>)
```

producirá que se produzcan tantas instancias como hechos contengan en el termino `edad` el valor 30. Pero ¿podemos establecer restricciones donde el valor sea variable?, es decir ¿que el valor que restringe venga dado por el valor devuelto por una función?. La respuesta es afirmativa.

```

1: CLIPS> (defrule NoPuedoCruzar
2:         (Semaforo ~verde)
3:         =>
4:         (printout t "No Puedo Cruzar" crlf)) ←
5: CLIPS> (defrule PuedoCruzar
6:         (Semaforo ~rojo&~amarillo)
7:         =>
8:         (printout t "Puedo Cruzar" crlf)) ←
9: CLIPS> (defrule MeAtrevoACruzar
10:        (Semaforo amarillo|verde)
11:        =>
12:        (printout t "Me Atrevo a Cruzar" crlf)) ←
13: CLIPS> (assert (Semaforo rojo)) ←
14: <Fact-0>
15: CLIPS> (run) ←
16: No Puedo Cruzar
17: CLIPS> (retract 0) ←
18: CLIPS> (assert (Semaforo amarillo)) ←
19: <Fact-1>
20: CLIPS> (run) ←
21: No Puedo Cruzar
22: Me Atrevo a Cruzar
23: CLIPS> (retract 1) ←
24: CLIPS> (assert (Semaforo verde)) ←
25: <Fact-2>
26: CLIPS> (run) ←
27: Puedo Cruzar
28: Me Atrevo a Cruzar
29: CLIPS> (assert (Semaforo amarillo)) ←
30: <Fact-3>
31: CLIPS> (run) ←
32: No Puedo Cruzar
33: Me Atrevo a Cruzar
34: CLIPS> (assert (Semaforo rojo)) ←
35: <Fact-4>
36: CLIPS> (run) ←
37: No Puedo Cruzar

```

Sesión 10.11: Restricciones Conectivas

```

1: CLIPS> (defrule NoPuedoCruzar
2:         (Semaforo ?x&~verde)
3:         =>
4:         (printout t "(" ?x ") No Puedo Cruzar" crlf)) ←
5: CLIPS> (defrule PuedoCruzar
6:         (Semaforo ?x&~rojo&~amarillo)
7:         =>
8:         (printout t "(" ?x ") Puedo Cruzar" crlf)) ←
9: CLIPS> (defrule MeAtrevoACruzar
10:        (Semaforo ?x&amarillo|verde)
11:        =>
12:        (printout t "(" ?x") Me Atrevo a Cruzar" crlf)) ←
13: CLIPS> Realizar los pasos, a partir del 13:, de la Sesión 10.11.

```

Sesión 10.12: Restricciones Conectivas con Variables

CLIPS permite sustituir literales por los valores retornados en la llamada a funciones. Esta sustitución se realiza con la **restricción de valores de retorno** (=) denotado por la expresión **=<llamada-a-la-función>** en la Sintaxis 10.1.

La restricción de valores de retorno (=) permite al usuario llamar a funciones externas desde un patrón. El valor devuelto por la función debe ser necesariamente uno de los tipos de datos primitivos y se incorpora en el patrón justo en el lugar donde se hizo la llamada a la función **<llamada-a-la-función>** como si fuera una restricción literal.

Imagine que quiere definir una siguiente regla para el siguiente propósito: Imprimir el nombre y edad de una persona cuya edad sea el doble de un número dado. ¿Cómo lo haría?. ¿Y si la edad de esa persona es el doble que el número dado o difiere en dos unidades del número dado?. ¡Hágase!. Una solución puede verla en la Sesión 10.13.

```

1: CLIPS> (defrule Regla1
2:         (dato ?x)
3:         (Persona (edad ?edad&=(* 2 ?x))
4:                  (Nombre ?nombre))
5:         =>
6:         (printout t ?nombre " tiene " ?edad " años" crlf))
7: CLIPS> (defrule Regla2
8:         (dato ?x)
9:         (Persona (edad ?edad&=(* 2 ?x) | =(+ ?x 2) | =(- ?x 2))
10:                  (Nombre ?nombre))
11:         =>
12:         (printout t ?nombre " tiene " ?edad " años" crlf))

```

Sesión 10.13: Restricciones de Valores de Retorno

10.10 Restricción Predicado :

En ocasiones es necesario restringir un campo basado en la veracidad de una **expresión booleana**. Una restricción de este tipo recibe el nombre de **restricción predicado** que se denota por : y que responde a la expresión :<función-predicado> en la Sintaxis 10.1.

Una función predicado es una función que devuelve el símbolo FALSE cuando no se satisface y devuelve un valor no-FALSE cuando se satisface. Si la función predicado devuelve no-FALSE entonces la restricción predicado se satisface, pero si devuelve el símbolo FALSE la restricción predicado no se satisface.

El modo usual de usar esta restricción es utilizando restricciones conectivas con variables. La expresión más simple de este tipo es como sigue:



?x&:<restricción-con-?x>

debe leerse como:

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

Pero pasemos a la práctica. Imagine que desea establecer una regla que haga una cierta acción cuando cierto hecho ordenado **valor** contenga un cantidad positiva. Un modo de definirlo es el siguiente:

```
1: CLIPS> (defrule Regla
2:         (valor ?x)
3:         (test (> ?x 0)) % Vea apartado 11.2
4:         =>
5:         (acciones))
```

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:

```
1: CLIPS> (defrule Regla
2:         (valor ?x&:(> ?x 0))
3:         =>
4:         (acciones))
```

Otro ejemplo es el siguiente: Realizar una regla que se dispare si el valor de un cierto campo de un cierto hecho es un número. Una solución puede ser:

```
1: CLIPS> (defrule Regla2
2:         (Hecho (valor ?x&:(integerp ?x)|:(floatp ?x) ) )
3:         % Vea apartado 16.3.2
4:         =>
5:         (acciones))
```

Lectura 11

OTROS ELEMENTOS CONDICIONALES

La sintaxis general de un elemento condicional puede verse en la Sintaxis 11.1

```
<Elemento-Condicional> ::= <restriccion> |  
                           <Captura-Dirección> |  
                           <Conectivas-entre-EC> |  
                           <EC-test> |  
                           <EC-Existe> |  
                           <EC-Para-Cada> |  
                           <EC-Soporte-Lógico>
```

Sintaxis 11.1: Sintaxis General de un Elemento Condicional

En la Lección 10 nos hemos centrado en las dos primeras opciones de la Sintaxis 11.1:

- ◇ La expresión **<restriccion>** es la que responde a la Sintaxis 10.1.
- ◇ La expresión **<Captura-Dirección>** es la de la Sintaxis 10.2.

Pero como puede comprobar existen más expresiones, no basadas en patrones, y que son las que se mostrarán en esta lección.



El comando **reset** debe utilizarse antes de utilizar la mayoría de los EC que se verán en esta lección.

Para más información sobre estas condiciones lea el manual de CLIPS.

11.1 Conectivas entre Elementos Condicionales

Las restricciones conectivas establecen restricciones del tipo **no**, **y** y **o** en los valores de una casilla, es decir, establecen patrones (ver Sección 10.8 para detalles.)

Esta idea puede extenderse para establecer conectividad entre los distintos elementos condicionales que componen el antecedente de una regla.

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

```
1: (defrule LLuvia "Mi primera regla"  
2:   (EstaLloviendo si)  
3:   (TengoParaguas no)  
4:   =>  
5:   (MeMojare si))
```


establece **implícitamente** 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 se muestran en la Sintaxis 11.2.

	<code><Conectivas-entre-EC> ::= <EC-or> <EC-and> <EC-not></code>
donde	
	<code><EC-or> ::= (or <elementos-condicionales>+)</code>
	<code><EC-and> ::= (and <elementos-condicionales>+)</code>
	<code><EC-not> ::= (not <elemento-condicional>)</code>

Sintaxis 11.2: Elementos Condicionales **or**, **and** y **not**

Los siguientes ejemplos son autoexplicativos:

- ⇨ Ejemplo del EC **or**: Me voy de excursión ya haga buen tiempo **o** mal tiempo.

