Técnicas heurísticas en juegos

José A. Alonso y Francisco J. Martín

Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Ejemplo de juego: Nim

- **Situación inicial: Una pila con N fichas.**
- **Jugadas: Coger 1, 2 ó 3 fichas de la pila.**
- **Objetivo: Obligar al adversario a coger la última ficha.**

```
> (juego :orden 8 :nivel 8 :empieza-la-maquina? t)
8 Mi turno.
5 Tu turno: 1
4 Mi turno.
1 Tu turno: 1
0 La maquina ha ganado
NIL
```

Representación del Nim

- **Representación de jugadores**

```
(defun contrario (jugador)
  (if (eq jugador 'max)
      'min
      'max))
```

- **Representación de estados**

```
(defstruct (estado (:constructor crea-estado)
                   (:conc-name))
  tablero
  jugador
  valor)
```

Representación del Nim

- **Generador de juegos**

```
(defvar *orden* 8)

(defun crea-juego (jugador n)
  (setf *orden* n)
  (crea-estado-inicial jugador)
  (crea-movimientos)
  (list 'nim n))
```

Representación del Nim

- **Estado inicial**

```
(defvar *estado-inicial*)

(defun crea-estado-inicial (jugador)
  (setf *estado-inicial*
        (crea-estado :tablero *orden*
                     :jugador jugador)))
```

- **Estados finales**

```
(defun es-estado-final (estado)
  (= (tablero estado) 0))

(defun es-estado-ganador (estado jugador)
  (and (es-estado-final estado)
       (eq (jugador estado) jugador)))
```

Representación del Nim

- **Movimientos**

```
(defvar *movimientos*)

(defun crea-movimientos ()
  (setf *movimientos* '(3 2 1)))

(defun es-movimiento-legal (movimiento estado)
  (<= 1 movimiento (min 3 (tablero estado))))

(defun aplica-movimiento (movimiento estado)
  (crea-estado :tablero (- (tablero estado) movimiento)
               :jugador (contrario (jugador estado))))
```

Representación del Nim

- **Sucesores**

```
(defun sucesores (estado)
  (loop for movimiento in *movimientos*
    when (es-movimiento-legal movimiento estado)
    collect (aplica-movimiento movimiento estado)))
```

- **Escritura de estados**

```
(defun escribe-estado (estado)
  (format t "~&~a " (tablero estado)))
```

Representación del Nim

- **Función de evaluación estática**

```
(defparameter *minimo-valor* -1)

(defparameter *maximo-valor* 1)

(defun f-e-estatica (estado)
  (if (es-estado-final estado)
      (if (eq (jugador estado) 'max)
          *maximo-valor*
          *minimo-valor*)
      0))
```

Elementos del juego

- **Jugadores:**
  - **MIN: humano**
  - **MAX: máquina**
- **Estados:**
  - **Tablero.**
  - **Jugador.**
  - **Valor.**

Procedimientos para definir juegos

```
(CONTRARIO JUGADOR)
(CREA-ESTADO &KEY TABLERO JUGADOR VALOR)
(TABLERO ESTADO) (JUGADOR ESTADO) (VALOR ESTADO)

(CREA-JUEGO JUGADOR N)
*ESTADO-INICIAL* (CREA-ESTADO-INICIAL JUGADOR)
(ES-ESTADO-FINAL ESTADO)
(ES-ESTADO-GANADOR ESTADO JUGADOR)
*MOVIMIENTOS* (CREA-MOVIMIENTOS)
(ES-MOVIMIENTO-LEGAL MOVIMIENTO ESTADO)
(APLICA-MOVIMIENTO MOVIMIENTO ESTADO)
(SUCESORES ESTADO)
*MINIMO-VALOR* *MAXIMO-VALOR* (F-E-ESTATICA ESTADO)
(ESCRIBE-ESTADO ESTADO)
```

