Tema 1(I): Programación basada en reglas con CLIPS

José A. Alonso Jiménez
Francisco Jesús Martín Mateos
José Luis Ruiz Reina

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Programación basada en reglas

- **Paradigma de la programación basada en reglas**
  - **Hechos: pieza básica de información**
  - **Reglas: describen el comportamiento del programa en función de la información existente**
- **Modelo de regla:**

  `<Condiciones> => <Acciones>`
- **Condiciones acerca de:**
  - **Existencia de cierta información**
  - **Ausencia de cierta información**
  - **Relaciones entre datos**
- **Acciones:**
  - **Incluir nueva información**
  - **Eliminar información**
  - **Presentar información en pantalla**

Definición de hechos

- **Estructura de un hecho:** `(<simbolo><datos>*)`
  - **Ejemplo:** `(conjunto a b 1 2 3)`
  - `(1 2 3 4)` **no es un hecho válido**
- **La acción de añadir hechos:** `(assert <hecho>*)`
- **Hechos iniciales:**

```
(deffacts <nombre>
   <hecho>*)
```

Definición de reglas

- **Estructura de una regla (I):**

```
(defrule <nombre>
  <condicion>*
  =>
  <accion>*)

<condicion> := <hecho>
```

- **Ejemplo:**

```
(defrule mamifero-1
  (tiene-pelos)
  =>
  (assert (es-mamifero)))
```

Interacción con el sistema

- **Cargar el contenido de un archivo:**

  `(load <archivo>)`

- **Trazas:**
  - **Hechos añadidos y eliminados:**

    `(watch facts)`

  - **Activaciones y desactivaciones de reglas:**

    `(watch activations)`

  - **Utilización de reglas:**

    `(watch rules)`

Interacción con el sistema

- **Inicialización:**

  `(reset)`

- **Ejecución:**

  `(run)`

- **Limpiar la base de conocimiento:**

  `(clear)`

- **Ayuda del sistema:**

  `(help)`

Ejemplo de base de conocimiento (hechos y reglas, I)

```
(deffacts hechos-iniciales
  (tiene-pelos)
  (tiene-pezugnas)
  (tiene-rayas-negras))

(defrule mamifero-1
  (tiene-pelos)
  =>
  (assert (es-mamifero)))

(defrule mamifero-2
  (da-leche)
  =>
  (assert (es-mamifero)))

(defrule ungulado-1
  (es-mamifero)
  (tiene-pezugnas)
  =>
  (assert (es-ungulado)))
```

Ejemplo de base de conocimiento (hechos y reglas, II)

```
(defrule ungulado-2
  (es-mamifero)
  (rumia)
  =>
  (assert (es-ungulado)))

(defrule jirafa
  (es-ungulado)
  (tiene-cuello-largo)
  =>
  (assert (es-jirafa)))

(defrule cebra
  (es-ungulado)
  (tiene-rayas-negras)
  =>
  (assert (es-cebra)))
```

Tabla de seguimiento

- **El modelo de ejecución en CLIPS**
  - **Base de hechos**
  - **Base de reglas**
  - **Activación de reglas y agenda**
  - **Disparo de reglas**
- **Tabla de seguimiento:**

Hechos	E	Agenda	D
`f0 (initial-fact)`	0		
`f1 (tiene-pelos)`	0	`mamifero-1: f1`	1
`f2 (tiene-pezugnas)`	0		
`f3 (tiene-rayas-negras)`	0		
`f4 (es-mamifero)`	1	`ungulado-1: f4,f2`	2
`f5 (es-ungulado)`	2	`cebra: f5,f3`	3
`f6 (es-cebra)`	3		

Un ejemplo de sesión en CLIPS

```
CLIPS> (load "animales.clp")
$******
TRUE
CLIPS> (watch facts)
CLIPS> (watch rules)
CLIPS> (watch activations)
CLIPS> (reset)
==> f-0     (initial-fact)
==> f-1     (tiene-pelos)
==> Activation 0      mamifero-1: f-1
==> f-2     (tiene-pezugnas)
==> f-3     (tiene-rayas-negras)
CLIPS> (run)
FIRE    1 mamifero-1: f-1
==> f-4     (es-mamifero)
==> Activation 0      ungulado-1: f-4,f-2
FIRE    2 ungulado-1: f-4,f-2
==> f-5     (es-ungulado)
==> Activation 0      cebra: f-5,f-3
FIRE    3 cebra: f-5,f-3
==> f-6     (es-cebra)
```

Plantillas y variables

- **Estructura de una plantilla (I):**

```
(deftemplate <nombre>
  <campo>*)

<campo> := (slot <nombre-campo>)
```

- **Ejemplos:**

```
(deftemplate persona
  (slot nombre)
  (slot ojos))
```

- **Variables:** `?x, ?y, ?gv32`
  - **Toman un valor simple**

Restricciones

- **Restricciones:**
  - **Condiciones sobre las variables que se comprueban en el momento de verificar las condiciones de una regla**
- **Algunos tipos de restricciones:**
  - **Negativas:**

    ```
    (dato ?x&~a)
    ```

  - **Disyuntivas:**

    ```
    (dato ?x&a|b)
    ```

  - **Conjuntivas:**

    ```
    (dato ?x&~a&~b)
    ```

Ejemplo

```
(deftemplate persona
  (slot nombre)
  (slot ojos))

(deffacts personas
  (persona (nombre Ana)    (ojos verdes))
  (persona (nombre Juan)   (ojos negros))
  (persona (nombre Luis)   (ojos negros))
  (persona (nombre Blanca) (ojos azules)))

(defrule busca-personas
  (persona (nombre ?nombre1)
           (ojos ?ojos1&azules|verdes))
  (persona (nombre ?nombre2&~?nombre1)
           (ojos negros))
  =>
  (printout t ?nombre1
              " tiene los ojos " ?ojos1 crlf)
  (printout t ?nombre2
              " tiene los ojos negros" crlf)
  (printout t "-----------------------" crlf))
```

