



# Reglas

El constructor *defrule*  
y otros comandos relacionados



# Reglas

- **Introducción**
- Definición de reglas
- Ciclo básico de ejecución de reglas
- Sintaxis del antecedente
- Propiedades de una regla
- Comandos



# Introducción

- Partes de la regla:
  - Antecedente (condiciones)
  - Consecuente (acciones)
- Semántica:
  - Si el **antecedente** es cierto según los **hechos** almacenados en la lista de hechos, entonces **pueden** realizarse las **acciones** especificadas en el **consecuente**



# Introducción

- **Base de conocimiento.** Conjunto de reglas que describen el problema a resolver.
- **Activación o disparo de reglas.** Entidad patrón: Hechos ordenados o plantillas, e instancias de clases.
- **Motor de inferencia.** Comprueba antecedente de las reglas y aplica el consecuente.



# Reglas

- Introducción
- Definición de reglas
- Ciclo básico de ejecución de reglas
- Sintaxis del antecedente
- Propiedades de una regla
- Comandos



# Definición de reglas

## Sintaxis del constructor *defrule*

```

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



# Definición de reglas

## Ejemplo

```
(defrule FrigorificoMal
```

```
  "Qué pasa si usamos mal el frigorífico"
```

```
  (frigorifico luz encendida)
```

```
  (frigorifico puerta abierta)
```

```
=>
```

```
  (assert (frigorifico comida estropeada)))
```



# Definición de reglas

## Consideraciones

- Una regla con el mismo nombre que otra, aun siendo errónea, machaca a la anterior
- No hay límite en el número de elementos condicionales y acciones de una regla
- Puede no haber ningún elemento condicional en el antecedente y se usa automáticamente (*initial-fact*) como elemento condicional
- Puede no haber ninguna acción en el consecuente, y la ejecución de la regla no tiene ninguna consecuencia
- El antecedente es de tipo conjuntivo
- Defrule Manager muestra la base de conocimiento





# Definición de reglas

## HolaMundo 1 y 2

