INTRODUCCIÓN A LA PROGRAMACIÓN BASADA EN REGLAS

Lawrence Mandow Andaluz
Dpto. de Lenguajes y Ciencias de la Computación,
Universidad de Málaga

José Luis Pérez de la Cruz Molina
Dpto. de Lenguajes y Ciencias de la Computación,
Universidad de Málaga

Octubre de 2003

Índice general

Capítulo 1

INTRODUCCIÓN

1.1. Reglas y lenguajes de reglas.

1.1.1. El concepto de regla.

Durante los años de esplendor de los Sistemas Expertos, la idea de Sistema Basado en Reglas gozó de gran predicamento en los ambientes de investigación y desarrollo en Inteligencia Artificial y Ciencia Cognitiva. Hoy este esplendor se ha visto en cierta forma atenuado, lo cual no impide que la programación basada en reglas de producción siga siendo una herramienta conceptual especialmente adecuada para muchos dominios, ni que existan muchas herramientas para el desarrollo e implementación de sistemas basados en este paradigma.

Pero conviene desde un principio clarificar las diversas clases de reglas que aparecen en nuestro discurso ordinario y en los programas así implementados.

En primer lugar, podemos distinguir las *reglas declarativas* de las *reglas procedimentales*. Una regla declarativa puede leerse como "Si p es verdad, entonces q también es verdad"; por el contrario, una regla procedimental puede leerse como "Si p es verdad, entonces llevar a cabo la acción a".

Dentro de las reglas declarativas, podemos distinguir a su vez varios tipos, según la relación que establezcan entre la parte "SI" —llamada convencionalmente *izquierda* o LHS de la regla— y la parte "ENTONCES" —llamada convencionalmente *derecha* o, en inglés, RHS de la regla. Una regla es *deductiva* cuando establece una relación puramente lógica, en el sentido del cálculo de proposiciones clásico, entre su LHS y su RHS, llamados entonces más propiamente *antecedente* y *consecuente*. Por ejemplo, *SI el recuento de leucocitos es*

inferior a 6000, ENTONCES hay leucopenia. En los sistemas expertos, estas reglas suelen reflejar definiciones, convenciones o leyes.

Pero más frecuentemente las reglas declarativas establecen un nexo más difícil de formalizar entre su parte "SI" y su parte "ENTONCES". En unos casos se trata de un nexo *causal*, como por ejemplo en *SI llueve, ENTONCES las calles están mojadas.* Pero en otros casos se invierte el punto de vista y la misma información se ofrece como *SI las calles están mojadas, ENTONCES llueve.* Se trata entonces de la expresión de un paso de *razonamiento abductivo*, es decir, de la marcha de las evidencias o síntomas a las causas que los producen.

Dentro de las reglas procedimentales, podemos distinguir también dos tipos. De una parte tenemos las que pretenden actuar sobre el mundo exterior al programa, por ejemplo, *SI la presión es mayor de 200, ENTONCES desconectar la alimentación Y activar la alarma.* Pero otras reglas procedimentales actúan sobre la ejecución del mismo programa; por ejemplo, *SI la fase es diagnosticar y se ha hallado el diagnóstico, ENTONCES borrar que la fase es diagnosticar Y añadir que la fase es determinar el tratamiento.* El empleo de estas reglas permite simular mediante reglas de producción cualquier algoritmo imperativo.

En general, las reglas puden servir como elemento estructural de un programa; sin embargo, su empleo exclusivo lleva a sistemas difíciles de mantener, depurar y modificar. Es por ello conveniente emplear además elementos adicionales que estructuren los datos (plantillas, marcos, objetos) y las mismas reglas (módulos.)

La regla como elemento cognitivo básico también ha sido discutida, aunque hay escuelas psicológicas (por ejemplo, la de J. R. Anderson [1], [2] o la de A. Newell [24], [19], [23]) que la siguen considerando como un mecanismo válido para explicar la mayor parte de la cognición en los seres humanos y, por tanto, para proporcionar capacidades cognitivas a los sistemas artificiales.

1.1.2. Lenguajes de reglas.

Uno de los más exitosos sistemas expertos de los años 80 fue R1/XCON, cuyo dominio era la configuración de ordenadores Vax a partir de descripciones incompletas de los pedidos [22], [3]. John McDermott y sus colaboradores implementaron el sistema en Lisp; pero, sacando partido del carácter extensible de este lenguaje de programación, enseguida definieron sobre él un nivel adicional, constituido por un lenguaje de reglas y un motor de inferencias basado en el algoritmo Rete [13]. Esta herramienta (en principio denominada OPS [14]) es la que acabó conociéndose como OPS5 [12], [10], [18]. OPS5 fue un lenguaje empleado con relativa frecuencia en los "años dorados" de los Sistemas Expertos, e

incluso llegó a comercializarse. De él procede el núcleo de los lenguajes actuales de reglas de producción en lo que se refiere tanto a la sintaxis como a muchos aspectos de la implementación.

Una derivación comercial de OPS5 fue ART. ART proporcionaba un potente entorno de desarrollo y diversas extensiones y complementos del modelo de cálculo básico. Independientemente de sus potencialidades, ART corrió la misma triste suerte que la mayoría de los productos basados en la I. A. que durante aquellos años salieron al mercado.

En un laboratorio de la NASA, sin embargo, surgió la idea de reimplementar las ideas de OPS5 y ART para obtener así una herramienta más barata y más adaptable a diversas plataformas que pudiera emplearse en los proyectos internos de la casa. De esta forma nació CLIPS, es decir, *C Language Production System*, lenguaje mantenido y desarrollado durante los años noventa en el Johnson Space Center de Houston y cuya última versión es por ahora CLIPS 6.21 [8], [9]. Esta versión añade al modelo básico de cálculo basado en reglas de producción muchas características, como por ejemplo, elementos de programación orientada a objetos (modelos de paso de mensajes y de funciones genéricas), posibilidad de estructurar el sistema en módulos o subsistemas, elementos condicionales de tipo "logical", etc.

Ante el auge de Internet y de sus lenguajes asociados, en otro laboratorio del Gobierno de los EE.UU. surge de nuevo la idea de reimplementar este paradigma de programación, esta vez usando como soporte el lenguaje Java; de esta forma surge JESS [15], *Java Expert System Shell*. JESS fue en un principio un clon de CLIPS que permitía además integrar de manera fluida todos los recursos dados por Java (programación por paso de mensajes, hebras, etc.) Las versiones actuales de CLISP y JESS, sin embargo, tienen grandes diferencias, debido a que los respectivos desarrolladores han ido adicionando características muy diferentes (las ya mencionadas para CLIPS 6, frente a elementos de encadenamiento hacia atrás, por ejemplo, en JESS.)

Ninguno de los lenguajes anteriormente citados tiene incorporadas construcciones para el manejo directo de datos imprecisos o reglas inciertas; pero en un laboratorio del Gobierno canadiense se han desarrollado versiones difusas de CLISP y JESS llamadas FuzzyCLIPS [25] y FuzzyJESS [26], respectivamente.

1.2. Introducción a CLIPS.

El ya mencionado lenguaje CLIPS (*C Language Integrated Production System*) favorece tres estilos de programación diferentes: procedural, orientado a

objetos y basado en reglas.

Aquí se presenta el lenguaje principalmente desde el punto de vista de la programación basada en reglas, que es la idea original para la que CLIPS fue concebido. Las ideas básicas serán pues el *encadenamiento hacia delante* y la *correspondencia de patrones*.

Supondremos en principio que nuestros programas CLIPS se interpretan en el entorno de desarrollo de programas proporcionado con el lenguaje (figura 1.1). La llamada al intérprete abre una ventana de diálogo donde se pueden

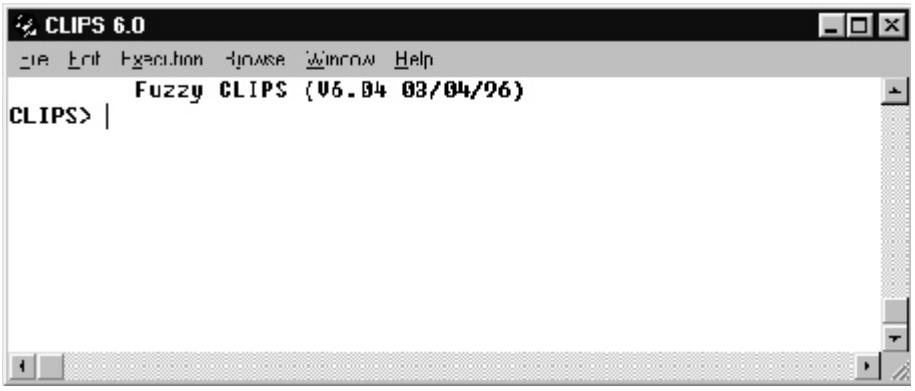

Figura 1.1: Ventana del entorno CLIPS

realizar llamadas a funciones u órdenes CLIPS. Estas llamadas devuelven un resultado y/o modifican el entorno de CLIPS. A través de esta interacción es posible definir un programa (o base de conocimiento), ejecutarlo, o trazar su ejecución paso a paso. Para salir del entorno CLIPS se puede ejecutar la orden `(exit)`. Como veremos más adelante, existen diversas formas de examinar el entorno de CLIPS y las modificaciones que se producen en el mismo.

1.2.1. Definición de un programa.

El entorno CLIPS puede almacenar el conjunto de definiciones que constituyen un programa. Los elementos básicos de un programa basado en reglas son los *hechos* y las propias *reglas*. Un programa CLIPS básico incorporará, por tanto, definiciones relativas a:

- Los tipos de hechos que va a manipular el programa.

- Los hechos iniciales, disponibles antes de que se ejecute el programa.
- Las reglas encargadas de manipularlos, y que siguen un formato *condición-acción*.

Más adelante estudiaremos otras construcciones que permiten a CLIPS dar mayor claridad y organización en programas complejos.

Normalmente las definiciones de un programa CLIPS no se escriben directamente en el intérprete, sino que se escriben en un fichero de *construcciones* o definiciones, que luego se carga utilizando la orden

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

También puede cargarse un fichero usando la opción "Load Constructs..."del menú 'File'. Cargar un fichero con definiciones de programa tiene el mismo efecto que si se escribiesen una a una directamente en el intérprete. La extensión por defecto de estos ficheros es `.clp`

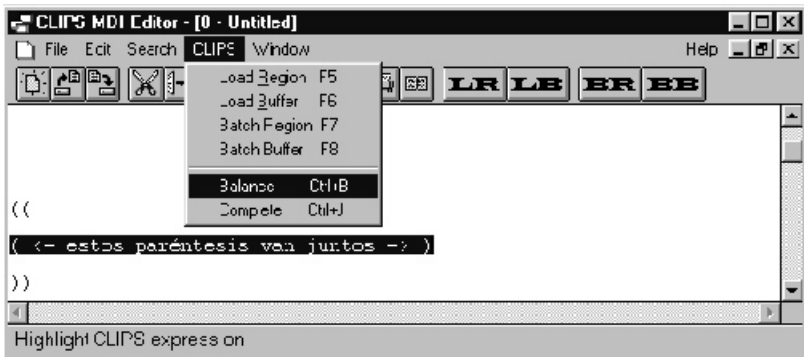

Figura 1.2: Ventana del editor integrado

Para el desarrollo de programas CLIPS se puede emplear cualquier editor avanzado, que al menos balancee los paréntesis. Además, el entorno CLIPS incluye un editor de texto integrado (figura 1.2) al que se accede desde el menú 'File', que incorpora algunas características que facilitan el desarrollo y prueba de los programas. Desde el editor es posible crear o abrir un fichero, incluir en él definiciones de programa, y a continuación cargarlas directamente. Una forma es utilizar dos botones que aparecen en el menú del editor:

- LB (load buffer): carga el contenido completo del fichero en el entorno CLIPS.

- LR (load region): carga el contenido de la región seleccionada en el entorno CLIPS.

Ambas opciones están disponibles también en el menú 'CLIPS' del editor y disponen de teclas de función asociadas.

Otra utilidad casi imprescindible del editor es la opción 'Balance' que localiza el compañero de cualquier paréntesis cerrado o abierto.

1.2.2. Ejecución de un programa.

En las representaciones basadas en reglas se considera que el conocimiento está constituido fundamentalmente por hechos y reglas. Los hechos son afirmaciones incondicionales acerca de algún aspecto del dominio del sistema; las reglas son afirmaciones condicionales de la forma SI *combinación de hechos del caso* ENTONCES *acciones sobre los hechos del caso.*

La base de conocimientos (reglas y hechos) y la memoria de trabajo (hechos) son manejadas por otro módulo separado, el llamado motor de inferencias.

Según esto, el intérprete o motor de inferencias CLIPS manipulará dos componentes (en esta manipulación es precisamente en lo que consiste la ejecución de un programa a partir de su definición):

- La lista de hechos o *memoria de trabajo.* Es una lista que contiene todos los hechos o datos conocidos o disponibles para el programa. Antes de ejecutar un programa hay que preparar la lista de hechos borrando todo su contenido e incluyendo a continuación los hechos que el programa describe como iniciales. El contenido de la lista de hechos cambia normalmente a medida que se ejecuta un programa.

- La *agenda.* Es una lista con todas las reglas activadas del programa. Una regla se activa cuando hay una correspondencia entre su parte izquierda (antecedente o condición) y el contenido de la lista de hechos. Siempre que hay más de una regla activada en la agenda se utiliza un mecanismo de resolución de conflictos para decidir cuál de ellas será la que finalmente se ejecute (o, como se dice en la jerga de la programación basada en reglas, se "dispare"). El efecto de disparar una regla consiste normalmente en una modificación de la lista de hechos. Cada vez que la lista de hechos se ve modificada, es necesario recalcular el contenido de la agenda. Antes

de ejecutar un programa hay que preparar la agenda, borrando todo su contenido e incluyendo a continuación las reglas activadas por el contenido inicial de la lista de hechos.

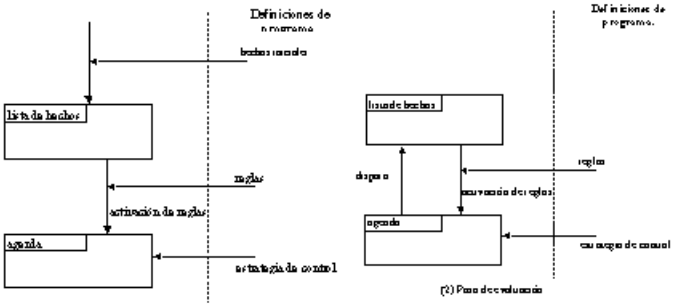

Figura 1.3: Ciclo de evaluación CLIPS

El funcionamiento básico del entorno CLIPS durante la ejecución de un programa puede resumirse de la siguiente forma (figura 1.3):

1. Inicialización:

   a) se borra el contenido de la lista de hechos y la agenda,

   b) se añaden los hechos iniciales a la lista de hechos,

   c) se añaden a la agenda las reglas activadas por los hechos iniciales.

2. Evaluación. Se desencadena un proceso cíclico consistente en los siguientes pasos:

   a) Se selecciona la primera regla de la agenda. Con frecuencia, durante un ciclo de ejecución, es posible que varias reglas estén activadas en la agenda. Para decidir cuál de ellas resulta elegida se utiliza una estrategia de control (o resolución de conflictos) que puede formar parte de la definición del programa (vd. las secciones 2.2).

   b) Se dispara la regla, es decir, se ejecutan las acciones de su parte derecha. Como norma general, las reglas se utilizan para modificar el contenido de la lista de hechos.

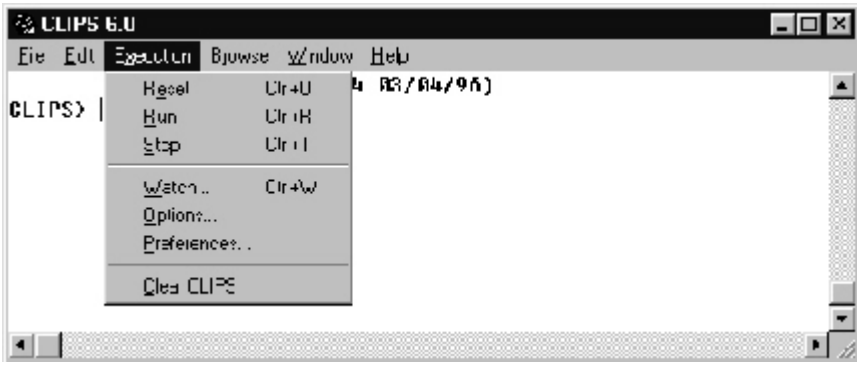

Figura 1.4: Menú '*Execution*'

c) Se recalcula el contenido de la agenda (conjunto de reglas activadas) en función del nuevo estado de la lista de hechos.

Este proceso se repite hasta que o bien la agenda queda vacía, o bien alguna regla provoca explícitamente la terminación del programa invocando la orden `(halt)`.

Las órdenes básicas para la ejecución de programas en CLIPS son:

- `(clear)`. Esta llamada elimina todas las definiciones de programa (tipos de hechos, reglas, etc.) que pudiese haber almacenadas en el entorno. En otras palabras, deja el entorno CLIPS listo para la definición de un nuevo programa, y su efecto es el equivalente salir del entorno CLIPS y abrirlo de nuevo.

- `(reset)`. Esta llamada realiza todas las tareas previas a la ejecución del programa cargado actualmente en el entorno CLIPS. Concretamente borra el contenido de la lista de hechos y la agenda. Guarda en la lista de hechos los hechos iniciales y en la agenda las reglas activadas por ellos.

- `(run)`. Inicia el ciclo de evaluación del programa, que se detendrá bien cuando la agenda quede vacía o se ejecute la orden (halt).

- `(step)`. Realiza un único ciclo de evaluación, es decir, se dispara únicamente una regla. Esta órden es muy útil para trazar paso a paso la ejecución de un programa.

De este modo es posible cargar o definir un programa, ejecutarlo tantas veces como sea necesario, y volver a cargar o definir un programa nuevo, todo ello sin necesidad de salir del entorno.

Todas estas órdenes pueden ejecutarse también desde el menú '*Execution*' en la ventana de la interfaz (figura 1.4).

1.2.3. FAQs de programadores expertos.

P.– Lo que se cuenta en los párrafos anteriores parece describir una interfaz dependiente de algún sistema operativo concreto, ¿no?
R.– No, no es así. El entorno está disponible para MS-Windows, para MacOS y para Linux y Unix en general (X Window system release 11R6.4).
P.– Un amigo mío odia los entornos de ventanitas y opina que lo único que vale la pena usar es la "línea de comandos". ¿Le será posible programar en CLIPS?
R.– Por supuesto; el entorno de ventanas puede considerarse como un conjunto de módulos añadidos a un intérprete; cada opción del menú es traducible a una definición, orden o función CLIPS. Así que tu amigo puede cargar un intérprete "sin ventanas", también disponible.
P.– Por cierto, ¿cómo se pueden conseguir todas estas herramientas?
R.– Todo el código fuente de CLIPS y los ejecutables para MS-DOS, MS-Windows y MacOS estan disponibles en el sitio web de CLIPS, que en el momento de escribir estas páginas reside en `http://www.ghg.net/clips/CLIPS.html`. El código fuente de CLIPS puede compilarse con un compilador ANSI C, o también con un compilador C++. Los compiladores que no sean ANSI C deben al menos soportar por completo el estilo de prototipos de función ANSI y el tipo de datos `void`.
P.– ¿Y en qué condiciones se pueden adquirir y emplear CLIPS y su entorno de programación?
R.– Gratis y para cualquier uso. Dice el manual: "Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the Software), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so. CLIPS is released as public domain software and as such you are under no obligation to pay for its use. However, if you derive commercial or monetary benefit from use of the software or just want to show support, please consider making a voluntary payment based on the worth of the software to you as compensation for the time and effort required to develop and

maintain CLIPS."
P.– ¿Y es necesario estar dentro del entorno de ventanas o estar ejecutando el intérprete para ejecutar programas CLISP?
R.– No. En primer lugar, es posible invocar al intérprete con la opción `-f`, de forma que se al cargarlo se ejecute automáticamente un fichero "batch" (esto se explica en la sección 2.4).

Y, para aplicaciones más ambiciosas, también es posible compilar un programa CLIPS junto con el intérprete, creando un programa ejecutable al que se puede llamar directamente desde el SO (es lo que se suele llamar un *run-time*). La forma de hacer esto, un poco engorrosa, se explica con todo detalle en el manual avanzado de programación [8], sección 5 *Creating a CLIPS Run-time Program*, y algo de ello mencionaremos más adelante en estas lecciones.
P.– ¿Puede llamarse un programa CLIPS desde otro programa externo? Y, a la inversa, ¿se puede llamar un programa externo desde un programa CLIPS?
R.– La respuesta a ambas preguntas es afirmativa. Esta cuestión también está explicada con detalle en el manual avanzado de programación [8], en las secciones 3 (*Integrating CLIPS with External Functions*), 4 (*Embedding CLIPS*) y 6 (*Combining CLIPS with Languages Other Than C*).
P.– ¿Es lo que se explica en estas lecciones específico de CLISP, o sirve también para los restantes lenguajes de la familia?
R.– Depende. El estilo general de la programación basada en reglas y la mayoría de las construcciones aquí expuestas son comunes a todos estos lenguajes.
P.– ¿Por qué se ha elegido el lenguaje CLIPS en lugar de otros más modernos, como JESS?
R.– Hay varias razones. En primer lugar, la latitud de la licencia CLIPS permite que cualquier programador desarrolle sus aplicaciones y las venda, si así lo desea, o que modifique el código fuente para adaptarlo a sus necesidades. Otra gran ventaja es que CLIPS está perfectamente documentado; incluso existe un libro de texto sobre él [16]. Además, CLIPS ocupa un lugar central en la familia de implementaciones de lenguajes de producción, es razonablemente estable y aún está mantenido por su creador Gary Riley. Por otra parte, hemos de reconocer que JESS tiene la gran ventaja de estar implementado como una clase Java, con lo cual la integración con el mundo exterior resulta mucho más fácil.

Capítulo 2

CONSTRUCCIONES BÁSICAS

2.1. Hechos y Reglas.

2.1.1. Hechos.

Todo sistema de producción dispone de una base de datos o memoria de trabajo donde almacenar los hechos conocidos en cada momento. En CLIPS esta memoria de trabajo se implementa mediante una lista de hechos. Cada hecho de la lista tiene una dirección y un índice (un número entero no negativo) que lo identifican unívocamente. Los índices se van asignando consecutivamente a los hechos, empezando por el número 0.

Es posible en todo momento visualizar el contenido de la lista de hechos mediante la función `(get-fact-list)` (que devuelve una lista de direcciones de hechos) o bien mediante la función `(facts)` que, al igual que la ventana *Facts* del entorno (figura 2.1), muestra los hechos explícitamente con sus correspondientes índices.

Los hechos más simples en CLIPS son los llamados "hechos ordenados". Su sintaxis es la siguiente:

```
(<nombre-de-relación> <valor>*)
```

Un hecho ordenado consta de un nombre de relación seguido de cero, uno o más valores constantes. Por ejemplo, son hechos ordenados:

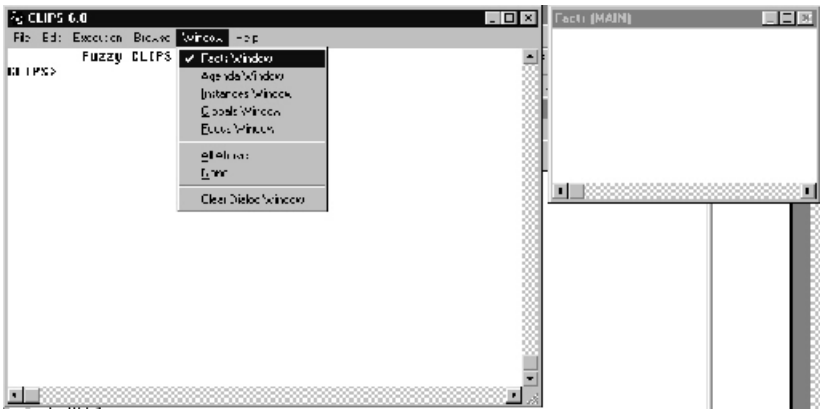

Figura 2.1: Activación de la ventana de hechos

```
(alarma-conectada)
(temperatura 20)
(es-un piolin animal)
```

De esta forma se pueden representar los enunciados básicos del lenguaje natural o, dicho de otra forma, los átomos de la lógica de predicados. Por ejemplo, si tenemos la afirmación "Pepe es alto", como sabemos, podemos simbolizarla como alto(pepe) o como altura(pepe, grande). También podemos emplear las ternas *objeto-atributo-valor*, definiendo una única relación `oav` que tendrá como argumentos el objeto del que hablamos, el atributo considerado y el valor que toma el atributo en el objeto; en este caso sería oav(pepe, altura, grande). En CLIPS tendríamos:

```
(alto pepe)
(altura pepe grande)
(oav pepe altura grande)
```

La definición o construcción `deffacts` permite describir el contenido inicial de la lista de hechos. Su sintaxis es la siguiente:

```
(deffacts <nombre>
   [<comentario>]
   <hecho>*)
```

Cada vez que se ejecute la orden `(reset)`, se borrará todo el contenido de actual de la lista de hechos; y, a continuación, los hechos indicados en todas las órdenes `deffacts` se añadirán a la lista de hechos como preparación ante la ejecución del programa.

Puede observarse que tras ejecutar la orden `(reset)` aparece siempre un hecho inicial llamado `(initial-fact)`. Es decir, la lista de hechos nunca estará vacía al iniciar la ejecución de un programa, ya que en tal caso ninguna regla se encontraría activa en la agenda. Así que la siguiente construcción está predefinida en CLIPS:

```
(deffacts initial-fact (initial-fact))
```

Es un error intentar redefinir el `deffacts initial-fact`. Además, al ejecutar `(reset)` se pondrá el contador de hechos a 0, con lo cual el índice de `(initial-fact)` será 0 y el del siguiente hecho asertado será 1.

En efecto, además de los hechos iniciales, a lo largo de la ejecución de un programa es posible añadir nuevos hechos, y también borrar hechos existentes. Ello se consigue con las siguientes funciones:

`(assert <hecho>+)`: añade uno o mas hechos a la lista. El hecho se especifica explícitamente por su contenido. La función devuelve la dirección del último hecho añadido. Por ejemplo:

```
> (assert (kk 1))
<Fact-234>
> (assert (oav peso pepe excesivo))
<Fact-235>
```

`(load-facts <nom-fichero>)`: lee el contenido del fichero `<nom-fichero>`, que debe ser un fichero de texto que contenga un conjunto de hechos correctamente escritos. La función devuelve TRUE si los hechos se cargaron con éxito, FALSE en otro caso. Por ejemplo, supongamos que el fichero "kk.txt" contiene una sola línea `(a 1) (b 2) (c 3)`. Entonces

```
> (clear)
> (load-facts "kk.txt")
TRUE
> (facts)
0: (a 1)
1: (b 2)
2: (c 3)
For a total of 3 facts.
```

`(retract <índice-de-hecho>|<direccion-de-hecho>+)`: borra uno o más hechos de la lista (y no devuelve nada). Es un error intentar borrar un hecho inexistente.

Ya que normalmente no se conocerán los índices ni las direcciones de los hechos, habitualmente se empleará como argumento de `retract` una variable que se ha ligado a la dirección de un hecho en la parte derecha de una regla; el lector deberá esperar al capítulo 4 para una explicación completa de ello.

Ejemplo 2.1 Esta podría ser una sesión interactiva con el intérprete CLIPS:

```
>(reset)
>(get-fact-list)
(<Fact-0>)
> (assert (kk 1))
<Fact-1>
> (assert (oav peso mari ok))
<Fact-2>
> (retract 1)
>(get-fact-list)
(<Fact-0> <Fact-2>)
> (assert (kk 1))
<Fact-3>
> (retract 1)
Error: no pude encontrar el hecho f-1.
```

◁

Ejercicio 2.1 Trazar el contenido de la memoria de trabajo al ir ejecutando la siguiente sucesión de llamadas a `assert/retract`:

```
(reset)
(assert (fase 1) (nivel 12))
(assert (oav presion s1 ok) (oav temperatura s2 alta))
(retract 3)
(retract 1)
(assert (fase 2))
(retract 5)
```

◁

También es posible almacenar en un fichero los hechos de la lista de hechos actual. Ello se consigue con la función

`(save-facts <nom-fichero>)`: crea un fichero de texto `<nom-fichero>` y almacena en él la lista de hechos, de forma que posteriormente se pueda cargar con `load-facts`. La función devuelve TRUE si los hechos se almacenaron con éxito, FALSE en otro caso.

Ejercicio 2.2 Trazar el contenido de la memoria de trabajo al ir ejecutando la siguiente sucesión de funciones:

```
(clear)
(reset)
(assert (fase 1) (nivel 12))
(retract 0)
(save-facts "kk.txt")
(clear)
(assert (oav presion s1 ok) (oav temperatura s2 alta))
(load-facts "kk.txt")
(retract 0)
(assert (fase 2))
```

◁

2.1.2. Reglas.

Las reglas son los elementos que describen la forma en que deben procesarse los hechos. En términos generales, el objetivo de una regla es modificar el contenido de la memoria de trabajo. Esto se consigue normalmente añadiendo, borrando o modificando uno o más hechos. El espíritu de la programación basada en reglas exige que, en la medida de lo posible, cada regla represente un heurístico o inferencia básica completa. En otras palabras, si es posible, ninguna regla debe depender de otra.

Las reglas en CLIPS se definen utilizando la definición o construcción `defrule`:

```
(defrule <nom-regla>
   [<comentario>]
   <elemento-condicional>*
 =>
   <acción>*)
```

Cuando todos los elementos condicionales se satisfacen (es decir, existe una correspondencia con la lista de hechos), entonces decimos que la regla se activa o que hay una activación de la regla. El disparo de una regla activa o activación provoca la ejecución secuencial de sus acciones. Si no aparece ningún elemento condicional, se supone que el antecedente de la regla es el hecho `(initial-fact)`. Las acciones de una regla pueden ser cero, una o más llamadas a órdenes y/o funciones (normalmente una regla sin acciones no tendrá ninguna utilidad.)

Existen diversas clases de elementos condicionales. Los más básicos son los llamados *patrones*, que describen mediante constantes, variables y restricciones hechos que pueden estar presentes en la memoria de trabajo. Por ahora sólo emplearemos constantes, así que nuestros patrones serán simplemente hechos. Dado un hecho h como patrón, estará en correspondencia con la memoria de trabajo y se satisfará si y sólo si h pertenece en ese momento a la memoria de trabajo.

Con todo esto ya estamos en condiciones de escribir nuestro primer programa CLIPS completo.

Ejemplo 2.2 Supóngase que deseamos determinar a qué lugar ir un fin de semana, en función de la estación del año y de la actividad que deseamos realizar (más bien gastronómica o más bien deportiva.) Para ello distinguimos hasta tres niveles de refinamiento en la respuesta: destino (campo o playa), zona (suclases de destinos) y lugar (instancia concreta de destino). El siguiente programa `VAC1.CLP` implementa en CLIPS algunas reglas sencillas para resolver este problema:

```
; Programa VAC1.CLP
; POSIBLES HECHOS INICIALES DEL PROGRAMA
; (estacion primavera|verano|otonyo|invierno): la estacion del año
; (actividad gastronomica|deportiva): tipo de actividad deseada
; POSIBLES HECHOS INFERIDOS POR EL PROGRAMA
; (destino campo|playa)  : posibles clases de destinos turisticos
; (zona   <nombre-de-zona>) : subclases de destinos turisticos
; (lugar   <nombre-de-lugar>) :instancias de destinos turisticos
;es primavera y deseamos hacer una excursion gastronomica
(deffacts hechos-iniciales
   (estacion primavera)
   (actividad gastronomica))
;si es primavera, los destinos posibles son campo y playa
(defrule posibilidades-primaverales
   (estacion primavera)
=>
   (assert (destino campo)
      (destino playa)))
;si un destino posible es playa, una zona recomendada es Malaga
(defrule playa-de-Malaga
   (destino playa)
=>
   (assert (lugar Malaga)))
;si un destino posible es playa, una zona recomendada es Torremolinos
(defrule playa-de-Torroles
   (destino playa)
=>
   (assert (lugar Torremolinos)))
;si un destino posible es campo y una actividad deseada es gastronomica,
```

```
;entonces una zona recomendada es iberico
(defrule campo-gastronomico
   (destino campo)
   (actividad gastronomica)
=>
   (assert (zona iberico)))
;si un destino posible es campo y una actividad deseada es deporte,
;entonces una zona recomendada es alta-montanya
(defrule campo-deportivo
   (destino campo)
   (actividad deporte)
=>
   (assert (zona alta-montanya)))
;si una zona posible es iberico, un lugar recomendado es Aracena
(defrule para-iberico-Aracena
   (zona iberico)
=>
   (assert (lugar Aracena)))
;si una zona posible es alta-montanya, un lugar recomendado es Aracena
(defrule Sierra-Nevada
   (zona alta-montanya)
=>
   (assert (lugar Sierra-Nevada)))
```

◁

Ejercicio 2.3 Modificar el programa anterior para responder a las siguientes nuevas especificaciones:

- Estamos en primavera y la actividad es deportiva.
- El programa debe pararse tras deducir el primer lugar recomendado.
- Deben expresarse también algún conocimiento acerca de los lugares recomendados en invierno.

◁

Ejercicio 2.4 Consideremos las siguientes reglas relativas a la calificación de los alumnos matriculados en *"Papiroflexia Avanzada"*:

Se supera la asignatura si se superan tanto la teórica como la práctica. Se supera la teórica si la calificación en cada uno de los dos exámenes teóricos ha sido "apto". Se supera la práctica si la calificación en cada una de las dos prácticas ha sido "apto", o bien si en una ha sido "no apto" y se ha realizado un trabajo complementario.

1. Dar un conjunto de objetos, atributos y valores adecuado para simbolizar toda esta información.

2. Representar la información anterior mediante un conjunto de reglas CLIPS, empleando un predicado "objeto-atributo-valor".

3. Escribir un programa CLIPS que calcule qué alumnos han superado la asignatura. Sabemos además que las calificaciones de un alumno de la clase han sido las siguientes:

T1	T2	P1	P2	TRABAJO
APTO	APTO	NO APTO	APTO	OK

   Introducir esta información en un `deffacts` y calcular mediante CLIPS si este alumno ha superado la asignatura.

4. Id. para otro alumno dado por

T1	T2	P1	P2
APTO	APTO	NO APTO	APTO

◁

2.1.3. Elementos condicionales booleanos.

Otros elementos condicionales frecuentes son `and` y `or`, cuyo significado, que coincide con el de la lógica clásica, no presenta ninguna dificultad. Su sintaxis es

```
(and <elemento-condicional>+)
(or <elemento-condicional>+)
```

Nótese que, al nivel superior de la regla, hay por defecto una conjunción de elementos condicionales, por lo que `and` aparecerá normalmente en el interior de otros elementos condicionales. Por ejemplo:

```
(defrule r1
   (hecho uno)
   (or (and (hecho tres)
            (hecho cuatro))
       (hecho cinco))
=>...)
```

se activará siempre que la memoria de trabajo contenga al menos uno de los dos conjuntos de hechos siguientes:

(hecho uno) (hecho tres) (hecho cuatro)
(hecho uno) (hecho cinco)

También disponemos del elemento condicional `not`, cuyo significado **no** coincide exactamente con el de la negación de la lógica clásica. En efecto, si no tenemos demostrado `p`, no por ello podremos decir en lógica clásica que tenemos demostrado ¬`p`. Sin embargo, esto es lo que significa el elemento `not`: `(not <elemento-condicional>)` se satisface si y sólo si `<elemento-condicional>` no se satisface. Considerando los únicos elementos condicionales que por ahora conocemos, que son simplemente hechos, ello quiere decir que `(not (h))` se satisface si y sólo si `(h)` no está en la memoria de trabajo. Por ejemplo,

```
(defrule r2
   (not (hecho uno))
   (hecho tres)
=>
   (assert (hecho cinco)))
```

se activará si la memoria de trabajo contiene únicamente `(hecho tres)`, y también si contiene además `(hecho cuatro)`, pero no se activará si contiene además `(hecho uno)`.

