Sistema CLIPS

(Apuntes)

Autor: Aitor San Juan Sánchez (aitorsj@hotmail.com)

Título: "Sistema CLIPS (Apuntes)"

Área: Sistemas Basados en el Conocimiento (I.A.)

N.º Págs.: 100

Prólogo

Este documento pretende ser una herramienta base de trabajo y de consulta. No intenta sustituir al manual, pero sí está pensado para no hacerle imprescindible. Por otra parte, se suponen conocimientos básicos sobre sistemas de mantenimiento de la verdad, sistemas de producción y sistemas basados en objetos estructurados (*frames*).

Se pueden distinguir tres partes en este documento:

I. Introducción y programación básica en CLIPS [*cap. 1*, *2 y 3*],
II. Control de la ejecución (desarrollo modular y variables globales) [*cap. 4 y 5*], y
III. Programación orientada a objetos en un sistema de producción: CLIPS [*cap. 6 y 7*, *principalmente el capítulo 6*].

Así mismo, se incluyen varios apéndices que tratan temas adicionales sobre CLIPS.

Convenciones tipográficas

cursiva	Indica extranjerismos, así como símbolos, signos o conjuntos de signos que no constituyan vocablos. Ej.: "... la construcción *deftemplate* ..."
`fuente fija`	Código en CLIPS tal y como se tecletaría (Nota: ignórense los signos de acentuación y la letra *ñ*, ya que CLIPS no los reconocerá). Ej.: `(printout t "ERROR" crlf)`

Bibliografía

- *CLIPS User's Guide*: ofrece una introducción a CLIPS. Se compone de 2 volúmenes:

  Volume I: Rules, introducción a la programación basada en reglas usando CLIPS.

  Volume II: Objects, introducción a la programación orientada a objetos usando COOL (parte de CLIPS para la programación orientada a objetos).

- *CLIPS Reference Manual*: consta de 3 volúmenes, de los cuales se han utilizado el I y II.

  Volume I: The Basic Programming Guide, descripción de la sintaxis de CLIPS y ejemplos de uso.

  Volume II: Advanced Programming Guide, describe en detalle las características más sofisticadas de CLIPS para programación de aplicaciones avanzadas.

  NASA Lyndon B. Johnson Space Center, 1993

- *Expert Systems: Principles and Programming*
  Joseph Giarratano & Gary Riley. 2nd Edition.
  PWS Publishing Company, 1994

ÍNDICE

Página

1. INTRODUCCIÓN 5

1.1 ¿QUÉ ES CLIPS? 5
1.2 REPRESENTACIÓN DEL CONOCIMIENTO 5
1.3 ESTRUCTURA DE UN PROGRAMA EN CLIPS 6

2. PROGRAMACIÓN BÁSICA EN CLIPS 6

2.1 MEMORIA DE TRABAJO (MT) 6
2.1.1 Estructura: lista de hechos y lista de instancias 6
2.1.2 Elementos básicos de programación: tipos de datos, funciones y constructores 6
2.1.3 Abstracción de datos: hechos, objetos y variables globales 8
2.1.3.1 Hechos: ordenados y no ordenados. Constructor deftemplate. Atributos. 8
2.1.3.2 Objetos 13
2.2 BASE DE CONOCIMIENTO (BC) 13
2.2.1 Estructura de las reglas: construcción *defrule* 13
2.2.2 Elementos condicionales (EC): tipos. Variables. Operadores lógicos. Predicados. 14
2.2.2.1 Tipos de elementos condicionales: pattern, test, or, and, not, exists, forall, logical. 14
2.2.2.2 EC's que "obtienen" la dirección de elementos de la MT 19
2.2.3 Comandos (acciones) predefinidos 20
2.2.4 Funciones predefinidas: de E/S, matemáticas, de conversión, de *strings*, de manejo de valores multicampo y funciones de entorno. Nombres lógicos predefinidos. 22
2.2.5 Funciones definidas por el usuario: constructor *deffunction* 32
2.2.6 Propiedades de las reglas: *salience* y *auto-focus* 32
2.3 MOTOR DE INFERENCIA 33
2.3.1 Ciclo básico 33
2.3.2 Filtrado (*match*) 34
2.3.3 Resolución de conflictos: estrategias (*select*) 36
2.3.4 Ejecución (*act*) 38

3. EJECUCIÓN DE PROGRAMAS 38

3.1 EL ENTORNO DE TRABAJO 38
3.2 INICIALIZACIÓN DE LA MT 38

4. CONTROL DE LA EJECUCIÓN 39

4.1 INTRODUCCIÓN 39
4.2 DISEÑO MODULAR: CONSTRUCCIÓN DEFMODULE 39
4.2.1 Importación y exportación. 41
4.3 MÓDULOS Y CONTROL DE LA EJECUCIÓN 42
4.3.1 El comando *focus*. 42
4.3.2 El comando *return*. 43

5. VARIABLES GLOBALES: CONSTRUCCIÓN *DEFGLOBAL* 46

6. PROGRAMACIÓN ORIENTADA A OBJETOS EN CLIPS: COOL 48

6.1 INTRODUCCIÓN. ¿QUÉ ES COOL? 48
6.2 CLASES PREDEFINIDAS POR EL SISTEMA 49
6.3 DEFINICIÓN DE CLASES 51
6.3.1 Construcción *defclass* 51
6.3.2 Herencia múltiple: reglas 51
6.3.3 Especificadores de clase 53

6.3.4 Atributos (*slots*) y propiedades 53
6.3.5 Proceso de *pattern-matching* con objetos 59
6.3.6 Documentación de gestores de mensajes 60
6.4 GESTORES DE MENSAJES (MESSAGE HANDLERS): CREACIÓN. 61
6.4.1 Parámetros 62
6.4.2 Acciones 63
6.4.3 Demonios (*daemons*) 64
6.4.4 Gestores de mensajes predefinidos 64
6.5 DISPATCHING 66
6.6 MANIPULACIÓN DE INSTANCIAS: CREACIÓN, MODIFICACIÓN Y BORRADO 67
6.7 MANIPULACIÓN DE CONJUNTOS DE INSTANCIAS: CONSULTAS. OPERACIONES. 71

7. FUNCIONES GENÉRICAS 75

7.1 CARACTERÍSTICAS. 75
7.2 CONSTRUCCIÓN DEFGENERIC: CABECERAS. 75
7.3 MÉTODOS: CONSTRUCCIÓN DEFMETHOD. 76
7.3.1 Tipos de parámetros de un método: simples o múltiples. 77
7.4 PROCESO DE DISPATCH GENÉRICO: APLICABILIDAD Y REGLAS DE PRECEDENCIA. 78
7.5 MÉTODOS "OSCURECIDOS": TÉCNICAS DECLARATIVA E IMPERATIVA 79
7.6 ERRORES DE EJECUCIÓN EN MÉTODOS 79
7.7 VALOR DEVUELTO POR UNA FUNCIÓN GENÉRICA 80

APÉNDICE A: ESPECIFICACIÓN BNF DE CLIPS 81

APÉNDICE B: ENCADENAMIENTO HACIA ATRÁS EN CLIPS 90

APÉNDICE C: CONSTRUCCIÓN DE UN PROGRAMA EJECUTABLE A PARTIR DEL CÓDIGO FUENTE CLIPS 93

APÉNDICE D: INTERFAZ CON C, ADA Y VISUAL BASIC 95

INTERFAZ CON C 95
INTERFAZ CON ADA 95
INTERFAZ CON VISUAL BASIC 99

APÉNDICE E: SITIOS WEB INTERESANTES SOBRE LA IA Y CLIPS 100

CLIPS 6.0

1. INTRODUCCIÓN

1.1 ¿Qué es CLIPS?

CLIPS *(C Language Integrated Production System)* es una herramienta para el desarrollo de sistemas expertos (SE) creada por la *Software Technology Branch (STB), NASA/Lyndon B. Johnson Space Center*. Los orígenes de CLIPS se remontan a 1984.

Se diseñó para facilitar el desarrollo de software que modele el conocimiento humano *(expertise)*:

- Con propósitos específicos: alta portabilidad, bajo coste, y facilidad de integración.
- CLIPS permite integración completa con otros lenguajes de programación como C o Ada.
- Puede ser llamado desde un lenguaje procedural, realizando su función y devolver el control al programa que le llamó.
- También se puede definir código procedural como funciones externas llamadas desde CLIPS. Cuando el código externo finaliza su ejecución devuelve el control a CLIPS.

CLIPS es un entorno completo para la construcción de SE basados en reglas y/o objetos. La versión estándar de CLIPS proporciona un entorno de desarrollo interactivo orientado a texto, incluyendo una herramienta de depuración, ayuda *on-line* y un editor integrado, aunque se han desarrollado interfaces visuales para plataformas Macintosh, Windows 3.*x* y el sistema X Window.

CLIPS distingue mayúsculas y minúsculas (*case-sensitive*), igual que el lenguaje *C*.

1.2 Representación del conocimiento

CLIPS ofrece paradigmas heurísticos y procedurales para representar el conocimiento.

A) Conocimiento heurístico: reglas

- Las reglas se utilizan para representar heurísticos que especifican un conjunto de acciones a realizar para una situación dada.
- El creador del SE define una colección de reglas que, en conjunto, resuelven un problema.
- Se puede pensar que las reglas son como sentencias IF-THEN de lenguajes procedurales como *C* o Ada. Sin embargo, las reglas actúan más bien como sentencias SIEMPRE QUE-ENTONCES.

B) Conocimiento procedural: funciones y objetos.

Este tipo de conocimiento se expresa mediante funciones definidas por el usuario (*deffunctions*), funciones genéricas y la programación orientada a objetos (POO). Ésta en CLIPS soporta las cinco características generalmente aceptadas: clases, mensajes, abstracción, encapsulamiento, herencia y polimorfismo.

Es posible desarrollar software utilizando sólo reglas, sólo objetos, o una mezcla de ambos.

1.3 Estructura de un programa en CLIPS

El *shell* (parte de CLIPS que realiza inferencias o razonamiento) provee los elementos básicos de un SE:

1. **memoria global de datos** (memoria de trabajo, MT): conocimiento factual (*fact-list* e *instance-list*).
2. **base de conocimiento** *(knowledge base)*: contiene las reglas de la base de reglas.
3. **motor de inferencia** *(inference engine)*: controla la ejecución global de las reglas: decide qué reglas deben ejecutarse y cuándo.

Un programa escrito en CLIPS puede consistir en reglas, hechos y objetos.

Un SE basado en reglas escrito en CLIPS es un programa dirigido por los datos *(data driven)*, es decir, hechos y objetos. Las reglas pueden *matchear* con objetos y hechos, aunque los objetos pueden usarse por sí solos (mediante el envío de mensajes) sin utilizar el motor de inferencia.

2. PROGRAMACIÓN BÁSICA EN CLIPS

2.1 MEMORIA DE TRABAJO (MT)

2.1.1 Estructura: lista de hechos y lista de instancias

Todos los elementos de la memoria de trabajo (MT) son entidades que corresponden o bien a hechos o bien a instancias de una clase de objetos.

La MT consta de una lista de hechos *(fact-list)* y de una lista de instancias *(instance-list)*.

Un hecho es una forma básica de alto nivel para representar información. Es la unidad de datos fundamental utilizada en las reglas.

Un hecho se compone de una serie de campos. Un campo es un lugar, con o sin nombre, que puede llevar asociado un valor.

2.1.2 Elementos básicos de programación: tipos de datos, funciones y constructores

CLIPS proporciona tres elementos básicos para escribir programas:

- tipos primitivos de datos: para representar información.
- funciones: para manipular los datos.
- constructores: para añadir conocimiento a la BC.

Tipos primitivos de datos también llamados valores de un único campo (*single-field values*)

INFORMACIÓN SIMBÓLICA

- Símbolos (`SYMBOL`): cualquier secuencia de caracteres ASCII imprimibles.
- Cadenas de caracteres (`STRING`): un conjunto de cero o más caracteres ASCII imprimibles encerrados entre comillas dobles (`"`).

INFORMACIÓN NUMÉRICA
- Enteros (`INTEGER`).
- Coma flotante (`FLOAT`).

DIRECCIONES (de la MT cada elemento de la MT tiene su dirección)
- externa (`EXTERNAL-ADDRESS`): dirección de una estructura de datos externa devuelta por alguna función (escrita en otro lenguaje, como *C* o Ada) que ha sido integrada con CLIPS.
- de hechos (`FACT-ADDRESS`).
- de instancias (`INSTANCE-ADDRESS`).

Funciones.

Una función es un fragmento de código ejecutable (identificado por un nombre) que devuelve un valor o que tiene un efecto lateral útil. Distinguiremos ambos tipos de funciones denominando comandos o acciones a aquellas funciones que no devuelven valores, pero que generalmente tendrán algún efecto lateral útil.

Hay varios tipos de funciones:

a) funciones (predefinidas) del sistema: definidas internamente por el entorno de CLIPS.

b) funciones definidas por el usuario. Se distinguen:
- funciones externas: escritas en otro lenguaje (*C*, Ada) y *linkadas* con el entorno de CLIPS.
- funciones definidas directamente en CLIPS utilizando sintaxis de CLIPS (*deffunctions*).
- funciones genéricas: son funciones que permiten ejecutar diferentes fragmentos de código dependiendo del tipo y número de parámetros.

Las llamadas a funciones usan notación prefija: los argumentos de una función aparecen después del nombre de ésta y todo ello entre paréntesis (al estilo de LISP).

NOTA: mientras que una función se refiere al fragmento de código ejecutable, llamaremos expresión a una función que tiene especificados sus parámetros (que pueden ser o no funciones también).

Ej.: `(+ 3 (* 8 9) 4)`
`(* 8 9)` = expresión

Constructores.

Son estructuras sintácticas identificadas por una palabra reservada del lenguaje que permiten definir funciones, reglas, hechos, clases, etc., que alteran el entorno de CLIPS añadiéndolas a la base de conocimiento. Los constructores no devuelven ningún valor. Su sintaxis también es similar a la definición de funciones.

Comentarios.

CLIPS también permite comentar el código. Todos los constructores (excepto uno llamado *defglobal*) permiten incorporar en su definición un comentario directamente entre comillas (`"`). En las demás partes, los comentarios pueden intercalarse con el código usando el punto y coma (`;`) (igual que en LISP).

2.1.3 Abstracción de datos: hechos, objetos y variables globales

Existen tres formas de representar la información en CLIPS: hechos, objetos y variables globales (estas últimas se describen con más detalle en la sección 5).

2.1.3.1 Hechos: ordenados y no ordenados. Constructor *deftemplate*. Atributos.

Un **hecho** es una lista de valores atómicos que pueden ser referenciados por la posición que ocupan dentro del hecho (hechos ordenados) o bien por un nombre (hechos no ordenados). La forma de acceder a un hecho de la MT es mediante un índice o dirección (*fact index*).

Un hecho puede ser asertado (añadido) a la MT, eliminado, modificado o duplicado:
- explícitamente por el usuario (mediante comandos), ó
- mediante un programa CLIPS.

HECHOS ORDENADOS

Secuencia de cero o más campos separados por espacios y delimitados por paréntesis.

- El campo inicial suele expresar una relación entre los campos siguientes.

  Ej.: `(altitud es 1000 metros)`

  `(lista-de-la-compra pan leche arroz)`

- Los campos de un hecho ordenado pueden ser de cualquier tipo primitivo de datos, excepto el primero, que debe ser un símbolo.
- No es necesario declararlos.
- No existe restricción alguna en el orden de los campos, pero
- Para *matchear* con una regla sus campos deben aparecer en el mismo orden que indique la regla. Es decir, los hechos ordenados "codifican" la información según la posición. Para acceder a esa información, el usuario debe saber la información que almacena el hecho y qué campo la contiene.

HECHOS NO ORDENADOS

- Los hechos no ordenados proporcionan al usuario la habilidad de abstraerse de la estructura del hecho, asignando un nombre a cada campo del mismo De esta forma, podemos acceder a los diferentes campos por su nombre.

Un hecho no ordenado es una secuencia de cero o más campos con nombre separados por espacios y delimitados por paréntesis.

El constructor `deftemplate` crea una plantilla o patrón que se usa para acceder, por su nombre, a los campos (*slots*) de un hecho no ordenado. Este constructor es análogo a la definición de un registro en lenguajes como Ada o las estructuras de C.

Sintaxis: `(deftemplate <nombre> [<comentario>] <definición-slot>*)`

Ejemplo: El problema de la asignación de habitaciones.

Hay 4 clases de habitaciones: simples, dobles, triples, y cuádruples.
Es más económico llenar las habitaciones más grandes.
Los estudiantes de una habitación deben ser del mismo sexo.
Los ocupantes de una habitación deben ser todos fumadores o todos no fumadores.

Nombre	**Atributos**	**Nombre**	**Atributos**
Estudiante	Nombre Sexo Fuma? Alojado	Habitación	Número Capacidad Sexos Fuman? Plazas-libres Ocupantes

Ej.:
```
(deftemplate habitación
      (slot número)
      (slot capacidad)
      (slot sexos)
      (slot fuman?)
      (slot plazas-libres)
      (multislot ocupantes))
```

Ej.:
```
(deftemplate estudiante
      (slot nombre)
      (slot sexo)
      (slot fuma?)
      (slot alojado))
```

- Los hechos no ordenados se distinguen de los ordenados mediante el primer campo. El primer campo de cualquier hecho debe ser un símbolo, pero si éste coincide con el nombre de un constructor *deftemplate*, entonces se considera no ordenado.

- No importa el orden en el que se referencia el campo y/o el orden en el que aparezcan los campos.

  Ej.: Los siguientes hechos son el mismo:

```
(habitación (número 23)(capacidad 4)(sexos femenino)
            (fuman? no)(plazas-libres 2)(ocupantes María Jone))

(habitación (capacidad 4)(sexos femenino)(número 23)
            (plazas-libres 2)(ocupantes María Jone)(fuman? no))
```

Además, podemos restringir el tipo, valor y rango numérico, entre otros, de los *slots* de un hecho no ordenado: son los atributos de los *slots*.

ATRIBUTOS DE LOS *SLOTS* *(campos)*

Se puede restringir su tipo, valor, rango numérico y la cardinalidad (el número mínimo y máximo de valores para un *slot*); se pueden especificar valores por defecto. Todas estas características ayudan en el desarrollo y mantenimiento de un SE proporcionando un fuerte tipado y comprobación de restricciones.

Atributo *type* `(type <especificación-tipo>)`

Define el tipo de datos que puede tener el *slot*.
Los tipos válidos son: `SYMBOL`, `STRING`, `LEXEME`, `INTEGER`, `FLOAT`, `NUMBER`. Si se especifica `?VARIABLE`, significa que el *slot* puede tomar cualquier tipo de dato (por defecto, se supone este). Si se especifican uno ó más tipos válidos, el tipo del *slot* queda restringido a uno de los tipos especificados. `LEXEME` equivale a especificar `SYMBOL` y `STRING` conjuntamente, y `NUMBER` equivale a `INTEGER` y `FLOAT`.

Ej.:
```
(deftemplate persona
      (multislot nombre (type SYMBOL))
      (slot edad (type INTEGER)))
```

Atributo *allowed-*

Especifica los valores concretos permitidos para un tipo específico.

Ej.:
```
(deftemplate persona
      (multislot (nombre (type SYMBOL))
      (slot edad (type INTEGER))
      (slot sexo (type SYMBOL)(allowed-symbols hombre mujer)))
```

Existen siete atributos de esta clase: `allowed-symbols`, `allowed-strings`, `allowed-lexemes`, `allowed-integers`, `allowed-floats`, `allowed-numbers` y `allowed-values`. Cada uno de estos atributos debería ser seguido por `?VARIABLE` (que indica que cualquier valor del tipo especificado es legal) ó por una lista de valores del tipo que sigue al prefijo *allowed-*. El atributo por defecto es:
```
(allowed-values ?VARIABLE)
```

Nótese que estos atributos *allowed-* no restringen el tipo del *slot*. En el ejemplo previo, el *allowed-symbols* no restringe el tipo del *slot* `sexo` a ser un símbolo. Sólo indica que si el valor del *slot* es un símbolo, entonces debe ser el símbolo `hombre` ó el símbolo `mujer`. En el ejemplo previo, cualquier *string*, entero o real sería legal para el *slot sexo* si no se especificara `(type SYMBOL)`.

Atributo *range* `(range <límite-inferior> <límite-superior>)`

Permite restringir los valores legales de un tipo numérico a un rango determinado. Tanto el límite inferior como el límite superior pueden ser un valor numérico o `?VARIABLE`.

Ej.: `(range ?VARIABLE 3)` representa el rango -∞ .. 3
`(range 14 ?VARIABLE)` representa el rango 14 .. +∞
`(range ?VARIABLE ?VARIABLE)` representa el rango -∞ .. +∞ (este el rango por defecto si no se especifica el atributo `range`)
Al igual que los atributos *allowed-*, el atributo *range* sólo restringe los valores numéricos legales de un *slot* a los del rango especificado, si el valor del *slot* es numérico.

Ej.:
```
(deftemplate persona
      (multislot nombre (type SYMBOL))
      (slot edad (type INTEGER)(range 0 125)))
```

Atributo *cardinality* `(cardinality <límite-inferior> <límite-superior>)`

Permite especificar el número mínimo y máximo que un *slot* puede contener. Ambos límites pueden ser un entero positivo ó `?VARIABLE`, que indica que no hay número mínimo o máximo de valores que el *slot* puede contener (dependiendo si se especifica como limite inferior, como superior, o en ambos). Por defecto se supone `?VARIABLE` para ambos límites.

Ej.: El siguiente *deftemplate* representa un equipo de voleibol que debe tener 6 jugadores y puede tener hasta 2 jugadores alternativos.
```
(deftemplate equipo-voleibol
      (slot nombre-equipo (type STRING))
      (multislot jugadores (type STRING) (cardinality 6 6))
      (multislot alternativos (type STRING) (cardinality 0 2)))
```

Atributo *default* `(default <especificación>)`

Permite especificar un valor por defecto para un *slot* cuando no se añade o especifica explícitamente (por ejemplo, en un comando *assert*). La <especificación> puede ser: `?DERIVE`, `?NONE` ó una expresión simple (si se trata de un *slot* simple), ó cero o más expresiones (si se trata de un *slot* multicampo, es decir, un *multislot*).

- Si se especifica `?DERIVE`, entonces se deriva un valor para el *slot* que satisfaga todos los atributos del *slot*. Si en un *slot* no se especifica nada acerca de *default*, se supondrá `(default ?DERIVE)`. En caso de un *slot* simple, esto significa que se elige un valor que satisfaga el tipo, rango y los valores permitidos para el *slot* (según indique el atributo *allowed-*). En el caso de un *slot* multicampo, el valor por defecto será una lista de valores idénticos que es la cardinalidad mínima permitida para el *slot* (cero por defecto). Si aparecen uno o más valores en el atributo *default* de un *slot* multicampo, entonces cada valor satisfará el tipo, rango y los atributos *allowed-* del *slot*.

