

# Tema 7: Introducción a la programación basada en reglas

José A. Alonso Jiménez  
Miguel A. Gutiérrez Naranjo  
Francisco J. Martín Mateos

Dpto. de Ciencias de la Computación e Inteligencia Artificial  
UNIVERSIDAD DE SEVILLA

# Contenido y bibliografía

- Programación basada en reglas
  - Definición de hechos y reglas
  - Plantillas, variables y restricciones
  - Variables múltiples y eliminaciones
  - Ejemplo de no terminación
  - Restricciones evaluables
  - Restricciones y funciones
- Bibliografía
  - Giarratano, J.C. y Riley, G. “Expert Systems Principles and Programming (2nd ed.)” (PWS Pub. Co., 1994)
    - \* Cap. 7: “Introduction to Clips”
  - Giarratano, J.C. “CLIPS 6.0 User’s Guide”

# 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:**

```
(defacts <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-pezognas)
  (tiene-rayas-negras))
```

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

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

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

```
(defrule unguado-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-pezuñas)	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      ungalado-1: f-4,f-2
FIRE      2 ungalado-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 restricciones

- 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))
```

```
(def facts 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) (ojos verdes))	0		
f2 (persona (nombre Juan) (ojos negros))	0	busca-personas: f1, f2	4
f3 (persona (nombre Luis) (ojos negros))	0	busca-personas: f1, f3	3
f4 (persona (nombre Blanca) (ojos azules))	0	busca-personas: f4, f2	2
		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

```
(defacts 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
- Trigonómicas: 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) (lados 3 4 5))	0	3			
f2 (tri (nombre B) (lados 6 8 9))	0	1			
f3 (tri (nombre C) (lados 6 8 10))	0	4	elimina-tri-no-rect: f2	1	
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 ordena: f2	2 -	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:**

```
(defacts 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))
```