```

(defrule HolaMundo1
=>
  (printout t "Hola Mundo" crlf) )
  
```

```

(defrule HolaMundo2
(initial-fact)
=>
  (printout t "Hola Mundo" crlf))
  
```



# Reglas

- Introducción
- Definición de reglas
- Ciclo básico de ejecución de reglas
- Sintaxis del antecedente
- Propiedades de una regla
- Comandos



# Ciclo básico de ejecución de reglas: Conceptos

- Una regla se activa cuando se satisface el antecedente
- Una regla puede activarse para distintos conjuntos de hechos (instancias de una regla)
- Activación: Se indica por el Nombre de la regla e Indices de los hechos que la satisfacen
- Una regla se dispara cuando el motor de inferencia decide ejecutar las acciones de su consecuente
- Las instancias de regla se almacenan en la agenda
- Las instancias de regla tienen asignada una prioridad entre -10000 y 10000 (0 por defecto)
- La estrategia de resolución de conflictos decide qué regla se dispara si hay varias con la misma prioridad



# Ciclo básico de ejecución de reglas

1. Las reglas se ejecutan con el comando  
(`run [<máximo>]`)
2. Si se ha alcanzado el máximo de disparos, se para la ejecución
3. Se actualiza la agenda según la lista de hechos
4. Se selecciona la instancia de regla a ejecutar de acuerdo a prioridades y estrategia de resolución de conflictos
5. Se dispara la instancia seleccionada, se incrementa número disparos y se elimina de la agenda
6. Volver al paso 2



# Ciclo básico de ejecución de reglas

Fichero con definición de reglas y hechos.

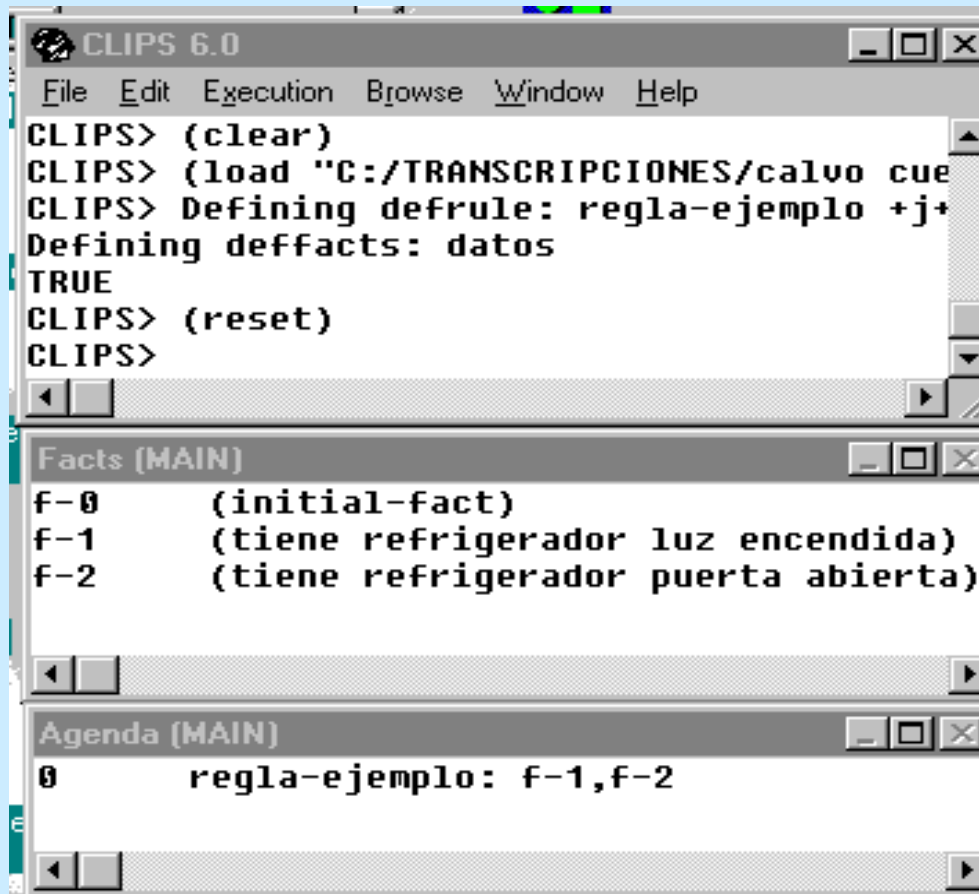
```
eje3.clp - Bloc de notas
Archivo Edición Buscar Ayuda

(defrule regla-ejemplo "Ejemplo simple"
  (tiene refrigerador luz encendida)
  (tiene refrigerador puerta abierta)
  =>
  (assert (tiene refrigerador comida estropeada)))

(deffacts datos
  (tiene refrigerador luz encendida)
  (tiene refrigerador puerta abierta))
```

# Ciclo básico de ejecución de reglas

Entorno CLIPS tras inicializar



```
CLIPS 6.0
File Edit Execution Browse Window Help
CLIPS> (clear)
CLIPS> (load "C:/TRANSCRIPCIONES/calvo cue
CLIPS> Defining defrule: regla-ejemplo +j+
Defining deffacts: datos
TRUE
CLIPS> (reset)
CLIPS>

Facts (MAIN)
f-0      (initial-fact)
f-1      (tiene refrigerador luz encendida)
f-2      (tiene refrigerador puerta abierta)

Agenda (MAIN)
0        regla-ejemplo: f-1,f-2
```

# Ciclo básico de ejecución de reglas

Entorno CLIPS tras ejecutar

```

CLIPS 6.0
File Edit Execution Browse Window Help
CLIPS> (clear)
CLIPS> (load "C:/TRANSCRIPCIONES/calvo cuenca/I
CLIPS> Defining defrule: regla-ejemplo +j+j
Defining deffacts: datos
TRUE
CLIPS> (reset)
CLIPS> (run)
CLIPS>

