# ESTRUCTURAS DE DATOS TEORÍA 2016/2017

## PILAS

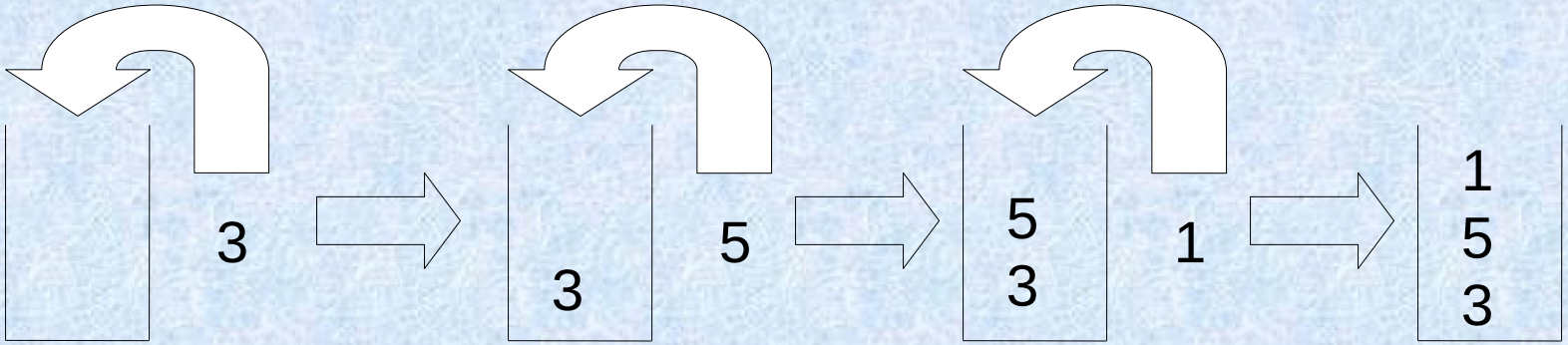
# PILAS

Una pila P es una estructura lineal tal que las inserciones, las consultas y las eliminaciones solo se permiten en un único punto.

- La pila puede no tener nada, situación que se denomina *pila vacía*.

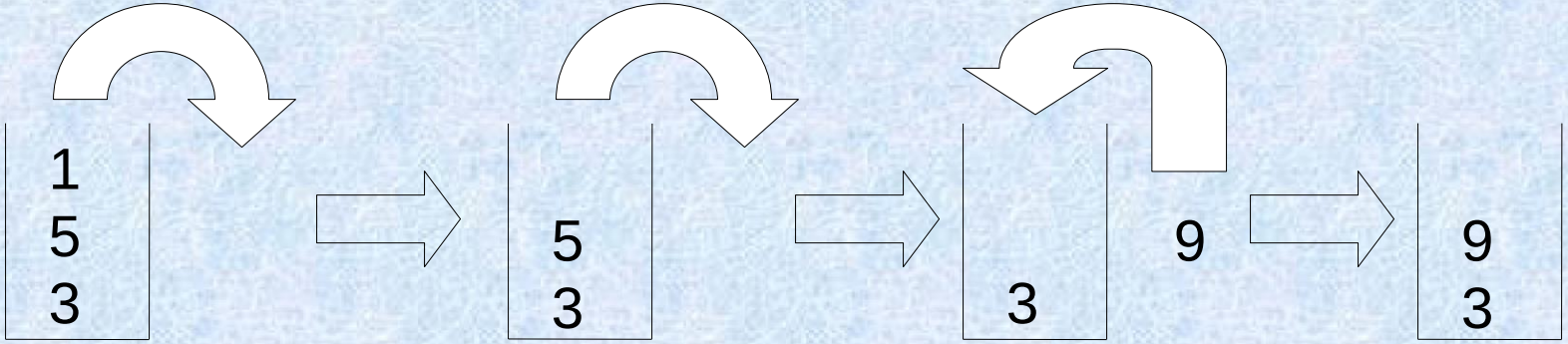
Las pilas son estructuras denominadas LIFO (Last In, First Out), nombre que hace referencia al modo en que se accede a los elementos.