Nota: Por razones de implementación, si el primer elemento condicional de una regla es un `not`, se supone que antes aparece el elemento condicional `(initial-fact)`.

Una alternativa más simple y más acorde al espíritu del lenguaje es el uso de *restricciones conectoras* para describir patrones. Una restricción conectora puede ir donde iría un argumento de un hecho (es decir, en cualquier posición del patrón salvo la primera.) La sintaxis de una restricción es

```
<termino-1>|<termino-2> ... |<termino-n>
~<termino>
<termino-1>&<termino-2> ... &<termino-n>
```

La restricción `&` se satisface si se satisfacen ambas restricciones adyacentes. La restricción `|` se satisface si se satisface alguna de las dos restricciones adyacentes. Por ejemplo,

`(oav pepe peso|altura excesivo)`

se satisfará si tenemos en la memoria de trabajo `(oav pepe peso excesivo)`, o bien `(oav pepe altura excesivo)`, o bien ambos, en cuyo caso habrá dos formas de satisfacerla. Por su parte, la restricción `~c` se satisface si se satisface `c'` y `c'`$\neq$`c`. Por ejemplo,

`(oav pepe peso ~excesivo)`
se satisfará si tenemos en la memoria de trabajo `(oav pepe peso normal)`.

La restricción `~` es la de mayor precedencia, seguida de `&`, seguida de `|`. Por ejemplo,

`(oav pepe peso ~excesivo&~bajo)`
se interpreta como "que el peso de Pepe sea ni excesivo ni bajo", y no como "que el peso de Pepe no sea a la vez excesivo y no bajo".

Ejemplo 2.3 Supongamos que la memoria de trabajo contiene los siguientes hechos:

0. `(fase 2)`	3. `(oav sensor1 presion alto)`
1. `(nivel a)`	4. `(oav sensor1 estado dudoso)`
2. `(nivel c)`	5. `(oav sensor2 temperatura normal)`
	6. `(oav sensor2 estado ok)`

Veamos si se satisfacen estas partes izquierdas de reglas:
`(oav sensor1 presion alto) (nivel b)`: no.
`(oav sensor1 presion alto) (or (fase 1) (nivel b)`: sí.
`(oav sensor1|sensor2 presion alto) (nivel  b)`: sí.

```
(or (and (oav sensor1 presion alto)
         (oav sensor1 estado ok))
    (and (oav sensor2 temperatura alto)
         (oav sensor2 estado ok)))
```
: sí.
`(not (oav sensor3 estado ok))`: sí.
`(oav sensor3 estado ~ok))`: no.
`(nivel ~a))`: sí, contra lo que pueda pensar el lector poco atento. ◁

Ejercicio 2.5 Supongamos que la memoria de trabajo contiene los siguientes hechos:

0. `(fase 1)`	4. `(oav sensor2 presion alto)`
1. `(nivel b)`	5. `(oav sensor2 estado dudoso)`
2. `(oav sensor1 presion normal)`	6. `(oav sensor3 temperatura alto)`
3. `(oav sensor1 estado ok)`	

Determinar si se satisfacen estas partes izquierdas de reglas:

```
(oav sensor1 presion ~alto) (nivel ~b)
```

```
(oav sensor2 ~presion ~alto)

(oav sensor1|sensor2 presion alto) (fase ~2)

(or (oav sensor4 presion alto|normal)
    (oav sensor3 estado ~ok)
    (and (oav sensor2 estado ~ok)
         (oav sensor2 presion|temperatura alto)))
```

◁

Ejercicio 2.6 Consideremos de nuevo el problema presentado en el ejercicio 2.4, con las siguientes modificaciones: ahora se considera que las calificaciones posibles en cada examen y cada práctica son "suspenso", "aprobado", "notable" y "sobresaliente". Las reglas pasan a ser las siguientes:

Se supera la asignatura si se superan tanto la teórica como la práctica. Se supera la teórica si la calificación en cada uno de los dos exámenes teóricos ha sido distinta de "suspenso". Se supera la práctica si la calificación en cada una de las dos prácticas ha sido distinta de "suspenso", o bien si en una ha sido "suspenso" pero en otra ha sido "sobresaliente" o se ha realizado un trabajo complementario.

1. Suponiendo un solo alumno, dar un conjunto de atributos y valores adecuado para simbolizar toda esta información.

2. Representar la información anterior mediante un conjunto de reglas CLIPS, empleando un predicado "objeto-atributo-valor".

3. Escribir un programa CLIPS que calcule si el alumno ha superado la asignatura.

4. Sabemos además que las calificaciones del alumno han sido las siguientes:

T1	T2	P1	P2	TRABAJO
apr.	not.	susp.	not.	ok

   Introducir esta informacion en un `deffacts` y calcular mediante CLIPS si este alumno ha superado la asignatura.

◁

2.1.4. Consulta del programa activo.

La forma más sencilla de consultar las definiciones de hechos por defecto y reglas que han sido cargadas en el entorno CLIPS es utilizar las opciones correspondientes del menú 'Browse' (figura 2.2). Desde ellas es posible consultar o eliminar fácilmente dichas definiciones. Un uso frecuente consiste en eliminar las definiciones 'deffacts' para cargar otras nuevas, y así probar un mismo programa con distintos conjuntos de hechos iniciales.

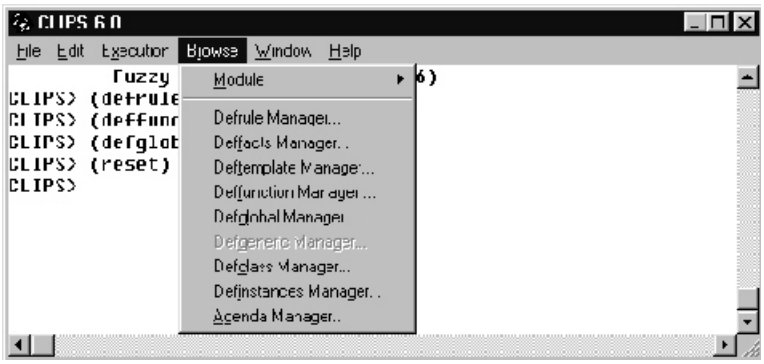

Figura 2.2: Menú *Browse*

2.2. La agenda y su ordenación.

El funcionamiento básico del motor de inferencia, como se expuso en la sección 1.2.2, consiste en ejecutar de forma cíclica estos tres procesos:

1. Selección de una de las activaciones de la agenda.
2. Disparo de la activación seleccionada.
3. Nuevo cálculo del contenido de la agenda.

Describamos ahora con algo más de detalle el algoritmo que sigue CLISP para ejecutar el paso 1 (selección de una de las activaciones de la agenda). En general, las activaciones se ordenan aplicando sucesivamente estos tres criterios (nótese que el orden en que aparecen escritas las reglas en el fichero del programa **no** es uno de ellos):

1. Prioridad o *saliencia* de la regla correspondiente a la activación.

2. Estrategia de resolución de conflictos activa.
3. Arbitrariamente (no aleatoriamente).

Primeramente se ordenan las activaciones según la prioridad o "saliencia" declarada para cada regla. Esta saliencia es un número entero entre -10000 y 10000. Si no se indica explícitamente, se supone que todas las reglas tienen la misma saliencia (igual a 0).

Nota.- Existe la tentación de declarar para cada regla una prioridad explícita diferente, convirtiendo así un programa CLIPS en algo muy parecido a un programa convencional. Para evitar esta tentación, no explicaremos cómo realizar esta declaración hasta que el lector esté más familiarizado con el estilo de programación basada en reglas.

Las activaciones con la misma saliencia (de momento, todas) se ordenan a su vez aplicando una regla de *resolución de conflictos* que toma en consideración ciertas características de la regla o de la activación.

Una de estas característica es el tiempo que la activación lleva en la agenda. Atendiendo a esto, se puede dar preferencia a la activación más reciente (estrategia `depth` o *primero en profundidad*) o, al contrario, se puede dar preferencia a la activación más antigua (estrategia `breadth` o *primero en amplitud*). La estrategia `depth` es la seguida por defecto en CLIPS.

Otra posible característica es la *especificidad* de las operaciones que aparecen en la parte izquierda de la regla. Hablando informalmente, la especificidad es el número de comparaciones que es necesario realizar con la memoria de trabajo para saber si la parte izquierda se satisface. La especificidad está definida para cualquier tipo de elemento condicional (vd. la sección 5.3.3 de [9]), pero si empleamos únicamente patrones constantes coincide precisamente con el número de estos patrones. Atendiendo a la especificidad, se puede dar preferencia a la activación menos específica (estrategia `simplicity` o *primero la más sencilla*) o, al contrario, se puede dar preferencia a la activación más específica (estrategia `complexity` o *primero la más compleja*).

Otra característica que se puede tomar en consideración es la antigüedad, no de la activación, sino de los hechos que la apoyan. Atendiendo a esta característica se definen las estrategias `LEX` y `MEA` (vd. las sección 5.3.5 y 5.3.6 de [9])

Por último, la estrategia `random` establece una ordenación aleatoria, pero repetible en sucesivas sesiones.

Para las reglas que tienen la misma prioridad y no pueden ordenarse en base a la estrategia seleccionada, el lenguaje no establece orden alguno. Es decir, se ordenan arbitrariamente según algún criterio no especificado y que puede cambiar en diversas implementaciones del lenguaje.

La estrategia de control activa puede modificarse desde la opción "Options" del menú "Execution", o bien utilizando las siguientes órdenes (nótese que las órdenes no pueden aparecer directamente al nivel superior de un fichero de construcciones, pero sí en la parte derecha de las reglas, en la ventana de interacción o en un fichero con extensión `.bat`):

`(get-strategy)` : devuelve el valor de la estrategia actual

`(set-strategy depth|breadth|random|lex|mea|simplicity|complexity)`

Cada vez que se cambia la estrategia, la agenda se reordena adecuadamente.

El paso 2 del ciclo de evaluación del programa es el llamado *disparo* de la activación seleccionada, que será la primera de la agenda. Este disparo, en general, causará modificaciones en la memoria de trabajo.

Por ello, el paso 3 del ciclo de evaluación es la actualización de la agenda. En él se determinan qué reglas están *activas* (es decir, se satisfacen) en la nueva configuración de la memoria de trabajo. Podría pensarse que este cálculo, que ha de repetirse en cada paso del motor de inferencias, consume unos recursos temporales excesivos, de forma que la ejecución del programa es inaceptablemente lenta. Sin embargo, el empleo de astutos algoritmos (como el algoritmo Rete, [13]) hace que en la mayoría de los casos reales el recálculo de la agenda se lleve a cabo de forma muy rápida. Las ideas básicas son: (i) preprocesar el conjunto de reglas y generar el llamado *grafo Rete*, que contiene sin repeticiones y de forma ordenada todos los elementos condicionales que aparecen en las partes izquierdas de las reglas; (ii) empleando este grafo, recalcular en cada paso del motor únicamente aquellos elementos condicionales afectados por la aserción o borrado de nuevos hechos.

Ejemplo 2.4 Sea el siguiente programa y supongamos ejecutado `(reset)`:

```
(deffacts f1
  (a) (b))

(defrule r1
  (a) => (assert (x)))

(defrule r2
  (a)(b) => (assert (y)))

(defrule r3
  (a)(x) => (assert (z)))
```

La agenda contendrá dos activaciones: a_1, correspondiente a `r1` y apoyada por `(a)`, y a_2, correspondiente a `r2` y apoyada por `(a)` y `(b)`. Si la estrategia es primero en profundidad o en amplitud, ambas activaciones están empatadas (son igualmente antiguas); se disparará una de ellas, dependiendo de la implementación. Si se ha dicho que la estrategia sea `simplicity`, la activación disparada

será siempre a_1. Si se ha dicho que la estrategia sea `complexity`, la activación disparada será siempre a_2.

Supongamos ahora que se ha disparado `r1` y la memoria de trabajo contiene `(a)`, `(b)` y `(x)` y la agenda contiene a_2 (aparecida en el paso anterior) y una nueva activación a_3 correspondiente a `r3` y apoyada por `(a)` y `(x)`. Si la estrategia es primero en profundidad, a_3 es prioritaria y será la disparada. Por el contrario, si la estrategia es amplitud, a_2 es prioritaria y será la que se dispare. ◁

Abriendo la ventana *Agenda* del entorno (menú *Window*) es posible visualizar en cada momento las reglas activadas así como su ordenación. Además, para cada activación se muestran los hechos que la sustentan.

Las reglas en CLIPS son "refractarias", es decir, cuando se dispara una regla el motor de inferencia guarda los índices de los hechos que la activaron, y ya no vuelve a activarla nunca con los hechos de ese mismo índice. (Nota: este comportamiento puede alterarse por medio de la función `(refresh <nom-regla>)`, que añade de nuevo una regla disparada a la agenda, siempre que los hechos que la activaron sigan estando en la lista de hechos. No emplearemos en nuestros ejemplos esta posibilidad.)

2.3. Plantillas.

2.3.1. Definición de plantillas.

Normalmente, los programas CLIPS utilizan diversos tipos de hechos para guardar distintos tipos de datos. Además de los hechos ordenados básicos (sección 2.1.1), pueden definirse tipos de hechos más complejos usando una *plantilla* o *template*, en inglés. En la plantilla se describen el número de campos (o datos básicos) del hecho, así como su nombre, tipo y otras restricciones. La sintaxis es la siguiente:

```
(deftemplate <nom-relación>
   [<comentario>]
   ([slot|multislot] <nom-slot> atributo*)*)
```

Cada tipo de hecho tiene un nombre de relación, y puede contener cero, uno o más campos (denominados en CLIPS *ranuras*, del inglés *slots*), identificados por un nombre.

Cada ranura puede ser:
- Sencilla (*slot*): contiene obligatoriamente un único dato.
- Múltiple (*multislot*): puede contener cero, uno o más datos.

Por ejemplo:

```
; un ejemplo de deftemplate
(deftemplate persona
   "datos de una persona"
   (slot nombre)                    ;nombre de la persona
   (slot edad)                      ;edad de la persona
   (slot e-civil)                   ;estado civil de la persona
   (multislot estudios))            ;estudios realizados
```

Como puede verse en este ejemplo, además de emplear comentarios señalados por `;`, todas las definiciones CLIPS pueden documentarse incluyendo tras su nombre una cadena de caracteres.

Una vez definida una plantilla para un tipo de hechos, es posible añadir o borrar hechos de dicho tipo mediante `deffacts`, `assert` y `retract`. Por ejemplo, si suponemos ejecutado el `deftemplate` anterior, podremos ejecutar la siguiente secuencia de instrucciones:

```
> (deffacts personas-1
     (persona (nombre luci) (edad 15)
        (e-civil soltero) (estudios egb))
     (persona (nombre pepe)(e-civil soltero)
        (estudios) (edad 23))
     (persona (edad 44) (e-civil casado) (nombre mari)
        (estudios derecho economicas)))
> (reset)
> (retract 3)
> (assert (persona (nombre juan) (edad 75)
             (e-civil viudo) (estudios ats)))
<Fact-4>
```

y el contenido de la memoria de trabajo será

```
0. (initial-fact)
1. (persona (nombre luci) (edad 15) (e-civil soltero) (estudios egb))
2. (persona (nombre pepe)(e-civil soltero) (estudios) (edad 23))
4. (persona (nombre juan) (edad 75) (e-civil viudo) (estudios ats))
```

Nótese que, al indicar el contenido de uno de estos hechos, es irrelevante el orden en que aparezcan sus campos. Por esta razón, los hechos definidos a partir de una plantilla se suelen denominar *hechos desordenados*.

Además de añadir o borrar un hecho, tenemos la posibilidad de *modificar* un hecho mediante

`(modify <índice-de-hecho>|<direccion-de-hecho>`
`(<nom-slot><valor-slot>)+)`: modifica los slots especificados del hecho referenciado. En realidad, el intérprete destruye el hecho anterior y añade uno nuevo con las modificaciones. la función `modify` devuelve la dirección del nuevo hecho. Por ejemplo, continuando la sesión anterior, podríamos tener

```
> (modify 4 (edad 85) (e-civil casado))
<Fact-5>
> (modify 1 (nombre lucia))
<Fact-6>
```

y el contenido de la memoria de trabajo será

```
0. (initial-fact)
2. (persona (nombre pepe)(e-civil soltero) (estudios) (edad 23))
5. (persona (nombre juan) (edad 85) (e-civil casado) (estudios ats))
6. (persona (nombre lucia) (edad 15) (e-civil soltero) (estudios egb))
```

Alternativamente, el lugar de `modify` podemos emplear `duplicate`:

`(duplicate <índice-de-hecho>|<direccion-de-hecho>`
`(<nom-slot><valor-slot>)+)`: funciona igual que `modify`, pero conserva en la lista tanto el hecho original como el nuevo. Por ejemplo, si ahora decimos

```
> (duplicate 2 (nombre paco))
<Fact-7>
```

tendremos en la memoria de trabajo

```
0. (initial-fact)
2. (persona (nombre pepe)(e-civil soltero) (estudios) (edad 23))
5. (persona (nombre juan) (edad 85) (e-civil casado) (estudios ats))
6. (persona (nombre lucia) (edad 15) (e-civil soltero) (estudios egb))
7. (persona (nombre paco)(e-civil soltero) (estudios) (edad 23))
```

Nota: recuérdese lo dicho en la sección 2.1.1 al hablar de los índices y direcciones de hechos. La forma más natural y habitual de emplear `modify` y `duplicate` será escribirlos en la parte derecha de una regla, teniendo como argumento una variable que en la parte izquierda se habrá ligado a una dirección de hecho.

Recuérdese que los elementos condicionales básicos que pueden aparecer en la parte izquierda de una regla son los patrones. Los patrones pueden corresponder a hechos desordenados presentes en la memoria de trabajo, teniendo en cuenta que si en el patrón no figura una ranura, ello quiere decir que no restingimos su valor, por lo cual no será relevante a la hora de calcular la correspondencia. Por ejemplo, supongamos el estado de la memoria de trabajo descrito más arriba, y sea una regla de la forma

```
(defrule prueba-slots
  (persona (e-civil soltero))
=>
  ...)
```

El patrón de la parte izquierda estará en correspondencia con los hechos 2, 6 y 7, por lo que habrá tres activaciones de la regla en la agenda.

Por otra parte, si la regla fuera de la forma

```
(defrule prueba-slots
  (persona (e-civil soltero) (estudios))
=>
  ...)
```

solamente los hechos 2 y 7 estarían en correspondencia con él; y si fuera

```
(defrule prueba-slots
  (persona (e-civil soltero) (edad 34) (estudios))
=>
  ...)
```

ya no habría ningún hecho en correspondencia con el patrón.

2.3.2. Ranuras y atributos.

En principio, el valor de una ranura o campo podría ser cualquiera. Las definiciones de plantilla, sin embargo, permiten asociar a cada campo cero, uno,

o más atributos (en la terminología clásica de los marcos, "facetas") para establecer determinadas restricciones sobre los valores que se pueden almacenar en él, y también permiten asignarle valores por defecto. Estos atributos son los siguientes:

Atributo `type` (tipo).

> `(type <tipo-permitido>+)`: permite limitar los valores de un slot a los tipos especificados. Olvidándonos de hechos e instancias, los tipos básicos que maneja CLIPS son `INTEGER` y `FLOAT` (números enteros y en coma flotante, ambos subtipos de `NUMBER`), `SYMBOL` (símbolo) y `STRING` (cadena de caracteres) Estos dos últimos son subtipos de `LEXEME`. Los enteros y flotantes se escriben con las convenciones habituales, p. ej., `2`, `34`, `-5`, `3.5e10`, `4.89`. Los símbolos son similares a los de Lisp, pero sus nombres son sensibles al uso de mayúsculas o minúsculas, es decir, `HOLA` no es igual a `hola`. Las cadenas se escriben entre comillas dobles, y el carácter de escape es '\', p. ej., `"Una cadena de CARACTERES"`, `"Una cadena con \"comillas\""`. De esta forma, al definir
>
> ```
> (deftemplate t1 (slot s1 (type FLOAT)))
> ```
>
> se está diciendo que el valor de `s1` sólo puede ser un número en coma flotante, y al decir
>
> ```
> (deftemplate t1 (slot s1 (type INTEGER LEXEME)))
> ```
>
> se está diciendo que el valor de `s1` puede ser un entero, un símbolo o una cadena de caracteres.

Atributo `allowed-values` (valores permitidos).

> `(allowed-values <valor>*)` nos permite definir un tipo enumerado: los únicos valores permitidos para el slot serán los indicados. Por ejemplo, al definir
>
> ```
> (deftemplate t1 (slot s1 (allowed-values 1 2 3)))
> ```
>
> se está diciendo que el valor de `s1` ha de ser uno de los enteros 1, 2 o 3.

Atributos de constantes permitidas.

> `(allowed-<tipo>s <valor>*)+`, donde `<tipo>` es uno de los anteriormente citados. Es una versión más fina de lo anterior, ya que restringe solamente los valores del `<tipo>` indicado. Por ejemplo, al definir
>
> ```
> (deftemplate t1 (slot s1 (allowed-symbols pepe juan)))
> ```

se está diciendo que el valor de `s1`, en caso de que sea un símbolo, ha de ser `pepe` o `juan` (pero puede tomar cualquier valor, caso de no ser un símbolo.)

Atributo `range` (rango).

`(range <límite-inferior><límite-superior>)` Sólo disponible para slots numéricos. Si no existe alguno de los dos límites puede utilizarse como valor el símbolo `?VARIABLE`.

Atributo `cardinality` (cardinalidad).

`(cardinality <límite-inferior><límite-superior>)` Sólo disponible para multislots. Indica el número mínimo y máximo de valores que puede contener. Si no se desea establecer alguno de los dos puede utilizarse como valor el símbolo `?VARIABLE`.

Atributo `default` (valor por defecto).

`(default <valor-por-defecto>)` El valor por defecto puede ser una expresión cualquiera o bien los símbolos `?NONE` (en cuyo caso el valor debe proporcionarse obligatoriamente cuando se aserten los hechos correspondientes) o `?DERIVE` (en cuyo caso el intérprete calculará un valor válido arbitrario, pero no aleatorio). Si no se especifica un valor por defecto para un slot, CLIPS supone `(default ?DERIVE)`.

Atributo `default-dynamic`.

`(default-dynamic <expresión>)`. El atributo `default` evalúa la expresión proporcionada cuando se ejecuta el `deffacts`, de forma que todas las veces que se aserta un hecho se considera el mismo valor por defecto. Por el contrario, el atributo `default-dynamic` evalúa la expresión en el momento de asertar cada hecho, de forma que el valor por defecto puede variar a lo largo de la ejecución del programa.

Nota.- CLIPS realiza por defecto comprobaciones de tipo estáticas (es decir, en el momento de leer las expresiones correspondientes), pero también puede realizarlas de forma dinámica (es decir, en tiempo de ejecución). Este comportamiento puede manipularse explícitamente utilizando las funciones

```
(get-static-constraint-checking)
(set-static-constraint-checking <valor>)
(get-dynamic-constraint-checking)
(set-dynamic-constraint-checking <valor>)
```

donde `<valor>` puede ser una las constantes `TRUE` (para activar el control de tipos) o `FALSE` (para desactivarlo). También es posible modificar estos valores utilizando el menú "Execution/Options".

Ejemplo 2.5 Supongamos que queremos escribir una construcción `deftemplate` para recoger los datos de un médico. Una posibilidad podría ser la siguiente:

```
(deftemplate medico  "datos de un medico"
   (slot nombre (type STRING) (default ?NONE))
   (slot edad (type INTEGER)(range 0 ?VARIABLE))
   (slot sexo (allowed-values hombre mujer))
   (multislot guardias (type INTEGER) (range 1 31)
      (cardinality 0 4)))
```

El nombre del médico será una cadena de caracteres, y no será posible crear un hecho de tipo 'medico' sin proporcionar el nombre. La edad será un número entero no negativo. El slot `sexo` sólo podrá contener los valores 'hombre' o 'mujer', nótese que no se admiten valores como 'HOMBRE', 'MUJER', 'Hombre' o 'Mujer'. Por último, 'guardias' es un multislot que podrá contener entre cero y cuatro valores enteros comprendidos entre el 1 y el 31. ◁

Ejercicio 2.7 Introducir la definición anterior en el intérprete y a continuación evaluar las siguientes órdenes:

```
> (assert (medico))
> (assert (medico (nombre "Pepe") (edad -5))
> (assert (medico (nombre "María") (guardias 7 10 20 24 28)))
```

Las tres incumplen alguna condición sobre los valores permitidos en los campos definidos en la plantilla medico. Analizar los mensajes de error producidos. ◁

Ejercicio 2.8 Consideremos de nuevo el problema presentado en el ejercicio 2.6. Recuérdese que un alumno habrá realizado dos exámenes y dos prácticas, con las posibles calificaciones "suspenso", "aprobado", "notable" y "sobresaliente"; y también es posible que haya realizado un trabajo complementario. Escribir un `deftemplate` adecuado para representar la información inicial de un alumno, y escribir de nuevo el correspondiente programa CLIPS.

Nota.- Los valores calculados por el programa no formarán parte de la plantilla, sino que se almacenarán en hechos ordenados `(oav practica <valor>)`, `(oav teoria <valor>)` y `(oav asignatura <valor>)`. ◁

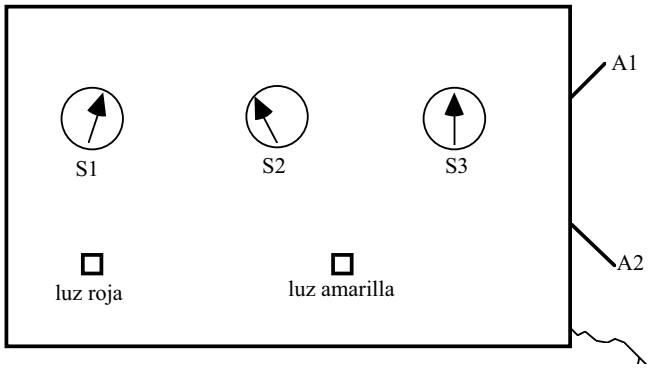

Figura 2.3: La máquina STT.

Ejercicio 2.9 La superturbotostadora o STT (vd. figura 2.3) es una compleja máquina que vale varios millones de euromortadelos. Su funcionamiento correcto es esencial para la buena marcha de la fábrica.

La experta operaria María Pérez (para los amigos, Mari) es la encargada de controlar el funcionamiento de la STT, actuando sobre las palancas $A1$ y $A2$ y la conexión a la red eléctrica.

Concepción López (para los amigos, Cuqui) es una hábil ingeniera del conocimiento encargada de implementar un sistema experto capaz de controlar la STT como lo hace Mari. Para ello ha mantenido con ella la siguiente entrevista:
C.– Observo que a veces usted apaga la STT. Es una medida extrema, ¿no?
M.– Sí. Apago la STT cuando la temperatura y la presión son demasiado elevadas.
C.– ¿Y cómo sabe cuándo la temperatura es demasiado elevada?
M.– Para eso está el sensor S2. Cuando está hacia la izquierda, sé que la temperatura es excesiva.
C.– ¿Y la presión?
M.– Compruebo si el sensor S1 está indicando a la derecha. Cuando ocurre esto, la presión aún no es muy elevada.
C.– Creo que está claro. Otras veces he observado que usted actúa sobre la palanca A1.
M.– Sí, lo hago para bajar la presión. Cuando la presión es alta, acciono A1.
C.– ¿Pero no hemos dicho que lo que hacemos es apagar la STT?

M.– No, eso lo hacemos cuando son a la vez excesivas la presión y la temperatura.
C.– Ya entiendo. Entonces, cuando sólo la presión es excesiva, no hace falta apagarla.
M.– Eso es.
C.– Entonces, se actúa sobre A1 sólo cuando S1 apunta a la izquierda, ¿no?.
M.– Sí... Bueno, cuando S1 está casi vertical, considero que la presión es un poco elevada y acciono la palanca A2.
C.– Entonces A2 sólo se toca en este caso, cuando S1 está casi vertical.
M.– No. A2 también me sirve para controlar la temperatura. Cuando es excesiva, la acciono.
C.– ¿Y para qué sirve la luz amarilla?
M.– Se me olvidaba: siempre que la luz amarilla está encendida, la presión debe considerarse demasiado alta.
C.– Queda por saber qué pasa cuando se enciende la luz roja.
M.– Nada. La verdad es que nunca la miro, ni tampoco el sensor S3.
Se pide:

1. Definir las plantillas adecuadas para representar sensores, luces y acciones, y plasmar el conocimiento extraído en la anterior entrevista en un conjunto de reglas CLIPS.

2. Ejecutar el programa CLIPS con los hechos iniciales correspondientes a la situación en que los sensores S1 y S3 apuntan completamente a la derecha, el S2 completamente a la izquierda y las luces están apagadas.

◁

2.4. Carga y ejecución de programas CLIPS.

A lo largo de estas páginas estamos suponiendo que el lector interactúa con CLIPS a través de un entorno integrado de ventanas (lo que en la jerga informática se suele llamar un IDE o "Interactive Development Environment"). En este entorno hemos introducido código de tres formas distintas:
(a) Pulsando opciones de los diversos menús.
(b) Escribiendo el código en la ventana de interacción.
(c) Cargando un fichero que contiene construcciones (extensión .clp) (orden `load`).

Aún hay otras dos posibilidades de introducir elementos CLIPS en el entorno:

(d) Cargando un fichero "batch" (extensión .bat) (orden `batch`).

(e) Cargando un fichero binario (orden `bload`).

Conviene aclarar que **no** todas estos métodos son equivalentes; no todos ellos permiten introducir el mismo código, ni tienen las mismas consecuencias. Para explicar este punto debemos introducir una distinción entre *construcciones* ("constructs"), de un lado, y *órdenes* ("commands") y *funciones* ("functions"), de otro.

Los nombres de las construcciones comienzan siempre por la cadena `def`. Hasta ahora hemos mencionado `deffacts`, `defrule` y `deftemplate`. Hay otras que veremos más adelante: `defmodule`, `defglobal` y `deffunction`. Y, por último, hay construcciones relacionadas con la programación orientada a objetos en CLIPS: `defclass`, `definstances`, `defmessage-handler`, `defgeneric` y `defmethod`. Conceptualmente, una llamada a una construcción no es parte de la ejecución del programa, sino una modificación del programa, es decir, de la "base de conocimientos" (hechos iniciales y reglas).

Por el contrario, una llamada a una función es lo que constituye un paso en el disparo de una regla y, por tanto, en la ejecución de un programa. Normalmente cada llamada devolverá un valor, aunque también puede producir un efecto colateral, por ejemplo sobre la lista de hechos (`assert`, etc.)

Las opciones (b) y (d) son las las más flexibles. Tanto desde la ventana de interacción como desde un fichero "batch" se puede introducir cualquier elemento, sea construcción o función. Recomendamos al lector que lo compruebe, escribiendo en la ventana una sucesión cualquiera de llamadas a `defrule`, `assert` y `retract`, y visualizando en cada paso el contenido de la agenda y la lista de hechos.

La opción (c) permite cargar ficheros que contengan solamente construcciones. Por tanto, si el fichero "kk.clp" tiene este contenido:

```
(set-strategy random)
(deffacts f1 (a 1))
(defrule r1 (a 1) => (assert (b 1)))
(reset)
(run)
```

la llamada a `(load "kk")` producirá varios errores `Expected the beginning of a construct`, ya que ni `(set-strategy random)` ni `(reset)` ni `(run)` son construcciones. Por el contrario, la orden `(batch "kk.clp")` no produciría ningún error.

Sin embargo, no es buen estilo emplear la orden `batch` para cargar construcciones. La manera ortodoxa de almacenar el código CLIPS mostrado más arriba

sería dividirlo en un fichero con las órdenes y otro con las construcciones:

```
------------------FICHERO KK.BAT-----------------
(set-strategy random)
(load "kk.clp")
(reset)
(run)
------------------FICHERO KK.CLP-----------------
(deffacts f1 (a 1))
(defrule r1 (a 1) => (assert (b 1)))
-------------------------------------------------
```

Es posible cargar el intérprete CLIPS sin el entorno de ventanas, invocando desde el SO la orden `clips`. Aún en este caso tendremos acceso a la lista de hechos, la agenda y demás elementos que en el entorno se visualizan, llamando a las adecuadas órdenes o funciones. La orden `clips` tiene la sintaxis

`clips [-f <nom-fichero> | -l <nom-fichero>]*`

con el siguiente significado:

`-f` carga un fichero con la orden `batch`.

`-l` carga un fichero con la orden `load`.

En particular, la última orden que figure en el fichero puede ser `(exit)`, que finaliza la ejecución del intérprete. De esta sencilla manera puede conseguirse ejecutar automáticamente un programa CLIPS y volver al SO.

En el ejemplo anterior:

```
------------------FICHERO KK.BAT-----------------
(set-strategy random)
(load "kk.clp")
(reset)
(run)
(exit)
------------------FICHERO KK.CLP-----------------
(deffacts f1 (a 1))
(defrule r1 (a 1) => (assert (b 1)))
-------------------------------------------------
```

Por último, mencionemos la posibilidad de preprocesar un programa CLIPS, generando un fichero binario apto para ser procesado por el intérprete. Ello se consigue con la orden

`(bsave <nom-fichero>)`

Esta orden almacena en el fichero indicado todas las construcciones presentes en el entorno actual. El formato es tal que posteriormente se podrán cargar estas construcciones con la orden

`(bload <nom-fichero>)`

con mayor rapidez que si se cargaran de un fichero de texto (extensión .clp) con la orden `load`.

Nota.– Las restricciones asociadas a las ranuras no se almacenarán, a menos que se haya activado la opción `set-dynamic-constraint-checking`.

Recomendamos al lector que ejecute esta sucesión de órdenes, comprobando en cada paso el estado de la lista de hechos y de la agenda:

```
(clear)
(deffacts f1 (a 1))
(defrule r1 (a 1) => (assert (b 1)))
(reset)
(run)
(bsave "kk.bin")
(clear)
(bload "kk.bin")
(run)
(reset)
(run)
```

Tras ejecutar `bload`, ya no es posible añadir nuevas construcciones, pues el programa ha sido precompilado y no admite ser modificado. Un posible continuación de la sesión anterior con el intérprete sería:

```
> (defrule r2 (b 1) => (assert (c 1)))
Error: Cannot load defrule construct whith binary load in effect.
> (clear)
> (defrule r2 (b 1) => (assert (c 1)))
> (assert (b 1))
Fact-0
> (run)
> (facts)
Fact-0 (b 1)
Fact-1 (c 1)
For a total of 2 facts.
```

Capítulo 3

UN EJEMPLO DE AGENTE REACTIVO.

No hemos hecho más que empezar nuestro estudio de los lenguajes basados en reglas; sin embargo, ya estamos en condiciones de abordar algunos problemas relativamente interesantes. Concretamente, programaremos un agente autónomo reactivo.

Un *agente* es un ente que por medio de *sensores* percibe ciertas características del medio que le rodea, y tras cierto procesamiento lleva a cabo ciertas acciones en dicho medio (modificando quizás su estado interno.)

Los agentes más sencillos son los denominados *agentes reactivos* (figura 3.1). Estos agentes carecen de un modelo simbólico interno, y actúan simplemente ligando ciertas combinaciones de estímulos a ciertas respuestas. Por tanto, los programas basados en reglas son especialmente adecuados para describir la conducta de estos agentes.

A continuación programaremos la conducta de un agente reactivo bastante simple.

