

# Tema 6: Reglas

El constructor *defrule*

# Reglas

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

# 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**.

# Reglas

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

# Definición de reglas

## Sintaxis del constructor *defrule*

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

# Definición de reglas

## Ejemplo

```
(defrule restaurante  
  "Cuándo ir al restaurante"  
  (tengo hambre)  
  (tengo dinero)  
=>  
  (assert (ir-a restaurante)))
```

# Definición de reglas

## Consideraciones

- Si se define una regla con el mismo nombre que otra, aun siendo errónea, machaca a la anterior.
- Puede no haber ningún elemento condicional en el antecedente.
- En el caso anterior se usa automáticamente (*initial-fact*) como elemento condicional.
- Puede no haber ninguna acción en el consecuente.
- En el caso anterior la ejecución de la regla no tiene ninguna consecuencia.
- El antecedente es de tipo conjuntivo.

# Definición de reglas

## Ejemplos

```
(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 relacionados
- Ejemplos

# 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 o instancia de regla: nombre de la regla e identificadores de los hechos que la satisfacen.
- Una regla se dispara cuando el motor de inferencia decide ejecutar las acciones de su consecuente.
- Las activaciones se almacenan en la agenda.
- Cada regla tiene una prioridad entre -10000 y 10000 (0 por defecto).
- Se aplica una estrategia de resolución de conflictos para decidir 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 y se elimina de la agenda.
6. Volver al paso 2.

# Ciclo básico de ejecución

## Ejemplo 1

```
CLIPS> (deffacts ejemplo-restaurante
  (tengo hambre)
  (tengo dinero))
CLIPS> (defrule restaurante
  (tengo hambre)
  (tengo dinero)
  =>
  (assert (ir-a restaurante)))
CLIPS> (reset)
CLIPS> (facts)
f-0      (initial-fact)
f-1      (tengo hambre)
f-2      (tengo dinero)
For a total of 3 facts.
CLIPS> (agenda)
0        restaurante: f-1,f-2
For a total of 1 activation.
CLIPS> (run)
```

# Ciclo básico de ejecución

## Ejemplo 1

```
CLIPS> (facts)
f-0      (initial-fact)
f-1      (tengo hambre)
f-2      (tengo dinero)
f-3      (ir-a restaurante)
For a total of 4 facts.
CLIPS> (agenda)
CLIPS> (retract 2)
CLIPS> (assert (tengo dinero))
<Fact-4>
CLIPS> (facts)
f-0      (initial-fact)
f-1      (tengo hambre)
f-3      (ir-a restaurante)
f-4      (tengo dinero)
For a total of 4 facts.
CLIPS> (agenda)
0        restaurante: f-1,f-4
For a total of 1 activation.
```

# Ciclo básico de ejecución

## Ejemplo 2

```
CLIPS> (def facts ejemplo-restaurante
  (tengo hambre)
  (tengo dinero))
CLIPS> (defrule restaurante
  (tengo hambre)
  (tengo dinero)
  =>
  (assert (ir-a restaurante)))
CLIPS> (reset)
CLIPS> (facts)
f-0      (initial-fact)
f-1      (tengo hambre)
f-2      (tengo dinero)
For a total of 3 facts.
CLIPS> (agenda)
0        restaurante: f-1,f-2
For a total of 1 activation.
```

# Ciclo básico de ejecución

## Ejemplo 2

```
CLIPS> (run)
CLIPS> (facts)
f-0      (initial-fact)
f-1      (tengo hambre)
f-2      (tengo dinero)
f-3      (ir-a restaurante)
For a total of 4 facts.
CLIPS> (agenda)
CLIPS> (reset)
CLIPS> (facts)
f-0      (initial-fact)
f-1      (tengo hambre)
f-2      (tengo dinero)
For a total of 3 facts.
CLIPS> (agenda)
0        restaurante: f-1,f-2
For a total of 1 activation.
```

# Ciclo básico de ejecución

## Ejemplo 3

