Paso de Parámetros

Paso de Parámetros

- Arreglos (Arrays)
- Llamada por referencia (call-by-reference)
- Llamada por valor-resultado y llamada por resultado
- Valores denotados y valores expresados
- Llamada por nombre y llamada por necesidad
- Argumentos opcionales y *keywords*

Paso de Parámetros

- Arreglos (Arrays)
- Llamada por referencia (call-by-reference)
- Llamada por valor-resultado y llamada por resultado
- Valores denotados y valores expresados
- Llamada por nombre y llamada por necesidad
- Argumentos opcionales y *keywords*

Arreglos (Arrays)

- Agregados:
  - Estructuras de datos compuestas de varios elementos
  - Se representan en localidades consecutivas de memoria
  - Paso de parámetros: Apuntador a dirección base
  - Hay implicaciones para el diseño de los lenguajes
- En Scheme:
  - vectores
  - pares punteados
  - cadenas

Paso de parámetros:

- Representación de datos agregados ***indirecta***
  - Acceso a elementos en el tipo de datos por referencia a la dirección base
  - Asignación: cambio de apuntador al dato agregado
- Representación de datos agregados ***directa***
  - Las asignación afecta directamente a los elementos del tipo agregado

Arreglos (arrays)

- Definición de arreglos:
  - Creación de ligas
  - Acceso a elementos de los arreglos
  - Asignación de elementos a arreglos

Arreglos (arrays)

- Extensión del intérprete: sintaxis concreta
  - <form> ::= definearray <var> <exp>

    <exp> ::= letarray <arraydecls> in <exp>

    | <array-exp> [<exp>]

    | <array-exp> [<exp>] := <exp>

    <array-exp> ::= <varref> | (<exp>)

    <arraydecls> ::= <arraydecl> {;<arraydecl>}*)

    <arraydecl> ::= <var> [<exp>]

Arreglos (arrays)

- Extensión del intérprete: sintaxis abstracta
  - (define-record definearray (var) (len-exp)
  - (define-record letarray (arraydecls body))
  - (define-record arrayref (array index))
  - (define-record arrayassign (array index exp))
  - (define-record decl (var exp))
- Un arreglo es una secuencia de celdas que contienen valores expresados
  - Array = {Cell(Valor Expresado)}*
  - Valor Expresado = Número + Procedimiento + Arreglo
  - Valor denotado = Cell(Valor Expresado)

Ejemplos

```
> define p = proc (b) b[0] := 3
> letarray a[2] in begin
                    a[0] := 1;
                    a[1] := 2;
                    p(a);
                    a[0]
                end;
3
```


ADT para arrays usando vectors

```
> (define make-array
      (lambda (length)
          (let ((array (make-vector (+ length 1))))
              (vector-set! array 0 '*array*)
              array)))

> (define array?
      (lambda (x)
          (and (vector? x) (eq? (vector-ref x 0) '*array*))))
```


ADT para arrays usando vectors

```
> (define array-ref
      (lambda (array index)
          (vector-ref array (+ index 1))))

> (define array-set!
      (lambda (array index value)
          (vector-set! array (+ index 1) value)))
```


ADT para arrays usando vectors

```
> (define array-whole-set     ;copia todo un arreglo
    (lambda (dest-array source-array)
        (let ((source-len (vector-length source-array)))
            (if (> source-len (vector-length dest-array))
                (error "arreglo muy grande" source-array)
                (letrec ((loop (lambda (n)
                                (if (< n source-len)
                                    (begin
                                    (vector-set! dest-array n
                                                 (vector-ref source-array n))
                                    (loop (+ n 1)))))))
                    (loop 1))))))
```

ADT para arrays usando vectors

```
> (define array-copy
    (lambda (array)
      (let ((new-array (make-array
                         (- vector-length array) 1))))
        (array-whole-set! new-array array)
        new-array)))
```


Intérprete para arrays

```
> (define eval-exp
    (lambda (exp env)
      (variant-case exp
        (varref (var) (denoted->expressed  (apply-env env var))
        (app (rator rands)
             (apply-proc (eval-rator rator env) (eval-rands  rands env)))
        (varassign (var exp)
             (denoted-value-assign! (apply-env env var) (eval-exp exp env)))
        (letarray (arraydecls body)
             (eval-exp body
               (extend-env (map decl->var  arraydecls)
                           (map (lambda (decl)
                                  (do-letarray (eval-exp (decls->exp decl) env)
                             arraydecls)
                           env)))
        ...
```


