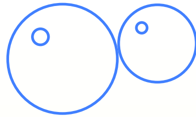
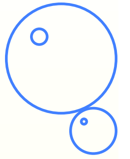
© José Angel Bañares

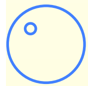


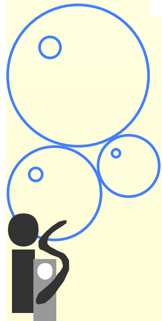


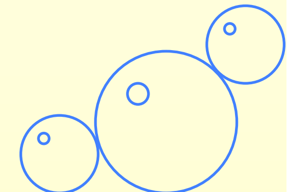
Representación del Conocimiento

Conocimiento Profundo

- Esquemas de representación/integración de paradigmas
  - Reglas, Frames y procedimientos
  - Acciones prodedimentales en CLIPS
  - Objetos en CLIPS (COOL)
  - Ejemplos

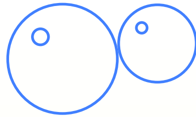


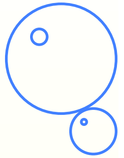




© José Angel Bañares 27/10/09

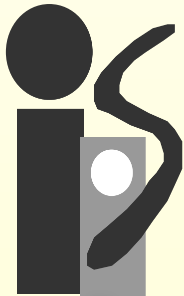
© José Angel Bañares



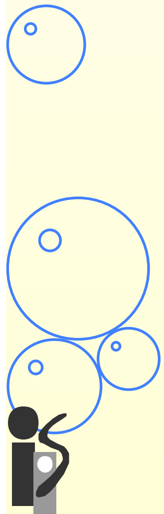


Lenguajes basados en reglas y objetos

Integración de distintos modelos de representación del conocimiento

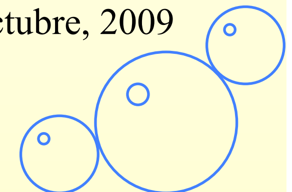


Departamento de Informática e Ingeniería de Sistemas
C.P.S. Universidad de Zaragoza

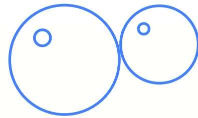


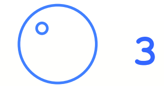
Copyright © 1998 José Angel Bañares

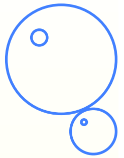
Última revisión: Octubre, 2009



© José Angel Bañares





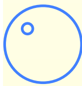


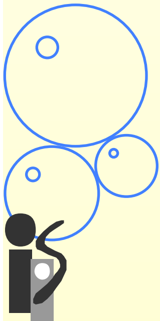
Two paradigms are better than one, but multiple paradigms are even better

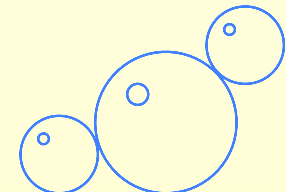
by A. K. Majumdar & J. F. Sowa,

http://www.jfsowa.com/pubs/paradigm.pdf

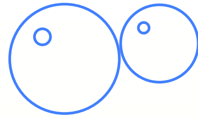
This is a slightly revised version of a paper that was published in the proceedings of ICCS 2009, edited by S. Rudolph, F. Dau, and S.O. Kuznetsov, LNAI 5662, Springer, pp. 32–47, 2009.

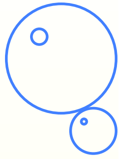




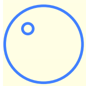


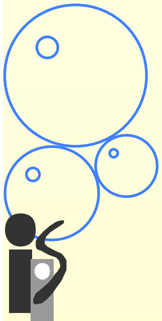
© José Angel Bañares

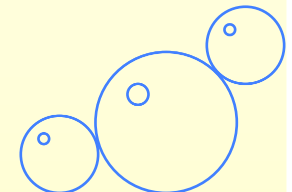




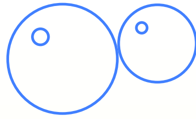
- **Índice**
  - 1. Integración de funciones, reglas y objetos.
  - 2. Acciones procedimentales y Funciones.
  - 3. Utilizando Técnicas orientadas a objetos.
    - Objetos CLIPS
    - Objetos y reglas
    - Preguntas y acciones sobre grupos de objetos
  - 4. Ejemplos de programas en CLIPS
    - Granjero con objetos
    - Granjero con objetos y reglas
    - ELECTRNC.CLP.

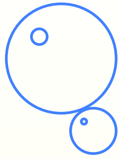






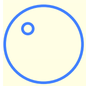
© José Angel Bañares

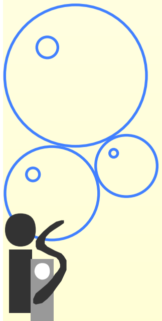


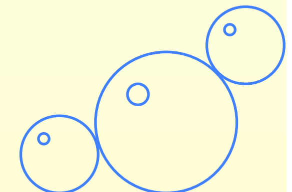


- **Bibliografía**

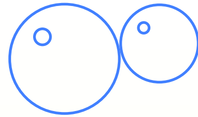
  - Giarratano and Riley. "Expert Systems. Principles and Programming" .PWS Publishing Company, second Edition 1994.
  - CLIPS Reference Manual. Volume I - The Basic Programming Guide". Report JSC-25012 Software Thechnology Branch, Lyndon B. Johnson Space Center.

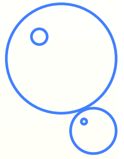






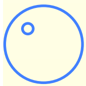
© José Angel Bañares

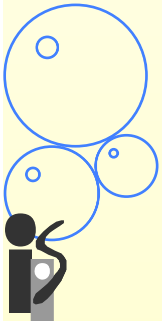


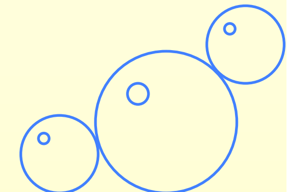


1. Introducción

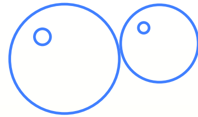
- **Selección de formalismo para representar conocimientos del dominio**
  - **Frames**, para representar el conocimiento **estático** del dominio
    - Permiten **construir taxonomías** de conceptos por especialización de conceptos generales en conceptos más específicos
    - La **herencia** permite compartir propiedades y evita la presencia de conocimiento redundante.
    - Las propiedades se pueden representar de forma **declarativa** o **procedimental**.
    - La estructura interna de los marcos permite mantener internamente las **restricciones de integridad semántica** que existen entre los elementos de un dominio.
    - Facilitan el diseño y mantenimiento de la BC
    - Permiten representar **valores por omisión** y **excepciones**.
    - Las redes semánticas no incorporan información procedural, y son difíciles de comprender y mantener cuando aumenta la BC.

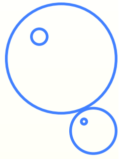






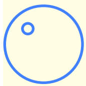
© José Angel Bañares



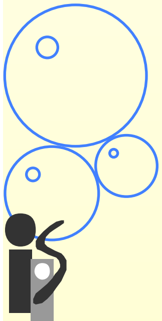


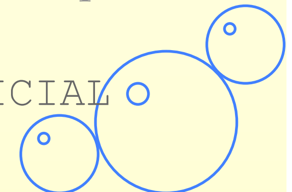
Reglas y Frames

- **Reglas** para representar conocimiento de tipo **dinámico**, es decir conocimientos sobre la resolución del problema
  - Este conocimiento también se puede representar con *frames*
  - Los sistemas basados en *frames*, pueden inferir dinámicamente en tiempo de ejecución, el valor de una propiedad utilizando valores de otras propiedades utilizando demonios
  - Los *frames* hacen uso de demonios para especificar acciones que deberían realizarse frente a ciertas circunstancias
    - Los demonios permanecen inactivos hasta que se solicitan sus servicios
    - Las reglas son chequeadas continuamente

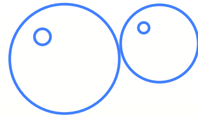


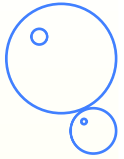
- Resumen:
  - Frames apropiados para representar comportamiento estático y dinámico
    - FRAMES PARA REPRESENTAR CONOCIMIENTO PROFUNDO
  - Reglas para representar comportamiento dinámico que no puede expresarse mediante procedimientos.
    - REGLAS PARA REPRESENTAR CONOCIMIENTO SUPERFICIAL





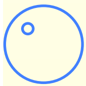
© José Angel Bañares

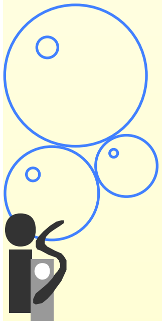


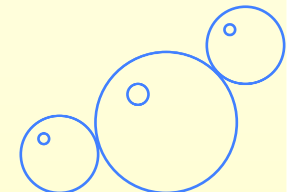


Integración

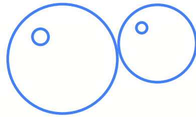
- **Es difícil que un único esquema de representación sea adecuado para representar todos los tipos de conocimiento**
- **Ventajas de la Integración**
  - Permite construir modelos del comportamiento de dominios que incluyen descripciones estructurales y especificaciones declarativas de:
    - El comportamiento de los objetos del dominio (Conocimiento profundo)
      - En lugar de extraer reglas (causa-efecto) de un experto en un sistema que para nosotros es una caja negra, nos creamos un **modelo del sistema.**
    - El comportamiento de los expertos que trabajan con los objetos del dominio (Conocimiento superficial)
  - La combinación de reglas y objetos ofrece un método versátil y natural de modelar los estados y la composición de un sistema.
- **Ejemplos de lenguajes y entornos de desarrollo que integran distintos formalismos**
  - CLIPS, KEE

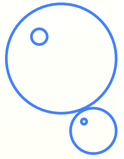






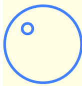
© José Angel Bañares

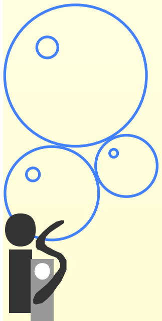


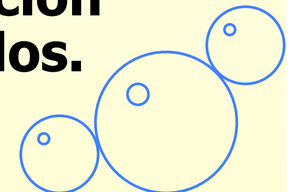


CLIPS

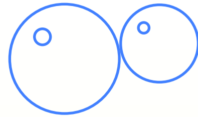
- **CLIPS fue diseñado por el centro espacial Jonshon (NASA) con los siguientes propósitos:**
  - Herramienta de desarrollo de sistemas expertos portable y de bajo costo
  - Fácil integración con otros sistemas.
- **Hoy CLIPS es una de las herramientas más difundidas para el desarrollo de sistemas expertos**
  - Los ejecutables, los programas fuentes en C y la documentación de CLIPS son públicos, y existen implementaciones de CLIPS en otros lenguajes (Ada, y Java).
  - Integración con otras herramientas:
    - CLISP puede ser llamado como una subrutina (en C) a la que se puede pasar y/o de la que se puede recibir información.
    - CLIPS puede utilizar funciones externas definidas en otro lenguaje (en C).
- **Esencialmente es un sistema de Producción de encadenamiento progresivo con una sintaxis extendida del OPS5, y permite utilizar el paradigma de programación estructurada y OO cuando éstos sean más apropiados.**







