Sistemas de producción: CLIPS

Francisco J. Martín Mateos

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

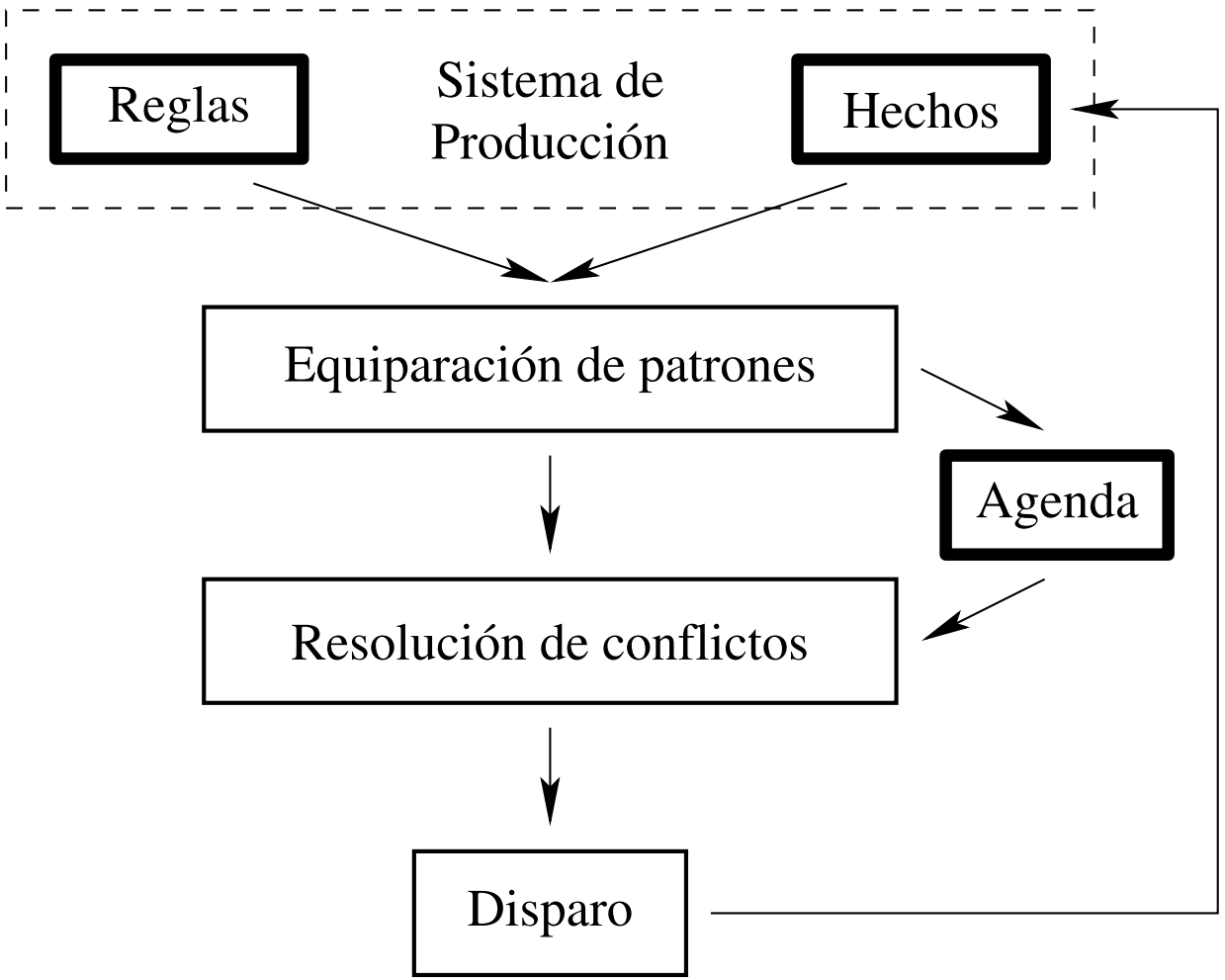
Sistemas de producción

- Reglas para identificar un animal:
  - Si el animal tiene pelos entonces es mamífero.
  - Si el animal produce leche entonces es mamífero.
  - Si el animal es mamífero y tiene pezuñas entonces es ungulado.
  - Si el animal es mamífero y rumia entonces es ungulado.
  - Si el animal es ungulado y tiene el cuello largo entonces es una jirafa.
  - Si el animal es ungulado y tiene rayas negras entonces es una cebra.
- ¿Cómo identificamos un animal que tiene pelos, pezuñas y rayas negras?

