Tema 1: Programación basada en reglas con CLIPS

José A. Alonso Jiménez
Francisco J. Martín Mateos
José L. Ruiz Reina

Dpto. de Ciencias de la Computación e Inteligencia Artificial

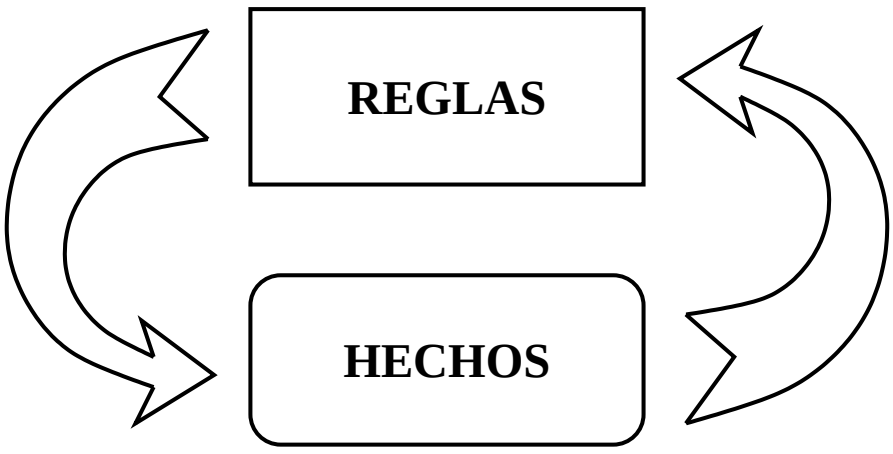
Universidad de Sevilla

Programación basada en reglas

- **Modelo de regla:**

  `<Condiciones> => <Acciones>`

- **Condiciones**
  - **Existencia de cierta información**
  - **Ausencia de cierta información**
  - **Relaciones entre datos**
- **Acciones**
  - **Incluir nueva información**
  - **Eliminar información**
  - **Presentar información en pantalla**



Definición de hechos y reglas

- **Estructura de un hecho**

  `(<simbolo> <datos>*)`

- **Ejemplos:**
  - `(conjunto a b 1 2 3)`
  - `(1 2 3 4)` **no es un hecho válido**
- **Acción: Añadir hechos**

  `(assert <hecho>*)`

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

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

<condicion> := <hecho>
```

- **Ejemplo:**

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

- **Hechos iniciales:**

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

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)`

- **Inicialización:**

  `(reset)`

- **Ejecución:**

  `(run)`

- **Limpiar la base de conocimiento:**

  `(clear)`

- **Ayuda del sistema:**

  `(help)`

Definición de hechos y reglas

- BC `animales.clp`

```
(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)))

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

Definición de hechos y reglas

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

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

- **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)`			

Definición de hechos y reglas

- **Sesión**

```
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, variables y restriciones

- **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 (I): Condiciones sobre las variables que se comprueban en el momento de verificar las condiciones de una regla**
  - **Negativas:**

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

  - **Disyuntivas:**

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

  - **Conjuntivas:**

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

Plantillas, variables y restricciones

- **Acción: Presentar información en pantalla**

```
(printout t <dato>*)
```

- **BC** `busca-personas.clp`

```
(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))
```

Plantillas, variables y restricciones

- **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

Plantillas, variables y restricciones

- **Sesión**

```
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.
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 eliminaciones

- **Variables:** `$?x, $?y, $?gv32`
  - **Toman un valor múltiple**
- **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>
```

- **Variables mudas: Toman un valor que no es necesario recordar**
  - **Simple:** `?`
  - **Múltiple:** `$?`

Variables múltiples y eliminaciones

- **Unión de conjuntos**
  - **BC** `union.clp`

```
(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))
```

Variables múltiples y eliminaciones

```
(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)))
```

Variables múltiples y eliminaciones

- **Tabla de seguimiento:**

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)`		4	`union-base: f6,f1,f5`	4
`f7 (union a b c)`	4			
`f8 (escribe-solucion)`	4		`eescribe-solucion: f8,f7`	5

Variables múltiples y eliminaciones

- **Sesión**

```
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)
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)
```

Variables múltiples y eliminaciones

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

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

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

- **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`

Variables múltiples y eliminaciones

- **Busca triángulos rectángulos**
  - **BC** `busca-triangulos-rect.clp`

```
(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)))

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

Variables múltiples y eliminaciones

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

- **Sesión**

```
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)
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)
```

Variables múltiples y eliminaciones

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)`	1	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	
			`fin: f0,,f5`	-	4
`f6 (tri-rect A C)`	4		`fin: f0,,f6`	5	

Ejemplo de no terminación

- **Suma de áreas de rectángulos**
  - **BC** `suma-areas-1.clp`

```
(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)))))
```

Ejemplo de no terminación

- **Sesión**

```
CLIPS> (clear)
CLIPS> (load "suma-areas-1.clp")
%$*
TRUE
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 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)
FIRE    4 suma-areas-de-rectangulos: f-1,f-8
<== f-8     (suma 162)
==> f-9     (suma 216)
...
```

Ejemplo de no terminación

- **BC** `suma-areas-2.clp`

```
(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)))

(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))))
```

Ejemplo de no terminación

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

- **Sesión**

```
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))
```

Ejemplo de no terminación

```
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)
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>)
```

- **Ordenación: 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)
```

  - **BC** `ordenacion.clp`

```
(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))
```

Restricciones evaluables

- **Traza**

```
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>
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)
```

Restricciones evaluables

- **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	

Restricciones evaluables

- **Máximo: Dada una lista de números determinar el máximo.**
  - **BC** `maximo.clp`

```
(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>*)
```

- **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

....
```

  - **Programa** `cuadrado-magico.clp`**:**

```
(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))
```