Facts (MAIN)
f-0      (initial-fact)
f-1      (tiene refrigerador luz encendida)
f-2      (tiene refrigerador puerta abierta)
f-3      (tiene refrigerador comida estropeada)

Agenda (MAIN)

```



# Reglas

- Introducción
- Definición de reglas
- Ciclo básico de ejecución de reglas
- Sintaxis del antecedente
- Propiedades de una regla
- Comandos





# Sintaxis del antecedente

- El antecedente se compone de una serie de Elementos Condicionales (EC).
- Hay 8 tipos de EC:
  - EC patrón
  - EC test
  - EC and
  - EC or
  - EC not
  - EC exists
  - EC forall
  - EC logical



# EC patrón

- Consiste en un conjunto de restricciones
- Se usan para verificar si se cumple un campo o slot de una entidad patrón
- Hay varios tipos de restricciones
  - Literales
  - Comodines
  - Variables
  - Conectivas
  - Predicados
  - Valores devueltos
  - Direcciones de hechos



# EC patrón

## Restricciones

- Restricción para hechos ordenados  
(`<restricción-1> ... <restricción-n>`)
- Restricción para hechos no ordenados  
(`<nombre-deftemplate>`  
(`<nombre-casilla-1> <restricción-1>`)  
.  
.  
.  
(`<nombre-casilla-n> <restricción-n>`)



# EC patrón

## Restricciones literales

- Restricciones más básicas
- Define el valor exacto del campo, sin comodines, ni variables
- Contienen sólo constantes

`<restricción> ::= <constante>`



# EC patrón

## Restricciones literales

```

(deffacts dato-hechos
  (dato 1.0 azul "rojo")
  (dato 1 azul)
  (dato 1 azul rojo)
  (dato 1 azul ROJO)
  (dato 1 azul rojo 6.9))

(deftemplate persona
  (slot nombre)
  (slot edad)
  (multislot amigos))

(deffacts gente
  (persona (nombre Juan) (edad 20))
  (persona (nombre Juan) (edad 20))
  (persona (nombre Juan) (edad 34))
  (persona (nombre Ana) (edad 34))
  (persona (nombre Ana) (edad 20)))
  
```

# EC patrón

## Restricciones literales

```
CLIPS> (defrule encontrar-datos (datos 1 azul
  rojo) => )
```

```
CLIPS> (reset)
```

```
CLIPS> (facts)
```

```
f-0      (initial-fact)
```

```
f-1      (datos 1.0 azul "rojo")
```

```
f-2      (datos 1 azul)
```

```
f-3      (datos 1 azul rojo)
```

```
f-4      (datos 1 azul ROJO)
```

```
f-5      (datos 1 azul rojo 6.9)
```

For a total of 6 facts.

```
CLIPS> (agenda)
```

```
0        encontrar-datos: f-3
```

For a total of 1 activation.



# EC patrón

## Restricciones literales

```
CLIPS> (defrule encontrar-juan (persona (nombre juan)
    (edad 20)) => )
```

```
CLIPS> (defrule encontrar-ana (persona (edad 34)
    (nombre ana)) => )
```

```
CLIPS> (reset)
```

```
CLIPS> (facts)
```

```
f-0      (initial-fact)
```

```
f-1      (persona (nombre pepe) (edad 20) (amigos))
```

```
f-2      (persona (nombre juan) (edad 20) (amigos))
```

```
f-3      (persona (nombre pepe) (edad 34) (amigos))
```

```
f-4      (persona (nombre ana) (edad 34) (amigos))
```

```
f-5      (persona (nombre ana) (edad 20) (amigos))
```

For a total of 6 facts.