```
1: (defrule Excursion
2:   (or (tiempo bueno) (tiempo malo))
3:   =>
4:   (printout t "Me voy de excursión" crlf)
```

- ⇨ Elemento condicional **and**: Podré conducir siempre y cuando tenga un coche **y** las llaves.

```
1: (defrule Conducir
2:   (and (tengo coche) (tengo llaves))
3:   =>
4:   (printout t "Puedo conducir" crlf)
```

- ⇨ Elemento condicional **not**. Tendré clase si el centro **no** está cerrado.

```
1: (defrule VoyAClase
2:   (not (centro cerrado))
3:   =>
4:   (printout t "Tengo clase" crlf)
```

11.2 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 se muestra en la Sintaxis 11.3.

```
<EC-test> ::= (test <expresión>)
```

Sintaxis 11.3: Elemento Condicional `test`

```
1: CLISP> Realizar la Sesión 10.1
2: CLIPS> (reset) ←
3: CLIPS> (defrule NoTanJoven
4:           (Persona (nombre ?nombre) (edad ?edad))
5:           (test (> ?edad 25))
6:           =>
7:           (printout t ?nombre " ya no es tan joven. Tiene "
8:             ?edad "años" crlf)) ←
9: ==> Activation 0    NoTanJoven: f-1
10: ==> Activation 0    NoTanJoven: f-3
11: ==> Activation 0    NoTanJoven: f-4
12: ==> Activation 0    NoTanJoven: f-5
```

Sesión 11.1: Un Ejemplo con el Elemento Condicional `test`

La Sesión 11.1 muestra cómo puede utilizarse.

Este elemento condicional también puede utilizarse con elementos condicionales conectivos. Por ejemplo, en la Sesión 11.2 se muestran dos ejemplos de elementos condicionales que sirven para determinar si el valor de una variable es entera y se encuentra en el rango 1-10.



Se aconseja no usar `test` como el primer elemento condicional de una regla.

Para más información sobre estas condiciones lea el manual de CLIPS.

11.3 Elemento Condicional `exists`

En ocasiones interesa saber si existe al menos una entidad que verifique un cierto patron sin importarnos demasiado que entidad es la que lo verifica.

Este tipo de situaciones también pueden establecerse en CLIPS mediante el EC `exists` que sigue la Sintaxis 11.4.

Por ejemplo, un profesor puede establecer que está dispuesto a impartir clase con que haya un alumno que asista al aula, sin importar de que alumno se trate. ¿Cómo lo haría? ¡Hágase!.

11.4 Elemento Condicional `forall`

En ocasiones interesa realizar acciones para cada ocurrencia de un grupo de elementos condicionales. Este tipo de situaciones se reflejan en CLIPS mediante el EC `forall` que sigue la Sintaxis 11.5.

Por ejemplo, un profesor puede querer detectar si hay algún alumno que haya superado el examen de teoría y de prácticas. Una solución la tiene en la Sesión 11.3. Complete una sesión en la que se muestre cómo puede utilizarse la regla definida en la Sesión 11.3.

```

1: CLISP> (defrule Test1
2:           (test (and (integerp ?valor)
3:                       (>= ?valor 1) % Vea Sección 16.
4:                       (<= valor 10)))
5:           =>
6:           (acciones))
7:
8: CLIPS> (defrule Test2
9:           (test not (or (not (integerp ?valor)
10:                          (< ?valor 1)
11:                          (> valor 10)))
12:           =>
13:           (acciones))
14:

```

Sesión 11.2: Otro Ejemplo con el Elemento Condicional test

<EC-existe> ::= (exists <elemento-condicional>+)

Sintaxis 11.4: Elemento Condicional exists

<EC-Para-Cada> ::= (forall <elemento-condicional> <elemento-condicional>+)

Sintaxis 11.5: Elemento Condicional forall

```

1: CLISP> (defrule Supera
2:           (forall ((Alumno ?x) (Teoria ?x) (Practica ?x)))
3:           =>
4:           (printout t "Supera" crlf))

```

Sesión 11.3: Un Ejemplo con el Elemento Condicional forall

11.5 Elemento Condicional logical

Note que todos los ejemplos que se han visto en las secciones previas las entidades tienen **soporte lógico incondicional**. Es decir, las entidades las ha afirmado via comandos (p.e. con **assert**), constructores (p.e. con **deffacts**) o a partir de reglas (e.d. como acciones de un **defrule**). En definitiva, ha creado entidades porque según su criterio existían condiciones lógicas para que se afirmaran. Pero cuando suprime ese condiciones lógico lo único que consigue es simplemente eso: elimina el soporte lógico pero no las entidades que afirmó. Por ejemplo, si elimina una regla que da soporte a una entidad que se afirmó en un momento de su programa CLIPS, sólo elimina la regla pero no la entidad (aún cuando la entidad se apoyaba en esa regla para su existencia).

CLIPS proporciona mecanismos de **mantenimiento de la verdad** para entidades (hechos o instancias) creadas por reglas. Es decir, una entidad creada en el consecuente de una regla se puede hacer lógicamente dependiente de ciertas entidades que se encuentren en el antecedente de dicha regla. Es decir, la entidad del consecuente existe (tiene soporte lógico) mientras exista las entidades del antecedente. Cuando algunas de las entidades que dan soporte lógico (del antecedente) se elimina, entonces la entidad que se creó (del consecuente) también se elimina, ya que su existencia estaba soportada por la entidad que se eliminó primero. Estos elementos condicionales que dan soporte lógico a la existencia de elementos en el consecuente de una regla siguen la Sintaxis 11.6. Implícitamente todos los EC logical están unidos por el elemento condicional **and**, pero puede usarlos también con **or** y **not**.

<EC-Soporte-Lógico> ::= (logical <elemento-condicional>+)

Sintaxis 11.6: Elemento Condicional logical



Sólo los N-primeros patrones de las reglas pueden tener el EC logical.

Por ejemplo, las siguientes reglas obtenidas del manual de CLIPS son ilegales:

```
1: (defrule not-ok-1      (defrule not-ok-2      (defrule not-ok-3
2:   (logical (a))        (a)                    (or (a)
3:   (b)                  (logical (b))          (logical (b)))
4:   (logical (c))        (logical (c))          (logical (c))
5:   =>                   =>                     =>
6:   (assert (d)))        (assert (d)))          (assert (d)))
```

Un ejemplo que le puede ayudar a entender la situación es la siguiente. Se supone que usted puede llegar a tener una vivienda mientras tenga un puesto de trabajo y tenga dinero suficiente para pagar la hipoteca. Teniendo en cuenta los tiempos que andan, en el momento que deje de trabajar o deje de tener dinero para la hipoteca, dejaría de tener su vivienda. Tanto el trabajo como el dinero, le dan el soporte lógico para poder afirmar de que puede tener una vivienda. Pero en cuanto uno de esos soportes deje de existir, su vivienda desaparecería. Una sesión que muestra esta situación es la Sesión 11.4. Observe que el hecho (**Tengo Trabajo**) se elimina porque se dispara la regla **EstarDespedido**, pero el hecho (**Tengo Vivienda**) se elimina porque se suprime uno de sus soportes lógicos, (**Tengo Trabajo**).

