CLIPS 6.4
User's Guide

by Joseph C. Giarratano, Ph.D.

Spanish edition

Revisión: martes, 21 de mayo de 2024

Tabla de Contenidos

Léame

> *El primer paso en el camino hacia la sabiduría es admitir la ignorancia. El segundo paso es darse cuenta de que no hay que cotorrearlo al mundo.*

Esta sección se llamaba antes *Prefacio*, pero como nadie la leía, le cambié el nombre por otro más convencional que los usuarios de ordenadores están obligados a obedecer. Otra sugerencia fue llamar a esta sección *"No me lea"*, pero como hoy en día la gente se cree todo lo que lee, temí que realmente no la leyeran.

El propósito de un *Prefacio*, perdón, de un *Léame*, es proporcionar **metaconocimiento** sobre el conocimiento contenido en un libro. El término *metaconocimiento* significa conocimiento sobre el conocimiento. Así que esta descripción del *Léame* es en realidad metaconocimiento. Si en este punto estás confundido o intrigado, sigue adelante y lee este libro de todos modos porque necesito todos los lectores que pueda conseguir.

¿Qué es CLIPS?

CLIPS es una herramienta de sistemas expertos desarrollada originalmente por la Software Technology Branch (STB), NASA/Lyndon B. Johnson Space Center. Desde su primera versión en 1986, CLIPS se ha ido perfeccionando y mejorando continuamente. Actualmente lo utilizan miles de personas en todo el mundo.

CLIPS está diseñado para facilitar el desarrollo de software que modele el conocimiento o la experiencia humana.

Existen tres formas de representar el conocimiento en CLIPS:

- *Reglas*, destinadas principalmente al conocimiento heurístico basado en la experiencia.
- *Deffunctions* y *funciones genéricas*, destinadas principalmente al conocimiento procedimental.
- *Programación orientada a objetos*, también pensada principalmente para el conocimiento procedimental. Se admiten las cinco características generalmente aceptadas de la programación orientada a objetos: clases, controladores de mensajes, abstracción, encapsulación, herencia y polimorfismo. Las reglas se pueden hacer corresponder con patrones de objetos y hechos.

Se puede desarrollar software utilizando sólo reglas, sólo objetos o una mezcla de ambos.

CLIPS también se ha diseñado para su integración con otros lenguajes, como C y Java. De hecho, CLIPS es el acrónimo de *C Language Integrated Production System* (sistema de producción integrado en lenguaje C). Las reglas y los objetos también forman un sistema integrado, ya que las reglas se pueden hacer corresponder con patrones de hechos y objetos. Además de utilizarse como herramienta independiente, CLIPS puede invocarse desde un lenguaje procedimental, realizar su función y, a continuación, devolver el control al programa invocador. Del mismo modo, el código procedimental puede definirse como funciones externas y llamarse desde CLIPS. Cuando el código externo completa su ejecución, el control vuelve a CLIPS.

Si ya estás familiarizado con la programación orientada a objetos en otros lenguajes como Smalltalk, C++, Objective C o Java, conoces las ventajas de los objetos en el desarrollo de software. Si no estás familiarizado con la programación orientada a objetos, CLIPS te resultará una herramienta excelente para aprender este nuevo concepto de desarrollo de software.

De qué trata este libro

La *Guía del usuario de CLIPS* es un tutorial introductorio sobre las características básicas de CLIPS. No pretende ser una discusión exhaustiva de toda la herramienta. El volumen que acompaña a este libro es el *Manual de Referencia CLIPS*, que proporciona una discusión completa y detallada de todos los temas tratados aquí y mucho más.

Quién debe leer este libro

El propósito de la *Guía del usuario de CLIPS* es proporcionar una introducción básica y fácil de leer a los sistemas expertos para personas con poca o ninguna experiencia en ellos.

La *Guía del usuario de CLIPS* se puede utilizar en aulas o como autoaprendizaje. El único prerrequisito es tener conocimientos elementales de programación en un lenguaje de alto nivel como Java, Ada, FORTRAN, C (de acuerdo, BASIC si acaso, pero no lo admitiremos en público y desmentiremos esta afirmación si se nos pregunta).

Cómo utilizar este libro

La Guía del usuario de CLIPS está diseñada para personas que quieren una introducción rápida y práctica a la programación de sistemas expertos. Los ejemplos son de carácter muy general. Además, dado que aprender un nuevo lenguaje puede ser una experiencia frustrante, la redacción tiene un estilo ligero y humorístico (espero) en comparación con los

libros de texto universitarios, serios, voluminosos e intimidantes. Esperemos que este humor no ofenda a nadie.

Para sacar el máximo provecho, es recomendable teclear los programas de ejemplo que aparecen en el texto a medida que lees el libro. Al teclear los ejemplos, verás cómo deberían funcionar los programas y qué mensajes de error aparecen si cometes un error. La salida de los ejemplos se muestra o se describe después de cada ejemplo. Finalmente, deberías leer el material correspondiente en el Manual de Referencia CLIPS a medida que cubres cada capítulo de esta Guía del Usuario CLIPS.

Como cualquier otro lenguaje de programación, sólo aprenderás a programar en CLIPS escribiendo programas en él. Para aprender realmente a programar sistemas expertos, deberías elegir un problema de tu interés y escribirlo en CLIPS.

Agradecimientos

Agradezco enormemente los consejos y las reseñas de este libro por parte de muchas personas. Gracias a Gary Riley, Chris Culbert, Brian Dantes, Bryan Dulock, Steven Lewis, Ann Baker, Steve Mueller, Stephen Baudendistel, Yen Huynh, Ted Leibfried, Robert Allen, Jim Wescott, Marsha Renals, Pratibha Boloor, Terry Feagin y Jack Aldridge. Un agradecimiento especial a Bob Savely por apoyar el desarrollo de CLIPS.

Nota del editor

Se han introducido pequeñas modificaciones en el documento original del Dr. Giarratano para reflejar los cambios introducidos en las nuevas versiones de CLIPS. — Gary Riley.

Nota del traductor

Le invito a disfrutar de mi traducción de la Guía del usuario de CLIPS. Soy licenciado en Ciencias Físicas en la especialidad de Electrónica y Computación por la Universidad de Santiago de Compostela (Galiza, España, UE), máster en Sistemas de Información Geográfica por ESRI. Soy funcionario de la Administración Pública y trabajo como técnico superior en tecnologías de la información (TIC) - David Martínez Suárez.

Capítulo 1

Sólo los hechos

Si ignoras los hechos, nunca te preocupará equivocarte

Este capítulo introduce los conceptos básicos de un sistema experto. Verás cómo insertar y eliminar hechos en CLIPS.

Introducción

CLIPS es un tipo de lenguaje informático diseñado para escribir aplicaciones denominadas **sistemas expertos**.

Un sistema experto es un programa destinado específicamente a modelar la experiencia o los conocimientos humanos. En cambio, los programas convencionales, como los de nóminas, los procesadores de texto, las hojas de cálculo, los juegos de ordenador, etc., no pretenden modelar esa experiencia o esos conocimientos. (Una definición de experto es alguien que se encuentra a más de 80 kilómetros de casa y que lleva un portafolios).

CLIPS se denomina **herramienta** de sistemas expertos porque es un entorno completo para desarrollar sistemas expertos que incluye características como un editor integrado y una herramienta de depuración. La palabra **shell** se reserva para la parte de CLIPS que realiza **inferencias** o razonamientos. El shell de CLIPS proporciona los elementos básicos de un sistema experto: 1. **lista de hechos** y **lista de instancias**: Memoria global para datos. 2. **base de conocimiento**: Contiene todas las reglas, la **base de reglas**. 3. **motor de inferencia**: Controla la ejecución global de las reglas. Un programa escrito en CLIPS puede constar de reglas, hechos y objetos. El motor de inferencia decide qué reglas deben ejecutarse y cuándo. Un sistema experto basado en reglas escrito en CLIPS es un programa basado en datos donde los hechos, y los objetos si se quiere, son los datos que estimulan la ejecución a través del motor de inferencia.

Este es un ejemplo de cómo CLIPS difiere de lenguajes procedimentales como Java, Ada, BASIC, FORTRAN y C. En los lenguajes procedimentales, la ejecución puede realizarse sin datos, es decir, las sentencias en esos lenguajes son suficientes para causar la ejecución. Por ejemplo, una sentencia como PRINT 2 + 2 podría ejecutarse inmediatamente en BASIC. Esta es una sentencia completa que no requiere ningún dato adicional para producir su ejecución. Sin embargo, en CLIPS, se requieren datos para iniciar la ejecución de reglas.

Al principio, CLIPS tenía capacidades para representar sólo reglas y hechos. Sin embargo, las mejoras de la versión 6.0 permiten que las reglas hagan correspondencia tanto con objetos como con hechos. Además, los objetos pueden utilizarse sin reglas mediante el envío de mensajes, por lo que el motor de inferencia ya no es necesario si sólo se usan objetos. En los capítulos 1 a 7, discutiremos los hechos y las reglas de CLIPS. Las características de los objetos de CLIPS se tratan en los capítulos 8 a 12.

El principio y el fin

Para iniciar CLIPS, basca con introducir el comando de ejecución apropiado para tu sistema. Deberías ver aparecer un prompt de CLIPS como el siguiente:

```
CLIPS>
```

A partir de este momento, puedes empezar a teclear **comandos** directamente en CLIPS. El modo en el que estás introduciendo comandos directamente se llama **nivel superior**. Si dispones de una versión de CLIPS con interfaz gráfica de usuario (GUI), también puedes **seleccionar** algunos comandos utilizando el ratón o las teclas de flecha en lugar de escribirlos. Por favor, consulta la *Guía de Interfaces CLIPS* para una discusión de los comandos soportados por las diferentes GUIs de CLIPS. Por simplicidad y uniformidad en este libro, asumiremos que los comandos se teclean.

El modo normal de salir de CLIPS es con el comando **exit**. Simplemente escribe

```
(exit)
```

en respuesta al prompt de CLIPS y luego presiona la tecla de retorno de carro.

Elaborar una lista

Al igual que otros lenguajes de programación, CLIPS reconoce ciertas palabras clave. Por ejemplo, si quieres poner datos en la lista de hechos (fact-list), puedes usar el comando assert.

Como ejemplo de *assert*, introduce lo siguiente justo después del prompt CLIPS tal como se muestra a continuación:

```
CLIPS> (assert (duck))
```

Aquí el comando assert recibe (duck) como argumento. Asegúrate de presionar siempre la tecla de retorno de carro para enviar la línea a CLIPS.

Verás la respuesta

<Fact-1>

que indica que CLIPS ha almacenado el hecho duck en la lista de hechos y le ha dado el identificador 1. Los **corchetes angulares** se utilizan como delimitador en CLIPS para rodear el nombre de un elemento. CLIPS nombrará automáticamente los hechos usando un número secuencialmente creciente y listará el índice de hechos más alto cuando uno o más hechos sean afirmados.

Fíjate que el comando (assert) y su argumento (duck) están entre paréntesis.

Como muchos otros lenguajes de sistemas expertos, CLIPS tiene una sintaxis similar a LISP que utiliza paréntesis como delimitadores. Aunque CLIPS no está escrito en LISP, el estilo de este último ha influido en el desarrollo de aquel.

Y comprobarlo dos veces

Supongamos que quieres ver lo que hay en la lista de hechos. Si tu versión de CLIPS soporta una GUI, puedes simplemente seleccionar el comando apropiado del menú. Alternativamente, puedes introducir comandos desde el teclado. A continuación, describiremos los comandos de teclado ya que las selecciones de ventana se explican por sí mismas.

El comando de teclado para ver los hechos es con el **comando facts**. Escribe (facts) en respuesta al prompt de CLIPS y éste responderá con un listado de hechos en la lista de hechos (fact-list). Asegúrate de poner paréntesis alrededor del comando o CLIPS no lo aceptará. El resultado del comando (facts) en este ejemplo debería ser

```
CLIPS> (facts)
f-1 (duck)
For a total of 1 fact.
CLIPS>
```

El término f-1 es el **identificador de hecho** asignado por CLIPS al hecho (duck). A cada hecho insertado en la lista de hechos se le asigna un identificador de hecho único que empieza por la letra "f " y va seguido de un número entero llamado **fact-index**. Al iniciar CLIPS, y después de ciertos comandos como **clear** y **reset** (que se discutirán con más detalle más adelante), el índice de hechos se pondrá a uno, y luego se incrementará en uno a medida que se afirme cada nuevo hecho.

¿Qué pasa si intentas poner un segundo pato en la lista de hechos? Probemos a ver. Afirma un nuevo (duck), luego ejecuta un comando (facts) tal como se muestra a continuación

```
CLIPS> (assert (duck))
<Fact-1>
CLIPS> (facts)
f-1 (duck)
For a total of 1 fact.
CLIPS>
```

CLIPS devuelve el mensaje <Fact-1> para indicar que el hecho ya existe. Verás sólo el original "f-1 (duck)". Esto muestra que CLIPS no aceptará una entrada duplicada de un hecho.

Sin embargo, existe un comando de anulación, **set-fact-duplication**, que permitirá la entrada duplicada de un hecho.

Por supuesto, puedes introducir otros hechos diferentes. Por ejemplo, afirma un hecho (quack) y luego emite un comando (facts). Verás lo siguiente

```
CLIPS> (assert (quack))
<Fact-2>
CLIPS> (facts)
f-1 (duck)
f-2 (quack)
For a total of 2 facts.
CLIPS>
```

Fíjate que el hecho (quack) está ahora en la lista de hechos.

Los hechos se pueden eliminar o **retractarse**. Cuando se retracta un hecho, no se modifican los índices de los demás hechos, por lo que pueden "faltar" índices de hechos (fact-indices). A modo de analogía, cuando un jugador de fútbol abandona un equipo y no es sustituido, los números de las camisetas de los demás jugadores no se ajustan debido al número que falta (a menos que odien a ese tipo y quieran olvidar que alguna vez jugó para ellos).

Esclarecer los hechos

El comando **clear** elimina todos los hechos de la memoria, tal como se muestra a continuación.

```
CLIPS> (facts)
f-1 (duck)
```

```
f-2 (quack)
For a total of 2 facts.
CLIPS> (clear)
CLIPS> (facts)
CLIPS>
```

El comando (clear) esencialmente restaura CLIPS a su estado inicial. Limpia la memoria de CLIPS y reinicia el identificador de hechos a uno. Para ver esto, afirma (animal-is-duck), luego comprueba la lista de hechos.

```
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS> (facts)
f-1 (animal-is duck)
For a total of 1 fact.
CLIPS>
```

Fíjate que (animal-is-duck) tiene un identificador de hecho de f-1 porque el comando (clear) reinició los identificadores de hechos. El comando (clear) hace algo más que eliminar hechos. Además de eliminar todos los hechos, (clear) también elimina todas las reglas, como verás en el próximo capítulo.

Campos sensibles y sorbentes

Se dice que un hecho como (duck) o (cuak) consta de un único **campo**. Un campo es un marcador de posición (con o sin nombre) que puede tener un valor asociado. Como analogía sencilla, puedes pensar en un campo como si fuera el marco de una foto. El marco puede contener una imagen, tal vez una foto de tu pato mascota (para aquellos de vosotros que teneis curiosidad por saber cómo es una imagen de un "cuac", podría ser (1) una foto de una traza de osciloscopio de un pato diciendo "cuak", donde la entrada de señal proviene de un micrófono, o (2) para aquellos de vosotros que sois más científicos, una Transformada Rápida de Fourier de la señal "cuak", o (3) un comerciante de TV vendiendo una cura milagrosa para las arrugas, perder peso, etc.). Los marcadores de posición con nombre sólo se utilizan con **deftemplate**, que se describen con más detalle en el capítulo 5.

El hecho (duck) tiene un único marcador de posición sin nombre para el valor duck. Este es un ejemplo de un hecho de **un solo campo**. Un campo es un contenedor para un valor. Como analogía a los campos, piensa en platos (campos) para contener comida (valores).

El **orden** de los campos sin nombre es importante. Por ejemplo, si se define un hecho

```
(Brian duck)
```

y una regla lo interpreta como que el cazador Brian disparó a un pato, entonces el hecho

(duck Brian)

significaría que el cazador duck disparó a Brian. En cambio, el orden de los campos con nombre no es significativo, como se verá más adelante con deftemplate.

En realidad, es buena práctica de ingeniería de software comenzar el hecho con una relación que describa los campos. Un hecho mejor descrito sería

(hunter-game duck Brian)

para dar a entender que el primer campo es el cazador y el segundo el juego.

Ahora son necesarias algunas definiciones. Una **lista** es un grupo de elementos sin orden implícito. Decir que una lista está **ordenada** significa que la posición en la lista es significativa. Un campo **múltiple** es una secuencia de campos, cada uno de los cuales puede tener un valor. Los ejemplos de (duck Brian) y (Brian duck) son hechos multicampo. Si un campo no tiene valor, se puede utilizar el símbolo especial **nil**, que significa "nada", para un campo vacío como marcador de posición. Por ejemplo,

(duck nil)

significaría que hoy el pato asesino no ha conseguido ningún trofeo.

Ten en cuenta que el *nil* es necesario para indicar un marcador de posición, aunque no tenga valor. Por ejemplo, piensa que un campo es análogo a un buzón de correo. Hay una gran diferencia entre un buzón vacío y la ausencia total de buzón. Sin el *nil*, el hecho se convierte en un hecho de un solo campo (duck). Si una regla depende de dos campos, no funcionará con un solo campo, como se verá más adelante.

Hay varios **tipos** de campos disponibles: **float**, **integer**, **symbol**, **string**, **external-address**, **fact-address**, **instance-name** e **instance-address**. El tipo de cada campo viene determinado por el tipo de **valor** almacenado en el campo. En un campo sin nombre, el tipo viene determinado implícitamente por el tipo que se pone en el campo. En las deftemplate, puedes declarar *explícitamente* el tipo de valor que puede contener un campo. La utilización de tipos explícitos refuerza los conceptos de **ingeniería de software**, que es una disciplina de programación para producir software de calidad.

Un **símbolo** es un tipo de campo que comienza con un carácter ASCII imprimible y va seguido opcionalmente por cero o más caracteres imprimibles. Los campos suelen estar **delimitados** por uno o varios espacios o paréntesis. Por ejemplo,

(duck-shot Brian Gary Rey)

Sin embargo, esto podría ser un hecho deftemplate legal si " shot " se define como el nombre de un campo, mientras que "Brian Gary Rey" son los valores asociados con ese nombre del campo.

CLIPS distingue entre mayúsculas y minúsculas. Además, algunos caracteres tienen un significado especial para CLIPS.

" ( ) & | < ~ ; ? $

Los caracteres "&", "|" y "~" no se pueden utilizar como símbolos independientes ni como parte de un símbolo.

Algunos caracteres actúan como delimitadores en la *terminación* de un símbolo. Los siguientes caracteres actúan como delimitadores de símbolos.

- cualquier carácter ASCII no imprimible, incluidos espacios, retornos de carro, tabulaciones y saltos de línea
- comillas dobles, **"**
- paréntesis de apertura y cierre, **()**
- ampersand, **&**
- barra vertical, **|**
- menos que, **<** . Ten en cuenta que puede ser el primer carácter de un símbolo
- tilde, **~**
- punto y coma, **;** indica el inicio de un comentario, un retorno de carro lo termina
- **?** y **$?** pueden no comenzar un símbolo, pero pueden estar dentro de él

El punto y coma actúa como inicio de un comentario en CLIPS. Si intentas introducir un punto y coma, CLIPS pensará que estás introduciendo un comentario y esperará a que termines. Si accidentalmente introduces un punto y coma en el nivel superior, simplemente escribe un paréntesis de cierre y un retorno de carro. CLIPS responderá con un mensaje de error y el prompt de CLIPS reaparecerá (Esta es una de las pocas ocasiones permitidas en la vida en las que es necesario hacer algo mal para hacer algo bien).

A medida que leas este manual, aprenderás los significados especiales de los caracteres anteriores. A excepción de "&", "|" y "~", puedes utilizar los demás como se ha indicado. Sin embargo, puede resultar confuso para alguien que lea tu programa e intente entender lo que hace. En general, es mejor evitar el uso de estos caracteres en los símbolos a menos que tengas alguna buena razón para usarlos.

Los siguientes son ejemplos de símbolos.

duck
duck1
duck_soup
duck-soup
duck1-1_soup-soup
d!?#%^

El segundo tipo de campo es la **cadena**. Una cadena debe empezar y terminar con comillas dobles. Las comillas dobles forman parte del campo. Entre las comillas dobles pueden aparecer cero o más caracteres de cualquier tipo. A continuación, se muestran algunos ejemplos de cadenas.

"duck"
"duck1"
"duck/soup"
"duck soup"
"duck soup is good!!!"

El tercer y cuarto tipo de campo son los **campos numéricos**. Un campo que representa un número puede ser de tipo **entero** o de tipo **coma flotante**. Un tipo de punto flotante se conoce comúnmente como **float**.

Todos los números en CLIPS se tratan como enteros "long long" o flotantes de **doble precisión**.

Los números sin punto decimal se tratan como enteros a menos que estén fuera del rango de los enteros.

El rango depende del número de bits, N, utilizados para representar el número entero de la siguiente manera.

- 2N-1
.
.
.
2N-1 - 1

Para los enteros "long long" de 64 bits, el rango de números es

- 9,223,372,036,854,775,808
.

.
.
9,223,372,036,854,775,807

Como algunos ejemplos de números, afirma los siguientes datos donde el último número está en notación exponencial, y utiliza la "e" o "E" para la potencia de diez.

```
CLIPS> (clear)
CLIPS> (facts)
CLIPS> (assert (number 1))
<Fact-1>
CLIPS> (assert (x 1.5))
<Fact-2>
CLIPS> (assert (y -1))
<Fact-3>
CLIPS> (assert (z 65))
<Fact-4>
CLIPS> (assert (distance 3.5e5))
<Fact-5>
CLIPS> (assert (coordinates 1 2 3))
<Fact-6>
CLIPS> (assert (coordinates 1 3 2))
<Fact-7>
CLIPS> (facts)
f-1 (number 1)
f-2 (x 1.5)
f-3 (y -1)
f-4 (z 65)
f-5 (distance 350000.0)
f-6 (coordinates 1 2 3)
f-7 (coordinates 1 3 2)
For a total of 7 facts.
CLIPS>
```

Como puedes ver, CLIPS imprime el número introducido en notación exponencial como 350000.0 porque convierte de formato de potencia de diez a coma flotante si el número es lo suficientemente pequeño.

Ten en cuenta que cada hecho debe empezar por un símbolo como "número", "x", "y", etc. Antes de la versión 6.0 de CLIPS, sólo era posible introducir un número como dato. Sin embargo, ahora se requiere un símbolo como primer campo. Además, ciertas palabras

reservadas utilizadas por CLIPS no se pueden utilizar como primer campo, pero sí para otros campos. Por ejemplo, la palabra reservada *not* se utiliza para indicar un patrón de negación y no puede utilizarse como primer campo de un hecho.

Un **hecho** se compone de uno o varios campos encerrados entre paréntesis a la izquierda y a la derecha. Para simplificar, sólo hablaremos de hechos en los siete primeros capítulos, pero la mayor parte de la discusión sobre la comparación de patrones se aplica también a los objetos. Se exceptúan algunas funciones, como assert y retract, las cuales sólo se aplican a hechos y no a objetos. Las correspondientes formas de gestinar objetos se discuten en los capítulos 8-12.

Un hecho puede estar **ordenado** o **desordenado**. Todos los ejemplos que has visto hasta ahora son hechos ordenados porque el orden de los campos es importante. Por ejemplo, fíjate que CLIPS considera los siguientes hechos como diferentes, aunque ambos usan los mismos valores "1", "2", y "3".

```
f-6 (coordinates 1 2 3)
f-7 (coordinates 1 3 2)
```

Los hechos ordenados deben utilizar la posición del campo para definir los datos. Por ejemplo, el hecho ordenado (duck Brian) tiene dos campos, que también los tiene (Brian duck). Sin embargo, CLIPS los considera como dos hechos separados porque el orden de los valores de los campos es diferente. Por el contrario, el hecho (duck-Brian) sólo tiene un campo debido al "-" que concatena los dos valores.

Los hechos Deftemplate, que se describirán con más detalle más adelante, no están ordenados porque utilizan campos con nombre para definir los datos. Esto es análogo al uso de estructuras (structs) en C y otros lenguajes.

Los campos múltiples normalmente están separados por **espacios en blanco** que consisten en uno o más espacios, tabulaciones, retornos de carro o saltos de línea. Por ejemplo, introduce los siguientes ejemplos y verás que cada dato almacenado es el mismo.

```
CLIPS> (clear)
CLIPS> (assert (The duck says "Quack"))
<Fact-1>
CLIPS> (facts)
f-1 (The duck says "Quack")
For a total of 1 fact.
CLIPS> (clear)
CLIPS> (assert (The duck says "Quack" ))
<Fact-1>
```

```
CLIPS> (facts)
f-1 (The duck says "Quack")
For a total of 1 fact.
CLIPS>
```

Los retornos de carro también pueden utilizarse para mejorar la legibilidad. En el siguiente ejemplo, se escribe un retorno de carro después de cada campo y el hecho afirmado es el mismo que antes, cuando se introducía en una línea.

```
CLIPS> (clear)
CLIPS> (assert (The
duck
says
"Quack"))
<Fact-1>
CLIPS> (facts)
f-1 (The duck says "Quack")
For a total of 1 fact.
CLIPS>
```

Sin embargo, ten cuidado si insertas un retorno de carro dentro de una cadena, como muestra el siguiente ejemplo.

```
CLIPS> (assert (The
duck
says
"Quack
"))
<Fact-2>
CLIPS> (facts)
f-1 (The duck says "Quack")
f-2 (The duck says "Quack
")
For a total of 2 facts.
CLIPS>
```

Como puedes ver, el retorno de carro incrustado en las comillas dobles se ha emitido con la cadena para poner la comilla doble de cierre en la línea siguiente. Esto es importante porque CLIPS considera el hecho f-1 como distinto del hecho f-2.

Fíjate también que CLIPS preservó las mayúsculas y minúsculas en los campos del hecho.

Es decir, la "T" de "The" y la "Q" de "Quack" son mayúsculas. Se dice que CLIPS **distingue entre mayúsculas y minúsculas** porque las diferencia entre ellas. Por ejemplo, afirma los hechos (duck) y (Duck) y luego emite un comando (facts). Verás que CLIPS te permite afirmar (duck) y (Duck) como hechos diferentes porque CLIPS distingue entre mayúsculas y minúsculas.

El siguiente ejemplo es un caso más realista en el que se utilizan retornos de carro para mejorar la legibilidad de una lista. Para ver esto, afirma el siguiente hecho donde los retornos de carro y los espacios se usan para poner campos en lugares apropiados en diferentes líneas. Los guiones o signos menos (-) se utilizan intencionadamente para crear campos individuales, de modo que CLIPS tratará elementos como " fudge-sauce " como un campo único.

```
CLIPS> (clear)
CLIPS>
(assert (grocery-list
 ice-cream
cookies
candy
fudge-sauce))
<Fact-1>
CLIPS> (facts)
f-1 (grocery-list ice-cream cookies candy fudge-sauce) For a total of 1 fact.
CLIPS>
```

Como puedes ver, CLIPS ha sustituido los retornos de carro y los tabuladores por espacios simples. Aunque el uso de espacios en blanco para separar los hechos es adecuado para una persona que lee un programa, CLIPS los convierte en espacios simples.

Una cuestión de estilo

Es un buen estilo de programación basado en reglas utilizar el primer campo de un hecho para describir la *relación* de los campos siguientes. Cuando se utiliza de esta forma, el primer campo se denomina *relación*. Los campos restantes del hecho se utilizan para valores específicos. Un ejemplo es (grocery-list ice-cream cookies candy fudge-sauce). Los guiones se utilizan para que quepan varias palabras en un solo campo.

Una buena documentación es aún más importante en un sistema experto que en lenguajes como Java, C, Ada, etc., porque las reglas de un sistema experto no suelen ejecutarse de forma secuencial. CLIPS ayuda al programador a escribir hechos descriptivos como éstos mediante deftemplate.

Otros ejemplos de hechos relacionados son (duck), (horse) y (cow). Un buen estilo de programación es referirse a ellos como

```
(animal-is duck)
(animal-is horse)
(animal-is cow)
```

o como el hecho único

```
(animals duck horse cow)
```

ya que la relación *animal-is* o *animals* describe su relación y proporciona así cierta documentación a la persona que lee el código.

Las relaciones explícitas, animal-is y animals, tienen más sentido para una persona que el significado implícito de (duck), (horse) y (cow). Aunque este ejemplo es lo suficientemente sencillo como para que cualquiera pueda averiguar las relaciones implícitas, es fácil caer en la trampa de escribir hechos en los que la relación *no* es tan obvia (de hecho, es mucho más fácil hacer algo más complicado que fácil, ya que a la gente le impresiona más la complejidad que la simplicidad).

Espaciándose

Dado que los espacios se utilizan para separar campos múltiples, se deduce que los espacios no pueden incluirse simplemente en los hechos. Por ejemplo,

```
CLIPS> (clear)
CLIPS> (assert (animal-is walrus))
<Fact-1>
CLIPS> (assert ( animal-is walrus ))
<Fact-1>
CLIPS> (assert ( animal-is walrus ))
<Fact-1>
CLIPS> (facts)
f-1 (animal-is walrus)
For a total of 1 fact.
CLIPS>
```

Sólo se afirma un hecho, (animal-is walrus), ya que CLIPS ignora los espacios en blanco y considera que todos estos hechos son equivalentes. Así, CLIPS responde con <Fact-1> cuando intentas introducir los dos últimos hechos duplicados. CLIPS normalmente no permite que se introduzcan hechos duplicados a menos que cambies el parámetro set-fact-duplication.

Si quieres incluir espacios en un dato, debes utilizar comillas dobles. Por ejemplo,

```
CLIPS> (clear)
CLIPS> (assert (animal-is "duck"))
<Fact-1>
CLIPS> (assert (animal-is "duck "))
<Fact-2>
CLIPS> (assert (animal-is " duck"))
<Fact-3>
CLIPS> (assert (animal-is " duck "))
<Fact-4>
CLIPS> (facts)
f-1 (animal-is "duck")
f-2 (animal-is "duck ")
f-3 (animal-is " duck")
f-4 (animal-is " duck ")
For a total of 4 facts.
CLIPS>
```

Ten en cuenta que los espacios hacen que cada uno de estos datos sean diferentes para CLIPS, aunque el significado sea el mismo para un humano.

¿Qué ocurre si quieres incluir las comillas dobles en un campo? La forma correcta de poner comillas dobles en un hecho es con la **barra invertida**, "\", como muestra el siguiente ejemplo.

```
CLIPS> (clear)
CLIPS> (assert (single-quote "duck"))
<Fact-1>
CLIPS> (assert (double-quote "\"duck\""))
<Fact-2>
CLIPS> (facts)
f-1 (single-quote "duck")
f-2 (double-quote ""duck"")
For a total of 2 facts.
CLIPS>
```

Retractar ese hecho

Ahora que ya sabes cómo poner hechos en la lista de hechos, es hora de aprender a eliminarlos.

La eliminación de hechos de la lista de hechos se denomina *retracción* y se realiza con el comando **retract**. Para retraer un hecho, debes especificar el índice del hecho como argumento de retract. Por ejemplo, configura tu lista de hechos de la siguiente manera.

```
CLIPS> (clear)
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS> (assert (animal-sound quack))
<Fact-2>
CLIPS> (assert (The duck says "Quack."))
<Fact-3>
CLIPS> (facts)
f-1 (animal-is duck)
f-2 (animal-sound quack)
f-3 (The duck says "Quack.")
For a total of 3 facts.
CLIPS>
```

Para eliminar el último hecho con el índice f-3, introduce el comando retract y, a continuación, comprueba tus hechos del siguiente modo.

```
CLIPS> (retract 3)
CLIPS> (facts)
f-1 (animal-is duck)
f-2 (animal-sound quack)
For a total of 2 facts.
CLIPS>
```

¿Qué pasa si intentas retractarte de un hecho ya retractado, o de un hecho inexistente? Intentémoslo y veamos.

```
CLIPS> (retract 3)
[PRNTUTIL1] Unable to find fact f-3.
CLIPS>
```

Fíjate en que CLIPS emite un mensaje de error si intentas retractarte de un hecho inexistente. La moraleja de esto es que no puedes retractarte de lo que no has dado.

Ahora vamos a retractarnos de los otros hechos de la siguiente manera.

```
CLIPS> (retract 2)
CLIPS> (facts)
```

```
f-1 (animal-is duck)
For a total of 1 fact.
CLIPS> (retract 1)
CLIPS> (facts)
CLIPS>
```

También se pueden retractar varios hechos a la vez, como se muestra a continuación.

```
CLIPS> (clear)
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS> (assert (animal-sound quack))
<Fact-2>
CLIPS> (assert (The duck says "Quack."))
<Fact-3>
CLIPS> (retract 1 3)
CLIPS> (facts)
f-2 (animal-sound quack)
For a total of 1 fact.
CLIPS>
```

Para retraer varios hechos, basta con enumerar los números de fact-id en el comando (retract).

