1. **DESARROLLO DE APLICACIONES CON CLIPS**

**5.1. Introducción a CLIPS**

Desarrollado originalmente en la NASA a mediados de los 80, puede integrarse con C/C++ (en ambas direcciones). De hecho, su nombre es un acrónimo derivado de “C Language Integrated Production System”. La sintaxis (y el nombre) de CLIPS están inspirados en OPS (“Official Production System”), un shell para el desarrollo de sistemas expertos creado en 1977 por Charles Forgy durante su doctorado con Allen Newell en la Carnegie-Mellon University.

La primera implementación de OPS se realizo en LISP, de ahí la sintaxis de CLIPS que puede resultar algo extraña para el que solo haya utilizado lenguajes imperativos tipo C, C++ o Java. No en vano, hay quien interpreta el nombre de LISP [LISt Processing] como “Lots of Insipid Silly Parentheses”.

CLIPS es ahora ampliamente utilizado en todo el gobierno, la industria y el mundo académico.

**5.1.1. ¿Qué es CLIPS?**

CLIPS es un shell para el desarrollo de sistemas expertos que utiliza una estrategia de control irrevocable con encadenamiento hacia adelante.

**Características claves**

* Representación del Conocimiento: CLIPS proporciona una herramienta de cohesión para el manejo de una amplia variedad de conocimientos con el apoyo de tres paradigmas de programación diferentes: basado en normas, orientado a objetos y de procedimiento. Programación basada en reglas, permite el conocimiento para ser representada como heurística, o " reglas de oro ", que especifican un conjunto de acciones que se deben realizar para una situación dada. Programación orientada a objetos, permite que los sistemas complejos para ser modelados como componentes modulares (que puede ser fácilmente reutilizados para modelar otros sistemas o para crear nuevos componentes). Las capacidades de programación de procedimiento previstas por CLIPS son similares a las capacidades que se encuentran en lenguajes como C, Java, Ada, y LISP.
* Portabilidad: CLIPS está escrito en C para la portabilidad y velocidad y, se ha instalado en muchos sistemas operativos diferentes y sin cambios en el código. Los sistemas operativos en los que CLIPS ha sido probado incluyen Windows XP, MacOS X y Unix. CLIPS puede ser portado a cualquier sistema que tiene un C compatible ANSI o compilador de C++. CLIPS viene con todo el código fuente que puede ser modificada o adaptada para satisfacer las necesidades específicas del usuario.
* Integración / extensibilidad: CLIPS se pueden incrustar en el código de procedimiento, llamado como un subprograma, e integrados con lenguajes como C, Java , FORTRAN y ADA. Los clips pueden ser extendidos fácilmente por un usuario mediante el uso de varios protocolos bien definidos.
* Desarrollo Interactivo: La versión estándar de CLIPS proporciona un texto orientado entorno de desarrollo interactivo, incluidas las ayudas de depuración, ayuda en línea, y un editor integrado. Sus interfaces proporcionan características tales como: menús desplegables, editores integrados y múltiples ventanas se han desarrollado para el MacOS, Windows XP y ambientes Windows.
* Verificación / Validación: CLIPS incluye una serie de características para apoyar la verificación y validación de los sistemas expertos, incluido el apoyo para el diseño modular y la partición de una base de conocimientos, la comprobación de restricción estático y dinámico de los valores tragamonedas y argumentos de la función, y el análisis semántico de los patrones de reglas para determinar si las inconsistencias podrían impedir una regla de despido o generar un error.
* Plenamente documentados: CLIPS viene con una extensa documentación que incluye un manual de referencia y una guía de usuario.
* Bajo Costo: CLIPS se mantiene como el software de dominio público.

**5.1.2. Elementos básicos de una herramienta de un Sistema Inteligente**

Los elementos básicos que posee cualquier herramienta de generación de sistemas expertos son:

* **Hechos**: Una lista de hechos y ejemplos que describen el estado actual del sistema.
* **Base de Conocimiento**: Reglas que actúan sobre los hechos para producir resultados.
* **Motor de inferencia**: Controla la ejecución de las reglas.

**5.1.3. Entrada y salida de CLIPS**

* Existen dos versiones de CLIPS montadas sobre UNIX. Una de ellas para terminales texto y otra para terminales gráficos. Ésta última tiene un interface muy parecido al de la versión de Windows.