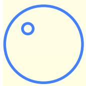
© José Angel Bañares

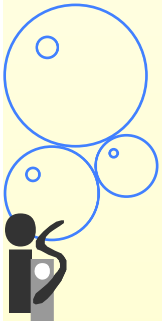


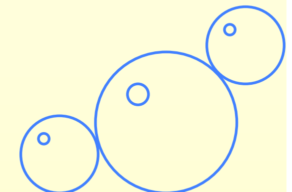
2. Acciones Procedimentales y Funciones

- **Función if and read**

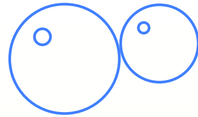
```
(defrule eleccion-humana
?turno <- (turno h)
? pila <- (numero-de-piezas ?n&(> ?n 1))
=>
(retract ?turno)
(printput t "Escribe el numero de piezas que coges: ")
(bind ?m (read))
(if (and (integerp ?m) (>= ?m 1) (<= ?m 3) (< ?m ?n))
    then (retract ?pila)
         (bind ?nuevo-numero-de-piezas (-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

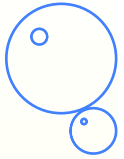






© José Angel Bañares

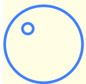


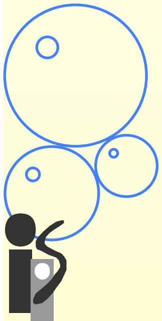


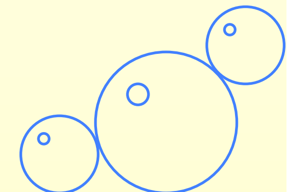
La función while

- **(While <expresion> [do] <accion>*)**

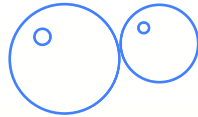
```
(defrule elige-jugador
?fase <- (fase elige-jugador)
=>
(retract ?fase)
(printout "Elige quien empieza, computadora o humano (c/h):")
(bind  ?jugador (read))
(while (not (or (eq ?jugador c) (eq jugador h))) do
       (print t ?jugador " es distinto de c y h" crlf)
       (print t "Elige quien empieza, computadora o humano (c/h):")
       (bind ?jugador (read)))
(assert (turno ?jugador))
(assert (fase elige-numero-de-piezas)))
```

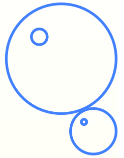






© José Angel Bañares

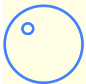


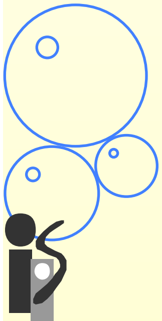


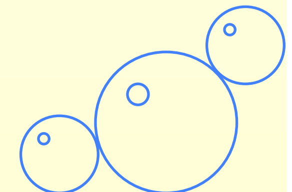
Definición de funciones

- **Deffunction**

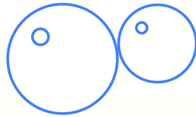
```
(deffunction turno-de-jugador-elegido ()
 (printout “Elige quien empieza, computadora o humano (c/h):“)
 (bind  ?jugador (read))
 (while (not (or (eq ?jugador c) (eq jugador h))) do
        (print t ?jugador “ es distinto de c y h” crlf)
        (print t “Elige quien empieza, computadora o humano (c/h):“)
        (bind ?jugador (read)))
 (assert (turno ?jugador)))

(defrule elige-jugador
?fase <- (fase elige-jugador)
=>
(turno-de-jugador-elegido)
(assert (fase elige-numero-de-piezas)))
```

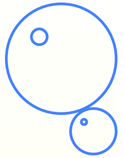






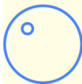
© José Angel Bañares

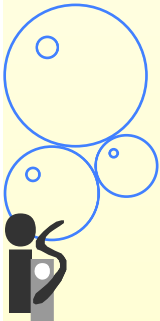


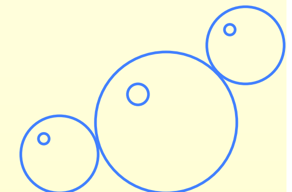


Relación de funciones procedimentales

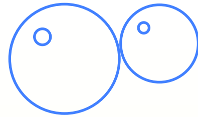
- **Funciones procedimentales definidas**
  - (bind <variable> <expresion>*)
  - (if <expresion> then <accion>* [else <accion>*])
  - (while <expresion> [do] <accion>*)
  - (loop-for-count <rango> [do] accion*)
  - (progn <expresion>*)
  - (return [<expresion>])
  - (break)

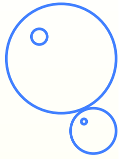






© José Angel Bañares

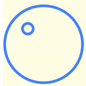


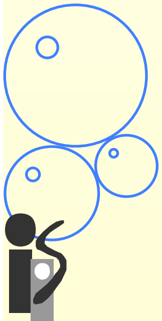


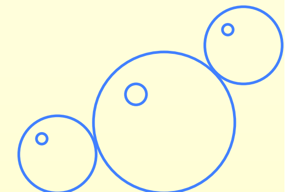
Funciones E/S I

- **(Assert-string <cadena>)**

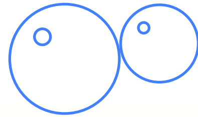
```
CLIPS>(defrule inserta-hecho
          =>
      (print-out t "Escribe un hecho como cadena" crlf)
      (assert-string (read)))
CLIPS>(reset)
CLIPS>(run)
Escribe un hecho como cadena
"(color verde)"
CLIPS>(facts)
f-0     (initial-facts)
f-1     (color-verde)
For a total of 2 facts
CLIPS>
```

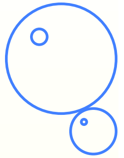






© José Angel Bañares

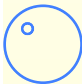


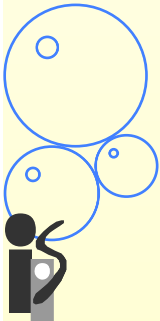


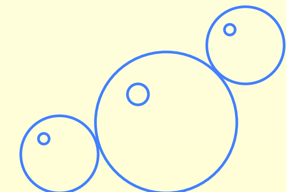
Funciones E/S II

- **(readline) (str-cat <cadena>*)**

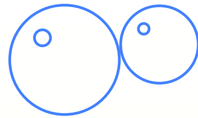
```
CLIPS>(defrule lee-linea
        =>
      (printout t "Introduce datos." crlf)
      (bind ?cadena (readline))
      (assert-string (str-cat "(" ?cadena ")")))
CLIPS>(reset)
CLIPS>(run)
Introduce datos.
colores verde azul ambar rojo
CLIPS>(facts)
f-0     (initial-fact)
f-1     (colores verde azul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

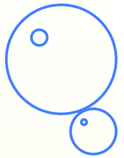






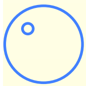
© José Angel Bañares

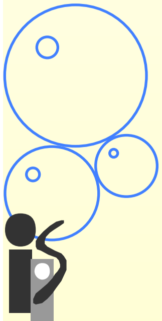


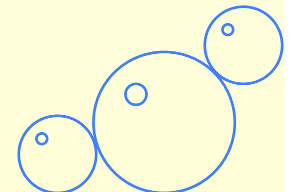


Otras funciones

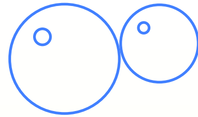
- **Funciones de cadenas:**
  - `str-cat, sym-cat, substring, str-index, eval, build, etc.`
- **Funciones de campos múltiples:**
  - `create, nth, member, delete, explode, implode, etc.`
- **Funciones matemáticas**
  - Trigonométricas: `acos, acosh, acot, acoth, acsc, acsh, etc.`
  - Básicas: `+, -, *, /, div, max, min, abs, float, integer, deg-grad, deg-rad, grad-deg, rad-deg, pi, **, sqrt, exp, log, log1-, round, mod`
- **Otras:**
  - `gensym, random, length`
- **Documentación funciones:**
  - (help)

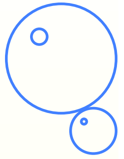






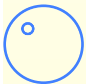
© José Angel Bañares

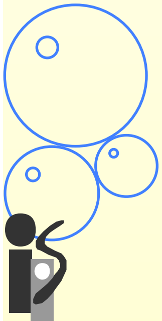


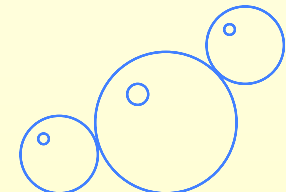


3. COOL (CLIPS Object Oriented Language)

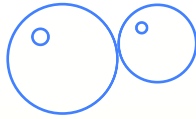
- **Características del la POO en CLIPS**
  - Abstracción
    - La definición de una nueva clase crea (implementa) un nuevo TAD
  - Encapsulación
    - Instancias de clases definidas por el usuarios deben ser accedidas a través de mensajes (**Excepto** en la lectura de atributos en la LHS de las reglas y en preguntas sobre conjuntos de instancias)
  - Herencia
    - Una clase puede definirse a partir de otra u otras
  - Polimorfismo
    - La implementación de un mensaje puede ser diferente para distintas clase
  - Ligadura dinámica
    - Variables y referencias a instancias pueden referirse dinámicamente a diferentes objetos.
  - Modelo de objetos *inspirado en CLOS y en el modelo clásico.*

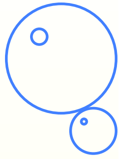






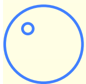
© José Angel Bañares

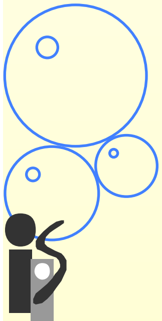


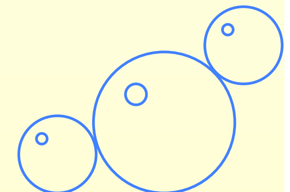


Desviaciones de la POO pura en CLIPS

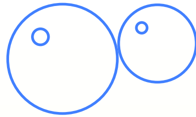
- **No todo es un objeto en CLIPS:**
  - Hechos y reglas se manipulan de la forma tradicional
- **Preguntas sobre conjuntos de instancias y reglas pueden acceder directamente al valor de los atributos (sin necesidad de mensajes), comprometiendo de esta forma la encapsulación**
  - El beneficio son mayores prestaciones
  - EL paso de mensajes se seguirá utilizando para invocar servicios de instancias ligadas en la RHS de las reglas, o como acciones invocadas a los objetos resultantes de *instance-set queries*

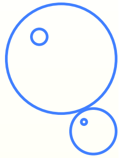






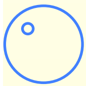
© José Angel Bañares

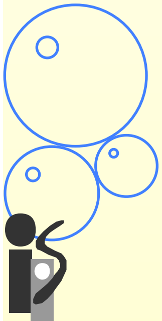


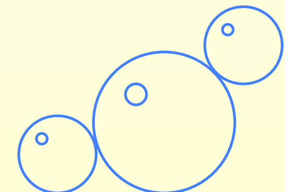


3.1 Estructura de los objetos

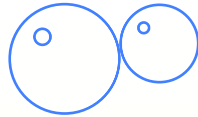
- **La construcción** `defclass` **especifica los slots de una nueva clase de objetos,**
  - Los *facets* permiten especificar propiedades del *slot*, tal como acceso para lectura y escritura, o reactividad ante reconocimiento de patrones
- **La construcción** `defmessage-handler` **permite definir los métodos para una clase de objetos**
- **La herencia múltiple permite especializar una clase existente**
  - La nueva clase hereda sus slots y sus métodos