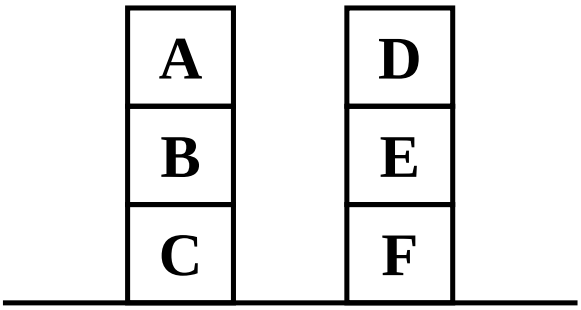
Restricciones y funciones

```
(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)))

(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))
```

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

- **Reglas:**

```
;;; 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

- **Reglas:**

```
;;; 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

- **Reglas:**

```
;;; 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)))

;;; 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)))
```

Mundo de los bloques

- **Sesión:**

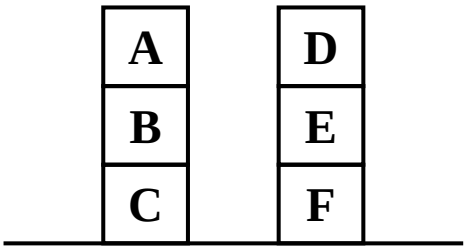
```
CLIPS> (clear)
CLIPS> (unwatch all)
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (watch rules)
CLIPS> (load "bloques.clp")
$****
TRUE
CLIPS> (reset)
==> f-0     (initial-fact)
==> f-1     (pila A B C)
==> f-2     (pila D E F)
==> f-3     (objetivo C esta-encima-del E)
==> Activation 0 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

Mundo de los bloques

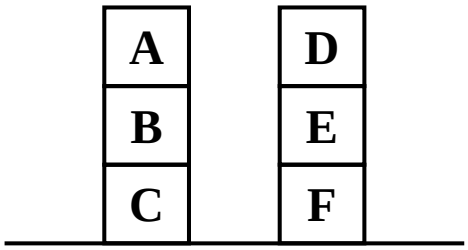
- **Sesión:**

```
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)
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>
```

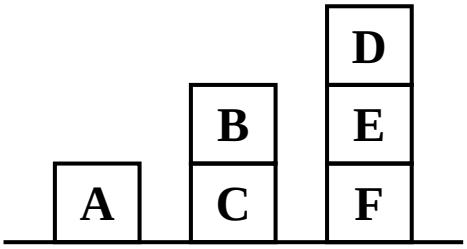
Mundo de los bloques



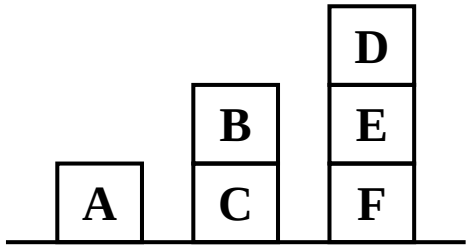
Objetivos: C/E
Agenda: Lib C
Lib E



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

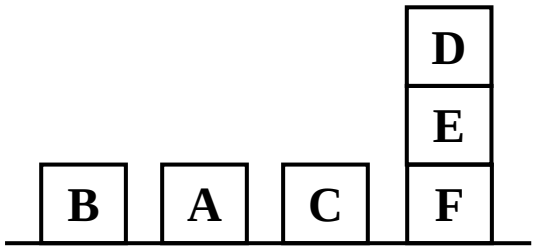


Objetivos: C/E
Agenda: Lib C
Lib E

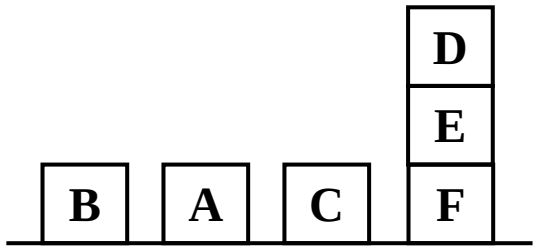


Objetivos: B/Suelo
C/E
Agenda: Mover B
Lib E

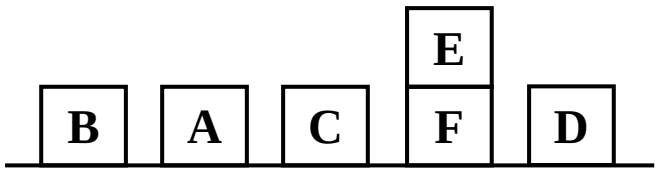
Mundo de los bloques



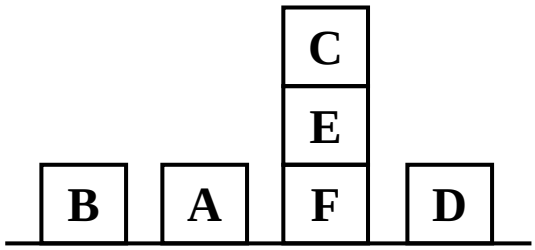
Objetivos: C/E
Agenda: Lib E



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



Objetivos: C/E
Agenda: Mover C



Elementos condicionales

- **Reglas disyuntivas:** `ej-1.clp`

```
(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:** `ej-2.cl`

```
(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> (unwatch all)
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (watch rules)
CLIPS> (load "ej-2.clp")
*$
TRUE
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:** `ej-3.clp`

```
(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)
CLIPS> (unwatch all)
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (watch rules)
CLIPS> (load "ej-3.clp")
*$
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>
```

Limitación de disparos disyuntivos

- **Programa equivalente sin elementos condicionales disyuntivos:** `ej-4.clp`

```
(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))

(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:** `ej-5.clp`

```
(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))
```

Eliminación de causas disyuntivas

- **Sesión**

```
CLIPS> (clear)
CLIPS> (unwatch all)
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (load "ej-5.clp")
*$
TRUE
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:** `ej-6.clp`

```
(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))
```

Conjunción

- **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**

```
(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))

(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))
```

Lectura de datos y variables globales

- **Lectura de datos:**

```
(read)

(readline)
```

- **Variables globales:**

```
(defglobal
  ?*<simbolo>* = <valor>)
```

- **Ejemplo: Adivina el número**

```
CLIPS> (reset)
CLIPS> (run)
Escribe un numero: 3
3 es bajo
Escribe un numero: 9
9 es alto
Escribe un numero: 7
7 es correcto
CLIPS>
```

- **Código: Adivina el número**