  Ej.:
  ```
  (deftemplate ejemplo
        (slot a) ;;; el tipo se supondrá SYMBOL, y el valor por defecto será nil
        (slot b (type INTEGER)) ;;; el valor por defecto para INTEGER es 0
        (slot c (allowed-values rojo verde azul))
        (multislot d) ;;; por defecto contendrá el valor ()
        (multislot e (cardinality 2 2)
                     (type FLOAT)
                     (range 3.5 10.0)))
  ```
  Al introducir un hecho en la MT sin dar ningún valor a ninguno de sus *slots*, el hecho realmente contendría lo siguiente:
  ```
  (ejemplo (a nil) (b 0) (c rojo) (d) (e 3.5 3.5))
  ```

- Si se indica `?NONE`, entonces se debe dar un valor obligatoriamente al *slot* cuando se aserte el hecho. Es decir, no hay valor por defecto. Si no se le da un valor, se producirá un error.
- Si se utiliza una o más expresiones en el atributo *default*, entonces se evalúan las expresiones y el valor resultante se almacena en el *slot* siempre que no se especifique un

valor para ese *slot* en un comando *assert*. En el caso de un *slot* simple, la especificación del atributo *default* debe ser exactamente una sola expresión. Si no se especifica expresión alguna cuando se trata de un *slot* multicampo, se usará un multicampo de longitud cero para el valor por defecto. En caso contrario, el *slot* multicampo contendrá cada uno de los valores devueltos por las respectivas expresiones.

Ej.:
```
(deftemplate ejemplo
     (slot a (default 3))
     (slot b (default (+ 3 4)))
     (multislot c (default a b c))
     (multislot d (default (+ 1 2) (+ 3 4))))
```

Al introducir un hecho en la MT sin especificar un valor para cada uno de los *slots*, el hecho realmente contendrá lo siguiente por defecto:

```
(ejemplo (a 3) (b 7) (c a b c) (d 3 7))
```

Ej.:
```
(deftemplate objeto
     (slot nombre
          (type SYMBOL)
          (default ?DERIVE))
     (slot localización
          (type SYMBOL)
          (default ?DERIVE))
     (slot sobre
          (type SYMBOL)
          (default suelo))
     (slot peso
          (allowed-values ligero pesado)
          (default ligero))
     (slot contenidos
          (type SYMBOL)
          (default ?DERIVE)))
```

Definición de hechos iniciales.

El constructor `deffacts` permite especificar un conjunto de hechos como conocimiento inicial.

Sintaxis:
```
(deffacts <nombre-colección-hechos> [<comentario>]
     <patrón-RHS>*)
```

Ej.:
```
(deffacts arranque "Estado inicial del frigorífico"
     (frigorífico interruptor encendido)
     (frigorífico puerta abierta)
     (frigorífico temperatura (get-temp)))
```

Ej.:
```
(deffacts estudiante "Todos los estudiantes iniciales"
     (estudiante (nombre Juan)(sexo varón)(fuma? no)(alojado no))
     (estudiante (nombre Pepe)(sexo varón)(fuma? si)(alojado no))
     (estudiante (nombre Luisa)(sexo mujer)(fuma? no)(alojado no))
     (estudiante (nombre Pedro)(sexo varón)(fuma? no)(alojado no)))
```

Los hechos de las sentencias `deffacts` son añadidas a la MT utilizando el comando **reset**. El comando *reset* elimina todos los hechos que hubiera en la lista de hechos actual, y a continuación añade los hechos correspondientes a sentencias `deffacts`.

Sintaxis: `(reset)`

Un programa CLIPS puede ser ejecutado mediante el comando **run**. Pero puesto que las reglas requieren de hechos para ejecutarse, el comando *reset* es el método clave para iniciar o reiniciar un sistema experto en CLIPS. Así, el comando *reset* provoca la activación de algunas reglas, y el comando *run* inicia la ejecución del programa.

2.1.3.2 Objetos

Una instancia es una instanciación o ejemplo específico de una clase (que representa un conjunto de objetos con las mismas propiedades). En CLIPS un objeto puede ser cualquier valor de un tipo primitivo de datos (un entero, un *string*, un símbolo, una dirección externa, etc.) o una instancia de una clase definida por el usuario.

Los objetos vienen descritos por sus propiedades y su conducta.

Clase: patrón para propiedades comunes y conducta de los objetos que son instancias.

Los objetos se dividen en dos categorías: tipos primitivos e instancias de clases definidas por el usuario. Estos dos tipos difieren en la forma en que se referencian, en cómo se crean y se borran, y en cómo se especifican sus propiedades.

⇨ Diferencia hecho no ordenado y objeto: herencia.

Herencia: permite definir las propiedades y conducta de una clase en función de otras clases.

⇨ Los hechos **no** tienen herencia.

2.2 BASE DE CONOCIMIENTO (BC)

2.2.1 Estructura de las reglas: construcción *defrule*

- Una regla consta de un antecedente -también denominado parte "si" o parte izquierda de la regla (LHS)[1]- y de un consecuente -también denominado parte "entonces" o parte derecha de la regla (RHS)[2].

- El antecedente está formado por un conjunto de condiciones -también denominadas elementos condicionales (EC)- que deben satisfacerse para que la regla sea aplicable. (Existe un *and* implícito entre todas las condiciones en la parte izquierda de la regla).

  ¿Cómo se satisfacen los EC de una regla?

  ⇨ La satisfactibilidad de un EC se basa en la existencia o no existencia en la MT de los hechos especificados o las instancias de clases definidas por el usuario en la regla.

[1] Iniciales de *Left-Hand Side*.
[2] Iniciales de *Right-Hand Side*.

- El consecuente de una regla es un conjunto de acciones a ser ejecutadas cuando la regla es aplicable. Estas acciones se ejecutan cuando el motor de inferencia de CLIPS es instruido para que comience la ejecución de las reglas aplicables.

Una regla CLIPS es una entidad independiente: no es posible el paso de datos entre dos reglas.

Sintaxis:
```
(defrule <nombre-regla> [<comentario>]
     [<declaración>]
     <elemento-condición>* ; Parte izquierda (LHS)
     =>
     <acción>*)  ; Parte dcha. (RHS) de la regla
```

Ej.:
```
(defrule regla-ejemplo "Ejemplo de regla"
     (frigorífico interruptor encendido)
     (frigorífico puerta abierta)
     =>
     (assert (frigorífico comida estropeada)))
```

- Si se introduce en la base de reglas una nueva regla con el mismo nombre que el de una existente, la nueva regla reemplazará a la antigua.

- El comentario, que debe ser un *string*, se usa normalmente para describir el propósito de la regla o cualquier otra información que el programador desee. Estos comentarios pueden ser visualizados junto con el resto de la regla usando el comando *ppdefrule*:

  Sintaxis: `(ppdefrule <nombre-regla>)`

- Si una regla no tiene parte izquierda, es decir, no tiene elementos condicionales, entonces el hecho `(initial-fact)` actuará como el elemento condicional para ese tipo de reglas, y la regla se activará cada vez que se ejecute un comando **reset**.

2.2.2 Elementos condicionales (EC): tipos. Variables. Operadores lógicos. Predicados.

Elemento condicional (EC): especifica restricciones sobre elementos de las listas de hechos e instancias: sólo se satisface si existe una entidad (hecho o instancia) que cumple las restricciones expresadas.

Una regla se ejecuta cuando:

1. todos sus elementos condición son satisfechos por la lista de hechos y/o la lista de instancias.
2. el motor de inferencia la selecciona.

2.2.2.1 Tipos de elementos condicionales: *pattern*, *test*, *or*, *and*, *not*, *exists*, *forall*, *logical*.

pattern

Colección de restricciones de campos, comodines, y variables que se usan para restringir el conjunto de hechos o instancias que satisfacen el *pattern*.

Ej.: Con restricciones literales.

```
(altitud es 1000 metros)
```
← hecho ordenado

```
(habitación (número 23)(plazas-libres 3))
```

Ej.: Con comodines simples y multicampo.

comodín para un solo campo ↓

```
(altitud es ? metros)

(habitación (número ?)(plazas-libres 3))

(habitación (número ?)(plazas-libres 3)(ocupantes $?))
```

comodín multivalor ↗

Ej.: Con variables simples y multicampo.

```
(altitud es ?x metros)

(habitación (número ?y)(plazas-libres 3))

(habitación (número ?y)(plazas-libres 3)(ocupantes $?z))
```

Variables

Sintaxis: `?<símbolo>`

No se declaran.
El ámbito de una variable es la regla donde se utiliza.
Se instancia la primera vez que aparece en una regla y mantiene su valor instanciado en sucesivas apariciones (dentro de la misma regla).
Se utilizan para relacionar diferentes entidades de las listas de hechos o instancias.

Ejercicio: Obtener el número de plazas libres de la habitación ocupada por Abilio.

Ej.: EC con operadores lógicos.

```
(habitación (plazas-libres ~0))

(habitación (plazas-libres 1|2|3))

(habitación (plazas-libres ~0 & ~4))

(habitación (plazas-libres ?p & ~0))

(habitación (plazas-libres ?p & 1|2))
```

Operadores lógicos:

Negación: ~
Disyunción: |
Conjunción: &

Ej.: EC con predicados o llamadas a funciones. En este caso deben ir precedidos del signo '`:`'

```
(habitación (capacidad ?c) (plazas-libres ?p & : (> ?c ?y)))

(datos $?x & : (> (length ?x) 2))
```

Predicados:

- De tipo (todos terminan en *p*): *numberp*, *floatp*, *symbolp*, ...
- Comparaciones numéricas: `=`, `<>`, `<`, `<=`, `>`, `>=`
- Igualdad (desigualdad) en tipo y valor: `eq` (`neq`)
- Predicados definidos por el usuario.

test `(test <llamada-a-función>)`

Se usa para evaluar expresiones en la parte izquierda de una regla, interviniendo en el proceso de *pattern-matching*.
El EC *test* se satisface si la llamada a la función que aparezca dentro de él devuelve cualquier valor distinto de `FALSE`; en caso contrario, este EC no se satisface.

Ej.:
```
(defrule ejemplo-test
     (datos ?x)
     (valor ?y)
     (test (>= (abs (- ?y ?x)) 3))
     =>)
```

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

Este EC se satisface cuando al menos uno de los componentes que aparece se satisface.

Ej.:
```
(defrule fallo-del-sistema
     (error-status desconocido)
     (or (temperatura alta)(válvula rota)
          (bomba (estado apagada)))
     =>
     (printout t "El sistema ha fallado." crlf))
```

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

Recuérdese que CLIPS supone que todas las reglas tienen un *and* implícito que rodea todos los elementos condicionales de la LHS. Esto significa que todos los EC que aparezcan en la LHS de la regla deben satisfacerse para que la regla se active.

¿Para qué sirve entonces el EC *and* explícito? → Se proporciona para permitir la combinación de *and*'s y *or*'s. Un EC *and* se satisface si todos los EC's que contenga se satisfacen.

Ej.:
```
(defrule fallo-1
     (error-status confirmado)
     (or (and (temperatura alta)(válvula cerrada))
         (and (temperatura baja)(válvula abierta)))
     =>
     (printout t "El sistema tiene el fallo 1." crlf))
```

not `(not <elemento-condicional>)`

Un elemento condicional negativo se satisface si no existe ninguna entidad que cumpla las restricciones expresadas.

Ej.:
```
(not (habitación (plazas-libres ?p & ~0)
     (capacidad ?c & : (> ?c ?p))))
```

Ejercicio: Determinar si no existe ninguna habitación vacía.

```
(not (habitación (capacidad ?c) (plazas-libres ?c))
```

exists `(exists <elemento-condicional>+)`

Este EC permite que se produzca el *pattern matching* cuando al menos exista un hecho que satisfaga la regla, sin tener en cuenta el número total de hechos que pudiesen *matchear*. Esto permite una sola activación para una regla con la que *matcheen* un conjunto de hechos.

El EC *exists* está implementado mediante una combinación de *and*'s y *not*'s. Los EC's dentro de un *exists* se incluyen dentro de un *and* y luego dentro de dos *not*.

Ej.: `(exists (emergencia)) ≡ (and (not (not (and (emergencia)))))`

forall `(forall <primer-EC> <resto-de-ECs>+)`

Permite el *matching* basado en un conjunto de EC's que son satisfechos por cada ocurrencia de otro EC. Su funcionamiento es el contrario que el de *exists*. Para que el EC *forall* se satisfaga, todo lo que *matchee* con `<primer-EC>` debe tener hechos que *matcheen* todos los demás EC que aparecen a continuación de `<primer-EC>`.

Este EC también puede expresarse en función de una combinación de *and*'s y *not*'s:

```
(not (and <primer-EC> (not (and <resto-de-ECs>+))))
```

logical `(logical <elemento-condicional>+)`

Este EC proporciona capacidad de mantenimiento de verdad para entidades (hechos o instancias) creados por una regla que usa el EC *logical*.

- Este EC permite establecer una dependencia lógica entre entidades creadas en las RHS (o como resultado de las acciones realizadas en la RHS) y las entidades que *matchearon* los patrones contenidos en el EC *logical* de la LHS de las reglas. Las entidades que *matcheen* con los patrones contenidos en el EC *logical* proporcionan un soporte lógico a los hechos e instancias que se crearon en la RHS de las reglas.
- **Si una entidad que justifique a otra desaparece, esta última también se elimina.**

Ejemplo: Supongamos las siguientes reglas:

R1: G F
R2: F H
R3: H I

a) Supongamos que F es cierto y apliquemos todas las reglas posibles.
b) Supongamos que aparece información que indica que H es falso.

```
(defrule R1          | (defrule R2          | (defrule R3
   (logical (G))     |    (logical (F))     |    (logical (H))
   =>                |    =>                |    =>
   (assert (F)))     |    (assert (H)))     |    (assert (I)))
```

```
CLIPS> (load logical.clp)  ;; Carga el fichero que contiene las reglas anteriores
Defining defrule: R1 +j
Defining defrule: R2 +j
Defining defrule: R3 +j
TRUE
CLIPS> (watch rules)  ;; Activamos la traza de reglas
CLIPS> (watch facts)  ;; Activamos la traza de hechos
CLIPS> (reset)  ;; Inicializa la MT
==> f-0     (initial-fact) ;; Este hecho lo aserta CLIPS por defecto
CLIPS> (assert (F))  ;; F es cierto (asertamos F)
==> f-1     (F)
<Fact-1>
CLIPS> (run)  ;; Se aplican todas las reglas posibles
FIRE    1 R2: f-1
==> f-2     (H)
FIRE    2 R3: f-2
==> f-3     (I)
CLIPS> (facts)  ;; Muestra el contenido de la MT
f-0     (initial-fact)
f-1     (F)
f-2     (H)
f-3     (I)
For a total of 4 facts.
CLIPS> (retract 2)  ;; Ahora H es falso (lo eliminamos de la MT, sabiendo que tiene índice
2)
<== f-2     (H) ;; Se eliminan automáticamente H e I
<== f-3     (I)
CLIPS> (facts) ;; Muestra todos los hechos que estén en la MT
f-0     (initial-fact)
f-1     (f)
For a total of 2 facts.
CLIPS>
```

- Si se crea una entidad sin el EC *logical*, se dice que esta entidad tiene justificación incondicional. Si se elimina una regla que apoya la justificación para una entidad, se elimina la justificación de tal entidad, pero no se elimina la entidad misma.
- Si en el antecedente (LHS) de una regla aparece uno o varios EC *logical*, éstos deben aparecer los primeros:

Regla legal	Regla ilegal

`(defrule correcta`
`  (logical (a))`
`  (logical (b))`
`  (c)`
`  =>`
`  (assert (d)))`
	`(defrule incorrecta-1`
	`  (a)`
	`  (logical (b))`
	`  (logical (c))`
	`  =>`
	`  (assert (d)))`

Regla ilegal	**Regla ilegal**

`(defrule incorrecta-2`
`  (logical (a))`
`  (b)`
`  (logical (c))`
`  =>`
`  (assert (d)))`
	`(defrule incorrecta-3`
	`  (or (a)`
	`  (logical (b)))`
	`  (logical (c))`
	`  =>`
	`  (assert (d)))`

2.2.2.2 EC's que "obtienen" la dirección de elementos de la MT

Ligan la dirección de las entidades de la MT (hechos o instancias) que satisfacen el elemento condicional a una variable para poder realizar acciones sobre ellos.

- Es un error ligar una variable con la dirección de un EC *not*.

Sintaxis: `?<var-dirección> <- (<elemento-condicional>)`

Ej.:
```
(defrule hab-vacía
     ?vacía <- (habitación (capacidad ?c)
                           (plazas-libres ?c)
                           (número ?n))
     =>
     (printout t "Número habitación vacía: " ?n crlf)
     (retract ?vacía))
```

2.2.3 Comandos (acciones) predefinidos

Las acciones o comandos (para distinguirlas de las funciones) son operaciones que no devuelven un valor, pero tienen algún efecto lateral útil.

❶ Acciones que modifican la MT: lista de hechos.

para crear hechos	(**assert** <hecho>+) (**duplicate** <especificador-hecho>)
para borrar hechos	(**retract** <especificador-hecho>)
para modificar elementos	(**modify** <especificador-hecho> <RHS-slot>*)

donde <especificador-hecho> puede ser:

a) una variable previamente ligada a la dirección del hecho a duplicar, borrar o modificar, ó
b) un índice de hecho (aunque generalmente éste no se conoce durante la ejecución de un programa).

Nota: Puede utilizarse el símbolo '*' con el comando *retract* para eliminar todos los hechos.

Ej.: *(Véase un ejemplo del caso (b) en el ejemplo del EC logical, pág.18)*

```
(assert (habitación (número 101)(capacidad 3)(plazas-libres 3)))

(defrule ...
      ?hab <- (habitación (número ?num))
      =>
      (modify ?hab (capacidad 3)(plazas-libres 3))
      ...)

(defrule ...
      ?hab <- (habitación)
      =>
      (retract ?hab)
      ...)
```

Ejercicio:

Escribir una regla que aloje a un estudiante en la mayor habitación libre, si no existe ninguna parcialmente ocupada "compatible" (de estudiantes del mismo sexo).

SI existe un estudiante sin piso asignado
y
no existe una habitación compatible parcialmente ocupada
y
existe una habitación libre
y
es la habitación libre más grande

ENTONCES
Alojar al estudiante
Registrar al estudiante en la habitación

Solución:

```
(defrule alojar-mayor-hab-libre
     ?vagabundo <- (estudiante (nombre ?nom)
                               (sexo ?s)
                               (fuma? ?f)
                               (alojado nil))

     (not (habitación (plazas-libres ?p1 & : (> ?p1 0))
                      (capacidad ?c & : (> ?c ?p1))
                      (sexos ?s)
                      (fuman? ?f)))
     ?alojarlo-en <- (habitación (número ?num)
                                 (plazas-libres ?p2)
                                 (capacidad ?c1 & ?p2))
     (not (habitación (capacidad ?c2 & : (> ?c2 ?c1))
                      (sexos nil)
                      (fuman? nil)))
     =>
     (modify ?vagabundo (alojado ?num))
     (modify ?alojarlo-en (plazas-libres (- ?p2 1))
                          (ocupantes (create$ ?nom))
                          (sexos ?s)
                          (fuman? ?f)))
```

❷ Acciones procedurales.

Ligaduras: La función *bind* sirve para asignar un valor a una variable, es decir, su efecto es el de una asignación, similar a la que permiten otros lenguajes con el símbolo "`:=`".

Sintaxis: `(bind <variable> <expresión>*)`

if - then - else: `(if <expresión> then <acción>+ [else <acción>+])`

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

loop-for-count: `(loop-for-count (<var> <inicio> <final>) [do] <acción>*)`

progn$: permite realizar un conjunto de acciones sobre cada campo de un valor multicampo.

Sintaxis: `(progn$ <expresión-lista> <expresión>*)`

`<expresión-lista> ::= <expresión-multicampo> |`
`(<variable> <expresión-multicampo>)`

Ej.:
```
CLIPS>(progn$ (?var (create$ abc def ghi))
           (printout t "-->" ?var "<--" crlf))
--> abd <--
--> def <--
--> ghi <--
CLIPS>
```

return: termina la ejecución de la función actual; si se incluye un argumento, devuelve un valor.

Sintaxis: `(return [<expresión>])`

break: terminación del ciclo actual: `(break)`

switch: procedimiento de selección múltiple (≡ *case*)

```
(switch <expresión-test>
      <sentencia-case>
      <sentencia-case>+
      [(default <acción>*)])

<sentencia-case> ::= (case <comparación> then <acción>*)
```

❸ Comandos sobre la agenda

`(agenda [<nombre-módulo>])`
Visualiza por pantalla la lista de reglas activadas del módulo dado, o del actual, si no se especifica módulo (*véase la sección 4.2*).

`(clear)`
Elimina toda la información contenida en el entorno actual de CLIPS. Esta información incluye: construcciones, la lista de hechos. Y define automáticamente las siguientes construcciones:

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

`(halt)`
Detiene la activación de reglas. La agenda permanece intacta, y la ejecución puede reanudarse con el comando *run*. No devuelve ningún valor.

`(run [n])`
Comienza la ejecución de reglas en el foco actual. El parámetro *n*, que es opcional, es un número entero que indica que sólo *n* reglas, de las que estén activas, deben ejecutarse; si no se especifica, o si *n* es `-1`, entonces se activarán tantas reglas como haya en la agenda.

2.2.4 Funciones predefinidas: de E/S, matemáticas, de conversión, de *strings*, de manejo de valores multicampo y funciones de entorno. Nombres lógicos predefinidos.

Nombres lógicos.

Los nombres lógicos permiten referenciar un dispositivo de E/S sin tener que entender los detalles de la implementación de la referencia. Muchas funciones en CLIPS usan nombres lógicos. Un nombre lógico puede ser un símbolo, un número, ó un *string*. A continuación se muestra una lista de los nombres lógicos predefinidos en CLIPS:

Nombre	*Descripción*
`stdin`	Las funciones *read* y *readline* leen de *stdin* si se especifica `t` como el nombre lógico. *stdin* es el dispositivo de entrada por defecto.
`stdout`	Las funciones *format* y *printout* envían sus salidas a *stdout* si se especifica `t` como el nombre lógico. *stdout* es el dispositivo de salida por defecto.
`wclips`	El *prompt* de CLIPS se envía al dispositivo asociado con este nombre lógico.
`wdialog`	A este nombre lógico se envía cualquier mensaje de información.
`wdisplay`	Las peticiones de presentar información (tal como hechos o reglas), se envían al dispositivo asociado con este nombre lógico.
`werror`	A este nombre lógico se envían todos los mensajes de error.
`wwarning`	A este nombre lógico se envían todos los mensajes de aviso.
`wtrace`	Toda la información sobre las trazas habilitadas se envía a este nombre lógico (con la excepción de las compilaciones que son enviadas a *wdialog*).

Nota: Cualquiera de estos nombres lógicos puede ser utilizado allí donde se requiera un nombre lógico.

❶ Funciones de E/S

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

Abre un fichero con el modo especificado ("`r`", "`w`", "`r+`", "`a`") asociando el nombre de fichero dado al nombre lógico especificado. Devuelve `TRUE` si la apertura tiene éxito, en caso contrario `FALSE`. Si no se especifica modo, se supone "`r`".

Ej.:
```
(open "MS-DOS\\directory\\file.clp" entrada)
(open "mifichero.clp" salida "w")
```

`(close [<nombre-lógico>])`

Cierra el fichero asociado con el nombre lógico especificado. Si no se especifica el nombre lógico, se cierran todos los ficheros. Devuelve `TRUE` si no se produce ningún error al cerrar el fichero, en caso contrario `FALSE`.

`(printout <nombre-lógico> <expresión>*)`

Evalúa e imprime como salida al nombre lógico 0 o más expresiones no formateadas.

`(read [<nombre-lógico>])`

Lee un campo simple (*single field*) desde el nombre lógico especificado. Si no se especifica nombre lógico, se supondrá *stdin* (entrada estándar). Devuelve el elemento leído (que será siempre de un tipo primitivo) si tiene éxito, o `EOF` si no hay entrada disponible.