3.1. El robot SLID.

SLID es un robot móvil experimental desarrollado en la Universidad de Agalam. SLID es capaz de desplazarse por una carretera tal como muestra la figura 3.2. La carretera, cuyo trazado es desconocido para el robot, tiene una anchura constante de 9 metros y puede discurrir recta, o bien zigzaguear a derecha o

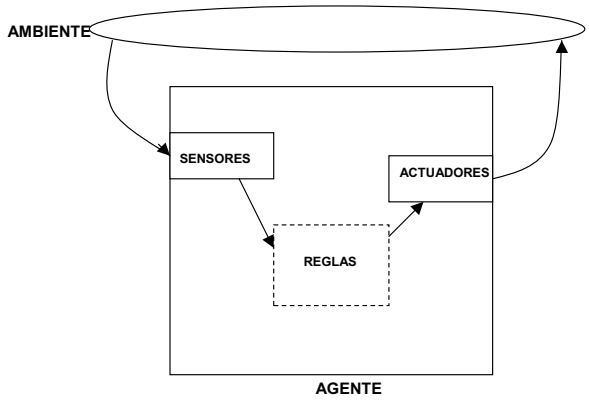

Figura 3.1: Esquema de un agente reactivo

izquierda. El zigzag se produce exactamente en diagonal, avanzando o retrocediendo el trazado de la carretera una unidad en el sentido del eje X , por cada unidad avanzada en el del eje Y.

SLID está dotado de dos sensores laterales situados uno a cada lado del robot. Estos sensores se activan cuando el margen correspondiente de la carretera se encuentra a una distancia inferior o igual a 3 metros, y se desactivan únicamente cuando la distancia es superior a 3 metros. La lectura de estos sensores no es fiable cuando el móvil se sale de la carretera. Una vez en marcha, el robot se desplaza de forma inevitable e inexorable a una velocidad constante de un metro por segundo en la dirección del eje Y. No obstante, el robot dispone de dos impulsores que le permiten desplazarse a voluntad un metro a derecha o izquierda por cada metro de avance. Es decir, si la posición del robot en un instante dado es la indicada por p1 en la figura 3.3, un segundo después su posición podrá ser únicamente:

–p2: si decidió desplazarse a la derecha (según el sentido de la marcha).
–p3: si decidió no realizar ninguna acción.
–p4: si decidió desplazarse a la izquierda.

El robot parte siempre de una posición inmediatamente adyacente al margen derecho de la carretera (según el sentido de la marcha). Su objetivo es avanzar durante un tiempo ilimitado sin salirse de los márgenes marcados por la carretera.

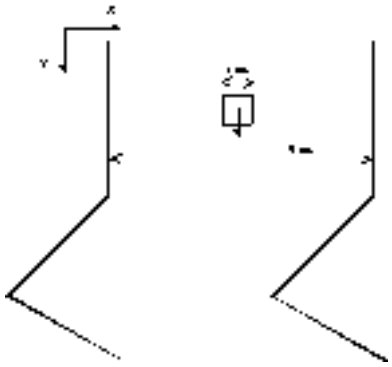

Figura 3.2: El entorno del robot SLID

3.2. El simulador S-SLID.

S-SLID es un simulador del robot SLID y de su medio natural. Una vez cargado el fichero `sim-c1.clp`, es posible definir un conjunto de construcciones CLIPS capaces de decidir las acciones que el robot debe llevar a cabo para conseguir su objetivo. En definitiva se trata de definir un conjunto de reglas que describan un comportamiento racional para el robot. El funcionamiento del simulador es cíclico. En cada iteración de la simulación se realizan los siguientes pasos:

–Se simula la acción del robot (avance y, posiblemente, desplazamiento a izquierda o derecha).

–Se simula la medición de los sensores en función del entorno simulado.

–Se evalúan las reglas del programa que controla el robot en un entorno donde es conocido el estado de los sensores. El resultado debe ser la selección de una (o ninguna) acción.

La medida de los sensores se simula añadiendo a la lista de hechos un hecho del tipo 'sensor-lateral´:

```
(deftemplate sensor-lateral
  (slot dcha  (allowed-values TRUE FALSE))    ;sensor de la dcha.
  (slot izqda (allowed-values TRUE FALSE)))   ;sensor de la izqda.
```

El valor TRUE denota que el sensor correspondiente está activado, y FALSE desactivado.

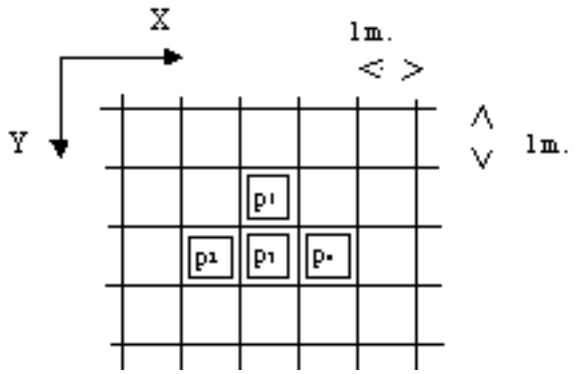

Figura 3.3: Las acciones del robot SLID

Para simular la ejecución de las acciones, las reglas que controlan el robot deben generar hechos del tipoa `accion`:

```
(deftemplate accion
  (slot tipo (allowed-values desplaz-dcha desplaz-izqda)))
```

El valor `desplaz-dcha` denota que el robot debe realizar un desplazamiento a la derecha, y `desplaz-izqda` a la izquierda.

Para que el simulador funcione correctamente, las construcciones CLIPS que controlan el robot (`deftemplate`, `defrule`, etc.) deben tener nombres que comiencen con el símbolo `AGENTE::`, por ejemplo:

```
(defrule AGENTE::regla1
  (hecho 1)
  (hecho 2)
  ...
=>
  (assert (accion (tipo..)))


(defrule AGENTE::regla2
  (hecho 3)
  (hecho 4)
  ...
```

```
=>
  (assert (accion (tipo..)))
```

Para probar el simulador puede realizarse la siguiente secuencia:

1. Limpiar el entorno CLIPS de otros programas: `(clear)`.
2. Cargar el simulador: `(load "s-slid.clp")`.
3. Cargar el programa CLIPS que controla el robot: `(load <nom-fich>)`
4. Evaluar `(reset)` y `(run)`.

Es posible realizar nuevas simulaciones evaluando sucesivamente `(reset)` y `(run)`. Si se omite el paso 3 (cargar el programa que controla el robot) puede observarse que este no realizará ninguna acción.

3.3. Un programa para controlar el robot SLID.

Es fácil escribir un programa para controlar el robot SLID:

–Cuando el sensor derecho esté activado, será preciso desplazar el robot a la izquierda.

–Cuando el sensor izquierdo esté activado, será preciso desplazar el robot a la derecha.

–En otro caso, el robot no necesita realizar ninguna acción.

Las siguientes reglas CLIPS describen perfectamente el comportamiento anterior:

```
(defrule AGENTE::alejarse-del-margen-derecho
  "si sensor derecho está activado, moverse hacia la izquierda"
  (sensor-lateral (dcha TRUE))
=>
  (assert (accion (tipo desplaz-izqda))))

(defrule AGENTE::alejarse-del-margen-izquierdo
  "si sensor izquierdo está activado, moverse hacia la derecha"
  (sensor-lateral (izqda TRUE))
=>
  (assert (accion (tipo desplaz-dcha))))
```

Nótese que el programa describe un comportamiento completamente reactivo, es decir, cada regla establece la acción adecuada para un conjunto de percepciones del medio. No es necesario almacenar el estado pasado del mundo ni razonar sobre su evolución en el futuro. El programa anterior se encuentra en el

fichero "slid1.clp". Puede probarse su resultado en el simulador con la siguiente secuencia de órdenes:

```
>(clear)
>(load "s-slid.clp")
>(load "slid1.clp")
>(reset)
>(run)
```

Ejercicio 3.1 Modificar el programa que describe el comportamiento del robot SLID eliminando la regla que permite desplazarse a la derecha. Realizar varios experimentos y analizar el comportamiento resultante.

Repetir el experimento, pero eliminando esta vez la regla que permite desplazarse a la derecha. ◁

Ejercicio 3.2 Los científicos de la Universidad de Agalam, presionados por dificultades presupuestarias, han modificado el diseño del robot SLID, eliminando el sensor lateral de la izquierda. El robot resultante se ha denominado SLD. Construir un programa que describa un comportamiento reactivo que garantice que SLD se mantendrá dentro de los márgenes de la carretera en base únicamente a la información recibida del sensor derecho. El simulador del nuevo robot SLD se encuentra en el fichero "s-sld.clp". ◁

Ejercicio 3.3 Los científicos de la Universidad de Agalam han recibido un nuevo encargo. Esta vez se trata de diseñar un robot móvil dotado de nuevos sensores y que se desenvuelve en un nuevo medio:

–El medio: esta vez la carretera discurre siempre recta, pero puede estar plagada de obstáculos. Los obstáculos son siempre lineales, perpendiculares al sentido de la carretera y tienen siempre 3 metros de ancho. Pueden estar situados a una distancia de exáctamente 0, 1, 2, 3, 4, 5, 6, 7 u 8 metros del margen derecho de la carretera. La posición y distancia exacta entre obstáculos es desconocida, pero se sabe que los obstáculos están distanciados entre sí al menos 5 metros en el eje Y.

–Las acciones: las acciones del nuevo robot son idénticas a las del robot SLID.

–Los sensores: además de los sensores laterales a derecha e izquierda ya utilizados por SLID, el nuevo robot incorpora un nuevo sensor frontal triple, capaz de detectar obstáculos justo en frente del robot o bien un metro a su derecha o izquierda (figura 3.4). El alcance del sensor es de 4 metros. Este

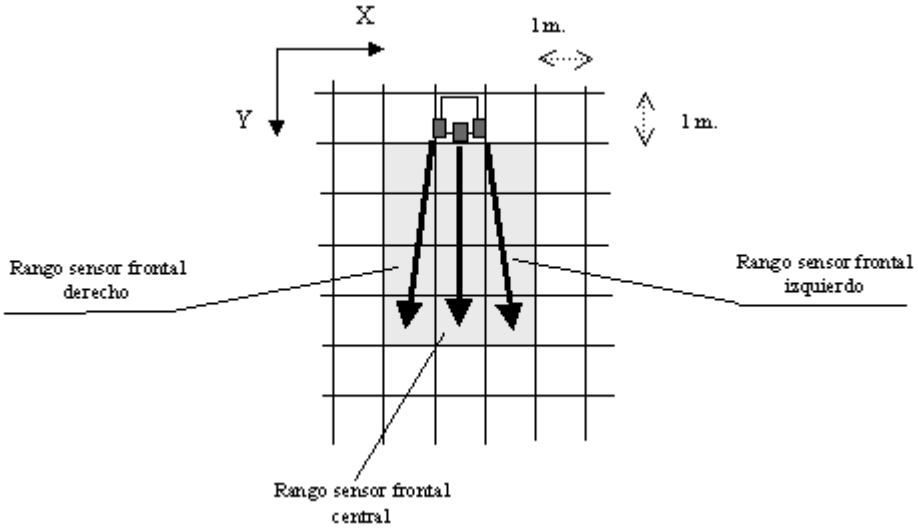

Figura 3.4: Sensores del robot SLID-2

nuevo sensor se simula con un hecho del tipo `sensor-frontal`, que sí es capaz de medir la distancia del obstáculo:

```
(deftemplate sensor-frontal
(slot dist   (type INTEGER) (range 0 4))  ;distancia del obstáculo
(slot centro (allowed-values TRUE FALSE)) ;medición central
(slot dcha   (allowed-values TRUE FALSE)) ;medición a la izquierda
(slot izqda  (allowed-values TRUE FALSE)));medición a la derecha
```

Algunos ejemplos ilustrativos de funcionamiento aparecen en las figuras 3.5, 3.6, 3.7, 3.8 y 3.9. En las figuras 3.5 y 3.6 el obstáculo está fuera del alcance de los sensores (demasiado al frente, y demasiado a su izquierda respectivamente), por lo que su lectura no registra ningún obstáculo:

```
(sensor-frontal (dist 0) (dcha FALSE)
     (centro FALSE) (izqda FALSE))
```

En la figura 3.7 el obstáculo está dentro del alcance de los sensores, por lo que su lectura registra su posición y distancia:

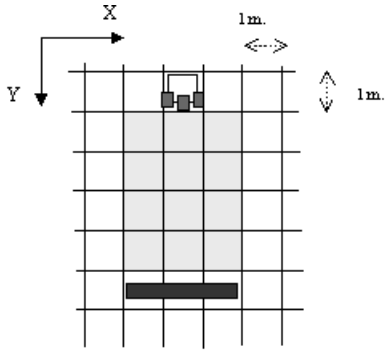

Figura 3.5:

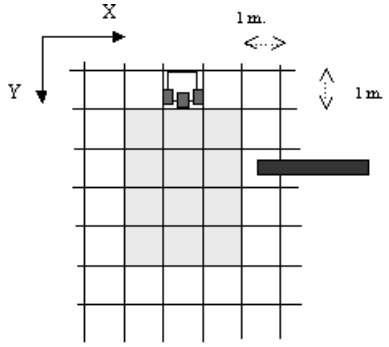

Figura 3.6:

```
(sensor-frontal (dist 3) (dcha TRUE)
     (centro TRUE) (izda TRUE))
```

En la figura 3.8 el obstáculo está dentro del alcance de los sensores, por lo que su lectura registra su posición y distancia. Nótese que el sensor frontal izquierdo no registra el obstáculo, al estar este desplazado a la derecha respecto al robot:

```
(sensor-frontal (dist 2) (dcha TRUE)
     (centro TRUE) (izda FALSE))
```

Por último, en la figura 3.9 el obstáculo está dentro del alcance de los sensores, por lo que su lectura registra su posición y distancia. Nótese que el único sensor

activo es el izquierdo:

```
(sensor-frontal (dist 3) (dcha FALSE)
     (centro FALSE) (izda TRUE))
```

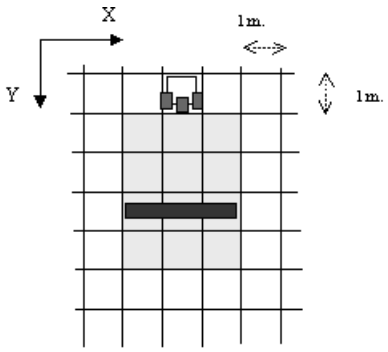

Figura 3.7:

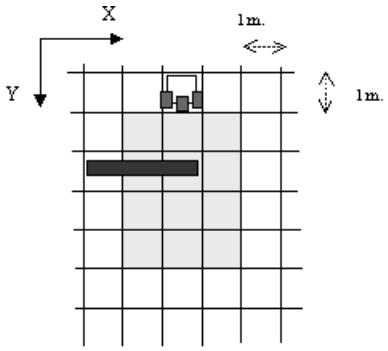

Figura 3.8:

Existe un simulador del nuevo robot, en el fichero "s-slidf.clp". Con su ayuda, diseñar un programa CLIPS capaz de provocar un comportamiento reactivo en el nuevo robot de modo que no se salga de los márgenes de la carretera, ni choque con los obstáculos.

◁

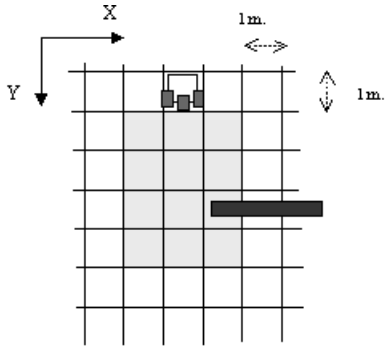

Figura 3.9:

Ejercicio 3.4 A la luz de los experimentos considerados anteriormente puede ser interesante considerar el siguiente texto de Herbert A. Simon (tomado de su ya clásico libro *The Sciences of the Artificial* [29]):

> Observamos a una hormiga abrir su laborioso camino a través de una playa moldeada por el viento y las olas. Avanza, tuerce a la derecha para facilitar su escalada de una empinada dunilla, se desvía para rodear un guijarro, se detiene un momento para intercambiar información con una colega.
>
> Así va y viene, deteniéndose en su camino de vuelta a casa. Para no antropomorfizar sus propósitos, trazo el camino sobre un papel. Se trata de una secuencia de segmentos angulares, irregulares –no es un paseo completamente aleatorio, ya que posee un sentido subyacente de dirección, de apuntar hacia una meta.
>
> Muestro el trazo sin letrero alguno a un amigo. ¿Quién describió un camino así? Tal vez un esquiador, haciendo slalom en una fuerte pendiente algo rocosa. O un balandro navegando contra el viento por un canal salpicado de islas y bancos de arena. Tal vez se trata de un camino en un espacio más abstracto: el curso de la búsqueda de un estudiante que intenta demostrar un teorema en geometría.
>
> Quienquiera que trazase el camino, y sea cual sea el espacio, ¿por qué no es recto? ¿por qué no apunta directamente desde el punto de partida hacia su meta? En el caso de la hormiga (y, a todos los

efectos, en el de los demás) conocemos la respuesta. Ella tiene un sentido general de donde se encuentra su casa, pero no puede prever todos los obstáculos que hay enmedio. Debe adaptar su curso repetidas veces a las dificultades que encuentra, y a menudo desviarse para evitar barreras infranqueables. Su horizonte es cercano, de modo que aborda cada obstáculo a medida que lo encuentra. Explora formas de salvarlo sin pensar mucho en los obstáculos futuros. Es fácil atraparla en largos desvíos.

Considerado como una figura geométrica, el camino de la hormiga es irregular, complejo y difícil de describir. Pero su complejidad es en realidad una complejidad en la superficie de la playa, no una complejidad en la hormiga. Sobre esa misma playa, otra pequeña criatura con su casa en el mismo lugar que la de la hormiga, bien podría seguir un camino muy parecido.

[...]Estas especulaciones sugieren una hipótesis [...] *Una hormiga, considerada como un sistema con comportamiento, es bastante simple. La aparente complejidad de su comportamiento a lo largo del tiempo es en gran medida un reflejo de la complejidad del medio en que se encuentra.*

Me gustaría examinar esta hipótesis, pero con la palabra *persona* en lugar de *hormiga*: *Una persona, considerada como un sistema con comportamiento, es bastante simple. La aparente complejidad de su comportamiento a lo largo del tiempo es en gran medida un reflejo de la complejidad del medio en que se encuentra.*

(*Continuará...*)

Hasta aquí el texto de H. Simon. Se sugieren las siguientes actividades:

a) Lee el texto con detenimiento.
b) Reflexiona sobre el mismo durante al menos 5 minutos.
c) Coméntalo mañana en la cafetería con tus compañeros. ◁

Capítulo 4

CORRESPONDENCIA DE PATRONES.

4.1. Patrones con variables.

Como se indicó en la sección 2.1.2, las reglas en CLIPS se definen utilizando la construcción `defrule`:

```
(defrule <nom-regla>
   [<comentario>]
   <elemento-condicional>*
 =>
   <acción>*)
```

y los elementos condicionales más básicos son los llamados *patrones*, que describen mediante constantes, variables y restricciones hechos que pueden estar presentes en la memoria de trabajo. Hasta ahora sólo hemos empleado constantes, pero con ello no hemos hecho uso de la auténtica potencia de los lenguajes de esta familia. Veamos cómo se pueden definir patrones con variables.

En CLIPS los nombres de variables se forman prefijando un signo de interrogación a un nombre de símbolo. Así pues, son nombres de variables válidos `?x`, `?alumno1`, `?punto-sobre-i`. Las variables en CLIPS no tienen tipo, es decir, pueden quedar ligadas a un campo de cualquiera de los tipos básicos.

Sea P un patrón con variables $V_1, \ldots, V_n$ y sea H un hecho de la memoria de trabajo. P se corresponde con H si y sólo si es posible sustituir las variables

$V_1, \ldots, V_n$ que aparecen en P de forma que el patrón constante P' que resulte coincide con H. En ese caso, las variables $V_1, \ldots, V_n$ quedan ligadas a esos valores en el resto de la regla (tanto en el resto de la parte izquierda como en la parte derecha de la regla.) Podemos decir pues que el ámbito de una variable está constituido por la regla donde aparece; es un error encontrar una variable no ligada en la parte derecha de una regla.

Si existen varios hechos $H_1, \ldots, H_m$ que se corresponden con el patrón P, se generará una activación diferente de la regla por cada uno de estos hechos.

Ejemplo 4.1 Supongamos un programa con las siguientes plantillas, hechos iniciales y reglas:

```
(deftemplate persona
  (slot nom (type SYMBOL))) ;el nombre
(deftemplate hombre
  (slot nom (type SYMBOL)) ;el nombre
  (slot color-ojos (allowed-values azul gris verde turquesa marron)))
(deftemplate mujer
  (slot nom (type SYMBOL))
  (slot color-ojos (allowed-values azul gris verde turquesa marron)))
(deffacts hombres-y-mujeres
  (hombre (nom maurice)     (color-ojos azul))
  (hombre (nom jean-michel) (color-ojos marron))
  (mujer  (nom mary)        (color-ojos azul))
  (mujer  (nom sally)       (color-ojos verde))
  (hombre (nom olof)        (color-ojos turquesa))
  (hombre (nom gandalf)     (color-ojos azul))
  (mujer  (nom rosa)        (color-ojos verde)))
(defrule hombres  "los hombres son personas"
  (hombre (nom ?nombre))
=>
  (assert (persona (nom ?nombre)))
(defrule mujeres  "las mujeres son personas"
 (mujer (nom ?nombre))
=>
 (assert (persona (nom ?nombre)))
```

El contenido de la lista de hechos y la agenda tras cargar el programa y ejecutar `(reset)` sería:

Agenda	*Facts*
mujeres: f-7	0. (initial-fact)
hombres: f-6	1. (hombre (nom maurice)(color-ojos azul))
hombres: f-5	2. (hombre (nom jean-michel)(color-ojos gris))
mujeres: f-4	3. (mujer (nom mary)(color-ojos azul))
mujeres: f-3	4. (mujer (nom sally)(color-ojos verde))
hombres: f-2	5. (hombre (nom olof)(color-ojos turquesa))
hombres: f-1	6. (hombre (nom gandalf)(color-ojos marron))
	7. (mujer (nom rosa)(color-ojos verde))

Nótese que cada hecho de tipo `hombre` se corresponde con el patrón de la regla `hombres`, y que cada hecho de tipo `mujer` se corresponde con el patrón de la regla `mujeres`. En cada caso, la variable `?nombre` queda ligada al valor correspondiente.

Tras evaluar el programa ejecutando `(run)`, el contenido de la lista de hechos será el siguiente:

Facts
0. (initial-fact)
1. (hombre (nom maurice)(color-ojos azul))
2. (hombre (nom jean-michel)(color-ojos gris))
3. (mujer (nom mary)(color-ojos azul))
4. (mujer (nom sally)(color-ojos verde))
5. (hombre (nom olof)(color-ojos turquesa))
6. (hombre (nom gandalf)(color-ojos marron))
7. (mujer (nom rosa)(color-ojos verde))
9. (persona (nom rosa)
10. (persona (nom gandalf)
11. (persona (nom olof)
12. (persona (nom sally)
13. (persona (nom mary)
14. (persona (nom jean-michel)
15. (persona (nom maurice)

◁

Cuando una variable aparece varias veces en la parte izquierda de una regla, todas sus apariciones deben ligarse al mismo valor. Por ejemplo, la regla

```
(defrule mismo-color-ojos  "dos mujeres con ojos del mismo color"
```

```
  (mujer (nom ?n1) (color-ojos ?col))
  (mujer (nom ?n2) (color-ojos ?col))
=>
  (assert (mismo-color-de-ojos ?n1 ?n2)))
```

afirmaría en la lista de hechos los siguientes hechos:

```
(mismo-color-de-ojos rosa rosa)
(mismo-color-de-ojos rosa sally)
(mismo-color-de-ojos sally rosa)
(mismo-color-de-ojos sally sally)
(mismo-color-de-ojos mary mary)
```

Nótese que nada obliga a que los dos hechos de tipo `mujer` que aparecen en el antecedente de la regla deban ser diferentes. Tan sólo se obliga a que el color de ojos sea igual, sin establecer ninguna restricción o relación entre los valores de `?n1`, `?n2`.

Las variables que aparecen por primera vez dentro de un elemento condicional `not` no pueden recibir, por razones obvias, ningún valor. Es por tanto un error utilizarlas a continuación como si lo tuvieran. Por ejemplo, la siguiente regla es errónea:

```
(defrule regla-erronea
  "esta regla hace un uso erroneo de la variable ?color"
  (not (hombre (color-ojos ?color)))
=>
  (assert (falta-color ?color)))
```

El intérprete no aceptará la definición anterior, ya que la variable `?color` utilizada en la parte derecha de la regla no puede haber recibido ningún valor en la parte izquierda.

Algo parecido puede ocurrir con el elemento condicional `or`. Considérese por ejemplo la siguiente regla:

```
(defrule regla-erronea-2
  "esta regla usa las variables incorrectamente"
  (or (hombre (color-ojos ?color-1))
      (mujer  (color-ojos ?color-2)))
=>
  (assert (uno-de-dos ?color-1 ?color-2)))
```

La regla se activa si:

–Algún hecho de la lista se corresponde con el patrón `(hombre (color-ojos ?color-1))`, en cuyo caso la variable `?color-1` quedará adecuadamente ligada, mientras que `?color-2` no recibirá ningún valor.

–Algún hecho de la lista se corresponde con el patrón `(mujer (color-ojos ?color-2))`, en cuyo caso la variable `?color-2` quedará adecuadamente ligada, mientras que `?color-1` no recibirá ningún valor.

En cualquier caso una de las dos variables quedará sin ligar, por lo que no pueden utilizarse con seguridad en la parte derecha de la regla. El intérprete detecta esta circunstancia y provocará un error si se introduce la definición anterior.

Ejercicio 4.1 ¿Será correcta la siguiente definición?

```
(defrule regla-misteriosa-1
  (initial-fact)
  (not (hombre (color-ojos ?color)))
  (mujer (color-ojos ?color))
=>
  (assert (color-femenino ?color)))
```

¿Es semánticamente equivalente la siguiente regla?

```
(defrule regla-misteriosa-2
  (initial-fact)
  (mujer (color-ojos ?color))
  (not (hombre (color-ojos ?color)))
=>
  (assert (color-femenino ?color)))
```

◁

Ejemplo 4.2 Volvamos a considerar el problema del cálculo de la calificación de los alumnos de una clase. Al disponer de variables, podemos ahora calcular "de golpe", en un solo programa, las calificaciones de todos los alumnos. Supongamos, por ejemplo, que se han realizado los exámenes $n1$ y $n2$, con las posibles calificaciones *suspenso/aprobado/notable/sobresaliente*, y que la calificación final es "apto" cuando las notas en ambos exámenes son distintas de "suspenso". Si Pepe ha obtenido aprobado y notable, Mari aprobado y suspenso, y Loli sólo se presentó al primer examen obteniendo notable, el programa completo para el cálculo de sus notas sería éste:

```
(deffacts notas
  (oav pepe n1 aprobado)
  (oav pepe n2 notable)
  (oav mari n1 aprobado)
  (oav mari n2 suspenso)
  (oav loli n1 notable))
(defrule regla-apto
```

```
  (oav ?n n1 ~suspenso)
  (oav ?n n2 ~suspenso)
=>
  (assert (oav ?n total apto)))
```

◁

Ejercicio 4.2 Suponiendo los datos del ejemplo anterior, escribir el programa correspondiente a estas nuevas especificaciones: se considera apto al alumno que ha aprobado ambos exámenes, o bien ha suspendido uno de ellos pero en el otro ha obtenido sobresaliente. ◁

4.2. Funciones en CLIPS.

A lo largo de las páginas anteriores hemos mencionado en diversos lugares que CLIPS permite el uso de *funciones* (por ejemplo, en la sección 1.2 dijimos que se podían invocar desde la ventana de interacción; en la sección 2.1.2 dijimos que la parte derecha de una regla consiste en una sucesión de llamadas a funciones). También hemos hecho uso de algunas funciones predefinidas para el manejo de los hechos y la agenda (`assert` y demás).

Como ya habrá notado el lector, las funciones en CLIPS utilizan la llamada *notación prefija*, es decir, para llamar a una función se escriben entre paréntesis primero el nombre de la función y luego los argumentos separados por espacios en blanco:

```
(nombre-funcion arg1 arg2 ... argn)
```

Estos argumentos pueden ser, a su vez, llamadas a otras funciones.

En CLIPS están predefinidas, entre otras, las funciones aritméticas elementales:

`+`, `-`, `*`, `/`.

Por ejemplo:

```
> (+ 3 5)
> 8
> (* (+ 1 2) (+ 3 4))
> 21
> (- (* (+ 1 2) (+ 3 4)) (+ 3 5))
> 13
```

En CLIPS están predefinidas tmbién las funciones de comparación numéricas:

`>`, `<`, `>=`, `<=`, `=` y `<>`.

y las funciones de comparación más generales:

`eq` y `neq`.

Por ejemplo:

```
> (> (+ 1 2) (+ 0 3))
> FALSE
> (eq  hola hola)
> TRUE
> (neq "hola" "hola")
> FALSE
```

4.3. Elementos condicionales *test*.

Otro tipo de elemento condicional, que puede aparecer en la parte izquierda de las reglas, es el dado por

`(test <llamada-a-predicado>)`

El elemento se satisface si y sólo si el resultado devuelto por la llamada al predicado es TRUE. Si el primer elemento de una regla es un elemento condicional `test`, se supone que antes aparece el patrón `(initial-fact)`. No debe confundirse con un tipo de hecho que aparezca en la memoria de trabajo. De hecho es imposible definir hechos cuyo nombre de relación sea `test`, ya que es una palabra reservada del lenguaje. Por ejemplo:

```
> (assert (test))
[PATTERN1] The symbol test has special meaning and may not be
used as a relation name.
>(deftemplate test (slot uno))
[PATTERN1] The symbol test has special meaning and may not be
used as a relation name.
```

Estos elementos condicionales se pueden emplear para efectuar comprobaciones acerca de valores ya obtenidos mediante la correspondencia de patrones con variables. Por ejemplo, esta regla permite comprobar que el valor de una variable ya ligada `?e` es mayor que 64:

```
(deftemplate persona
  (slot nom)
  (slot edad))
(defrule jubilados-con-test
  (persona (nom ?n) (edad ?e))
  (test (> ?e 64))
=>
  (assert (jubilado ?n)))
```

Ejercicio 4.3 Volvamos a considerar el problema del cálculo de la calificación de los alumnos del ejemplo 4.2. Recordemos que la calificación final es “apto”

cuando las notas en ambos exámenes son distintas de "suspenso". Consideremos ahora que las calificaciones en los exámenes $n1$ y $n2$ son numéricas. Si Pepe ha obtenido 5,5 y 8, Mari 6 y 3,5, y Loli sólo se presentó al primer examen obteniendo 7, escribir un programa para el cálculo de sus notas finales. ◁

4.4. Funciones dentro de los patrones.

Hasta este momento, hemos visto que se pueden emplear las siguientes restricciones para describir un patrón:

–una constante, p. ej. `(color azul)`

–una variable, p. ej. `(color ?c)`

–la composición de las anteriores con las conectoras `|`, `&`, `~`, p. ej. `(color azul|rojo)`

Aún hay otras dos posibilidades, dadas por la llamada a una función dentro del patrón.

La primera de ellas viene dada por

```
:<llamada-a-predicado>
```

que se utiliza normalmente junto a alguna conectora `|`, `&`, `~`. Permite comprobar que determinado valor cumple una condición. Es equivalente muchas veces a usar un elemento condicional `test` después de un patrón, aunque el método ahora expuesto es más eficiente. Por ejemplo, para realizar la comprobación anterior de que una persona tiene más de 64 años podríamos decir:

```
(defrule jubilados-con-restriccion-de-campo
  (persona (nom ?n) (edad ?e&:(> ?e 64)))
=>
  (assert (jubilado ?n)))
```

donde `?e&:(> ?e 64)` puede leerse como "unifica `?e` y además comprueba que sea mayor que 64".

Otra forma de establecer restricciones dentro de un patrón viene dada por

```
=<llamada-a-funcion>
```

El valor que devuelve la llamada a la función es el que se empleará como restricción en la correspondencia de patrones. Por ejemplo, las siguientes construcciones

```
(deffacts numeros
 (numero 1)
 (numero 2)
 (numero 3)
 (numero 4))

(defrule que-casualidad
  (numero ?x)
  (numero =(* ?x ?x))
=>
  (assert (presente-cuadrado ?n)))
```

provocarían la aserción de `(presente-cuadrado 1)` y `(presente-cuadrado 2)`

Ejercicio 4.4 Repetir el ejercicio 4.3, empleando ahora patrones con restricciones en lugar de elementos condicionales de tipo `test`. ◁

4.5. Patrones asignados.

El índice de un hecho puede guardarse en una variable con ayuda del operador `<-`. Ello se consigue mediante los elementos condicionales de tipo *patrón asignado*, que tienen la siguiente sintaxis:

```
<variable> <- <patron>
```

Estos elementos condicionales tienen gran importancia, ya que permiten borrar (`retract`), modificar (`modify`) o duplicar (`duplicate`) en la parte derecha de las reglas los hechos con los que se ha establecido correspondencia en la parte izquierda. Por ejemplo:

```
(defrule borra-h-ojos-azules
  "borra a los hombres con ojos azules"
  ?p <- (hombre (color-ojos azul))
=>
  (retract ?p))
```

Mediante el disparo de una regla se pueden pues borrar cualesquiera hechos de la memoria, o modificar el valor de cualquiera de sus ranuras (lo que, recuérdese lo dicho en la sección 2.3.1, equivale a borrar el hecho antiguo y asertar uno nuevo.) De esta forma, abandonamos la tierra firme de la programación declarativa; pues, como es sabido, en lógica clásica una vez que se ha deducido o asertado un hecho, no tiene sentido desdecirse de ello. Pero los lenguajes basados en reglas, en realidad, no pretenden ser demostradores automáticos, sino herramientas de propósito más general. Y, en este contexto más general, algunas de las aplicaciones típicas de `retract` son las siguientes:

–Representar un ambiente cambiante. Por ejemplo, supóngase un sistema de control que recibe la señal `(leido s1 <t>)` de un sensor s1 y la almacena en un hecho `(oav s1 temperatura <t>)`. Una regla para procesar esto puede ser

```
(defrule actualiza-temp
  "actualiza el valor de la temperatura de s1"
  ?h <- (oav s1 temperatura ?t1)
  (leido s1 ?t2)
=>
  (retract ?h)
  (assert (oav s1 temperatura ?t2)))
```

–Retirar los valores por defecto. Por ejemplo, en este programa se supone al principio que el sensor s1 funciona bien, pero la regla `sensor-chungo` puede modificar esta circunstancia:

```
(deftemplate sensor
  (id (allowed-values s1 s2 s3 s4) (default ?NONE))
  (estado (allowed-values roto ok) (default ok))
  (lectura (type REAL)))
(deffacts f1
  (sensor (id s1) (lectura 27)))
(defrule sensor-chungo
  "si la lectura del sensor es absurda, el sensor esta roto"
  ?h1 <- (sensor (id ?s) (estado ok))
  (leido ?sensor ?valor&:(< ?valor 0))
=>
  (modify ?h1 (estado roto)))
```

–Definir algoritmos. A veces se emplean ciertos hechos para describir, no el conocimiento del dominio, sino un algoritmo de control. Estos hechos deben ir cambiando para reflejar los pasos del algoritmo. Por ejemplo, supóngase un sistema de reparación de averías que primeramente diagnostica cuál es el fallo, a continuación determina el procedimiento más adecuado para tratarlo y finalmente genera un informe al usuario. Podría emplearse un hecho de la forma `(fase <f>)` para indicar cuál de las tres fases se está y habría reglas de la forma

```
(defrule paso-a-tratamiento
  "si ha acabado el diagnóstico, pasar al tratamiento"
  ?h1 <- (fase diagnostico)
=>...
  (retract ?h1)
  (assert (fase tratamiento)))
```

El uso de `modify` puede dar alguna sorpresa al programador principiante. El siguiente ejemplo aclarará lo que queremos decir.

Ejemplo 4.3 Supongamos el siguiente programa:

```
(deffacts f1
  (oav pepe n1 aprobado)
  (oav pepe n2 aprobado))
(defrule nota-total
  (oav ?a n1 aprobado)
  (oav ?a n2 aprobado)
=>
  (assert (oav ?a total aprobado)))
```

Tras inicializarlo, la memoria de trabajo contiene `0: (initial-fact)` `1: (oav pepe n1 aprobado)` y `2: (oav pepe n2 aprobado)`, y el contenido de la agenda es una activación de `nota-total`. La activación se disparará y se retirará de la agenda, se añadirá `(oav pepe total aprobado)` a la memoria, la agenda quedará vacía (pues, aunque se sigue satisfaciendo el antecedente de la regla `nota-total`, lo hace con los mismos hechos que originaron su disparo, recordemos lo dicho en la sección 2.2) y el programa terminará.