Intérprete para arrays

```
...
    (arrayref (array index)
        (array-ref  (eval-array-exp array env) (eval-exp index env)))
    (arrayassign (array index exp)
        (array-set! (eval-array-exp array env)
                    (eval-exp index env)
                    (eval-exp exp env)))
...)
```


Intérprete (otros proc.)

```
> (define eval-rator
    (lambda (rator env)
      (eval-exp rator env)))

> (define eval-rands
    (lambda (rands env)
      (map (lambda (rand) (eval-rand rand env)) rands)))

>(define eval-rand
    (lambda (rand env)
      (expressed->denoted (eval-exp exp env))))
```


Intérprete (otros proc.)

```
> (define apply-proc
    (lambda (proc args)
      (variant-case proc
        (prim-proc (prim-op)
            (appy-prim-op prim-op
                (map denoted->expressed args)))
        (closure (formals body env)
            (eval-exp body (extend-env formal args env)))
        (else (error "Procedimiento inválido:" proc)))))
```


Los nuevos procedimientos

- La manera de definir los nuevos procedimientos:
  - denoted->expressed
  - denoted-value-assign!
  - do-letarray
  - eval-array-exp
  - expressed->denoted

  define la modalidad de paso de parámetros.

- Diferentes definiciones producen los diferentes mecanismos de paso de parámetros

Los nuevos procedimientos

- Definiciones para la representación indirecta o llamado por valor (call-by-value):
  - (define denoted->expressed cell-ref)
  - (define denoted-value-assign! cell-set!)
  - (define do-letarray (compose make-cell make-array)
  - (define eval-array-exp eval-exp)
  - (define expressed->denoted make-cell)

Intérprete para llamada-por-valor

```
> (define eval-exp
    (lambda (exp env)
      (variant-case exp
        (varref (var) (cell-ref  (apply-env env var))
        (app (rator rands)
             (apply-proc (eval-rator rator env) (eval-rands  rands env)))
        (varassign (var exp)
             (cell-set!  (apply-env env var) (eval-exp exp env)))
        (letarray (arraydecls body)
             (eval-exp body
                  (extend-env (map decl->var  arraydecls)
                              (map (lambda (decl)
                                     ((compose make-cell make-array)
                                          (eval-exp (decls->exp decl) env)
                                   arraydecls)
                              env)))
```


Intérprete para arrays

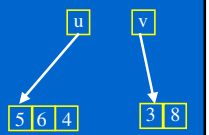
```
...
        (arrayref (array index)
                  (array-ref  (eval-exp array env) (eval-exp index env)))
        (arrayassign (array index exp)
                  (array-set! (eval-exp array env)
                              (eval-exp index env)
                              (eval-exp exp env)))
        ...)
```


Modelos directo e indirecto de paso de arreglos

```
> letarray u[3]; v[2]
  in begin
       u[0] := 5; u[1] := 6; u[2] := 4;
       v[0] := 3; v[1] := 8
       let p = proc (x)
           begin
               x[1] := 7; x := v; x[1] := 9
           end
       in p(u)
  end
```

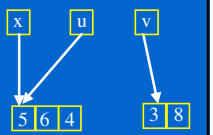

Modelo indirecto

```
> letarray u[3]; v[2]
  in begin
       u[0] := 5; u[1] := 6; u[2] := 4;
       v[0] := 3; v[1] := 8
       let p = proc (x)
           begin
               x[1] := 7; x := v; x[1] := 9
           end
       in p(u)
  end
```



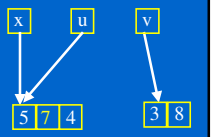
Modelo indirecto

```
> letarray u[3]; v[2]
  in begin
       u[0] := 5; u[1] := 6; u[2] := 4;
       v[0] := 3; v[1] := 8
       let p = proc (x)
           begin
               x[1] := 7; x := v; x[1] := 9
           end
       in p(u)
  end
```