Procedimiento de control de juegos

```
(defun juego (&key orden
                   (nivel 2)
                   (empieza-la-maquina? nil)
                   (procedimiento #'minimax))
  (setf *maxima-profundidad* nivel
        *procedimiento*      procedimiento)
  (cond (empieza-la-maquina?
         (crea-juego 'max orden)
         (escribe-estado *estado-inicial*)
         (if (es-estado-final *estado-inicial*)
             (analiza-final *estado-inicial*)
             (jugada-maquina *estado-inicial*)))
        (t (crea-juego 'min orden)
           (escribe-estado *estado-inicial*)
           (jugada-humana *estado-inicial*))))
```

Procedimiento de control de juegos

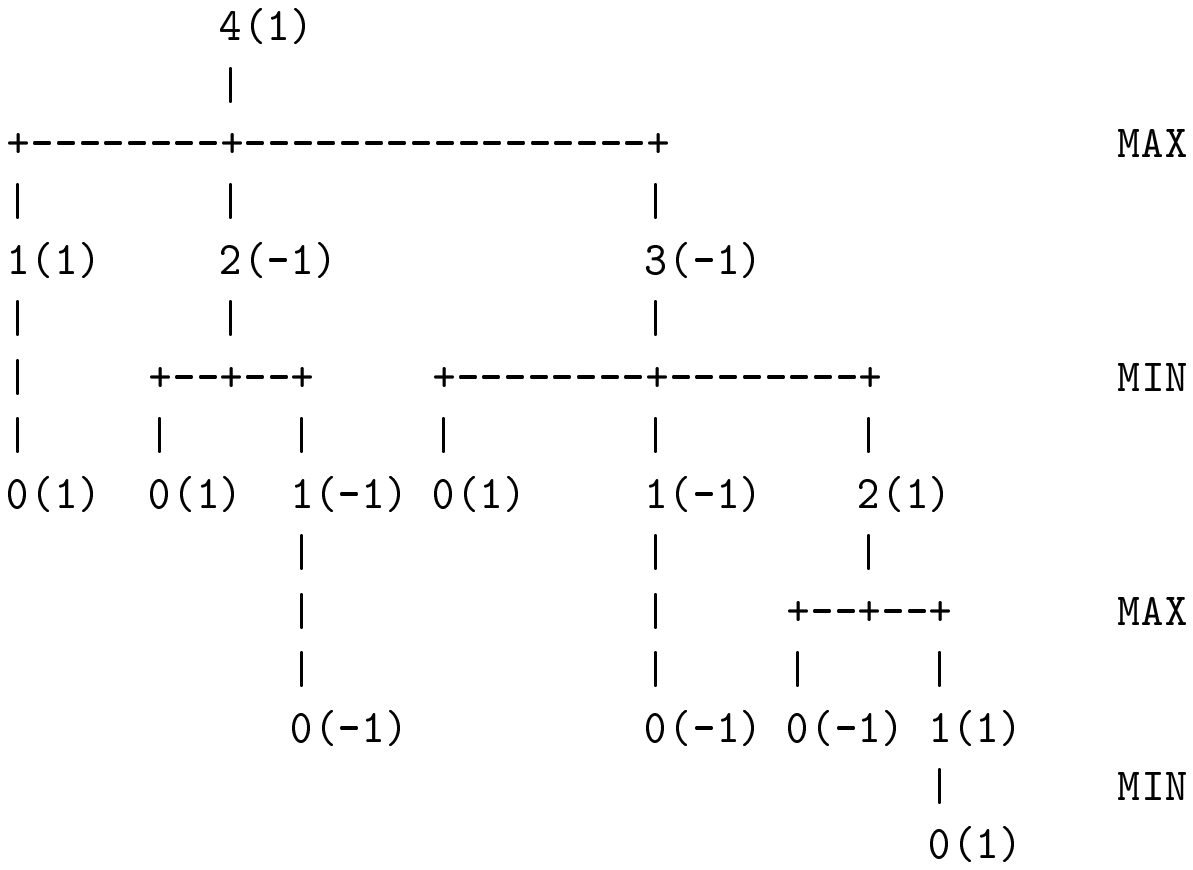
```
(defun jugada-humana (estado)
  (format t "Tu turno: ")
  (let ((m (read)))
    (cond ((es-movimiento-legal m estado)
           (let ((siguiente (aplica-movimiento m estado)))
             (escribe-estado siguiente)
             (if (es-estado-final siguiente)
                 (analiza-final siguiente)
                 (jugada-maquina siguiente))))
          (t (format t "~&   ~a es ilegal. " m)
             (jugada-humana estado)))))
```