```
(defglobal
  ?*numero* = 7)

(defrule juego
  =>
  (assert (lee)))
```

Lectura de datos y variables globales

- **Ejemplo: Adivina el número (I)**

```
(defrule lee
  ?h <- (lee)
  =>
  (retract ?h)
  (printout t "Escribe un numero: ")
  (assert (numero (read))))

(defrule bajo
  ?h <- (numero ?n&:(< ?n ?*numero*))
  =>
  (retract ?h)
  (printout t ?n " es bajo" crlf)
  (assert (lee)))

(defrule alto
  ?h <- (numero ?n&:(> ?n ?*numero*))
  =>
  (retract ?h)
  (printout t ?n " es alto" crlf)
  (assert (lee)))

(defrule exacto
  ?h <- (numero ?n&:(= ?n ?*numero*))
  =>
  (retract ?h)
  (printout t ?n " es correcto" crlf))
```

Lectura de hechos como cadenas

- **Sesión**

```
CLIPS> (defrule inserta-hecho
         =>
         (printout t "Escribe un hecho como cadena"
                     crlf)
         (assert-string (read)))
CLIPS> (reset)
CLIPS> (run)
Escribe un hecho como cadena
"(color verde)"
CLIPS> (facts)
f-0     (initial-fact)
f-1     (color verde)
For a total of 2 facts.
CLIPS>
```

- **Añadir un hecho expresado como una cadena:**
  - `(assert-string <cadena>)`

Lectura de líneas

- **Sesión**

```
CLIPS> (defrule lee-linea
         =>
         (printout t "Introduce datos." crlf)
         (bind ?cadena (readline))
         (assert-string (str-cat "(" ?cadena ")")))
CLIPS> (reset)
CLIPS> (run)
Introduce datos.
colores verde azul ambar rojo
CLIPS> (facts)
f-0     (initial-fact)
f-1     (colores verde azul ambar rojo)
For a total of 2 facts.
CLIPS>
```

- **Concatenación:**
  - `(str-cat <cadena>*)`

Acciones procedimentales: `if` y `bind`

- **Condicional:**

```
(if <condicion>
  then <accion>*
  [else <accion>*])
```

- **Asignacion:**

```
(bind <variable> <valor>)
```

- **Ejemplo: Adivina el número (II)**

```
(defrule lee
  ?h <- (lee)
  =>
  (retract ?h)
  (printout t "Escribe un numero: ")
  (bind ?n (read))
  (if (not (numberp ?n))
    then
      (printout t "Eso no es un numero." crlf)
      (assert (lee))
    else
      (assert (numero ?n))))
```

Acciones procedimentales: `while`

- **Bucle:**

```
(while <condicion> do
  <accion>*)
```

- **Ejemplo: Adivina el número (III)**

```
(defrule lee
  ?h <- (lee)
  =>
  (retract ?h)
  (printout t "Escribe un numero: ")
  (bind ?n (read))
  (while (not (numberp ?n)) do
    (printout t "Eso no es un numero." crlf)
    (printout t "Escribe un numero: ")
    (bind ?n (read)))
  (assert (numero ?n)))
```

Nim

- **Sesión:**

```
CLIPS> (clear)
CLIPS> (unwatch all)
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (watch rules)
CLIPS> (load "nim-1.clp")
$*****$*
TRUE
CLIPS> (reset)
<== f-0     (initial-fact)
==> f-0     (initial-fact)
==> f-1     (turno h)
==> f-2     (numero-de-piezas 11)
==> Activation 0      eleccion-humana: f-1,f-2
==> f-3     (computadora-coge 1 cuando-el-resto-es 1)
==> f-4     (computadora-coge 1 cuando-el-resto-es 2)
==> f-5     (computadora-coge 2 cuando-el-resto-es 3)
==> f-6     (computadora-coge 3 cuando-el-resto-es 0)
CLIPS> (run)
FIRE    1 eleccion-humana: f-1,f-2
<== f-1     (turno h)
Quedan 11 pieza(s)
Cuantas piezas coges: 2
```

Nim

- **Sesión:**

```
==> f-7     (eleccion-humana 2)
==> Activation 0  correcta-eleccion-humana: f-7,f-2
FIRE    2 correcta-eleccion-humana: f-7,f-2
<== f-7     (eleccion-humana 2)
<== f-2     (numero-de-piezas 11)
==> f-8     (numero-de-piezas 9)
==> f-9     (turno c)
==> Activation 0  eleccion-computadora: f-9,f-8,f-3
FIRE    3 eleccion-computadora: f-9,f-8,f-3
<== f-9     (turno c)
<== f-8     (numero-de-piezas 9)
Quedan 9 pieza(s)
La computadora coge 1 pieza(s)
==> f-10    (numero-de-piezas 8)
==> f-11    (turno h)
==> Activation 0      eleccion-humana: f-11,f-10
FIRE    4 eleccion-humana: f-11,f-10
<== f-11    (turno h)
Quedan 8 pieza(s)
Cuantas piezas coges: 3
```

Nim

- **Sesión:**

```
==> f-12    (eleccion-humana 3)
==> Activation 0  correcta-eleccion-humana: f-12,f-10
FIRE    5 correcta-eleccion-humana: f-12,f-10
<== f-12    (eleccion-humana 3)
<== f-10    (numero-de-piezas 8)
==> f-13    (numero-de-piezas 5)
==> f-14    (turno c)
==> Activation 0  eleccion-computadora: f-14,f-13,f-3
FIRE    6 eleccion-computadora: f-14,f-13,f-3
<== f-14    (turno c)
<== f-13    (numero-de-piezas 5)
Quedan 5 pieza(s)
La computadora coge 1 pieza(s)
==> f-15    (numero-de-piezas 4)
==> f-16    (turno h)
==> Activation 0      eleccion-humana: f-16,f-15
FIRE    7 eleccion-humana: f-16,f-15
<== f-16    (turno h)
Quedan 4 pieza(s)
Cuantas piezas coges: 3
```

Nim

- **Sesión:**

