DECSAI
Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

Sistemas Inteligentes de Gestión

Tutorial de CLIPS

© *Juan Carlos Cubero & Fernando Berzal*

Índice

CLIPS

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

Desarrollado originalmente en la NASA a mediados de los 80, puede integrarse con C/C++ (en ambas direcciones). De hecho, su nombre es un acrónimo derivado de "C Language Integrated Production System"

La sintaxis (y el nombre) de CLIPS están inspirados en OPS ("Official Production System"), un shell para el desarrollo de sistemas expertos creado en 1977 por Charles Forgy durante su doctorado con Allen Newell en la Carnegie-Mellon University.

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

> NOTA: Posteriormente, OPS se reprogramó, por cuestiones de eficiencia, en BLISS ["Basic Language for Implementation of System Software" o "System Software Implementation Language, Backwards"], un lenguaje de programación de sistemas diseñado en CMU que pasó a un segundo plano debido al auge del lenguaje de programación C.
>
> http://en.wikipedia.org/wiki/BLISS_programming_language

Referencias

- Sitio web oficial de CLIPS:

  http://clipsrules.sourceforge.net/

- Jess [Java Expert System Shell], descendiente de CLIPS para Java:

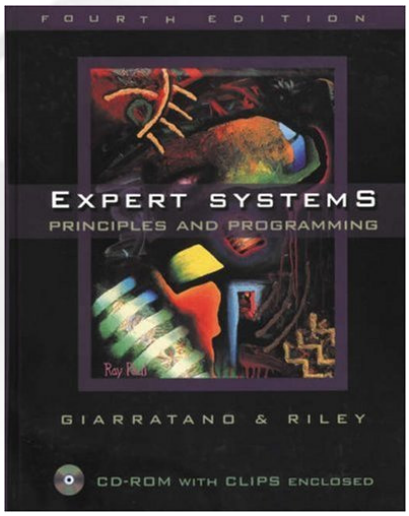
  http://www.jessrules.com/

- Libro de texto:

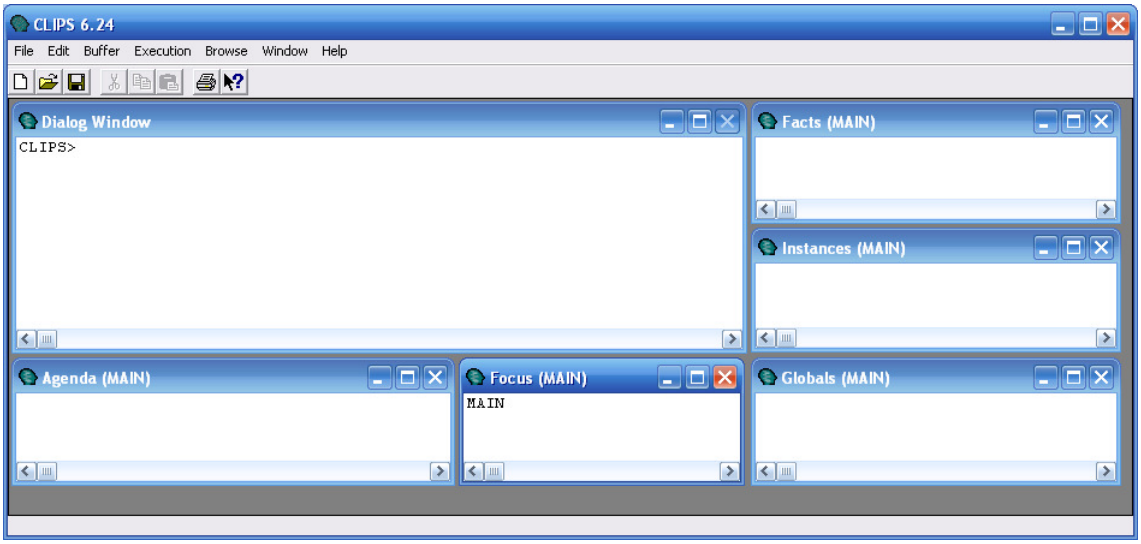
  Joseph C. Giarratano & Gary D. Riley:
  Expert Systems:
  Principles and Programming
  Thomson, 4th edition, 2005.
  ISBN 0534384471

- Más información:

  http://es.wikipedia.org/wiki/CLIPS



Hechos y reglas



El entorno de trabajo de CLIPS nos muestra:

- La Memoria de trabajo (ventana *Facts*).
- El "conjunto conflicto" (ventana *Agenda*).

Además, nos permite inspeccionar otros elementos (por ejemplo, instancias de objetos).

A la hora de representar datos, CLIPS nos permite utilizar:

1. Vectores ordenados de características.
2. Registros.
3. Instancias de clases (objetos).

A continuación veremos cómo se usan los dos primeros:

Vectores ordenados de características

`(Pedro 45 V NO)` sería un ejemplo de vector ordenado.

Cada valor corresponde, según un orden definido por el programador, a un atributo.

Desde la línea de comandos de CLIPS, puede "asertarse" un hecho mediante `(assert)`:

```
CLIPS> (assert (Pedro 45 V NO))
```

Obviamente, este hecho será distinto al hecho `(Pedro 45 NO V)`, dado que el orden es fundamental en la definición de hechos de esta forma.

Una buena práctica consiste en anteponer un nombre de relación al vector ordenado de características, como por ejemplo: `(Persona Pedro 45 NO V)`

Registros

Para evitar errores asociados a la utilización de vectores ordenados de características, normalmente resulta más recomendable definir registros (*templates*).

Primero, definiremos el tipo de dato registro con `(deftemplate)`. A continuación, utilizaremos `(assert)` para establecer uno o varios hechos del tipo definido.

```
CLIPS> (deftemplate Persona
                    (field Nombre)
                    (field Edad)
                    (field Sexo)
                    (field EstadoCivil) )

CLIPS> (assert (Persona
                  (Nombre Juan)
                  (Edad 30)
                  (EstadoCivil casado)
                  (Sexo V)))
```

- Cada vez que se añade un hecho (vector o registro), CLIPS devuelve un identificador de hecho, empezando desde 0 (a no ser que se haya asertado automáticamente con `(reset)` el hecho inicial, en cuyo caso, éste es el que tiene índice 0).

- CLIPS es sensible a mayúsculas y minúsculas, por lo que `persona` sería distinto a `Persona`.

- Si no se especifica un campo de un registro, CLIPS le asigna automáticamente un valor nulo (`nil` en CLIPS).

- No pueden definirse estructuras anidadas, con un template dentro de otro. Para hacer esto tendríamos que utilizar clases.

NOTA: Aunque actualmente se haga referencia a los campos de los registros con el término `field`, antes de la versión 6 de CLIPS, se utilizaba el término `slot` (tradicionalmente usado en I.A. para modelos de representación del conocimiento basados en marcos).

multifield

Dentro de un template, si queremos que un campo sea un vector ordenado de características, usamos el identificador `multifield`, en vez de `field`:

```
(deftemplate Persona
      (field Nombre)
      (field Edad)
)

(assert (Persona
      (Nombre Juan Carlos Cubero)
      (Edad 39))
)
```

Error: "The single field slot Nombre can only contain a single field"

```
(deftemplate Persona
      (multifield Nombre)
      (field Edad)
)

(assert (Persona
      (Nombre Juan Carlos Cubero)
      (Edad 33))
)
```

Correcto!!

Tanto con `field` como con `multifield`, podríamos haber utilizado comillas dobles:

```
(assert (Persona
      (Nombre "Juan Carlos Cubero")
      (Edad 33))
)
```

Uso de la memoria de trabajo

(deffacts)

Para no tener que estar asertando hechos desde línea de comandos de CLIPS, podemos utilizar el comando `(deffacts)` para asertarlos desde un fichero (con extensión `.clp` por convención).

personas.clp

```
; Datos de personas

(deftemplate Persona
      (field Nombre)
      (field Edad)
      (field Sexo)
      (field EstadoCivil)
)

(deffacts VariosHechos
      (Persona
            (Nombre JuanCarlos)
            (Edad 33))
      (Persona
            (Nombre Maria)
            (Sexo M))
)

(deffacts OtrosHechos
      (NumeroDeReactores 4)
)
```

El fichero comienza con un comentario (que en CLIPS se indica mediante un punto y coma al comienzo de la línea), tras el cual se define el tipo de registro Persona y se establecen tres hechos (dos de tipo Persona y un vector ordenado).

Para cargar el fichero escribimos lo siguiente desde la línea de comandos de CLIPS:

```
CLIPS> (load personas.clp)
```

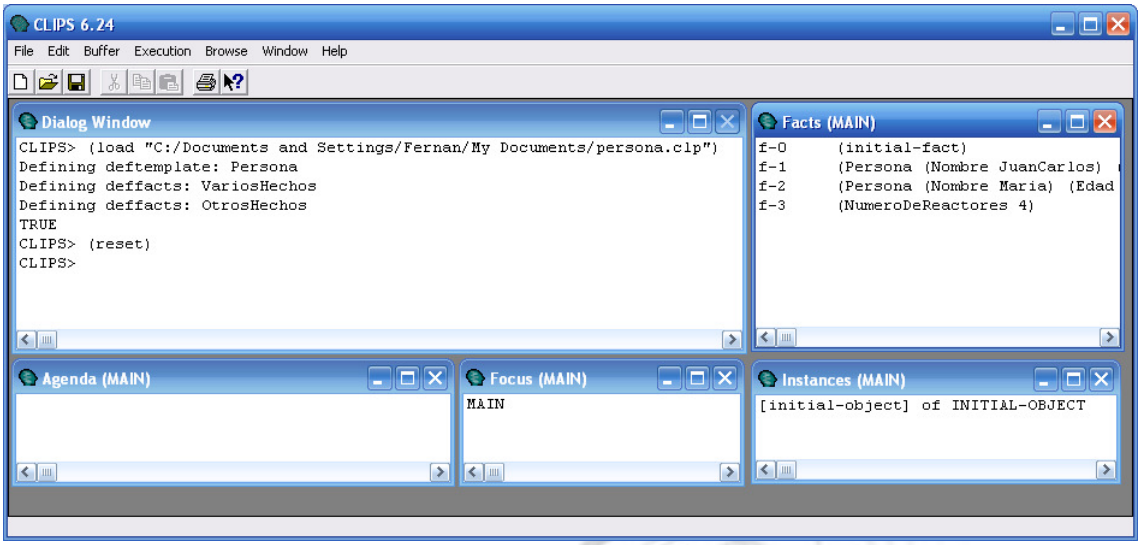
Pero los tres hechos no pasan a formar parte de nuestra memoria de trabajo hasta que ejecutamos:

```
CLIPS> (reset)
```

`(reset)` elimina primero todos los hechos que estuviesen en la memoria de trabajo. Las definiciones de los templates no se borran con `(reset)`, pero sí los hechos de la memoria de trabajo, ya sean vectores o registros.

`(reset)` también añade un hecho inicial con identificador 0 (del que más adelante veremos su uso).

A continuación, se cargan los `(deffacts)` existentes en los ficheros que hayamos cargado con `(load)`. Si, por ejemplo, hubiésemos escrito `(load f1.clp)` y `(load f2.clp)`, entonces, después de ejecutar `(reset)`, tendríamos en la memoria de trabajo tanto los hechos definidos con `(deffacts)` en `f1.clp` como los definidos en `f2.clp`.



Resumiendo, CLIPS nos proporciona los siguientes mecanismos para añadir hechos a la memoria de trabajo:

- Desde línea de comandos con `(assert)`.
- Desde un fichero que contenga `(deffacts)`, que deberemos cargar con (load) y posteriormente ejecutar un `(reset)`.
- Como acción ejecutada en una regla.

(retract)

```
(retract <índice de hecho>+ )
```

```
CLIPS> (retract 1 2)
```

Suprime los hechos con identificadores 1 y 2.

```
CLIPS> (retract *)
```

borra todos los hechos

NOTA: Resulta bastante incómodo tener que recordar los identificadores de los hechos, por lo que veremos una forma mejor de hacerlo más adelante.

(modify) & (duplícate)

```
(modify <índice de hecho>
      (<nombre campo> <nuevo valor>))
```

```
CLIPS> (modify 1 (edad 27))
```

cambia la edad asociada al hecho #1, que deberá tener definido el campo edad.

IMPORTANTE: El identificador del hecho cambia después del `(modify)`.

Para añadir una copia de un hecho a la vez que modificamos alguno de sus campos:

```
CLIPS> (duplicate 1
            (Nombre "Pedro Pérez")
            (Edad 40))
```

(clear)

```
CLIPS> (clear)
```

borra todo lo que se hubiese definido en la sesión de trabajo, ya sea desde la línea de comandos o desde un fichero (todos los hechos que existan en la memoria de trabajo las definiciones de registros, los deffacts, las reglas…).

También es posible borrar selectivamente lo que nos interese, con `(undeffacts ...)`, `(undefrules ...)`, etc.

Reglas

```
(defrule <nombre> <comentario>
      <patrón antecedente> +
      =>
      <consecuente> +
)
```

La parte izquierda de la regla suele llamarse LHS (Left Hand Side), mientras que la parte derecha es RHS (Right Hand Side).

La LHS está constituida por varios patrones. En ellos se establecen las condiciones que han de darse sobre los elementos de la memoria de trabajo para que la regla pueda activarse.

