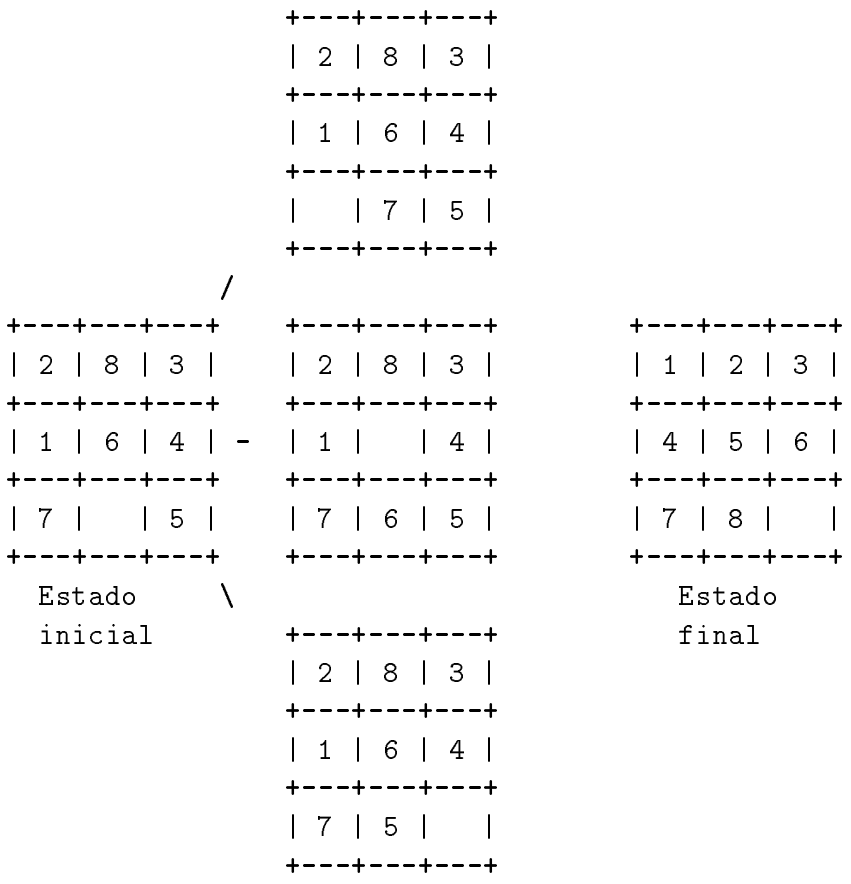
Tema 2:
Problemas y espacios de estados.

Descripción de problemas

- Elementos que describen un problema:
  - Estado inicial
  - Operadores
  - Estados finales
- Suposiciones subyacentes:
  - Agente único
  - Conocimiento completo

Descripción de problemas: El 8–puzzle

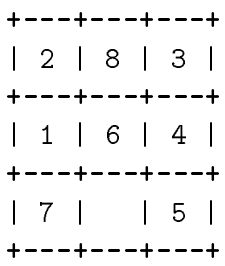


Representación de estados

- Propiedades de las representaciones:
  - Representación suficiente
  - Representación necesaria
- Ejemplo: 8–puzzle: Elementos de la representación:
  - Localización de cada bloque y del hueco;
  - tipo de material de los bloques;
  - colores de los bloques, ...

Representación de estados

- Ejemplo: 8–puzzle: Representaciones del estado



  - **Lista:** `(2 8 3 1 6 4 7 H 5)`, `(2 8 3 4 H 7 1 6)`
  - **Matriz:** `((2 8 3)(1 6 4)(7 H 5))`
  - **Hechos:** `((primera-izquierda 2) (primera-centro 8) ...)`
  - **Número de estados** $= 9! = 362.880$

- Criterio para elegir una representación

Operadores

- Definición de operador
- Elementos de un operador:
  - Precondiciones
  - Postcondiciones
- Criterio para elegir operadores
- Ejemplo: Operadores del 8–puzzle:
  - Según los movimientos de los bloques = 32
  - Según los movimientos del hueco = 4

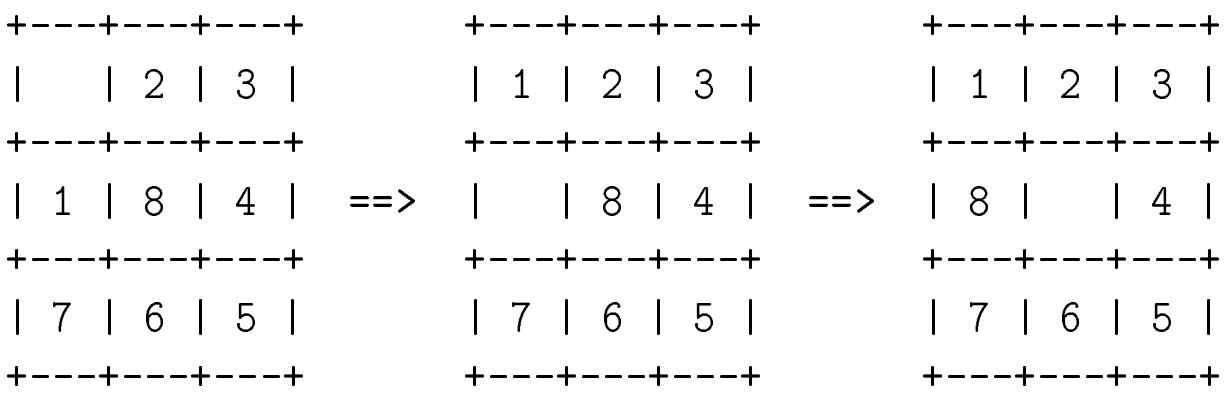
Estados finales

- Clasificación de problemas por el número de estados finales:
  - Unico estado final: 8–puzzle
  - Múltiples estados finales: 8–reinas

- Formas de implementar los estados finales:
  - Enumerativa
  - Declarativa

Soluciones de un problema

- Definición de solución de un problema
- Ejemplo: Solución del 8–puzzle:




```
(mover-hueco-abajo mover-hueco-derecha)
```

Soluciones de un problema

- Tipos de problemas:
  - Buscar una solución.
  - Buscar la solución más corta.
  - Buscar la solución menos costosa.
  - Buscar cualquier solución lo más rápidamente posible.
  - Buscar todas las soluciones.
  - Determinar si existe solución y encontrar un estado final.