Ej.:
```
(assert (habitación (número (read))))
```

(readline [<nombre-lógico>])

Lee una línea completa desde el nombre lógico especificado (supondrá *stdin*, si no se especifica ninguno). Devuelve un *string* si tuvo éxito, o EOF si no había entrada disponible.

Ej.:
```
(defrule poner-fecha
     ?primera <- (initial-fact)
     =>
     (retract ?primera)
     (printout t "Introduzca la fecha: " crlf)
     (assert (hoy fecha (readline))))
```

Ej.:
```
(defrule obtener-nombre
     =>
     (printout t "Introduzca su nombre: ")
     (bind ?nom (readline))
     (assert (nombre-usuario ?nom)))
```

(format <nombre-lógico> <string-control> <parámetros>*)

Envía al dispositivo asociado con el nombre lógico especificado un *string* formateado. El *string* de control contiene unos *flags* de formato que indican cómo serán impresos los parámetros. El funcionamiento es similar a la función *printf* del lenguaje *C*. La función devuelve el *string* impreso.

<nombre-lógico> puede ser:

nil, no se produce salida alguna, pero devuelve el *string* formateado.
t, se imprime en la salida standard.

<string-control> son de la forma: %[-][M][.N]x, donde:

El '-' es opcional y significa justificación a la izda. (a la dcha. por defecto).
M indica la anchura del campo en columnas. Como mínimo se imprimirán M caracteres.
N especifica, opcionalmente, el número de dígitos a la derecha del punto decimal. Por defecto, se toman 6 para los números reales.
x especifica el formato de impresión y puede ser:

d	entero.
f	formato decimal.
e	formato exponencial (potencias de 10).
g	general (numérico). Imprimir con el formato más corto.
o	octal. Número sin signo. (Especificador N no aplicable).
x	hexadecimal. Ídem anterior.
s	*string*.
n	salto de línea (CR + LF).
r	retorno de carro.
%	el carácter "%".

Ej.:
```
CLIPS> (format nil "Nombre: %-15s Edad: %3d"
                   "John Smith" 35)
"Nombre: John Smith     Edad:  35"
CLIPS>
```

(rename <nombre-fichero-antiguo> <nombre-fichero-nuevo>)

Cambia de nombre a un fichero. Devuelve `TRUE` si tuvo éxito; `FALSE` en otro caso.

`(remove <nombre-fichero>)`
Borra el fichero de nombre el especificado. Devuelve `TRUE` si tuvo éxito; `FALSE` en caso contrario.

❷ Funciones matemáticas, de conversión y de *strings*: similares a las de LISP y *C*.

- **aritméticas:** `abs, div, float, integer, max, min, +, -, *, /`

`(abs <expresión-numérica>)`
Devuelve el valor absoluto del argumento dado.

`(div <expresión-numérica> <expresión-numérica>+)`
Devuelve el valor del primer argumento dividido por cada uno de los sucesivos parámetros. La división es entera.

`(float <expresión-numérica>)`
Devuelve el argumento convertido al tipo *float*.

`(integer <expresión-numérica>)`
Devuelve el argumento convertido al tipo *integer*.

`(max | min <expresión-numérica> <expresión-numérica>+)`
Devuelve el valor del argumento mayor (menor).

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

donde `<op> ::= + | - | * | /`

+	Devuelve la suma de todos sus argumentos.
-	Devuelve el valor del primer argumento menos la suma de todos los sucesivos.
*	Devuelve el producto de todos sus argumentos.
/	Devuelve el valor del primer argumento dividido por cada uno de los sucesivos argumentos.

- **funciones matemáticas extendidas:**

`(exp <expresión-numérica>)`
Devuelve el valor del número *e* elevado a la potencia que indica el argumento.

`(log <expresión-numérica>)`
Devuelve el logaritmo en base *e* del argumento.

`(log10 <expresión-numérica>)`
Devuelve el logaritmo en base 10 del argumento.

`(mod <expresión-numérica> <expresión-numérica>)`
Devuelve el resto de dividir el primer argumento entre el segundo.

`(pi)` Devuelve el valor del número π.

`(round <expresión-numérica>)`

Devuelve el valor del argumento redondeado al entero más cercano.

`(sqrt <expresión-numérica>)`
Devuelve la raíz cuadrada del argumento.

`(** <expresión-numérica> <expresión-numérica>)`
Devuelve el valor del primer argumento elevado a la potencia del segundo.

`(random)` Devuelve un número entero aleatorio.

- **trigonométricas:** Sintaxis general: `(<f> <expresión-numérica>)`

  Donde `<f> ::= sin | cos | tan | cot | sec | tanh | sinh | ...`

- **de conversión:** Sintaxis general: `(<f> <expresión-numérica>)`

  donde `<f> ::= deg-grad | deg-rad | grad-deg | rad-deg`

`deg-grad`	Conversión de grados sexagesimales (º) a grados centesimales (^{g})
`deg-rad`	Conversión de grados sexagesimales a radianes
`grad-deg`	Conversión de grados centesimales a grados sexagesimales
`rad-deg`	Conversión de radianes a grados sexagesimales

$360º = 400^g$ $360º = 2\pi$ rad.

- **funciones con *strings*:**

`(lowcase <expresión-simbólica-o-de-cadena>)`
Convierte a minúsculas el símbolo o *string* dado.

Ej.:
```
> (lowcase "Este eS un ejEMplo de lowcase")
"este es un ejemplo de lowcase"
> (lowcase Una_Palabra_DE_Ejemplo_Para_lowcase)
una_palabra_de_ejemplo_para_lowcase
```

`(upcase <expresión-simbólica-o-de-cadena>)`
Convierte a mayúsculas el símbolo o *string* dado.

Ej.:
```
> (upcase "Este es un ejemplo de upcase")
"ESTE ES UN EJEMPLO DE UPCASE"
> (upcase Una_Palabra_DE_Ejemplo_Para_Upcase)
UNA_PALABRA_DE_EJEMPLO_PARA_UPCASE
```

`(str-cat <expresión>*)`

Concatena todos los parámetros y los devuelve como un *string*. Cada <expresión> debe ser de uno de los siguientes tipos: *symbol*, *string*, *integer*, *float* ó *instance-name*.

Ej.:
```
> (str-cat "vaca" CaJa 0.961)
"vacaCaJa0.961"
```

`(sym-cat <expresión>*)`

Concatena todos los parámetros y los devuelve como un único símbolo. Cada <expresión> debe ser de uno de los siguientes tipos: *symbol*, *string*, *integer*, *float* ó *instance-name*.

Ej.:
```
> (sym-cat "vaca" CaJa 0.961)
vacaCaJa0.961
```

`(str-compare <expresión-simbólica-o-de-cadena>`
`<expresión-simbólica-o-de-cadena>)`

Devuelve un número entero que representa el resultado de la comparación:

`0`	si ambos parámetros son idénticos.
`1`	si el 1er *string* (o símbolo) > 2º *string* (o símbolo)
`-1`	si el 1er *string* (o símbolo) < 2º *string* (o símbolo)

Ej.:
```
> (str-compare "abcd" "abcd")
0
> (str-compare "zoo" "perro")
1
> (str-compare "casa" "mesa")
-1
> (str-compare "casa" "MEsa")
1
```

`(str-index <expresión-lexema> <expresión-lexema>)`

Comprueba si el primer argumento está contenido en el segundo, y en caso afirmativo devuelve la posición (empezando desde 1) en el segundo parámetro donde aparece el primero. Si no está contenido, devuelve `FALSE`.

Ej.:
```
CLIPS> (str-index "def" "abcdefghi")
4
CLIPS> (str-index "qwerty" "qwertypoiuyt")
1
CLIPS> (str-index "qwerty" "poiuytqwer")
FALSE
CLIPS>
```

`(str-length <expresión-simbólica-o-de-cadena>)`
Devuelve la longitud (en caracteres) de un *string* o un símbolo.

```
Ej.:
CLIPS> (str-length "abcd")
4
CLIPS> (str-length xyz)
3
CLIPS>
```

`(sub-string <inicio> <fin> <expresión-de-cadena>)`
Devuelve una subcadena de la expresión de cadena dada que está comprendida entre las posiciones de <inicio> y <fin>.

```
Ej.:
CLIPS> (sub-string 3 8 "abcdefghijkl")
"cdefgh"
CLIPS> (sub-string 6 3 "abcdefgh")
""
CLIPS> (sub-string 7 10 "abcdefgh")
"gh"
```

❸ Manipulación de valores multicampo

`(create$ <expresión>*)`
Crea un valor multicampo con las expresiones dadas como parámetros.
Devuelve el valor creado.

`(delete$ <expresión-multicampo> <posición-inicio> <posición-final>)`
Borra todos los campos en el rango especificado de la expresión multicampo y devuelve el resultado.

`(explode$ <expresión-string>)`
Devuelve un valor multicampo creado a partir de los campos contenidos en el *string*.

`(first$ <expresión-multicampo>)`
Devuelve el primer campo de la expresión multicampo. (Ídem a LISP)

`(implode$ <expresión-multicampo>)`
Devuelve un *string* que contiene los campos del valor multicampo especificado.

`(insert$ <expr-multicampo> <expr-entera> <expresión-simple-o-multicampo>+)`
Inserta todos los valores simples o de multicampo en la expresión de multicampo antes del *n*-ésimo valor (definido por la expresión entera) de la expresión multicampo dada.

`(length$ <expresión-multicampo>)`
Devuelve el número de campos del parámetro dado. (Similar a la homóloga de LISP).

`(member$ <expresión-campo-simple> <expresión-multicampo>)`
Devuelve la posición del primer argumento en el segundo, o devuelve `FALSE` si el primer argumento no está en el segundo.

`(nth$ <expresión-entera> <expresión-multicampo>)`
Devuelve el *n*-ésimo campo (<expresión-entera>) contenido en la expresión multicampo.

`(replace$ <expresión-multicampo> <inicio> <final>`
`<expresión-simple-o-multicampo>)`
Reemplaza los campos en el rango especificado por <inicio> y <final> en la expresión multicampo con todos los del último argumento y devuelve el resultado.

`(rest$ <expresión-multicampo>)`
Devuelve un valor multicampo con todos los elementos originales excepto el primero.

`(subseq$ <expresión-multicampo> <inicio> <final>)`
Extrae los campos en el rango `inicio..final` del primer argumento, y los devuelve en un valor multicampo.

Ejemplos:

```
>(create$ (+ 3 4) xyz "texto" (/ 8 4))
(7 xyz "texto" 2.0)

>(nth$ 3 (create$ a b c d e f))
c

>(nth$ 9 (create$ 1 2 3 4))
nil

>(member$ azul (create$ rojo 3 "texto" 8.7 azul))
5

>(subsetp (create$ 1 2 3)(create$ 1 3 4 5 6 2))
TRUE

>(delete$ (create$ a b c d e) 3 4)
(a b e)

>(delete$ (a b c d e) 3 4) ==> ERROR

>(explode$ "1 2 abc \" el coche\"")
(1 2 abc " el coche")

>(implode$ (create$ 1 azul de 2))
"1 azul de 2"

>(subseq$ (create$ a b c d) 3 4)
(c d)

>(replace$ (create$ a b c) 3 3 x)
(a b x)
```

```
>(replace$ (create$ a b c) 3 3 (x y z)) ==> ERROR

>(replace$ (create$ a b c) 3 3 (create$ x y z))
(a b x y z)

>(insert$ (create$ a b c d) 1 x)
(x a b c d)
```

❹ Funciones de entorno *(environment functions)*

`(load <nombre-de-fichero>)`
Carga las construcciones contenidas en un fichero. La función devuelve el símbolo `TRUE`, si no ocurre ningún error al cargar; `FALSE`, en caso contrario.

Ej.: `(load "B:reglas.clp")`

```
(load "B:\\ia\\clips\\ejemplos\\reglas.clp")
```

`(save <nombre-de-fichero>)`
Guarda en disco el conjunto de construcciones actualmente en la memoria de trabajo en el fichero especificado.

`(exit)` Sale del entorno CLIPS.

`(system <expresión>+)`
Ejecuta comandos del sistema operativo desde dentro de CLIPS.

Ej.: `(system "dir")`

```
(defrule listar-directorio
     (listar-directorio ?dir)
     =>
     (system "ls " ?dir))
```

`(batch <nombre-de-fichero>)`
Permite ejecutar un fichero de comandos CLIPS. Podemos guardar en un fichero de texto una serie de comandos CLIPS que tecleemos con frecuencia y luego ejecutarlos secuencialmente desde dentro de CLIPS [algo similar a los *scripts* de UNIX o los ficheros *batch* (.BAT) del MS-DOS]. Devuelve `TRUE` si se ejecutó satisfactoriamente el fichero de comandos, ó `FALSE` en caso contrario.

`(dribble-on <nombre-de-fichero>)`
Este comando puede ser utilizado para registrar toda entrada (desde teclado) y salida (a pantalla) en un fichero de texto especificado. Una vez ejecutado este comando, todas las salidas a pantalla y todas las entradas desde teclado serán enviadas al fichero especificado, así como a la pantalla.

`(dribble-off)` Este comando cancela los efectos del comando *dribble-on*.

2.2.5 Funciones definidas por el usuario: constructor *deffunction*

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

Una *deffunction* se compone de cinco elementos: **1)** un nombre (que debe ser un símbolo), **2)** un comentario opcional, **3)** una lista de cero o más parámetros requeridos (variables simples), **4)** un parámetro comodín opcional que sirve para manejar un número variable de argumentos (variable multicampo), y **5)** una secuencia de acciones o expresiones que serán ejecutadas en orden cuando se llame a la función.

El valor devuelto por la función es la última acción o expresión evaluada dentro de la función. Si una *deffunction* no tiene acciones, devolverá el símbolo `FALSE`. Si se produce algún error mientras se ejecuta la función, cualquier otra acción de la función aún no ejecutada se abortará, y la función devolverá el símbolo `FALSE`.

Ej.:
```
(deffunction mostrar-params (?a ?b $?c)
      (printout t ?a " " ?b " and " (length ?c) " extras: " ?c crlf))

(mostrar-params 1 2)    1 2 and 0 extras: ()
(mostrar-params a b c d)    a b and 2 extras: (c d)
```

Declaraciones incompletas (*forward declarations*).

Consisten en declarar una función, sin especificar ni los parámetros ni el cuerpo (acciones). La definición completa de la función puede estar en un fichero diferente al de la declaración.

Ej.: `(deffunction factorial () )` ;;; declaración incompleta

2.2.6 Propiedades de las reglas: *salience* y *auto-focus*

Esta característica permite definir las propiedades de una regla. Estas propiedades son: *salience* o *auto-focus*. Las propiedades se declaran en la parte izquierda de una regla utilizando la palabra clave ***declare***. Una regla sólo puede tener una sentencia `declare` y debe aparecer antes del primer elemento condicional.

salience

- Permite al usuario asignar una prioridad a una regla. La prioridad asignada debe estar en el rango -10000 .. +10000. Por defecto, la prioridad de una regla es 0.
- Si la agenda incluye varias reglas, se desencadenará en primer lugar aquella que tenga mayor prioridad.

auto-focus

- Permite la ejecución automática de un comando ***focus*** cada vez que la regla se activa. Por defecto, la propiedad *auto-focus* de una regla es `FALSE`. (*Véase 4.3: Módulos y control de la ejecución*).

Ejercicio:

Queremos desalojar al último estudiante asignado a la habitación 111 (se supone que el piso está ocupado por más personas).

```
(defrule desalojar
     ?hab <- (habitación  (número 111)
                          (ocupantes $?quien)
                          (plazas-libres ?p))
     ?ocupa <- (estudiante (alojado 111))
     =>
     (modify ?ocupa (alojado nil))
     (modify ?hab (plazas-libres (+ ?p 1)
                  (ocupantes (delete$ $?quien 1 1))))
```

2.3 MOTOR DE INFERENCIA

2.3.1 Ciclo básico

1. Si se ha alcanzado el número de ciclo de desencadenamiento expresados o no hay foco actual, entonces PARAR.

   Si no, seleccionar para ejecutar la regla tope de la agenda del módulo actual.

   Si no hay reglas en dicha agenda, entonces eliminar el foco actual de la pila de focos y actualizar al siguiente de la pila.

   Si la pila de focos está vacía, entonces PARAR, si no ejecutar 1. de nuevo.

2. Se ejecutan las acciones de la parte derecha de la regla seleccionada. Incrementar el nº de reglas desencadenadas.

   Si se usa ***return***, entonces eliminar el foco actual de la pila de focos.

3. Como resultado de 2., se activan o desactivan nuevas reglas.

   Las reglas activadas se sitúan en la agenda del módulo en el que están definidas. La situación concreta depende de la prioridad de la regla ***(salience)*** y de la estrategia de resolución de conflictos actual. Las reglas desactivadas salen de la agenda.

4. Si se utiliza prioridad dinámica ***(dynamic salience)***, entonces se reevalúan los valores de prioridad de las reglas de la agenda. Ir al paso 1.

2.3.2 Filtrado (*match*)

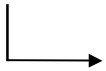
"Construye una instanciación para cada conjunto de elementos que satisfacen una regla"

Instanciación: `<prioridad> <nombre-regla> : fact-index*`

Fact index Formato: `f-<número-entero>`

Entero único asignado por CLIPS a cada entidad (hecho) de la memoria de trabajo. Se crea en tiempo de ejecución.

Representa el orden temporal en el que ha sido incorporado a la memoria de trabajo.



Se refiere al orden de aparición en el tiempo. El hecho más reciente tendrá el *fact index* más alto.

Conjunto conflicto

La **agenda** es la lista de todas las reglas cuyas condiciones se satisfacen y todavía no se han ejecutado.

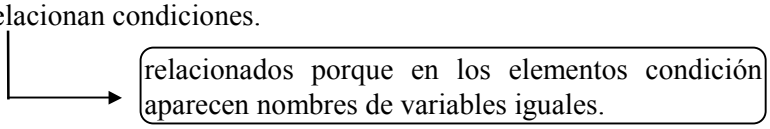
```
CLIPS>(agenda)
h-vacia: f-3
h-ocupada: f-2
h-vacia: f-1
For a total of 3 activations.
```

```
CLIPS>(facts)
f-0   (initial-fact)
f-1   (habitación (número 1)(capacidad 5)(plazas-libres 5))
f-2   (habitación (número 2)(capacidad 5)(plazas-libres 3))
f-3   (habitación (número 3)(capacidad 5)(plazas-libres 5))
For a total of 4 facts.

CLIPS>(rules)
h-vacia
h-ocupada
For a total of 2 defrules.

CLIPS>(ppdefrule h-ocupada)
(defrule MAIN::h-ocupada
      ?vacía<-(habitación(capacidad ?c)(plazas-libres ?p &:(> ?c ?p))(número ?n))
      =>
      (printout t "Número habitación semiocupada: " ?n crlf)
      (retract ?vacía))

CLIPS>(ppdefrule h-vacia)
(defrule MAIN::h-vacia
      ?vacía <- (habitación(capacidad ?c)(plazas-libres ?p & ?c)(número ?n))
      =>
      (printout t "Número habitación vacía: " ?n crlf)
      (modify ?vacía (plazas-libres (- ?c 1))))
```

Proceso de filtrado de una regla.

Fase 1: Se toman todos los elementos que satisfacen cada una de sus condiciones.

Fase 2: Se toman las combinaciones de elementos que satisfacen la restricción impuesta por las variables que relacionan condiciones.



Ej.:

MT

```
...
(habitación (número 221)(capacidad 1)(plazas-libres 1)(ocupantes))
(habitación (número 346)(capacidad 2)(plazas-libres 1)(ocupantes))
(habitación (número 222)(capacidad 1)(plazas-libres 1)(ocupantes))
(estudiante (nombre pepa)(sexo mujer)(alojado 221))
(estudiante (nombre luis)(sexo varón)(alojado 346))
(estudiante (nombre abilio)(sexo varón)(alojado 222))
For a total of 10 facts.
```

```
(defrule MAIN::ejem-instanciaciones
      (estudiante (nombre ?quien)(alojado ?num))
      (habitación (numero ?num)(capacidad 1)(plazas-libres 1))
      =>)
```

```
CLIPS>(matches ejem-instanciaciones)
Matches for Pattern 1
f-7
f-8
f-9
Matches for Pattern 2
f-4
f-6
Partial matches for CEs 1 - 2
f-9,f6
f-7, f-4
Activations
f-9, f6
f-7, f-4
```

Agenda: Conjunto conflicto

```
0     ejem-instanciaciones: f-9, f-6

0     ejem-instanciaciones: f-7, f-4
```

2.3.3 Resolución de conflictos: estrategias (*select*)

La agenda es la lista de todas las reglas cuyas condiciones se satisfacen y todavía no se han ejecutado.

Cada módulo tiene su propia agenda, y su funcionamiento es similar al de una pila: la regla tope de la agenda es la primera que se ejecuta.

Cuando una regla se activa de nuevo, se sitúa en la agenda en base a los siguientes factores:

a) las nuevas reglas activadas se sitúan por encima de todas las reglas de menor prioridad *(salience)* y por debajo de todas las reglas de mayor prioridad.

b) entre las reglas de igual prioridad, se utiliza la estrategia de resolución de conflictos actual para determinar su situación relativa.

c) si los pasos (a) y (b) no son suficientes para determinar el orden, las reglas se colocan arbitrariamente en relación con las otras reglas (no aleatoriamente, sino que depende de la implementación de las reglas).

CLIPS posee siete estrategias: profundidad ***(depth)***, anchura ***(breadth)***, simplicidad ***(simplicity)***, complejidad ***(complexity)***, aleatoria ***(random)***, orden lexicográfico ***(lex)***, y ***means-end analysis***.

Sintaxis: `(set-strategy <nombre-estrategia>)`

⇨ devuelve el nombre de la estrategia anterior. Además, la agenda se reordena para reflejar la nueva estrategia de resolución del conjunto conflicto.

- **`depth`** (estrategia por defecto)
  Las nuevas reglas se sitúan sobre todas las de la misma prioridad.

- **`breadth`**
  Las nuevas reglas se sitúan por debajo de todas las de la misma prioridad.

- **`simplicity`**
  Entre las reglas de la misma prioridad, las nuevas reglas se sitúan sobre todas las activaciones de reglas con mayor o igual especificidad (*specificity*).

  especificidad: determinada por el número de comparaciones que deben realizarse en la LHS de la regla.

- **`complexity`**
  Entre las reglas de la misma prioridad, las nuevas reglas se sitúan sobre todas las activaciones de reglas con menor o igual especificidad.

- **`random`**
  A cada activación se le asigna un número aleatorio que se utilizará para determinar su lugar entre las activaciones de igual prioridad (este número se mantiene entre cambios de estrategia).

- **`lex`**
    1. Eliminación de las instanciaciones disparadas anteriormente (una vez ejecutada una instancia se elimina de la agenda).

    2. Supresión de las instanciaciones que no contengan el mayor *time-tag* de los aparecidos en el conjunto conflicto. Este proceso se repite sucesivamente hasta resolver el conflicto o encontrar un subconjunto de instanciaciones con el mismo *time-tag*.

    3. Supresión de las instanciaciones menos específicas.

    4. Selección aleatoria.

- **Means-End Analysis `(mea)`**
    1. Se comparan los *time-tag*s de los elementos instanciados por la primera condición de las reglas, y se eliminan las instanciaciones que no posean el mayor de los *time-tag*.

    2. Supresión de las instanciaciones que no contengan el mayor *time-tag* de los aparecidos en el conjunto conflicto. Este proceso se repite sucesivamente hasta resolver el conflicto o encontrar un subconjunto de instanciaciones con el mismo *time-tag*.