```
==> f-17    (eleccion-humana 3)
==> Activation 0  correcta-eleccion-humana: f-17,f-15
FIRE    8 correcta-eleccion-humana: f-17,f-15
<== f-17    (eleccion-humana 3)
<== f-15    (numero-de-piezas 4)
==> f-18    (numero-de-piezas 1)
==> f-19    (turno c)
==> Activation 0     pierde-la-computadora: f-19,f-18
FIRE    9 pierde-la-computadora: f-19,f-18
Queda 1 pieza
La computadora coge la ultima pieza
He perdido
CLIPS>
```

Nim

- **Código:**

```
(deffacts datos-iniciales
  (turno h)
  (numero-de-piezas 11))

(defrule pierde-la-computadora
  (turno c)
  (numero-de-piezas 1)
  =>
  (printout t "Queda 1 pieza" crlf)
  (printout t "La computadora coge la ultima pieza"
            crlf)
  (printout t "He perdido" crlf))

(deffacts heuristica
  (computadora-coge 1 cuando-el-resto-es 1)
  (computadora-coge 1 cuando-el-resto-es 2)
  (computadora-coge 2 cuando-el-resto-es 3)
  (computadora-coge 3 cuando-el-resto-es 0))

(defrule eleccion-computadora
  ?turno <- (turno c)
  ?pila <- (numero-de-piezas ?n&:(> ?n 1))
  (computadora-coge ?m cuando-el-resto-es
                    =(mod ?n 4))
  =>
  (retract ?turno ?pila)
  (printout t "Quedan " ?n " pieza(s)" crlf)
  (printout t "La computadora coge " ?m
              " pieza(s)" crlf)
  (assert (numero-de-piezas (- ?n ?m))
  (turno h)))
```

Nim

- **Código:**

```
(defrule pierde-el-humano
  (turno h)
  (numero-de-piezas 1)
  =>
  (printout t "Queda 1 pieza" crlf)
  (printout t "Tienes que coger la ultima pieza"
            crlf)
  (printout t "Has perdido" crlf))

(defrule eleccion-humana
  ?turno <- (turno h)
  ?pila <- (numero-de-piezas ?n&:(> ?n 1))
  =>
  (retract ?turno)
  (printout t "Quedan " ?n " pieza(s)" crlf)
  (printout t "Cuantas piezas coges: ")
  (assert (eleccion-humana (read))))
```

Nim

- **Código:**

```
(defrule incorrecta-eleccion-humana
  ?eleccion <- (eleccion-humana ?m)
  (numero-de-piezas ?n&:(> ?n 1))
  (test (not (and (integerp ?m)
                  (>= ?m 1)
                  (<= ?m 3)
                  (< ?m ?n))))
  =>
  (retract ?eleccion)
  (printout t "Tiene que elegir "
            "un numero entre 1 y 3" crlf)
  (assert (turno h)))

(defrule correcta-eleccion-humana
  ?eleccion <- (eleccion-humana ?m)
  ?pila <- (numero-de-piezas ?n&:(> ?n 1))
  (test (and (integerp ?m)
             (>= ?m 1)
             (<= ?m 3)
             (< ?m ?n)))
  =>
  (retract ?eleccion ?pila)
  (assert (numero-de-piezas (- ?n ?m))
  (turno c)))
```

Nim con `if`

- **Regla:** `eleccion-humana`

```
(defrule eleccion-humana
  ?turno <- (turno h)
  ?pila <- (numero-de-piezas ?n&:(> ?n 1))
  =>
  (retract ?turno)
  (printout t "Quedan " ?n " pieza(s)" crlf)
  (printout t "Cuantas piezas coges: ")
  (bind ?m (read))
  (if (and (integerp ?m)
           (>= ?m 1)
           (<= ?m 3)
           (< ?m ?n))
    then
      (bind ?nuevo (- ?n ?m))
      (retract ?pila)
      (assert (numero-de-piezas ?nuevo)
              (turno c))
    else
      (printout t "Tiene que elegir "
                "un numero entre 1 y 3" crlf)
      (assert (turno h))))
```

Nim con `while`

- **Regla:** `eleccion-humana`

```
(defrule eleccion-humana
  ?turno <- (turno h)
  ?pila <- (numero-de-piezas ?n&:(> ?n 1))
  =>
  (retract ?turno ?pila)
  (printout t "Quedan " ?n " pieza(s)" crlf)
  (printout t "Cuantas piezas coges: ")
  (bind ?m (read))
  (while (not (and (integerp ?m)
                   (>= ?m 1)
                   (<= ?m 3)
                   (< ?m ?n))) do
    (printout t "Tiene que elegir "
              "un numero entre 1 y 3" crlf)
    (printout t "Quedan " ?n " pieza(s)" crlf)
    (printout t "Cuantas piezas coges: ")
    (bind ?m (read)))
  (assert (numero-de-piezas (- ?n ?m))
  (turno c)))
```

Nim con funciones definidas

- **Regla:** `eleccion-humana`

```
(deffunction piezas-cogidas-de (?m ?n)
  (while (not (and (integerp ?m)
                   (>= ?m 1)
                   (<= ?m 3)
                   (< ?m ?n))) do
    (printout t "Tiene que elegir "
              "un numero entre 1 y 3" crlf)
    (printout t "Quedan " ?n " pieza(s)" crlf)
    (printout t "Cuantas piezas coges: ")
    (bind ?m (read)))
  ?m)

(defrule eleccion-humana
  ?turno <- (turno h)
  ?pila <- (numero-de-piezas ?n&:(> ?n 1))
  =>
  (retract ?turno ?pila)
  (printout t "Quedan " ?n " pieza(s)" crlf)
  (printout t "Cuantas piezas coges: ")
  (bind ?m (piezas-cogidas-de (read) ?n))
  (assert (numero-de-piezas (- ?n ?m))
          (turno c)))
```

Nim con acciones definidas

- **Regla:** `eleccion-humana`