Procedimiento de control de juegos

```
(defun analiza-final (estado-final)
  (cond ((es-estado-ganador estado-final 'max)
         (format t "La maquina ha ganado"))
        ((es-estado-ganador estado-final 'min)
         (format t "El humano ha ganado"))
        (t (format t "Empate"))))

(defun jugada-maquina (estado)
  (format t "Mi turno.")
  (let ((siguiente (funcall *procedimiento* estado 0)))
    (escribe-estado siguiente)
    (if (es-estado-final siguiente)
        (analiza-final siguiente)
        (jugada-humana siguiente))))
```

Procedimiento Minimax: Ejemplo



Procedimiento Minimax: Implementación

```
(defun minimax (estado profundidad)
  (if (or (es-estado-final estado)
          (es-bastante-profunda profundidad))
      (crea-estado :valor (f-e-estatica estado)
                   :jugador (contrario (jugador estado)))
      (let ((sucesores (sucesores estado)))
        (if (null sucesores)
            (crea-estado :valor (f-e-estatica estado)
                         :jugador (contrario (jugador estado)))
            (if (eq (jugador estado) 'max)
                (maximizador sucesores profundidad)
                (minimizador sucesores profundidad))))))

(defun es-bastante-profunda (profundidad)
  (>= profundidad *maxima-profundidad*))
```

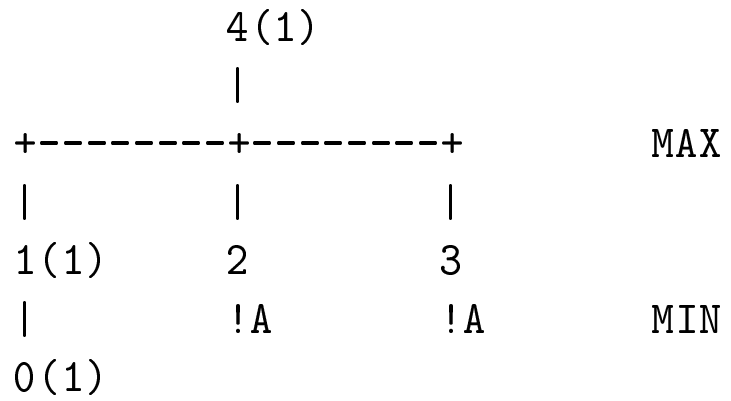
Procedimiento Minimax: Implementación

```
(defun maximizador (sucesores profundidad)
  (let ((mejor-sucesor (first sucesores))
        (mejor-valor *minimo-valor*))
    (loop for sucesor in sucesores do
      (setf valor (valor (minimax sucesor (1+ profundidad))))
      (when (> valor mejor-valor)
            (setf mejor-valor valor)
            (setf mejor-sucesor sucesor)))
    (crea-estado :valor mejor-valor
                 :jugador 'min
                 :tablero (tablero mejor-sucesor))))
```