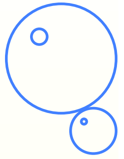






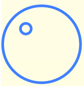
© José Angel Bañares

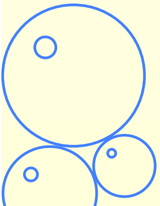


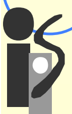


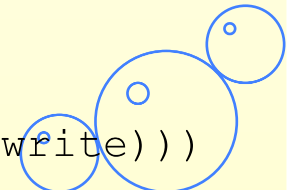
Ejemplo de Objetos

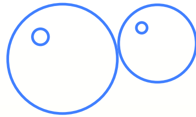
```
(defclass POSESION (is-a USER)
   (role concrete)
   (pattern-match reactive)
   (slot localizacion
        (type STRING)
        (create-accessor read-write))
   (slot valor
        (type NUMBER)
        (create-accessor read-write))
(defclass COCHE (is-a USER)
   (slot color
        (type STRING)
        (create-accessor read-write))
   (slot velocidad-maxima
        (type NUMBER)
        (create-accessor read-write))
(defclass UTILITARIO (is-a POSESION COCHE)
   (slot no-licencia (type SYMBOL) (create-accessor read-write)))
```

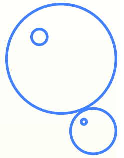








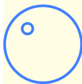




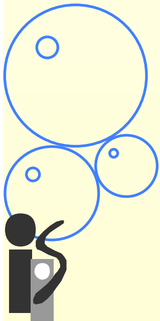
© José Angel Bañares

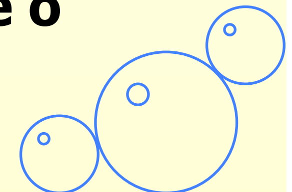
Acceso a objetos (Paso de mensajes)

- **Sólo se puede acceder a los slots mediante los métodos para leer o escribir en ellos.**
  - CLIPS puede crear estos métodos si se indica en el facet `create-accessor`
- **CLIPS es una mezcla de lenguajes OO como Smalltalk y CLOS**
  - Permite enviar mensajes a una instancia: **`send`.**
    - Permite distintos tipos de métodos: **primario, before, after y around**
    - Programación declarativa e imperativa
  - Permite definir **funciones genéricas**

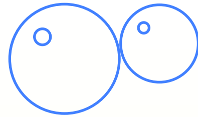


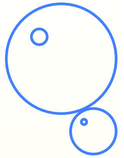
- **Los nombres de instancias se indican entre `[]`**
- **Una instancia se crea mediante la función `make-instance`, la cual envía al nuevo objeto el mensaje `init`.**
- **Las instancias pueden ser referenciadas por nombre o dirección**





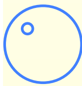
© José Angel Bañares

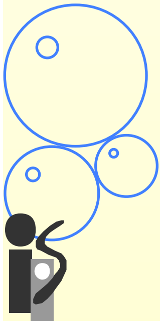


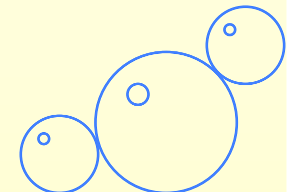


Ejemplo acceso a objetos

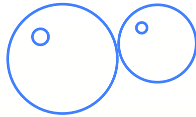
```
CLIPS> (make-instance [oc1] of UTILITARIO)
[oc1]
CLIPS> (send [oc1] print)
[oc1] of UTILITARIO
(color nil)
(velocidad-maxima 0)
(localizacion "")
(valor 0)
(np-licencia nil)
CLIPS> (send [oc1] put-color VERDE)
VERDE
CLIPS> (send [oc1] get-color)
VERDE
CLIPS>
```

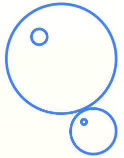




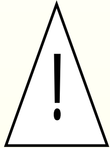


© José Angel Bañares





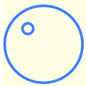
DEFMESSAGE-HANDLER

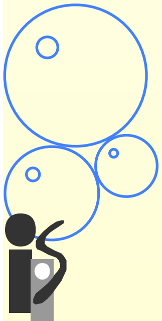


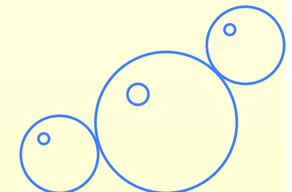
– **Defmessage** no es para definir los métodos de una función genérica, sino los métodos de una clase.

- **Métodos after y before (DEMONS ---> Igual que en CLOS)**

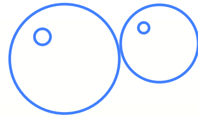
```
CLIPS> (clear)
CLIPS> (defclass A (is-a USER) (role concrete))
CLIPS> (defmessage-handler A delete before ()
              (printout t "Borrando una instancia de la clase A" crlf))
CLIPS>(defmessage-handler A delete after ()
         (printout t "Se ha borrado una instancia de la clase A" crlf))
CLIPS>(watch instances)
CLIPS>(make-instance a of A)
==> instance [a] of A
[a]
CLIPS>(send [a] delete)
Borrando una instancia de la clase A
<== instance [a] of A
Se ha borrado una instancia de la clase A
```

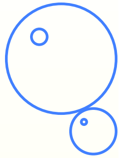






© José Angel Bañares

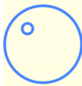


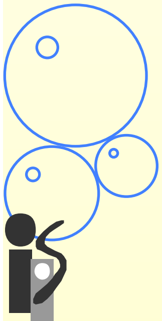


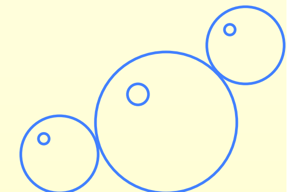
Instancia activa y parámetros

- **`self` designa la instancia que recibe el mensaje**

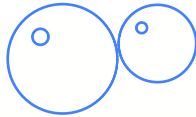
```
CLIPS> (clear)
CLIPS> (defclass A (is-a USER) (role concrete))
CLIPS> (make-instance a of A)
[a]
CLIPS> (defmessage-handler A print-args(?a ?b $?c)
          (printout t (instance-name ?self) "" ?a "" ?b " y "
                 (length$ ?c) " extras: " ?c crlf))
CLIPS>(send [a] print-args a b c d)
[a] a b y 2 extras: (c d)
CLIPS>
```

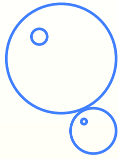






© José Angel Bañares

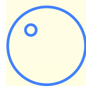


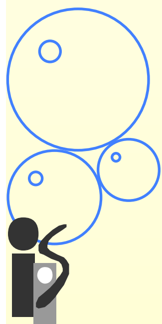


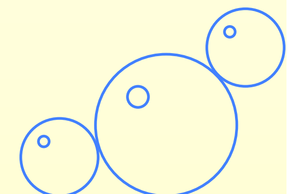
Acceso atributos dentro del método

- **Dentro de un método se puede acceder directamente a los atributos:**
  - ?self:<nombre-slot> y (bind ?self:<nombre-slot> <valor>)

```
CLIPS>(clear)
CLIPS>
(defclass A (is-a USER)
   (role concrete)
    (slot atr-1 (default 1))
   (slot atr-2 (default 2)))
CLIPS> (defmessage-handler A print-todos-slots ()
                 (printout t ?self:atr-1 " " ?self:atr-2 crlf))
CLIPS>(make-instance a of A)
[a]
CLIPS>(send [a] print-todos-slots)
1 2
CLIPS>(defmessage-handler A set-atr-1 (?valor)
          (bind ?self:atr-1 ?valor))
CLIPS> (send [a] set-atr-1 34)
34
```







© José Angel Bañares

Funciones genéricas

- **Una función genérica puede examinar la clase de sus argumentos**
  - En el cuerpo de los métodos se pueden enviar mensajes a los argumentos que son instancias de alguna clase.
  - Permite la sobrecarga de operadores

```
CLIPS>(clear)
CLIPS>
(defmethod + ((?a STRING) (?b STRING))
     (str-cat ?a ?b))
CLIPS>(+ 1 2)
3
CLIPS> (+ "Ho" "la")
"Hola"
CLIPS> (+ "Cara" "co" "la")
[GENREXE1] No applicable methods for +.
FALSE
```

© José Angel Bañares

Facets

- **Los facets describen características de los slots:**
  - Tipo del slot: `Multislot` o `slot`.
  - Valor por defecto `(default <value>)`
  - Storage: `(storage shared)` o `(storage local)`
  - Acces: `read-write`, `read-only`, e `initialize-only`.

    Como read-only, pero también se puede dar valor con init y make-instance.

    Sólo se puede dar valor por default en la definición de clase
  - Inheritance: `(propagation inherit)` o `(propagation non-inherit)`

    Sólo las instancias de la clase tendrán los slots
  - Source: `(source exlusive)` o `(source composite)`

    Hereda facets del padre

    Hereda facets del padre y SUPERCLASES
  - Pattern-Match Reactivity: `(pattern-match reactive)` o `(pattern-match non-reactive)`

© José Angel Bañares

- Visibility: `(visibility public)` o `(visibility private)`.

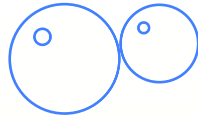
  Los manejadores de mensajes de subclases pueden acceder al slot

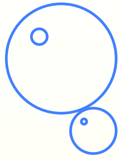
- Create-accessor: Si se especifica `read`, `writer` o `read-writer` crea los métodos primarios `get-<slot>` y `put-<slot>`.

  Por defecto, make-instance, initialize-instance, etc. ponen valor
  Con put-<nombre-slot>. Se puede especificar otro método.

- Override Message: Permite que se utilice un método específico al utilizar la forma estándar de acceder slots.

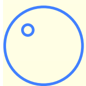
© José Angel Bañares

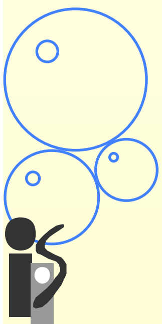


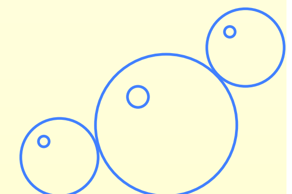


3.2 Objetos y reglas

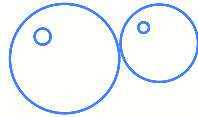
- **Los patrones con objetos son como los patrones con hechos, excepto en:**
  - Las **reglas sólo pueden reconocer clases de objetos que han sido definidas antes que las reglas**. Una clase y cualquiera de los *slots* que es reconocido debe de ser "**reactivo**". -> (Que cuando se produzca algún cambio en sus atributos se entere el motor de inferencia).
    - Un Patrón que reconoce una superclase, reconocerá las instancias de una subclase
  - A diferencia de los hechos, los cambios en los slots son locales y no afectan a ninguna regla que explícitamente no tenga un patrón con ese slot.
  - Una instancia debe de ser reactiva para que sea reconocida
  - Se permiten restricciones especiales sobre el nombre y la clase (`is-a`, `name`).