**Ejemplo:** Poner los datos 3, 5 y 1 en una pila vacía.

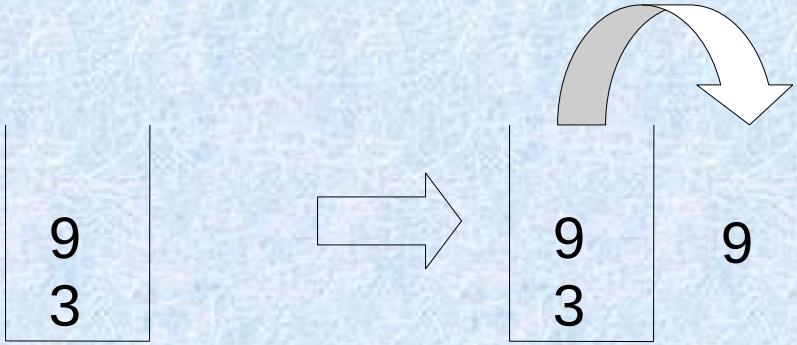


# PILAS

**Ejemplo:** Quitar dos datos de la pila y poner un 9.




**Ejemplo:** Comprobar qué hay en la pila.



# PILAS EN COMPUTACIÓN

Las pilas se utilizan, por ejemplo, en la implementación de la recursión…

- Los entornos locales se resuelven en orden inverso al que se crean.
  - Factorial(3) = 3 * Factorial(2) = 3 * (2 * Factorial(1) ) …o

    en la evaluación de expresiones matemáticas.
- Notación Polaca Inversa (o RPN, Reverse Polish Notation)
  - 5 + ( (1 + 2) * 4) se escribe como 5 1 2 + 4 * +

# ESPECIFICACIÓN: PILAS

*{Como no sabemos de qué va a ser la pila, ponemos una especificación genérica y usamos un parámetro formal}*