Procedimiento Minimax: Implementación

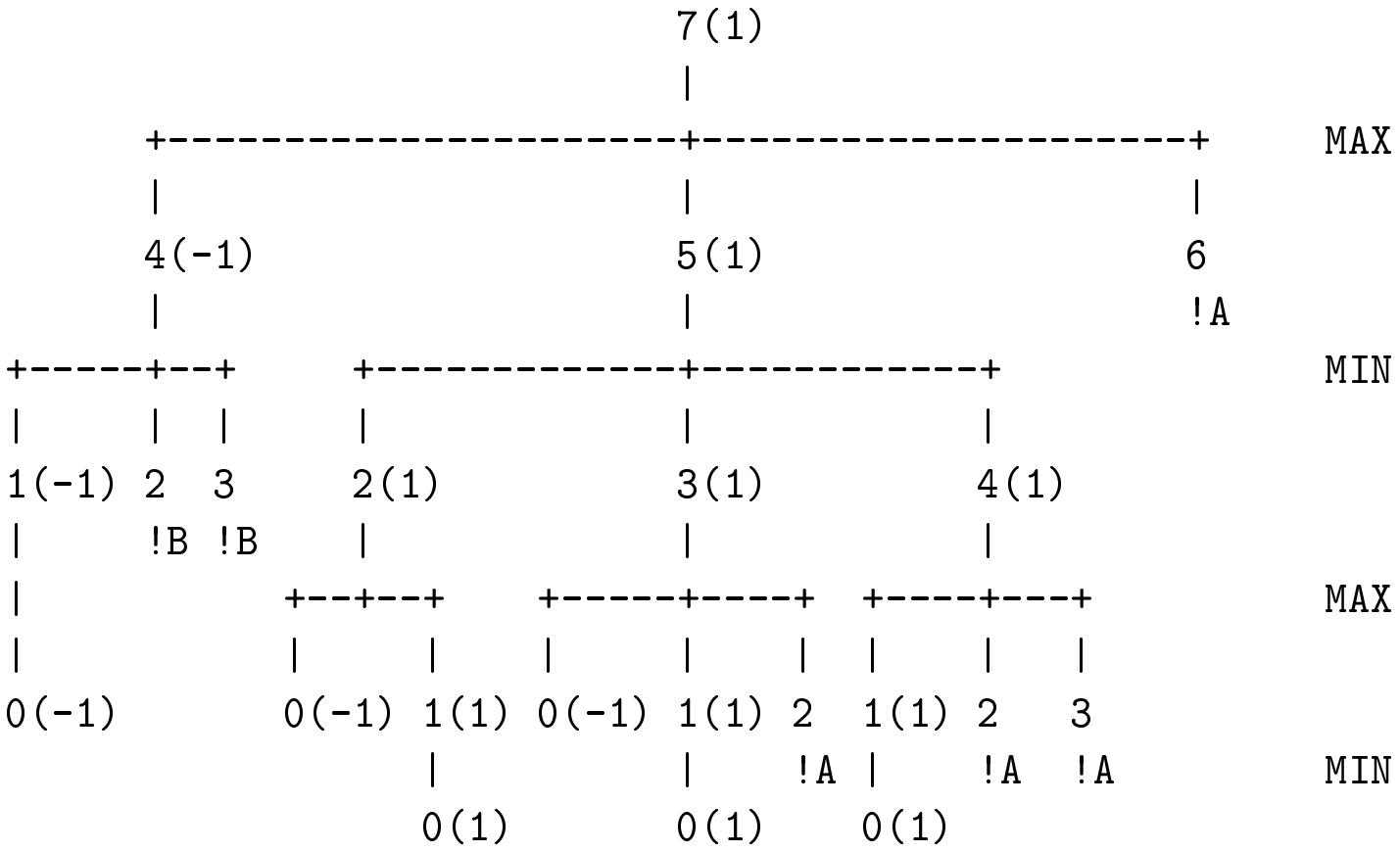
```
(defun minimizador (sucesores profundidad)
  (let ((mejor-sucesor (first sucesores))
        (mejor-valor *maximo-valor*))
    (loop for sucesor in sucesores do
      (setf valor (valor (minimax sucesor (1+ profundidad))))
      (when (< valor mejor-valor)
            (setf mejor-valor valor)
            (setf mejor-sucesor sucesor)))
    (crea-estado :valor mejor-valor
                 :jugador 'max
                 :tablero (tablero mejor-sucesor))))
```