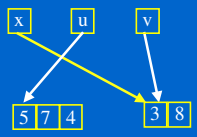
Modelo indirecto

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



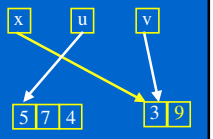
Modelo indirecto

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



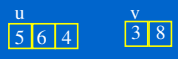
Modelo indirecto

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



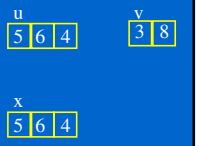
Modelo directo

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



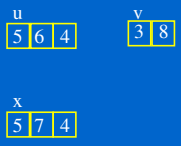
Modelo directo

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



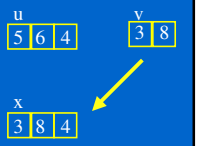
Modelo directo

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



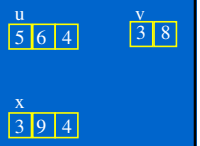
Modelo directo

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



Modelo directo

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



Definición modo directo

- Representación directa
  - Array = {Cell(Número + Procedimiento)}*
  - Valor Expresado = Número + Procedimiento + Arreglo
  - Valor denotado = Cell(Número) + Cell(procedimiento) + Arreglo

Definición modo directo

- Representación directa
  - Array = {Cell(Número + Procedimiento)}*
  - Valor Expresado = Número + Procedimiento + Arreglo
  - Valor denotado = Cell(Número) + Cell(procedimiento) + Arreglo
- Los arreglos no contienen arreglos!
- Representación indirecta
  - Array = {Cell(Valor Expresado)}*
  - Valor Expresado = Número + Procedimiento + Arreglo
  - Valor denotado = Cell(Valor Expresado)

Representación directa (llamada por valor)

```
> (define denoted->expressed
      (lambda (den-val)
          (if (array? den-val) den-val (cell-ref den-val))))

> (define denoted-value-assign!
      (lambda (den-val exp-val)
          (cond
              ((not (array? den-val)) (cell-set! den-val exp-val)
              ((array? exp-val) (array-whole-set! den-val exp-val)
              (else (error  "se debe asignar un arreglo:" den-val)))))
```


Representación directa (llamada por valor)

```
> (define do-letarray make-array)

> (define eval-array-exp eval-exp)

> (define expressed->denoted
      (lambda (exp-val)
          (if  (array? exp-val)
              (array-copy exp-val)
              (make-cell exp-val))))
```

Intérprete para arrays

```
> (define eval-exp
    (lambda (exp env)
      (variant-case exp
        (varref (var) (denoted->expressed  (apply-env env var))
        (app (rator rands)
             (apply-proc (eval-rator rator env) (eval-rands  rands env)))
        (varassign (var exp)
             (denoted-value-assign! (apply-env env var) (eval-exp exp env)))
        (letarray (arraydecls body)
             (eval-exp body
                 (extend-env (map decl->var  arraydecls)
                             (map (lambda (decl)
                                    (do-letarray (eval-exp (decls->exp decl) env)
                                  arraydecls)
                             env)))
                 ...
```


Intérprete para arrays

```
...
        (arrayref (array index)
             (array-ref  (eval-array-exp array env) (eval-exp index env)))
        (arrayassign (array index exp)
             (array-set! (eval-array-exp array env)
                         (eval-exp index env)
                         (eval-exp exp env)))
        ...)
```


Paso de Parámetros

- Arreglos (Arrays)
- Llamada por referencia (call-by-reference)
- Llamada por valor-resultado y llamada por resultado
- Valores denotados y valores expresados
- Llamada por nombre y llamada por necesidad
- Argumentos opcionales y *keywords*

Llamada por referencia

- Cuando un procedimiento hace una llamada y el parámetro formal (una variable) es cambiado por el procedimiento llamado, es dicho cambio visible en el procedimiento que hace la llamada?
- En el modo indirecto de llamada por valor, dicho cambio es visible para los arreglos, pero no para variables de otro tipo.
- En la llamada por valor, los cambios **NO** son visibles
- La llamada por referencia (call-by-reference) habilita dicha visibilidad

Llamada por referencia

```
> let p = proc (x) x := 5
  in let a = 3;
         b = 4
     in begin
          p(a);
          p(b);
          +(a, b)
        end
```

- Llamada por valor ⇒ 7 (el argumento se guarda en una localidad nueva)
- Llamada por referencia ⇒ 10

Llamada por referencia

- Se necesita la llamada por referencia?
- Hay procedimientos que no pueden escribirse con una llamada por valor?

```
> define swap = proc (x, y)
                  let temp  = 0;
                  in begin temp := x; x := y; y:= temp end;
> define  c = 3
> let b = 4 in begin swap(c,  b); b end;
3
> c
4
```