**espec** *PILA[ELEMENTO]*

**usa** *BOOLEANOS*

**parametro formal**

**generos** *elemento*

**fparametro**

**generos** *pila*

# ESPECIFICACIÓN: PILAS (2)

**operaciones**

*{crear una pila vacía}*
*pvacía: → pila*

*{poner un elemento en la pila}*
*apilar: elemento pila → pila*

Generadoras

*{quitar un elemento de la pila}*
**parcial** *desapilar: pila → pila*

Modificadoras

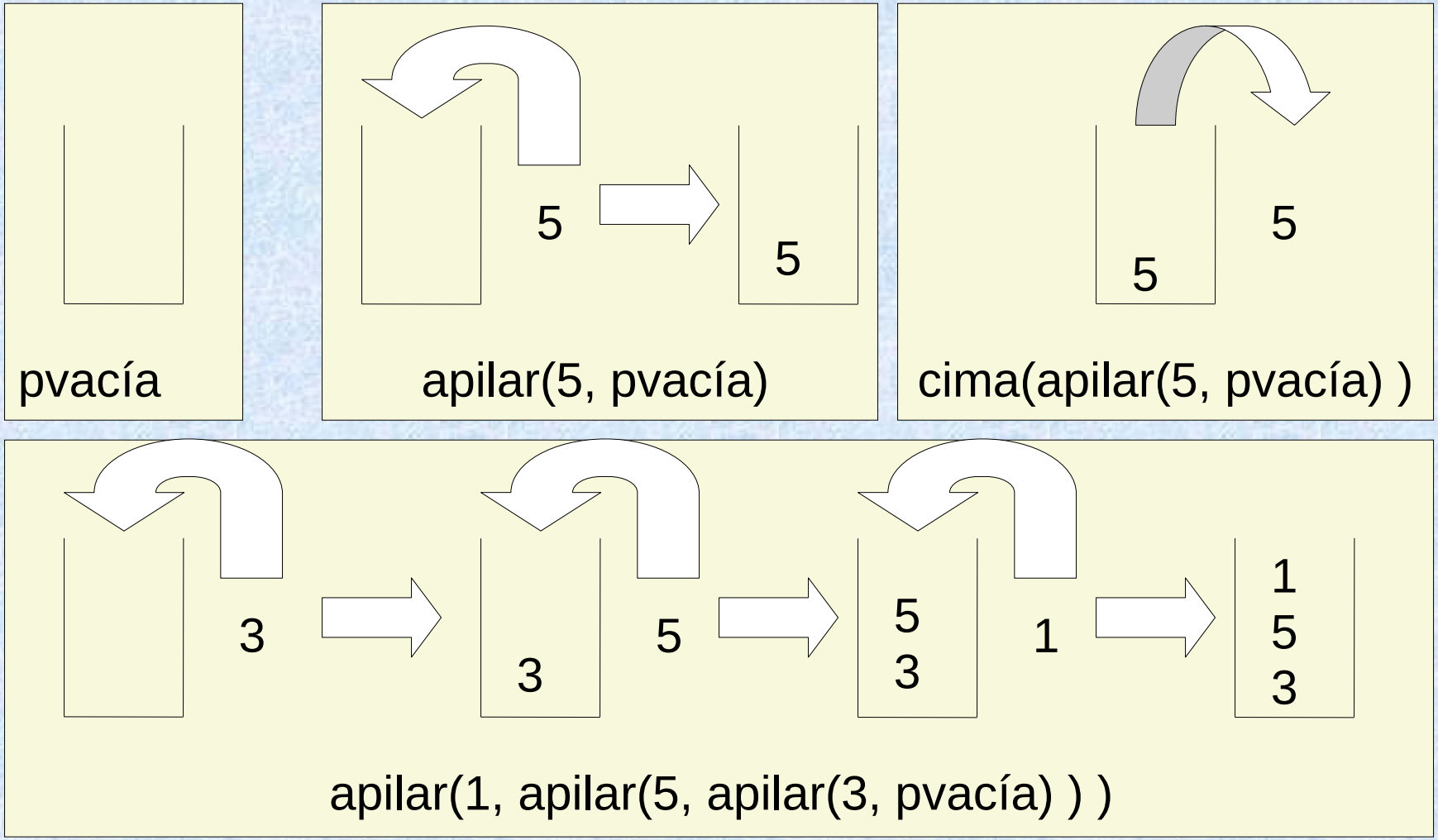
*{observar la cima de la pila}*
**parcial** *cima: pila → elemento*

*{para ver si la pila está vacía}*
*vacía?: pila → bool*

Observadoras

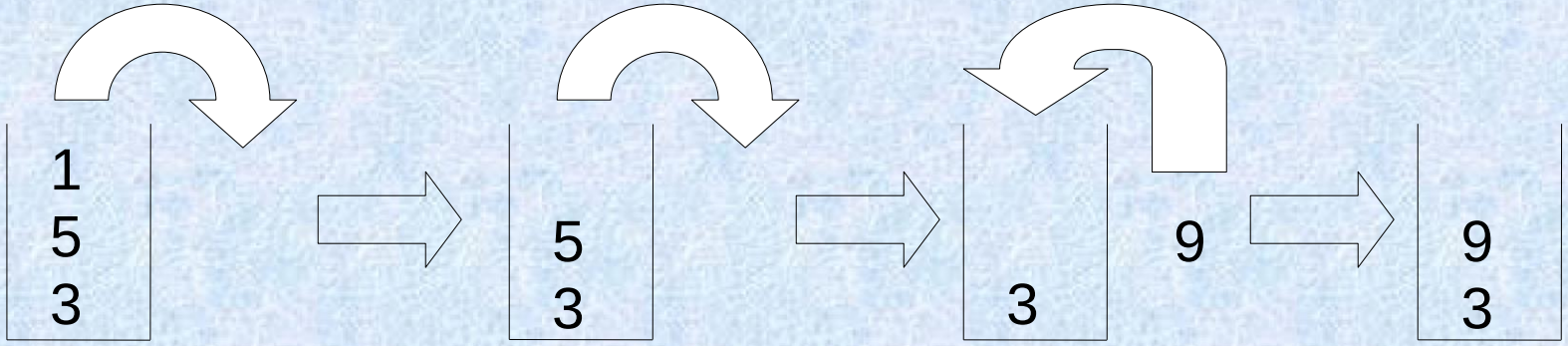
# REPRESENTACIÓN DE LAS PILAS (1)

¿Qué podemos construir con números y estas operaciones?