Poda alfa–beta: Ejemplo



- **Principio alfa–beta: Si se tiene una buena (mala) idea, no perder el tiempo en averiguar lo buena (mala) que es.**

Poda alfa–beta: Ejemplo



Poda alfa–beta: Implementación

```
(defun minimax-a-b (estado profundidad
                    &optional (alfa *minimo-valor*) (beta *maximo-valor*))
  (if (or (es-estado-final estado)
          (es-bastante-profunda profundidad))
      (crea-estado :valor (f-e-estatica estado)
                   :jugador (contrario (jugador estado)))
      (let ((sucesores (sucesores estado)))
        (if (null sucesores)
            (crea-estado :valor (f-e-estatica estado)
                         :jugador (contrario (jugador estado)))
            (if (eq (jugador estado) 'max)
                (maximizador-a-b sucesores profundidad alfa beta)
                (minimizador-a-b sucesores profundidad alfa beta))))))
```

Poda alfa–beta: Implementación

```
(defun maximizador-a-b (sucesores profundidad alfa beta)
  (let ((mejor-sucesor (first sucesores))
        (valor 0))
    (loop for sucesor in sucesores do
      (setf valor
            (valor (minimax-a-b sucesor (1+ profundidad) alfa beta)))
      (when (> valor alfa)
            (setf alfa valor)
            (setf mejor-sucesor sucesor))
      (when (>= alfa beta)
            (return)))
    (crea-estado :tablero (tablero mejor-sucesor)
                 :jugador (jugador mejor-sucesor)
                 :valor alfa)))
```

Poda alfa–beta: Implementación

```
(defun minimizador-a-b (sucesores profundidad alfa beta)
  (let ((mejor-sucesor (first sucesores))
        (valor 0))
    (loop for sucesor in sucesores do
      (setf valor
            (valor (minimax-a-b sucesor (1+ profundidad) alfa beta)))
      (when (< valor beta)
            (setf beta valor)
            (setf mejor-sucesor sucesor))
      (when (>= alfa beta)
            (return)))
    (crea-estado :tablero (tablero mejor-sucesor)
                 :jugador (jugador mejor-sucesor)
                 :valor beta)))
```

Complejidad de minimax y alfa–beta

- **Complejidad:**
  - r: factor de ramificación.
  - p: profundidad de la búsqueda.
  - Complejidad en tiempo de minimax: $O(r^p)$.
  - Complejidad en tiempo de minimax con alfa–beta: $O(r^{p/2})$.
- **Aplicación al ajedrez**
  - Factor de ramificación $= 35$
  - Profundidad de la búsqueda $= 100$.
  - Complejidad en tiempo de minimax: $O(16^{33}) \cong 10^{40}$.

Complejidad de minimax y alfa–beta

- **Estados evaluados**
  - **empezando la máquina,**
  - **con profundidad 20 y**
  - **eligiendo el humano siempre 1.**

	8	12	16	20
minimax	88	1.051	11.624	133.039
minimax-a-b	6	25	98	379

Complejidad de minimax y alfa–beta

- **Tiempo y espacio para orden 16:**

	Tiempo	Espacio
minimax	17.28 sec.	1.427.868 bytes
minimax-a-b	0.15 sec.	19.356 bytes

Nim: Segunda función de evaluación

```
(defun f-e-estatica-2 (estado)
  (case (jugador estado)
        (max (if (= (rem (tablero estado) 4) 1)
                 *minimo-valor*
                 *maximo-valor*))
        (min (if (= (rem (tablero estado) 4) 1)
                 *maximo-valor*
                 *minimo-valor*))))
```