Llamada por referencia

- Llamada por referencia: arreglos y ligas a variables

```
> define b = 2;
> letarray a[3]
   in begin
            a[1] := 5;
            swap(a[1], b);
            a[1]
      end;
2
> b;
5
```


Llamada por referencia

- Llamada por referencia: los operandos pueden ser expresiones
- Se asigna una localidad de memoria al valor del operando
- Dicho valor no es accesible por el proc. que hace la llamda

```
> define b = 3;
> define p = proc (x) x := 5
>  begin
            p(add1(b));
            b
   end;
3
```

- Las llamadas por valor y por referencia tienen el mismo efecto

Llamada por referencia

- Ligas "alias" (aliasing): más de un parámetro en una llamada por referencia puede referirse a la misma localidad

```
> let b = 2;
      p = proc (x, y)
            begin
                  x:= 4; y
            end
   in p(b, b)
4
```

- Las llamadas tienen efectos laterales "perniciosos"
- x & y se refieren a la celda que contiene a *b*

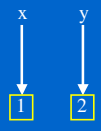
Llamada por referencia

- Si se pasan referencias a arreglos como argumentos es muy costoso detectar "aliasing" (i.e. Si swap(a[1], a[f(b)]) cómo detectar que f(b) = 1?
- *Aliasing* puede darnos sorpresas muy desagradables:

```
> define swap2 = proc (x, y)
                  begin
                        x := +(x, y);
                        y := -(x, y);
                        x := -(x, y)
                  end;
```

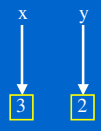

Llamada por referencia

```
> define b = 1
> define c = 2
> swap2(b, c)
            begin
                  x := +(x, y);
                  y := -(x, y);
                  x := -(x, y)
            end;
```



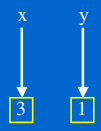
Llamada por referencia

```
> define b = 1
> define c = 2
> swap2(b, c)
            begin
                  x := +(x, y);
                  y := -(x, y);
                  x := -(x, y)
            end;
```



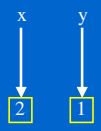
Llamada por referencia

```
> define b = 1
> define c = 2
> swap2(b, c)
        begin
                x := +(x, y);
                y := -(x, y);
                x := -(x, y)
        end;
```



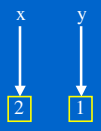
Llamada por referencia

```
> define b = 1
> define c = 2
> swap2(b, c)
        begin
                x := +(x, y);
                y := -(x, y);
                x := -(x, y)
        end;
```



Llamada por referencia

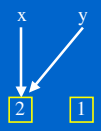
```
> define b = 1
> define c = 2
> swap2(b, c)
> b;
2
> c;
> 1
```



Llamada por referencia

- Pero...

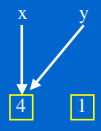
```
> swap2(b, b)
        begin
                x := +(x, y);
                y := -(x, y);
                x := -(x, y)
        end;
```



Llamada por referencia

- Pero...

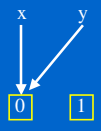
```
> swap2(b, b)
        begin
                x := +(x, y);
                y := -(x, y);
                x := -(x, y)
        end;
```



Llamada por referencia

- Pero...

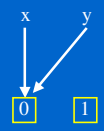
```
> swap2(b, b)
        begin
                x := +(x, y);
                y := -(x, y);
                x := -(x, y)
        end;
```



Llamada por referencia

- Pero...

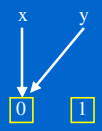
```
> swap2(b, b)
        begin
                x := +(x, y);
                y := -(x, y);
                x := -(x, y)
        end;
```



Llamada por referencia

- Pero...

```
> swap2(b, b)
        begin
                x := +(x, y);
                y := -(x, y);
                x := -(x, y)
        end;
> b;
0
```

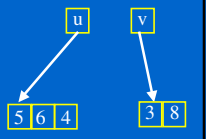


Cómo modelar la llamada por referencia?

- En todos los casos los valores denotados por el procedimiento que hace la llamada son los valores donotados del procedimiento llamado!
- Por lo tanto, si el operando es una variable se puede pasar su liga directamente al procedimiento llamado, en vez de copiar el contenido a una nueva celda.