Tabla de seguimiento en el ejemplo

- **La acción de presentar información en pantalla:**

  `(printout t <dato>*)`

- **Tabla de seguimiento:**

Hechos	E	Agenda	D
`f0 (initial-fact)`	0		
`f1 (persona (nombre Ana)`	0		
`(ojos verdes))`			
`f2 (persona (nombre Juan)`	0	`busca-personas: f1,f2`	4
`(ojos negros))`			
`f3 (persona (nombre Luis)`	0	`busca-personas: f1,f3`	3
`(ojos negros))`			
`f4 (persona (nombre Blanca)`	0	`busca-personas: f4,f2`	2
`(ojos azules))`		`busca-personas: f4,f3`	1

Sesión (I)

```
CLIPS> (clear)
CLIPS> (load "busca-personas.clp")
%$*
TRUE
CLIPS> (reset)
CLIPS> (facts)
f-0 (initial-fact)
f-1 (persona (nombre Ana) (ojos verdes))
f-2 (persona (nombre Juan) (ojos negros))
f-3 (persona (nombre Luis) (ojos negros))
f-4 (persona (nombre Blanca) (ojos azules))
For a total of 5 facts.
CLIPS> (agenda)
0      busca-personas: f-4,f-3
0      busca-personas: f-4,f-2
0      busca-personas: f-1,f-3
0      busca-personas: f-1,f-2
For a total of 4 activations.
```

Sesión (II)

```
CLIPS> (run)
Blanca tiene los ojos azules
Luis tiene los ojos negros
-----------------------
Blanca tiene los ojos azules
Juan tiene los ojos negros
-----------------------
Ana tiene los ojos verdes
Luis tiene los ojos negros
-----------------------
Ana tiene los ojos verdes
Juan tiene los ojos negros
-----------------------
```

Variables múltiples y mudas

- **Variables:** `$?x, $?y, $?gv32`
  - **Toman un valor múltiple**
- **Variables mudas: toman un valor que no es necesario recordar**
  - **Simple:** `?`
  - **Múltiple:** `$?`

Eliminaciones

- **Estructura de una regla (II):**

```
(defrule <nombre>
  <condicion>*
  =>
  <accion>*)

<condicion> := <hecho> |
               (not <hecho>) |
               <variable-simple> <- <hecho>
```

- **Acción: Eliminar hechos:**

```
(retract <identificador-hecho>*)

<identificador-hecho> := <variable-simple>
```

Ejemplo: unión de conjuntos (I)

```
(deffacts datos-iniciales
  (conjunto-1 a b)
  (conjunto-2 b c))

(defrule calcula-union
  =>
  (assert (union)))

(defrule union-base
  ?union <- (union $?u)
  ?conjunto-1 <- (conjunto-1 $?e-1)
  ?conjunto-2 <- (conjunto-2)
  =>
  (retract ?conjunto-1 ?conjunto-2 ?union)
  (assert (union ?e-1 ?u))
  (assert (escribe-solucion)))

(defrule escribe-solucion
  (escribe-solucion)
  (union $?u)
  =>
  (printout t "La union es " ?u crlf))
```

Ejemplo: unión de conjuntos (II)

```
(defrule union-con-primero-compartido
  (union $?)
  ?conjunto-2 <- (conjunto-2 ?e $?r-2)
  (conjunto-1 $? ?e $?)
  =>
  (retract ?conjunto-2)
  (assert (conjunto-2 ?r-2)))

(defrule union-con-primero-no-compartido
  ?union <- (union $?u)
  ?conjunto-2 <- (conjunto-2 ?e $?r-2)
  (not (conjunto-1 $? ?e $?))
  =>
  (retract ?conjunto-2 ?union)
  (assert (conjunto-2 ?r-2)
          (union ?u ?e)))
```

Tabla de seguimiento en el ejemplo

Hechos	E	S	Agenda	D
`f0 (initial-fact)`	0		`calcula-union: f0`	1
`f1 (conj-1 a b)`	0	4		
`f2 (conj-2 b c)`	0	2		
`f3 (union)`	1	3	`union-con-p-c: f3,f2,f1`	2
`f4 (conj-2 c)`	2	3	`union-con-p-no-c: f3,f4,`	3
`f5 (conj-2)`	3	4		
`f6 (union c)`	3	4	`union-base: f6,f1,f5`	4
`f7 (union a b c)`	4			
`f8 (escribe-solucion)`	4		`eescribe-solucion: f8,f7`	5

Sesión (I)

```
CLIPS> (load "union.clp")
$*****
TRUE
CLIPS> (watch facts)
CLIPS> (watch rules)
CLIPS> (reset)
==> f-0     (initial-fact)
==> f-1     (conjunto-1 a b)
==> f-2     (conjunto-2 b c)
CLIPS> (run)
FIRE    1 calcula-union: f-0
==> f-3     (union)
FIRE    2 union-con-primero-compartido: f-3,f-2,f-1
<== f-2     (conjunto-2 b c)
==> f-4     (conjunto-2 c)
FIRE    3 union-con-primero-no-compartido: f-3,f-4,
<== f-4     (conjunto-2 c)
<== f-3     (union)
==> f-5     (conjunto-2)
==> f-6     (union c)
```