Puedes utilizar **(retract *)** para retraer todos los hechos, donde "*" indica todos los hechos.

```
CLIPS> (clear)
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS> (assert (animal-sound quack))
<Fact-2>
CLIPS> (assert (The duck says "Quack."))
<Fact-3>
CLIPS> (facts)
f-1 (animal-is duck)
f-2 (animal-sound quack)
f-3 (The duck says "Quack.")
For a total of 3 facts.
CLIPS> (retract *)
CLIPS> (facts)
CLIPS>
```

Observar ese hecho

CLIPS proporciona varios comandos para ayudarte a depurar programas. Uno de estos comandos te permite **ver los hechos** que se afirman y se retractan. Esto es más práctico que tener que teclear el comando (facts) una y otra vez y tratar de averiguar qué ha cambiado en la lista de hechos.

Para empezar a ver hechos, introduce el comando **(watch facts)** como se muestra en el siguiente ejemplo.

```
CLIPS> (clear)
CLIPS> (watch facts)
CLIPS> (assert (animal-is duck))
==> f-1 (animal-is duck)
<Fact-1>
CLIPS>
```

El símbolo de **doble flecha a la derecha**, ==>, significa que un hecho está entrando en la memoria, mientras que la **doble flecha a la izquierda** indica que un hecho está saliendo de la memoria, tal como se ve a continuación.

```
CLIPS> (reset)
<== f-1 (animal-is duck)
CLIPS> (assert (animal-is duck))
==> f-1 (animal-is duck)
<Fact-1>
CLIPS> (retract 1)
<== f-1 (animal-is duck)
CLIPS> (facts)
CLIPS>
```

El comando (watch facts) proporciona un registro que muestra el estado **dinámico** o cambiante de la lista de hechos. Por el contrario, el comando (facts) muestra el estado **estático** de la lista de hechos, ya que indica el contenido actual de la lista de hechos. Para desactivar la visualización de hechos, introduce **(unwatch facts)**.

Hay varias cosas que se pueden ver. Entre ellas se incluyen las siguientes, que se describen con más detalle en el *Manual de referencia de CLIPS*. El **comentario** en CLIPS comienza con un **punto y coma**. CLIPS ignora todo lo que va después del punto y coma.

```
(watch facts)
; instances used with objects
```

```
(watch instances)
; slots used with objects
(watch slots)
(watch rules)
(watch activations)
; messages used with objects
(watch messages)
; message-handlers used with objects
(watch message-handlers)
(watch generic-functions)
(watch methods)
(watch deffunctions)
; compilations are watched by default
(watch compilations)
(watch statistics)
(watch globals)
(watch focus)
; all watches everything
(watch all)
```

A medida que uses más las capacidades de CLIPS, encontrarás estos comandos (watch) muy útiles en la depuración. Para desactivar un comando (watch), introduce un comando **unwatch**. Por ejemplo, para desactivar la visualización de compilations, introduce (unwatch compilations).

Capítulo 2

Seguir las reglas

Si quieres llegar a alguna parte en la vida, no rompas las reglas, ¡haz las reglas!

Crear buenas reglas

Para realizar un trabajo útil, un sistema experto debe tener reglas además de hechos. Ya que has visto cómo se afirman y se retractan los hechos, es hora de ver cómo funcionan las reglas. Una regla es similar a una sentencia IF THEN en un lenguaje procedimental como Java, C o Ada. Una regla IF THEN puede expresarse en una mezcla de lenguaje natural y lenguaje informático de la siguiente manera:

```
IF certain conditions are true
THEN execute the following actions.
```

Otro término para la afirmación anterior es **pseudocódigo**, que literalmente significa *código falso*. Aunque el pseudocódigo no puede ser ejecutado directamente por el ordenador, sirve de guía muy útil para escribir código ejecutable. El pseudocódigo también es práctico para documentar reglas. Una traducción de reglas del lenguaje natural a CLIPS no es muy difícil si se tiene en cuenta esta analogía IF THEN.

A medida que aumente tu experiencia con CLIPS, descubrirás que escribir reglas en CLIPS resulta muy sencillo. Puedes escribir las reglas directamente en CLIPS o cargarlas desde un archivo de reglas creado con un editor de texto.

El pseudocódigo para una regla sobre sonidos de patos podría ser

```
IF the animal is a duck
THEN the sound made is quack
```

A continuación se muestra un hecho y una regla llamada *duck*, que es el pseudocódigo anterior expresado en sintaxis CLIPS. El nombre de la regla sigue inmediatamente después de la palabra clave *defrule*. Aunque se puede introducir una regla en una sola línea, es habitual poner las diferentes partes en líneas separadas para facilitar la lectura y la edición.

```
CLIPS> (unwatch facts)
CLIPS> (clear)
```

```
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS>
(defrule duck
(animal-is duck)
=>
(assert (sound-is quack)))
CLIPS>
```

Si escribes la regla correctamente tal como se ha indicado previamente, debería aparecer el prompt CLIPS. Si no, aparecerá un mensaje de error. Si recibes un mensaje de error, es probable que hayas escrito mal una palabra clave o que hayas omitido un paréntesis. Recuerda, en una sentencia el número de paréntesis izquierdo y derecho siempre debe coincidir.

A continuación, se muestra la misma regla con comentarios añadidos para que coincida con las partes de la regla. También se muestra el comentario opcional del **encabezado de la regla** entre comillas, "Here comes the quack". Sólo puede haber un comentario de encabezado de regla y debe colocarse después del nombre de la regla y antes del primer **patrón**. Aunque ahora sólo estamos hablando de la comparación de patrones con hechos, de forma más general un patrón puede compararse con una **entidad patrón**. Una *entidad patrón* puede ser un hecho o una instancia de una clase definida por el usuario. La comparación de patrones con objetos se tratará más adelante.

CLIPS intenta comparar el patrón de la regla con una entidad patrón. Por supuesto, se pueden utilizar espacios en blanco consistentes en espacios, tabuladores y retornos de carro para separar los elementos de una regla y mejorar la legibilidad. Otros comentarios comienzan con un punto y coma y continúan hasta que se pulsa la tecla de retorno de carro para terminar una línea. CLIPS ignora los comentarios.

```
; Rule header
(defrule duck
        ; Comment
        "Here comes the quack"
        ; Pattern
        (animal-is duck)
        => ; THEN arrow
        ; Action
        (assert (sound-is quack)))
```

❖ *En CLIPS sólo puede existir un nombre de regla a la vez.*

Introducir el mismo nombre de regla, en este caso "duck", reemplazará cualquier regla existente con ese nombre. Es decir, aunque puede haber muchas reglas en CLIPS, sólo puede haber una regla que se llame "duck". Esto es análogo a otros lenguajes de programación en los que sólo se puede utilizar un nombre de procedimiento para identificarlo de forma unívoca.

La sintaxis general de una regla es la siguiente.

```
(defrule rule_name "optional_comment"
        (pattern_1) ; Left-Hand Side (LHS)
        (pattern_2) ; of the rule consisting
        . ; of elements before
        . ; the "=>"
        .
        (pattern_N)
        =>
        (action_1) ; Right-Hand Side (RHS)
        (action_2) ; of the rule consisting
        . ; of elements after
        . ; the "=>". The last ")"
        . ; balances the opening
        (action_M)) ; "(" to the left of
        ; "defrule". Be sure all
        ; your parentheses
        ; balance or you will
        ; get error messages
```

La regla completa debe ir entre paréntesis. Cada uno de los patrones de reglas y **acciones** debe ir entre paréntesis. Una acción es en realidad una función que normalmente no tiene **valor de retorno**, pero realiza alguna operación útil, como (assert) o (retract). Por ejemplo, una acción podría ser (assert (duck)). Aquí el nombre de la función es "assert" y su argumento es "duck". Fíjate que no queremos ningún valor de retorno, como un número. En su lugar, queremos que se asevere el hecho (duck). En CLIPS, una **función** es una pieza de código ejecutable identificada por un nombre específico, que devuelve un valor útil o realiza un efecto colateral útil, como (printout).

Una regla a menudo tiene múltiples patrones y acciones. El número de patrones y acciones no tiene por qué ser igual, por lo que se han elegido índices diferentes, N y M, para los patrones y las acciones de las reglas.

Se pueden escribir cero o más patrones después del encabezado de la regla. Cada patrón consta de uno o varios campos. En la regla del pato, el patrón es (animal-is-duck), donde los campos son "animal-is" y "duck". CLIPS intenta comparar los patrones de las reglas con los hechos de la lista de hechos (fact-list). Si todos los patrones de una regla concuerdan con los hechos, la regla se **activa** y se incluye en la **agenda**. La agenda es una colección de **activaciones** que son aquellas reglas que concuerdan con entidades patrón. Puede haber cero o más activaciones en la agenda.

El símbolo "=> " que sigue a los patrones en una regla se denomina **flecha**. La flecha representa el comienzo de la parte THEN de una regla IF-THEN (y puede leerse como "implica").

La última parte de una regla es la lista de cero o más acciones que se ejecutarán cuando se **dispare** la regla. En nuestro ejemplo, la única acción es afirmar el hecho (sound-is quack). El término *dispara* significa que CLIPS ha seleccionado una determinada regla de la agenda para su ejecución.

- *Un programa dejará de ejecutarse cuando no haya activaciones en la agenda.*

Cuando hay varias activaciones en la agenda, CLIPS determina automáticamente qué activación es la adecuada para disparar. CLIPS ordena las activaciones de la agenda en términos de prioridad o **prominencia** creciente.

La parte de la regla que se encuentra antes de la flecha se denomina lado izquierdo (**LHS**) y la parte de la regla que se encuentra después de la flecha se denomina lado derecho (**RHS**). Si no se especifica ningún patrón, CLIPS activa automáticamente la regla cuando se introduce un comando **(reset)**.

Hagamos cuack

CLIPS siempre ejecuta las acciones del lado derecho de la regla de mayor prioridad de la agenda. A continuación, esta regla se elimina de la agenda y se ejecutan las acciones de la nueva regla de mayor prioridad. Este proceso continúa hasta que no hay mas activaciones o se encuentra una orden de parada.

Puedes comprobar qué hay en la agenda con el comando **agenda**. Por ejemplo,

```
CLIPS> (agenda)
0 duck: f-1
For a total of 1 activation.
CLIPS>
```

El primer número "0" es la prioridad de la activación "duck", y "f-1" es el identificador del hecho (animal-is-duck) que coincide con la activación. Si la prioridad de una regla no se declara explícitamente, CLIPS le asigna el valor predeterminado de cero, donde los posibles valores van de -10.000 a 10.000. En este libro, usaremos la definición del término **por defecto** en el *sentido estándar*.

Si sólo hay una regla en la agenda, esa regla se disparará. Dado que el patrón LHS de la regla duck-sound es

```
(animal-is duck)
```

este patrón será satisfecho por el hecho (animal-is-duck) y por lo tanto la regla duck-sound debe dispararse.

Se dice que cada campo del patrón es una **restricción literal**. El término **literal** significa que tiene un valor constante, a diferencia de una variable cuyo valor se espera que cambie. En este caso, los literales son "animal-is" y "duck".

Para hacer que un programa se ejecute, basta con introducir la orden **run**. Escribe (run) y pulsa la tecla de retorno de carro. A continuación, haz un (facts) para comprobar que el hecho fue afirmado por la regla.

```
CLIPS> (run)
CLIPS> (facts)
f-1 (animal-is duck)
f-2 (sound-is quack)
For a total of 2 facts.
CLIPS>
```

Antes de continuar, vamos a guardar la regla del pato con el comando **save** para que no tengas que escribirla de nuevo (si no la has guardado ya en un editor). Simplemente introduce un comando como (save "duck.clp") para guardar la regla desde la memoria CLIPS al disco y nombra el archivo como "duck.clp" donde ".clp" es sencillamente una extensión práctica para recordarnos que éste es un fichero de código fuente CLIPS. Recuerda que guardar el código de la memoria CLIPS de esta manera sólo conservará el comentario opcional del encabezado de la regla entre comillas y no cualquier otro.

Patea tu pato

Puede que en este momento se te ocurra una pregunta interesante. ¿Qué pasa si se ejecuta (run) de nuevo? Hay una regla y un hecho que satisface la regla, por lo que la regla debería dispararse. Sin embargo, si lo intentas y ejecutas (run) de nuevo, verás que la regla no se

dispara. Esto puede ser algo frustrante. Sin embargo, antes de hacer algo drástico para aliviar tu frustración -como patear a tu pato mascota- necesitas saber un poco más sobre algunos principios básicos de los sistemas expertos.

Una regla se activa si sus patrones concuerdan con

1. una entidad patrón completamente nueva que no existía antes o,

2. una entidad patrón que sí existía antes pero que fue retractada y reafirmada, es decir, un "clon" de la antigua entidad patrón y que, por tanto, ahora es una nueva entidad patrón.

La regla y los índices de los patrones concordantes constituyen la activación. Si la regla o la entidad del patrón, o ambas, cambian, se elimina la activación. Una activación también puede ser eliminada por un comando o una acción de otra regla que se disparó antes y eliminó las condiciones necesarias para hacer efectiva la activación.

El motor de inferencia ordena las activaciones en función de su prioridad. Este proceso de ordenación se denomina **resolución de conflictos** porque elimina el conflicto de decidir qué regla debe activarse a continuación. CLIPS ejecuta el RHS de la regla con mayor prioridad en la agenda y elimina la activación. Esta ejecución se denomina disparo de la regla, por analogía con el disparo de una neurona. Una neurona emite un pulso eléctrico cuando se le aplica un estímulo adecuado. Después de que una neurona se ha disparado, sufre una **refracción** y no puede volver a dispararse durante un cierto periodo de tiempo. Sin refracción, las neuronas seguirían disparándose una y otra vez ante exactamente el mismo estímulo.

Sin refracción, los sistemas expertos siempre quedarían atrapados en bucles triviales. Es decir, en cuanto se dispara una regla, seguiría disparándose sobre ese mismo hecho una y otra vez. En el mundo real, el estímulo que provocó el disparo acabaría desapareciendo. Por ejemplo, un pato real podría huir nadando o conseguir un trabajo en el cine. Sin embargo, en el mundo del ordenador, una vez que se almacenan los datos, permanecen allí hasta que se eliminan explícitamente o se desconecta la alimentación.

El siguiente ejemplo muestra las activaciones y disparos de una regla. Fíjate que los comandos (watch) se utilizan para mostrar más cuidadosamente cada hecho y activación. La flecha que va hacia la derecha significa un hecho o activación entrante mientras que una flecha hacia la izquierda significaría un hecho o activación saliente.

```
; Comments in blue/italics have been
; added for explanation. You will
; not see these in the actual output
CLIPS> (clear)
CLIPS>
```

```
(defrule duck
(animal-is duck)
=>
(assert (sound-is quack)))
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (assert (animal-is duck))
==> f-1 (animal-is duck)
; Activation salience is 0 by default,
; then rule name:pattern entity index
==> Activation 0 duck: f-1
<Fact-1>
; Notice that duplicate fact
; cannot be entered
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS> (agenda)
0 duck: f-1
For a total of 1 activation.
CLIPS> (run)
==> f-2 (sound-is quack)
; Nothing on agenda after rule fires
; Even though fact matches rule,
; refraction will not allow this
; activation because the rule already
; fired on this fact
CLIPS> (agenda)
CLIPS> (facts)
f-1 (animal-is duck)
f-2 (sound-is quack)
For a total of 2 facts.
CLIPS> (run)
CLIPS>
```

Puedes hacer que la regla se dispare otra vez si retractas el hecho y luego lo afirmas de nuevo.

Enséñame las reglas

A veces puedes querer ver una regla mientras estás en CLIPS. Existe un comando llamado **ppdefrule** - pretty print rule - que imprime una regla. Para ver una regla, especifica el nombre de la regla como argumento de *ppdefrule*. Por ejemplo,

```
CLIPS> (ppdefrule duck)
(defrule MAIN::duck
(animal-is duck)
=>
(assert (sound-is quack)))
CLIPS>
```

CLIPS coloca diferentes partes de la regla en diferentes líneas para facilitar la lectura. Los patrones de la regla antes de la flecha se siguen considerando su LHS y las acciones después de la flecha se siguen considerando su RHS. El término *MAIN* se refiere al módulo MAIN en el que se encuentra esta regla por defecto.

Puedes definir módulos para colocar reglas de forma análoga a las sentencias que pueden colocarse en diferentes paquetes, módulos, procedimientos o funciones de otros lenguajes de programación. El uso de módulos facilita la escritura de sistemas expertos con muchas reglas, ya que éstas pueden agruparse con sus propias agendas para cada módulo. Para más información, consulta el *Manual de referencia de CLIPS*.

¿Qué pasa si quieres imprimir una regla, pero no recuerdas su nombre? No hay problema, simplemente utiliza el comando **rules** en respuesta a un prompt de CLIPS y te imprimirá los nombres de todas las reglas. Por ejemplo, introduce

```
CLIPS> (rules)
duck
For a total of 1 defrule.
CLIPS>
```

Escríbeme

Además de afirmar hechos en el RHS de las reglas, también puedes imprimir información utilizando la función **printout**. CLIPS también tiene una palabra clave de retorno de carro/salto de línea llamada **crlf** que es muy útil para mejorar la apariencia de la salida formateándola en diferentes líneas. Para variar, la *crlf* no se cierra entre paréntesis. A modo de ejemplo,

```
CLIPS>
(defrule duck
(animal-is duck)
=>
;; Be sure to type in the "t"
(printout t "quack" crlf))
```

```
==> Activation 0 duck: f-1
CLIPS> (run)
quack
CLIPS>
```

La salida es el texto entre comillas dobles. Asegúrate de escribir la letra "t" después del comando printout. Esto le indica a CLIPS que envíe la salida al **dispositivo de salida estándar** de tu ordenador. Generalmente, el dispositivo de salida estándar es tu terminal (de ahí la letra "t" después de printout). Sin embargo, esto puede redefinirse para que el dispositivo de salida estándar sea algún otro dispositivo, como un módem o un disco.

Otras características

El comando **declare salience** proporciona un control explícito sobre qué reglas se incluirán en la agenda. Debes tener cuidado al utilizar esta función con demasiada ligereza, no sea que tu programa se vuelva demasiado rígido (procedimental). Una forma de hacer que una regla se dispare de nuevo es forzar la reactivación de la regla mediante el comando **refresh** rule.

El comando **load** carga la regla que habías guardado previamente en el disco en el fichero "duck.clp" o cualquier otro nombre y directorio en el que la hubieras archivado. Puedes cargar un archivo de reglas hecho en un editor de texto en CLIPS usando el comando load.

Una forma más rápida de cargar archivos es guardarlos primero en un formato binario legible por la máquina con el comando para archivar binarios llamado **bsave**. El comando para cargar binarios, **bload**, se puede entonces usar para leer estas reglas binarias en la memoria CLIPS mucho más rápido, ya que los archivos no tienen que ser reinterpretados por CLIPS.

Otros dos comandos útiles permiten guardar y cargar hechos usando un archivo. Estos son **save-facts** y **load-facts**. El comando (save-facts) guardará todos los hechos de la lista de hechos en un archivo, mientras que (load-facts) cargará los hechos de un *archivo* en la *lista de hechos*.

El comando por lotes **batch** te permite ejecutar comandos desde un archivo como si se hubieran escrito en el nivel superior. Otro comando útil que proporciona una interfaz con tu sistema operativo, el comando **system,** permite la ejecución de comandos o ejecutables del sistema operativo dentro de CLIPS. Para más información sobre todos estos temas, consulte el *Manual de Referencia de CLIPS*.

Capítulo 3

Añadir detalles

El problema no es el panorama general, sino los detalles.

En los dos primeros capítulos, aprendiste los fundamentos de CLIPS. Ahora verás cómo construir sobre esa base para crear programas más potentes.

Parar y seguir

Hasta ahora, sólo has visto el tipo de programa más sencillo, que consiste en una sola regla. Sin embargo, los sistemas expertos que constan de una sola regla no son muy útiles. Los sistemas expertos prácticos pueden constar de cientos o miles de reglas. Veamos ahora una aplicación que requiere múltiples reglas.

Supongamos que queremos escribir un sistema experto para determinar cómo debe responder un robot móvil ante un semáforo. Lo mejor es escribir este tipo de problema utilizando múltiples reglas. Por ejemplo, las reglas para las situaciones de luz roja y verde se pueden escribir de la siguiente manera.

```
(defrule red-light
        (light red)
        =>
        (printout t "Stop" crlf))

(defrule green-light
        (light green)
        =>
        (printout t "Go" crlf))
```

Una vez introducidas las reglas en CLIPS, afirma un hecho (light red) y ejecuta (run). Verás impreso "Stop". Ahora afirma un hecho (light green) y ejecútalo de nuevo. Deberías ver impreso "Go".

De paseo

Si lo piensas bien, existen otras posibilidades además de los sencillos casos rojo, verde y amarillo. Algunos semáforos tienen también una flecha verde para los giros autorizados a la

izquierda. Algunos tienen una manecilla que se ilumina para indicar si una persona puede pasar o no. Algunos tienen señales que dicen camine o pare. Por lo tanto, dependiendo de si nuestro robot está caminando o conduciendo, tendrá que prestar atención a diferentes señales.

La información sobre caminar o conducir debe afirmarse además de la información sobre el estado de la luz. Se pueden hacer reglas para cubrir estas condiciones, pero deben tener más de un patrón. Por ejemplo, supongamos que queremos una regla que se dispare si el robot está caminando y si la señal dice caminar. Una regla podría escribirse de la siguiente manera:

```
(defrule take-a-walk
        (status walking)
        (walk-sign walk)
        =>
        (printout t "Go" crlf))
```

La regla anterior tiene dos patrones. Ambos patrones deben ser satisfechos por los hechos en la lista de hechos para que la regla se dispare. Para ver cómo funciona, introduce la regla y, a continuación, afirma los hechos (status walking) y (walk-sign walk). Cuando ejecutas (run), el programa imprimirá "Go" ya que ambos patrones se satisfacen y la regla se dispara.

Puedes tener cualquier número de patrones o acciones en una regla. Lo importante es darse cuenta de que la regla se incluye en la agenda si y sólo si todos los patrones se satisfacen con hechos. Este tipo de restricción se denomina **elemento condicional (CE) lógico AND** en referencia a la relación AND de la lógica booleana. Se dice que una relación AND es verdadera si y sólo si todas sus condiciones son verdaderas.

Dado que los patrones son del tipo lógico AND, la regla no se activará si sólo se cumple uno de los patrones. Todos los hechos deben estar presentes antes de que el LHS de una regla se satisfaga y ésta se coloque en la agenda.

Una cuestión de estrategia

La palabra **estrategia** era originalmente un término castrense para referirse a la planificación y las operaciones militares.

Hoy en día, el término *estrategia* se utiliza habitualmente en los negocios (porque los negocios son la guerra) para referirse a los planes de alto nivel de una organización para alcanzar sus objetivos, por ejemplo: "¡Ganar mucho dinero vendiendo más hamburguesas grasientas que nadie en el mundo!".

En los sistemas expertos, uno de los usos del término estrategia es en la resolución de conflictos de activaciones. Se podría decir: "Bueno, diseñaré mi sistema experto de forma que sólo se pueda activar una regla a la vez. Entonces no habrá necesidad de resolución de conflictos". La buena noticia es que, si tienes éxito, la resolución de conflictos es, en efecto, innecesaria. La mala noticia es que este éxito demuestra que tu aplicación puede representarse prefectamente mediante un programa secuencial. Así que deberías haberla codificado en C, Java o Ada en primer lugar y no molestarte en escribirla como un sistema experto.

CLIPS ofrece siete modos diferentes de resolución de conflictos: profundidad, amplitud, LEX, MEA, complejidad, simplicidad y aleatorio (depth, breadth, LEX, MEA, complexity, simplicity, y random). Es difícil decir que uno es claramente mejor que otro sin tener en cuenta una aplicación específica. Incluso entonces, puede ser difícil juzgar cuál es "mejor". Para más información sobre los detalles de estas estrategias, consulta el *Manual de referencia de CLIPS*.

La **estrategia en profundidad** es la **estrategia estándar por defecto** de CLIPS. La configuración por defecto se establece automáticamente cuando CLIPS se inicia por primera vez. Después, puedes cambiar la configuración predeterminada. En la estrategia en profundidad, las nuevas activaciones se colocan en la agenda después de las activaciones con mayor prioridad, pero antes de las activaciones con igual o menor prioridad. Esto significa simplemente que la agenda se ordena de mayor a menor prioridad.

- *En este libro, todas las discusiones y ejemplos presupondrán una estrategia en profundidad.*

Ahora que todos estos ajustes opcionales diferentes están disponibles, asegúrate de que antes de ejecutar un sistema experto desarrollado por otra persona, tus ajustes sean los mismos que los suyos. De lo contrario, es posible que el funcionamiento resulte ineficaz o incluso incorrecto. De hecho, es una buena idea codificar explícitamente todos los ajustes en cualquier sistema que desarrolles para que se configure adecuadamente.

Dame Deffacts

A medida que trabajes con CLIPS, puedes cansarse de teclear las mismas aserciones desde el nivel superior. Si vas a utilizar las mismas aserciones cada vez que se ejecute un programa, puedes cargarlas primero desde un disco utilizando un fichero batch. Una forma alternativa de introducir hechos es utilizando la palabra clave de definición de hechos, **deffacts**. Por ejemplo,

```
CLIPS> (unwatch facts)
CLIPS> (unwatch activations)
CLIPS> (clear)
```

```
CLIPS>
(deffacts walk "Some facts about walking"
; status fact to be asserted
(status walking)
; walk-sign fact to be asserted
(walk-sign walk))
CLIPS> (reset)
; reset causes facts from
; deffacts to be asserted
CLIPS> (facts)
f-1 (status walking)
f-2 (walk-sign walk)
For a total of 2 facts.
CLIPS>
```

El nombre requerido de esta sentencia deffacts, walk, sigue a la palabra clave deffacts. Tras el nombre hay un comentario opcional entre comillas dobles. Al igual que el comentario opcional de una regla, el comentario (deffacts) se conservará con la sentencia (deffacts) después de que CLIPS la haya cargado. Después del nombre o comentario se encuentran los hechos que se afirmarán en la lista de hechos. Los hechos de una sentencia deffacts se afirman mediante el comando CLIPS (reset).

El comando (reset) tiene una ventaja comparado con el comando (clear) y es que (reset) no se deshace de todas las reglas. El comando (reset) deja tus reglas intactas, al contrario que (clear), que elimina todas las reglas activadas de la agenda y también elimina todos los hechos antiguos de la lista de hechos. Dar una orden (reset) es una forma recomendable de iniciar la ejecución de un programa, especialmente si el programa ya se ha ejecutado antes y la lista de hechos está llena de hechos antiguos.

En resumen, (reset) hace dos cosas para los hechos.

1. Elimina los hechos existentes de la lista de hechos, lo que puede eliminar las reglas activadas de la agenda.

2. Afirma hechos a partir de declaraciones (deffacts) existentes.

En efecto, la orden (reset) también realiza las operaciones equivalentes en los objetos: Borra y afirma instancias de **definstances**.

Eliminación selectiva

El comando **undeffacts** elimina a (deffacts) de la afirmación de hechos, eliminando los deffacts de la memoria. Por ejemplo,

```
CLIPS> (undeffacts walk)
CLIPS> (reset)
CLIPS> (facts)
CLIPS>
```

Este ejemplo demuestra cómo se ha eliminado el (deffacts) walk. Para restaurar una sentencia deffacts después de un comando (undeffacts), debes introducir de nuevo la sentencia deffacts. Además de los hechos, CLIPS también permite eliminar reglas de forma selectiva mediante el comando **undefrule**.

¡Atención!

Puedes observar el disparo de reglas con **watch rules** y ver las activaciones con **watch activations** en la agenda. La visdualización de estadísticas **watch statistics** imprimen información sobre el número de reglas disparadas, el tiempo de ejecución, las reglas por segundo, el número medio de hechos, el número máximo de hechos, el número medio de activaciones y el número máximo de activaciones. La información estadística se ve mediante **tuning up** y puede ser útil para ajustar un sistema experto y optimizar su velocidad. Otro comando, llamado **watch compilations**, muestra información cuando se están cargando reglas. El comando **watch all** verá todo.

La impresión de la información de visualización en la pantalla o en el disco con el comando **dribble** ralentizará un poco el programa porque CLIPS utiliza más tiempo para imprimir o guardar en el disco. El comando **dribble-on** almacenará toda la entrada y salida introducida en el prompt de comandos en un fichero de disco hasta que se introduzca el comando **dribble-off**. Esto es útil para tener un registro permanente de todo lo que ocurre. Estos comandos son los siguientes.

```
(dribble-on <filename>)
(dribble-off <filename>)
```

Otro comando de depuración útil es (run) que toma como argumento opcional el número de disparos de una regla. Por ejemplo, un comando (run 21) le diría a CLIPS que ejecute el programa y se detenga después de 21 disparos de una regla. Un comando (run 1) te permite ejecutar un programa disparando una regla a la vez.

Al igual que muchos otros lenguajes de programación, CLIPS también te da la capacidad de establecer **puntos de interrupción**. Un punto de interrupción es simplemente un indicador para que CLIPS detenga la ejecución justo antes de ejecutar una regla específica. Un punto de

interrupción se establece mediante el comando **set-break**. El comando **remove-break** eliminará un punto de interrupción que haya sido creado previamente. El comando **show-breaks** listará todas las reglas que tienen puntos de interrupción establecidos. La sintaxis de estos comandos para el argumento <rulename> se muestra a continuación.

```
(set-break <rulename>)
(remove-break <rulename>)
(show-breaks)
```

Una buena concordancia

Puedes encontrarte con una situación en la que estés seguro de que una regla debería estar activada pero no lo esté. Aunque es posible que esto se deba a un error en CLIPS, no es muy probable debido a la gran habilidad de las personas que lo programaron (NOTA: ANUNCIO COMERCIAL PAGADO PARA LOS DESARROLLADORES).

En la mayoría de los casos, el problema se produce por la forma en que se escribió la regla. Como ayuda para la depuración, CLIPS tiene un comando llamado **matches** que puede decirte qué patrones de una regla concuerdan con qué hechos. Los patrones que no concuerdan impiden que la regla se active. Una razón común para que un patrón no coincida con un hecho es debido a escribir mal un elemento en el patrón o en la aserción del hecho.

El argumento de (matches) es el nombre de la regla que se comprobará si concuerda. Para ver cómo funciona (matches), primero haz (clear) y luego introduce la siguiente regla.

```
(defrule take-a-vacation
        ; Conditional element 1
        (work done)
        ; Conditional element 2
        (money plenty)
        ; Conditional element 3
        (reservations made)
        =>
        (printout t "Let's go!!!" crlf))
```

A continuación, se muestra cómo se utiliza (matches). Introduce los comandos como se indica a continuación. Fíjate que (watch facts) está activado. Esto es una buena idea cuando estás afirmando hechos manualmente ya que le das a CLIPS la oportunidad de comprobar la ortografía.

```
CLIPS> (watch facts)
CLIPS> (assert (work done))
```

```
==> f-1 (work done)
<Fact-1>
CLIPS> (matches take-a-vacation)
Matches for Pattern 1
f-1
Matches for Pattern 2
None
Matches for Pattern 3
None
; CE is conditional element
Partial matches for CEs 1 - 2
None
Partial matches for CEs 1 - 3
None
Activations
 None
; The return value indicates the total patterns
; matched, the total partial matches, and the
; total activations
(1 0 0)
CLIPS>
```

El hecho con identificador de hecho f-1 concuerda con el primer patrón o elemento condicional de la regla y se notifica mediante (matches). Dado que una regla tiene N patrones, el término **concordancias parciales** se refiere a cualquier conjunto de concordancias de los primeros N-1 patrones con hechos. Es decir, las concordancias parciales comienzan con el primer patrón de una regla y terminan con cualquier patrón hasta el último (n-enésimo), pero sin incluirlo. En el momento en que no se pueda realizar una concordancia parcial, CLIPS dejará de realizar más comprobaciones. Por ejemplo, una regla con cuatro patrones podría tener concordancias parciales del primer y segundo patrón y también del primer, segundo y tercer patrón. Si todos los N patrones concuerdan, la regla se activará.

Otras características

Algunos comandos adicionales son útiles junto con deffacts. Por ejemplo, el comando **list-deffacts** listará los nombres de los deffacts actualmente cargados en CLIPS. Otro comando útil es **ppdeffacts** que imprime los hechos almacenados en una deffacts.

Otras funciones permiten manipular cadenas fácilmente:

assert-string: Realiza una aserción de cadena tomando una cadena como argumento y afirmada como un hecho no-string.
str-cat: Construye una cadena de comillas simples a partir de elementos individuales mediante concatenación de cadenas.
str-index: Devuelve un índice de cadena de la primera aparición de una subcadena.
sub-string: Devuelve una subcadena de una cadena.
str-compare: Realiza una comparación de cadenas.
str-length: Devuelve la longitud de una cadena.
sym-cat: Devuelve un símbolo concatenado.

Si quieres imprimir una variable multicampo sin paréntesis, la forma más sencilla es utilizar la función de implosión de cadenas, implode$.

Capítulo 4

Variables de interés

Nada cambia más que el cambio

El tipo de reglas que has visto hasta ahora ilustra la correspondencia simple de patrones con hechos. En este capítulo, aprenderás formas muy efectivas de emparejar y gestionar hechos.