```
CLIPS> (defrule hola-mundo
=>
  (printout t "Hola mundo" crlf))
CLIPS> (reset)
CLIPS> (facts)
f-0      (initial-fact)
For a total of 1 fact.
CLIPS> (agenda)
0        hola-mundo: f-0
For a total of 1 activation.
CLIPS> (run)
Hola mundo
CLIPS> (facts)
f-0      (initial-fact)
For a total of 1 fact.
CLIPS> (agenda)
```



# Reglas

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

# 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.
- El primer campo de un patrón debe ser un símbolo.
- Elementos:
  - Literales
  - Comodines
  - Variables
  - Conectores lógicos
  - Predicados
  - Valores devueltos por funciones

# EC patrón

## Restricciones

- Restricción para hechos ordenados

(`<símbolo>` `<restricción-1>` ... `<restricción-n>`)

- Restricción para hechos no ordenados

(`<nombre-deftemplate>`

`<slot-1>` `<restricción-1>` ... `<restricción-n>`)

.

.

.

(`<slot-1>` `<restricción-1>` ... `<restricción-m>`)

# EC patrón

## Restricciones literales

- Contienen sólo constantes.

`<restricción> ::= <constante>`

# 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 rojo azul)
```

```
f-6      (datos 1 azul rojo 6.9)
```

For a total of 7 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

- Comodín monocampo: ?
- Comodín multicampo: \$?
- Una casilla no especificada en un patrón basado en una plantilla se compara con un comodín implícito.

`<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

(dato \$? VERDE \$?)

Emparejaría con:

(dato VERDE)

(dato VERDE rojo azul)

(dato rojo VERDE azul)

(dato rojo azul VERDE)

(dato VERDE azul VERDE)

# EC patrón

## Restricciones con comodines

Si tenemos una plantilla  
`(deftemplate persona`  
     `(multislot nombre)`  
     `(slot edad))`

El patrón

`(persona (nombre Juan))`

equivale al patrón

`(persona (nombre Juan) (edad ?))`

El patrón

`(persona (edad 40))`

equivale al patrón

`(persona (nombre $?) (edad 40))`

# EC patrón

## Restricciones con variables

- 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

- Las variables son similares a los comodines, pero tienen nombre y conservan su valor dentro de su ámbito.
- La primera vez que aparece una variable en el antecedente de una regla, si dicha regla concuerda con algún conjunto de hechos, se asignará un valor a la variable, que conservará dentro de su ámbito.
- La ligadura de un valor a una variable se mantiene únicamente en el ámbito de la regla en que ocurre.
- Las variables son locales a una regla.
- Cada regla mantiene una lista privada de nombres de variables y valores asociados.

# EC patrón

## Restricciones con variables

```
CLIPS> (defrule prueba
```

```
=>
```

```
(printout t ?x crlf))
```

```
[PRCCODE3] Undefined variable x  
referenced in RHS of defrule.
```

ERROR:

```
(defrule MAIN::prueba
```

```
=>
```

```
(printout t ?x crlf))
```

# 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 conectivas

- 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 indefinida 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> (defrule ejemplo1-1 (datos-A ~azul) => )
CLIPS> (defrule ejemplo1-2 (datos-B (valor ~rojo&~verde)) =>
)
CLIPS> (defrule ejemplo1-3 (datos-B (valor verde|rojo)) => )
CLIPS> (facts)
f-0      (initial-fact)
f-1      (datos-A verde)
f-2      (datos-A azul)
f-3      (datos-B (valor rojo))
f-4      (datos-B (valor azul))
For a total of 5 facts.
CLIPS> (agenda)
0        ejemplo1-3: f-3
0        ejemplo1-2: f-4
0        ejemplo1-1: f-1
For a total of 3 activations.
```

# EC patrón

## Restricciones conectivas

```
(defrule regla1
  (dato (valor ?x&~rojo&~verde))
  =>
  (printout t "Valor: " ?x crlf))
```

```
CLIPS> (reset)
```

```
CLIPS> (facts)
```

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

```
f-1      (dato (valor rojo))
```

```
f-2      (dato (valor azul))
```

For a total of 3 facts.