Los consecuentes pueden consistir en la adición o supresión de un elemento a la memoria de trabajo (ejecutando `(assert)` o `(retract)`) o la ejecución de algún procedimiento (como, por ejemplo, `(printout)`).

reglas.clp

```
(deffacts VariosHechosVectores
      (Persona Pedro 45 V SI)
      (Persona Maria 34 M NO)
)
(defrule ECivilPedro_Soltero
      (Persona Pedro 45 V NO)
      =>
      (printout t crlf "Pedro está soltero")
)
(defrule ECivilPedro_Casado
      (Persona Pedro 45 V SI)
      =>
      (printout t crlf "Pedro está casado")
)
(defrule ECivilMaria_Soltera
      (Persona Maria 34 M NO)
      =>
      (printout t crlf "Maria está soltera")
)
```

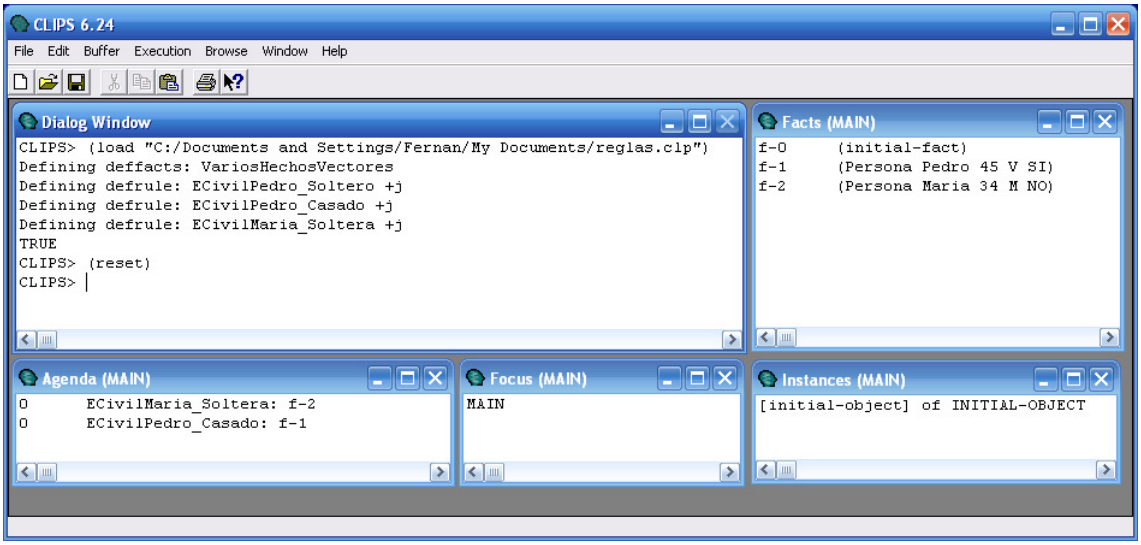
```
CLIPS> (clear)
CLIPS> (load datos.clp)
CLIPS> (reset)
```

RECORDATORIO: `(reset)` es necesario para añadir los hechos definidos con `(deffacts)` a la memoria de trabajo.

El ciclo de control de CLIPS

Tres fases constituyen el ciclo de control de un motor de inferencia con encadenamiento hacia adelante: emparejamiento, resolución de conflictos y ejecución.

Veamos cómo se realiza este proceso en CLIPS:



Después de `(reset)`, CLIPS realiza el emparejamiento de todas las reglas con todos los hechos de la memoria de trabajo y, en ese momento, ya son aplicables dos reglas que pasan a formar parte de la agenda. En realidad, entran en la agenda las activaciones de las reglas con los hechos correspondientes (es decir, la misma regla podría aparecer varias veces en la agenda si fuese aplicable a distintos datos):

Memoria de trabajo	Agenda
f-1:(P Pedro 45 V SI) f-2:(P Maria 34 M NO)	ECMS(f-2) ECPC(f-1)

A continuación, CLIPS ordena dichas reglas según algún criterio y aquí finalizan las acciones realizadas por `(reset)`.

La primera regla de la agenda no se dispara hasta que no lo digamos explícitamente con:

```
CLIPS> (run 1)
```
o bien `CONTROL+T`

lo que mostrará en pantalla el mensaje "María está soltera" y eliminará de la agenda la regla que se ha aplicado:

Memoria de trabajo	Agenda
f-1:(P Pedro 45 V SI) f-2:(P Maria 34 M NO)	ECPC(f-1)

Cada vez que ejecutamos `(run 1)`, se realiza una iteración completa del ciclo de control de CLIPS, que consta de las siguientes fases:

1. **Ejecución y refracción:** Se ejecuta la primera regla de la agenda y ésta desaparece de la misma por refracción.

2. **Emparejamiento:** Se emparejan los hechos de la memoria de trabajo con las reglas existentes.

    a. Si en la parte derecha de la regla ejecutada se hubiese añadido algún elemento a la memoria de trabajo, algunas reglas nuevas podrían ser aplicables (por lo que entrarían en la agenda).

    b. Si en la parte derecha de la regla ejecutada se hubiese suprimido algún elemento de la memoria de trabajo, algunas reglas que estaban en la agenda podrían desaparecer de ésta.

    NOTA: En el ejemplo anterior, no se daba ninguna de los dos casos, por lo que no se modificaba la agenda (más allá de la regla aplicada, que desaparecía).

3. **Resolución de conflictos:** Según el criterio definido por CLIPS, se ordenan las reglas (activaciones) de la agenda y se selecciona una de ellas como la primera, que será la que se ejecute en el siguiente ciclo de control.

En nuestro ejemplo, como sólo queda una regla aplicable, la resolución es inmediata:

Memoria de trabajo	Agenda
f-1:(P Pedro 45 V SI) f-2:(P Maria 34 M NO)	ECPC(f-1)

```
CLIPS> (run 1)
```

mostrará ahora en pantalla el mensaje "Pedro está casado" y eliminará de la agenda la regla aplicada, con lo que la agenda se queda vacía:

Memoria de trabajo	Agenda
f-1:(P Pedro 45 V SI) f-2:(P Maria 34 M NO)	

También podríamos haber utilizado

```
CLIPS> (run n)
```

dónde n es el número de iteraciones que queremos que se ejecuten del ciclo de control de CLIPS, o

```
CLIPS> (run)
```

que va ejecutando automáticamente todas las reglas de la agenda hasta que ésta quede vacía (equivale a `CONTROL+R`). ¡Mucho cuidado con los ciclos infinitos!

Emparejamiento en vectores ordenados de características

Podemos incluir variables en los patrones de las reglas para no tener que especificar las reglas asociadas a cada hecho, como hicimos en la sección anterior.

Sintaxis: **?<nombre de variable>**

EJEMPLO

Dado el hecho: `(Persona Pedro 45 V NO)`
Y el patrón: `(Persona ?Nombre 45 V ?Casado)`

Ambos se pueden emparejar con las sustituciones:

`?Nombre` por `Pedro`
`?Casado` por `NO`

Si no queremos usar algún valor del vector, podemos usar una variable anónima **?**

EJEMPLO

```
(defrule ImprimeSolteros
      (Persona ?Nombre  ?  ?  NO)
      =>
      (printout t crlf ?Nombre " está soltero"))
```

- Una variable se liga en la parte izquierda de la regla (LHS), en alguno de los patrones que aparezcan en su antecedente.
- Una vez ligada a un valor (cuando se ha producido un emparejamiento con alguno de los hechos de la memoria de trabajo), permanece con dicho valor.
- Aunque dos variables tengan el mismo nombre, si aparecen en distintas reglas, se consideran variables diferentes (como las variables locales en las funciones de un lenguaje de programación imperativo).

Los dos puntos anteriores pueden resumirse en:

El ámbito de una variable es la regla en la que se encuentra.

Una misma regla puede activarse con varios hechos distintos. Por cada emparejamiento de una regla con los hechos correspondientes, tendremos una activación de la regla en la agenda:

```
(defrule ImprimeSolteros
      (Persona ?Nombre  ?  ?  NO)
      =>
      (printout t crlf ?Nombre " está soltero"))
```

Memoria de trabajo	Agenda
f-1:(Persona Pedro 45 V SI) f-2:(Persona Juan 35 V NO) f-3:(Persona Maria 34 M NO) …	IS(f-2) IS(f-3)

Como siempre, el usuario ejecutará `(run n)` y se completarán `n` ciclos, en cada uno de los cuales se ejecutará una sola activación.

- No puede llegar una variable sin ningún valor instanciado a la parte derecha de la regla (RHS). En ese caso, se produce un error de compilación:

```
(defrule ReglaInvalida
      (Persona ?Nombre 45 V ?)
      =>
      (printout t crlf ?Nombre ?Casado))

Error: Undefined variable Casado referenced in RHS of defrule.
```

- Una variable podría aparecer una sola vez en una regla (en su LHS), pero esto no tiene demasiado sentido. Para eso usaremos la variable anónima `?`.

```
(defrule ImprimeSolteros                    ; Demasiadas variables
      (Persona ?Nombre ?Edad ?Sexo NO)
      =>
      (printout t crlf ?Nombre))

(defrule ImprimeSolteros                    ; Versión mejorada :-)
      (Persona ?Nombre ? ? NO)
      =>
      (printout t crlf ?Nombre))
```

- No puede ponerse una variable como primer valor de un vector ordenado de características:

```
(defrule ReglaInvalida
      (?Relacion Pedro 45 V Si)
      =>
      (printout t crlf ?Relacion))

Syntax error:
Check appropriate syntax for the first field of a pattern.
```

- Si se quiere utilizar una variable que englobe más de un valor, se usa el comodín `$?`, que representa cero, uno ó más valores de un patrón.

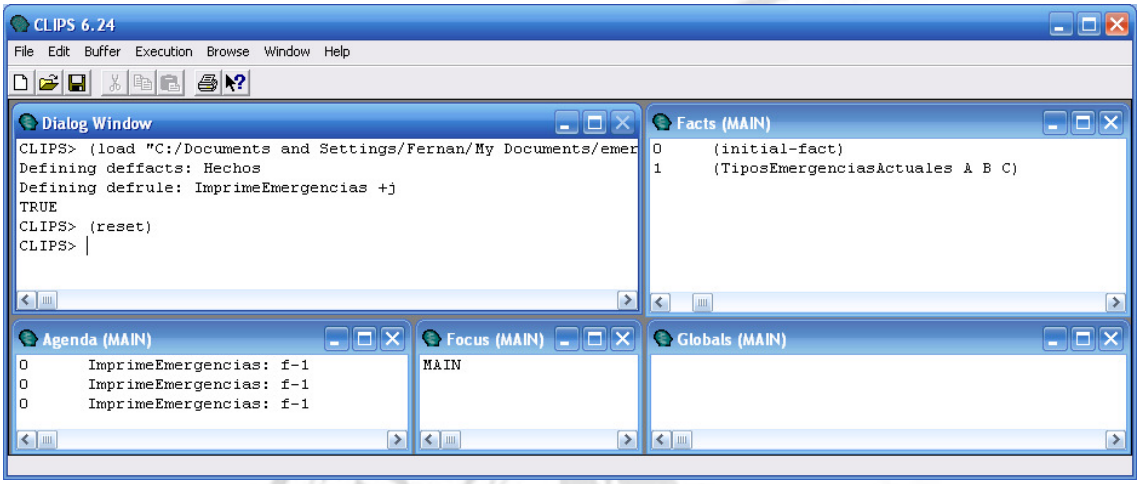
```
(defrule ImprimeNombresPersonas
      (Persona ?Nombre $?)
      =>
      (printout t crlf ?Nombre))
```

Es importante destacar que, al usar comodines, podríamos tener en la agenda varias activaciones de la misma regla con el mismo hecho:

```
(deffacts Hechos
      (TiposEmergenciasActuales A B C)
)

(defrule ImprimeEmergencias
      (TiposEmergenciasActuales $? ?T $?)
      =>
      (printout t "Emergencia -> " ?T crlf))
```

En la agenda, tendremos tres activaciones de la regla `ImprimeEmergencias`:



Al ejecutar `(run)`, obtendremos el siguiente resultado:

```
Emergencia -> A
Emergencia -> B
Emergencia -> C
```

Reglas con varios patrones

En la parte izquierda de una regla (LHS) podremos poner más de un patrón. Implícitamente, irán conectados con Y. Es decir, todos ellos deben proporcionar emparejamientos con hechos de la memoria de trabajo para que la regla sea aplicable.

¿Qué ocurre cuando una variable aparece varias veces en distintos patrones de la LHS?

La primera vez que aparece no implica ninguna restricción. Simplemente, se realiza una sustitución. Sin embargo, las siguientes veces que aparece, como ya tiene un valor concreto, sí impone una restricción en la posición en la que se encuentra:

En la siguiente regla tenemos 3 patrones en su LHS, que implícitamente van conectados por Y:

```
(defrule DiagnosticoEccema
      (FichaPaciente    ?Nombre $?)
      (DatosExploracion ?Nombre $? picor $? vesiculas $?)
      =>
      (printout t crlf  ?Nombre "tiene un eccema"))
```

1. La primera vez que aparece `?Nombre` no se aplica ninguna restricción. Simplemente, se captura el nombre de un paciente.

2. La segunda vez que aparece `?Nombre` sí se realiza restricción: Buscamos el nombre de un paciente concreto (el correspondiente al valor al que se ligó la variable `?Nombre` en el primer patrón de la regla).