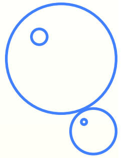






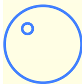
© José Angel Bañares

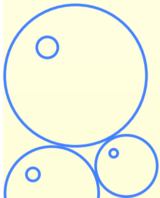


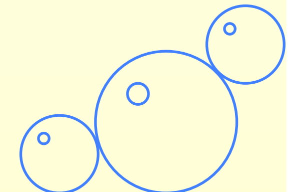


Ejemplos objetos y reglas

```
CLIPS> (make-instance [oc1] of POSESION)
[i1]
CLIPS> (make-instance [oc2] of UTILITARIO)
[oc2]
CLIPS> (send [oc1] put-value 100)
CLIPS>
(defrule muestra-valores-con-valor
   ?oi <- (object (is-a POSESION) (valor ?x&:(> ?x 0)))
=>
   (printout t (instance-name ?oi) " tiene un valor de " ?x clrf))
CLIPS>(run)
[oc1] tiene un valor de 100
CLIPS> (send [oc1] put-value 36)
36
CLIPS> (send [oc2] put-value 99)
99
CLIPS> (run)
[oc2] tiene un valor de 99
[oc1] tiene un valor de 36
```







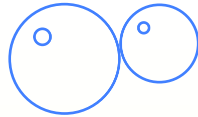
© José Angel Bañares

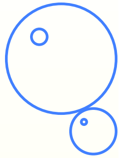
3.3 Preguntas y Acciones sobre conjuntos de instancias

- **COOL permite obtener y realizar acciones sobre conjuntos de instancias que cumplen ciertas propiedades *(instance-set queries* y *distributed actions)***

Función	Propósito
any-instancep	Determina si una o más instancias satisfacen una pregunta
find-instance	Devuelve la primera instancia que satisface la pregunta
find-all-instance	Agrupa y devuelve todas las instancias que satisfacen la pregunta
do-for-instance	Realiza una acción sobre ala primera instancia que satisface la pregunta
do-for-all-instances	Realiza una acción por cada instancia que satisface una pregunta
delayed-do-for-all-instances	Agrupa todas las instancias que satisfacen una pregunta e itera una acción sobre el grupo

© José Angel Bañares

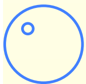


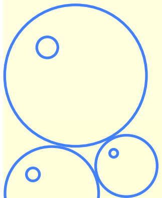


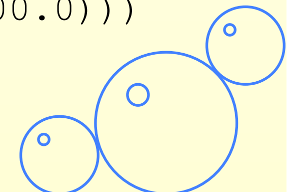
Ejemplos Instance-set

- **Ejemplo**

```
(defclass PERSONA (is-a USER)
  (role ab<stract)
  (slot sexo (access read-only)
             (storage shared))
  (slot edad (type NUMBER)
             (visibility public)))

(defmessage-handler PERSONA
                 put-edad (?value)
  (dynamic-put edad ?value))

(defclass FEMENINO (is-a PERSONA)
  (role abstract)
  (slot sexo (source composite)
             (default FEMENINO)))

(defclass MASCULINO (is-a PERSONA)
  (role abstract)
  (slot sexo (source composite)
             (default MASCULINO)))

(defclass NIGNA (is-a FEMENINO)
  (role concrete)
  (slot edad (source composite)
             (default 4)
             (range 0.0 17.9)))

(defclass MUJER (is-a FEMENINO)
  (role concrete)
  (slot edad (source composite)
             (default 25)
             (range 18.0 100.0)))

(defclass NIGNO (is-a MASCULINO)
  (role concrete)
  (slot edad (source composite)
             (default 4)
             (range 0.0 17.9)))

(defclass HOMBRE (is-a MASCULINO)
  (role concrete)
  (slot edad (source composite)
             (default 25)
             (range 18.0 100.0)))
```







© José Angel Bañares

Vinculación dinámica de manejadores que acceden a slots.

- **Los manejadores que acceden directamente a slots se vinculan estáticamente.**

```
(defclass A (is-a USER)
  (slot foo
      (create-accessor read)))

(defclass B (is-a USER)
(role concrete)
  (slot foo))
```

¡Ojo si la clase redefine el slot!

```
CLIPS>(make-instance b of B)
[b]
CLIPS>(send [b] get-foo)
```

¡Error!, La referencia estática al slot foo de la clase A no se aplica a [b] de B

```
(defclass A (is-a USER)
  (slot foo
      (create-accessor read)))

(defmessage-handler A get-foo ()
    (dynamic-get foo))

(defclass B (is-a USER)
(role concrete)
  (slot foo (visibility public)))

CLIPS>(make-instance b of B)
[b]
CLIPS>(send [b] get-foo)
NIL
CLIPS>
```

© José Angel Bañares

Ejemplo acción sobre instance-set

```
(definstances GENTE
   (HOMBRE-1 of HOMBRE (edad 18))
   (HOMBRE-2 of HOMBRE (edad 60))
   (MUJER-1 of MUJER (edad 18))
   (MUJER-2 of MUJER (edad 60))
   (MUJER-3 of MUJER)
   (NIGNO-1 of NIGNO (edad 8))
   (NIGNO-2 of NIGNO)
   (NIGNO-3 of NIGNO)
   (NIGNO-4 of NIGNO)
   (NIGNA-1 of NIGNA (edad 8))
   (NIGNA-2 of NIGNA))
CLIPS> (do-for-all-instances ((?hombre-o-nigno NIGNO HOMBRE)
                              (?mujer-o-nigna NIGNA MUJER))
  (and (>= ?hombre-o-nigno:edad 18)
   (>= ?mujer-o-nigna:edad 18))
    (printout t ?hombre-o-nigno " " ?mujer-o-nigna crlf))

[HOMBRE-1] [MUJER-1]
[HOMBRE-1] [MUJER-2]
[HOMBRE-1] [MUJER-3]
[HOMBRE-2] [MUJER-1]
[HOMBRE-2] [MUJER-2]
[HOMBRE-2] [MUJER-3]
```

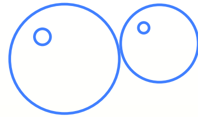
instance-set member member variables

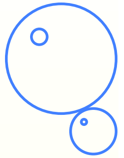
instance-set member class restrictions

instance-set query

instance-set distributed action

© José Angel Bañares



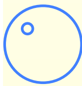


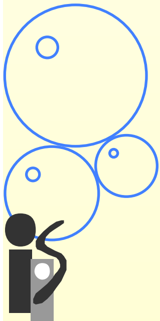
4.1 Problema del granjero representado sólo con objetos

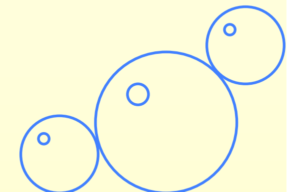
- El problema del granjero, el lobo, la cabra y la col
  - Utilización de técnicas de POO para resolver el problema

- Definición de clases

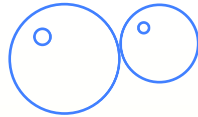
```
;;; Las instancias de estado representan los nodos del árbol
;;; de búsquedas y contienen la información sobre el estado

(defclass estado
  (is-a USER) (role concrete)
  (slot granjero
    (create-accessor write)(default orilla-1))
  (slot lobo
    (create-accessor write)(default orilla-1))
  (slot cabra
    (create-accessor write)(default orilla-1))
  (slot col
    (create-accessor write)(default orilla-1))
  (slot padre
    (create-accessor write)(default sin-padre))
  (slot profundidad
    (create-accessor write)(default 1))
  (slot ultimo-movimiento
    (create-accessor write)(default no-mueve)))
```

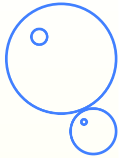






© José Angel Bañares

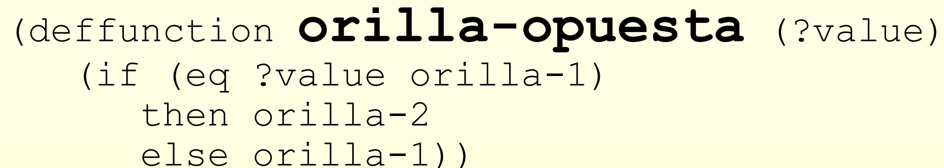




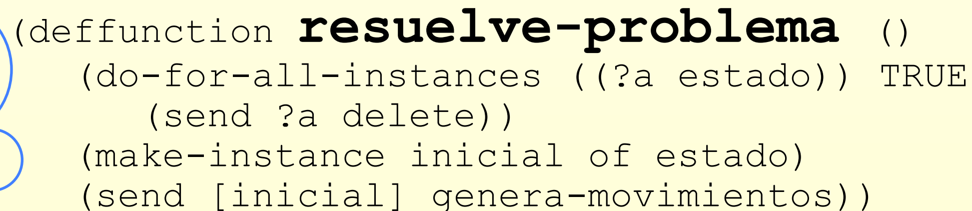
Funciones

- **Definición de funciones**

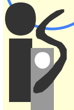
```
(deffunction contradiccion (?granjero ?lobo ?cabra ?col ?profundidad)
   (if (or (and (eq ?lobo ?cabra) (neq ?granjero ?lobo))
           (and (eq ?cabra ?col) (neq ?granjero ?cabra)))
      then TRUE
      else
      (any-instancep ((?e estado))
        (and (eq ?e:granjero ?granjero)
          (eq ?e:lobo ?lobo)
          (eq ?e:cabra ?cabra)
          (eq ?e:col ?col)
          (< ?e:profundidad ?profundidad)))))
```

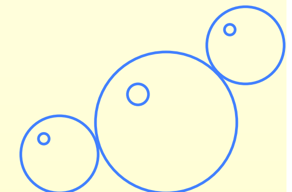


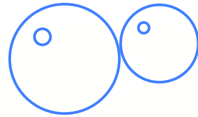
```
(deffunction orilla-opuesta (?value)
   (if (eq ?value orilla-1)
      then orilla-2
      else orilla-1))
```



```
(deffunction resuelve-problema ()
   (do-for-all-instances ((?a estado)) TRUE
      (send ?a delete))
   (make-instance inicial of estado)
   (send [inicial] genera-movimientos))
```





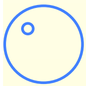


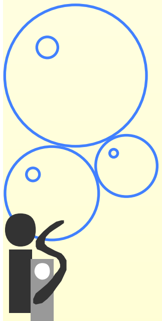
© José Angel Bañares

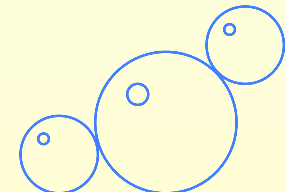
Reglas

- Ejecución normal de programas CLIPS
  - Para ejecutar load, reset y run

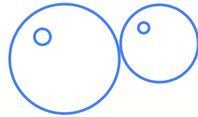
```
(defrule inicializa
 =>
  (resuelve-problema))
```