```
(deffunction coge-piezas (?n)
  (printout t "Quedan " ?n " pieza(s)" crlf)
  (printout t "Cuantas piezas coges: ")
  (bind ?m (read))
  (while (not (and (integerp ?m)
                   (>= ?m 1)
                   (<= ?m 3)
                   (< ?m ?n))) do
    (printout t "Tiene que elegir "
              "un numero entre 1 y 3" crlf)
    (printout t "Cuantas piezas coges: ")
    (bind ?m (read)))
  (assert (numero-de-piezas (- ?n ?m))
  (turno c)))

(defrule eleccion-humana
  ?turno <- (turno h)
  ?pila <- (numero-de-piezas ?n&:(> ?n 1))
  =>
  (retract ?turno ?pila)
  (coge-piezas ?n))
```

Nim con fases

- **Sesión**

```
CLIPS> (load "nim.clp")
CLIPS> $***********$*
TRUE
CLIPS> (reset)
CLIPS> (run)
Elige quien empieza: computadora o Humano (c/h) c
Escribe el numero de piezas: 15
La computadora coge 2 pieza(s)
Quedan 13 pieza(s)
Escribe el numero de piezas que coges: 3
Quedan 10 pieza(s)
La computadora coge 1 pieza(s)
Quedan 9 pieza(s)
Escribe el numero de piezas que coges: 2
Quedan 7 pieza(s)
La computadora coge 2 pieza(s)
Quedan 5 pieza(s)
Escribe el numero de piezas que coges: 4
Tiene que elegir un numero entre 1 y 3
Escribe el numero de piezas que coges: 1
Quedan 4 pieza(s)
La computadora coge 3 pieza(s)
Quedan 1 pieza(s)
Tienes que coger la ultima pieza
Has perdido
CLIPS>
```

Nim con fases

- **Elección del jugador**

```
(deffacts fase-inicial
  (fase elige-jugador))

(defrule elige-jugador
  (fase elige-jugador)
  =>
  (printout t "Elige quien empieza: ")
  (printout t "computadora o Humano (c/h) ")
  (assert (jugador-elegido (read))))

(defrule correcta-eleccion-de-jugador
  ?fase <- (fase elige-jugador)
  ?eleccion <- (jugador-elegido ?jugador&c|h)
  =>
  (retract ?fase ?eleccion)
  (assert (turno ?jugador))
  (assert (fase elige-numero-de-piezas)))

(defrule incorrecta-eleccion-de-jugador
  ?fase <- (fase elige-jugador)
  ?eleccion <- (jugador-elegido ?jugador&~c&~h)
  =>
  (retract ?fase ?eleccion)
  (printout t ?jugador " es distinto de c y h" crlf)
  (assert (fase elige-jugador)))
```

Nim con fases

- **Elección del número de piezas**

```
(defrule elige-numero-de-piezas
  (fase elige-numero-de-piezas)
  =>
  (printout t "Escribe el numero de piezas: ")
  (assert (numero-de-piezas (read))))

(defrule correcta-eleccion-del-numero-de-piezas
  ?fase <- (fase elige-numero-de-piezas)
  ?eleccion <- (numero-de-piezas ?n&:(integerp ?n)
                                 &:(> ?n 0))
  =>
  (retract ?fase ?eleccion)
  (assert (numero-de-piezas ?n)))

(defrule incorrecta-eleccion-del-numero-de-piezas
  ?fase <- (fase elige-numero-de-piezas)
  ?eleccion <- (numero-de-piezas ?n&~:(integerp ?n)
                                 |:(<= ?n 0))
  =>
  (retract ?fase ?eleccion)
  (printout t ?n " no es un numero entero mayor que 0"
            crlf)
  (assert (fase elige-numero-de-piezas)))
```

Nim con fases

- **Jugada humana**

```
(defrule pierde-el-humano
  (turno h)
  (numero-de-piezas 1)
  =>
  (printout t "Tienes que coger la ultima pieza" crlf)
  (printout t "Has perdido" crlf))

(defrule eleccion-humana
  (turno h)
  (numero-de-piezas ?n&:(> ?n 1))
  =>
  (printout t "Escribe el numero de piezas que coges: ")
  (assert (piezas-cogidas (read))))
```

Nim con fases

```
(defrule correcta-eleccion-humana
  ?pila <- (numero-de-piezas ?n)
  ?eleccion <- (piezas-cogidas ?m)
  ?turno <- (turno h)
  (test (and (integerp ?m)
             (>= ?m 1)
             (<= ?m 3)
             (< ?m ?n)))
  =>
  (retract ?pila ?eleccion ?turno)
  (bind ?nuevo-numero-de-piezas (- ?n ?m))
  (assert (numero-de-piezas ?nuevo-numero-de-piezas))
  (printout t "Quedan "
            ?nuevo-numero-de-piezas " pieza(s)" crlf)
  (assert (turno c)))

(defrule incorrecta-eleccion-humana
  (numero-de-piezas ?n)
  ?eleccion <- (piezas-cogidas ?m)
  ?turno <- (turno h)
  (test (or (not (integerp ?m))
            (< ?m 1)
            (> ?m 3)
            (>= ?m ?n)))
  =>
  (retract ?eleccion ?turno)
  (printout t "Tiene que elegir un numero entre 1 y 3"
            crlf)
  (assert (turno h)))
```

Nim con fases

- **Jugada de la computadora**

```
(defrule pierde-la-computadora
  (turno c)
  (numero-de-piezas 1)
  =>
  (printout t "La computadora coge la ultima pieza"
            crlf "He perdido" crlf))

(deffacts heuristica
  (computadora-coge 1 cuando-el-resto-es 1)
  (computadora-coge 1 cuando-el-resto-es 2)
  (computadora-coge 2 cuando-el-resto-es 3)
  (computadora-coge 3 cuando-el-resto-es 0))