    3. Supresión de las instanciaciones menos específicas.

    4. Selección aleatoria.

Ejercicio:

Para cada uno de los siguientes conjuntos conflicto, indicar qué instanciación será ejecutada en primer lugar, para cada una de las estrategias LEX y MEA.

```
regla-uno     34 67 4     (especificidad 8)
regla-dos     36 2  3     (especificidad 4)


regla-tres    12 36 2     (especificidad 5)
regla-cuatro  12 2  36 7  (especificidad 10)


regla-cinco   13 8  5     (especificidad 6)
regla-seis    5  13 8     (especificidad 9)


regla-siete   34 67 4     (especificidad 8)
regla-ocho    5  34       (especificidad 4)
```

2.3.4 Ejecución (*act*)

Una vez seleccionada una regla, ésta se ejecuta.

Las acciones se ejecutan en el orden en el que han sido escritas.

A los elementos creados o modificados, se les asigna un índice de hecho (*fact index*) mayor que todos los existentes.

Las modificaciones realizadas en la MT son propagadas al conjunto conflicto (agendas).

3. EJECUCIÓN DE PROGRAMAS

3.1 El entorno de trabajo

Un programa CLIPS se ejecuta de la siguiente forma:

1. Se carga en el entorno mediante una instrucción *load*.
2. Se ejecuta un comando *reset*.
3. Se lanza el comando *run*.

A partir del momento en el que se ejecuta el comando *run*, empiezan a activarse reglas y, posteriormente, se ejecutan.

3.2 Inicialización de la MT

- Mediante comandos *assert* No práctico, salvo casos excepcionales.

```
CLIPS> (assert ...)
```

- Inicialización mediante *deffacts*: incorporación del conocimiento inicial.

```
(deffacts <nombre>
     (...)
     (...))
```

- Mediante una regla, que se suele llamar regla de inicio (*start-up rule*):

```
(defrule inicializar
     (initial-fact)
     =>
     (assert ...)
     (assert ...))
```

4. CONTROL DE LA EJECUCIÓN

4.1 Introducción

¿Qué ocurre si la memoria de producción (base de reglas) es grande?

Problemas:

Muy costoso el proceso de filtrado.
En el mantenimiento y depuración.

Solución: ***diseño modular***

Técnicas de programación para:

- Organizar la base de reglas mediante ***módulos***
- Dirigir la ejecución de las reglas mediante el ***foco***

4.2 Diseño modular: construcción *defmodule*

CLIPS permite el control de la ejecución y desarrollo modular de bases de conocimiento con la construcción ***defmodule***.

Los módulos permiten agrupar un conjunto de construcciones de modo que se mantenga un control explícito sobre la restricción del acceso de otros módulos: ámbito global estrictamente jerárquico en una sola dirección. Esto quiere decir que si un módulo *A* puede ver construcciones de un módulo *B*, entonces no es posible que el módulo *B* pueda ver las construcciones del módulo *A* (no se pueden establecer ciclos de visibilidad).

⇨ ≡ ámbito local y global de lenguajes como Ada o C.

Restringiendo el acceso a construcciones (*deftemplate* y *defclass*), los módulos pueden funcionar como pizarras (*blackboards*), permitiendo únicamente a otros módulos ver ciertos hechos e instancias.

Agrupando las reglas en módulos es posible controlar la ejecución de éstas.

Definición de módulos.

Sintaxis:
```
(defmodule <nombre-módulo> [<comentario>]
     <especificación-acceso>*)

<especificación-acceso> ::= (export <ítem>) |
                            (import <nombre-módulo> <ítem>)

<ítem> ::= ?ALL | ?NONE |
           <construcción> ?ALL | <construcción> ?NONE |
           <construcción> <nombre-construcción>+

<construcción> ::= deftemplate | defclass | defglobal |
                   deffunction | defgeneric
```

Una construcción *defmodule* no puede ser ni redefinida ni borrada una vez que se define (con la excepción del módulo `MAIN` que puede ser redefinido una vez). La única forma de eliminar un módulo es con el comando *clear*. Al principio de una sesión CLIPS o después de un comando *clear*, CLIPS define automáticamente el módulo `MAIN` mediante:

```
(defmodule MAIN)
```

Todas las clases predefinidas pertenecen al módulo `MAIN`. Sin embargo, no es necesario ni importar ni exportar las clases del sistema; siempre son accesibles. Excepto éstas, por defecto el módulo predefinido `MAIN` ni importa ni exporta construcción alguna.

Ej.:
```
(defmodule OBTENER-DATOS)

(defmodule ANALIZAR-DATOS)

(defmodule PROPONER-SOLUCIONES)
```

Con los módulos, es posible tener una construcción con el mismo nombre en dos módulos diferentes, pero no se permite que ambas sean visibles al mismo tiempo.

¿Cómo se incluye una construcción *(defrule, deffacts)* en un módulo?
→ Existen dos formas de especificar un nombre (de una construcción, función, regla, ...) perteneciente a un módulo: explícita ó implícita.

- explícita: se escribe el nombre del módulo (que es un símbolo) seguido de dos puntos (`::`)[3] y a continuación el nombre.

  Ej.: `DETECCIÓN::sensor` se refiere al nombre `sensor` del módulo `DETECCIÓN`.

- implícita: sin especificar el nombre del módulo ni los dos puntos `::`, ya que siempre existe un módulo "actual". El módulo actual cambia siempre que:
  - se defina una construcción *defmodule*, ó
  - se especifica el nombre de un módulo en una construcción (usando `::`), ó

[3]El símbolo `::` se denomina *separador de módulo*.

- se utilice la función *set-current-module*. El módulo `MAIN` es definido automáticamente por CLIPS y es el módulo actual por defecto cuando se inicia CLIPS por primera vez o después de un comando *clear*.

  Sintaxis: `(set-current-module <nombre-módulo>)`

  Ej.: El nombre `sensor` se referiría a un identificador definido en el módulo `DETECCIÓN` si éste fuera el actual.

4.2.1 Importación y exportación.

La especificación ***export*** se usa para indicar qué construcciones serán accesibles (visibles) a otros módulos que quieran utilizar alguna construcción del módulo dado.

La especificación ***import*** se usa para indicar qué construcciones, que se utilizan en el módulo que se está definiendo, serán de otros módulos.

Sólo se puede exportar e importar: *deftemplate*, *defclass*, *defglobal*, *deffunction* y *defgeneric*.
No se pueden exportar ni importar las construcciones: *deffacts, defrule* y *definstances*.

Existen tres modos diferentes de exportación / importación:

(Para importar, basta con sustituir la palabra clave *export* por la palabra clave *import*; y en el texto, donde dice *exportar* por *importar*).

1. Un módulo puede exportar todas las construcciones válidas que le son visibles. Esto se consigue mediante la palabra clave `?ALL`:
   Ej.: `(defmodule A (export ?ALL))`
   El módulo *A* exporta todas sus construcciones.

2. Un módulo puede exportar todas las construcciones válidas de un tipo particular que le son visibles. Para ello, se escribe la palabra clave *export* seguida del nombre de la construcción y ésta seguida de la palabra clave `?ALL`.
   Ej.: `(defmodule B (export deftemplate ?ALL))`
   El módulo *B* exporta todos sus *deftemplates*.

3. Un módulo puede exportar construcciones específicas de un tipo determinado que le son visibles. La sintaxis es escribir a continuación de la palabra clave *export*, el nombre del tipo de construcción seguido del nombre de uno o más construcciones visibles del tipo especificado.
   Ej.: `(defmodule DETECCIÓN (export deftemplate sensor evento-calor))`
   El módulo `DETECCIÓN` exporta los *deftemplate* `sensor` y `evento-calor`.

Podemos indicar que no se exporta o importa construcción alguna desde un módulo o que ninguna construcción de un tipo particular se exporta o importa sustituyendo la palabra clave `?ALL` por la palabra clave `?NONE`.

Un módulo debe estar definido antes de poder usarlo en una importación. Además, si se especifican construcciones en la especificación de importación, éstas deben estar ya definidas en el módulo que las exporta. No es necesario importar las construcciones en un módulo en el que se definen para poder usarlas. Una construcción puede ser importada indirectamente en un módulo que directamente importa y luego exporta el módulo a usar.

Exportación / Importación de hechos o instancias

A diferencia de las reglas y construcciones *deffacts*, sí se pueden compartir las construcciones *deftemplate* (y todos los hechos que utilicen ese *deftemplate*) con otros módulos. Así, un módulo es "propietario" de todos los hechos y/o instancias que incluyen su correspondiente *deftemplate* o *defclass*; por contra, **no** son propiedad del módulo que los crea.

Los hechos y/o instancias son visibles sólo para aquellos módulos que importan la correspondiente *deftemplate* o *defclass*.

> **Una base de conocimiento puede dividirse de forma que las reglas y otros constructores puedan ver solamente aquellos hechos o instancias que sean de su interés.**

Nótese que el *deftemplate* `(initial-fact)` y la clase `INITIAL-OBJECT` deben ser importadas explícitamente desde el módulo `MAIN`, ya que, si no se importan explícitamente, las reglas que tengan en su parte izquierda los *patterns* `(initial-fact)` o `(initial-object)` no se activarán.

4.3 Módulos y control de la ejecución

Además de controlar la importación y exportación, la construcción *defmodule* también puede utilizarse para controlar la ejecución de las reglas. Cada módulo tiene su propia agenda (conjunto conflicto). Entonces la ejecución puede controlarse seleccionando una agenda, y en ésta se elegirán reglas para ejecutar. De esta forma, se elige un conjunto determinado de reglas a ejecutar.

¿Cómo se hace esa elección de agenda? → Mediante el comando *focus*.

4.3.1 El comando *focus*.

Ahora que hay distintas agendas con reglas, ¿qué sucede cuando se ejecuta un comando ***run***?
→ Se disparan aquellas reglas del módulo que tiene el foco actual. CLIPS mantiene un foco actual que determina sobre qué agenda tendrá efecto un comando *run*. Recuérdese que los comandos *reset* y *clear*, establecen automáticamente como módulo actual al módulo `MAIN`. El foco actual no cambia cuando cambia de módulo actual.

Para cambiar el foco actual se utiliza el comando *focus*, cuya sintaxis es la siguiente:

Sintaxis: `(focus <nombre-módulo>+)`

El comando *focus* no sólo modifica el foco actual, sino que también "recuerda" el valor del foco actual anterior. Realmente el foco actual es el valor tope de una estructura de datos similar a una pila, denominada pila de focos. Siempre que el comando *focus* cambia el foco actual, de hecho lo que hace es empilar el nuevo foco actual en la cima de la pila de focos, desplazando a los focos actuales previos.

La ejecución de las reglas continúa hasta que:

- cambia el foco a otro módulo,
- no haya reglas en la agenda, ó
- se ejecute ***return*** en la RHS de una regla.

Cuando se vacía la agenda de un módulo (foco actual), el foco actual se elimina de la pila de focos, y el siguiente módulo en la pila de focos se convierte en el foco actual.

Nota 1: El efecto de dos comandos *focus* para empilar dos módulos es que se ejecutarán primero las reglas del módulo del segundo comando *focus* antes que las del módulo que se especificó en el primer comando *focus*. Sin embargo, si en un mismo comando se especifica más de un módulo, los módulos se empilan en la pila de focos de derecha a izquierda.

Nota 2: Un mismo módulo puede estar más de una vez en la pila de focos, pero ejecutar un comando *focus* sobre un módulo que ya es el foco actual no tiene efecto alguno.

Comandos sobre la pila de focos.

`(list-focus-stack)`	Lista el contenido de la lista de focos.
`(clear-focus-stack)`	Elimina todos los focos de la pila de focos.
`(get-focus-stack)`	Devuelve un valor multicampo con los nombres de los módulos en la pila de focos
`(get-focus)`	Devuelve el nombre del módulo del foco actual, o el símbolo `FALSE` si la pila de focos está vacía.
`(pop-focus)`	Elimina el foco actual de la pila de focos y devuelve el nombre del modulo, ó `FALSE` si la pila de focos está vacía.

4.3.2 El comando *return*.

Es posible detener prematuramente la ejecución de las reglas activadas en la agenda de un módulo específico, esto es, antes de que la agenda del módulo se quede vacía. El comando *return* se puede utilizar para finalizar inmediatamente la ejecución de la parte derecha de una regla y retirar el foco actual de la pila de focos, devolviendo así el control de ejecución al siguiente módulo en la pila de focos. Cuando se utilice en la parte derecha de las reglas, el comando *return* no debe tener argumentos. Debe notarse que el comando *return* detiene inmediatamente la ejecución de la RHS de una regla. Por tanto, cualquier otro comando que siguiese al comando *return* en la RHS no se ejecutaría, mientras que si se usara en su lugar un comando *pop-focus*, éste permitiría la terminación completa de las acciones en la RHS de la regla.

La propiedad **auto-focus** ejecuta automáticamente un comando `focus` cuando la regla se activa, convirtiendo en foco actual al módulo al que pertenezca dicha regla. Este efecto se consigue con el atributo *auto-focus*, que se especifica en la sentencia *declare* junto con el atributo *salience*. Para activar el efecto del *auto-focus*, se establece a `TRUE`. Por defecto, este atributo tiene el valor `FALSE`.

Sintaxis: `(auto-focus TRUE | FALSE)`

La propiedad *auto-focus* es particularmente útil para reglas que se encargan de detectar violaciones de restricciones. Puesto que el módulo que contiene a dicha regla se convierte inmediatamente en el foco actual, es posible obrar en consecuencia cuando ocurre la violación, en lugar de tener un fase específica para la detección de las excepciones.

Ej.: Programa que prueba el uso de diferentes módulos:

```
(defmodule a)
(defmodule b)
(deffacts coleb (hechob 1)(hechob 2))
(deffacts a::colea (hechoa 2)(hechoa 1))

(defrule MAIN::inicio
     (initial-fact)
     =>
     (focus a b))

(defrule a::r2
     (hechoa 2)
     =>
     (printout t "cambio de foco." crlf)
     (return))

(defrule a::r1
     (hechoa 1)
     =>
     (printout t crlf "hechoa 1" crlf))

(defrule b::r2
     (hechob 2)
     =>
     (printout t "He pasado a b2." crlf))

(defrule b::r1
     (hechob 1)
     =>
     (printout t "He pasado a b1." crlf))
```

```
TRUE
CLIPS>(load "ejemplo.clp")
Defining defmodule: a
Defining defmodule: b
Defining deffacts: coleb
Defining deffacts: colea
Defining defrule: inicio +j
Defining defrule: r2 +j
Defining defrule: r1 +j
Defining defrule: r2 +j
Defining defrule: r1 +j
TRUE

CLIPS>(reset)
<== Focus MAIN
==> Focus MAIN
==> f-0    (initial-fact)
==> Activation 0 inicio: f-0
==> f-1    (hechoa 2)
==> Activation 0r2: f-1
==> f-2    (hechoa 1)
==> Activation 0r1: f-2
==> f-3    (hechob 1)
==> Activation 0r1: f-3
==> f-4    (hechob 2)
==> Activation 0r2: f-4

CLIPS>(run 1)
FIRE 1 inicio: f-0
==> Focus b from MAIN
==> Focus a from b

CLIPS>(agenda MAIN)

CLIPS>(agenda a)
0     r1: f-2
0     r2: f-1
For a total of 2 activations.

CLIPS>(agenda b)
0     r2: f-4
0     r1: f-3
For a total of 2 activations.

CLIPS>(get-focus)
a

CLIPS>(run 1)
FIRE 1 r1: f-2
hechoa 1

CLIPS>(agenda MAIN)

CLIPS>(agenda a)
0     r2: f-1
For a total of 1 activation.

CLIPS>(agenda b)
0     r2: f-4
0     r1: f-3
For a total of 2 activations.

CLIPS>(get-focus)
a

CLIPS>(run 1)
FIRE 1 r2: f-1
cambio de foco.
<== Focus a to b

CLIPS>(get-focus)
b

CLIPS>(agenda a)

CLIPS>(agenda b)
0     r2: f-4
0     r1: f-3
For a total of 2 activations.

CLIPS>(agenda MAIN)

CLIPS>(run 1)
FIRE 1 r2: f-4
He pasado a b2.

CLIPS>(run 1)
FIRE 1 r1: f-3
He pasado a b1.
<== Focus b to MAIN
<== Focus MAIN

CLIPS>(get-focus)
FALSE

CLIPS>(dribble-off)
```

5. VARIABLES GLOBALES: construcción *defglobal*

La construcción *defglobal* permite definir variables que sean visibles globalmente en todo el entorno de CLIPS. Es decir, una variable global puede ser accedida desde cualquier punto en CLIPS y retiene su valor independientemente de otras construcciones.

(Sin embargo, las variables globales en CLIPS están tipadas débilmente. Esto significa que no están restringidas a contener un valor de un tipo de datos determinado).

Las variables globales se pueden acceder como parte del proceso de *pattern-matching*, pero el hecho de modificarlas no invoca tal proceso.

Sintaxis: `(defglobal [<nombre-módulo>] <asignación>*)`

`<asignación> ::= ?*<nombre-variable-global>* = <expresión>`

donde `<nombre-módulo>` indica el módulo en el que se define(n) la(s) variable(s) global(es). Si no se especifica, las definiciones globales se colocarán en el módulo actual. Si se usa una variable ya definida anteriormente con una construcción *defglobal*, su valor se reemplazará con el valor especificado en la nueva construcción *defglobal*.

Para establecer el valor de una variable se utiliza la función *bind*.

Las variables globales recuperan su valor inicial cuando:

- se utiliza la función *bind* con la variable global sin especificar un valor, ó
- se ejecuta un comando *reset*.

Este comportamiento, que es el que presenta CLIPS por defecto, se puede modificar mediante la siguiente función:

Sintaxis: `(set-reset-globals TRUE | FALSE)`
(por defecto se supone `TRUE`, que corresponde al comportamiento por defecto)

Los siguientes comandos permiten eliminar las variables globales del entorno:

- el comando *clear*
- el comando *undefglobal*:

  Sintaxis: `(undefglobal <nombre-variable-global>)`

  Nota: La variable global se especifica sin el '`?`' y sin los asteriscos. Además la variable no debe estar actualmente en uso. En caso contrario, se produce un error. Se puede usar el símbolo '`*`' para eliminar todas las variables globales, a menos que exista una variable global cuyo nombre sea `*`, en tal caso se elimina ésta, y no todas. Esta función no devuelve ningún valor.

Se puede visualizar el valor de una variable global introduciendo su nombre en el *prompt*, ó mediante el siguiente comando:

Sintaxis: `(show-defglobals [<nombre-módulo>])`

Muestra el valor de todas las variables globales del módulo especificado, o del actual, si no se especifica. Si se da como nombre de módulo el símbolo '*', presenta todas las variables (y sus valores) de todos los módulos.

Ejemplos:

```
>(defglobal ?*A* = 12)
>?*A*
12
>(defglobal ?*B* = 1 ?*C* = 2 ?*D* = 3)
>?*B*
1
>(undefglobal ?*B*)
ERROR ;;; de sintaxis
>(undefglobal B) ;;; no se deben especificar los asteriscos ni el símbolo '?'
>?*B*
ERROR ;;; variable no definida. Acaba de ser eliminada.
>?*C*
2
>(defglobal ?*A* = "EJEMPLO") ;;; se machaca el valor y "cambia" el tipo de la variable A
"EJEMPLO"
>(defmodule M1)
>(defglobal M1 ?*X* = "EQUIS")
>(defglobal ?*y* = Reloj) ;;; se supone M1, porque es el actual.
>(defglobal ?*Zeta* = "Letra zeta") ;;; se supone M1, porque es el actual.
>(show-defglobals *)
MAIN:
   ?*A* = "EJEMPLO"
   ?*C* = 2
   ?*D* = 3
M1:
   ?*X* = "EQUIS"
   ?*y* = Reloj
   ?*Zeta* = "Letra zeta"
>(defglobal ?*y* = 85.017) ;;; se machaca el valor y "cambia" el tipo de la variable y
>(show-defglobals M1)
?*X* = "EQUIS"
?*y* = 85.017
?*Zeta* = "Letra zeta"
>(defglobal MAIN ?*y* = "Y GRIEGA")
>(show-defglobals *)
MAIN:
   ?*A* = "EJEMPLO"
   ?*C* = 2
   ?*D* = 3
   ?*y* = "Y GRIEGA"
M1:
   ?*X* = "EQUIS"
   ?*y* = 85.017
   ?*Zeta* = "Letra zeta"
>?*C* ;;; Intento de acceso a una variable perteneciente a un módulo desde otro distinto.
ERROR ;;; variable no definida. El módulo actual es B, y C no es visible en este módulo.
```

Para que un módulo *A* pueda acceder a una variable global de otro módulo *B*, este último debe exportarla, y *A* debe importarla.

6. PROGRAMACIÓN ORIENTADA A OBJETOS EN CLIPS: COOL

6.1 Introducción. ¿Qué es COOL?

Objeto:

Colección de información (los datos) que representa una entidad del mundo real y la forma de manipularla (el código asociado). Los objetos vienen descritos por sus propiedades y conducta.

Propiedades: Especificadas en términos de los atributos (*slots*) de la clase del objeto.

Conducta:

Especificada en términos de código procedural (gestores de mensajes) asociado a la clase del objeto.

NOTA: La mayoría de los sistemas de POO ofrecen esta conducta bien mediante el envío de mensajes (ej. Smalltalk) o mediante funciones genéricas (ej. CLOS). CLIPS ofrece ambos mecanismos, aunque las funciones genéricas no pertenecen estrictamente al LOO de CLIPS. Sin embargo, el hecho de que CLIPS ofrezca ambos mecanismos puede llevar a confusiones en la terminología. En los sistemas OO que soportan sólo envío de mensajes, se utiliza el término "*método*" para designar ese código procedural que responde a mensajes. En aquellos otros sistemas OO que soportan únicamente las funciones genéricas, también se denomina "*método*" a las diferentes implementaciones de una función genérica para distintos conjuntos de restricciones de parámetros. Por tanto, para evitar tal confusión, en la terminología de CLIPS se ha optado por denominar gestor de mensajes al código procedural que responde al envío de un mensaje, y dejar el término método (*method*) para designar a los diferentes códigos que implementan una función genérica.

Los objetos se manipulan mediante el **paso (envío) de mensajes**.

Principios de la programación orientada a objetos

Abstracción:

Representación intuitiva de mayor nivel para un concepto complejo.
Definición de nuevas clases para describir las propiedades comunes y el comportamiento de un grupo de objetos.

Encapsulación:

Mecanismo que permite la ocultación de los detalles de implementación de un objeto mediante el uso de un interfaz externo bien definido.

Herencia: Proceso por el cual una clase puede ser definida en términos de otra(s) clase(s).

Polimorfismo:

Habilidad de diferentes objetos para responder al mismo mensaje de una forma especializada.

Ligadura dinámica (*dynamic binding*):

Capacidad para seleccionar en tiempo de ejecución el gestor de mensajes que corresponde a un mensaje concreto.

COOL: CLIPS Object-Oriented Language

Proporciona mecanismos de abstracción de datos y representación del conocimiento.

Es un híbrido de las características de varios lenguajes orientados a objetos y nuevas ideas:
- ⇨ encapsulación similar a la de SmallTalk,
- ⇨ reglas de herencia múltiple similares a las de CLOS (Common Lisp Object System).

La definición de nuevas clases permite la abstracción de nuevos tipos de datos. Los *slots* y los gestores de mensajes de las clases describen las propiedades y la conducta de un nuevo grupo de objetos.