Sesión (II)

```
FIRE    4 union-base: f-6,f-1,f-5
<== f-1     (conjunto-1 a b)
<== f-5     (conjunto-2)
<== f-6     (union c)
==> f-7     (union a b c)
==> f-8     (escribe-solucion)
FIRE    5 escribe-solucion: f-8,f-7
La union es (a b c)
```

Plantillas con campos múltiples

- **Estructura de una plantilla (II):**

```
(deftemplate <nombre>
  <campo>*)

<campo> := (slot <nombre-campo>) |
           (multislot <nombre-campo>)
```

Comprobaciones en las condiciones de una regla

- **Estructura de una regla (III):**

```
(defrule <nombre>
  <condicion>*
  =>
  <accion>*)

<condicion> := <hecho> |
               (not <hecho>) |
               <variable-simple> <- <hecho> |
               (test <llamada-a-una-funcion>)
```

- **Funciones matemáticas:**
  - **Básicas:** `+`, `-`, `*`, `/`
  - **Comparaciones:** `=`, `!=`, `<`, `<=`, `>`, `>=`
  - **Exponenciales:** `**`, `sqrt`, `exp`, `log`
  - **Trigonométricas:** `sin`, `cos`, `tan`

Ejemplo: busca triángulos rectángulos (I)

```
(deftemplate triangulo
  (slot nombre)
  (multislot lados))

(deffacts triangulos
  (triangulo (nombre A) (lados 3 4 5))
  (triangulo (nombre B) (lados 6 8 9))
  (triangulo (nombre C) (lados 6 8 10)))

(defrule inicio
  =>
  (assert (triangulos-rectangulos)))

(defrule almacena-triangulo-rectangulo
  ?h1 <- (triangulo (nombre ?n) (lados ?x ?y ?z))
  (test (= ?z (sqrt (+ (** ?x 2) (** ?y 2)))))
  ?h2 <- (triangulos-rectangulos $?a)
  =>
  (retract ?h1 ?h2)
  (assert (triangulos-rectangulos $?a ?n)))
```

Ejemplo: busca triángulos rectángulos (II)

```
(defrule elimina-triangulo-no-rectangulo
  ?h <- (triangulo (nombre ?n) (lados ?x ?y ?z))
  (test (!= ?z (sqrt (+ (** ?x 2) (** ?y 2)))))
  =>
  (retract ?h))

(defrule fin
  (not (triangulo))
  (triangulos-rectangulos $?a)
  =>
  (printout t "Lista de triangulos rectangulos: "
              $?a crlf))
```

Sesión (I)

```
CLIPS> (load "busca-triangulos-rect.clp")
%$****
TRUE
CLIPS> (watch facts)
CLIPS> (watch rules)
CLIPS> (reset)
==> f-0     (initial-fact)
==> f-1     (triangulo (nombre A) (lados 3 4 5))
==> f-2     (triangulo (nombre B) (lados 6 8 9))
==> f-3     (triangulo (nombre C) (lados 6 8 10))
CLIPS> (run)
FIRE    1 elimina-triangulo-no-rectangulo: f-2
<== f-2     (triangulo (nombre B) (lados 6 8 9))
FIRE    2 inicio: f-0
==> f-4     (triangulos-rectangulos)
```

Sesión (II)

```
FIRE    3 almacena-triangulo-rectangulo: f-1,f-4
<== f-1     (triangulo (nombre A) (lados 3 4 5))
<== f-4     (triangulos-rectangulos)
==> f-5     (triangulos-rectangulos A)
FIRE    4 almacena-triangulo-rectangulo: f-3,f-5
<== f-3     (triangulo (nombre C) (lados 6 8 10))
<== f-5     (triangulos-rectangulos A)
==> f-6     (triangulos-rectangulos A C)
FIRE    5 fin: f-0,,f-6
Lista de triangulos rectangulos: (A C)
```

Tabla de seguimiento

Hechos	E	S	Agenda	D	S
`f0 (initial-fact)`	0		`inicio: f0`	2	
`f1 (tri (nombre A)`	0	3			
`(lados 3 4 5))`					
`f2 (tri (nombre B)`	0	1			
`(lados 6 8 9))`			`elimina-tri-no-rect: f2`	1	
`f3 (tri (nombre C)`	0	4			
`(lados 6 8 10))`					
`f4 (tri-rect)`	2	3	`almacena-tri-rect: f3,f4`	-	3
			`almacena-tri-rect: f1,f4`	3	
`f5 (tri-rect A)`	3	4	`almacena-tri-rect: f3,f5`	4	
`f6 (tri-rect A C)`	4		`fin: f0,,f6`	5	

Ejemplo de no terminación: suma áreas rectángulos

```
(deftemplate rectangulo
  (slot nombre)
  (slot base)
  (slot altura))

(deffacts informacion-inicial
  (rectangulo (nombre A) (base 9) (altura 6))
  (rectangulo (nombre B) (base 7) (altura 5))
  (rectangulo (nombre C) (base 6) (altura 8))
  (rectangulo (nombre D) (base 2) (altura 5))
  (suma 0))

(defrule suma-areas-de-rectangulos
  (rectangulo (base ?base) (altura ?altura))
  ?suma <- (suma ?total)
  =>
  (retract ?suma)
  (assert (suma (+ ?total (* ?base ?altura)))))
```