Problema de las jarras de agua

- Enunciado:
  - Se tienen dos jarras, una de 4 litros de capacidad y otra de 3.
  - Ninguna de ellas tiene marcas de medición.
  - Se tiene una bomba que permite llenar las jarras de agua.
  - Averiguar cómo se puede lograr tener exactamente 2 litros de agua en la jarra de 4 litros de capacidad?.
- Representación de estados: `(x y)`, `x` $\in \{0,1,2,3,4\}$, `y` $\in \{0,1,2,3\}$
- Número de estados $= 20$

Problema de las jarras de agua

- Estado inicial: `(0 0)`
- Estados finales: `(2 n)`
- Operadores:
  - Llenar la jarra de 3 litros con la bomba.
  - Llenar la jarra de 4 litros con la bomba.
  - Llenar la jarra de 3 litros con la jarra de 4 litros.
  - Llenar la jarra de 4 litros con la jarra de 3 litros.
  - Vaciar la jarra de 3 litros en la jarra de 4 litros.
  - Vaciar la jarra de 4 litros en la jarra de 3 litros.
  - Vaciar la jarra de 3 litros en el suelo.
  - Vaciar la jarra de 4 litros en el suelo.

Problema del granjero, el lobo, la cabra y la col

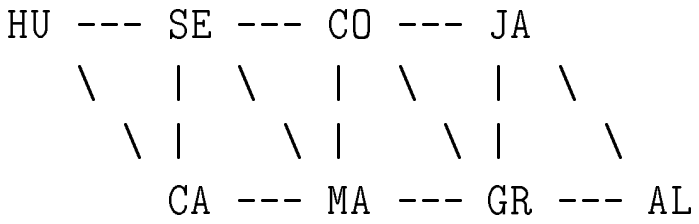
- Enunciado:
  - Un granjero está con un lobo, una cabra y una col en una orilla de un río.
  - Desea pasarlos a la otra orilla.
  - Dispone de una barca en la que sólo puede llevar una cosa cada vez.
  - l lobo se come a la cabra si no está el granjero.
  - La cabra se come la col si no está el granjero.
- Representación de estados: `(x y z u)` $\in \{i, d\} \times \{i, d\} \times \{i, d\} \times \{i, d\}$
- Número de estados $= 16$

Problema del granjero, el lobo, la cabra y la col

- Estado inicial: `(i i i i)`
- Estado final: `(d d d d)`
- Operadores:
  - Pasa el granjero sólo.
  - Pasa el granjero con el lobo.
  - Pasa el granjero con la cabra.
  - Pasa el granjero con la col.

Problema del viaje

- Enunciado:
  - Nos encontramos en una capital andaluza (p.e. Sevilla).
  - Deseamos ir a otra capital andaluza (p.e. Almería).
  - La línea de autobuses sólo tiene viajes de cada capital a sus vecinas.



- Representación de estados: x
  - $x \in \{almeria, cadiz, cordoba, granada, huelva, jaen, malaga, sevilla\}$
- Número de estados $= 8$.

Problema del viaje

- Estado inicial: sevilla
- Estado final: almeria
- Operadores:
  - Ir a Almería.
  - Ir a Cádiz.
  - Ir a Córdoba.
  - Ir a Granada.
  - Ir a Huelva.
  - Ir a Jaen.
  - Ir a Málaga.
  - Ir a Sevilla.

Lisp: Funciones aritméticas

```
* (+ X-1 ... X-N)
      (+ 3 7 5)  =>  15
      (+ 3)      =>  3
      (+)        =>  0
* (- X-1 ... X-N)
      (- 123 7 5)  =>  111
      (- 3)        =>  -3
* (* X-1 ... X-N)
      (* 2 7 5)          =>  70
      (* 3)              =>  3
      (*)                =>  1
* (/ X Y)
      (/ 6 2)   =>  3.0
      (/ 5 2)   =>  2.5
* (MOD X Y)
      (mod 7 2)   =>  1
```

Lisp: Definición de funciones

```
* (DEFUN NOMBRE LISTA FORMA-1 ... FORMA-N)
      > (defun cuadrado (n) (* n n))
      CUADRADO
      > (cuadrado 3)
      9
      > (defun suma-cuadrados (x y)
           (+ (cuadrado x) (cuadrado y)))
      SUMA-CUADRADOS
      > (suma-cuadrados 3 4)
      25
```

Lisp: Funciones de construcción de listas

```
* (CONS X Y)
      (cons 'a 'b)                          =>  (A . B)
      (cons 'a '(b c))                      =>  (A B C)
      (cons 'a (cons 'b (cons 'c '())))     =>  (A B C)
      (cons '(a b) '(c d))                  =>  ((A B) C D)
* (LIST X-1 X-1 ... X-N)
      (list 'a 'b 'c)                         =>  (A B C)
      (list '(a b) '(c d))                    =>  ((A B) (C D))
      (list)                                  =>  NIL
      (list (list 'a 'b) (list 'c 'd 'e))     =>  ((A B) (C D E))
* (APPEND L-1 ... L-N)
      (append '(a) '(b) () '(x y))  =>  (A B C X Y)
      (append '(a b) '(c d))        =>  (A B C D)
* (REVERSE L)
      (reverse '(a (b c) d))  =>  (D (B C) A)
```

Lisp: Funciones de acceso a listas

```
* (FIRST L)
      (first '(a b c))  =>  A
      (first ())        =>  NIL
* (REST L)
      (rest '(a b c))  =>  (B C)
      (rest ())        =>  NIL
* (SECOND L)
      (second '(a b c d))  =>  B
      (second '(a))        =>  NIL
* (NTH N L)
      (nth  2 '(a b c d))  =>  C
      (nth  2 '(a b))      =>  NIL
* (LENGTH L)
      (length '(a (b c) d))  =>  3
```

Lisp: Valores lógicos y predicados aritméticos

- Valores lógicos: `NIL`, `()`, `T`
- Predicados aritméticos:

```
* (= X-1 ... X-N)
      (= 10 (+ 3 7))      =>  T
      (= 2 2.0 (+ 1 1))   =>  T
      (= 1 2 3)           =>  NIL
      (= 1 2 1)           =>  NIL
* (> X-1 ... X-N)
      (> 4 3 2 1)  =>  T
      (> 4 3 3 2)  =>  NIL
* (>= X-1 ... X-N)
      (>= 4 3 3 2)  =>  T
      (>= 4 3 3 5)  =>  NIL
* (< X-1 ... X-N)  y (<= X-1 ... X-N)
```

Lisp: Predicados de igualdad

```
* (EQ X Y)
      (eq 3 3)                                  => T
      (eq 3 3.0)                                => NIL
      (eq 3.0 3.0)                              => NIL
      (eq (first '(a b c)) 'a)                  => T
      (eq (cons 'a '(b c)) '(a b c))            => NIL
* (EQL X Y)
      (eql 3.0 3.0)                              => T
* (EQUAL X Y)
      (equal (cons 'a '(b c)) (cons 'a '(b c))) => T
```