Vamos a variar

Al igual que otros lenguajes de programación, CLIPS dispone de **variables** para almacenar valores. A diferencia de un hecho, que es **estático** o inmutable, el contenido de una variable es **dinámico** a medida que cambian los valores que se le asignan. Por el contrario, una vez que se afirma un hecho, sus campos sólo se pueden modificar retrayendo el hecho y afirmando uno nuevo con los campos cambiados, Incluso la acción *modify* (descrita más adelante en el capítulo sobre deftemplate) actúa retrayendo y afirmando un hecho modificado, como puedes ver comprobando el fact-index.

El nombre de una variable, o **identificador de variable**, se escribe siempre con un signo de interrogación seguido de un símbolo que es el nombre de la variable (sin espacio entre ellos). El formato general es

```
?<variable-name>
```

Las variables globales, que se describirán con más detalle más adelante, tienen una sintaxis ligeramente distinta.

Al igual que en otros lenguajes de programación, los nombres de las variables deben tener sentido para lograr un buen estilo de programación.

A continuación, se ofrecen algunos ejemplos de nombres válidos de variables.

```
?x
?noun
?color
?sensor
?valve
?ducks-eaten
```

Antes de poder utilizar una variable es necesario asignársele un valor. Como ejemplo de un caso en el que no se asigna un valor, intenta introducir lo siguiente y CLIPS responderá con el mensaje de error tal como se ve a continuación.

```
CLIPS> (unwatch all)
CLIPS> (clear)
CLIPS>
(defrule test
=>
(printout t ?x crlf))
[PRCCODE3] Undefined variable x referenced in RHS of defrule.
ERROR:
(defrule MAIN::test
=>
(printout t ?x crlf))
CLIPS>
```

CLIPS muestra un mensaje de error cuando no puede encontrar un valor **ligado** a ?x. El término *ligado* significa la asignación de un valor a una variable. Sólo las variables globales están ligadas a *todas* las reglas. El resto de variables sólo están ligadas *dentro* de una regla. Antes y después de que se dispare una regla, las variables no globales no están ligadas, por lo que CLIPS dará un mensaje de error si intentas consultar una variable no ligada.

Ser asertivo

Un uso común de las variables es hacer coincidir un valor en el LHS y luego afirmar esta variable ligada en el RHS. Por ejemplo, introduce

```
(defrule make-quack
        (duck-sound ?sound)
        =>
        (assert (sound-is ?sound)))
```

Ahora afirma (duck-sound quack), luego ejecuta el programa con (run). Comprueba los hechos y verás que la regla ha producido (sound-is quack) porque la variable ?sound estaba ligada a quack.

Por supuesto, también puedes utilizar una variable más de una vez. Por ejemplo, introduce lo siguiente, asegurándote de hacer un (reset) y aseverar (duck-sound quack) de nuevo.

```
(defrule make-quack
        (duck-sound ?sound)
```

```
    =>
    (assert (sound-is ?sound ?sound)))
```

Cuando la regla se dispara, producirá (sound-is quack quack) ya que la variable ?sound se ha utilizado dos veces en la aserción.

Qué dijo el pato

Las variables también se utilizan comúnmente en la salida de impresión, como en (defrule make-quack)

```
(duck-sound ?sound)
    =>
    (printout t "The duck said "
    ?sound crlf))
```

Haz un (reset), introduce esta regla, afirma el hecho y luego haz (run) para averiguar lo que dijo el pato. ¿Cómo modificarías la regla para poner comillas dobles alrededor de quack en la salida?

Se puede utilizar más de una variable en un patrón, como muestra el siguiente ejemplo.

```
CLIPS> (clear)
CLIPS>
(defrule whodunit
(duckshoot ?hunter ?who)
=>
(printout t ?hunter " shot "
?who crlf))
CLIPS> (assert (duckshoot Brian duck))
<Fact-1>
; Duck dinner tonight!
CLIPS> (run)
Brian shot duck
CLIPS> (assert (duckshoot duck Brian))
<Fact-2>
; Brian dinner tonight!
CLIPS> (run)
duck shot Brian
; Missing third field
CLIPS> (assert (duckshoot duck))
<Fact-3>
```

```
; Rule doesn't fire,
; no output
CLIPS> (run)
CLIPS>
```

Observa la gran diferencia que supone el orden de los campos a la hora de determinar quién disparó a quién. También puedes ver que la regla *no* se activó cuando se afirmó el hecho (duckshoot duck). La regla no se activó porque ningún campo del hecho coincidía con la segunda restricción de patrón, ?who.

El soltero feliz

La retracción es muy útil en los sistemas expertos y suele realizarse en el RHS en lugar de en el nivel superior. Antes de que un hecho pueda ser retraído, se deberá especificar a CLIPS. Para retraer un hecho de una regla, la **dirección del hecho** primero debe estar ligada a una variable en el LHS.

Hay una gran diferencia entre vincular una variable al contenido de un hecho o vincular una variable a la dirección del hecho. En los ejemplos que has visto como (duck-sound ?sound), una variable estaba ligada al valor de un campo. Es decir, ?sound estaba ligado a quack. Sin embargo, si quieres eliminar el hecho cuyo contenido es (duck-sound quack), primero debes decirle a CLIPS la *dirección* del hecho a retraer.

La dirección del hecho se especifica usando la **flecha izquierda**, "<-". Para crearla, basta con escribir un símbolo "<" seguido de un "-". Un ejemplo de retracción de hecho de una regla es el siguiente,

```
CLIPS> (clear)
CLIPS> (assert (bachelor Dopey))
<Fact-1>
CLIPS> (facts)
f-1 (bachelor Dopey)
For a total of 1 fact.
CLIPS>
(defrule get-married
?duck <- (bachelor Dopey)
=>
(printout t "Dopey is now happily married "
?duck crlf)
(retract ?duck))
CLIPS> (run)
```

```
Dopey is now happily married <Fact-1>
CLIPS> (facts)
CLIPS>
```

Fíjate que (printout) imprime el fact-index de ?duck, <Fact-1>, ya que la flecha izquierda ligó la dirección del hecho a ?duck. Además, no hay ningún hecho (bachelor Dopey) porque ha sido retraído.

Las variables se pueden utilizar para recoger un valor de hecho al mismo tiempo que una dirección, como se muestra en el siguiente ejemplo. Por comodidad, también se ha definido una (deffacts).

```
CLIPS> (clear)
CLIPS>
(defrule marriage
?duck <- (bachelor ?name)
=>
(printout t ?name
" is now happily married"
crlf)
(retract ?duck))
CLIPS>
(deffacts good-prospects
(bachelor Dopey)
(bachelor Dorky)
(bachelor Dicky))
CLIPS> (reset)
CLIPS> (run)
Dicky is now happily married
Dorky is now happily married
Dopey is now happily married
CLIPS>
```

Fíjate cómo la regla se dispara en *todos* los hechos que concuerdan con el patrón (bachelor ?name). CLIPS también tiene una función llamada **fact-index** que se puede utilizar para devolver el índice de hecho de una dirección de hecho.

No tiene importancia

En lugar de vincular un valor de campo a una variable, la presencia de un campo no vacío puede detectarse por sí sola utilizando un **comodín**. Por ejemplo, supongamos que tienes un

servicio de citas para patos y una patita afirma que sólo sale con patos cuyo nombre de pila sea Dopey. En realidad, hay dos criterios en esta especificación, ya que hay una implicación de que el pato debe tener más de un nombre. Así que un simple (bachelor Dopey) no es adecuado porque sólo hay un nombre en el hecho.

Este tipo de situación, en la que sólo se especifica una parte del hecho, es muy común y muy importante. Para resolver este problema, se puede utilizar un comodín para hacer coincidir los Dopeys.

La forma más sencilla de comodín se denomina **comodín de campo único** y se muestra mediante un signo de interrogación, "? ". El "?" también se denomina restricción de campo único. Un comodín de un solo campo representa exactamente un campo, tal como podemos ver a continuación.

```
CLIPS> (clear)
CLIPS>
(defrule dating-ducks
(bachelor Dopey ?)
=>
(printout t "Date Dopey"
crlf))
CLIPS>
(deffacts duck
(bachelor Dicky)
(bachelor Dopey)
(bachelor Dopey Mallard)
(bachelor Dinky Dopey)
(bachelor Dopey Dinky Mallard))
CLIPS> (reset)
CLIPS> (run)
Date Dopey
CLIPS>
```

El patrón incluye un comodín para indicar que el apellido de Dopey no es importante. Siempre que el nombre sea Dopey y haya cualquier apellido (pero ningún segundo nombre), la regla se cumplirá y se disparará. Dado que el patrón tiene tres campos, de los cuales uno es un comodín de un solo campo, sólo los hechos de *exactamente* tres campos pueden satisfacerlo. En otras palabras, sólo los Dopeys con exactamente dos nombres pueden satisfacer a esta patita.

Supongamos que quieres especificar Dopeys con exactamente tres nombres. Todo lo que tendrías que hacer es escribir un patrón como

```
(bachelor Dopey ? ?)
```

o, si sólo quieres las personas con tres nombres cuyo segundo nombre sea Dopey, como

```
(bachelor ? Dopey ?)
```

o, si sólo quieres que el apellido sea Dopey, como:

```
(bachelor ? ? Dopey)
```

Otra posibilidad interesante se da si Dopey *debe* ser el primer nombre, pero sólo son aceptables los Dopeys con dos o tres nombres. Una forma de resolver este problema es escribir dos reglas. Por ejemplo

```
(defrule eligible
      (bachelor Dopey ?)
      =>
      (printout t "Date Dopey" crlf))

(defrule eligible-three-names
      (bachelor Dopey ? ?)
      =>
      (printout t "Date Dopey" crlf))
```

Escribe y ejecuta esto y verás que se imprimen los Dopeys con dos y tres nombres. Por supuesto, si no quieres fechas anónimas, necesitas enlazar los nombres de Dopey con una variable e imprimirlos.

A lo loco

En lugar de escribir reglas separadas para cada campo, es mucho más fácil utilizar el **comodín multicampo**. Se trata de un signo de dólar seguido de un signo de interrogación, "**$?** ", y representa *cero o más campos*. Fíjate cómo esto contrasta con el comodín de un solo campo, que debe coincidir exactamente con uno y solamente uno.

Las dos reglas para las fechas se pueden escribir ahora en una únca regla de la siguiente manera.

```
CLIPS> (clear)
CLIPS>
```

```
(defrule dating-ducks
(bachelor Dopey $?)
=>
(printout t "Date Dopey" crlf))
CLIPS>
(deffacts duck
(bachelor Dicky)
(bachelor Dopey)
(bachelor Dopey Mallard)
(bachelor Dinky Dopey)
(bachelor Dopey Dinky Mallard))
CLIPS> (reset)
CLIPS> (run)
Date Dopey
Date Dopey
Date Dopey
CLIPS>
```

Los comodines tienen otro uso importante porque pueden adjuntarse a un campo simbólico para crear una variable como ?x, $?x, ?name o $?name. La variable puede ser de **campo único** o una **variable multicampo** dependiendo de si se utiliza "?" o "$?" en el LHS. Ten en cuenta que en el lado derecho sólo se utiliza ?x, donde "x" puede ser cualquier nombre de variable. Puedes pensar en "$" como una función cuyo argumento es un comodín de campo único o una variable de campo único y devuelve un comodín multicampo o una variable multicampo, respectivamente.

Como ejemplo de variable multicampo, la siguiente versión de la regla también imprime el/los campo(s) de nombre del hecho concuerdante porque una variable se equipara al/los campo(s) de nombre que concuerdan:

```
CLIPS>
(defrule dating-ducks
(bachelor Dopey $?name)
=>
(printout t "Date Dopey " ?name crlf))
CLIPS> (reset)
CLIPS> (run)
Date Dopey (Dinky Mallard)
Date Dopey (Mallard)
Date Dopey ()
CLIPS>
```

Como puedes ver, en el LHS, el patrón multicampo es *$?name* pero es *?name* cuando se usa como variable en el RHS. Cuando introduzcas el programa y lo ejecutes, verás los nombres de todos los Dopeys elegibles. El comodín multicampo se encarga de cualquier número de campos. Fíjaste también que los valores multicampo se devuelven encerrados entre paréntesis.

Supongamos que quieres obtener una correspondencia de todos los patos que tengan Dopey en algún lugar de su nombre, no necesariamente como nombre de pila. La siguiente versión de la regla haría corresponder todos los hechos con un Dopey en ellos y luego imprimiría los nombres:

```
CLIPS>
(defrule dating-ducks
(bachelor $?first Dopey $?last)
=>
(printout t "Date "
?first
" Dopey "
?last crlf))
CLIPS> (reset)
CLIPS> (run)
Date () Dopey (Dinky Mallard)
Date (Dinky) Dopey ()
Date () Dopey (Mallard)
Date () Dopey ()
CLIPS>
```

El patrón coincide con cualquier nombre que tenga un Dopey en cualquier parte.

Se pueden combinar comodines de uno o varios campos. Por ejemplo, el patrón (bachelor ? $? Dopey ?) significa que el nombre y el apellido pueden ser cualquier cosa y que el nombre justo antes del último debe ser Dopey. Este patrón también requiere que el hecho concordante tenga al menos cuatro campos, ya que el "$?" coincide con cero o más campos y todos los demás deben coincidir exactamente con cuatro.

Aunque las variables multicampo pueden ser esenciales para la concordancia de patrones en muchos casos, su uso excesivo puede causar mucha ineficacia debido al aumento de los requisitos de memoria y a una ejecución más lenta.

- *Como regla general de estilo de programación, debes utilizar $? sólo cuando no conozcas la longitud de los campos. No utilices $? simplemente para facilitar la escritura.*

El soltero ideal

Las variables utilizadas en los patrones tienen una propiedad importante y útil, que puede enunciarse del siguiente modo.

- *La primera vez que se vincula una variable, conserva ese valor sólo dentro de la regla, tanto en el LHS como en el RHS, a menos que se cambie en el RHS.*

Por ejemplo, en la regla siguiente

```
(defrule bound
        (number-1 ?num)
        (number-2 ?num)
        =>)
```

Si hay algunos hechos

```
f-1 (number-1 0)
f-2 (number-2 0)
f-3 (number-1 1)
f-4 (number-2 1)
```

entonces la regla sólo puede ser activada por el par f-1, f-2, y el otro par f-3, f-4. Es decir, el hecho f-1 no puede concordar con f-4 porque cuando ?num está ligado a 0 en el primer patrón, el valor de ?num en el segundo patrón también debe ser 0. Del mismo modo, cuando ?num está ligado a 1 en el primer patrón, el valor de ?num en el segundo patrón *debe* ser 1. Fíjate que la regla se activará dos veces por estos cuatro hechos: una activación para el par f-1, f-2, y la otra activación para el par f-3, f-4.

Como ejemplo más práctico, introduce la siguiente regla. Fíjate que la misma variable, ?name, se utiliza en ambos patrones. Antes de hacer un (reset) y un (run), introduce también un comando (watch all) para que puedas ver lo que ocurre durante la ejecución.

```
CLIPS> (clear)
CLIPS>
(defrule ideal-duck-bachelor
(bill big ?name)
(feet wide ?name)
=>
(printout t "The ideal duck is "
?name crlf))
CLIPS>
```

```
(deffacts duck-assets
(bill big Dopey)
(bill big Dorky)
(bill little Dicky)
(feet wide Dopey)
(feet narrow Dorky)
(feet narrow Dicky))
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (reset)
==> f-1 (bill big Dopey)
==> f-2 (bill big Dorky)
==> f-3 (bill little Dicky)
==> f-4 (feet wide Dopey)
==> Activation 0 ideal-duck-bachelor: f-1,f-4
==> f-5 (feet narrow Dorky)
==> f-6 (feet narrow Dicky)
CLIPS> (run)
The ideal duck is Dopey
CLIPS>
```

Cuando se ejecuta el programa, el primer patrón coincide con Dopey y Dorky, ya que ambos tienen billetes grandes (bill big). La variable ?name está asociada a cada nombre. Cuando CLIPS intenta emparejar con el segundo patrón de la regla, sólo la variable ?name que está ligada a Dopey también satisface el segundo patrón de (feet wide).

El pato afortunado

En la vida se dan muchas situaciones en las que conviene hacer las cosas de forma sistemática. De ese modo, si tus expectativas no funcionan, puedes volver a intentarlo repetidamente (como el algoritmo habitual para encontrar al Cónyuge Perfecto casándose una y otra vez).

Una forma de organizarse es llevar una lista. (Nota: si *de verdad* quieres impresionar a la gente, enséñales una lista de tus listas.) En nuestro caso, mantendremos una lista de patos solteros, con el candidato más probable al inicio. Una vez identificado el pato soltero ideal, lo pondremos al principio de la lista como pato afortunado.

El siguiente programa muestra cómo hacerlo añadiendo un par de reglas a la regla ideal-duck-bachelor.

```
(defrule ideal-duck-bachelor
        (bill big ?name)
        (feet wide ?name)
        =>
        (printout t "The ideal duck is " ?name crlf)
        (assert (move-to-front ?name)))

(defrule move-to-front
        ?move-to-front <- (move-to-front ?who)
        ?old-list <- (list $?front ?who $?rear)
        =>
        (retract ?move-to-front ?old-list)
        (assert (list ?who ?front ?rear))
        (assert (change-list yes)))

(defrule print-list
        ?change-list <- (change-list yes)
        (list $?list)
        =>
        (retract ?change-list)
        (printout t "List is : " ?list crlf))

(deffacts duck-bachelor-list
        (list Dorky Dinky Dicky))

(deffacts duck-assets
        (bill big Dicky)
        (bill big Dorky)
        (bill little Dinky)
        (feet wide Dicky)
        (feet narrow Dorky)
        (feet narrow Dinky))
```

La lista original se da en las deffacts duck-bachelor-list. Cuando se ejecute el programa, proporcionará una nueva lista de candidatos probables.

```
CLIPS> (unwatch all)
CLIPS> (reset)
CLIPS> (run)
The ideal duck is Dicky
List is : (Dicky Dorky Dinky)
```

```
CLIPS>
```

Observa la aserción (change-list yes) en la regla move-to-front. Sin esta aserción, la regla print-list siempre se dispararía en la lista original. Esta aserción es un ejemplo de un **hecho de control** que sirve para controlar el disparo de otra regla. Los hechos de control son muy importantes para regular la activación de determinadas reglas, y deberías estudiar este ejemplo cuidadosamente para entender por qué se usan. Otro método de control son los módulos, como se discute en el *Manual de Referencia CLIPS*.

La regla move-to-front retira la lista antigua y afirma la lista nueva. Si la lista antigua no fuera retirada, estarían dos activaciones en la agenda para la regla print-list, pero sólo una se dispararía.

Sólo se activará una porque la regla print-list elimina el hecho de control necesario para la siguiente activación de la misma regla. No se sabría de antemano cuál se activaría, por lo que podría imprimirse la lista antigua en lugar de la nueva.

Capítulo 5

Hacerlo con estilo

Hoy a la moda, mañana no

En este capítulo, aprenderás sobre una palabra clave llamada deftemplate, que significa definir plantilla. Esta función puede ayudarte a escribir reglas cuyos patrones tengan una estructura bien definida.

Sr. Estupendo

Deftemplate es análoga a una definición struct en C. Es decir, la deftemplate define un grupo de campos relacionados en un patrón similar a la forma en que un struct en C es un grupo de datos relacionados. Una deftemplate es una lista de campos con nombre llamados slots (ranuras). Deftemplate permite el acceso por nombre en lugar de especificar el orden de los campos. Deftemplate contribuye a un buen estilo en los programas de sistemas expertos y es una valiosa herramienta de ingeniería de software.

Un slot es una ranura simple o múltiple con nombre. Una ranura simple contiene exactamente un campo, mientras que una ranura múltiple contiene cero o más campos. Se puede utilizar cualquier número de ranuras simples o múltiples en una plantilla. Para escribir una ranura, indica el nombre del campo (atributo) seguido del valor del campo.

Ten en cuenta que una ranura múltiple con un valor no es estrictamente lo mismo que una ranura simple. Como analogía, piensa en un armario (la ranura mútiple) que puede contener platos. Un armario con un plato no es lo mismo que un plato (ranura única). Sin embargo, el valor de un slot (o variable) de una sola ranura puede coincidir con un slot (o variable) de varias ranuras que contenga un campo.

Como ejemplo de relación deftemplate, consideremos los atributos de un pato que podría pasar por un buen pretendiente a casarse:

name
    "Dopey Wonderful"
assets
    rich
age
    99

Se puede definir una deftemplate para el *futuro* de relación de la siguiente manera, en la que se utilizan espacios en blanco y comentarios para facilitar la lectura y la explicación.

```
; name of deftemplate relation
(deftemplate prospect
        ; optional comment in quotes
        "vital information"
        ; name of field
        (slot name
        ; type of field
        (type STRING)
        ; default value of field name
        (default ?DERIVE))
        ; name of field
        (slot assets
        ; type of field
        (type SYMBOL)
        ; default value of field assets
        (default rich))
        ; name of field
        (slot age
        ; type. NUMBER can be
        ; INTEGER or FLOAT
        (type NUMBER)
        ; default value of field age
        (default 80)))
```

En este ejemplo, los componentes de deftemplate se estructuran como:

- Un nombre de relación deftemplate.
- Atributos denominados *campos*.
- El tipo de campo, que puede ser cualquiera de los tipos permitidos: símbolo, cadena, número y otros.
- El valor predeterminado para el valor del campo.

Esta deftemplate en particular tiene tres slots de una sola ranura llamadas *name*, *assets* y *age*.

Los **valores** por defecto de la deftemplate son insertados por CLIPS cuando se realiza un (reset) si no se definen valores explícitos. Por ejemplo, introduce la deftemplate para prospect después de un comando (clear), y afirmalo tal como se muestra a continuación.

```
CLIPS> (assert (prospect))
<Fact-1>
CLIPS> (facts)
f-1 (prospect (name "") (assets rich) (age 80)) For a total of 1 fact.
CLIPS>
```

Como puedes ver, CLIPS ha insertado el valor por defecto de la cadena nula, "", para el campo name, ya que es el valor por defecto para una STRING. Del mismo modo, CLIPS también ha insertado los valores por defecto de los campos asset y age. Los diferentes tipos tienen diferentes símbolos por defecto, como la cadena nula, "", para STRING; el entero 0 para INTEGER; el float 0.0 para FLOAT; etc. La palabra clave ?DERIVE selecciona el tipo de restricción adecuado para ese slot, por ejemplo, la cadena nula , "", para una ranura de tipo STRING.

Puedes definir explícitamente los valores de campo, como se ilustra en el siguiente ejemplo.

```
CLIPS>
(assert (prospect (age 99)
(name "Dopey")))
<Fact-2>
CLIPS> (facts)
f-1 (prospect (name "") (assets rich) (age 80)) f-2 (prospect (name "Dopey") (assets rich) (age 99)) For a total of 2 facts.
CLIPS>
```

Ten en cuenta que el orden en el que se escriben los campos no importa, ya que se trata de campos con nombre.

En la deftemplate, es importante darse cuenta de que NUMBER *no* es un tipo de campo primitivo como symbol, string, integer o float. El NUMBER es realmente un tipo compuesto que puede ser integer *o* float. Se utiliza para casos en los que al usuario no le importa qué tipo de números se almacenan. Una alternativa a NUMBER sería especificar los tipos de la siguiente manera.

```
(slot age (type INTEGER FLOAT) (default 80)))
```

Adiós

En resumen, una deftemplate con N ranuras tiene la siguiente estructura general:

```
(deftemplate <name>
        (slot-1)
```

(slot-2)
.
.
.
(slot-N))

En una deftemplate, los valores de los atributos pueden especificarse con mayor precisión que un simple valor como *80* o *rich*. Por ejemplo, en la anterior deftemplate, se especificó un **tipo** de valor.

Los valores de campo se pueden especificar mediante una lista explícita o dando un rango de valores.

Los *valores permitidos* pueden ser de cualquier tipo primitivo, como SYMBOL, STRING, INTEGER, FLOAT, etc. A continuación, se muestran algunos ejemplos deftemplate de valores enumerados:

allowed-symbols
rich filthy-rich loaded
allowed-strings
"Dopey" "Dorky" "Dicky"
allowed-numbers
1 2 3 4.5 -2.001 1.3e-4
allowed-integers
-100 53
allowed-floats
-2.3 1.0 300.00056
allowed-values
"Dopey" rich 99 1.e9

No tiene sentido especificar tanto un rango numérico como valores permitidos para el mismo campo de una deftemplate. Por ejemplo, si especificas (allowed-integers 1 4 8), esto contradice una especificación de rango de 1 a 10 mediante (range 1 10). Si los números son secuenciales, como 1, 2, 3, podrías especificar un rango que coincidiera exactamente: (rango 1 3). Sin embargo, el rango sería redundante para la especificación de los enteros permitidos. Por lo tanto, rango y valores permitidos *se excluyen mutuamente*. Es decir, si especificas un rango, no puedes especificar valores permitidos y *viceversa*.

En general, el atributo range no puede utilizarse junto con allowed-values, allowed-numbers, allowed-integers o allowed-floats.

A continuación, se muestra la deftemplate sin esta información opcional y una regla que la utiliza.

```
CLIPS> (clear)
CLIPS>
; name of deftemplate
(deftemplate prospect
; name of field
(slot name
; default value of field name
(default ?DERIVE))
; name of field
(slot assets
; default value of field assets
(default rich))
; name of field
(slot age
; default value of field age
(default 80)))
CLIPS>
(defrule matrimonial_candidate
(prospect (name ?name)
(assets ?net_worth)
(age ?months))
=>
(printout t "Prospect: "
?name crlf
?net_worth crlf
?months " months old"
crlf))
CLIPS>
(assert
(prospect (name "Dopey Wonderful")
(age 99)))
<Fact-1>
CLIPS> (run)
Prospect: Dopey Wonderful
rich
99 months old
CLIPS>
```

Fíjate en que se ha utilizado el valor por defecto de rich para Dopey, ya que no se ha especificado el campo assets en el comando assert.

Si el campo assets tiene un valor específico como *poor*, el valor especificado para los *assets* de *poor* anula el valor por defecto de *rich* como se muestra en el siguiente ejemplo sobre el sobrino pobre de Dopey.

```
CLIPS> (reset)
CLIPS>
(assert
(prospect (name "Dopey Notwonderful")
(assets poor)
(age 95)))
<Fact-1>
CLIPS> (run)
Prospect: "Dopey Notwonderful"
poor
95 months old
CLIPS>
```

Un patrón deftemplate puede utilizarse como cualquier patrón ordinario. Por ejemplo, la siguiente regla eliminará los prospect no deseados.

```
CLIPS> (undefrule matrimonial_candidate)
CLIPS>
(defrule bye-bye
?bad-prospect <-
(prospect (assets poor)
(name ?name))
=>
(retract ?bad-prospect)
(printout t "bye-bye " ?name crlf))
CLIPS> (reset)
CLIPS>
(assert
 (prospect (name "Dopey Wonderful") (assets rich)))
<Fact-1>
CLIPS>
(assert
(prospect (name "Dopey Notwonderful")
(assets poor)))
<Fact-2>
CLIPS> (run)
bye-bye Dopey Notwonderful
CLIPS>
```

Sin cadenas para mí

Fíjate que en los ejemplos anteriores sólo se han utilizado campos individuales. Es decir, los valores de los campos *name*, *assets*, y *age* eran valores únicos. En algunos tipos de reglas, es posible que quieras utilizar varios campos. Deftemplate permite el uso de múltiples valores en un *multislot*.

Como ejemplo de multislot, supongamos que quieres tratar el nombre de la relación *prospect* como campos múltiples. Esto proporcionaría más flexibilidad en el procesamiento de consultas, ya que cualquier parte del nombre podría coincidir con un patrón. A continuación se muestra la definición del modelo que utiliza la regla multislot y la regla revisada para la concordancia de patrones en varios campos. Fíjate que ahora se utiliza un patrón multislot, $?name, para que coincida con todos los campos que componen el nombre. Por conveniencia, también se proporciona una (deffacts).

```
CLIPS> (clear)
CLIPS>
(deftemplate prospect
(multislot name
(type SYMBOL)
(default ?DERIVE))
(slot assets
(type SYMBOL)
(allowed-symbols
poor rich wealthy loaded)
(default rich))
(slot age
(type INTEGER) ; The older
(range 80 ?VARIABLE) ; the
 (default 80))) ; better!!!
CLIPS>
(defrule happy_relationship
(prospect (name $?name)
(assets ?net_worth)
(age ?months))
=>
(printout t "Prospect: "
; Note: not $?name
?name crlf
?net_worth crlf
?months " months old"
crlf))
```

```
CLIPS>
(deffacts duck-bachelor
(prospect (name Dopey Wonderful)
(assets rich)
(age 99)))
CLIPS> (reset)
CLIPS> (run)
Prospect: (Dopey Wonderful)
rich
99 months old
CLIPS>
```

En la salida, CLIPS pone los paréntesis alrededor del nombre de Dopey para indicar que se trata de un valor multislot. Si comparas la salida de esta versión multislot con la versión single-slot, verás que las comillas dobles alrededor de "Dopey Wonderful" han desaparecido. La *ranura* del nombre no es una cadena en la versión multislot, por lo que CLIPS trata name como dos campos independientes, *Dopey* y Wonderful.

Qué hay detrás de un nombre

Deftemplate simplifica enormemente el acceso a un campo específico en un patrón porque el campo requerido puede identificarse por su nombre de ranura. La acción **modify** puede utilizarse para modificar un hecho en una sola acción especificando una o más ranuras de plantilla a modificar.

Como ejemplo, consideremos las siguientes reglas que muestran lo que ocurre cuando el pato soltero Dopey Wonderful pierde todo su pescado comprando pósters del Pato Donald y fishsplits de plátano para su nueva patita Dixie.

```
CLIPS> (undefrule *)
CLIPS>
(defrule make-bad-buys
?prospect <- (prospect (name $?name)
(assets rich)
(age ?months))
=>
(printout t "Prospect: "
; Note: not $?name
?name crlf
"rich" crlf
?months " months old"
crlf crlf)
```

```
(modify ?prospect (assets poor)))
CLIPS>
(defrule poor-prospect
?prospect <- (prospect (name $?name)
(assets poor)
(age ?months))
=>
(printout t "Ex-prospect: "
; Note: not $?name
?name crlf
poor crlf
?months " months old"
crlf crlf))
CLIPS>
(deffacts duck-bachelor
(prospect (name Dopey Wonderful)
(assets rich)
(age 99)))
CLIPS> (reset)
CLIPS> (run)
Prospect: (Dopey Wonderful)
rich
99 months old
Ex-prospect: (Dopey Wonderful)
poor
99 months old
CLIPS>
```

Si ejecutas (facts) como se indica a continuación, verás que el hecho f-1 que originalmente correspondía a (prospect (assets rich) (age 99) (name Dopey Wonderful)) se ha modificado y la ranura assets se ha establecido en poor.

```
CLIPS> (facts)
f-1 (prospect (name Dopey Wonderful)
(assets poor) (age 99))
For a total of 1 fact.
CLIPS>
```

La regla make-bad-buys se activa por un rich prospect especificado por la ranura assets. Esta regla cambia los assets a *poor* utilizando la acción modify. Fíjate que se puede acceder a la ranura *assets* por su nombre. Sin una deftemplate, sería necesario enumerar todos los

campos con variables individuales o utilizar un comodín, lo que es menos eficiente. El propósito de la regla poor-prospect es simplemente imprimir los poor prospects, demostrando así que la regla make-bad-investments efectivamente modificó los assets.

Capítulo 6

Ser funcional

La funcionalidad es lo contrario del estilo

En este capítulo, aprenderás funciones más eficaces para emparejar patrones y otras que resultan muy prácticas con variables multicampo. También aprenderás cómo se hacen cálculos numéricos.

No es mi obligación

Reconsideremos el problema de diseñar un sistema experto para ayudar a un robot a cruzar una calle. Una regla que habría que seguir es.

```
(defrule green-light
    (light green)
    =>
    (printout t "Walk" crlf))
```

Otra regla cubriría el caso de un semáforo en rojo.

```
(defrule red-light
    (light red)
    =>
    (printout t "Don't walk" crlf))
```

Una tercera regla cubriría el caso en el que una señal de paso dijera que no se camine. Esto tendría prioridad sobre una luz verde.

```
(defrule walk-sign
    (walk-sign-says dont-walk)
    =>
    (printout t "Don't walk" crlf))
```

Las reglas anteriores están simplificadas y no cubren todos los casos, como la avería del semáforo. Por ejemplo, ¿qué hace el robot si el semáforo está en rojo o amarillo y walk-sign indica walk?

Una forma de tratar este caso es utilizar una **restricción de campo** para limitar los valores que puede tener un patrón en el LHS. La restricción de campo actúa como una restricción sobre patrones.

Un tipo de restricción de campo se denomina restricción **conectiva**. Existen tres tipos de restricciones conectivas. La primera se denomina restricción **~**. Su símbolo es la **tilde**, "**~**". La restricción ~ actúa sobre el valor que le sigue inmediatamente y *no* permitirá ese valor.

Como ejemplo sencillo de la restricción ~, supongamos que quieres escribir una regla que imprima "No camine" si el semáforo no está en verde. Un enfoque sería escribir reglas para cada posible condición de la luz, incluyendo todos los posibles fallos de funcionamiento: amarillo, rojo, amarillo parpadeante, rojo parpadeante, verde parpadeante, amarillo y rojo intermitente, y así sucesivamente. Sin embargo, un enfoque mucho más simple es utilizar la restricción ~ como se muestra en la siguiente regla:

```
(defrule walk
      (light ~green)
      =>
      (printout t "Don't walk" crlf))
```

Al utilizar la restricción ~, esta regla hace el trabajo de muchas otras que requerían especificar cada condición de luz.