```

1: CLIPS> (deffacts CondicionesIniciales "Condiciones Iniciales"
2:         (Tengo Trabajo)
3:         (Tengo Dinero))
4: CLIPS> (defrule TenerVivienda "Establece la posibilidad de tener Vivienda"
5:         (logical (Tengo Trabajo))
6:         (logical (Tengo Dinero))
7:         =>
8:         (assert (Tengo Vivienda)))
9: CLIPS> (defrule EstarDespedido "Condiciones para perder el Trabajo"
10:        (Estar Despedido)
11:        ?x <- (Tengo Trabajo)
12:        =>
13:        (retract ?x))
14: CLIPS> (reset)
15: CLIPS> (watch facts)
16: CLIPS> (agenda)
17: 0      TenerVivienda: f-1,f-2
18: For a total of 1 activation.
19: CLIPS> (run)
20: ==> f-3      (Tengo Vivienda)
21: CLIPS> (facts)
22: f-0      (initial-fact)
23: f-1      (Tengo Trabajo)
24: f-2      (Tengo Dinero)
25: f-3      (Tengo Vivienda)
26: For a total of 4 facts.
27: CLIPS> (assert (Estar Despedido))
28: ==> f-4      (Estar Despedido)
29: <Fact-4>
30: CLIPS> (run)
31: <== f-1      (Tengo Trabajo)
32: <== f-3      (Tengo Vivienda)
33: CLIPS> (facts)
34: f-0      (initial-fact)
35: f-2      (Tengo Dinero)
36: f-4      (Estar Despedido)
37: For a total of 3 facts.

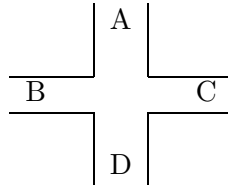
```

Sesión 11.4: Un Ejemplo con el Elemento Condicional logical

11.6 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 enlazarás 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, simultáneamente, 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, tío, 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 7.6. 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 7.6.

El apartado 15.4.3 puede serle útil.

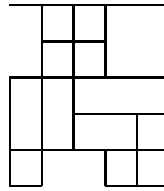
Ejercicio 4 Implemente en CLIPS el tipo de dato abstracto pila con los operadores:

- ◇ **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.

Copia para la impartición de la asignatura
Idaniel en um.es
Ingeniería del conocimiento y S.I.

Lectura 12

ESTRATEGIAS 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 añade 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 reordenarlas.
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.

12.1 Prioridad de una Regla: *salience*

La prioridad (en inglés, *salience*) de una regla no es más 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 en el término `<propiedades>` de la Sintaxis 9.1, entre el nombre de la regla y el primer elemento condicional de la regla, utilizando la expresión

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

con lo que la Sintaxis 9.1 queda modificada según la Sintaxis 12.1.

```
(defrule <nombre-de-la-regla> [<comentario-string>]
  [(declare (salience <expresión-numérica>))]
  <antecedente>*
  =>
  <acciones>* )
```

Sintaxis 12.1: Definición general de una regla con *salience*

En la Sesión 12.1 se definen dos reglas que incluye prioridades: en la regla R1 se establece un valor fijo de prioridad con valor 100, en la regla R2 se establece que el valor de prioridad viene dado por el resultado de sumar 10 al valor de la variable global `?*variable*` (ver apartado 10.7).

Se propone que defina 3 reglas (R1, R2 y R3) según la sintaxis


```

1: CLIPS> (defrule R1
2:           (declare (salience 100))
3:           (Elemento-Condicional-1)
4:           =>
5:           (printout t "R1 disparada" crlf))
6: CLIPS> (defrule R2
7:           (declare (salience (+ ?*variable* 10)))
8:           (Elemento-Condicional-2)
9:           =>
10:          (printout t "R2 disparada" crlf))

```

Sesión 12.1: Ejemplos sencillos de definición de reglas con preferencias

```

1: (defrule R<i>
2:   (declare (salience (* <i> 10)))
3:   =>
4:   (printout t "R<i>" crlf))

```

donde <i> tome los valores 1, 2 y 3. Ejecute el programa. ¿Cuál es el orden de ejecución de las reglas?.

12.1.1 Funciones relacionadas con salience

Otras funciones relacionadas con la prioridad de las reglas son las siguientes.

(set-salience-evaluation <valor>)

Establece cuándo debe valorarse la prioridad de la regla. El parámetro <valor> es alguno de los siguientes valores: **when-defined**, **when-activated**, o **every-cycle**. Es decir, los valores de prioridad pueden evaluarse en alguno de los siguientes momentos:

- ✧ **when-defined**: Cuando se defina la regla. La prioridad se establece al realizar el **defrule** de la regla. Esta es la opción por defecto de CLIPS.
- ✧ **when-activated**: Cuando la regla se active. La prioridad se establece al realizar el **defrule** de la regla y posteriormente cuando la regla se activa.
- ✧ **every-cycle**: En cada ciclo de ejecución (ver apartado 9.3). La prioridad se establece al realizar el **defrule** de la regla, posteriormente cuando la regla se activa, y después de cada disparo.

(get-salience-evaluation)

Esta función devuelve el actual comportamiento de CLIPS sobre la evaluación de prioridades. Los valores que puede devolver son: **when-defined**, **when-activated**, o **every-cycle**.

Esta función fuerza a la revaluación 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.

12.2 El uso de salience puede ser problemático

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. DESENGAÑASE. Si opta por abusar de esta potencia debe de ser consciente de que hace un manejo explícito del control:

1. Está convirtiendo esta técnica de programación en una programación imperativa (o procedural) ya que estará indicando la secuencia de ejecución. Entonces, ¿para qué utilizar CLIPS?. Conformes 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.

Como ejemplo veamos cómo puede suprimirse por completo el **salience** en algunas situaciones. Suponga que desea jugar al tic-tac-toe (gato o tres en raya) 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 la casilla está libre, entonces poner una ficha en la casilla.

Esta estrategia está implementada en la Sesión 12.2. Notar lo siguiente:

- ⇒ Si está disponible más de un tipo de casilla, se activarán las 3 reglas y se dispararán en el orden R1, R2 y R3.

```

1: CLIPS> (defrule Casilla-Ganadora ; R1
2:         (declare (salience 100))
3:         ?mov <- (muevoYo) ; Le toca al ordenador
4:         (casilla ganadora)
5:         =>
6:         (retract ?mov)
7:         (assert (mover-A ganadora)) )
8: CLIPS> (defrule Casilla-Bloqueo ; R2
9:         (declare (salience 50))
10:        ?mov <- (muevoYo) ; Le toca al ordenador
11:        (casilla bloquea)
12:        =>
13:        (retract ?mov)
14:        (assert (mover-A bloquea)) )
15: CLIPS> (defrule Casilla-Ganadora ; R3
16:        ?mov <- (muevoYo) ; Le toca al ordenador
17:        (casilla ?x&esquina|medio|lateral)
18:        =>
19:        (retract ?mov)
20:        (assert (mover-A ?x)) )

```

Sesión 12.2: Implementación *imperativa* para el tres en raya

- ◇ 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 este caso, **saliente expresa una relación implícita y condicionada entre las reglas**.



En general, deberá de eliminar por completo, en la medida de lo posible, la interacción entre las reglas con objeto de que actúen oportunísticamente (que es lo correcto) y no condicionalmente (que es lo incorrecto).

Bueno, bueno. Todo esto está muy bien pero ... ¿eso cómo lo hago?. La respuesta se obtiene teniendo en cuenta otro concepto básico en la programación basada en reglas:



Una regla debe de representar lo más **completamente** posible una heurística. De este modo pueden eliminarse las relaciones condicionadas y/o implícitas anteriores añadiendo adecuadamente más patrones (conocimiento) en las reglas.

Así, aplicando este modo de proceder, las mismas reglas de la Sesión 12.2 pueden expresarse como se muestra en la Sesión 12.3.

