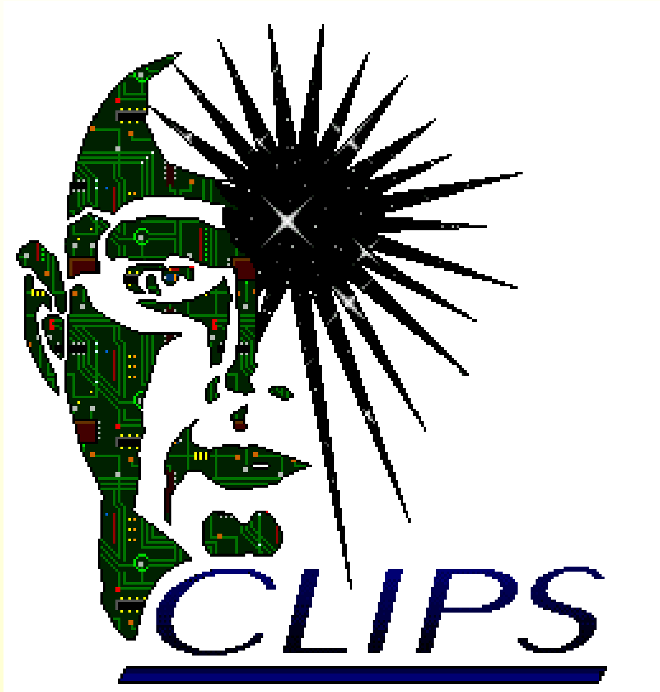
© José Angel Bañares

Lenguajes basados en reglas



Representación del Conocimiento
- Representación de hechos
- Representación de reglas

Motor de Inferencia

© José Angel Bañares 23/09/04

© José Angel Bañares

Índice

1. Sistemas de Producción/ Lenguajes Basados en reglas
2. Arquitectura de los Lenguajes Basados en Reglas
3. Representación Memoria de trabajo y producción
4. Proceso de Reconocimiento. Patrones con variables.
5. El proceso de razonamiento
   - 5.1 Encadenamiento progresivo
   - 5.2 Encadenamiento regresivo
   - 5.3 Encadenamiento/Razonamiento
   - 5.4 Estrategias de control
6. Ventajas y desventajas de los LBR
7. El sistema CLIPS

© José Angel Bañares

Bibliografía

- Peter Jackson. “Introduction to Expert System”. Second Edition. Addison Wesley, 1990.
- Avelino J. Gonzalez and Douglas D. Dankel. “The Engineering of Knowledge Bases Systems”. Prentice Hall 1993.
- Brownston y col. “Programming Expert Systems in OPS5”. Addison Wesley, 1985.

© José Angel Bañares

1. Sistemas de Producción / Lenguajes basados en reglas

- **Un Sistema de Producción ofrece un** mecanismo de control **dirigido por patrones para la resolución de problemas**
  - Representación del conocimiento experto

Se puede apreciar que las nubes están oscuras y está levantándose viento, y cuando se dan estos hechos siempre llueve copiosamente.

```
(defrule lluvia
    (nubes (color oscuro))
    (atmosfera (viento moderado))
 =>
    (assert (fenomeno (lluvia))))
```

Patrón: `(nubes (color oscuro))`, `(atmosfera (viento moderado))`

Acción: `(assert (fenomeno (lluvia)))`

© José Angel Bañares

Requisitos para un Sistema de Producción

– Herramienta para implementar búsquedas en el espacio de estados
  - **Representación del estado del sistema**

```
(deftemplate estado (slot garrafa (type INTEGER)))
```

  - **Estado Inicial**

```
(estado (garrafa 0))
```

  - **Estado Final**

```
(estado (garrafa 3))
```

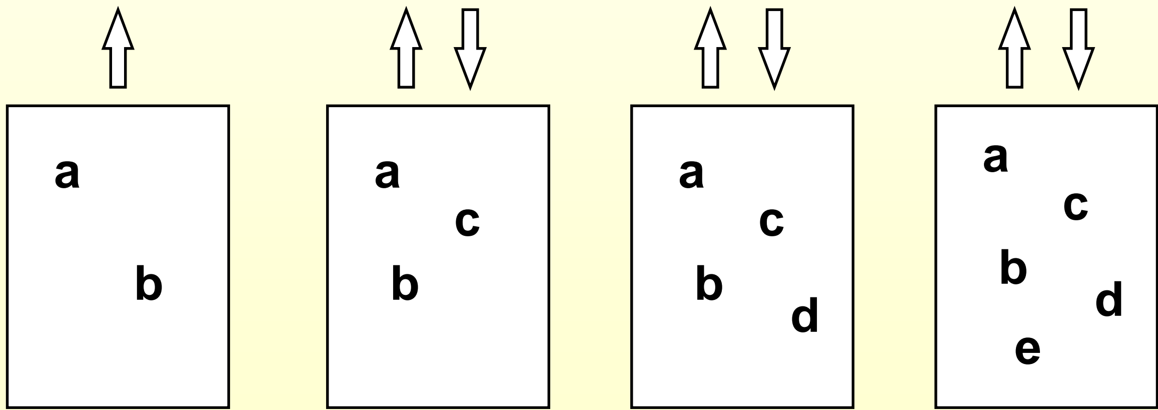
  - **OPERADORES**: Reglas de producción actuando sobre estados

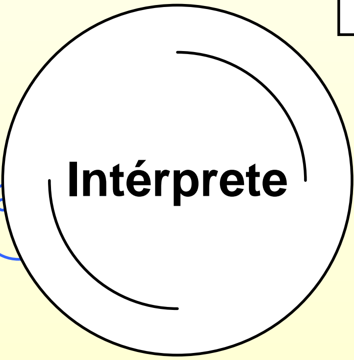
```
(defrule Agnade-Un-Litro
 ?estado <- (estado (garrafa ?cantidad))
   =>
(modify ?estado (garrafa (+ ?cantidad 1))))
```

© José Angel Bañares

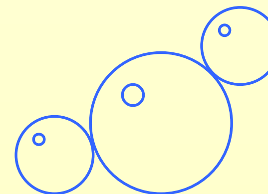
Inferencia en lenguajes basados en reglas

- **Una regla no aporta mucho, pero un conjunto de reglas pueden formar una cadena capaz de alcanzar una conclusión significativa.**





a = La temperatura ambiente es de 32º
b = La humedad relativa es mayor que el 65%
c = El tiempo es caluroso
d = La atmósfera está húmeda
e = se va formar una tormenta



© José Angel Bañares

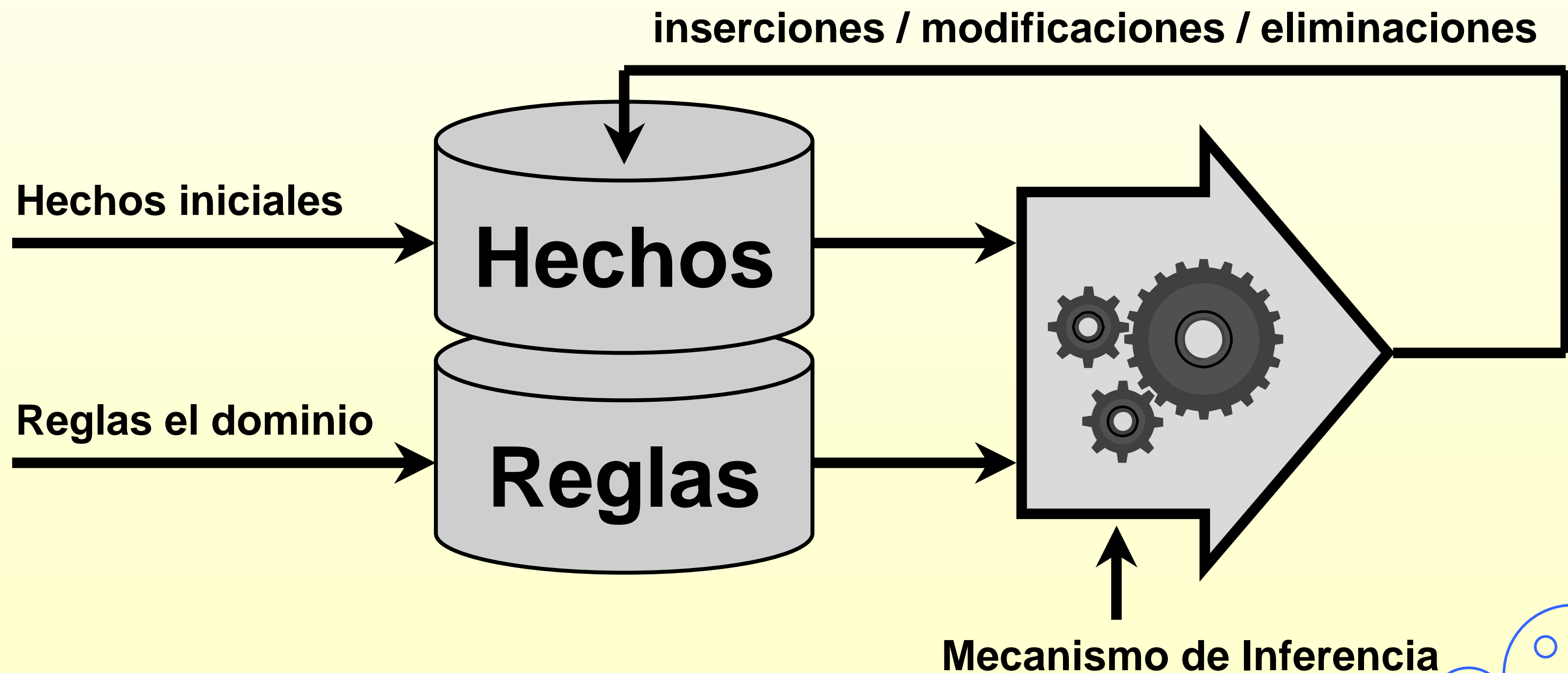
Cuando utilizar un lenguaje basado en reglas

- **Problema :**
  - No siempre es fácil (e incluso posible) obtener la manera de resolver un problema mediante una solución algorítmica
  - Se busca simular el razonamiento humano en dominios en los que el conocimiento es "evolutivo" y en los cuales no existe un método determinista seguro:
    - Ejemplos en las finanzas, la economía, las ciencias sociales
    - Ejemplos en la medicina (diagnóstico medico), en el mantenimiento
    - Ejemplos de demostración automática de teoremas, …
- **Método :**
  - Aislar y modelar un subconjunto del mundo real
  - Modelar el problema en términos de hechos iniciales o de objetivos a conseguir.
  - Ejecutar un programa que simula el razonamiento humano.