Sé prudente

La segunda restricción conectiva es la **restricción de barra**, "|". La restricción conectiva "|" se utiliza para permitir que coincida cualquiera de un grupo de valores.

Por ejemplo, supongamos que quieres una regla que imprima "Be cautious" si la luz es amarilla o amarilla intermitente. El siguiente ejemplo muestra cómo se hace utilizando la restricción "|".

```
CLIPS> (clear)
CLIPS>
(defrule cautious
(light yellow|blinking-yellow)
=>
(printout t "Be cautious" crlf))
CLIPS> (assert (light yellow))
<Fact-1>
CLIPS> (assert (light blinking-yellow))
<Fact-2>
CLIPS> (agenda)
```

```
0 cautious: f-2
0 cautious: f-1
For a total of 2 activations.
CLIPS>
```

Y allá vamos

El tercer tipo de restricción conectiva es la **restricción conectiva** &. El símbolo de la restricción conectiva & es el **ampersand**, "**& "**. La restricción & obliga a que las restricciones conectadas coincidan en la unión, como verás en los ejemplos siguientes. Normalmente, la restricción & sólo se utiliza con las demás restricciones; de lo contrario, no tiene mucha utilidad práctica. Como ejemplo, supongamos que quieres tener una regla que se active por un hecho amarillo o amarillo parpadeante. Esto es bastante fácil - sólo tienes que utilizar el conector de restricción | como hiciste en un ejemplo anterior. Pero supongamos que también quieres identificar el color de la luz.

La solución es vincular una variable que se empareja al color utilizando "&" y luego imprimirla. Aquí es donde el "&" es útil, como se muestra a continuación.

```
(defrule cautious
        (light ?color&yellow|blinking-yellow)
        =>
        (printout t
        "Be cautious because light is " ?color crlf))
```

La variable ?color se vinculará a cualquier color que coincida con el campo yellow|blinking-yellow.

El "&" también es útil con el "~". Por ejemplo, supongamos que quieres una regla que se active cuando la luz no sea amarilla ni roja.

```
(defrule not-yellow-red
        (light ?color&~red&~yellow)
        =>
        (printout t "Go, since light is " ?color crlf))
```

Es elemental

Además de tratar con hechos simbólicos, CLIPS también puede realizar cálculos numéricos. Sin embargo, debes tener en cuenta que un lenguaje de sistema experto como CLIPS no está diseñado principalmente para realizar cálculos numéricos. Aunque las funciones matemáticas de CLIPS son muy potentes, en realidad están pensadas para modificar los números sobre los que razona el programa de aplicación.

Otros lenguajes, como FORTRAN, son mejores para el cálculo numérico en el que se realiza poco o ningún razonamiento simbólico. Encontrarás la capacidad computacional de CLIPS útil en muchas aplicaciones.

CLIPS proporciona funciones aritméticas y matemáticas básicas +, /, *, -, div, max, min, abs, float e integer. Para más detalles, consulta el *Manual de Referencia de CLIPS*.

Las expresiones numéricas se representan en CLIPS según el estilo de LISP. Tanto en LISP como en CLIPS, una expresión numérica que habitualmente se escribiría como 2 + 3 debe escribirse en **forma prefija**, (+ 2 3). En la forma prefija de CLIPS, la función precede a los argumentos, y los paréntesis deben rodear la expresión numérica. La forma habitual de escribir expresiones numéricas se denomina forma **infija** porque las funciones matemáticas se fijan entre los **argumentos**.

Las funciones pueden utilizarse en el LHS y en el RHS. Por ejemplo, a continuación se muestra cómo se utiliza la operación aritmética de adición en el lado derecho de una regla para afirmar un hecho que contiene la suma de dos números ?x e ?y. Fíjate que los comentarios están en notación infija sólo para tu información ya que la infija no puede ser evaluada por CLIPS.

```
CLIPS> (clear)
CLIPS>
(defrule addition
(numbers ?x ?y)
=>
; Add ?x + ?y
(assert (answer-plus (+ ?x ?y))))
CLIPS> (assert (numbers 2 3))
<Fact-1>
CLIPS> (run)
CLIPS> (facts)
f-1 (numbers 2 3)
f-2 (answer-plus 5)
For a total of 2 facts.
CLIPS>
```

Se puede utilizar una función en el LHS si se utiliza un **signo igual**, **=**, para indicar a CLIPS que evalúe la siguiente expresión en lugar de utilizarla literalmente para la comparación de patrones. El siguiente ejemplo muestra cómo se calcula la hipotenusa en el LHS y se utiliza para comparar patrones con algunos artículos de stock. La función de **exponenciación**, "**", se utiliza para elevar al cuadrado los valores x e y. El primer argumento de la función de exponenciación es el número que se eleva a la potencia del segundo argumento.

```
CLIPS> (clear)
```

```
CLIPS>
(deffacts database
(stock A 2.0)
(stock B 5.0)
(stock C 7.0))
CLIPS>
(defrule addition
(numbers ?x ?y)
; Hypotenuse
(stock ?ID =(sqrt (+ (** ?x 2)
(** ?y 2))))
=>
(printout t "Stock ID=" ?ID crlf))
CLIPS> (reset)
CLIPS> (assert (numbers 3 4))
<Fact-4>
; Stock ID matches
; hypotenuse calculated
CLIPS> (run)
Stock ID=B
CLIPS>
```

Amplios argumentos

Los argumentos de una expresión numérica pueden ampliarse a más de dos para muchas de las funciones matemáticas.

La misma secuencia de cálculos aritméticos se realiza para más de dos argumentos. El siguiente ejemplo ilustra cómo se utilizan tres argumentos. La evaluación se realiza de izquierda a derecha. Sin embargo, antes de introducirlos, es posible que se quiera hacer un (clear) para deshacerse de cualquier dato y regla antiguos.

```
(defrule addition
        (numbers ?x ?y ?z)
        =>
        ; ?x + ?y + ?z
        (assert (answer-plus (+ ?x ?y ?z))))
```

Introduce el programa anterior y afirma (numbers 2 3 4). Después de ejecutarlo, verás los siguientes hechos. Ten en cuenta que los índices de los hechos pueden ser diferentes si has hecho un (reset) en lugar de un (clear) antes de cargar este programa.

```
CLIPS> (facts)
f-1 (numbers 2 3 4)
f-2 (answer-plus 9)
For a total of 2 facts.
CLIPS>
```

El equivalente infijo de una expresión CLIPS con múltiples argumentos puede expresarse como arg [function arg] donde los corchetes significan que puede haber múltiples términos.

Además de las funciones matemáticas básicas, CLIPS tiene **funciones matemáticas extendidas** que incluyen trigonométricas, hiperbólicas, etc. Para obtener una lista completa, consulta el *Manual de referencia de CLIPS.* Se llaman funciones matemáticas *extendidas* porque no se consideran funciones matemáticas básicas como "+", "-", etc.

Resultados variados

Al tratar con expresiones, CLIPS intenta mantener el mismo tipo que los argumentos. Por ejemplo,

```
; both integer arguments
; give integer result
CLIPS> (+ 2 2)
4
; both floating-point arguments
; give floating-point result
CLIPS> (+ 2.0 2.0)
4.0
; mixed arguments
; give floating-point result
CLIPS> (+ 2 2.0)
4.0
CLIPS>
```

Fíjate que, en el último caso de argumentos combinados, CLIPS convierte el resultado al tipo estándar de coma flotante de doble precisión.

Puedes convertir explícitamente un tipo a otro usando las funciones float e integer, como se demuestra en los siguientes ejemplos.

```
; convert integer
; to float
CLIPS> (float (+ 2 2))
```

```
4.0
; convert float
; to integer
CLIPS> (integer (+ 2.0 2.0))
4
CLIPS>
```

Los paréntesis se utilizan para especificar explícitamente, si se quiere, el orden de evaluación de la expresión. En el ejemplo de ?x + ?y * ?z, la forma infija habitual de evaluarla es multiplicar ?y por ?z y luego sumar el resultado a ?x. Sin embargo, en CLIPS, debe escribir la precedencia explícitamente si quieres este orden de evaluación, tal como se indica a continuación.

```
(defrule mixed-calc
    (numbers ?x ?y ?z)
    =>
    ; ?y * ?z + ?x
    (assert (answer (+ ?x (* ?y ?z)))))
```

En esta regla, primero se evalúa la expresión entre paréntesis; así, ?y se multiplica por ?z. El resultado se suma a ?x.

Bachilleres

El análogo a asignar un valor a una variable en el LHS mediante la correspondencia de patrones es **vincular** un valor a una variable en el RHS mediante la **función bind**. Es práctico enlazar variables en el lado derecho si se van a utilizar los mismos valores repetidamente.

Como ejemplo sencillo en un cálculo matemático, primero ligemos la respuesta a una variable y luego imprimamos la **variable ligada.**

```
CLIPS> (clear)
CLIPS>
(defrule addition
(numbers ?x ?y)
=>
(assert (answer (+ ?x ?y)))
(bind ?answer (+ ?x ?y))
(printout t "answer is "
?answer crlf))
CLIPS> (assert (numbers 2 2))
```

```
<Fact-1>
CLIPS> (run)
answer is 4
CLIPS> (facts)
f-1 (numbers 2 2)
f-2 (answer 4)
For a total of 2 facts.
CLIPS>
```

La función (bind) también se puede utilizar en el RHS para vincular valores individuales o multicampo a una variable. La función (bind) se utiliza para asociar cero, uno o más valores a una variable *sin el operador "$"*. Recuerda que, en el LHS, sólo puedes crear un patrón multicampo utilizando el operador "$" en un campo, como "$? x". Sin embargo, el "$" es innecesario en el lado derecho porque los argumentos de (bind) le dicen explícitamente a CLIPS exactamente cuántos valores vincular. De hecho, el "$" es un apéndice inútil en el RHS.

La siguiente regla ilustra algunos enlaces de variables en el RHS. La **función** de **valor multicampo**, **create$**, se utiliza para crear un valor multicampo. Su sintaxis general es la siguiente.

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

donde se puede añadir cualquier número de argumentos para crear un valor multicampo. Este valor multicampo, o un valor de campo único, puede entonces vincularse a una variable como se muestra en las acciones RHS de la siguiente regla.

```
CLIPS> (clear)
CLIPS>
(defrule bind-values-demo
=>
(bind ?duck-bachelors
(create$ Dopey Dorky Dinky))
(bind ?happy-bachelor-mv
(create$ Dopey))
(bind ?none (create$))
(printout t
"duck-bachelors "
?duck-bachelors crlf
"duck-bachelors-no-() "
(implode$ ?duck-bachelors) crlf
"happy-bachelor-mv "
```

```
?happy-bachelor-mv crlf
"none " ?none crlf))
CLIPS> (reset)
CLIPS> (run)
duck-bachelors (Dopey Dorky Dinky)
duck-bachelors-no-() Dopey Dorky Dinky
happy-bachelor-mv (Dopey)
none ()
CLIPS>
```

Haz lo que quieras

Al igual que otros lenguajes, CLIPS te permite definir tus propias funciones con **deffunction**. La deffunction es conocida globalmente, lo que te ahorra el esfuerzo de introducir las mismas acciones una y otra vez.

Las deffunction también ayudan a la legibilidad. Puedes llamar a una deffunction como a cualquier otra función. Una deffunction también puede utilizarse como argumento de otra función. Un (printout) puede usarse en *cualquier* parte de una deffunction incluso si no es la última acción porque imprimir es un efecto colateral de llamar a la función (printout).

A continuación, se muestra la sintaxis general de una función.

```
(deffunction <function-name>
        [optional comment]
        ; argument list. Last one may
        ; be optional multicampo arg.
         (?arg1 ?arg2 ... ?argM [$?argN])
        ; action1 to action(K-1)
        ; do not return a value
        ; only last action returned
        (<action1>
        <action2>
        ...
        <action(K-1)>
        <actionK>)
```

Los *?arg* son **argumentos ficticios**, lo que significa que los nombres de los argumentos no entrarán en conflicto con los nombres de las variables de una regla si son iguales. El término argumento ficticio a veces se denomina **parámetro** en otros libros.

Aunque cada acción puede haber devuelto valores de llamadas a funciones dentro de la acción, éstos son bloqueados por la deffunction de no ser devueltos al usuario. La deffunction sólo devolverá el valor de la *última* acción, <acciónK>. Esta acción puede ser una función, una variable o una constante.

A continuación, se muestra un ejemplo de cómo se define una deffunction para calcular la hipotenusa y cómo se utiliza en una regla. Aunque los nombres de las variables en la regla sean los mismos que los de los argumentos ficticios, no hay conflicto. Por eso son *ficticios*, porque no significan nada.

```
CLIPS> (clear)
CLIPS>
(deffunction hypotenuse ; name
; dummy arguments
(?a ?b)
; action
(sqrt(+ (* ?a ?a) (* ?b ?b))))
CLIPS>
(defrule calculate-hypotenuse
(dimensions ?base ?height)
=>
(printout t "Hypotenuse="
(hypotenuse ?base ?height)
crlf))
CLIPS> (assert (dimensions 3 4))
<Fact-1>
CLIPS> (run)
Hypotenuse=5.0
CLIPS>
```

Las deffunction pueden utilizarse con valores multicampo, como ilustra el siguiente ejemplo.

```
CLIPS> (clear)
CLIPS>
(deffunction count ($?arg)
(length$ $?arg))
CLIPS> (count 1 2 3 a duck "quacks")
6
CLIPS>
```

Otras características

A continuación, se describen otras funciones útiles. Para más información, consulta el *Manual de referencia de CLIPS*.

round: Redondea hacia el entero más cercano. Si está exactamente entre dos enteros, redondea hacia el infinito negativo.
integer: Trunca la parte decimal de un número.
format: Da formato a la salida.
list-deffunctions: Lista todas las funciones.
ppdeffunction: "Pretty print" de una función.
undeffunction: Elimina una función si no se está ejecutando actualmente y no se hace referencia a ella en ningún otro lugar. Especificando "*" para <nombre> se borran todas.
length$: Número de campos, o el número de caracteres de una cadena o símbolo.
nth$: Campo especificado si existe, si no nil.
member$: Número del campo si existe literal o variable, en caso contrario FALSE.
subsetp: Devuelve TRUE si un valor multicampo es un subconjunto de otro valor multicampo, en caso contrario FALSE.
delete$: Dado un número de campo, borra su valor.
explode$: Devuelve cada elemento de cadena como parte de un nuevo valor multicampo.
subseq$: Devuelve un rango especificado de campos.
replace$: Sustituye un valor especificado.

Capítulo 7

Cómo tener el control

> *Cuando eres joven, el mundo te controla, cuando eres mayor, debes controlar el mundo*

Hasta este momento, has estado aprendiendo la sintaxis básica de CLIPS. Ahora verás cómo aplicar la sintaxis que has aprendido a programas más potentes y complejos. También aprenderás una nueva sintaxis para la entrada, y verás cómo comparar valores y generar bucles.

Empecemos a leer

Además de hacer correspondencias con un patrón, una regla puede obtener información de otra manera. CLIPS puede leer la información que escribes desde el teclado utilizando la función **read**.

El siguiente ejemplo muestra cómo se usa (read) para introducir datos. Fíjate que no se necesita (crlf) extra después de (read) para poner el cursor en una nueva línea. La función (read) reinicia automáticamente el cursor en una nueva línea.

```
CLIPS> (clear)
CLIPS>
(defrule read-input
=>
(printout t "Name a primary color"
crlf)
(assert (color (read))))
CLIPS>
(defrule check-input
?color <-
(color ?color-read&red|yellow|blue)
=>
(retract ?color)
(printout t "Correct" crlf))
CLIPS> (reset)
CLIPS> (agenda)
0 read-input: *
```

```
For a total of 1 activation.
CLIPS> (run)
Name a primary color
red
Correct
CLIPS> (reset)
CLIPS> (run)
Name a primary color
green
CLIPS> ; No "correct"
```

La regla está diseñada para utilizar la entrada de teclado en el RHS, por lo que es práctico activar la regla sin especificar ningún patrón en el LHS para que se active automáticamente cuando se produzca un (reset).

Cuando se muestra la activación para la regla read-input mediante el comando (agenda), se imprime un * en lugar de un identificador de hecho como f-1. El * se utiliza para indicar que el patrón se satisface, pero no por un hecho específico.

La función (read) no es una función de propósito general que leerá cualquier cosa que escribas en el teclado. Una limitación es que (read) sólo leerá un campo. Así que, si intentas leer que el color primario es el rojo sólo se leerá el primer campo "primario". Para leer toda la entrada, debes encerrar la entrada entre comillas dobles. Por supuesto, una vez que la entrada está entre comillas dobles, se convierte en un campo literal único.

Puedes acceder a las subcadenas "primary", "color", "is" y "red" con las funciones **str-explode** o **sub-string**.

La segunda limitación de (read) es que no puedes introducir paréntesis a menos que estén entre comillas dobles. Del mismo modo que no puedes afirmar un hecho que contenga paréntesis, no puedes leer paréntesis directamente excepto como literales.

La función **readline** se utiliza para leer múltiples valores hasta que terminan con un retorno de carro.

Esta función lee los datos como una cadena. Para aseverar los datos de (readline) se utiliza una función (assert-string) para afirmar que el hecho no es una cadena, del mismo modo que la entrada por (readline). A continuación se muestra un ejemplo de nivel superior de (assert-string).

```
CLIPS> (clear)
CLIPS>
```

```
(assert-string "(primary color is red)")
<Fact-1>
CLIPS> (facts)
f-1 (primary color is red)
For a total of 1 fact.
CLIPS>
```

Fíjate que el argumento de (assert-string) debe ser una cadena A continuación se muestra cómo afirmar un hecho de múltiples campos a partir de (readline).

```
CLIPS> (clear)
CLIPS>
(defrule test-readline
=>
(printout t "Enter input" crlf)
(bind ?string (readline))
(assert-string
(str-cat "(" ?string ")")))
CLIPS> (reset)
CLIPS> (run)
Enter input
primary color is red
CLIPS> (facts)
f-1 (primary color is red)
For a total of 1 fact.
CLIPS>
```

Dado que (assert-string) requiere paréntesis alrededor de la cadena que se va a afirmar, se utiliza la función (str-cat) para ponerlos alrededor de la cadena.

Tanto (read) como (readline) también pueden utilizarse para leer información de un fichero especificando su nombre lógico como argumento. Para más información, consulta el *Manual de referencia de CLIPS.*

Ser eficiente

CLIPS es un lenguaje basado en reglas que utiliza un algoritmo de correspondencia de patrones muy eficaz llamado **Algoritmo Rete**, ideado por Charles Forgy, de la Universidad Carnegie-Mellon, para su shell OPS.

El término *Rete* significa *red* en latín y describe la arquitectura de software del proceso de comparación de patrones.

Es muy difícil proporcionar reglas precisas que mejoren siempre la eficiencia de un programa que se ejecute bajo el Algoritmo Rete. Sin embargo, las siguientes directrices generales pueden ayudar a ello:

1. Coloca primero los patrones más específicos en una regla. Los patrones con variables no ligadas y comodines deben ir más abajo en la lista de patrones de reglas. Un hecho de control debe ponerse en primer lugar dentro de los patrones.
2. Los patrones con menos hechos correspondencias deben ir primero para minimizar las concordancias parciales.
3. Los patrones que se retractan y afirman con frecuencia, los **patrones volátiles**, deben colocarse al final de la lista de patrones.

Como puedes ver, estas directrices son potencialmente contradictorias. Un patrón no específico puede tener pocas correspondencias (véanse las directrices 1 y 2). ¿Dónde debe ir? La directriz general es minimizar los cambios de las correspondencias parciales de un ciclo del motor de inferencia al siguiente. Esto puede requerir mucho esfuerzo por parte del programador a la hora de supervisar las correspondencias parciales. Una solución alternativa es simplemente comprar un ordenador más rápido, o una placa aceleradora. Esto es cada vez más atractivo ya que el precio del hardware siempre baja mientras que el precio del trabajo humano siempre sube. Como CLIPS está diseñado para la portabilidad, cualquier código desarrollado en una máquina debería funcionar en otra.

Otras características

El **elemento condicional test** proporciona una forma muy eficaz de comparar números, variables y cadenas en el LHS. El comando (test) se utiliza como patrón en el LHS. Sólo se activará una regla si se cumple el (test) junto con otros patrones.

CLIPS proporciona muchas funciones predefinidas.

Las funciones lógicas son:

not: Booleana *not*.
and: Booleana *and*.
or: Booleana *or*.

Las funciones aritméticas son:

/: División
*****: Multiplicación
+: Adición

-: Sustracción

Las funciones de comparación son:

eq: Igual (cualquier tipo). Compara tipo y magnitud.
neq: No igual (cualquier tipo).
=: Igual (tipo numérico). Compara la magnitud.
<> : No igual (tipo numérico).
>=: Mayor o igual que.
> : Mayor que.
<=: Menor o igual que.
< : Menor que.

Todas las funciones de comparación excepto "eq" y "neq" darán un mensaje de error si se utilizan para comparar un número y un no-número. Si no se conoce de antemano el tipo, deben utilizarse las funciones "eq" y "neq". La función eq comprueba la misma magnitud *y* tipo de sus argumentos mientras que la función "=" sólo comprueba la magnitud de sus argumentos (numéricos) y no le importa si son enteros o de coma flotante.

Las **funciones lógicas** de CLIPS son **and**, **or** y **not**. Pueden utilizarse en expresiones como funciones booleanas. En CLIPS, verdadero y falso se representan mediante los símbolos TRUE y FALSE.

Ten en cuenta que en CLIPS *deben* utilizarse mayúsculas para los valores lógicos.

Además de todas las funciones predefinidas, puedes escribir **funciones externas** o **funciones definidas por el usuario** en C, Ada u otros lenguajes procedimentales y enlazarlas a CLIPS. Estas funciones externas se utilizan como cualquier función predefinida.

CLIPS también te da la capacidad de especificar explícitamente un **elemento condicional and**, un **elemento condicional or** y un **elemento condicional not** en el LHS. La ausencia de un hecho se especifica como un patrón en el LHS utilizando el elemento condicional "not".

La alteración de nuestra información para ajustarla a la realidad se denomina **mantenimiento de la verdad**. Es decir, intentamos mantener el estado de nuestra mente para que sólo contenga información verdadera, de modo que se minimicen los conflictos con el mundo real.

Mientras que las personas pueden hacer esto con bastante facilidad (la práctica hace al maestro), es difícil para los ordenadores porque normalmente no saben qué entidades patrón son **lógicamente dependientes** de otras entidades patrón. CLIPS dispone de una función de mantenimiento de la verdad que etiqueta internamente las entidades modelo que

dependen lógicamente de otras. Si estas otras entidades patrón se retraen, CLIPS retraerá automáticamente las entidades lógicamente dependientes. El **elemento condicional lógico** utiliza la palabra clave **logical** alrededor de un patrón para indicar que las entidades de patrón concordantes proporcionan **soporte lógico** a las aserciones del lado RHS.

Aunque el soporte lógico funciona para las afirmaciones, *no reafirma* los hechos retractados. La moraleja es que, si pierdes algo debido a una información errónea, no puedes recuperarlo (como perder dinero por los consejos de tu corredor de bolsa).

CLIPS tiene dos funciones para ayudar con el soporte lógico. La función **dependencies** enumera las concordancias parciales de las que recibe soporte lógico una entidad patrón, o ninguna si lo no hay. La segunda función lógica es **dependents**, que lista todas las entidades patrón que reciben soporte lógico de una entidad patrón.

Las restricciones conectivas utilizan "&", "|" o "~". Otro tipo de restricción de campo se denomina **restricción de predicado** y suele utilizarse para la correspondencia de patrones de campos más complejos. El propósito de una restricción de predicado es restringir un campo en función del resultado de una expresión booleana. Si la expresión booleana devuelve FALSE, la restricción no se cumple y falla la comparación de patrones. Encontrará que la restricción de predicado es muy útil con patrones numéricos.

Una **función de predicado** es aquella que devuelve un valor FALSE o no FALSE. Los dos **puntos** ":" seguidos de una función de predicado se denominan **restricción de predicado**. Los dos puntos ":" pueden ir precedidos de "&", "|" o "~" o pueden ir solos, como en el patrón (fact :(> 2 1)). Normalmente se utiliza en combinación con la restricción conectiva & en forma de **"&:"**. La siguiente lista contiene algunas de las funciones de predicado definidas por CLIPS:

(**evenp** <arg>): Comprueba si <arg> es un número par.
(**floatp** <arg>): Comprueba si <arg> es un múmero en coma flotante.
(**integerp** <arg>): Comprueba si <arg> es entero.
(**lexemep** <arg>): Comprueba si <arg> es un símbolo o cadena.
(**multicampop** <arg>): Comprueba si <arg> es un valor multicampo.
(**numberp** <arg>): Comprueba si <arg> es flotante o entero.
(**oddp** <arg>): Comprueba si <arg> es un número impar.
(**pointerp** <arg>): Comprueba si <arg> es una dirección externa.
(**stringp** <arg>): Comprueba si <arg> es una cadena.
(**symbolp** <arg>): Comprueba si <arg> es un símbolo.

A menudo hay casos en los que es útil disponer de valores conocidos globalmente en un sistema experto. Por ejemplo, es ineficiente tener que redefinir constantes universales como π.

CLIPS proporciona la construcción **defglobal** para que los valores puedan ser conocidos universalmente por todas las reglas.

Otro tipo de función útil son los números aleatorios. CLIPS tiene una función **random** que devuelve un valor entero "aleatorio". La función de números aleatorios de CLIPS en realidad devuelve números **pseudoaleatorios**, lo que significa que no son realmente aleatorios, sino que son generados por una fórmula matemática. Para la mayoría de los propósitos, los números pseudoaleatorios estarán bien. Ten en cuenta que la función aleatoria de CLIPS utiliza la función rand de la librería ANSI C, que puede no estar disponible en todos los ordenadores que no se adhieran a este estándar. Para más información sobre todos estos temas, consulta el *Manual de Referencia de CLIPS*.

Además de los hechos de control para controlar la ejecución de los programas, CLIPS proporciona una forma más directa de control mediante la asignación explícita de prioridad a las reglas. El principal problema asociado con el uso explícito de la prioridad cuando se está empezando a aprender CLIPS es la tendencia a sobreutilizarla y escribir programas secuenciales. Este uso excesivo va en contra del propósito de utilizar un lenguaje basado en reglas, que es proporcionar un medio natural para aquellas aplicaciones mejor representadas por reglas. Del mismo modo, los lenguajes procedimentales son mejores para aplicaciones orientadas fuertemente al control, mientras que los lenguajes orientados a objetos son mejores para representar dichos objetos. CLIPS dispone de palabras clave denominadas **declare salience**, que pueden utilizarse para establecer explícitamente la prioridad de las reglas.

La prioridad se establece utilizando un valor numérico que va desde el valor más pequeño de -10000 al más alto de 10000. Si una regla no tiene una prioridad explícitamente asignada por el programador, CLIPS asume una prioridad de cero. Fíjate que una prioridad cero está a medio camino entre los valores de prioridad mayor y menor. Una prioridad cero no significa que la regla no la tenga, sino que simplemente tiene un nivel de prioridad intermedio.

CLIPS proporciona algunas estructuras de programación procedimental que se pueden utilizar en la RHS.

Estas estructuras son el **while** y el **if then else** que también se encuentran en lenguajes modernos de alto nivel como Ada, C y Java.

Otra función útil con los bucles (while) es **break** que finaliza el bucle (while) que se está ejecutando en ese momento. La función **return** finaliza inmediatamente una deffunction, una función genérica, un método o un controlador de mensajes que se esté ejecutando en ese instante.

Cualquier función puede ser llamada desde el RHS, lo que contribuye en gran medida a la potencia de CLIPS.

Existen muchas otras funciones CLIPS que pueden devolver números, símbolos o cadenas.

Estas funciones pueden ser utilizadas por sus valores de retorno o por sus **efectos colaterales**. Un ejemplo de una función que sólo se utiliza por su efecto colateral es (printout). El valor devuelto por (printout) no tiene sentido.

La importancia de (printout) radica en su efecto colateral de salida. En general, las funciones pueden tener argumentos anidados si es apropiado para el efecto deseado.

Antes de poder acceder a un fichero para leerlo o escribirlo, debe abrirse mediante la función **open**. El número de ficheros que se pueden abrir a la vez depende del sistema operativo y del hardware. Cuando ya no necesites acceder a un fichero, debes cerrarlo con la función **close**. A menos que se cierre un fichero, no hay garantía de que se guarde la información escrita en él.

El **nombre lógico** de un archivo es la forma en que CLIPS lo identifica. El nombre lógico es un nombre global por el cual CLIPS conoce este archivo en todas las reglas. Aunque el nombre lógico puede ser idéntico al nombre del fichero, puede que quieras usar otro diferente. Otra ventaja de un nombre lógico es que se puede sustituir fácilmente por un nombre de archivo diferente sin hacer grandes cambios en el programa.

La función para leer datos de un fichero es la conocida (read) o (readline). Lo único nuevo que tienes que hacer es especificar como argumento de (read) o (readline) el nombre lógico del que leer.

Para leer (read) más de un campo, debes utilizar un bucle. Incluso con (readline), es necesario un bucle para leer varias líneas. Se puede escribir un bucle haciendo que una regla desencadene otra o con un bucle while.

El bucle no debe intentar leer más allá del final del fichero o el sistema operativo emitirá un mensaje de error. Para ayudar a prevenir esto, CLIPS devuelve un campo simbólico EOF si se intenta leer más allá del final del archivo (EOF).

La función de **evaluación**, **eval**, se utiliza para evaluar cualquier cadena o símbolo, excepto las construcciones de tipo "def " como defrule, deffacts, etc., igual que si se hubiesen introducido en el nivel superior. La función **build** se encarga de las construcciones de tipo "def ". La función (build) es el complemento de (eval). La función (build) evalúa una cadena o símbolo como si se hubiera introducido en el nivel superior y devuelve TRUE si el argumento es una construcción legal de tipo def como (defrule), (deffacts), etc.

Capítulo 8

Cuestiones de herencia

> *La forma más fácil de obtener riqueza es heredarla; la segunda mejor forma es hacerla con el trabajo de otros; casarse con la riqueza se parece demasiado al trabajo.*

Este capítulo es una visión general de la programación orientada a objetos en CLIPS. A diferencia de la programación basada en reglas, en la que puedes escribir directamente una regla sin preocuparte de qué más hay en el sistema, la programación orientada a objetos requiere un material de base esencial.

Cómo ser objetivo

Una característica clave de un buen diseño de programas es la flexibilidad. Por desgracia, la rígida metodología de las técnicas de programación estructurada no proporciona la flexibilidad necesaria para realizar cambios rápidos, fiables y eficaces. El paradigma de la **programación orientada a objetos (POO)** proporciona esta flexibilidad.

El término *paradigma* procede de la palabra griega *paradeigma*, que significa modelo, ejemplo o patrón. En informática, un *paradigma* es una metodología coherente y organizada para intentar resolver un problema. En la actualidad, existen muchos paradigmas de programación, como la programación orientada a objetos, la **procedimental**, la **basada en reglas** y la **conexionista**. El término **sistemas neuronales artificiales** es un sinónimo moderno del antiguo término conexionista.

La programación tradicional es procedimental porque hace hincapié en algoritmos o procedimientos para resolver problemas. Se han desarrollado muchos lenguajes para soportar este paradigma procedimental, como Pascal, C, Ada, FORTRAN o BASIC. Estos lenguajes también se han adaptado al **diseño orientado a objetos** (OOD), ya sea añadiendo extensiones o imponiendo una metodología de diseño a los programadores. En cambio, se han desarrollado nuevos lenguajes para ofrecer POO, que no es lo mismo que OOD. Se puede hacer OOD en cualquier lenguaje, incluso en lenguaje ensamblador.

CLIPS proporciona tres paradigmas: reglas, objetos y procedimientos. Aprenderás más sobre los objetos en el **Lenguaje Orientado a Objetos de CLIPS (COOL)** que está integrado con los paradigmas basados en reglas y procedimientos de CLIPS. CLIPS soporta el paradigma

procedimental a través de funciones genéricas, deffuntion y funciones externas definidas por el usuario. Dependiendo de la aplicación, se pueden utilizar reglas, objetos, procedimientos o una combinación.

En lugar de imponer un único paradigma al usuario, nuestra filosofía es que una variedad de herramientas especializadas, un enfoque **multiparadigma**, es mejor que intentar obligar a todo el mundo a utilizar una única herramienta de propósito general. A modo de analogía, aunque se podría utilizar un martillo y clavos para sujetarlo todo, hay casos en los que se prefieren otros elementos de fijación. Por ejemplo, imagina que te sujetas los pantalones con un martillo y clavos en lugar de con una cremallera (NOTA: si alguien utiliza un martillo y clavos en sus pantalones, por favor, póngase en contacto con el Libro Guiness de los Récords).