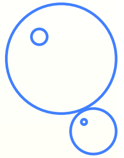






© José Angel Bañares

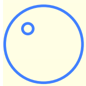


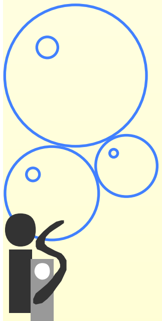


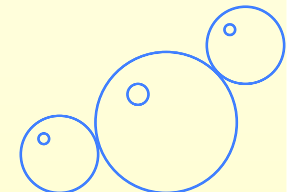
Manejadores de Mensajes

- **Definición de métodos**
  - Operaciones aplicables al estado

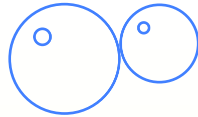
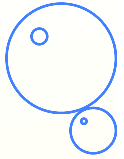
```
(defmessage-handler estado mueve-granjero ()
   (if (not (contradiccion (orilla-opuesta ?self:granjero) ?self:lobo
                            ?self:cabra ?self:col ?self:profundidad))
      then
      (bind ?x (make-instance (gensym) of estado
         (granjero (orilla-opuesta ?self:granjero))
         (lobo ?self:lobo)
         (cabra ?self:cabra)
         (col ?self:col)
         (ultimo-movimiento granjero)
         (padre ?self)
         (profundidad (+ ?self:profundidad 1))))
      (if (not (send ?x solucion?))
         then
         (send ?x genera-movimientos))))
```





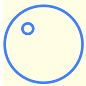
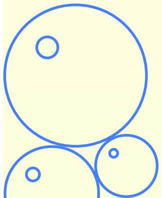


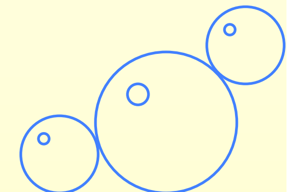
© José Angel Bañares

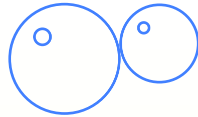
Mueve cabra

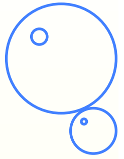
```
(defmessage-handler estado mueve-cabra ()
  (if (and (eq ?self:granjero ?self:cabra)
           (not (contradiccion (orilla-opuesta ?self:granjero)
                    ?self:lobo (orilla-opuesta ?self:cabra)
                    ?self:col ?self:profundidad)))
    then
    (bind ?x (make-instance (gensym) of estado
      (granjero (orilla-opuesta ?self:granjero))
      (lobo ?self:lobo)
      (cabra (orilla-opuesta ?self:granjero))
      (col ?self:col)
      (ultimo-movimiento cabra)
      (padre ?self)
      (profundidad (+ ?self:profundidad 1))))
    (if (not (send ?x solucion?))
      then
      (send ?x genera-movimientos))))
```



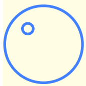
© José Angel Bañares

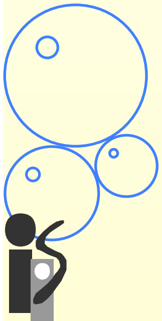


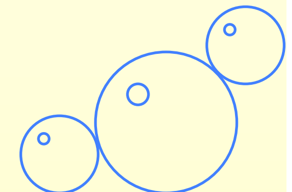


Mueve lobo

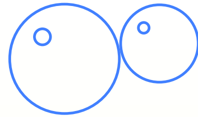
```
(defmessage-handler estado mueve-lobo ()
  (if (and (eq ?self:granjero ?self:lobo)
           (not (contradiccion (orilla-opuesta ?self:granjero)
                               (orilla-opuesta ?self:lobo)
                            ?self:cabra ?self:col ?self:profundidad)))
    then
    (bind ?x (make-instance (gensym) of estado
      (granjero (orilla-opuesta ?self:granjero))
      (lobo (orilla-opuesta ?self:granjero))
      (cabra ?self:cabra)
      (col ?self:col)
      (ultimo-movimiento lobo)
      (padre ?self)
      (profundidad (+ ?self:profundidad 1))))
    (if (not (send ?x solucion?))
      then
      (send ?x genera-movimientos))))
```

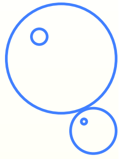






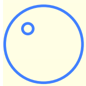
© José Angel Bañares

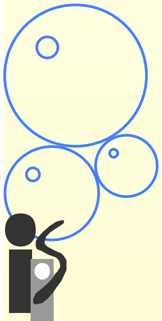


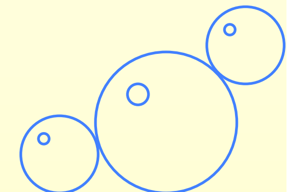


Mueve col

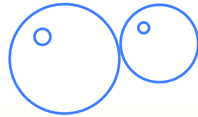
```
(defmessage-handler estado mueve-col ()
  (if (and (eq ?self:granjero ?self:col)
          (not (contradiccion (orilla-opuesta ?self:granjero)
                              ?self:lobo ?self:cabra
                              (orilla-opuesta ?self:col)
                              ?self:profundidad)))
    then
    (bind ?x (make-instance (gensym) of estado
       (granjero (orilla-opuesta ?self:granjero))
       (lobo ?self:lobo)
       (cabra ?self:cabra)
       (col (orilla-opuesta ?self:granjero))
       (ultimo-movimiento col)
       (padre ?self)
       (profundidad (+ ?self:profundidad 1))))
    (if (not (send ?x solucion?))
       then
       (send ?x genera-movimientos))))
```

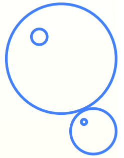






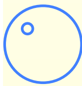
© José Angel Bañares

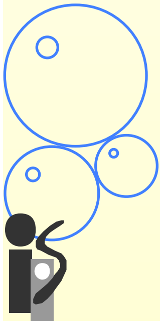


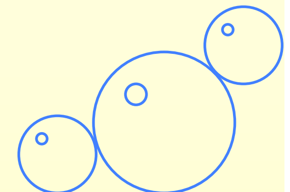


Aplica operadores a estado

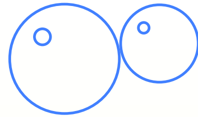
```
(defmessage-handler estado genera-movimientos ()
  (send ?self mueve-granjero)
  (send ?self mueve-lobo)
  (send ?self mueve-cabra)
  (send ?self mueve-col))
```

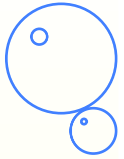






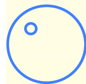
© José Angel Bañares

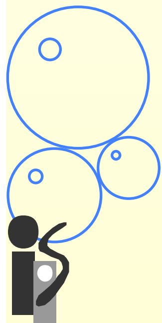


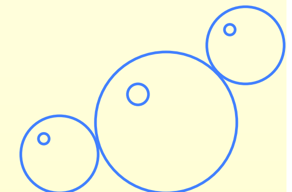


Reconoce solución y la escribe

```
(defmessage-handler estado escribe-solucion ()
  (if (neq ?self:padre sin-padre)
    then
    (send ?self:padre escribe-solucion)
    (bind ?mueve-dest (dynamic-get ?self:ultimo-movimiento))
    (if (eq ?self:ultimo-movimiento granjero)
      then
      (printout t "granjero se mueve solo a " ?mueve-dest "." crlf)
      else
      (printout t "granjero se mueve con "
          ?self:ultimo-movimiento " a " ?mueve-dest "." crlf))))

(defmessage-handler estado solucion? ()
  (if (and (eq ?self:granjero orilla-2) (eq ?self:lobo orilla-2)
        (eq ?self:cabra orilla-2) (eq ?self:col orilla-2))
    then
    (printout t crlf "solucion encontrada" crlf crlf)
    (send ?self escribe-solucion)
    TRUE
    else
    FALSE))
```







© José Angel Bañares

4.2 Problema del granjero con objetos y reglas

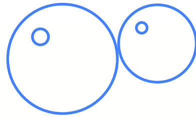
o El problema del granjero, el lobo, la cabra y la col

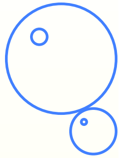
Utilización de objetos y reglas para resolver el problema

- Definición de clases

```
;;; Instancias de estado representan los nodos del árbol
(defclass estado (is-a USER)(role concrete)(pattern-match reactive)
  (slot profundidad (create-accessor write)
    (type INTEGER) (range 1 ?VARIABLE) (default 1))
  (slot padre (create-accessor write)
    (type INSTANCE-ADDRESS) (default ?DERIVE))
  (slot localizacion-granjero (create-accessor write)(type SYMBOL)
    (allowed-symbols orilla-1 orilla-2)(default orilla-1))
  (slot localizacion-lobo (create-accessor write)
    (type SYMBOL)(allowed-symbols orilla-1 orilla-2)
    (default orilla-1))
  (slot localizacion-cabra (create-accessor write)
    (type SYMBOL)(allowed-symbols orilla-1 orilla-2)
    (default orilla-1))
  (slot localizacion-col (create-accessor write)(type SYMBOL)
    (allowed-symbols orilla-1 orilla-2) (default orilla-1))
  (slot ultimo-movimiento(create-accessor write)(type SYMBOL)
    (allowed-symbols sin-movimiento solo lobo cabra col)
    (default sin-movimiento)))
```

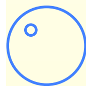
© José Angel Bañares





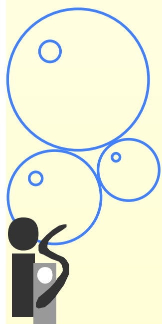
clases (cont.) y estado inicial

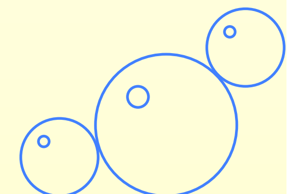
```
;;; Instancias de movimientos contienen la información de los
;;; movimientos realizados para alcanzar un estado

(defclass movimientos (is-a USER) (role concrete) (pattern-match reactive)
   (slot id
      (create-accessor write)
      (type INSTANCE))
   (multislot lista-movimientos
      (create-accessor write)
      (type SYMBOL)
      (allowed-symbols sin-movimiento solo lobo cabra col)))

(defclass opuesto-de (is-a USER) (role concrete)  (pattern-match reactive)
   (slot valor (create-accessor write))
   (slot valor-opuesto (create-accessor write)))
```



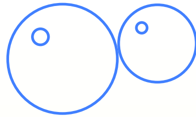
- Estado inicial

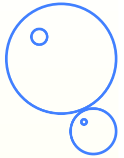
```
(definstances Inicializa
  (of estado)
  (of opuesto-de (valor orilla-1) (valor-opuesto orilla-2))
  (of opuesto-de (valor orilla-2) (valor-opuesto orilla-1)))
```





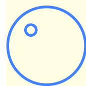
© José Angel Bañares

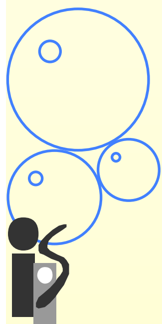


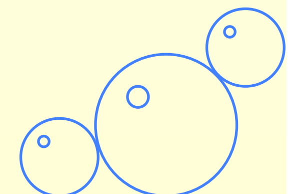


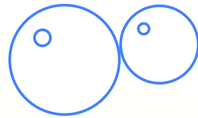
Reglas para representar los operadores

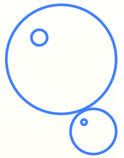
```
(defrule mueve-solo
  ?nodo <- (object (is-a estado) (profundidad ?num)
                  (localizacion-granjero ?lg))
  (object (is-a opuesto-de) (valor ?lg) (valor-opuesto ?nl))
  =>
  (duplicate-instance ?nodo (profundidad (+ 1 ?num))(padre ?nodo)
    (localizacion-granjero ?nl) (ultimo-movimiento solo)))

(defrule mueve-con-lobo
  ?nodo <- (object (is-a estado) (profundidad ?num)
                  (localizacion-granjero ?lg)
                  (localizacion-lobo ?lg))
  (object (is-a opuesto-de) (valor ?lg) (valor-opuesto ?nl))
  =>
  (duplicate-instance ?nodo (profundidad (+ 1 ?num))(padre ?nodo)
    (localizacion-granjero ?nl)(ultimo-movimiento lobo)
    (localizacion-lobo ?nl)))
```