Para iniciar la sesión con CLIPS, escribiremos FZ\_CLIPS para la versión texto o FZ\_XCLIPS para la versión gráfica.

seneca> FZ\_XCLIPS

* Versiones de CLIPS montadas sobre Windows. Hacer doble click sobre el icono de CLIPS.

En cualquiera de estos casos, nos encontraremos dentro de la ventana principal del programa. En esta ventana, el símbolo del sistema es: CLIPS>

La forma de introducir mandatos en CLIPS es análoga a la de cualquier otro programa con una interfaz texto (DOS, UNIX, etc.), es decir, introducimos el mandato apropiado, finalizando dicha introducción con la tecla <Intro>.

Para salir del sistema, utilizaremos el comando (exit).

CLIPS>(exit)

**5.2. Elementos básicos de CLIPS**

**5.2.1. Hechos en CLIPS**

La información que CLIPS utiliza para conocer el estado del sistema se maneja mediante hechos. Los hechos pueden tener un solo campo, aunque lo más normal es que tengan más de uno. En los siguientes ejemplos se verán las formas más comunes de expresar hechos en CLIPS. Para cada una de estas formas, indicaremos primero su formato general (mediante la notación BNF: **notación de Backus-Naur[[1]](#footnote-1)**, también conocida por sus denominaciones inglesas **Backus-Naur form** (**BNF**), **Backus-Naur formalism** o **Backus normal form**, es un [metalenguaje](https://es.wikipedia.org/wiki/Metalenguaje) usado para expresar [gramáticas libres de contexto](https://es.wikipedia.org/wiki/Gram%C3%A1tica_libre_de_contexto): es decir, una manera formal de describir [lenguajes formales](https://es.wikipedia.org/wiki/Lenguaje_formal)).

Los hechos nos permiten alimentar la base de conocimientos.

**Atributo - valor**

Formato: **(assert (<atributo> <valor>))**

Ejemplos:

CLIPS> (assert (es-animal perro))

<Fact-0>  
CLIPS> (assert (presion-sanguinea alta))

<Fact-1>  
CLIPS> (assert (velocidad 5))

<Fact-2>  
CLIPS> (assert (velocidad 5))

FALSE  
CLIPS>

En este último ejemplo se muestra que, cuando pretendemos introducir un hecho que ya se ha introducido, CLIPS devuelve "FALSE", ante la imposibilidad de duplicarlo. A diferencia de otros lenguajes de programación, CLIPS no sobreescribe los hechos sino que, para modificarlos hay que primero eliminarlos (utilizando la instrucción (retract)) y luego reescribirlos.

**Objeto-atributo-valor**

Formato: **(assert (<objeto> <atributo> <valor>))**

Ejemplos:

CLIPS> (clear)

CLIPS> (assert (Lassie especie perro))

<Fact-0>  
CLIPS> (assert (Lassie domestico si))

<Fact-1>  
CLIPS>

**Relacional**

Formato: **(assert (<relación> <atributo> <valor>))**

Ejemplos:

CLIPS> (assert (tratado-con Perez penicilina))

<Fact-2>  
CLIPS> (assert (admitido-por Perez Dr-Lopez))

<Fact-3>  
CLIPS>

**Forma de ver los hechos**

La instrucción (facts) muestra todos los hechos que se han introducido con su número de identificador.

Ejemplo:

CLIPS>(facts)  
f-0 (Lassie especie perro) CF 1.00

f-1 (Lassie domestico si) CF 1.00

f-2 (tratado-con Perez penicilina) CF 1.00

f-3 (admitido-por Perez Dr-Lopez) CF 1.00

For a total of 4 facts.

CLIPS>(reset)  
CLIPS>(facts)  
f-0 (initial-fact) CF 1.00

For a total of 1 fact.

CLIPS>

En este ejemplo se muestra también que el comando (reset) elimina todos los hechos que se han creado (al igual que el comando (clear)). Sin embargo, en este caso, el comando inserta siempre un hecho, el denominado **hecho inicial**. Posteriormente se verá la utilidad de este hecho en la creación de programas.