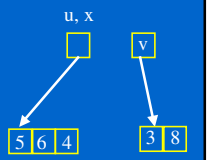
Llamada por referencia (modo indirecto)

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



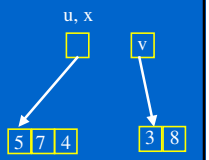
Llamada por referencia (modo indirecto)

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



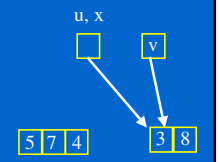
Llamada por referencia (modo indirecto)

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



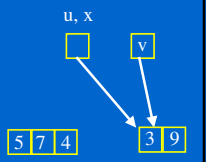
Llamada por referencia (modo indirecto)

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



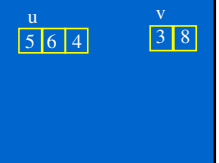
Llamada por referencia (modo indirecto)

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



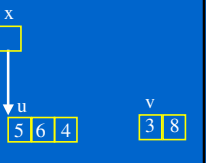
Llamada por referencia (modo directo)

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



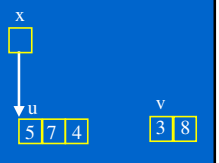
Llamada por referencia (modo directo)

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



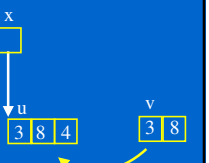
Llamada por referencia (modo directo)

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



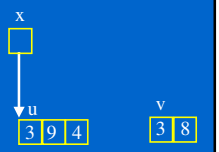
Llamada por referencia (modo directo)

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



Llamada por referencia (modo directo)

```
> letarray u[3]; v[2]
  in begin
        u[0] := 5; u[1] := 6; u[2] := 4;
        v[0] := 3; v[1] := 8
        let p = proc (x)
            begin
                x[1] := 7; x := v; x[1] := 9
            end
        in p(u)
  end
```



Intérprete llamada por valor versus referencia

- Cambiar:

```
<exp> ::= <integer-literal>
        | <varref>
        | (<operatos><operands>)
  <operatos> ::= <varref>  | (<exp>)
  <operand> ::= ( ) | (<operand> {,<operand>}*)
  <operand> ::= <exp>
    <varref> ::= <var>
```

- a:

```
<operand> ::= <varref>
```


Intérprete llamada por valor versus referencia

- También cambiar:

```
> (define eval-rand
      (lambda (rand env)
        (expressed->denoted (eval-exp exp env))))
```

a:

```
> (define eval-rand
      (lambda (rand env)
        (variant-case rand
          (varref (var) (apply-env env var))
          (else (error "op. inválida:" rand)))))
```


Paso de arreglos por referencia

- Hay que pasar una referencia al arreglo
- En Scheme no podemos pasar apuntadores directamente, por lo que definimos array-elements como sigue:

```
(define-record ae (array index))
```

- Definimos L-value como sigue:

```
L-value = Cell(Expressed-value) +
          Array Element (Expressed Value)
Denoted value = L-value
```


Intérprete para llamada por ref. (arrays)

- Con esto hay dos tipos de operandos posibles en una llamada por referencia: variable y arreglos
- Generalizando, se puede permitir pasar cualquier tipo de objeto por referencia, copiando sus valores a celdas nuevas, generadas con dicho propósito.
- Sintaxis concreta

```
<operand> ::= <varref>
            | <array-exp> [<exp>]      array ref (array index)
            | <exp>
```


Definición del intérprete (a partir del intérprete para llamada por valor)

```
> (define eval-rand
      (lambda (rand env)
        (variant-case rand
          (varref (var) (apply-env en var)
          (arrayref (array index)
              (make-ae (eval-array-exp array env)
                       (eval-exp index env)))
          (else (make-cell (eval-exp rand env)))))
```

Definición del intérprete

- Como el conjunto de valores denotados ha cambiado, es necesario reflejar los cambios en *denoted->expressed*

```
> (define denoted->expressed
    (lambda (den-val)
      (cond
        ((cell? den-val) (cell-ref den-val))
        ((ae? den-val) (array-ref (ae->array den-val)
                                  (ae->index den-val)))
        (else (error "no se pueden extraer los valores:" den-val)))))
```


Definición del intérprete

- Como el conjunto de valores denotados ha cambiado, es necesario reflejar los cambios en *denote-value-assign!*

```
> (define denoted-value-assign!
    (lambda (den-val val)
      (cond
        ((cell? den-val) (cell-set! den-val val))
        ((ae? den-val) (array-set! (ae->array den-val)
                                   (ae->index den-val) val))
        (else (error "no se puede asignar :" den-val)))))
```