© José Angel Bañares

2. Arquitectura de los lenguajes basados en reglas

- **Base de hechos :** Contiene hechos iniciales, más los deducidos
- **Base de Reglas :** Contienen las reglas que explotan los hechos
- **Motor de inferencia :** Aplica las reglas a los hechos

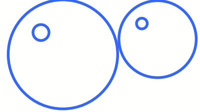


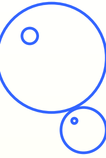
© José Angel Bañares

Dos partes de un Sistema Basado en Reglas

- **Parte declarativa :**
  - **Hechos** : conocimiento declarativo (información) sobre el mundo real
  - **Reglas** : conocimiento declarativo de la gestión de la base de hechos de gestión de la base de hechos
    - En lógica *monótona* : únicamente se permiten añadir hechos
    - En lógica *no-monótona* : inserciones, modificaciones y eliminación de hechos
  - **Meta-reglas** : conocimiento declarativo sobre el empleo de las reglas
- **Parte algorítmica/imperativa :**
  - **Motor de inferencias** : Software que efectúa los *razonamientos* sobre el conocimiento declarativo disponible :
    - Aplica las reglas de la memoria de producción a los hechos en memoria de trabajo
    - Se basa en uno o varios esquemas de razonamiento (ex : deducción)
    - Se puede consultar la traza del proceso de razonamiento:
      - Durante las fases de diseño y depuración
      - Para obtener explicaciones sobre la solución

© José Angel Bañares





Ejemplos de deducción

Modus Ponens — Modus Tollens

Inferencias sin variables :

Modus Ponens

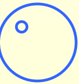
(hombre Sócrates) es cierto
(hombre Sócrates) → (mortal Sócrates)

(mortal Sócrates) es cierto

Modus Tollens

(mortal Agustín) es falso
(hombre Agustín) → (mortal Agustín)

(hombre Agustín) es falso



Inferencias con variables :

Modus Ponens

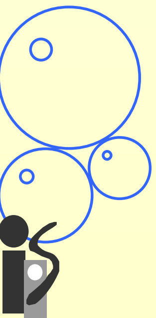
(hombre Sócrates) es cierto
(hombre ?x) → (mortal ?x)

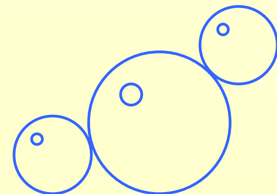
(mortal Sócrates) es cierto

Modus Tollens

(mortal Agustín) es falso
(hombre ?x) → (mortal ?x)

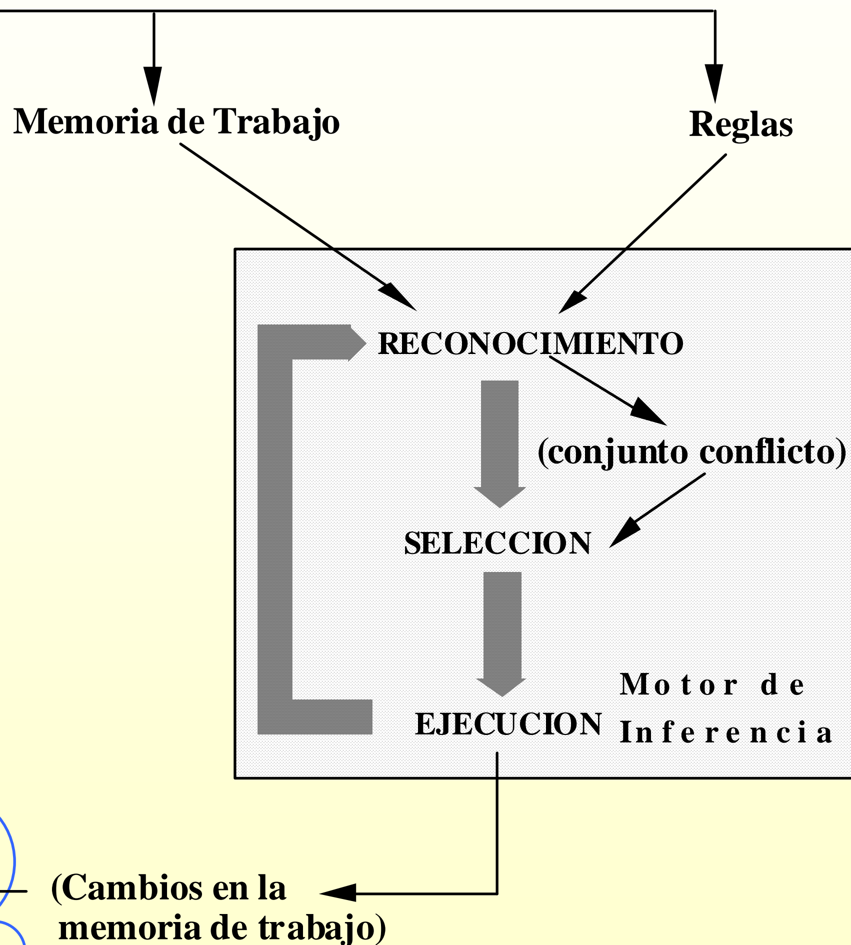
(hombre Agustín) es falso





© José Angel Bañares

Motor de Inferencia (OPS5/CLIPS)



- **La interpretación de las reglas conlleva los siguiente pasos básicos:**
  - Reconocimiento: Comparación de los patrones en las reglas con los elementos de la memoria de trabajo.
  - Resolución de conflictos: Se elige una regla entre las satisfechas por la memoria de trabajo y se ejecuta su parte ENTONCES.
  - Ejecución: La ejecución de las reglas da lugar a cambios en la memoria de trabajo. (También podrían añadirse nuevas reglas)
- **La estrategia de control especifica La forma de resolver conflictos**

© José Angel Bañares

3. Representación Hechos y Reglas

- **MEMORIA DE TRABAJO, Representación de hechos en CLIPS:**
  - Pepe nació en Zaragoza en 1966

```
(Persona (Nombre Pepe) (Edad 30) (Trabajo Ninguno))
```

- **MEMORIA DE PRODUCCION, Representación de reglas en CLIPS**
  - antecedentes **que reconocen estructuras de símbolos**
  - consecuentes **que contienen operadores especiales que manipulan las estructuras de símbolos.**

```
(defrule Pepe-desempleado
   (Persona (Nombre Pepe) (Edad 30) (Trabajo ninguno))
 =>
   (assert (Tarea (Reclamar Paro) (Para Pepe))))

(defrule Inscripcion-paro
   (Tarea (Reclamar Paro) (Para Pepe))))
 =>
   (assert (Inscrito_Paro (Nombre Pepe))))
```

© José Angel Bañares

4. Sistemas de Producción. Patrones con variables

1. Variables en las reglas CLIPS

- variable ::= ?<nombre>

```
CLIPS> (deftemplate personaje  (slot nombre) (slot ojos)
                              (slot pelo))
CLIPS>(defrule busca-ojos-azules
        (personaje (nombre ?nombre) (ojos azules))
             =>
        (printout t ?nombre " tiene los ojos azules." crlf))
CLIPS> (deffacts gente
(personaje (nombre Juan) (ojos azules) (pelo castagno))
(personaje (nombre Luis) (ojos verdes) (pelo rojo))
(personaje (nombre Pedro) (ojos azules) (pelo rubio))
(personaje (nombre Maria) (ojos castagnos) (pelo negro)))
CLIPS> (reset)
CLIPS> (run)
Pedro tiene los ojos azules.
Juan tiene los ojos azules.
```

© José Angel Bañares

Reconocimiento de patrones

```
(defrule COMER
          (HAMBRIENTO ?PERSONA)
          (COMESTIBLE ?ALIMENTO)
=>
(assert (Come ?PERSONA el ?ALIMENTO))
```

BASE de REGLAS

H1: (HAMBIENTO PEDRO)
H2: (HAMBRIENTO PABLO)
H3:(COMESTIBLE MANZANA)
H4:(COMESTIBLE MELOCOTON)
H5:(COMESTIBLE PERA)

BASE de HECHOS

I1: (?PERSONA = PEDRO, ?ALIMENTO = MANZANA) (H1,H3)
I2: (?PERSONA = PEDRO, ?ALIMENTO = MELOCOTON (H1,H4)
I3: (?PERSONA = PEDRO, ?ALIMENTO = PERA (H1,H5)
I4: (?PERSONA = PABLO, ?ALIMENTO = MANZANA (H2,H3))
I5: (?PERSONA = PABLO, ?ALIMENTO = MELOCOTON (H2 ,H4))
I6 :(?PERSONA = PABLO, ?ALIMENTO = PERA (H2 ,H5))

© José Angel Bañares

Variable que se repiten en distintos patrones

2. Variables en las reglas CLIPS: Restricción por ligaduras consistentes

```
CLIPS> (undefrule *)
CLIPS>
(deftemplate busca (slot ojos))
CLIPS>
(defrule busca-ojos
  (busca (ojos ?color-ojos))
  (personaje (nombre ?nombre) (ojos ?color-ojos))
  =>
  (printout t ?nombre " tiene los ojos " ?color-ojos "."  crlf))
CLIPS> (reset)
CLIPS> (assert (busca (ojos azules)))
<Fact-5>
CLIPS> (run)
Pedro tiene los ojos azul.
Juan tiene los ojos azul.
```

© José Angel Bañares

Funciones avanzadas para la correspondencia

- **Ligar un hecho a una variable**

```
CLIPS> (deftemplate persona (slot nombre) (slot direccion))
CLIPS>  (deftemplate cambio (slot nombre) (slot direccion))
CLIPS>(defrule procesa-informacion-cambios
         ?h1 <- (cambio (nombre ?nombre) (direccion ?direccion))
         ?h2 <- (persona (nombre ?nombre))
       =>
         (retract ?h1)
         (modify ?h2 (direccion ?direccion)))
CLIPS> (deffacts ejemplo
  (persona (nombre "Pato Donald") (direccion "Disneylandia"))
  (cambio (nombre"Pato Donald") (direccion "Port Aventura")))
CLIPS> (reset)
CLIPS> (run)
CLIPS> (facts)
f-0  (initial-fact)
f-3  (persona (nombre "Pato Donald") (direccion "Port Aventura"))
For a total of 2 facts.
```

© José Angel Bañares

Comodines

- **Variable que no liga valor y reconoce cualquier valor: ?**

```
CLIPS> (clear)
CLIPS> (deftemplate persona (multislot nombre) (slot dni))
CLIPS>
(deffacts ejemplo
  (persona (nombre Jose L. Perez) (dni 22454322))
  (persona (nombre Juan Gomez) (dni 23443325)))
CLIPS>
(defrule imprime-dni
   (persona (nombre ? ? ?Apellido) (dni ?dni))
  =>
  (printout t ?dni " " ?Apellido crlf))
CLIPS> (reset)
CLIPS> (run)
22454322 Perez
```

© José Angel Bañares

Comodines para varios valores

- **Comodín que reconoce cero o más valores de un atributo multivalor: $?**

```
CLIPS>
(defrule imprime-dni
   (persona (nombre $?nombre ?Apellido) (dni ?dni))
  =>
  (printout t ?dni " " ?nombre " " ?Apellido crlf))
CLIPS> (reset)
CLIPS> (run)
23443325 (Juan) Gomez
22454322 (Jose L.) Perez
```

© José Angel Bañares

Restricciones sobre el valor de un atributo

- **Operador ~**

```
(persona (nombre ?nombre) (pelo ~rubio))
```

- **Operador |**

```
(persona (nombre ?nombre) (pelo castagno | pelirojo))
```

- **Operador &** (En combinación con los anteriores para ligar valor a una variable)

```
(defrule pelo-castagno-o-rubio
  (persona (nombre ?nombre) (pelo  ?color&rubio|castagno))
   =>
  (printout t ?nombre " tiene el pelo " ?color crlf))
```

© José Angel Bañares

Restricciones sobre el valor

- **Predicados sobre el valor de un atributo**

```
(defrule mayor-de-edad
  (persona (nombre $?nombre) (edad ?edad&:(> ?edad 18))
   =>
  (printout t ?nombre " es mayor de edad. Edad:"  ?edad crlf))
```

- **Predicados sobre el valor de un atributo (igualdad)**

```
(defrule mayor-de-edad
  (persona (nombre $?nombre) (edad =(+ 18 2))
   =>
  (printout t ?nombre
      " tiene dos angos sobre la mayoria de edad." crlf))
```

- **Función Test**

```
(test (> ?edad  18))
```

- **Función Bind** (liga un valor no obtenido mediante reconocimiento a una variable)

```
(bind ?suma (+ ?a ?b))
```

© José Angel Bañares

Patrones en sistemas basados en reglas

- **Utilización de implementaciones sofisticadas de algoritmos para reconocimiento de patrones para**
  - ligar valores a variables,
  - restringir los valores que pueden tomar las variables
- **Las relaciones entre las reglas y los hechos se determinan en tiempo de ejecución.**
  - Se utilizan hechos con múltiples valores (tuplas objeto-atributos-valor), y patrones con variables que pueden reconocer diferentes hechos.
  - Los lenguajes basados en reglas con patrones permiten elaborar restricciones complejas para la identificación de hechos en la base de datos.

© José Angel Bañares

5. El proceso de razonamiento (adelante y atrás)

- **El proceso de razonamiento es una progresión desde un conjunto de datos hacia una solución, respuesta o conclusión.**
- **Dos situaciones posibles:**
  - Hay pocos datos iniciales, y muchas soluciones conclusiones. Lo razonable es progresar desde los datos iniciales hasta una solución.
  - Hay muchos datos iniciales, pero solo unos pocos son relevantes
    - *Por ejemplo, cuando vamos al médico solo le contamos los síntomas anormales (dolor de cabeza, nauseas). El médico intentará probar hipótesis preguntando cuestiones adicionales.*
- **Razonamiento dirigido por los datos o encadenamiento progresivo:**
  - Comienza con todos los datos conocidos y progresa hasta la conclusión
- **Razonamiento dirigido por los objetivos o encadenamiento regresivo:**
  - Selecciona una conclusión posible e intenta probar su validez buscando evidencias que la soporten.