**Tipos de hechos. Tipos de campos**

Los hechos presentados hasta ahora se denominan **hechos sin etiqueta**, porque los campos que integran el hecho no llevan ningún tipo de etiqueta identificativa. Esta razón hace que este tipo de hechos sean sensibles al orden. Es decir, estos dos hechos son diferentes:

CLIPS> (assert (cazado-por zorro Juan))

<Fact-1>  
CLIPS> (assert (cazado-por Juan zorro))

<Fact-2>  
CLIPS>

Posteriormente se estudiarán los **hechos con etiqueta**, también llamados **plantillas**. Este tipo de hechos es insensible al orden, ya que los datos se introducen indicando de forma explícita el campo al que pertenecen.

El tipo al que pertenecen los campos de un hecho sin etiqueta en CLIPS se asigna de forma automática cuando éste se almacena. En el caso de los hechos con etiqueta, por el contrario, al definirlos tendremos que indicar a qué tipo de campo pertenece cada uno. Los tipos de campos posibles en CLIPS son:

* float
* integer
* symbol
* string
* externaladress
* factadress
* instancename
* instanceadress

**Eliminando hechos**

La instrucción (retract n) donde n es el número de un hecho introducido, elimina el hecho en cuestión. Puede utilizarse el símbolo '\*' para eliminar todos los hechos introducidos hasta el momento.

**Introducción remota de hechos**

Existe una forma de introducir un conjunto de hechos en forma remota o diferida. Utilizando la instrucción (deffacts) lo hechos no se cargan en memoria directamente, sino que se cargan cuando hayamos reiniciado el sistema mediante (reset). El formato de la instrucción es el siguiente:

(deffacts (<hecho-1>)

(<hecho-2>)

...

(<hecho-n>)

)

Ejemplo:

(deffacts estado-inicial

(alta Jose-Martinez iop)

(anterior Jose-Martinez infarto)

(dolor Jose-Martinez lado-izquierdo)

(paciente Juan-Lopez asmatico)

)

**5.2.2. Reglas en CLIPS**

Las reglas son las verdaderas responsables de almacenar el conocimiento en el sistema experto, ya que realizan una o unas acciones determinadas cuando se cumplen una serie de condiciones (estas condiciones son las que realmente almacenan el conocimiento). En CLIPS todas las reglas son de la forma:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Si** |  |  |
|  |  | *condición-1* |  |
|  |  | *condición-2* |  |
|  |  | *condición-3* |  |
|  | **entonces** |  |  |
|  |  | *acción-1* |  |
|  |  | *acción-2* |  |
|  |  | *acción-n* |  |

**Activación de una regla**

Como se acaba de ver, en una regla existen dos miembros. El miembro izquierdo de la misma o **antecedente** está formado por una serie de condiciones. Cuando **todas** estas condiciones (hechos) se cumplen, la regla se activa, y se realizan las acciones del miembro derecho de la misma, llamado **consecuente**. Estas acciones suelen ser afirmaciones, eliminaciones de hechos u otras acciones distintas.

La ejecución de un sistema basado en reglas se basa en el cumplimiento de dichas reglas por los patrones (hechos) que se introducen. Cuando no hay patrones que cumplan las condiciones necesarias, la ejecución del sistema se detiene.

**La agenda**

La agenda es una estructura que muestra las reglas que se han activado y que van a ser ejecutadas, puestas por orden de ejecución. En fuzzyCLIPS, podemos ver la ventana agenda eligiendo la opción **Agenda Window** del menú Window. En la versión para terminales texto, utilizaremos el comando (watch agenda), que nos muestra en nuestro terminal cuando se ha activado una regla en cuestión.

**Definición de reglas**

La construcción (defrule) se utiliza para definir reglas en CLIPS. Su formato general es el siguiente:

(defrule nombre\_regla "Descripción opcional entre comillas"

(<patrón-1>); Miembro izquierdo de la regla (condiciones)

(<patrón-2>)

...

(<patrón-n>)

=>

(<acción-1>); Miembro derecho de la regla (acciones)

(<acción-2>)

...