```
1: CLIPS> (defrule Casilla-Ganadora ; R1
2:      ?mov <- (muevoYo) ; Le toca al ordenador
3:      (casilla ganadora)
4:      =>
5:      (retract ?mov)
6:      (assert (mover-A ganadora)) )
7: CLIPS> (defrule Casilla-Bloqueo ; R2
8:      ?mov <- (muevoYo) ; Le toca al ordenador
9:      (casilla bloquea)
10:     (not (casilla ganadora))
11:     =>
12:     (retract ?mov)
13:     (assert (mover-A bloquea)) )
14: CLIPS> (defrule Casilla-Ganadora ; R3
15:      ?mov <- (muevoYo) ; Le toca al ordenador
16:      (casilla ?x&esquina|medio|lateral)
17:      (not (casilla ganadora))
18:      (not (casilla bloquea))
19:      =>
20:      (retract ?mov)
21:      (assert (mover-A ?x)) )
```

Sesión 12.3: Implementación *declarativa* para el tres en raya

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 bloquee.
3. Si la casilla está libre, no es ganadora y no bloquea al contrario, entonces poner una ficha en la casilla.

12.3 Estrategias Implementadas en CLIPS: set-strategy y get-strategy

Suponemos que con el ejemplo del tres en raya ya ha aprendido de que salience no debe utilizarse. Claro está que si no establecer ningún **salience** en las reglas, entonces CLIPS las dispara de manera automática pero ... ¿cómo lo hace exactamente? ¿puede cambiarse?. El modo exacto en que lo hace es mediante una estrategia. Una estrategia no es más que un criterio que permite establecer el orden en que se ejecutarán las reglas. CLIPS proporciona varias estrategias “de fábrica”: en profundidad, en anchura, aleatoria, basada en simplicidad, basada en complejidad, la lex y la mea.

```
(set-strategy <estrategia>)

donde

    <estrategia> ::= depth | breadth | random | simplicity |
                  complexity | lex | mea
```

Sintaxis 12.2: Sintaxis de set-strategy

El comando **set-strategy** de la Sintaxis 12.2 le permite cambiar de estrategia y **get-strategy** (Sintaxis 12.3) para conocer la que está utilizando en un momento dado. Elija la que elija todas siguen la siguiente idea:

- ✧ Se considera una pila en la que se van colocando las instancias de las reglas.
- ✧ En cada ejecución, las que tiene la misma preferencia se colocan en una parte de la pila que dependerá de la estrategia seleccionada.
- ✧ La regla que quede en la parte alta de la pila será la siguiente en ser disparada.

```
(get-strategy)
```

Sintaxis 12.3: Sintaxis de get-strategy

Veamos con algo más de detenimiento en que consiste cada estrategia con un ejemplo. Para ello, tomaremos como referencia el conjunto de reglas que se muestra en la Sesión 12.4 que responde al árbol de búsqueda de la Figura 2.

```

1: CLIPS> (defrule R0      => (assert (Hecho 1)))
2: CLIPS> (defrule R1 (Hecho 1) => (assert (Hecho 2) (Hecho 3)))
3: CLIPS> (defrule R2 (Hecho 2) => (assert (Hecho 4) (Hecho 5)))
4: CLIPS> (defrule R3 (Hecho 3) => (assert (Hecho 6) (Hecho 7)))
5: CLIPS> (defrule R4 (Hecho 4) => )
6: CLIPS> (defrule R5 (Hecho 5) => )
7: CLIPS> (defrule R6 (Hecho 6) => (assert (Hecho 8) (Hecho 9)))
8: CLIPS> (defrule R7 (Hecho 7) => )
9: CLIPS> (defrule R8 (Hecho 8) => )
10: CLIPS> (defrule R9 (Hecho 9) => )
11: CLIPS> (reset)
12: CLIPS> (watch rules)

```

Sesión 12.4: Definición del árbol de búsqueda de la Figura 2

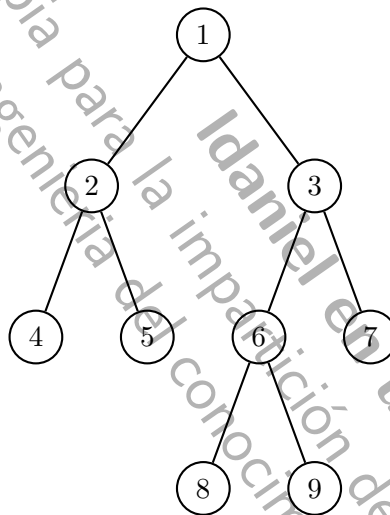


Figura 2: Árbol de búsqueda para la Sesión 12.4

12.3.1 Estrategia en Profundidad: depth

Con esta estrategia, las nuevas reglas con la misma preferencia se colocarán en el cima de la pila. Imagine que tiene dos reglas: R1 que se activa para f-1, y R2 se activa para f-2. Caso de que f-1 se afirmara antes que f-2, entonces R2 se colocará por encima de R1.

En la Sesión 12.5 tiene el resultado de la ejecución de las reglas de la Sesión 12.4 cuando se especifica la estrategia en profundidad.

12.3.2 Estrategia en Anchura: breadth

En este caso, las nuevas 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, entonces R1 se mantendrá por encima de R2.

```

1: CLIPS> Cargar la sesión 12.4.
2: CLIPS> (set-strategy depth) ↔
3: depth
4: CLIPS> (run) ↔
5: FIRE      1 R0: f-0
6: FIRE      2 R1: f-1
7: FIRE      3 R3: f-3
8: FIRE      4 R7: f-5
9: FIRE      5 R6: f-4
10: FIRE     6 R9: f-7
11: FIRE     7 R8: f-6
12: FIRE     8 R2: f-2
13: FIRE     9 R5: f-9
14: FIRE    10 R4: f-8

```

Sesión 12.5: Estrategia en Profundidad

La Sesión 12.6 es semejante a la Sesión 12.5 pero ahora estableciendo la estrategia basada en anchura.

```

1: CLIPS> Cargar la sesión 12.4.
2: CLIPS> (set-strategy breadth) ↔
3: breadth
4: CLIPS> (run) ↔
5: FIRE      1 R0: f-0
6: FIRE      2 R1: f-1
7: FIRE      3 R2: f-2
8: FIRE      4 R3: f-3
9: FIRE      5 R4: f-4
10: FIRE     6 R5: f-5
11: FIRE     7 R6: f-6
12: FIRE     8 R7: f-7
13: FIRE     9 R8: f-8
14: FIRE    10 R9: f-9

```

Sesión 12.6: Estrategia en Anchura

12.3.3 Estrategia Aleatoria: random

A cada activación de una regla se asigna a un número aleatorio que es el que se utilizará para determinar su lugar en la pila entre todas las activaciones de igual preferencia (con el mismo salience). Este número aleatorio se guarda incluso cuando se cambia de estrategia para que se produzca el mismo orden cuando la estrategia aleatoria se vuelva a seleccionar.

La sesión 12.7 es el resultado de aplicar las reglas de la sesión 12.4 cuando se establece la estrategia aleatoria.

```
1: CLIPS> Cargar la sesión 12.4.
2: CLIPS> (set-strategy random) ↵
3: random
4: CLIPS> (run) ↵
5: FIRE    1 R0: f-0
6: FIRE    2 R1: f-1
7: FIRE    3 R3: f-3
8: FIRE    4 R6: f-4
9: FIRE    5 R9: f-7
10: FIRE   6 R8: f-6
11: FIRE    7 R2: f-2
12: FIRE    8 R4: f-8
13: FIRE    9 R5: f-9
14: FIRE   10 R7: f-5
```

Sesión 12.7: Estrategia Aleatoria

12.3.4 Estrategia basada en la Simplicidad: simplicity

Esta estrategia se basa en la especificidad de una regla, entendiendo por tal el número de comparaciones que deben de realizarse en el antecedente de una regla. A menor número de comparaciones, menor especificidad.

Con esta estrategia, las nueva reglas, con la misma preferencia, se colocan por encima de aquellas instancias que tengan igual o mayor especificidad. Es decir, las que tenga menor especificidad serán las primeras en dispararse, y las de mayor especificidad (las más complejas) se dejarán para el final.

Pero ¿cómo se calcula ese valor de especificidad? La especificidad es una cantidad que incrementa 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 añada especificidad a la regla.

Como ejemplo considere la siguiente regla:

```
1: (defrule UnEjemplo
2:   (item ?x ?y ?x)
```



```
3: (test (and (numberp ?x) (> ?x (+ 10 ?y)) (< ?x 100)))
4: =>
```

¿Ve que tiene un valor de especificidad 5?, ¿no?. Observe:

1. 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`.
2. 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 `+`.
3. La unión de los términos establece que la regla tiene especificidad $1+4=5$.

12.3.5 Estrategia basada en la Complejidad: complexity

Esta estrategia es totalmente opuesta a la anterior. Las nuevas reglas, con la misma preferencia, se colocan por encima de aquellas instancias que tengan igual o menor especificidad. Las que sean más complejas se dispararán primero.

12.3.6 Estrategias temporales: `lex` y `mea`

Estas estrategias se basan en la estrategia OPS5. La idea intuitiva es establecer una ordenación temporal entre las reglas que tienen la misma prioridad. Ese orden temporal viene dado por el momento en que la regla se activa (lea el manual de CLIPS para más detalles.)

La estrategia `lex` lo que hace es poner en primer lugar los que tenga un *menor valor* temporal; es decir, aquellas que se hayan activado más recientemente.

Por contra, la estrategia `mea` coloca primero aquellas que tienen un *mayor valor* temporal, es decir aquellas que llevan mayor tiempo activadas.

DEPURACIÓN DE PROGRAMAS

En esta lección se le indicará algunos aspectos a tener en cuenta para que diseñe un programa más o menos coherente y eficiente en CLIPS.

13.1 Eficiencia: matches

Para conseguir un programa eficiente en tiempo de ejecución debemos conocer en primer lugar cómo funciona CLIPS en la activación de las reglas o, en otras palabras, cómo CLIPS hace casar los antecedentes de las reglas con los hechos.

El proceso general 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 suele variar 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)sactivación 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 tenga que gestionar la lista de hechos que satisfacen (parcialmente) su antecedente.

Por ejemplo, imagine que tiene una regla con antecedentes A, B y C. Imagine que pasado cierto tiempo de la ejecución de su programa se satisfacen A y B. Y, 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. Sólo el hecho C es de interés, pues éste ha sido lo único que se ha añadido a la lista de hechos. Así, aunque sea necesario utilizar un poco más memoria obviamente estamos ganando a cambio un menor tiempo de ejecución.

Es claro que esto conlleva que cada regla **debe de recordar** qué es lo que ya se satisfacía. Este 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 componen el antecedente. Más 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, AC, ABC. Cuando todas las coincidencias se cumplen, la regla se activa. Esta forma de trabajar de CLIPS nos puede ayudar para mejorar la eficiencia de un programa.

(matches <nombre-regla>)

Sintaxis 13.1: Sintaxis de matches

Para que el usuario conozca cuáles son las coincidencias parciales de una regla, CLIPS proporciona el comando **matches** (Sintaxis 13.1), que resulta muy útil para conocer qué reglas generan un número elevado de coincidencias parciales. Por ejemplo, ejecute el fichero batch que se muestra en la Sesión 13.1 y analice el resultado.

```
1:      (defrule regla (a) (b) (c) => )
2:      (assert (a) (b))
3:      (matches regla)
4:      (agenda)
5:      (retract 1)
6:      (matches regla)
7:      (agenda)
8:      (assert (c))
9:      (matches regla)
10:     (agenda)
11:     (assert (b))
12:     (matches regla)
13:     (agenda)
```

Sesión 13.1: Fichero batch que muestra el uso de matches

Ya que CLIPS utiliza el algoritmo **Rete Pattern Matching**, que como hemos dicho se basa en comprobar las coincidencias parciales, es importante que no se generen un número muy elevado de dichas coincidencias parciales.



Cuanto mayor sea el número de coincidencias parciales que se generen en una regla más ineficiente es esa regla.

El fichero batch que se muestra en la Sesión 13.2 muestra claramente las diferencias entre un número pequeño y un número elevado de coincidencias parciales.

Así, la pregunta obvia es ¿existe alguna forma de actuar para hacer que las reglas sean más eficientes?. Bueno, realmente no existe unos criterios que establezcan de un modo tajante cómo deben ordenarse los patrones para conseguir programas más eficientes. Pero si le podemos mostrar una pequeña guía de cómo deberían ordenarse los patrones.

```

1:      (deffacts Informacion
2:          (datos a c)
3:          (elemento a)
4:          (elemento b)
5:          (elemento c))
6:
7:      (defrule R1
8:          (datos ?x ?y)
9:          (elemento ?x)
10:         (elemento ?y) => )
11:
12:      (defrule R2
13:          (elemento ?x)
14:          (elemento ?y)
15:          (datos ?x ?y) => )
16:
17:      (reset) (matches R1) (matches R2)

```

Sesión 13.2: Un fichero batch: ¡Páralo ... si puedes!



- ◇ Los patrones más específicos, 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 provocará un menor 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, si se colocan al principio reducirán el número de coincidencias parciales.

Insistimos. Estos criterios no los interprete cómo normas estrictas que debe seguirse al pie de la letra porque, 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 rápido.

Otro criterio interesante es el siguiente:



La función test debe ponerse lo antes posible

Por ejemplo, en la Sesión 13.3 la regla R1 puede *optimizarse* si se reescribe como se muestra en la regla R2. Sin embargo, recuerde la advertencia del Apartado 11.2.

```
1: (defrule R1
2:     ?p1 <- (valor ?x)
3:     ?p2 <- (valor ?y)
4:     ?p3 <- (valor ?z)
5:     (test (and (neq ?p1 ?p2) (neq ?p2 ?p3) (neq ?p1 ?p3)))
6:     => )
7:
8: (defrule R2
9:     ?p1 <- (valor ?x)
10:    ?p2 <- (valor ?y)
11:    (test (neq ?p1 ?p2))
12:    ?p3 <- (valor ?z)
13:    (test (and (neq ?p2 ?p3) (neq ?p1 ?p3)))
14:    => )
```

Sesión 13.3: La regla R2 es una optimización de R1

13.2 Comentarios

Incluir comentarios en un programa en CLIPS, o en cualquier otro programa, siempre es una buena idea. Ayudan a entender los constructores difíciles y a explicar los qué se hace en el programa. Un comentario en CLIPS es cualquier texto que comienza con un punto y coma hasta el final de un retorno de carro.

La sesión 13.4 muestra un modo de usarlos.

13.2.1 Puntos de Corte: set-break, remove-break, show-break

Se entiende por punto de corte a una regla que permiten detener la ejecución de un programa antes de su disparo. Es decir, cuando una regla llega a un punto de corte significa que el programa se parará antes de que la regla sea disparada.

Su utilidad es evidente. Al establecer un punto de corte, se puede detener el programa en reglas que consideremos claves en su ejecución para observar el comportamiento de la estrategia, el contenido de las variables o de la agenda, etc. Al ejecutar de nuevo (run) el programa seguirá justo en el punto donde se quedó.

Los comandos asociados con los puntos de corte son **set-break**, **show-breaks** y **remove-break** y ya se mostraron en el Apartado .

```

1: ;*****
2: ;* Autor: Luis Daniel Hernández Molinero *
3: ;* Fichero: Inicio.clp *
4: ;* Creado el: 25/3/2003 *
5: ;* Ultima modificación: 26/3/2003 *
6: ;* Objetivo: Definir el conocimiento inicial del problema *
7: ;* Prerrequisitos: Ninguno. *
8: ;* Requerido por: Todos los demás ficheros. *
9: ;*****
10:
11: ; Definición de hechos iniciales
12: (def facts Informacion "Información del problema")
13:   (datos si no)
14:   (elemento si))
15:
16: ; Sólo tengo una regla. Debería de pensar en más.
17: (defrule R1 "Primera regla" ; esto también es un comentario
18:   (datos ?x ?y)
19:   (elemento ?x)
20:   =>
21:   (coche ?y))

```

Sesión 13.4: Uso de comentarios

13.3 Mostrando Toda la Memoria: [un]watch y list-watch-items

La función **watch** es el que nos permite visualizar todo lo que ocurre en la memoria del ordenador. Nos permite visualizar todo lo que se carga y se destruye en ella. Si desea dejar de visualizar lo que ocurre en la memoria de trabajo utilice la función **unwatch**. Las sintaxis de ambos comandos se muestran en la Sintaxis 13.2.