© José Angel Bañares

Encadenamiento progresivo y regresivo

- **Las siguientes reglas de producción pueden utilizarse de dos formas:**
  - (**P1**) $ -> a$a
  - (**P2**) $ -> b$b
  - (**P3**) $ -> c$c
  - Encadenamiento hacia adelante: utilizar las reglas para generar palíndromos. Dado cualquier símbolo inicial como p.e. **c** , y la secuencia de reglas **P1**, **P2**, **P3**, **P2**, **P3**, generará la siguiente secuencia de cadenas:

    aca bacab cbacabc bcbacabcb cbcbacabcbc

  - Encadenamiento hacia atrás: utilizar las reglas para reconocer palíndromos. Dado un palíndromo como **bacab**, podemos trazar la secuencia de reglas que llevan a su construcción:

    bacab satisface la parte derecha de **P2,** la parte izquierda de **P2** que daría **bacab** es **aca,**

    aca satisface la parte derecha de **P1,** y la parte izquierda de **P1** que da **aca** es el símbolo **c.**

© José Angel Bañares

Ejemplo encadenamiento

```
REGLAS PARA IDENTIFICAR FRUTA

Regla 1: SI Forma = alargada y
            Color = verde o amarillo
         ENTONCES Fruta = banana
Regla 2: SI Forma = redonda u ovalada
            Diametro > 1.6 cm
         ENTONCES claseFruta = planta
Regla 3: SI Forma = redonda y
           Diametro < 1.6 cm
         ENTONCES claseFruta = árbol
Regla 4: SI numSemillas = 1
         ENTONCES claseSemilla = hueso
Regla 5: SI numSemillas > 1
      ENTONCES claseSemilla = multiple
Regla 6: SI claseFruta = planta y
            Color = verde
         ENTONCES Fruta = sandía
Regla 7: SI Forma = planta y
            Color = amarillo
         ENTONCES Fruta = melón

Regla 8: SI claseFruta = árbol y
            Color = naranja y
            claseSemilla = hueso
         ENTONCES Fruta = albaricoque
Regla 9: SI claseFruta = árbol y
            Color = naranja y
            claseSemilla = multiple
         ENTONCES Fruta = naranja
Regla 10: SI claseFruta = árbol y
             Color = rojo y
             claseSemilla = hueso
          ENTONCES Fruta = cereza
Regla 11: SI claseFruta = árbol y
             Color = naranja y
             claseSemilla = hueso
          ENTONCES Fruta = melocotón
Regla 12: SI claseFruta = árbol y
       Color = rojo o amarillo o verde y
             claseSemilla = múltiple
          ENTONCES Fruta = manzana
Regla 13: SI claseFruta = árbol y
            Color = morado y
            claseSemilla = hueso
         ENTONCES Fruta = ciruela
```

© José Angel Bañares

5.1 Intérprete con encadenamiento progresivo

- **Pasos del intérprete**
  1. Reconocimiento: Encuentra reglas aplicables y márcalas.
  2. Resolución de conflictos: Desactiva reglas que no añadan hechos nuevos.
  3. Acción: Ejecuta la acción de la regla **aplicable con menor número**. Si no hay reglas aplicables se detiene el intérprete.
  4. Reset: Vacía la lista de reglas aplicables y vuelve al paso 1.
- **Si la memoria de trabajo tiene los siguientes hechos iniciales:**

```
Diametro = 0.4 cm, forma = redonda, Numsemillas = 1, color = rojo
```

Ciclo de ejecución	*Reglas aplicables*	*regla seleccionada*	*Hecho derivado*
1	3,4	3	claseFruta = árbol
2	3,4	4	claseSemilla = hueso
3	3,4,10	10	Fruta = cereza
4	3,4, 10	_	

© José Angel Bañares

5.1 Intérprete con encadenamiento progresivo

- **Pasos del intérprete**
  1. Reconocimiento: Encuentra reglas aplicables y márcalas.
  2. Resolución de conflictos: Desactiva reglas que no añadan hechos nuevos.
  3. Acción: Ejecuta la acción de la regla **aplicable con menor número**. Si no hay reglas aplicables se detiene el intérprete.
  4. Reset: Vacía la lista de reglas aplicables y vuelve al paso 1.
- **Si la memoria de trabajo tiene los siguientes hechos iniciales:**

```
Diametro = 0.4 cm, forma = redonda, Numsemillas = 1, color = rojo
```

Ciclo de ejecución	*Reglas aplicables*	*regla seleccionada*	*Hecho derivado*
1	3,4	3	claseFruta = árbol
2	3,4	4	claseSemilla = hueso
3	3,4,10	10	Fruta = cereza
4	3,4, 10	_	

© José Angel Bañares

Diametro = 0.4 cm, forma = redonda, Numsemillas = 1, color = rojo

REGLAS PARA IDENTIFICAR FRUTA

```
Regla 1: SI Forma = alargada y
            Color = verde o amarillo
         ENTONCES Fruta = banana
Regla 2: SI Forma = redonda u ovalada
          y Diametro > 1.6 cm
         ENTONCES claseFruta = planta
Regla 3: SI Forma = redonda y
          Diametro < 1.6 cm
         ENTONCES claseFruta = árbol
Regla 4: SI numSemillas = 1
         ENTONCES claseSemilla = hueso
Regla 5: SI numSemillas > 1
      ENTONCES claseSemilla = multiple
Regla 6: SI claseFruta = planta y
            Color = verde
         ENTONCES Fruta = sandía
Regla 7: SI Forma = planta y
            Color = amarillo
         ENTONCES Fruta = melón
Regla 8: SI claseFruta = árbol y
            Color = naranja y
            claseSemilla = hueso
         ENTONCES Fruta = albaricoque
Regla 9: SI claseFruta = árbol y
            Color = naranja y
            claseSemilla = multiple
         ENTONCES Fruta = naranja
Regla 10: SI claseFruta = árbol y
             Color = rojo y
             claseSemilla = hueso
          ENTONCES Fruta = cereza
Regla 11: SI claseFruta = árbol y
             Color = naranja y
             claseSemilla = hueso
          ENTONCES Fruta = melocotón
Regla 12: SI claseFruta = árbol y
        Color = rojo o amarillo o verde y
             claseSemilla = múltiple
          ENTONCES Fruta = manzana
Regla 13: SI claseFruta = árbol y
            Color = morado y
            claseSemilla = hueso
         ENTONCES Fruta = ciruela
```

© José Angel Bañares

5.1 Intérprete con encadenamiento progresivo

- **Pasos del intérprete**
  1. Reconocimiento: Encuentra reglas aplicables y márcalas.
  2. Resolución de conflictos: Desactiva reglas que no añadan hechos nuevos.
  3. Acción: Ejecuta la acción de la regla **aplicable con menor número**. Si no hay reglas aplicables se detiene el intérprete.
  4. Reset: Vacía la lista de reglas aplicables y vuelve al paso 1.
- **Si la memoria de trabajo tiene los siguientes hechos iniciales:**

```
Diametro = 0.4 cm, forma = redonda, Numsemillas = 1, color = rojo
```

Ciclo de ejecución	*Reglas aplicables*	*regla seleccionada*	*Hecho derivado*
1	3,4	3	claseFruta = árbol
2	3,4	4	claseSemilla = hueso
3	3,4,10	10	Fruta = cereza
4	3,4, 10	_	

© José Angel Bañares

Diametro = 0.4 cm, forma = redonda, Numsemillas = 1, color = rojo, claseFruta=arbol

REGLAS PARA IDENTIFICAR FRUTA

```
Regla 1: SI Forma = alargada y
            Color = verde o amarillo
         ENTONCES Fruta = banana
Regla 2: SI Forma = redonda u ovalada
          y Diametro > 1.6 cm
         ENTONCES claseFruta = planta
Regla 3: SI Forma = redonda y
          Diametro < 1.6 cm
         ENTONCES claseFruta = árbol
Regla 4: SI numSemillas = 1
         ENTONCES claseSemilla = hueso
Regla 5: SI numSemillas > 1
      ENTONCES claseSemilla = multiple
Regla 6: SI claseFruta = planta y
            Color = verde
         ENTONCES Fruta = sandía
Regla 7: SI Forma = planta y
            Color = amarillo
         ENTONCES Fruta = melón
```

```
Regla 8: SI claseFruta = árbol y
            Color = naranja y
            claseSemilla = hueso
         ENTONCES Fruta = albaricoque
Regla 9: SI claseFruta = árbol y
            Color = naranja y
            claseSemilla = multiple
         ENTONCES Fruta = naranja
Regla 10: SI claseFruta = árbol y
             Color = rojo y
             claseSemilla = hueso
          ENTONCES Fruta = cereza
Regla 11: SI claseFruta = árbol y
             Color = naranja y
             claseSemilla = hueso
          ENTONCES Fruta = melocotón
Regla 12: SI claseFruta = árbol y
        Color = rojo o amarillo o verde y
             claseSemilla = múltiple
          ENTONCES Fruta = manzana
Regla 13: SI claseFruta = árbol y
            Color = morado y
            claseSemilla = hueso
         ENTONCES Fruta = ciruela
```

© José Angel Bañares

5.1 Intérprete con encadenamiento progresivo

- **Pasos del intérprete**
  1. Reconocimiento: Encuentra reglas aplicables y márcalas.
  2. Resolución de conflictos: Desactiva reglas que no añadan hechos nuevos.
  3. Acción: Ejecuta la acción de la regla **aplicable con menor número**. Si no hay reglas aplicables se detiene el intérprete.
  4. Reset: Vacía la lista de reglas aplicables y vuelve al paso 1.
- **Si la memoria de trabajo tiene los siguientes hechos iniciales:**

```
Diametro = 0.4 cm, forma = redonda, Numsemillas = 1, color = rojo
```

Ciclo de ejecución	*Reglas aplicables*	*regla seleccionada*	*Hecho derivado*
1	**3,4**	**3**	**claseFruta = árbol**
2	**3,4**	**4**	**claseSemilla = hueso**
3	3,4,10	10	Fruta = cereza
4	3,4, 10	_	

© José Angel Bañares

Diametro = 0.4 cm, forma = redonda, Numsemillas = 1, color = rojo
, claseSemilla=hueso, clasefruta= arbol

REGLAS PARA IDENTIFICAR FRUTA

Regla 1: SI Forma = alargada y
Color = verde o amarillo
ENTONCES Fruta = banana

Regla 2: SI Forma = redonda u ovalada
y Diametro > 1.6 cm
ENTONCES claseFruta = planta

Regla 3: SI Forma = redonda y
Diametro < 1.6 cm
ENTONCES claseFruta = árbol

Regla 4: SI numSemillas = 1 y
ENTONCES claseSemilla = hueso

Regla 5: SI numSemillas > 1
ENTONCES claseSemilla = multiple

Regla 6: SI claseFruta = planta y
Color = verde
ENTONCES Fruta = sandía

Regla 7: SI Forma = planta y
Color = amarillo
ENTONCES Fruta = melón

Regla 8: SI claseFruta = árbol y
Color = naranja y
claseSemilla = hueso
ENTONCES Fruta = albaricoque

Regla 9: SI claseFruta = árbol y
Color = naranja y
claseSemilla = multiple
ENTONCES Fruta = naranja

Regla 10: SI claseFruta = árbol y
Color = rojo y
claseSemilla = hueso
ENTONCES Fruta = cereza

Regla 11: SI claseFruta = árbol y
Color = naranja y
claseSemilla = hueso
ENTONCES Fruta = melocotón

Regla 12: SI claseFruta = árbol y
Color = rojo o amarillo o verde y
claseSemilla = múltiple
ENTONCES Fruta = manzana

Regla 13: SI claseFruta = árbol y
Color = morado y
claseSemilla = hueso
ENTONCES Fruta = ciruela

© José Angel Bañares

5.1 Intérprete con encadenamiento progresivo

- **Pasos del intérprete**
  1. Reconocimiento: Encuentra reglas aplicables y márcalas.
  2. Resolución de conflictos: Desactiva reglas que no añadan hechos nuevos.
  3. Acción: Ejecuta la acción de la regla **aplicable con menor número**. Si no hay reglas aplicables se detiene el intérprete.
  4. Reset: Vacía la lista de reglas aplicables y vuelve al paso 1.
- **Si la memoria de trabajo tiene los siguientes hechos iniciales:**

```
Diametro = 0.4 cm, forma = redonda, Numsemillas = 1, color = rojo
```

Ciclo de ejecución	*Reglas aplicables*	*regla seleccionada*	*Hecho derivado*
1	**3,4**	**3**	**claseFruta = árbol**
2	**3,4**	**4**	**claseSemilla = hueso**
3	**3,4,10**	**10**	**Fruta = cereza**
4	3,4, 10	_	

© José Angel Bañares

5.2 Encadenamiento regresivo

- **Se comienza por un objetivo (conclusión que se desea probar) y decide si los hechos soportan el objetivo.**
  - Se comienza con una **base de hechos**: ( ), que suele estar vacía y
  - **una lista de objetivos** para la que el sistema intenta derivar hechos. Los objetivos se ordenan de forma que sean los más fácilmente alcanzables primero. Por ejemplo, en el problema de identificar fruta el objetivo es dar valor al parámetro "Fruta". **Objetivos**: (Fruta).
  - El encadenamiento regresivo utiliza la lista de objetivos para coordinar su búsqueda a través de la base de reglas.