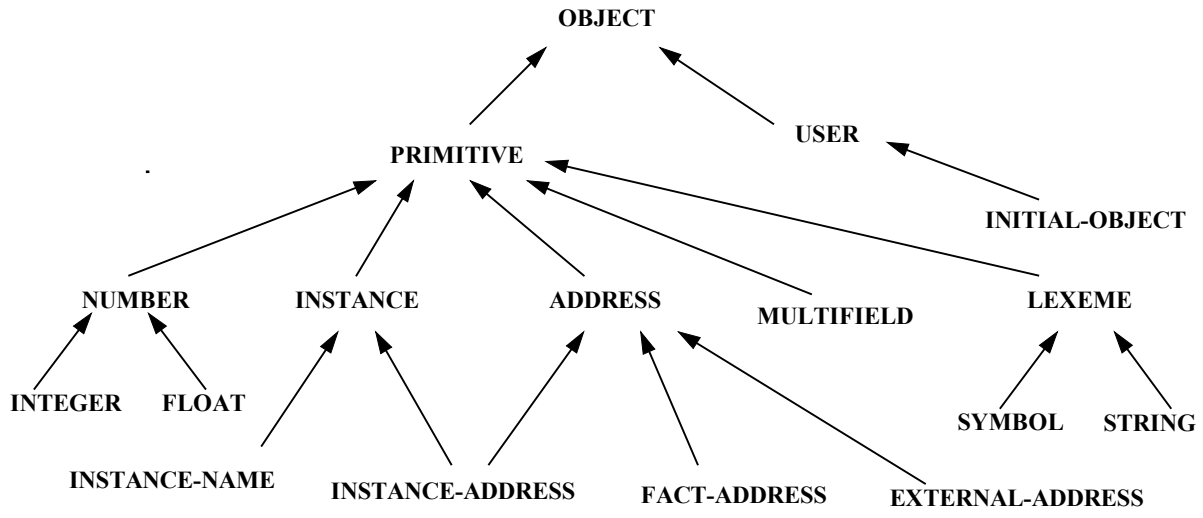
COOL soporta la encapsulación: requiere el paso de mensajes para manipular las instancias de las clases definidas por el usuario. Una instancia no puede responder a un mensaje para el que no tenga definido el correspondiente gestor de mensajes.

COOL permite especificar las propiedades y la conducta de una clase en términos de una o más superclases no relacionadas: **herencia múltiple**.

Para cada nueva clase, COOL utiliza la jerarquía de clases existente para establecer un orden lineal llamado **lista de precedencia de clases**. Las instancias de estas clases pueden heredar propiedades (*slots*) y conducta (gestores de mensajes) de cada una de las clases en la lista de precedencia.

6.2 Clases predefinidas por el sistema

COOL proporciona 17 clases predefinidas que no pueden ser ni eliminadas ni modificadas por el usuario. El esquema de la jerarquía de las clases es el siguiente:



Excepto `INITIAL-OBJECT`, todas son clases abstractas: clases que únicamente se utilizan por razones de herencia y no permiten instancias directas.

Ninguna de estas clases tiene *slots*, y excepto `USER`, ninguna tiene gestores de mensajes. Sin embargo, el usuario puede definir para estas clases sus propios gestores de mensajes, excepto para `INSTANCE`, `INSTANCE-ADDRESS` e `INSTANCE-NAME`.

6.3 Definición de clases

6.3.1 Construcción *defclass*

Se utiliza para especificar las propiedades o atributos (*slots*) y la conducta (gestores de mensajes) de una clase de objetos.

Sintaxis:
```
(defclass <nombre> [<comentario>]
      (is-a <nombre-de-superclase>+)
      [(role concrete | abstract)]
      [(pattern-match reactive | non-reactive)]
      <slot>* ;;; definición de los atributos de la clase
      <documentación-handler>*)
```

Al redefinir una clase existente, se eliminan todas las subclases actuales y todos los gestores de mensajes asociados. Se producirá un error si existen instancias de la clase o de alguna de las subclases.

6.3.2 Herencia múltiple: reglas

Si la clase *A* hereda de la clase *B*, entonces se dice que *A* es una **subclase** de *B*, y *B* es una **superclase** de *A*.

Toda clase definida por el usuario debe tener al menos una superclase directa.

La herencia múltiple ocurre cuando una clase tiene más de una superclase directa.

COOL examina cada nueva clase utilizando una lista de superclases para establecer un orden lineal. Tal lista se denomina **lista de precedencia de clases**. La nueva clase hereda *slots* y gestores (conducta) de cada una de las clases de la lista de precedencia. "Precedencia" significa que las propiedades y conducta de una clase que está primero en la lista, elimina (anula) las definiciones conflictivas de alguna otra clase que pueda aparecer más tarde en la lista. Una clase que aparezca antes que otra en la lista se dice que es más **específica**. Todas las listas de precedencia terminan en la clase del sistema `OBJECT`, y la mayoría (si no todas) de las listas de clases definidas por el usuario terminarán en las clases `USER` y `OBJECT`.

La lista de precedencia de clases se puede listar mediante la función *describe-class*: esta función presenta una descripción detallada de una clase. No devuelve ningún valor.

Sintaxis:
```
(describe-class <nombre-de-clase>)
```

Reglas de herencia múltiple

COOL utiliza las siguientes reglas para determinar la lista de precedencia de una nueva clase:

1. Una clase tiene mayor precedencia que cualquiera de sus superclases.
2. Una clase especifica la precedencia entre sus superclases directas.

Si más de una lista de clases de precedencia satisficiese estas dos reglas, COOL elige la que más se asemeje a un recorrido preorden en profundidad.

Ejemplos:

```
(defclass A (is-a USER))
```
⇨ La clase `A` hereda información directamente de la clase `USER`.
⇨ La lista de precedencia de `A` sería: `A USER OBJECT`

```
(defclass B (is-a USER))
```
⇨ La clase `B` hereda información directamente de la clase `USER`.
⇨ La lista de precedencia de `B` sería: `B USER OBJECT`

```
(defclass C (is-a A B))
```
⇨ La clase `C` hereda información directamente de las clases `A` y `B`.
⇨ La lista de precedencia de `C` sería: `C A B USER OBJECT`

```
(defclass D (is-a B A))
```
⇨ La clase `D` hereda información directamente de las clases `B` y `A`.
⇨ La lista de precedencia de `D` sería: `D B A USER OBJECT`

```
(defclass E (is-a A C))
```
⇨ **Error**: por la regla 2. `A` debe preceder a `C`. Sin embargo, como `C` es una subclase de `A`, entonces la lista de precedencia sería (`A C A`) en la que existe un bucle.

```
(defclass E (is-a C A))
```
⇨ Decir que `E` herede de `A` es un poco raro, ya que `C` hereda de `A`. Sin embargo es legal.
⇨ La lista de precedencia de `E` sería: `E C A B USER OBJECT`

```
(defclass F (is-a C B))
```
⇨ Decir que `F` herede de `B` es un poco raro, ya que `C` hereda de `B`. Sin embargo es legal.
⇨ La lista de precedencia de `F` sería: `F C A B USER OBJECT`. Obsérvese que la lista de superclases en el *defclass* de `F` dice que `B` debe seguir a `C` en la lista de clases de precedencia de `F`, pero no que deba seguirle inmediatamente.

```
(defclass G (is-a C D))
```
⇨ **Error.**

Ejercicio: ¿Cuál sería la lista de clases de precedencia en los siguientes casos?

1. `(defclass H (is-a A))`
2. `(defclass I (is-a B))`
3. `(defclass J (is-a H I A B))`
4. `(defclass K (is-a H I))`

6.3.3 Especificadores de clase

Clases abstractas y clases concretas: `(role concrete | abstract)`

- Una clase abstracta sólo se utiliza para la herencia; no se permite crear instancias directas de este tipo de clases.
- Una clase concreta permite crear instancias de ella.

Si no se especifica ni *abstract* ni *concrete* para una clase, éste se determinará por herencia.

Clases reactivas y no reactivas: `(pattern-match reactive | non-reactive)`

- Los objetos de una clase reactiva pueden *matchear* con los elementos objeto (*object pattern*) de las reglas.
- Los objetos de una clase no reactiva no pueden *matchear* con los elementos objeto (*object pattern*) de las reglas.
- Una clase abstracta no puede ser reactiva.

Si no se especifica ni *reactive* ó *non-reactive* para una clase, éste se determinará por herencia.

6.3.4 Atributos (*slots*) y propiedades

Los atributos de una clase, denominados ***slots***, son lugares para valores asociados con instancias de clases definidas por el usuario. Se puede especificar si un atributo es heredable o no.

Cada instancia tiene una copia del conjunto de *slots* especificados por las clases inmediatas y los obtenidos por herencia. El número de *slots* queda limitado por la cantidad de memoria disponible.

Para determinar el conjunto de *slots* de una instancia, se examina la lista de precedencia de más específico a más general (es decir, de izda. a dcha.). Una clase es más específica que sus superclases.

Ej.:
```
(defclass A (is-a USER)          (defclass B (is-a A)
      (slot colorA)                    (slot colorB)
      (slot alturaA))                  (slot alturaB))
```

Las instancias de `A` tendrán dos *slots*: `colorA` y `alturaA`. Las instancias de `B` tendrán cuatro *slots*: `colorB`, `alturaB`, `colorA`, `alturaA`.

Tipo de campo del *slot*.

```
(slot <nombre> <facet>*)
(single-slot <nombre> <facet>*)
(multislot <nombre> <facet>*)
```

Propiedades de los atributos (*facets*).

Así como los atributos (*slots*) forman una clase, las *facets* forman un atributo. Las *facets* son las propiedades de los atributos, las cuales son ciertas para todos los objetos que comparten dichos *slots*. Las posibles *facets* son: `default`, `storage`, `access`, `propagation`, `source`, `pattern-match`, `visibility`, `create-accessor`, `override-message` y `constraint`.

a) **`default`** y **`default-dynamic:`** contiene(n) el(los) valor(es) inicial(es) por defecto para un atributo, si no se da uno explícitamente.
   - La propiedad `default` es estática, esto es, la expresión se evalúa una sola vez cuando se define la clase y a continuación se almacena para la clase. Por defecto se supone `?DERIVE`, que hace que el valor del atributo se obtenga por derivación a partir de sus propiedades. Si se especifica `?NONE`, no se asigna ningún valor por defecto al atributo, y se requerirá uno al crear las instancias.
   - La propiedad `default-dynamic` es dinámica, esto es, la expresión dada se evalúa cada vez que se crea una instancia, y luego se almacena en el atributo.

Sintaxis: `(default ?DERIVE | ?NONE | <expresión>*)`
`(default-dynamic <expresión>*)`

Ej.:
```
(defclass camara (is-a USER)
      (role concrete)
      (slot numero-serie (default-dynamic (gensym)))
      (slot cargada (default no))
      (slot tapa-cierre (default cerrada))
      (slot seguro (default puesto))
      (slot num-fotos-sacadas (default 0)))

CLIPS> (make-instance mi-camara of camara)
[mi-camara]

CLIPS> (make-instance camara2 of camara)
[camara2]

CLIPS> (send [mi-camara] print)
[mi-camara] of camara
(numero-serie gen1)
(cargada no)
(tapa-cierre cerrada)
(seguro puesto)
(num-fotos-sacadas 0)

CLIPS> (send [camara2] print)
[camara2] of camara
(numero-serie gen2)
(cargada no)
(tapa-cierre cerrada)
(seguro puesto)
(num-fotos-sacadas 0)
```

b) **`storage:`** permite especificar si el valor del atributo se almacena en la instancia (`local`, por defecto) ó bien en la propia clase (`shared`). Si el valor del atributo se almacena localmente, entonces cada instancia tendrá su propio valor del atributo; sin embargo, si es compartido (almacenado en la clase), todas las instancias tendrán el mismo valor para ese atributo. Además, si se modifica el valor de ese atributo, se modificará para todas las instancias.

Sintaxis: `(storage local | shared)`

Ej.: Supongamos la definición siguiente:

```
(defclass camion-volvo (is-a USER)
      (role concrete)
      (slot motor     (create-accessor write)
                      (storage shared)
                      (default v207))
      (slot ruedas    (create-accessor write)
                      (storage shared)
                      (default-dynamic 10))
      (slot carga     (create-accessor write)
                      (storage local)))

CLIPS> (make-instance camion1 of camion-volvo)
[camion1]
CLIPS> (send [camion1] print)
[camion1] of camion-volvo
(motor v207)
(ruedas 10)
(carga nil) ;;; Se ha supuesto para carga el tipo SYMBOL, cuyo valor por defecto es nil
CLIPS> (send [camion1] put-motor v300)
v300  ;;; se modifica el valor en la definición de la clase
CLIPS> (send [camion1] put-ruedas 16)
16
CLIPS> (make-instance camion2 of camion-volvo)
[camion2]  ;;; En este momento, ya se han recalculado todos los valores por defecto
           ;;; de los atributos compartidos.
CLIPS> (send [camion2] print)
[camion2] of camion-volvo
(motor v300)
(ruedas 10)
(carga nil)
CLIPS> (send [camion2] put-motor v350)
v350  ;;; valor que compartirán ambas instancias
CLIPS> (send [camion2] put-carga fruta)
fruta  ;;; valor sólo para la instancia [camion2]
CLIPS> (send [camion1] print)
[camion1] of camion-volvo
(motor v350)
(ruedas 10)
(carga nil)
CLIPS> (send [camion2] print)
[camion2] of camion-volvo
(motor v350)
(ruedas 10)
(carga fruta)
```

c) **access:** establece el tipo de acceso permitido sobre un *slot*. El acceso puede ser: `read-write`, `read-only` e `initialize-only`. Por defecto, se supone el acceso `read-write` y significa que el *slot* puede ser leído y modificado. El acceso `read-only` indica que el *slot* solamente puede ser leído; la única forma de establecer este tipo de acceso es mediante la propiedad `default` en la definición de la clase. Por último, `initialize-only` es como `read-only` excepto que se le puede dar un valor sólo en el momento de crear una instancia mediante `make-instance`.

Sintaxis: `(access read-write | read-only | initialize-only)`

Ej.:
```
CLIPS>(defclass camión-volvo (is-a USER)
           (role concrete)
           (slot motor     (access read-only)
                           (default v350))
           (slot ruedas    (create-accessor write)
                           (access initialize-only))
           (slot carga     (create-accessor write)
                           (access read-write)))

CLIPS>(defmessage-handler camión-volvo put-motor (?valor)
CLIPS>(make-instance camion1 of camion-volvo (carga frutas)
(ruedas 16))
[camion1]
CLIPS>(send [camion1] put-motor v250)
ERROR motor slot in [camion1] of camion-volvo: write access denied.
FALSE
CLIPS> (send [camion1] put-ruedas 10)
ERROR ruedas slot in [camion1] of camion-volvo: write access denied.
FALSE
CLIPS> (send [camion1] print)
[camion1] of camion-volvo
(motor v350)
(ruedas 16)
(carga frutas)
```

d) **propagation:** esta propiedad indica si el atributo correspondiente es heredable por subclases. Los valores que puede tomar son: `inherit` (valor por defecto) ó `no-inherit`.
El valor `no-inherit` implica que sólo instancias directas de la clase heredarán el atributo (*slot*).

Sintaxis: `(propagation inherit | no-inherit)`

Ej.:
```
CLIPS>(defclass deposito (is-a USER)
           (role concrete)
           (slot capacidad (propagation inherit))
           (slot estado (propagation no-inherit))
           (slot contiene-litros (propagation inherit)))

CLIPS>(defclass deposito-viejo (is-a deposito))
CLIPS>(make-instance deposito1 of deposito)
[deposito1]
CLIPS>(make-instance deposito-viejo1 of deposito-viejo))
[deposito-viejo1]
CLIPS>(send [deposito1] print)
[deposito1] of deposito
(capacidad nil)
(estado nil)
(contiene-litros nil)
CLIPS>(send [deposito-viejo1] print)
[deposito-viejo1] of deposito-viejo
(capacidad nil)
(contiene-litros nil)
```

e) **source:** (origen de otras *facets*) al obtener los atributos de la lista de precedencia de clases durante la creación de una instancia, por defecto se toman las *facets* de la clase más específica y se dan los valores por defecto a las *facets* no especificadas. Los valores para esta propiedad son: `exclusive` y `composite`. El valor `exclusive` corresponde al comportamiento por defecto explicado anteriormente. El efecto del valor `composite` es que las propiedades no especificadas explícitamente por la clase más específica se toman de la siguiente clase más específica.

Sintaxis: `(source exclusive | composite)`

f) **pattern-match:** sirve para especificar si cualquier cambio en un atributo de una instancia se considerará por el proceso de *pattern-matching*. Los valores de esta propiedad son: `reactive` (valor por defecto: sí se considera un cambio) ó `non-reactive` (no se considera el cambio).

Sintaxis: `(pattern-match reactive | non-reactive)`

Ejemplo:

```
(defclass camión (is-a USER)
      (role concrete)
      (pattern-match reactive)
      (slot carga (create-accessor write)
                  (pattern-match reactive)))
```

;;; El "*pattern-match*" del atributo *carga* no es necesario, pues el primer *pattern-match*, ;;; al no ir asociado a ningún atributo concreto, se asociará con todos los atributos de la ;;; clase.

```
(defclass coche (is-a USER)
      (role concrete)
      (pattern-match reactive)
      (slot carga (create-accesor write)
                  (pattern-match non-reactive)))
```

;;; modificaciones en el atributo *carga* de esta clase no provocarán cambios en ;;; la agenda (base de reglas).

```
(defrule crear
      ?instancia <- (object (is-a coche | camión))
      =>
      (printout t "Crear " (instance-name ?instancia) crlf))

(defrule acceso-carga
      ?instancia <- (object (is-a coche | camión) (carga ?))
      =>
      (printout t "Acceso-carga " (instance-name ?instancia) crlf))

CLIPS>(make-instance camión1 of camión)
[camión1]
CLIPS>(make-instance coche1 of coche)
[coche1]
CLIPS>(run)
Crear [coche1]
Crear [camión1]
Acceso-carga [camión1]
CLIPS>(send [camión1] put-carga fruta)
fruta
CLIPS>(send [coche1] put-carga fruta)
fruta
CLIPS>(run)
```

```
Acceso-carga [camión1]
```

g) **visibility:** determina la visibilidad de los atributos a los gestores de mensajes de las subclases o superclases. Por defecto, sólo pueden acceder a los atributos definidos en una clase los gestores de mensajes ligados a la propia clase (`private`). Es decir, por defecto la visibilidad de un atributo se supone `private`. Sin embargo, también es posible permitir dicho acceso a los gestores de mensajes de superclases o subclases que heredan el atributo (especificando el valor `public`).

Sintaxis: `(visibility private | public)`

Ejemplo:

```
>(defclass persona (is-a USER)
      (slot edad (visibility private)))
            ;;; el atributo edad no puede ser accedido
            ;;; por gestores que no estén asociados a la clase persona.

>(defclass hombre (is-a persona)
      (role concrete))
      ;;; CLIPS no permitirá definir un gestor asociado a la clase hombre que intente
acceder
      ;;; al atributo edad de la clase persona. Si se intenta, se producirá un error.
```

h) **create-accessor:** el usuario debe definir explícitamente los gestores de mensajes necesarios para acceder (leer y/o escribir) a cada atributo. Esta propiedad hace que CLIPS los cree automáticamente. Por defecto, no se crean. Además, estos gestores de mensajes no pueden ser modificados por el usuario.

Sintaxis: `(create-accessor ?NONE | read | write | read-write)`

Ejemplo:

```
>(defclass persona (is-a USER)
      (slot concrete)
      (slot empresa (create-accessor write))
      (slot edad))
            ;;; Si al crear una instancia se intenta dar un valor inicial al atributo edad,
            ;;; el sistema generará un error porque no existe ningún gestor asociado a la
            ;;; modificación de este atributo. Con esta definición, es imposible acceder al
            ;;; atributo edad.
```

i) **override-message:** Esta propiedad permite al usuario definir sus propios mensajes de acceso a los atributos, de forma que puedan ser utilizados directamente por las funciones COOL. Por defecto, estos mensajes tienen la sintaxis `put-<nombre-slot>`. Con esta propiedad, se puede definir otro formato distinto.

Sintaxis: `(override-message ?DEFAULT | <nombre-mensaje>)`

Ejemplo:

```
>(defclass persona (is-a USER)
      (role concrete)
      (slot nombre (override-message nombre-put)))
      ;;; El mensaje que se debe usar para modificar el atributo nombre debe ser nombre-
      put.
```

j) **constraint:** COOL permite una serie de restricciones sobre los atributos (tipo, rango, cardinalidad) que pueden ser comprobadas de forma estática (por defecto) y/o de forma dinámica (hay que activarlo explícitamente[4]) sobre las clases y sus instancias.

- comprobación estática: se realiza cuando se analizan (*parse*) construcciones o comandos que especifican información sobre atributos.
- comprobación dinámica: se comprueban los valores de nuevas instancias cada vez que se modifican.

Ej.:

```
>(defclass persona (is-a USER)
      (role concrete)
      (multislot empleos    (create-accessor write)
                            (type SYMBOL)
                            (cardinality 0 2)))
>(set-dynamic-constraint-checking TRUE)
FALSE  ;;; activa la comprobación dinámica (es decir, en tiempo de ejecución) de restricciones.
       ;;; Devuelve el valor que tuviese anteriormente.
>(make-instance Ana of persona)
[Ana]
>(make-instance Maria of persona (empleos medico))
[Maria]
>(make-instance Ana of persona (empleos 1.5 medico))
ERROR slot empleos does not match the allowed types.
FALSE
>(instances) ;;; Lista las instancias actualmente en la MT
[Maria] of persona
For a total of 1 instance. ;;; Atención: la instancia antigua Ana se ha eliminado.
>(make-instance Pepe of persona (empleos actor poeta piloto)))
ERROR slot empleos does not satisfy the cardinality restrictions.
FALSE
```

6.3.5 Proceso de *pattern-matching* con objetos

Al igual que los hechos, las instancias de clases definidas por el usuario también pueden hacerse corresponder (hacer "*matching*") con patrones en la LHS de las reglas. Estos patrones solamente pueden emparejarse ("*matchear*") con objetos (instancias) de clases ya existentes y que estén en el ámbito (es decir, que sean visibles) del módulo actual.

Las modificaciones en atributos no reactivos o en instancias de clases no reactivas no tendrán efecto alguno en las reglas.

Sintaxis: `(object <atributo-de-restricción>)`

```
<atributo-de-restricción> ::=
                        (is-a <restricción>) |
```

[4]Se activa mediante la función `(set-dynamic-constraint-checking TRUE)`

```
                    (name <restricción>) |
                    (<nombre-atributo> <restricción>*)
```

- El atributo *is-a* se utiliza para especificar restricciones de clase del estilo "¿Es este objeto miembro de la clase *C*?"
- El atributo *name* se utiliza para referirse a una instancia específica, concreta con un nombre determinado, esto es, contiene el nombre de la instancia. La evaluación de este atributo debe ser de tipo `INSTANCE-NAME`, no un símbolo. El nombre de una instancia va encerrado entre corchetes ([]). No se pueden usar restricciones multicampo (es decir, el símbolo `$?`) con *is-a* o *name*.
- Las restricciones utilizadas con atributos de objetos (instancias) se usan de igual forma que con los atributos de *deftemplate*s.