Cosas de clase

En programación orientada a objetos, una clase es una plantilla que describe las características o atributos comunes de los objetos. Nótese que este uso del término plantilla no es el mismo que *deftemplate* descrito en un capítulo anterior. Aquí, la palabra plantilla se utiliza en el sentido de una herramienta que se utiliza para construir objetos que tienen atributos comunes. Por analogía, una regla es una plantilla para dibujar líneas rectas, mientras que un cortador de galletas es una plantilla para dibujar curvas.

Las clases de objetos se organizan en una jerarquía o en un gráfico para describir las relaciones de los objetos en un sistema. Cada clase es una **abstracción** de un sistema del mundo real o de algún otro sistema lógico que intentamos modelar. Por ejemplo, un modelo abstracto de un sistema real podría ser un automóvil. Otro modelo abstracto de un sistema lógico podrían ser instrumentos financieros como acciones y bonos, o números complejos. El término *abstracción* se refiere a (1) la descripción abstracta de un objeto del mundo real o de otro sistema que intentamos modelar, o (2) el proceso de representar un sistema en términos de clases. La abstracción es una de las cinco características generalmente aceptadas de un verdadero lenguaje de programación orientada a objetos. Las otras son la **herencia**, la **encapsulación**, el **polimorfismo** y el **enlace dinámico**. Estos términos serán explicados en detalle a medida que leas este libro. CLIPS soporta estas cinco características.

El término *abstracto* significa que no nos ocupamos de los detalles no esenciales. Una descripción abstracta de un sistema complejo es una descripción simplificada que se concentra en la información relevante para un fin concreto. Así, el sistema se representa mediante un modelo más sencillo y fácil de entender. Como ejemplo conocido, cuando algunas personas conducen coches, utilizan un modelo abstracto de conducción que consta de dos elementos: el volante y el acelerador. Es decir, estas personas no se preocupan de los cientos de componentes que forman un automóvil, ni de la teoría de los motores de

combustión interna, las leyes de tráfico, etcétera. Saber sólo cómo utilizar el volante y el acelerador es su modelo abstracto de conducción.

Una de las cinco características fundamentales de la programación orientada a objetos es la herencia. Las clases se organizan en una jerarquía con las clases más generales en la parte superior y las clases más especializadas a continuación. Esto permite definir fácilmente nuevas clases como refinamientos especializados o modificaciones de clases existentes.

El uso de la herencia puede acelerar enormemente el desarrollo de software y aumentar la fiabilidad, ya que no es necesario crear nuevo software desde cero cada vez que se hace un nuevo programa. La programación orientada a objetos facilita la utilización de **código reutilizable**. Los programadores de POO suelen utilizar bibliotecas de objetos compuestas por cientos o miles de objetos. Estos objetos pueden utilizarse o modificarse como se quiera en un nuevo programa. Además de las bibliotecas de objetos de dominio público, varias empresas comercializan bibliotecas de objetos. Aunque el concepto de componentes de software reutilizables existe desde los primeros días de las bibliotecas de subrutinas FORTRAN en los años 60, nunca antes se había utilizado con tanto éxito para el desarrollo general de software.

Para definir una clase, es necesario especificar una o varias **clases padre** o **superclases** de la clase que se va a definir. Como analogía a las superclases, todas las personas tienen padres; las personas no surgen espontáneamente (aunque a veces puede preguntarse si ciertas personas realmente tenían padres). Lo contrario de una superclase es una **clase hija** o **subclase**.

Esto determina la herencia de la nueva clase. Una subclase hereda **atributos** de una o más superclases. El término atributo en COOL se refiere a las **propiedades** de un objeto, que son **ranuras** con nombre que lo describen. Por ejemplo, un objeto que represente a una persona puede tener ranuras para nombre, edad, dirección, etc.

Una **instancia** es un objeto que tiene *valores* para las ranuras, como John Smith, 28, 1000 Main St., Clear Lake City, TX. Las clases de nivel inferior heredan automáticamente sus ranuras de las clases de nivel superior, a menos que las ranuras se bloqueen explícitamente. Se definen nuevas ranuras además de las heredadas para establecer todos los atributos que describen la clase.

El **comportamiento** de un objeto viene definido por sus **controladores de mensajes**, o **controladores** para simplificar. Un controlador de mensajes para un objeto responde a los mensajes y realiza las acciones requeridas. Por ejemplo, enviar el mensaje

```
(send [John_Smith] print)
```

haría que el controlador de mensajes apropiado imprimiera los valores de las ranuras de la instancia John_Smith. Las instancias se especifican generalmente entre **corchetes**, **[ ]**. Un mensaje comienza con la función **send**, seguida del nombre de la instancia, el nombre del mensaje y los argumentos necesarios. Por ejemplo, en el caso del mensaje print, no hay argumentos. Un **objeto** en CLIPS es una instancia de una clase.

La encapsulación de ranuras y controladores dentro de un objeto es otra de las cinco características generalmente aceptadas de la programación orientada a objetos. El término *encapsulado* significa que una clase se define en términos de sus ranuras y controladores. Aunque un objeto de una clase puede heredar ranuras y controladores de sus superclases, con algunas excepciones que se discutirán más adelante, *los valores de las ranuras del objeto no pueden ser alterados o examinados sin enviar un mensaje al objeto*.

La **clase raíz** o simplemente **raíz** de CLIPS es una **clase predefinida del sistema** llamada **OBJECT**.

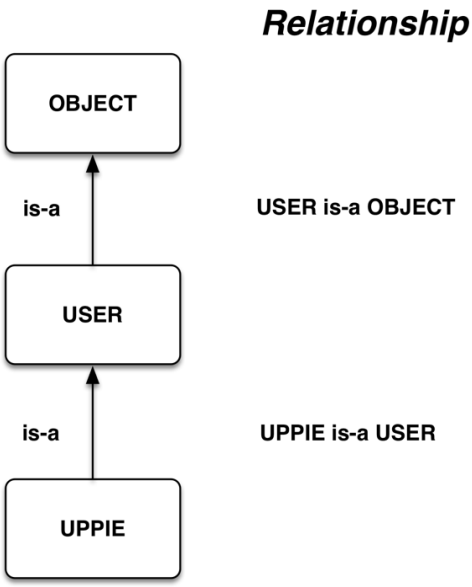
La clase predefinida **USER** es una subclase de OBJECT.

Cómo los UPPIES consiguen lo suyo

Como ejemplo, supongamos que queremos definir una clase llamada UPPIE, que es un término coloquial para un **profesional urbano**. Ten en cuenta que en este libro seguiremos la convención de escribir las clases en mayúsculas.

La Fig. 8.1 ilustra cómo los UPPIEs obtienen su herencia hasta llegar a la clase raíz OBJECT. Fíjate que UPPIE se define como una subclase de USER. Las cajas o **nodos** representan clases, mientras que las flechas de conexión se denominan **enlaces**. A menudo se utilizan líneas en lugar de flechas para simplificar el dibujo. Además, como CLIPS sólo soporta enlaces **is-a**, la relación "is-a" no se escribirá explícitamente junto a cada enlace a partir de ahora.

Fig. 8.1 La clase UPPIE



La convención que seguiremos para la **relación** entre clases es que el extremo de la flecha está en la subclase mientras que la cabeza apunta a la superclase. Las relaciones de la Fig. 8.1 siguen esta convención. Otra convención posible es utilizar flechas para señalar las subclases.

El enlace is-a indica la herencia de ranuras de una clase a su subclase. Una clase puede tener cero o más subclases. Todas las clases excepto OBJECT deben tener una superclase. Como UPPIE también hereda todos los slots de USER, y USER hereda todos los slots de OBJECT, se deduce que UPPIE también hereda todos los slots de OBJECT. El mismo principio de herencia se aplica también a los controladores de mensajes de cada clase. Por ejemplo, UPPIE hereda todos los controladores de USER y OBJECT.

La herencia de ranuras y controladores es particularmente importante en la programación orientada a objetos, ya que significa que no tienes que redefinir las propiedades y el comportamiento de cada nueva clase de objetos que se declara. En su lugar, cada nueva clase hereda todas las propiedades y el comportamiento de sus clases de nivel superior. Dado que el nuevo comportamiento se hereda, puede reducir sustancialmente la **verificación y validación (V&V)** de los controladores. V&V significa esencialmente que el producto se ha construido correctamente y que cumple los requisitos. La tarea de verificar y validar el software puede llevar más tiempo y dinero que el propio desarrollo del software, especialmente si el software afecta a la vida humana y a la propiedad. La herencia de controladores permite una reutilización eficaz del código existente y acelera el desarrollo.

Las clases se definen en CLIPS utilizando la construcción defclass. La clase UPPIE se define en una sentencia como la siguientes.

```
(defclass UPPIE (is-a USER))
```

Fíjate en la similitud entre la relación UPPIE-USER de la Fig. 8.1 y la construcción (defclass).

No es necesario introducir las clases USER u OBJECT ya que son clases predefinidas y por tanto CLIPS ya conoce su relación. De hecho, si intentas definir USER u OBJECT, aparecerá un mensaje de error ya que no puedes cambiar las clases predefinidas, a menos que cambies el código fuente de CLIPS.

Dado que CLIPS **distingue entre mayúsculas y minúsculas**, los comandos y funciones *deben* introducirse en minúsculas.

Las clases predefinidas del sistema como USER y OBJECT *deben* introducirse en mayúsculas. Aunque puedes introducir las clases definidas por el usuario en minúsculas o mayúsculas, seguiremos la convención de utilizar todas en mayúsculas para las clases, en aras de la legibilidad.

El formato básico del comando defclass para definir clases solamente, y no ranuras, es,

```
(defclass <class>
    (is-a <direct-superclasses>))
```

La lista de clases, <superclases-directas>, se denomina **lista de precedencia de superclases directas** porque define cómo se vincula una clase con sus superclases directas. Las superclases directas de una clase son una o más clases nombradas después de la palabra clave is-a. En nuestro ejemplo, la clase DUCKLING es la superclase directa de DUCK. Ten en cuenta que al menos una superclase directa debe figurar en la lista de precedencia de superclases directas.

Si la lista de superclases directas fuera la siguiente,

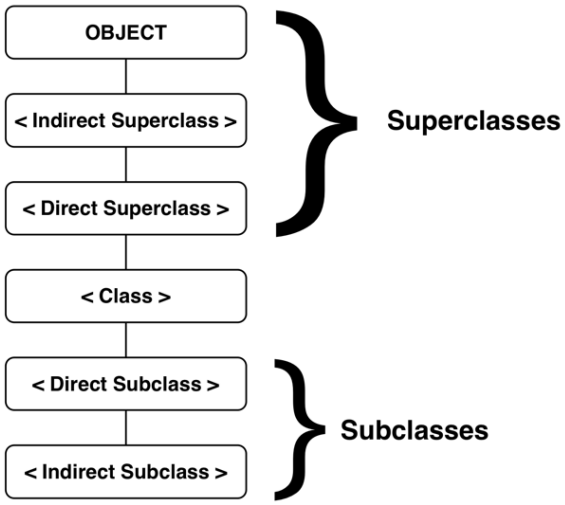
```
(defclass DUCK
    (is-a DUCKLING USER OBJECT))
```

entonces USER y OBJECT también serían superclases directas de DUCK. En este ejemplo, no hay diferencia si USER y OBJECT se especifican además de DUCKLING. De hecho, dado que USER y OBJECT son clases predefinidas que siempre están enlazadas de tal forma que USER is-a OBJECT, y OBJECT es la raíz, nunca es necesario especificarlas excepto cuando se define una subclase de USER. Dado que USER sólo hereda de OBJECT, no es necesario especificar OBJECT si se especifica USER.

Las **superclases indirectas** de una clase son todas las clases no nombradas después de "is-a" que aportan ranuras y controladores de mensajes por herencia. En nuestro ejemplo, las superclases indirectas son USER y OBJECT. Una clase hereda ranuras y controladores de mensajes de todas sus superclases directas e indirectas. Así, DUCK hereda de DUCKLING, USER y OBJECT.

Una **subclase directa** está conectada por un único enlace a la clase superior. Una subclase indirecta tiene más de un enlace. La Fig. 8.2 resume la terminología de las clases.

Fig. 8.2 Relaciones de clase



La clase raíz OBJECT es la *única* clase que no tiene una superclase.

El uso de esta nueva terminología nos permite enunciar el *Principio de Herencia de la programación orientada a objetos*: Una clase puede heredar de todas sus superclases.

Este es un concepto simple, pero poderoso, plenamente explotado en la programación orientada a objetos. Este principio significa que las ranuras y los controladores de mensajes pueden heredarse para ahorrarnos la molestia de redefinirlos para nuevas subclases. Además, las ranuras pueden personalizarse fácilmente para nuevas subclases como modificaciones y como compuestos de ranuras de superclases. Al permitir una reutilización fácil y flexible del código existente, se reducen el tiempo y el coste de desarrollo del programa. Además, la reutilización del código existente minimiza las tareas de verificación y validación. Todas estas ventajas facilitan el mantenimiento de las tareas de depuración, modificación y mejora del programa una vez que el código es liberado.

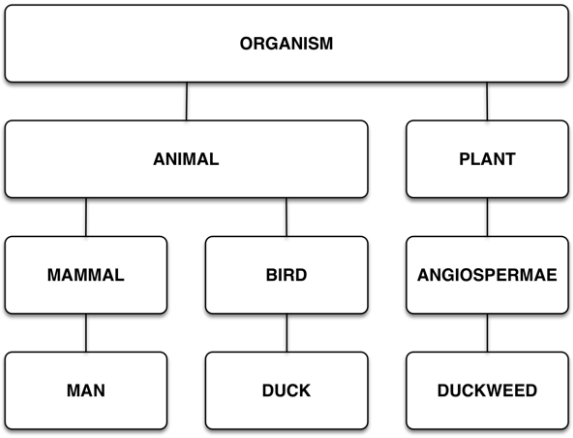
La razón de utilizar *may* en el principio es destacar que la herencia de ranuras de una clase puede bloquearse incluyendo una faceta no-inherit en la definición de ranura de la clase.

Las clases directas e indirectas de una clase son todas aquellas que se encuentran en una **ruta de herencia** hacia OBJECT. Una *ruta de herencia* es un conjunto de nodos conectados entre la clase y OBJECT. En nuestro ejemplo, el único camino de herencia de DUCK es DUCK, DUCKLING, USER, y OBJECT.

Verá ejemplos más adelante, como la Fig. 8.5, en la que una clase tiene **múltiples caminos** de herencia hacia OBJECT.

La Fig. 8.3 ilustra una **taxonomía** muy simplificada de organismos que ilustra la herencia en la Naturaleza. El término *taxonomía* significa clasificación. Las taxonomías biológicas están diseñadas para mostrar el parentesco de los organismos. Es decir, una taxonomía biológica enfatiza las similitudes entre organismos agrupándolos.

Fig. 8.3 Taxonomía simple de los organismos vivos con enlaces is-a



En una taxonomía como la de la Fig. 8.3, las líneas de conexión son enlaces is-a. Por ejemplo, un DUCK es un BIRD. Un BIRD es un ANIMAL. Un ANIMAL es un ORGANISM y así sucesivamente. Aunque la herencia genética de cada individuo es diferente, las características de MAN y de DUCK son las mismas para cada especie.

En la Fig. 8.3, observe que la clase más general, ORGANISM, está en la parte superior, mientras que las clases más especializadas están más abajo en la taxonomía. En terminología CLIPS, diríamos que cada subclase hereda las ranuras de sus clases padre. Por ejemplo, dado que los mamíferos son de sangre caliente y dan a luz a crías, con la excepción del ornitorrinco, la clase MAN hereda los atributos de la clase madre MAMMAL. La superclase directa de MAN es ANIMAL y la subclase directa de MAMMAL es MAN. La superclase indirecta de MAMMAL es ORGANISM.

Las otras clases, como BIRD, DUCK, etc., no tienen relación con MAMMAL porque no están en una **ruta de herencia** de la clase más general ORGANISM. Una ruta de herencia es cualquier

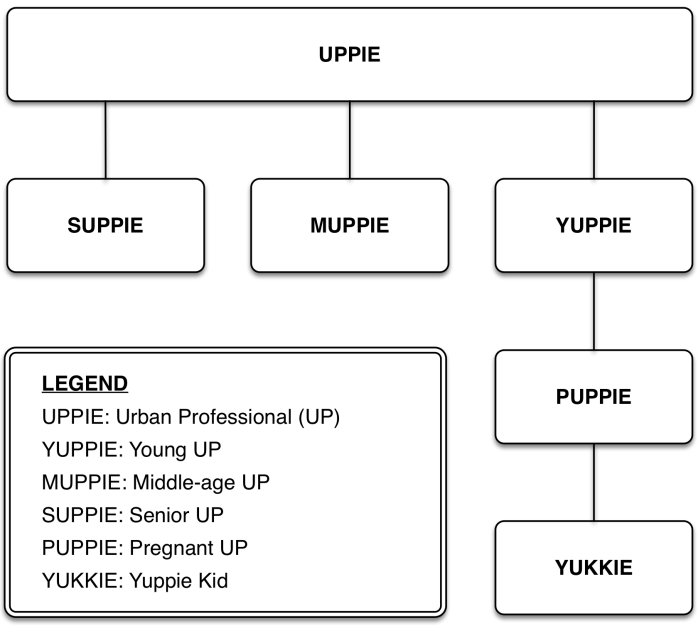
camino de una clase a otra que no implique **retroceder** o desandar el camino. Una clase como PLANT no está en una ruta de herencia hacia MAMMAL porque tendríamos que retroceder hasta ORGANISM antes de continuar hacia MAMMAL. Por lo tanto, MAMMAL no obtiene automáticamente ninguna ranura de PLANT ni de ninguna otra clase que no esté en la ruta de herencia hacia MAMMAL. Este modelo de herencia refleja el mundo real, ya que, de lo contrario, podríamos tener hierba creciendo en nuestras cabezas en lugar de pelo.

El YUKKIE ilegítimo

Ahora que ya tienes la idea básica de las clases, vamos a añadir algunas clases adicionales al diagrama UPPIE de la Fig. 8.1 para hacer el ejemplo más realista. Este tipo de desarrollo añadiendo clases de nivel inferior es la forma en que se hace la programación orientada a objetos, añadiendo clases desde las más generales a las más específicas.

La Fig. 8.4 muestra el diagrama de herencia del YUKKIE ilegítimo. Por simplicidad, no se muestran las clases OBJECT y USER. La jerarquía de la Fig. 8.4 es un **árbol** porque cada nodo tiene exactamente un padre.

Fig. 8.4 El YUKKIE ilegítimo



Un ejemplo familiar de estructura organizativa en árbol es la que suelen utilizar las empresas que tienen una **jerarquía** formada por presidente, vicepresidentes, jefes de departamento, gerentes y así hasta el empleado más bajo. En este caso, la estructura jerárquica refleja la autoridad de las personas en la organización. Los árboles se suelen utilizar para organizaciones de personas porque cada persona tiene exactamente un jefe, excepto el

mandamás, que no tiene jefe. Los nodos del organigrama representan los cargos, como presidente, vicepresidente, etc. Las líneas que conectan los cargos son las **ramas** que indican la división de responsabilidades. Los enlaces suelen denominarse ramas en un árbol.

En la Fig. 8.4, todas las clases excepto YUKKIE son legales o **legítimas**. Por ejemplo, un SUPPIE is-a UPPIE. Un MUPPIE is-a UPPIE. Un YUPPIE is-a UPPIE. También nos gustaría decir que un YUKKIE is-a YUPPIE y un YUKKIE is-a UPPIE por herencia. Sin embargo, no queremos decir que un YUKKIE is-a PUPPIE, que es lo que significa el vínculo is-a entre YUKKIE y PUPPIE.

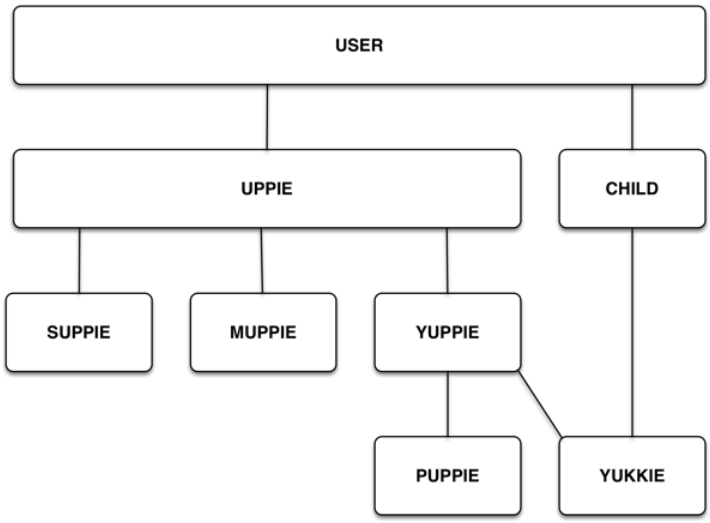
El vínculo is-a entre YUKKIE y PUPPIE es un error natural que puede cometer una persona, ya que un YUKKIE es hijo de una PUPPIE (en realidad, de una ex-PUPPIE después de dar a luz.) Aunque establecer un vínculo is-a entre YUKKIE y PUPPIE permite al YUKKIE heredar de YUPPIE y UPPIE como se quiera, también produce una relación ilegítima al decir que un YUKKIE is-a PUPPIE. Esto significa que un YUKKIE heredará todas las ranuras de un PUPPIE.

Suponiendo que una de las ranuras de PUPPIE especifica cuántos meses de embarazo tiene el PUPPIE, esto significa que cada bebé Yuppie también tendrá una ranura para indicar ¡cuántos meses de embarazo tiene!

Es posible corregir la figura. Sin embargo, tenemos que utilizar un grafo en lugar de un árbol. A diferencia de los árboles, en los que cada nodo excepto la raíz tiene exactamente un padre, cada nodo de un grafo puede tener cero o más nodos conectados a él. Un ejemplo familiar de grafo es un mapa de carreteras en el que las ciudades son nodos y las carreteras son los enlaces que las conectan. Otra diferencia entre árboles y grafos es que la mayoría de los tipos de árboles tienen una estructura jerárquica, mientras que los tipos generales de grafos no la tienen.

La Fig. 8.5 muestra la clase legítima YUKKIE. Se ha creado una nueva clase CHILD y se han establecido vínculos entre YUKKIE y sus dos superclases, YUPPIE y CHILD. Fíjate que ya no existe un vínculo ilegítimo entre YUKKIE y PUPPIE.

Fig. 8.5 El YUKKIE legítimo



Se trata de un grafo porque la clase YUKKIE tiene dos superclases directas en lugar de una sola como en un árbol. También es un grafo jerárquico porque las clases se ordenan mediante enlaces is-a desde la más general, USER, a las más específicas, SUPPIE, MUPPIE, PUPPIE y YUKKIE. Utilizando la Fig. 8.5, podemos decir que un YUKKIE is-a YUPPIE, y también que un YUKKIE is-a CHILD.

A continuación, se muestran los comandos para añadir las subclases mostradas en la Fig. 8.5.

```
CLIPS> (clear)
CLIPS> (defclass UPPIE (is-a USER))
CLIPS> (defclass CHILD (is-a USER))
CLIPS> (defclass SUPPIE (is-a UPPIE))
CLIPS> (defclass MUPPIE (is-a UPPIE))
CLIPS> (defclass YUPPIE (is-a UPPIE))
CLIPS> (defclass PUPPIE (is-a YUPPIE))
CLIPS> (defclass YUKKIE (is-a YUPPIE CHILD))
```

El orden de definición de las clases debe ser tal que una clase se defina antes que sus subclases. Así,

```
(defclass CHILD
      (is-a USER))
```

debe introducirse *antes* de

```
(defclass YUKKIE
      (is-a YUPPIE CHILD))
```

CLIPS emitirá un mensaje de error si intentas introducir la clase YUKKIE antes que la clase CHILD.

Observa el orden de izquierda a derecha en que se dibujan SUPPIE, MUPPIE y YUPPIE en la Fig. 8.5. Esto corresponde al orden en que estas clases se introducen en CLIPS y es la convención que seguiremos. También puedes ver por qué CHILD está dibujado a la derecha de UPPIE ya que fue introducido después de la clase UPPIE.

En la Fig. 8.5, fíjate que el enlace YUKKIE-YUPPIE está dibujado a la izquierda del enlace YUKKIE-CHILD. Otra convención que seguiremos es escribir las superclases directas de izquierda a derecha en la lista de precedencia según su orden de izquierda a derecha dibujado en un grafo. La ordenación YUPPIE-CHILD en la lista de precedencia de YUKKIE se hace para satisfacer esta convención.

Enséñame

CLIPS proporciona una serie de funciones para mostrar información sobre las clases, como funciones de predicado para comprobar si una clase es una superclase o subclase de otra.

La función **superclassp** devuelve TRUE si <class1> es una superclase de <class2>, y FALSE en caso contrario. La función **subclassp** devuelve TRUE si <class1> es una subclase de <class2> y FALSE en caso contrario. La forma general de ambas funciones es (función <class1> <class2>).

Por ejemplo,

```
CLIPS> (superclassp UPPIE YUPPIE)
TRUE
CLIPS> (superclassp YUPPIE UPPIE)
FALSE
CLIPS> (subclassp YUPPIE UPPIE)
TRUE
CLIPS> (subclassp UPPIE YUPPIE)
FALSE
CLIPS>
```

Ahora comprobemos si CLIPS ha aceptado todas estas nuevas clases. Una forma de hacerlo es con el comando **list-defclasses**. A continuación, se muestra la salida del comando.

```
CLIPS> (list-defclasses)
FLOAT
INTEGER
```

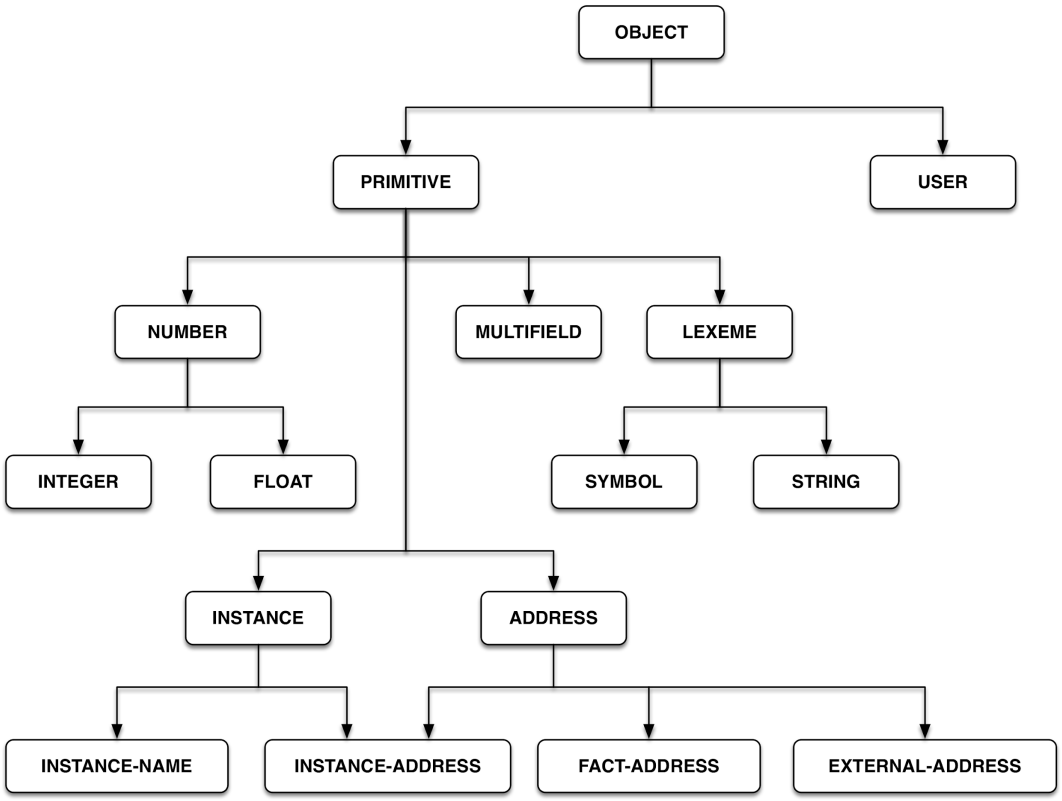
SYMBOL
STRING
MULTICAMPO
EXTERNAL-ADDRESS
FACT-ADDRESS
INSTANCE-ADDRESS
INSTANCE-NAME
OBJECT
PRIMITIVE
NUMBER
LEXEME
ADDRESS
INSTANCE
USER
UPPIE
CHILD
SUPPIE
MUPPIE
YUPPIE
PUPPIE
YUKKIE
For a total of 23 defclasses.
CLIPS>

Fíjate que el comando (list-defclasses) no indica la estructura jerárquica de las clases. Es decir, list-defclasses no indica qué clases son subclases o superclases de otras.

Si miras hacia abajo en esta lista, verás todas las clases definidas introducidas por el usuario: UPPIE, CHILD, YUPPIE, MUPPIE, SUPPIE, PUPPIE y YUKKIE. Además de las clases predefinidas del sistema OBJECT y USER que se han discutido hasta ahora, existen también otras clases predefinidas. Deberías reconocer la mayoría de ellas por tener el mismo nombre que los tipos familiares de CLIPS que aprendiste en capítulos anteriores. Los tipos predefinidos de CLIPS también están definidos como clases para que puedan ser utilizados con COOL.

El diagrama general de herencia de las clases predefinidas del *Manual de Referencia CLIPS*, se muestra en la Fig. 8.6, donde las flechas apuntan a las subclases.

Fig. 8.6 Las clases CLIPS predefinidas



La clase OBJECT es la raíz del árbol y está conectada por **ramas** a sus subclases. Los términos *rama*, **arista**, *enlace* y **arco** son básicamente sinónimos en el sentido de que todos indican una conexión entre nodos. Cada subclase está por debajo de su(s) superclase(s).

Cada subclase tiene mayor **especificidad** que su superclase. El término *especificidad* significa que una clase es más restrictiva. Por ejemplo, LEXEME es una superclase de SYMBOL y STRING. Si te dijeran que un objeto es de la clase LEXEME, sabrías que sólo puede ser un SYMBOL o una STRING. Sin embargo, si un objeto es un SYMBOL, no puede ser una STRING y viceversa. Por lo tanto, las clases SYMBOL y STRING son más específicas que LEXEME.

El comando **browse-classes** muestra la jerarquía de clases mediante indentación.

```
CLIPS> (browse-classes)
OBJECT
PRIMITIVE
NUMBER
INTEGER
FLOAT
LEXEME
SYMBOL
```

```
STRING
 MULTICAMPO
ADDRESS
EXTERNAL-ADDRESS
FACT-ADDRESS
INSTANCE-ADDRESS *
INSTANCE
INSTANCE-ADDRESS *
INSTANCE-NAME
USER
UPPIE
SUPPIE
MUPPIE
YUPPIE
PUPPIE
YUKKIE *
CHILD
YUKKIE *
CLIPS>
```

El asterisco tras el nombre de una clase indica que tiene varias superclases.

El comando (browse-classes) tiene un argumento opcional que especifica la clase inicial para las subclases que quieres ver. Esto es útil si no estás interesado en listar todas las clases.

Los siguientes ejemplos ilustran cómo mostrar porciones del grafo YUPPIE de la Fig. 8.5, llamados **subárboles** o **subgrafos** dependiendo de si los nodos y enlaces forman un árbol o un grafo.

```
CLIPS> (browse-classes UPPIE)
UPPIE
SUPPIE
MUPPIE
YUPPIE
PUPPIE
YUKKIE *
CLIPS> (browse-classes YUPPIE)
YUPPIE
PUPPIE
YUKKIE *
CLIPS> (browse-classes YUKKIE)
```

YUKKIE *
CLIPS>

Una **clase abstracta** está diseñada únicamente para la herencia. La clase abstracta USER no puede tener **instancias directas** definidas para ella, que son instancias definidas directamente para una clase. Además de la información sobre las clases, se describe la **precedencia de herencia** de las clases. Se trata de una **lista ordenada** en la que el orden de izquierda a derecha indica la precedencia de mayor a menor que aportan las clases por herencia. La precedencia de herencia enumera todas las superclases de una clase hasta la clase raíz OBJECT. También puedes ver que la información de superclases directas indica la superclase que está un eslabón por encima de una clase mientras que la lista de precedencia por herencia muestra *todas* las superclases.

Aunque una clase no tenga instancias directas, tendrá **instancias indirectas** si tiene subclases que tengan instancias. Las instancias indirectas de una clase son todas las instancias de sus subclases. Una **clase concreta** puede tener instancias directas. Por ejemplo, dada una clase concreta COW, la instancia directa Elsie sería la famosa vaca vendedora de televisión. Normalmente, las clases que heredan de clases abstractas son también clases abstractas. Sin embargo, las clases que heredan de clases del sistema, como USER, se consideran concretas a menos que se especifique lo contrario. Por tanto, UPPIE y YUPPIE también son clases concretas.