Supongamos ahora el siguiente programa:

```
(deftemplate alumno
  (slot id)
  (slot n1) (slot n2)
  (slot total (default np)))
(deffacts f1
  (alumno (id pepe) (n1 aprobado)
  (n2 aprobado)))
(defrule nota-total
  ?h1 <- (alumno (n1 aprobado)
 (n2 aprobado))
=>
  (modify ?h1 (total aprobado)))
```

Si el lector lo ejecuta (recomendamos que lo haga paso a paso), comprobará que no termina. ¿Qué es lo que ocurre? Tras inicializarlo, la memoria de trabajo contiene `0: (initial-fact)` y `1: (alumno (id pepe) (n1 aprobado) (n2 aprobado) (total np))`. El contenido de la agenda, como antes, es una activación de `nota-total`. La activación se disparará y se retirará de la agenda, y el disparo producirá la "modificación" del hecho 1. Pero, en realidad, lo que ocurre es que el hecho 1 se borra y en su lugar se aserta un nuevo hecho `2: (alumno (id pepe) (n1 aprobado) (n2 aprobado) (total aprobado))`. Al ser un nuevo hecho (aunque el programador quizás lo haya conceptualizado como "el mismo de antes"), se genera una nueva activación de `nota-total`, que borra 2 de la memoria y añade un hecho 3 idéntico a 2, y así sucesivamente...

La solución más sencilla será exigir explícitamente en la regla que la nota total no se haya calculado:

```
(defrule nota-total
  ?h1 <- (alumno (n1 aprobado)
                 (n2 aprobado)
                 (total np))
=>
  (modify ?h1 (total aprobado)))
```

Ahora sólo se generará la primera activación de la regla, pues tras su disparo ya no se cumplirá `(total np)`. ◁

Ejercicio 4.5 Volvamos a considerar el problema del cálculo de la calificación de los alumnos de una clase tal como se expuso en el ejemplo 4.2. Empleando una plantilla para almacenar todos los datos de un alumno (datos iniciales o calculados por el programa), escribir un programa que calcule la calificación final. ◁

4.6. Comodines.

Un comodín corresponde a una variable anónima. En CLIPS los comodines se representan mediante el signo de interrogación `?`.

Los comodines permiten comprobar la presencia de un valor en un multicampo o en un hecho ordenado, sin necesidad de utilizar una variable. Por ejemplo, supongamos presentes los siguientes hechos y la siguiente regla:

```
(deffacts hechos-padre-de
  "(padre-de X Y) significa que X es padre de Y"
  (padre-de maurice jean-michel)
  (padre-de juan    juanito)
  (padre-de luis    margarita))
(defrule paternidad
  "si es padre de alguien, entonces tiene hijos"
  (padre-de ?nom  ?)
=>
  (assert (tiene-hijos ?nom)))
```

La ejecución del programa añadiría los siguientes hechos:

```
(tiene-hijos luis)
(tiene-hijos juan)
(tiene-hijos maurice)
```

4.7. Variables multicampo.

Las variables multicampo permiten establecer correspondencia con cero, uno o más valores de un multicampo. En la parte izquierda de las reglas, se indica que una variable (en particular, puede ser una variable comodín) debe corresponderse con un valor multicampo precediendo su nombre con el signo `$`. Así, las siguientes variables son multicampo: `$?notas`, `$?asignaturas`, `$?`.

Consideremos por ejemplo las siguientes construcciones:

```
(deftemplate padre
  (slot nombre)
  (multislot hijos))
(deffacts padres
  (padre (nombre pepe)    (hijos))
  (padre (nombre maurice) (hijos jean-michel))
  (padre (nombre carlos)  (hijos margarita luis pepe)))
(defrule hermanos
  (padre (hijos $?h))
=>
  (assert (hermanos ?h))
```

Nótese que en la parte derecha de la regla no es necesario emplear el símbolo `$` delante de la variable. De hecho, este símbolo no forma parte del nombre de la variable, y se utiliza únicamente para indicar que al establecer la correspondencia el valor de `?h` será un multicampo. La ejecución del programa produciría el siguiente resultado:

```
(hermanos margarita luis pepe)
(hermanos jean-michel)
(hermanos)
```

También es posible utilizar comodines para que una regla procese todos los elementos de un multicampo. Por ejemplo, supongamos los hechos anteriores y la siguiente regla:

```
(defrule ver-hijos-de-carlos
  "aserta los hijos de carlos"
  (padre (nombre carlos) (hijos $? ?hijo $?))
=>
  (assert (hijo-de-carlos ?hijo)))
```

Analicemos los hechos de la memoria y el patrón de la regla y determinemos todas las posibles formas de establecer la correspondencia entre ambos. Recordemos que el comodín multicampo $? puede ligarse a un multicampo con cero,

uno, o más campos, mientras que la variable `?hijo` debe ligarse obligatoriamente a un campo:

Hecho	`(padre (hijos margarita luis pepe))`
Patrón	`(padre (hijos $? ?hijo $?))`

Las posibles ligaduras de cada uno de los elementos del patrón son:

`$?`	`?hijo`	`$?`
`()`	`margarita`	`(luis pepe)`
`(margarita)`	`luis`	`(pepe)`
`(margarita luis)`	`pepe`	`()`

Nótese que el primer comodín multicampo podría corresponderse con el valor `(margarita luis pepe)`, pero en tal caso la variable `?hijo` ya no puede corresponderse con ningún campo. Por tanto la ejecución del programa anterior añadiría a la memoria de trabajo

```
(hijo-de-carlos margarita)
(hijo-de-carlos luis)
(hijo-de-carlos pepe)
```

Las variables y comodines multicampo son muy versátiles, como muestran los siguientes ejemplos.

Ejemplo 4.4 Queremos definir en CLIPS hechos y reglas que permitan gestionar una pila. Para almacenar el contenido de la pila emplearemos hechos ordenados `(pila <elemento>*)`:

```
(deffacts la-pila
  "una pila inicialmente vacia"
  (pila))
```

Cada vez que deseemos introducir un elemento en la pila añadiremos a la memoria de trabajo un hecho `(push <elemento>)`, y cada vez que queramos sacar un elemento añadiremos un hecho `(pop)`.

Adoptando esta representación, las siguientes tres reglas se encargan de hacer el trabajo:

```
(defrule incluir-en-pila
  "añadir un elemento a la pila"
  ?p <- (pila $?contenido)
  ?o <- (push ?x)
```

```
=>
  (retract ?p ?o)
  (assert (pila ?x ?contenido)))
(defrule sacar-de-pila  "sacar un elemento de la pila"
  ?p <- (pila ?cima $?contenido)
  ?o <- (pop)
=>
  (retract ?p ?o)
  (assert (pila ?contenido)))
(defrule error-al-sacar-de-pila  "sacar de la pila vacia"
  (pila)
  ?o <- (pop)
=>
  (retract ?o)
  (error "ERROR - LA PILA YA ESTA VACIA"))
```

◁

Ejemplo 4.5 Vamos ahora a programar el conocido algoritmo de resolución para cláusulas del cálculo de proposiciones. Recordemos que una cláusula es una fórmula $L_1 \vee L_2 \vee \ldots \vee L_n$, $(n > 0)$ donde cada L_i es un literal, es decir, una proposición atómica afirmada o negada; por ejemplo, son cláusulas $p \vee \neg q \vee r$, p, $\neg q$. La regla de resolución dice que si en una cláusula C_1 aparece afirmada una proposición p y en otra claúsula C_2 aparece negada, se puede generar una nueva cláusula con todos los literales que aparezcan en cualquiera de las dos. Por ejemplo, si tenemos $C_1 = p \vee \neg q \vee s$ $C_2 = w \vee q \vee r$, y consideramos la proposición q, ponemos obtener $C_3 = p \vee s \vee w \vee r$.

Representemos cada cláusula con un hecho `(clausula <literal>*)` donde cada literal será `si <proposicion>` o `no <proposicion>`. Por ejemplo, la cláusula $p \vee \neg q \vee \neg r \vee p \vee \neg r$ se representará por `(clausula si p no q no r si p no r)`. La regla de resolución es simplemente

```
(defrule resolver
   "Regla de resolucion"
   ?h1 <- (clausula $?x1 si ?y $?z1)
   ?h2 <- (clausula $?x2 no ?y $?z2)
   (test (neq ?h1 ?h2)) ;nunca resolveremos una cláusula consigo misma
=>
   (assert (clausula $?x1 $?z1 $?x2 $?z2)))
```

El método acaba con éxito si alcanzamos la llamada cláusula vacía, que es un caso especial de cláusula que no contiene ningún literal; en otro caso, si se agotan las posibilidades de resolver sin alcanzar la cláusula vacía, el método acaba con fracaso. Bastará pues inspeccionar el estado de la memoria al final de programa para saber si se ha acabado con éxito o fracaso. Pero de esta forma no garantizamos que el programa pare al alcanzar la cláusula vacía; resolverá siempre

que sea posible, hasta agotar todas las posibilidades. Una posible solución es comprobar si la cláusula asertada es la vacía, y en caso afirmativo parar. Para ello emplearemos, adelantándonos un poco a lo que veremos en un capítulo posterior, la función `create$`, que llamada sin argumentos devuelve un multicampo vacío (otra posibilidad hubiera sido emplear la función `length$`).

```
(defrule resolver
   "Regla de resolucion (clausula no vacia)"
   ?h1 <- (clausula $?x1 si ?y $?z1)
   ?h2 <- (clausula $?x2 no ?y $?z2)
   (test (neq ?h1 ?h2)) ;nunca resolveremos una cláusula consigo misma
   (or (test (neq $?x1 (create$))) ;comprobamos si algún trozo es no vacío
       (test (neq $?z1 (create$)))
       (test (neq $?x2 (create$)))
       (test (neq $?z2 (create$))))
=>
   (assert (clausula $?x1 $?z1 $?x2 $?z2)))
(defrule resolver-acabar
   "Regla de resolucion (clausula vacia y parar)"
   ?h1 <- (clausula si ?y)
   ?h2 <- (clausula no ?y)
=>
   (assert (clausula))
   (halt))
```

El programa no simplifica las cláusulas iniciales, ni las que que va calculando. Por ejemplo, al resolver `(clausula si p no q si r)` y `(clausula no p no q si r)` obtiene `(clausula no q si r no q si r)` en lugar de `(clausula no q si r)`. Ello hace que a veces no termine; por ejemplo, pruébese con

```
(deffacts mala-idea
  (clausula si p si p si p)
  (clausula no p no p no p))
```

Pero por ahora lo dejaremos así... ◁

Ejercicio 4.6 Repetir el ejemplo 4.5, pero representando ahora una cláusula mediante una plantilla

```
(deftemplate (clausula)
  (multislot afirmados)
  (multislot negados))
```

◁

4.8. Elementos condicionales exists y forall.

El elemento condicional `exists` tiene la siguiente sintaxis:

`(exists <elemento-condicional>+)`

Este elemento se satisface si se satisface al menos de una forma el conjunto de elementos condicionales que incluye. Podría pensarse que este elemento es redundante, ya que el proceso de correspondencia de patrones consiste precisamente en averiguar si "existe" algún hecho en la memoria que corresponda con el elemento condicion. Pero hay dos diferencias esenciales entere usar y no usar `exists`:

1) Aunque el elemento condicional pueda satisfacerse de varias formas, `exists` sólo considera una de ellas a efectos de generar activaciones. Por ejemplo, supongamos en la memoria de trabajo los hechos

`(oav pepe altura 186)` `(oav juan altura 191)` `(oav pepe altura 190)`.

Supongamos la regla

```
(defrule regla-sin-exists
  (oav ?n altura ?h&:(>?h 185))
=>
  (assert (oav ?n altura-abs alto)))
```

Habrá tres activaciones de ella en la agenda, cuyos disparos asertarán los respectivos hechos `(oav <n>altura-abs alto)`.

Con el mismo estado de la memoria, consideremos ahora la regla

```
(defrule regla-con-exists
  (exists (oav ?n altura ?h&:(>?h 185)))
=>
  (assert (hay-alto)))
```

Habrá una única activación de ella en la agenda, pese a que hay tres hechos que están en correspondencia con `(oav ?n altura ?h&:(>?h 185))`.

2) Las ligaduras de variables establecidas dentro de un elemento condicional `exists` no son visibles desde el exterior, es decir, las variables que aparecen dentro de un elemento condicional `exists` son locales al mismo. Por tanto, esta regla es sintácticamente errónea:

```
(defrule regla-con-exists-erronea
  (exists (oav ?n altura ?h&:(>?h 185)))
=>
  (assert (hay-alto ?n)))
```

Nota.- `(exists <elemento-condicional>+)` es equivalente a `(not (not (and <elemento-condicional>+)))`. En consecuencia:

(1) Si el primer elemento condicional de una regla es exists, se supone que antes aparece `(initial-fact)`.

(2) Como acabamos de decir, las variables son locales.

El elemento condicional `forall` tiene la siguiente sintaxis:

`(forall <elemento-condicional-1><elemento-condicional>+)`

Este elemento se satisface si para todas las formas de satisfacer el primer elemento condicional, se satisface de alguna forma el conjunto de los restantes elementos condicionales `<elemento-condicional>+`. Quizás sean convenientes algunos ejemplos para aclarar esta semántica.

Ejemplo 4.6 Sea el elemento condicional e_1 = `(forall (a ?x) (b ?x))`. Consideremos que la memoria contiene únicamente

```
0. (initial-fact)
```

Entonces e_1 se satisface. Consideremos ahora que la memoria contiene únicamente

```
0. (initial-fact)        1. (b 1)
```

Entonces e_1 también se satisface. Consideremos ahora que la memoria contiene

```
0. (initial-fact)        1. (a 1)
```

Entonces e_1 no se satisface. Pero si la memoria contiene

```
0. (initial-fact)        1. (a 1)        2. (b 1)
```

Entonces e_1 se satisface, al igual que si contiene

```
0. (initial-fact)        1. (a 1)        2. (b 1)
3. (b 3)        4. (b 4)        5. (b 0)
```

◁

Ejemplo 4.7 Supongamos que tenemos la plantilla `alumno` con ranuras `id` y `nota-final`. Si todos los alumnos han obtenido más de 6, prometemos llevarlos de excursión a Disneylandia. Ello se puede representar mediante la regla

```
(defrule r1
  (forall (alumno (id ?n)) (alumno (id ?n) (nota-final ?f&:(> ?f 6))))
=>
  (assert (excursion disneylandia prometido)))
```

cuya lectura sería: "Si para todo $?n$ tal que hay un alumno de identificador $?n$ existe un alumno de identificador $?n$ con nota mayor de 6, entonces asertar que una excursión a Disneylandia está prometida". Ya que se supone que el identificador identifica unívocamente a cada alumno, ello equivale a: "Si para todo alumno ese alumno tiene más de 6, entonces asertar que una excursión a Disneylandia está prometida" ◁

Ejercicio 4.7 Supongamos la representación de los alumnos de una clase dada en el ejemplo anterior (ejemplo 4.7). Escribir reglas CLIPS correspondientes a los siguientes enunciados en lenguaje natural:

–"Si existe algún alumno cuya nota sea menor que 3, hay que repetir el examen".

–"Si existen dos alumnos cuyas notas sean menores que 3, hay que repetir el examen".

–"Si todos los alumnos tienen una nota menor que 3, hay que cambiar al profesor".

–"Si todos los alumnos salvo uno tienen una nota menor que 3, hay que felicitar a ese alumno". ◁

Al igual que ocurre con los elementos `exists`, las ligaduras de variables establecidas dentro de un elemento condicional `forall` no son visibles desde el exterior, es decir, las variables que aparecen dentro de un elemento condicional `forall` son locales al mismo. Por tanto, esta regla es sintácticamente errónea:

```
(defrule regla-con-forall-erronea
  (forall (oav ?n altura ?h&:(>?h 185)))
=>
  (assert (hay-alto ?n)))
```

Nota.- `(forall <elemento-condicional-1><elemento-condicional>+)` es equivalente a

```
(not (and <elemento-condicional-1>
          (not (and <elemento-condicional>+))))
```

En consecuencia:

(1) Si el primer elemento condicional de una regla es un `forall`, se supone que antes aparece `(initial-fact)`.

(2) Como hemos dicho, las variables que aparecen dentro de un elemento condicional `forall` son locales al mismo.

Ejemplo 4.8 Considérese el caso de un sistema de control que posee 5 dispositivos. Periódicamente se comprueba si hay alguno activado. Si hay al menos un sensor activado hay que dar una señal de alarma:

```
(deftemplate sensor
  "datos de un sensor"
  (slot id)       ;el nombre del sensor
  (slot estado))  ;puede ser activado o desactivado

(deffacts sensores
```

```
  (sensor (id 1) (estado desactivado))
  (sensor (id 2) (estado activado))
  (sensor (id 3) (estado activado))
  (sensor (id 4) (estado desactivado))
  (sensor (id 5) (estado activado)))

(defrule aviso-de-sensores-activados
  "avisa cuando al menos hay un sensor activado"
  (exists (sensor (estado activado)))
=>
  (assert (alarma sensores-activados)))
```

La ejecución del programa anterior provocaría una única activación de la regla, aunque existan tres sensores activados.

Considérese que cada vez que se envía un equipo de personal a comprobar un sensor se afirma en memoria un hecho del tipo `(sensor-en-revision <id>)`. La siguiente regla afirma que todo está bajo control si para todos los sensores activados hay un equipo que ya los está revisando:

```
(defrule sensores-bajo-supervision
  (forall (sensor (id ?nom) (estado activado))
          (sensor-en-revision ?nom))
=>
  (assert (alarma bajo-control)))
```

◁

Ejemplo 4.9 Vamos a programar ahora en CLIPS el conocido algoritmo de consistencia en arcos para el proprocesamiento de un problema de satisfacción de restricciones.

Supongamos que los valores posibles en cada dominio X se representan por hechos `(valor x <valor>)`. Por ejemplo, si hay tres dominios X, Y, Z, con valores 1, 2, 3, tendremos

```
(deffacts dominios
   (valor x 1) (valor x 2) (valor x 3)
   (valor y 1) (valor y 2) (valor y 3)
   (valor z 1) (valor z 2) (valor z 3))
```

Supongamos que si existe una restricción entre las variables X e Y, ello se representa mediante un hecho `(arco x y)` y un conjunto de hechos `(compatible x y <valor-1><valor-2>)` que representan todos los pares de valores compatibles con la restricción. Por ejemplo, si las restricciones son $X \leq Y$, $Y \leq Z$, $X \leq Z$, tendremos

```
(deffacts arcos
   (arco x y) (arco x z) (arco y x)
```

```
   (arco y z) (arco z x) (arco z y))
(deffacts compatibles
   (compatible x y 1 2) (compatible x y 1 3) (compatible x y 2 3)
   (compatible y x 2 1) (compatible y x 3 1) (compatible y x 3 2)
   (compatible x z 1 2) (compatible x z 1 3) (compatible x z 2 3)
   (compatible z x 2 1) (compatible z x 3 1) (compatible z x 3 2)
   (compatible y z 1 2) (compatible y z 1 3) (compatible y z 2 3)
   (compatible z y 2 1) (compatible z y 3 1) (compatible z y 3 2))
```

Ahora tendremos que expresar el algoritmo de filtrado o eliminación de valores: "para todo dominio D y todo valor V de D, si hay una restricción que liga D con D' y V es incompatible con todo valor V' de D', se debe eliminar V de D". Lo cual en CLIPS es simplemente:

```
(defrule eliminar-valor
   ?hecho<-(valor ?dom1 ?v1)
   (arco ?dom1 ?dom2)
   (forall (valor ?dom2 ?v2)
      (not (compatible ?dom1 ?dom2 ?v1 ?v2)))
 =>
   (retract ?hecho))
```

Tras ejecutar el programa, el lector puede comprobar que los únicos valores que quedan en la memoria son `(valor x 1)`, `(valor y 2)`, `(valor z 3)`. ◁

4.9. De hormigas y de hombres.

Ejercicio 4.8 ¿Recuerda el lector la historia de la hormiga de Simon (ejercicio 3.4)? No todo el mundo está de acuerdo con la moraleja que Herbert Simon proponía deducir de ella, como muestra el siguiente texto de Joseph Weizenbaum [32]:

> En una oscura noche, un policía se encuentra en la calle a un borracho. El hombre está de rodillas y obviamente busca algo a la luz de un farol. A la pregunta del policía responde que está buscando sus llaves, que acaba de perder "por allí". Y señala la oscuridad. El policía le pregunta que si efectivamente perdió las llaves "por allí" por qué las busca junto al farol, y el borracho contesta: "Porque la luz es mucho mejor aquí". De la misma forma procede la ciencia. Es importante recapacitar sobre este hecho, irrelevante e inútil, para censurar el modo en que a veces opera la ciencia, aunque, en realidad, cuando se busca algo es preciso que haya luz [...] Dos cosas importan: el tamaño del círculo de luz, que es el universo de nuestra

búsqueda, y el espíritu de esa búsqueda que debe incluir una clara noción de que hay una oscuridad exterior y fuentes de iluminación de las que se sabe todavía muy poco.

[...] Uno de los testimonios más explícitos de la forma en que la ciencia deliberada y conscientemente se propone deformar la realidad, para continuar después aceptando esta deformación como "un informe completo y exhaustivo", es el del científico Herbert. A. Simon, especialista en ordenadores, referente a su propia orientación teórica fundamental:

"Una hormiga, considerada como un sistema con comportamiento [*... se refiere al texto del ejercicio 3.4*]"

¡Mediante un simple giro de pluma, sustituyendo sólamente "hormiga" por "hombre", los supuestos irrelevantes detalles microscópicos del contexto interior de la hormiga, respecto a su comportamiento, han sido elevados a la irrelevancia del contexto interior del hombre, en su conjunto, con respecto a su comportamiento! Veintitrés años antes que Simon, como si las palabras de éste hubieran resonado en sus oídos, Huxley escribía [Aldous Huxley, *Science, Liberty and Peace*, 1946]:

"A causa del prestigio de la ciencia, como una fuente de poder, y del abandono general de la filosofía, la visión popular del mundo (*Weltanschauung*) de nuestros tiempos muestra un vasto elemento de lo que puede llamarse pensamiento 'no es más que'. Se supone, más o menos tácitamente, que los seres humanos no son más que cuerpos, animales, incluso máquinas... los valores no son más que ilusiones que se han mezclado de algún modo con nuestra experiencia del mundo; los hechos mentales no son más que epifenómenos... espiritualmente no es más que... y así sucesivamente."

Salvo, naturalmente, que aquí no se trata de una visión popular del mundo (*Weltanschauung*) sino de uno de los más prestigiosos científicos americanos. Y no es que la hipótesis de Simon acerca de lo que es irrelevante para el comportamiento del hombre sea "más o menos tácita"; al contrario, la ha hecho muy explícita.

[...] La hipótesis que ha de probarse aquí es, en parte, que el contexto interior del hombre es poco significativo en cuanto a su comportamiento. Podría suponerse que, para probarla, podría estudiarse el comportamiento humano ante el dolor y la desgracia o una profunda experiencia religiosa. Pero estos casos no se prestan fácilmente a los métodos para el estudio del ser humano desarrollados en

los laboratorios psicológicos. Tampoco es probable que condujeran a los hechos simples que las hipótesis prometían. Permanecen en las sombras, en las cuales el teórico ha perdido sus llaves; pero la luz es mucho mejor bajo el farol que él mismo ha erigido.

Así pues, no hay posibilidad de demostrar la falsedad de hipótesis como la surgida en la mente de Simon o en la de sus colegas.

(*Continuará...*)

Hasta aquí el texto de J. Weizenbaum. Se sugieren las siguientes actividades:

a) Lee el texto con detenimiento.
b) Reflexiona sobre el mismo durante al menos 5 minutos.
c) Coméntalo mañana en la cafetería con tus compañeros. ◁

Capítulo 5

CONSTRUCCIONES PROCEDIMENTALES.

5.1. Funciones predefinidas.

En la sección 4.2 explicamos cómo se emplean las funciones en CLIPS y citamos algunas funciones predefinidas. Ahora completamos esta materia, introduciendo más funciones predefinidas y explicando cómo se pueden definir nuevas funciones en un programa.

Operadores lógicos. Están predefinidos los operadores lógicos `and`, `or` y `not`. Por ejemplo:

```
> (not TRUE)
 FALSE
> (and TRUE (= 3 3.0))
 TRUE
> (or (= 3 (+ 2 1))  (symbolp "h"))
 TRUE
```

Estas funciones coinciden en nombre con los elementos condicionales `not`, `and` y `or`. Su significado también es parecido; pero debe notarse que son elementos sintácticos diferentes.

Ejercicio 5.1 Para cada una de estas reglas, decir si son sintácticamente correctas, y en caso de que lo sean indicar si se satisfacen cuando la memoria de trabajo contiene exactamente

```
   (a 1)         (a 2)         (b 3)

(defrule r1
  (and (a ?x) (b ?x))
=>
  (assert (c ?x)))
(defrule r2
  (test (and (a ?x) (b ?x)))
=>
  (assert (c ?x)))
(defrule r3
  (and (test (a ?x)) (test (b ?x))))
=>
  (assert (c ?x)))
(defrule r4
  (and (a ?x) (> ?x 1))
=>
  (assert (c ?x)))
(defrule r5
  (and (a ?x) (test (> ?x 1)))
=>
  (assert (c ?x)))
(defrule r6
  (a ?x)
  (test (and (> ?x 0) (< ?x 2)))
=>
  (assert (c ?x)))
(defrule r7
  (a ?x)
  (and (> ?x 0) (< ?x 2))
=>
  (assert (c ?x)))
```

◁

Predicados de tipo. Existen predicados de tipo predefinidos para los tipos de datos básicos. Cada uno recibe como argumento un valor y devuelve como resultado `TRUE` o `FALSE` según el valor sea o no del tipo indicado. Los predicados correspondientes a los tipos básicos estudiados en la lección 2 son:

`(integerp <expr>)` `(floatp <expr>)`

`(symbolp <expr>)` `(stringp <expr>)`.

Adicionalmente existen los predicados

`(numberp <expr>)` verdadero si `<expr>` es entero o flotante

`(lexeme <expr>)` verdadero si `<expr>` es símbolo o cadena.

Otros predicados interesantes son:

`(evenp <expr>)` verdadero si `<expr>` es un número entero par

`(oddp <expr>)` verdadero si `<expr>` es un número entero impar.

Ejemplo 5.1 Vamos a emplear predicados de tipo, funciones y restricciones para implementar una regla que suma enteros:

```
(deffacts total-e-inicial
  (total-e 0))

(deffacts valores
  (valor 10)
  (valor 10.5)
  (valor 7))

(defrule sumar-valor-entero
  "acumula en total-e la suma de valores enteros"
  ?v <- (valor ?n&:(integerp ?n))
  ?t <- (total-e ?suma)
=>
  (retract ?t ?v)
  (assert (total-e (+ ?suma ?n))))
```

donde `?n&:(integerp ?n)` puede leerse como "unifica `?n` y comprueba que sea un número entero".

Ahora vamos a sumar enteros o flotantes (lo mismo puede hacerse de forma más simple mediante `numberp`):

```
(deffacts total- inicial
  (total 0))

(defrule sumar-valor-entero-o-flotante
  "acumula la suma de valores enteros o flotantes"
  ?v <- (valor ?x&:(integerp ?x)|:(floatp ?x))
  ?t <- (total ?suma)
=>
  (retract ?t ?v)
  (assert (total (+ ?suma ?x))))
```

◁

Funciones para multicampos. Las siguientes funciones permiten manipular multicampos:

`(create$ <campo>*)`: devuelve un multicampo formado por todos los valores recibidos.

`(first$ <multicampo>)`: devuelve el primer valor de un multicampo.

`(rest$ <multicampo>)`: devuelve un valor multicampo igual al recibido tras eliminar su primer valor.

`(nth$ <n> <multicampo>)`: devuelve el n-ésimo valor de un multicampo.

`(member$ <campo> <multicampo>)`: devuelve la posición del campo dentro del multicampo, o FALSE si no se pudo encontrar.

`(length$ <multicampo>)`: devuelve el número de elementos que forman el multicampo.

Otras funciones para multicampos son `delete$`, `delete-member$`, `explode$`, `implode$`, `insert$`, `replace$`, `replace-member$`, `subseq$` y `subsetp`, cuya definición exacta puede encontrar el lector en el manual básico, sección 12.2.

Funciones para cadenas. La manipulación elemental de cadenas se lleva a cabo mediante las siguientes funciones:

`(str-cat <expresion>*)`. Concatena las expresiones (que pueden ser lexemas o números) y devuelve una cadena.

`(str-compare <expr-cadena1>) <expr-cadena2>)`. Compara las expresiones y devuelve 0 si son iguales, un entero negativo si la primera es menor según el orden lexicográfico, y un entero positivo si la primera es mayor).

`(str-index <expr-cadena1>) <expr-cadena2>)`. Comprueba si la primera expresión aparece en la segunda, y devuelve FALSE, o bien la posición en la que empieza la primera aparición de la primera expresión en la segunda.

`(str-length <expr-cadena>)`. Devuelve la longitud de la expresión.

`(sub-string <expr-entera1> <expr-entera2> <expr-cadena>)` Devuelve el trozo de la cadena comprendido entre las posiciones `expr-entera1` y `expr-entera2`, ambas incluidas.
Por ejemplo:

```
> (sub-string 3 4 (str-cat "abc" "xyz"))
 "cx"
> (str-index "cx" (sub-string 3 4 (str-cat "abc" "xyz")))
 1
> (str-compare "abc" "a")
 1
> (str-compare "a" "aa")
 -1
```

Las siguientes funciones realizan tareas bastante más complejas:

`(eval <expresion-cadena>)`

`(build <expresion-cadena>)`.

Ejecutan la cadena, tal y como si se hubiera introducido por teclado. `eval` espera una llamada a función u orden, mientras que `build` espera la definición de una construcción. `eval` no permite emplear variables locales. Ni `eval` ni `build` pueden emplearse en ficheros binarios ni en *run-times*.

`(check-syntax <expresion-cadena>)`.

Comprueba que la cadena es correcta sintácticamente, como si se hubiese leído de teclado. Devuelve `FALSE`, o bien la cadena que corresponde al mensaje de error que se generaría.
Por ejemplo:

```
> (eval (sub-string 1 7 (str-cat "(* " "2 3)()")))
 6
> (clear)
> (build "(defrule r1 (a ?x) => (assert (b ?x)))")
 TRUE
> (rules)
 r1
 For a total of 1 defrule.
> (build (sub-string 1 7 (str-cat "(* " "2 3)()")))
 FALSE
> (eval "(defrule r1 (a ?x) => (assert (b ?x)))")
 Missing function declaration for defrule.
> (check-syntax "(defrule r1 (a ?x) => (assert (b ?x)))")
 FALSE
> (check-syntax "(dephrule r1 (a ?x) => (assert (b ?x)))")
 ("
   Missing function declaration for dephrule.
  " FALSE)
```

Otras funciones predefinidas para manejar cadenas son `lowcase`, `string-to-field`, `sym-cat` y `upcase`.

Otras funciones. Existen otras funciones predefinidas en CLIPS. Citaremos por ejemplo

`(random)` - genera un número entero pseudoaleatorio.

`(seed <expresión-entera>)` - proporciona una semilla para el generador de números pseudoaleatorios. La secuencia de números pseudoaleatorios obtenida con `(random)` será siempre la misma si la semilla es también la misma.

5.2. Entrada/salida en CLIPS.

La entrada/salida se realiza a traves de "routers" o *enrutadores*. Los dispositivos son identificados por medio de nombres lógicos, que pueden ser símbolos, números o cadenas de caracteres. Para referirse a los dispositivos de entrada y salida predefinidos debe emplearse el símbolo `t`.

Las funciones que permiten leer y escribir son

- `(printout <nom-logico> <expresion>*)`. No devuelve ningún valor. Cada expresión se evalúa y se escribe, sin espacios ni líneas adicionales. El símbolo `crlf` usado como expresión forzará un salto de línea. Análogamente, los símbolos `tab` y `ff` forzarán un salto de tabulador o de página, respectivamente.

- `(read [<nom-logico>])`. Permite introducir valores desde un dispositivo. Los valores deben ser adecuados para un campo simple. Si se omite el

nombre lógico, se emplea el dispositivo predefinido de entrada. Si se intenta leer más allá del fin del fichero, se devuelve `EOF`.

- `(readline [<nom-logico>])`. Es similar a `read`, pero lee una línea completa (es decir, hasta encontrar un salto de línea o un `;`) y la devuelve como cadena de caracteres.

- `(format <nom-logico> <expresion-cadena> <expresion>*)` Permite una salida formateada según las indicaciones dadas en `<expresion-cadena>`.

Mostremos algún ejemplo del uso de estas funciones con los dispositivos estándar (lo introducido por teclado en respuesta a las funciones de lectura va precedido del símbolo "?"):

```
> (printout t "Hola a todos" crlf "Esto es un saludo" crlf)
 Hola a todos
 Esto es un saludo
> (printout t Hola a todos crlf Esto es un saludo crlf)
 Holaatodos
 Estoesunsaludo
> (+ (read) (read))
? 3
? 5
 8
>(readline)
? abd dfg
 "abd dfg"
```

Para emplear otros dispositivos de entrada/salida, deberemos hacer uso de `open` y `close`:

- `(open <nom-fichero> <nom-logico> [<modo>])`. Abre un fichero y le asigna un nombre lógico. El modo puede ser:
  `"r"` sólo lectura.
  `"w"` sólo escritura.
  `"r+"` lectura y escritura.
  `"a"` sólo añadir.
  `"wb"` sólo escritura en binario.

  Si no figura ningún modo, se supone acceso para sólo lectura.

- `(close [<nom-logico>])`. Si se llama sin argumentos, se cierran todos los ficheros abiertos. En otro caso, se cierra el fichero cuyo nombre lógico se indica.

Una pequeña sesión de ejemplo:

```
> (open "kk.txt" f1 "w")
 TRUE
> (printout f1 hola crlf)
> (printout f1 adios)
> (close f1)
> (open "kk.txt" g1)
 TRUE
> (readline g1)
 "hola"
> (read g1)
 adios
> (read g1)
 EOF
```

Remitimos al lector interesado en profundizar en estas cuestiones al manual básico, sección 12.4.

5.3. Funciones definidas por programa.

La construcción `deffunction` permite programar funciones en CLIPS. Estas funciones son interpretadas por el entorno, por lo cual no son muy eficientes. El manual advierte: "Care should be taken with recursive deffunctions; too many levels of recursion can lead to an overflow of stack memory (especially on PC-type machines)".

La construcción `deffunction` permite definir funciones que pueden ser utilizadas del mismo modo que las funciones predefinidas. Su sintaxis es la siguiente:

```
(deffunction <nombre> (<parámetro-simple>* [<parámetro-multicampo>])
  [<comentario>]
  <expresión>*)
```

Cada función puede tener cero, uno o más parámetros, cuyos nombres siguen la sintaxis de las variables simples en CLIPS. Opcionalmente, el último parámetro puede ser una variable multicampo, lo que permite construir funciones que reciban un número cualquiera de argumentos. Para que una llamada sea correcta debe aparecer exactamente el mismo número de argumentos que de parámetros simples (si no se utiliza el parámetro multicampo), o al menos el mismo número de argumentos que de parámetros simples (si se utiliza el parámetro multicampo). En este último caso todos los argumentos sobrantes se agrupan en un único valor multicampo que se asigna al parámetro multicampo.

El comentario, opcional, debe ser una cadena de caracteres.

Por último, el cuerpo de la función está formado por cero, una o más expresiones, que pueden ser llamadas a otras funciones predefinidas o definidas por el usuario u órdenes (en particular, se pueden invocar `assert` y `retract`).