(defrule eleccion-computadora
  ?turno <- (turno c)
  ?pila <- (numero-de-piezas ?n&:(> ?n 1))
  (computadora-coge ?m cuando-el-resto-es =(mod ?n 4))
  =>
  (retract ?turno ?pila)
  (printout t "La computadora coge " ?m " pieza(s)"
            crlf)
  (bind ?nuevo-numero-de-piezas (- ?n ?m))
  (printout t "Quedan " ?nuevo-numero-de-piezas
            " pieza(s)" crlf)
  (assert (numero-de-piezas ?nuevo-numero-de-piezas))
  (assert (turno h)))
```

Control empotrado en las reglas

- **Fases en el Nim:**
  - **Elección del jugador.**
  - **Elección del número de piezas.**
  - **Turno del humano.**
  - **Turno de la computadora.**
- **Hechos de control:**
  - `(fase elige-jugador)`
  - `(fase elige-numero-de-piezas)`
  - `(turno h)`
  - `(turno c)`
- **Inconvenientes del control empotrado en las reglas:**
  - **Dificultad para entenderse.**
  - **Dificultad para precisar la conclusión de cada fase.**

Técnicas de control

- **Ejemplo de fases de un problema:**
  - **Detección.**
  - **Aislamiento.**
  - **Recuperación.**
- **Técnicas de control:**
  - **Control empotrado en las reglas.**
  - **Prioridades.**
  - **Reglas de control.**
  - **Módulos.**

Prioridades

- **Ejemplo:** `ej-1.clp`

```
(deffacts inicio
  (prioridad primera)
  (prioridad segunda)
  (prioridad tercera))

(defrule regla-1
  (prioridad primera)
  =>
  (printout t "Escribe primera" crlf))

(defrule regla-2
  (prioridad segunda)
  =>
  (printout t "Escribe segunda" crlf))

(defrule regla-3
  (prioridad tercera)
  =>
  (printout t "Escribe tercera" crlf))
```

Prioridades

- **Sesión**

```
CLISP> (load "ej-1.clp")
CLIPS> $***
TRUE
CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (prioridad primera)
f-2     (prioridad segunda)
f-3     (prioridad tercera)
For a total of 4 facts.
CLIPS> (rules)
regla-1
regla-2
regla-3
For a total of 3 defrules.
CLIPS> (agenda)
0      regla-3: f-3
0      regla-2: f-2
0      regla-1: f-1
For a total of 3 activations.
CLIPS> (run)
Escribe tercera
Escribe segunda
Escribe primera
CLIPS>
```

Prioridades

- **Ejemplo:** `ej-2.clp`

```
(deffacts inicio
  (prioridad primera)
  (prioridad segunda)
  (prioridad tercera))

(defrule regla-1
  (declare (salience 30))
  (prioridad primera)
  =>
  (printout t "Escribe primera" crlf))

(defrule regla-2
  (declare (salience 20))
  (prioridad segunda)
  =>
  (printout t "Escribe segunda" crlf))

(defrule regla-3
  (declare (salience 10))
  (prioridad tercera)
  =>
  (printout t "Escribe tercera" crlf))
```

Prioridades

- **Sesión**

```
CLIPS> (load "ej-2.clp")
CLIPS> $***
TRUE
CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (prioridad primera)
f-2     (prioridad segunda)
f-3     (prioridad tercera)
For a total of 4 facts.
CLIPS> (rules)
regla-1
regla-2
regla-3
For a total of 3 defrules.
CLIPS> (agenda)
30     regla-1: f-1
20     regla-2: f-2
10     regla-3: f-3
For a total of 3 activations.
CLIPS> (run)
Escribe primera
Escribe segunda
Escribe tercera
CLIPS>
```

Prioridades

- **Sintaxis:**
  - `(declare (salience <numero>))`
- **Valores:**
  - **Mínimo: -10000**
  - **Máximo: 10000**
  - **Defecto: 0**
- **Inconvenientes:**
  - **Abuso.**
  - **Contradicción con el objetivo de los sistemas basados en reglas.**

Reglas de control

- **Reglas de cambio de fases** `ej-3.clp`

```
(defrule deteccion-a-aislamiento
  (declare (salience -10))
  ?fase <- (fase deteccion)
  =>
  (retract ?fase)
  (assert (fase aislamiento)))

(defrule aislamiento-a-recuperacion
  (declare (salience -10))
  ?fase <- (fase aislamiento)
  =>
  (retract ?fase)
  (assert (fase recuperacion)))

(defrule recuperacion-a-deteccion
  (declare (salience -10))
  ?fase <- (fase recuperacion)
  =>
  (retract ?fase)
  (assert (fase deteccion)))

(defrule detecta-fuego
  (fase deteccion)
  (luz-a roja)
  =>
  (assert (problema fuego)))

(deffacts inicio
  (fase deteccion) (luz-a roja))
```

Reglas de control

- **Sesión**

```
CLIPS> (load "ej-3.clp")
CLIPS> ****$
TRUE
CLIPS> (watch rules)
CLIPS> (reset)
CLIPS> (run 7)
FIRE    1 detecta-fuego: f-1,f-2
FIRE    2 deteccion-a-aislamiento: f-1
FIRE    3 aislamiento-a-recuperacion: f-4
FIRE    4 recuperacion-a-deteccion: f-5
FIRE    5 detecta-fuego: f-6,f-2
FIRE    6 deteccion-a-aislamiento: f-6
FIRE    7 aislamiento-a-recuperacion: f-7
CLIPS>
```

Reglas de control

- **Cambio de fase mediante una regla** `ej-4.clp`

```
(deffacts control
  (fase deteccion)
  (siguiente-fase deteccion aislamiento)
  (siguiente-fase aislamiento recuperacion)
  (siguiente-fase recuperacion deteccion))

(defrule cambio-de-fase
  (declare (salience -10))
  ?fase <- (fase ?actual)
  (siguiente-fase ?actual ?siguiente)
  =>
  (retract ?fase)
  (assert (fase ?siguiente)))

(defrule detecta-fuego
  (fase deteccion)
  (luz-a roja)
  =>
  (assert (problema fuego)))