© José Angel Bañares

Pasos del intérprete con encadenamiento regresivo

1. **Forma una pila con todos los objetivos iniciales.**
2. **Reunir todas las reglas capaces de satisfacer el primer objetivo.**
3. **Para cada una de estas reglas, examinar sus premisas:**
   a) Si las premisas son satisfechas, entonces se ejecuta esta regla para derivar sus conclusiones. Elimina el objetivo de la pila y vuelve al paso 2.
   b) Si una de las premisas no se cumple, busca las reglas que pueden derivar esta premisa. Si se encuentra alguna regla, entonces se considera la premisa como subobjetivo, se coloca éste al principio de la pila, y se va al paso 2.
   c) Si el paso b no puede encontrar una regla que derive el valor especificado para el objetivo en curso, entonces preguntar al usuario por el valor del parámetro, y añade éste a la memoria de trabajo. Si este valor satisface la premisa en curso, continúa con la siguiente premisa de esta regla. Si la premisa en curso no queda satisfecha por el valor continúa con la siguiente regla.
4. **Si todas las reglas que pueden satisfacer el objetivo actual se han intentado y han fallado, entonces el objetivo en curso permanece indeterminado. Saca éste de la pila y vuelve al paso 2. Si la pila de objetivos está vacia, el intérprete se detiene.**

© José Angel Bañares

Ejemplo de encadenamiento regresivo I

- **Supongamos que queremos examinar una cereza. La traza de ejecución de las reglas para ver si son capaces de derivar cereza como valor de fruta se como sigue:**
  - Paso 1. `Objetivos:  (Fruta)`
  - Paso 2. La lista de reglas que pueden satisfacer este objetivo son: 1, 6,7, 8, 9, 10, 11, 12, y 13.
  - Paso 3.
    - Se considera regla 1: La primera premisa (Forma= alargada) no se encuentra en la memoria de trabajo. No hay reglas que deriven éste valor así que el intérprete pregunta por este valor:
      - `¿Cuál es el valor de Forma?  redondo.`
      - `Memoria de Trabajo: ((Forma = redondo))`
    - Se considera regla 6: La primera premisa de esta regla es (claseFruta = planta), y no se encuentra la memoria de trabajo. Reglas 2 y 3 pueden derivar éste valor, así que añadimos claseFruta en la lista de objetivos:
      - `Objetivos:  (claseFruta, Fruta)`

© José Angel Bañares

Objetivos: (Fruta)

Hechos: Forma= redondo

¿Cuál es el valor de forma?: Redondo

REGLAS PARA IDENTIFICAR FRUTA

Regla 1: SI Forma = alargada y
Color = verde o amarillo
ENTONCES Fruta = banana

Regla 2: SI Forma = redonda u ovalada
Diametro > 1.6 cm
ENTONCES claseFruta = planta

Regla 3: SI Forma = redonda y
Diametro < 1.6 cm
ENTONCES claseFruta = árbol

Regla 4: SI numSemillas = 1
ENTONCES claseSemilla = hueso

Regla 5: SI numSemillas > 1
ENTONCES claseSemilla = multiple

Regla 6: SI claseFruta = planta y
Color = verde
ENTONCES Fruta = sandía

Regla 7: SI Forma = planta y
Color = amarillo
ENTONCES Fruta = melón

Regla 8: SI claseFruta = árbol y
Color = naranja y
claseSemilla = hueso
ENTONCES Fruta = albaricoque

Regla 9: SI claseFruta = árbol y
Color = naranja y
claseSemilla = multiple
ENTONCES Fruta = naranja

Regla 10: SI claseFruta = árbol y
Color = rojo y
claseSemilla = hueso
ENTONCES Fruta = cereza

Regla 11: SI claseFruta = árbol y
Color = naranja y
claseSemilla = hueso
ENTONCES Fruta = melocotón

Regla 12: SI claseFruta = árbol y
Color = rojo o amarillo o verde y
claseSemilla = múltiple
ENTONCES Fruta = manzana

Regla 13: SI claseFruta = árbol y
Color = morado y
claseSemilla = hueso
ENTONCES Fruta = ciruela

© José Angel Bañares

¿Cuál es el valor de Diametro?:0.4

Objetivos: (ClaseFruta ,Fruta)

Hechos: Forma= redondo , Diametro = 0.4

REGLAS PARA IDENTIFICAR FRUTA

```
Regla 1: SI Forma = alargada y        [
            Color = verde o amarillo
         ENTONCES Fruta = banana
Regla 2: SI Forma = redonda u ovalada
            Diametro > 1.6 cm
         ENTONCES claseFruta = planta
Regla 3: SI Forma = redonda y
           Diametro < 1.6 cm
         ENTONCES claseFruta = árbol
Regla 4: SI numSemillas = 1
         ENTONCES claseSemilla = hueso
Regla 5: SI numSemillas > 1
      ENTONCES claseSemilla = multiple
Regla 6: SI claseFruta = planta y
            Color = verde
         ENTONCES Fruta = sandía
Regla 7: SI Forma = planta y
            Color = amarillo
         ENTONCES Fruta = melón
```

```
Regla 8: SI claseFruta = árbol y
            Color = naranja y
            claseSemilla = hueso
         ENTONCES Fruta = albaricoque
Regla 9: SI claseFruta = árbol y
            Color = naranja y
            claseSemilla = multiple
         ENTONCES Fruta = naranja
Regla 10: SI claseFruta = árbol y
             Color = rojo y
             claseSemilla = hueso
          ENTONCES Fruta = cereza
Regla 11: SI claseFruta = árbol y
             Color = naranja y
             claseSemilla = hueso
          ENTONCES Fruta = melocotón
Regla 12: SI claseFruta = árbol y
       Color = rojo o amarillo o verde y
             claseSemilla = múltiple
          ENTONCES Fruta = manzana
Regla 13: SI claseFruta = árbol y
            Color = morado y
            claseSemilla = hueso
         ENTONCES Fruta = ciruela
```

© José Angel Bañares

Ejemplo de encadenamiento regresivo II

- Examinamos regla 2, la primera premisa (Forma redondo o alargado) es satisfecha puesto que el valor de "Forma" es redondo. Se continúa con la siguiente premisa, puesto que no existe un valor de diámetro ni se puede derivar de otras reglas se pregunta al usuario:
  - **¿Cuál es el valor del diametro? 0.4**
  - **Memoria de Trabajo:**
    ((Forma = redondo)(Diámetro = 0.4))
- La regla 2 falla. El intérprete lo intenta con la regla 3.Ambas premisas se cumplen, por lo que se deriva que claseFruta =árbol
  - **Memoria de Trabajo:**
    ((Forma = redondo)(Diámetro = 0.4)
    (claseFruta = árbol))
- Como se ha encontrado un valor para el el objetivo claseFruta se elimina éste de la lista de objetivos. Se vuelve al objetivo Fruta y a la regla 6. Falla la segunda premisa claseFruta=planta. Lo mismo ocurre con la regla 7.

© José Angel Bañares

Objetivos: (Fruta)

Hechos: Forma= redondo, Diametro = 0.4, claseFruta = arbol

REGLAS PARA IDENTIFICAR FRUTA

```
Regla 1: SI Forma = alargada y         [
            Color = verde o amarillo
         ENTONCES Fruta = banana
Regla 2: SI Forma = redonda u ovalada
            Diametro > 1.6 cm          [
         ENTONCES claseFruta = planta
Regla 3: SI Forma = redonda y
           Diametro < 1.6 cm
         ENTONCES claseFruta = árbol
Regla 4: SI numSemillas = 1
         ENTONCES claseSemilla = hueso
Regla 5: SI numSemillas > 1
      ENTONCES claseSemilla = multiple
Regla 6: SI claseFruta = planta y      [
            Color = verde
         ENTONCES Fruta = sandía
Regla 7: SI Forma = planta y
            Color = amarillo
         ENTONCES Fruta = melón
Regla 8: SI claseFruta = árbol y
            Color = naranja y
            claseSemilla = hueso
         ENTONCES Fruta = albaricoque
Regla 9: SI claseFruta = árbol y
            Color = naranja y
            claseSemilla = multiple
         ENTONCES Fruta = naranja
Regla 10: SI claseFruta = árbol y
             Color = rojo y
             claseSemilla = hueso
          ENTONCES Fruta = cereza
Regla 11: SI claseFruta = árbol y
             Color = naranja y
             claseSemilla = hueso
          ENTONCES Fruta = melocotón
Regla 12: SI claseFruta = árbol y
       Color = rojo o amarillo o verde y
             claseSemilla = múltiple
          ENTONCES Fruta = manzana
Regla 13: SI claseFruta = árbol y
            Color = morado y
            claseSemilla = hueso
         ENTONCES Fruta = ciruela
```

© José Angel Bañares

Ejemplo de encadenamiento regresivo III

- La regla 8 tiene su primera premisa satisfecha (claseFruta = árbol), la siguiente premisa color no está ni se puede derivar:
  - ¿Cuál es el valor del color? rojo
  - Memoria de Trabajo:
    ((Forma = redondo)(Diámetro = 0.4)
    (claseFruta = árbol)(color rojo))
- Fallan reglas 8 y 9 porque sus premisas "color" no son rojo. La regla 10 cumple las 2 primeras premisas ((claseFruta = arbol) y (color = rojo)). No hay valor para la tercera premisa (claseSemilla = hueso) ni hay reglas que puedan derivarlo:
  - ¿Cuál es el valor de claseSemilla? hueso
  - Memoria de Trabajo:
    ((Forma = redondo)(Diámetro = 0.4)
    (claseFruta = árbol)(color rojo) (claseSemilla = hueso))
- La regla 10 es satisfecha completamente, se deriva el valor de fruta y queda la memoria de trabajo con el valor de fruta. La pila de objetivos se vacia:
  - Memoria de Trabajo:
    ((Forma = redondo)(Diámetro = 0.4)
    (claseFruta = árbol)(color rojo)
    (claseSemilla = hueso)(Fruta = cereza))
  - Objetivos:()

© José Angel Bañares

¿Cuál es el valor de color?: rojo

¿Cuál es el valor de claseSemilla?: hueso

Objetivos: (Fruta)

Hechos: Forma= redondo, Diametro = 0.4, claseFruta = arbol ,color = rojo ,claseSemilla = hueso

REGLAS PARA IDENTIFICAR FRUTA

Regla 1: SI Forma = alargada y
Color = verde o amarillo
ENTONCES Fruta = banana [

Regla 2: SI Forma = redonda u ovalada
Diametro > 1.6 cm [
ENTONCES claseFruta = planta

Regla 3: SI Forma = redonda y
Diametro < 1.6 cm
ENTONCES claseFruta = árbol

Regla 4: SI numSemillas = 1
ENTONCES claseSemilla = hueso

Regla 5: SI numSemillas > 1
ENTONCES claseSemilla = multiple

Regla 6: SI claseFruta = planta y [
Color = verde
ENTONCES Fruta = sandía

Regla 7: SI Forma = planta y
Color = amarillo
ENTONCES Fruta = melón

Regla 8: SI claseFruta = árbol y [
Color = naranja y
claseSemilla = hueso
ENTONCES Fruta = albaricoque

Regla 9: SI claseFruta = árbol y [
Color = naranja y
claseSemilla = multiple
ENTONCES Fruta = naranja

Regla 10: SI claseFruta = árbol y
Color = rojo y
claseSemilla = hueso
ENTONCES Fruta = cereza

Regla 11: SI claseFruta = árbol y
Color = naranja y
claseSemilla = hueso
ENTONCES Fruta = melocotón

Regla 12: SI claseFruta = árbol y
Color = rojo o amarillo o verde y
claseSemilla = múltiple
ENTONCES Fruta = manzana

Regla 13: SI claseFruta = árbol y
Color = morado y
claseSemilla = hueso
ENTONCES Fruta = ciruela

© José Angel Bañares

5.3 Distinción Razonamiento / Encadenamiento

- **Distinción entre razonamiento y encadenamiento progresivo y regresivo:**
  - Razonamiento bottom-up o progresivo: De los hechos a los objetivos
  - Encadenamiento progresivo: comparar la parte derecha (SI) de las reglas con los datos de la memoria de trabajo, y ejecutar la parte derecha (ENTONCES) de las reglas satisfechas.
  - Razonamiento top-down o regresivo: De los objetivos a los hechos:
  - Encadenamiento regresivo:Se comienza por un objetivo y se busca las reglas capaces de satisfacer el objetivo en su parte derecha. Para cada una de estas reglas se miran sus precondiciones.
- **Un sistema de producción como CLIPS tienen un mecanismo de encadenamiento hacia adelante. Sin embargo, en CLIPS se puede hacer razonamiento regresivo si se hace un control explícito del encadenamiento:**
  - El encadenamiento implementa el razonamiento
  - La estrategia de razonamiento puede controlar el encadenamiento

© José Angel Bañares

5.4 Estrategias de control

- **En cada ciclo de interpretación puede haber más de una instancia de regla candidata a la ejecución.**
- **Hay dos aproximaciones generales al control de los sistemas basados en reglas**
  - **Control global**: Control independiente del dominio de aplicación.
    - Estrategias implementadas en el intérprete
    - No son modificables por el programador
  - **Control local**: Control dependiente del dominio de aplicación.
    - Reglas especiales que permiten razonar sobre el control: METAREGLAS.
    - El programador escribe reglas explícitas para controlar el sistema.