Lisp: Operadores lógicos

```
* (NOT X)
      (not (= (+ 1 1) 2))  =>  NIL
      (not (= (+ 1 1) 3))  =>  T
* (OR E-1 ... E-N)
      (or nil 2 3)                 =>  2
      (or (eq 'a 'b) (eq 'a 'c))   =>  NIL
* (AND E-1 ... E-N)
      (and 1 nil 3)  =>  NIL
      (and 1 2 3)    =>  3
```

Lisp: Condicionales

```
* (IF TEST ENTONCES [EN-CASO-CONTRARIO])
      (if t   1 2)  =>  1
      (if nil 1)    =>  NIL
* (WHEN TEST E-1 ... E-N)
      (when t   1 2 3)  =>  3
      (when nil 1 2 3)  =>  NIL
* (UNLESS TEST E-1 ... E-N)
* (COND L-1 ... L-N)
      > (defun notas (n)
          (cond ((< n 5) 'suspenso)
                ((< n 7) 'aprobado)
                ((< n 9) 'notable)
                (t 'sobresaliente) ))
      NOTAS
      > (notas 8)
      NOTABLE
```

Lisp: Definiciones recursivas

- Definición de factorial

```
> (defun factorial (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1)))))
FACTORIAL
> (factorial 3)
6
```

```
* (TRACE F-1 ... F-N)
        > (trace factorial *)
        ;; Tracing function FACTORIAL.
        ;; Tracing function *.
        (FACTORIAL *)
```

Lisp: Definiciones recursivas

```
> (factorial 3)
1. Trace: (FACTORIAL ´3)
2. Trace: (FACTORIAL ´2)
3. Trace: (FACTORIAL ´1)
4. Trace: (FACTORIAL ´0)
4. Trace: FACTORIAL ==> 1
4. Trace: (* ´1 ´1)
4. Trace: * ==> 1
3. Trace: FACTORIAL ==> 1
3. Trace: (* ´1 ´2)
3. Trace: * ==> 2
2. Trace: FACTORIAL ==> 2
2. Trace: (* ´2 ´3)
2. Trace: * ==> 6
1. Trace: FACTORIAL ==> 6
6
```

Lisp: Definiciones recursivas

```
> (trace)
(* FACTORIAL)

* (UNTRACE F-1 ... F-N)
```

Lisp: Declaración de variables

```
* (DEFPARAMETER NOMBRE VALOR-INICIAL)
      > (defparameter *estado-inicial* '(4 0))
      *ESTADO-INICIAL*
      > (first *estado-inicial*)
      4
```

Implementación de problemas

- La implementación de un problema general consta de:
  - Una estructura para representar los estados.
  - Una lista de operadores: `*OPERADORES*`.
  - Una definición de cada operador.
- La implementación de un problema particular necesita:
  - `*estado-inicial*`
  - `(es-estado-final x)`

Problema del granjero, el lobo, la cabra y la col

- Enunciado:
  - Un granjero está con un lobo, una cabra y una col en una orilla de un río.
  - Desea pasarlos a la otra orilla.
  - Dispone de una barca en la que sólo puede llevar una cosa cada vez.
  - l lobo se come a la cabra si no está el granjero.
  - La cabra se come la col si no está el granjero.
- Representación de estados: `(x y z u)` $\in \{i, d\} \times \{i, d\} \times \{i, d\} \times \{i, d\}$
- Número de estados $= 16$

Implementación del problema del granjero usando listas

- Representación de estados

```
(defun crea-estado (a b c d)
  (list a b c d))

(defun posicion-granjero (estado)
  (first estado))

(defun posicion-lobo (estado)
  (second estado))

(defun posicion-cabra (estado)
  (third estado))

(defun posicion-col (estado)
  (fourth estado))
```

Implementación del problema del granjero usando listas

- Estado inicial

```
(defparameter *estado-inicial*
  (crea-estado 'i 'i 'i 'i))
```

- Estado final

```
(defparameter *estado-final*
  (crea-estado 'd 'd 'd 'd))

(defun es-estado-final (estado)
  (equal estado *estado-final*))
```

Implementación del problema del granjero usando listas

- Funciones auxiliares

```
(defun opuesta (posicion)
  (cond ((eq posicion 'i) 'd)
        ((eq posicion 'd) 'i)))

(defun es-seguro (estado)
  (when (and (if (eq (posicion-lobo estado) (posicion-cabra estado))
   (eq (posicion-granjero estado) (posicion-lobo estado))
   t)
(if (eq (posicion-cabra estado) (posicion-col estado))
   (eq (posicion-granjero estado) (posicion-cabra estado))
   t))
  estado))
```

Implementación del problema del granjero usando listas

- Operadores

```
(defparameter *operadores*
  '(pasa-granjero-solo
    pasan-granjero-y-lobo
    pasan-granjero-y-cabra
    pasan-granjero-y-col))

(defun pasa-granjero-solo (estado)
  (es-seguro (crea-estado
              (opuesta (posicion-granjero estado))
              (posicion-lobo estado)
              (posicion-cabra estado)
              (posicion-col estado))))
```

Implementación del problema del granjero usando listas

```
(defun pasan-granjero-y-lobo (estado)
  (when (eq (posicion-granjero estado) (posicion-lobo estado))
        (es-seguro (crea-estado
                    (opuesta (posicion-granjero estado))
                    (opuesta (posicion-lobo estado))
                    (posicion-cabra estado)
                    (posicion-col estado)))))

(defun pasan-granjero-y-cabra (estado)
  (when (eq (posicion-granjero estado) (posicion-cabra estado))
        (es-seguro (crea-estado
                    (opuesta (posicion-granjero estado))
                    (posicion-lobo estado)
                    (opuesta (posicion-cabra estado))
                    (posicion-col estado)))))
```

Implementación del problema del granjero usando listas

```
(defun pasan-granjero-y-col (estado)
  (when (eq (posicion-granjero estado) (posicion-col estado))
        (es-seguro (crea-estado
                    (opuesta (posicion-granjero estado))
                    (posicion-lobo estado)
                    (posicion-cabra estado)
                    (opuesta (posicion-col estado))))))
```

Lisp

```
* (NULL X)
        (null (rest '(a b)))  =>  NIL
        (null (rest '(a)))    =>  T

* (SETF SIMBOLO-1 E-1 SIMBOLO-2 E-2 ... SIMBOLO-N E-N)
        (setf x 3 y (+ x 2))  =>  5
        y                     =>  5

* (FUNCALL FN E-1 ... E-N)
        (funcall #'+ 1 2 3)  =>  6
```