3. La tercera vez que aparece `?Nombre`, ya en la RHS de la regla, tampoco se realiza restricción alguna. Sólo usamos el valor de la variable `?Nombre` para imprimirla

Obsérvese que, si un paciente no está fichado (no existe el dato correspondiente a su `FichaPaciente`), la regla `DiagnosticoEccema` no se activará con dicho paciente:

Memoria de trabajo	Agenda
f-1:(FP Pedro …) f-2:(FP Juan …) … f-30:(DE Pedro … … picor … vesiculas …) f-34:(DE Juan … … picor … vesiculas …) f-35:(DE Antonio … … picor … vesiculas …)	DE(f-1,f-30) DE(f-2,f-34)

Para mejorar la eficiencia de CLIPS, procuraremos poner primero los patrones que impliquen una mayor restricción. Por ejemplo, si los datos de la exploración del paciente son dinámicos y se atienden de uno en uno, cambiaríamos la regla anterior por:

```
(defrule DiagnosticoEccema
      (DatosExploracion ?Nombre $? picor $? vesiculas $?)
      (FichaPaciente    ?Nombre $?)
      =>
      (printout t crlf  "tiene un eccema"))
```

Emparejamiento en registros

En las reglas sólo se incluyen aquellos campos de los registros sobre los que queremos especificar una restricción, p.ej.:

alarma.clp

```
(deftemplate Emergencia
      (field tipo)
      (field sector)
      (field campo)
)

(deftemplate SistemaExtincion
      (field tipo)
      (field status)
      (field UltimaRevision)
)

(defrule Emergencia-Fuego-ClaseB
      (Emergencia
            (tipo ClaseB))
      (SistemaExtincion
            (tipo DioxidoCarbono)
            (status operativo))
      =>
      (printout t "Usar extintor CO2" crlf)
)
```

Para que la regla `Emergencia-Fuego-ClaseB` sea aplicable, debe existir un hecho del template `Emergencia` que tenga en el campo `tipo` el valor `ClaseB` y, ADEMÁS, otro hecho del template `SistemaExtincion` que tenga en su campo `tipo` el valor `DioxidoCarbono` y en su campo `status` el valor `operativo`.

alarma.datos.clp

```
(deffacts HechosSistemaExtincion
      (SistemaExtincion
            (tipo DioxidoCarbono)
            (status activado)
            (UltimaRevision diciembre)))
)
```

Normalmente, dividiremos nuestra **base de conocimiento** en

- Reglas (estáticas): Fichero `alarma.clp`
- Datos (más o menos estáticos): Fichero `alarma.datos.clp`

Para cargar nuestra base de conocimiento en CLIPS, usamos

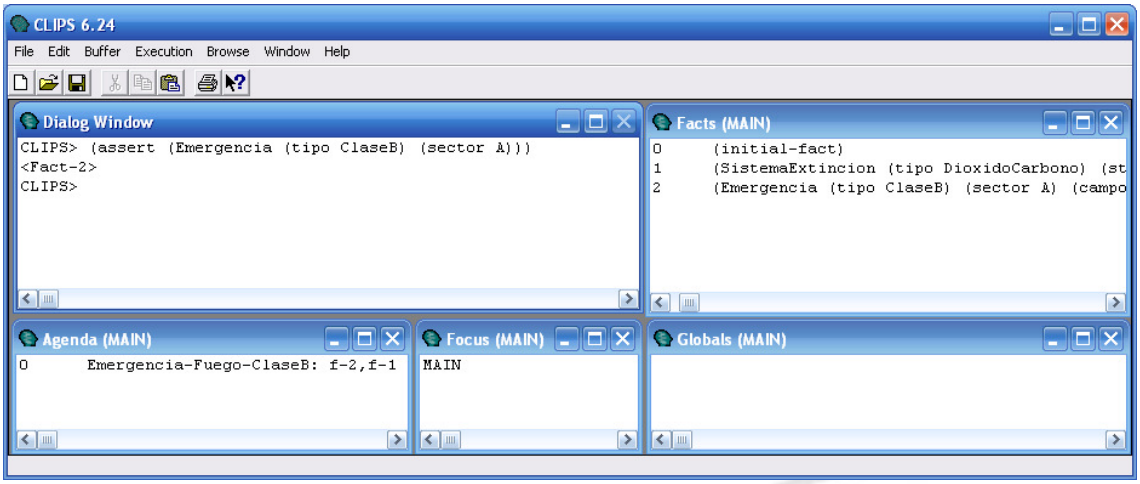
```
CLIPS> (clear)
CLIPS> (load alarma.clp)            con CONTROL+L
CLIPS> (load alarma.datos.clp)
CLIPS> (reset)                      con CONTROL+E
```

Observe que la regla `Emergencia-Fuego-ClaseB` todavía no es aplicable.

- Los **datos dinámicos** se asertarán en la memoria de trabajo con (`assert`) desde línea de comandos de CLIPS (aunque también podrían leerse desde un fichero u obtenerse automáticamente desde algún dispositivo auxiliar).

```
CLIPS> (assert (Emergencia
                  (tipo ClaseB)
                  (sector A)))
```

En este momento, la regla Emergencia-Fuego-ClaseB se activa, por lo que pasa a formar parte de la agenda:



La regla se disparará cuando se ejecute (`run 1`), por ejemplo, lo que mostrará en pantalla el mensaje "`Usar extintor CO2`".

EJEMPLO: Diagnóstico de eccemas usando templates para representar los datos de la exploración por una parte y la ficha del paciente por otra.

```
(deftemplate FichaPaciente
      (field Nombre)
      (field Casado)
      (field Direccion))

(deftemplate DatosExploracion
      (field Nombre)
      (multifield Sintomas)
      (field GravedadAfeccion))

(defrule DiagnosticoEccema
  (DatosExploracion
      (Nombre ?N)
      (Sintomas $? picor $? vesiculas $?))
  (FichaPaciente
      (Nombre ?N))
  =>
  (printout t "Posible diagnóstico para el paciente " ?N ": Eccema " crlf)
)
```

EJERCICIO: Defina los hechos necesarios para que la memoria de trabajo y la agenda de CLIPS queden como se muestra a continuación:

Memoria de trabajo	Agenda
f-1:(FP (Nombre Pedro) ....) f-2:(FP (Nombre Juan) ....) f-30: (DE (Nombre Pedro) (Sintomas (…picor vesiculas…)) …) f-34: (DE (Nombre Juan) (Sintomas (…picor vesiculas…)) …)	DE(f-30,f-1) DE(f-34,f-2)

Si nos hace falta, podemos guardar todos los valores de un `multifield` en una única variable, para lo que usaremos la notación `$?Variable`:

```
(deftemplate Persona
      (multifield Nombre)
      (field Edad)
      (field Casado)
)

(deffacts VariasPersonas
      (Persona
            (Nombre Juan Pedro Olivar)
            (Edad 33)
            (Casado No))
      …
)

(defrule ImprimeEdadSolteros
      (Persona
            (Nombre $?Nombre)
            (Edad ?Edad)
            (Casado No))
      =>
      (printout t $?Nombre " tiene " ?Edad " años" crlf)
)
```

NOTA FINAL: Obviamente, si quisiéramos, también podríamos haber mezclado patrones de vectores y patrones de registros en una misma regla.

Refracción

Cuando se ejecuta una regla, CLIPS la borra directamente de la agenda. Sólo si volvemos a añadir los hechos que la hicieron aplicable, entonces la regla volverá a entrar en la agenda.

```
CLIPS> (clear)
CLIPS> (load alarma.clp)
CLIPS> (load alarma.datos.clp)
CLIPS> (reset)
```

```
f-1
(SistemaExtincion
      (tipo DioxidoCarbono)
      (status operativo)
      (UltimaRevision diciembre)))
```

```
CLIPS> (assert (emergencia
                  (tipo ClaseB)
                  (sector A)))
```

```
f-2
(emergencia
(tipo ClaseB)
(sector A)))
```

Memoria de trabajo	Agenda
f-1 f-2	EFCB(f-2,f-1)

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

CLIPS no volverá a asignar el identificador 2 a ningún hecho.

Memoria de trabajo	Agenda
f-1	

```
CLIPS> (assert (emergencia
                  (tipo ClaseB)
                  (sector A)))
```

```
f-3
(emergencia
(tipo ClaseB)
(sector A)))
```

Memoria de trabajo	Agenda
f-1 f-3	EFCB(f-3,f-1)

Direcciones de hechos

No se permiten hechos con variables:

```
CLIPS> (assert (d ?))

      Syntax Error…

(defrule Invalida
      (a b)
      =>
      (assert (d ?)))

      Syntax error…
```

Supongamos que quisiéramos definir una regla que nos permitiese borrar más de un elemento de la memoria de trabajo. Sin embargo, NONO podemos poner una variable libre en la parte derecha de la regla:

```
(defrule Invalida
      (a b)
      =>
      (retract (d ?)))

      Syntax error…
```

Para hacer algo así, debemos calcular primero la dirección del hecho que se quiere borrar y luego hacer el `(retract)` correspondiente a su dirección en la memoria de trabajo:

```
(defrule BorraVarios
      (a b)
      ?Borrar <- (d ?)
      =>
      (retract ?Borrar))
```

En la regla anterior, `(d ?)` es un patrón de restricción usual y `<-` guarda la dirección del hecho con el que se empareja en la variable `?Borrar`

La regla `BorraVarios` se activará cuando exista un hecho `(a b)` y, además, algún hecho de la forma de la forma `(d ?)`, que será el que eliminemos de la memoria de trabajo:

Memoria de trabajo	Agenda
(a b) (d 1) (d 2)	BorraVarios(a b)(d 1) BorraVarios(a b)(d 2)
(a b) (d 2)	BorraVarios(a b)(d 2)
(a b)	

Observe que no se borran todos los hechos de golpe, sino conforme se va ejecutando cada iteración del ciclo de control de CLIPS (realizando aplicaciones individuales de las reglas de la agenda).

Supongamos ahora que tenemos dos reglas, la anterior más `Otra`:

```
(defrule BorraVarios
      (a b)
      ?Borrar <- (d ?)
      =>
      (retract ?Borrar))

(defrule Otra
      (d ?)
      =>
      (printout t "OK"))
```

Memoria de trabajo	Agenda
(a b) (d 1) (d 2)	BorraVarios(a b)(d 1) BorraVarios(a b)(d 2) Otra(d 1) Otra(d 2)
(a b) (d 2)	BorraVarios(a b)(d 2) Otra(d 2)
(a b)	

La regla `Otra` nunca llegó a aplicarse porque antes de poder hacerlo, se suprimieron las condiciones que hacían posible su aplicabilidad, por lo que se eliminó de la agenda tras las aplicaciones de `BorraVarios`.

Restricciones sobre campos

not ~

EJEMPLO: Mostrar las personas solteras cuyo color de pelo no sea marrón.

```
(defrule SolteroNoMarron
      (Persona
            (Nombre ?N)
            (ColorPelo ~Marron)
            (Casado No))
      =>
      (printout t ?N " no tiene pelo marrón" crlf))
```

or |

EJEMPLO: Mostrar las personas con el pelo de color marrón o negro.

```
(defrule PersonaMarronONegro
      (Persona
            (Nombre ?N)
            (ColorPelo Marron|Negro))
      =>
      (printout t ?N " tiene pelo marrón ó negro" crlf))
```

and &

EJEMPLO: Mostrar las personas cuyo color de pelo no sea ni marrón ni negro.

```
(defrule PersonaNiMarronNiNegro
      (Persona
            (Nombre ?N)
            (ColorPelo ~Marron & ~Negro))
      =>
      (printout t ?N " no tiene pelo marrón ni negro" crlf))
```

& también se utiliza para utilizar restricciones sobre campos en combinación con variables (para asociar a una variable el valor que tenga un hecho en el campo sobre el que se está definiendo la restricción):

```
(defrule PersonaMarronONegro
      (Persona
            (Nombre ?N)
            (ColorPelo ?Color & Marron|Negro))
      =>
      (printout t "El color de pelo de " ?N " es " ?Color crlf))
```

¡OJO! La variable (?) debe preceder a la restricción (&).

Lo siguiente, obviamente no tiene sentido en CLIPS:

```
(defrule SinSentido
      (Persona
            (Nombre ?N)
            (ColorPelo Marron & Negro))
      =>…
```

El orden de precedencia de los operadores que se utilizan para definir restricciones sobre campos es el habitual en cualquier lenguaje de programación: ~ & |

Sólo hay una excepción: Cuando & se usa al principio de la restricción en combinación con variables, entonces tiene máxima prioridad (en realidad, & está "sobrecargado").

EJERCICIO:
Mostrar el nombre y color de pelo de dos personas que:

- Una de ellas tenga, o bien los ojos azules, o bien los ojos verdes, pero que no tenga el pelo negro.
- La otra, que no tenga el mismo color de ojos que la primera y que tenga, o bien el pelo rojo, o bien el mismo color de pelo que la primera.

```
(defrule ParejaComplicada
      (Persona
            (Nombre ?N1)
            (ColorOjos ?Ojos1 & azul|verde)
            (ColorPelo ?Pelo1 & ~negro))
      (Persona
            (Nombre ?N2 & ~?N1)
            (ColorOjos ?Ojos2 & ~?Ojos1)
            (ColorPelo ?Pelo2 & rojo | ?Pelo1))
      =>
      (printout t ?N1 " tiene los ojos " ?Ojos1
                  " y el pelo " ?Pelo1 crlf
                ?N2 " tiene los ojos " ?Ojos2
                  " y el pelo " ?Pelo2 crlf))
```

Se usa ?N2 & ~?N1 por claridad (y también por eficiencia, ya que la restricción de los ojos hará que el mismo hecho no se empareje con los dos patrones).