© José Angel Bañares

Control Global

- **Determina que reglas participan en el proceso de reconocimiento de patrones y como se elegirá entre las instancias de regla en el caso de que exista más de una candidata.**
- **Dos mecanismos:**
  - **Estrategias de resolución del conjunto conflicto**: Estrategias para la selección de una regla del conjunto conflicto para ser disparada en cada ciclo.
  - **Proceso de filtrado**: Decide que reglas y con que datos se intenta realizar el proceso de reconocimiento de patrones.
    - En CLIPS se comparan todas las reglas con todos los hechos de la memoria de trabajo. Pero se pueden definir módulos.
    - En KEE, es posible agrupar las reglas en clases de forma que las inferencias se realicen solo con instancias de una cierta clase (de reglas).

© José Angel Bañares

Resolución de conflictos

- **Los criterios de selección persiguen que el sistema sea sensible y estable**
  (Brownston y col. 1985, Programming Expert Systems in OPS5, capítulo 7)
  - Sensible: Se responda a cambios reflejados en la memoria de trabajo
  - Estable: Haya continuidad en la línea de razonamiento.

- **Los mecanismos de resolución de conflicto son muy diversos, pero los siguientes criterios de selección son muy populares**
  - **Refracción(refraction)**: Una regla no debe poderse disparar más de una vez con los mismo hechos.
  - **Novedad (recency)**: Las instancias de reglas que utilizan hecho más recientes son preferidas a las que utilizan hechos más viejos.
  - **Especificidad**: Instancias derivadas de reglas más específicas (con mayor número de condiciones, es decir más difíciles de cumplir) son preferidas.
  - **Prioridades** asociadas a las reglas.

© José Angel Bañares

Estrategias LEX y MEA (OPS5, CLIPS)

- **LEX**
  1. Considera **refracción**
  2. Ordena instancias restantes de acuerdo a su novedad. Considera aquellas que tienen mayor valor de novedad y descarta el resto. Si lo hay, se mira el valor de novedad del siguiente elemento de la memoria de trabajo reconocido por la regla. Se continúa hasta que queda una única instancia, o todas las instancias reconocen hechos de la misma novedad
  3. Se aplica el principio de **especificidad** sobre el conjunto resultante.
  4. Si quedan más de una regla se elige una **aleatoriamente**.

- **MEA (Means End Analysis)**
  – Igual que la estrategia LEX, pero inmediatamente después de considerar la refracción, se mira la **novedad de la primera condición**. Si no hay una que domine, se pasa el conjunto restante por los mismos pasos que la estrategia LEX
  – MEA vienen de **means-ends** analysis. Facilita el manejo de subobjetivos. Si el primer elemento de la regla es siempre un objetivo, el sistema no se vera afectado por la novedad de un elemento reciente que no es un objetivo.

© José Angel Bañares

6. Ventajas y desventajas de los LBR

Ventajas

- Modularidad: Los lenguajes basados en reglas son muy modulares.
  - Cada regla es una unidad del conocimiento que puede ser añadida, modificada o eliminada independientemente del resto de las reglas.
  - Se puede desarrollar una pequeña porción del sistema, comprobar su correcto funcionamiento y añadirla al resto de la base de conocimiento.
- Uniformidad:
  - Todo el conocimiento es expresado de la misma forma.
- Naturalidad:
  - Las reglas son la forma natural de expresar el conocimiento en cualquier dominio de aplicación.
- Explicación:
  - La traza de ejecución permite mostrar el proceso de razonamiento
  - En CLIPS

```
(watch rules)
```

© José Angel Bañares

Ventajas y desventajas de los LBR

Desventajas

- Ineficiencia: La ejecución del proceso de reconocimiento de patrones es muy ineficiente.
- Opacidad: Es difícil examinar una base de conocimiento y determinar que acciones van a ocurrir.
  - La división del conocimiento en reglas hace que cada regla individual sea fácilmente tratable, pero se pierde la visión global.
- Dificultad en cubrir todo el conocimiento:
  - Aplicaciones como el control de tráfico aéreo implicarían una cantidad de reglas que no serían manejables.

© José Angel Bañares

7. El sistema CLIPS

- **Un entorno de desarrollo completo :**
  - Características :
    - Tres formalismos para la representación del conocimiento
    - Motor de inferencia con encadenamiento progresivo
    - Implementación en C del algoritmo de RETE (eficiente)
    - Lenguaje de control y de programación al estilo LISP
  - Modos de control :
    - Modo batch (EL sistema se lanza con un fichero de control)
    - Modo en línea (Depuración, uso amater)
    - Interfaz gráfica con varias ventanas (Windows, Macintosh, Unix)
- **Las claves del éxito:**
  - Software libre escrito en C (código legible y documentado)
  - Comunidad de desarrolladores muy activa
  - Sitios Web, varias extensiones, lista de distribución

© José Angel Bañares

Breve historia

- **CLIPS (C-Language Integrated Production System) :**
  - 1985 : release 1.0 (prototipo)
    - Sección de Inteligencia Artificial — Lyndon B. Johnson Space Center
    - Implementación en C del motor de inferencia más eficiente : OPS 5
  - 1986 : release 3.0 (primera difusión oficial)
  - 1988 : release 4.2 (reescritura completa del código)
  - 1991 : release 5.0 (aspectos procedurales + COOL)
    - Modelo de objetos inspirado en CLOS (Common Lisp Object System)
  - 1993 : release 6.0 (modulos + filtro de objetos COOL)
  - 1998 : release 6.1 (funciones de usuario + compatibilidad C++)
- **JESS (the Java Expert System Shell) :**
  - 1995 : release 1.0 (primera versión)
    - Objetivo : alternativa Java a CLIPS
  - 1999 : release 4.4 (última versión estable)

© José Angel Bañares

El mundo de los bloques

- **Se quiere escribir la parte I.A. de un robot móvil :**
  - El robot debe tomar decisiones (acciones) en función del conocimiento que dispone del entorno
  - Las decisiones se traducen en fines/objetivos a conseguir
  - Un objetivo puede dar lugar a subobjetivos (planificación)
  - Una acción modifica la configuración del entorno
- **Representación del conocimiento :**
  - Conocimientos del robot :
    - Conocimientos sobre el mundo y sobre el mismo
    - Implementados como hechos en memoria de trabajo
  - Razonamiento del robot :
    - Implementado como reglas del tipo (condiciones) → (acciones)
    - Las condiciones se sustentan en hechos iniciales o deducidos
    - Las acciones son modificaciones de los hechos

© José Angel Bañares

Conocimientos del robot

- **Representación de objetivos como listas :**

```
(objetivo Roby coger cubo)
(objetivo Roby ir-hacia almacen)
(objetivo <sujeto> <verbo> <complento>)
```

- **Se pueden representar las entidades como *plantillas* :**

```
(robot (nombre Roby) (localizacion hangar))
(objeto (nombre cubo) (localizacion almacen))
```

- **Justificación de las representaciones :**
  - Los objetivos son "casi" frases que se comprenden fácilmente
  - Los objetos son entidades con ciertas propiedades
  - Una lista como `(robot Roby hangar)` sería poco explícita
  - Una estructura tipo *plantilla (registro)* es a la larga más apropiada

© José Angel Bañares

Hechos ordenados

- **Hecho ordenado o *patrón* :**
  - Lista constituida por un símbolo (la relación) seguida de una secuencia (eventualmente vacía) de datos separados por espacios
    - Un símbolo es una secuencia de caracteres ASCII con la siguiente excepción : ( ) < > & $ | ; ? ~
      - Los hechos CLIPS en Windows no aceptan caracteres acentuados
    - Un dato debe ser de uno de los ocho tipos que permite CLIPS (ver tipología traspa 59)
- **Ejemplo de patrones :**

```
(flag)
(estado motor encendido)
(altitud 10000 metros)
(ventas 1999 3 4 6 8 7 5 4 3 9 5 4 2)
(objetivo Roby coger cubo)
```

© José Angel Bañares

Hechos estructurados

- **Hechos estructurados o *template (plantillas):***
  - Estructura de tipo *frame* que permiten especificar y acceder a los atributos (slots) de los hechos (clase de registro)
  - !Las *templates* se deben declarar antes de utilizarse !
    - El orden de los atributos no importa
    - Los atributos pueden ser monovaluados o multivaluados
      - Un atributo multivaluado se maneja como un hecho ordenado
    - Se puede especificar el dominio de cada atributo :
      - Tipo, intervalo, cardinalidad, valor por defecto, *etc*.
- **Ejemplos *plantillas :***

```
(cliente (nombre "Luis Sereno") (id X9345A))
(lista (tamano 3) (contenido pan leche huevos))
(coordenadas (x 10) (y 24) (z -12))
(robot (nombre Roby) (localizacion hangar))
```

© José Angel Bañares

Elección del tipo de hecho

- **Hechos ordenados :**
  - Ventaja : Se pueden utilizar sin declaración previa
  - Inconveniente : Ningún control sobre el tipo de datos
  - Inconveniente : poco explícito (fuente potencial de errores)
  - Atención : !La posición de un valor puede tener importancia !

```
(empleado "Fernandez" "Juan" 27 PATC)
```

- **Hechos estructurados :**
  - Ventaja : Mucho más explícitos que los hechos ordenados
  - Ventaja : Control de los tipos de datos
  - Inconveniente : Precisa de declaración previa

```
(empleado (nobre " Juan ")
          (Apellido " Fernandez ")
          (edad 27)
          (tipo-contrato PATC))
```

© José Angel Bañares

Creación de nuevos hechos

- **Hechos ordenados :**
  - Instrucción de creación : `(assert <pattern>+)`
  - Sin declaración previa
  - Ejemplo creación: `(assert (objetivo Roby coger cubo))`
- **Hechos estructurados :**
  - Idem mais il faut déclarer leur structure avant de les utiliser !
  - Creación previa de la plantilla (*template*) robot :

```
(deftemplate robot
  "Los robots del mundo de bloques"
  (slot nombre (type SYMBOL) (default ?NONE))
  (slot localizacíon (type SYMBOL) (default hangar))
  (slot sostiene (type SYMBOL) (default ?DERIVE)))
```

  - Ejemplo creación : `(assert (robot (nombre Roby)))`

© José Angel Bañares

Sintaxis deftemplate

```
(deftemplate <deftemplate-name> [<comment>] <slot-definition>*)

<slot-definition> ::= <single-slot-def> | <multislot-def>
<single-slot-def> ::= (slot <slot-name> <template-attrib>*)
<multislot-def> ::= (multislot <slot-name> <template-attrib>*)

<template-attrib> ::= <default-attrib> | <constraint-attrib>

<default-attrib> ::= (default ?DERIVE | ?NONE | <expression>*) |
                     (default-dynamic <expression>*)

<constraint-attrib> ::= (type <allowed-type>+) |
                        (allowed-values <value>+) |
                        (allowed-symbols <symbol>+) |
                        .........
                        (range <number> <number>) |
                        (cardinality <integer> <integer>)
```