Sesión

```
CLIPS> (clear)
CLIPS> (load "suma-areas-1.clp")
%$*
TRUE
CLIPS> (reset)
==> f-0     (initial-fact)
==> f-1     (rectangulo (nombre A) (base 9) (altura 6))
==> f-2     (rectangulo (nombre B) (base 7) (altura 5))
==> f-3     (rectangulo (nombre C) (base 6) (altura 8))
==> f-4     (rectangulo (nombre D) (base 2) (altura 5))
==> f-5     (suma 0)
CLIPS> (run)
FIRE    1 suma-areas-de-rectangulos: f-1,f-5
<== f-5     (suma 0)
==> f-6     (suma 54)
FIRE    2 suma-areas-de-rectangulos: f-1,f-6
<== f-6     (suma 54)
==> f-7     (suma 108)
FIRE    3 suma-areas-de-rectangulos: f-1,f-7
<== f-7     (suma 108)
==> f-8     (suma 162)
.......
```

Consiguiendo la terminación en la suma áreas rectángulos

```
(deftemplate rectangulo
  (slot nombre)
  (slot base)
  (slot altura))

(deffacts informacion-inicial
  (rectangulo (nombre A) (base 9) (altura 6))
  (rectangulo (nombre B) (base 7) (altura 5))
  (rectangulo (nombre C) (base 6) (altura 9))
  (rectangulo (nombre D) (base 2) (altura 5)))
```

Consiguiendo la terminación en la suma áreas rectángulos

```
(defrule inicio
  =>
  (assert (suma 0)))

(defrule areas
  (rectangulo (nombre ?n) (base ?b) (altura ?h))
  =>
  (assert (area-a-sumar ?n (* ?b ?h))))

(defrule suma-areas-de-rectangulos
  ?nueva-area <- (area-a-sumar ? ?area)
  ?suma <- (suma ?total)
  =>
  (retract ?suma ?nueva-area)
  (assert (suma (+ ?total ?area))))

(defrule fin
  (not (area-a-sumar ? ?))
  (suma ?total)
  =>
  (printout t "La suma es " ?total crlf))
```

Sesión (I)

```
CLIPS> (load "suma-areas-2.clp")
%$****
TRUE
CLIPS> (reset)
CLIPS> (run)
La suma es 153
CLIPS> (watch facts)
CLIPS> (watch rules)
CLIPS> (reset)
==> f-0     (initial-fact)
==> f-1     (rectangulo (nombre A) (base 9) (altura 6))
==> f-2     (rectangulo (nombre B) (base 7) (altura 5))
==> f-3     (rectangulo (nombre C) (base 6) (altura 9))
==> f-4     (rectangulo (nombre D) (base 2) (altura 5))
```

Sesión (II)

```
CLIPS> (run)
FIRE    1 areas: f-4
==> f-5     (area-a-sumar D 10)
FIRE    2 areas: f-3
==> f-6     (area-a-sumar C 54)
FIRE    3 areas: f-2
==> f-7     (area-a-sumar B 35)
FIRE    4 areas: f-1
==> f-8     (area-a-sumar A 54)
FIRE    5 inicio: f-0
==> f-9     (suma 0)
FIRE    6 suma-areas-de-rectangulos: f-5,f-9
<== f-9     (suma 0)
<== f-5     (area-a-sumar D 10)
==> f-10    (suma 10)
```

Sesión (III)

```
FIRE    7 suma-areas-de-rectangulos: f-6,f-10
<== f-10    (suma 10)
<== f-6     (area-a-sumar C 54)
==> f-11    (suma 64)
FIRE    8 suma-areas-de-rectangulos: f-7,f-11
<== f-11    (suma 64)
<== f-7     (area-a-sumar B 35)
==> f-12    (suma 99)
FIRE    9 suma-areas-de-rectangulos: f-8,f-12
<== f-12    (suma 99)
<== f-8     (area-a-sumar A 54)
==> f-13    (suma 153)
FIRE   10 fin: f-0,,f-13
La suma es 153
```

Restricciones evaluables

- **Restricciones (II):**
  - **Evaluables:**

```
(dato ?x&:<llamada-a-un-predicado>)
```

- **Ejemplo: dada una lista de números obtener la lista ordenada de menor a mayor.**
  - **Sesión**

```
CLIPS> (assert (vector 3 2 1 4))
La ordenacion de (3 2 1 4) es (1 2 3 4)
```

Ejemplo: ordenación de listas numéricas

```
(defrule inicial
  (vector $?x)
  =>
  (assert (vector-aux ?x)))

(defrule ordena
  ?f <- (vector-aux $?b ?m1 ?m2&:(< ?m2 ?m1) $?e)
  =>
  (retract ?f)
  (assert (vector-aux $?b ?m2 ?m1 $?e)))

(defrule final
  (not (vector-aux $?b ?m1 ?m2&:(< ?m2 ?m1) $?e))
  (vector $?x)
  (vector-aux $?y)
  =>
  (printout t "La ordenacion de " ?x " es " ?y crlf))
```

Sesión (I)

```
CLIPS> (load "ordenacion.clp")
***
TRUE
CLIPS> (watch facts)
CLIPS> (watch rules)
CLIPS> (reset)
==> f-0     (initial-fact)
CLIPS> (assert (vector 3 2 1 4))
==> f-1     (vector 3 2 1 4)
==> Activation 0      inicial: f-1
<Fact-1>
```

Sesión (II)

```
CLIPS> (run)
FIRE    1 inicial: f-1
==> f-2     (vector-aux 3 2 1 4)
FIRE    2 ordena: f-2
<== f-2     (vector-aux 3 2 1 4)
==> f-3     (vector-aux 2 3 1 4)
FIRE    3 ordena: f-3
<== f-3     (vector-aux 2 3 1 4)
==> f-4     (vector-aux 2 1 3 4)
FIRE    4 ordena: f-4
<== f-4     (vector-aux 2 1 3 4)
==> f-5     (vector-aux 1 2 3 4)
FIRE    5 final: f-0,,f-1,f-5
La ordenacion de (3 2 1 4) es (1 2 3 4)
```