Cuando se realiza una llamada a una función, los parámetros reciben ordenadamente los valores de los argumentos de la llamada. Las expresiones se evalúan entonces secuencialmente. El valor devuelto por la función coincide con el devuelto por la última de las expresiones evaluadas. Por ejemplo, sean las siguientes definiciones:

```
(deffunction sin-argumentos ()
  "función sin argumentos que escribe un saludo por pantalla"
  (printout t "Hola, soy una función sin argumentos." crlf))
(deffunction suma-3 (?x ?y ?z)
  "recibe tres números y devuelve su suma"
  (+ ?x ?y ?z))
(deffunction dos-o-mas-args (?x ?y $?resto)
  "puede recibir dos o más argumentos y los muestra"
  (printout t crlf "Valor de ?x : " ?x)
  (printout t crlf "Valor de ?y : " ?y)
  (printout t crlf "Valor de ?resto : " ?resto crlf))
```

Las siguientes llamadas producirán los resultados indicados:

```
> (sin-argumentos)
Hola, soy una función sin argumentos.
> (suma-3 1 2)
[ARGACCES4] Function suma-3 expected exactly 3 argument(s).
> (suma-3 1 2 3)
6
> (dos-o-mas-args 10)
[ARGACCES4] Function dos-o-mas-args expected at least 2 argument(s).
> (dos-o-mas-args 10 20 30 40 50)}
Valor de ?x : 10
Valor de ?y : 20
Valor de ?resto : (30 40 50)
```

5.4. Funciones procedimentales.

CLIPS proporciona un juego de funciones predefinidas con cuyo uso se puede programar en un estilo imperativo convencional. Estas funciones son `bind`, `break`, `if`, `loop-for-count`, `progn`, `progn$`, `return`, `switch` y `while`. Estas funciones pueden invocarse directamente en la interacción con el intérprete, en un fichero batch, o en cualquier punto de un programa en el que pueda aparecer una llamada a función. En particular, se suelen emplear en la parte derecha de las reglas y en la definición de funciones por programa.

La función `bind` es el equivalente a la sentencia de asignación en los lenguajes imperativos. Puede utilizarse entre las acciones de una regla para ligar una variable a un valor que se va a utilizar repetidas veces. Su sintaxis es

```
(bind <variable> <valor>)
```

Por ejemplo

```
(deftemplate objeto
  (slot tipo)
  (slot lado))
(deffacts objetos
  (objeto (tipo cuadrado) (lado 10))
  (objeto (tipo triangulo) (lado 7)))
(defrule mostrar-area-insistente
  (objeto (tipo cuadrado) (lado ?l))
=>
  (bind ?area (* ?l ?l))
  (printout t "El area del cuadrado es "  ?area  crlf)
  (printout t "Repito : "  ?area  crlf))
```

Nótese que la variable `?area` es local a la regla `mostrar-area-insistente`.

También puede emplearse la función `bind` dentro de una función definida con `deffunction`:

```
(deffunction funcion-pesada (?x)
  (bind ?c (* ?x ?x))
  (printout t "El cuadrado de " ?x " es "
    ?c ". Repito : " ?c crlf))
> (funcion-pesada 5)
El cuadrado de 5 es 25. Repito : 25
```

La función `if...then...else` permite la ejecución condicional de conjuntos de acciones. También permite definir funciones recursivas (aunque, como se ha dicho, estas no son recomendables). Su sintaxis es la siguiente:

```
(if <condición>
  then <expresión>*
  [else <expresión>*])
```

La función consta de una condición y cero, una o más expresiones detrás del símbolo `then` que se evalúan si la condición devuelve un valor distinto de FALSE . Opcionalmente pueden incluirse cero, una, o más expresiones detrás del símbolo `else`, que se evaluarán si la condición se evalúa a FALSE. La función `if...then...else` devuelve el valor de la última expresión evaluada. En caso de que no haya ninguna, o bien la condición sea falsa y no exista la cláusula `else`, devuelve FALSE.

Una nota de cautela: en general **no** se considera un buen estilo de programación incluir llamadas a funciones `if...then...else` en la parte derecha de una regla. Así, en lugar de:

```
(defrule analizar-temperatura
  (temperatura ?t)
=>
  (if (>= ?t 0)                        ;estilo deplorable
    then (assert (temperatura alta))
    else (assert (temperatura baja))))
```

es mejor utilizar dos reglas diferentes:

```
(defrule analizar-temperatura-alta    ;estilo ok
  (temperatura ?t&:(>= ?t 0))
=>
  (assert (temperatura alta)))
(defrule analizar-temperatura-baja
  (temperatura ?t&:(>= ?t 0))
=>
  (assert (temperatura baja)))
```

Por tanto, el lugar más apropiado para el empleo de estas funciones es la definición de funciones por programa. Por ejemplo, la siguiente función recursiva recibe una pregunta y una lista de cero, una o más respuestas. Realiza la pregunta al usuario hasta que su respuesta coincide con una de las proporcionadas en la lista:

```
(deffunction pregunta-r (?mensaje $?opciones)
  "muestra ?mensaje y lee la respuesta hasta que este en ?opciones"
  (printout t crlf ?mensaje)
  (bind ?result (read))
  (if (member$ ?result ?opciones)
    then ?result
    else (pregunta-r ?mensaje ?opciones)))
```

La siguiente llamada produciría el resultado indicado:

```
> (pregunta-r ">Te gusta CLIPS (S/N)? " S N)
>Te gusta CLIPS (S/N)? QUE TONTERÍA
>Te gusta CLIPS (S/N)? QUE ME DEJES
>Te gusta CLIPS (S/N)? PIÉRDETE
>Te gusta CLIPS (S/N)? s
>Te gusta CLIPS (S/N)? S
S
```

La última 'S' corresponde al valor devuelto por la función.

La función `while` permite realizar bucles condicionales. Su sintaxis es

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

Para evaluar una llamada a `while`, primeramente se evalúa `<expresión>`. Si es FALSE, se devuelve FALSE; en caso contrario, se evalúan secuencialmente las `<acción>`es y se vuelve a evaluar `<expresión>`. El proceso continúa mientras que `<expresión>` se evalúe a un valor no FALSE.

Por ejemplo, la función recursiva anterior se puede implementar mediante `while`:

```
(deffunction preguntar (?mensaje $?opciones)
  "muestra ?mensaje y lee la respuesta hasta que este en ?opciones"
  (printout t crlf ?mensaje " " $?opciones  "? ")
  (bind ?result (read))
  (while (not (member$ ?result ?opciones))
    do (printout t crlf ?mensaje " " $?opciones  "? ")
       (bind ?result (read)))
  ?result)  ;while devuelve FALSE!!
```

El lector puede consultar el manual básico de CLIPS (sección 12.6) para conocer en detalle las restantes funciones procedurales:

`loop-for-count`: iteración;
`break`: terminación prematura de bucles;
`progn` y `progn$`: secuenciación;
`return`: terminación prematura de funciones;
`switch`: selección de alternativas.

5.5. Variables globales.

La construcción `defglobal` permite definir variables globales en CLIPS. Los nombres de variables globales deben empezar por `?*` y acabar por `*`. La sintaxis de `defglobal` es la siguiente:

```
(defglobal <asignación>*)
<asignación> ==  ?*<símbolo>* = <expresión>
```

En una misma construcción pueden definirse varias variables globales. Para cada una de ellas debe proporcionarse una expresión que se evaluará para calcular el valor inicial de la variable. La sintaxis de las variables globales es igual a la de las variables normales, salvo que el nombre debe ir entre asteriscos. Los valores de estas variables pueden consultarse desde el intérprete y utilizarse en cualquier función o en la parte derecha de las reglas. Por ejemplo:

```
(defglobal ?*interes* = 7
           ?*doble-interes* = (* 2 ?*interes*))
```

```
(deffunction calcula-intereses (?saldo)
  (printout t "Intereses : " (/ (* ?saldo ?*interes*) 100) crlf))
>?*interes*
7
>?*doble-interes*
14
>(calcula-intereses 200)
Intereses : 14
```

Los valores de las variables globales pueden cambiarse utilizando la función `bind`. Cuando se ejecuta `(reset)` el valor de las variables globales vuelve a ser su valor inicial.

Las variables globales también pueden emplearse en la parte izquierda de las reglas, pero al hacerlo hay que tener en cuenta varias cosas. En primer lugar, estas variables no pueden utilizarse directamente para definir patrones. Por ejemplo, supongamos definida `(defglobal ?*temp-max* = 100)`. Entonces la siguiente regla producirá un error al cargar el programa:

```
(defrule ***REGLA-ERRONEA***
  "esta regla producira un error al cargarse"
  (temperatura ?*temp-max*)
=>
  (printout t "Peligro" crlf))
```

Lo correcto sería usar:

```
(defrule regla-correcta
  (temperatura ?t&:(= ?t ?*temp-max*))
=>
  (printout t "Peligro" crlf))
```

Además,por el hecho de cambiar el valor de una variable global no se recalcula el grafo Rete (vd. sección 2.2); por tanto, la satisfacción de los elementos condicionales sigue calculándose según el valor anterior de la variable global. Por ejemplo, en el caso anterior, si en la parte derecha de otra regla se ejecutara `(bind ?*temp-max* 200)`, el valor almacenado en el patrón de la regla `regla-correcta` seguiría siendo 100. Por ello, emplear variables globales en la parte izquierda de las reglas y cambiar su valor a lo largo de la ejecución del programa puede ser fuente de errores difíciles de depurar.

Ejercicio 5.2 ¿Recuerda el lector a nuestra amiga Cuqui, hábil ingeniera del conocimiento (ejercicio 2.9)? Tras el éxito alcanzado con la superturbotostadora, sus jefes le han encargado desarrollar un nuevo sistema, esta vez para el control de la VIP (Vaporizadora Iónica Panorámica). Esta es la entrevista celebrada con Tomás Elu, encargado de su manejo.

C.– Nunca he visto una VIP. ¿Puede decirme cómo es su panel de control?
E.– Tiene tres pantallitas en las que aparecen números y dos botones, uno azul y otro rojo.
C.– ¿Para qué sirven los botones?
E.– Los aprieto cuando la máquina se sale de su régimen normal de funcionamiento. Si ioniza demasiado, pulso el botón rojo; si ioniza demasiado poco, pulso el botón azul.
C.– ¿Cómo sabe si está ionizando demasiado?
E.– Cuando la suma de los valores de las tres pantallitas supera el valor de referencia establecido semanalmente.
C.– Ya veo. ¿Y cómo sabe si está ionizando demasiado poco?
E.– Bueno, si en alguna de las tres pantallitas aparece un número menor que 10.
C.– ¿O sea, si en una se lee -10, por ejemplo?
E.– No, los números negativos no tienen sentido. Son errores del sensor que no se toman en cuenta.
C.– Muchas gracias, señor Elu.

Se pide escribir un programa CLIPS que refleje adecuadamente el conocimiento expresado por don Tomás en esta entrevista. El programa deberá definir las funciones y variables globales convenientes para que el código resulte más legible. ◁

5.6. Saliencia.

Como ya se dijo en la sección 2.2, la *saliencia* es un número entero entre -10000 y 10000 (por defecto, 0), que establece la prioridad que tendrán las activaciones de una regla. Sintácticamente, ello se consigue por medio de una *declaración*

```
(declare (salience expr-entera>))
```

situada al inicio del `defrule`. Por ejemplo

```
(defrule regla-prioritaria
  "Las activaciones de esta regla son muy prioritarias"
  (declare (salience 10000))
  (fuego)
=>
  (huir))
```

También se dijo en la sección 2.2 que las saliencias deben usarse con parsimonia. Sin embargo, ello no quiere decir que su empleo esté siempre desaconsejado, es-

pecialmente cuando las reglas CLIPS se emplean, no para describir conocimiento declarativo de un dominio, sino para definir un algoritmo.

Por ejemplo, volvamos a la implementación del algoritmo de resolución dada en el ejercicio 4.5. Recordemos que quedó imperfecta, pues no implementamos la simplificación de las cláusulas dadas o generadas, con lo cual podía caerse en un cálculo infinito. En realidad, las reglas de simplificación de una cláusula son bien sencillas. Por ejemplo, esta sería la regla para suprimir una aparicion repetida de un literal afirmado (habría otra regla análoga para suprimir una aparicion repetida de un literal negado):

```
(defrule simplificar-1
   ?h <- (clausula $?x si ?y $?z si ?y $?v)
=>
   (retract ?h)
   (assert (clausula $?x si ?y $?z $?v)))
```

Pero, ¿qué nos garantiza que estas reglas de simplificación, aun activadas, vayan a dispararse? En ausencia de saliencias, la estrategia de resolución de conflictos puede preferir siempre otras activaciones de las infinitas que se van generando. Lo que necesitamos es pues asegurar que estas reglas, siempre que se activen, serán disparadas de inmediato. Para ello bastará declarar una saliencia mayor que la de las reglas que generan nuevas cláusulas.

Y, ya que estamos estableciendo prioridades, parece sensato generar la cláusula vacía siempre que sea posible, independientemente de que se pueda simplificar o no. Para ello le daremos una saliencia aún mayor.

Por último, parece conveniente que el programa nos informe explícitamente por pantalla en caso de que fracase. La forma más sencilla de conseguirlo es mediante una "regla por defecto", es decir, una regla que se satisface siempre, pero con una prioridad muy baja. De esta forma, se disparará solamente cuando se haya agotado el conjunto de las demás activaciones. Con todo ello, el programa quedaría

```
(defrule resolver
   "Regla de resolucion (clausula no vacia)"
   ?h1 <- (clausula $?x1 si ?y $?z1)
   ?h2 <- (clausula $?x2 no ?y $?z2)
   (test (neq ?h1 ?h2)) ;nunca resolveremos una cláusula consigo misma
   (or (test (= 0 (length $?x1)));comprobamos si algún trozo es no vacío
       (test (= 0 (length $?z1)))
       (test (= 0 (length $?x2)))
       (test (= 0 (length $?z2))))
=>
   (assert (clausula $?x1 $?z1 $?x2 $?z2)))

(defrule resolver-acabar
```

```
   "Regla de resolucion (clausula vacia y parar)"
   (declare (salience 200))
   ?h1 <- (clausula si ?y)
   ?h2 <- (clausula no ?y)
=>
   (assert (clausula))
   (printout t "EXITO!!!" crlf)
   (halt))
(defrule simplificar-1
   (declare (salience 100))
   ?h <- (clausula $?x si ?y $?z si ?y $?v)
=>
   (retract ?h)
   (assert (clausula $?x si ?y $?z $?v)))
(defrule simplificar-2
   (declare (salience 100))
   ?h <- (clausula $?x no ?y $?z no ?y $?v)
=>
   (retract ?h)
   (assert (clausula $?x no ?y $?z $?v)))
(defrule fracaso-acabar
   "Si no hay nada mas que resolver, acabar"
   (declare (salience -100))
=>
   (printout t "FRACASO." crlf))
```

El lector puede comprobar que este programa termina para cualquier entrada.

Ejercicio 5.3 Modificar el código CLIPS dado en el ejemplo 4.9 de forma que se escriban en la pantalla todos los valores de las variables que no hayan sido eliminados por el algoritmo. ◁

5.7. Algunos trucos.

¿Es posible usar CLIPS para programar tareas "vulgares" como contar, sumar, ordenar...? Evidentemente, la respuesta es afirmativa, como muestran los siguientes ejemplos.

Ejemplo 5.2 Supongamos que deseamos calcular la nota media de los alumnos de una clase.

Cualquier algoritmo se puede implementar en CLIPS. En particular, si el algoritmo se expresa en forma de reglas, la implementación será bastante directa. Expresemos pues un algoritmo para calcular la media aritmética mediante tres reglas: i) Si no se ha contado nada todavia, la suma parcial es 0 y la cuenta parcial es 0; ii) Si la suma parcial es s y la cuenta parcial es c y hay algún alumno que no se ha contabilizado cuya nota es n, hacer que la suma parcial

sea $s + n$ y la cuenta sea $c + 1$; iii) Si no quedan alumnos sin contabilizar, la media es la suma parcial dividida entre la cuenta parcial.

Supongamos que los alumnos se representan mediante hechos ordenados `(oav <id>nota <nota>)`. Supongamos además que estos hechos no son necesarios en el resto del programa; entonces podremos borrarlos cuando los contabilicemos. El programa sería

```
(defrule contar-1
  "Si no hay ningún valor parcial, inicializar a 0"
  (not (suma ?x))
=>
  (assert (suma 0))
  (assert (cuenta 0)))

(defrule contar-2
  "Si hay valores parciales y un alumno, borrar al alumno y
   borrar los antiguos valores parciales y asertar los nuevos"
  ?h1<-(suma ?s)
  ?h2<-(cuenta ?c)
  ?h3<-(oav ? nota ?x)
=>
  (retract ?h1 ?h2 ?h3)
  (assert (cuenta (+ 1 ?c)))
  (assert (suma (+ ?x ?s))))

(defrule promediar
  "Si hay valores parciales y no hay ningun alumno,
   calcular la media, asertarla e imprimirla"
  (suma ?s)
  (cuenta ?c)
  (not (oav ?a nota ?n))
=>
  (bind ?m (/ ?s ?c))
  (assert (media ?m))
  (printout t "La calificacion media es: " ?m crlf))
```

◁

Ejercicio 5.4 Suponer ahora que cada alumno pertenece a un grupo y que los alumnos se representan mediante hechos desordenados de la forma

`(alumno (id <id>) (grupo <grupo>) (nota <n>))`

Escribir un programa CLIPS que calcule las medias por grupos. ◁

Ejercicio 5.5 Modificar el código del ejemplo 5.2 para que no dé error de división por 0 en el caso de que no haya ningún alumno. ◁

Ejemplo 5.3 Supongamos ahora que deseamos ordenar a los alumnos del ejemplo 5.2 en orden decreciente (o, más exactamente, no creciente) de calificaciones. El algoritmo de ordenación por selección directa se expresa de forma natural mediante una sola regla: “si hay un valor no contabilizado y ese valor es mayor o

igual que los restantes no contabilizados, entonces seleccionarlo como el siguiente en la ordenación".

Representemos la ordenación mediante un hecho ordenado `(lista a1 a2 ...)` que contendrá los identificadores de los alumnos en orden decreciente de calificaciones. Por variar un poco respecto a lo visto en el ejemplo 5.2, vamos a suponer ahora que la inicialización se lleva a cabo en un `deffacts` que aserta el hecho `(lista)`. Además, los alumnos no pueden borrarse, pues deben seguir en memoria para otros procesamientos. Así que asertaremos hechos `(ordenado <id>)` para recordar que hemos contabilizado al alumno `<id>`. El programa será:

```
(deffacts inicio-orden
  (lista))

(defrule ordenar
  (oav ?a nota ?n)
  (not (ordenado ?a))
  ?h<-(lista $?x)
  (forall (oav ?a1 nota ?n1)
  (or (ordenado ?a1)
      (test (>= ?n ?n1))))
=>
  (assert (ordenado ?a))
  (retract ?h)
  (assert (lista $?x ?a))
  (printout t "La nota de " ?a " es " ?n crlf))
```

◁

Ejercicio 5.6 Suponer ahora que cada alumno pertenece a un grupo y que los alumnos se representan mediante hechos desordenados de la forma

`(alumno (id <id>) (grupo <grupo>) (nota <n>))`

Escribir un programa CLIPS que ordene a los alumnos primero por grupos y dentro de cada grupo por calificación. ◁

5.8. De hormigas y de hombres.

Ejercicio 5.7 ¿Recuerda el lector la historia de la hormiga de Simon (ejercicio 3.4) y las críticas de J. Weizenbaum (ejercicio 4.8)? Herbert Simon se defiende de ellas en una carta a su hija Bárbara, reproducida en su autobiografía [30], págs. 273–275:

21 de mayo de 1977

Querida Bar:

Me preguntas por Weizenbaum y todo eso. Es una historia larga y compleja, que tiene al menos tres niveles. En primer lugar, está todo el conjunto de preguntas sobre qué ha conseguido la inteligencia artificial, y qué conseguirá imitando el pensamiento humano y otros procesos humanos. Al menos en principio, se trata de una pregunta que debería responderse imparcialmente después de considerar los hechos.

En segundo lugar, están las preguntas sobre qué signican los logros de la inteligencia artificial, tanto si son grandes como si son pobres, para la sociedad humana y la gente en general. Estas son también preguntas empíricas, pero su respuesta depende en parte de las respuestas al primer conjunto de preguntas.

En tercer lugar, están las preguntas sobre qué siente la gente hacia las máquinas inteligentes y sus relaciones con la gente. Estas son preguntas sobre la emoción y los valores que no son subceptibles a prueba y demostración.

Sería bueno que pudieramos aclarar los conjuntos de preguntas primero y segundo, los basados en hechos, más o menos independientemente del tercero, los emotivos. Pero no es así como ha sido. Desde los primeros comienzos de la I.A., allá en los años cincuenta, esta ha generado mucho miedo, ansiedad, e incluso enfado en los corazones de algunas personas. En este sentido, la I.A. ha tenido un efecto muy parecido al anuncio de Darwin de la Teoría de la Evolución. Ambas hicieron surgir en algunas personas ansiedades sobre su porpia unicidad, valor y dignidad. Así, muy en los comienzos, un ingeniero llamado Mortimer Taube (creo) escribió una airada carta a [la revista] "Science" sobre el primer trabajo de GPS que Al [su amigo y colaborador Allen Newell] y yo publicamos allí, y un libro incluso más airado. Le siguió Richard Bellman... a Bellman le siguió el filósofo humanista Hubert Dreyfus [...], y a Dreyfus le siguió Weizenbaum. Ha habido, por supuesto, muchos otros, pero esos son los que consiguieron más atención.

En general no he respondido a estos ataques... No se llega muy lejos discutiendo con alguien sobre su religión, y se trata esencialmente de asuntos religiosos para los Dreyfuses y los Weizenbaums del mundo. El libro de Weizenbaum, por ejemplo, vacila entra las posturas de que (1) las afirmaciones de la I.A. están muy exageradas, (2) existe el peligro de que estas afirmaciones se lleven a cabo, (3) es inmoral que alguien intente llevarlas a cabo (a veces emplea

el término .ºbsceno", y compara a dicha gente con los Nazis). Comprendo algunos motivos por los que Joe está molesto (el mismo fue un refugiado a causa de los Nazis), pero no por qué se ha fijado en los ordenadores como el objeto de todas sus ansiedades. En cualquier caso, no creo que valga la pena discutir con él.

Mi posición ha sido esta: mi trabajo científico, y el de otros investigadores de la I.A., determinará, a la larga, cuántos de los procesos de pensamiento humanos pueden ser simulados. Creo que en el fondo todos pueden serlo, pero no siento una gran urgencia en intentar demostrarselo a otros que piensan diferente. En la ciencia, son los hechos los que nos dan la respuesta definitiva.

Con respecto a las consecuencias sociales, creo que cada investigador tiene alguna responsabilidad en valorar, y en tratar de informar a los otros de, las posibles consecuencias sociales de los productos que está intentando crear. Al y yo hemos intentado descargarnos de esa responsabilidad [en 1958] en nuestro artículo de [la revista] OR con sus predicciones, y yo también lo he intentado en las tres ediciones de mi "New Science of Management Decision" y "Sciences of the Artificial". Desde luego no puedo esperar que otros estén siempre, o a menudo, de acuerdo con mis predicciones. Algo que me hace perder la calma, sin embargo, es que me ataquen por hacerlas... Sin embargo, aunque lamento tales ataques, no me sorprenden. La calma... rara vez se conserva en las controversias científicas, y no debería esperarlo en este caso, donde los nervios que se tocan son tan sensibles...

Sólo un par de comentarios para terminar. En primer lugar, no creo que tengamos que afirmar ninguna unicidad particular del hombre, o separación del resto de la naturaleza, para encontrar valor en la vida. Personalmente, encuentro más agradable pensar que el hombre es parte de la naturaleza, que pensar que está separado y por encima de ella, aunque otros pueden tener gustos diferentes en este asunto. En segundo lugar, creo que aquellos que ponen objeciones a mi caracterización del hombre como sencillo, quieren de algún modo retener un misterio profundo en su esencia —una negación, de nuevo, de su relación integral con el resto de la naturaleza. En cuanto a mí, demostrar que algo cuyo comportamiento parece muy complejo y errático está realmente construido a partir de la combinatoria de componentes muy sencillos es hermoso, no degradante. Me parece que todo científico debería pensar así, ya que todo el propósito

de la ciencia es encontrar simplicidad con sentido en medio de la complejidad desordenada.

Pero basta de filosofía por hoy. Espero que estos comentarios te den una idea de mi reacción ante los críticos.

Con cariño,
Papá.

(*Continuará...*)

Hasta aquí el texto de H. Simon. Se sugieren las siguientes actividades:

a) Lee el texto con detenimiento.
b) Reflexiona sobre el mismo durante al menos 5 minutos.
c) Coméntalo mañana en la cafetería con tus compañeros. ◁

Capítulo 6

INTEGRACIÓN CON EL LENGUAJE C

6.1. Uso de CLIPS

Hasta ahora hemos empleado el intérprete CLIPS para desarrollar, definir y probar nuestros programas basados en reglas. Sin embargo, tal como su propio nombre indica, una de las principales características del lenguaje CLIPS (C Language Integrated Production System) es que permite su integración con otros programas escritos en lenguaje C. En realidad, la lista de lenguajes con los que es posible la integración se ha ampliado con el paso del tiempo, e incluye C++, ADA, FORTRAN y VisualBasic. El desarrollo de diversos dialectos de CLIPS, como JESS (Java Expert System Shell) y LISA (Lisp-based Intelligent Software Agents), ha ampliado la lista a los lenguajes Java y Common Lisp respectivamente.

En este capítulo analizaremos cómo integrar un programa CLIPS sencillo con programas escritos en C. Todos los programas mostrados se han compilado con GNU GCC 3.3.1 empleando el código fuente de CLIPS 6.21. Existen dos posibilidades a la hora de integrar CLIPS con otros lenguajes:

- Añadir a CLIPS funciones externas definidas en otros lenguajes.
- Permitir que un programa escrito en otros lenguaje acceda a CLIPS.

Para una descripción detallada de las posibilidades de integración de CLIPS con otros lenguajes consultar la *Guía de Programación Avanzada* de CLIPS.

6.2. Cómo incluir funciones externas definidas en lenguaje C

Es importante tener en cuenta que CLIPS y C son lenguajes con filosofías y jerarquías de tipos muy diferentes. Tal vez conviene recordar aquí que los tipos de datos más usuales en CLIPS son `lexeme` (con los subtipos `symbol` y `string`) y `number` (con los subtipos `integer` y `float`). Además, las funciones CLIPS pueden admitir un número variable de argumentos, mientras que en C este número es invariable. Por tanto, la parte más compleja a la hora de incluir en CLIPS una nueva función escrita en C está relacionada con el paso de argumentos y la recuperación del valor devuelto por la función. En esta sección analizaremos únicamente un caso sencillo, y no consideraremos por tanto el caso de funciones con número variable de argumentos ni el paso de valores multicampo como argumento.

Para emplear en CLIPS una función de usuario escrita en lenguaje C seguiremos el siguiente procedimiento:

1. Escribir la nueva función.
2. Escribir una función interfaz que:
   a) Pase el número y tipo de argumentos adecuado a la nueva función.
   b) Ejecute la nueva función.
   c) Devuelva un resultado del tipo adecuado.
3. Integrar las funciones nueva e interfaz en el código fuente de CLIPS.
4. Compilar el código fuente para crear un nuevo ejecutable CLIPS.

Supongamos que queremos definir una función que calcule el factorial de un número natural. El objetivo es poder usarla luego desde CLIPS como cualquier otra función predefinida, por ejemplo con la siguiente sintaxis:

```
> (factorial 3)
6
> (factorial 4)
24
```

Una función ingenua en C para hacer este cálculo podría ser, por ejemplo, la siguiente:

```
long factorial (n)
     long n;
{
  if (n == 0) return(1);
  else return(n * factorial(n - 1));

}
```

Una forma elegante de acceder a dicha función desde CLIPS es emplear una función interfaz encargada de recoger los argumentos y devolver el resultado. Veamos cómo.

6.2.1. Cómo se pasan los argumentos

Cuando se llama a una función externa, CLIPS guarda los argumentos en un almacenamiento intermedio, y llama a la función interfaz correspondiente *sin argumentos*. Las funciones externas son responsables de acceder al almacenamiento intermedio, determinar si el número y tipo de argumentos almacenados es correcto, y recuperar sus valores. Para ello pueden utilizarse diversas funciones:

- Para saber el número de argumentos con que se llamó a la función desde CLIPS, y que se encuentran ahora en el almacenamiento intermedio:

  ```
  int RtnArgCount();
  ```

- Para recuperar argumentos de los tipos básicos de CLIPS pueden emplearse las siguientes funciones, que reciben la posición del argumento correspondiente (estando el primero en la posición 1).

  ```
  char   *RtnLexeme(posicionArgumento);
  double  RtnDouble(posicionArgumento);
  long    RtnLong(posicionArgumento);

  int posicionArgumento;
  ```

  La función `RtnLexeme` es adecuada para recuperar argumentos tipo `symbol` o `string`. La función de usuario es responsable de reservar nuevas posiciones de memoria y copiar dichos valores si desean guardarse de forma

permanente, ya que CLIPS podría invalidar el puntero devuelto más adelante. Las funciones `RtnDouble` y `RtnLong` son adecuadas para recuperar argumentos de los tipos `integer` y `float`, que serán convertidos a los tipos C double o long, según la función empleada.

Por ejemplo, para nuestra sencilla función factorial podríamos definir una función interfaz como la siguiente:

```
interfaz_clips_factorial_1()        // función incompleta
{
  long fact, n;

  //Recuperar los argumentos:
  if (RtnArgCount != 1) return(-1L); //error
  n = RtnLong(1);

  //Realizar el cálculo:
  fact = factorial(n);

  //Devolver el resultado
  // continuará...

}
```

- También es posible recuperar argumentos de tipo desconocido. Para ello CLIPS proporciona un tipo de datos capaz de almacenar toda la información necesaria sobre el argumento (la estructura `DATA_OBJECT`), y un conjunto de funciones que permiten consultar dicha información:

```
DATA_OBJECT *RtnUnknown(posicionArgumento, &argumento);
int          GetType(argumento);

DATA_OBJECT argumento;
```

La función `RtnUnknown` recibe la posición de un argumento y un puntero a una estructura `DATA_OBJECT`, y devuelve un puntero a esa misma estructura. Como efecto secundario, la estructura recibe toda la información del argumento correspondiente.

Una vez comprobado el tipo del argumento con la función `GetType`, se puede recuperar su valor empleando las siguentes funciones:

```
char   *DOToString(argumento);
double  DOToDouble(argumento);
float   DOToFloat(argumento);
long    DOToLong(argumento);
int     DOToInteger(argumento);
```

Por ejemplo, para nuestra sencilla función factorial podríamos definir una función interfaz como la siguiente:

```
interfaz_clips_factorial_2()         // función incompleta
{
  DATA_OBJECT argumento;
  long fact, n;

  //Recuperar los argumentos:
  if (RtnArgCount != 1) return(-1L);  //error
  RtnUnknown(1, &argumento);
  if (GetType(argumento) == INTEGER)
    n = DOToLong(argumento);
  else
    return(-1);   //error

  //Realizar el cálculo:
  fact = factorial(n);

  //Devolver el resultado
  // continuará...

}
```

Veamos ahora cómo devolver a CLIPS el resultado calculado por una función.

6.2.2. Cómo se devuelven los resultados

Una vez que la función interfaz ha realizado el cálculo correctamente, es necesario devolver el resultado a CLIPS. Algunos tipos de datos pueden de-

volverse directamente sin problemas. Este es el caso, por ejemplo, de `double`, `float`, `integer`, `long integer`, o `character`. En otros casos, como los tipos `symbol`, `string`, o los valores lógicos (`TRUE`, `FALSE`), deben tenerse en cuenta consideraciones especiales. Trataremos a continuación todos estos casos. Para los restantes puede consultarse la *Guía de Programación Avanzada* de CLIPS.

Ejemplo 1: función factorial

Comencemos por el caso más sencillo. Cualquiera de las funciones interfaz definidas en 6.2.1 puede completarse añadiendo la orden `return` correspondiente, ya que el valor de vuelto es de tipo `long`. Por ejemplo:

```
long interfaz_clips_factorial()  //interfaz CLIPS de factorial
{
  long n, resultado;

  //Recuperar los argumentos:
  if (RtnArgCount() != 1) return(-1L); //error
  n = RtnLong(1);

  //Realizar el cálculo:
  resultado = factorial(n);

  //Devolver el resultado:
  return(resultado);
}
```

Ejemplo 2: función str_space

En el caso de funciones que devuelven lexemas, es necesario tener en cuenta que CLIPS almacena todas las cadenas y símbolos utilizados en una *tabla de símbolos*. Antes de devolver cualquier objeto de estos tipos, es preciso incluirlo previamente en la tabla de símbolos empleando la siguiente función, que devuelve un puntero al objeto en cuestión.

```
VOID *AddSymbol(string);
char *string;
```

Por ejemplo, supongamos que deseamos enriquecer CLIPS con una nueva función `str_space`, que reciba un número entero n, y devuelva una cadena con

n espacios en blanco. Desde CLIPS esta función podrá usarse con la siguiente sintaxis:

```
>(str_space 2)
"  "
>(str_space 0)
""
>(str_space 10)
"          "
```

La función C podría ser:

```
void str_space(l,s) //llena s con l espacios
     int l;
     char s[l];
{int i;
 for(i=0; i < l-1; i++) s[i] = ' ';}
```

La función interfaz puede definirse entonces de la siguiente forma:

```
#include <stdlib.h>                //malloc, free
VOID *interfaz_clips_str_space()   //interfaz CLIPS de str_space
{
  long  n;
  char *cadena;
  DATA_OBJECT argumento;
  VOID *resultado;

  //Recuperar el argumento:
  if (RtnArgCount() != 1) return(-1L);  //error
  RtnUnknown(1, &argumento);
  if (GetType(argumento) == INTEGER)
    n = DOToLong(argumento);
  else
    return(-1);                       //error

  //Realizar el cálculo:
  cadena = (char *) malloc(n+1);
  str_space(n, cadena);

  //Devolver el resultado
```

```
  resultado = AddSymbol(cadena);
  free(cadena);
  return(resultado);
}
```

¿Cómo distingue CLIPS si el valor devuelto debe interpretarse como una cadena o un símbolo? Este misterio quedará resuelto en la sección 6.2.3 cuando expliquemos cómo declarar las funciones interfaz.

Ejemplo 3: función zerop

Por último consideremos el caso de un predicado, es decir, una función que devuelve los valores `TRUE` o `FALSE`. En este caso, la función puede devolver uno de los símbolos CLIPS predefinidos:

```
VOID *FalseSymbol;
VOID *TrueSymbol;
```

A modo de ejemplo, la función `zerop` recibirá un número (entero o flotante) y devolverá `TRUE` si el número es cero, y `FALSE` en otro caso:

```
>(zerop (mod 120 5))
TRUE
>(zerop (/ 5 3))
FALSE
>(zerop (sin 0))
TRUE
```

Las funciones C correspondientes podrían ser:

```
int zeropf(f)
     float f;
{return(f == 0.0);}

int zeropi(i)
     long i;
{return(i == 0L);}
```

La función interfaz puede definirse entonces de la siguiente forma:

```
VOID *interfaz_clips_zerop()        //interfaz CLIPS de zerop
{
```

```
  long  n;
  float f;
  DATA_OBJECT argumento;
  int resultado;

  //Recuperar el argumento:
  if (RtnArgCount() != 1) return(-1);  //error
  RtnUnknown(1, &argumento);
  switch (GetType(argumento))
    {
    case INTEGER : n = DOToLong(argumento);
      break;
    case FLOAT   : f = DOToFloat(argumento);
      break;
    default      : return(-1);      //error
    }

  //Realizar el cálculo:
  switch (GetType(argumento))
    {
    case INTEGER : resultado = zeropi(n);
      break;
    case FLOAT   : resultado = zeropf(f);
      break;
    }

  //Devolver el resultado
  if (resultado)  return(TrueSymbol);
  else            return(FalseSymbol);
}
```

6.2.3. Cómo integrar las nuevas funciones en CLIPS

Para poder utilizar nuevas funciones externas en CLIPS es necesario declararlas en el lugar adecuado, y compilarlas junto a los ficheros fuente de CLIPS. Esto producirá un nuevo ejecutable del intérprete donde las nuevas funciones son accesibles.