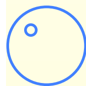


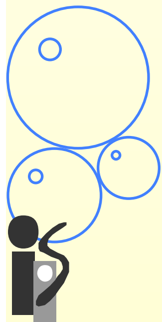


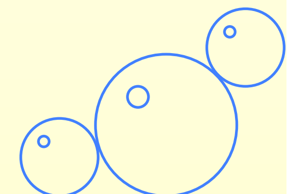
© José Angel Bañares

Operadores (reglas)

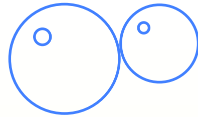
```
(defrule mueve-con-cabra
  ?nodo <- (object (is-a estado) (profundidad ?num)
                (localizacion-granjero ?lg)
                (localizacion-cabra ?lg))
  (object (is-a opuesto-de) (valor ?lg) (valor-opuesto ?nl))
  =>
  (duplicate-instance ?nodo (profundidad (+ 1 ?num))(padre ?nodo)
    (localizacion-granjero ?nl)(ultimo-movimiento cabra)
    (localizacion-cabra ?nl)))

(defrule mueve-con-col
  ?nodo <- (object (is-a estado)(profundidad ?num)
                (localizacion-granjero ?lg)
                (localizacion-col ?lg))
  (object (is-a opuesto-de) (valor ?lg) (valor-opuesto ?nl))
  =>
  (duplicate-instance ?nodo (profundidad (+ 1 ?num))(padre ?nodo)
    (localizacion-granjero ?nl)(ultimo-movimiento col)
    (localizacion-col ?nl)))
```

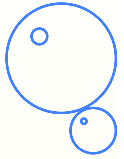






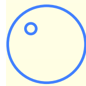
© José Angel Bañares

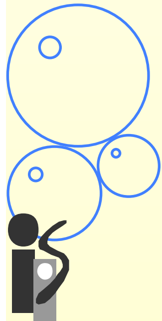


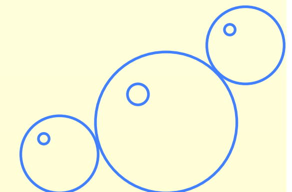


Reglas que detectan violación de restricciones

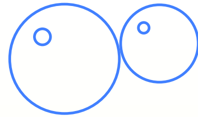
```
(defrule lobo-come-cabra
  (declare (salience 200))
  ?nodo <- (object (is-a estado)(localizacion-granjero ?s1)
           (localizacion-lobo ?s2&~?s1) (localizacion-cabra ?s2))
  =>
  (unmake-instance ?nodo))

(defrule cabra-come-col
  (declare (salience 200))
  ?nodo <- (object (is-a estado)(localizacion-granjero ?s1)
          (localizacion-cabra ?s2&~?s1)(localizacion-col ?s2))
  =>
  (unmake-instance ?nodo))

(defrule camino-circular
  (declare (salience 200))
  (object (is-a estado)(profundidad ?sd1)
         (localizacion-granjero ?lg) (localizacion-lobo ?xs)
         (localizacion-cabra ?gs)(localizacion-col ?cs))
  ?nodo <- (object (is-a estado)(profundidad ?sd2&:(< ?sd1 ?sd2))
              (localizacion-granjero ?lg)(localizacion-lobo ?xs)
              (localizacion-cabra ?gs)(localizacion-col ?cs))
  =>
  (unmake-instance ?nodo))
```

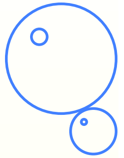






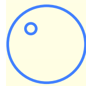
© José Angel Bañares

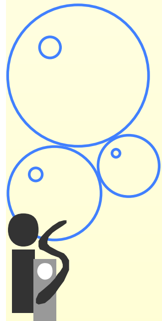


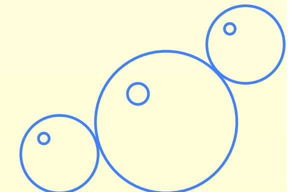


Reglas para reconocer solución y escribirla

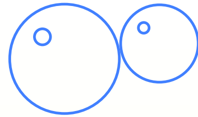
```
(defrule reconoce-solucion
  (declare (salience 100))
  ?nodo <- (object (is-a estado) (padre ?padre)
                   (localizacion-granjero orilla-2)
                   (localizacion-lobo orilla-2)
                   (localizacion-cabra orilla-2)
                   (localizacion-col orilla-2)
                   (ultimo-movimiento ?mueve))
  =>
  (unmake-instance ?nodo)
  (make-instance of movimientos (id ?padre) (lista-movimientos ?mueve)))

(defrule agnade-solucion
  (declare (salience 100))
  ?estado <- (object (is-a estado)
                     (padre ?padre)
                     (ultimo-movimiento ?mueve))
  ?mv <- (object (is-a movimientos)
                (id ?estado) (lista-movimientos $?rest))
  =>
  (modify-instance ?mv (id ?padre) (lista-movimientos ?mueve ?rest)))
```

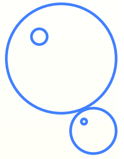






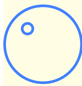
© José Angel Bañares

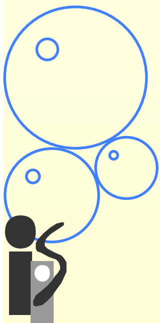


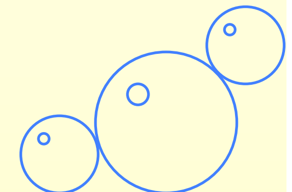


Escribe solución

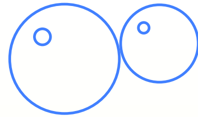
```
(defrule escribe-solucion
  (declare (salience 100))
  ?mv <- (object (is-a movimientos)
               ;(id [sin-padre])
               (lista-movimientos sin-movimiento $?m))
  =>
  (unmake-instance ?mv)
  (printout t t  "Solucion encontrada " t t)
  (bind ?length (length ?m))
  (bind ?i 1)
  (bind ?orilla orilla-2)
  (while (<= ?i ?length)
    (bind ?cosa (nth$ ?i ?m))
    (if (eq ?cosa solo)
      then (printout t "Granjero se mueve solo a " ?orilla "." t)
      else (printout t "Granjero se mueve con " ?cosa " a "
                    ?orilla "." t))
    (if (eq ?orilla orilla-1)
      then (bind ?orilla orilla-2)
      else (bind ?orilla orilla-1))
    (bind ?i (+ 1 ?i))))
```

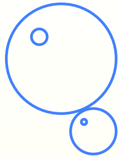






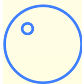
© José Angel Bañares

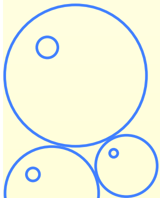


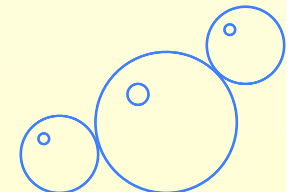


4.3 Ejemplo ELECTRNC.CLP

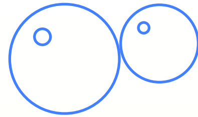
```
;;;=============================================================
;;;   Sistema Experto que simplifica la tabla de verdad de
;;;   un circuito que consta de entradas (SOURCES) y salidas
;;;   (LEDS)
;;;
;;;     El proceso de simplificación consiste en:
;;;     1) Se inicializan las conexiones entre los componente
;;;     2) Se determina la respuesta del circuito cuando
;;;        todas las entradas son cero
;;;     3) Se cambian los valores de las entradas una a una
;;;        y se determinan las respuestas.
;;;        Se itera a través de todas las entradas posibles
;;;        utilizando un código gray (un sistema de representación
;;;        numérica que utiliza dígitos binarios en los cuales
;;;        el siguiente entero sólo difiere en un dígito).
;;;        Por ejemplo, los códigos gray para los enteros 0 a 7
;;;        son 0 = 000, 1 = 001, 2 = 011, 3 = 010, 4 = 110,
;;;        5 = 111, 6 = 101, 7 = 100. Utilizando un código gray,
;;;        sólo cambia una SOURCE cada vez para determinar la
;;;        siguiente respuesta más rápidamente.
```

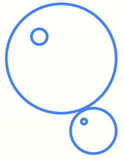






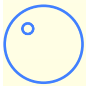
© José Angel Bañares

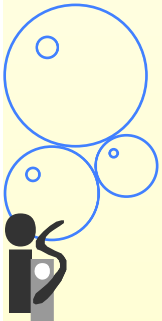


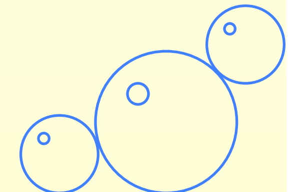


Definición del problema

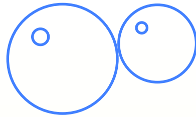
```
;;;      4)Según se determinan las respuestas, una reglas
;;;         si dos conjuntos de entradas con la misma
;;;         respuesta difieren en una única entrada. Si es así,
;;;         esa entrada puede ser reemplazada con un *
;;;         (indicando que no importa el valor de la entrada).
;;;         por ejemplo,  si la entrada 0 1 0 dió una respuesta
;;;         de 1 0 y la entrada 0 0 0 dió la misma respuesta,
;;;         entonces la tabla de decisión puede simplificarse
;;;         indicando que 0 * 0 da como respuesta 1 0.
;;;      5) Una vez determinadas todas las respuestas y
;;;         simplificaciones, se imprime la tabla de decisión.
;;;
;;;      CLIPS Version 6.0 Example
;;;
;;;      Para ejecutar, load este fichero, load uno de los
;;;      ficheros circuito (circuit1.clp, circuit2.clp, or
;;;      circuit3.clp), reset, and run.
;;;======================================================
```

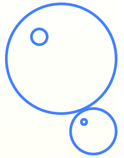






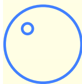
© José Angel Bañares

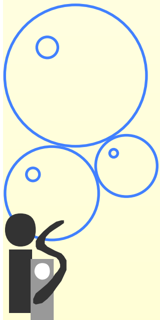


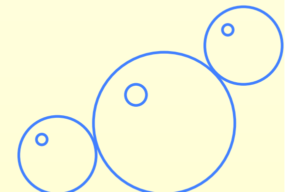


Recursos utilizados en el ejemplo

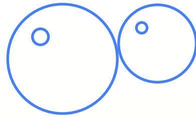
- **Este ejemplo ilustra la mayoria de las construcciones disponibles en CLIPS**
  - COOL es integrado con las reglas
  - Las funciones genéricas se emplean para conectar las componentes
  - Classes, message-handlers, y deffunctions determinan la respuesta del circuito ante las entradas
  - Reglas, deffunctions, y variables globales se utilizan para el control de la ejecución, iterar a través de las entradas, simplificar la tabla de decisión e imprimir la tabla.