```
CLIPS> (agenda)
```

```
0        encontrar-ana: f-4
```

```
0        encontrar-juan: f-2
```

For a total of 2 activations.



# EC patrón

## Restricciones con comodines

- Indican que cualquier valor en esa posición de la entidad patrón es válido para emparejar con la regla.
- Tipos:
  - Comodín monocampo: ?  
(empareja con 1 campo)
  - Comodín multicampo: \$?  
(empareja con 0 o más campos)

`<restricción> ::= <constante> | ? | $?`





# EC patrón

## Restricciones con comodines

```
CLIPS> (defrule encontrar-datos (datos ? azul
  rojo $?) => )
```

```
CLIPS> (facts)
```

```
f-0      (initial-fact)
```

```
f-1      (datos 1.0 azul "rojo")
```

```
f-2      (datos 1 azul)
```

```
f-3      (datos 1 azul rojo)
```

```
f-4      (datos 1 azul ROJO)
```

```
f-5      (datos 1 azul rojo 6.9)
```

For a total of 6 facts.

```
CLIPS> (agenda)
```

```
0        encontrar-datos: f-5
```

```
0        encontrar-datos: f-3
```

For a total of 2 activations.



# EC patrón

## Restricciones con comodines

- Las restricciones multicampo y literal se pueden combinar para especificar restricciones complejas.

```
(dato $? AMARILLO $?)
```

Emparejaría con:

```
(dato AMARILLO)
```

```
(dato AMARILLO rojo azul)
```

```
(dato rojo AMARILLO azul)
```

```
(dato rojo azul AMARILLO)
```



# EC patrón

## Restricciones con variables

- Almacena el valor de un campo para después utilizarlo en otros elementos condicionales o consecuente de la regla.
- Tipos:
  - Variable monocampo: ?<nombre-variable>
  - Variable multicampo: \$<nombre-variable>

```
<restricción> ::=      <constante> | ? | $? |
                        <variable-monocampo> |
                        <variable-multicampo>
```

```
<variable-monocampo> ::= ?<nombre-variable>
```

```
<variable-multicampo> ::= $<nombre-variable>
```



# EC patrón

## Restricciones con variables

- Cuando la variable aparece por 1<sup>o</sup> vez, actúa como un comodín, pero el valor queda ligado al valor del campo.
- Si vuelve a aparecer la variable, ahora debe de coincidir con el valor ligado.
- La ligadura sólo se cumple dentro del alcance de la regla.



# EC patrón

## Restricciones con variables

```

CLIPS> (defrule encontrar-datos-triples
        (datos ?x ?y ?z)
        =>
        (printout t ?x " : " ?y " : " ?z crlf))
  
```

```

CLIPS> (facts)
f-0      (initial-fact)
f-1      (datos 2 azul verde)
f-2      (datos 1 azul)
f-3      (datos 1 azul rojo)
  
```

For a total of 4 facts.

```

CLIPS> (run)
1 : azul : rojo
2 : azul : verde
  
```

# EC patrón

## Restricciones con variables

```

eje9.clp - Bloc de notas
Archivo Edición Buscar Ayuda

(defacts datos
  (dato 1 azul)
  (dato 1 azul rojo)
  (dato 1 azul rojo 6.9))

(defrule encuentra-dato-1
  (dato ?x $?y ?z)
  =>|
  (printout t "?x = " ?x crlf
            "?y = " ?y crlf
            "?z = " ?z crlf
            "-----" crlf))
  
```

# EC patrón

## Restricciones con variables

```

CLIPS 6.0
File Edit Execution Browse Window Help
Redefining defrule: encuentra-dato-1 +j
TRUE
CLIPS> (reset)
CLIPS> (run)
?x = 1
?y = (azul rojo)
?z = 6.9
-----
?x = 1
?y = (azul)
?z = rojo
-----
?x = 1
?y = ()
?z = azul
-----
CLIPS>

```

Agenda (MAIN)

Facts (MAIN)