```
(watch <argumento>)
(unwatch <argumento>)
(list-watch-items [<argumento>])
donde
    <argumento> ::= all | facts | activations | rules | compilations |
                  statistics | globals | deffunctions
```

Sintaxis 13.2: Sintaxis de [un]watch y list-watch-items

Son muchos los argumentos de esta función (vea el manual de CLIPS si desea conocerlos todos). Destacamos algunos:

- ✧ **facts**. Ver Apartado 7.4.
- ✧ **activations**. Ver Apartado 9.4.
- ✧ **rules**. Ver Apartado 9.5.
- ✧ **compilations**. Ver apartado 15.2.
- ✧ **statistics**. Información sobre el tiempo, número medio de hechos, número medio de activaciones, etc.
- ✧ **globals**. Se mostrarán las asignaciones de las variables globales.
- ✧ **deffunctions**. Se mostraran el inicio y final de los constructores deffunctions.
- ✧ **all**. Se mostrará todo.

Para conocer qué elementos de la memoria se están mostrando o el estado de un elemento concreto se utiliza el comando **list-watch-items** sin argumento o con el argumento del elemento de interés, respectivamente (ver Sintaxis 13.2.)

13.4 Traza de los programas: dribble-on y dribble-off

Todo lo que se muestra en la consola de CLIPS, y en otros dispositivos internos de CLIPS (ver manual), se puede volcar en un fichero. Es decir, toda la traza de un programa puede volcarla en algún dispositivo para, posteriormente, analizarla detenidamente. Para activar la traza debe utilizar **dribble-on** y para desactivarla **dribble-off** (Sintaxis 13.3). El argumento **<fichero>** de **dribble-on** será el fichero físico de su dispositivo donde se volcará la traza.

(dribble-on <fichero>)
(dribble-off)

Sintaxis 13.3: Sintaxis de dribble-on y dribble-off

Lectura 14

LENGUAJE IMPERATIVO

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



Las llamadas de **todas** las funciones que vienen a continuación se colocan en el **consecuente** de las reglas.

14.1 Definición de Funciones Propias

A parte de las funciones que implementa CLIPS, puede definir sus propias funciones con el constructor `deffunction` según la Sintaxis 14.1.

```
(deffunction ::= <nombre> [<comentario-opcional>]
              (<parámetro>* [<parámetro-comodín>])
              <acción>*)
```

donde

```
<parámetro>      ::= <variable-unicampo>
<parámetro-comodín> ::= <variable-multicampo>
```

Sintaxis 14.1: Sintaxis del constructor `deffunction`

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 (como por ejemplo `length` o `nth`).
5. Una secuencia de acciones, o expresiones, que se realizarán en el orden especificado cuando se llame a la función.

El siguiente ejemplo del manual de CLIPS le ayudará a entender cómo funcionan las componentes <parámetro> y <parámetro-comodín>:

```
1: CLIPS>(deffunction argumentos (?a ?b $?c)
2:          (printout t ?a " " ?b " y " (length ?c) " extras: " ?c crlf))
3: CLIPS> (argumentos 1 2)
4: 1 2 y 0 extras: ()
5: CLIPS> (argumentos a b c d)
6: a b y 2 extras: (c d)
```

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 símbolo 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.

14.2 Asignación de Variables: bind

Puede crear variables o modificar el valor de éstas mediante la función **bind**. Esta función ya la estudió en la Sección 10.7.2 y se mostró un ejemplo de cómo utilizar dicho comando con variables globales en la Sesión 10.10.

Pero la función **bind** también puede utilizarse con variables locales. Como ejemplo, nos planteamos cómo resolver el problema de realizar una sumatoria usando un sistema de reglas. Una primera alternativa es la sesión 14.1. Es fácil comprobar que esta primera alternativa no permite solucionar el problema. Así, se opta por diseñar una regla más depurada, la de la sesión 14.2. Aunque esta regla sirve para nuestros propósitos presenta como inconveniente que el cálculo (+ ?total ?x) ha tenido que realizarse dos veces: una para hacer la sumatoria y otra para mostrar el resultado. Una alternativa más es la de la sesión 14.3 en la que se utiliza la función **bind** para almacenar el resultado de la suma.



Aunque CLIPS permite lenguaje imperativo, intente aplicar la siguiente regla para las funciones que vienen a continuación:

No abuse de ellas y si es posible **no las utilice**.

14.3 Función if...then...else

Esta función proporciona la conocida estructura de control if...then...else del lenguaje imperativo. Responde a la Sintaxis 14.3.

Si la <expresión> se evalúa con no FALSE se realizarán las 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:

```
1: (defrule SuperNota                                (defrule Nota
2:   (calificacion 10)                                (califacion ?valor&~10)
```

```

1: CLIPS> (deffacts ValoresIniciales
2:         (valor 1) (valor 2) (valor 3) (valor 4) (suma 0)) ←
3: CLIPS> (watch activations) ←
4: CLIPS> (watch rules) ←
5: CLIPS> (reset) ←
6: CLIPS> (defrule Sumatoria
7:         (valor ?x)
8:         ?suma <- (suma ?total)
9:         =>
10:        (retract ?suma)
11:        (assert (suma (+ ?total ?x)))) ←
12: ==> Activation 0    Sumatoria: f-4,f-5
13: ==> Activation 0    Sumatoria: f-3,f-5
14: ==> Activation 0    Sumatoria: f-2,f-5
15: ==> Activation 0    Sumatoria: f-1,f-5
16: CLIPS> (run 2) ←
17: FIRE    1 Sumatoria: f-1,f-5
18: <== Activation 0    Sumatoria: f-2,f-5
19: <== Activation 0    Sumatoria: f-3,f-5
20: <== Activation 0    Sumatoria: f-4,f-5
21: ==> Activation 0    Sumatoria: f-4,f-6
22: ==> Activation 0    Sumatoria: f-3,f-6
23: ==> Activation 0    Sumatoria: f-2,f-6
24: ==> Activation 0    Sumatoria: f-1,f-6
25: FIRE    2 Sumatoria: f-1,f-6
26: <== Activation 0    Sumatoria: f-2,f-6
27: <== Activation 0    Sumatoria: f-3,f-6
28: <== Activation 0    Sumatoria: f-4,f-6
29: ==> Activation 0    Sumatoria: f-4,f-7
30: ==> Activation 0    Sumatoria: f-3,f-7
31: ==> Activation 0    Sumatoria: f-2,f-7
32: ==> Activation 0    Sumatoria: f-1,f-7

```

Sesión 14.1: Primera Sumatoria

```

1: CLIPS> Repetir 1: - 5: de la Sesión 14.1.
2: CLIPS> (defrule Sumatoria
3:         ?valor <- (valor ?x)
4:         ?suma <- (suma ?total)
5:         =>
6:         (retract ?suma ?valor)
7:         (assert (suma (+ ?total ?x)))
8:         (printout t "Suma=" (+ ?total ?x) crlf)) ←
9: CLIPS> (run) ←
10: FIRE    1 Sumatoria: f-1,f-5
11: Suma=1
12: FIRE    2 Sumatoria: f-2,f-6
13: Suma=3
14: FIRE    3 Sumatoria: f-3,f-7
15: Suma=6
16: FIRE    4 Sumatoria: f-4,f-8
17: Suma=10

```

Sesión 14.2: Segunda Sumatoria

```

1: CLIPS> Repetir 1: - 5: de la Sesión 14.1.
2: CLIPS> (defrule Sumatoria
3:         ?valor <- (valor ?x)
4:         ?suma <- (suma ?total)
5:         =>
6:         (retract ?suma ?valor)
7:         (bind ?sub-total (+ ?x ?valor))
8:         (assert (suma ?sub-total))
9:         (printout t "Suma Parcial=" ?sub-total crlf)) ←
10: CLIPS> (run) ←
11: FIRE    1 Sumatoria: f-1,f-5
12: Suma Parcial = 1
13: FIRE    2 Sumatoria: f-2,f-6
14: Suma Parcial = 3
15: FIRE    3 Sumatoria: f-3,f-7
16: Suma Parcial = 6
17: FIRE    4 Sumatoria: f-4,f-8
18: Suma Parcial = 10

```

Sesión 14.3: Tercera Sumatoria

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

Sintaxis 14.2: Función if...then...else

```
3: =>                                =>
4: (printout t "Matricula" crlf))    (printout t ?valor " no es matricula" crlf))
```

Puede unir las dos reglas en tan sólo una como sigue:

```
1: (defrule Calificacion
2:   (calificacion ?valor)
3:   =>
4:   (if (= ?valor 10) then
5:     (printout t "Tiene matricula" crlf)
6:   else
7:     (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.