Lisp: Escritura

```
* (FORMAT DESTINO CADENA-DE-CONTROL X-1 ... X-N)
        > (format t "~&Linea 1 ~%Linea 2")
        Linea 1
        Linea 2
        NIL
        > (format t "~&El cuadrado de ~a es ~a" 3 (* 3 3))
        El cuadrado de 3 es 9
        NIL
        > (setf l '(a b c))
        (A B D)
        > (format t
                  "~&La longitud de la lista ~a es ~a"
                  l (length l))
        La longitud de la lista (A B C) es 3
        NIL
```

Lisp: Funciones con argumentos opcionales

```
> (defun factorial (n &optional (resultado 1))
    (if (= n 0)
        resultado
        (factorial (- n 1) (* n resultado))))
> FACTORIAL
> (trace factorial *)
;; Tracing function FACTORIAL.
;; Tracing function *.
(FACTORIAL *)
```

Lisp: Funciones con argumentos opcionales

```
> (factorial 3)
1. Trace: (FACTORIAL ´3)
2. Trace: (* ´3 ´1)
2. Trace: * ==> 3
2. Trace: (FACTORIAL ´2 ´3)
3. Trace: (* ´2 ´3)
3. Trace: * ==> 6
3. Trace: (FACTORIAL ´1 ´6)
4. Trace: (* ´1 ´6)
4. Trace: * ==> 6
4. Trace: (FACTORIAL ´0 ´6)
4. Trace: FACTORIAL ==> 6
3. Trace: FACTORIAL ==> 6
2. Trace: FACTORIAL ==> 6
1. Trace: FACTORIAL ==> 6
6
```

Verificación de soluciones

```
> (load "granjero-1.lsp")
T
> (load "verifica.lsp")
T
> (verifica ´(pasan-granjero-y-cabra
              pasa-granjero-solo))
(I I I I) PASAN-GRANJERO-Y-CABRA
(D I D I) PASA-GRANJERO-SOLO
(I I D I) NO ES ESTADO FINAL
NIL
```

Verificación de soluciones

```
> (verifica ´(pasan-granjero-y-cabra
              pasa-granjero-solo
              pasan-granjero-y-col
              pasan-granjero-y-cabra
              pasan-granjero-y-lobo
              pasa-granjero-solo
              pasan-granjero-y-cabra))
(I I I I) PASAN-GRANJERO-Y-CABRA
(D I D I) PASA-GRANJERO-SOLO
(I I D I) PASAN-GRANJERO-Y-COL
(D I D D) PASAN-GRANJERO-Y-CABRA
(I I I D) PASAN-GRANJERO-Y-LOBO
(D D I D) PASA-GRANJERO-SOLO
(I D I D) PASAN-GRANJERO-Y-CABRA
(D D D D) ESTADO FINAL
T
```

Verificación de soluciones

```
(defun verifica (plan &optional (estado *estado-inicial*))
  (cond ((null plan)
         (cond ((es-estado-final estado)
                (format t "~&~a ~a" estado "Estado final")
                t)
               (t (format t "~&~a ~a" estado "No es estado final")
                  nil)))
        (t (format t "~&~a ~a" estado (first plan))
           (verifica (rest plan) (funcall (first plan) estado)))))
```

LISP: Estructuras

```
(DEFSTRUCT (NOMBRE (:CONSTRUCTOR FUNCION-CONSTRUCTURA)
                   (:CONC-NAME PREFIJO-)
                   (:PRINT-FUNCTION FUNCION-DE-ESCRITURA))
  CAMPO-1
  ...
  CAMPO-N)
```

```
> (defstruct (punto (:constructor crea-punto)
                    (:conc-name coordenada-))
    x
    y)
PUNTO
```

LISP: Estructuras

```
> (setf *punto-1* (crea-punto :x 2 :y 3))
#S(PUNTO :X 2 :Y 3)
> (coordenada-y *punto-1*)
3
> (setf (coordenada-y *punto-1*) 5)
5
> *punto-1*
#S(PUNTO :X 2 :Y 5)
> (punto-p *punto-1*)
T
> (punto-p ´(2 5))
NIL
> (setf *punto-2* (copy-punto *punto-1*))
#S(PUNTO :X 2 :Y 5)
```

LISP: Estructuras

```
> (equal *punto-1* *punto-2*)
NIL
> (equalp *punto-1* *punto-2*)
T
> (setf (coordenada-y *punto-2*) 3)
3
> *punto-2*
#S(PUNTO :X 2 :Y 3)
> *punto-1*
#S(PUNTO :X 2 :Y 5)
> (setf *punto-3* (crea-punto :y 3 :x 2))
#S(PUNTO :X 2 :Y 3)
> (equalp *punto-2* *punto-3*)
T
```

LISP: Estructuras

```
> (defstruct (estado (:constructor crea-estado)
                     (:conc-name posicion-)
                     (:print-function escribe-estado-granjero))
    granjero
    lobo
    cabra
    col)
ESTADO
> (defun escribe-estado-granjero (estado &optional (canal t) profundidad)
    (format canal "<Gr=~a, Lo=~a, Ca=~a, Co=~a>"
            (posicion-granjero estado)
            (posicion-lobo estado)
            (posicion-cabra estado)
            (posicion-col estado)))
ESCRIBE-ESTADO-GRANJERO
```

LISP: Estructuras

```
> (setf *estado-inicial*
    (crea-estado
     :granjero ´i
     :lobo     ´i
     :cabra    ´i
     :col      ´i))
<Gr=I, Lo=I, Ca=I, Co=I>
> (posicion-granjero *estado-inicial*)
I
> (posicion-cabra *estado-inicial*)
I
> (setf (posicion-granjero *estado-inicial*) ´d)
D
> *estado-inicial*
<Gr=D, Lo=I, Ca=I, Co=I>
```

LISP: Estructuras

```
> (estado-p *estado-inicial*)
T
> (setf *estado-1* (copy-estado *estado-inicial*))
<Gr=D, Lo=I, Ca=I, Co=I>
> (setf (posicion-cabra *estado-inicial*) 'd)
D
> *estado-inicial*
<Gr=D, Lo=I, Ca=D, Co=I>
> *estado-1*
<Gr=D, Lo=I, Ca=I, Co=I>
> (equalp *estado-inicial* *estado-1*)
NIL
> (setf (posicion-cabra *estado-1*) 'd)
D
> (equalp *estado-inicial* *estado-1*)
T
```