(deffacts inicio
  (fase deteccion)
  (luz-a roja))
```

Reglas de control

```
CLIPS> (load "ej-4.clp")
$**$
TRUE
CLIPS> (watch rules)
CLIPS> (watch facts)
CLIPS> (reset)
CLIPS> (reset)
==> f-0     (initial-fact)
==> f-1     (fase deteccion)
==> f-2     (siguiente-fase deteccion aislamiento)
==> f-3     (siguiente-fase aislamiento recuperacion)
==> f-4     (siguiente-fase recuperacion deteccion)
==> f-5     (luz-a roja)
CLIPS> (run 5)
FIRE    1 detecta-fuego: f-1,f-5
==> f-6     (problema fuego)
FIRE    2 cambio-de-fase: f-1,f-2
<== f-1     (fase deteccion)
==> f-7     (fase aislamiento)
FIRE    3 cambio-de-fase: f-7,f-3
<== f-7     (fase aislamiento)
==> f-8     (fase recuperacion)
FIRE    4 cambio-de-fase: f-8,f-4
<== f-8     (fase recuperacion)
==> f-9     (fase deteccion)
FIRE    5 detecta-fuego: f-9,f-5
```

Reglas de control

- **Cambio de fase mediante una regla y sucesión de fases** `ej-5.clp`

```
(deffacts control
  (fase deteccion)
  (sucesion-de-fases aislamiento
                     recuperacion
                     deteccion))

(defrule cambio-de-fase
  (declare (salience -10))
  ?fase <- (fase ?actual)
  (sucesion-de-fases ?siguiente $?resto)
  =>
  (retract ?fase)
  (assert (fase ?siguiente))
  (assert (sucesion-de-fases ?resto ?siguiente)))
```

Reglas de control

```
CLIPS> (clear)
CLIPS> (load "ej-5.clp")
$**$
TRUE
CLIPS> (watch facts)
CLIPS> (watch rules)
CLIPS> (reset)
==> f-0     (initial-fact)
==> f-1     (fase deteccion)
==> f-2     (sucesion-de-fases
             aislamiento recuperacion deteccion)
==> f-3     (luz-a roja)
CLIPS> (run 5)
FIRE    1 detecta-fuego: f-1,f-3
==> f-4     (problema fuego)
FIRE    2 cambio-de-fase: f-1,f-2
<== f-1     (fase deteccion)
==> f-5     (fase aislamiento)
==> f-6     (sucesion-de-fases
             recuperacion deteccion aislamiento)
FIRE    3 cambio-de-fase: f-5,f-6
<== f-5     (fase aislamiento)
==> f-7     (fase recuperacion)
==> f-8     (sucesion-de-fases
             deteccion aislamiento recuperacion)
FIRE    4 cambio-de-fase: f-7,f-8
<== f-7     (fase recuperacion)
==> f-9     (fase deteccion)
FIRE    5 detecta-fuego: f-9,f-3
```

Nim con módulos

- **Módulo** `MAIN`

```
(defmodule MAIN
  (export deftemplate numero-de-piezas
                      turno
                      initial-fact))

(defrule MAIN::inicio
  =>
  (focus INICIO))

(defrule MAIN::la-computadora-elige
  ?turno <- (turno c)
  (numero-de-piezas ~0)
  =>
  (focus COMPUTADORA)
  (retract ?turno)
  (assert (turno h)))

(defrule MAIN::el-humano-elige
  ?turno <- (turno h)
  (numero-de-piezas ~0)
  =>
  (focus HUMANO)
  (retract ?turno)
  (assert (turno c)))
```

Nim con módulos

- **Módulo** `INICIO`

```
(defmodule INICIO
  (import MAIN deftemplate numero-de-piezas
                           turno
                           initial-fact))

(defrule INICIO::elige-jugador
  (not (turno ?))
  =>
  (printout t "Elige quien empieza: "
              "computadora o Humano (c/h) ")
  (assert (turno (read))))

(defrule INICIO::incorrecta-eleccion-de-jugador
  ?eleccion <- (turno ?jugador&~c&~h)
  =>
  (retract ?eleccion)
  (printout t ?jugador " es distinto de c y h" crlf))

(defrule INICIO::elige-numero-de-piezas
  (not (numero-de-piezas-elegidas ?))
  =>
  (printout t "Escribe el numero de piezas: ")
  (assert (numero-de-piezas-elegidas (read))))
```

Nim con módulos

```
(defrule INICIO::incorrecta-eleccion-del-numero-de-piezas
  ?e <- (numero-de-piezas-elegidas ?n&~:(integerp ?n)
                                        |:(<= ?n 0))
  =>
  (retract ?e)
  (printout t ?n " no es un numero entero mayor que 0"
              crlf))

(defrule INICIO::correcta-eleccion-del-numero-de-piezas
  (numero-de-piezas-elegidas ?n&:(integerp ?n)
                                 &:(> ?n 0))
  =>
  (assert (numero-de-piezas ?n))
  (return))
```

- **Módulo** HUMANO

```
(defmodule HUMANO
  (import MAIN deftemplate numero-de-piezas))

(defrule HUMANO::pierde-el-humano
  ?h <- (numero-de-piezas 1)
  =>
  (printout t "Tienes que coger la ultima pieza" crlf)
  (printout t "Has perdido" crlf)
  (retract ?h)
  (assert (numero-de-piezas 0)))
```

Nim con módulos

```
(defrule HUMANO::eleccion-humana
  (numero-de-piezas ?n&:(> ?n 1))
  (not (piezas-cogidas ?))
  =>
  (printout t "Escribe el numero de piezas que coges: ")
  (assert (piezas-cogidas (read))))

(defrule HUMANO::correcta-eleccion-humana
  ?pila <- (numero-de-piezas ?n)
  ?eleccion <- (piezas-cogidas ?m)
  (test (and (integerp ?m)
             (>= ?m 1)
             (<= ?m 3)
             (< ?m ?n)))
  =>
  (retract ?pila ?eleccion)
  (bind ?nuevo-numero-de-piezas (- ?n ?m))
  (assert (numero-de-piezas ?nuevo-numero-de-piezas))
  (printout t "Quedan " ?nuevo-numero-de-piezas
              " pieza(s)" crlf)
  (return))