Sistemas de producción

- Un *sistema de producción* es un mecanismo computacional basado en *reglas de producción* de la forma: "Si se cumplen las *condiciones* entonces se ejecutan las *acciones*."
- El conjunto de las reglas de producción forma la *base de conocimiento* que describe como evoluciona un sistema.
  - Las reglas de producción actúan sobre una *memoria de trabajo* o *base de datos* que describe el estado actual del sistema.
  - Si la condición de una regla de producción se satisface entonces dicha regla está *activa*.
  - El conjunto de reglas de producción activas en un instante concreto forma el *conjunto de conflicto* o *agenda*.
  - La *estrategia de resolución* selecciona una regla del conjunto de conflicto para ser ejecutada o *disparada*, modificando así la memoria de trabajo.

Sistemas de producción



Sistemas de producción

- Componentes:
  - Base de hechos (*memoria de trabajo*): Elemento dinámico.
  - Base de reglas (*base de conocimiento*): Elemento estático.
  - Motor de inferencia: Produce los cambios en la memoria de trabajo.
- Elementos adicionales:
  - *Algoritmo de equiparación de patrones*: Algoritmo para calcular **eficientemente** la agenda.
  - *Estrategia de resolución de conflictos*: Proceso para decidir en cada momento qué regla de la agenda debe ser disparada.

Estrategia de resolución de conflictos

- Una activación sólo se produce una vez.
- Estrategias más comunes:
  - Tratar la agenda como una pila.
  - Tratar la agenda como una cola.
  - Elección aleatoria.
  - Regla más específica (número de condiciones).
  - Activación más reciente (en función de los hechos).
  - Regla menos utilizada.
  - Mejor (pesos).

Reglas de producción

- 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.

CLIPS

- CLIPS ≡ C Language Integrated Production Systems.
  - `http://www.clipsrules.net/`
  - Versión 6.31
- Lenguaje basado en reglas de producción.
- Desarrollado en el *Johnson Space Center* de la NASA.
- Relacionado con OPS5 y ART.
- Características:
  - Conocimiento: reglas, objetos y procedimental.
  - Portabilidad: implementado en C.
  - Integración y Extensibilidad: C, Java, FORTRAN, ADA.
  - Documentación.
  - Bajo coste: software libre.

Hechos en CLIPS

- Estructura de un hecho simple:
  `(<simbolo> <datos>*)`
- Ejemplos:
  - `(conjunto A 1 2 3)`
  - `(1 2 3 4)` no es un hecho válido.
- Conjunto de hechos iniciales:

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

- Ejemplo:

```
(deffacts datos-iniciales
  (conjunto A 1 2 3 4)
  (conjunto B 1 3 5))
```

Reglas en CLIPS

- Estructura de una regla (I):

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

- Las condiciones son patrones que se equiparan con los hechos de la memoria de trabajo.
- Acción: Añadir hechos.

```
(assert <hecho>*)
```

- Ejemplo:

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

Interacción con CLIPS

- Limpiar la base de conocimiento: `(clear)`
- Cargar el contenido de un archivo: `(load <archivo>)`
- Inicializar el sistema de producción: `(reset)`
- Visualizar la memoria de trabajo: `(facts)`
- Visualizar la agenda: `(agenda)`
- Ejecutar el sistema de producción: `(run)`
  - Ejecutar una regla en el sistema de producción: `(run 1)`
- Salir del sistema: `(exit)`

Ejemplo

- Reglas:

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

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

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

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

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

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

- Conjunto de hechos iniciales:

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

Seguimiento de la ejecución

- Visualizar las entradas y salidas de hechos:

```
(watch facts)
```

- Visualizar las activaciones y desactivaciones de las reglas:

```
(watch activations)
```

- Visualizar los disparos de las reglas:

```
(watch rules)
```

- Desactivar el seguimiento:

```
(unwatch facts)
(unwatch activations)
(unwatch rules)
```

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)`			

Variables en CLIPS

- Variables simples: `?x`, `?y`
  - Toman un valor simple (número, símbolo o cadena de texto).
- Variables múltiples: `$?x`, `$?y`
  - Toman como valor una secuencia de valores simples.
- Variables mudas: Toman un valor que no es necesario recordar.
  - Simple: `?`
  - Múltiple: `$?`

Reglas en CLIPS

- Estructura de una regla (II):

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

- Condiciones positivas y negativas:

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

  - Condiciones positivas: comprueban la presencia de un hecho.
  - Condiciones negativas: comprueban la ausencia de un hecho.