```
CLIPS> (run)
```

```
Valor: azul
```

# EC patrón

## Restricciones conectivas

```
(defrule regla2
  (dato (valor ?x & azul | verde))
  (dato (valor ?y&~?x))
  =>
  (printout t ?x " y " ?y crlf))
```

CLIPS> (reset)

CLIPS> (facts)

f-0 (initial-fact)

f-1 (dato (valor rojo))

f-2 (dato (valor azul))

For a total of 3 facts.

CLIPS> (agenda)

0 regla2: f-2,f-1

For a total of 1 activation.

CLIPS> (run)

azul y rojo

# 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> (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 de valor devuelto

- Se usa el valor devuelto por una función para restringir un campo.
- Se indica mediante el carácter '='.
- El valor devuelto se incorpora en el patrón en la posición en la que se encuentra la llamada a la función y actúa como una restricción literal.

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

# 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> (facts)
```

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

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

```
f-2      (dato 3)
```

For a total of 3 facts.

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

```
CLIPS> (run)
```

```
CLIPS> (facts)
```

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

```
f-2      (dato 3)
```

For a total of 2 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 test

- Bajo determinadas circunstancias, en las reglas en las que aparece un EC test, *CLIPS* puede añadir el EC (`initial-fact`) al antecedente, por lo que se debe ejecutar el comando *reset* para garantizar el correcto funcionamiento bajo todas las circunstancias.

# 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 activará varias veces.

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

# EC or

```
CLIPS> (defrule posibles-tostadas
        (tengo pan)
        (or   (tengo mantequilla)
              (tengo aceite))
        =>
        (assert (desayuno tostadas)))
```

```
CLIPS> (facts)
f-0      (initial-fact)
f-1      (tengo pan)
f-2      (tengo mantequilla)
f-3      (tengo aceite)
```

For a total of 4 facts.

```
CLIPS> (agenda)
0        posibles-tostadas: f-1,f-3
0        posibles-tostadas: f-1,f-2
```

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.
- No se puede asignar a una variable la dirección de un EC not.

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

# EC not

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

# EC not

- Las variables a las que se asigna un valor dentro de un EC not mantienen su valor sólo dentro del EC not.

# EC not

```

CLIPS> (defrule numero-mayor
  (numero ?x)
  (not (numero ?y&:(> ?y ?x)))
  =>
  (printout t ?x " es el mayor número" crlf))
CLIPS>
(defrule numero-no-mayor
  (numero ?x)
  (not (numero ?y&:(> ?y ?x)))
  =>
  (printout t ?y " no es el mayor número" crlf))

```

[PRCCODE3] Undefined variable y referenced in RHS of defrule.

# EC not

- Bajo determinadas circunstancias, en las reglas en las que aparece un EC not, *CLIPS* puede añadir el EC (`initial-fact`) al antecedente, por lo que se debe ejecutar el comando *reset* para garantizar el correcto funcionamiento en todas las circunstancias.

# EC not y la propiedad de refracción

```
CLIPS> (defrule tenis
  (not (tiempo lluvioso))
  =>
  (printout t "Podemos jugar al tenis." crlf))
CLIPS> (agenda)
CLIPS> (reset)
CLIPS> (agenda)
0      tenis: f-0,
For a total of 1 activation.
CLIPS> (run)
Podemos jugar al tenis.
CLIPS> (agenda)
CLIPS> (assert (tiempo lluvioso))
<Fact-1>
CLIPS> (retract 1)
CLIPS> (agenda)
0      tenis: f-0,
For a total of 1 activation.
```

# EC exists

- Permite comprobar si una serie de ECs se satisface por al menos un conjunto de hechos.

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



# EC exists

```
CLIPS> (defrule tostadas-mermelada
  (tengo pan)
  (tengo mermelada ?)
  =>
  (printout t "Desayuno tostadas con mermelada." crlf))
```

```
CLIPS> (facts)
f-0      (initial-fact)
f-1      (tengo pan)
f-2      (tengo mermelada fresa)
f-3      (tengo mermelada albaricoque)
f-4      (tengo mermelada naranja)
```

For a total of 5 facts.

```
CLIPS> (agenda)
0        tostadas-mermelada: f-1,f-4
0        tostadas-mermelada: f-1,f-3
0        tostadas-mermelada: f-1,f-2
```

For a total of 3 activations.