(<acción-m>)

); Paréntesis de cierre

Ejemplos:

(defrule semaforo-rojo

(luz roja)

=> (printout t "Detengase" crlf))

(defrule trata-infeccion "Tratamiento empírico"

(Perez riesgo-infeccion si)

(Perez infeccion-antes si)

=> (assert (Perez dar penicilina)

)

**5.2.3. Variables en CLIPS**

Como en otros lenguajes de programación, CLIPS ofrece la posibilidad de utilizar variables para almacenar valores. Estas variables se diferencian de los hechos en que son dinámicas y que los valores que se les asignan pueden cambiar.

El nombre de una variable, o identificador de variable, siempre se escribe precedido de un signo de interrogación seguido de una símbolo (palabra) que es el nombre de la variable. La sintaxis es:

**?<nombredeVariable>**

Es importante recordar que en CLIPS las variables no pueden ser el primer campo de un hecho.

**Variables en el antecedente de una regla**

Un uso corriente de las variables es almacenar una valor en el antecedente de una regla y luego utilizarlo en el consecuente

Ejemplos

(defrule colorcoche (coche ?color)  
=>  
   (assert (coche ?color))

(defrule quienesquien (cazador ?cazador ?cazado)  
=>  
   (printout t ?cazador “dispara al” ?cazado crlf))

En el segundo de estos casos, comprobamos que en una afirmación puede haber más de una variable.

**Variables que almacenan direcciones**

Una variable puede también almacenar la dirección de un hecho. Esto es especialmente útil si queremos eliminar un hecho que se ha introducido y no conocemos la posición que ocupa en la memoria. Para asignar una dirección a una variable, utilizaremos el operador **<-**

Ejemplo

(defrule quienesquien2

   (?asesinato <- (cazador ?cazador ?cazado)

=>  
(printout t "Hecho numero " ?asesinato ":" crlf ?cazador "ha matado a" ?cazado crlf)

Note la diferencia entre la impresión del contenido de ?asesinato y las otras dos variables.

**Comodines**

Como se ha visto anteriormente, existen algunas situaciones en las que sólo se especifica una parte de un hecho, y el resto de éste puede contener cualquier valor o estar vacío. Para evitar tener que asignar el valor de estos campos a una variable, la presencia de un campo no vacío puede también detectarse usando un comodín.

El símbolo **?** podría denominarse **comodín monocampo**, ya que sustituye exactamente a un campo sencillo en una afirmación particular.

Ejemplo:

CLIPS> (defrule agencia-matrimonial (nombre Pedro ?)  
=>  
(assert (encontrado-Pedro-apellido si)))  
CLIPS> (assert (Pedro Gonzalez Lopez))  
CLIPS> fact-0  
CLIPS> (assert (Pedro))  
CLIPS> fact-1  
CLIPS> (assert (Pedro Lopez))  
CLIPS> fact-2  
CLIPS>

De los tres hechos anteriores, sólo el hecho-2 activa la regla, ya que es el único que posee el símbolo Pedro seguido exactamente de un símbolo.

**Comodines multicampo. Variables multicampo**

El símbolo **$?** se denomina **comodín multicampo**, ya que equivale a cero o más campos en un hecho. Así, en el ejemplo anterior, si cambiamos los símbolos de los comodines, los tres hechos activarían la regla.

El mismo símbolo antepuesto a un nombre indica una **variable multicampo**.

**5.2.4. La cola de activaciones**

CLIPS utiliza una estructura en cola para almacenar y operar con las reglas que se han activado en un instante dado. El mecanismo de activación de las reglas utilizado por CLIPS es el siguiente: CLIPS realiza un ciclo con todas las reglas en el orden en que éstas se han introducido, comprobando qué reglas cumplen las condiciones necesarias para ser activadas (es decir, que se cumplan todos y cada uno de los patrones que hay en su antecedente). Cuando esto sucede, la pone en la cola, de manera que, la última regla que se ha activado, sería la última en ejecutarse.

Además de la estructura en cola que CLIPS utiliza para almacenar las reglas, existe un *mecanismo de prioridades manipulable por el usuario*. Así, si dos reglas se activan simultáneamente, el orden en el que se colocarán en la cola es tal que la de mayor prioridad se coloque primero. Más adelante se verá cómo utilizar el mandato (salience) para modificar la prioridad de ejecución de una regla.