- Acción: eliminar hechos.

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

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

Unión de conjuntos

- Reglas:

```
(defrule inicio
  =>
  (assert (union)))

(defrule union
  ?h <- (union $?u)
  (conjunto ? $? ?e $?)
  (not (union $? ?e $?))
  =>
  (retract ?h)
  (assert (union ?e $?u)))
```

- Conjunto de hechos iniciales:

```
(deffacts datos-iniciales
  (conjunto A 1 2 3 4)
  (conjunto B 1 3 5))
```

- Ejercicio: Intersección de conjuntos.

Unión de conjuntos: `(reset)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
- Agenda:

```
(defrule inicio
  =>
  (assert (union)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
- Agenda:

```
(defrule inicio
  =>
  (assert (union)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union)`
- Agenda:

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 5 3 1)`
- Agenda:

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 5 3 1)`
- Agenda:

```
(defrule union
  ?h <- (union $?u)
  (conjunto ? $? ?e $?)
  (not (union $? ?e $?))
  =>
  (retract ?h)
  (assert (union ?e $?u)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 5 3 1)`
- Agenda:

```
(defrule union
  ?h <- (union $?u)
  (conjunto ? $? ?e $?)
  (not (union $? ?e $?))
  =>
  (retract ?h)
  (assert (union ?e $?u)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 5 3 1)`
- Agenda:

```
(defrule union
  ?h <- (union $?u)
  (conjunto ? $? ?e $?)
  (not (union $? ?e $?))
  =>
  (retract ?h)
  (assert (union ?e $?u)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 5 3 1)`
- Agenda:

```
(defrule union
  ?h <- (union 5 3 1)
  (conjunto ? $? ?e $?)
  (not (union $? ?e $?))
  =>
  (retract ?h)
  (assert (union ?e 5 3 1)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 5 3 1)`
- Agenda:

```
(defrule union
  ?h <- (union 5 3 1)
  (conjunto ? $? ?e $?)
  (not (union $? ?e $?))
  =>
  (retract ?h)
  (assert (union ?e 5 3 1)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 5 3 1)`
- Agenda:

```
(defrule union
  ?h <- (union 5 3 1)
  (conjunto ? $? ?e $?)
  (not (union $? ?e $?))
  =>
  (retract ?h)
  (assert (union ?e 5 3 1)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 5 3 1)`
- Agenda:

```
(defrule union
  ?h <- (union 5 3 1)
  (conjunto ? $? 1 $?)
  (not (union $? 1 $?))
  =>
  (retract ?h)
  (assert (union 1 5 3 1)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 5 3 1)`
- Agenda:

```
(defrule union
  ?h <- (union 5 3 1)
  (conjunto ? $? 1 $?)
  (not (union $? 1 $?))
  =>
  (retract ?h)
  (assert (union 1 5 3 1)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 5 3 1)`
- Agenda:

```
(defrule union
  ?h <- (union 5 3 1)
  (conjunto ? $? ?e $?)
  (not (union $? ?e $?))
  =>
  (retract ?h)
  (assert (union ?e 5 3 1)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 5 3 1)`
- Agenda:

```
(defrule union
  ?h <- (union 5 3 1)
  (conjunto ? $? ?e $?)
  (not (union $? ?e $?))
  =>
  (retract ?h)
  (assert (union ?e 5 3 1)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 5 3 1)`
- Agenda:

```
(defrule union
  ?h <- (union 5 3 1)
  (conjunto ? $? 2 $?)
  (not (union $? 2 $?))
  =>
  (retract ?h)
  (assert (union 2 5 3 1)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 5 3 1)`
- Agenda:

```
(defrule union
  ?h <- (union 5 3 1)
  (conjunto ? $? 2 $?)
  (not (union $? 2 $?))
  =>
  (retract ?h)
  (assert (union 2 5 3 1)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 5 3 1)`
- Agenda:

```
(defrule union
  ?h <- (union 5 3 1)
  (conjunto ? $? 2 $?)
  (not (union $? 2 $?))
  =>
  (retract ?h)
  (assert (union 2 5 3 1)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 5 3 1)`
- Agenda:

```
(defrule union
  ?h <- (union 5 3 1)
  (conjunto ? $? 2 $?)
  (not (union $? 2 $?))
  =>
  (retract ?h)
  (assert (union 2 5 3 1)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 2 5 3 1)`
- Agenda:

```
(defrule union
  ?h <- (union 5 3 1)
  (conjunto ? $? 2 $?)
  (not (union $? 2 $?))
  =>
  (retract ?h)
  (assert (union 2 5 3 1)))
```

Unión de conjuntos: `(run)`

- Memoria de trabajo:
  - `(conjunto A 1 2 3 4)`
  - `(conjunto B 1 3 5)`
  - `(union 2 5 3 1)`
- Agenda:

Unión de conjuntos

Hechos	E	S	Agenda	D	S
f0 (initial-fact)	0		inicio: *	1	
f1 (conjunto A 1 2 3 4)	0				
f2 (conjunto B 1 3 5)	0				
f3 (union)	1	2	union: f3,f1(?e=4),*		2
			union: f3,f1(?e=3),*		2
			union: f3,f1(?e=2),*		2
			union: f3,f1(?e=1),*		2
			union: f3,f2(?e=5),*		2
			union: f3,f2(?e=3),*		2
			union: f3,f2(?e=1),*	2	
f4 (union 1)	2	3	union: f4,f1(?e=4),*		3
			union: f4,f1(?e=3),*		3
			union: f4,f1(?e=2),*		3
			union: f4,f2(?e=5),*		3
			union: f4,f2(?e=3),*	3	
f5 (union 3 1)	3	4	union: f5,f1(?e=4),*		4
			union: f5,f1(?e=2),*		4
			union: f5,f2(?e=5),*	4	
f6 (union 5 3 1)	4	5	union: f6,f1(?e=4),*		5
			union: f6,f1(?e=2),*	5	
f7 (union 2 5 3 1)	5	6	union: f7,f1(?e=4),*		6
f8 (union 4 2 5 3 1)	6				

Plantillas

- Estructura de una plantilla:

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

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

  - Un campo simple `slot` tiene que almacenar exactamente un valor simple.
  - Un campo múltiple `multislot` puede almacenar cualquier secuencia de valores.

- Ejemplos:

```
(deftemplate conjunto
  (slot nombre)
  (multislot datos))

(conjunto (nombre A)
          (datos 1 2 3 4))

(conjunto (nombre B)
          (datos 1 3 5))
```

Unión de conjuntos con plantillas

- Plantillas:

```
(deftemplate conjunto
  (slot nombre)
  (multislot datos))
```

- Reglas:

```
(defrule inicio
  =>
  (assert (conjunto (nombre union) (datos))))

(defrule union
  ?h <- (conjunto (nombre union) (datos $?u))
  (conjunto (datos $? ?e $?))
  (not (conjunto (nombre union) (datos $? ?e $?)))
  =>
  (retract ?h)
  (assert (conjunto (nombre union) (datos ?e $?u))))
```

Restricciones sobre las variables

- Condiciones sobre las variables que se comprueban en el momento de analizar las condiciones de una regla.
  - Negativas: `(dato ?x&~a)`
  - Disyuntivas: `(dato ?x&a|b)`
  - Conjuntivas: `(dato ?x&~a&~b)`
  - Igualdad: `(dato ?x&=<llamada-a-funcion>)`
  - Evaluables: `(dato ?x&:<llamada-a-predicado>)`
- Funciones y predicados en CLIPS.
  - Guía de Programación Básica, sección 12.

Reglas en CLIPS

- Estructura de una regla (III):

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

- Condiciones explícitas: comprueban propiedades de las variables.

```
<condicion> := <patron> |
               (not <patron>) |
               <variable-simple> <- <patron> |
               (test <llamada-a-predicado>)
```

- Acción: presentar información en pantalla.