Problema del granjero con estructuras

- Representación de estados

```
(defstruct (estado (:constructor crea-estado)
                   (:conc-name posicion-)
                   (:print-function escribe-estado-granjero))
  granjero
  lobo
  cabra
  col)

(defun escribe-estado-granjero (estado &optional (canal t) profundidad)
  (format canal "<Gr=~a, Lo=~a, Ca=~a, Co=~a>"
          (posicion-granjero estado)
          (posicion-lobo estado)
          (posicion-cabra estado)
          (posicion-col estado)))
```

Problema del granjero con estructuras

- Estado inicial

```
(defparameter *estado-inicial*
  (crea-estado :granjero 'i :lobo 'i :cabra 'i :col 'i))
```

- Estado final

```
(defparameter *estado-final*
  (crea-estado :granjero 'd :lobo 'd :cabra 'd :col 'd))

(defun es-estado-final (estado)
  (equalp estado *estado-final*))
```

Problema del granjero con estructuras

- Funciones auxiliares

```
(defun opuesta (posicion)
  (cond ((eq posicion 'i) 'd)
        ((eq posicion 'd) 'i)))

(defun es-seguro (estado)
  (when (and (when (eq (posicion-lobo estado) (posicion-cabra estado))
                   (eq (posicion-granjero estado) (posicion-lobo estado)))
             (when (eq (posicion-cabra estado) (posicion-col estado))
                   (eq (posicion-granjero estado) (posicion-cabra estado))))
        estado))
```

Problema del granjero con estructuras

- Operadores

```
(defparameter *operadores*
  '(pasa-granjero-solo
    pasan-granjero-y-lobo
    pasan-granjero-y-cabra
    pasan-granjero-y-col))

(defun pasa-granjero-solo (estado)
  (es-seguro (crea-estado
              :granjero (opuesta (posicion-granjero estado))
              :lobo (posicion-lobo estado)
              :cabra (posicion-cabra estado)
              :col (posicion-col estado))))
```

Problema del granjero con estructuras

```
(defun pasan-granjero-y-lobo (estado)
  (when (eq (posicion-granjero estado) (posicion-lobo estado))
        (es-seguro (crea-estado
                    :granjero (opuesta (posicion-granjero estado))
                    :lobo (opuesta (posicion-lobo estado))
                    :cabra (posicion-cabra estado)
                    :col (posicion-col estado)))))

(defun pasan-granjero-y-cabra (estado)
  (when (eq (posicion-granjero estado) (posicion-cabra estado))
        (es-seguro (crea-estado
                    :granjero (opuesta (posicion-granjero estado))
                    :lobo (posicion-lobo estado)
                    :cabra (opuesta (posicion-cabra estado))
                    :col (posicion-col estado)))))
```

Problema del granjero con estructuras

```
(defun pasan-granjero-y-col (estado)
  (when (eq (posicion-granjero estado) (posicion-col estado))
        (es-seguro (crea-estado
                    :granjero (opuesta (posicion-granjero estado))
                    :lobo (posicion-lobo estado)
                    :cabra (posicion-cabra estado)
                    :col (opuesta (posicion-col estado))))))
```

Verificación de solución del problema del granjero

```
> (verifica '(pasan-granjero-y-cabra
              pasa-granjero-solo
              pasan-granjero-y-col
              pasan-granjero-y-cabra
              pasan-granjero-y-lobo
              pasa-granjero-solo
              pasan-granjero-y-cabra))
<Gr=I, Lo=I, Ca=I, Co=I> PASAN-GRANJERO-Y-CABRA
<Gr=D, Lo=I, Ca=D, Co=I> PASA-GRANJERO-SOLO
<Gr=I, Lo=I, Ca=D, Co=I> PASAN-GRANJERO-Y-COL
<Gr=D, Lo=I, Ca=D, Co=D> PASAN-GRANJERO-Y-CABRA
<Gr=I, Lo=I, Ca=I, Co=D> PASAN-GRANJERO-Y-LOBO
<Gr=D, Lo=D, Ca=I, Co=D> PASA-GRANJERO-SOLO
<Gr=I, Lo=D, Ca=I, Co=D> PASAN-GRANJERO-Y-CABRA
<Gr=D, Lo=D, Ca=D, Co=D> Estado final
T
```

Lisp: Variables locales

```
* (LET ((VAR-1 VAL-1)...(VAR-M VAL-M)) E-1 ... E-N)
        (setf a 9 b 7)                      =>  7
        (let ((a 2)(b 3)) (+ a b))          =>  5
        (+ a b)                             =>  16
        (let ((a 2)(b (+ 1 a))) (+ a b))    =>  Error

* (LET* ((VAR-1 VAL-1) ... (VAR-N VAL-N)) E-1 ... E-M)
        (let* ((a 2)(b (+ 1 a))) (+ a b))   =>  5
```

Implementación del problema de las jarras con listas

- Representación de estados

```
(defun crea-estado (x y)
  (list x y))

(defun contenido-jarra-4 (estado)
  (first estado))

(defun contenido-jarra-3 (estado)
  (second estado))
```

Implementación del problema de las jarras con listas

- Estado inicial

```
(defparameter *estado-inicial*
  (crea-estado 0 0))
```

- Estados finales

```
(defun es-estado-final (estado)
  (= 2 (contenido-jarra-4 estado)))
```

Implementación del problema de las jarras con listas

- Operadores

```
(defparameter *operadores*
  '(llenar-jarra-4
    llenar-jarra-3
    vaciar-jarra-4
    vaciar-jarra-3
    llenar-jarra-4-con-jarra-3
    llenar-jarra-3-con-jarra-4
    vaciar-jarra-3-en-jarra-4
    vaciar-jarra-4-en-jarra-3))

(defun llenar-jarra-4 (estado)
  (when (< (contenido-jarra-4 estado) 4)
        (crea-estado 4
                     (contenido-jarra-3 estado))))
```

Implementación del problema de las jarras con listas

```
(defun llenar-jarra-3 (estado)
  (when (< (contenido-jarra-3 estado) 3)
        (crea-estado (contenido-jarra-4 estado)
                     3)))

(defun vaciar-jarra-4 (estado)
  (when (> (contenido-jarra-4 estado) 0)
        (crea-estado 0
                     (contenido-jarra-3 estado))))

(defun vaciar-jarra-3 (estado)
  (when (> (contenido-jarra-3 estado) 0)
        (crea-estado (contenido-jarra-4 estado)
                     0)))
```

Implementación del problema de las jarras con listas

```
(defun llenar-jarra-4-con-jarra-3 (estado)
  (let ((x (contenido-jarra-3 estado))
        (y (contenido-jarra-4 estado)))
    (when (and (> x 0)
               (< y 4)
               (> (+ y x) 4))
          (crea-estado 4 (- x (- 4 y))))))

(defun llenar-jarra-3-con-jarra-4 (estado)
  (let ((x (contenido-jarra-3 estado))
        (y (contenido-jarra-4 estado)))
    (when (and (> y 0)
               (< x 3)
               (> (+ y x) 3))
          (crea-estado (- y (- 3 x)) 3))))
```