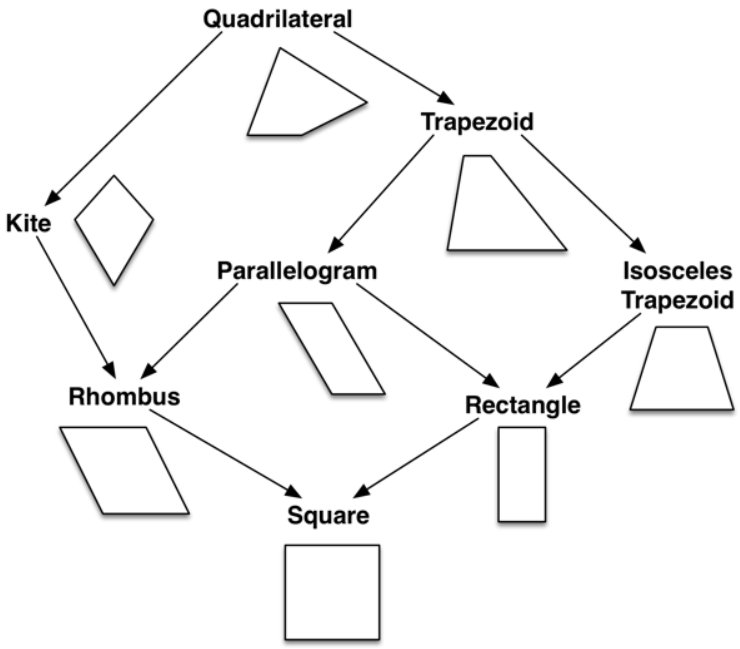
Se *recomienda encarecidamente* que *todas las clases que definas en CLIPS sean subclases de USER*. CLIPS proporcionará automáticamente controladores para print, init y delete si tus clases son subclases de USER.

La clase YUKKIE tiene **herencia múltiple** ya que tiene dos superclases directas, CHILD y YUPPIE. Si recordamos la analogía de la herencia con los jefes de una organización, la cuestión de la herencia múltiple plantea una pregunta interesante: ¿quién es el jefe? En el caso de una estructura de árbol, cada clase sólo tiene una superclase directa (boss), por lo que es fácil saber de quién recibir órdenes. Sin embargo, en el caso de la herencia múltiple, parece haber múltiples "jefes" de igual autoridad que son las superclases directas.

Para el caso de clases dispuestas en un árbol, es decir, herencia simple, la herencia es simplemente todas las clases a lo largo de la ruta de herencia de vuelta a OBJECT. El camino de herencia en un árbol es simplemente el camino más corto desde una clase hasta OBJECT. Este concepto de herencia también se aplica a un subgrafo de un grafo que es un árbol. Por ejemplo, la precedencia de herencia del subgrafo UPPIE, SUPPIE, MUPPIE, YUPPIE y PUPPIE es un árbol ya que cada uno de ellos tiene un único padre. Por lo tanto, la precedencia hereditaria de cada uno de ellos es la secuencia más corta de enlaces hacia OBJETO. Por ejemplo, la precedencia de herencia de PUPPIE es PUPPIE, YUPPIE, UPPIE, USER y OBJECT.

La Fig. 8.7 es otro ejemplo de grafo. Fíjate que algunos nodos, como el rombo, tienen más de un padre.

Fig. 8.7 El gráfico de cuadriláteros



Para simplificar, sólo discutiremos la herencia simple en este libro. Para más detalles sobre la herencia múltiple, consulta el *Manual de Referencia CLIPS*.

Otras características

Algunas otras funciones útiles con las clases se muestran a continuación.

ppdefclass: "Pretty-print" la estructura interna de declass.
undefclass: Eliminar la clase.
describe-class: Información adicional sobre las clases.
class-abstractp: La función de predicado devuelve TRUE si la clase es abstracta.

Para obtener más información sobre estas funciones y los temas mencionados en este capítulo, consulta el *Manual de referencia de CLIPS*.

Capítulo 9

Mensajes con sentido

> *Siempre es mejor complacer a los superiores que a los subordinados.*
>
> *- Bowen*

En este capítulo aprenderás más sobre las clases y los objetos llamados instancias. Verás cómo especificar los atributos de las clases utilizando ranuras y cómo enviar mensajes a los objetos.

Los pájaros y las abejas

En el Capítulo 1 aprendiste las ideas básicas de la herencia. La razón por la que la herencia es tan importante en la programación orientada a objetos es que la herencia permite la fácil construcción de **software personalizado**.

Por *software personalizado*, no queremos decir que el software se construya desde cero. Más bien, es como tomar un artículo producido en masa y modificarlo para una aplicación especial. El artículo producido en masa puede considerarse un producto de una **fábrica de software** que produce de forma rápida, económica y fiable artículos de un tipo general que están destinados a ser fácilmente personalizados.

El corazón del paradigma de la programación orientada a objetos es la creación de una jerarquía de clases para producir software de forma rápida, sencilla y fiable. Generalmente, este software es una modificación del software existente para que los programadores no estén siempre "reinventando el bucle".

Aunque en el pasado se ha intentado proporcionar código reutilizable a través de mecanismos como las bibliotecas de subrutinas, el paradigma **de la POO pura** en lenguajes como Smalltalk lleva el concepto de código reutilizable a su conclusión lógica al intentar construir *todo* el software de un sistema como código reutilizable. En Smalltalk, todo es un objeto, incluso las clases. En CLIPS, las instancias de los tipos primitivos como NUMBER, SYMBOL, STRING, etc., así como las instancias definidas por el usuario son objetos. Las clases *no* son objetos en CLIPS. Por ejemplo, el NUMBER 1, el SYMBOL Duck, la STRING " Duck ", y la instancia definida por el usuario Duck son todos objetos.

El paradigma de la POO es bastante diferente del enfoque de la biblioteca de subrutinas, en la que los fragmentos de código de subrutinas pueden o no ser utilizados dependiendo del capricho del programador. El paradigma de la programación orientada a objetos fomenta y apoya el código modular -los controladores de mensajes- que puede modificarse y

mantenerse fácilmente. Esta característica de la mantenibilidad del código desempeña un papel cada vez más importante a medida que aumentan el tamaño y el coste de los sistemas.

En la programación orientada a objetos, una clase es como una fábrica de software que contiene información sobre el diseño de un objeto. En otras palabras, una clase es como una plantilla que se puede utilizar para producir objetos idénticos que son las instancias de la clase. La analogía clásica es que una clase es como el plano de una vaca, y el objeto que produce leche, como Elsie, es la instancia.

La sintaxis general de un nombre de instancia es simplemente un símbolo rodeado de corchetes, [ ], como se indica a continuación.

[<name>]

En realidad, los corchetes *no* forman parte del nombre de instancia, que es un símbolo, como Elsie. Los corchetes se utilizan para rodear un nombre de instancia si existe peligro de ambigüedad en el uso del nombre. Esto puede ocurrir en la función (send), y es por eso por lo que se utilizan corchetes en (send). En caso de duda, utiliza corchetes, ya que no hace daño.

A continuación, se muestran algunos ejemplos de diferentes tipos de objetos en CLIPS:

SYMBOL
Dorky_Duck
STRING
"Dorky_Duck"
FLOAT
1.0
INTEGER
1
MULTICAMPO
(1 1.0 Dorky_Duck "Dorky_Duck")
DUCK
[Dorky_Duck]

Las clases **SYMBOL**, **STRING**, **FLOAT**, **INTEGER**, y **MULTIFIELD** tienen los mismos nombres con los que estás familiarizado de la programación basada en reglas en CLIPS. Se denominan tipos **primitivos** de objetos porque son proporcionados por CLIPS y mantenidos automáticamente según sea necesario. Estos tipos primitivos se proporcionan principalmente para su uso en **funciones genéricas**. Dos **clases compuestas** son **NUMBER** que es FLOAT o INTEGER, y **LEXEME** que es SYMBOL o STRING. Las clases compuestas se proporcionan por practicidad si el tipo número, o el tipo string no importa.

En cambio, los **tipos de objetos definidos** por el usuario son los que tú defines mediante clases definidas por el usuario. Si te refieres de nuevo a las clases predefinidas de CLIPS de

la Fig. 8.6 en el capítulo 8, verás que las clases primitivas y definidas por el usuario son la división de nivel superior de las clases en CLIPS.

Dos funciones convierten un símbolo en un nombre de instancia y *viceversa*. La función **symbol-to-instance-name** convierte un símbolo en un nombre de instancia, como se muestra a continuación.

```
; Get rid of any old classes
CLIPS> (clear)
CLIPS>
(symbol-to-instance-name Dorky_Duck)
[Dorky_Duck]
CLIPS>
(symbol-to-instance-name
(sym-cat Dorky "_" Duck))
[Dorky_Duck]
CLIPS>
```

Fíjate cómo las funciones estándar de CLIPS como (sym-cat), que concatena elementos, pueden utilizarse con el sistema de objetos de CLIPS.

La función opuesta, **instance-name-to-symbol**, convierte un nombre de instancia en un símbolo, como muestran los siguientes ejemplos.

```
CLIPS>
(instance-name-to-symbol [Dorky_Duck])
Dorky_Duck
CLIPS>
(str-cat
(instance-name-to-symbol [Dorky_Duck])
" is a DUCK")
"Dorky_Duck is a DUCK"
CLIPS>
```

Pato patoso

Existe una diferencia entre la Naturaleza y la programación orientada a objetos. En la Naturaleza, los objetos sólo se reproducen a partir de objetos similares, como los pájaros y las abejas (el huevo y la gallina son las excepciones a esta regla). Sin embargo, en la programación orientada a objetos las instancias sólo se crean utilizando la plantilla de clase. En una POO pura como Smalltalk, una instancia de una clase específica se crea enviando a la clase un mensaje. De hecho, el corazón de la programación orientada a objetos consiste en enviar diferentes tipos de mensajes de un objeto a otro, incluso de un objeto a sí mismo.

Para ver cómo funcionan los mensajes, empecemos escribiendo los siguientes comandos para crear una clase DUCK definida por el usuario y comprobar que está introducida. Observa que no se especifica ningún descriptor de **rol** para la clase DUCK. Si una clase tiene un **rol concreto**, se pueden crear instancias directas de la clase. Si el rol no está especificado, CLIPS determina el rol por herencia. Para determinar el rol por herencia, las clases del sistema se comportan como clases concretas. Así, por defecto, cualquier clase que herede de USER es una clase concreta y no necesita ser declarada como tal para permitir la creación de instancias directas.

Si una clase tiene **rol abstracto** no se pueden hacer instancias directas de ella. Las clases abstractas se definen únicamente a efectos de herencia. Por ejemplo, se podría definir una clase abstracta llamada PERSON cuyas propiedades como name, address, age, height, weight etc. son heredadas por las clases concretas MAN y WOMAN. Una instancia directa de MAN podría ser un man-person llamado Harold, y una instancia directa de WOMAN es una woman-person llamada Henrietta.

```
CLIPS> (defclass DUCK (is-a USER))
CLIPS> (describe-class DUCK)
==============================
*****************************************************
Concrete: direct instances of this class can be created.
Reactive: direct instances of this class can match defrule patterns.
Direct Superclasses: USER
Inheritance Precedence: DUCK USER OBJECT
Direct Subclasses:
-------------------------------------------------------------
Recognized message-handlers:
init primary in class USER
delete primary in class USER
create primary in class USER
print primary in class USER
direct-modify primary in class USER
message-modify primary in class USER
direct-duplicate primary in class USER
message-duplicate primary in class USER
*****************************************************
==============================
CLIPS>
```

Como las clases no son objetos en CLIPS, no podemos enviarles un mensaje para crear un objeto. En su lugar, se utiliza la función **make-instance** para crear un objeto instancia. La sintaxis básica es la siguiente.

(make-instance [<instance-name>] of <class> <slot-override>)

Normalmente, se especifica un nombre de instancia. Sin embargo, si no lo haces, CLIPS generará uno utilizando la función gensym*. También se pueden especificar valores de ranura.

Ahora que tenemos una fábrica de patos, hagamos algunas instancias como sigue, donde el nombre de la instancia está entre paréntesis. Observa el uso de la palabra clave "**of**" para separar el nombre de la instancia del nombre de la clase. Debes incluir "of " o se producirá un error de sintaxis. Además, ten en cuenta que los paréntesis en el código significan un nombre de instancia, mientras que los paréntesis en la metasintaxis como for (make-instance) arriba, significan *opcional*.

```
CLIPS> (make-instance [Dorky] of DUCK)
[Dorky]
CLIPS> (make-instance [Elsie] of COW)
[PRNTUTIL1] Unable to find class COW.
CLIPS> (make-instance Dorky_Duck of DUCK)
[Dorky_Duck]
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 2 instances.
CLIPS>
```

Una vez que la instancia se ha creado correctamente, CLIPS responde con el nombre de la instancia. Si *no* es posible crear una instancia, CLIPS responde con un FALSE. También, como los comandos (rules) y (facts), CLIPS tiene una función **instances** para imprimir las instancias de una clase.

Para el caso de (make-instance), los corchetes alrededor del nombre de la instancia son opcionales para una clase definida por el USUARIO. Como ejemplo, vamos a crear el primo de Dorky, Dorky_Duck, sin corchetes de la siguiente manera.

```
CLIPS> (make-instance Dorky_Duck of DUCK)
; Instance Dorky_Duck is made.
[Dorky_Duck]
CLIPS>
```

Hay que tener en cuenta dos reglas importantes sobre las instancias.

- En un módulo sólo puede utilizarse una instancia del mismo nombre.
- No se puede redefinir una clase si existen instancias de la misma.

Por ejemplo, vamos a hacer un clon de Dorky_Duck de la siguiente manera. (Es este clon malvado el que siempre mete a Dorky_Duck en problemas porque nadie puede distinguirlos. Los niños también tienen a menudo clones como este, y algunos adultos).

```
CLIPS> (make-instance Dorky_Duck of DUCK)
[Dorky_Duck]
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
; Still only one Dorky_Duck
For a total of 2 instances.
CLIPS>
```

Mucho lío con las instancias

Si se emite un comando (reset), se borran todas las instancias de la memoria.

```
CLIPS> (reset)
CLIPS> (instances)
CLIPS>
```

Al igual que (deffacts) define hechos, también hay una (**definstances**) para definir instancias cuando se emite un (reset). La siguiente (definstances) también pone el comentario opcional entre comillas dobles después del nombre de la instancia, DORKY_OBJECTS.

```
CLIPS>
(definstances DORKY_OBJECTS "The Dorky Cousins"
 (Dorky of DUCK)
(Dorky_Duck of DUCK))
CLIPS> (instances)
CLIPS> (reset)
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 2 instances.
CLIPS>
```

El pato desaparecido

Aunque un (reset) borrará todas las instancias, también creará nuevas instancias a partir de (definstances). Si quieres borrar permanentemente una instancia, la función **unmake-**

instance borrará una o todas las instancias, dependiendo de su argumento. Para borrar *todas* las instancias, utiliza el "*".

Los siguientes ejemplos ilustran el comando (unmake-instance).

```
; Delete all instances
CLIPS> (unmake-instance *)
TRUE
; Check that all are gone
CLIPS> (instances)
; Create new instances again
CLIPS> (reset)
; Check new instances created
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 2 instances.
; Delete a specific instance
CLIPS> (unmake-instance [Dorky])
TRUE
CLIPS> (instances)
[Dorky_Duck] of DUCK
For a total of 1 instance.
CLIPS>
```

Otra forma de eliminar una instancia específica es **send** un mensaje de **eliminación**. La sintaxis general de la función (send) es la siguiente.

```
(send <instance-name> <message>)
```

- *Sólo se puede especificar un nombre de instancia en un comando y debe ir entre corchetes si es un nombre definido por el usuario.*

Por ejemplo, lo siguiente hará que Dorky_Duck desaparezca.

```
; Create new instances again
CLIPS> (reset)
; Check new instances created
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 2 instances.
CLIPS> (send [Dorky_Duck] delete)
```

```
TRUE
CLIPS> (instances)
[Dorky] of DUCK
For a total of 1 instance.
CLIPS>
```

El "*" en un (send) *no* funcionará para borrar todas las instancias. El "*" sólo funciona con la función (unmake). Otra alternativa es definir tu propio controlador para eliminar, el cual aceptará el "*" y, por lo tanto, te permitirá enviar mensajes (send <nombre-de-instancia> my_delete *).

Un mensaje (send) sólo es ejecutado por un **objeto destino** que tenga un controlador apropiado.

CLIPS proporciona automáticamente controladores para print, init, delete, etc. para cada clase definida por el usuario. Es importante darse cuenta de que el mensaje (send [Dorky_Duck] delete) funciona sólo porque *esta instancia es una* clase definida por el usuario. Si defines clases que *no* heredan de USER, como una subclase de INTEGER, *debes* crear también los controladores apropiados para llevar a cabo todas las tareas deseadas, como imprimir, crear y borrar instancias. Es mucho más fácil definir subclases de USER y aprovechar los controladores proporcionados por el sistema.

¿Qué desayunaste?

La función (send) es el núcleo del funcionamiento de la programación orientada a objetos, ya que es la *única forma adecuada de que los objetos se comuniquen*. De acuerdo con el principio de encapsulación de objetos, un objeto sólo debe poder acceder a los datos de otro mediante el envío de un mensaje.

Por ejemplo, si alguien quiere saber qué has desayunado, generalmente te lo preguntará, es decir, enviará un mensaje. Una alternativa de mala educación sería abrirte la boca de un tirón y mirar por tu garganta. Si no se sigue el principio de encapsulación de objetos, cualquier objeto puede trastear con las partes privadas de otros objetos, con resultados potencialmente desastrosos.

Una aplicación útil de (send) es imprimir información sobre un objeto. Hasta ahora todos los ejemplos de objetos que has visto no tienen estructura. Sin embargo, al igual que deftemplate da estructura a un patrón de reglas, las ranuras dan estructura a un objeto. Tanto para deftemplate como para los objetos, una ranura es una ubicación con nombre en la que se pueden almacenar datos. Sin embargo, a diferencia de las ranuras deftemplate, los objetos obtienen sus ranuras de las clases, y las clases utilizan la herencia. Por lo tanto, la información de las ranuras de los objetos puede ser heredada por los objetos de las subclases. Una ranura **no vinculada** es aquella que no tiene valores asignados. Todas las ranuras *deben* estar vinculadas.

Como ejemplo sencillo, vamos a crear un objeto con ranuras para guardar información personal y enviarle mensajes. Los siguientes comandos configurarán primero el entorno CLIPS con las construcciones apropiadas. Las ranuras denominadas *sound* y *age* inicialmente no contienen datos, es decir, valores *nulos*.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
(slot sound)
(slot age))
CLIPS>
(definstances DORK_OBJECTS
(Dorky_Duck of DUCK))
CLIPS> (reset)
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound nil)
(age nil)
CLIPS>
(send [Dorky_Duck] put-sound quack)
quack
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(age nil)
CLIPS>
```

Fíjate que las ranuras se imprimen en el orden definido en la clase. Sin embargo, si la instancia hereda ranuras de más de una clase, las ranuras de las clases más generales se imprimirán primero.

El valor de una ranura se cambia utilizando mensaje put-. Por defecto, CLIPS crea un put-handler para cada ranura de una clase para manejar los mensaje put-. Fíjate en el guión, "-", al final de "put-". El guión es una parte esencial de la sintaxis del mensaje ya que separa el "put" del nombre de la ranura. Sólo se permite un "put-" en un (envío). Por lo tanto, para cambiar varias ranuras o la misma ranura de muchas instancias, debes enviar varios mensajes. En lugar de hacer esto manualmente, es posible escribir una función para hacer múltiples envíos, o utilizar la función **modify-instance**.

El valor de una ranura puede establecerse mediante un slot-override en un make-instance. A modo de ejemplo,

```
CLIPS>
(make-instance Dixie_Duck of DUCK
```

```
(sound quack) (age 2))
[Dixie_Duck]
CLIPS> (send [Dixie_Duck] print)
[Dixie_Duck] of DUCK
(sound quack)
(age 2)
CLIPS>
```

El mensaje complementario a "put-" es *get-* , que obtiene los datos de una ranura, como se muestra en el siguiente ejemplo. Si un *put-* tiene éxito devuelve el nuevo valor, mientras que si *get-* tiene éxito devuelve los datos apropiados. Si el *put-* o el *get-* no tienen éxito, se devolverá un mensaje de error. Los siguientes ejemplos muestran cómo funciona.

```
; No slot color
CLIPS>
(send [Dorky_Duck] put-color white)
[MSGFUN1] No applicable primary message-handlers found for put-color.
FALSE
CLIPS> (send [Dorky_Duck] get-age)
nil
; Value put in age
CLIPS> (send [Dorky_Duck] put-age 1)
1
; Check value is correct
CLIPS> (send [Dorky_Duck] get-age)
1
CLIPS>
```

A diferencia del mensaje *put-*, el mensaje *get- devuelve* el valor de una ranura. Puesto que se devuelve el valor de *get-*, puede ser utilizado por otra función, asignado a una variable, etcétera. Por el contrario, un valor que se imprime no puede ser utilizado por otra función, ni asignado a una variable, etc. porque el valor va al dispositivo de salida estándar. Una forma de evitar este problema es imprimir a un archivo y luego leer los datos del archivo. Aunque no es una solución elegante, funciona. Otra forma es escribir tu propio controlador de mensajes de impresión que también devuelva valores.

Un punto muy importante sobre las ranuras es que no puedes modificar las ranuras de una clase añadiendo ranuras, eliminando ranuras o cambiando sus características. La única forma de modificar una clase es (1) borrar todas las instancias de la clase, y (2) utilizar una (defclass) con el mismo nombre de clase y las ranuras requeridas. Esta situación es análoga a modificar una regla cargando una nueva regla con el mismo nombre.

Etiqueta de clase

Ahora que ya conoces las ranuras y las instancias, es hora de hablar de la **etiqueta clase**. El término *etiqueta* se refiere a un conjunto de pautas para hacer algo.

En contraste con la programación procedimental estándar, el paradigma de la programación orientada a objetos está **orientado a clases**.

Cada objeto está intrínsecamente relacionado con una clase, y esa clase forma parte de una jerarquía de clases. En lugar de concentrarse en las acciones en primer lugar, el desarrollador de programación orientada a objetos tiene en cuenta la jerarquía general de clases o la **arquitectura de clases**, y cómo se enviarán los mensajes entre los objetos. Así, las acciones en los programas procedimentales habituales se realizan explícitamente, mientras que las acciones se realizan implícitamente en la programación orientada a objetos. En cualquier caso, el resultado final es el mismo. Sin embargo, el sistema POO puede verificarse, validarse y mantenerse más fácilmente.

El uso apropiado de las clases se resume en las siguientes *Reglas de Etiqueta de las Clases:*

1. La jerarquía de clases debe estar en incrementos lógicos especializados utilizando enlaces is-a.
2. Una clase es innecesaria si sólo tiene una instancia.
3. Una clase no debe tener el nombre de una instancia y viceversa.

La primera regla desaconseja la creación de una única clase para su aplicación. Si una sola clase es suficiente, entonces probablemente no necesites la programación orientada a objetos. Al crear clases en incrementos, puedes verificar, validar y mantener tu código más fácilmente. Además, las jerarquías de clases incrementales pueden colocarse fácilmente en bibliotecas de clases para facilitar enormemente la creación de nuevo código. Este concepto de bibliotecas de clases es análogo a las bibliotecas de subrutinas para operaciones. Sólo se pueden utilizar enlaces is-a, ya que es la única relación que admite la versión 6.3 de CLIPS.

La segunda regla fomenta la idea de que las clases están pensadas como una plantilla para producir múltiples objetos del mismo tipo. Por supuesto, se puede empezar con cero o una instancia.

Sin embargo, si nunca vas a necesitar más de una instancia en una clase, deberías considerar modificar tu superclase para acomodar la instancia en lugar de definir una nueva subclase. Si todas tus clases sólo tienen una instancia, es probable que tu aplicación simplemente no esté bien adaptada a la programación orientada a objetos y que lo mejor sea codificar en un lenguaje procedimental.

La tercera regla significa que las clases no deben llamarse igual que las instancias y *viceversa*, para evitar confusiones.

Otras características

Hay una serie de funciones de predicado útiles para las ranuras. Si utilizas estas funciones de predicado para probar los valores apropiados para las funciones, tu programa será más robusto contra fallos. En general, si una función no devuelve TRUE, devuelve FALSE. Las funciones de ranura proporcionadas por CLIPS incluyen:

class-slot-exists: Devuelve TRUE si la ranura de clase existe.
slot-existp: Devuelve TRUE si la ranura de instancia existe.
slot-boundp: Devuelve TRUE si la ranura especificada tiene un valor.
instance-address: Devuelve la dirección de la máquina en la que se almacena la instancia especificada.
instance-name: Devuelve el nombre dado a una instancia.
instancep: Devuelve TRUE si su argumento es una instancia.
instance-addressp: Devuelve TRUE si su argumento es una dirección de instancia.
instance-namep: Devuelve TRUE si su argumento es un nombre de instancia.
instance-existp: Devuelve TRUE si la instancia existe.
list-definstances: Lista todas las definstancias.
ppdefinstances: "Pretty-print" de la definstance.
watch instances: Lista todas las instancias definidas. Permite ver la creación y eliminación de instancias.
unwatch instances: Desactiva la visualización de instancias.
save-instances: Guarda las instancias en un archivo.
load-instances: Carga instancias desde un archivo.
undefinstances: Elimina la definstancia nombrada.

Capítulo 10

Facetas fascinantes

Si quieres tener clase, actúa, viste y habla como tus amigos.

En este capítulo aprenderás más sobre las ranuras y cómo especificar sus características utilizando **facetas**. Al igual que las ranuras describen instancias, las facetas describen ranuras. El uso de facetas es una buena ingeniería de software porque hay más probablidades de que CLIPS señale un valor ilegal en lugar de arriesgarse a un error en tiempo de ejecución o a un fallo. Hay muchos tipos de facetas que se pueden usar para especificar ranuras, como se resume en la siguiente lista:

default and **default-dynamic**: Establece los valores iniciales de las ranuras.
cardinality: Número de valores multicampo.
access: Acceso de lectura-escritura, sólo lectura, sólo inicialización.
storage: Ranura local en instancia o ranura compartida en clase.
propagation: Ranuras heredadas o no heredadas.
source: Herencia compuesta o exclusiva.
override-message: Indica el mensaje a enviar para la anulación de ranura.
create-accessor: Crear controladores put y get.
visibility: Público o privado sólo para la clase definidora.
reactive: Los cambios en una ranura activan la comprobación de patrones.

Por razones de espacio, sólo describiremos algunas facetas con más detalle en el resto de este capítulo. Para más detalles, consulta el *Manual de referencia de CLIPS.*

Una ranura llamada Default

La **faceta por defecto** establece el valor por defecto de una ranura cuando se crea o inicializa una instancia, como se muestra en el siguiente ejemplo.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
 (slot sound (default quack))
(slot ID)
(slot sex (default male)))
CLIPS> (make-instance Dorky_Duck of DUCK)
[Dorky_Duck]
```

```
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID nil)
(sex male)
CLIPS>
```

Como se puede ver, los valores por defecto para las ranuras sex y sound fueron establecidos por los valores por defecto de la faceta.

Después de la palabra clave default puede ir cualquier expresión CLIPS válida que no implique una variable. Por ejemplo, la expresión por defecto de la ranura sound es el símbolo quack. Se pueden utilizar funciones en la **expresión de la faceta** como se mostrará en el siguiente ejemplo.

Esta faceta por defecto es una faceta **estática por defecto** porque el valor de su expresión de faceta se determina cuando se define la clase y nunca cambia a menos que se redefina la clase. Por ejemplo, establezcamos el valor por defecto del ID de ranura en la función (gensym*) que devuelve un nuevo valor que no está en el sistema cada vez que se llama.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
(slot sound (default quack))
(slot ID (default (gensym*)))
(slot sex (default male)))
CLIPS> ; Dorky_Duck #1
(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen1)
(sex male)
CLIPS> ; Dorky_Duck #2
(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen1)
(sex male)
CLIPS>
```

Como puedes ver, el ID es siempre gen1 ya que (gensym*) sólo se evalúa una vez, y no de nuevo cuando se crea la segunda instancia. Ten en cuenta que los valores de (gensym*) pueden ser diferentes en tu ordenador si ya has llamado a (gensym*), ya que se incrementa en uno cada vez que se llama, y *no* se restablece con un (clear). La función (gensym*) vuelve a su valor inicial sólo si reinicias CLIPS.

Ahora supongamos que queremos hacer un seguimiento de las diferentes instancias de Dorky_Duck que se han creado. En lugar de utilizar el valor por defecto estático, podemos utilizar la faceta llamada **dinámica por defecto** que evaluará su expresión de faceta cada vez que se cree una nueva instancia. Observa la diferencia entre el siguiente ejemplo y el anterior.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
(slot sound (default quack))
(slot ID (default-dynamic (gensym*)))
(slot sex (default male)))
CLIPS> ; Dorky_Duck #1
(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen2)
(sex male)
CLIPS> ; Dorky_Duck #2
(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen3) ; Note ID is different
(sex male) ; from Dorky_Duck #1
CLIPS>
```

En este ejemplo, que utiliza el valor por defecto dinámico, el ID de la segunda instancia, gen3, es diferente del de la primera instancia, gen2. En cambio, en el ejemplo anterior estático por defecto, los valores de ID eran los mismos, gen1, ya que el (gensym*) sólo se evaluaba una vez cuando se definía la clase y no para cada nueva instancia en el caso de dinámico por defecto.

Propiedades cardinales

La **cardinalidad** de una ranura hace referencia a uno de los dos tipos de campos que puede contener una ranura: (1) campo único, o (2) multicampo. El término *cardinalidad* se refiere a un recuento. Una ranura delimitada de un solo campo contiene un solo campo, mientras que una ranura delimitada multicampo puede contener cero o más campos. Las ranuras de campo único y multicampo contienen *un* valor cada una. Sin embargo, un valor multicampo puede contener varios campos. Por ejemplo, (a b c) es un único valor multicampo con tres campos. La cadena vacía "" es un valor de un solo campo, al igual que "a b c". Por el contrario, una ranura no vinculada no tiene ningún valor.

Como analogía a las variables de campo único y multicampo, piensa en una ranura como tu buzón de correos. A veces puedes encontrar un correo basura que no tiene sobre. En su lugar, sólo se ha pegado una etiqueta de dirección en un trozo de papel doblado dirigido a "Residente". Otras veces puedes encontrar un sobre con varios anuncios. La pieza única de correo basura sin sobre es como un valor de campo único, mientras que el sobre con múltiples anuncios es como el valor multicampo. Si el distribuidor de correo basura se equivoca y te envía un sobre sin nada dentro, esto corresponde a la variable multicampo vacía. (Ahora que lo pienso, si el sobre de correo basura está vacío, ¿realmente has recibido correo basura?).

Una **faceta múltiple** con la palabra clave **multislot**, se utiliza para almacenar un valor multicampo como se muestra en el siguiente ejemplo.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
(multislot sound (default quack quack)))
CLIPS> (make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack quack)
CLIPS>
```

Se puede acceder a un valor multicampo utilizando get- y put- como se muestra en los siguientes ejemplos, que ilustra cómo realizar un seguimiento de los quacks.

```
CLIPS>
(send [Dorky_Duck] put-sound
quack1 quack2 quack3)
(quack1 quack2 quack3)
CLIPS> (send [Dorky_Duck] get-sound)
(quack1 quack2 quack3)
```

```
CLIPS>
```

Se pueden utilizar funciones CLIPS estándar como nth$ para obtener el enésimo campo de un valor multislot. El siguiente ejemplo muestra cómo elegir un determinado cuack.