Tabla de seguimiento:

Hechos	E	S	Agenda	D	S
`f0 (initial-fact)`	0				
`f1 (vector 3 2 1 4)`	0		`inicial: f1`	1	
`f2 (vector-aux 3 2 1 4)`	1	2	`ordena: f2`	2	
			`ordena: f2`	-	2
`f3 (vector-aux 2 3 1 4)`	2	3	`ordena: f3`	3	
`f4 (vector-aux 2 1 3 4)`	3	4	`ordena: f4`	4	
`f5 (vector-aux 1 2 3 4)`	4		`final: f0,,f1,f5`	5	

Ejemplo: cálculo del máximo de una lista numérica

```
(defrule maximo
  (vector $? ?x $?)
  (not (vector $? ?y&:(> ?y ?x) $?))
  =>
  (printout t "El maximo es " ?x crlf))
```

Sesión

```
CLIPS> (load "maximo.clp")
*
TRUE
CLIPS> (watch facts)
CLIPS> (watch rules)
CLIPS> (reset)
==> f-0     (initial-fact)
CLIPS> (assert (vector 3 2 1 4))
==> f-1     (vector 3 2 1 4)
<Fact-1>
CLIPS> (run)
FIRE    1 maximo: f-1,
El maximo es 4
CLIPS> (assert (vector 3 2 1 4 2 3))
==> f-2     (vector 3 2 1 4 2 3)
<Fact-2>
CLIPS> (run)
FIRE    1 maximo: f-2,
El maximo es 4
```

Restricciones y funciones

- **Funciones:**

```
(deffunction <nombre>
  (<argumento>*)
  <accion>*)
```

- **Ejemplo: problema de cuadrados mágicos**
  - **Enunciado**

```
ABC         {A,B,C,D,E,F,G,H,I} = {1,2,3,4,5,6,7,8,9}
DEF         A+B+C = D+E+F = G+H+I = A+D+G = B+E+F
GHI               = C+F+I = A+E+I = C+E+G
```

  - **Sesión**

```
CLIPS> (run)
Solucion 1:
   492
   357
   816

....
```

Ejemplo: cuadrados mágicos (I)

```
(deffacts datos
  (numero 1) (numero 2) (numero 3) (numero 4)
  (numero 5) (numero 6) (numero 7) (numero 8)
  (numero 9) (solucion 0))

(deffunction suma-15 (?x ?y ?z)
  (= (+ ?x ?y ?z) 15))
```

Ejemplo: cuadrados mágicos (II)

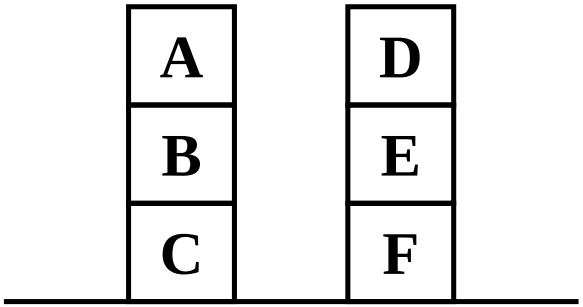
```
(defrule busca-cuadrado
  (numero ?e)
  (numero ?a&~?e)
  (numero ?i&~?e&~?a&:(suma-15 ?a ?e ?i))
  (numero ?b&~?e&~?a&~?i)
  (numero ?c&~?e&~?a&~?i&~?b&:(suma-15 ?a ?b ?c))
  (numero ?f&~?e&~?a&~?i&~?b&~?c&:(suma-15 ?c ?f ?i))
  (numero ?d&~?e&~?a&~?i&~?b&~?c&~?f
          &:(suma-15 ?d ?e ?f))
  (numero ?g&~?e&~?a&~?i&~?b&~?c&~?f&~?d
          &:(suma-15 ?a ?d ?g)&:(suma-15 ?c ?e ?g))
  (numero ?h&~?e&~?a&~?i&~?b&~?c&~?f&~?d&~?g
          &:(suma-15 ?b ?e ?h)&:(suma-15 ?g ?h ?i))
  =>
  (assert (escribe-solucion ?a ?b ?c ?d ?e
                            ?f ?g ?h ?i)))
```

Ejemplo: cuadrados mágicos (III)

```
(defrule escribe-solucion
  ?f <- (escribe-solucion ?a ?b ?c
                          ?d ?e ?f
                          ?g ?h ?i)
  ?solucion <- (solucion ?n)
  =>
  (retract ?f ?solucion)
  (assert (solucion (+ ?n 1)))
  (printout t "Solucion " (+ ?n 1) ":" crlf)
  (printout t "   " ?a ?b ?c crlf)
  (printout t "   " ?d ?e ?f crlf)
  (printout t "   " ?g ?h ?i crlf)
  (printout t crlf))
```

Ejemplo: mundo de los bloques

- **Enunciado**



  - **Objetivo: Poner `C` encima de `E`**

- **Representación**

```
(deffacts estado-inicial
  (pila A B C)
  (pila D E F)
  (objetivo C esta-encima-del E))
```