Se debe utilizar el EC "`<-`" para ligar una variable con la dirección en la MT de una instancia, de forma similar a como se hace con *deftemplate*s:

Ejemplo: La siguiente regla, perteneciente a un SE que controla un sistema de calefacción doméstico, ilustra el uso de los atributos anteriores.

```
(defrule UNIDAD-CONTROL::UC7
     "Apaga la caldera si la habitación está vacía"
     (object (is-a Termostato) (temperatura ?t)
          (temperatura-seleccionada ?ts & :(>= ?t (- ?ts 3)))
          (caldera ?c) (modo calor) (habitación ?h))
     ?inst <- (object (is-a Caldera) (name ?c)
                      (estado encendida))
     (object (is-a Habitación) (name ?h) (ocupación vacía))
     =>
     (send ?inst Apagar))
```

6.3.6 Documentación de gestores de mensajes

COOL permite al usuario declarar por anticipado los gestores de mensajes de una clase dentro de la construcción *defclass*. El propósito de estas declaraciones es sólo para documentación y son ignoradas por CLIPS. No es necesario declarar previamente los gestores de una clase para poder definirlos y asociarlos a una clase.

En cualquier caso, para definir un gestor se debe utilizar el constructor *defmessage-handler*.

```
Ej.:  (defclass rectángulo (is-a USER)
           (role concrete)
           (slot base (create-accessor read-write) (default 1))
           (slot altura (create-accessor read-write) (default 1))
           (message-handler calcular-área))  ;;; declaración, no definición
```

6.4 Gestores de mensajes (*message handlers*): creación.

Los objetos se manipulan mediante el envío de mensajes utilizando la función `send`.

La ejecución de un mensaje produce como resultado un valor o un efecto lateral. La implementación de un mensaje consta de una serie de fragmentos de código procedural llamados gestores de mensajes (*message-handlers*).

La construcción `defmessage-handler` se utiliza para especificar la conducta de los objetos de una clase en respuesta a un mensaje particular.

Dentro de una clase, los gestores para un mensaje particular pueden ser de cuatro tipos diferentes:

Tipo	Papel que juega para la clase
primary	Realiza la mayoría del trabajo del mensaje.
before	Realiza trabajo auxiliar para un mensaje antes de que se ejecute el gestor primario (usado sólo para efectos laterales; se ignora el valor producido).
after	Realiza trabajo auxiliar para un mensaje después de que se ejecute el gestor primario (usado sólo para efectos laterales; se ignora el valor producido).
around	Inicializa un entorno para la ejecución del resto de los gestores.

El gestor primario proporciona la parte de la implementación del mensaje que es más específica al objeto (el definido en la clase más cercana anula los de las clases superiores).

Un mensaje utiliza todos los gestores *before* o *after* de todas las clases de un objeto. Esto quiere decir que si se envía un determinado mensaje a una clase *C*, se ejecuta el gestor primario de *C*, y todos los gestores *before* y *after* de sus superclases → forma declarativa.

Cuando el tipo de gestor por sí solo especifica los gestores que se ejecutan y su orden de ejecución, se dice que el mensaje está implementado de forma declarativa. En otro caso, se dice de forma imperativa.

Sintaxis:
```
(defmessage-handler <nombre-clase> <nombre-mensaje>
      [<tipo-handler>] [<comentario>]
      (<parámetro>* [<parámetro-comodín>])
      <acción>*)
```

Un gestor de mensajes consta de 7 partes:

1. un nombre de clase (ya definida) a la que el gestor estará asociado;
2. el nombre del mensaje al que el gestor responderá;
3. el tipo de gestor (por defecto se supondrá `primary` si no se especifica);
4. un *string* de comentario (opcional);
5. una lista de parámetros que se pasará al gestor durante la ejecución;
6. un parámetro comodín (para gestionar múltiples parámetros), y
7. una secuencia de acciones o expresiones que serán ejecutadas por el gestor.

- El valor devuelto por un gestor de mensajes es el resultado de la evaluación de la última acción.
- Los gestores de mensajes se identifican unívocamente por la clase, el nombre, y su tipo.

- Nunca se llama directamente a un gestor de mensajes.

Ej.: Partiendo del anterior:

```
>(defmessage-handler rectángulo calcular-área ()
      (* ?self:base ?self:altura)) ;;; Devuelve el área (base por altura).

>(defmessage-handler rectángulo imprimir-área ()
      (printout t (send ?self calcular-área) crlf))
```

> Todos los gestores de mensajes tienen un parámetro implícito llamado **?self**, que contiene la instancia activa[5] para ese mensaje. Este parámetro es reservado y, por tanto, ni puede aparecer en la lista de parámetros del gestor, ni su valor puede ser machacado dentro del cuerpo del gestor.

Ej.:

```
>(defclass A (is-a USER) (role concrete))
>(defmessage-handler A borrar before ()
      (printout t "Borrando una instancia de la clase A..."
      crlf))
>(defmessage-handler A borrar ()
      (send ?self delete))
>(defmessage-handler USER borrar after ()
  (printout t "El sistema terminó el borrado de una instancia" crlf))
>(watch instances) ;;; activa la traza de instancias
>(make-instance a of A) ;;; crea la instancia a
==> instance [a] of A
[a] ;;; Los nombres de instancias aparecen entre corchetes
>(send [a] borrar) ;;; envía el mensaje "borrar" a la instancia a
Borrando una instancia de la clase A...
<== instance [a] of A ;;; efecto lateral debido a la ejecución del gestor primary
El sistema terminó el borrado de una instancia
TRUE
>(unwatch instances) ;;; desactiva la traza de instancias
```

6.4.1 Parámetros

El número de parámetros de un gestor de mensajes puede ser fijo o variable, dependiendo de que se proporcione un parámetro comodín. Un parámetro comodín es una variable multicampo:

Sintaxis: `$?<nombre-variable>`

Los parámetros actuales asignados al parámetro comodín se tratan como un valor multicampo, y el número de éstos debe ser mayor o igual que el mínimo aceptado.

Ej.:

```
>(defclass COCHE (is-a USER)
```

[5] El término *instancia activa* se refiere a la instancia que en ese momento está respondiendo a un mensaje.

```
      (role concrete)
      (slot asiento-delantero)
      (multislot maletero)
      (slot número-bultos))

>(defmessage-handler COCHE meter-en-el-coche (?ítem $?resto)
      ...) ;;; definición del cuerpo más adelante.
```

6.4.2 Acciones

El cuerpo de un gestor de mensajes es una secuencia de expresiones que se ejecutan en el orden definido.

Las acciones de los gestores tienen acceso directo a los atributos: son considerados como parte de la encapsulación del objeto.

Se proporciona una abreviatura para acceder a los atributos de una instancia activa desde dentro de un gestor de mensajes.

Sintaxis: `?self:<nombre-slot>` ← sintaxis a usar si el acceso es de lectura

Ej.:

```
(defclass auto (is-a USER)
      (role concrete)
      (slot puertas (default 4))
      (multislot seguridad (default cinturones)))

(defmessage-handler auto imprimir-slots ()
      (printout t ?self:puertas " " ?self:seguridad crlf))
            ;;; esta acción se ejecutará cuando se envía el mensaje imprimir-slots
            ;;; a una instancia de la clase coche.
```

La función `bind` también puede utilizar esta abreviatura para modificar el valor de un atributo:

Sintaxis: `(bind ?self:<nombre-slot> <valor>*)`

sintaxis a usar si el acceso al *slot* es de escritura (modificaciones).

Ej.:

```
>(defmessage-handler COCHE meter-en-el-coche (?ítem $?resto)
      (bind ?self:asiento-delantero ?ítem)
      (bind ?self:maletero ?resto)
      (bind ?self:número-bultos (length$ ?resto)))
>(defmessage-handler auto actualizar-seguridad ($?valores)
      (bind ?self:seguridad $?valores))
            ;;; asigna al atributo seguridad el valor pasado en el parámetro valores.
>(make-instance Renault-19 of auto)
[Renault-19]
>(send [Renault-19] actualizar-seguridad Airbag ABS)
(Airbag ABS)
```

```
>(send [Renault-19] imprimir-slots)
4 (Airbag ABS)
```

6.4.3 Demonios (*daemons*)

Los demonios son fragmentos de código que se ejecutan implícitamente cuando se realiza alguna acción básica sobre una instancia, como inicialización, borrado, ó lectura y escritura de atributos. Todas estas acciones básicas se implementan con gestores primarios asociados a la clase de la instancia.

Los demonios pueden ser fácilmente implementados como gestores de mensajes de tipo *before* o *after*, que reconozcan los mismos mensajes. De esta forma, estos fragmentos de código se ejecutarán siempre que se realice alguna operación básica sobre la instancia.

Ej.: El gestor que se define será invocado cada vez que se cree una instancia y justo antes de inicializar sus atributos.

```
(defclass A (is-a USER) (role concrete))
(defmessage-handler A init before ()
      (printout t
       "Inicializando nueva instancia de la clase A..." crlf))
```

6.4.4 Gestores de mensajes predefinidos

CLIPS define 7 gestores de mensajes primitivos ligados a la clase `USER`. Éstos no pueden ser ni modificados ni borrados.

a) Inicialización de instancias: mensaje `init`

Sintaxis: `(defmessage-handler USER init primary () )`

- Este gestor se encarga de inicializar una instancia con sus valores por defecto después de crearla.
- Las funciones para crear instancias (`make-instance`) e inicializar instancias (`initialize-instance`) envían automáticamente el mensaje `init` a una instancia; el usuario no debe enviar nunca este mensaje directamente.
- Los gestores `init` definidos por el usuario no deben impedir al gestor de mensajes del sistema poder responder a un mensaje `init`.

b) Eliminación de instancias: mensaje `delete`

Sintaxis: `(defmessage-handler USER delete primary () )`

- Se encarga de borrar una instancia.
- El usuario debe enviar directamente un mensaje de borrado a una instancia; los gestores de este tipo definidos por el usuario no deben impedir al gestor de mensajes del sistema responder a un mensaje `delete`.

- Este gestor devuelve `TRUE` si logra borrar con éxito la instancia; devuelve `FALSE` en caso contrario.

c) Visualización de instancias: mensaje `print`

Sintaxis:
```
(defmessage-handler USER print primary () )
```

- Imprime en pantalla los atributos y sus valores de una instancia dada.

d) Modificación directa de instancias: mensaje `direct-modify`

Sintaxis:
```
(defmessage-handler USER direct-modify primary
      (?expresiones-de-anulación-de-slot) )
```

- Modifica directamente los atributos (*slots*) de una instancia.
- Las expresiones se pasan al gestor como un objeto EXTERNAL_ADDRESS.
- Este mensaje (`direct-modify`) lo utilizan las funciones `modify-instance` y `active-modify-instance`.

Ej.: El siguiente gestor de tipo *around* se encargaría de asegurar que todas la expresiones de anulación (sobreescritura) de *slots* se gestionan mediante mensajes *put-*.

```
(defmessage-handler USER direct-modify around (?anulaciones)
     (send ?self direct-modify ?anulaciones))
```

e) Modificación de instancias mediante mensajes: mensaje `message-modify`

Sintaxis:
```
(defmessage-handler USER message-modify primary
      (?expresiones-de-anulación-de-slot) )
```

- Modifica los atributos (*slots*) de una instancia mediante mensajes *put-*... para actualizar un atributo.
- Las expresiones se le pasan como objetos EXTERNAL_ADDRESS.
- El mensaje `message-modify` lo utilizan las funciones `message-modify-instance` y `active-message-modify-instance`.

f) Duplicación directa de instancias: mensaje `direct-duplicate`

Sintaxis:
```
(defmessage-handler USER direct-duplicate primary
      (?nombre-nueva-instancia)
      (?expresiones-de-anulación-de-slot) )
```

- Este gestor duplica una instancia sin usar mensajes *put-* asignando las expresiones de anulación.
- Si el nombre de la nueva instancia creada coincide con el de una existente, ésta última se elimina sin hacer uso de mensajes.
- Las expresiones de anulación se pasan como objetos EXTERNAL_ADDRESS al gestor.

- Este mensaje lo usan las funciones `duplicate-instance` y `active-duplicate-instance`.

g) Duplicación de instancias mediante mensajes: mensaje `message-duplicate`

Sintaxis: `(defmessage-handler USER message-duplicate primary (?expresiones-de-anulación-de-slot) )`

- Este gestor duplica instancias usando mensajes.
- Los valores de los atributos de la instancia original y las expresiones de anulación se copian usando mensajes *get-* y mensajes *put-* (*véase 6.6*).
- Si el nombre de la nueva instancia creada coincide con el de una existente, ésta última se elimina mediante un mensaje `delete`.
- Después de crear la instancia, se la envía un mensaje `init`.
- Las expresiones de anulación se pasan como objetos EXTERNAL_ADDRESS al gestor.
- Este mensaje lo utilizan las funciones `message-duplicate-instance` y la función `active-message-duplicate-instance`.

6.5 *Dispatching*

Cuando un objeto recibe un mensaje mediante la función `send`, CLIPS examina la lista de precedencia de clases para determinar el conjunto de gestores de mensajes aplicables al mensaje.

Un gestor de mensajes es aplicable a un mensaje, si su nombre coincide con el mensaje y el gestor está ligado (asociado) a alguna clase de la lista de precedencia a la cual pertenezca la instancia que recibe el mensaje.

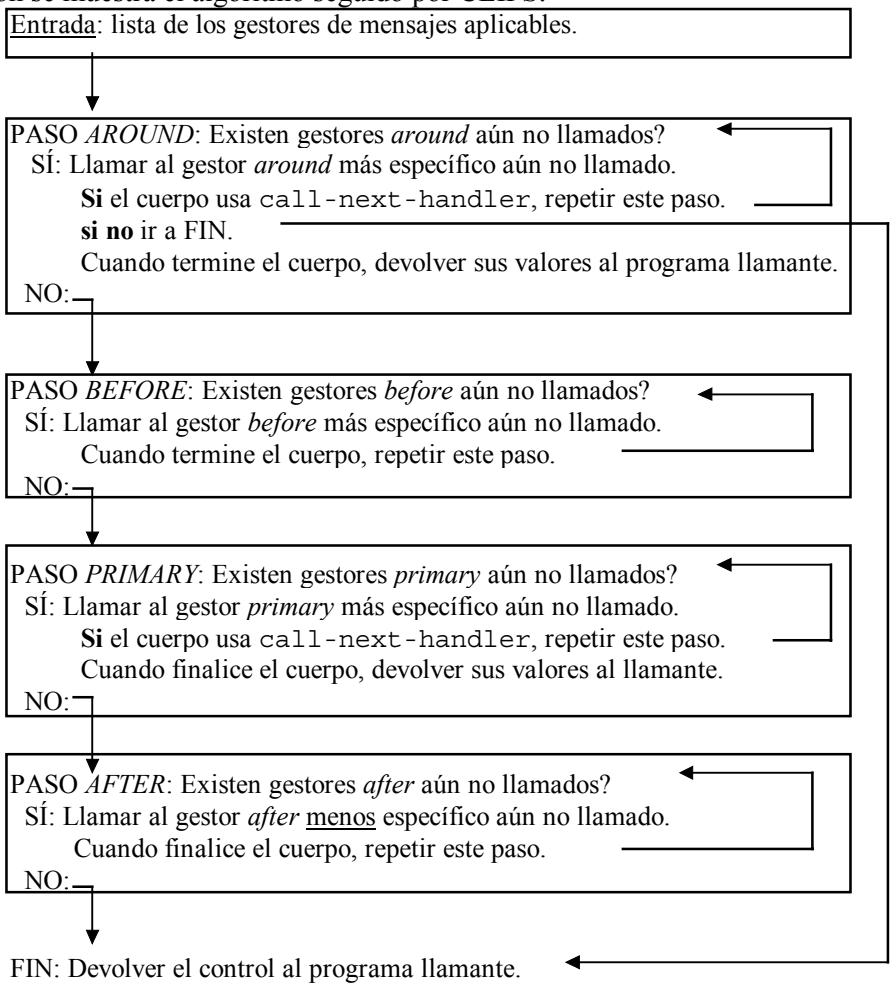
El conjunto de gestores de mensajes aplicables se ordena de acuerdo a su tipo o *role* (el orden es: *around*, *before*, *primary*, *after*), y a continuación se ordena por especificidad de clase:

⇨ los gestores *around*, *before* y *primary* → se ordenan de más especifico a más general
⇨ los gestores *after* → se ordenan de más general a más específico

Es preferible utilizar únicamente los gestores *before, after* y el *primary* más específico (técnica declarativa).

El orden de ejecución es como sigue **1)** los gestores *around* comienzan a ejecutarse desde el más específico al más general (cada gestor *around* debe permitir explícitamente la ejecución a los otros gestores); **2)** se ejecutan los gestores *before* (uno tras otro) de más específico a más general; **3)** se ejecutan los gestores *primary*, desde el más específico al más general (los *primary* más específicos deben permitir explícitamente la ejecución de los más generales); **4)** los gestores *primary* finalizan su ejecución, desde el más general al más específico; **5)** los gestores *after* se ejecutan (uno tras otro) de más general a más específico, y **6)** finaliza la ejecución de los gestores *around* del más general al más específico.

A continuación se muestra el algoritmo seguido por CLIPS:



6.6 Manipulación de instancias: creación, modificación y borrado

Los objetos se manipulan mediante el envío de mensajes de unos a otros. Esto se consigue usando la función ***send***, que toma como argumentos el objeto destino del mensaje, el mensaje mismo, y otros parámetros que debieran ser pasados a los gestores.

Sintaxis: `(send <expresión-de-objeto> <nombre-de-mensaje> <expresión>*)`

Si ocurre cualquier error durante la ejecución de un gestor de mensajes, cualquier otro gestor que se esté ejecutando actualmente será abortado, los gestores que todavía no hubiesen iniciado su ejecución serán ignorados, y la función *send* devolverá `FALSE`.

La función *send* devuelve el valor del último gestor (*primary* o *around*) ejecutado.

Creación de instancias.

Sintaxis: `(make-instance [<nombre-instancia>] of <nombre-clase>`

```
          (<nombre-slot> <expresión>)*)
```

- El valor devuelto por `make-instance` es el nombre de la nueva instancia (si se tuvo éxito), ó el símbolo `FALSE` en caso contrario.
- La evaluación del nombre de la instancia puede ser un nombre de instancia ó un símbolo. Si no se proporciona un nombre de instancia, CLIPS generará uno arbitrario (usando automáticamente la función `gensym*`).

Ej.:
```
(declass A (is-a USER)
     (role concrete)
     (slot x (default 34) (create-accessor write))
     (slot y (default abcd)))
(defmessage-handler A put-x before (?valor)
     (printout t "Slot x actualizado con mensaje." crlf))
(defmessage-handler A delete after ()
     (printout t "Borrada la instancia antigua." crlf))
CLIPS>(make-instance a of A)
[a]  ;;; Los nombres de instancia aparecen entre corchetes siempre, excepto en make-instance
CLIPS>(send [a] print) ;;; envía el mensaje predefinido print
[a] of A
(x 34)
(y abcd)
CLIPS>(make-instance a of A (x 65)) ;;; se machacará la instancia anterior
Borrada la instancia antigua.
Slot x actualizado con mensaje.
[a]
CLIPS>(send [a] print)
[a] of A
(x 65)
(y abcd)
CLIPS>(send [a] delete) ;;; envía el mensaje predefinido delete
Borrada la instancia antigua.
TRUE
```

Construcción *definstances*.

Esta construcción, análoga a la construcción *deffacts*, permite especificar un conjunto de instancias iniciales que actúen como conocimiento inicial (ó conocimiento *a priori*).

Sintaxis:
```
(definstances <nombre-colección-instancias>
     [active] [<comentario>]
     (<definición-de-instancia>)*)
```

¿Cuál es el funcionamiento de esta construcción?

⇨ Cuando se da un comando *reset*, todas las instancias actuales en la MT reciben un mensaje de eliminación, CLIPS la borra, y a continuación se realiza, automáticamente, una llamada a *make-instance* con cada una de las instancias dentro de *definstances*.

Nota: Al inicio de una sesión con CLIPS y cada vez que se ejecuta un comando *clear*, se define automáticamente la siguiente construcción *definstances*:

```
(definstances initial-object
     (initial-object of INITIAL-OBJECT))
```

La clase predefinida `INITIAL-OBJECT` no se puede borrar, pero sí esta construcción.

Reinicialización de instancias existentes.

Sintaxis: `(initialize-instance <nombre-instancia>`
`<expresión-anulación-slot>*)`

- Esta función permite inicializar nuevamente una instancia con los valores por defecto de la clase y nuevos valores de anulación de *slot*(s).
- El valor que devuelve esta función es el nombre de la instancia, en caso de éxito, ó el símbolo `FALSE`, en caso contrario.
- El nombre de instancia proporcionado puede ser un nombre de instancia concreto, una dirección de instancia (*instance-address*) ó un símbolo.
- Esta función suspende momentáneamente toda actividad correspondiente al proceso de *pattern-matching*, hasta que haya procesado todas las expresiones de anulación de *slot*(s).
- Si no se especifica ninguna anulación de *slot* o la clase no tiene valor por defecto para el *slot*, éste permanecerá inalterado.
- Si ocurre algún error, la instancia no se elimina, pero los valores de los atributos de la instancia pueden quedar en un estado inconsistente.

Obtención (lectura) de los valores de los atributos (*slots*) de una instancia.

Fuentes externas al objeto (tales como *defrules* o *deffunctions*) pueden leer los valores de los *slots* de un objeto enviando al objeto concreto un mensaje *get-*. Este mensaje tiene la sintaxis:

Sintaxis: `get-<nombre-atributo>`

Los gestores de mensajes asociados al objeto que respondan al mensaje, pueden acceder al atributo para leer su valor directamente o mediante mensajes.

Existe una serie de funciones proporcionadas por COOL que comprueban la existencia de atributos y sus valores.

Modificación (escritura) de los valores de los atributos (*slots*) de una instancia.

Fuentes externas al objeto (tales como *defrules* o *deffunctions*) pueden modificar (escribir) los valores de los *slots* de un objeto enviándole un mensaje *put-*. Este mensaje tiene la sintaxis:

Sintaxis: `put-<nombre-atributo>`

Ej.:
```
>(defclass A (is-a USER)
      (role concrete)
      (slot x (create-accessor read) (default 34))
      (slot y (create-accessor write) (default abc)))
>(make-instance a of A)
[a]
>(send [a] get-x)
34
>(send [a] put-y "Nuevo valor.")
```

```
        "Nuevo valor."
```

Borrado de instancias.

Podemos eliminar una instancia, enviándola el mensaje `delete`.

Ej.: `>(send [a] delete)`
`TRUE`

Al manipular instancias (creación, modificación, eliminación) es posible retrasar la actividad del proceso de *pattern-matching* de las reglas, hasta que hayan finalizado todas las manipulaciones. Esto se consigue mediante la función *object-pattern-match-delay*:

Sintaxis: `(object-pattern-match-delay <acción>*)`

Esta función actúa de forma idéntica a la función *progn$*, pero además tiene el efecto lateral de que cualquier acción que pudiese afectar al proceso de *pattern-matching* de las reglas se retrasa hasta que se salga de la función.

Modificación de instancias.

CLIPS proporciona 4 funciones para modificar instancias. Estas funciones permiten la modificación de los atributos en bloques, sin requerir una serie de mensajes *put-*.
Estas funciones devuelven `TRUE` si se ejecutan con éxito, y `FALSE`, en caso contrario.

Sintaxis:

```
(modify-instance <instancia> <anulación-slot>*)
(active-modify-instance <instancia> <anulación-slot>*)
(message-modify-instance <instancia> <anulación-slot>*)
(active-message-modify-instance <instancia> <anulación-slot>*)
```

Las funciones que empiezan por *active-* permiten que el proceso de *pattern-matching* ocurra a la vez que se van realizando las modificaciones de los *slots*.

Duplicación de instancias.