```
CLIPS> (nth$ 1 (send [Dorky_Duck] get-sound))
quack1
CLIPS> (nth$ 2 (send [Dorky_Duck] get-sound))
quack2
CLIPS> (nth$ 3 (send [Dorky_Duck] get-sound))
quack3
CLIPS>
```

Otras características

CLIPS dispone de varias **funciones de ranura multicampo**, como se muestra en la siguiente lista.

slot-replace$: Sustituye el rango especificado.
slot-insert$: Inserta el rango especificado.
slot-delete$: Borra el rango especificado.

Una ranura multicampo sin valores, por ejemplo, el valor multicampo vacío (), puede asignarse a una ranura con una faceta (multiple). Ten en cuenta que hay una diferencia entre una ranura con un valor multicampo vacío () y una ranura no vinculada. Si se piensa en un valor multicampo vacío como análogo a un autobús vacío, se puede ver que hay una diferencia entre ninguna persona (unbound slot) y un autobús sin personas (empty multifield value, ()).

Una faceta **create-accessor** indica a CLIPS si debe crear controladores **put-** y **get-** para una ranura. Por defecto, todas las ranuras tienen un accesorio de creación de **lectura-escritura**, por lo que no es necesario especificar esta faceta para crear controladores. Si defines tus propios controladores, entonces necesitas usar ?NONE con la faceta create-accessor. Los otros tipos de facetas para create *accessor* son **read** y **write.**

Una **faceta de almacenamiento** define uno de los dos lugares en los que se almacena un valor de ranura: (1) en una instancia, o (2) en la clase. Un valor de ranura almacenado en la instancia se denomina **local** porque el valor sólo lo conoce la instancia. Por lo tanto, pueden existir diferentes instancias con diferentes valores locales. Por el contrario, un valor de ranura almacenado en una clase se denomina **compartido** porque es el mismo para *todas las* instancias de la clase.

Un valor local se especifica mediante la **faceta local**, que es la predeterminada para una ranura. Un valor compartido se especifica mediante la faceta **shared** y *todas las* instancias

con este tipo de ranura tendrán su valor de ranura cambiado automáticamente si *una* de ellas lo cambia. Una faceta de **acceso** define uno de los tres tipos de acceso para una ranura, tanto si se utilizan los controladores creados por CLIPS como si se definen los tuyos propios. El tipo por defecto, **lectura-escritura**, permite tanto leer como escribir el valor de la ranura. Los otros tipos son **sólo lectura** y **sólo inicialización**.

Otra forma de establecer los valores de instancia es con la función **initialize-instance**. Se puede llamar a (initialize-instance) en cualquier momento para restablecer los valores predeterminados y conservar los valores en las ranuras no-(default).

Un (reset) se puede considerar como una inicialización **en frío**, ya que todos los valores de las ranuras no (default) se borran, mientras que un (initialize-instance) se puede considerar como una **inicialización en caliente**, ya que los valores no (default) se conservan. Por supuesto, sólo (definstances) se puede inicializar en frío, ya que las no (definstances) simplemente se borrarán. Además, las anulaciones de ranura pueden utilizarse en (initialize-instance) como muestra el último ejemplo.

Se han diseñado dos funciones de predicado para su uso con las facetas de acceso. Ambas funciones de predicado devuelven un mensaje de error si la ranura o instancia especificada no existe. La función de predicado **slot-writablep** es una función de predicado que devuelve TRUE si se puede escribir en una ranura y FALSE si no se puede. La función de predicado **slot-initablep** devuelve TRUE si la ranura se puede inicializar (es inicializable) y FALSE si no se puede (no lo es). El término *inicializable* significa que *no* es de sólo lectura.

La **faceta inherit**, que es la predeterminada, especifica que las instancias indirectas de una clase pueden heredar esta ranura de la clase. Como recordarás, las instancias indirectas de una clase son las instancias de sus subclases, mientras que las instancias directas son las definidas específicamente para la clase. Las instancias indirectas de una clase son instancias directas de las subclases en las que están definidas. Por ejemplo, [Dorky_Duck] es una instancia directa de DUCK y una instancia indirecta de USER, que es la superclase de DUCK. La **faceta no-inherit** especifica que la ranura de clase sólo tiene una instancia directa.

Es importante tener en cuenta que la faceta (no-inherit) sólo prohíbe la herencia de la clase y no de sus superclases. Esto significa que una instancia aún puede heredar de superclases de la clase (no-inherit).

La **faceta composite** establece que las facetas que no están definidas *explícitamente* en la clase de precedencia más alta toman sus facetas de la clase de precedencia inmediatamente superior. Si la faceta no está definida explícitamente en la clase de precedencia inmediatamente superior *y también* es compuesta, CLIPS prueba en la inmediatamente superior y así sucesivamente hasta que la faceta esté definida o no haya más clases. Si la clase inmediatamente superior no es compuesta, CLIPS no realiza más comprobaciones. Lo contrario a la faceta compuesta es la **faceta exclusive**, que es la predeterminada. Para obtener más información, consulta el *Manual de referencia de CLIPS*.

Capítulo 11

Controlar los controladores

Hay dos pasos para aprender a utilizar una pala:

(1) averiguar qué parte es el mango, y (2) qué hacer con el mango.

Los controladores son esenciales en la programación orientada a objetos porque soportan la encapsulación de objetos. La única forma correcta en que los objetos pueden responder a los mensajes es teniendo un controlador apropiado para recibir el mensaje y tomar la acción correspondiente. En este capítulo aprenderás cómo son interpretados los mensajes por los objetos. Verás cómo modificar los controladores de mensajes existentes y cómo escribir los tuyos propios.

Tu yo primitivo

Hasta ahora has aprendido sobre la estructura estática de las clases a través de la jerarquía de herencia y las ranuras. Sin embargo, la parte dinámica de un objeto está determinada por sus controladores de mensajes, o controladores para abreviar, que reciben mensajes y en respuesta realizan acciones. Los controladores son responsables de las propiedades dinámicas de un objeto, que determinan su comportamiento. Ya has utilizado un controlador muchas veces - el *print* en un (send).

Polimorfismo; es una de las cinco características generalmente aceptadas de un verdadero lenguaje OOP. Por ejemplo, el mismo tipo de mensaje, (send <nombre-de-instancia> print), puede tener diferentes acciones dependiendo de la clase del objeto que lo recibe. Los objetos DUCK pueden imprimir sound, ID y age, mientras que los objetos DUCKLING sólo imprimen sound y age.

En lenguajes sin polimorfismo, habría que definir una función, (send-duck-print) para los tipos duck y otra función, (send-duckling-print) para el tipo duckling. Sin embargo, en POO, no importa cuántas clases se definan, el mismo mensaje, (send <instance-name> print), imprimirá las ranuras de los objetos. Esto mejora enormemente la eficiencia del desarrollo de programas, ya que no es necesario definir nuevas funciones para cada nuevo tipo.

El polimorfismo puede llevarse al extremo haciendo que el mismo mensaje haga cosas completamente diferentes. Por ejemplo, se podría definir un controlador de mensajes de impresión que *imprimiera* objetos de una determinada clase y *borrara* objetos de otra clase. Otra posibilidad extrema sería que el controlador de impresión no imprimiera nada. En su

lugar, borraría objetos de una clase, guardaría objetos de otra clase, añadiría objetos de otra clase, y así sucesivamente.

Este uso extremo del polimorfismo crearía una Torre de Babel para el programador y dificultaría mucho la comprensión del código, ya que todo dependería del tiempo de ejecución. Definir controladores de mensajes del mismo nombre en diferentes clases que hacen cosas completamente diferentes va en contra del Principio del Menor Sorpresa.

Otro ejemplo de polimorfismo podría definirse utilizando un controlador de mensajes para "+". Si un mensaje para "+" se envía a cadenas o símbolos, es decir, objetos LEXEME, se concatenarán debido a un controlador definido para la clase LEXEME. Si el "+" se envía a números ordinarios, el resultado será una suma, debido al controlador predefinido por el sistema para la suma de números. Si el "+" se envía a números complejos, definidos como una subclase de USER llamada COMPLEX, un controlador definido para la clase COMPLEX hará la suma de números complejos. Por lo tanto, el "+" hace tipos de operaciones afines que coincide con nuestra intuición, y no nos sorprende.

Además de los controladores predefinidos como *print*, puedes definir tus propios controladores. Empecemos escribiendo un controlador para sumar números a través de mensajes.

Como puedes ver en la Fig. 8.6 del capítulo 8, la clase NUMBER tiene subclases INTEGER y FLOAT. Dado que estas son clases predefinidas, se vería natural hacer cálculos numéricos enviando mensajes a los números. Intentémoslo de la siguiente manera.

```
CLIPS> (clear)
CLIPS> (send 1 + 2)
[MSGFUN1] No applicable primary message-handlers found for +.
FALSE
CLIPS>
```

Bueno, como puedes ver, este ejemplo no funcionó. La razón está implícita en el mensaje de error.

Vamos a comprobarlo obteniendo más información sobre la clase INTEGER ya que era el objeto de destino del mensaje de impresión.

```
CLIPS> (describe-class INTEGER)
==========================
**************************************************
Abstract: direct instances of this class cannot be created.
Direct Superclasses: NUMBER
Inheritance Precedence: INTEGER NUMBER PRIMITIVE OBJECT
```

```
Direct Subclasses:
**************************************************
==========================
CLIPS>
```

El problema es que la clase INTEGER no tiene ningún controlador para "+". De hecho, no tiene ningún controlador ya que no aparece ninguno en la lista. Como recordarás, la clase USER y sus subclases siempre tienen controladores de impresión, borrado y otros definidos automáticamente por CLIPS. Si queremos enviar mensajes de impresión a un objeto como 1 de la clase INTEGER, tendremos que hacer nuestro propio controlador de impresión.

Antes de escribir este controlador, vamos a responder a una pregunta que puedes tener acerca de cómo INTEGER puede tener instancias. Dado que INTEGER es una clase abstracta, te preguntarás cómo puede tener instancias como 1, 2, 3, etc. Aunque no se pueden crear instancias directas de una clase abstracta, sí se pueden utilizar instancias existentes. En el caso de la clase predefinida del sistema INTEGER, todos los enteros hasta el máximo permitido están disponibles como objetos. Del mismo modo, las cadenas y los símbolos están disponibles para las clases abstractas STRING y SYMBOL definidas por el sistema, y así sucesivamente para las demás clases predefinidas.

A continuación, se muestra la definición de un controlador para la clase NUMBER que controlará la suma por mensajes. El controlador se define para NUMBER en lugar de INTEGER porque también nos gustaría controlar objetos FLOAT. En lugar de definir el mismo controlador para FLOAT y para INTEGER, es más fácil definir un controlador para la superclase NUMBER. *Si un controlador para un mensaje no está definido en la clase del objeto, CLIPS prueba todos los controladores en la lista de precedencia de herencia.* Dado que "+" no está definido para INTEGER, CLIPS prueba con NUMBER, encuentra el controlador aplicable y devuelve el resultado de 3.

```
CLIPS>
; ?arg is argument of handler
(defmessage-handler NUMBER + (?arg)
; Function addition of handler
(+ ?self ?arg))
CLIPS> (send 1 + 2)
3
CLIPS> (send 2.5 + 3)
5.5
CLIPS> (send 2.5 + 2.6)
5.1
CLIPS> (describe-class NUMBER)
==========================
```

```
**************************************************
Abstract: direct instances of this class cannot be created.
Direct Superclasses: PRIMITIVE
Inheritance Precedence: NUMBER PRIMITIVE OBJECT
Direct Subclasses: INTEGER FLOAT
--------------------------------------------------------
Recognized message-handlers:
+ primary in class NUMBER
**************************************************
==============================
CLIPS>
```

La variable ? **self** es una variable especial en la que CLIPS almacena la **instancia activa**. La *variable*?self es una palabra reservada que no puede ser incluida explícitamente en un argumento de controlador, ni puede ser ligada a un valor diferente. La instancia activa es la instancia a la que se envió el mensaje. En nuestro ejemplo, todas las clases predefinidas como NUMBER, INTEGER y FLOAT son subclases de la clase PRIMITIVE. Esto contrasta con USER, que es la otra subclase principal de OBJECT.

Como otro ejemplo, escribamos un controlador para concatenar cadenas, símbolos o ambos.

```
CLIPS>
; ?arg is argument of handler
(defmessage-handler LEXEME + (?arg)
; Function concatenation of handler
(sym-cat ?self ?arg))
; SYMBOL + STRING
CLIPS> (send Dorky_ + "Duck")
Dorky_Duck ; SYMBOL result
CLIPS>
```

Observa que el controlador está definido para la clase LEXEME, ya que es una superclase tanto de SYMBOL como de STRING. En este caso, el controlador devuelve un SYMBOL ya que se utiliza (sym-cat).

Este ejemplo también ilustra por qué los corchetes pueden ser necesarios en un (send). Como se muestra en este ejemplo, el mensaje va al SYMBOL *Dorky_* sin corchetes. Con corchetes, el mensaje va a un objeto *[* Dorky_] de una clase definida por el usuario. Aquí asumimos que *[Dorky_]* podría ser un objeto de una clase definida por el usuario, como DUCK.

Hacerles pagar con un riñon

La verdadera utilidad de los controladores es con las subclases de USER ya que se pueden definir instancias de estas clases. Para ver cómo funcionan los controladores en este caso, primero configuremos el entorno de la siguiente manera.

```
CLIPS> (clear)
CLIPS>
(defclass DUCKLING (is-a USER)
(slot sound (default quack))
(slot age (visibility public)))
CLIPS>
(defclass DUCK (is-a DUCKLING)
(slot sound (default quack)))
CLIPS>
(definstances DUCKY_OBJECTS
(Dorky_Duck of DUCK (age 2))
(Dinky_Duck of DUCKLING (age .1)))
CLIPS> (reset)
CLIPS>
```

Como ejemplo sencillo, escribamos un controlador que imprima las ranuras de la instancia activa. Podemos hacer uso de la función **ppinstance** para imprimir las ranuras de la instancia activa. Esta función *no* devuelve ningún valor y sólo se utiliza para su efecto colateral de imprimir en el dispositivo estándar. Además, sólo se puede utilizar desde dentro de un controlador, ya que sólo allí se conoce la instancia activa. A continuación, se muestra un controlador definido por USER llamado print-slots que imprime las ranuras de la instancia activa utilizando (ppinstance).

```
CLIPS>
(defmessage-handler USER print-slots ()
(ppinstance))
CLIPS> (send [Dorky_Duck] print-slots)
[Dorky_Duck] of DUCK
(age 2)
(sound quack)
CLIPS>
```

Aunque en este caso el controlador podría definirse sólo para DUCK, un controlador definido para USER será llamado para todas las subclases de USER, no sólo para DUCKLING y DUCK. Así, el controlador print-slots funcionará para *todas las* subclases que podamos definir de USER.

Por supuesto, es posible dejarse llevar y definir todos los controladores de mensajes como controladores de USER. Sin embargo, es de buen estilo y mejora la eficiencia definir los

controladores tan cerca como sea posible de la clase o clases para las que están destinados. La eficiencia mejora porque CLIPS no tiene que buscar entre un montón de clases para encontrar el controlador aplicable.

Cómo desplazarse

Examinemos ahora los controladores de mensajes con más detalle. Definiremos un controlador para imprimir una cabecera cuando un objeto reciba un mensaje para imprimirse. El controlador de mensajes se define utilizando una construcción defmessage-handler tal como se indica a continuación.

```
CLIPS>
(defmessage-handler USER print before ()
(printout t "*** Starting to print ***" crlf))
CLIPS> (send [Dorky_Duck] print)
*** Starting to print ***
[Dorky_Duck] of DUCK
(age 2)
(sound quack)
CLIPS>
```

La razón por la que se imprime una cabecera en lugar de un resumen al final tiene que ver con el **tipo de controlador**. Un controlador de tipo **before** se utiliza *antes* del mensaje de impresión. Para hacer un resumen, utiliza el controlador de tipo **after** como se muestra en el siguiente ejemplo.

```
CLIPS>
(defmessage-handler USER print after ()
(printout t "*** Finished printing ***"
crlf))
CLIPS> (send [Dorky_Duck] print)
*** Starting to print ***
[Dorky_Duck] of DUCK
(age 2)
(sound quack)
*** Finished printing ***
CLIPS>
The general format of a message-handler is as follows.
(defmessage-handler <class-name>
<message-name> [handler-type]
[comment]
(<parameters>* [wildcard-parameter])
```

<action>*)

Aunque puede haber varias acciones en un controlador, *sólo se devuelve el valor de la última acción*. Fíjate que esto es igual que una (deffunción).

Como [Dorky_Duck] es de la clase DUCK, una subclase de USER, podemos aprovechar el controlador de **impresión** que está predefinido por CLIPS para la clase USER. Todas las subclases de USER pueden aprovechar los controladores de USER, lo que te ahorra la molestia de escribir controladores para cada clase que definas. Observa cómo el concepto de herencia de USER a su subclase DUCK simplifica el desarrollo del programa al permitir la reutilización del código existente, es decir, el controlador de impresión de USER.

Los paréntesis vacíos () que siguen al tipo de controlador *before* significan que no hay parámetros ni un parámetro comodín. En otras palabras, el controlador de cabecera no recibe argumentos, por lo que los paréntesis están vacíos, pero son necesarios. Ten en cuenta que, aunque se pueden utilizar varios parámetros, sólo puede haber un comodín.

Consideraciones principales

Como puedes ver, el controlador resumen es el mismo que el controlador de cabecera excepto que se usa un controlador tipo *after*, y el texto de la acción es diferente. Así, un tipo de controlador before realiza su tarea antes del tipo de controlador **primary**, y un controlador after realiza su tarea después del controlador primary. Un *primary* está destinado a realizar la tarea principal. Un controlador de tipo "around" está pensado para preparar el entorno para el resto de los controladores. Los tipos before y after están pensados para tareas menores como inicializar variables o imprimir, mientras que el primary realiza la tarea principal.

La siguiente lista resume el papel de la clase y el valor de retorno de cada tipo de controlador:

around: Configura el entorno para otros controladores. Devuelve un valor.
before: Trabajo auxiliar antes que primario. Sin valor de retorno.
primary: Realiza la tarea principal del mensaje. Devuelve un valor.
after: Trabajo auxiliar después del primario. Sin valor de retorno.

Los tipos de controladores se listan en el orden en que normalmente son llamados durante la ejecución de un mensaje. Dependiendo del tipo de controlador, CLIPS sabe cuando ejecutarlo. Es decir, un controlador **around** comienza antes que cualquier controlador before. Un controlador before se ejecuta antes que cualquier controlador primary, al que siguen los controladores after. La excepción a esta ejecución secuencial de controladores es el controlador de tipo around. Si se define un controlador around, comenzará la ejecución antes que cualquiera de los otros, realizará las acciones especificadas, y luego completará sus acciones *después de* que todos los otros tipos de controladores hayan terminado. Pronto verás un ejemplo detallado de la ejecución de estos controladores.

El **rol de la clase** describe el propósito de cada tipo. El valor de retorno describe si el tipo de controlador está destinado generalmente a un **valor de retorno** o simplemente a proporcionar un efecto colateral útil como la impresión**.** Esta consideración dependerá del controlador. Por ejemplo, muchos controladores primarios definidos por el usuario pueden escribirse para devolver un valor como resultado de algún cálculo numérico u operación de cadena. Una excepción a la devolución de un valor de retorno útil es un controlador primario de impresión cuya tarea principal es el efecto colateral de imprimir, y no tiene un valor de retorno.

A continuación, se muestra la lista de controladores primarios predefinidos de USER y su función en la clase. Por herencia, están disponibles para todas las subclases de USER.

init: Inicializa una instancia.
delete: Elimina una instancia.
print: Imprime una instancia.
direct-modify: Modifica directamente las ranuras.
message-modify: Modifica las ranuras mediante mensajes put-.
direct-duplicate: Duplica una instancia sin utilizar mensajes put-.
message-duplicate: Duplica una instancia utilizando mensajes.

Estos controladores primarios están predefinidos y no pueden ser modificados a menos que cambies el código fuente de CLIPS. Sin embargo, puedes definir los tipos de controladores *before*, *after* y *around* para estos primarios. Ya has visto un ejemplo de cómo cambiar los tipos de controlador antes y después para el controlador de impresión USER. Ahora veamos algunos ejemplos de definición de los tipos de controlador before y after para el controlador primario **init**.

```
CLIPS>
(defmessage-handler USER init before ()
(printout t
"*** Starting to make instance ***"
 crlf))
CLIPS>
(defmessage-handler USER init after ()
(printout t
"*** Finished making instance ***"
crlf))
CLIPS> (reset)
*** Starting to make instance ***
*** Finished making instance ***
*** Starting to make instance ***
*** Finished making instance ***
CLIPS>
```

```
(make-instance Dixie_Duck of DUCK
(age 1))
*** Starting to make instance ***
*** Finished making instance ***
[Dixie_Duck]
CLIPS> (instances)
[Dorky_Duck] of DUCK
[Dinky_Duck] of DUCKLING
[Dixie_Duck] of DUCK
For a total of 3 instances.
CLIPS>
```

El poder de creer

El parámetro self es útil porque puede utilizarse para *leer* un valor de ranura de la forma

```
?self:<slot-name>
```

También puede utilizarse para *escribir* un valor de ranura utilizando la función bind. La notación "?self:" es más eficiente que el envío de mensajes, pero sólo se puede utilizar desde dentro de un controlador en su instancia activa.

En cambio, las funciones **dynamic-get-** y **dynamic-put-** pueden utilizarse desde dentro de un controlador para leer y escribir un valor de ranura de la instancia activa. Aunque se pueden utilizar mensajes desde dentro de un controlador, como el siguiente, no es eficiente.

```
(send ?self dynamic-get-<slot>)
(send ?self dynamic-put-<slot>)
```

Como ejemplo de dynamic-get-, cambiemos nuestro ejemplo de la siguiente manera.

```
CLIPS>
(defmessage-handler USER print-age primary ()
(printout t "*** Starting to print ***"
crlf
"Age = " (dynamic-get age)
crlf
"*** Finished printing ***"
crlf))
CLIPS> (send [Dorky_Duck] print-age)
*** Starting to print ***
Age = 2
*** Finished printing ***
```

```
CLIPS>
```

El parámetro ?self:age sólo puede utilizarse en una clase y sus subclases que hereden la ranura age. El parámetro ?

self:<nombre-de-la-ranura> se evalúa de forma *estática* a través de la herencia. Esto significa que si una subclase redefine una ranura, un controlador de mensajes de la superclase fallará si intenta acceder directamente a la ranura usando ?

```
self:<slot-name>.
```

Por el contrario, los métodos dynamic-get- y dynamic-put- pueden ser utilizados por superclases y subclases porque comprueban las ranuras *dinámicamente*. Sin embargo, para que una superclase pueda hacer referencia dinámicamente a una ranura, la faceta de visibilidad de la ranura debe ser pública. El siguiente ejemplo *no* funcionaría si DUCK se cambia por USER.

```
CLIPS>
(defmessage-handler DUCK print-age primary ()
(printout t "*** Starting to print ***"
crlf
"Age = " ?self:age
crlf
"*** Finished printing ***"
crlf))
CLIPS> (send [Dorky_Duck] print-age)
*** Starting to print ***
Age = 2
*** Finished printing ***
CLIPS>
```

Como ejemplo del uso de una función dynamic-put- en un controlador, supongamos que queremos ayudar a Dorky_Duck a recuperar algo de su juventud. El siguiente ejemplo muestra cómo se puede cambiar su edad utilizando un controlador. Este ejemplo también ilustra cómo se puede pasar un valor a un controlador a través de la variable ?change.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
(slot sound (default quack))
(slot age))
CLIPS>
(defmessage-handler DUCK lie-about-age
```

```
(?change)
(bind ?new-age (- ?self:age ?change))
(dynamic-put age ?new-age)
(printout t "*** Starting to print ***"
crlf
"I am only " ?new-age
crlf
"*** Finished printing ***" crlf))
CLIPS>
(make-instance [Dorky_Duck] of DUCK (age 3))
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] lie-about-age 2)
*** Starting to print ***
I am only 1
*** Finished printing ***
CLIPS>
```

Como puedes ver, la creencia de Dorky_Duck es tan fuerte que la edad cambiada se pone en su ranura age.

Fíjate cómo el controlador utiliza la variable ?new-age para almacenar la edad cambiada, que luego se pone en la ranura age de la instancia.

El demonio de la verdad

Un **daemon** es un controlador que se ejecuta cada vez que se realiza alguna **acción básica** como inicializar, borrar, obtener o poner en una instancia. Una regla no puede considerarse un demonio porque no es seguro que se ejecute sólo porque se cumplan sus patrones LHS. Lo único seguro es que una regla se *activará* cuando se satisfagan sus LHS — no que se ejecutará.

No hay una palabra clave explícita para daemon ya que es sólo un concepto. Los controladores before y after que imprimían cadenas pueden considerarse demonios de impresión. Estos controladores esperaron un mensaje (send [Dorky_Duck] print-age) para desencadenar su acción. Primero el controlador before imprimió su cadena, luego el controlador primary imprimió, y finalmente el controlador after imprimió. Un demonio es el controlador after que espera un mensaje de impresión. El segundo demonio es el controlador after que espera a que el primario de impresión termine de imprimir.

La impresión no se considera una acción básica porque no hay valor de retorno asociado a un (send <instance> print). El mensaje print sólo se envía para el efecto colateral de imprimir. Por el contrario, un mensaje (send <instance> get-<slot>) devolverá un valor que puede ser utilizado por otro código.

Del mismo modo, la inicialización, la eliminación, y poner todos tienen un efecto sobre una instancia y por lo tanto se consideran acciones básicas como get.

Los demonios se implementan fácilmente utilizando controladores before y after, ya que se ejecutarán antes y después de su controlador principal. Implementar demonios de esta manera se llama **implementación declarativa** porque no es necesaria ninguna acción explícita por parte del controlador para que se ejecute. Es decir, CLIPS siempre ejecutará un controlador before antes de su primario y siempre ejecutará un controlador after después de su primario. En una implementación declarativa de demonios, el funcionamiento normal de CLIPS hará que los demonios se activen cuando llegue su momento. Así, la implementación declarativa está implícita en la operación normal.

Lo opuesto a la ejecución implícita es la implementación **imperativa** en la que las acciones se programan explícitamente. Usar el controlador around es muy práctico para demonios imperativos. La idea básica del controlador around es la siguiente.

1. Comienza antes que cualquier otro controlador.
2. Llama al siguiente controlador usando **call-next-handler** para pasar los mismos argumentos u **override-next-handler** para pasar otros diferentes.
3. Continúa la ejecución cuando finalice el último controlador.
4. Después de que cualquier otro controlador around, before, primary o after termine, el controlador around reanuda la ejecución.

La palabra clave *call-next-handler* se utiliza para llamar a los siguientes controladores. Se dice que un controlador está *en la sombra* de un **shadower** si debe ser llamado desde el shadower por una función como call-next-handler. La función call-next-handler puede utilizarse varias veces para llamar a los mismos controladores.

Se utiliza una función de predicado llamada **next-handlerp** para comprobar la existencia de un controlador antes de realizar la llamada. Si no existe ningún controlador, next-handlerp devolverá FALSE.

El siguiente ejemplo ilustra el controlador around de un demonio sincero que delata a Dorky_Duck cada vez que miente sobre su edad.

```
CLIPS>
(defmessage-handler DUCK lie-about-age around
(?change)
(bind ?old-age ?self:age)
(if (next-handlerp) then
(call-next-handler))
(bind ?new-age ?self:age)
(if (<> ?old-age ?new-age) then
(printout t "Dorky_Duck is lying!"
```

```
crlf
"Dorky_Duck is lying!"
crlf
"He's really " ?old-age
crlf)))
CLIPS>
(make-instance [Dorky_Duck] of DUCK (age 3))
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] lie-about-age 1)
*** Starting to print ***
I am only 2
*** Finished printing ***
Dorky_Duck is lying!
Dorky_Duck is lying!
He's really 3
CLIPS>
```

Aunque Dorky_Duck siga mintiendo sobre su edad, el demonio dice la verdad.

Fíjate en el argumento ?change. Aunque el controlador around no usa ?change, el primario lie-about-age que es llamado por el controlador call-next si lo necesita para cambiar la edad. Por lo tanto, el controlador "around" debe pasar ?change al primario. Si se omite ?change, aparecerá un mensaje de error.

El call-next-handler *siempre* pasa los argumentos del shadower al controlador shadowed.

Es posible pasar *diferentes* argumentos a un controlador shadowed mediante el uso de la función override-next-handler como se muestra en el siguiente ejemplo.

```
CLIPS>
(defmessage-handler DUCK lie-about-age around
(?change)
(bind ?old-age ?self:age)
(if (next-handlerp) then
; Divide age in half!
(override-next-handler (/ ?change 2)))
(bind ?new-age ?self:age)
(if (<> ?old-age ?new-age) then
(printout t "Dorky_Duck is lying!"
crlf
"Dorky_Duck is lying!"
crlf
```

```
"He's really " ?old-age
crlf)))
CLIPS>
(make-instance [Dorky_Duck] of DUCK (age 3))
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] lie-about-age 1)
*** Starting to print ***
I am only 2.5
*** Finished printing ***
Dorky_Duck is lying!
Dorky_Duck is lying!
He's really 3
CLIPS>
```

- *Es importante tener en cuenta que el valor de retorno de call-next-handler y override-next-handler es el de los controladores en la sombra.*

A continuación, se muestran las reglas de envío de mensajes:

1. Todos los controladores around comienzan la ejecución. A continuación, los controladores before, primary y after comienzan y terminan, seguidos por la finalización de la ejecución de around.
2. Los controladores around, before y primary son llamados en orden de mayor a menor clase de precedencia.
3. Los controladores after son llamados de menor a mayor precedencia.
4. Cada controlador around debe llamar explícitamente al siguiente controlador shadowed.
5. Los primary de mayor precedencia deben llamar explícitamente a los primary de menor precedencia (shadowed) para poder ejecutarse.

Sin embargo, ten en cuenta el siguiente prerrequisito para cualquier gestión de mensajes: *Debe haber al menos un controlador primary aplicable.*

Dado que sólo los controladores around y primary pueden devolver valores, y los around hacen sombra a los primary, se deduce que el valor de retorno de un (send) será el valor de retorno around. Si no hay around, entonces el valor de retorno será el del primary de mayor precedencia.

La siguiente lista resume el valor de retorno de los tipos de controladores.

around: Ignorar o capturar el valor de retorno de la siguiente vuelta más específica o primaria.
before: Ignorar. Sólo efecto colateral.

primary: Ignorar o capturar el valor de retorno del ámbito primario más general.
after: Ignorar. Sólo efecto colateral.

Entiende el punto

Hasta ahora, hemos hablado de la herencia utilizando únicamente relaciones is-a. Como has visto, este tipo de relación de herencia es buena para definir clases cada vez más especializadas. Es decir, empiezas definiendo las clases más generales como una subclase de USER, y luego defines clases más especializadas con más ranuras en los niveles inferiores de la jerarquía de clases.

Normalmente, se diseñan nuevas clases como especializaciones de otras ya existentes. Este paradigma es la **herencia por especialización**. Como recordará del ejemplo del cuadrilátero de la Fig. 8.7 del capítulo 8, el nivel más alto es cuadrilátero, y los niveles inferiores son trapezoide, paralelogramo, rectángulo, y luego cuadrado a lo largo de una ruta de herencia. Trapezoide es una clase especial de cuadrilátero, paralelogramo es una clase especial de trapezoide, rectángulo es una clase especial de paralelogramo, y cuadrado es una clase especial de rectángulo.

La herencia también puede utilizarse para construir clases más complejas. Sin embargo, esto no es tan directo en CLIPS. Como ejemplo, la clase básica para la geometría es un POINT que contiene una única ranura point1. Una LINE puede definirse añadiendo el point2 al POINT. Un TRIANGLE se define añadiendo el point3 a la LINE, y así sucesivamente para QUADRILATERALS, PENTAGONS, etc.

```
(defclass POINT (is-a USER)
      (multislot point1))

(defclass LINE (is-a POINT)
      (multislot point2))

(defclass TRIANGLE (is-a LINE)
      (multislot point3))
```

Fijate cómo cada clase es una especialización de su clase padre heredando los puntos de la superclase y añadiendo después un nuevo punto.

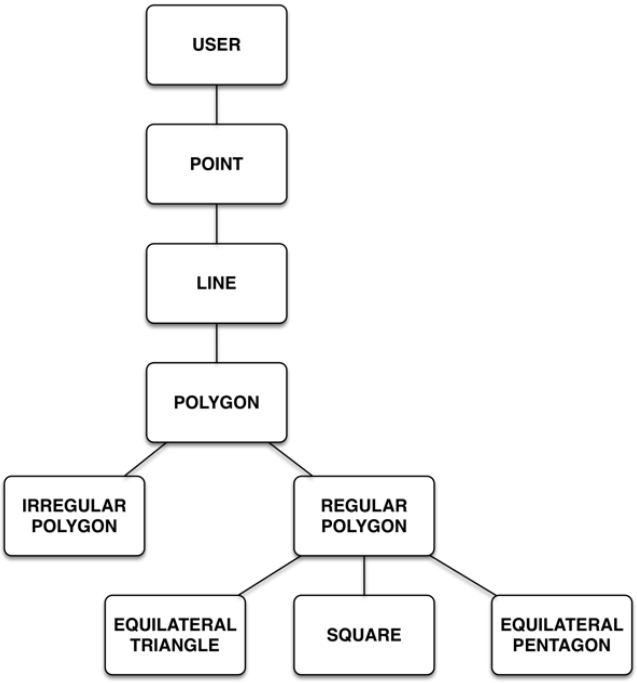
El paradigma opuesto es la **herencia por generalización**, en la que se construyen clases más generales a partir de clases simples. Por ejemplo, se considera que una LINE está formada por dos puntos. Un TRIANGLE está formado por tres líneas. Un QUADRILATERAL está formado por cuatro líneas, y así sucesivamente. Este sería un buen paradigma para construir un sistema de dibujo orientado a objetos en el que se pudieran construir nuevos objetos a partir de otros más simples.

Hay una sutil pero importante diferencia entre este ejemplo de especialización y generalización. En la especialización, las nuevas clases se construyen añadiendo ranuras especializadas que son del mismo tipo que las ranuras de la superclase. Así, POINT, LINE y TRIANGLE tienen ranuras de tipo point.

En cambio, la generalización se construye utilizando nuevos tipos de ranuras definidos para cada clase. La clase POINT tiene una ranura de tipo point. LINE tiene dos ranuras de tipo point. TRIANGLE tiene tres ranuras de tipo line. CUADRILATERAL tiene cuatro ranuras de tipo line, y así sucesivamente. La generalización es buena para la **síntesis**, es decir, para la construcción. Lo contrario de la síntesis es el **análisis**, que significa desmontar o simplificar. El modelo de análisis es la especialización.

La Fig. 11.1 ilustra un esquema de herencia para polígonos en el que las clases se construyen por herencia. En este caso, el vínculo entre clases sería "is-made-of ". Así, una LINE está hecha de POINT. Un POLYGON está hecho de LINE, y así sucesivamente.