© José Angel Bañares

Deftemplate

```
(deftemplate objeto
   "Las entidades manipulables"
   (slot nombre (type SYMBOL) (default ?NONE))
   (slot id (default-dynamic (gensym)))
   (slot loc (type SYMBOL) (default ?DERIVE))
   (slot peso (allowed-values ligero pesado) (default ligero))
   (multislot punteros (type FACT-ADDRESS) (default ?DERIVE)))
```

```
CLIPS> (assert (objeto))
[TMPLTRHS1] Slot nombre requires a value because of its (default ?NONE)
   attrib
CLIPS> (assert (objeto (nombre cubo)))
<fact-0>
CLIPS> (assert (objeto (nombre bloque) (peso pesado))
<fact-1>
CLIPS> (facts)
f-0 (objeto (nombre cubo) (id gen1) (loc nil) (peso ligero) (punteros))
f-1 (objeto (nombre cubo) (id gen2) (loc nil) (peso pesado) (punteros))
For a total of 2 facts.
```

© José Angel Bañares

Tipología CLIPS

- **Valores permitidos** `<allowed-type>` **de un atributos:**
  - **INTEGER** : números enteros
  - **FLOAT** : números en coma flotante
  - **STRING** : cadenas de caracteres
  - **SYMBOL** :símbolos (secuencias de caracteres)
  - **FACT-ADDRESS** : punteros para acceder directamente a hechos
  - **INSTANCE-NAME** : nombres de objetos COOL
  - **INSTANCE-ADDRESS** : punteros para acceder a objetos
  - **EXTERNAL-ADDRESS** : punteros a datos externos

© José Angel Bañares

Constructores CLIPS

- **Permiten la declaración de conocimientos (lista de constructores CLIPS) :**
  - `deftemplate` : definición de *template*
  - `deffacts` : definición de hechos iniciales
  - `defrule` : definición de una regla
  - `defglobal` : definición de variable global
  - `deffunction` : definición de una función
  - `defmodule` : definición de un módulo
- **Respecto a COOL (CLISP Object Oriented Language):**
  - `defclass` : Definición de una clase (con herencia)
  - `defmessage-handler` : Definición de un método asociado a una instancia
  - `definstance` : definición de una instancia
  - `defmethod` : declaración de una función genérica.

© José Angel Bañares

Sintaxis de defrule

- **Las reglas en CLIPS :**
  - Formadas por una secuencia de condiciones seguidas por una secuencia de acciones a ejecutar si se verifican las condiciones
  - Se basan en el filtrado de hechos o *pattern-matching*

```
(defrule <identificateur>
     [<commentaire>]          ; cadena de caracteres
     [<declaration>]          ; Propiedades de la regla
     <condition>*             ; LHS (Left-Hand Side)
     =>
     <action>*)               ; RHS (Right-Hand Side)
```

  - No existe límite en el número de condiciones o acciones
  - Las condiciones tienen un AND implícito

© José Angel Bañares

Filtrado de hechos (1)

- **Definición de un filtro o *Pattern-Conditional Element* :**
  - Estructura que contiene constantes, comodines y variables :
    - Constantes : datos simbólicos o (alpha-) numéricos
    - comodines (*wildcards*) : monovaluados (`?`) o multivaluados (`$?`)
    - Variables : monovaluados (`?toto`) o multivaluados (`$?titi`)
      - Los filtros CLIPS pueden contener restricciones más complejas
- **Filtro de un hecho o *pattern matching* :**
  - Consiste en la comparación progresiva de un filtro y un hecho :
    - Las constantes son sólo iguales a ellas mismas
    - Los comodines absorben los datos encontrados
    - Las variables libres se emparejan con los datos encontrados
    - Las variables emparejadas comparan su valor con los datos encontrados.

© José Angel Bañares

Filtrado de hechos (2)

- **Ejemplos :**
  - `(objetivo ?robot coger ?objet)`
    `(objetivo Roby coger cubo)`
  - `(objetivo ?robot coger ?)`
    `(objetivo Roby coger cubo)`
  - `(objetivo Roby $?action)`
    `(objetivo Roby coger cubo)`
  - `(a $?x d $)`
    `(a b c d c d e f)`
- **Elección de la regla a disparar :**
  - Aproximación al algoritmo :

    Para todas las reglas candidatas **Rc** de la base de reglas repetir :
    - Para todos los filtros **P** de la regla **Rc** repetir :
      - Para todos los hechos **F** de la base de hechos repetir :
        - Si el hecho **F** es reconocido por el filtro **P** entonces satisface **P**
    - Si todos los filtros de **Rc** se satisfacen entonces **Rc** es disparable

    Elegir una regla **Rd** entre todas las disparable

    Disparar la regla **Rd**

© José Angel Bañares

La nevera está abierta

```
(deffacts hechos-iniciales
    (refrigerator door open))

(defrule enciende-chivato
    (refrigerator door open)
    =>
    (assert (refrigerator light on)))
```

Hofrqwxfwru **ghiidfwv** shuplwh ghfoduxpd dwd gh khfkrv lqlfldohv +dtx,xq ~qlfr khfkr,

```
CLIPS> (reset)
CLIPS> (facts)
f-0 (initial-fact)
f-1 (refrigerator door open)
For a total of 2 facts.
CLIPS> (run)
CLIPS> (facts)
f-0 (initial-fact)
f-1 (refrigerator door open)
f-2 (refrigerator light on)
For a total of 3 facts.
```

Od lqvwuxfflöq **uhvhw** yd,d od edvh gh khfkrv/ fuhd xq khfkr +lqlwldo0 idfw,/ fuhd orv khfkrv ghfoudgrv frq hofrqvwuxfwru **ghiidfw** h lqlfldo}d hop rwrugh lqihuhqfld

Od lqvwuxfflöq **uxq** hmhfxwd hop rwrugh lqihuhqfld

© José Angel Bañares

El mundo de bloques

```
(deftemplate objeto
   (slot nombre (type SYMBOL) (default ?NONE))
   (slot localizacion (type SYMBOL) (default almacen)))

(deftemplate robot
   (slot nombre (type SYMBOL) (default ?NONE))
   (slot localizacion (type SYMBOL) (default hangar))
   (slot sostiene (type SYMBOL) (default ?DERIVE)))

(deffacts hechos-iniciales
   (robot (nombre Roby))
   (objeto (nombre cubo))
   (objetivo Roby coger cubo))

(defrule crea-objetivo-ir-hacia
   "Crea un subobjetivo ir-hacia a partir de un objetivo coger"
   (objetivo ?robot coger ?objeto)
   (objeto (nombre ?objeto) (localizacion ?loc))
   =>
   (assert (objetivo ?robot ir-hacia ?loc)))
```

© José Angel Bañares

Inicialización

```
CLIPS> (reset)
CLIPS> (facts)
f-0 (initial-fact)
f-1 (robot (nombre Roby) (localizacion hangar) (sostiene nil))
f-2 (objet (nombre cube) (localizacion almacen))
f-3 (objetivo Roby coger cubo)
For a total of 4 facts.
CLIPS> (agenda)
0 crea-objetivo-ir-hacia: f-3,f-2
For a total of 1 activation.
CLIPS> (run)
CLIPS> (facts)
f-0 (initial-fact)
f-1 (robot (nombre Roby) (localizacion hangar) (sostiene nil))
f-2 (objeto (nombre cube) (localizacion almacen))
f-3 (objetivo Roby coger cubo)
f-4 (objetivo Roby ir-hacia almacen)
For a total of 5 facts.
```

Od lqvwuxff‰q <u>djhqgd</u> shuplwh
ylvxdol}du uhjodv fdqglgdwdv

© José Angel Bañares

Gestión de hechos

- **Como suprimir o modificar un hecho :**
  - Se memoriza su FACT-ADDRESS en una variable :

    `?p <- <filtre>`

  - Supresión de un *pattern* o de un *template* : `(retract ?p)`
  - Modificación de un *template* : `(modify ?p (<slot> <valeur>)+)`
    - Toda modificación consiste en un `retract` seguido de un `assert`
- **El robot se puede desplazar ahora :**

```
(defrule acción-desplazar-el-robot
     "El robor cambia de localización"
     ?objetivo <- (objetivo ?robot ir-hacia ?loc)
     ?rob <- (robot (nombre ?robot))
     =>
     (retract ?objetivo)
     (modify ?rob (localizacion ?loc)))
```

© José Angel Bañares

Prueba del programa…

```
CLIPS> (load mundo-bloques.clp)
Defining deftemplate: objet
Defining deftemplate: robot
Defining deffacts: hechos-iniciales
Defining defrule: crea-objetivo-ir-hacia +j+j
Defining defrule: accion-desplazar-robot +j+j
TRUE
CLIPS> (reset)
CLIPS> (facts)
f-0 (initial-fact)
f-1 (robot (nombre Roby) (localizacion hangar) (sostiene nil))
f-2 (objeto (nombre cubo) (localizacion almacen))
f-3 (objetivo Roby coger cubo)
For a total of 4 facts.
CLIPS> (run)
CLIPS> (facts)
f-0 (initial-fact)
f-2 (objeto (nombre cubo) (localizacion almacen))
f-3 (objetivo Roby coger cubo)
f-5 (robot (nombre Roby) (localizacion almacen) (sostiene nil))
For a total of 4 facts.
```

Od lqwxff‰q ordg shuplwh fdujduhoilfkhur frq odv frqwuxfwruhv

úG‰ogh hvwh hokhfkr i07 B

© José Angel Bañares

Traza de las inferencias

- **Muestra de las trazas de las inferencias :**
  - **(watch <item>)** Permite añadir una información a trazar
  - **(unwatch <item>)** Permite eliminar información a trazar
  - **(dribble-on <file>)** Envio de la traza a un fichero de texto
  - **(dribble-off)** Cerrar el fichero de traza

```
CLIPS> (reset)
CLIPS> (watch rules)
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (run)
FIRE    1 crea-objetivo-ir-hacia: f-3,f-2
==> f-4     (objetivo Roby ir-hacia almacen)
==> Activation 0    acion-desplazar-robor: f-4,f-1
FIRE    2 acion-desplazar-robot : f-4,f-1
<== f-4     (objetivo Roby ir-hacia almacen)
<== f-1     (robot (nombre Roby) (localizacion hangar) (sostiene nil))
==> f-5     (robot (nombre Roby) (localizacion almacen) (sostiene nil))
```

© José Angel Bañares

Se depura el programa...

- ¿Que ocurre en la siguiente base de hechos?

```
(deffacts hechos-iniciales
   (objet (nombre cubo))
   (robot (nombre Roby) (localizacion almacen))
   (objetivo Roby coger cubo))
```

```
CLIPS> (reset)
CLIPS> (facts)
f-0 (initial-fact)
f-1 (robot (nombre Roby) (localizacion almacen) (sostiene nil))
f-2 (objeto (nombre cubo) (localizacion almacen))
f-3 (objetivo Roby coger cubo)
CLIPS> (run)
................ ???
```

© José Angel Bañares

Uno se da cuenta de que…

… hay inferencias inútiles :

- ¡Roby no debe desplazarse a un lugar en el que se encuentre ya !
- Es preciso verificar la localización de Roby en "crear-objetivo-ir-hacia"
- Se precisa un test y un operador de negación

- **Reescritura de crear-objetivo-ir-hacia :**

```
(defrule crear-objetivo-ir-hacia
   "Crea un subobjetivo ir-hacia a partir de un objetivo coger"
   (objetivo ?robot coger ?objeto)
   (objeto (nombre ?objeto) (localizacion ?loc-objeto))
   (robot (nombre ?robot) (localizacion ?loc-robot))
   (test (neq ?loc-objeto ?loc-robot))
   =>
   (assert (objetivo ?robot ir-hacia ?loc-objeto)))
```

© José Angel Bañares

Podemos continuar...