La declaración de las nuevas funciones debe realizarse dentro de la función `UserFunctions`, que normalmente se encuentra definida en el fichero `main.c`

Código	Tipo de datos
c	character
d	double precission float
f	single precission float
i	integer
l	long integer
s	string
w	symbol

Cuadro 6.1: Algunos códigos de tipos de datos devueltos por funciones externas.

que acompaña los ficheros fuente de CLIPS. Concretamente, puede realizarse una llamada a la función `DefineFunction` por cada nueva función externa:

```
int DefineFunction(nombreFuncion, tipoFuncion, punteroFuncion,
nombreRealFuncion);

char *nombreFuncion, tipoFuncion, *nombreRealFuncion;
int  (*punteroFuncion)();
```

Veamos el significado de los argumentos que recibe `DefineFunction`:

- `nombreFuncion` es una cadena con el nombre que se utilizará desde CLIPS para llamar a la función externa.
- `tipoFuncion` es un caracter que indica el tipo del valor que la función devolverá a CLIPS (véase la tabla 6.1).
- `punteroFuncion` es un puntero a la función interfaz correspondiente a la función externa (una declaración `extern` de la función puede ser apropiada).
- `nombreRealFuncion` es una cadena de caracteres con el nombre de la función interfaz correspondiente a la función externa.

Es importante destacar que las funciones definidas por el usuario tienen precedencia sobre las predefinidas. Si el usuario define una nueva función con el mismo nombre que otra predefinida, la segunda queda anulada.

Podemos entonces declarar las funciones de ejemplo presentadas en las secciones 6.2.1 y 6.2.2 de la siguiente forma:

```
void UserFunctions()
  {
    extern long  interfaz_clips_factorial();
    extern VOID *interfaz_clips_str_space();
    extern VOID *interfaz_clips_zerop();

    DefineFunction("factorial",
                   'l',
                   PTIF interfaz_clips_factorial,
                   "interfaz_clips_factorial");

    DefineFunction("str_space",
                   's',
                   PTIF interfaz_clips_str_space,
                   "interfaz_clips_str_space");

    DefineFunction("zerop",
                   'w',
                   PTIF interfaz_clips_zerop,
                   "interfaz_clips_zerop");
  }
```

Por último, basta compilar y linkar todos los ficheros fuente de CLIPS con los ficheros fuente que contengan las definiciones necesarias para las nuevas funciones externas. Es importante señalar que todos los ficheros donde se empleen funciones CLIPS (en nuestro caso, en las funciones de interfaz) deben ir precedidas por la declaración,

```
#include <clips.h>
```

6.3. Cómo usar CLIPS desde un programa C

Una de las posibilidades más interesantes de CLIPS es la de poder ser manipulado directamente desde un programa C. Obviamente, las necesidades de integración variarán de una aplicación a otra. Presentaremos a continuación algunos esquemas generales que pueden resultar útiles como guía para casos más particulares.

6.3.1. Cómo ejecutar órdenes básicas

Para que un programa C pueda acceder a CLIPS será necesario disponer de los ficheros fuente de CLIPS.

El fichero principal del programa C deberá incluir las siguientes declaraciones:

```
#include <stdio.h>
#include "clips.h"
```

Esto da acceso a muchas de las órdenes, funciones y construcciones disponibles en el entorno CLIPS. En nuestro ejemplo emplearemos las siguientes:

- Iniciar el entorno CLIPS. Todo programa que utilice CLIPS debe llamar esta función una sola vez, y antes de realizar ninguna llamada a otras funciones CLIPS.

  ```
  void InitializeEnvironment();
  ```

- Manipulación básica del entorno CLIPS. Las siguientes funciones proporcionan la funcionalidad de las órdenes `clear y reset`.

  ```
  void Clear();
  void Reset();
  ```

- Carga de construcciones CLIPS. La siguiente función permite cargar un fichero con construcciones CLIPS de modo análogo a la orden `load`. Recibe como argumento una cadena con el nombre del fichero a cargar. Devuelve un valor entero: 0 si no se pudo abrir el fichero; -1 si se abrió el fichero pero ocurrió un error al cargarlo; 1 si se abrió el fichero y no hubo errores en la carga. Si se producen errores sintácticos, estos se escriben en *werror* y continúa la carga.

  ```
  int Load(fileName);
  char *fileName;
  ```

- Ejecución de programas CLIPS. La siguiente función combina la funcionalidad de las órdenes `run y step`. Acepta como argumento el número de reglas a disparar, o un número negativo si se desea ejecutar el programa hasta que la agenda quede vacía. Devuelve el número de reglas disparadas.

  ```
  long int Run(runLimit);
  long int runLimit;
  ```

Ejemplo: ejecución de CLIPS desde C

La siguiente función ilustra la interacción básica con CLIPS: inicia el entorno, carga un programa y lo ejecuta. Por simplicidad, el nombre del fichero se ha definido como una constante.

```
#define PATH_FICHERO_CLP  "c:\\prueba.clp "
#include <stdio.h>
#include "clips.h"

main()    //main ejecución de CLIPS desde C
{
  InitializeEnvironment();
  Load(PATH_FICHERO_CLP);
  Reset();
  Run(-1L);
}
```

La función `main` reemplazará a la incluida en el fichero `main.c` proporcionado en los ficheros fuente de CLIPS. A continuación se deben compilar y linkar todos los ficheros para obtener un nuevo ejecutable. Puede probarse con el siguiente fichero `prueba.clp`:

```
(defrule hola
  (initial-fact)
=>
  (printout t "Hola C" crlf))
```

6.3.2. Cómo manipular la lista de hechos

Salvo en los casos más sencillos, el programador C deseará una interacción más sofisticada con el entorno CLIPS. En particular se puede controlar la ejecución del programa CLIPS añadiendo hechos en la lista de hechos. También será frecuente la necesidad de consultar el contenido de la lista de hechos una vez completada la ejecución.

La función `AssertString` permite añadir un hecho definido previamente mediante una cadena de caracteres.

```
VOID *AssertString(cadena);
char *cadena;
```

El resultado devuelto por `AssertString` es un puntero a una estructura que corresponde al hecho asertado. Posteriormente, el hecho podría eliminarse de la lista de hechos utilizando dicho puntero y la siguiente función:

```
int  Retract(prtHecho);
VOID *ptrHecho;
```

La consulta de la lista de hechos puede requerir funciones muy diversas dependiendo de si son ordenados o con plantilla, o si contienen o no multicampos. Trataremos sólo el caso de hechos con plantilla y sin multicampos. Para ello bastará normalmente con utilizar las siguientes funciones:

- `GetNextFact` recibe un puntero a un hecho, y devuelve el siguiente de la lista. Para acceder al primero se puede llamar a la función con NULL. Si no hay más hechos en la lista devuelve NULL.

  ```
  VOID *GetNextFact(ptrHecho);
  VOID *ptrHecho;
  ```

- `GetFactPPForm` recibe un array de caracteres (buffer), el número máximo de caracteres que puede contener (sin incluir el caracter nulo final), y un puntero a un hecho. Como efecto secundario guarda en el array una representación del hecho que puede ser impresa posteriormente.

  ```
  VOID *GetFactPPForm(buffer, longBuffer, ptrHecho);
  char *buffer;
  int   longBuffer;
  VOID *ptrHecho;
  ```

- `GetFactSlot` recibe un puntero a un hecho con plantilla, una cadena con el nombre de un slot, y un puntero a una estructura genérica `DATA_OBJECT`. Como efecto secundario, la estructura `DATA_OBJECT` recibe el valor del slot. Además devuelve verdadero o falso según la operación tuviera éxito o no.

  ```
  int   GetFactSlot(ptrHecho, nomSlot, &elValor);
  VOID *ptrHecho;
  char *nomSlot;
  DATA_OBJECT elValor;
  ```

Para recuperar el valor de la estructura `DATA_OBJECT` pueden usarse las funciones ya descritas en la sección 6.2.1.

- `FactDeftemplate` recibe un puntero a un hecho y devuelve otro a la estructura deftemplate que le corresponde.

```
VOID *FactDeftemplate(ptrHecho);
VOID *ptrHecho;
```

- `GetDeftemplateName` recibe un puntero a una estructura `deftemplate` y devuelve una cadena que contiene el nombre del deftemplate.

```
char *GetDeftemplateName(ptrDeftemplate);
VOID *ptrDeftemplate;
```

Ejemplo: modificación y consulta de la lista de hechos

Consideremos que el siguiente programa CLIPS se encuentra en el fichero `c:\\prueba2.clp`:

```
(deftemplate numero (slot valor (type INTEGER FLOAT)))
(deftemplate par    (slot valor (type INTEGER)))
(deftemplate impar  (slot valor (type INTEGER)))

(defrule numero-par
  ?h <- (numero (valor ?v))
  (test (= 0 (mod ?v 2)))
=>
  (retract ?h)
  (assert (par (valor ?v))))

(defrule numero-impar
  ?h <- (numero (valor ?v))
  (test (= 1 (mod ?v 2)))
=>
  (retract ?h)
  (assert (impar (valor ?v))))
```

El siguiente programa carga un programa CLIPS, añade diversos hechos de tipo `numero` a la lista de hechos, ejecuta el programa y muestra el contenido final de la lista de hechos por pantalla.

```
#define PATH_FICHERO_CLP  "c:\\prueba2.clp "
#include <stdio.h>
#include "clips.h"

main()    //main ejecución de CLIPS desde C
{
  InitializeEnvironment();
  Load(PATH_FICHERO_CLP);
  Reset();
  aserta_numero_entero(2);
  aserta_numero_entero(5);
  aserta_numero_entero(7);
  aserta_numero_flotante(2.5);

  Run(-1L);

  procesa_lista_hechos();
}
//-------------------------------------------------------
aserta_numero_entero (n)
     int n;
{char cadena[100];

 sprintf(cadena, "(numero (valor %d))", n);
 AssertString(cadena);
}

aserta_numero_flotante (f)
     float f;
{char cadena[100];

 sprintf(cadena, "(numero (valor %f))", f);
 AssertString(cadena);
}
```

```
//-------------------------------------------------------
char *NomFactDeftemplate(ptrHecho) //nom. del deftempl. del hecho
     VOID *ptrHecho;
{
  return(GetDeftemplateName(FactDeftemplate(ptrHecho)));
}

//-------------------------------------------------------
#include <string.h>     //strcmp
procesa_lista_hechos()
{
  VOID *ptrHecho1, *ptrHecho2;
  DATA_OBJECT   valor;
  int   n;
  float f;

  ptrHecho2 = GetNextFact(NULL);
  while (ptrHecho2 != NULL)
    {
      if (strcmp(NomFactDeftemplate(ptrHecho2), "par") == 0)
        {
          GetFactSlot(ptrHecho2, "valor", &valor);
          n = DOToInteger(valor);
          printf("Número par: %d\n", n);
        }
      else if (strcmp(NomFactDeftemplate(ptrHecho2), "impar")  == 0)
        {
          GetFactSlot(ptrHecho2, "valor", &valor);
          n = DOToInteger(valor);
          printf("Número impar: %d\n", n);
        }
      else if (strcmp(NomFactDeftemplate(ptrHecho2), "numero")  == 0)
        {
          GetFactSlot(ptrHecho2, "valor", &valor);
          f = DOToFloat(valor);
          printf("Número real: %f\n", f);
        }
      else
        printf("Hecho desconocido\n");
```

```
      //siguiente hecho
      ptrHecho1 = ptrHecho2;
      ptrHecho2 = GetNextFact(ptrHecho1);
    }
}
```

La ejecución del programa provocará la siguiente salida por pantalla:

```
Hecho desconocido
Número real: 2.500000
Número impar: 7
Número impar: 5
Número par: 2
```

El hecho desconocido corresponde al hecho por defecto `(initial-fact)`, que también se encuentra presente en la lista de hechos.

6.3.3. Cómo consultar variables globales

Por último mencionaremos un método más simple para consultar el resultado de la ejecución de un programa CLIPS. Con frecuencia dicho resultado se reduce a un solo valor, por lo que no es necesario recuperar toda la lista de hechos.

En tales casos el programa CLIPS puede facilitar la labor guardando el valor de resultado en una variable global. Los valores de las variables globales pueden consultarse fácilmente desde un programa C empleando la siguiente función:

```
int         GetDefglovalValue(nomVar, &vPtr);
char        nomVar;
DATA_OBJECT vPtr;
```

El nombre de la variable `nomVar` será una cadena de caracteres con el mismo nombre de la variable global tras eliminar el signo `?` y los asteriscos inicial y final. Es decir, si el nombre de la variable es `?*resultado*`, la cadena de caracteres será `"resultado"`.

La función devolverá como resultado 0 (si no se encontró la variable global) o 1 (si la operación se realizó con éxito). Como efecto secundario, la variable `vPtr` recibirá la información relativa al valor de la variable global.

6.4. Guía rápida

Concluiremos este capítulo con un resumen de las funciones que hemos estudiado, todas ellas disponibles en CLIPS 6.21. La *Guía de programación avanzada* de CLIPS proporciona una relación completa de todas las funciones disponibles para los programadores C.

Paso de argumentos

```
int RtnArgCount();
```

```
char   *RtnLexeme(posicionArgumento);
double  RtnDouble(posicionArgumento);
long    RtnLong(posicionArgumento);

int posicionArgumento;
```

```
DATA_OBJECT *RtnUnknown(posicionArgumento, &argumento);
int GetType(argumento);
char   *DOToString(argumento);
double  DOToDouble(argumento);
float   DOToFloat(argumento);
long    DOToLong(argumento);
int     DOToInteger(argumento);

DATA_OBJECT argumento;
```

Devolución de resultados

```
VOID *AddSymbol(string);
char *string;
```

```
VOID *FalseSymbol;
VOID *TrueSymbol;
```

Declaración de funciones externas

```
int DefineFunction(nombreFuncion, tipoFuncion, punteroFuncion,
nombreRealFuncion);

char *nombreFuncion, tipoFuncion, *nombreRealFuncion;
int  (*punteroFuncion)();
```

Órdenes básicas del entorno CLIPS

```
void InitializeEnvironment();
```

```
void Clear();
void Reset();
```

```
int Load(fileName);
char *fileName;
```

```
long int Run(runLimit);
long int runLimit;
```

Gestión de la lista de hechos
`VOID *AssertString(cadena);` `char *cadena;`
`int  Retract(prtHecho);` `VOID *ptrHecho;`
`VOID *GetNextFact(ptrHecho);` `VOID *ptrHecho;`

`VOID *GetFactPPForm(buffer, longBuffer, ptrHecho);`
`char *buffer;`
`int  longBuffer;`
`VOID *ptrHecho;`

`int  GetFactSlot(ptrHecho, nomSlot, &elValor);`
`VOID *ptrHecho;`
`char *nomSlot;`
`DATA_OBJECT elValor;`

`VOID *FactDeftemplate(ptrHecho);` `VOID *ptrHecho;`
`char *GetDeftemplateName(ptrDeftemplate);` `VOID *ptrDeftemplate;`

Acceso a variables globales

```
int         GetDefglovalValue(nomVar, &vPtr);
char        nomVar;
DATA_OBJECT vPtr;
```

Capítulo 7

INTEGRACIÓN CON EL LENGUAJE Java

7.1. Jess y Java

Trataremos en este capítulo sobre la integración de Jess (Java Expert System Shell) con el lenguaje Java. Jess es un intérprete de reglas inspirado en CLIPS, pero implementado integramente en Java. Aunque existe una gran coincidencia entre Jess y CLIPS se trata en definitiva de lenguajes distintos, de modo que existen características de CLIPS que no están disponibles en Jess y viceversa. No obstante las construcciones básicas (hechos y reglas) son idénticas en ambos lenguajes.

Todos los programas mostrados se han compilado empleando Java 2 SDK Standard Edition v1.4.2_02 y el código fuente de Jess 6.1p5. Las posibilidades de integración de Jess y Java son múltiples. Tal como indica Ernest Friedman-Hill en "Jess, The Expert System Shell for the Java Platform", "el paso más importante en el desarrollo de una aplicación Jess es elegir una arquitectura de entre el rango de posibilidades casi ilimitado". Nos centraremos aquí en dos casos frecuentes:

- Añadir a Jess funciones externas definidas en lenguaje Java.
- Permitir que un programa escrito en Java acceda a Jess.

Para ello trabajaremos con algunas de las múltiples clases predefinidas en Jess:

- `jess.Rete`, el motor de inferencias de Jess, encargado de coordinar todos los aspectos relacionados con la ejecución de programas Jess.
- `jess.RU`, es una clase sin constructor que contiene distintos métodos y campos `static`. Se trata de utilidades generales para la clase `jess.Rete`.
- `jess.Value`, representa un valor junto con su tipo en Jess. En general, estos valores pueden ser estáticos (átomos, números, cadenas) o dinámicos (variables, llamadas a funciones).
- `jess.ValueVector`, un vector que contiene objetos de la clase `jess.Value`.
- `jess.Fact`, que representa un hecho en Jess.
- `jess.Context`, un contexto de ejecución. Más concretamente, un contexto representa un ámbito en el que puede haber variables declaradas. Los contextos se emplean para interpretar, o *resolver*, el valor de objetos de la clase `jess.Value` que representan variables u otras expresiones dinámicas.
- `JessException` es la clase padre de todas las excepciones lanzadas por los métodos públicos de la biblioteca de clases Jess.

7.2. Cómo incluir nuevas funciones en Jess

Para añadir una nueva función al intérprete Jess es necesario seguir el siguiente procedimiento:

1. Escribir una nueva clase que implemente la interfaz `jess.Userfunction`. De este modo la clase representará una nueva función Jess.
2. Crear una única instancia de la clase anterior e instalarla en un objeto `jess.Rete` empleando el método `Rete.addUserfunction()`.

Supongamos que queremos definir una función que calcule el factorial de un número natural. El objetivo es poder usarla luego desde Jess como cualquier otra función predefinida, por ejemplo con la siguiente sintaxis:

```
> (factorial 3)
6
> (factorial 4)
24
```

Nótese que el argumento de la llamada puede ser una constante, como en los ejemplos anteriores, o una expresión dinámica, como la variable `?n` en el siguiente ejemplo:

```
(defrule aserta-fact-de-n
  (numero ?n)
 =>
  (assert (fact-de-n ?n (factorial ?n))))
```

La interfaz `jess.Userfunction` posee únicamente dos métodos:

- `getName()` devuelve simplemente una cadena con el nombre con el que se utilizará la función en Jess.
- `call()` debe encargarse de realizar el cálculo de la función y devolver el resultado adecuado. Recibirá como argumento un objeto `ValueVector` que contendrá una representación de la llamada a la función realizada desde Jess. De este modo, el primer elemento del `ValueVector` será un símbolo (tipo `RU.ATOM`) con el nombre de la función (lo que permite que una sola clase gestione todas las funciones externas), y los restantes serán objetos `Value` correspondientes a los distintos argumentos de la llamada. Estos últimos pueden ser estáticos (átomos, números, cadenas) o dinámicos (variables u otras expresiones que deben ser interpretadas en el contexto de la llamada).

El siguiente fichero Factorial.java puede situarse en el mismo directorio que Main.java:

```
package jess;
import jess.*;

public class Factorial implements Userfunction
{
  //Nótese que no es necesario definir constructor

  //Funciones de la interface Userfunction
  public String getName() { return "factorial"; }

  public Value call (ValueVector vv, Context c) throws JessException
   {
    //recuperamos el argumento
```

```
    //realizamos el cálculo

    //devolvemos el resultado
    //(continuará...)

   }

  //Función factorial
  private long factorial (int n)
   {
    if (n == 0) return 1;
    else return n * factorial(n - 1);
   }
}
```

Veamos más detalladamente los aspectos relacionados con la recuperación de los argumentos y la devolución del resultado.

7.2.1. Cómo se pasan los argumentos

Cuando se llama a una función externa, Jess guarda una representación de la llamada en un objeto `ValueVector`, donde su primer elemento es un símbolo (tipo `RU.ATOM`) correspondiente al nombre de la función, y los demás son objetos `Value` correspondientes a los distintos argumentos de la llamada (constantes, variables, o llamadas a otras funciones). A continuación se llama al método `call` de la clase correspondiente a la función, pasándole el objeto `ValueVector` y un contexto. Este contexto servirá para interpretar el valor de los argumentos en caso de que se trate de variables o llamadas a otras funciones.

Los objetos `Value` representan datos autodescritos. Veamos algunos métodos útiles sobre objetos `Value`:

- `type()` devuelve el tipo del dato representado. Concretamente el tipo será una de las siguientes constantes definidas en la clase `jess.RU`: `NONE`, `ATOM`, `STRING`, `INTEGER`, `VARIABLE`, `FACT`, `FLOAT`, `FUNCALL`, `LIST`, `DESCRIPTOR`, `EXTERNAL_ADDRESS`, `INTARRAY`, `MULTIVARIABLE`, `SLOT`, `MULTISLOT`, `LONG`.

- Los siguientes métodos permiten extraer el dato, interpretándolo en su contexto si es necesario:

```
public Object  externalAddressValue(Context c) throws JessException
public String  stringValue         (Context c) throws JessException
public Fact    factValue           (Context c) throws JessException
public Funcall funcallValue        (Context c) throws JessException
public ValueVector listValue       (Context c) throws JessException
public double  floatValue          (Context c) throws JessException
public double  numericValue        (Context c) throws JessException
public int     intValue            (Context c) throws JessException
```

Vale la pena señalar aquí que existe una clase `LongValue`, heredera de `Value`, que permite almacenar valores del tipo `long` de Java. Para extraer el dato de esta clase puede emplearse el método:

```
public long     longValue            (Context c) throws JessException
```

Por ejemplo, para nuestra sencilla función factorial podríamos definir el método `call()` del siguiente modo:

```
//función incompleta
public Value call (ValueVector vv, Context c) throws JessException
  {int n;
  long f;

  //recuperamos el argumento
  n = vv.get(1).intValue(c);

  //realizamos el cálculo
  f = factorial(n);

  //devolvemos el resultado
  //continuará...
 }
```

Veamos ahora cómo devolver a Jess el resultado calculado por una función.

7.2.2. Cómo se devuelven los resultados

Una vez que la función interfaz ha realizado el cálculo correctamente, es necesario devolver el resultado a Jess. En general, se construirá y devolverá siempre

un objeto `Value` adeduado. En caso de que el valor a devolver sea `nil`, debe emplearse la variable `public static final jess.Funcall.NIL`.

Los siguientes constructores permiten crear objetos `Value` que representen distintos tipos de datos:

```
public Value(boolean b)                throws JessException
public Value(String s, int type)       throws JessException
public Value(double d)                 throws JessException
public Value(int value, int type)      throws JessException
```

El primer constructor permite construir uno de los objetos `TRUE` o `FALSE`. En el caso del segundo constructor, el valor del parámetro `type` puede ser, entre otros, `RU.ATOM` o `RU.STRING`. En el cuarto, el valor del parámetro `type` puede ser `RU.INTEGER`.

Para crear objetos de la clase `LongValue` puede emplearse el constructor:

```
public LongValue(long l)               throws JessException
```

De este modo, podemos completar el método `call` de nuestra clase `Factorial` del siguiente modo:

```
  public Value call (ValueVector vv, Context c) throws JessException
    {int n;
    long f;
    JessException ex;

    //recuperamos el argumento (producirá una excepción si el
    //número de argumentos es incorrecto)
    if (vv.size() != 2)
        {
            ex = new JessException("Factorial.call",
                                   "Número incorrecto de argumentos",
                                   "");
            throw ex;
        }
    n = vv.get(1).intValue(c);

    //realizamos el cálculo
    f = factorial(n);
```

```
    //devolvemos el resultado
    return new LongValue(f);
  }
```

Nótese que hemos completado la función comprobando que la longitud del `ValueVector vv` es 2 (el nombre de la función y el argumento). En caso contrario utilizamos el siguiente constructor de objetos `JessException` para lanzar una excepción:

```
public JessException (java.lang.String rutina,
                      java.lang.String mensaje,
                      java.lang.String datos)
```

donde `rutina` es una cadena con el nombre de la rutina donde se produjo la excepción, `mensaje` es una cadena con un mensaje informativo sobre la excepción, y `datos` es una cadena con datos adicionales que se concatena a la anterior.

7.2.3. Cómo integrar las nuevas funciones en Jess

Para poder utilizar nuevas funciones externas en Jess es necesario llamar a `Rete.addUserfunction()` con una instancia de la nueva clase como argumento. Por ejemplo, la función `main` de la clase `Main` en el fichero Main.java, se puede sustituir por el siguiente código:

```
  public static void main(String[] argv) {
      Main m = new Main();
      Rete r = new Rete();

      r.addUserfunction(new Factorial()); //declaración
      m.initialize(argv, r);
      m.execute(true);
  }
```

Por supuesto, será necesario compilar los ficheros Factorial.java y Main.java. Es importante señalar que todos los ficheros donde se empleen clases de Jess deben ir precedidas por la declaración,

```
import jess.*;
```

7.3. Cómo usar Jess desde un programa Java

Una de las posibilidades más interesantes de Jess es la de poder ser manipulado directamente desde un programa Java. Obviamente, las necesidades de integración variarán de una aplicación a otra. Presentaremos a continuación algunos esquemas generales que pueden resultar útiles como guía para casos más particulares.

7.3.1. Cómo ejecutar órdenes básicas

Para que un programa Java pueda acceder a Jess será necesario disponer de los ficheros con las clases de Jess. De entre ellas, la más importante es sin duda la clase `Rete`. Cada nueva instancia de la clase `Rete` es un nuevo intérprete de reglas, con su propia lista de hechos, agenda, variables globales, etc.

Algunas de las órdenes más frecuentes de Jess pueden invocarse como métodos de la clase `Rete`. Entre ellos se encuentran `run()`, `run(int)`, `reset()`, `clear()`, y `halt()`

Otro método muy importante de la clase `Rete` es `executeCommand(String)`, que permite ejecutar cualquier orden Jess que se le pasa mediante una cadena de caracteres.

Ejemplo: ejecución de Jess desde Java

La siguiente función ilustra la interacción básica con Jess: crea una nueva instancia del intérprete Jess, carga un programa y lo ejecuta. Por simplicidad, el nombre del fichero a cargar se ha definido como una variable privada.

```
//Ejemplo1.java

package jess;

import jess.*;
public class Ejemplo1  {

    private static String nomf = "\"c:\\\\hola.clp\"";

    public static void main(String[] argv) throws JessException {

        Rete r = new Rete();
```

```
        r.executeCommand("(batch " + nomf + ")");
        r.reset();
        r.run();
    }
}
```

Nótese que la orden `load` de CLIPS no existe en Jess con tal nombre. La orden equivalente es `batch`. Puede probarse con el siguiente fichero `prueba.clp`:

```
(defrule hola
  (initial-fact)
=>
  (printout t "Hola Java" crlf))
```

7.3.2. Cómo intercambiar valores: mecanismo store/fetch

Salvo en los casos más sencillos, el programador Java deseará una interacción más sofisticada con el entorno Jess. En esta sección estudiaremos un método sencillo para intercambiar valores entre Jess y un programa Java. Esto nos permite controlar la ejecución de un programa Jess y obtener un resultado del mismo.

Los objetos `jess.Rete` poseen una tabla en la que es posible guardar y consultar valores indexados por un nombre. Para acceder a dicha tabla desde Java pueden emplearse los siguientes métodos de la clase `jess.Rete`:

```
public Value store(String nombre, Value val);
public Value store(String nombre, Object val);
public Value fetch(String nombre);
public void  clearStorage();
```

Las siguientes funciones permiten realizar las mismas operaciones desde un programa Jess:

```
(store <nombre>, <valor>)
(fetch <nombre>)
(clearStorage)
```

En ambos casos, la operación `store` permite guardar un nuevo valor en la tabla indexándolo con un nombre. Además `store` devolverá el anterior valor asociado al nombre o `null/nil` si no tenía ninguno. Asociar el valor null (en Java) o nil (en Jess) a un nombre equivale a borrarlo de la tabla.

La operación `fetch` permite consultar el valor asociado a un nombre en la tabla, mientras que `clearStorage` borra todo el contenido de la tabla.

Ejemplo: paso de valores entre Jess y Java

Supongamos un programa Jess que recibe dos valores de entrada y produce otro como resultado. Nos centraremos aquí en la forma de comunicar dichos datos, independientemente del proceso que se realice con ellos. Los datos de entrada serán dos hechos de la forma:

```
(dato1 <valor>)
(dato2 <valor>)
```

El programa generará como resultado un hecho de la forma:

```
(resultado <valor>)
```

La comunicación entre nuestro programa Java y Jess se realizará de la siguiente forma:

1. Emplear `store` para almacenar los dos valores de `dato1` y `dato2`.
2. Cargar y ejecutar el programa Jess. Este se encargará de recuperar los valores, realizar el cálculo, y emplear `store` para almacenar el valor `resultado`.
3. Emplear `fetch` para recuperar el resultado calculado por el programa Jess.

Consideremos que el siguiente programa CLIPS que realiza la suma de dos números se encuentra en el fichero `c:\suma-store-fetch.clp`:

```
(deffacts datos-entrada
   "Los datos de entrada se recuperan con fetch al inicio del programa"
   (dato1 (fetch dato1))
   (dato2 (fetch dato2)))

(defrule suma
  "Esta regla representa el cálculo del programa"
  (dato1 ?n1)
  (dato2 ?n2)
=>
  (assert (resultado (+ ?n1 ?n2))))
```

```
(defrule guarda-resultado
  "Guarda el resultado con store y termina el programa"
  (resultado ?r)
=>
  (store resultado ?r)
  (halt))
```

El siguiente método devuelve el contexto de evaluación global de un objeto `Rete`, que será necesario para interpretar correctamente los objetos `Value` que queramos consultar:

```
Context getGlobalContext()
```

El siguiente programa realiza la comunicación descrita con el programa Jess.

```
//EjemploStoreFetch.java

package jess;

import jess.*;
import java.io.*;

public class EjemploStoreFetch  {

    private static String nomf = "\"c:\\\\suma-store-fetch.clp\"";

    public static void main(String[] argv) throws JessException {
        int  n1, n2;
        Value v;
        Context c;
        Rete r = new Rete();

        //pide los datos y los guarda en la tabla
        n1 = pideEntero("Escribe el primer valor entero : ");
        n2 = pideEntero("Escribe el segundo valor entero : ");
        r.store("dato1", new Value(n1, RU.INTEGER));
        r.store("dato2", new Value(n2, RU.INTEGER));

        //carga y ejecuta el programa Jess
```

```
        r.executeCommand("(batch " + nomf + ")");
        r.reset();
        r.run();

        //recupera y muestra el resultado
        v = r.fetch("resultado");
        c = r.getGlobalContext();
        System.out.println("El resultado es : " + v.intValue(c));
    }

    //-------------------------------------------------------

    private static int pideEntero (String msj) {
    int n;

    System.out.println(msj);
    try {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String entrada = br.readLine();
        n = (Integer.valueOf(entrada)).intValue();
    }
    catch(IOException           ex) {n = pideEntero(msj);}
    catch(NumberFormatException ex) {n = pideEntero(msj);}

    return n;
  }
}
```

La función auxiliar `pideEntero` simplemente escribe un mensaje por pantalla y lee un número entero de teclado. La ejecución del programa provocará la siguiente salida por pantalla:

```
Escribe el primer valor entero :
10
Escribe el segundo valor entero :
20
El resultado es : 30
```

7.3.3. Cómo manipular la lista de hechos

En esta sección estudiaremos un tipo de interacción más sofisticada con el entorno Jess. Concretamente analizaremos cómo se puede controlar la ejecución del programa Jess añadiendo hechos en la lista de hechos. También será frecuente la necesidad de consultar el contenido completo de la lista de hechos una vez completada la ejecución.

Para manipular la lista de hechos será necesario emplear diversos métodos de las clases `Rete` y `Fact`. La consulta de la lista de hechos puede requerir métodos distintos dependiendo de si son ordenados o con plantilla, o si contienen o no multicampos. Trataremos sólo el caso de hechos con plantilla y sin multicampos.

Para asertar nuevos hechos puede emplearse el método `executeCommand(String)` de la clase `Rete` ya comentado en la sección 7.3.1.

Puede obtenerse un iterador de la lista de hechos (es decir, un objeto de la clase `java.util.Iterators`) empleando el siguiente método de la clase `Rete`:

```
Iterator listFacts()
```

Los siguientes métodos de la clase `Fact` serán de utilidad para consultar un hecho concreto:

- `String getName()` devuelve una cadena con el nombre del hecho.
- `Value getSlotValue(String nomslot)` recibe una cadena con el nombre de un slot y devuelve un objeto de la clase `Value` con el valor del slot. Para recuperar dicho valor pueden usarse los métodos ya descritos en la sección 7.2.1 para objetos `Value` .
- `String toString()` devuelve una cadena con una representación del hecho que puede ser impresa posteriormente.

Ejemplo: modificación y consulta de la lista de hechos

Consideremos que el siguiente programa CLIPS se encuentra en el fichero `c:\prueba3.clp` (nótese que el cualificador de slots `type` no está favorecido en Jess y es más limitado que el de CLIPS:

```
(deftemplate numero (slot valor (type NUMBER)))
(deftemplate par    (slot valor (type INTEGER)))
(deftemplate impar  (slot valor (type INTEGER)))
```

```
(defrule numero-par
  ?h <- (numero (valor ?v))
  (test (and (integerp ?v) (= 0 (mod ?v 2))))
=>
  (retract ?h)
  (assert (par (valor ?v))))

(defrule numero-impar
  ?h <- (numero (valor ?v))
  (test (and (integerp ?v) (= 1 (mod ?v 2))))
=>
  (retract ?h)
  (assert (impar (valor ?v))))
```

El siguiente programa carga un programa CLIPS, añade diversos hechos de tipo `numero` a la lista de hechos, ejecuta el programa y muestra el contenido final de la lista de hechos por pantalla.

```
//Ejemplo2.java

package jess;

import jess.*;
import java.util.Iterator;

public class Ejemplo2  {

    private static String nomf = "\"c:\\\\prueba3.clp\"";

    public static void main(String[] argv) throws JessException {

        Rete r = new Rete();

        r.executeCommand("(batch " + nomf + ")");
        r.reset();
        r.executeCommand("(assert (numero (valor " + 2 + ")))");
        r.executeCommand("(assert (numero (valor " + 5 + ")))");
        r.executeCommand("(assert (numero (valor " + 7 + ")))");
        r.executeCommand("(assert (numero (valor " + 2.5 + ")))");
```

```
        r.run();

        procesaListaHechos(r);
    }

    //--------------------------------------------------------

    static void procesaListaHechos(Rete r)  throws JessException {
        Iterator it;
        Fact     hecho;
        Value    v;
        String   s;
        Context  c = r.getGlobalContext();
        int   n;
        double f;

        it = r.listFacts();
        while (it.hasNext()) {
            hecho = (Fact)it.next();
            s = hecho.getName();
            if (s.equals("MAIN::par")) {
               v = hecho.getSlotValue("valor");
               n = v.intValue(c);
               System.out.println("Número par: " + n);
            } else if (s.equals("MAIN::impar")) {
                v = hecho.getSlotValue("valor");
                n = v.intValue(c);
                System.out.println("Número impar: " + n);
            } else if (s.equals("MAIN::numero")) {
                v = hecho.getSlotValue("valor");
                f = v.floatValue(c);
                System.out.println("Número real: " + f);
            } else
                System.out.println("Hecho desconocido");
        }
    }
}
```

La ejecución del programa provocará la siguiente salida por pantalla:

```
Hecho desconocido
Número real: 2.5
Número impar: 7
Número impar: 5
Número par: 2
```

El hecho desconocido corresponde al hecho por defecto `(initial-fact)`, que también se encuentra presente en la lista de hechos.