```

f-0      (initial-fact)
f-1      (dato 1 azul)
f-2      (dato 1 azul rojo)
f-3      (dato 1 azul rojo 6.9)

```



# EC patrón

## Restricciones conectivas

- Permiten unir restricciones y variables
- Utilizan los conectores lógicos & (and), | (or), ~ (not)

`<restricción> ::= ? | $? | <restricción-conectiva>`

`<restricción-conectiva>`

`::= <restricción-simple> |`

`<restricción-simple> & <restricción-conectiva> |`

`<restricción-simple> | <restricción-conectiva>`

`<restricción-simple> ::= <término> | ~<término>`

`<término> ::= <constante> |`

`<variable-monocampo> |`

`<variable-multicampo>`





# EC patrón

## Restricciones conectivas: Precedencia

- Orden de precedencia:  $\sim$ ,  $\&$ ,  $|$
- Excepción: si la primera restricción es una variable seguida de la conectiva  $\&$ , la primera restricción (la variable) se trata como una restricción aparte

`?x&rojo|azul`

equivale a

`?x&(rojo|azul)`



# EC patrón

## Restricciones conectivas

```
CLIPS> (deftemplate dato-B (slot valor))
```

```
CLIPS> (defacts AB
```

```
  (dato-A verde)
```

```
  (dato-A azul)
```

```
  (dato-B (valor rojo))
```

```
  (dato-B (valor azul)))
```

```
CLIPS> (defrule ejemplo1-1 (datos-A ~azul) => )
```

```
CLIPS> (defrule ejemplo1-2 (datos-B (valor  
  ~rojo&~verde)) => )
```

```
CLIPS> (defrule ejemplo1-3 (datos-B (valor verde|rojo))  
=> )
```

# EC patrón

## Restricciones conectivas

```

CLIPS 6.0
File Edit Execution Browse Window Help
CLIPS> Defining deftemplate: dato-B
Defining deffacts: AB
Defining defrule: Ejemplo1-1 +j
Defining defrule: Ejemplo1-2 +j
Defining defrule: Ejemplo1-3 +j
TRUE
CLIPS> (reset)
CLIPS>

```

Agenda (MAIN)		Facts (MAIN)	
0	Ejemplo1-2: f-4	f-0	(initial-fact)
0	Ejemplo1-3: f-3	f-1	(dato-A verde)
0	Ejemplo1-1: f-1	f-2	(dato-A azul)
		f-3	(dato-B (valor rojo))
		f-4	(dato-B (valor azul))



# EC patrón

## Restricciones predicado

- Se restringe un campo según el valor de verdad de una expresión lógica
- Se indica mediante dos puntos (:) seguidos de una llamada a una función predicado
- La restricción se satisface si la función devuelve un valor no FALSE
- Normalmente se usan junto a una restricción conectiva y a una variable

```

<término> ::= <constante> |
              <variable-monocampo> |
              <variable-multicampo> |
              :<llamada-a-función->
    
```



# EC patrón

## Restricciones predicado

```
CLIPS> (defrule predicado1 (datos ?x&:(numberp ?x)) => )
```

```
CLIPS> (assert (datos 1) (datos 2) (datos rojo))
```

```
<Fact-2>
```

```
CLIPS> (facts)
```

```
f-0      (datos 1)
```

```
f-1      (datos 2)
```

```
f-2      (datos rojo)
```

For a total of 3 facts.

```
CLIPS> (agenda)
```

```
0        predicado1: f-1
```

```
0        predicado1: f-0
```

For a total of 2 activations.



# EC patrón

## Restricciones predicado

- CLIPS proporciona funciones predicado:
  - (evenp <arg>)
  - (floatp <arg>)
  - (integerp <arg>)
  - (numberp <arg>)
  - (oddp <arg>)
  - (stringp <arg>)
  - (symbolp <arg>)



# EC patrón

## Restricciones predicado

- Y funciones de comparación:
  - (eq <expression> <expression>+)
  - (neq <expression> <expression>+)
  - (= <numeric-expression> <numeric-expression>+)
  - (<> <numeric-expression> <numeric-expression>+)
  - (> <numeric-expression> <numeric-expression>+)
  - (< <numeric-expression> <numeric-expression>+)
  - (>= <numeric-expression> <numeric-expression>+)
  - (<= <numeric-expression> <numeric-expression>+)



# EC patrón

## Restricciones de valor devuelto

- Se usa el valor devuelto por una función para restringir un campo
- El valor devuelto debe ser de uno de los tipos primitivos de datos y se sitúa en el patrón como si se tratase de una restricción literal en las comparaciones
- Se indica mediante el carácter `='`