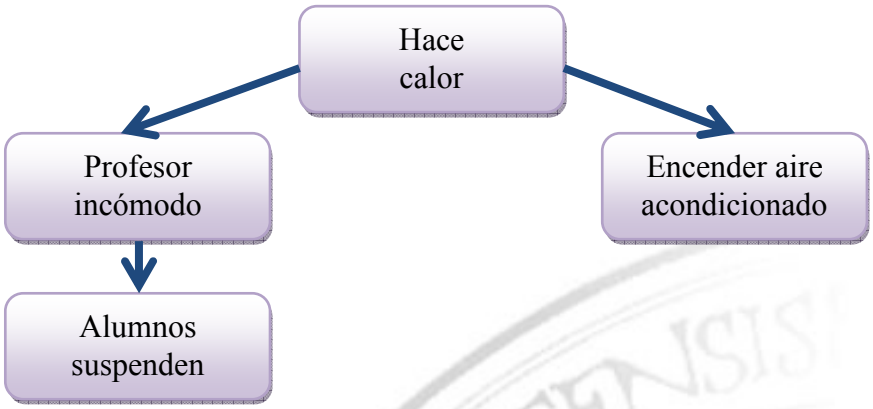
Estrategias de resolución de conflictos

¿Qué criterios generales puede seguir un SEBR hacia adelante a la hora de escoger qué regla de la agenda es la que se va a aplicar?

EJEMPLOS

Si tenemos varios pacientes, dependiendo del orden de aplicación de las reglas, o bien obtenemos primero todos los diagnósticos y luego se determinan las terapias adecuadas para cada uno, o bien se calcula un diagnóstico para una persona, a continuación su terapia, y se repite el proceso iterativamente con el resto de los pacientes.

A veces, elegir una regla antes que otra, puede tener consecuencias más importantes:



En el caso de CLIPS, podemos utilizar el comando

```
(set-strategy <strategy>)

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

Estrategia "primero en profundidad" (LIFO)

En inglés: Depth-first / LIFO (Last In, First Out)

Se ejecutan antes aquellas reglas que acaban de entrar en la agenda.

Supongamos las siguientes reglas, donde + representa `(assert)`:

```
R1: A => +C          R3: H => +I
R2: A => +B          R4: H => +J
```

```
CLIPS> (assert (A))
                     ; Se activan las reglas R1 y R2

CLIPS> (assert (H))
                     ; Se activan las reglas R3 y R4

CLIPS> (run)
                     ; R3 y R4 se aplicarán antes que R1 y R2.
                     ; Entre R3 y R4 no hay definido ningún orden específico.
                     ; Lo mismo ocurre con R1 y R2.
```

Estrategia "primero en anchura" (FIFO)

En inglés: Breadth-first / FIFO (First In, First Out)

Se ejecutan antes aquellas reglas que llevan más tiempo en la agenda:

```
R1: A => +C          R3: H => +I
R2: A => +B          R4: H => +J
```

```
CLIPS> (assert (A))
                     ; Se activan las reglas R1 y R2
CLIPS> (assert (H))
                     ; Se activan las reglas R3 y R4
CLIPS> (run)

                     ; R1 y R2 se aplicarán antes que R3 y R4.
                     ; Entre R1 y R2 no hay definido ningún orden específico.
                     ; Lo mismo ocurre con R3 y R4
```

Por complejidad

Complexity en CLIPS

Se ejecutan antes aquellas reglas que tienen un mayor número de antecedentes, ya que se consideran reglas más específicas (menos generales).

```
R1:
Si el paciente tiene insomnio, ansiedad, dolores de cabeza,
   irritabilidad, taquicardia, inquietud y consumo excesivo de cafeína
Entonces
   Diagnóstico = Adición a la cafeína

R7:
Si el paciente tiene insomnio, ansiedad, dolores de cabeza,
   irritabilidad, taquicardia, inquietud
Entonces
   Diagnóstico = Ansiedad Neurótica.
```

Por antigüedad

En inglés: Recency of facts matching patterns

Se ejecutan antes aquellas reglas que se emparejan con los hechos más recientes de la memoria de trabajo (los que se supone aportan la información más novedosa0.

Usaremos un contador de ciclos de control (emparejamiento → resolución de conflictos → aplicación de una regla) y, a cada hecho de la memoria de trabajo, le asociaremos el valor del contador correspondiente al ciclo en el que se añadió a la memoria de trabajo.

Se aplicarán antes aquellas reglas que tengan algún patrón que se empareje con el hecho más reciente. En caso de empate, se procede a comparar el siguiente emparejamiento 'más reciente'.

EJEMPLO

```
R1: A B => +C, +D
R2: A C => +E
R3: C D => +F
```

Memoria de trabajo	Agenda	Resolución	Acción
(A,0) (B,0)	R1	R1	+C +D
(A,0) (B,0) (C,1) (D,1)	R2 R3	R3	+F
(A,0) (B,0) (C,1) (D,1) (F,2)	R2	R2	+E
(A,0) (B,0) (C,1) (D,1) (F,2) (E,3)			

EJEMPLO

```
R1 : A B => +C, +D
R1b: A H F => -F
R2 : A C => +E
R3 : C D H => +G -H
```

Memoria de trabajo	Agenda	Resolución	Acción
(A,0) (B,0) (H,0) (F,0)	R1 R1b	R1 (arbitrario)	+C +D
(A,0) (B,0) (H,0) (F,0) (C,1) (D,1)	R1b R2, R3	R3	+G -H
(A,0) (B,0) (F,0) (C,1) (D,1) (G,2)	R2	R2	+E
(A,0) (B,0) (F,0) (C,1) (D,1) (G,2) (E,3)			

NOTA: En el segundo ciclo, la estrategia FIFO habría elegido la regla `R1b`, mientras que la estrategia LIFO se habría decantado indistintamente por `R2` o `R3`. Sin embargo, según el criterio de ordenación por antigüedad, la resolución de conflicto se hará siempre a favor de `R3`.

LEX

Es la estrategia resultante de aplicar primero el criterio de ordenación por antigüedad, y en caso de empate, aplicar el criterio de complejidad.

NOTA: LEX y LIFO se usan cuando se quiere seguir una línea de razonamiento concreta (primero en profundidad) y no llevar varias a la vez en paralelo (para lo cual es preferible una estrategia primero en anchura).

Especificación de prioridades

Las estrategias de resolución de conflicto son métodos generales para ordenar las activaciones de las reglas dentro de la agenda, pero el programador puede indicar expresamente que una regla tiene más prioridad que otra, por lo que figuraría antes el la agenda independientemente del criterio de resolución de conflictos que se utilice:

```
(defrule Meningitis
      (declare (salience 1000))
      (Sintomas ?N $? manchas violáceas $?)
      =>
      (assert (DiagnosticoPosible Meningitis)))
```

La regla `Meningitis` se situará en la agenda por encima de todas aquellas reglas que tengan prioridad menor a 1000.

NOTA: Por defecto, las reglas tienen prioridad 0.

Funciones aritméticas y lógicas

Funciones aritméticas

Operadores aritméticos en CLIPS:

+ - * / mod

En CLIPS se usa una notación prefija para las expresiones aritméticas, por lo que no existen relaciones de precedencia entre operadores (la evaluación se realiza de izquierda a derecha):

```
CLIPS> (+ 2 3 4)

      9

CLIPS> (assert (hecho (+ 2 3)))

      (hecho 5)

CLIPS> (assert (hecho + 2 3))      ; ¡OJO!

      (hecho + 2 3)
```

Para evitar realizar cálculos de tipo aritmético dentro de una regla, tanto en su LHS como en su RHS, hay que utilizar un operador de asignación:

(bind ?Variable expresión)

EJEMPLO

Supongamos que tenemos datos de varias personas, entre los que se incluyen sus ingresos mensuales. Para todos ellos, queremos averiguar la suma de sus ingresos junto con los de su cónyuge:

```
(defrule CalculaIngresosTotalesFamilia
      (Persona
            (Nombre ?N1)
            (Sexo V)
            (CasadoCon ?Esposa)
            (Ingresos ?Ing1))
      (Persona
            (Nombre ?Esposa)
            (Ingresos ?Ing2))
      =>
      (bind ?Total (+ ?Ing1 ?Ing2))
      (printout t "Ingresos Totales de " ?N1 " = " ?Total)
```

Lectura de datos

La instrucción `bind` también puede usarse para leer desde teclado:

```
(defrule ObtenerNombreUsuario
      =>
      (printout t "Introduzca su nombre" crlf)
      (bind ?Respuesta (read))
      (assert (PacienteNuevo ?Respuesta)))
```

NOTA: Obsérvese la LHS vacía de la regla (se empareja con el hecho inicial f-0) y cómo añadimos a la memoria de trabajo un hecho a partir de los datos introducidos por el usuario.

Funciones lógicas

CLIPS: Predicate functions

Las funciones lógicas de devuelven verdadero (símbolo `TRUE`) o falso (símbolo `FALSE`):

```
CLIPS> (and (> 4 3) (> 4 5))

      FALSE

CLIPS> (integerp 3.5)

      FALSE
```

En el manual de referencia de CLIPS se puede encontrar una lista completa de las funciones lógicas que podemos utilizar, algunas de las cuales son:

`=  <>  >  <  >=  <=`

`and  or  eq  neq  integerp  oddp`

NOTA: `eq (neq)` compara que sean iguales dos valores en tipo y valor, mientras que `= (<>)` sólo tiene en cuenta el valor.

Una función lógica no puede aparecer directamente como patrón de la LHS de una regla. Necesitamos anteponer el condicional lógico `test`:

```
(test (neq ?Variable valor))
```

EJEMPLO

Mostrar el nombre de las personas mayores de edad

```
(defrule MuestraMayor18
      (Persona
            (Nombre ?N)
            (Edad ?Edad))
      (test (> ?Edad 18))
      =>
      (printout t ?N "es mayor de edad"))
```

EJERCICIO

```
R0:   Si hay placas (puntos blancos) en la garganta
      Entonces Posible Diagnóstico es Infección garganta

R1:   Si garganta inflamada
      Y sospechamos infección bacteriana
      Entonces posible diagnóstico es Infección garganta

R2:   Si temperatura paciente > 37
      Entonces paciente tiene fiebre

R3:   Si paciente enfermo más de una semana
      Y paciente tiene fiebre
      Entonces sospechamos infección bacteriana
```

Use un template `ExploracionPaciente`, un template `DiagnosticoPaciente` y vectores para representar hechos intermedios del tipo `(Paciente Juan TieneFiebre)` o `(Paciente Juan PosibleInfeccionBacteriana)`.

SOLUCIÓN EN LA PÁGINA SIGUIENTE…

```
(defrule Plaquetas
      (ExploracionPaciente
            (Nombre  ?N)
            SintomasGarganta $? Plaquetas $?))
      =>
      (assert (DiagnosticoPaciente
                  (Nombre ?N)
                  (Diagnostico InfeccionBacterianaGarganta)))
)

(defrule PosibleInfeccionBacteriana
      (ExploracionPaciente
            (Nombre ?N)
            (NumSemanas ?Sem))
      (test (> ?Sem 2))
      (Paciente ?N TieneFiebre)
      =>
      (assert (Paciente ?N PosibleInfeccionBacteriana))
)

(defrule PacienteConFiebre
      (ExploracionPaciente
            (Nombre ?N)
            (Temperatura ?T))
      (test (> ?T 37))
      =>
      (assert (Paciente ?N TieneFiebre))
)

(defrule InfecBactGarg
      (ExploracionPaciente
            (Nombre  ?N)
            (SintomasGarganta  $? Inflamacion $?))
      (Paciente ?N PosibleInfeccionBacteriana)
      =>
      (assert (DiagnosticoPaciente
                  (Nombre ?N)
                  (Diagnostico InfeccionBacterianaGarganta)))
)

(defrule MuestraDiagnostico
      (DiagnosticoPaciente
            (Nombre ?N)
            (Diagnostico ?D))
      =>
      (printout t crlf "El diagnostico para " ?N " es: " ?D crlf)
)
```

Al principio, se añadirán los hechos del template `ExploracionPaciente` y se irán encadenando las reglas. Si también tuviésemos reglas para la terapia, entonces, dependiendo de la estrategia de control usada, o bien se mostrarían todos los diagnósticos y luego todas las terapias (en anchura), o bien se mostrarían el diagnóstico y terapia de cada persona (en profundidad).

Emparejamiento y restricciones avanzadas

Restricciones sobre campos
[Field constraints]

Restricciones con valores devueltos por expresiones aritméticas (=)
[return value field constraints]

Si queremos establecer una restricción sobre un campo numérico, podemos usar:

```
(Persona
      (Edad 27))
```

Pero si queremos hacer lo mismo usando el valor devuelto por una función numérica:

```
(Persona
      (Edad = (+ 24 3))
```

NOTA: Es necesario utilizar el conectivo =.

Restricciones con valores devueltos por expresiones lógicas (:)
[predicate field constraints]

Para comparar con el valor `TRUE` o `FALSE` devuelto por una función lógica, en vez de utilizar = igual que en las expresiones aritméticas, ahora usaremos **:**

Además, como este valor lo emplearemos como parte de la expresión lógica que usemos para imponer restricciones sobre campos, también necesitaremos el conectivo **&**

```
(defrule MuestraMayor18
      (Persona
            (Nombre ?N)
            (Edad ?Edad & :(> ?Edad 18)))
      =>
      (printout t ?N "es mayor de edad")
)
```

¡OJO! La siguiente regla no sería válida:

```
(defrule Invalida
      (Persona (Edad (> 18))
      …
)
```

Elementos condicionales
[Conditional elements]

- Para conectar patrones con otros conectivos lógicos: `or`, `and`, `not`...
- Para realizar otro tipo de comprobaciones que no correspondan a emparejamientos con hechos de la memoria de trabajo: `test`, `exists`,...

(test)

Se usa normalmente en conjunción con una función lógica, para poder especificar un patrón en la LHS de una regla, como ya vimos anteriormente.

(or), (and)

Si tenemos una regla de la forma:

```
si (A o B) y C entonces D
```

Podríamos desglosarla en dos aplicando las leyes de Morgan:

```
si A y C entonces D
si B y C entonces D
```

Pero también podríamos usar directamente el formato inicial de la regla con los elementos condicionales `or` y `and` (por lo general, preferiremos desglosar las reglas).

ADVERTENCIA: Mucho cuidado con el uso de variables en reglas con `or` (que se descomponen en reglas distintas y, por tanto, el ámbito de sus variables es distinto).