Nim: Segunda función de evaluación

```
> (time (juego :orden 16 :nivel 1 :empieza-la-maquina? t
               :procedimiento #'minimax))
16 Mi turno.
13 Tu turno: 1
12 Mi turno.
9 Tu turno: 1
8 Mi turno.
5 Tu turno: 1
4 Mi turno.
1 Tu turno: 1
0 La maquina ha ganado
Real time: 2.483688 sec.
Run time: 0.02 sec.
Space: 3388 Bytes
NIL
```

3 en raya: Ejemplo de juego

```
> (juego :empieza-la-maquina? t :procedimiento #'minimax-a-b)
 . . .   0 1 2
 . . .   3 4 5
 . . .   6 7 8
Mi turno.
 . . .   0 1 2
 . X .   3 4 5
 . . .   6 7 8
Tu turno: 1
 . O .   0 1 2
 . X .   3 4 5
 . . .   6 7 8
```

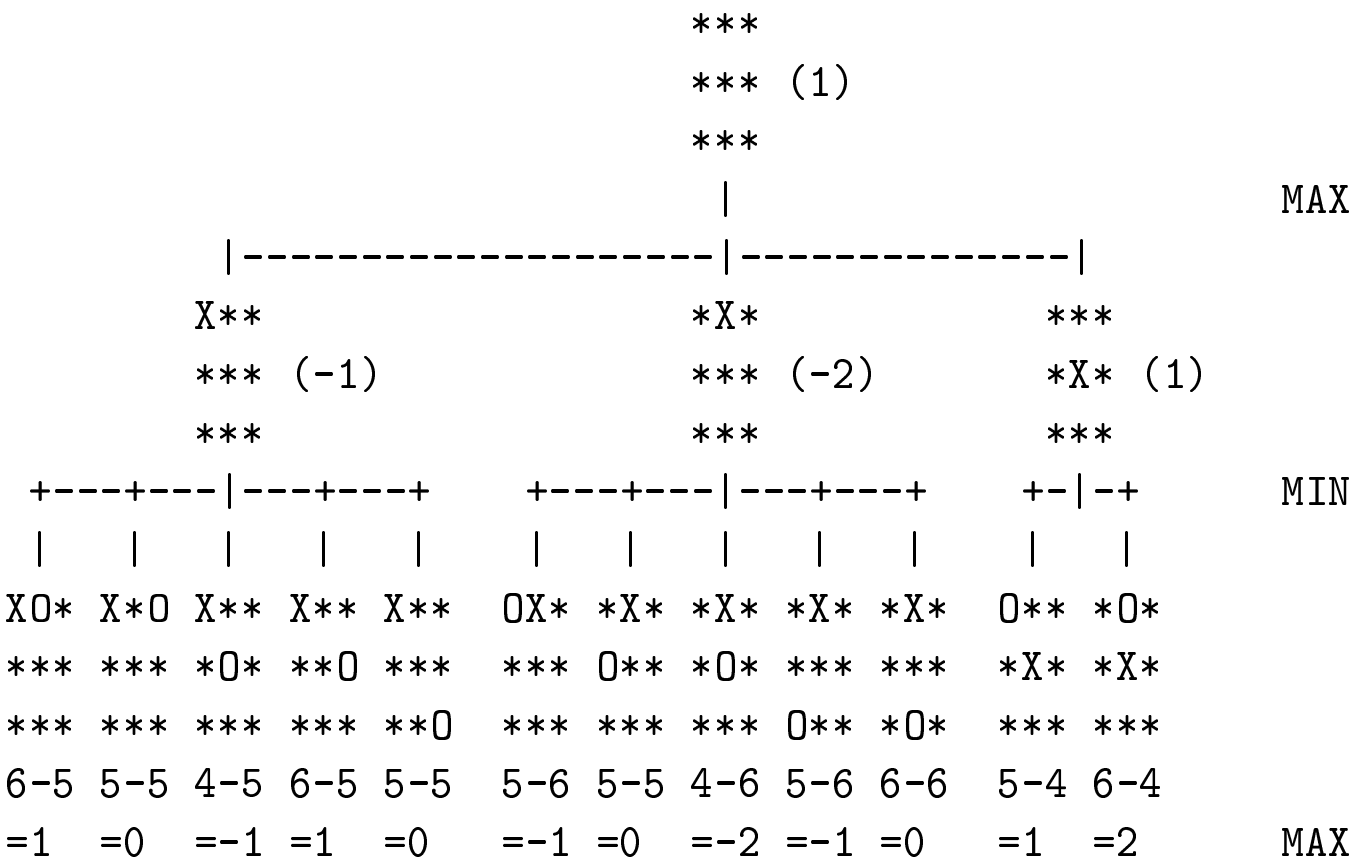
3 en raya: Ejemplo de juego

```
Mi turno.
 X O .   0 1 2
 . X .   3 4 5
 . . .   6 7 8
Tu turno: 8
 X O .   0 1 2
 . X .   3 4 5
 . . O   6 7 8
Mi turno.
 X O X   0 1 2
 . X .   3 4 5
 . . O   6 7 8
```