**5.2.5. Hechos más complejos**

* **La construcción Deftemplate**

Los hechos con etiqueta o plantillas son denominados **templates**. Esta estructura de datos es muy parecida a la estructura tipo registro en Pascal o a las variables estructura de C. Consiste en una serie de campos (slots) que almacenarán datos relativos a un hecho determinado. Dichos campos están etiquetados, siendo innecesario introducir sus valores en un orden predeterminado. Además, en este tipo de estructuras es posible introducir un valor por defecto para los campos, de forma que el usuario se evita la necesidad de introducir todos y cada uno de los valores de los campos al afirmar un hecho perteneciente a un tipo establecido.

* **Definición de plantillas**

La forma general de definir una plantilla es mediante la instrucción (deftemplate), cuya sintaxis general es la siguiente:

(deftemplate <nombre de plantilla> "Comentario opcional entre comillas"

<definición campo-1>

<definición campo-2>

...

<definición campo-n>

)

Donde:

<definición campo> = (**Slot** <nombre campo> [(type <tipo de dato al que pertenece> ) [default <valor por defecto>)] |

**Multislot** <nombre campo\_múltiple> [(type <tipo de dato al que pertenece> ) [default <valor por defecto>)]...)

Los puntos suspensivos indican que existen otras primitivas (también opcionales) que se pueden introducir al definir un campo en una plantilla.

Ejemplo: La siguiente instrucción define una plantilla llamada 'paciente'

(deftemplate paciente

(slot nombre)

(multislot apellidos)

(slot edad)

(slot sexo)

(slot volumen\_pulmonar)

)

y esta otra instrucción sirve para introducir un hecho de tipo paciente con sus distintos campos.

(assert (paciente

(nombreapellidos Jose Quintero Garcia)

(edad 52)

(sexo varon)

(volumen\_pulmonar 20))

)

Cuando se declara una plantilla se pueden incluir algunas primitivas para describir el tipo de campo que se está definiendo. Así, por ejemplo, la primitiva (type <tipo-campo>) nos indica el tipo de dato en el campo, y primitiva (default <valor-por-defecto>) nos indica el valor por defecto, que es el valor que toma el campo cuando no se introduce o cuando se hace (reset). La primitiva (default ?DERIVE) selecciona el valor apropiado para cada tipo de campo. La primitiva (range <valor-inicial> <valor-final>) se utiliza para definir el rango de valores permitidos en el campo. Por último, las primitivas (allowed-symbols), (allowed-strings), (allowed-numbers), (allowed-integers), (alloewd-floats), (allowed-values) se utilizan para enumerar los valores permitidos para un campo concreto.

En el siguiente ejemplo, modificamos la definición de la plantilla 'paciente' añadiendo valores por defecto y otras primitivas.

(deftemplate paciente

(slot nombre

(type string)

(default ?DERIVE))

(multislot apellidos

(type SYMBOL

(default ?DERIVE))

(slot edad

(type INTEGER)

(default 30))

(slot sexo

(type SYMBOL)

(allowed-symbols F f M m))

(slot volumen\_pulmonar

(type REAL)

(default 2.5))

(slot estado

(type SYMBOL)

(default baja)

(allowed-symbols alta baja))

)

* **Uso de plantillas en reglas**

Los hechos estructurados o plantillas se pueden utilizar en reglas de la misma forma que los hechos no estructurados. Veamos los siguientes ejemplos:

(defrule BuscoaJacques  
   (paciente (nombre Jacques) (apellidos $?apellidos))  
=>  
   (printout t "Te he encontrado, Santiago " ?apellidos crlf)  
)

(defrule adios  
   ?sano <- (paciente (nombre ?nombre) (estado alta))  
=>  
   (printout t "El paciente " ?nombre "ha sido dado de alta" crlf)  
   (retract ?sano)  
)

Este último ejemplo muestra cómo podemos manipular un hecho tipo plantilla en una regla.

Para terminar esta sección, mostraremos la instrucción (modify) que se utiliza para modificar los campos de una plantilla.