```
(defrule Plaquetas
      (or
            (ExploracionPaciente
                  (Nombre ?N1)
                  (SintomasGarganta $? Plaquetas $?))
            (ExploracionPaciente
                  (Nombre ?N2)
                  (SintomasGarganta  $? Inflamacion $?))
      )
      =>
      (assert
            (DiagnosticoPaciente
                  (Nombre ?N3)
                  (Diagnostico InfeccionBacterianaGarganta)))
)
```

Interpretación de la variable `?N`:
Si se satisface el patrón en que aparece $?N_1$, $?N_3$ se queda con el valor de $?N_1$ (ni se llega a evaluar $?N_2$). Si $?N_1$ no se puede asignar a un valor que satisfaga el primer patrón, $?N_2$ se considera una nueva variable.

(not)

Se usa para activar una regla cuando no exista ningún hecho que empareje con un patrón determinado.

```
(defrule InformaEmergencia
      (tarea ObtenerInforme)
      (emergencia (tipo ?T))
      =>
      (printout t "Solucionando emergencia del tipo" ?T))


(defrule NoHayEmergenciasB
      (tarea ObtenerInforme)
      (not (emergencia (Tipo B)))
      =>
      (printout t "No hay ninguna emergencia de tipo B"))


(defrule NoHayEmergencias
      (tarea ObtenerInforme)
      (not (emergencia))
      =>
      (printout t "No hay ninguna emergencia"))
```

`not` también puede usarse en expresiones lógicas con `test`:

```
(defrule Valida
      (vector ?numero)
      (not (test (> ?numero 5)))
      =>
      (printout t "ok"))
```

aunque, obviamente, este ejemplo quedaría mejor como:

```
(defrule Valida
      (vector ?numero)
      (test (<= ?numero 5))
      =>
      (printout t "ok"))
```

EJEMPLO:
Compuebe que no se produjeron nacimientos en una fecha concreta.

```
(defrule NoNacidosDia
      (tarea ComprobarNoNacidos ?fecha)
      (not (Persona (FechaNac ?fecha)))
      =>
      (printout t "No se produjeron nacimientos el " ?fecha crlf))
```

Nuevamente, hay que tener cuidado con el uso de variables. Dentro del ámbito de un `not`, una misma variable conserva su valor. Al salir del ámbito, incluso dentro de la LHS, se desliga el valor que tuviese.

```
(defrule NoNacidosDiaINCORRECTA
      (not (Persona (FechaNac ?fecha)))
      (tarea ComprobarNoNacidos ?fecha)
      =>
      (printout t "No se produjeron nacimientos el " ?fecha crlf))
```

¡OJO! Después del `not`, al llegar a `tarea`, ¡`?fecha` es una variable libre!

EJERCICIO:
Determine si no hay dos personas distintas que hayan nacido en la misma fecha.

```
(defrule NoNacidosMismaFechaINCORRECTA
      (Persona (Nombre ?N)
               (FechaNac ?fecha))
      (not (Persona (Nombre ~?N)
                    (FechaNac ?fecha))))
      =>
      (printout t “No hay dos personas nacidas el mismo día” crlf))
```

La regla se activa en cuanto haya dos personas que no hayan nacido el mismo día, por lo que nuestro primer intento no funciona correctamente.

```
(defrule NoNacidosMismaFecha
      (not  (and  (Persona (Nombre ?N)
                           (FechaNac ?fecha))
                  (Persona (Nombre ~?N)
                           (FechaNac ?fecha))))
      =>
      (printout t "No hay dos personas nacidas el mismo día" crlf))
```

(exists)

```
(defrule AvisaOperador
      (emergencia (tipo B))
      =>
      (printout t "Emergencia B: Avisar operador"))
```

Si hay varias emergencias de tipo B activas, se activará varias veces la regla `AvisaOperador`. Si queremos evitar esos avisos duplicados, usamos el condicional `exists`:

```
(defrule AvisaOperador
      (exists (emergencia (tipo B)))
      =>
      (printout t "EmergenciaB: Avisar operador"))
```

NOTA: Cuando se active la regla anterior, aparecerá en la agenda: `AvisaOperador f-0`. Si tenemos una regla `R` con tres patrones en la LHS, el segundo correspondiente a un `exists`, y que se activa con los hechos `f-7` (primer patrón) y `f-35` (el tercer patrón), en la agenda aparecerá: `R f-7,,f-35`

Podemos consultar si existe cualquier hecho de tipo emergencia:

```
(defrule AvisaOperador
      (exists (emergencia))
      =>
      (printout t "Emergencia: Avisar operador"))
```

Igual que con `not`, debemos tener mucho cuidado con el uso de variables:

```
(defrule Invalida
      (exists (emergencia (tipo ?T)))
      =>
      (printout t "Emergencia tipo: " ?T))
```

La regla no es válida ya que, aunque puede aparecer una variable con el `exists`, ésta no se liga a ningún valor concreto, por lo que `?T` aparece libre en la RHS y se produce un error de compilación.

```
(defrule AvisaOperador
      (TiposEmergenciasActuales $? ?T $?)
      (exists (emergencia (tipo ?T)))
      =>
      (printout t "Emergencia tipo: " ?T))
```

Aquí ?T si tiene un valor asociado ☺

El condicional `exists` también puede utilizarse sobre los vectores ordenados de características:

```
CLIPS> (assert (emergencia B normal))
CLIPS> (assert (emergencia C urgente))
CLIPS> (assert (emergencia D urgente))
```

```
(defrule AvisarOperador
      (exists (emergencia $?))
      =>
      (printout t "Emergencia!"))
```

O, si se prefiere ser más explícito:

```
(defrule AvisarOperadorUrgente
      (exists (emergencia $? urgente ))
      =>
      (printout t "Emergencia urgente"))

(defrule AvisarOperadorNormal
      (exists (emergencia $? normal))
      =>
      (printout t "Emergencia normal"))
```

Uso de hechos de control

Los hechos de control son hechos añadidos por el programador para controlar la activación de la reglas. Usualmente, se utilizan para representar fases y tareas.

EJEMPLO
Se tienen datos de personas e información sobre posibles cambios de domicilio:

```
(deftemplate Persona
      (field Nombre)
      (field Direccion))

(deftemplate CambioDomicilio
      (field Nombre)
      (field NuevaDireccion))
```

Escribimos una regla que actualiza la dirección de aquéllos que se van a mudar.

```
(defrule ActualizaDireccion
      (CambioDomicilio
            (Nombre ?N)
      (NuevaDireccion ?D))
      ?Persona <- (Persona (Nombre ?N))
      =>
      (modify ?Persona (Direccion ?D)))
```

En primer lugar, incluimos el patrón `CambioDomicilio` por razones obvias de eficiencia. Normalmente, habrá muchas personas registradas que cambios de domicilio. Por lo tanto, primero fijamos el nombre de la persona que se va a mudar y luego buscamos sus datos personales.

Pero… LA REGLA **NO** FUNCIONA.

Supongamos que la regla se activa con los hechos:

```
f-34  (Cambio
            (Nombre Juan)
            (NuevaDireccion AvGranada))

f-6   (Persona
            (Nombre Juan)
            (Direccion AvCadiz))
```

Al ejecutarse la regla, se modifican los datos de la persona `f-6`, y CLIPS considera que se trata de un nuevo hecho, con su identificador asociado (por ejemplo, `f-50`). Por lo tanto, la regla `ActualizaDireccion` es de nuevo aplicable a los hechos `f-34` y `f-50`. Entramos en un bucle infinito…

SOLUCIÓN

Hay que ver interpretar el template `CambioDomicilio` como una tarea. En cuanto hayamos realizado la tarea de modificar el domicilio, la suprimiremos de la memoria de trabajo:

```
(defrule ActualizaDireccion
      ?tareaCambio <- (Cambio (Nombre ?N)
                              (NuevaDireccion ?D))
      ?Persona <- (Persona (Nombre ?N))
      =>
      (retract ?tareaCambio)
      (modify ?Persona (Direccion ?D)))
```

EJEMPLO

Supongamos que tenemos datos de varias personas, incluyendo sus ingresos mensuales, y queremos modificando los ingresos de cada persona que esté casada, de tal forma que aparezca la suma de sus ingresos y los de su cónyuge.

```
(defrule ComputaIngresosTotales
      ?Persona1 <- (Persona (Nombre ?N1)
                            (Sexo V)
                            (CasadoCon ?Esposa)
                            (Ingresos ?Ing1))
      ?Persona2 <- (Persona (Nombre ?Esposa)
                            (Ingresos ?Ing2))
      =>
      (bind ?Total (+ ?Ing1 ?Ing2))
      (modify ?Persona1 (Ingresos ?Total))
      (modify ?Persona2 (Ingresos ?Total)) )
```

El uso de `bind` es correcto, pero la regla tiene un fallo: Al modificar los ingresos de un cónyuge, obtenemos un nuevo hecho que vuelve a disparar la misma regla, por lo que se entra en un bucle infinito.

Para intentar evitarlo, separaremos los ingresos propios de los ingresos de la unidad familiar:

```
(deftemplate Persona
      (field Nombre)
      …
      (field Ingresos)
      (field IngresosUF)
)
```

Veamos cómo evitar el ciclo infinito de dos formas diferentes:

SOLUCIÓN A

Obligando a que el valor del campo IngresosUF esté a `nil`. En cuanto se modifique este valor, la regla no volverá a ser aplicable con la misma persona, ya que `nil` sólo se empareja con `nil`.

```
(defrule IngresosTotalesCasados
      ?P1 <- (Persona (Sexo V)
                      (Ingresos ?I1)
                      (CasadoCon ?Esposa)
                      (IngresosUF nil))
      ?P2 <- (Persona (Sexo F)
                      (Nombre ?Esposa)
                      (Ingresos ?I2))
      =>
      (bind ?Total (+ ?I1 ?I2))
      (modify ?P1 (IngresosUF ?Total))
      (modify ?P2 (IngresosUF ?Total)))


(defrule IngresosTotalesSolteros
      ?P1 <- (Persona (Sexo V)
                      (Ingresos ?I1)
                      (IngresosUF nil)
                      (CasadoCon nil))
      =>
      (modify ?P1 (IngresosUF ?I1)))
```

En un sistema estático, no habría problema si hacemos algo así, pero si suponemos que los ingresos de una persona pueden cambiar durante la ejecución de CLIPS, entonces la regla `IngresosTotalesCasados` no funcionará como desearíamos (ya que el campo `IngresosUF` no es igual a `nil` una vez activada la regla, por lo que no se actualizarán adecuadamente los ingresos de la unidad familiar si cambias los ingresos de la persona).

Pero tampoco podemos eliminar la restricción `nil` sin entrar de nuevo en un bucle infinito…

SOLUCIÓN B

Vamos a cambiar nuestra estructura de datos creando otro template en el que se almacene la información de cada persona que resulta relevante para Hacienda, incluyendo los ingresos de la unidad familiar.

```
(deftemplate Persona
      (field Nombre)
      (field Edad)
      (field CasadoCon)
      (field Ingresos)
      (field Sexo))

(deftemplate Hacienda
      (field NombreUF)
      (field IngresosUF))


(defrule IngresosTotalesCasados
      (Persona (Sexo V)
               (Nombre ?N1)
               (Ingresos ?I1)
               (CasadoCon ?Esposa))
      (Persona (Sexo F)
               (Nombre ?Esposa)
               (Ingresos ?I2))
      =>
      (assert (Hacienda
                 (NombreUF ?N1)
                 (IngresosUF (+ ?I1 ?I2) )))

(defrule IngresosTotalesSolteros
      (Persona (Nombre ?N)
               (Ingresos ?I)
               (CasadoCon nil))
      =>
      (assert (Hacienda
                 (NombreUF ?N)
                 (IngresosUF ?I))))
```

Pero supongamos de nuevo que modifico el valor de los ingresos de cualquier persona. Entonces, la regla `IngresosTotalesCasados` es aplicable con el nuevo hecho (resultado de la modificación del antiguo) y se añadiría otro hecho de tipo `Hacienda` con el mismo valor de `NombreUF` pero distinto valor de `IngresosUF`.

Tenemos que refinar un poco más nuestra solución…

Incluyamos en la regla `IngresosTotalesCasados` una distinción adicional: Si la persona no estaba ya fichada por Hacienda, añadiremos el hecho correspondiente con `assert`; mientras que, si ya estaba fichada, realizaremos un `modify`.

