

Grado en Ingeniería Informática, Universidad de Córdoba

Sistemas Inteligentes

CLIPS

Tema 5: Reglas II

Aurora Esteban Toscano
aestebant@uco.es

José Manuel Alcalde Llergo
i72allj@uco.es

Marzo de 2023



- Conocer las posibilidades de configuración del antecedente a través de los *Elementos Condicionales* de CLIPS.
- Aplicar los Elementos Condicionales tanto en hechos ordenados como en no ordenados.



Si el **antecedente** es cierto según los hechos almacenados en la **base de afirmaciones**, entonces **pueden** realizarse las acciones especificadas en el **consecuente**.

Estructura de las reglas

Antecedente \implies Consecuente

- Antecedente: cero o más *cláusulas* que deben cumplirse para que la regla pueda ejecutarse (dispararse).
 - Entidad patrón: elementos en los que se basa la activación de una regla. Pueden ser hechos ordenados, plantillas e instancias de clases.
 - Elemento condicional (EC): son cada una de las condiciones que pueden encadenarse para componer el antecedente de la regla.
- Consecuente: cero o más *acciones* que se llevarán a cabo si la regla se dispara.
 - Entre esas acciones puede estar crear (afirmar) más hechos, eliminar hechos obsoletos, llegar a conclusiones finales...



Tipos de Elementos Condicionales

Los EC se basan en aplicar restricciones sobre entidades patrón y en la concatenación de sus resultados para construir el antecedente de la regla.

- **EC patrón:** verificar los elementos de una entidad patrón.
- **EC test:** comprobar el valor devuelto por una función.
- **EC and:** comprobar si en una cadena de EC todos se cumplen.
- **EC or:** comprobar si en una cadena de EC al menos uno se cumple.
- **EC not:** invertir el resultado del EC que contiene.
- **EC exists:** comprobar si los EC que contiene se satisfacen por algún conjunto de hechos.
- **EC forall:** comprobar si los EC que contiene se satisfacen para toda ocurrencia de otro EC.
- **EC logical:** asegurar el mantenimiento de verdad para hechos que se basan en la presencia de otros hechos.



Los EC patrón se usan para verificar si se cumplen campos de una entidad patrón en base a ciertos tipos de restricciones.

- Tipos de restricciones: literales, comodines, variables, sentencias conectivas, predicados, valores devueltos, direcciones de hechos.

Estructura: hechos ordenados

```
(defrule <nombre> [comentario] [propiedades]
  (<hecho ordenado> <restriccion>*)
  =>
  <consecuente>
)
```

Estructura: hechos no ordenados

```
(defrule <nombre> [comentario] [propiedades]
  (<nombre plantilla>
    (<campo> <restriccion>)*
  )
  =>
  <consecuente>
)
```



EC patrón: restricciones literales

- Verifican que la entidad patrón toma un valor exacto.
- Restricciones más básica, basadas únicamente en constantes.

Por ejemplo

```
(def facts prueba1
  (ejemplo 1.0 azul "rojo")
  (ejemplo 1 azul)
  (ejemplo 1 azul rojo)
  (ejemplo 1 azul ROJO)
  (ejemplo 1 azul rojo 6.9)
  (ejemplo 1.0 amarillo ROJO 33)
  (ejemplo 3 rojo amarillo 9)
)
(defrule buscar-literales
  (ejemplo 1 azul rojo) =>
)
```

Por ejemplo

```
(def template persona
  (slot nombre)
  (slot edad)
  (multislot amigos))
(def facts prueba2
  (persona (nombre Aurora) (edad 26))
  (persona (nombre Juan) (edad 20))
  (persona (nombre Jose) (edad 20)
    (amigos Adela Jorge))
  (persona (nombre Adela) (edad 9)
    (amigos Jose))
  (persona (nombre Carlos) (edad 40)
    (amigos María Antonio Lourdes)))
(defrule buscar-edad (persona (edad 20)) => )
```



- Indican que cualquier valor en la posición indicada de la entidad patrón es válido para la regla.
- Dos tipos:
 - Comodín monocampo ?: Busca exactamente un campo.
 - Comodín multicampo \$?: Busca cualquier número de campos (incluso 0).