Implementación del problema de las jarras con listas

```
(defun vaciar-jarra-3-en-jarra-4  (estado)
  (let ((x (contenido-jarra-3 estado))
        (y (contenido-jarra-4 estado)))
    (when (and (> x 0)
               (<= (+ y x) 4))
          (crea-estado (+ x y) 0))))

(defun vaciar-jarra-4-en-jarra-3 (estado)
  (let ((x (contenido-jarra-3 estado))
        (y (contenido-jarra-4 estado)))
    (when (and (> y 0)
               (<= (+ y x) 3))
          (crea-estado 0 (+ x y)))))
```

Problema de las jarras: Verificación de soluciones

```
> (verifica ´(llenar-jarra-4
              llenar-jarra-3-con-jarra-4
              vaciar-jarra-3
              vaciar-jarra-4-en-jarra-3
              llenar-jarra-4
              llenar-jarra-3-con-jarra-4))
(0 0) LLENAR-JARRA-4
(4 0) LLENAR-JARRA-3-CON-JARRA-4
(1 3) VACIAR-JARRA-3
(1 0) VACIAR-JARRA-4-EN-JARRA-3
(0 1) LLENAR-JARRA-4
(4 1) LLENAR-JARRA-3-CON-JARRA-4
(2 3) Estado final
T
```

Problema de las jarras con estructuras

- Representación de estados

```
(defstruct (estado (:conc-name contenido-)
                   (:constructor crea-estado)
                   (:print-function escribe-estado-jarras))
  jarra-4
  jarra-3)

(defun escribe-estado-jarras (estado &optional (canal t) profundidad)
  (format canal "<~a en 4 y ~a en 3>"
          (contenido-jarra-4 estado)
          (contenido-jarra-3 estado)))
```

Problema de las jarras con estructuras

- Estado inicial

```
(defparameter *estado-inicial*
  (crea-estado :jarra-4 0
               :jarra-3 0))

;;; > *estado-inicial*
;;; <0 en 4 y 0 en 3>
```

- Estados finales

```
(defun es-estado-final (estado)
  (= 2 (contenido-jarra-4 estado)))
```

Problema de las jarras con estructuras

- Operadores

```
(defparameter *operadores*
  '(llenar-jarra-4
    llenar-jarra-3
    vaciar-jarra-4
    vaciar-jarra-3
    llenar-jarra-4-con-jarra-3
    llenar-jarra-3-con-jarra-4
    vaciar-jarra-3-en-jarra-4
    vaciar-jarra-4-en-jarra-3))

(defun llenar-jarra-4 (estado)
  (when (< (contenido-jarra-4 estado) 4)
        (crea-estado :jarra-4 4
                     :jarra-3 (contenido-jarra-3 estado))))
```

Problema de las jarras con estructuras

```
(defun llenar-jarra-3 (estado)
  (when (< (contenido-jarra-3 estado) 3)
        (crea-estado :jarra-4 (contenido-jarra-4 estado)
                     :jarra-3 3)))

(defun vaciar-jarra-4 (estado)
  (when (> (contenido-jarra-4 estado) 0)
        (crea-estado :jarra-4 0
                     :jarra-3 (contenido-jarra-3 estado))))

(defun vaciar-jarra-3 (estado)
  (when (> (contenido-jarra-3 estado) 0)
        (crea-estado :jarra-4 (contenido-jarra-4 estado)
                     :jarra-3 0)))
```

Problema de las jarras con estructuras

```
(defun llenar-jarra-4-con-jarra-3 (estado)
  (let ((x (contenido-jarra-3 estado))
        (y (contenido-jarra-4 estado)))
    (when (and (> x 0)
               (< y 4)
               (> (+ y x) 4))
          (crea-estado :jarra-4 4 :jarra-3  (- x (- 4 y))))))

(defun llenar-jarra-3-con-jarra-4 (estado)
  (let ((x (contenido-jarra-3 estado))
        (y (contenido-jarra-4 estado)))
    (when (and (> y 0)
               (< x 3)
               (> (+ y x) 3))
          (crea-estado :jarra-4 (- y (- 3 x)) :jarra-3 3))))
```

Problema de las jarras con estructuras

```
(defun vaciar-jarra-3-en-jarra-4  (estado)
  (let ((x (contenido-jarra-3 estado))
        (y (contenido-jarra-4 estado)))
    (when (and (> x 0)
               (<= (+ y x) 4))
          (crea-estado :jarra-4 (+ x y)
                       :jarra-3  0))))

(defun vaciar-jarra-4-en-jarra-3 (estado)
  (let ((x (contenido-jarra-3 estado))
        (y (contenido-jarra-4 estado)))
    (when (and (> y 0)
               (<= (+ y x) 3))
          (crea-estado :jarra-4 0
                       :jarra-3  (+ x y)))))
```

Problema de las jarras:Verificación de soluciones

```
> (verifica ´(llenar-jarra-4
              llenar-jarra-3-con-jarra-4
              vaciar-jarra-3
              vaciar-jarra-4-en-jarra-3
              llenar-jarra-4
              llenar-jarra-3-con-jarra-4))
<0 en 4 y 0 en 3> LLENAR-JARRA-4
<4 en 4 y 0 en 3> LLENAR-JARRA-3-CON-JARRA-4
<1 en 4 y 3 en 3> VACIAR-JARRA-3
<1 en 4 y 0 en 3> VACIAR-JARRA-4-EN-JARRA-3
<0 en 4 y 1 en 3> LLENAR-JARRA-4
<4 en 4 y 1 en 3> LLENAR-JARRA-3-CON-JARRA-4
<2 en 4 y 3 en 3> Estado final
T
```

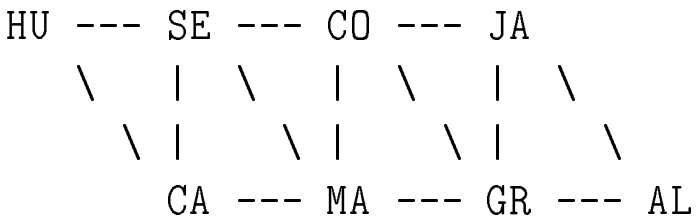
Lisp: Predicado de pertenencia

```
* (MEMBER E L)
        (member 'x '(a x b x c))          => (X B X C)
        (member 'x '(a (x) b))            => NIL
        (setq l' ((a b) (c d)))           => ((A B) (C D))
        (member '(c d) l)                 => NIL
        (member 2.0 '(1 2 3))             => NIL

* (MEMBER E L :TEST #'PREDICADO)
        (member '(c d) l)                 => NIL
        (member '(c d) l :test #'equal)   => ((C D))
        (member 2.0 '(1 2 3))             => NIL
        (member 2.0 '(1 2 3) :test #'=)   => (2 3)
        (member 2.0 '(1 2 3) :test #'<)   => (3)
```