```
(defrule IngresosTotalesCasadosNuevos
      (Persona (Sexo V)
               (Nombre ?N1)
               (Ingresos ?I1)
               (CasadoCon ?Esposa))
      (Persona (Sexo F)
               (Nombre ?Esposa)
               (Ingresos ?I2))
      (not (Hacienda
               (NombreUF ?N1)))
      =>
      (assert (Hacienda
                 (NombreUF ?N1)
                 (IngresosUF (+ ?I1 ?I2) ))))

(defrule IngresosTotalesCasadosFichados
      (Persona (Sexo V)
               (Nombre ?N1)
               (Ingresos ?I1)
               (CasadoCon ?Esposa))
      (Persona (Sexo F)
               (Nombre ?Esposa)
               (Ingresos ?I2))
      ?H <- (Hacienda (NombreUF ?N1))
      =>
      (modify ?H (IngresosUF (+ ?I1 ?I2))))
```

Pero ahora tenemos el mismo problema que al principio, ya que cada vez que modificamos un hecho con `(modify ?H (IngresosUF (+ ?I1 ?I2)))` resulta que volvemos a activar la misma regla, por lo que entra en un **bucle infinito**.

NOTA: Dentro de un `assert` se puede indicar directamente el cálculo de una función sin necesidad de utilizar `bind`

```
(assert (Hacienda
            (NombreUF ?N1)
            (IngresosUF (+ ?I1 ?I2) ))))
```

SOLUCIÓN DEFINITIVA

Usar tareas como fases intermedias en la resolución de nuestro problema.

```
(defrule IngresosTotalesCasados
      (Persona (Sexo V)
               (Nombre ?N1)
               (Ingresos ?I1)
               (CasadoCon ?Esposa))
      (Persona (Sexo F)
               (Nombre ?Esposa)
               (Ingresos ?I2))
      =>
      (assert (Tarea ActualizarIngresos ?N1 (+ ?I1 ?I2))))

(defrule IngresosTotalesCasadosAuxNuevos
      ?Tarea <- (Tarea ActualizarIngresos ?N ?I)
      (not (Hacienda (NombreUF ?N)))
      =>
      (assert (Hacienda (NombreUF ?N)
                        (IngresosUF ?I)))
      (retract ?Tarea)
)

(defrule IngresosTotalesCasadosAuxFichados
      ?Tarea <- (Tarea ActualizarIngresos ?N ?I)
      ?H <- (Hacienda (NombreUF ?N))
      =>
      (modify ?H  (IngresosUF ?I))
      (retract ?Tarea))

(defrule IngresosTotalesSolteros
      (Persona (Nombre ?N)
               (Ingresos ?I)
               (CasadoCon nil))
      =>
      (assert (Hacienda (NombreUF ?N)
                        (IngresosUF ?I))))
```

Es necesario suprimir la tarea `(ActualizarIngresos ?N ?I)` en las reglas `IngresosTotalesCasadosAux*`. Si no lo hacemos, CLIPS entendería que aún no hemos terminado la tarea y se volvería a ejecutar la regla correspondiente (los fichados entrarían en un bucle infinito y los nuevos pasarían a estar fichados, por lo que también entrarían en un bucle infinito).

Intuitivamente, si tenemos un sistema dinámico, es lógico pensar que las acciones que haya que realizar vendrán determinadas por las tareas pendientes. En el momento en el que se completa una tarea, ya no es necesario realizar nuevas acciones.

MUY IMPORTANTE: Suprimir siempre las tareas resueltas.

TMS

[Truth Maintenance System]

Los sistemas TMS tienen como finalidad mantener la coherencia de la base de datos conforme se van añadiendo y suprimiendo hechos en sistemas dinámicos.

TMS usando la hipótesis de mundo cerrado

Supongamos datos sobre pacientes y una regla que aserta un hecho del tipo `(Paciente Juan TieneFiebre)` cuando su temperatura sea mayor de 37 grados.

```
(deftemplate ExploracionPaciente
      (field Nombre)
      (field Temperatura)
      (field NumSemanas)
      (multifield SintomasGarganta)
      (multifield SintomasBronquios))

(defrule PacienteConFiebre
      (ExploracionPaciente
            (Nombre      ?N)
            (Temperatura ?T))
      (test (> ?T 37))
      =>
      (assert (Paciente ?N TieneFiebre))
)
```

Imaginemos que, igual que almacenamos el dato `(Paciente Juan TieneFiebre)`, también queremos almacenar `(Paciente Juan NoTieneFiebre)`

Esto sería imprescindible cuando la temperatura del paciente pudiese variar y hubiese que ir modificando el correspondiente tratamiento. Pero tenemos que tener cuidado para evitar hechos contradictorios en la memoria de trabajo.

¿Podemos cambiar la regla anterior por la siguiente?

```
(defrule PacienteConFiebre
      (ExploracionPaciente
            (Nombre      ?N)
            (Temperatura ?T))
      (test (> ?T 37))
      ?PF <- (Paciente ?N NoTieneFiebre)
      =>
      (retract ?PF)
      (assert (Paciente ?N TieneFiebre))
)
```

Si tenemos un nuevo paciente `X`, con 40 grados de fiebre, como no tenemos el hecho `(Paciente X NoTieneFiebre)`, resulta que la regla no se activaría.

Así pues, necesitamos otra regla para los pacientes nuevos:

```
(defrule PacienteNuevoConFiebre
      (ExploracionPaciente
            (Nombre      ?N)
            (Temperatura ?T))
      (test (> ?T 37))
      (not (Paciente ?N NoTieneFiebre))
      =>
      (assert (Paciente ?N TieneFiebre))
)
```

¿Qué pasa si durante la sesión de trabajo aumenta la temperatura del paciente?
Tenemos otro hecho distinto, luego se activa `PacienteNuevoConFiebre`. Si se dispara dicha regla, se ejecuta: `(assert (Paciente Juan TieneFiebre))`

¿Qué hace CLIPS?

Existe una opción en CLIPS para permitir, o no, hechos duplicados.

Si no se permiten hechos duplicados, al ejecutar el `assert` de un hecho ya existente, no se produce ningún cambio en la memoria de trabajo y no se lanza ninguna regla nueva. En caso contrario, tendremos dos hechos iguales con distintos identificadores

```
      (Paciente Juan TieneFiebre) -> f-34
      (Paciente Juan TieneFiebre) -> f-56
```

Como el segundo es un hecho nuevo, volverían a lanzarse las mismas reglas de diagnóstico. No hay ninguna contradicción, pero volverían a obtenerse los mismos resultados que ya se tenían anteriormente.

Resumiendo, necesitamos dos reglas y no permitir hechos duplicados:

```
(defrule PacienteConFiebre
      (ExploracionPaciente
            (Nombre          ?N)
            (Temperatura     ?T))
      (test (> ?T 37))
      ?PF <- (Paciente ?N NoTieneFiebre)
      =>
      (retract ?PF)
      (assert (Paciente ?N TieneFiebre)))

(defrule PacienteNuevoConFiebre
      (ExploracionPaciente
            (Nombre          ?N)
            (Temperatura     ?T))
      (test (> ?T 37))
      (not (Paciente ?N NoTieneFiebre))
      =>
      (assert (Paciente ?N TieneFiebre)))
```

En cualquier caso, igual que tenemos dos reglas para las subidas de temperatura, necesitamos otras dos reglas para las bajadas. En total, cuatro reglas que complican nuestra implementación.

Para solucionar este problema, impongamos la **Hipótesis de Mundo Cerrado** (si no se incluye explícitamente un hecho, se supone que éste es falso):

```
; Una regla más cercana a nuestra forma de pensar

(defrule PacienteConFiebre
      (ExploracionPaciente
            (Nombre           ?N)
            (Temperatura      ?T))
      (test (> ?T 37))
      =>
      (assert (Paciente ?N TieneFiebre))
)
```

¿Qué pasa si la temperatura aumenta?

Tenemos otro hecho distinto, por lo que se activa `PacienteConFiebre`. Si se dispara dicha regla, se añade de nuevo que tiene fiebre. Si no permitimos hechos duplicados, no hay problemas.

Pero, ¿qué pasa si la temperatura disminuye?

Ahora mismo, tenemos el hecho de que tiene fiebre, por lo que necesitamos por tanto otra regla que contemple el caso de la disminución de temperatura.

```
(defrule PacienteConFiebre
      (ExploracionPaciente
            (Nombre           ?N)
            (Temperatura      ?T))
      (test (> ?T 37))
      =>
      (assert (Paciente ?N TieneFiebre))
)

(defrule PacienteSinFiebre
      (ExploracionPaciente
            (Nombre           ?N)
            (Temperatura      ?T))
      (test (<= ?T 37))
      ?PF <- (Paciente ?N TieneFiebre)
      =>
      (retract ?PF)
)
```

Resumiendo:

SOLUCIÓN CON HIPÓTESIS DE MUNDO CERRADO

- Si la memoria de trabajo contiene un hecho del tipo `(Juan TieneFiebre)`, supondremos que es verdad.
- En caso contrario, supondremos que es falso.
- No permitiremos hechos duplicados

NOTA: El diagnóstico y la terapia de un paciente pueden depender de si una persona tiene fiebre o no. Por lo tanto, las reglas relativas a la fiebre deben aplicarse antes que las reglas relativas al diagnóstico y a la terapia. Esto se puede conseguir mediante prioridades, hechos de control o, de forma más elegante, con módulos.

TMS con (logical)

Lo que hemos hecho antes es diseñar nuestras reglas de forma que no haya hechos contradictorios en la base de datos (para mantener "la verdad"). En CLIPS también podemos implementar un TMS usando patrones con condición lógica especial: `logical`

OBJETIVO: Que CLIPS borre automáticamente los hechos deducidos con ciertas reglas cuando desaparecen las condiciones que hicieron que éstas se aplicaran.

```
(defrule R
      (logical (HaceCalor))
      =>
      (assert (JC Incomodo)))


CLIPS> (assert (HaceCalor))  ; Añade HaceCalor <f-1>


CLIPS> (run)                 ; CLIPS añade JCIncomodo


CLIPS> (retract 1)           ; Se borra HaceCalor
                             ; CLIPS borra automáticamente JCIncómodo
```

¿Qué pasa cuando `logical` se aplica sobre varios patrones?

```
(defrule R
      (logical    (A)
                  (B))
      =>
      (assert (D)))
```

La supresión de cualquiera de los dos elementos (`A` o `B`) hará que CLIPS borre `D`.

```
(defrule R1
      (logical (A))
      (C)
      =>
      (assert (D)))
```

La supresión de `A` hará que CLIPS borre `D`, pero la supresión de `C` no cambiará `D`.

EJEMPLO

```
(defrule HumosNocivosAire
      (Emergencia (tipo fuego))
      (HayHumosNocivos)
      =>
      (assert (UsarMascaraGas)))
```

Supongamos que el fuego se apaga y que, además, ya no hay humos nocivos. En la memoria de trabajo seguiría existiendo el hecho `UsarMascaraGas` cuando ya no debería estar. Veamos posibles soluciones a este problema.

Un primer intento fallido

```
(defrule HumosNocivosAire
      (logical
            (Emergencia (tipo fuego))
            (HayHumosNocivos))
      =>
      (assert (UsarMascaraGas)))
```

Si se suprime cualquier hecho que empareje con cualquier patrón incluido en el `logical`, se borrará automáticamente el hecho `UsarMascaraGas`. Pero el fuego puede estar extinguido y aún existir humos nocivos en el aire.

Mejor usamos dos reglas (sin permitir hechos duplicados):

```
(defrule MascaraGasHumos
      (logical  (HayHumosNocivos))
      =>
      (assert (UsarMascaraGas)))

(defrule MascaraGasFuego
      (logical (Emergencia (tipo fuego)))
      =>
      (assert (UsarMascaraGas)))
```

`(UsarMascaraGas)` no se borrará hasta que no se supriman los hechos `(HayHumosNocivos)` y `(Emergencia …)`.

IMPORTANTE: Lo anterior NO es equivalente a:

```
(defrule MascaraGasHumosFuego
      (logical  (HayHumosNocivos))
      (logical (Emergencia (tipo fuego)))
      =>
      (assert (UsarMascaraGas)))
```

Esta regla establece la misma dependencia que si `logical` hubiese abarcado a los dos patrones.

Si queremos usar la máscara de gas en el caso de que haya una emergencia de fuego y además tengamos humos nocivos, pero queramos quitarla en cuanto desaparezcan los humos (aunque la emergencia siga), necesitaríamos definir la siguiente regla:

```
(defrule HumosNocivosAire
      (logical (HayHumosNocivos))
      (Emergencia (tipo fuego))
      =>
      (assert (UsarMascaraGas)))
```

- El patrón `logical` debe estar siempre al principio de la regla (puede haber varios `logical`, pero todos al principio).
- Las dependencias entre los hechos sólo se establecen cuando la regla se ejecuta.
- Si tenemos reglas que se van encadenando, es importante incluir `logical` en todas. En caso contrario, se rompería la cadena y no se borrarían automáticamente los hechos asertados con las reglas.