- **Se implementa una nueva acción del robot :**

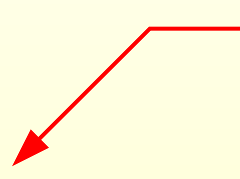
```
(defrule accion-coger-objeto
   ?objetivo <- (objetivo ?robot coger ?objeto)
   ?obj <- (objeto (nombre ?objeto) (localizacion ?loc))
   ?rob <- (robot (nombre ?robot) (localizacion ?loc))
   =>
   (retract ? objetivo )
   (modify ?rob (sostiene ?objeto))
   (modify ?obj (localizacion ?robot)))

CLIPS> (reset)
CLIPS> (run)
CLIPS> (facts)
f-0 (initial-fact)
f-6 (robot (nombre Roby) (localizacion almacen) (sostiene cubo))
f-7 (objeto (nombre cubo) (localizacion Roby))
For a total of 3 facts.
```

© José Angel Bañares

Probamos el programa…

- ¿Qué ocurre con la siguiente memoria de trabajo ?

```
(deffacts hechos-iniciales
   (robot (nombre Roby))
   (objeto (nombre cubo-1))
   (objeto (nombre cubo-2))
   (objetivo Roby coger cubo-1)
   (objetivo Roby coger cubo-2))
```



úUhvxowdgr B

+Kdfhuodv bqihuhqfldv d pdqr,

```
CLIPS> (reset)
CLIPS> (run)
CLIPS> (facts)
f-0  (initial-fact)
f-9  (objeto (nombre cubo-1) (localizacion Roby))
f-10 (robot (nombre Roby) (localizacion almacen) (sostiene cubo-2))
f-11 (objeto (nombre cubo-2) (localizacion Roby))
For a total of 4 facts.
```

© José Angel Bañares

¿Error de modelado ?

- **¡Nuestro robot no puede tener dos objetos a la vez !**
  - Es preciso dejar un objeto antes de coger otro:

```
(defrule crear-objetivo-dejar-objeto
      (objetivo ?robot coger ?objeto)
      (objeto (nombre ?objeto) (localizacion ?loc))
      (robot (nombre ?robot) (localizacion ?loc) (sostiene ?algo))
      (test (neq ?algo nil))
      =>
      (assert (objetivo ?robot dejar ?algo)))

(defrule accion-dejar-objeto
      ? objetivo <- (objetivo ?robot dejar ?objeto)
      ?rob <- (robot (nombre ?robot) (localizacion ?loc))
      ?obj <- (objeto (nombre ?objeto))
      =>
      (retract ? objetivo )
      (modify ?rob (sostiene nil))
      (modify ?obj (localizacion ?loc)))
```

Fuhdfl%q gh
vxerehwlyrv

Dffl%q gh ghmdu

© José Angel Bañares

Corregido

```
CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (robot (nombre Roby) (localizacion hangar) (sostiene nil))
f-2     (objeto (nombre cubo-1) (localizacion almacen))
f-3     (objeto (nombre cubo-2) (localizacion almacen))
f-4     (objetivo Roby coger cubo-1)
f-5     (objetivo Roby coger cubo-2)
For a total of 6 facts.
CLIPS> (watch rules)
CLIPS> (run)
FIRE    1 crear-objetivo-ir-hacia: f-5,f-3,f-1
FIRE    2 accion-desplazar-robot: f-6,f-1
FIRE    3 accion-coger-objeto: f-4,f-2,f-7
FIRE    4 crear-objetivo-dejar-objeto: f-5,f-3,f-8
FIRE    5 accion-dejar-objeto: f-10,f-8,f-9
FIRE    6 accion-coger-objeto: f-5,f-3,f-11
CLIPS> (facts)
f-0     (initial-fact)
f-12    (objeto (nombre cubo-1) (localizacion almacen))
f-13    (robot (nombre Roby) (localizacion almacen) (sostiene cubo-2))
f-14    (objeto (nombre cubo-2) (localizacion Roby))
For a total of 4 facts.
```

Wud}dghorvglsdurv

© José Angel Bañares

Otra posibilidad...

... si el robot tiene dos manos !

- Se puede manipular una lista :

- **Los *patrones y los atributos* multivaluados son listas**
  - Hechos ordenados :
    - Creación de una lista vacía : `(assert (objets))`
    - Creación de una lista con datos : `(assert (objets obj1 obj2))`
    - Manipulación de listas : (ver siguiente transparencia)
  - Atributos multivaluados :

```
(deftemplate robot
    (slot nombre (type SYMBOL) (default ?NONE))
    (slot localizacion (type SYMBOL) (default hangar))
    (multislot sostiene (type SYMBOL) (default ?DERIVE)))
```

Txhgd frpr hmhuflflr prglilfduhohwr ghosurjudpdý

© José Angel Bañares

Gestión de una lista

– Búsqueda de un dato : `(assert (buscar <dato>))`

```
(defrule encuentra-dato
      ?objetivo <- (buscar ?x)
      (lista $ ?x $)
      =>
      (retract ? objetivo )
      (assert (dato-encontrado ?x)))
```

– Retirar un dato : `(assert (retirar <dato>))`

```
(defrule retira-dato
      ? objetivo <- (retirar ?x)
      ?lista <- (lista $?antes ?x $?despues)
      =>
      (retract ?objetivo ?lista)
      (assert (lista ?antes ?despues)))
```

– Añadir un dato : `(assert (agnadir <dato>))`

```
(defrule añadir-dato
      ?objetivo <- (agnadir ?x)
      ?lista <- (lista $?contenido)
      =>
      (retract ?objetivo ?lista)
      (assert (lista ?contenido ?x)))
```

© José Angel Bañares

Escritura de condiciones (1)

❶ Condición de tipo filtro :

```
<pattern-CE> ::= (<constraint> ... <constraint>) |
                 (<deftemplate-name> (<slot-name> <constraint>)*)
<constraint> ::= <constant> | ? | $? | ?<var-symbol> | $?<var-symbol>
```

- Los símbolo **?** et **$?** Son los comodines monovaluados y multivaluados
- Las variables multivaluadas se ligan a listas
- Posibilidad de escribir restricciones complejas

❷ Condición booleana :

```
<test-CE> ::= (test <function-call>)
```

- La sintaxis de llamadas a funciones son tipo Lisp :

```
(defrule operator-condition
       (data ?oper ?x)
       (value ?oper ?y)
       (test (> (abs (- ?y ?x)) 3))
       =>
       (assert (valid ?oper TRUE)))
```

© José Angel Bañares

Escritura de condiciones(2)

❸ Operadores booleanos

```
<not-CE> ::= (not <conditional-element>)
<or-CE> ::= (or <conditional-element>+)
<and-CE> ::= (and <conditional-element>+)
```

– Para el NOT : Leer la página 50 de *Basic Programming Guide*

```
(defrule WARNING::high-flow-rate
      (temp high)
      (valve open)
      (not (error-status confirmed))
      =>
      (assert (warning "High Temp - Recommend closing of valve")))
```

– El AND sólo es útil en combinación con el OR :

```
(defrule ALERT::system-flow
      (error-status confirmed)
      (or (and (temp high) (valve closed))
          (and (temp low) (valve open)))
      =>
      (assert (alert (level 3) (text "Flow problem"))))
```

© José Angel Bañares

Escritura de condiciones (3)

❹ Condición existencial :

```
<exists-CE> ::= (exists <conditional-element>+)
```

– Permite comprobar si un grupo de filtros se satisface para el menos un conjunto de hechos (no genera las posibles combinaciones de instancias de regla disparables) :

```
(deffacts hechos-iniciales
      (nosotros estamos en peligro)
      (super-heroe "Super Man" ocupado)
      (super-heroe "Spider Man" disponible)
      (super-heroe "Wonder Woman" disponible)
      (super-heroe "Flash Gordon" ocupado))

(defrule dont-worry
      ?p <- (nosotros estamos en peligro)
      (exists (super-heroe ? disponible))
      =>
      (retract ?p))
```

© José Angel Bañares

Escritura de condiciones(4)

❺ Condición Universal :

```
<forall-CE> ::= (forall <conditional-element> <conditional-element>+)
```

– Permite comprobar su un grupo de filtros se satisfacen por cada ocurrencia de otro filtro:

```
(defrule todos-los-estudiantes-pasan
      ?p <- (demanda comprobacion ?classe)
      (forall (alumno ?nom ?classe)
           (lectura-OK ?nom)
           (escritura-OK ?nom)
           (math-OK ?nom))
      =>
      (retract ?p)
      (assert (comprobacion (classe ?classe) (estado OK))))
```

© José Angel Bañares

Restricciones complejas (1)

- **CLIPS nos da la posibilidad de introducir restricciones complejas en el interior de los filtros :**
  - Reescritura de una reglas :

```
(defrule crea-objetivo-ir-hacia
        (objetivo ?robot coger ?objeto)
        (objeto (nombre ?objeto) (localizacion ?loc-objeto))
        (robot (nombre ?robot) (localizacion ?loc-robot))
        (test (neq ?loc-objeto ?loc-robot))
        =>
        (assert (objetivo ?robot ir-hacia ?loc-objeto)))
```

Dqwhv

```
(defrule crea-objetivo-ir-hacia
        (objetivo ?robot coger ?objeto)
        (objeto (nombre ?objeto) (localizacion ?loc))
        (robot (nombre ?robot) (localizacion ~?loc))
        =>
        (assert (objetivo ?robot ir-hacia ?loc)))
```

Ghvsx‡v

© José Angel Bañares

Restricciones complejas(2)

- **Extensión de la sintaxis de restricciones en filtros :**
  - Se puede añadir conectores lógicos, llamadas funcionales, *etc*.
  - Nueva defición de <constraint> (ver trasparencia 62) QHZ $

```
<constraint> ::= ? | $? | <connected-constraint>
<connected-constraint> ::= <single-constraint> |
                           <single-constraint>&<connected-constraint> |
                           <single-constraint>|<connected-constraint>
<single-constraint> ::= <term> | ~<term>
<term> ::= <constant> | <single-field-variable> | <multified-variable> |
           <predicate-function-call> | <return-value-constraint>
<single-field-var> ::= ?<variable-symbol>
<multifield-var> ::= $?<variable-symbol>
<predicate-function-call> ::= :<function-call>
<return-value-constraint> ::= =<function-call>
```

QHZ $

© José Angel Bañares

Restricciones complejas(3)

- Restricciones con negaciones :

```
(persona (edad ~30))
(persona (edad ?x&~20))
(persona (edad ?x&~20&~30))
```

- Restricciones con llamadas funcionales:

```
(persona (edad ?x&~:(oddp ?x)))
(persona (edad ?x&:(> ?x 30)&:(< ?x 40)))
```

- Restricciones muy complejas :

```
(defrule regla-compleja
    (persona (nombre ?x) (edad ?y))
    (persona (nombre ~?x) (edad ?w&?y|=(* 2 ?y)))
    =>
    (assert (regla muy compleja)))
```

© José Angel Bañares

Aspectos funcionales

- **Sintaxis al gusto de los programadores Lisp :**

Yhuvlrq OLVS

```
(defun corta (lista indice)
   (if (eq indice 1)
      lista
      (corta (cdr lista)
             (- indice 1))))

LISP> (corta (list 1 2 3 4) 3)
(3 4)
```

Yhuvlrq FOLSV

```
(deffunction corta$ (?lista ?indice)
   (if (eq ?indice 1)
      then ?lista
      else (corta$ (rest$ ?lista )
                   (- ?indice 1))))

CLIPS> (corta$ (create$ 1 2 3 4) 3)
(3 4)
```

– Las funciones definidas con `deffunction` (como las primitivas) se pueden utilizar en las reglas

© José Angel Bañares

Predicados

Funciones que devuelven TRUE o FALSE

- **Predicados de tipo :**
  - (numberp <expr>)(integerp <expr>)(floatp <expr>)
  - (stringp <expr>)(symbolp <expr>)(multifieldp <expr>)...
- **Predicados de comparación :**
  - (eq <expr> <expr>+)(neq <expr> <expr>+)(= <expr> <expr>+)
  - (> <expr> <expr>+)(>= <expr> <expr>+)(<> <expr> <expr>+)...
- **Predicados booleanos :**
  - (and <expr>+)(or <expr>+)(not <expr>)
- **Otros predicados :**
  - (oddp <expr>)(evenp <expr>)
  - (subsetp <expr> <expr>)...