Por ejemplo

```
(defrule buscar-comodin
  (ejemplo ? azul rojo $?) =>
)
```

Por ejemplo

```
(defrule buscar-un-amigo
  (persona (amigos ?)) =>
)
```



Los EC patrón literal y comodín se pueden combinar para especificar restricciones complejas.

- Buscar un literal en cualquier posición: (... \$? <literal> \$? ...)

Por ejemplo

(defrule encontrar-literal (ejemplo \$? AMARILLO \$?) =>)

Emparejaría con

(ejemplo AMARILLO), (ejemplo 2.5 AMARILLO),
(ejemplo "rojo" AMARILLO 89), (ejemplo "rojo" 33 AMARILLO)...

- Buscar un multcampo con al menos un elemento: (... ? \$?)

Por ejemplo

(defrule al-menos-un-amigo (persona (amigos ? \$?)))



- Almacenan un valor para luego emplearse en otros EC del antecedente o en el consecuente de una regla.
- Dos tipos:
 - Variable monocampo
?<nombre>: almacena exactamente un campo.
 - Variable multicampo
\${<nombre>: almacena cualquier número de campos consecutivos.

OjO

La primera vez que la variable aparece actúa como un comodín, pero su valor queda ligado al valor del campo. Si la variable vuelve a aparecer, debe coincidir el valor ligado.

OjO'

El ámbito de la variable sólo abarca la regla en la que aparece.



Por ejemplo

```
(defrule encontrar-terna
  (ejemplo ?x ?y ?z)
  =>
  (printout t ?x " : " ?y " : " ?z crlf)
)
```

Por ejemplo

```
(defrule mis-amigos
  (persona (nombre ?n) (amigos $?a))
  =>
  (printout t "Me llamo " ?n " y mis
    amigos son " $?a crlf)
)
```



- Permiten modificar EC patrón de tipo *restricción* mediante conectores lógicos básicos: negación (\sim), y lógico ($\&$), o lógico ($|$).

Estructura

(... <restricción simple> <|/&> <restricción simple> ...)

<restriccion simple> :: [~] <literal | var. monocampo | var. multicampo>

- Precedencia: $\sim > \& > |$
 - Excepción: en situaciones compuestas de <variable>&<restriccion>*, la variable se trata a parte.

$?x\&\text{rojo}|\text{azul} \equiv ?x\&(\text{rojo}|\text{azul})$



Por ejemplo

```
(defrule busca-rojo (ejemplo ? azul rojo|ROJO|"rojo" $?) => )  
(defrule excluye-azul  
  (ejemplo ? ?x&~azul $?) => (printout t "No soy azul, soy " ?x crlf)  
)
```

Por ejemplo

```
(defrule no-profes  
  (persona (nombre ?n&~Aurora&~Carlos))  
  =>  
  (printout t ?n " no es profesor" crlf)  
)
```



- Restringen el campo según el resultado de una expresión lógica → la función debe devolver no FALSE para que el patrón evaluado empareje con la regla.

Estructura

(... <restricción simple> : <función> ...)

- Uso típico: variable + conectiva & + EC patrón predicado.
- Algunas funciones de CLIPS que devuelven expresiones lógicas:
 - Tipos de datos: integerp, floatp, numberp, symbolp, stringp, lexemep
 - Booleana: eq, neq, and, or, not
 - Lógica mates: =, <>, <, <=, >, >=, evenp, oddp



Por ejemplo

```
(defrule busca-cadena (ejemplo ? ? ?x&:(stringp ?x) $?) => )  
(defrule buca-1 (ejemplo ?x&:(= ?x 1) $?) => )  
(defrule buca-1-mal (ejemplo ?x&:(eq ?x 1) $?) => )
```

Por ejemplo

```
(defrule busca-jovenes  
  (persona (nombre ?n) (edad ?x&:(> ?x 15)&:(< ?x 25)))  
  =>  
  (printout t ?n " es joven" crlf)  
)
```



- Restringe un campo en base al valor devuelto por una función → el resultado debe ser uno de los tipos de datos primitivos.
- El resultado de la función actuará como un literal limitando los valores de los EC a ese resultado.