Implementación del problema del viaje

- Enunciado



- Representación de estados

```
;;; almeria, cadiz, cordoba, granada, huelva, jaen, malaga, sevilla.
```

Implementación del problema del viaje

- Estado inicial

```
(defparameter *estado-inicial* 'sevilla)
```

- Estado final

```
(defparameter *estado-final* 'almeria)

(defun es-estado-final (estado)
  (eq estado *estado-final*))
```

Implementación del problema del viaje

- Operadores

```
(defparameter *operadores*
  '(ir-a-almeria
    ir-a-cadiz
    ir-a-cordoba
    ir-a-granada
    ir-a-huelva
    ir-a-jaen
    ir-a-malaga
    ir-a-sevilla))

(defun ir-a-almeria (estado)
  (when (member estado '(granada jaen))
        'almeria))
```

Implementación del problema del viaje

```
(defun ir-a-cadiz (estado)
  (when (member estado '(huelva sevilla malaga))
        'cadiz))

(defun ir-a-cordoba (estado)
  (when (member estado '(sevilla malaga granada jaen))
        'cordoba))

(defun ir-a-granada (estado)
  (when (member estado '(cordoba malaga jaen almeria))
        'granada))

(defun ir-a-huelva (estado)
  (when (member estado '(sevilla cadiz))
        'huelva))
```

Implementación del problema del viaje

```
(defun ir-a-jaen (estado)
  (when (member estado '(cordoba granada))
        'jaen))

(defun ir-a-malaga (estado)
  (when (member estado '(sevilla cadiz cordoba granada))
        'malaga))

(defun ir-a-sevilla (estado)
  (when (member estado '(cadiz cordoba huelva malaga))
        'sevilla))
```

Solución del problema del viaje

```
> (load "viaje.lp")
T
> (load "verifica.lsp")
T
> (verifica ´(ir-a-cordoba ir-a-granada ir-a-almeria))
SEVILLA IR-A-CORDOBA
CORDOBA IR-A-GRANADA
GRANADA IR-A-ALMERIA
ALMERIA Estado final
T
```

Lisp: Matrices

- Creación:

```
* (MAKE-ARRAY DIMENSIONES :INITIAL-CONTENTS EXPRESION)
      > (make-array '(2 2))
      #2A((NIL NIL) (NIL NIL))
      > (make-array '(2 1))
      #2A((NIL) (NIL))
      > (make-array '(1 2))
      #2A((NIL NIL))
      > (make-array '(2 2 1))
      #3A(((NIL) (NIL)) ((NIL) (NIL)))
      > (make-array '(2 1 2))
      #3A(((NIL NIL)) ((NIL NIL)))
```

Lisp: Matrices

```
> (make-array '(3 3)
              :initial-contents '((1 2 3)
                                  (8 H 4)
                                  (7 6 5)))
#2A((1 2 3) (8 H 4) (7 6 5))
> (setf *estado-final*
    (make-array '(3 3)
                :initial-contents '((1 2 3)
                                    (8 H 4)
                                    (7 6 5))))
#2A((1 2 3) (8 H 4) (7 6 5))
```

Lisp: Matrices

- Lectura:

```
* (AREF MATRIZ INDICE-1 ... INDICE-N)
      > *estado-final*
      #2A((1 2 3) (8 H 4) (7 6 5))
      > (aref *estado-final* 0 0)
      1
      > (aref *estado-final* 1 1)
      H
      > (aref *estado-final* 2 2)
      5
```

Lisp: Matrices

- Modificación:

```
* (SETF (AREF MATRIZ INDICE-1 ... INDICE-N) EXPRESION)
      > *estado-final*
      #2A((1 2 3) (8 H 4) (7 6 5))
      > (setf (aref *estado-final* 1 2) 'h)
      H
      > (setf (aref *estado-final* 1 1) 4)
      4
      > *estado-final*
      #2A((1 2 3) (8 4 H) (7 6 5))
```

Lisp: Bucles

```
> (let ((res nil))
    (loop for x from 1 to 7 do
          (setf res (cons x res)))
    res)
(7 6 5 4 3 2 1)
> (loop for x from 1 to 7
        collect (* x x))
(1 4 9 16 25 36 49)
> (loop for x from 1 to 7
        when (evenp x)
        collect (* x x))
(4 16 36)
> (loop for x from 1 to 7
        when (evenp x)
        summing (* x x))
56
```

Lisp: Bucles

```
> (loop for x from 1 to 7 by 2
        collect (* x x))
(1 9 25 49)
> (loop for x in '(1 3 5)
        summing x)
9
> (let ((x 3)
        (res nil))
    (loop while (> x 0) do
          (setf res (cons x res)
                x (- x 1)))
    res)
(1 2 3)
```

Lisp: Bucles

```
> (let ((x 3)
        (res nil))
    (loop until (<= x 0) do
          (setf res (cons x res)
                x (- x 1)))
    res)
(1 2 3)
> (defun factor (x)
    (or (loop for i from 2 to (sqrt x)
              thereis (when (= (mod x i) 0)
                            i))
        x))
FACTOR
> (factor 35)
5
```

Lisp: Bucles

```
> (defun es-primo (x)
    (= x (factor x)))
ES-PRIMO
> (es-primo 7)
T
> (es-primo 35)
NIL
> (loop for x from 2 to 100
       count (es-primo x))
25
> (defun primos-gemelos (x)
    (when (and (es-primo x)
               (es-primo (+ x 2)))
          (list x (+ x 2))))
PRIMOS-GEMELOS
```

Lisp: Bucles

```
> (loop for i from 200 to 2000
        thereis (primos-gemelos i))
(227 229)
> (loop for x from 2 to 100
        count (primos-gemelos x))
8
> (defun suma-primeros-impares (x)
    (let ((resultado 0))
      (loop until (null x) do
            (if (= (mod (first x) 2) 0)
                (return resultado)
                (setf resultado (+ resultado (first x))
                      x (rest x))))))
SUMA-PRIMEROS-IMPARES
> (suma-primeros-impares '(1 3 2 5 6))
4
```