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

Valores repetidos en un tablero

- Plantillas:

```
(deftemplate casilla
  (slot fila)
  (slot columna)
  (slot valor))
```

- Conjuntos de hechos iniciales:

1	2	1
4	1	3
5	2	4

Valores repetidos en un tablero

- Plantillas:

```
(deftemplate casilla
  (slot fila)
  (slot columna)
  (slot valor))
```

- Conjuntos de hechos iniciales:

```
(deffacts datos-iniciales
  (casilla (fila 1) (columna 1) (valor 1))
  (casilla (fila 1) (columna 2) (valor 2))
  (casilla (fila 1) (columna 3) (valor 1))
  (casilla (fila 2) (columna 1) (valor 4))
  (casilla (fila 2) (columna 2) (valor 1))
  (casilla (fila 2) (columna 3) (valor 3))
  (casilla (fila 3) (columna 1) (valor 5))
  (casilla (fila 3) (columna 2) (valor 2))
  (casilla (fila 3) (columna 3) (valor 4)))
```

Valores repetidos en un tablero

- Valores repetidos en la misma fila:

```
(defrule repetidos-en-la-misma-fila
  (casilla (fila ?f) (columna ?c1) (valor ?v))
  (casilla (fila ?f) (columna ?c2&~?c1) (valor ?v))
  =>
  (printout t "Repetidos en fila " ?f
            " y columnas " ?c1 " y " ?c2 crlf))
```

- Valores repetidos en la misma columna:

```
(defrule repetidos-en-la-misma-columna
  (casilla (fila ?f1) (columna ?c) (valor ?v))
  (casilla (fila ?f2&~?f1) (columna ?c) (valor ?v))
  =>
  (printout t "Repetidos en columna " ?c
            " y filas " ?f1 " y " ?f2 crlf))
```

Valores repetidos en un tablero

- Valores repetidos en la misma diagonal (1):

```
(defrule repetidos-en-la-misma-diagonal-1
  (casilla (fila ?f1) (columna ?c1) (valor ?v))
  (casilla (fila ?f2)
           (columna ?c2&~?c1&:(= (abs (- ?f1 ?f2))
                                 (abs (- ?c1 ?c2))))
           (valor ?v))
  =>
  (printout t "Repetidos en diagonal V1 "
            "(" ?f1 "," ?c1 ") y (" ?f2 "," ?c2 ")" crlf))
```

Valores repetidos en un tablero

- Valores repetidos en la misma diagonal (2):

```
(defrule repetidos-en-la-misma-diagonal-2
  (casilla (fila ?f1) (columna ?c1) (valor ?v))
  (casilla (fila ?f2) (columna ?c2&~?c1) (valor ?v))
  (test (= (abs (- ?f1 ?f2)) (abs (- ?c1 ?c2))))
  =>
  (printout t "Repetidos en diagonal V2 "
            "(" ?f1 "," ?c1 ") y (" ?f2 "," ?c2 ")" crlf))
```

Valores repetidos en un tablero

- Valores repetidos en distintas líneas (1):

```
(defrule repetidos-en-distintas-lineas-1
  (casilla (fila ?f1) (columna ?c1) (valor ?v))
  (casilla (fila ?f2&~?f1)
           (columna ?c2&~?c1) (valor ?v))
  =>
  (printout t "Repetidos en distintas líneas V1 "
            "(" ?f1 "," ?c1 ") y (" ?f2 "," ?c2 ")" crlf))
```

Valores repetidos en un tablero

- Valores repetidos en distintas casillas (1):

```
(defrule repetidos-en-distintas-casillas-1
  (casilla (fila ?f1) (columna ?c1) (valor ?v))
  (casilla (fila ?f2)
           (columna ?c2&:(!= ?f1 ?f2)|:(!= ?c1 ?c2))
           (valor ?v))
  =>
  (printout t "Repetidos en casillas V1 "
            "(" ?f1 "," ?c1 ") y (" ?f2 "," ?c2 ")" crlf))
```

Valores repetidos en un tablero

- Valores repetidos en distintas casillas (2):

```
(defrule repetidos-en-distintas-casillas-2
  ?h1 <- (casilla (fila ?f1) (columna ?c1) (valor ?v))
  ?h2 <- (casilla (fila ?f2) (columna ?c2) (valor ?v))
  (test (neq ?h1 ?h2))
  =>
  (printout t "Repetidos en casillas V2 "
            "(" ?f1 "," ?c1 ") y (" ?f2 "," ?c2 ")" crlf))
```

Ordenación de un vector

- Reglas:

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

(defrule resultado
  (vector $?x)
  (not (vector $?b ?m1 ?m2&:(< ?m2 ?m1) $?e))
  =>
  (printout t "El vector ordenado es " $?x crlf))
```

- Conjuntos de hechos iniciales:

```
(deffacts datos-iniciales
  (vector 3 2 1 4))
```

Ordenación de un vector

Hechos	E	S	Agenda	D	S
`f0 (initial-fact)`	0				
`f1 (vector 3 2 1 4)`	0	1	`ordena: f1(?m1=2,?m2=1)`		1
			`ordena: f1(?m1=3,?m2=2)`	1	
`f2 (vector 2 3 1 4)`	1	2	`ordena: f2(?m1=3,?m2=1)`	2	
`f3 (vector 2 1 3 4)`	2	3	`ordena: f3(?m1=2,?m2=1)`	3	
`f4 (vector 1 2 3 4)`	3		`resultado: *,f4`	4	

Reglas en CLIPS

- Estructura de una regla (IV):

```
(defrule <nombre>
  <cond-combinada>*
  =>
  <accion>*)
```

- Condiciones combinadas: aplicación de operadores lógicos.

```
<cond-combinada> := <condicion> |
                    (not <cond-combinada>) |
                    (and <cond-combinada>+) |
                    (or <cond-combinada>+) |
                    (exists <cond-combinada>+) |
                    (forall <cond-combinada>
                            <cond-combinada>+)
```

Reglas en CLIPS

- Negación: Comprueba la ausencia de hechos en la base de datos que cumplan una condición combinada.
- Conjunción: Comprueba la existencia de hechos en la base de datos que cumplan una serie de condiciones combinadas.
  - Es equivalente a escribir la regla indicando explícitamente todas las condiciones combinadas de la conjunción.
- Disyunción: Comprueba la existencia de hechos en la base de datos que cumplan alguna de una serie de condiciones combinadas.
  - Es equivalente a escribir varias reglas, indicando en cada una de ellas explícitamente una de las condiciones combinadas de la disyunción.

Valores repetidos en un tablero

- Valores repetidos en la misma diagonal (3):

```
(defrule repetidos-en-la-misma-diagonal-3
  (and (casilla (fila ?f1) (columna ?c1) (valor ?v))
       (casilla (fila ?f2) (columna ?c2&~?c1) (valor ?v))
       (test (= (abs (- ?f1 ?f2)) (abs (- ?c1 ?c2)))))
  =>
  (printout t "Repetidos en diagonal V3 "
            "(" ?f1 "," ?c1 ") y (" ?f2 "," ?c2 ")" crlf))
```

Valores repetidos en un tablero

- Valores repetidos en distintas casillas (3):

```
(defrule repetidos-en-distintas-casillas-3
  (casilla (fila ?f1) (columna ?c1) (valor ?v))
  (or (casilla (fila ?f2&~?f1) (columna ?c2) (valor ?v))
      (casilla (fila ?f2) (columna ?c2&~?c1) (valor ?v)))
  =>
  (printout t "Repetidos en distintas casillas V3 "
            "(" ?f1 "," ?c1 ") y (" ?f2 "," ?c2 ")" crlf))
```

Valores repetidos en un tablero

- Equivalencia de la disyunción con varias reglas:

```
(defrule repetidos-en-distintas-casillas-3-1
  (casilla (fila ?f1) (columna ?c1) (valor ?v))
  (casilla (fila ?f2&~?f1) (columna ?c2) (valor ?v))
  =>
  (printout t "Repetidos en casillas V3-1 "
            "(" ?f1 "," ?c1 ") y (" ?f2 "," ?c2 ")" crlf))

(defrule repetidos-en-distintas-casillas-3-2
  (casilla (fila ?f1) (columna ?c1) (valor ?v))
  (casilla (fila ?f2) (columna ?c2&~?c1) (valor ?v))
  =>
  (printout t "Repetidos en casillas V3-2 "
            "(" ?f1 "," ?c1 ") y (" ?f2 "," ?c2 ")" crlf))
```

Reglas en CLIPS

- Existencia: Comprueba la existencia de hechos en la base de datos que cumplan una serie de condiciones combinadas, pero no almacena la información que contienen.
  - Es equivalente a una doble negación de la conjunción de todas las condiciones combinadas.
- Universal: Comprueba que para todos los hechos que cumplen la primera de las condiciones combinadas, también se cumplen todas las demás.
  - Es útil para comprobar que un proceso ha terminado.

Valores repetidos en un tablero

- Existen valores repetidos en distintas casillas:

```
(defrule existen-repetidos-en-casillas-distintas
  (exists (casilla (fila ?f1) (columna ?c1) (valor ?v))
          (casilla (fila ?f2) (columna ?c2) (valor ?v))
          (test (or (!= ?f1 ?f2) (!= ?c1 ?c2))))
  =>
  (printout t "Existen valores repetidos" crlf))
```

Valores repetidos en un tablero

- Existen valores que no se repiten:

```
(defrule existen-valores-que-no-se-repiten
  (exists (casilla (fila ?f1) (columna ?c1) (valor ?v))
          (not (and (casilla (fila ?f2)
                             (columna ?c2) (valor ?v))
                    (test (or (!= ?f1 ?f2)
                              (!= ?c1 ?c2))))))
  =>
  (printout t "Existen valores que no se repiten" crlf))
```

Valores repetidos en un tablero

- Todos los valores pares se repiten:

```
(defrule todos-los-valores-se-repiten
  (forall (casilla (fila ?f1) (columna ?c1)
                   (valor ?v&:(evenp ?v)))
          (casilla (fila ?f2) (columna ?c2) (valor ?v))
          (test (or (!= ?f1 ?f2) (!= ?c1 ?c2))))
  =>
  (printout t "Todos los valores pares se repiten" crlf))
```

Problemas de terminación

- Una misma regla se puede activar con distinto conjunto de hechos dando lugar a una iteración infinita en el proceso de deducción.

```
(deffacts datos-iniciales
  (hecho))

(defrule bucle-infinito
  ?h <- (hecho)
  =>
  (retract ?h)
  (assert (hecho)))
```

Problemas de terminación

- También se puede producir en otras situaciones:

```
(deffacts suma-inicial
  (suma 0))

(defrule suma-valores
  (casilla (fila ?f1) (columna ?c1) (valor ?v))
  ?h <- (suma ?total)
  =>
  (retract ?h)
  (assert (suma (+ ?total ?v))))
```

Problemas de terminación

- Suma de los valores de todas las casillas:

```
(deffacts suma-inicial
  (suma 0))

(defrule suma-valores
  (casilla (fila ?f1) (columna ?c1) (valor ?v))
  (not (sumado ?f1 ?c1))
  ?h <- (suma ?total)
  =>
  (retract ?h)
  (assert (suma (+ ?total ?v))
          (sumado ?f1 ?c1)))

(defrule suma-valores-final
  (forall (casilla (fila ?f1) (columna ?c1))
          (sumado ?f1 ?c1))
  (suma ?total)
  =>
  (printout t "La suma de los valores es " ?total crlf))
```

Plantillas

- En la definición de las plantillas se pueden incluir restricciones sobre los atributos
  - Tipo de dato: `(type <TIPO>)`, donde el `<TIPO>` puede ser SYMBOL, STRING, LEXEME, INTEGER, FLOAT o NUMBER entre otros.
  - Valores permitidos: `(allowed-<TIPO> <lista>)`
  - Rango: `(range <rango-min> <rango-max>)`, donde los límites del rango pueden ser números o `?VARIABLE`.
  - Cardinalidad: `(cardinality <min> <max>)`, donde los límites de la cardinalidad pueden ser números o `?VARIABLE`.
  - Valor por defecto: `(default <val>)`, donde el valor por defecto puede ser `?DERIVE`, `?NONE`, un valor concreto
  - Valor generado por una expresión: `(default-dynamic <expresion>)`

Plantillas

- Definición restrictiva de un tipo de dato:

```
(deftemplate elemento
  (slot id
        (type INTEGER)
        (default ?DERIVE)
        (allowed-integers 0 1 2 3 4 5 6 7 8 9))
  (multislot valor
             (type FLOAT)
             (default ?NONE)
             (range 0 100)
             (cardinality 0 3)))
```

Plantillas: Generación dinámica

- Variable global contadora

```
(defglobal ?*id* = 0)
```

- Función de incremento

```
(deffunction incrementaId ()
  (bind ?*id* (+ ?*id* 1))
  ?*id*)
```

- Plantilla

```
(deftemplate restriccion
  (slot id (default-dynamic (incrementaId)))
  (multislot datos))
```

Bibliografía

- Giarratano, J.C. y Riley, G.
  "Expert Systems Principles and Programming (4th ed.)",
  PWS Pub. Co., 2005.
  - Cap. 7: "Introduction to Clips"
- Giarratano, J.C.
  "CLIPS User's Guide",
  `http://www.clipsrules.net/ug631.pdf`.
- Giarratano, J.C.
  "CLIPS Basic Programming Guide",
  `http://www.clipsrules.net/bpg631.pdf`.