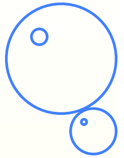






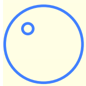
© José Angel Bañares

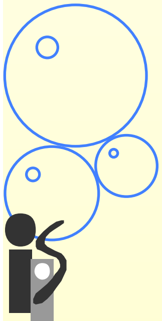


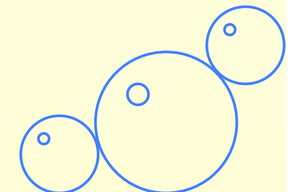


Clases abstractas: componente, sin-salidas

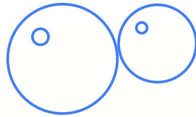
```
;;;***********
;;; DEFCLASSES
;;;***********
(defclass COMPONENT
  (is-a USER)
  (slot ID# (create-accessor write)))

(defclass NO-OUTPUT
  (is-a USER)
  (slot number-of-outputs (access read-only)
                          (default 0)
                          (create-accessor read)))

(defmessage-handler NO-OUTPUT compute-output ())
```

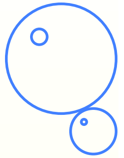






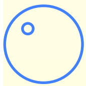
© José Angel Bañares

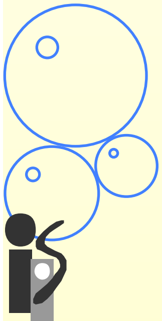


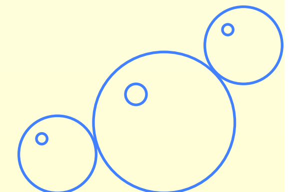


Una-salida. Demon que propaga salida

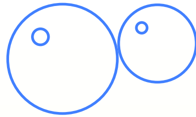
```
(defclass ONE-OUTPUT
  (is-a NO-OUTPUT)
  (slot number-of-outputs (access read-only)
                          (default 1)
                          (create-accessor read))
  (slot output-1 (default UNDEFINED) (create-accessor write))
  (slot output-1-link (default GROUND) (create-accessor write))
  (slot output-1-link-pin (default 1) (create-accessor write)))

(defmessage-handler ONE-OUTPUT put-output-1 after (?value)
   (send ?self:output-1-link
         (sym-cat put-input- ?self:output-1-link-pin)
         ?value))
```

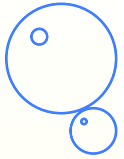






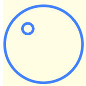
© José Angel Bañares

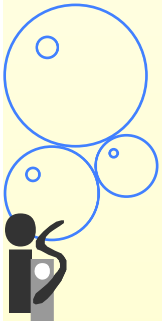


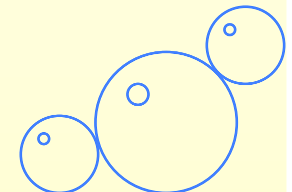


Dos-salidas. Demon que propaga salida

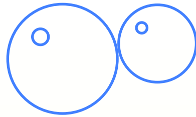
```
(defclass TWO-OUTPUT
  (is-a ONE-OUTPUT)
  (slot number-of-outputs (access read-only)
                          (default 2)
                          (create-accessor read))
  (slot output-2 (default UNDEFINED) (create-accessor write))
  (slot output-2-link (default GROUND) (create-accessor write))
  (slot output-2-link-pin (default 1) (create-accessor write)))

(defmessage-handler TWO-OUTPUT put-output-2 after (?value)
   (send ?self:output-2-link
         (sym-cat put-input- ?self:output-2-link-pin)
         ?value))
```

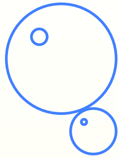






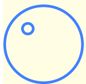
© José Angel Bañares

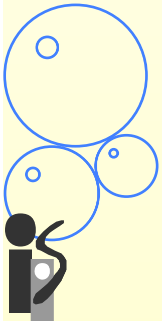


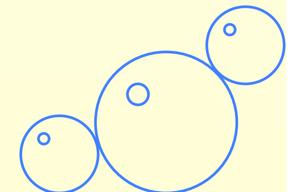


Sin-entradas. Una-entrada. Demon que recalcula salida

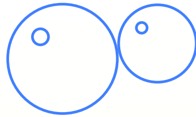
```
(defclass NO-INPUT
  (is-a USER)
  (slot number-of-inputs (access read-only)
                         (default 0)
                         (create-accessor read)))

(defclass ONE-INPUT
  (is-a NO-INPUT)
  (slot number-of-inputs (access read-only)
                         (default 1)
                         (create-accessor read))
  (slot input-1 (default UNDEFINED)
                (visibility public)
                (create-accessor read-write))
  (slot input-1-link (default GROUND) (create-accessor write))
  (slot input-1-link-pin (default 1) (create-accessor write)))

(defmessage-handler ONE-INPUT put-input-1 after (?value)
   (send ?self compute-output))
```

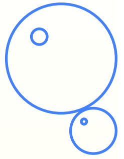






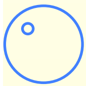
© José Angel Bañares

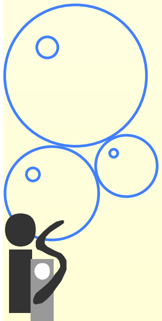


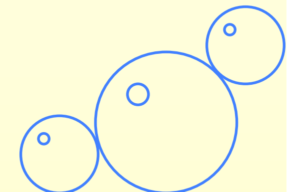


Dos-entradas. Demon que recalcula salida

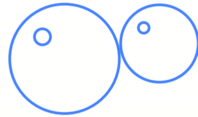
```
((defclass TWO-INPUT
  (is-a ONE-INPUT)
  (slot number-of-inputs (access read-only)
                         (default 2)
                         (create-accessor read))
  (slot input-2 (default UNDEFINED)
                (visibility public)
                (create-accessor write))
  (slot input-2-link (default GROUND) (create-accessor write))
  (slot input-2-link-pin (default 1) (create-accessor write)))

(defmessage-handler TWO-INPUT put-input-2 after (?value)
   (send ?self compute-output))
```

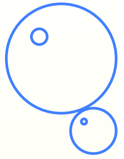






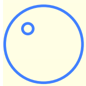
© José Angel Bañares

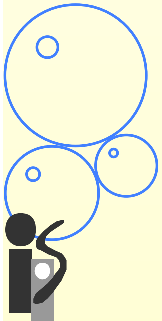


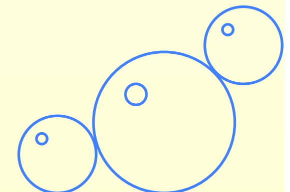


Componentes instanciables

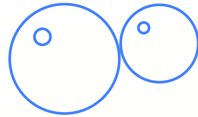
```
(defclass SOURCE
  (is-a NO-INPUT ONE-OUTPUT COMPONENT)
  (role concrete)
  (slot output-1 (default UNDEFINED) (create-accessor write)))

(defclass SINK
  (is-a ONE-INPUT NO-OUTPUT COMPONENT)
  (role concrete)
  (slot input-1 (default UNDEFINED) (create-accessor read-write)))
```

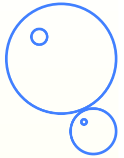






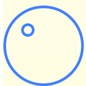
© José Angel Bañares

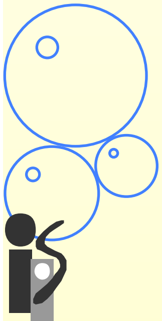


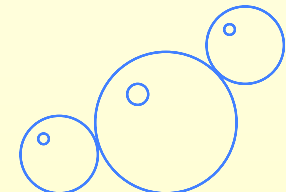


Puertas. Métododos para cálcular las salidas

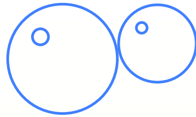
```
;;;*******************
;;; NOT GATE COMPONENT
;;;*******************

(defclass NOT-GATE
  (is-a ONE-INPUT ONE-OUTPUT COMPONENT)
  (role concrete))

(deffunction not# (?x) (- 1 ?x))

(defmessage-handler NOT-GATE compute-output ()
   (if (integerp ?self:input-1) then
       (send ?self put-output-1 (not# ?self:input-1))))
```

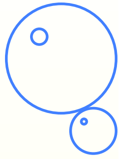






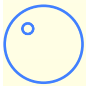
© José Angel Bañares

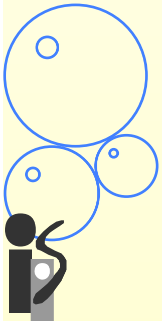


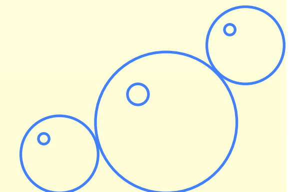


Consultas sobre instancias de objetos, funciones

```
;;;*************
;;; LED COMPONENT
;;;*************

(defclass LED
  (is-a ONE-INPUT NO-OUTPUT COMPONENT)
  (role concrete))

;;; Devuelve el valor de cada instancia LED
;;; en una lista
(deffunction LED-response ()
   (bind ?response (create$))
   (do-for-all-instances ((?led LED)) TRUE
      (bind ?response (create$ ?response (send ?led get-
   input-1))))
```





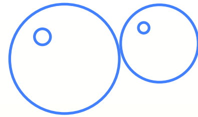


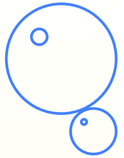
© José Angel Bañares

Funciones genéricas y métodos para conectar

```
;;;**************************
;;; DEFGENERICS AND DEFMETHODS
;;;**************************
(defgeneric connect)
;;; Conecta un componente de una salida a uno de una entrada.
(defmethod connect ((?out ONE-OUTPUT) (?in ONE-INPUT))
   (send ?out put-output-1-link ?in)
   (send ?out put-output-1-link-pin 1)
   (send ?in  put-input-1-link ?out)
   (send ?in  put-input-1-link-pin 1))

;;: Conecta un componente de una salida a un pin de un componente de
;;; dos entradas
(defmethod connect ((?out ONE-OUTPUT) (?in TWO-INPUT)
                    (?in-pin INTEGER))
   (send ?out put-output-1-link ?in)
   (send ?out put-output-1-link-pin ?in-pin)
   (send ?in  (sym-cat put-input- ?in-pin -link) ?out)
   (send ?in  (sym-cat put-input- ?in-pin -link-pin) 1))
```

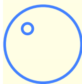
© José Angel Bañares

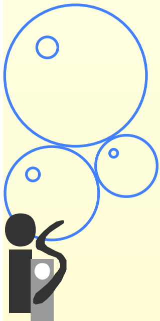


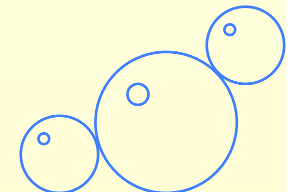


Métodos para conectar

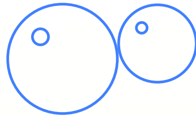
```
;;; Conecta un pin de un componente de dos salidas a uno de una
;;; entrada
(defmethod connect ((?out TWO-OUTPUT) (?out-pin INTEGER)
                    (?in ONE-INPUT))
   (send ?out (sym-cat put-output- ?out-pin -link) ?in)
   (send ?out (sym-cat put-output- ?out-pin -link-pin) 1)
   (send ?in put-input-1-link ?out)
   (send ?in put-input-1-link-pin ?out-pin))

;;; Conecta un pin de un componente con dos salidas
;;; a un pin de un componente con dos entradas.
(defmethod connect ((?out TWO-OUTPUT) (?out-pin INTEGER)
                    (?in TWO-INPUT) (?in-pin INTEGER))
   (send ?out (sym-cat put-output- ?out-pin -link) ?in)
   (send ?out (sym-cat put-output- ?out-pin -link-pin) ?in-pin)
   (send ?in  (sym-cat put-input- ?in-pin -link) ?out)
   (send ?in  (sym-cat put-input- ?in-pin -link-pin) ?out-pin))
```