7.4. Guía rápida

Concluiremos este capítulo con un resumen de las clases, interfaces y métodos que hemos estudiado, todas ellas disponibles en Jess 6.1p5. La documentación de la API de Jess proporciona una relación completa de todas las clases y métodos disponibles para los programadores Java.

class Fact

```
String getName()
Value  getSlotValue(String nomslot)
String toString()
```

class JessException

```
JessException (java.lang.String rutina,
               java.lang.String mensaje,
               java.lang.String datos)
```

class LongValue

```
LongValue(long l) throws JessException
long     longValue(Context c) throws JessException
```

class Rete
`UserFunction addUserFunction(UserFunction uf)`

`void clear()`
`void halt()`
`void reset()`
`int  run()`
`int  run(int max)`

`Value store(String nombre, Value val);`
`Value store(String nombre, Object val);`
`Value fetch(String nombre);`
`void  clearStorage();`

`Value executeCommand(java.lang.String cmd)`
`Iterator listFacts()` `Context  getGlobalContext()`

interface Userfunction
`String getName()` `Value  call (ValueVector vv, Context c) throws JessException`

class Value

```
int   type()
```

```
Value(boolean b)                throws JessException
Value(String s, int type)       throws JessException
Value(double d)                 throws JessException
Value(int value, int type)      throws JessException
```

```
Object  externalAddressValue(Context c) throws JessException
String  stringValue          (Context c) throws JessException
Fact    factValue            (Context c) throws JessException
Funcall funcallValue         (Context c) throws JessException
ValueVector listValue        (Context c) throws JessException
double  floatValue           (Context c) throws JessException
double  numericValue         (Context c) throws JessException
int     intValue             (Context c) throws JessException
```

class ValueVector

```
Value get(int i)
int   size()
```

Capítulo 8

SISTEMAS BASADOS EN EL CONOCIMIENTO.

Un *Sistema Basado en el Conocimiento* o SBC es un sistema informático que exhibe ciertas capacidades de razonamiento y resolución de problemas en un dominio limitado como, por ejemplo, el diagnóstico médico o el diseño de circuitos electrónicos. Un nombre más antiguo para estos sistemas es el de *Sistema Experto*. Las capacidades de estos sistemas se deben en gran medida al "conocimiento" que el sistema tiene acerca del dominio, conocimiento que aparece explícitamente representado. El carácter explícito de la representación diferencia los SBC de los sistemas informáticos tradicionales.

Los primeros esfuerzos de los investigadores en Inteligencia Artificial se encaminaban hacia la búsqueda de un modelo del razonamiento humano sencillo y aplicable al conjunto de la actividad intelectiva. Esta línea de investigación no resultó, en general, excesivamente fecunda. En contraste con ello, cada SBC razona en un dominio muy específico y bien delimitado, y gracias a esta limitación puede resolver bastante bien los problemas que se le proponen.

Los sistemas expertos aparecieron a principios de los años 70. Entre los primeros sistemas que proporcionaron prestaciones satisfactorias destacan:

–DENDRAL, desarrollado para un dominio de conocimiento muy específico: infería la estructura molecular de compuestos desconocidos a partir de resultados de espectrometría de masas [5], [21], [20].

–PROSPECTOR, una herramienta para ayudar a los geólogos a identificar formaciones geológicas que pueden contener depósitos minerales. Se desarro-

lló entre 1974 y 1983, pero finalmente no se llegó a comercializar [11].

–MYCIN, sistema de diagnóstico médico que determina el agente infeccioso presente en la sangre de un paciente y especifica un tratamiento para dicha infección [28], [6].

–R1/XCON, cuyo dominio era la configuración de ordenadores Vax a partir de descripciones incompletas de los pedidos [22], [3].

8.1. Estructura de un SBC.

Un SBC consta de los siguientes módulos:

- La *base de conocimientos*, que contiene el conjunto de conocimientos aplicables al dominio considerado del mundo real. Esta base permanece constante a lo largo del proceso de razonamiento y también –salvo cuando interviene el módulo de adquisición– de una sesión a otra. Los formalismos más extendidos para representar el conocimiento son las reglas, los marcos, las redes semánticas y las redes bayesianas. Modernamente, otros sistemas también representan el conocimiento en forma de restricciones, leyes cualitativas, etc.

- El *motor de inferencias*, que implementa un algoritmo de manipulación de los conocimientos de forma que se alcancen soluciones a los problemas propuestos. Este elemento y el anterior constituyen el núcleo del sistema.

- La *memoria de trabajo*, que contiene los datos proporcionados y el objetivo propuesto por el usuario, y los resultados intermedios –datos deducidos o subobjetivos generados– obtenidos por el sistema en su proceso de razonamiento. Obviamente, el contenido de esta memoria de trabajo va siendo diferente a lo largo de cada sesión del sistema.

Además de estos tres elementos básicos, serán necesarios otros dos para la conexión del sistema con su entorno:

- La *interfaz con el usuario*, que se encarga de presentar de forma comprensible las respuestas del sistema y de aceptar las entradas del usuario. Una parte fundamental es el servicio de explicaciones o de justificación, que debe ser capaz de exponer al usuario el proceso de razonamiento seguido por el sistema, de forma que se explique o justifique la respuesta proporcionada en la consulta.

- El *módulo de adquisición de conocimientos*, que permite la modificación de la base de conocimientos. La adquisición del conocimiento se realiza generalmente a través del llamado "ingeniero del conocimiento". En este caso, existen tres elementos en el proceso de adquisición:

  –El experto humano, que posee los conocimientos del dominio, si bien en forma a veces implícita y vagamente consciente.

  –El ingeniero del conocimiento, que se encarga de extraer y formular explícitamente el conocimiento del experto humano.

  –El sistema experto, cuya base de conocimientos es modificada directamente por el ingeniero del conocimiento, que ha de conocer para ello los formalismos y convenciones empleados en el sistema.

El proceso de adquisición del conocimiento es el auténtico "cuello de botella" del desarrollo del sistema, por lo que modernamente se tiende, siempre que ello sea posible, a emplear técnicas de aprendizaje automático, que permitan sintetizar la base de conocimientos a partir de un conjunto de ejemplos resueltos previamente.

En la figura 8.1 se representan estos elementos así como las interacciones entre ellos.

8.2. Ciclo de vida.

Las fases generalmente aceptadas son, con pequeñas variantes, las siguientes:
–Identificación.
–Conceptualización.
–Prototipado.
–Prueba y redefinición.
–Mantenimiento de la base de conocimientos.

En la fase de *identificación* se determinan los requisitos del sistema. Primeramente, es necesario determinar el experto o equipo de expertos humanos cuyo conocimiento se pretende transferir al sistema, y el ingeniero del conocimiento que va a diseñarlo. Es imprescindible que el experto humano esté muy interesado en el desarrollo del proyecto, y que disponga de tiempo bastante para interactúar con el ingeniero del conocimiento. También es necesario identificar el tipo de problemas concretos que se pretende resolver, señalando el objetivo global y los subproblemas en que se descompone. Todo ello llevará a un listado de los recursos necesarios para la tarea: fuentes del conocimiento, como expertos

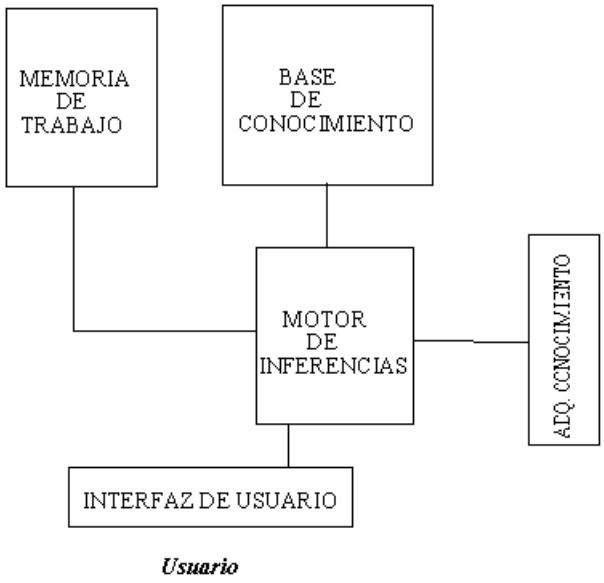

Figura 8.1: Estructura de un SBC

humanos, colecciones de casos resueltos; tiempo necesario; recursos informáticos; y dinero. A partir de este listado será posible determinar si la construcción del sistema es viable o no.

La fase de *conceptualización*, o conceptualización y formalización, va encaminada a encontrar los conceptos necesarios para representar el conocimiento en el dominio considerado, y fijar el formalismo que el sistema empleará para ello. Un procedimiento frecuente para ello es el siguiente: el ingeniero del conocimiento propone al experto casos prácticos sencillos, y elabora los correspondientes *protocolos* descriptivos de su proceso de razonamiento. El ingeniero abstrae entonces lo que parece ser el procedimiento de resolución seguido por el experto, lo formula informalmente y lo aplica, delante del experto, a la resolución de casos sencillos. El experto hará notar los pasos erróneos dados en el razonamiento, los conocimientos pertinentes que se han omitido, y conocimientos que había olvidado hacer explícitos en las anteriores sesiones. En esta fase no se pretende acumular todo el conocimiento necesario para la resolución de casos reales: el objetivo simplemente es hacer posible un primer prototipo del sistema.

La fase de *prototipado* consiste en la representación en un sistema informático del prototipo conceptualizado en la fase previa. A diferencia del software clásico, muchas inconsistencias y errores no se detectan hasta esta fase, ya que no se dispone de especificaciones exactas del sistema. El prototipo tendrá una gran importancia en el diseño global del sistema: de una parte, servirá para probar lo adecuado de la formalización elegida; de otra, será la base sobre la cual se construya el sistema definitivo. También en esta fase hay que implementar las interfaces con el usuario, que se encargarán de preguntar al usuario no informático los datos necesarios para la resolución del problema, y de justificar la decisión tomada.

Cuando se ha conseguido que el prototipo resuelva por completo algunos ejemplos de casos, deberá probarse intensivamente con casos variados, para detectar y corregir sus limitaciones. Esta es la fase de *prueba y redefinición*.

Capítulo 9

CLASIFICACIÓN HEURÍSTICA.

9.1. Las tareas de clasificación.

Todos utilizamos procesos mentales de clasificación de manera mas o menos consciente. De forma natural tendemos a agrupar en *clases* o *categorías* sucesos y objetos con características observables distintas. Clasificar algo (un objeto, patrón, medida, etc.) es identificarlo como miembro de una clase conocida.

Las clases expresan regularidades, de modo que todos los miembros de una clase comparten ciertas propiedades. Consideraremos que en un sistema de clasificación existe siempre un conjunto finito y predefinido de soluciones o clases. La determinación cuáles son estas clases y su definición es un problema distinto y, en general, más complicado que la tarea de clasificación.

Las tareas de clasificación, tal como se han definido aquí, comprenden muchos procesos que llevamos a cabo cotidianamente:

- la *selección* de entre un conjunto prefijado de opciones, a partir de una descripción de las preferencias del agente.
- el *diagnóstico* de una enfermedad o avería a partir de un conjunto de síntomas (en realidad el problema del diagnóstico es más amplio, vd. la discusión más abajo).
- la *clasificación* propiamente dicha, es decir, la determinación del valor

de un atributo distinguido llamado *clase* a partir de los valores de otros atributos.

Existen diversas formas de realizar las tareas de clasificación. Algunas de ellas son apropiadas cuando los datos son fundamentalmente numéricos, de forma que el problema se puede reducir a calcular distancias en ciertos espacios vectoriales.

Nosotros abordamos aquí, siguiendo a Clancey [7], el proceso de clasificación típico de los sistemas basados en el conocimiento, proceso que suele denominarse *clasificación heurística*. En este caso, el proceso de clasificación puede caracterizarse mediante las siguientes fases:

(1) abstracción de datos,
(2) correspondencia con una taxonomía de soluciones preenumeradas, y
(3) refinado dentro de la taxonomía de soluciones,

donde cada fase puede descomponerse a su vez en varias etapas de abstracción, correspondencia y refinado.

Por ejemplo, consideremos el problema de diagnostico planteado por el sistema pionero MYCIN [28], [6]. Se trata de identificar una enfermedad infecciosa (si la hay), a partir de los datos médicos de un paciente. Esta tarea puede modelarse de forma natural como una tarea de clasificación, donde cada clase o solución corresponde a una determinada enfermedad. Su implementación se realizó mendiante reglas con encadenamiento hacia atrás. Un ejemplo de estas reglas lo tenemos en la figura 9.1, donde se establece una relación directa entre el número de glóbulos blancos de la muestra de sangre de un paciente y una posible causa de infección.

```
SI existe un análisis de sangre Y
        el número de glóbulos blancos es inferior a 2500
ENTONCES las siguientes bacterias pueden estar causando infección:
        E. coli (0.75),
        Pseudomonas-aeruginosa (0.5),
        Klebsiella-pneumoniae (0.5).
```

Figura 9.1: El conocimiento en MYCIN

Pero, ¿qué significa en realidad esta regla? ¿Cuál es su justificación? La asociación heurística subyacente puede resumirse de la siguiente forma:

1. Abstracción de datos. Un tipo de *condición de riesgo* de infección (aunque no la única) es la *inmunosupresión*, en la que el grado de respuesta del

sistema inmunológico se encuentra inhibido por efecto de medicamentos, radiaciones o algún trastorno del sistema inmunológico. La *leucopenia* es una clase particular de inmunosupresión en la que hay una deficiencia de glóbulos blancos. Suele considerarse que un umbral inferior en el número de glóbulos blancos es 2500. Dicho en orden inverso, si el número de glóbulos blancos es inferior a 2500, entoces el paciente tiene leucopenia, que es una situación de inmunosupresión. Los pacientes con condición inmunosupresora estan en condición de *riesgo de infección*.

2. Correspondencia entre datos y soluciones. Cuando un paciente se encuentra en una condición de riesgo de infección, las bacterias que habitualmente se encuentran en las zonas no estériles del cuerpo pueden provocar una infección.

3. Refinado dentro de la taxonomía de soluciones. Algunas de estas zonas no estériles son la piel, las vias altas respiratorias y el tracto digestivo. La zona más propensa a una infección es el tracto gastrointestinal donde se encuentran habitualmente *enterobacterias* como *E. coli*, *Pseudomonas* o *Klebsiella*.

De esta forma, el conocimiento que MYCIN expresaba de forma "compilada" en la regla de la figura 9.1, en realidad es el expresado de forma explícita en la figura 9.2 (adaptada de [7]).

Aclaraciones terminológicas.

Diagnóstico y clasificación. Hemos dicho antes que diagnosticar es clasificar. Ello es cierto en el caso de MYCIN y en muchos otros; pero no siempre. En primer lugar, la tarea de diagnosticar es más amplia, pues parte de ella es *reconocer anomalías y conflictos*; pues, en general, no tendremos una lista de síntomas, sino un conjunto de datos que habrán de ponerse en relación con los esperados para detectar el posible mal funcionamiento. En segundo lugar, el conocimiento de diagnóstico puede que no se presente en la forma aquí supuesta, sino en otra más *profunda*, es decir, en forma de *modelos de funcionamiento*, *modelos causales*, ... Las técnicas empleadas serán entonces diferentes: simulación, razonamiento causal, mantenimiento de la verdad, ... Por último, como mencionaremos más adelante, la salida de la tarea de diagnóstico puede consistir no en una sola enfermedad, sino en un conjunto de varias enfermedades, todas ellas presentes en el caso diagnosticado.

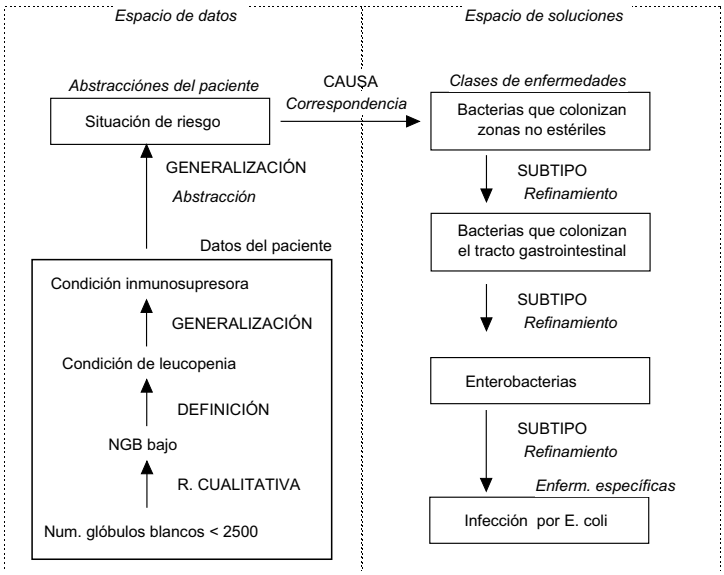

Figura 9.2: El conocimiento subyacente a MYCIN

Valoración y clasificación. En algunas versiones de la metodología KADS [4] no se habla de clasificación, sino solamente de *valoración* (*assessment*). La valoración es la asignación del caso considerado a una clase de decisión; por ejemplo, ante una solicitud S de un préstamo hipotecario, el proceso de valoración asignará S a una de las categorías "concedido/no concedido". Aquí preferimos emplear la palabra "clasificación" para englobar todas estas tareas.

9.2. Árboles de decisión.

En el caso de MYCIN los procesos de abstracción de datos, correspondencia datos/soluciones y refinamiento de soluciones aparecen en toda su complejidad. Vamos a considerar ahora un caso mucho más sencillo, correspondiente a la clasificación mediante un *árbol de decisión*.

El árbol está constituido por nodos; cada nodo n está etiquetado por un atributo $a(n)$. Cada nodo n tiene tantos hijos n_i como valores posibles tiene el atributo $a(n)$. Es decir, cada posible valor de $a(n)$ etiqueta un arco que sale de n

y llega a uno de sus hijos. De esta manera, partiendo de la raíz, se va transitando en cada paso al hijo que corresponda al valor hallado para el atributo del último nodo considerado. Finalmente, se llegará a un nodo hoja que estará etiquetado con el valor de la clase correspondiente a la combinación de valores hallados para los atributos considerados. Un ejemplo se muestra en la figura 9.3.

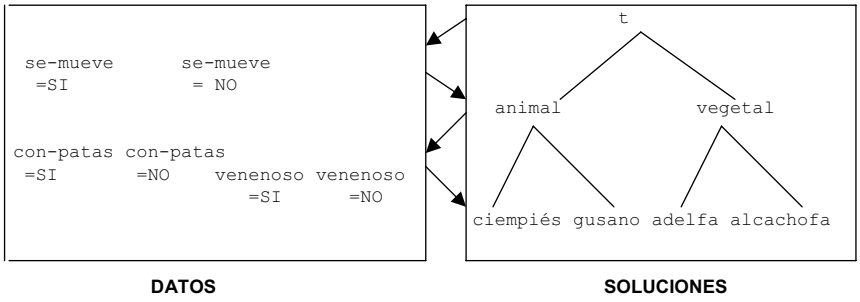

Figura 9.3: Un árbol de decisión

En este caso, la relación entre datos y soluciones es muy directa: en cada nivel de refinamiento de la solución existe un atributo discriminante, es decir, un atributo cuyo valor determinará cuál es la solución inmediatamente siguiente en la jerarquía correpondiente al caso considerado. Los nodos hoja dan ya una solución con el máximo nivel posible de refinamiento.

Diversas variantes de los árboles de decisión –más o menos sofisticadas– aparecen ya compiladas en numerosos casos reales. Piénsese en las "claves taxonómicas", en las guías de campo (del tipo *¿Cómo identificar los pájaros de España?*), en las claves de diagnóstico de los libros de medicina o psicología, en los manuales de detección de averías, etc.

Sin embargo, como cualquiera que haya usado este material sabe, consultar un árbol de decisión no suele ser suficiente para resolver un problema de clasificación; pues puede ocurrir: a) que el usuario sea incapaz de discernir el valor del atributo, pese a que éste sea evidente para el experto (ejemplo: *¿aparecen pequeñas pintas parduzcas en el lomo?* El usuario inexperto no sabe exactamente qué se quiere decir con los difusos identificadores "pequeño" y "parduzco"); b) que el valor del atributo no sea de ninguna manera determinable en la situación considerada, en cuyo caso será necesario buscar una vía alternativa de razonamiento (por ejemplo: *¿Cuántos pétalos tiene la flor?* Si no estamos en la época de floración, difícilmente se podrá averiguar); c) que el auténtico experto

detecte que la casificación dada por el árbol no es la correcta por tratarse de un "caso excepcional". Por todo ello, es un claro abuso de lenguaje decir que un árbol de decisión simple es un procedimiento "inteligente"de clasificación.

Ejemplo 9.1 En el programa siguiente ARBOL-DEC1.CLP se muestra una implementación directa en CLIPS del árbol de decisión de la figura 9.3.

```
;;;ARBOL-DEC1.CLP
;;;Ejemplo de arbol de decision binario
;;;en el cual cada arco esta compilado en una regla
;;;no hay interaccion con el usuario
;;;;;;;;;;;;;;;;;;;datos del caso;;;;;;;;;;;;;;;;;;;;;;
(deffacts datos
  (clase t)
  (oav se-mueve si)
  (oav patas no)
  (oav veneno no))
;;;;;;;;;;;;;;conocimiento del dominio;;;;;;;;;;;;;;;;;
(defrule animal
  (clase t)
  (oav se-mueve si)
=>
  (assert (clase animal)))

(defrule vegetal
  (clase t)
  (oav se-mueve no)
=>
  (assert (clase vegetal)))

(defrule ciempies
 (clase animal)
 (oav patas si)
=>
 (assert (solucion ciempies)))

(defrule gusano
 (clase animal)
 (oav patas no)
=>
 (assert (solucion gusano)))

(defrule adelfa
 (clase vegetal)
 (oav veneno si)
=>
 (assert (solucion adelfa)))

(defrule alcachofa
 (clase vegetal)
 (oav veneno no)
=>
 (assert (solucion alcachofa)))
```

◁

Ejercicio 9.1 Los habitantes del reino de Kakastán se dividen en nobles y plebeyos. A su vez, los nobles pueden ser príncipes, caballeros o hidalgos; y los plebeyos, propietarios o proletarios. Cuentan los viajeros que los nobles llevan siempre un hermoso gorro colorado, prenda que les está prohibida a los plebeyos. Además, los príncipes calzan sandalias doradas, los caballeros botas de cuero y los hidalgos chanclas; en cuanto a los plebeyos, llevan el calzado que buenamente pueden. Otra curiosa norma obliga a los plebeyos propietarios a dejarse la barba, mientras que los proletarios han de ir completamente afeitados. Se pide:

1. Formalizar este conocimiento en forma de árbol de decisión y escribir un conjunto de reglas CLIPS que lo implementen (una regla por cada arco del árbol).

2. Trazar el funcionamiento del programa al clasificar a un sujeto de gorro rojo, calzado con chanclas y que lleva barba.

◁

Ejercicio 9.2 Considérese de nuevo el problema del control de la STT, tal como se planteó en el ejercicio 2.9. Formalizarlo como un árbol de decisión e implementarlo en CLIPS. ◁

El enfoque anterior para implementar árboles de decisión no es demasiado elegante; además, se suponen todos los datos dados desde un principio, lo cual no es siempre razonable: parece preferible que el programa los vaya preguntando conforme los necesite. Por ello proporcionamos una implementación más abstracta, en la que los arcos serán **hechos** CLIPS, y el algoritmo de razonamiento en el árbol de decisión se expresará mediante **reglas** CLIPS.

Ejemplo 9.2 En el programa siguiente ARBOL-DEC.CLP se muestra una implementación genérica del algoritmo del árbol de decisión. El conocimiento del dominio de la figura 9.3 se almacena en el fichero COSAS1.CLP. Ejecutando el fichero COSAS1.BAT, se llevaría a cabo el razonamiento en este dominio.

```
;;;ARBOL-DEC.CLP
;;;Arbol de decision generico.
;;La raiz se representa por un hecho (clase t).
;;Cada arco,por un hecho ordenado
;;    (arco (clase-padre c1) (oav a1 v1) (subclase c2))
;;con el siguiente significado:
;;  si estamos en la clase c1 y el atributo a1 tiene el valor v1,
;;entonces estamos en la subclase c2 de c1.
(deftemplate atributo
```

```
  (slot nombre (type SYMBOL) (default ?NONE))
  (slot pregunta (type STRING))
  (multislot valores))

(deftemplate arco
  (slot clase-padre (type SYMBOL))
  (multislot oav (cardinality 2 2))
  (slot subclase (type SYMBOL)))

(deffunction preguntar (?mensaje $?opciones)
  "muestra ?mensaje y lee la respuesta hasta que este en ?opciones"
  (printout t crlf ?mensaje " " $?opciones  "? ")
  (bind ?result (read))
  (while (not (member$ ?result ?opciones))
    do (printout t crlf ?mensaje " " $?opciones  "? ")
       (bind ?result (read)))
  ?result)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defrule preguntar (clase ?c)
 (arco (clase-padre ?c) (oav ?a ?))
 ?h<-(atributo (nombre ?a) (pregunta ?p) (valores $?vs))
 (not (preguntadop ?a))
 =>
 (bind ?r (preguntar ?p ?vs))
 (assert (preguntadop ?a))
 (assert (oav ?a ?r)))

(defrule refinar
 (clase ?c1)
 (oav ?a ?v)
 (arco (clase-padre ?c1) (oav ?a ?v) (subclase ?c2))
=>
 (assert (clase ?c2))
 (printout t "Deducida la solucion: " ?c2 crlf)
 (printout t "Desea refinarla (si/no)?" crlf " ")
 (bind ?r (read))
 (if (eq ?r no) then (halt)))

(defrule acabar
 (declare (salience -100))
 (initial-fact)
=>
 (printout t "No es posible refinar mas la solucion" crlf))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;fichero COSAS1.CLP
;;;arbol de la figura
;;representado en forma procesable por ARB-DEC.CLP
;;La raiz se representa por un hecho (clase t).

(deffacts atributos
  (atributo (nombre se-mueve)
    (pregunta "Se mueve")
    (valores si no))
  (atributo (nombre patas)
    (pregunta "Tiene patas")
    (valores si no))
  (atributo (nombre veneno)
```

```
    (pregunta "Es venenosa")
    (valores si no)))
(deffacts raiz
  (clase t))
(deffacts arcos
  (arco (clase-padre t) (oav se-mueve si) (subclase animal))
  (arco (clase-padre t) (oav se-mueve no) (subclase vegetal))
  (arco (clase-padre animal) (oav patas si) (subclase ciempies))
  (arco (clase-padre animal) (oav patas no) (subclase gusano))
  (arco (clase-padre vegetal) (oav veneno si) (subclase adelfa))
  (arco (clase-padre vegetal) (oav veneno no) (subclase alcachofa)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;fichero COSAS1.BAT
;;llamada a los ficheros anteriores
(load "arb-dec.clp")
(load "cosas1.clp")
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

◁

Ejercicio 9.3 Repetir los ejercicios 9.1 y 9.2, haciendo uso ahora del programa ARBOL-DEC.CLP del ejemplo 9.2. ◁

9.3. Abstracción de datos.

Las tareas de elicitar, representar y verificar el conocimiento relativo a la abstracción de datos no suelen resultar demasiado difíciles. Además, el nivel de certeza de estas relaciones suele ser muy elevado.

Podemos distinguir varios tipos de relaciones de abstracción:

(a) Abstracción por definición. Frecuentemente se emplea un vocabulario u ontología que es definible en términos más básicos. Por ejemplo, "Se define la presión efectiva como 1,5 veces la presión medida"; o, siguiendo con el dominio de MYCIN: "Se entiende por leucopenia un nivel de glóbulos blancos bajo".

(b) Abstracción por generalización. Otras veces los datos forman de manera natural una jerarquía de clases, por la que se debe ascender hasta llegar al nivel en que de realizará la correspondencia con las soluciones. Un claro ejemplo aparecía en el análisis del conocimiento de diagnóstico de MYCIN efectuado en la sección 9.1: "La leucopenia es una situación de inmunosupresión".

(c) Abstracción cualitativa. Los datos cuantitativos, expresión de magnitudes continuas, no suelen intervenir en bruto en el proceso de razonamiento; generalmente se consideran ciertos intervalos significativos y a cada uno de ellos se le da una etiqueta lingüística. Por ejemplo: "Si el número de glóbulos blancos es menor de 2500, entonces el nivel de glóbulos blancos es bajo".

Ejemplo 9.3 Nuestra amiga Cuqui tiene que averiguar cúando hay que apagar la máquina VIP. Para ello interroga hábilmente a Tomás Elu.
C.– ¿Cuándo apaga usted la máquina?
E.– Cuando la presión compensada es muy grande.
C.– ¿Podría explicarme qué es la presión compensada?
E.– Es fácil, llamamos presión compensada a la presión media multiplicada por el factor de corrección.
C.– ¿Factor de corrección?
E.– Sí, un valor que da el fabricante. Para esta máquina es 0,95
C.– Ya veo. ¿Y la presión media cuál es?
E.– Está claro; la media aritmética de las tres presiones que leemos en cada una de estas pantallitas.
C.– Vale, entonces apagaremos la máquina cuando el número que calculamos según estas reglas es "muy grande". Pero, ¿qué quiere decir exactamente eso de "muy grande"?
E.– Las instrucciones de uso de la máquina dicen que la presión compensada es muy grande si supera los 10000 KPa, grande si está entre 5000 y 10000 Kpa, normal si está entre 1000 y 5000, y baja en otro caso.
C.– Por curiosidad, ¿cuál es el rango total de variación de la presión?
E.– Hombre, como mínimo puede valer 0, y el valor máximo es digamos 20000.
C.– Bueno, creo que ya lo tengo todo. Vamos a tomar un café.

Representemos mediante un conjunto de reglas CLIPS el conocimiento extraído por Cuqui en la anterior entrevista. Parece claro que se han identificado los objetos *sensor 1*, *sensor 2* y *sensor 3*, con el atributo *presión*, y el objeto *máquina VIP* con los atributos *presión media* y *presión compensada*. Los valores de estos atributos son números reales, y además para la presión compensada se definen etiquetas cualitativas. Representaremos todos los datos mediante ternas *oav*.

Primeramente tendremos las reglas referentes a la abstracción por definición:

```
;;;ABSTRAC1.CLP
;;;ejemplos de calculos de magnitudes
;;
(defglobal ?*alfa* = 0.95)
(defrule presion-compensada
  (oav vip presion-media ?p)
=>
  (assert (oav vip presion-compensada (* ?*alfa* ?p))))
(defrule presion-media
  (oav s1 presion ?p1)
  (oav s2 presion ?p2)
  (oav s3 presion ?p3)
```

```
=>
  (assert (oav vip presion-media (/ (+ ?p1 ?p2 ?p3) 3))))
```

Ahora representaremos el conocimiento de abstracción cualitativa. Un primer enfoque puede ser escribir una regla por intervalo:

```
;;;ABSTRAC2.CLP
;;;abstraccion cualitativa intervalar, implementacion especifica
;;que pasaria si asertaramos  (oav ?o presion-compensada ?e)?
(defrule abs-pre-1
  (oav ?o presion-compensada ?p&:(<= 0 ?p)&:(< ?p 1000))
=>
  (assert (oav ?o presion-compensada-abs baja)))
(defrule abs-pre-2
  (oav ?o presion-compensada ?p&:(<= 1000 ?p)&:(< ?p 5000))
=>
  (assert (oav ?o presion-compensada-abs normal)))
(defrule abs-pre-3
  (oav ?o presion-compensada ?p&:(<= 5000 ?p)&:(< ?p 10000))
=>
  (assert (oav ?o presion-compensada-abs grande)))
(defrule abs-pre-4
  (oav ?o presion-compensada ?p&:(<= 10000 ?p)&:(< ?p 20000))
=>
  (assert (oav ?o presion-compensada-abs muy-grande)))
```

Ningún programador decente se quedará muy conforme con este enfoque. ¿No sería más elegante definir como hechos los intervalos de una magnitud y tener una sola regla que abstraiga? La respuesta está en este programa:

```
;;;ABSTRAC3.CLP
;;;abstraccion cualitativa intervalar, implementacion generica
;;
(deftemplate abstractor
  (slot magnitud)
  (multislot etiquetas))
(defrule abstraer
  (oav ?o ?a ?v&:(numberp ?v))
  (abstractor (magnitud ?a)
      (etiquetas $?e1 ?x1&:(numberp ?x1) ?e ?x2 $?e2))
      (test (<= ?x1 ?v))
      (test (<  ?v ?x2))
=>
  (assert (oav ?o ?a ?e)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;ABSTRAC-P.CLP
;;;hechos relativos a la abstraccion de la presion
;;
(deffacts abs1
  (abstractor (magnitud presion-compensada)
      (etiquetas 0 baja 1000 normal 5000 grande 10000 muy-grande 20000)))
```

◁

Ejercicio 9.4 Plasmar en reglas CLIPS el conocimiento de abstracción del dominio MYCIN expresado en la sección 9.1 ◁

9.4. Correspondencia y recubrimiento.

Quizás la tarea más compleja dentro de la clasificación heurística sea la de establecer la correspondencia entre datos y soluciones (clases). Pues, en primer lugar, las relaciones entre datos observables y clases no tienen siempre un carácter determinístico, sino que van moduladas por una "fuerza" o "peso" lingüístico o numérico. Por ejemplo, "la gripe suele cursar con fiebre" o "el 76 % de los habitantes de Kakastán son rubios". Ello obliga a i) elucidar qué significan estas "fuerzas" o "pesos"; ii) definir las leyes de composición adecuadas para manejarlos e integrarlos.

En los primeros tiempos de los SBC se desarrollaron varios enfoques más o menos "ad hoc" para resolver las cuestiones i) y ii). El más famoso es el modelo MYCIN (en realidad, EMYCIN; los conceptos y reglas que utilizaba MYCIN eran levemente diferentes). Pero en los últimos años se prefiere abordar estas cuestiones con las herramientas formales proporcionadas por la teoría de la probabilidad, como se hace en las *redes bayesianas*.

Sin embargo, en este capítulo consideraremos únicamente relaciones determinísticas entre datos y soluciones, tales como "La gripe siempre cursa con fiebre" o "Todos los habitantes de Kakastán son rubios". Estas relaciones podrían representarse en lógica clásica con las fórmulas $\mathsf{gripe} \rightarrow \mathsf{fiebre}$ y $\mathsf{kakastan} \rightarrow \mathsf{rubio}$, respectivamente.

Consideremos por ejemplo el conocimiento expresado en $\mathsf{kakastan} \rightarrow \mathsf{rubio}$. Si sabemos que $\neg\mathsf{rubio}$, la lógica clásica nos premite deducir válidamente $\neg\mathsf{kakastan}$; es la llamada argumentación por *modus tollens*:

$\mathsf{A} \rightarrow \mathsf{B}$, $\neg\mathsf{B}$, luego $\neg\mathsf{A}$.

Pero si sabemos que rubio, la lógica clásica **no** nos permite deducir nada acerca de $\mathsf{kakastan}$; la siguiente argumentación, llamada *argumentación por abducción* **no** es lógicamente válida:

$\mathsf{A} \rightarrow \mathsf{B}$, B, luego A.

El sentido común está de acuerdo con la lógica; en este ejemplo, sabiendo únicamente que el sujeto es rubio, no podemos deducir que sea de Kakastán, pues es claro que puede haber además otras personas rubias. Sin embargo, el sentido común tiende a decir que los hechos "apoyan" o "justifican" la hipótesis

kakastan, sobre todo si el porcentaje de rubios es pequeño fuera de este país. El razonamiento estadístico bayesiano permite expresar de forma rigurosa esta intuición.

En el caso límite, si suponemos que todas las demás clases tales que algunos de sus miembros son rubios han sido descartadas por unas u otras razones, el hecho de ser rubio implicará el origen kakastanés.

Los algoritmos de recubrimiento intentan sistematizar e implementar las reflexiones anteriores. Sea S_k un posible valor de *Clase* (por ejemplo, kakastan). Sea un atributo D_i (por ejemplo, colorpelo), y sea V_{ij} uno de sus posibles valores (por ejemplo, rubio). Supongamos que la solución S_k implica uno de los valores $V_{ij_1}, \ldots, V_{ij_n}$, es decir, $Clase = S_k \rightarrow (D_i = V_{ij_1} \vee \ldots \vee D_i = V_{ij_n})$.

Supongamos que en el caso que estamos clasificando está presente el atributo D_i con el valor V_k. Si V_k es uno de los $V_{ij_1}, \ldots, V_{ij_n}$, se dice que la solución S_k *explica* el dato. Si hacemos uso del razonamiento por abducción, podemos decir que el dato $D_i = V_{ij}$ *aporta evidencia a favor* de la solución S_k.

Por el contrario, supongamos que V_k no es ninguno de los $V_{ij_1}, \ldots, V_{ij_n}$. Entonces se dice que la solución *es incompatible* con el dato. Si hacemos uso del razonamiento deductivo por *modus tollens*, podemos decir que el dato *excluye* la solución.

En cualquier otro caso (es decir, si no existe ninguna relación conocida entre la solución S_k y el atributo D_i), el valor de D_i es irrelevante para la solución. Si para todas las posibles soluciones el atributo es irrelevante, entonces no vale la pena investigar el valor concreto que ha tomado en el caso analizado.

El razonamiento por recubrimiento consistirá básicamente en aplicar un test `ajustadop(S, D)` a cada posible solución `S` a la vista del conjunto presente `D` de datos (valores de atributos) y descartar aquellas soluciones que no lo superen. Se dice que el razonamiento es *conservador* cuando se considera ajustada toda solución `S` que no es incompatible con los datos. Se dice que se sigue el criterio del *recubrimiento positivo* cuando además se exige que `S` explique algún dato. Se dice que se sigue el criterio de la *explicación completa* cuando se exige además que la solución explique todos los datos.

Este último criterio suele ser demasiado exigente. Para satisfacerlo habrá que considerar, no soluciones individuales, sino *conjuntos* de soluciones compatibles entre sí (por ejemplo, si consideramos averías de máquinas, es posible que para explicar todas las anomalías sea necesario suponer que han fallado varios elementos). El problema de hallar diagnósticos compuestos presenta una complejidad mayor (en el sentido técnico) y no lo trataremos en estas páginas.

Por concretar, presentemos e implementemos un algoritmo de recubrimiento con las siguientes características:

– Todos los datos posiblemente relevantes están presentes al comienzo del proceso de razonamiento (ello implica que son relativamente pocos y relativamente fáciles de conseguir).

– El usuario elegirá si desea seguir el criterio de inclusión conservadora, recubrimiento positivo o recubrimiento total.

– Hay un procedimiento `ordenar(L)` que ordena las soluciones restantes según algún criterio (mayor certidumbre heurística, mayor simplicidad, etc.)

En estas condiciones, el algoritmo puede ser el que muestra el cuadro 9.1.