CLIPS proporciona 4 funciones para duplicar instancias, de modo que la copia y modificaciones de los *slots* se realice en un todo, sin requerir una serie de mensajes *put-*.
Estas funciones devuelven el nombre de la instancia si se ejecutan con éxito, o bien `FALSE`.

Sintaxis:

```
(nombre-función <instancia>
                [to <nombre-instancia>]
                <anulación-slot>*)
```

donde *nombre-función* puede ser:

```
duplicate-instance
active-duplicate-instance
message-duplicate-instance
active-message-duplicate-instance
```

Las funciones que empiezan por *active-* permiten que el proceso de *pattern-matching* ocurra a la vez que se van realizando las modificaciones de los *slots*.

6.7 Manipulación de conjuntos de instancias: consultas. Operaciones.

COOL proporciona un sistema de consulta para determinar y realizar acciones sobre conjuntos de instancias de clases definidas por el usuario.

El sistema de consultas de instancias de COOL proporciona 6 funciones, cada una de las cuales opera sobre conjuntos de instancias determinados por criterios de consulta definidos por el usuario.

A continuación se muestra un cuadro-resumen de estas 6 funciones:

Función	Propósito
`any-instancep`	Determina si uno o más conjuntos de instancias satisfacen una consulta.
`find-instance`	Devuelve el primer conjunto de instancias que satisfaga una consulta.
`find-all-instances`	Agrupa y devuelve todos los conjuntos de instancias que satisfacen una consulta.
`do-for-instance`	Realiza una acción sobre el primer conjunto de instancias que satisfaga una consulta.
`do-for-all-instances`	Realiza una acción con cada conjunto de instancias que satisface una consulta según se van encontrando.
`delayed-do-for-all-instances`	Agrupa todos los conjuntos de instancias que satisfacen una consulta y realiza una acción iterativamente sobre este grupo.

Supongamos las siguientes definiciones, a las que se hace referencia en los ejemplos:

```
(defclass PERSONA (is-a USER)
      (role abstract)
      (slot sexo (access read-only) (storage shared))
      (slot edad (type NUMBER) (visibility public)))

(defmessage-handler PERSONA put-edad (?valor)
      (dynamic-put edad ?valor))

(defclass HEMBRA (is-a PERSONA)
      (role abstract)
      (slot sexo (source composite) (default hembra)))

(defclass VARÓN (is-a PERSONA)
      (role abstract)
      (slot sexo (source composite) (default varón)))

(defclass CHICA (is-a HEMBRA)
      (role concrete)
      (slot edad (source composite) (default 4) (range 0.0 17.0)))
```

```
(defclass MUJER (is-a HEMBRA)
      (role concrete)
      (slot edad (source composite)(default 25)(range 18.0 100.0)))

(defclass CHICO (is-a VARÓN)
      (role concrete)
      (slot edad (source composite) (default 4) (range 0.0 17.0)))

(defclass HOMBRE (is-a VARÓN)
      (role concrete)
      (slot edad (source composite)(default 25)(range 18.0 100.0)))
```

Creación de plantillas de conjuntos de instancias.

- Un conjunto de instancias es una colección ordenada de instancias.
- Una plantilla de conjuntos de instancias es un conjunto de variables (que se instancian con nombres de instancias) y sus restricciones de clase asociadas.
- Las funciones de consulta usan tales plantillas para generar conjuntos de instancias.
- Se puede especificar el módulo de la clase; no es necesario que la clase sea visible dentro del módulo actual.

Sintaxis: `((<variable-monocampo> <nombre-clase>+)+)`

Ej.: pares ordenados de chicos/hombre y chicas/mujeres:

```
((?hombre-o-chico CHICO HOMBRE) (?mujer-o-chica CHICA MUJER))
```

ó, equivalentemente:

```
((?hombre-o-chico VARÓN) (?mujer-o-chica HEMBRA))
```

Ej.: pares (chica,mujer) y (mujer,chica)

```
((?f1 HEMBRA) (?f2 HEMBRA))
```

⇨ Se examina primero las instancias de la clase `CHICA` antes que las de la clase `MUJER`, porque la clase `CHICA` se definió antes.

Creación de consultas.

- Una consulta es una expresión booleana definida por el usuario.
- La expresión booleana se aplica a un conjunto de instancias para determinar si el conjunto satisface más restricciones definidas por el usuario.
- Si la evaluación de la expresión para un conjunto de instancias es cualquier cosa excepto el símbolo `FALSE`, se dice que el conjunto satisface la consulta.

Ej.: pares ordenados de chicos/hombres y chicas/mujeres en los que la edad del primero sea igual a la del segundo:

$$\{(h,m) \in VARÓN \times HEMBRA \mid edad(h) = edad(m)\}$$

```
(= (send ?hombre-o-chico get-edad) (send ?mujer-o-chica get-edad))
```

Otra forma de expresar lo mismo es la siguiente:

```
(= ?hombre-o-chico:edad ?mujer-o-chica:edad)
```

que responde a la sintaxis:

```
<variable>:<nombre-atributo>
```

⇨ Esta sintaxis se usa cuando no es necesario el paso de un mensaje para leer el valor de un atributo, ya que esta forma es más eficiente.

Funciones de consulta.

a) **`any-instancep:`** Determina si existe un conjunto de instancias que satisface una consulta devolviendo el símbolo `TRUE`; en caso contrario, devuelve el símbolo `FALSE`.

Sintaxis:
```
(any-instancep <especificación-cjto-instancia>
     <consulta>)
```

Ej.: ¿Existe algún hombre de más de 30 años?

```
(any-instacep ((?hombre HOMBRE)) (> ?hombre:edad 30))
```

b) **`find-instance:`** devuelve en una estructura multicampo (en la que cada campo es un nombre de instancia) el primer conjunto de instancias que satisface la consulta dada. Si no encuentra ningún conjunto, devuelve el valor multicampo de longitud cero `()`.

Sintaxis:
```
(find-instance <especificación-cjto-instancia>
     <consulta>)
```

Ej.: Encontrar el primer par formado por un hombre y una mujer que tengan la misma edad:

```
> (find-instance ((?h HOMBRE) (?m MUJER)) (= ?h:edad ?m:edad))
([nombre-instancia1] [nombre-instancia2])
```

c) **`find-all-instances:`** devuelve todos los conjuntos de instancias que satisfacen una consulta en una estructura multicampo.

Sintaxis:
```
(find-all-instances  <especificación-cjto-instancia>
          <consulta>)
```

Ej.: Encontrar todos los pares de hombres y mujeres con la misma edad:

```
(find-all-instances ((?h HOMBRE) (?m MUJER)) (= ?h:edad ?m:edad))
```

d) **do-for-instance:** ejecuta una acción sobre el primer conjunto de instancias que satisface una consulta. El valor devuelto es el resultado de la evaluación de la acción. Si no se encuentra ningún conjunto de instancias que satisfaga la consulta, devuelve el símbolo FALSE.

Sintaxis: `(do-for-instance <especificación-cjto-instancia> <consulta> <acción>)`

Ej.: Imprimir en pantalla el primer grupo de tres personas (diferentes) de la misma edad:

```
(do-for-instance ((?p1 PERSONA) (?p2 PERSONA) (?p3 PERSONA))
      (and (= ?p1:edad ?p2:edad ?p3:edad)
           (neq ?p1 ?p2) (neq ?p1 ?p3) (neq ?p2 ?p3))
      (printout t ?p1 " " ?p2 " " ?p3 crlf))
```

e) **do-for-all-instances:** ídem a la función anterior, pero ejecuta la acción para todos los conjuntos de instancias que encuentre que satisfagan la consulta.

Sintaxis: `(do-for-all-instances <cjto-de-instancias> <consulta> <acción>)`

Ej.: Mismo ejemplo anterior, pero para todos los conjuntos que encuentre.

```
(do-for-all-instances ((?p1 PERSONA) (?p2 PERSONA) (?p3 PERSONA))
      (and (= ?p1:edad ?p2:edad ?p3:edad)
           (> (str-compare ?p1 ?p2) 0)
           (> (str-compare ?p2 ?p3) 0))
      (printout t ?p1 " " ?p2 " " ?p3 crlf))
```

Nota: Las llamadas a la función *str-compare* impiden que se consideren ternas que sean simplemente permutaciones, limitando de esta forma el resultado a combinaciones. Así se evita que dos conjuntos de instancias que únicamente difieran en el orden satisfagan la consulta.

Ej.:

```
(do-for-all-instances ((?coche1 RENAULT SEAT) (?coche2 BMW))
      (> ?coche2:precio (* 1.5 ?coche1:precio)) ;;; esta es la pregunta
                                                ;;; que deben satisfacer
      (printout t ?coche1 crlf))  ;;; esta es la acción
```

7. FUNCIONES GENÉRICAS

7.1 Características.

- Las funciones genéricas son similares a las funciones definidas por el usuario (*deffunctions*) ya que también pueden:
  - ⇨ usarse para definir directamente nuevo código procedural, y
  - ⇨ ser invocadas como cualquier otra función.

- Las funciones genéricas son mucho más potentes porque pueden realizar distintas cosas dependiendo del tipo y número de parámetros.

  Ej.: Se podría definir un operador '+' que concatene *strings*, pero que también sume números.

- Las funciones genéricas están formadas por múltiples componentes llamados **métodos** (*methods*), donde cada método gestiona casos diferentes de parámetros para la función genérica. El conjunto de métodos constituye la definición (cuerpo) de la función genérica.

- Una función genérica que posea más de un método se dice que está **sobrecargada** (*overloaded*). Sin embargo, existe una serie de funciones del sistema que no pueden ser sobrecargadas.

- Las funciones definidas por el usuario (*deffunctions*) no pueden actuar como métodos de una función genérica; las *deffunctions* se proporcionan sólo con la finalidad de poder añadir nuevas funcionalidades cuando no interese la sobrecarga.

  Ej.: Una función genérica que tenga un solo método que restringe sólo el número de argumentos es equivalente a una *deffunction*.

- Los métodos de una función genérica no se invocan directamente (sin embargo, existe la función `call-specific-method` para tal efecto).

- CLIPS reconoce cuándo una llamada corresponde a una función genérica, y utiliza los parámetros para encontrar y ejecutar el método apropiado: este proceso de búsqueda y ejecución se denomina ***dispatching* genérico**.

7.2 Construcción *defgeneric*: cabeceras.

Una función genérica consta de una cabecera y cero o más métodos.

Sintaxis: `(`**`defgeneric`** `<nombre> [<comentario>])`

- La cabecera puede ser declarada explícitamente por el usuario, o implícitamente por la definición de, al menos, un método.

- Una función genérica debe ser declarada mediante una cabecera o mediante un método antes de poder llamarla desde:

- un método de otra función genérica,
- una función definida por el usuario (*deffunction*),
- un gestor de mensajes,
- una regla, ó
- desde el intérprete.

(La única excepción es una función genérica recursiva).

Cabeceras:

- ⇨ Una función genérica se identifica unívocamente por un nombre.
- ⇨ La cabecera debe existir explícitamente si se hace referencia a la función en otras construcciones y todavía no se ha definido ninguno de sus métodos.
- ⇨ La definición del primer método crea implícitamente la cabecera.

7.3 Métodos: construcción *defmethod*.

Cada método de una función genérica consta de 6 partes:

1. un nombre (que debe coincidir con el de la función genérica a la que pertenece),
2. un índice (opcional; si no se especifica, CLIPS asigna uno automáticamente),
3. un comentario (opcional),
4. una lista de parámetros obligatorios,
5. una especificación multicampo para gestionar un número variable de parámetros (opcional), y
6. una secuencia de acciones o expresiones que se ejecutarán en el orden en el que aparecen cuando se invoque el método.

Sintaxis:

```
(defmethod <nombre> [<índice>] [<comentario>]
     (<parámetro>* [<resto-parámetros>])
     <acción>*)

<parámetro> ::= <variable-simple> |
                (<variable-simple> <tipo>* [<consulta>])

<resto-parámetros> ::= <variable-multicampo> |
           (<variable-multicampo> <tipo>* [<consulta>])

<tipo> ::= <nombre-de-clase>

<consulta> ::= <variable-global> | <llamada-a-función>
```

- Un método está identificado unívocamente por:
  a) un nombre y un índice (número entero), ó
  b) un nombre y los parámetros.

- A cada método de una función genérica se le asigna un único índice.

- Si se define un nuevo método que tiene exactamente el mismo nombre y la misma especificación de parámetros que otro existente, éste es reemplazado automáticamente por el nuevo.

- Para reemplazar un método por otro con diferente especificación de parámetros, se pone como índice del nuevo método el del antiguo. Sin embargo, el nuevo método no puede tener distinta especificación de parámetros que otro existente con índice diferente).

7.3.1 Tipos de parámetros de un método: simples o múltiples.

- **Parámetros simples:** cada uno se identifica por una variable simple (usando '?')

  En los parámetros se puede especificar: el tipo y una consulta.

  Si para un parámetro no se especifica ningún tipo de restricción (ni tipo ni consulta), el método aceptará cualquier argumento en la posición correspondiente al parámetro formal.

  A) La restricción de tipo determina las clases de argumentos que serán aceptados. Los tipos que se pueden especificar son cualquier clase predefinida por el sistema o definida por el usuario. No se permite redundancia entre las clases especificadas.

  Ej.: La siguiente definición daría un error porque `INTEGER` es una subclase de `NUMBER`.

```
(defmethod f ((?x INTEGER NUMBER)))
```

  B) La restricción de consulta es un *test* booleano que debe satisfacerse para que se acepte un argumento.
  La consulta puede ser:
  - ⇨ un variable global, ó
  - ⇨ una llamada a una función.

  La consulta se satisface si su evaluación es distinta del símbolo `FALSE`.

  NOTA 1: Puesto que una consulta puede no satisfacerse, ésta no debe tener efectos laterales, pues los efectos laterales tendrán lugar al evaluarse tanto si se satisface como si no, incluso si el método resulta ser no aplicable.

  NOTA 2: Precedencia entre métodos.
  Cuando existe más de un método aplicable para unos parámetros determinados, el proceso de *dispatch* genérico establece un orden entre ellos y ejecuta el primero de ese orden.
  Si no existen métodos aplicables, se producirá un error.

  NOTA 3: Los parámetros se examinan de izquierda a derecha, por tanto las consultas que usen parámetros múltiples deben incluirse con el parámetro de más a la derecha.

  Ej.: En el siguiente método, la evaluación de la consulta se retrasa hasta que las clases de ambos parámetros hayan sido verificadas.

```
(defmethod f ((?n INTEGER) (?m INTEGER (> ?n ?m))))
```

- **Parámetros múltiples:** identificados todos ellos por una única variable multicampo (usando '$?').

  - Los parámetros simples (obligatorios, si existen) representan el mínimo número de argumentos que se deben pasar al método. Si existe la especificación de parámetros múltiples, al método se le puede pasar cualquier número de argumentos mayor o igual que el mínimo; en caso contrario, deben pasarse todos los parámetros simples.

  - Los parámetros que no correspondan a ninguno de los simples se agrupan en un único valor multicampo que puede ser referenciado por el parámetro comodín (`$?`) dentro del cuerpo del método.

VARIABLE ESPECIAL: **`?current-argument`** (*parámetro actual*)

Esta variable puede utilizarse solamente en las consultas de parámetros múltiples para referirse a un parámetro individual. Esta variable sólo es visible dentro de la propia consulta y no tiene significado en el cuerpo del método.

Las restricciones de tipo y consulta en los parámetros múltiples se aplican a cada uno de ellos.

Ej.: En este ejemplo, las funciones `>` y *length$* son invocadas 3 veces: una vez para cada argumento (en el ejemplo 3).

```
CLIPS>(defmethod f (($?r (> (length$ ?r) 2))) TRUE)
CLIPS>(f 1 rojo 3.75)
TRUE
CLIPS>(f)
TRUE  ;;; En este caso es aplicable porque, al no pasar parámetros,
      ;;;  la consulta no se evalúa nunca.
```

Ej.: Método que comprueba el número de parámetros. En este caso, el tipo de éstos es irrelevante y se puede colocar la consulta en el parámetro simple para mejorar eficiencia.

```
CLIPS>(defmethod f ((?args (> length$ ?r) 1)) $?r) TRUE)
CLIPS>(f)
ERROR  ;;; se produce error porque no existen métodos aplicables.
```

7.4 Proceso de *dispatch* genérico: aplicabilidad y reglas de precedencia.

Consiste en la selección y ejecución del método de mayor precedencia cuando se llama a una función genérica, para el cual se satisfacen todas las restricciones de sus parámetros. El valor devuelto por el método se devuelve como valor de la función genérica.

Aplicabilidad de métodos.

Un método es aplicable para una llamada a una función genérica si:

1. Su nombre coincide con el de la función genérica, y
2. Se aceptan, al menos, tantos parámetros como le fueron pasados a la función genérica, y
3. Cada argumento de la función genérica satisface la correspondiente restricción (si existe) del método.

Reglas de precedencia entre métodos.

Las siguientes reglas de precedencia se aplican cuando dos o más métodos son aplicables a la llamada de una función genérica particular.
CLIPS elige el método con mayor precedencia. En general, el método con las restricciones de parámetros más específicas tiene la mayor precedencia.

1. Las restricciones de ambos métodos son comparadas posicionalmente de izquierda a derecha dos a dos.
   1.1. Un parámetro simple tiene precedencia sobre un parámetro múltiple.
   1.2. La restricción de tipo más específica de un parámetro particular tiene prioridad: una clase es más específica que cualquiera de sus superclases.
   1.3. Un parámetro con consulta tiene más prioridad que otro que no tiene.

2. El método con mayor número de parámetros simples tiene más precedencia.
3. Un método que no posea parámetros múltiples tiene más precedencia que otro que tenga.
4. Un método definido antes que otro tiene prioridad.

Se puede utilizar el comando *list-defmethods* para examinar la precedencia de los métodos de una función genérica:

Sintaxis: `(list-defmethods [<nombre-función-genérica>])`

7.5 Métodos "oscurecidos": técnicas declarativa e imperativa

Si un método para ser ejecutado debe ser invocado por otro método, se dice que el primer método está **"oscurecido"** (*shadowed*) por el segundo método.

La técnica declarativa se refiere en dejar al proceso de *dispatch* genérico que se encargue de ejecutar el método con la mayor precedencia.

La técnica imperativa consiste en utilizar las funciones *call-next-method* y *override-next-method* que permiten a un método ejecutar el método que él está "oscureciendo". De esta forma, es como si el método que usa esas funciones interviniese en el proceso de *dispatch* genérico. Esto no se recomienda, a no ser que sea absolutamente necesario.

Sintaxis: `(call-next-method)`

Sintaxis: `(override-next-method <expresión>*)`

7.6 Errores de ejecución en métodos

Si ocurre un error durante la ejecución de un método correspondiente a la llamada de una función genérica, cualquier acción del método actual aún no ejecutada se aborta, cualquier método aún no invocado se aborta, y la función genérica devuelve el símbolo `FALSE`.

Recuérdese que la falta de métodos aplicables para un conjunto determinado de parámetros de una función genérica se considera un error de ejecución.

7.7 Valor devuelto por una función genérica

El valor que devuelve una función genérica es el que devuelva el método aplicable con la mayor precedencia. Cada método aplicable que se ejecute puede elegir entre ignorar o capturar el valor devuelto por cualquier método que él esté "oscureciendo".

El valor que devuelve cualquier método es el resultado de la evaluación de la última acción de ese método.

APÉNDICE A: Especificación BNF de CLIPS

Data Types

```
<symbol>          ::= <carácter-ASCII-imprimible-sin-delimitadores>+

<delimitador>     ::= carácter-de-ctrl  espacio tab retorno-de-carro avance-línea
                      " ( ) & | < ; ~

<string>          ::= " <carácter-ASCII-imprimible>* "

<integer>         ::= [+ | -] <dígito>+

<dígito>          ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<float>           ::= <integer> <exponente> |
                      <integer> . [exponente] |
                      . <entero-sin-signo> [exponente] |
                      <integer> . <entero-sin-signo> [exponente] |

     <entero-sin-signo> ::= <dígito>+

     <exponente> ::= e | E <integer>

<instance-name>   ::= A valid instance-name as specified in section 2.3.1

<number>          ::= <float> | <integer>

<lexeme>          ::= <symbol> | <string>

<constant>        ::= <symbol> | <string> | <integer> |
                      <float> | <instance-name>

<comment>         ::= <string>

<variable-symbol> ::= A symbol beginning with an alphabetic character

<function-name>   ::= Any symbol which corresponds to a system or
                      user defined function, a deffunction name, or  a defgeneric name

<file-name>       ::= A symbol or string which is a valid file name
                      (including path information) for the operating
                      system under which CLIPS is running

<slot-name>       ::= A valid deftemplate slot name

<...-name>        ::= A symbol where the ellipsis indicate what the symbol
                      represents. For example, <rule-name> is a symbol which
                      represents the name of a rule
```

Variables and Expressions

```
<single-field-variable> ::= ?<variable-symbol>

<multifield-variable>   ::= $?<variable-symbol>

<global-variable>       ::= ?*<symbol>*

<variable>              ::= <single-field-variable> |
                            <multifield-variable> |
                            <global-variable>

<function-call>         ::= (<function-name> <expression>*)

<expression>            ::= <constant> | <variable> |
                            <function-call>

<action>                ::= <expression>
```

`<...-expression>` ::= *An* `<expression>` *which returns the type indicated by the ellipsis. For example,* `<integer-expression>` *should return an integer.*

Constructs

```
<CLIPS-program> ::= <construct>*

<construct>     ::= <deffacts-construct> |
                    <deftemplate-construct> |
                    <defglobal-construct> |
                    <defrule-construct> |
                    <deffunction-construct> |
                    <defgeneric-construct> |
                    <defmethod-construct> |
                    <defclass-construct> |
                    <definstance-construct> |
                    <defmessage-handler-construct> |
                    <defmodule-construct>
```

Deffacts Construct

```
<deffacts-construct> ::= (deffacts <deffacts-name> [<comment>]
                                   <RHS-pattern>*)
```

Deftemplate Construct

```
<deftemplate-construct> ::= (deftemplate <deftemplate-name>
                                [<comment>] <slot-definition>*)

<slot-definition>  ::= <single-slot-definition> |
                       <multislot-definition>

<single-slot-definition>    ::=    (slot    <slot-name>    <template-
attribute>*)

<multislot-definition>
                  ::= (multislot <slot-name> <template-attribute>*)

<template-attribute>::=<default-attribute> | <constraint-attribute>

<default-attribute>
                  ::= (default ?DERIVE | ?NONE | <expression>*) |
                      (default-dynamic <expression>*)
```

Fact Specification

```
<RHS-pattern> ::= <ordered-RHS-pattern> | <template-RHS-pattern>

<ordered-RHS-pattern>   ::= (<symbol> <RHS-field>+)

<template-RHS-pattern>  ::= (<deftemplate-name> <RHS-slot>*)

<RHS-slot> ::= <single-field-RHS-slot> | <multifield-RHS-slot>

<single-field-RHS-slot> ::= (<slot-name> <RHS-field>)

<multifield-RHS-slot>   ::= (<slot-name> <RHS-field>*)

<RHS-field>       ::= <variable> | <constant> | <function-call>
```