```

<término> ::= <constante> |
              <variable-monocampo> |
              <variable-multicampo> |
              :<llamada-a-función->
              =<llamada-a-función->
    
```



# EC patrón

## Restricciones de valor devuelto

```

eje19.clp - Bloc de notas
Archivo  Edición  Buscar  Ayuda

(deftemplate dato (slot x) (slot y))

(defrule doble
(dato (x ?x) (y =(* 2 ?x)))
=>)

kdeffacts datos
(dato (x 2) (y 4))
(dato (x 3) (y 9)))
  
```



# EC patrón

## Restricciones de valor devuelto

```
CLIPS> (defrule doble
        (datos (x ?x) (y = (* 2 ?x))) => )
```

```
CLIPS> (facts)
```

```
f-0      (datos (x 2) (y 4))
```

```
f-1      (datos (x 3) (y 9))
```

For a total of 2 facts.

```
CLIPS> (agenda)
```

```
0        doble: f-0
```

For a total of 1 activation.



# EC patrón

## Captura de direcciones de hechos

- A veces se desea realizar modificaciones, duplicaciones o eliminaciones de hechos en el consecuente de una regla
- Para ello es necesario que en la regla se obtenga el índice del hecho sobre el que se desea actuar

```
<EC-patrón-asignado> ::=
```

```
?<nombre-variable> <- <EC-patrón>
```



# EC patrón

## Captura de direcciones de hechos

```
CLIPS> (deffacts hechos (dato 1) (dato 2))
```

```
CLIPS> (reset)
```

```
CLIPS> (facts)
```

```
f-0      (dato 1)
```

```
f-1      (dato 2)
```

For a total of 2 facts.

```
CLIPS> (defrule borra1
        ?hecho <- (dato 1)
        =>
        (retract ?hecho))
```

```
CLIPS> (run)
```

```
CLIPS> (facts)
```

```
f-1      (dato 2)
```

For a total of 1 facts.



# EC test

- El EC test comprueba el valor devuelto por una función
- El EC test se satisface si la función devuelve un valor que no sea FALSE
- El EC test no se satisface si la función devuelve un valor FALSE

`<EC-test> ::= (test <llamada-a-función>)`



# EC test

```

CLIPS> (defrule diferencia
        (dato ?x)
        (valor ?y)
        (test (>= (abs (- ?x ?y)) 3))
        => )
  
```

```

CLIPS> (assert (dato 6) (valor 9))
  
```

```

<Fact-1>
  
```

```

CLIPS> (facts)
  
```

```

f-0      (dato 6)
  
```

```

f-1      (valor 9)
  
```

For a total of 2 facts.

```

CLIPS> (agenda)
  
```

```

0        diferencia: f-0, f-1
  
```

For a total of 1 activation.



# EC or

- El EC or se satisface si se satisface cualquiera de los EC que lo componen
- Si se satisfacen varios ECs dentro del EC or, entonces la regla se disparará varias veces

`<EC-or> ::= (or <elemento-condicional>+)`

# EC or

```

CLIPS> (defrule posibles-desayunos
        (tengo pan)
        (or (tengo mantequilla)
            (tengo aceite))
        =>
        (assert (desayuno tostadas)))
CLIPS> (assert (tengo pan) (tengo mantequilla)
        (tengo aceite))
<Fact-2>
CLIPS> (agenda)
0      posibles-desayunos: f-0,f-2
0      posibles-desayunos: f-0,f-1
For a total of 2 activations.
  
```





# EC and

- El EC and se satisface si se satisfacen todos los EC que lo componen
- EL EC and permite mezclar ECs and y or en el antecedente

`<EC-and> ::= (and <elemento-condicional>+)`



# EC and