(defrule modificapaciente  
     ?erroneo <- (paciente (nombre pedrito))  
=>  
     (modify ?erroneo (nombre Pedro)  
)

**5.2.6. Otros Aspectos de Control**

A continuación se presenta algunos aspectos de CLIPS que ayudarán en la construcción de reglas. Por una parte, trataremos las denominadas **restricciones de campo**, una serie de operadores que se utilizan para restringir los valores que un patrón puede tener en el antecedente de una regla. El efecto que producen es equivalente a combinar afirmaciones en las reglas de CLIPS. Estas restricciones son la **restricción tilde**, que se utiliza de forma análoga a una negación, la **restricción barra**, que es en cierto modo equivalente a un operador de unión y la **restricción &**, que se usa en combinación con las otras restricciones. También se mostrará las **operaciones aritméticas en CLIPS**. Para terminar se verá una serie de funciones (**Bind**, **Create$**, **Read** y **Readline**) que van a permitir la entrada de datos por el usuario.

* **Restricción tilde ~**

La restricción conectiva tilde actúa negando el valor sobre el que actúa. Para aclarar este concepto, supongamos que queremos escribir una regla que niegue el paso a toda persona de la base de conocimiento que no tenga de nombre Juan. La regla sería algo parecido a:

(defrule prohibirpaso  
   (persona (nombre ~Juan))  
=>  
   (printout t "Prohibido el paso" crlf)  
   (assert (pasar no))  
)

Como podemos comprobar en el ejemplo, la regla se activará para todos los hechos del tipo persona cuyo nombre no sea Juan, y actuará dando un mensaje prohibiendo el paso y produciendo el hecho (pasar no), que se podría utilizar en una regla que sirviera, por ejemplo, para controlar la apertura de la puerta.

* **Restricción barra |**

La restricción conectiva barra se utiliza para combinar varios hechos. Supongamos que queremos modificar la regla anterior para que se permita el paso a Juan y Pedro. La regla que realizaría esta acción sería:

(defrule permitirpaso  
   (persona (nombre Juan|Pedro))  
=>  
   (printout t "Puedes pasar" crlf)  
   (assert (pasar si))  
 )

* **Restricción &**

Esta última restricción se utiliza para conectar varias restricciones en unión. Veamos el siguiente ejemplo:

(defrule permitirpaso2  
     (persona (nombre ?name&Juan|Pedro))  
=>  
   (printout t "Puedes pasar" ?name crlf)  
   (assert (pasar si))  
)

* **Operaciones aritméticas en CLIPS**

En CLIPS pueden también realizarse operaciones aritméticas sobre datos de tipo numérico. El lenguaje proporciona las operaciones y funciones aritméticas básicas **+**, **-**, **\***, **/**, **div**, **max**, **min**, **abs**, **float** e **integer**. Su significado es análogo al que tienen en cualquier otro lenguaje de programación.

Las expresiones numéricas se representan en el estilo conocido como **notación prefija**. Este estilo, también utilizado en LISP, consiste en anteponer el operador a los operandos de una operación o función dada. Así, para representar la suma de los enteros 2 + 3, deberíamos escribir:

(+ 2 3)

Las funciones pueden utilizarse en el antecedente y en el consecuente de una regla. Por ejemplo, se muestra cómo calcular la suma de dos números

(defrule Suma   
     (numeros ?x ?y)  
=>  
     (assert (resultado\_suma (+ ?x ?y)))  
)

Para evaluar una función en el antecedente de una regla tenemos que utilizar el signo igual (=), que informa a CLIPS de que debe evaluar la expresión que la sigue en lugar de usarla literalmente en una comprobación de patrones.

Ejemplo:

(defrule Hipotenusa   
     (numeros ?x ?y)  
     (stock ?ID =(sqrt(+ (\*\* ?x 2) (\*\* ?y 2))))  
=>  
     (printout t "Stock ID =" ?ID crlf)  
)

* **Función de asignación Bind. Create$.**

La analogía a la asignación de un valor a una variable en el antecedente de una regla mediante correspondencia de patrones es enlazar un valor a una variable en el consecuente usando la función **bind**. El siguiente ejemplo es una nueva versión de la función suma anterior:

(defrule Suma2   
     (numeros ?x ?y)  
=>  
     (assert (respuesta (+ ?x ?y)))  
     (bind ?respuesta (+ ?x ?y))  
     (printout t "La respuesta es " ?respuesta crlf)  
)

La función (bind) puede también usarse para enlazar valores mono o multicampo a una variable. Dicha función se usa para enlazar cero, uno o más valores a una variable sin necesidad de utilizar el operador $. Recuerde que en el antecedente de una regla, sólo podemos crear un patrón multicampo usando el operador $ sobre un campo, como en '$?x'. Sin embargo, el dólar '$' es innecesario sobre el antecedente ya que los argumentos de (bind) indican a CLIPS exactamente cuántos valores enlazar.