Mundo de los bloques (I)

```
;;; REGLA: mover-bloque-sobre-bloque
;;; SI
;;;   el objetivo es poner el bloque X encima del
;;;     bloque Y y
;;;   no hay nada encima del bloque X ni del bloque Y
;;; ENTONCES
;;;   colocamos el bloque X encima del bloque Y y
;;;   actualizamos los datos.

(defrule mover-bloque-sobre-bloque
  ?obj <- (objetivo ?blq-1 esta-encima-del ?blq-2)
  ?p-1 <- (pila ?blq-1 $?resto-1)
  ?p-2 <- (pila ?blq-2 $?resto-2)
  =>
  (retract ?ob ?p1 ?p2)
  (assert (pila $?resto-1))
  (assert (pila ?blq-1 ?blq-2 $?resto-2))
  (printout t ?blq-1 " movido encima del "
              ?blq-2 crlf))
```

Mundo de los bloques (II)

```
;;; REGLA: mover-bloque-al-suelo
;;; SI
;;;   el objetivo es mover el bloque X al suelo y
;;;   no hay nada encima de X
;;; ENTONCES
;;;   movemos el bloque X al suelo y
;;;   actualizamos los datos.

(defrule mover-bloque-al-suelo
  ?obj <- (objetivo ?blq-1 esta-encima-del suelo)
  ?p-1 <- (pila ?blq-1 $?resto)
  =>
  (retract ?objetivo ?pila-1)
  (assert (pila ?blq-1))
  (assert (pila $?resto))
  (printout t ?blq-1 " movido encima del suelo."
              crlf))
```

Mundo de los bloques (III)

```
;;; REGLA: libera-bloque-movible
;;; SI
;;;   el objetivo es poner el bloque X encima de Y
;;;     (bloque o suelo) y
;;;   X es un bloque y
;;;   hay un bloque encima del bloque X
;;; ENTONCES
;;;   hay que poner el bloque que está encima de X
;;;     en el suelo.

(defrule liberar-bloque-movible
  (objetivo ?bloque esta-encima-del ?)
  (pila ?cima $? ?bloque $?)
  =>
  (assert (objetivo ?cima esta-encima-del suelo)))
```

Mundo de los bloques (IV)

```
;;; REGLA: libera-bloque-soporte
;;; SI
;;;   el objetivo es poner el bloque X (bloque o
;;;     nada) encima de Y e
;;;   hay un bloque encima del bloque Y
;;; ENTONCES
;;;   hay que poner el bloque que está encima de Y
;;;     en el suelo.

(defrule liberar-bloque-soporte
  (objetivo ? esta-encima-del ?bloque)
  (pila ?cima $? ?bloque $?)
  =>
  (assert (objetivo ?cima esta-encima-del suelo)))
```

Sesión (I)

```
CLIPS> (clear)
CLIPS> (unwatch all)
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (watch rules)
CLIPS> (load "bloques.clp")
$****
TRUE
CLIPS> (reset)
==> f-0     (initial-fact)
==> f-1     (pila A B C)
==> f-2     (pila D E F)
==> f-3     (objetivo C esta-encima-del E)
==> Activation 0      liberar-bloque-soporte: f-3,f-2
==> Activation 0      liberar-bloque-movible: f-3,f-1
```

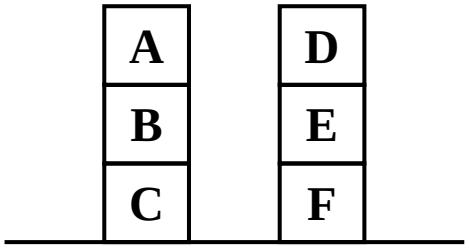
Sesión (II)

```
CLIPS> (run)
FIRE    1 liberar-bloque-movible: f-3,f-1
==> f-4     (objetivo A esta-encima-del suelo)
==> Activation 0      mover-bloque-al-suelo: f-4,f-1
FIRE    2 mover-bloque-al-suelo: f-4,f-1
<== f-4     (objetivo A esta-encima-del suelo)
<== f-1     (pila A B C)
==> f-5     (pila A)
==> f-6     (pila B C)
==> Activation 0      liberar-bloque-movible: f-3,f-6
A movido encima del suelo.
FIRE    3 liberar-bloque-movible: f-3,f-6
==> f-7     (objetivo B esta-encima-del suelo)
==> Activation 0      mover-bloque-al-suelo: f-7,f-6
FIRE    4 mover-bloque-al-suelo: f-7,f-6
<== f-7     (objetivo B esta-encima-del suelo)
<== f-6     (pila B C)
==> f-8     (pila B)
==> f-9     (pila C)
```

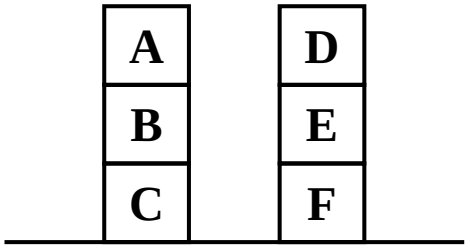
Sesión (III)

```
B movido encima del suelo.
FIRE    5 liberar-bloque-soporte: f-3,f-2
==> f-10    (objetivo D esta-encima-del suelo)
==> Activation 0      mover-bloque-al-suelo: f-10,f-2
FIRE    6 mover-bloque-al-suelo: f-10,f-2
<== f-10    (objetivo D esta-encima-del suelo)
<== f-2     (pila D E F)
==> f-11    (pila D)
==> f-12    (pila E F)
==> Activation 0
              mover-bloque-sobre-bloque: f-3,f-9,f-12
D movido encima del suelo.
FIRE    7 mover-bloque-sobre-bloque: f-3,f-9,f-12
<== f-3     (objetivo C esta-encima-del E)
<== f-9     (pila C)
<== f-12    (pila E F)
==> f-13    (pila)
==> f-14    (pila C E F)
C movido encima del E
CLIPS>
```