```
(defrule R1
      (logical (A))
      =>
      (assert (B)))

(defrule R2
      (logical (B))
      =>
      (assert (C)))
```

Memoria de trabajo	Agenda	Resolución	Acción
(A)	R1	R1	+B
(A) (B)	R2	R2	+c
(A) (B) (C)	∅		
Se suprime (A)			
∅			

Si nos hubiésemos olvidado del `logical` en la segunda regla, no se habría suprimido el hecho `C`, como vemos en el siguiente ejemplo:

```
(defrule R1
      (logical (A))
      =>
      (assert (B)))

(defrule R2
      (B)
      =>
      (assert (C)))
```

Memoria de trabajo	Agenda	Resolución	Acción
(A)	R1	R1	+B
(A) (B)	R2	R2	+c
(A) (B) (C)	∅		
Se suprime (A)			
(C)			

Lo usual será que queramos que nuestro sistema funcione como en nuestro primer ejemplo y borrar también `(C)` en caso de que se borre `(A)`.

Sin embargo, supongamos que `(A)` es un patrón de un template y simplemente hacemos un `modify` de un hecho que emparejase con dicho patrón. Entonces, corremos el peligro de volver a obtener resultados ya obtenidos previamente. Para ilustrarlo, recuperemos el ejemplo del paciente con fiebre.

Utilizando la hipótesis de mundo cerrado, teníamos las siguientes reglas:

```
(defrule PacienteConFiebre
      (ExploracionPaciente
            (Nombre           ?N)
            (Temperatura      ?T))
      (test (> ?T 37))
      =>
      (assert (Paciente ?N TieneFiebre))
)

(defrule PacienteSinFiebre
      (ExploracionPaciente
            (Nombre           ?N)
            (Temperatura      ?T))
      (test (<= ?T 37))
      ?PF <- (Paciente ?N TieneFiebre)
      =>
      (retract ?PF)
)
```

Nos plantemos usar ahora una única regla:

```
(defrule PCF
      (logical
            (ExploracionPaciente
                  (Nombre           ?N)
                  (Temperatura      ?T))
            (test (> ?T 37))
      )
      =>
      (assert (Paciente ?N TieneFiebre))
)
```

- Supongamos un paciente nuevo, `Juan`, con 39º, cuyos datos de exploración corresponden al hecho `f-10`. Se aplica `PCF` y se añade `(Paciente Juan TieneFiebre)`. Este hecho disparará las reglas de diagnóstico y terapia correspondientes.

- Supongamos ahora que baja su temperatura. Al hacer `(modify 10 (Temperatura 37))`, tenemos otro hecho, `f-20`, que sustituye al antiguo `f-10`. Por tanto, CLIPS borra `(Paciente Juan TieneFiebre)` y la regla `PCF` no se dispara ya que `test` devuelve `FALSE`.

- Supongamos que, en vez de bajar, sube más su temperatura. Al hacer `(modify 10 (Temperatura 40))`, tenemos otro hecho, `f-20`, que sustituye al antiguo `f-10`. Por tanto, CLIPS borra `(Paciente Juan TieneFiebre)` y la regla `PCF` se vuelve a disparar ya que `test` devuelve `TRUE`, lo que añade un nuevo hecho `(Paciente Juan TieneFiebre)`. Este hecho volverá a activar las reglas de diagnóstico y de terapia, ofreciéndose otra vez los mismos diagnósticos y terapias que ya se hubiesen obtenido. Si no es eso lo que queremos, deberíamos prescindir de `logical`, asumir HMC y usar el par de reglas anterior.

Cuando la parte derecha de una regla implica la ejecución de procedimientos y tareas, no suele utilizarse el conectivo `logical` en la parte izquierda de la misma.

```
(defrule EncenderAA
      (HaceCalor)
      (AparatoAA
            (Status OFF))
      =>
      (assert (tarea EncenderAA)))

(defrule ApagarAA
      (not (HaceCalor))
      (AparatoAA
            (Status ON))
      =>
      (assert (tarea ApagarAA)))
```

NOTA: Las tareas las suprimirán aquellas reglas que procedan a su resolución.

Esto podría acarrear un problema en la práctica: cuando se suprima `(HaceCalor)` antes de que se ejecute la regla encargada de resolver la tarea `(tarea EncenderAA)`, o bien se haya encendido el AA por alguna otra causa.

Si esto es posible, entonces es necesario usar `logical`, para que se borre automáticamente la tarea y la regla encargada de su resolución desaparezca de la agenda:

```
(defrule EncenderAA
      (logical
            (HaceCalor)
            (AparatoAA
                  (Status OFF)))
      =>
      (assert (tarea EncenderAA)))
```

Por contra, supongamos que la regla encargada de resolver la tarea se ha aplicado y se ha suprimido la tarea. ¿Qué pasará si se suprime `(HaceCalor)`? Debido al `logical`, CLIPS intentará borrar automáticamente el hecho `tarea`, pero este ¡ya no existe! No pasa nada, no se produce ningún error.

Con la regla `ApagarAA` habría que realizar las mismas consideraciones.

Pero, ¿cómo funciona un `not` dentro de un `logical`?

```
(defrule prueba
      (logical (not (A)))
      =>
      (assert (B)))
```

Si no existe `(A)`, se ejecuta la regla y se añade `(B)`.

Si se añade ahora `(A)`, automáticamente se suprime `(B)`.

```
(defrule ApagarAA
      (logical
            (not (HaceCalor))
            (AparatoAA
                  (Status ON)))
      =>
      (assert (tarea ApagarAA)))
```

Si se añade `(HaceCalor)` antes de resolver la tarea `(tarea ApagarAA)`, ésta se suprimirá automáticamente.

Modularización

En los sistemas expertos de diagnóstico se distinguen distintas fases:

1. **Detección del problema**: ¿Qué ha fallado?
2. **Diagnóstico**: ¿Por qué ha fallado?
3. **Resolución** (terapia): ¿Cómo resolver ó solucionar el fallo?

Para conseguir que primero se disparen las reglas de detección, luego las de diagnóstico y, finalmente las de terapia, podemos utilizar distintas estrategias:

- Asignarles a todas las reglas de una fase la misma prioridad, pero mayor que la prioridad de las reglas de una fase posterior.
- Utilizar hechos de control.
- Definir módulos de CLIPS.

Modularización con hechos de control

EJEMPLO: Diagnóstico médico.

```
(defrule LeerSintomas
      ?fase <- (fase LecturaSintomas ?Nombre)
      =>
      (printout t "Introduzca los síntomas de " ?Nombre " separados
      por espacios y ordenados alfabéticamente" crlf)
      (retract ?fase))

(defrule DiagnosticoEccema
      (Sintomas ?Nombre $? picor $? vesiculas $?)
      =>
      (printout t "El diagnóstico para " ?Nombre " es: Eccema" crlf))
```

```
CLIPS> (assert (fase LecturaSintomas Pedro))
CLIPS> (run)
  Introduzca los síntomas de Pedro separados por espacios y
  ordenados alfabéticamente
CLIPS> (assert (Sintomas Pedro picor vesiculas))
CLIPS> (run)
  El diagnóstico para Pedro es: Eccema
```

Podemos evitar que el usuario tenga que introducir órdenes específicas de CLIPS:

```
; Fase inicial

(defrule ObtenerNombreUsuario
      =>
      (printout t "Introduzca su nombre" crlf)
      (bind ?Respuesta (read))
      (assert (fase LecturaSintomas ?Respuesta)))


; Fase de lectura de síntomas

(defrule LeerSintomas
      ?fase <- (fase LecturaSintomas ?Nombre)
      =>
      (printout t "Introduzca los síntomas de " ?Nombre " separados
      por espacios y ordenados alfabéticamente" crlf)
      (bind ?Respuesta (explode$ (readline)))
      (assert (Sintomas ?Nombre ?Respuesta))
      (retract ?fase)
)


; Fase de diagnóstico

(defrule DiagnosticoEccema
      (Sintomas ?Nombre $? picor $? vesiculas $?)
      =>
      (printout t "El diagnóstico para " ?Nombre " es: Eccema" crlf))
```

```
CLIPS> (reset)
CLIPS> (run)
  Introduzca su nombre
  Pedro

  Introduzca los síntomas de Pedro separados por espacios y
  ordenados alfabéticamente

  picor vesículas

  El diagnóstico para Pedro es: Eccema
```

El problema se complica cuando tenemos que manejar más fases y hay varias reglas en una misma fase. Por ejemplo, puede que tengamos reglas que nos dicen si una persona es o no adulta, si tiene o no fiebre… y deseemos que estas reglas se lancen antes que cualquier otra regla de diagnóstico o de terapia.

```
; Fase inicial

(defrule ObtenerNombreUsuario
      =>
      (printout t "Introduzca su nombre" crlf)
      (bind ?Respuesta (read))
      (assert (fase LecturaSintomas ?Respuesta)))


; Fase de lectura de síntomas

(defrule ObtenerTemperaturaUsuario
      (fase LecturaSintomas ?Nombre)
      =>
      (printout t "Introduzca temperatura" crlf)
      (bind ?Respuesta (read)))

(defrule ObtenerSintomasUsuario
      (fase LecturaSintomas ?Nombre)
      =>
      (printout t "Introduzca los síntomas de " ?Nombre " separados
      por espacios y ordenados alfabéticamente" crlf)
      (bind ?Respuesta (explode$ (readline)))
      (assert (Sintomas ?Nombre ?Respuesta)))

…

; Última regla en aplicarse

(defrule CambiarDeFase
      (declare (salience -10))
      ?fase <- (fase LecturaSintomas ?Nombre)
      =>
      (retract ?fase)
      (assert (fase Diagnostico ?Nombre)))
```

Lo mismo se haría con los conjuntos de reglas de diagnóstico, terapia, salida de resultados, etcétera.

Por ejemplo, en todas las reglas relativas al diagnóstico, tendré como patrón en el antecedente `(fase Diagnostico)`

```
(defrule PacienteConFiebre
      (fase Diagnostico ?N)
      (ExploracionPaciente
            (Nombre           ?N)
            (Temperatura      ?T))
      (test (> ?T 37))
      =>
      (assert (Paciente ?N TieneFiebre)))

(defrule PacienteSinFiebre
      (fase Diagnostico ?N)
      (ExploracionPaciente
            (Nombre           ?N)
            (Temperatura      ?T))
      (test (<= ?T 37))
      ?PF <- (Paciente ?N TieneFiebre)
      =>
      (retract ?PF))

(defrule DiagnosticoEccema
      (fase Diagnostico ?N)
      (Sintomas ?N $? picor $? vesiculas $?)
      =>
      (assert (PosibleDiagnostico ?N Eccema)))
…
```

En vez de usar una regla para cambiar de fase en cada una de ellas, podemos usar una única regla, que además nos permitirá volver a empezar con la primera fase una vez concluida la última:

```
(deffacts InformacionControl
       (fase Deteccion)
       (SecuenciaFases Diagnostico Terapia Deteccion))

(defrule CambiarDeFase
      (declare (Salience -10))
      ?fase <- (fase ?)
      ?ListaFases <- (SecuenciaDeFases  ?Siguiente $?Resto)
      =>
      (retract ?fase ?ListaFases)
      (assert (fase ?Siguiente))
      (assert (SecuenciaDeFases $?Resto ?Siguiente)))
```

Supongamos que estamos en una fase cualquiera, por ejemplo `Diagnóstico`. Cuando ya no existan reglas aplicables con `(fase Diagnostico)` en su antecedente, se activará la regla `CambiarDeFase` que ya estaba en la agenda a la espera de ejecutarse (la última debido a su baja prioridad).

El único problema que se puede presentar es que si ejecutamos `(run)`, podríamos entrar en un ciclo infinito, ya que procedería a cambiar ininterrumpidamente de fase.

Sobre el uso de prioridades en un sistema experto

Normalmente, no resulta recomendable utilizar más de 4 ó 5 prioridades diferentes (aunque CLIPS admita cualquier entero entre `-10.000` y `+10.000`), ya que un uso excesivo de prioridades conduce a una programación imperativa y no declarativa.

Normalmente, de mayor a menor prioridad, esta es la forma en que se agrupan las reglas de un sistema experto:

- **Restricciones**: Entrada de valores no permitidos y detección de estados que violan alguna restricción de nuestro problema.

- **Experto**: Las reglas usuales del sistema experto, que son de mayor prioridad que las reglas relacionadas con la realización de preguntas al usuario (para no ir preguntándole a éste si la respuesta puede obtenerse con las reglas del sistema).

- **Preguntas**: Lectura de datos proporcionados por el usuario.

- **Control**: Por último, las reglas con menor prioridad corresponden a las reglas que gobiernan las transiciones entre distintas fases.

Modularización con (defmodule)

`(defmodule)` se usa en CLIPS para dividir la base de conocimiento en distintos módulos. Cada módulo tendrá una agenda y una memoria de trabajo diferente, por lo que nos servirán para controlar las distintas fases de un sistema experto.