La siguiente regla ilustra algunos ejemplos de uso de la función (bind). La función (create$) se utiliza para crear un valor multicampo. Su sintaxis general es:

**(create$ <arg1> <arg2> ...  <argN>)**

donde pueden añadirse cualquier número de argumentos para crear un valor multicampo. Veámoslo en el siguiente ejemplo:

(defrule demostracion\_bind   
     (initial-fact)  
=>  
     (bind ?paciente1 (create$ Pedro Sanchez))  
     (bind ?paciente2 (create$ Juan Jimenez)  
     (bind ?paciente3 (create$ Jose Perez))  
     (printout t "El paciente 1 es " ?paciente1 crlf  
                 "El paciente 2 es " ?paciente2 crlf  
                 "El paciente 3 es " ?paciente3 crlf "fin" crlf)  
)

* **Read y Readline**

La función (read) se utiliza para leer información introducida desde el teclado (o desde un fichero abierto previamente). En el siguiente ejemplo veremos cómo se utiliza dicha función:

(defrule lee-entrada  
     (initial-fact)  
=>  
     (printout t "Introduzca un color primario" crlf)  
     (assert (color (read)))  
)

Una limitación que presenta la función (read) es que **sólo lee un campo**. Si queremos introducir varias palabras separadas por espacio, debemos encerrarlas entre comillas y CLIPS interpretará la cadena como un solo hecho. Una vez hecho esto, existen funciones que permiten convertir estas cadenas en los correspondientes hechos multicampo.

Otra limitación de la función (read) es que **no pueden introducirse paréntesis a menos que vayan encerrados entre comillas**. Por tanto, no podemos afirmar un hecho que contiene paréntesis.

La función (readline) se usa para leer valores múltiples hasta que finalicemos introduciendo un retorno de carro. Esta función lee los datos como una cadena. Para afirmar hechos introducidos mediante una función (readline), se utiliza la función (assert-string), que convierte una cadena de caracteres en un hecho de tipo no cadena. Veamos el siguiente ejemplo

(defrule test-readline  
     (initial-fact)  
=>  
     (printout "Introduzca una cadena" crlf)  
     (bind ?cadena(readline))  
     (assert-string(str-cat "(" ?cadena ")"))  
)

Como la función (assert-string) requiere utilizar paréntesis en el argumento que se va a convertir en un hecho, se ha utilizado en este ejemplo la función str-cat, que concatena una serie de subcadenas produciendo una sola.

Algunas webgrafías:

CLIPS: A Tool for Building Expert Systems

<http://clipsrules.sourceforge.net/>

Sebastián Ventura Soto. CLIPS.

<http://www.uco.es/users/sventura/misc/TutorialCLIPS/TutorCLIPS00.htm>

Redes de comunicación de datos: CLIPS

<http://www.it.uc3m.es/rcrespo/docencia/irc/apuntes/04b_Clips.pdf>

CLIPS

[ftp://mozart.dis.ulpgc.es/pub/PCI/**CLIPS**\_I.pps](ftp://mozart.dis.ulpgc.es/pub/PCI/CLIPS_I.pps)

### Introducción a la Programación en Clips

[https://fdoperez.webs.ull.es/doc/**clips**.pdf](https://fdoperez.webs.ull.es/doc/clips.pdf)

El Sistema CLIPS

<http://www.cs.upc.edu/~bejar/ia/transpas/lab/clips-n.pdf>

Manual de CLIPS

<http://www.unistmo.edu.mx/~daniel.garcia/MaterialDescargablePrivado/DocumentosClips.pdf>

1. https://es.wikipedia.org/wiki/Notaci%C3%B3n\_de\_Backus-Naur [↑](#footnote-ref-1)