```
Algoritmo recubrimiento-1
1.        L <- TODAS-SOLS.
2.        Obtener el conjunto total de datos D.
4.        Para cada posible S∈L
5.                Si no ajustadop(S, D) entonces L <- L - {S}.
6.        L <- ordenar(L).
```

Cuadro 9.1: Algoritmo simple de recubrimiento.

```
;;;REC.CLP
;;;Algoritmo de clasificacion por recubrimiento
;;;Los datos se representan por (oav o a v)
;;;Las relaciones solucion-> datos se expresan por hechos desordenados
;;;El razonamiento se estructura en fases:
;;;1) eliminar soluciones incompatibles
;;;2) eliminar soluciones que no explican nada
;;;3) eliminar soluciones que no explican todo
;;las reglas del dominio son de la forma
;;"Si solucion es s, entonces el dato d vale v
(deftemplate solucion-imp-dato
 (slot si)
 (multislot entonces (cardinality 2 2)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deffunction preguntar (?mensaje $?opciones)
  "muestra ?mensaje y lee la respuesta hasta que este en ?opciones"
  (printout t crlf ?mensaje " " $?opciones  "? ")
  (bind ?result (read))
  (while (not (member$ ?result ?opciones))
    do (printout t crlf ?mensaje " " $?opciones  "? ")
       (bind ?result (read)))
  ?result)
(deffacts inicializacion
 (fase eliminar-1))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;FASE ELIMINAR SOLS. INCOMPATIBLES;;;;
```

```
;si el atributo a vale v1 y
;la clase c implica que a toma valores ninguno de los cuales es v1
;entonces c no es solucion (deduccion por modus tollens).
(defrule eliminar-sol-incompatible
 (fase eliminar-1)
 ?h<-(clase ?c)
 (oav ?a ?v1)
 (solucion-imp-dato (si ?c) (entonces ?a ?))
 (not (solucion-imp-dato (si ?c) (entonces ?a ?v1)))
=>
 (retract ?h))

(defrule escribir-soluciones-11
  (declare (salience -100))
  (fase eliminar-1)
  (clase ?c)
=>
  (printout t "Una posible solucion compatible con los datos es:" crlf)
  (printout t ?c crlf crlf))

(defrule escribir-soluciones-12
  (declare (salience -200))
  (fase eliminar-1)
=>
  (printout t "No hay mas soluciones compatibles." crlf)
  (bind ?r (preguntar "Desea continuar?" si no))
  (assert (respuesta ?r)))

(defrule cambiar-de-fase-11
  (fase eliminar-1)
  (respuesta no)
=>
  (printout t "Hasta pronto." crlf)
  (halt))

(defrule cambiar-de-fase-12
  ?f<-(fase eliminar-1)
  ?r<-(respuesta si)
=>
  (retract ?f ?r)
  (assert (fase eliminar-2)))

;;;;;;;;;;;;;;;;;;;;;;;;;FASE ELIMINAR SOLS. QUE NO EXPLICAN NADA;;;;;;;;
;si una solucion no explica ningun dato,
;eliminarla
(defrule eliminar-sol-que-no-explica-nada
  (fase eliminar-2)
  ?h<-(clase ?c)
  (forall (oav ?a ?v)
  (not (solucion-imp-dato (si ?c) (entonces ?a ?v))))
=>
  (retract ?h))

(defrule escribir-soluciones-21
  (declare (salience -100))
  (fase eliminar-2)
  (clase ?c)
=>
  (printout t "Una posible solucion que explica algun dato es:" crlf)
  (printout t ?c crlf crlf))
```

```
(defrule escribir-soluciones-22
  (declare (salience -200))
  (fase eliminar-2)
=>
  (printout t "No hay mas soluciones que expliquen algun dato." crlf)
  (bind ?r (preguntar "Desea continuar?" si no))
  (assert (respuesta ?r)))

(defrule cambiar-de-fase-21
  (fase eliminar-2)
  (respuesta no)
=>
  (printout t "Hasta pronto." crlf)
  (halt))

(defrule cambiar-de-fase-22
  ?f<-(fase eliminar-2)
  ?r<-(respuesta si)
=>
  (retract ?f ?r)
  (assert (fase eliminar-3)))

;;;;;;;;;;;;;;;;;;;;;;;FASE ELIMINAR SOLS. QUE NO EXPLICAN TODO;;;;;;;;
;si una solucion compatible deja sin explicar el valor de algun dato,
;eliminarla
(defrule eliminar-sol-que-no-explica-todo
 (fase eliminar-3)
 ?h<-(clase ?c)
 (oav ?a ?v)
 (not (solucion-imp-dato (si ?c) (entonces ?a ?v)))
=>
 (retract ?h))

(defrule escribir-soluciones-31
  (declare (salience -100))
  (fase eliminar-3)
  (clase ?c)
=>
  (printout t "Una posible solucion que explica todos los datos es:" crlf)
  (printout t ?c crlf crlf))

(defrule escribir-soluciones-32
  (declare (salience -200))
  (fase eliminar-3)
=>
  (printout t "No hay mas soluciones que expliquen todos los datos." crlf))
```

Ejemplo 9.4 El algoritmo anterior se puede aplicar a cualquier dominio en el cual el conocimiento se exprese en el formalismo indicado. Por ejemplo, supongamos que quiero saber la raza de un perro, del que me dicen que es peludo y grande. Hay tres posibles razas: chihuahua, que siempre es pequeño; dálmata, que siempre es blanco y negro; y sambernardo, que siempre es peludo y grande.

El código CLIPS correspondiente sería el de los ficheros REC-PRR.CLP (con el conocimiento del dominio) y REC-P1.CLP (con los datos de mi perro):

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;REC-PRR.CLP
;;CONOCIMIENTO DEL DOMINIO
;;hay tres clases posibles: chihuahua, dalmata y sanbernardo
(deffacts soluciones-posibles
 (clase chihuahua)
 (clase dalmata)
 (clase sanbernardo))
;;estas son las reglas del dominio
(deffacts reglas
 (solucion-imp-dato (si sanbernardo) (entonces talla grande))
 (solucion-imp-dato (si sanbernardo) (entonces pelo largo))
 (solucion-imp-dato (si dalmata) (entonces color blanco-negro))
 (solucion-imp-dato (si chihuahua) (entonces talla pequeno)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;REC-P1.CLP
;;DATOS DEL CASO
(deffacts datos
 (oav pelo largo)
 (oav talla grande))
```

◁

Ejercicio 9.5 Nuestra amiga Cuqui debe implementar un sistema para diagnosticar averías de la superturbotostadora iónica o STTI. La máquina se avería con cierta frecuencia, bien por atorarse el ciringulillo 1, bien por quemarse el cable K, bien por tener problemas en el refrigerador, bien por bloqueo del ionizador catastrófico. Tras estudiar concienzudamente varios manuales técnicos, Cuqui ha llegado a las siguientes conclusiones:

– Si se ha atorado el ciringulillo 1, entonces la presión es alta, la temperatura es baja y la luz amarilla se enciende.

– Si se ha quemado el cable K, entonces la presión es alta, la temperatura es media y la luz amarilla se enciende.

– Si hay problemas en el refrigerador, entonces la presión es normal, la temperatura es alta y la luz amarilla está apagada.

– Si el ionizador catastrófico se ha bloqueado, entonces la luz amarilla se enciende.

Se pide implementar en CLIPS este conocimiento y aplicarlo en un caso en el que la presión es alta y la luz amarilla está encendida. ◁

Ejercicio 9.6 Implementar una modificación del algoritmo de recubrimiento que ordene las hipótesis compatibles dando prioridad a las que expliquen más síntomas. ◁

9.5. Recapitulación.

Recordemos que hemos distinguido tres tareas dentro del razonamiento de clasificación: la abstracción de datos, la correspondencia entre datos y soluciones preenumeradas, y el refinado dentro de la taxonomía de soluciones. En las secciones 9.3 y 9.4 hemos estudiado estas dos últimas, y en la sección 9.2 vimos un caso muy sencillo de integración entre refinado y correspondencia. En un caso real, habrán de realizarse las tres tareas, integradas de formás más o menos compleja.

Por ejemplo, en los algoritmos de correspondencia hemos supuesto que los datos ya estaban abstraídos, y que además estaban todos presentes desde un principio. Pero ello puede resultar irreal: ¿qué ocurre si los atributos posibles son varios miles? ¿Los solicitaremos todos antes de empezar a razonar? Por otra parte, ¿qué ocurre si algún atributo es tal que la determinación de su valor resulta costosa? Lo lógico será pues que al comienzo estén presentes tan solo ciertos datos, y que a lo largo del proceso de razonamiento se requieran únicamente los valores que vayan resultando necesarios. Un algoritmo más apropiado para estos casos será entonces el de la tabla 9.2. En este contexto suele hablarse de razonamiento *dirigido por los datos* (*data-driven*) (se parte del conjunto de datos y se calculan las "mejores" soluciones), razonamiento *dirigido por las soluciones* (*solution-driven*) (se parte del conjunto de las soluciones y se van buscando los datos que podrían discriminar entre ellas) y razonamiento *oportunista* (se trabaja en las dos direcciones alternativamente).

El proceso de razonamiento puede complicarse aún más; por ejemplo, el algoritmo de la tabla 9.2 realiza una búsqueda irrevocable, pero es concebible que sea necesario efectuar retrocesos o tanteos. Las posibilidades son realmente innumerables.

Ejercicio 9.7 Supongamos que Cuqui ha seguido profundizando en las interioridades de de la STTI (ejercicio 9.5) y, además del conocimiento allí expresado, ha averiguado lo siguiente:

- La presión es alta cuando está entre 550 y 1000, baja en otro caso.
- La temperatura es alta, media o baja, siendo los valores límite 200 y 750.
- El valor numérico de la presión aparece indicado en el sensor s1. Cuando este sensor está roto (marca un valor negativo), hay que desmontar la tapa posterior y leer el sensor s21, multiplicando por 1,35 la lectura.
- La temperatura se lee en los sensores s2 y s3, promediando ambos resultados.
- Los problemas en el refrigerador pueden ser de sobreenfriamiento o de

```
Algoritmo recubrimiento-2
1.    L <- TODAS-SOLS.
2.    Obtener los datos iniciales D_0
3.    D'_0 <- abstraer(D_0)
4.    Para cada posible solución S del nivel inicial J_0
5.        Si no ajustadop(S, D'_0) entonces L <- L - {S}
6.    Para cada solución S de L
7.        Si S es refinable,
8.            L <- L -{S}
9.            Para cada S_j refinamiento inmediato de S
10.                Obtener los datos útiles D_j para el nivel de S_j
11.                D'_j <- abstraer(D_j)
12.                Si no ajustadop(S_j, D'_j)
13.                    L <- L -{S_j}
14.    L <- ordenar(L)
```

Cuadro 9.2: Un algoritmo más realista.

hiperventilación. El sobreenfriamiento produce charcos debajo de la máquina (que es bastante pesada y peligrosa de mover). La hiperventilación origina una elevada concentración de ozono (que se puede medir, aunque hay que traer un aparato especial para ello).

– El bloqueo del ionizador puede ser de tipo 1 o de tipo 2. El bloqueo de tipo 1 produce también una elevada concentración de ozono, mientras que el de tipo 2 se traduce en que la luz amarilla se enciende pero de manera intermitente.

Escribir un (conjunto de) fichero(s) CLIPS para representar y manejar todo este conocimiento. ◁

Capítulo 10

CONFIGURACIÓN

La configuración es, hoy por hoy, el área donde los SBC han conseguido mayores éxitos desde el punto de vista de su implantación industrial. Por ejemplo, podemos citar los sistemas R1/XCON y XSEL, para la configuración de ordenadores de la empresa Digital (hoy HP), o el sistema PROSE, para la configuración de sistemas de telefonía digital.

La configuración es la actividad que a partir de un conjunto predeterminado de *componentes*, cada uno de los cuales vendrá descrito por un conjunto predeterminado de *parámetros* con sus correspondientes valores y de un conjunto predeterminado de posibles *puertos* para relacionar los componentes, produce un *artefacto* que satisface ciertos *requisitos*.

Es decir, el proceso de configurar un artefacto parte de la siguiente información:

- ¿Qué condiciones se le exigen al artefacto? (requisitos)
- ¿De qué elementos se puede componer el artefacto? (componentes)
- ¿Cuáles son los atributos relevantes de cada componente, así como sus posibles valores? (parámetros)
- ¿Cuáles son las relaciones relevantes entre los componentes, así como sus posibles valores? (puertos–conectores)

A partir de esta información, se van seleccionando los elementos componentes y sus parámetros y se van organizando, a fin de satisfacer los requisitos. Algunos ejemplos de problemas de configuración pueden ser los anteriormente citados

relativos a equipos informáticos o telemáticos. Otros más cotidianos podrían ser hallar una dieta para una persona a partir de sus necesidades y preferencias y de un conjunto de platos; colocar un conjunto de muebles en una habitación de forma que se satisfagan todas las necesidades de funcionalidad;... Es obvio que los problemas de este tipo son más complejos que los de clasificación.

10.1. Satisfacción de restricciones.

Como es sabido, un problema de satisfacción de restricciones queda definido por los siguientes elementos:

- Un conjunto finito de variables $\{X_1, \ldots, X_n\}$.
- Para cada variable X_i, un conjunto finito D_i llamado dominio.
- Un conjunto finito de restricciones $\{C_1, \ldots, C_p\}$. Cada restricción C_j viene dada por un subconjunto no vacío de variables $\{X_{j,1}, \ldots, X_{j,a_j}\}$ y un conjunto $V_j \subseteq D_{j,1} \times \ldots \times D_{j,a_j}$.

El valor a_j es la aridad de la restricción y el conjunto V_j es el conjunto de asignaciones parciales que satisfacen la restricción. Habitualmente, una restricción C_j no viene dada explícitamente por el conjunto V_j, sino por una expresión booleana que para cada asignación devolverá "verdadero" o "falso".

Se llama asignación a una función que a cada variable le asigna un valor de su dominio. Resolver un problema es encontrar una asignación total que satisfaga todas las restricciones.

Muchos problemas de configuración se pueden expresar de forma directa como satisfacción de restricciones. Por ejemplo:

Se dispone de 5 dispositivos, A, B, C, D, E, descritos en la tabla adjunta. Queremos configurar un artefacto formado por un subconjunto de estos dispositivos tal que el coste total sea menor de 23 y cumpla las tres funcionalidades f_1, f_2, f_3.

	precio	función	incompatible-con	exige
A	10	f_1	C	B
B	5	f_2	-	A
C	4	f_3	A	E
D	20	f_1, f_2	E	-
E	7	f_2, f_3	D	-

Es fácil identificar en este problema:

–Cinco variables A, B, C, D, E con el dominio común {ausente, presente}.

–Las restricciones binarias dadas por las incompatibilidades y exigencias de las dos últimas columnas.

–Las restricciones de aridad variable que expresan que la suma de precios de los componentes presentes es menor que 23 y que la unión de funciones de los componentes presentes es $\{f_1, f_2, f_3\}$.

Muchas de estas restricciones (por ejemplo, la incompatibilidad de componentes, o el coste total inferior a 23) pueden calcularse en las asignaciones parciales, podando así muchas ramas del árbol de búsqueda. Si las variables se asignan en el orden indicado, el lector puede comprobar que se generan únicamente 8 de las 32 asignaciones totales, y de estas 8 solamente una satisface la restricción de funcionalidad (A, B, E presentes, C, D ausentes).

El problema de este enfoque radica en la *explosión combinatoria*, que surge porque el número de configuraciones posibles crece exponencialmente con el número de componentes y parámetros. Una búsqueda con retroceso puede ser por tanto inabordable, aún cuando se empleen ciertos heurísticos para ordenar las variables y sus valores, y aún cuando se empleen algoritmos más "astutos", como los de consistencia en arcos, para simplificar la búsqueda. Por ello, en los casos complejos es necesario emplear conocimiento heurístico más sofisticado y dependiente del problema para guiar el proceso de generación de las configuraciones.

10.2. Reglas heurísticas de producción.

La primera propuesta para superar la explosión combinatoria fue el uso de reglas de producción (como las conocidas de CLIPS) de forma que cada regla correspondía a un paso o decisión tomada en el proceso de configuración. Esta es la idea fundamental del ya mencionado sistema XCON ([22]), para configuración de ordenadores Vax. Un buen resumen del razonamiento seguido por XCON puede encontrarse en [31], pp. 625-633. Aquí damos un resumen grosero e inexacto, pero que puede servir de guía para el desarrollo de SBC basados en estos principios.

A partir del conocimiento extraído a los humanos que realizaban esta tarea, y empleando algunos "trucos", fue posible implementar un sistema con encadenamiento hacia adelante que a partir de la especificación inicial y el catálogo de componentes va añadiendo unas y otros hasta llegar a una solución completa, sin realizar tanteos ni retrocesos. Para ello:

1. Se organiza el razonamiento en varias fases consecutivas, a saber:

   a) Completar la especificación, sustituyendo o añadiendo componentes requeridos.
   b) Disponer adecuadamente los componentes
   c) Generar el *layout* y el cableado.

2. Cada fase está compuesta de diversas *subtareas*, implementada cada una de ellas mediante una o varias reglas de producción.

3. La selección y ordenación temporal de estas subtareas es diferente en cada problema; por ello, las reglas de producción deben contener en su parte izquierda las condiciones que aseguren su aplicabilidad en un problema dado. Ello se consigue preguntando si ciertos valores han sido ya asignados, y codificando de alguna forma lo que equivale a una "búsqueda con previsión" en términos de algoritmos de satisfacción de restricciones.

Aclaremos esto con un ejemplo de un dominio menos técnico, adaptado de [33]: supongamos que queremos realizar la compra en el supermercado y que recibimos la siguiente orden: "Traer comida para el desayuno y unas cervezas". La primera fase (a) consistirá en generar a partir de esto una lista completa de la compra, a partir de las restricciones del mandante y de la tienda, lo cual puede llevar a reglas como

```
;;refinar el componente ‘‘desayuno’’
SI desayuno es requerido
ENTONCES añadir cartón-leche es requerido Y
        añadir bote-cafe-soluble es requerido Y
        añadir paquete-galletas es requerido Y
        quitar desayuno es requerido.

;;añadir un componente no especificado inicialmente
SI pack-latas-cerveza es requerido Y papas-fritas es en-oferta
ENTONCES añadir bolsa-papas-fritas es requerido.
```

La segunda fase (b) consistirá en disponer en bolsas los objetos requeridos y comprados, lo cual puede expresarse mediante reglas como

```
;;colocar un objeto pesado en una nueva bolsa.
SI X es requerido Y X no está colocado Y X es pesado
ENTONCES añadir B = bolsa nueva Y
        añadir X está colocado en B
```

El problema de este enfoque, el típico de los primeros sistemas expertos, es la *fragilidad* de los sistemas resultantes, consistente en lo siguiente:

1. Una base de reglas, según se suele decir, representa el *conocimiento* del dominio. Pero esto no es enteramente verdad, pues una base de reglas no puede emplearse para realizar dos tareas que empleen el mismo conocimiento pero con finalidad levemente diferente; por ejemplo, si las reglas sirven para generar una solución, no podrán emplearse para comprobar si una propuesta es realmente una solución.

2. Una base de reglas resulta en general muy grande y muy difícil de depurar y mantener, ya que las reglas mezclan y representan de manera engañosamente uniforme el conocimiento cierto acerca de los componentes y sus parámetros, el conocimiento acerca de las especificaciones, y el conocimiento heurístico que simplifica la búsqueda. Como ejemplo, citemos que en 1989 XCON constaba de unas 15.000 reglas ([3]).

Ejercicio 10.1 La empresa FETUSA (Ferretería y Tubería, S.A.) es la principal fabricante mundial de superturbotostadoras. La fabricación es flexible, de forma que cada stt es diferente en función del pedido del cliente. He aquí un fragmento de la entrevista entre nuestra admirada Cuqui y la experta Encarni, encargada de procesar los pedidos y enviar a fábrica la configuración detallada requerida para cada stt:
C.– ¿Qué hace usted cuando recibe un pedido?
E.– Lo primero es comprobar que no se piden cosas absurdas, y que todos los elementos requeridos están presentes.
C.– ¿Cosas absurdas?
E.– Sí; por ejemplo, es frecuente que se pida un ionizador con menos fluctuancia de la que requieren los refrigeradores, o una caldera con menos exuperancia de la que requiere el ciringulillo.
C.– ¿Y qué hace usted entonces?
E.– Pongo el ionizador o la caldera más pequeña compatible con los refrigeradores o el ciringulillo que figuran en el pedido.
C.– Ya veo. ¿Y eso de los elementos que no están presentes?
E.– Sí, muchas veces se olvidan de poner un ionizador, habiendo pedido algún refrigerador, o un ciringulillo adecuado, habiendo pedido un bogómetro.
C.– ¿Y la caldera no es también necesaria?
E.– Sí, siempre, sea como sea el pedido.
C.– Bueno, supongamos que el pedido está ya completo. ¿Qué hace usted a continuación?
E.– Empiezo a asignar los componentes a la caja adecuada.
C.– ¿Y cómo se sabe cuál es esa caja?

E.– Es sencillo, hace falta una caja grande si hay más de tres componentes, o si la caldera es grande; en otro caso se ponen en una caja normal.
C.– Y siempre se disponen en una sola caja...
E.– No, no siempre; cuando hay más de cinco componentes hace falta una caja adicional.
C.– ¿Cómo se reparten entonces?
E.– Se ponen la caldera, el ciringulillo y el bogómetro en una caja y lo demás en la otra.
C.– Y con eso acaba su trabajo...
E.– De ninguna forma; queda todavía lo más lioso, que es determinar cómo se deben conectar los componentes.
C.– ¿Cómo se hace?
E.– Primero hay que considerar que cada dispositivo tiene una entrada y una salida, y que si hay dos cajas hay que poner un cable K para conectarlas.
C.– Supongo que la salida del ionizador es la entrada del refrigerador, ¿no?
E.– Efectivamente, a menos que se pida un ionizador sin refrigerador, en cuyo caso la salida del ionizador va a tierra. Si hay dos refrigeradores, hay que multiplexarlos a la entrada y a la salida. Y la salida del refrigerador es a la caja.
C.– ¿Y los demás componentes?
E.– Bien, la caldera sale al ciringulillo, el ciringulillo al bogómetro, y este a la caja.
C.– ¿Y si no hay ciringulillo ni bogómetro?
E.– Entonces la caldera sale a la caja.
C.– Creo que se me ha olvidado preguntar por las entradas del ionizador y de la caldera...
E.– Sí, son siempre el enchufe que hay en cada caja. Claro que si están en la misma caja hay que multiplexarlo.
C.– Muchas gracias, Encarni. No quiero cansarla más por hoy. Si al escribir el programa me surgen dudas, me temo que tendré que molestarla de nuevo. (*Nota: dada la limitada disponibilidad de la experta Encarni, en caso de dudas cada lector las resolverá como estime más lógico.*)

Se pide:

1. Escribir un programa en el que se exprese el conocimiento del dominio en forma de reglas CLIPS, de forma que se genere la descripción simbólica (componentes presentes, tipo de los mismos y conexiones entre ellos) y el *layout* de la configuración (bastará un dibujo formado por caracteres ASCII.)

2. Usando el programa anterior, generar la configuración correspondiente a

este pedido: una superturbotostadora con un bogómetro grande y un refrigerador normal.

ANEXO 1. CATÁLOGO DE COMPONENTES.

Componente	Versiones	Requiere
Bogómetro	grande, normal	Ciringulillo mayor o igual
Caja	grande, normal	
Caldera	grande, normal, pequeña	
Ciringulillo	grande, normal	Caldera mayor o igual
Ionizador	grande, normal, pequeño	
Refrigerador	grande, normal	Ionizador grande si hay varios

Componentes auxiliares: cable K, cables normales, multiplexores, enchufes de caja, tomas de tierra. ◁

10.3. Modelos avanzados.

Como ya hemos mencionado, al plantear el problema de configuración como una satisfacción de restricciones, se pierde casi por completo la estructuración que un ser humano tendría en cuenta a la hora de resolver el problema; y al representar uniformemente mediante reglas de producción todo el conocimiento de configuración, se pierde de vista en qué consiste exactamente este conocimiento. Por tanto, será más conveniente considerar explícitamente y distinguir en un primer nivel de análisis y diseño varias tareas diferentes de razonamiento, cada una de las cuales se implementará con los algoritmos o procedimientos que en cada caso resulten más oportunos. Desde un punto de vista general, y siguiendo la estrategia llamada "proponer-y-revisar" (*propose-and-revise*), estas tareas son las siguientes:

- "Hacer operativos" los requisitos iniciales, comprobando su consistencia y traduciéndolos a un lenguaje formal de especificación. Para ello será necesario a veces emplear conocimiento declarativo acerca del dominio y llevar a cabo sesiones de extracción de conocimiento. El resultado de ello será un conjunto inicial de especificaciones (restricciones, que habrán de satisfacerse en todo caso, y preferencias, que deberán respetarse en lo posible.)

- "Inicializar" la configuración. A partir de las especificaciones y del conocimiento declarativo, se genera un primer conjunto de componentes con sus parámetros y sus relaciones (estructura.)

- "Proponer" una modificación a la configuración existente. Esta modificación será un refinamiento de componentes o el añadido de nuevas instancias de componentes.

- "Verificar y criticar" la modificación, es decir, comprobar si viola alguna restricción y en caso afirmativo proponer una acción que evite tal violación. Para que esto sea eficaz suele ser necesario recurrir al llamado "conocimiento de arreglos" (fix knowledge) que indica qué acciones son más apropiadas para cada situación de violación.

- "Modificar" la configuración en función de la acción seleccionada, y en su caso también las especificaciones o restricciones. Una causa para esto último es precisamente la aparición de nuevos componentes o el refinamiento de los existentes. Otra, más profunda, es el descubrimiento de que las especificaciones iniciales eran ambiguas, o imposibles o muy difíciles de satisfacer; si es necesario modificar las especificaciones por esta última causa, estaremos en realidad ante un problema de *diseño*, no de configuración.

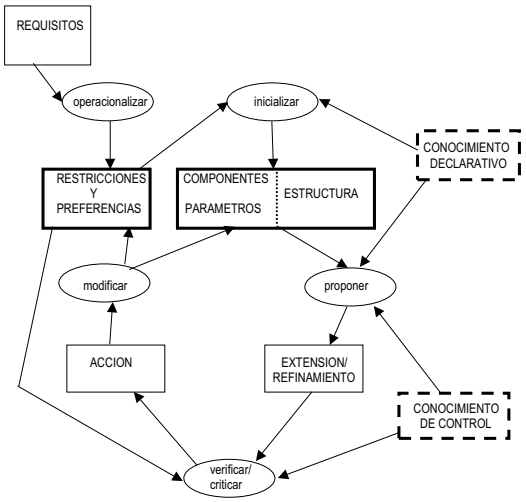

Figura 10.1: Razonamiento de configuración.

Por tanto, en general puede considerarse que la configuración se desarrolla en dos espacios, el de las *especificaciones* y el de la propia configuración, cada uno de ellos con cierta estructura. Todo esto se esquematiza en la figura 10.1, de elaboración propia a partir de varias que aparecen en [31], [27] y [4].

10.4. Razonamiento basado en casos

Este enfoque, radicalmente diferente de los anteriores, supone que se dispone previamente de una biblioteca B de *casos* previamente resueltos, consistente en un conjunto de pares *C(requisitos, solución)*. Dado un nuevo problema, es decir, unos nuevos requisitos R, el procedimiento puede describirse como sigue:

1. Hallar el caso $C_i(r_i, s_i)$ tal que la distancia de r_i a R sea mínima. Nótese que ello presupone que se ha definido una métrica adecuada en el conjunto de los requisitos.

2. Modificar s_i de forma que satisfaga todos los requisitos R. Nótese que ello supone definir los adecuados operadores de modificación de la solución y una estrategia (a ciegas o, mejor, heurística) que los vaya eligiendo y aplicando hasta conseguir el objetivo.

Si la biblioteca de casos resueltos es grande y cubre uniformemente el espacio de posibles problemas, y hay métricas claras y heurísticos eficientes, este método es muy efectivo y puede evitar casi por completo la búsqueda. En otro caso, la fase (2) del procedimiento será, en realidad, una instancia del problema de configuración tan complicada o más que la original.

Bibliografía

[1] John R. Anderson. *The Architecture of Cognition.* Harvard University Press, Cambridge, MA, 1983.

[2] John R. Anderson. *Rules of the Mind.* Lawrence Erlbaum, Hillsdale, NJ, 1993.

[3] V. Barker y D. O'Connor. Expert systems for configuration at Digital: XCON and beyond. *Communications ACM*, 32(3):298–318, 1989.

[4] Joost Breuker y Walter van de Velde, (eds.). *CommonKADS Library for Expertise Modelling.* IOS Press, Amsterdam, 1994.

[5] Bruce G. Buchanan y Edward A. Feigenbaum. DENDRAL and Meta-DENDRAL: Their applications dimension. *Artificial Intelligence*, 11:5–24, 1978.

[6] Bruce G. Buchanan y Edward H. Shortliffe. *Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley, Reading, MA., 1984.

[7] William J. Clancey. Heuristic classification. *Artificial Intelligence*, 27(3):289–350, 1985.

[8] *CLIPS Reference Manual. Volume II. Advanced Programming Guide. Version 6.21.* `http://www.ghg.net /clips/ download/ documentation/ apg.pdf`, June 2003.

[9] *CLIPS Reference Manual. Volume I. Basic Programming Guide. Version 6.21.* `http://www.ghg.net /clips/ download/ documentation/ bpg.pdf`, June 2003.

[10] T. Cooper y N. Wogrin. *Rule-based Programming with OPS5.* Morgan Kaufmann, San Mateo, CA., 1988.

[11] Richard Duda, John Gaschnig, y Peter E. Hart. Model design in the PROSPECTOR consultant system for mineral exploration. En Donald Michie, (ed.), *Expert systems in the micro-electronic age*, pp. 153–167. Edinburgh University Press, Edimburgo, 1979.

[12] Charles L. Forgy. OPS5 user's manual. Informe t'ecnico, Dept. of Computer Science, Carnegie Mellon University, 1981.

[13] Charles L. Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[14] Charles L. Forgy y John P. McDermott 1977: 933-939. Ops, a domain-independent production system language. En *IJCAI-77*, pp. 933–939, 1977.

[15] Ernest J. Friedman-Hill. Jess, the rule engine for the java platform. Informe T'ecnico SAND98-8206 (revised), Distributed Computing Systems. Sandia National Laboratories, Livermore, CA, 2003. http:// herzberg.ca.sandia.gov/ jess.

[16] Joseph Giarratano y Gary Riley. *Expert Systems. Principles and Programming.* PWS Publishing, Boston, Ma., 2nd ed., 1998. Hay traducción española [17].

[17] Joseph Giarratano y Gary Riley. *Sistemas Expertos. Principios y Programación.* Thompson International, México, 2001.

[18] Thaddeus J. Kowalski y Leon S. Levy. *Rule-Based Programming.* Kluwer, Dordrecht, 1996.

[19] John E. Laird, Allen Newell, y Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[20] Joshua Lederberg. How DENDRAL was conceived and born. En *ACM Symposium on the History of Medical Informatics.* National Library of Medicine, 1987.

[21] R. K. Lindsay, Bruce G. Buchanan, Edward A. Feigenbaum, y Joshua Lederberg. *Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project.* McGraw-Hill, New York, 1980.

[22] John P. McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19(1):39–88, 1982.

[23] Alan Newell. *Unified Theories of Cognition.* Harvard University Press, Cambridge, Ma., 1991.

[24] Alan Newell y Herbert A. Simon. *Human problem solving.* Prentice-Hall, Englewood Cliffs, NJ, 1972.

[25] Robert A. Orchard. *FuzzyCLIPS Version 6.04A User's Guide.* Integrated Reasoning, Institute for Information Technology, National Research Council Canada, October 1998. http:// ai.iit.nrc.ca/ IR_public/ fuzzy/ fuzzyClips/ fuzzyCLIPSIndex.html.

[26] Robert A. Orchard. *NRC FuzzyJ Toolkit for the Java(tm) Platform User's Guide Version 1.5a.* Integrated Reasoning, Institute for Information Technology, National Research Council Canada, May 2003. http:// ai.iit.nrc.ca/ IR_public/ fuzzy/ fuzzyJToolkit.html.

[27] Guus Schreiber, Hans Akkermans, Anjo Anjewieden, Robert de Hoog, Nigel Shadbolt, Walter van de Velde, y Bob Wielinga. *Knowledge Engineering and Management. The CommonKADS Methodology.* MIT Press, Cambridge, Ma., 2000.

[28] Edward H. Shortliffe. *Computer-based medical consultations: MYCIN.* Elsevier-North Holland, New York, 1976.

[29] Herbert A. Simon. *The sciences of the artificial.* MIT Press, Cambridge, MA., 1968.

[30] Herbert A. Simon. *Models of my life.* MIT Press, Cambridge, MA., 1996.

[31] Mark Stefik. *Introduction to knowledge systems.* Morgan Kaufmann, Los Altos, Ca., 1995.

[32] Joseph Weizenbaum. *Computer power and human reason.* W.H.Freeman, San Francisco, 1976. (Hay trad. española: La frontera entre el ordenador y la mente, Madrid, Pirámide 1978).

[33] Patrick Henry Winston. *Artificial Intelligence.* Addison-Wesley, Reading, Ma., 3rd ed., 1992.