¡Defina la función factorial de un número natural! Si no lo consigue tiene la solución en la Sesión 14.4

14.4 Función return

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

```
(return [<expresión>])
```

Sintaxis 14.3: Función return

Sin argumento, no retorna ningún valor. Con argumento, retornará el valor de la <expresión> especificada.

Defina una función que devuelva tres posibles mensajes, "No puedo cruzar", "Párate por seguridad" o "Puedes Pasar" según la luz encendida de un semáforo. La solución en la Sesión 14.7

14.5 Función while

Se utiliza para realizar un conjunto de acciones hasta que deje de cumplirse una condición (Sintaxis 14.4).

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

Sintaxis 14.4: Función while

Las acciones del bucle, <acción>, se realizarán hasta que <expresión> se evalúe con el valor FALSE. Pueden utilizarse las funciones **break** (Sección 14.6) y **return** (Sección 14.4) para finalizar el bucle prematuramente. La función devuelve el valor FALSE salvo que la función **return** se utilice para finalizar el bucle.

Defina una regla que imprima desde el valor entero positivo v, dado en el antecedente de la regla, hasta 1. La solución la tiene en la Sesión 14.6

14.6 Función break

Se utiliza para terminar la ejecución de un grupo de instrucciones que se encuentran en un flujo de control (cualquier tipo de bucle). (Sintaxis 14.5).

```
(break)
```

Sintaxis 14.5: Función break

Defina una función que imprima los valores desde del 0 hasta num-1, para un num dado. Una solución en la Sesión 14.7.

14.7 Función loop-for-count

La función **loop-for-count** permite realizar un conjunto de acciones un número determinado de veces (Sintaxis 14.6). Es el equivalente al **for** en los lenguajes imperativos.

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

Sintaxis 14.6: Función loop-for-count

Se desarrollan las acciones <acción>* tantas veces como se haya especificado en <rango>. Por ejemplo,

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

```
3: Saludo al mundo
4: FALSE
```

Si tan sólo se especifica el `<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 x veces las acciones, donde $x = \text{<índice-final>} - \text{<índice-inicio>} + 1$

Por ejemplo,

```
1: CLIPS>(loop-for-count (?contador1 2 4) do
2:      (loop-for-count (?contador2 1 3) do
3:      (printout t ?contador1 " " ?contador2 crlf)))
4: 2 1
5: 2 2
6: 2 3
7: 3 1
8: 3 2
9: 3 3
10: 4 1
11: 4 2
12: 4 3
13: FALSE
```

Por último indicar que la función retorna el valor FALSE salvo que se utilice la función `return` para finalizar.

14.8 Función switch

Permite realizar un conjunto de acciones para cada uno de los valores concretos de una variable. Responde a la Sintaxis 14.7.

```
(switch <condición>
      <sentencia-caso>
      <sentencia-caso>+
      [<sentencias-por-defecto>])
```

donde

```
<sentencia-caso>      ::= (case <comparación> then <acción>*)
<sentencias-por-defecto> ::= (default <acción>*)
```

Sintaxis 14.7: Función switch

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

1. La función `switch` evalúa <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>.

Ya que este tipo de instrucciones le son familiares ¿por qué no hace una función que indique el número de días que tiene el mes representado por un valor de entre 1 y 12?. Una solución en la Sesión 14.8.

14.9 Algunos Ejemplos con Solución

- ✧ Defina la función factorial de un número natural. Solución: Sesión 14.4.

```

1: (deffunction factorial (?a)
2:   (if (or (not (integerp ?a)) (< ?a 0)) then ; Si no es entero o positivo
3:     (printout t "Factorial Error!" crlf) ; no puede calcularse (?a)!
4:   else
5:     (if (= ?a 0) then ; se supone que el factorial de 0 es 1
6:       1
7:     else
8:       (* ?a ( factorial (- ?a 1) ) )
9:     )
10:  )
11: )

```

Sesión 14.4: Definición de la función factorial de un número natural

- ✧ Defina una función que devuelva tres posibles mensajes, "No puedo cruzar", "Párate por seguridad" o "Puedes Pasar" según la luz encendida de un semáforo. Solución: Sesión 14.5
- ✧ Defina una regla que imprima desde el valor entero positivo v, dado en el antecedente de la regla, hasta 1. Solución: Sesión 14.6
- ✧ Defina una función que imprima los valores desde del 0 hasta `num-1`, para un `num` dado. Solución: Sesión 14.7.
- ✧ Defina una función que indique el número de días que tiene un mes representado por un valor de entre 1 y 12. Solución: Sesión 14.8.


```

1: CLIPS> (deffunction semaforo (?color)
2:           (if (eq ?color rojo) then
3:               (return "No puedes cruzar")))
4:           (if (eq ?color amarillo) then
5:               (return "Parate por seguridad")))
6:           "Puedes Pasar")
7: CLIPS> (semaforo rojo)
8: "No puedes cruzar"
9: CLIPS> (semaforo amarillo)
10: "Parate por seguridad"
11: CLIPS> (semaforo verde)
12: "Puedes Pasar"

```

Sesión 14.5: Devuelve un mensaje según la luz encendida del semáforo

```

1: CLIPS> (defrule Cuenta-Atrás
2:         (valor ?v)
3:         =>
4:         (while (> ?v 0)
5:             (printout t "Valor " ?v crlf)
6:             (bind ?v (- ?v 1)) ) )

```

Sesión 14.6: Una regla que imprime una cuenta atrás

```

1: CLIPS>(deffunction Cuenta (?num)
2:         (bind ?i 0)
3:         (while TRUE do
4:             (if (>= ?i ?num) then
5:                 (break))
6:             (printout t ?i " ")
7:             (bind ?i (+ ?i 1)))
8:             (printout t crlf))
9: CLIPS> (Cuenta 1)
10: 0
11: CLIPS> (Cuenta 10)
12: 0 1 2 3 4 5 6 7 8 9

```

Sesión 14.7: Una función que imprime desde 0 hasta num-1

```

1: CLISP> (deffunction dias (?no)
2:         (bind ?resto (mod ?no 2)) ;Detectamos si es un mes par
3:         (bind ?mitad (<= ?no 7)) ;Hasta julio
4:         (switch ?resto
5:           (case 0 then          ;Es un mes par
6:             (switch ?mitad      ;¿anterior a agosto?
7:               (case TRUE then
8:                 (if (= ?no 2) then ;Para febrero
9:                   (printout t "28 dias" crlf)
10:                  else
11:                    (printout t "30 días" crlf)
12:                  )
13:               )
14:             (case FALSE then (printout t "31 días" crlf))
15:           )
16:         )
17:         (case 1 then          ;Es un mes impar
18:           (switch ?mitad      ;¿anterior a agosto?
19:             (case TRUE then (printout t "31 dias" crlf))
20:             (case FALSE then (printout t "30 días" crlf))
21:           )
22:         )
23:         (default algo falla)
24:       ))
25: CLIPS> (dias 6)
26: 30 días
27: CLIPS> (dias 7)
28: 31 dias
29: CLIPS> (dias 8)
30: 31 días

```

Sesión 14.8: Número de días que tiene un mes

Lectura 15

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.

15.1 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.