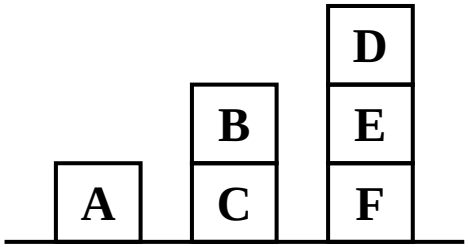
Mundo de los bloques



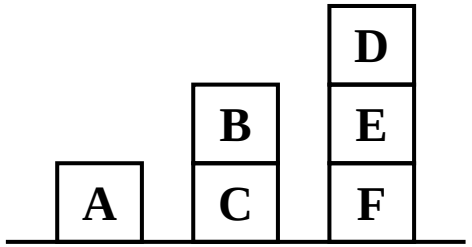
Objetivos: C/E
Agenda: Lib C
Lib E



Objetivos: A/Suelo
C/E
Agenda: Mover A
Lib E

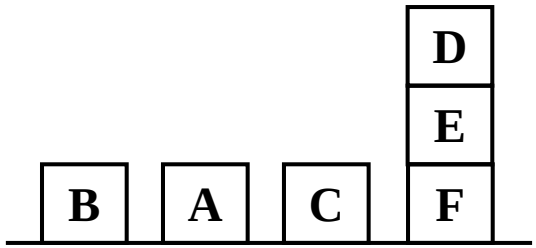


Objetivos: C/E
Agenda: Lib C
Lib E

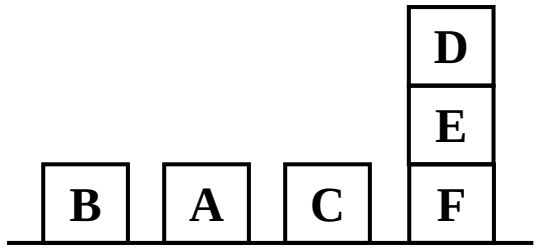


Objetivos: B/Suelo
C/E
Agenda: Mover B
Lib E

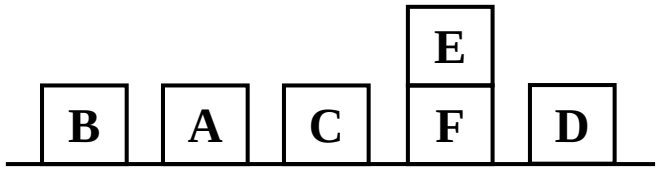
Mundo de los bloques



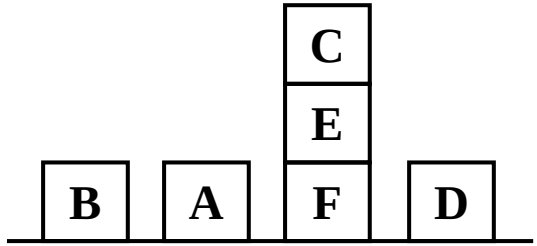
Objetivos: C/E
Agenda: Lib E



Objetivos: D/Suelo
C/E
Agenda: Mover D



Objetivos: C/E
Agenda: Mover C



Elementos condicionales

- **Reglas disyuntivas:**

```
(defrule no-hay-clase-1
  (festivo hoy)
  =>
  (printout t "Hoy no hay clase" crlf))

(defrule no-hay-clase-2
  (sabado hoy)
  =>
  (printout t "Hoy no hay clase" crlf))

(defrule no-hay-clase-3
  (hay-examen hoy)
  =>
  (printout t "Hoy no hay clase" crlf))

(deffacts inicio
  (sabado hoy)
  (hay-examen hoy))
```

Elementos condicionales

- **Reglas disyuntivas. Sesión:**

```
CLIPS> (clear)
CLIPS> (unwatch all)
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (watch rules)
CLIPS> (load "ej-1.clp")
***$
TRUE
CLIPS> (reset)
==> f-0     (initial-fact)
==> f-1     (sabado hoy)
==> Activation 0      no-hay-clase-2: f-1
==> f-2     (hay-examen hoy)
==> Activation 0      no-hay-clase-3: f-2
CLIPS> (run)
FIRE    1 no-hay-clase-3: f-2
Hoy no hay clase
FIRE    2 no-hay-clase-2: f-1
Hoy no hay clase
```

Disyunción

- **Elementos condicionales disyuntivos:**

```
(defrule no-hay-clase
  (or (festivo hoy)
      (sabado hoy)
      (hay-examen hoy))
  =>
  (printout t "Hoy no hay clase" crlf))

(deffacts inicio
  (sabado hoy)
  (hay-examen hoy))
```

Disyunción

- **Sesión**

```
CLIPS> (clear)
....
CLIPS> (reset)
==> f-0     (initial-fact)
==> f-1     (sabado hoy)
==> Activation 0      no-hay-clase: f-1
==> f-2     (hay-examen hoy)
==> Activation 0      no-hay-clase: f-2
CLIPS> (run)
FIRE    1 no-hay-clase: f-2
Hoy no hay clase
FIRE    2 no-hay-clase: f-1
Hoy no hay clase
CLIPS>
```

Limitación de disparos disyuntivos

- **Ejemplo:**

```
(defrule no-hay-clase
  ?periodo <- (periodo lectivo)
  (or (festivo hoy)
      (sabado hoy)
      (hay-examen hoy))
  =>
  (retract ?periodo)
  (assert (periodo lectivo-sin-clase))
  (printout t "Hoy no hay clase" crlf))

(deffacts inicio
  (sabado hoy)
  (hay-examen hoy))
```