```
(defmodule  <Nombre del Módulo>
      (export  <Construcciones que exporta>)
      (import  <Construcciones que importa>)
      …
)
```

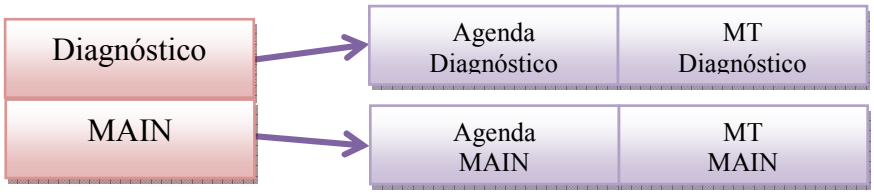
Todo lo que escribamos a partir de `(defmodule MODULO …)` pertenece a dicho módulo y, en principio, no es accesible desde otros módulos.

```
(defmodule Deteccion)
      (defrule TomarDatos …)
      …
)

(defmodule Diagnostico
      (defrule Eccema …)
      …
)

(defmodule Terapia
      …
)
```

CLIPS utiliza una pila de módulos para gestionar el módulo que está activo en cada momento, a la que denomina "*focus stack*" (de forma similar a como un lenguaje de programación imperativo utiliza una pila para gestionar las llamadas a subprogramas):



Cuando usamos `(run)`, se ejecutan las reglas del último módulo de la lista (`Diagnóstico` en la figura) y, cuando su agenda se quede vacía, automáticamente se borra el módulo `Diagnóstico` del "*focus stack*" y se carga el siguiente de la lista para proseguir la ejecución de las reglas que aparezcan en la agenda correspondiente.

En cualquier momento, podemos añadir un nuevo módulo a la lista, que pasará a ser el módulo activo, con

```
(focus <Nombre Módulo>)
```

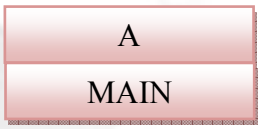
Esto lo podemos hacer desde línea de comandos de CLIPS o desde alguna RHS.

Al principio, la *"focus stack"* sólo tendrá MAIN, que siempre será el último módulo de la lista (como la función main de un programa en C). Cuando se ejecute `(focus Diagnóstico)` obtendremos una situación como la de la figura de arriba.

```
ejemplo.modulos.clp
(defmodule A)
…
(defmodule B)
…
```

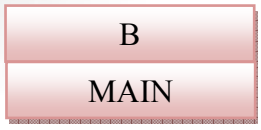
```
CLIPS> (clear)
CLIPS> (load ejemplo.modulos.clp)
CLIPS> (reset)
```


```
CLIPS> (focus A)
```




```
CLIPS> (run)
```


```
CLIPS> (focus B)
```




```
CLIPS> (run)
```

EJEMPLO

```
(defmodule M1)
      (deffacts HechosM1
            (A))
      (defrule R1
            (A)
            =>
            (assert (B)))

(defmodule M2)
      (deffacts HechosM2
            (C))
      (defrule R2
            (C)
            =>
            (assert (D)))



CLIPS> (reset)          ; Añade HechosM1 a la M.T. del módulo M1
                        ; Añade HechosM2 a la M.T. del módulo M2

CLIPS> (focus M1)       ; Se añade M1 a la 'focus stack'

CLIPS> (run)            ; Se ejecuta R1 y se añade B a la M.T. de M1

                        ; Como la agenda de M1 queda vacía,
                        ; M1 desaparece de la pila (sólo queda MAIN)

CLIPS> (focus M2)       ; Se añade M2 a la 'focus stack'

CLIPS> (run)            ; Se ejecuta R2 y se añade D a la M.T. de M2

                        ; Como la agenda de M2 queda vacía,
                        ; M2 desaparece de la pila (sólo queda MAIN)
```

EJEMPLO

```
(defmodule M1)
      (deffacts HechosM1
            (A))
      (defrule R1
            (A)
            =>
            (assert (B))
            (focus M2))

(defmodule M2)
      (deffacts HechosM2
            (C))
      (defrule R2
            (C)
            =>
            (assert (D)))
```

```
CLIPS> (reset)          ; Añade HechosM1 a la M.T. del módulo M1
                        ; Añade HechosM2 a la M.T. del módulo M2

CLIPS> (focus M1)       ; Se añade M1 a la 'focus stack'

CLIPS> (run)            ; Se ejecuta R1 y se añade B a la M.T. de M1

                        ; Se ejecuta (focus M2),
                        ; por lo que se añade M2 a la 'focus stack'

                        ; Se ejecuta R2 y se añade D a la M.T. de M2

                        ; Como la agenda de M2 queda vacía,
                        ; M2 desaparece de la pila (quedan M1 y MAIN)

                        ; Como la agenda de M1 está vacía,
                        ; M1 desaparece de la pila (sólo queda MAIN)
CLIPS>
```

Importar y exportar construcciones

Cada módulo tiene asociados hechos y reglas que son invisibles para otros módulos a no ser que se exporten. Para hacerlo, hemos de tener en cuenta que:

- No se pueden exportar hechos aislados (se exporta un `deftemplate` determinado y, automáticamente, se está exportando la definición del template junto con los hechos correspondientes).

- Un módulo no dice a quién va a exportar, pero el que importa sí debe decir de dónde lo va a hacer.

```
(defmodule MDeteccion
      (export deftemplate FichaPaciente))

(deftemplate FichaPaciente
      (field Paciente)
      (field Tipo))

(deffacts Hechos
      (FichaPaciente (Paciente Juan)
                     (Tipo     Interno)))

(defmodule MDiagnostico
      (import MDeteccion deftemplate FichaPaciente))

(defrule Regla
      (FichaPaciente (Tipo Interno)
                     (Paciente ?Nombre))
      …
```

`MDetección` exporta `FichaPaciente` y `MDiagnostico` la importa.

> NOTA: Si quisiéramos exportar los vectores ordenados de características que estén definidos dentro de un `deffacts`, debemos obligatoriamente exportar e importar TODOS los `deftemplates` definidos en el módulo.

Instrucciones de importación y exportación:

```
(export ?ALL)                       exporta todo.
(export ?NONE)                      no exporta nada (por defecto).
(export deftemplate ?ALL)           exporta todos los deftemplates y vectores.
(export deftemplate ?NONE)          no exporta ningún deftemplate.
(export deftemplate <nombre>+)      sólo exporta los deftemplates especificados.

(import <módulo> ?ALL)
(import <módulo> ?NONE)
(import <módulo> deftemplate ?ALL)
(import <módulo> deftemplate ?NONE)
(import <módulo> deftemplate <nombre>+)
```

Se pueden usar los hechos (definidos en un `deffacts`) de un módulo que los exporta, sin que tenga que pasar el foco por él.

```
(defmodule A (export ?ALL))

(deftemplate Registro (field F))

(deffacts Hechos
      (vector 1)
      (Registro (F 1)))


(defmodule B (import A ?ALL))

(defrule R
      (vector ?variable)
      (Registro (F 1))
      =>
      (printout t ?variable))
```

Obviamente, si los hechos no están definidos dentro de un `deffacts`, sino que son asertados en la RHS de alguna regla del módulo `A`, entonces el foco debe pasar antes por `A` (y la regla debe ejecutarse para que el hecho exista en la memoria de trabajo).

NOTA: No se pueden usar en el MAIN hechos asertados desde otros módulos, ni al revés (en los módulos no se conocen ni los hechos ni los deftemplates definidos en MAIN).

Control de fases mediante módulos

```
(deffacts InformacionDeControl
        (ListaFases MDeteccion MDiagnostico MTerapia))

(defrule CambiarFase
      ?Lista <- (ListaFases ?Siguiente $?Resto)
      =>
      (focus ?Siguiente)
      (retract ?Lista)
      (assert (ListaFases $?Resto ?Siguiente)))


(defmodule MDeteccion)
…

(defmodule MDiagnostico)
…

(defmodule MTerapia)
…
```

Obsérvese que ya no introducimos prioridades (`-10`) en la regla `CambiarFase`, ya que ahora no resultan necesarias.

Sin embargo, hay un problema si no existen reglas aplicables en ningún módulo, ya que al ejecutar `(run)` la regla `CambiarFase` se activa y ejecuta indefinidamente.

POSIBLE SOLUCIÓN: Incluir otro patrón en la regla de cambio de fase.

Para el ejemplo anterior, es lógico que si hemos resuelto el problema de un paciente y ya no hay más pacientes, entonces no se cambia de fase.

Su suponemos que el usuario introducirá algo del tipo:

```
CLIPS> (assert (BuscarTerapia Juan))
```

Entonces, la regla adecuada para cambiar de fase será de la forma:

```
(defrule CambiarFase
      (exists (BuscarTerapia ?))
      ?Lista <- (ListaFases ?Siguiente $?Resto)
      =>
      (focus ?Siguiente)
      (retract ?Lista)
      (assert (ListaFases $?Resto ?Siguiente)))
```

Ficheros

```
(save <nombre de fichero .clp>)
```

Graba en un fichero de texto las definiciones de `deftemplate`, `deffacts`, `defrule`… que en ese momento existan en la memoria de trabajo (pero no aserta los hechos que no estén englobados dentro de un `deffacts`).

```
(open <file-name> <logical-name> [<mode>])

<mode> ::= "r" | "w" | "r+" | "a" | "wb"
```

Permite acceder a un fichero, indicando el modo de acceso como en C:

`"r"`	(lectura)
`"w"`	(escritura)
`"r+"`	(ambos)
`"a"`	(añadir)
`"wb"`	(escritura en binario)

EJEMPLO

```
(open "fichero.txt" datos "w")
(printout datos "Texto al fichero")
(printout datos 7)
(close datos)
```

Como en cualquier lenguaje de programación, conviene que nos acordemos de cerrar el fichero, que en CLIPS se hace con

```
(close [<logical-name>])
```

Para escribir en un fichero, usaremos:

```
(printout <logical-name> <expresion>*)
(format <logical-name> <string-expression> <expression>*)
```

Si lo que queremos es leer datos, podemos usar:

```
(read [<logical-name>])
(readline [<logical-name>])
(read-number [<logical-name>])
(get-char [<logical-name>])
```

Apéndice: Atributos de los campos

Type

Se pueden especificar los tipos de datos admitidos para cada campo de un template:

```
(deftemplate Persona
      (field Nombre (type SYMBOL))
      (field Edad (type INTEGER))
      (field Salario (type NUMBER))
      (field Ciudad (type LEXEME))
)
```

NOTA:

```
(field Salario (type INTEGER FLOAT))
```
equivale a
```
(field Salario (type NUMBER))
```

```
(field Ciudad (type SYMBOL STRING))
```
equivale a
```
(field Ciudad (type LEXEME))
```

Un campo de tipo `STRING` no admite valores como `Carlos` o `NO` pero sí `"Carlos"` (justo lo contrario que el tipo `SYMBOL`). Ninguno de los dos acepta valores numéricos.

Cuando no se especifica un tipo de dato, se asume que puede ser cualquiera (la combinación de `NUMBER` y `LEXEME`), por lo que se admiten `Carlos`, `"Carlos"` y `4`.

Mucho cuidado con los tipos de datos y los emparejamientos:

```
(deftemplate Exploracion
      (field Nombre (type STRING))
      (field Sintomas))

(deftemplate Persona
      (field Nombre))

(defrule Invalida
      (Persona (Nombre ?N))
      (Exploracion (Nombre ?N))    ; Tipos incompatibles
      =>
      …
)
```

Allowed Values

allowed<tipo> <valores>

```
(deftemplate Persona
      (field Sexo (type SYMBOL)
                  (allowedsymbols V M))
)
```

NOTA: En el caso de `LEXEME`, habría que poner `allowed-lexemes`

Si suprimimos la especificación de tipo en este ejemplo `(type SYMBOL)`, entonces se aceptaría cualquier número, cualquier string, pero sólo los símbolos `V` y `M`.

Range

```
(deftemplate Persona
      (field Edad (range 0 ?VARIABLE))
)
```

`; ?VARIABLE` es una palabra reservada.

Con `range` hemos especificado que se acepta cualquier valor mayor o igual que 0.

Con `range` no se restringe el tipo. El campo `Edad`, tal como lo hemos definido, aceptaría símbolos y cadenas, pero sólo enteros mayores o iguales que 0. Para que sólo se admitan enteros, debemos especificar lo siguiente:

```
(deftemplate Persona
      (field Edad (type INTEGER)
                  (range 0 ?VARIABLE))
)
```

Cardinality

Restringe el número de valores que puede incluir un `multifield`:

```
(deftemplate EquipoBaloncesto
      (multifield JugadoresPista
            (type STRING)
            (cardinality 5 5))
      (multifield JugadoresBanquillo
            (type STRING)
            (cardinality 0 7))
)
```

Default

Establece un valor por defecto que CLIPS asignará al correspondiente campo cuando nosotros no le asignemos un valor:

```
(deftemplate Persona
      (field Sexo (type SYMBOL)
                  (allowedvalues V M)
                  (default M))
)
```

Si queremos que CLIPS seleccione automáticamente un valor que verifique las restricciones impuestas por los atributos del campo, pondremos:

```
      (default ?DERIVE)
```

Este es el comportamiento por defecto de CLIPS

Sin embargo, si tenemos:

```
      (field Edad (type INTEGER))
```

CLIPS asignará por defecto el valor 0
(no se le asigna `nil` porque `nil` no es un número).

Por último, si queremos que, obligatoriamente, siempre haya que indicar el valor de un campo, pondremos:

```
      (default ?NONE)
```