3 en raya: Ejemplo de juego

```
Tu turno: 7
 X O X   0 1 2
 . X .   3 4 5
 . O O   6 7 8
Mi turno.
 X O X   0 1 2
 . X .   3 4 5
 X O O   6 7 8
La maquina ha ganado
NIL
```

3 en raya: Evaluación



3 en raya: Representación

- **Jugadores**

```
(defun ficha (jugador)
  (if (eq jugador 'max)
      'x
      'o))
```

- **Representación de estados**

```
Tableros = Vectores
        012
        345
        678
```

3 en raya: Representación

- **Generador de juegos**

```
(defun crea-juego (jugador n)
  (crea-estado-inicial jugador)
  (crea-movimientos)
  (list '3-en-raya))
```

- **Estado inicial**

```
(defvar *estado-inicial*)

(defun crea-estado-inicial (jugador)
  (setf *estado-inicial*
        (crea-estado :tablero (make-array 9)
                     :jugador jugador)))
```

3 en raya: Representación

- **Estados finales**

```
(defun es-estado-final (estado)
  (or (es-estado-ganador estado 'max)
      (es-estado-ganador estado 'min)
      (es-estado-completo estado)))

(defun es-estado-ganador (estado jugador)
  (loop for linea in *lineas*
    thereis (es-linea-ganadora linea (tablero estado) jugador)))

(defvar *lineas* '((0 1 2) (3 4 5) (6 7 8)
                   (0 3 6) (1 4 7) (2 5 8)
                   (0 4 8) (2 4 6)))
```

3 en raya: Representación

```
(defun es-linea-ganadora (linea tablero jugador)
  (loop for lugar in linea
    always (eq (aref tablero lugar) (ficha jugador))))

(defun es-estado-completo (estado)
  (not (position nil (tablero estado))))
```

- **Movimientos**

```
(defvar *movimientos*)

(defun crea-movimientos ()
  (setf *movimientos*
        (loop for i from 0 to 8
          collect i)))
```

3 en raya: Representación

```
(defun es-movimiento-legal (movimiento estado)
  (and (numberp movimiento)
       (not (aref (tablero estado) movimiento))))

(defun aplica-movimiento (movimiento estado)
  (when (es-movimiento-legal movimiento estado)
        (let ((jugador (jugador estado))
              (nuevo-tablero (copia-tablero (tablero estado))))
          (setf (aref nuevo-tablero movimiento) (ficha jugador))
          (crea-estado :tablero nuevo-tablero
                       :jugador (contrario jugador)))))
```

3 en raya: Representación

```
(defun copia-tablero (tablero)
  (let ((nuevo-tablero (make-array 9)))
    (loop for i from 0 to 8 do
      (setf (aref nuevo-tablero i)
            (aref tablero i)))
    nuevo-tablero))
```