Lisp: Bucles

- **Opciones iniciales:**

```
(LOOP FOR <VARIABLE> FROM <INICIO> TO <FIN> ...)
(LOOP FOR <VARIABLE> FROM <INICIO> TO <FIN> BY <INCREMENTO> ...)
(LOOP FOR <VARIABLE> IN <LISTA> ...)
(LOOP WHILE <CONDICION> ...)
(LOOP UNTIL <CONDICION> ...)
```

- **Opciones centrales:**

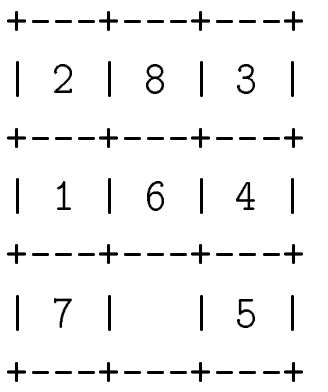
```
(LOOP ... WHEN <CONDICION> ...)
```

- **Opciones finales:**

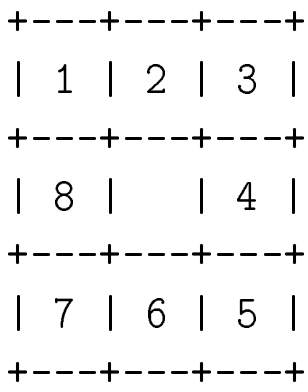
```
(LOOP ... DO <EXPRESION>)
(LOOP ... COLLECT <EXPRESION>)
(LOOP ... THEREIS <EXPRESION>)
(LOOP ... COUNT <EXPRESION>)
(LOOP ... SUMMING <EXPRESION>)
```

Implementación del 8–puzzle

- Enunciado:



Estado inicial



Estado final

Implementación del 8–puzzle

- Estado inicial

```
(defparameter *estado-inicial*
  (make-array '(3 3)
              :initial-contents '((2 8 3)
                                  (1 6 4)
                                  (7 h 5))))
```

- Estado final

```
(defparameter *estado-final*
  (make-array '(3 3)
              :initial-contents '((1 2 3)
                                  (8 h 4)
                                  (7 6 5))))
(defun es-estado-final (estado)
  (equalp estado *estado-final*))
```

Implementación del 8–puzzle

- Funciones auxiliares

```
;;;; (COPIA-TABLERO TABLERO)
;;;  > *estado-final*
;;;  #2A((1 2 3) (8 H 4) (7 6 5))
;;;  > (setf tablero-2 (copia-tablero *estado-final*))
;;;  #2A((1 2 3) (8 H 4) (7 6 5))
;;;  > (setf (aref tablero-2 2 1) 'h (aref tablero-2 1 1) 6)
;;;  8
;;;  > tablero-2
;;;  #2A((1 2 3) (8 6 4) (7 H 5))
;;;  > *estado-final*
;;;  #2A((1 2 3) (8 H 4) (7 6 5))
```

Implementación del 8–puzzle

```
(defun copia-tablero (tablero)
  (let ((nuevo-tablero (make-array '(3 3))))
    (loop for i from 0 to 2
          do (loop for j from 0 to 2
                   do (setf (aref nuevo-tablero i j)
                            (aref tablero i j))))
    nuevo-tablero))
```

Implementación del 8–puzzle

```
;;;; (COORDENADAS BLOQUE TABLERO)
;;;  *estado-final*                    =>  #2A((1 2 3) (8 H 4) (7 6 5))
;;;  (coordenadas 6 *estado-final*)    =>  (2 1)
;;;  (coordenadas 'h *estado-final*)   =>  (1 1)
(defun coordenadas (bloque tablero)
  (loop for i from 0 to 2
        thereis (loop for j from 0 to 2
                      thereis
                      (when (eq (aref tablero i j) bloque)
                            (list i j)))))
```

Implementación del 8–puzzle

- Operadores

```
(defparameter *operadores*
  '(mover-izquierda
    mover-arriba
    mover-derecha
    mover-abajo))

(defun mover-izquierda (estado)
  (let* ((lugar-del-hueco (coordenadas 'h estado))
         (i (first lugar-del-hueco))
         (j (second lugar-del-hueco))
         (nuevo-estado (copia-tablero estado)))
    (when (> j 0)
          (setf (aref nuevo-estado i j) (aref nuevo-estado i (- j 1)))
          (setf (aref nuevo-estado i (- j 1)) 'h)
          nuevo-estado)))
```

Implementación del 8–puzzle

```
(defun mover-arriba (estado)
  (let* ((lugar-del-hueco (coordenadas 'h estado))
         (i (first lugar-del-hueco))
         (j (second lugar-del-hueco))
         (nuevo-estado (copia-tablero estado)))
    (when (> i 0)
          (setf (aref nuevo-estado i j) (aref nuevo-estado (- i 1) j))
          (setf (aref nuevo-estado (- i 1) j) 'h)
          nuevo-estado)))
```

Implementación del 8–puzzle

```
(defun mover-derecha(estado)
  (let* ((lugar-del-hueco (coordenadas 'h estado))
         (i (first lugar-del-hueco))
         (j (second lugar-del-hueco))
         (nuevo-estado (copia-tablero estado)))
    (when (< j 2)
          (setf (aref nuevo-estado i j) (aref nuevo-estado  i (+ j 1)))
          (setf (aref nuevo-estado  i (+ j 1)) 'h)
          nuevo-estado)))
```

Implementación del 8–puzzle

```
(defun mover-abajo (estado)
  (let* ((lugar-del-hueco (coordenadas 'h estado))
         (i (first lugar-del-hueco))
         (j (second lugar-del-hueco))
         (nuevo-estado (copia-tablero estado)))
    (when (< i 2)
          (setf (aref nuevo-estado i j) (aref nuevo-estado (+ i 1) j))
          (setf (aref nuevo-estado (+ i 1) j) 'h)
          nuevo-estado)))
```

Solución del 8–puzzle

```
> (load "8-puzzle.lsp")
T
> (load "verifica.lsp")
T
> (verifica ´(mover-arriba
              mover-arriba
              mover-izquierda
              mover-abajo
              mover-derecha))
#2A((2 8 3) (1 6 4) (7 H 5)) MOVER-ARRIBA
#2A((2 8 3) (1 H 4) (7 6 5)) MOVER-ARRIBA
#2A((2 H 3) (1 8 4) (7 6 5)) MOVER-IZQUIERDA
#2A((H 2 3) (1 8 4) (7 6 5)) MOVER-ABAJO
#2A((1 2 3) (H 8 4) (7 6 5)) MOVER-DERECHA
#2A((1 2 3) (8 H 4) (7 6 5)) Estado final
T
```

Bibliografía

- Haton–91:
  - Cap. 2: "Resolución de problemas".
- Rich–94:
  - Cap. 2: "Problemas, espacios problema y búsqueda".