```

(defrule posibles-desayunos
  (tengo zumo-natural)
  (or (and (tengo pan)
           (tengo aceite))
      (and (tengo leche)
           (tengo cereales))))
=>
(assert (desayuno sano))
  
```



# EC not

- El EC not se satisface si no se satisface el EC que contiene
- Sólo puede negarse un EC

`<EC-not> ::= (not <elemento-condicional>)`



# EC not

```
(defrule Homer-loco
  (not (hay tele))
  (not (hay cerveza)))
=>
(assert (Homer pierde la cabeza)))
```



# EC exists

- Permite comprobar si una serie de ECs se satisface por algún conjunto de hechos

`<EC-exists> ::= (exists <elemento-condicional>+)`



# EC exists

```
CLIPS> (defrule dia-salvado
  (objetivo salvar-el-dia)
  (heroe (estado desocupado)))
=>
  (printout t "El día está salvado" crlf))
CLIPS> (facts)
f-0      (initial-fact)
f-1      (objetivo salvar-el-dia)
f-2      (heroe (nombre spider-man) (estado desocupado))
f-3      (heroe (nombre daredevil) (estado desocupado))
f-4      (heroe (nombre iron-man) (estado desocupado))
For a total of 5 facts.
CLIPS> (agenda)
0        dia-salvado: f-1,f-4
0        dia-salvado: f-1,f-3
0        dia-salvado: f-1,f-2
For a total of 3 activations.
```



# EC exists (continúa)

```

CLIPS> (defrule dia-salvado
  (objetivo salvar-el-dia)
  (exists (heroe (estado desocupado)))
  =>
  (printout t "El día está salvado" crlf))
CLIPS> (facts)
f-0      (initial-fact)
f-1      (objetivo salvar-el-dia)
f-2      (heroe (nombre spider-man) (estado desocupado))
f-3      (heroe (nombre daredevil) (estado desocupado))
f-4      (heroe (nombre iron-man) (estado desocupado))
For a total of 5 facts.
CLIPS> (agenda)
0        dia-salvado: f-1,
For a total of 1 activation.
  
```



# EC forall

- Permite comprobar si un conjunto de EC se satisface para toda ocurrencia de otro EC especificado

```
<EC-forall> ::= (forall <elemento condicional>  
                <elemento-condicional>+)
```

- Se satisface si, para toda ocurrencia del primer EC, se satisfacen los demás ECs





# EC forall

```
CLIPS> (defrule todos-limpios
  (forall (estudiante ?nombre)
    (lengua ?nombre)
    (matematicas ?nombre)
    (historia ?nombre)) => )
```

```
CLIPS> (reset)
```

```
CLIPS> (agenda)
```

```
0      todos-limpios: f-0,
```

```
For a total of 1 activation.
```

```
CLIPS> (assert (estudiante pepe) (lengua pepe)
  (matematicas pepe))
```

```
<Fact-3>
```

```
CLIPS> (agenda)
```

```
CLIPS> (assert (historia pepe))
```

```
<Fact-4>
```

```
CLIPS> (agenda)
```

```
0      todos-limpios: f-0,
```

```
For a total of 1 activation.
```



# EC logical

- Asegura el *mantenimiento de verdad* para hechos creados mediante reglas que usan EC logical
- Los hechos del antecedente proporcionan soporte lógico a los hechos creados en el consecuente
- Un hecho puede recibir soporte lógico de varios conjuntos distintos de hechos
- Un hecho permanece mientras permanezca alguno de los que lo soportan lógicamente
- Los ECs incluidos en un EC logical están unidos por un and implícito



# EC logical

- Puede combinarse con ECs and, or y not
- Sólo los primeros ECs del antecedente pueden ser de tipo logical

# EC logical