(defrule HUMANO::incorrecta-eleccion-humana
  (numero-de-piezas ?n)
  ?eleccion <- (piezas-cogidas ?m)
  (test (or (not (integerp ?m))
            (< ?m 1)
            (> ?m 3)
            (>= ?m ?n)))
  =>
  (printout t "Tiene que elegir un numero entre 1 y 3"
            crlf)
  (retract ?eleccion))
```

Nim con módulos

- **Módulo** `COMPUTADORA`

```
(defmodule COMPUTADORA
  (import MAIN deftemplate numero-de-piezas))

(defrule COMPUTADORA::pierde-la-computadora
  ?h <- (numero-de-piezas 1)
  =>
  (printout t "La computadora coge la ultima pieza" crlf)
  (printout t "He perdido" crlf)
  (retract ?h)
  (assert (numero-de-piezas 0)))

(deffacts heuristica
  (computadora-coge 1 cuando-el-resto-es 1)
  (computadora-coge 1 cuando-el-resto-es 2)
  (computadora-coge 2 cuando-el-resto-es 3)
  (computadora-coge 3 cuando-el-resto-es 0))

(defrule COMPUTADORA::eleccion-computadora
  ?pila <- (numero-de-piezas ?n&:(> ?n 1))
  (computadora-coge ?m cuando-el-resto-es =(mod ?n 4))
  =>
  (retract ?pila)
  (printout t "La computadora coge " ?m " pieza(s)" crlf)
  (bind ?nuevo-numero-de-piezas (- ?n ?m))
  (printout t "Quedan " ?nuevo-numero-de-piezas
              " pieza(s)" crlf)
  (assert (numero-de-piezas ?nuevo-numero-de-piezas))
  (return))
```

Nim con módulos

- **Sesión con traza:**

```
CLIPS> (watch facts)
CLIPS> (watch rules)
CLIPS> (watch focus)
CLIPS> (reset)
==> Focus MAIN
==> f-0     (initial-fact)
==> f-1     (computadora-coge 1 cuando-el-resto-es 1)
==> f-2     (computadora-coge 1 cuando-el-resto-es 2)
==> f-3     (computadora-coge 2 cuando-el-resto-es 3)
==> f-4     (computadora-coge 3 cuando-el-resto-es 0)
CLIPS> (run)
FIRE    1 inicio: f-0
==> Focus INICIO from MAIN
FIRE    2 elige-jugador: f-0,
Elige quien empieza: computadora o Humano (c/h) a
==> f-5     (turno a)
FIRE    3 incorrecta-eleccion-de-jugador: f-5
<== f-5     (turno a)
a es distinto de c y h
FIRE    4 elige-jugador: f-0,
Elige quien empieza: computadora o Humano (c/h) c
==> f-6     (turno c)
FIRE    5 elige-numero-de-piezas: f-0,
Escribe el numero de piezas: a
==> f-7     (numero-de-piezas-elegidas a)
FIRE    6 incorrecta-eleccion-del-numero-de-piezas: f-7
<== f-7     (numero-de-piezas-elegidas a)
a no es un numero entero mayor que 0
```

Nim con módulos

```
FIRE    7 elige-numero-de-piezas: f-0,
Escribe el numero de piezas: 5
==> f-8     (numero-de-piezas-elegidas 5)
FIRE    8 correcta-eleccion-del-numero-de-piezas: f-8
==> f-9     (numero-de-piezas 5)
<== Focus INICIO to MAIN
FIRE    9 la-computadora-elige: f-6,f-9
==> Focus COMPUTADORA from MAIN
<== f-6     (turno c)
==> f-10    (turno h)
FIRE   10 eleccion-computadora: f-9,f-1
<== f-9     (numero-de-piezas 5)
La computadora coge 1 pieza(s)
Quedan 4 pieza(s)
==> f-11    (numero-de-piezas 4)
<== Focus COMPUTADORA to MAIN
FIRE   11 el-humano-elige: f-10,f-11
==> Focus HUMANO from MAIN
<== f-10    (turno h)
==> f-12    (turno c)
FIRE   12 eleccion-humana: f-11,
Escribe el numero de piezas que coges: 4
==> f-13    (piezas-cogidas 4)
FIRE   13 incorrecta-eleccion-humana: f-11,f-13
Tiene que elegir un numero entre 1 y 3
<== f-13    (piezas-cogidas 4)
FIRE   14 eleccion-humana: f-11,
Escribe el numero de piezas que coges: 1
==> f-14    (piezas-cogidas 1)
```

Nim con módulos

```
FIRE   15 correcta-eleccion-humana: f-11,f-14
<== f-11    (numero-de-piezas 4)
<== f-14    (piezas-cogidas 1)
==> f-15    (numero-de-piezas 3)
Quedan 3 pieza(s)
<== Focus HUMANO to MAIN
FIRE   16 la-computadora-elige: f-12,f-15
==> Focus COMPUTADORA from MAIN
<== f-12    (turno c)
==> f-16    (turno h)
FIRE   17 eleccion-computadora: f-15,f-3
<== f-15    (numero-de-piezas 3)
La computadora coge 2 pieza(s)
Quedan 1 pieza(s)
==> f-17    (numero-de-piezas 1)
<== Focus COMPUTADORA to MAIN
FIRE   18 el-humano-elige: f-16,f-17
==> Focus HUMANO from MAIN
<== f-16    (turno h)
==> f-18    (turno c)
FIRE   19 pierde-el-humano: f-17
Tienes que coger la ultima pieza
Has perdido
<== f-17    (numero-de-piezas 1)
==> f-19    (numero-de-piezas 0)
<== Focus HUMANO to MAIN
<== Focus MAIN
```

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"
  - Cap. 10: "Diseño modular y control de la ejecución"
  - Cap. 11: "Eficiencia de los lenguajes basados en reglas"
  - Cap. 12: "Ejemplos de diseño de sistemas expertos"
  - 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)