Estructura

(... =<función> ...)

- Algunas funciones numéricas de CLIPS: +, -, *, /, div, mod, sqrt, **, round, abs, max, min

Por ejemplo

```
(defrule buscar-triple
  (ejemplo ?x $? =(* 3 ?x) $?) =>
)
```



Para actuar sobre hechos en el consiguiente de una regla es necesario capturar su dirección y guardarla en una variable.

- En el antecedente se utiliza el operador `<-`: `?dir <- <hecho>`
 - En el consecuente se pueden aplicar las operaciones:
 - Eliminación: `(retract ?dir)`
 - Modificación*: `(modify ?dir (<campo> <nuevo valor>))`
 - Duplicación*: `(duplicate ?dir (<campo> <nuevo valor>))`
- * Solo para hechos no ordenados

Por ejemplo

```
(defrule comenzar
  ?h <- (iniciar_programa)
  =>
  (retract ?h) (printout t "Iniciando..." crlf)
)
```




- Recupera el ejercicio sobre pilas y colas trabajado la semana pasada.
- Añade una modificación para que sólo admita datos de tipo numérico.
- Añade una modificación para que la pila se rellene automáticamente dado un dato inicial y cuántos elementos más añadir.

Por ejemplo, si se parte de los hechos (pila), (dato 1) y (elementos 5), el resultado final será:

(pila 5 4 3 2 1)



El EC test comprueba el valor devuelto por una función lógica. Se satisface si el resultado es TRUE. Por el contrario, no se satisface si la función devuelve FALSE.

- Uso típico: actúa sobre variables previamente capturadas en operaciones similares a los EC patrón predicados, pero más complejas.

Estructura

```
(defrule <nombre> [comentario] [propiedades]
  (<hecho> | <restricción> |...)*
  (test <función>)
  =>
  <consecuente>
)
```



Por ejemplo

```
(defrule pila-correcta
  (pila ?v ?w ?x ?y ?z)
  (test (> ?v ?w ?x ?y ?z))
  =>
  (printout t "La pila está correcta" crlf)
)
```



Estos EC sirven para encadenar otros EC mediante operaciones lógicas.

Estructura

(and | or <elemento condicional> +)

(not <elemento condicional>)

- EC or: se satisface si se satisface cualquiera de los EC que lo compone.
 - La regla se activará para tantas ocurrencias como EC dentro del or se cumplan.
- EC and: se satisface si todos los EC que lo componen se satisface.
 - Se puede utilizar en combinación con *or* en el antecedente. Utilizarlo sólo no es necesario ya que el comportamiento por defecto del antecedente es unir todas los EC mediante y lógico.
- EC not: se satisface si no se satisface el EC contenido.
 - Sólo puede abarcar un EC.
 - Uso típico: comprobar que un hecho no existe en la base de afirmaciones.



Por ejemplo

```
(defrule desayuno-sano
  (tengo zumo)
  (or (and (tengo pan) (tengo aceite))
      (and (tengo leche) (tengo cereales))))
  (not (tengo bollo))
  =>
  (printout t "Desayuno sano")
)
```

Por ejemplo

```
(defrule completa-bd
  (persona (nombre ?n1)
            (amigos $? ?n2 $?))
  (not (persona (nombre ?n2)))
  =>
  (assert (persona (nombre ?n2)
                  (amigos ?n1)))
)
```

Ejercicio 1: ¿cómo podría mejorarse la regla mis-amigos definida anteriormente para que no muestre personas sin amigos?

Ejercicio 2: ¿cómo se crearía una regla que inicialice la pila/cola de ejercicios anteriores en caso de que no exista?



El EC exists permite comprobar si una serie de EC se satisface por **algún** conjunto de hechos.

- El comportamiento normal de CLIPS es que la regla se active para **todos** los conjuntos de hechos que la satisfagan.