Limitación de disparos disyuntivos (sesión)

```
CLIPS> (clear)
....
TRUE
CLIPS> (reset)
==> f-0     (initial-fact)
==> f-1     (sabado hoy)
==> f-2     (hay-examen hoy)
CLIPS> (assert (periodo lectivo))
==> f-3     (periodo lectivo)
==> Activation 0      no-hay-clase: f-3,f-2
==> Activation 0      no-hay-clase: f-3,f-1
CLIPS> (run)
FIRE    1 no-hay-clase: f-3,f-1
<== f-3     (periodo lectivo)
<== Activation 0      no-hay-clase: f-3,f-2
==> f-4     (periodo lectivo-sin-clase)
Hoy no hay clase
CLIPS>
```

Programa equivalente sin disyunciones (I)

```
(defrule no-hay-clase-1
  ?periodo <- (periodo lectivo)
  (festivo hoy)
  =>
  (retract ?periodo)
  (assert (periodo lectivo-sin-clase))
  (printout t "Hoy no hay clase" crlf))

(defrule no-hay-clase-2
  ?periodo <- (periodo lectivo)
  (sabado hoy)
  =>
  (retract ?periodo)
  (assert (periodo lectivo-sin-clase))
  (printout t "Hoy no hay clase" crlf))
```

Programa equivalente sin disyunciones (II)

```
(defrule no-hay-clase-3
  ?periodo <- (periodo lectivo)
  (hay-examen hoy)
  =>
  (retract ?periodo)
  (assert (periodo lectivo-sin-clase))
  (printout t "Hoy no hay clase" crlf))

(deffacts inicio
  (sabado hoy)
  (hay-examen hoy))
```

Eliminación de causas disyuntivas

- **Ejemplo**

```
(defrule no-hay-clase
  ?periodo <- (periodo lectivo)
  (or ?causa <- (festivo hoy)
      ?causa <- (sabado hoy)
      ?causa <- (hay-examen hoy))
  =>
  (retract ?periodo ?causa)
  (assert (periodo lectivo-sin-clase))
  (printout t "Hoy no hay clase" crlf))

(deffacts inicio
  (sabado hoy)
  (hay-examen hoy)
  (periodo lectivo))
```

Sesión

```
CLIPS> (clear)
....
CLIPS> (reset)
==> f-0     (initial-fact)
==> f-1     (sabado hoy)
==> f-2     (hay-examen hoy)
==> f-3     (periodo lectivo)
==> Activation 0      no-hay-clase: f-3,f-2
==> Activation 0      no-hay-clase: f-3,f-1
CLIPS> (run)
<== f-3     (periodo lectivo)
<== Activation 0      no-hay-clase: f-3,f-2
<== f-1     (sabado hoy)
==> f-4     (periodo lectivo-sin-clase)
Hoy no hay clase
CLIPS> (facts)
f-0     (initial-fact)
f-2     (hay-examen hoy)
f-4     (periodo lectivo-sin-clase)
For a total of 3 facts.
CLIPS>
```

Conjunción

- **Conjunciones y disyunciones**

```
(defrule no-hay-clase
  ?periodo <- (periodo lectivo)
  (or (festivo hoy)
      (sabado hoy)
      (and (festivo ayer)
           (festivo manana)))
  =>
  (retract ?periodo)
  (assert (periodo lectivo-sin-clase))
  (printout t "Hoy no hay clase" crlf))

(deffacts inicio
  (periodo lectivo)
  (festivo ayer)
  (festivo manana))
```

Sesión

```
CLIPS> (clear)
CLIPS> (unwatch all)
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (load "ej-6.clp")
*$
TRUE
CLIPS> (reset)
==> f-0     (initial-fact)
==> f-1     (periodo lectivo)
==> f-2     (festivo ayer)
==> f-3     (festivo manana)
==> Activation 0      no-hay-clase: f-1,f-2,f-3
CLIPS> (run)
<== f-1     (periodo lectivo)
==> f-4     (periodo lectivo-sin-clase)
Hoy no hay clase
CLIPS>
```

Conjunción: reglas equivalentes (I)

```
(defrule no-hay-clase-1
  ?periodo <- (periodo lectivo)
  (festivo hoy)
  =>
  (retract ?periodo)
  (assert (periodo lectivo-sin-clase))
  (printout t "Hoy no hay clase" crlf))

(defrule no-hay-clase-2
  ?periodo <- (periodo lectivo)
  (sabado hoy)
  =>
  (retract ?periodo)
  (assert (periodo lectivo-sin-clase))
  (printout t "Hoy no hay clase" crlf))
```

Conjunción: reglas equivalentes (II)

```
(defrule no-hay-clase-3
  ?periodo <- (periodo lectivo)
  (festivo ayer)
  (festivo manana)
  =>
  (retract ?periodo)
  (assert (periodo lectivo-sin-clase))
  (printout t "Hoy no hay clase" crlf))
```

Bibliografía

- Giarratano, J.C. y Riley, G. *Sistemas expertos: Principios y programación (3 ed.)* (International Thomson Editores, 2001)
  - Cap. 7: "Introducción a CLIPS"
  - Cap. 8: "Comparación de patrones"
  - Cap. 9: "Comparación avanzada de patrones"
  - Apéndice E: "Resumen de comandos y funciones de CLIPS"
- Giarrantano, J.C. y Riley, G. *Expert Systems Principles and Programming (3 ed.)* (PWS Pub. Co., 1998).
- Giarratano, J.C. *CLIPS User's Guide (Version 6.20, March 31st 2001)*
- Kowalski, T.J. y Levy, L.S. *Rule–Based Programming* (Kluwer Academic Publisher, 1996)