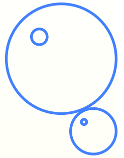






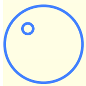
© José Angel Bañares

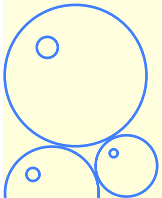


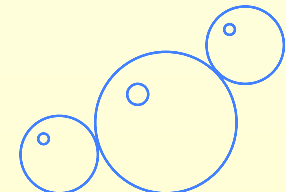


Variables globales y funciones

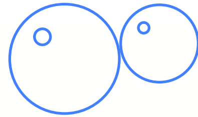
```
;;;**************************
;;; DEFGLOBALS AND DEFFUNCTIONS
;;;**************************
(defglobal ?*gray-code* = (create$)
           ?*sources* = (create$)
           ?*max-iterations* = 0)

;;; A partir de la iteración en curso, determina que bit
;;; cambiará en el código gray en la siguiente.
;;; Algoritmo por cortesía de John R. Kennedy (The BitMan).
(deffunction change-which-bit (?x)
   (bind ?i 1)
   (while (and (evenp ?x) (!= ?x 0)) do
      (bind ?x (div ?x 2))
      (bind ?i (+ ?i 1)))
   ?i)

;;; Declaración Forward puesto que la configuración inicial
;;; está en un fichero separado.
(deffunction connect-circuit ())
```

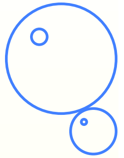






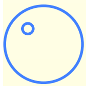
© José Angel Bañares

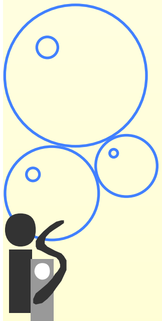


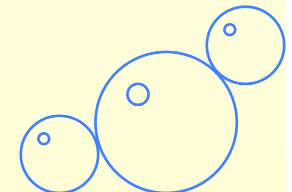


Regla para iniciar el sistema

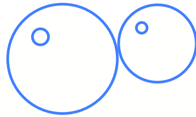
```
;;;*********
;;; DEFRULES
;;;*********

(defrule startup
  =>
  ;; Initialize the circuit by connecting the components
  (connect-circuit)
  ;; Setup the globals.
  (bind ?*sources* (find-all-instances ((?x SOURCE)) TRUE))
  (do-for-all-instances ((?x SOURCE)) TRUE
     (bind ?*gray-code* (create$ ?*gray-code* 0)))
  (bind ?*max-iterations* (round (** 2 (length ?*sources*))))
  ;; Do the first response.
  (assert (current-iteration 0)))
```

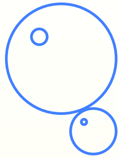






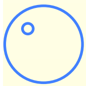
© José Angel Bañares



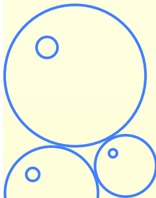


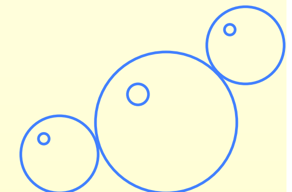
Cómputo para la primera entrada 0 .. 0

```
(defrule compute-response-1st-time
   ?f <- (current-iteration 0)
   =>
   ;; Pon todas las fuentes a cero.
   (do-for-all-instances ((?source SOURCE)) TRUE
               (send ?source put-output-1 0))
   ;; Determina la respuesta inicial de los LED.
   (assert (result ?*gray-code* =(str-implode (LED-response))))
   ;; Comienza la iteración a través de los códigos gray
   (retract ?f)
   (assert (current-iteration 1)))
```



```
CLIPS> (reset)
CLIPS> (run 2)
CLIPS>(facts)
f-0     (initial-fact)
f-2     (result 0 0 "1 1")
f-3     (current-iteration 1)
For a total of 3 facts.
```



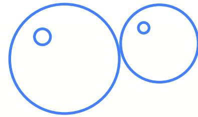


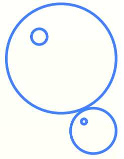
© José Angel Bañares

Cómputo de las demás respuestas

```
(defrule compute-response-other-times
   ?f <- (current-iteration ?n&~0&:(< ?n ?*max-iterations*))
   =>
   ;; cambia el código gray, guardando el bit cambiado.
   (bind ?pos (change-which-bit ?n))  ; posición
   (bind ?nv (- 1 (nth ?pos ?*gray-code*))) ; cambio valor
   (bind ?*gray-code* (replace$ ?*gray-code* ?pos ?pos ?nv));reemplaza
   ;; Cambia la fuente correspondiente
   (send (nth ?pos ?*sources*) put-output-1 ?nv)
   ;; Determina la respuesta de los LED.
   (assert (result ?*gray-code* =(str-implode (LED-response))))
   ;; Assert el nuevo hecho contador de iteracion
   (retract ?f)
   (assert (current-iteration =(+ ?n 1))))
CLIPS> (run 1)
CLIPS> (facts)
f-0     (initial-fact)
f-2     (result 0 0 "1 1")
f-4     (result 1 0 "1 1")
f-5     (current-iteration 2)
```

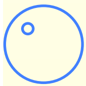
© José Angel Bañares

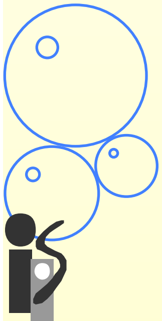


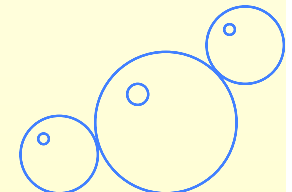


Mezcla respuestas

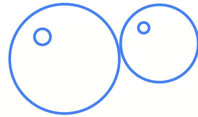
```
(defrule merge-responses
   (declare (salience 10))
   ?f1 <- (result $?b  ?x $?e ?response)
   ?f2 <- (result $?b ~?x $?e ?response)
   =>
   (retract ?f1 ?f2)
   (assert (result $?b * $?e ?response)))

CLIPS> (run 1)
CLIPS> (facts)
f-0     (initial-fact)
f-5     (current-iteration 2)
f-6     (result * 0 "1 1")
For a total of 3 facts.
CLIPS>
```

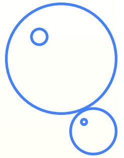






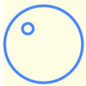
© José Angel Bañares

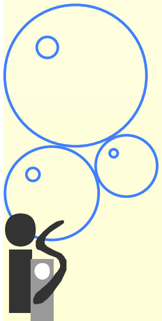


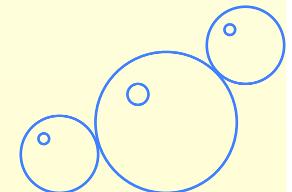


Imprime cabecera

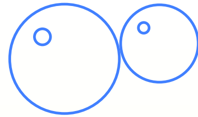
```
(defrule print-header
  (declare (salience -10))
   =>
  (assert (print-results))
  (do-for-all-instances ((?x SOURCE)) TRUE
        (format t " %3s " (sym-cat ?x)))
   (printout t " | ")
   (do-for-all-instances ((?x LED)) TRUE
        (format t " %3s " (sym-cat ?x)))
   (format t "%n")
   (do-for-all-instances ((?x SOURCE)) TRUE (printout t "-----"))
   (printout t "-+-")
   (do-for-all-instances ((?x LED)) TRUE (printout t "-----"))
   (format t "%n"))
; S-1  S-2  |  L-1  L-2
;-----------+----------
;  *    1   |   1    0
;  *    0   |   1    1
```

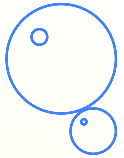






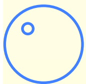
© José Angel Bañares

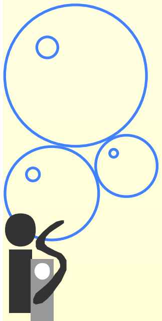


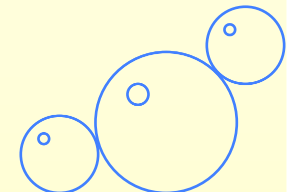


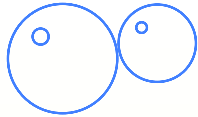
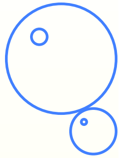
Imprime resultado

```
(defrule print-result
;;; Imprime ordenadamente salidas de menor a mayor
   (print-results)
   ?f <- (result $?input ?response)
   (not (result $?input-2
          ?response-2&:(< (str-compare ?response-2 ?response) 0)))
   =>
   (retract ?f)
   ;; Imprime la entrada para las fuentes.
   (while (neq ?input (create$)) do
      (printout t "  " (nth 1 ?input) "  ")
      (bind ?input (rest$ ?input)))
   ;; Imprime la salida para los lEDS.
   (printout t " | ")
   (bind ?response (str-explode ?response))
   (while (neq ?response (create$)) do
      (printout t "  " (nth 1 ?response) "  ")
      (bind ?response (rest$ ?response)))
   (printout t crlf))
```





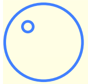
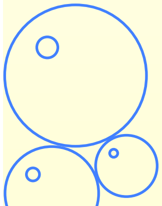


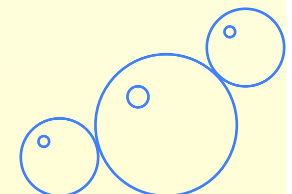



© José Angel Bañares

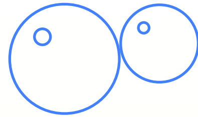
Circuito ejemplo

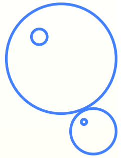
```
;;;======================================================
;;; LEGEND
;;; ------------
;;; S = Source
;;; P = Splitter
;;; N = NOT Gate
;;; O = OR Gate
;;; X = XOR Gate
;;; L = LED
;;;
;;;          /---N1>--\           /---------L1
;;; S1>--P1>-|         O1>---P2>--|
;;;          \--------/           |
;;;                               |
;;;                               |
;;;                               \---\
;;;                                    X2>--L2
;;; S2>-------------------------------/
;;;
;;;======================================================
```



© José Angel Bañares





circuit1.clp

```
(definstances circuit
  (S-1 of SOURCE) (S-2 of SOURCE)
  (P-1 of SPLITTER)
  (P-2 of SPLITTER)
  (N-1 of NOT-GATE)
  (O-1 of OR-GATE)
  (X-1 of XOR-GATE)
  (L-1 of LED) (L-2 of LED))

(deffunction connect-circuit ()
  (connect [S-1] [P-1])
  (connect [S-2] [X-1] 2)
  (connect [P-1] 1 [N-1])
  (connect [P-1] 2 [O-1] 2)
  (connect [N-1] [O-1] 1)
  (connect [O-1] [P-2])
  (connect [P-2] 1 [L-1])
  (connect [P-2] 2 [X-1] 1)
  (connect [X-1] [L-2]))
```