Defrule Construct

```
<defrule-construct>     ::= (defrule <rule-name> [<comment>]
                                [<declaration>]
                                <conditional-element>*
                                =>
                                <action>*)
```

```
<declaration>           ::= (declare <rule-property>+)

<rule-property>         ::= (salience <integer-expression>) |
                            (auto-focus <boolean-symbol>)

<boolean-symbol>        ::= TRUE | FALSE

<conditional-element>   ::= <pattern-CE> | <assigned-pattern-CE> |
                            <not-CE> | <and-CE> | <or-CE> |
                            <logical-CE> | <test-CE> |
                            <exists-CE> | <forall-CE>

<test-CE>    ::= (test <function-call>)

<not-CE>     ::= (not <conditional-element>)

<and-CE>     ::= (and <conditional-element>+)

<or-CE>      ::= (or <conditional-element>+)

<exists-CE>  ::= (exists <conditional-element>+)

<forall-CE>::=(forall <conditional-element> <conditional-element>+)

<logical-CE> ::= (logical <conditional-element>+)

<assigned-pattern-CE>   ::= ?<variable-symbol> <- <pattern-CE>

<pattern-CE>            ::= <ordered-pattern-CE> |
                            <template-pattern-CE> |
                            <object-pattern-CE>

<ordered-pattern-CE>    ::= (<symbol> <constraint>*)

<template-pattern-CE>   ::= (<deftemplate-name> <LHS-slot>*)

<object-pattern-CE>     ::= (object <attribute-constraint>*)

<attribute-constraint>  ::=     (is-a <constraint>) |
                                (name <constraint>) |
                                (<slot-name> <constraint>*)

<LHS-slot>     ::= <single-field-LHS-slot> | <multifield-LHS-slot>

<single-field-LHS-slot> ::= (<slot-name> <constraint>)

<multifield-LHS-slot>   ::= (<slot-name> <constraint>*)
```

```
<constraint>            ::= ? | $? | <connected-constraint>

<connected-constraint>
                ::= <single-constraint> |
                    <single-constraint> & <connected-constraint> |
                    <single-constraint> | <connected-constraint>

<single-constraint>     ::= <term> | ~<term>

<term>                  ::= <constant> |
                            <single-field-variable> |
                            <multifield-variable> |
                            :<function-call> |
                            =<function-call>
```

Defglobal Construct

```
<defglobal-construct>   ::= (defglobal [<defmodule-name>]
                                       <global-assignment>*)

<global-assignment>     ::= <global-variable> = <expression>

<global-variable>       ::= ?*<symbol>*
```

Deffunction Construct

```
<deffunction-construct> ::= (deffunction <name> [<comment>]
                      (<regular-parameter>* [<wildcard-parameter>])
                      <action>*)

<regular-parameter>  ::= <single-field-variable>

<wildcard-parameter> ::= <multifield-variable>
```

Defgeneric Construct

```
<defgeneric-construct> ::= (defgeneric <name> [<comment>])
```

Defmethod Construct

```
<defmethod-construct> ::= (defmethod <name> [<index>] [<comment>]
                                  (<parameter-restriction>*
                                  [<wildcard-parameter-restriction>])
                                  <action>*)
```

```
<parameter-restriction> ::=
                     <single-field-variable> |
                     (<single-field-variable> <type>* [<query>])

<wildcard-parameter-restriction> ::=
                <multifield-variable> |
                (<multifield-variable> <type>* [<query>])

<type> ::= <class-name>

<query> ::= <global-variable> | <function-call>
```

Defclass Construct

```
<defclass-construct> ::= (defclass <name> [<comment>]
                            (is-a <superclass-name>+)
                            [<role>]
                            [<pattern-match-role>]
                            <slot>*
                            <handler-documentation>*)

<role>  ::= (role concrete | abstract)

<pattern-match-role> ::= (pattern-match reactive | non-reactive)

<slot>  ::= (slot <name> <facet>*) |
            (single-slot <name> <facet>*) |
            (multislot <name> <facet>*)

<facet> ::=  <default-facet> | <storage-facet> |
             <access-facet> | <propagation-facet> |
             <source-facet> | <pattern-match-facet> |
             <visibility-facet> | <create-accessor-facet>
             <override-message-facet> | <constraint-attributes>

<default-facet> ::= (default ?DERIVE | ?NONE | <expression>*) |
                    (default-dynamic <expression>*)

<storage-facet> ::= (storage local | shared)

<access-facet>::= (access read-write | read-only | initialize-only)

<propagation-facet> ::= (propagation inherit | no-inherit)

<source-facet> ::= (source exclusive | composite)

<pattern-match-facet>  ::= (pattern-match reactive | non-reactive)
```

```
<visibility-facet> ::= (visibility private | public)

<create-accessor-facet>
     ::= (create-accessor ?NONE | read | write | read-write)

<override-message-facet>
     ::= (override-message ?DEFAULT | <message-name>)

<handler-documentation> ::=
                (message-handler <name> [<handler-type>])

<handler-type> ::= primary | around | before | after
```

Defmessage-handler Construct

```
<defmessage-handler-construct>  ::=
     (defmessage-handler <class-name> <message-name>
         [<handler-type>] [<comment>]
         (<parameter>* [<wildcard-parameter>])
         <action>*)

<handler-type> ::= around | before | primary | after

<parameter> ::= <single-field-variable>

<wildcard-parameter> ::= <multifield-variable>
```

Definstances Construct

```
<definstances-construct> ::=
           (definstances <definstances-name>
                [active] [<comment>] <instance-template>*)

<instance-template>   ::= (<instance-definition>)

<instance-definition> ::=
           <instance-name-expression> of <class-name-expression>
           <slot-override>*

<slot-override> ::= (<slot-name-expression> <expression>*)
```

Defmodule Construct

```
<defmodule-construct> ::= (defmodule <module-name> [<comment>]
                             <port-spec>*)

<port-specification>  ::= (export <port-item>) |
                          (import <module-name> <port-item>)
```

```
<port-item>           ::= ?ALL |
                          ?NONE |
                          <port-construct> ?ALL |
                          <port-construct> ?NONE |
                          <port-construct> <construct-name>+

<port-construct>      ::= deftemplate | defclass |
                          defglobal | deffunction |
                          defgeneric
```

Constraint Attributes

```
<constraint-attribute>::= <type-attribute> |
                          <allowed-constant-attribute> |
                          <range-attribute> |
                          <cardinality-attribute>

<type-attribute>      ::= (type <type-specification>)

<type-specification>  ::= <allowed-type>+ | ?VARIABLE

<allowed-type>        ::= SYMBOL | STRING | LEXEME |
                          INTEGER | FLOAT | NUMBER |
                          INSTANCE-NAME | INSTANCE-ADDRESS |
                          INSTANCE | EXTERNAL-ADDRESS |
                          FACT-ADDRESS


<allowed-constant-attribute>
                    ::= (allowed-symbols <symbol-list>) |
                        (allowed-strings <string-list>) |
                        (allowed-lexemes <lexeme-list> |
                        (allowed-integers <integer-list>) |
                        (allowed-floats <float-list>) |
                        (allowed-numbers <number-list>) |
                        (allowed-instance-names <instance-list>) |
                        (allowed-values <value-list>)

<symbol-list>         ::= <symbol>+ | ?VARIABLE

<string-list>         ::= <string>+ | ?VARIABLE

<lexeme-list>         ::= <lexeme>+ | ?VARIABLE

<integer-list>        ::= <integer>+ | ?VARIABLE

<float-list>          ::= <float>+ | ?VARIABLE

<number-list>         ::= <number>+ | ?VARIABLE
```

```
<instance-name-list>  ::= <instance-name>+ | ?VARIABLE

<value-list>          ::= <constant>+ | ?VARIABLE

<range-attribute>     ::= (range <range-specification>
                                 <range-specification>)

<range-specification> ::= <number> | ?VARIABLE

<cardinality-attribute>
            ::= (cardinality <cardinality-specification>
                             <cardinality-specification>)

<cardinality-specification> ::= <integer> | ?VARIABLE
```

APÉNDICE B: Encadenamiento hacia atrás en CLIPS

CLIPS no implementa directamente el encadenamiento hacia atrás como parte de su motor de inferencia. Sin embargo, el encadenamiento hacia atrás se puede emular mediante reglas que usen el encadenamiento hacia adelante de CLIPS.

(Nota: CLIPS está diseñado para ser utilizado como un lenguaje de encadenamiento hacia adelante. Por tanto, si para resolver un cierto problema el encadenamiento hacia atrás es más apropiado, entonces se debería usar un lenguaje que implemente directamente tal encadenamiento).

En este apéndice se construye un sistema simple de encadenamiento hacia atrás. Este sistema tiene las siguientes capacidades y limitaciones:

- Los hechos se representan como pares atributo-valor.
- El encadenamiento hacia atrás empezará con la aserción de un atributo objetivo inicial.
- Sólo se comprobará la igualdad de un atributo a un valor específico como condición en el antecedente de una regla.
- La única acción del antecedente de una regla será la asignación de un valor de un atributo simple.
- Si el valor de un atributo objetivo no puede ser determinado usando reglas, el sistema pedirá un valor para el atributo. No se puede asignar a un atributo un valor desconocido.
- Un atributo sólo puede contener un valor simple. El sistema no soporta razonamiento hipotético sobre valores de atributos diferentes provenientes de reglas diferentes.
- No se representa conocimiento incierto.

Algoritmo que seguirá el sistema

```
función Resolver_Objetivo (in obj) devuelve
     -- Entrada: atributo objetivo obj
     -- Salida: determina el valor para el atributo objetivo

     si el valor del atributo objetivo es conocido entonces
          devolver el valor del atributo objetivo
     fin si;

     para cada regla R cuyo consecuente sea el atributo objetivo
     hacer
          éxito := Intentar_Regla(R);
          si éxito entonces
               Asignar al atributo objetivo el valor indicado
               por el consecuente de la regla R;
               devolver el valor del atributo objetivo;
          fin si;
     fin para;
     Preguntar al usuario el valor del atributo objetivo;
     Asignar al atributo objetivo el valor proporcionado;
     devolver el valor del atributo objetivo;
fin función;
```

```
función Intentar_Regla (R : in Regla) devolver booleano
     -- Entrada: una regla R
     -- Salida: TRUE si se satisface el antecedente de R;
     --         FALSE, en caso contrario.

     para cada condición B en el antecedente de R hacer
          val := Resolver_Objetivo(B)
          si val ≠ valor requerido por B entonces
               devolver FALSE;
          fin si;
     fin para;
     devolver TRUE;
fin función;
```

Representación de las reglas de encadenamiento hacia atrás en CLIPS

Una regla de encadenamiento hacia atrás la representaremos como un hecho. Así, los antecedentes y consecuentes pueden ser examinados por las reglas CLIPS (*defrule*) las cuales actuarán como el motor de inferencia del encadenamiento hacia atrás.

```
(deftemplate regla
     (multislot if)     ; antecedente
     (multislot then))  ; consecuente
```

- Cada antecedente contendrá un par simple atributo-valor de la forma:

```
<antecedente> ::= <atributo> is <valor> (and <atributo> is <valor>)*
```

- El consecuente sólo podrá contener un par atributo-valor.

Según vaya procediendo el encadenamiento hacia atrás, se irán generando subobjetivos para determinar el valor de atributos. Utilizaremos otro *deftemplate* para representar un subobjetivo:

```
(deftemplate objetivo
     (slot atributo))
```

Cuando se determinen los valores de los atributos, habrá que almacenarlos. Para ello utilizaremos el siguiente *deftemplate*:

```
(deftemplate atributo
     (slot nombre)
     (slot valor))
```

A continuación se presenta el módulo CLIPS que hace el papel de motor de inferencia de encadenamiento hacia atrás:

```
;;; Módulo de Encadenamiento Hacia Atrás
(defmodule EHA (export deftemplate regla objetivo atributo))

(defrule EHA::INTENTAR-REGLA
     (objetivo (atributo ?nom-obj))
     (regla (if ?anteced $?)
```

```
               (then ?consec $?))
      (not (atributo (nombre ?anteced)))
      (not (objetivo (atributo ?anteced)))
      =>
      (assert (objetivo (atributo ?anteced))))

(defrule EHA::PEDIR-ATRIBUTO-VALOR
      ?obj <- (objetivo (atributo ?nom-obj))
      (not (atributo (nombre ?nom-obj)))
      (not (regla (then ?nom-obj $?)))
      =>
      (retract ?obj)
      (printout t "Cuál es el valor de " ?nom-obj "? ")
      (assert (atributo (nombre ?nom-obj) (valor (read)))))
```

APÉNDICE C: Construcción de un programa ejecutable a partir del código fuente CLIPS

La idea general es que el código fuente en CLIPS se traduce a *C*, y luego este último se compila y se *linka*.

Para crear un programa ejecutable a partir del código fuente CLIPS hay que seguir los siguientes pasos:

① Iniciar CLIPS y cargar todas las construcciones (es decir, todo el código fuente) que constituirán el módulo ejecutable.

② Utilizar la función *constructs-to-c* para traducir el código CLIPS a *C*:

Sintaxis: `(constructs-to-c <nombre-fichero> <id> [<max>])`

donde:

`<nombre-fichero>` es un *string* o un símbolo
`<id>` es un entero
`<max>` es un entero que limita el tamaño de los ficheros fuente .c e indica el nº máximo de estructuras que pueden ser colocadas en un *array* en los ficheros fuente. Útil cuando algún compilador imponga un límite determinado al tamaño de los ficheros fuente que pueda compilar.

③ Establecer a `1` la variable `RUN_TIME` que aparece en el fichero cabecera *setup.h* y compilar todos los ficheros fuente (.c) que la función *constructs-to-c* haya creado.

④ Recompilar todo el código fuente de CLIPS (la variable `RUN_TIME` debe estar aún a `1`). Se recomienda recompilar en otra librería separada de la versión completa de CLIPS.

⑤ Modificar el fichero `main.c`. Si no se utilizan los argumentos *argv* ni *argc*, éstos deberían ser eliminados. A continuación se lista un ejemplo de programa principal:

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

main()
{
   InitializeCLIPS();
   InitCImage_1();
        •
        •   /* Cualquier inicialización del usuario */
        •
   Reset();
   Run(-1L);
        •
        •   /* Otro código */
        •
}
UserFunctions()
{
  /* Esta función no se le llama
     en una versión ejecutable. */
```

```
}
```

APÉNDICE D: Interfaz con C, Ada y Visual Basic

CLIPS fue diseñado para ser embebido (*embedded*) dentro de otros programas. Cuando CLIPS es embebido, el usuario debe proporcionar un programa principal. Las llamadas a CLIPS se realizan como cualquier otra subrutina.

Interfaz con C

Para embeber CLIPS, añadir las siguientes líneas al programa principal:

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

El programa principal del usuario debe inicializar CLIPS llamando a la función `InitializeCLIPS` antes de cargar las construcciones (*deftemplate*s, *defrule*s, *defclass*es, ...).

Interfaz con Ada

Los siguientes listados son la especificación y cuerpo de un paquete Ada para algunas de las funciones CLIPS utilizadas en sistemas CLIPS embebidos. También se muestra el listado de un programa principal Ada de ejemplo de uso del paquete.

```
package CLIPS is

  procedure xInitializeCLIPS;
    -- Initializes the CLIPS environment upon program startup


  procedure xReset;
    -- Resets the CLIPS environment.


  function xLoad (File_Name : in STRING) return INTEGER;
    -- Loads a set of constructs into the CLIPS database.
    -- If there are syntactic error in the constructs,
    -- xLoadConstructs will still attempt to read the entire
    -- file, and error notices will be sent to werror.
    -- Returns: an integer, zero if an error occurs.


  function xRun (Run_Limit : in INTEGER := -1) return INTEGER;
    -- Allows Run_Limit rules to fire (execute).
    -- -1 allows rules to fire until the agenda is empty.
    -- Returns: Number of rules that were fired.


  procedure xFacts (Logical_Name : in STRING;
                    Module_Ptr   : in INTEGER;
                    First        : in INTEGER;
                    Last         : in INTEGER;
```

```
                    Max          : in INTEGER);
    -- Lists the facts in the fact-list.


  function xWatch (Watch_Item : in STRING) return INTEGER;
    -- Turns the watch facilities of CLIPS on.


  function xUnwatch (Watch_Item : in STRING) return INTEGER;
    -- Turns the watch facilities of CLIPS off.


  function xAssertString (Pattern : in STRING) return INTEGER;
    -- Asserts a fact into the CLIPS fact-list.
    -- The function version
    -- returns the Fact_Pointer required by xRetractFact.


  function xRetract (Fact_Pointer : in INTEGER) return INTEGER;
    -- Causes a fact asserted by the ASSERT_FACT function to be
    -- retracted.
    -- Returns: false if fact has already been retracted, else
    -- true.
    -- Input of any value not returned by ASSERT_FACT will
    -- cause CLIPS to abort.

  function xPrintCLIPS (Log_Name : in STRING;
                        Str      : in STRING) return INTEGER;
    -- Queries all active routers until it finds a router that
    -- recognizes the logical name associated with this I/O request
    -- to print a string. It then calls the print function
    -- associated with that router.


  function xUndefrule (Rule_Name : in STRING) return INTEGER;
    -- Removes a rule from CLIPS.
    -- Returns: false if rule not found, else true.

private

  -- El código de esta parte private es específico del compilador
  -- Ada de DEC. Otros compiladores de Ada proporcionarán
  -- capacidades similares variando únicamente algo la sintaxis.

  pragma INTERFACE(C, xInitializeCLIPS);
  pragma IMPORT_PROCEDURE (INTERNAL => xInitializeCLIPS,
                           EXTERNAL => InitializeCLIPS);

  pragma INTERFACE(C, xReset);
  pragma IMPORT_PROCEDURE (INTERNAL => xReset,
                           EXTERNAL => Reset);

  function cLoad (File_Name : in STRING) return INTEGER;
    pragma INTERFACE(C, cLoad);
    pragma IMPORT_FUNCTION (INTERNAL  => cLoad,
                            EXTERNAL  => Load,
                            MECHANISM => REFERENCE);
```

```
  pragma INTERFACE(C, xRun);
  pragma IMPORT_FUNCTION (INTERNAL  => xRun,
                          EXTERNAL  => Run,
                          MECHANISM => VALUE);

  procedure cFacts(Logical_Name : in STRING;
                   Module_Ptr   : in INTEGER;
                   First        : in INTEGER;
                   Last         : in INTEGER;
                   Max          : in INTEGER);
    pragma INTERFACE (C, cFacts);
    pragma IMPORT_PROCEDURE (INTERNAL => cFacts,
                             EXTERNAL => Facts,
                             MECHANISM => (REFERENCE, VALUE,
                                           VALUE, VALUE, VALUE));

  function cWatch (Item : in STRING) return INTEGER;
    pragma INTERFACE (C, cWatch);
    pragma IMPORT_FUNCTION (INTERNAL => cWatch,
                            EXTERNAL => Watch,
                            MECHANISM => REFERENCE);

  function cUnwatch (Item : in STRING) return INTEGER;
    pragma INTERFACE (C, cUnwatch);
    pragma IMPORT_FUNCTION (INTERNAL => cUnwatch,
                            EXTERNAL => Unwatch,
                            MECHANISM => REFERENCE);


  function cAssertString (Pattern : in STRING) return INTEGER;
    pragma INTERFACE (C, cAssertString);
    pragma IMPORT_FUNCTION (INTERNAL => cAssertString,
                            EXTERNAL => AssertString,
                            MECHANISM => REFERENCE);

  function cRetract (Fact_Pointer : in INTEGER) return INTEGER;
    pragma INTERFACE (C, cRetract);
    pragma IMPORT_FUNCTION (INTERNAL => cRetract,
                            EXTERNAL => Retract,
                            MECHANISM => VALUE);

  function cPrintCLIPS (Log_Name : in STRING ;
                        Str : in STRING) return INTEGER;
    pragma INTERFACE (C, cPrintCLIPS);
    pragma IMPORT_FUNCTION (INTERNAL => cPrintCLIPS,
                            EXTERNAL => PrintCLIPS,
                            MECHANISM => REFERENCE);

  function cUndefrule (Rule_Name : in STRING) return INTEGER;
    pragma INTERFACE (C, cUndefrule);
    pragma IMPORT_FUNCTION (INTERNAL  => cUndefrule,
                            EXTERNAL  => Undefrule,
                            MECHANISM => REFERENCE);

end CLIPS;
```

```
package body CLIPS is


  function ADA_TO_C_STRING (Input_String : in STRING)
           return STRING is
    Out_String : STRING (1..Input_String'LAST+1);
  begin
    for I in Input_String'RANGE loop
      if (Input_String(I) in ' ' ..  '~' or
          Input_String(I) = ASCII.CR or
          Input_String(I) = ASCII.LF) then
        Out_String(I) := Input_String(I);
      else
        Out_String(I) := ASCII.NUL;
      end if;
    end loop;
    Out_String(Out_String'LAST) := ASCII.NUL;
    return Out_String;
  end ADA_TO_C_STRING;


  function xLoad (File_Name : in STRING) return INTEGER is
  begin
    return cLoad(ADA_TO_C_STRING(File_Name));
  end xLoad;


  procedure xFacts (Logical_Name : in STRING;
                    Module_Ptr : in INTEGER;
                    First : in INTEGER;
                    Last : in INTEGER;
                    Max : in INTEGER) is
  begin
    cFacts(ADA_TO_C_STRING (Logical_Name),
           Module_Ptr, First, Last, Max);
  end xFacts;


  function xWatch (Watch_Item : in STRING) return INTEGER is
  begin
    return cWatch(ADA_TO_C_STRING (Watch_Item));
  end xWatch;


  function xUnwatch (Watch_Item : in STRING) return INTEGER is
  begin
    return cUnwatch(ADA_TO_C_STRING (Watch_Item));
  end xUnwatch;

  function xAssertString (Pattern : in STRING) return INTEGER is
  begin
    return cAssertString(ADA_TO_C_STRING (Pattern));
  end xAssertString;


  function xRetract (Fact_Pointer : in INTEGER) return INTEGER is
```

```
  begin
    return cRetract(Fact_Pointer);
  end xRetract;


  function xPrintCLIPS (Log_Name : in STRING;
                        Str      : in STRING) return INTEGER is
  begin
    return cPrintCLIPS (ADA_TO_C_STRING (Log_Name),
                        ADA_TO_C_STRING (Str));
  end xPrintCLIPS;


  function xUndefrule (Rule_Name : in STRING) return INTEGER is
  begin
    return cUndefrule(ADA_TO_C_STRING(Rule_Name));
  end xUndefrule;

end CLIPS;
```

```
with CLIPS;     use CLIPS;
with TEXT_IO;   use TEXT_IO;

procedure MAIN is
  File_Name        : STRING(1..50);
  File_Open_Status : INTEGER;
  Rules_Fired      : INTEGER;
begin
  xInitializeCLIPS;

  File_Name(1..7) := "mab.clp";
    -- Load rules
  File_Open_Status := xLoad(File_Name);

  if File_Open_Status = 1 then
    xReset;
    Rules_Fired := xRun(-1);
    PUT(INTEGER'IMAGE(Rules_Fired));
    PUT_LINE("  Rules Fired");
  else
    PUT_LINE ("Unable to open rules file!");
  end if;
end MAIN;
```

Interfaz con Visual Basic

Existen unas bibliotecas de funciones para entorno Windows 3.x/95/NT que hacen de interfaz entre Visual Basic y código fuente CLIPS. Estas bibliotecas están disponibles en la dirección de Internet que se comenta en el apéndice E.

APÉNDICE E: Sitios web interesantes sobre la IA y CLIPS

`[cossack.]cosmic.uga.edu`

News Group sobre CLIPS
`comp.ai.shells` (Un *News Group*)
`ftp.ensmp.fr` (mirar en /pub/clips)
`http://ourworld.compuserve.com/homepages/marktoml/clipstuf.html`
`jsc.nasa.gov`
`www.isphouston.com`
`www.kuai.se/~gillis/vb`
`www.qns.com/~robinson/vb/vb.html`