# EC exists (continúa)

```
CLIPS> (defrule tostadas-mermelada
  (tengo pan)
  (exists (tengo mermelada ?)))
=>
  (printout t "Desayuno tostadas con mermelada." crlf))
CLIPS> (facts)
f-0      (initial-fact)
f-1      (tengo pan)
f-2      (tengo mermelada fresa)
f-3      (tengo mermelada albaricoque)
f-4      (tengo mermelada naranja)
For a total of 5 facts.
CLIPS> (agenda)
0        tostadas-mermelada: f-1,
For a total of 1 activation.
```

# EC exists

- El EC exists está implementado internamente usando el EC not.
- Como el EC not puede hacer que se añada el EC (`initial-fact`) al antecedente, se debe ejecutar el comando *reset* para garantizar el correcto funcionamiento bajo todas las circunstancias.

# EC forall

- Permite comprobar si siempre que se satisface un determinado EC se satisfacen también otros ECs dados.

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

- Se satisface si siempre que se satisface el primer EC se satisfacen también los siguientes.

# 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 forall

- El CE forall está implementado internamente usando el EC not.
- Como el EC not puede hacer que se añada el EC (`initial-fact`) al antecedente, se debe ejecutar el comando *reset* para garantizar el correcto funcionamiento bajo todas las circunstancias.

# EC logical

- Permite indicar que la existencia de un conjunto de hechos depende de la existencia de otro conjunto de hechos.
- Los hechos del antecedente que formen parte de un EC logical proporcionan soporte lógico a los hechos creados en el consecuente.
- Un hecho puede recibir soporte lógico de varios conjuntos distintos de hechos (de varias reglas).
- Un hecho permanece mientras permanezca alguno de los conjuntos de hechos que lo soportan lógicamente.

# EC logical

- Los ECs incluidos en un EC logical están unidos por un and implícito.
- Sólo los primeros ECs del antecedente pueden ser de tipo logical.

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



# 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
- Ejemplos

# 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 relacionados
- Ejemplos

# Comandos relacionados defrule

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

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

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



# Comandos relacionados

## 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 relacionados agenda

(agenda [<nombre-módulo> | \*])

(run [<expresión-entera>])

# Comandos relacionados 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)
```

# Reglas

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

# Ejemplos

## Sumar números

```
(deffacts valores-a-sumar  
  (dato 1)  
  (dato 2)  
  (dato 3)  
  (dato 4)  
  (dato 5)  
  (suma 0))
```

# Ejemplos

## Sumar números

```
(defrule sumar-numeros
  ?s <- (suma ?total)
  ?t <- (dato ?valor)
  =>
  (retract ?s ?t)
  (assert (suma (+ ?total ?valor)))))
```

```
(defrule mostrar-suma
  (not (dato ?))
  ?s <- (suma ?total)
  =>
  (retract ?s)
  (printout t "La suma es: " ?total crlf))
```

# Ejemplos

## Pila

; La cabeza de la pila queda a la izquierda.

```
(deffacts inicio-pila
  (pila))
```

```
(defrule meter-valor-pila
```

```
  ?v <- (meter ?valor)
```

```
  ?p <- (pila $?resto)
```

```
=>
```

```
(retract ?v ?p)
```

```
(assert (pila ?valor $?resto))
```

```
(printout t "Metiendo valor: " ?valor crlf))
```

# Ejemplos

## Pila

```

(defrule sacar-valor-pila
  ?s <- (sacar)
  ?p <- (pila ?valor $?resto)
=>
  (retract ?s ?p)
  (assert (pila $?resto))
  (printout t "Sacando valor: " ?valor crlf))

```

```

(defrule intentar-sacar-valor-pila-vacia
  ?s <- (sacar)
  (pila)
=>
  (retract ?s)
  (printout "La pila está vacía" crlf))

```



# Ejemplos

## Pila

```

CLIPS> (reset)
CLIPS> (assert (meter primero))
<Fact-2>
CLIPS> (run)
Metiendo valor: primero
CLIPS> (assert (meter 2))
<Fact-4>
CLIPS> (run)
Metiendo valor: 2
CLIPS> (facts)
f-0      (initial-fact)
f-5      (pila 2 primero)
For a total of 2 facts.
CLIPS> (assert (sacar))
<Fact-6>
CLIPS> (run)
Sacando valor: 2
  
```