Fig. 11.1 Jerarquía de polígonos con algunas clases de polígonos regulares



Los enlaces como "is-made-of " pueden simularse en CLIPS mediante definiciones de ranura apropiadas, aunque en la versión 6.3 sólo se admitan enlaces is-a. Como ejemplo de generalización, construyamos una clase LINE como generalización de una clase POINT. La clase POINT proporcionará instancias que tienen una posición. Para que este ejemplo sea realista, asumiremos un número arbitrario de dimensiones definiendo la posición como (multiple). Así, un punto unidimensional tendrá un valor en la ranura position, un punto

bidimensional tendrá dos valores, y así sucesivamente. La definición de la clase POINT es muy sencilla.

```
CLIPS> (clear)
CLIPS>
(defclass POINT (is-a USER)
(multislot position
(propagation no-inherit)))
CLIPS>
```

La faceta (no-inherit) se utiliza para evitar que una LINE herede una ranura position. En su lugar, una LINE será definida por dos puntos llamados ranura point1 y ranura point2. Estas dos ranuras definirán la línea y es superfluo tener una ranura position adicional por herencia.

La definición de la clase LINE es un poco más compleja. La razón de la complejidad añadida es que los detalles de implementación se incluyen en la (defclass) porque la Versión 6.3 sólo soporta relaciones is-a.

```
(defclass LINE (is-a POINT)
(slot point1
(default-dynamic
(make-instance (gensym*) of POINT))
(propagation no-inherit))
(slot point2
(default-dynamic
(make-instance (gensym*) of POINT))
(propagation no-inherit))
(message-handler find-point)
(message-handler print-points)
(message-handler find-distance)
(message-handler print-distance))
```

Ten en cuenta que los controladores de mensajes de LINE se **declaran hacia adelante** a efectos de documentación.

En este momento te estarás preguntando por qué POINT y LINE no están definidas como subclases de USER ya que todas sus ranuras tienen facetas (no-inherit). Dado que todas las ranuras de POINT, LINE y la clase TRIANGLE que se definirá más adelante tienen facetas (no-inherit s), todas estas clases podrían definirse como subclases directas de USER en lugar de definir LINE como subclase de POINT y TRIANGLE como subclase de LINE.

Sin embargo, el objetivo de este ejemplo es ilustrar la herencia por generalización, un concepto lógico que la versión 6.3 no admite directamente. Por lo tanto, definir LINE como subclase de POINT y TRIANGLE como subclase de LINE se hace para documentar el concepto lógico de herencia por generalización. Es cierto que se podría añadir un comentario junto a (defclass LINE (is-a USER)) y (defclass TRIANGLE (is-a USER)) indicando que estamos intentando implementar la Herencia por Generalización, pero ver código en su lugar es una mejor documentación. Si alguna vez CLIPS soporta directamente la Herencia por Generalización, estas declaraciones (defclass) facilitarán la conversión.

La razón de incluir (make-instance (gensym*)) en las ranuras de LINE es proporcionar la herencia de la clase POINT. Con la herencia por especialización estándar sólo es posible una ranura position de LINE porque POINT sólo tiene una ranura position. No es posible que tanto el slot point1 como el slot point2 de LINE hereden el slot position de POINT. El valor real de la ranura de cada LÍNEA será un valor gensym*. Cada valor gensym* será el nombre de instancia de una instancia de punto. Se puede acceder a la posición del punto a través del valor gensym*. Así, los valores gensym* actúan como punteros a diferentes instancias.

Esta técnica de **acceso indirecto** de valores (gensym*) es análoga al uso de un puntero para acceder a un valor en un lenguaje procedimental. Así, los diferentes slots de LINE pueden heredar indirectamente los mismos slots de POINT. Es conveniente usar (gensym*) porque no nos importa cuáles son los nombres de los punteros de LINE, igual que no nos importa cuáles son las direcciones de los punteros en un lenguaje procedimental.

Los siguientes ejemplos muestran cómo se accede a los puntos para puntos unidimensionales en posición 0 y 1.

```
CLIPS>
(definstances LINE_OBJECTS
(Line1 of LINE))
CLIPS> (reset)
CLIPS>
(send (send [Line1] get-point1)
put-position 0)
(0)
CLIPS>
(send (send [Line1] get-point2)
put-position 1)
(1)
CLIPS> (send [Line1] print)
[Line1] of LINE
(point1 [gen1])
(point2 [gen2])
CLIPS>
```

```
(send (send [Line1] get-point1)
get-position)
(0)
CLIPS>
(send (send [Line1] get-point2)
get-position)
(1)
CLIPS>
```

Ahora que entiendes cómo funciona el acceso indirecto, vamos a definir algunos controladores para evitar la molestia de introducir los mensajes anidados (send), como en los dos últimos casos. Definamos un controlador llamado find-point para imprimir un valor de punto especificado, y un controlador llamado print-points para imprimir los valores de ambos puntos de LINE tal como se muetra a continuación. El argumento de find-point será un 1 para point1 o un 2 para point2.

```
CLIPS>
(defmessage-handler LINE find-point (?point)
(send (send ?self
(sym-cat "get-point" ?point))
get-position))
CLIPS>
(defmessage-handler LINE print-points ()
(printout t "point1 "
(send ?self find-point 1)
crlf
"point2 "
(send ?self find-point 2)
crlf))
CLIPS> (send [Line1] find-point 1)
(0)
CLIPS> (send [Line1] find-point 2)
(1)
CLIPS>
```

Para un uso real, sería mejor proporcionar detección de errores para que sólo se permita un 1 o 2.

Como puedes ver, el controlador funciona bien para puntos unidimensionales. Se puede probar para puntos bidimensionales de la siguiente manera. Asumiremos que el primer número para cada punto es el valor-X, y el segundo número es el valor-Y. Es decir, el punto 1 tiene el valor-X 1 y el valor-Y 2.

```
CLIPS>
(send (send [Line1] get-point1)
put-position 1 2)
(1 2)
CLIPS>
(send (send [Line1] get-point2)
put-position 4 6)
(4 6)
CLIPS> (send [Line1] print)
[Line1] of LINE
(point1 [gen1])
(point2 [gen2])
CLIPS>
(send (send [Line1] get-point1)
get-position)
(1 2)
CLIPS>
(send (send [Line1] get-point2)
get-position)
(4 6)
CLIPS>
```

Como era de esperar, el controlador también funciona correctamente para puntos bidimensionales.

Ahora que tenemos las posiciones de los dos puntos, es fácil encontrar la distancia entre los puntos, que es la longitud de la línea. La distancia se puede determinar definiendo un nuevo controlador llamado find-distance que utiliza el Teorema de Pitágoras para calcular la distancia como la raíz cuadrada de la suma de los cuadrados. Como no se han hecho suposiciones en cuanto al número de dimensiones, se utilizará la función (nth$) para escoger cada valor multicampo hasta el número máximo de coordenadas, almacenados en la variable ?len.

```
CLIPS>
(defmessage-handler LINE find-distance ()
(bind ?sum 0)
(bind ?index 1)
(bind ?len
(length$ (send ?self find-point 1)))
(bind ?Point1 (send ?self find-point 1))
(bind ?Point2 (send ?self find-point 2))
```

```
(while (<= ?index ?len)
(bind ?dif (- (nth$ ?index ?Point1)
(nth$ ?index ?Point2)))
(bind ?sum (+ ?sum (* ?dif ?dif)))
(bind ?index (+ ?index 1)))
(bind ?distance (sqrt ?sum)))
CLIPS>
(defmessage-handler LINE print-distance ()
(printout t "Distance = "
(send ?self find-distance)
crlf))
CLIPS> (send [Line1] print-distance)
Distance = 5.0
CLIPS>
```

Los valores 1, 2 para el point1 y 4, 6 para el point2 se eligieron para facilitar la comprobación del controlador, ya que estas coordenadas definen un triángulo 3-4-5. Como se puede ver, la distancia es 5,0, tal como se esperaba.

Mapas del tesoro

Ahora que las clases POINT y LINE han sido definidas por herencia generalizada, ¿por qué parar ahora? Continuemos con la siguiente clase más sencilla que puede definirse a partir de una línea: el triángulo.

A continuación, se muestran las tres defclases necesarias.

```
CLIPS> (clear)
CLIPS>
(defclass POINT (is-a USER)
(multislot position
(propagation no-inherit)))
CLIPS>
(defclass LINE (is-a POINT)
(slot point1
(default-dynamic
(make-instance (gensym*) of POINT))
(propagation no-inherit))
(slot point2
(default-dynamic
(make-instance (gensym*) of POINT))
```

```
(propagation no-inherit)))
CLIPS>
(defclass TRIANGLE (is-a LINE)
(slot line1
(default-dynamic
(make-instance (gensym*) of LINE))
(propagation no-inherit))
(slot line2
(default-dynamic
(make-instance (gensym*) of LINE))
(propagation no-inherit))
(slot line3
(default-dynamic
(make-instance (gensym*) of LINE))
(propagation no-inherit)))
CLIPS>
```

Fíjate en que las facetas (no-inherit) de TRIANGLE no son técnicamente necesarias, ya que no hay definida ninguna subclase de TRIANGLE. La razón para incluir las facetas (no-inherit) es la **programación defensiva**, que es análoga a la conducción defensiva. Si se añade otra subclase, ya sea por tí o por otra persona, la (defclass) de TRIANGLE debe ser modificada para incluir las facetas (no-inherit). Esto es de mal estilo porque significa que el código existente y depurado debe modificarse. Si vas a mejorar código existente y depurado, no deberías tener que modificarlo. Es mejor planificar con antelación las mejoras.

A continuación, definiremos una instancia de triángulo y comprobaremos las instancias generadas. Ten en cuenta que tus valores gen pueden ser diferentes de los mostrados a menos que acabes de arrancar o reiniciar CLIPS, o no hayas usado (gensym*) o (gensym) desde que empezaste.

```
CLIPS>
(definstances TRIANGLE_OBJECTS
(Triangle1 of TRIANGLE))
CLIPS> (reset)
CLIPS> (instances)
[Triangle1] of TRIANGLE
[gen3] of LINE
[gen4] of POINT
[gen5] of POINT
[gen6] of LINE
[gen7] of POINT
[gen8] of POINT
```

```
[gen9] of LINE
[gen10] of POINT
[gen11] of POINT
For a total of 10 instances.
CLIPS>
```

Al principio puede que te sorprendan todos los valores gensym* creados. Sin embargo, todos ellos son necesarios. En primer lugar, se creó gen3 para la ranura line1, que requería gen4 y gen5 para las ranuras point1 y point2 asociadas a ella por herencia. En segundo lugar, se creó gen6 para la ranura line2, que requirió gen7 y gen8 para sus ranuras point1 y point2. Por último, se ha creado gen9 para la ranura línea3, que requiere gen10 y gen11 para las ranuras point1 y point2 asociadas a ella. A continuación, se muestran las ranuras para [Triangle1] y uno de sus valores de puntero, [gen3].

```
CLIPS> (send [Triangle1] print)
[Triangle1] of TRIANGLE
(line1 [gen3])
(line2 [gen6])
(line3 [gen9])
CLIPS> (send [gen3] print)
[gen3] of LINE
(point1 [gen4])
(point2 [gen5])
CLIPS>
```

Ahora pongamos las coordenadas X, Y de [Triangle1] como sigue.

```
CLIPS>
(send (send (send [Triangle1] get-line1)
get-point1)
put-position -1 0)
(-1 0)
CLIPS>
(send (send (send [Triangle1] get-line1)
get-point2)
put-position 0 2)
(0 2)
CLIPS>
(send (send (send [Triangle1] get-line2)
get-point1)
put-position 0 2)
```

```
(0 2)
CLIPS>
(send (send (send [Triangle1] get-line2)
get-point2)
put-position 1 0)
(1 0)
CLIPS>
(send (send (send [Triangle1] get-line3)
get-point1)
put-position 1 0)
(1 0)
CLIPS>
(send (send (send [Triangle1] get-line3)
get-point2)
put-position -1 0)
(-1 0)
CLIPS>
```

Los valores almacenados son los siguientes.

```
CLIPS> (send [Triangle1] print)
[Triangle1] of TRIANGLE
(line1 [gen3])
(line2 [gen6])
(line3 [gen9])
CLIPS> (send [gen3] print)
[gen3] of LINE
(point1 [gen4])
(point2 [gen5])
CLIPS> (send [gen4] print)
[gen4] of POINT
(position -1 0)
CLIPS> (send [gen5] print)
[gen5] of POINT
(position 0 2)
CLIPS>
```

Como puedes ver, el puntero de la line1 [gen1] apunta a point1 y point2 con sus punteros [gen2] y [gen3]. Estos dos últimos punteros apuntan finalmente a los valores reales de (-1 0) y (0 2) que definen la line1 de [Triangle1]. Es análogo a Long John Silver encontrando un cofre del tesoro con un mapa, [gen1], que conduce a otro cofre con dos mapas, [gen2] y

[gen3], que conducen a los dos tesoros enterrados en las ubicaciones especificadas por [gen2] y [gen3].

Los valores almacenados para cada línea de [Triangle1] pueden recuperarse mediante un único comando utilizando mensajes anidados como los siguientes.

```
(send (send (send [Triangle1] get-line1)
                              get-point1)
                              get-position)
```

Aunque estos comandos funcionan, no es muy divertido teclearlos a menos que te paguen por horas y necesites practicar mecanografía. Como habrás adivinado por los controladores de LINE que definimos en la sección anterior, es posible definir controladores de TRIANGLE de la siguiente manera.

```
CLIPS>
(defmessage-handler TRIANGLE find-line-point
(?line ?point)
(send (send (send ?self
(sym-cat "get-line"
?line))
(sym-cat "get-point" ?point))
get-position))
CLIPS>
(defmessage-handler TRIANGLE print-line
(?line)
(printout t "point1 "
(send ?self find-line-point
?line 1)
crlf
"point2 "
(send ?self find-line-point
?line 2)
crlf))
CLIPS> (send [Triangle1] print-line 1)
point1 (-1 0)
point2 (0 2)
CLIPS>
```

Utilizar estos controladores es mucho más cómodo que teclear los mensajes anidados.

Llegados a este punto, puedes estar tentado de definir un controlador llamado find-line que devuelva ambos valores de punto de la línea especificada. Recuerda que find-line-point requiere la especificación tanto de la línea como de uno de los dos puntos que definen la línea. Entonces, ¿por qué no enviar dos mensajes en el mismo controlador para devolver los valores de ambos puntos de la línea especificada? A continuación se muestra el controlador para find-line y lo que devuelve para line1 de [Triangle1].

```
CLIPS>
(defmessage-handler TRIANGLE find-line (?line)
(send (send (send ?self
(sym-cat "get-line"
 ?line))
get-point1)
get-position)
(send (send (send ?self
(sym-cat "get-line"
?line))
get-point2)
get-position))
CLIPS> (send [Triangle1] find-line 1)
(0 2)
CLIPS>
```

Como puedes ver, el controlador sólo devuelve el último valor de mensaje de (0 2). Así, el primer valor de (-1 0) no es devuelto por CLIPS. Esto es como el caso de las deffunction que sólo devuelven la última acción. Una forma de evitar este problema y devolver los dos valores del punto de la línea es la que se muestra a continuación.

```
CLIPS>
(defmessage-handler TRIANGLE find-line (?line)
(create$
(send
(send (send ?self
(sym-cat "get-line"
?line))
get-point1)
get-position)
(send
(send (send ?self
(sym-cat "get-line"
?line))
```

```
get-point2)
get-position)))
CLIPS> (send [Triangle1] find-line 1)
(-1 0 0 2)
CLIPS>
```

Fíjate en que se ha utilizado la función (create$) para combinar ambos valores de punto en un único valor multicampo (-1 0 0 2) que luego se retornó.

Otras características

Existen otras funciones útiles para los controladores. Algunas de ellas son las siguientes.

undefmessage-handler: Elimina un controlador especificado.
list-defmessage-handlers: Lista de controladores.
delete-instance: Actúa sobre el controlador activo.
message-handler-existp: Devuelve TRUE si el controlador existe, en caso contrario FALSE.

Las **funciones de agrupación** agrupan los elementos COOL en una variable multicampo.

get-defmessage-handler-list: Agrupa los nombres de las clases, los nombres de los mensajes y los tipos (directos o heredados).
class-superclasses: Agrupa todos los nombres de superclases (directas o heredadas).
class-subclasses: Agrupa todos los nombres de subclases (directas o heredadas).
class-slots: Agrupa todos los nombres de ranura (definidos explícitamente o heredados).
slot-existp: Devuelve TRUE si la ranura de clase existe, en caso contrario FALSE.
slot-facets: Agrupa los valores de faceta de ranura especificados de una clase.
slot-sources: Agrupa los nombres de ranura de las clases que contribuyen a una ranura de la clase especificada.

La función **preview-send** es útil en depuración ya que muestra la secuencia de todos los controladores que *potencialmente* pueden estar involucrados en el procesamiento de un mensaje. La razón del término *potencialmente* es que los controladores en la sombra no se ejecutarán si el sombreador no utiliza call-next-handler o override-next-handler.

Los controladores se organizan en una **precedencia mensaje-controlador** que determina cómo son llamados. El proceso de determinar qué controladores deben ser llamados y en qué orden se denomina **envío de mensajes**. Cada vez que se envía un mensaje a un objeto, CLIPS organiza el envío de mensajes para los controladores aplicables de ese objeto, que

pueden verse mediante el comando (preview-send). Los controladores aplicables son todos los controladores en todas las clases a lo largo de la ruta de herencia del objeto que pueden responder al tipo de mensaje. A continuación, se indican otras funciones útiles para la comparación de patrones de objetos mediante reglas.

object-pattern-match-delay: Retrasa la concordancia de patrones de reglas hasta después de crear, modificar o eliminar instancias.

modify-instance: Modifica la instancia utilizando anulaciones de ranura. La concordancia de patrones de objetos se retrasa hasta después de las modificaciones.

active-modify-instance: Modifica los valores de la instancia concurrente con concordancia de patrón de objeto con mensaje de modificación directa.

message-modify-instance: Cambia los valores de la instancia. Retrasa la concordancia de patrones de objetos hasta que se cambien todas las ranuras.

active-message-modify-instance: Cambia los valores de la instancia de forma concurrente con la concordancia de patrones de objetos utilizando *message-modify*.

Capítulo 12

Preguntas y respuestas

> *La mejor manera de aprender es haciéndose preguntas; la mejor manera de arrepentirse es respondiendo a todas ellas.*

En este capítulo aprenderás cómo concordar patrones en instancias. Una forma es con reglas. También, CLIPS tiene un número de funciones de consulta para hacer concordar instancias. Además, los hechos de control y los demonios de ranura pueden usarse para la concordancia de patrones.

Lecciones de Objetos

Una de las novedades de la versión 6.0 es la posibilidad de que las reglas concuerden con patrones de objetos. El siguiente ejemplo muestra cómo el valor de la ranura sound se hace corresponder con un patrón mediante una regla.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
(multislot sound (default quack quack)))
CLIPS>
(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS>
(make-instance [Dinky_Duck] of DUCK)
[Dinky_Duck]
CLIPS>
(defrule find-sound
?duck <- (object (is-a DUCK)
(sound $?find))
=>
(printout t "Duck "
(instance-name ?duck)
" says " ?find crlf))
CLIPS> (run)
Duck [Dinky_Duck] says (quack quack)
Duck [Dorky_Duck] says (quack quack)
```

```
CLIPS>
```

El elemento condicional objeto-patrón va seguido de las clases y ranuras con las que se va a comparar. Tras is-a y slot-name puede haber expresiones de restricción que incluyan ?, $?, & y |.

Además, se pueden especificar nombres de instancia para la concordancia de patrones. El siguiente ejemplo muestra cómo se compara una sola instancia de la clase DUCK utilizando la **restricción** de nombre, **name**, de instancias. Recuerda que *name* es una palabra reservada y no se puede utilizar como nombre de ranura.

```
CLIPS>
(defrule find-sound
?duck <- (object (is-a DUCK)
(sound $?find)
(name [Dorky_Duck]))
=>
(printout t "Duck "
(instance-name ?duck)
" says " ?find crlf))
CLIPS> (run)
Duck [Dorky_Duck] says (quack quack)
CLIPS>
```

Objetos de la base de datos

Analicemos el siguiente tipo general de problema. Dadas algunas instancias, ¿cuántas satisfacen una condición especificada? Por ejemplo, a continuación se muestran las defclasses y definstances de Joe's Home que muestran los distintos tipos de sensores y los aparatos conectados a ellos. Fíjate que se define una clase abstracta DEVICE ya que tanto SENSOR como APPLIANCE heredan ranuras comunes type y location.

```
CLIPS> (clear)
CLIPS>
(defclass DEVICE (is-a USER)
(role abstract)
; Sensor type
(slot type (access initialize-only))
; Location
(slot loc (access initialize-only)))
CLIPS>
(defclass SENSOR (is-a DEVICE)
```

```
(role concrete)
(slot reading)
; Min reading
(slot min (access initialize-only))
; Max reading
(slot max (access initialize-only))
; SEN. APP.
(slot app (access initialize-only)))
CLIPS>
(defclass APPLIANCE (is-a DEVICE)
(role concrete)
; Depends on appliance
(slot setting)
; off or on
(slot status))
CLIPS>
(definstances ENVIRONMENT_OBJECTS
(T1 of SENSOR
(type temperature)
(loc kitchen)
(reading 110) ; Too hot
(min 20)
(max 100)
(app FR))
(T2 of SENSOR
(type temperature)
(loc bedroom)
(reading 10) ; Too cold
(min 20)
(max 100)
(app FR))
(S1 of SENSOR
(type smoke)
(loc bedroom)
(reading nil) ; Bad sensor nil reading
(min 1000)
(max 5000)
(app SA))
(W1 of SENSOR (type water)
(loc basement)
(reading 0) ; OK
```

(min 0)
(max 0)
(app WP))
(FR of APPLIANCE
(type furnace)
(loc basement)
(setting low) ; low or high
(status on))
(WP of APPLIANCE
(type water_pump)
(loc basement)
(setting fixed)
(status off))
(SA of APPLIANCE
(type smoke_alarm)
(loc basement)
(setting fixed)
(status off)))
CLIPS>

Supongamos que se plantean las siguientes preguntas o consultas. ¿Cuáles son todos los objetos de la base de datos?

¿Cómo están dispuestos todos los objetos? ¿Cuáles son las relaciones entre los objetos? ¿Cuáles son todos los dispositivos? ¿Cuáles son todos los sensores? ¿Cuáles son todos los aparatos? ¿Qué sensor está conectado a qué aparato? ¿Hay algún sensor cuyo tipo sea temperatura? ¿Qué sensores de tipo temperatura tienen una lectura entre el mínimo y el máximo? Una consulta aún más básica es si *hay* o no sensores presentes.

El **sistema de consulta** de COOL es un conjunto de seis funciones que se pueden utilizar para la comparación de patrones de un conjunto de **instancias** y la realización de acciones. Un *instance-set* es un conjunto de instancias, como las instancias de SENSOR. Las instancias del instance-set pueden proceder de varias clases que no tienen por qué estar relacionadas. En otras palabras, las clases no tienen por qué pertenecer a la misma ruta de herencia.

La siguiente lista del *Manual de Referencia CLIPS*, resume las **funciones de consulta** predefinidas que pueden utilizarse para el acceso a conjuntos de instancias.

any-instancep: Determina si uno o varios conjuntos de instancias satisfacen una consulta.
find-instance: Devuelve el primer conjunto de instancias que satisface una consulta.
find-all-instances: Agrupa y devuelve todos los conjuntos de instancias que satisfacen una consulta.

do-for-instance: Realiza una acción para el primer conjunto de instancias que satisface una consulta.
do-for-all-instances: Realiza una acción para cada conjunto de instancias que satisface una consulta a medida que se encuentran.
delayed-do-for-all-instances: Agrupa todos los conjuntos de instancias que satisfacen una consulta y, a continuación, itera una acción sobre este grupo.

Me quedo con cualquiera

La función **any-instancep** es una función de predicado que devuelve TRUE si hay una instancia que coincida con el patrón y FALSE en caso contrario. A continuación, se muestra un ejemplo de esta función de consulta utilizada con las clases e instancias SENSOR y APPLIANCE. La función de consulta determina si existe *alguna* instancia en la clase SENSOR.

```
CLIPS> (reset)
CLIPS> (instances)
[T1] of SENSOR
[T2] of SENSOR
[S1] of SENSOR
[W1] of SENSOR
[FR] of APPLIANCE
[WP] of APPLIANCE
[SA] of APPLIANCE
For a total of 7 instances.
; Function returns TRUE because
; there is a SENSOR instance
CLIPS> (any-instancep ((?ins SENSOR)) TRUE)
TRUE
; Evaluation error—Bad!
; No DUCK class
CLIPS> (any-instancep ((?ins DUCK)) TRUE)
[PRNTUTIL1] Unable to find class DUCK.
CLIPS>
```

El formato básico de una función de consulta incluye una **instance-set-template** para especificar las instancias y sus clases, una **instance-set-query** como la condición booleana que deben satisfacer las instancias y **acciones** para especificar las acciones que se deben realizar. La función de predicado **class-existp** devuelve TRUE si la clase existe y FALSE en caso contrario.

La combinación de un nombre de instancia seguido de una o varias **restricciones de clase** se denomina **instance-set-member-template**. En general, las funciones de consulta pueden utilizarse como cualquier otra función de CLIPS.

Hay dos pasos para intentar satisfacer una consulta. En primer lugar, CLIPS genera todos los instance-sets posibles que concuerdan con la instance-set-template. En segundo lugar, se aplica la consulta booleana instance-set-query a todos los instance-sets para ver cuáles, si los hay, satisfacen la consulta. Los instance-sets se generan mediante una **permutación** simple de los miembros de una plantilla, variando primero los miembros situados más a la derecha. Ten en cuenta que una permutación no es lo mismo que una **combinación** porque el orden importa en una permutación, pero no en una combinación.

La función **find-all-instances** devuelve un valor multicampo de todas las instancias que satisfacen la consulta, o un valor multicampo vacío para ninguna. La función de consulta **do-for-instance** es similar a find-instance, salvo que realiza una única **acción distribuida** cuando se satisface la consulta.

La función **do-for-all-instances** es similar a do-for-instance excepto en que realiza sus acciones para cada conjunto de instancias que satisface la consulta.

Decisiones de diseño

A diferencia de las reglas, que sólo se activan cuando se satisfacen sus patrones, las deffunction se llaman explícitamente y luego se ejecutan. El hecho de que una regla se active no significa que se vaya a ejecutar. Las deffunction son de naturaleza completamente procedimental porque una vez llamadas por su nombre, su código se ejecuta de manera procedimental, sentencia por sentencia. Además, no se utiliza ningún patrón que implique restricciones en una deffunction para decidir si sus acciones deben ejecutarse. En su lugar, cualquier argumento que coincida con el número esperado por la lista de argumentos de la deffunction satisfará la deffunction y hará que se ejecuten sus acciones.

La idea básica de las deffunction como código procedimental con nombre se lleva a un grado mucho mayor con las **defgenerics** y las **defmethods** que describen su implementación. Una defgeneric es como una deffunction, pero mucho más potente porque puede realizar diferentes tareas dependiendo de las restricciones y tipos de sus argumentos. La capacidad de una función genérica para realizar diferentes acciones dependiendo de las clases de sus argumentos se denomina **sobrecarga** del nombre de la función.

Mediante el uso adecuado de la sobrecarga de operadores, es posible escribir código más legible y reutilizable. Por ejemplo, se puede definir una defgeneric para la función "+" con diferentes defmethods. La expresión,

```
(+ ?a ?b)
```

podría sumar dos números reales representados por ?a y ?b, o dos números complejos, o dos matrices, o concatenar dos cadenas, y así sucesivamente dependiendo de si hay una defmethod definida para los argumentos de las clases. CLIPS hace esto reconociendo primero el tipo de los argumentos y luego llamando a la defmethod apropiada definida para esos tipos. Se definiría una defmethod sobrecargada separado para "+" para cada conjunto de tipos de argumentos excepto para los tipos predefinidos del sistema como los números reales. Una vez definida la defgeneric, es fácil reutilizarla en otros programas.

Cualquier **función con nombre**, definida por el sistema o externa, puede sobrecargarse utilizando una función genérica. Ten en cuenta que una deffunction no puede sobrecargarse. Un uso apropiado de una función **genérica** es sobrecargar una función con nombre. Si la sobrecarga no es necesaria, deberías definir una deffunction o una función externa.

La sintaxis de defgenerics es muy simple, consiste sólo en el nombre legal de un símbolo CLIPS y un comentario opcional.

```
(defgeneric <name> [<comment>])
```

Como ejemplo sencillo de funciones genéricas, veamos el siguiente intento en CLIPS de comparar dos cadenas utilizando la función ">".

```
CLIPS> (clear)
CLIPS> (> "duck2" "duck1")
[ARGACCES5] Function > expected argument #1 to be of type integer or float
CLIPS>
```

No es posible realizar esta comparación con la función ">" porque se esperan tipos NUMBER como argumentos.

Sin embargo, es fácil definir una (defgeneric) que sobrecargará el ">" para aceptar tipos STRING, así como tipos NUMBER. Por ejemplo, si los argumentos de ">" son de tipo STRING, la defgeneric hará una comparación de cadenas, letra a letra empezando por la izquierda hasta que los códigos ASCII difieran. Por el contrario, si los argumentos de ">" son de tipo NÚMERO, el sistema compara el signo y la magnitud de los números. La función ">" definida por el usuario para los tipos STRING es un **método explícito**, mientras que una función externa definida por el sistema o por el usuario, como ">" para el tipo NUMBER, es un **método implícito**.

La técnica de sobrecargar el nombre de una función para que el método que la implementa no se conozca hasta el tiempo de ejecución es otro ejemplo de vinculación dinámica. Cualquier referencia de objeto de nombre o dirección también puede ser vinculada en tiempo de ejecución en CLIPS a funciones a través de la vinculación dinámica.

Algunos lenguajes como Ada tienen un tipo de sobrecarga más restrictivo en el que el nombre de la función debe conocerse en tiempo de compilación y no en tiempo de ejecución. La *vinculación dinámica* en tiempo de ejecución es la menos restrictiva ya que se pueden crear métodos durante la ejecución mediante la sentencia (build). Sin embargo, debes tener cuidado al utilizar (build) ya que la creación dinámica de construcciones suele ser difícil de depurar. Además, el código resultante puede ser difícil de verificar y validar ya que tendrás que detener la ejecución para examinar el código. La vinculación dinámica es una característica de un verdadero lenguaje de programación orientado a objetos.

A continuación, se muestra un ejemplo de una defgeneric, ">", para tipos STRING y su método.

```
; Header declaration. Actually unnecessary
CLIPS> (defgeneric >)
CLIPS>
(defmethod > ((?a STRING) (?b STRING))
(> (str-compare ?a ?b) 0))
; The overload ">" works correctly
; in all three cases.
CLIPS> (> "duck2" "duck1")
TRUE
CLIPS> (> "duck1" "duck1")
FALSE
CLIPS> (> "duck1" "duck2")
FALSE
CLIPS>
```

La (defgeneric) actúa como una **declaración de cabecera** para definir el tipo de función que se sobrecarga. En realidad, no es necesario usar una defgeneric en este caso porque CLIPS deduce implícitamente el nombre de la función del nombre de la defmethod, que es el primer símbolo después de "defmethod". La cabecera es una *declaración hacia adelante* la cual es necesaria si los métodos (defgeneric) aún no han sido definidos pero otro código, como defrule, defmessage-handlers, etc., sí hace referencia al nombre (defgeneric).

Otras características

Comparado con las deffunction, un método tiene un índice de **método** opcional. Si no proporcionas este índice, CLIPS proporcionará un número de índice único entre los métodos para esa función genérica, que puede verse mediante el comando **list-defmethods**. El **cuerpo del método** puede imprimirse usando el comando **ppdefmethod**. Un método puede eliminarse con una llamada a la función **undefmethod**.

El rango de los métodos determina la precedencia de **método** de una función genérica. La precedencia del método es la que determina el orden de listado de los métodos. Los métodos de mayor precedencia se listan antes que los de menor precedencia. CLIPS también intentará probar primero el método de mayor precedencia.

Un método **sombreado** es aquel en el que un método debe ser llamado por otro. El proceso mediante el cual CLIPS elige el método con mayor precedencia se denomina **envío genérico**. Para obtener más información, consulta el *Manual de referencia de CLIPS*.

Información de soporte

Preguntas e información

La URL de la página web de CLIPS es http://www.clipsrules.net.

Las preguntas relacionadas con CLIPS pueden enviarse a uno de los diversos foros en línea, como el CLIPS Expert System Group, http://groups.google.com/group/CLIPSESG/, los foros CLIPS de SourceForge, http://sourceforge.net/forum/?group_id=215471, y Stack Overflow, http://stackoverflow.com/questions/tagged/clips.

Las consultas relacionadas con el uso o la instalación de CLIPS pueden enviarse por correo electrónico a support@clipsrules.net.

Código fuente y ejecutables de CLIPS

Los ejecutables y el código fuente de CLIPS están disponibles en http://sourceforge.net/projects/clipsrules/files.

Documentación

Los manuales de referencia y la guía del usuario de CLIPS están disponibles en formato de documento portable (PDF) en http://www.clipsrules.net/?q=Documentation.

Expert Systems: Principles and Programming, 4ª edición, de Giarratano y Riley (ISBN

0-534-38447-1) incluye un CD-ROM con los ejecutables de CLIPS 6.22 (DOS, Windows XP y Mac OS), documentación y código fuente. La primera mitad del libro está orientada a la teoría y la segunda a la programación basada en reglas utilizando CLIPS. Está publicado por Course Technology.