© José Angel Bañares

Otras funciones

- **Funciones para cadenas :**
  - `(str-cat <expr>*)(sub-string <int> <int> <expr>)`
  - `(str-index <expr> <expr>)(str-compare <expr> <expr>)`
  - `(str-length <expr>)(upcase <expr>)(lowcase <expr>)...`
- **Funciones aritméticas :**
  - `(+ <expr> <expr>+)(- <expr> <expr>+)(* <expr> <expr>+)`
  - `(/ <expr> <expr>+)(div <expr> <expr>)(mod <expr> <expr>)`
  - `(** <expr> <expr>)(exp <expr>)(log <expr>)(log10 <expr>)`
  - `(max <expr>+)(min <expr>+)(abs <expr>)(sqrt <expr>)...`
- **Funciones trigonométricas:**
  - `(sin <expr>)(cos <expr>)(tan <expr>)(sinh <expr>)...`
- **Conversiones de tipo :**
  - `(float <expr>)(integer <expr>)`
  - `(deg-rad <expr>)(rad-deg <expr>)...`

© José Angel Bañares

Funciones sobre listas

- (create$ a b c d) → (a b c d)
- (explode$ "a b c d") → (a b c d)
- (implode$ (create$ a b c d)) → "a b c d"
- (nth$ 1 (create$ a b c d)) → a
- (first$ (create$ a b c d)) → (a)
- (rest$ (create$ a b c d)) → (b c d)
- (length$ (create$ a b c d)) → 4
- (member$ b (create$ a b c d)) → 2
- (insert$ (create$ a c d) 2 b)) → (a b c d)
- (insert$ (create$ a d) 2 (create$ b c))) → (a b c d)
- (delete$ (create$ a b c d) 2 3) → (a d)
- (subseq$ (create$ a b c d) 2 3) → (b c)
- (replace$ (create$ a b c d) 2 3 x) → (a x d)
- (subsetp (create$ a c) (create$ a b c d)) → TRUE

© José Angel Bañares

Funciones "procedurales"

- Ligadura variable-valor :
  - `(bind <variable> <expression>)`
- Si ... entonces ... sino:
  - `(if <expression> then <action>* [else <action>*])`
- Mientras que :
  - `(while <expression> [do] <action>*)`
- Para :
  - `(loop-for-count <range> [do] <action>*)`
  - `<range> ::= <end-index> | (<variable> [<start> <end>])`
- Para cada :
  - `(progn$ (<variable> <expression>) <expression>*)`
- Seleccion :
  - `(switch <test> (case <expression> then <action>*)+)`

© José Angel Bañares

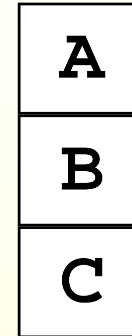
Entradas y salidas

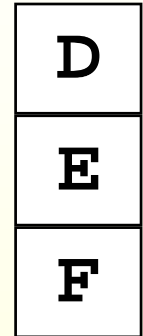
- **Sobre la noción de *stream* :**
  - `stdin` (entrada std) `stdout` (salida std) `wclips` (prompt) `werror` (errores) `wwarning` (warnings) `wtrace` (trazas) …
- **Operaciones sobre ficheros :**
  - `(open <file-name> <logical-name> [<mode>])`
  - `"r"` (solo lectura) `"w"` (sólo escritura) `"r+"` (lectura y escritura) `"a"` (añadir)
  - `(close [<logical-name>])`
  - `(rename <old-file-name> <new-file-name>)`
  - `(remove <file-name>)`
- **Lectura y escritura :**
  - `(read [<logical-name>])`
  - `(readline [<logical-name>])`
  - `(printout <logical-name> <expression>*)`
  - `(format <logical-name> <string> <expression>*)`
  - `(dribble-on <file-name>) (dribble-off`

© José Angel Bañares

Ejemplo programa en Clips

Enunciado: Objetivo Poner C Encima de E





Representación

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

© José Angel Bañares

Mundo de bloques

```
;;; Regla mover-bloque-sobre-bloque
;;; SI el objetivo es poner el objeto X encima del objeto Y y
;;;    tanto X como Y son bloques y
;;;    no hay nada encima del bloque X ni del bloque Y
;;; ENTONCES
;;;    colocamos el bloque X encima del bloque Y y
;;;    actualizamos datos

(defrule mover-bloque-sobre-bloque
   ?objetivo <- (objetivo ?objeto-1 esta-encima-del ?objeto-2)
   (bloque ?objeto-1)
   (bloque ?objeto-2)
   (estado nada esta-encima-del ?objeto-1)
   ?pila-1 <- (estado ?objeto-1 esta-encima-del ?objeto-3)
   ?pila-2 <- (estado nada esta-encima-del ?objeto-2)
  =>
   (retract ?objetivo ?pila-1 ?pila-2)
   (assert (estado ?objeto-1 esta-encima-del ?objeto-2))
   (assert (estado nada esta-encima-del ?objeto-3))
   (printout t ?objeto-1 " movido encima del " ?objeto-2 "." crlf))
```

© José Angel Bañares

Mundo de bloques

```
;;; Regla mover-bloque-al-suelo
;;; SI el objetivo es poner el objeto X al suelo
;;;    X es un bloque
;;;    no hay nada encima de X
;;; ENTONCES
;;;    movemos X al suelo y
;;;    actualizamos datos

(defrule mover-bloque-al-suelo
   ?objetivo <- (objetivo ?objeto-1 esta-encima-del suelo)
   (bloque ?objeto-1)
   (estado nada esta-encima-del ?objeto-1)
   ?pila <- (estado ?objeto-1 esta-encima-del ?objeto-2)
  =>
   (retract ?objetivo ?pila)
   (assert (estado ?objeto-1 esta-encima-del suelo))
   (assert (estado nada esta-encima-del ?objeto-2))
   (printout t ?objeto-1 " movido encima del suelo. " crlf))
```

© José Angel Bañares

Mundo de bloques

```
;;; Regla libera-bloque-movible
;;; SI el objetivo es poner el objeto X encima de Y
;;;    (bloque o suelo) y
;;;   X es un bloque y
;;;   y hay un bloque encima del bloque X
;;; ENTONCES
;;;    hay que poner el bloque que esta encima de X
;;;    en el suelo

(defrule libera-bloque-movible
   (objetivo ?objeto-1 esta-encima-del ?)
   (bloque ?objeto-1)
   (estado ?objeto-2 esta-encima-del ?objeto-1)
   (bloque ?objeto-2)
  =>
   (assert (objetivo ?objeto-2 esta-encima-del suelo)))
```

© José Angel Bañares

Mundo de bloques

```
;;; Regla libera-bloque-soporte
;;; SI
;;;   el objetivo es poner X (bloque o nada) encima de Y
;;;    (bloque o suelo) y
;;;   Y es un bloque y
;;;   hay un bloque encima del bloque Y
;;; ENTONCES
;;;    hay que poner el bloque que esta encima de Y
;;;    en el suelo

(defrule libera-bloque-soporte
   ?objetivo <- (objetivo ? esta-encima-del ?objeto-1)
   (bloque ?objeto-1)
   (estado ?objeto-2 esta-encima-del ?objeto-1)
   (bloque ?objeto-2)
  =>
   (assert (objetivo ?objeto-2 esta-encima-del suelo)))
```

© José Angel Bañares

Mundo de bloques

Sesión

```
CLIPS> (clear)
CLIPS> (load "bloques.clp")
Defining deffacts: estado-inicial
Defining defrule: mover-bloque-sobre-bloque +j+j+j+j+j+j
Defining defrule: mover-bloque-al-suelo +j+j+j+j
Defining defrule: libera-bloque-movible =j=j+j+j
Defining defrule: libera-bloque-soporte =j+j+j+j
TRUE
CLIPS> (watch facts)
CLIPS> (watch rules)
CLIPS> (watch activations)
```

© José Angel Bañares

Mundo de bloques

```
==> f-0      (initial-fact)
==> f-1      (bloque A)
==> f-2      (bloque B)
==> f-3      (bloque C)
==> f-4      (bloque D)
==> f-5      (bloque E)
==> f-6      (bloque F)
==> f-7      (estado nada esta-encima-del A)
==> f-8      (estado A esta-encima-del B)
==> f-9      (estado B esta-encima-del C)
==> f-10     (estado C esta-encima-del suelo)
==> f-11     (estado nada esta-encima-del D)
==> f-12     (estado D esta-encima-del E)
==> f-13     (estado E esta-encima-del F)
==> f-14     (estado F esta-encima-del suelo)
==> f-15     (objetivo C esta-encima-del E)
==> Activation 0      libera-bloque-soporte: f-15,f-5,f-12,f-4
==> Activation 0      libera-bloque-movible: f-15,f-3,f-9,f-2
```

© José Angel Bañares

Mundo de bloques

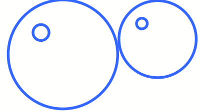
```
CLIPS> (run)
FIRE    1 libera-bloque-movible: f-15,f-3,f-9,f-2
==> f-16    (objetivo B esta-encima-del suelo)
==> Activation 0      libera-bloque-movible: f-16,f-2,f-8,f-1
FIRE    2 libera-bloque-movible: f-16,f-2,f-8,f-1
==> f-17    (objetivo A esta-encima-del suelo)
==> Activation 0      mover-bloque-al-suelo: f-17,f-1,f-7,f-8
FIRE    3 mover-bloque-al-suelo: f-17,f-1,f-7,f-8
<== f-17    (objetivo A esta-encima-del suelo)
<== f-8     (estado A esta-encima-del B)
==> f-18    (estado A esta-encima-del suelo)
==> f-19    (estado nada esta-encima-del B)
==> Activation 0      mover-bloque-al-suelo: f-16,f-2,f-19,f-9
A movido encima del suelo.
FIRE    4 mover-bloque-al-suelo: f-16,f-2,f-19,f-9
<== f-16    (objetivo B esta-encima-del suelo)
<== f-9     (estado B esta-encima-del C)
==> f-20    (estado B esta-encima-del suelo)
==> f-21    (estado nada esta-encima-del C)
B movido encima del suelo.
```

© José Angel Bañares

Mundo de bloques

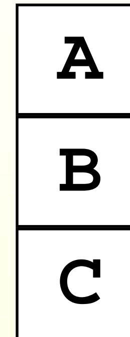
```
FIRE    5 libera-bloque-soporte: f-15,f-5,f-12,f-4
==> f-22    (objetivo D esta-encima-del suelo)
==> Activation 0      mover-bloque-al-suelo: f-22,f-4,f-11,f-12
FIRE    6 mover-bloque-al-suelo: f-22,f-4,f-11,f-12
<== f-22    (objetivo D esta-encima-del suelo)
<== f-12    (estado D esta-encima-del E)
==> f-23    (estado D esta-encima-del suelo)
==> f-24    (estado nada esta-encima-del E)
==> Activation 0      mover-bloque-sobre-bloque: f-15,f-3,f-5,f-
   21,f-10,f-24
D movido encima del suelo.
FIRE    7 mover-bloque-sobre-bloque: f-15,f-3,f-5,f-21,f-10,f-24
<== f-15    (objetivo C esta-encima-del E)
<== f-10    (estado C esta-encima-del suelo)
<== f-24    (estado nada esta-encima-del E)
==> f-25    (estado C esta-encima-del E)
==> f-26    (estado nada esta-encima-del suelo)
C movido encima del E.
CLIPS>
```

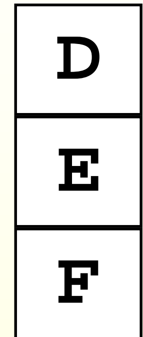
© José Angel Bañares



Mundo de bloques 2

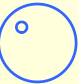
Enunciado: Objetivo Poner C Encima de E

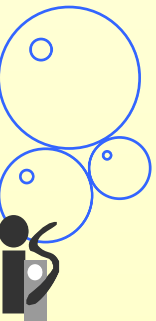


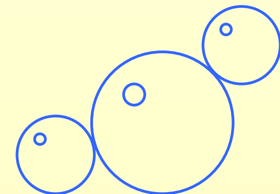


Representación

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







© José Angel Bañares

Mundo de bloques 2

```
;;; Reglas mover-bloque-sobre-bloque y mover-bloque-al-suelo
(defrule mover-bloque-sobre-bloque
      ?objetivo <- (objetivo ?bloque-1  esta-encima-del ?bloque-2)
      ?pila-1 <- (pila ?bloque-1 $?resto-1)
      ?pila-2 <- (pila ?bloque-2 $?resto-2)
     =>
      (retract ?objetivo ?pila-1 ?pila-2)
      (assert (pila $?resto-1))
      (assert (pila ?bloque-1 ?bloque-2 $?resto-2))
      (printout t ?bloque-1 " movido encima del " ?bloque-2 "." crlf))

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

© José Angel Bañares

Mundo de bloques 2

```
;;; Reglas libera-bloque-movible y libera-bloque-soporte

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


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