Estructura

(exists <elemento condicional> +)



Observa la regla del fichero disponible en Moodle superheroes.clp. Cárgalo en Moodle y observa el comportamiento:

```
CLIPS> (clear)
CLIPS> (load superheroes.clp)
CLIPS> (reset)
CLIPS> (facts)
f-1    (superhero (nombre spiderman) (estado libre))
f-2    (superhero (nombre viuda-negra) (estado ocupado))
f-3    (superhero (nombre dextrange) (estado libre))
f-4    (superhero (nombre capitana-marvel) (estado ocupado))
CLIPS> (agenda)
0      salvar-dia: f-3
0      salvar-dia: f-1
```

Solución

```
(defrule salvar-dia "Necesitamos que un súper heroe salve el día"
  (exists (superhero (estado libre))) => (printout t "El día está salvado" crlf)
)
```



El EC forall permite comprobar si para toda ocurrencia del primer EC se satisface el conjunto de los siguientes EC contenidos en el forall.

Estructura

(forall <elemento condicional> <elemento condicional>+)



Descarga y observa la estructura del fichero `estudiantes.clp`. ¿Qué es lo que falta para que se active la regla `todos-limpios`?

La regla

```
(defrule todos-limpios
  (forall (estudiante ?nombre)
    (aprobado lengua ?nombre) (aprobado matematicas ?nombre)
    (aprobado historia ?nombre)
  )
=>
(printout t "Todos mis estudiantes pasan de curso" crlf)
)
```

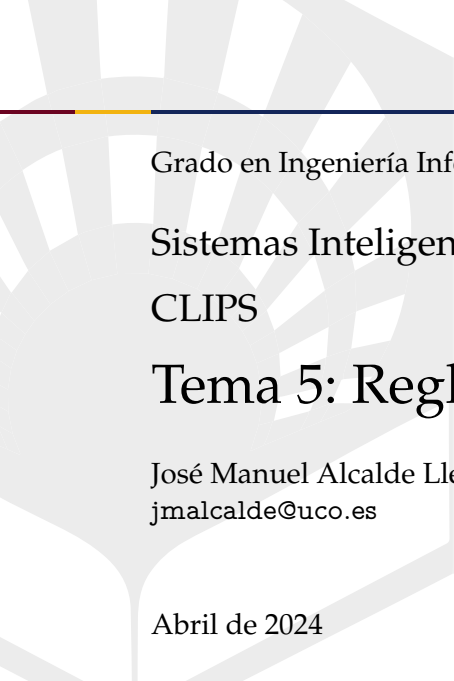


El EC logical asegura el *mantenimiento de verdad* de los hechos creados a partir de él.

- 1 Los hechos del antecedente proporcionan **soporte lógico** a los hechos que se han creado en el consecuente de una regla basada en el EC forall. → Los hechos permanecen en la base de hechos mientras permanezcan los que los soportan lógicamente.
- 2 Un hecho puede recibir soporte lógico de varios conjuntos distintos de hechos → para permanecer necesita que exista al menos uno de los conjuntos.
- 3 Los EC incluidos en un EC logical están unidos implícitamente por Y lógico.
- 4 Puede combinarse con EC *and*, *or* y *not*.
- 5 Sólo los primeros EC del antecedente pueden ser de tipo logical.



```
CLIPS> (defrule puedo-pasar (logical (or
    (and (semaforo verde) (not (viene ambulancia))) (and (semaforo rojo) (sin coches))))
    => (assert (puedo cruzar-paso)))
CLIPS> (assert (semaforo verde))
CLIPS> (run)
CLIPS> (facts)
f-1    (semaforo verde)
f-2    (puedo cruzar-paso)
CLIPS> (assert (viene ambulancia))
CLIPS> (facts)
f-1    (semaforo verde)
f-3    (viene ambulancia)
CLIPS> (retract 1 3)
CLIPS> (assert (semaforo rojo) (sin coches))
CLIPS > (run)
CLIPS> (facts)
f-4    (semaforo rojo)
f-5    (sin coches)
f-6    (puedo cruzar-paso)
CLIPS> (retract 5)
CLIPS> (facts)
f-4    (semaforo rojo)
```



Grado en Ingeniería Informática, Universidad de Córdoba

Sistemas Inteligentes

CLIPS

Tema 5: Reglas II

Aurora Esteban Toscano
aestebant@uco.es

José Manuel Alcalde Llergo
i72allj@uco.es

Marzo de 2023