- **Sucesores**

```
(defun sucesores (estado)
  (elimina-simetricos (todos-los-sucesores estado)))

(defun todos-los-sucesores (estado)
  (loop for movimiento in *movimientos*
    when (es-movimiento-legal movimiento estado)
    collect (aplica-movimiento movimiento estado)))
```

3 en raya: Representación

```
(defun elimina-simetricos (lista-de-tableros)
  (delete-duplicates lista-de-tableros
                     :test #'son-simetricos
                     :from-end t))

(defun son-simetricos (estado-1 estado-2)
  (let ((t-1 (tablero estado-1))
        (t-2 (tablero estado-2)))
    (or (son-simetricos-respecto-columna-central t-1 t-2)
        (son-simetricos-respecto-fila-central t-1 t-2)
        (son-simetricos-respecto-diagonal-principal t-1 t-2)
        (son-simetricos-respecto-diagonal-secundaria t-1 t-2))))
```

3 en raya: Representación

```
(defun fila (i tablero)
  (loop for x from (* 3 i) to (+ (* 3 i) 2)
    collect (aref tablero x)))

(defun columna (j tablero)
  (loop for x from j to (+ j (* 3 2)) by 3
    collect (aref tablero x)))

(defun son-simetricos-respecto-columna-central (tablero-1 tablero-2)
  (loop for j from 0 to 2
    always (equal (columna j tablero-1)
                  (columna (- 2 j) tablero-2))))
```

3 en raya: Representación

```
(defun son-simetricos-respecto-fila-central (tablero-1 tablero-2)
  (loop for i from 0 to 2
    always (equal (fila i tablero-1)
                  (fila (- 2 i) tablero-2))))

(defun son-simetricos-respecto-diagonal-secundaria (t-1 t-2)
  (loop for j from 0 to 2
    always (equal (columna j t-1)
                  (fila j t-2))))

(defun son-simetricos-respecto-diagonal-principal (t-1 t-2)
  (loop for j from 0 to 2
    always (equal (columna j t-1)
                  (reverse (fila (- 2 j) t-2)))))
```

3 en raya: Representación

- **Función de evaluación estática**

```
(defparameter *minimo-valor* -99999)

(defparameter *maximo-valor* 99999)

(defun f-e-estatica (estado)
  (cond ((es-estado-ganador estado 'max) *maximo-valor*)
        ((es-estado-ganador estado 'min) *minimo-valor*)
        ((es-estado-completo estado) 0)
        (t (- (valor-estatico estado 'max)
              (valor-estatico estado 'min)))))
```

3 en raya: Representación

```
(defun valor-estatico (estado jugador)
  (loop for linea in *lineas*
    counting (not (esta (tablero estado)
                        (ficha (contrario jugador))
                        linea))))

(defun esta (tablero ficha linea)
  (loop for i in linea
    thereis (eq (aref tablero i) ficha)))
```

3 en raya: Representación

- **Escritura de estados**

```
(defun escribe-estado (estado)
  (let ((tablero (tablero estado)))
    (loop for i from 0 to 6 by 3 do
      (format t "~%")
      (loop for j from 0 to 2 do
        (format t " ~d" (or (aref tablero (+ i j)) ".")))
      (format t "  ")
      (loop for j from 0 to 2 do
        (format t " ~d" (+ i j))))
    (format t "~%~%")))
```

Bibliografía

- [Cortés–93]
  Cap. 4.7.1: "Juegos: la estrategia minimax y sus modificaciones".
- [Mira–95]
  Cap. 4.4: "Búsqueda con adversarios".
- [Rich–91]
  Cap. 12: "Los juegos".
- [Russell–97]
  Cap. 5: "Juegos".
- [Winston–94]
  Cap. 5: "Arboles y búsqueda con adversario".