```

CLIPS> (defrule puedo-pasar
  (semaforo verde)
  =>
  (assert (puedo pasar)))
CLIPS> (assert (semaforo verde))
<Fact-0>
CLIPS> (run)
CLIPS> (facts)
f-0      (semaforo verde)
f-1      (puedo pasar)
For a total of 2 facts.
CLIPS> (retract 0)
CLIPS> (facts)
f-1      (puedo pasar)
For a total of 1 fact.
  
```



# EC logical (continúa)

```

CLIPS> (defrule puedo-pasar
  (logical (semaforo verde))
  =>
  (assert (puedo pasar)))
CLIPS> (assert (semaforo verde))
<Fact-0>
CLIPS> (run)
CLIPS> (facts)
f-0      (semaforo verde)
f-1      (puedo pasar)
For a total of 2 facts.
CLIPS> (retract 0)
CLIPS> (facts)
  
```



# Reglas

- Introducción
- Definición de reglas
- Ciclo básico de ejecución de reglas
- Sintaxis del antecedente
- Propiedades de una regla
- Comandos



# Propiedades de una regla

- La declaración de propiedades se incluye tras el comentario y antes del antecedente
- Se indica mediante la palabra reservada *declare*
- Una regla puede tener una única sentencia *declare*

`<declaración> ::= (declare <propiedad>)`

`<propiedad> ::= (salience <expresión-entera>)`



# Propiedades de una regla

## Prioridad

- Se indica en la declaración de propiedades con la palabra reservada *salience*
  - Puede tomar valores entre -10000 y 10000
  - El valor por defecto es 0
  - Cuándo puede evaluarse la prioridad:
    - Cuando se define la regla (por defecto)
    - Cuando se activa la regla
    - En cada ciclo de ejecución
- } Prioridad dinámica





# Propiedades de una regla

## Prioridad

```
CLIPS> (clear)
```

```
CLIPS> (defrule primera
  (declare (salience 10))
```

```
=>
```

```
(printout t "Me ejecuto la primera" crlf))
```

```
CLIPS> (defrule segunda
```

```
=>
```

```
(printout t "Me ejecuto la segunda" crlf))
```

```
CLIPS> (reset)
```

```
CLIPS> (run)
```

```
Me ejecuto la primera
```

```
Me ejecuto la segunda
```



# Reglas

- Introducción
- Definición de reglas
- Ciclo básico de ejecución de reglas
- Sintaxis del antecedente
- Propiedades de una regla
- Comandos



# Comandos defrule

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

```
(list-defrules [<nombre-módulo> | *])
```

```
(rules [<nombre-módulo> | *])
```

```
(undefrule <nombre-regla> | *)
```



# Comandos

## defrule

```

CLIPS> (defrule ej1 => (printout t "Ejemplo 1" crlf))
CLIPS> (defrule ej2 => (printout t "Ejemplo 2" crlf))
CLIPS> (list-defrules)
ej1
ej2
For a total of 2 defrules.
CLIPS> (ppdefrule ej2)
(defrule MAIN::ej2
  =>
    (printout t "Ejemplo 2" crlf))
CLIPS> (undefrule ej1)
CLIPS> (list-defrules)
ej2
For a total of 1 defrule.
CLIPS> (undefrule *)
CLIPS> (list-defrules)
  
```



# Comandos agenda

(agenda [<nombre-módulo> | \*])

(run [<expresión-entera>])



# Comandos agenda

```
CLIPS> (defrule ej1 => (printout t "Ejemplo 1"
  crlf))
```

```
CLIPS> (defrule ej2 => (printout t "Ejemplo 2"
  crlf))
```

```
CLIPS> (reset)
```

```
CLIPS> (agenda)
```

```
0      ej1: f-0
```

```
0      ej2: f-0
```

For a total of 2 activations.

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

```
Ejemplo 1
```

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

```
Ejemplo 2
```

```
CLIPS> (agenda)
```