15.2 Cargar y Salvar Constructores: load y save

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-defxxx>
```

donde **defxxx** indica el constructor que se está cargando y **<nombre-defxxx>** es el nombre que le dio 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.

15.3 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 simultáneamente. 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.



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

15.4 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.

15.4.1 Abriendo Fichero Lógicos: open.

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-lógico> [<modo>])
```

Donde

- ◇ <nombre-fichero> es el nombre del fichero físico que tiene en el disco (en general, es un dispositivo).
- ◇ <nombre-lógico> es el símbolo que usará para hacer referencia al fichero físico <nombre-fichero>.
- ◇ <modo> es alguno de los siguientes modos de apertura del fichero físico:
 - ✓ "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.

Por ejemplo, (open "datos.dat" datos "w") permite abrir un fichero para escritura.

15.4.2 Escritura sobre Ficheros Lógicos: printout y format.

El comando usual de escritura es la función

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

cuya interpretación es la siguiente: **printout** mostrará 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 estándar (pantalla) cuyo nombre lógico es **t**.

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 fácilmente la información volcada sobre el dispositivo. Estas secuencias de caracteres son:

- ◇ **crlf**, provoca un retorno de carro / nueva línea.
- ◇ **tab**, provoca una tabulación horizontal.

⇒ **vtab**, provoca una tabulación vertical.

⇒ **ff**, form feed.

El primer ejemplo de este tutorial en que se indica cómo mostrar mensajes en pantalla puede verlo en el paso 7: de la sesión 9.8. En la Sesión 15.1 se muestra cómo abrir un fichero para escritura sobre el que se vuelcan los valores "hola" y 10.5 en el dispositivo datos.

```
1: CLIPS> (open "datos.dat" datos "w") ↵
2: CLIPS> (printout datos "hola" crlf) ↵
3: CLIPS> (printout datos 10.5 crlf) ↵
```

Sesión 15.1: Un ejemplo de escritura en un fichero con **printout**

El formato de salida de la función **printout** está limitado a los caracteres de control indicados. Además la apariencia de los caracteres de control dependerá del sistema operativo que utilice por lo que resulta hasta cierto punto difícil escribir ficheros con cierta estética.

Si desea que el formato de salida sea algo más complejo es conveniente utilizar la función **format** en lugar de la función **printout**. Su sintaxis es:

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

donde **<expresión-string>** es un string usual pero que contiene secuencias de control. Esta expresión especifica 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 (de forma semejante a como lo hace el lenguaje C).

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 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 carácter.
 - 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 línea.
- r** , imprime un retorno de carro.
- %** , imprime el carácter %.

Por ejemplo, en el siguiente comando

```
(format nil "Nombre: %-15s Edad: %3d" "Jose Antonio" 25)
```

se tiene que

- ◇ **<nombre-logico>** es el dispositivo **nil**.
- ◇ **<expresión-string>** es la cadena "Nombre: %-15s Edad: %3d" con lo que imprimirá dos argumentos. El primero tendrá un espacio reservado de 15 caracteres y será justificado a la izquierda, y el segundo en un espacio reservado de 3 caracteres.
- ◇ **<expresion>*** son los valores "Jose Antonio" y 25.

En consecuencia, el comando escribirá "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.

15.4.3 Lectura desde Ficheros Lógicos: read y readline.

La función

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

permite leer información desde el dispositivo indicado en **<nombre-logico>**.

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

La información leída 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.

El siguiente ejemplo le permite leer **un dato** del teclado.

```
1: (defrule lectura-numero "Introduciendo un número"
2:   =>
3:   (printout t "cantidad? ")
4:   (bind ?numero (read))
5:   (assert (valor ?numero)))
```

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).

El siguiente ejemplo le permite leer **varios datos** del teclado.

```
1: (defrule lectura-calle "Introduciendo el nombre de una calle"
2:   =>
3:   (printout t "nombre de la calle? ")
4:   (bind ?nombre (readline))
5:   (assert (nombrecalle ?nombre)))
```

15.4.4 Cerrando Ficheros Lógicos: close.

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.

15.5 Borrado y Renombrado de Ficheros: remove y rename

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.

Lectura 16

ALGUNAS OPERACIONES CON ALGUNAS CLASES DE CLIPS



Para un listado completo de las funciones que dispone CLIPS consulte los manuales.

16.1 Operaciones Matemáticas

16.1.1 Funciones Estandares

- ⇨ Adición. Retorna la suma de sus argumentos.

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

✓ Ejemplo:

```
1: CLIPS> (+ 1 2 3) ↵
2: 6
3: CLIPS> (+ 1 2.0 3.0) ↵
4: 6.0
5: CLIPS> (+ 1 2.0 -3.0) ↵
6: 0.0
```

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

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

✓ Ejemplo:

```
1: CLIPS> (- 10 2 3) ↵
2: 5
3: CLIPS> (- 10 2.0 3) ↵
4: 5.0
5: CLIPS> (- 10.0 2.0 -3.0) ↵
6: 11.0
```

- ⇨ Multiplicación. Retorna el producto de todos sus argumentos.

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

✓ Ejemplo:

```
1: CLIPS> (* 1 2 3) ↵
2: 6
3: CLIPS> (* 1 2.0 -3) ↵
4: -6.0
5: CLIPS> (* 1.5 2.5 -3.5) ↵
6: -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>+)

✓ Ejemplo:

```
1: CLIPS> (/ 24 2) ↵
2: 12.0
3: CLIPS> (/ 24 6) ↵
4: 4.0
5: CLIPS> (/ 24 2 3) ↵
6: 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>)

16.1.2 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>)

16.1.3 Funciones Trigonómicas

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

16.1.4 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.

16.2 Operaciones con Lexemas

◇ (str-cat <expresión>*)

Concatena todos sus argumentos en un sólo string. Los argumentos pueden ser alguno de los siguientes tipos: lexemas, números o nombres de instancias.

◇ (sym-cat <expresión>*)

Concatena todos sus argumentos en un sólo símbolo. Los argumentos pueden ser alguno de los siguientes tipos: lexemas, números o nombres de instancias.

◇ (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>)

Evalúa 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.
- ◇ **(lowercase <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.

16.3 Operaciones Booleanas

16.3.1 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.

16.3.2 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.
- ◇ **lexemp.**
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.

16.3.3 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 $i + 1$.

16.3.4 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.

16.3.5 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.

16.4 Operaciones con Multicampos

- ◇ **(create\$ <expresión>*)**
Une la secuencia de valores <expresión>* para crear un valor multicampo. Si se llama sin argumentos crea un valor multicampo de longitud nula.

✓ Ejemplo:

```
1: CLIPS> (create$ Luis Daniel Hernández Molinero)
2: (Luis Daniel Hernández Molinero)
3: CLIPS> (create$ (+ 3 4) (* 2 3) (/ 8 4))
4: (7 6 2)
```

- ◇ **(nth\$ <entero> <multicampo>)**
Devuelve el campo especificado por <entero> del valor multicampo <multicampo>. El entero tiene que ser un valor estrictamente positivo y si es mayor al número de campos de la expresión multicampo devolverá nil.

✓ Ejemplo:

```
1: CLIPS> (nth$ 2 (create$ Luis Daniel Hernández Molinero))
2: Daniel
```

- ◇ **(subseq\$ <valor-multicampo> <inicio> <final>)**
Extrae los valores del valor multicampo <valor-multicampo> que se encuentran entre las posiciones (enteras) <inicio> y <final>.

✓ Ejemplo:

```
1: CLIPS> (subseq$ (create$ Luis Daniel Hernández Molinero) 1 2)
2: (Luis Daniel)
```

⇒ `(subseq$ <valor-multicampo> <inicio> <final>)`

Extrae los valores del valor multicampo `<valor-multicampo>` que se encuentran entre las posiciones (enteras) `<inicio>` y `<final>`.

✓ Ejemplo:

```
1: CLIPS> (subseq$ (create$ Luis Daniel Hernández Molinero) 1 2)
2: (Luis Daniel)
```

*Copia para la impartición de la asignatura
Idaniel en um.es
Ingeniería del conocimiento y S.I.*