# REPRESENTACIÓN DE LAS PILAS (2)

¿Qué podemos construir con números y estas operaciones?




apilar(9, desapilar(desapilar(apilar(1, apilar(5, apilar(3, pvacía))))))

esta es la pila de la página anterior

# ESPECIFICACIÓN: PILAS (3)

**var** *p: pila; x: elemento*

*{Como hay operaciones parciales hay que definir cuándo pueden usarse, es decir, sobre qué datos se aplican}*

*{Primera forma: utilizando las generadoras del tipo}*

**ecuaciones de definitud**

***Def( desapilar(apilar(x,p)) )***

***Def( cima(apilar(x,p)) )***

*{Segunda forma: utilizando propiedades de los datos}*

**ecuaciones de definitud**

***vacía?(p) = F ⇒ Def( desapilar(p) )***

***vacía?(p) = F ⇒ Def( cima(p) )***

# ESPECIFICACIÓN: PILAS (4)

***{Ahora que ya sabemos cuándo puede usarse una operación vamos a ver cómo se usa. Para ello ponemos los datos como si se hubiesen obtenido mediante las generadoras (cuando sea posible)}***

**ecuaciones**

```
desapilar( apilar(x,p) ) = p

cima( apilar(x,p) ) = x

vacía?( pvacía ) = T

vacía?( apilar(x,p) ) = F
```

**fespec**

# EJEMPLO 1

Ejemplo: Contar cuántos elementos tiene una pila.

- Es una operación observadora (devuelve un natural)

  ***contar*: *pila* → *natural***

- Las ecuaciones pueden ser

  ***contar( pvacía ) = 0***

  ***contar( apilar(x, p) ) = suc( contar( p ))***

- Otra opción para las ecuaciones

  ***vacía?(p) = T ⇒ contar(p) = 0***

  ***vacía?(p) = F ⇒***

  ***contar(p) = suc( contar(desapilar(p)) )***

***IMPORTANTE: ¡LA PILA SE VACÍA AL RECORRERLA!***

# EJEMPLO1. PSEUDOCÓDIGO

Ejemplo: Contar cuántos elementos tiene una pila.

```
func contar (p:pila) dev n:natural                    {recursiva}
    si vacia?(p) entonces Devolver 0
                 si no devolver 1+ contar(desapilar(p))
finsi
finfunc
```

# EJEMPLO1. PSEUDOCÓDIGO (2)

```
func contar (p:pila)dev n:natural                    {Iterativa}
    var cuantos:natural
    n=0
    mientras ¡vacia?(p) hacer
        desapilar(p)
        n←n+1
    finmientras
finfunc
```

## EJEMPLO 2

Ejemplo: Obtener la suma de los datos de una pila de enteros, considerando la pila vacía como valor 0.

- Es una operación observadora (devuelve un entero)

  $suma: pila \rightarrow entero$

- Usando generadores, las ecuaciones pueden quedar

  $suma(\ pvacía\ ) = 0$

  $suma(\ apilar(x,\ p)\ ) = x + suma(\ p\ )$

- Usando propiedades, las ecuaciones serían

  $vacía?(p) = T \Rightarrow suma(p) = 0$

  $vacía?(p) = F \Rightarrow$

  $suma(p) = cima(p) + suma(desapilar(p))$

# EJEMPLO 2. PSEUDOCÓDIGO

Ejemplo: Obtener la suma de los datos de una pila de enteros, considerando la pila vacía como valor 0.

**func** suma (p:pila) **dev** entero

**si** vacia?(p) **entonces Devolver** 0

**sino devolver** cima(p)+suma(desapilar(p))

**finsi**

**fin func**

***IMPORTANTE: Hay que tener en cuenta que si escribiésemos:*** `sino Devolver suma (desapilar (p)+ cima(p))`

***el resultado podría ser incorrecto (depende la implementación)***

# EJEMPLO 2. PSEUDOCÓDIGO (2)

Ejemplo: Obtener la suma de los datos de una pila de enteros, considerando la pila vacía como valor 0.

```
func suma (p:pila) dev sum:entero
var sum:entero
    sum=0
    mientras ¡vacia?(p) hacer
    sum←sum+cima(p)
    desapilar(p)
    finmientras
finfunc
```

## EJEMPLO 3

Ejemplo: Obtener la inversa de una pila, es decir, la pila resultante al cambiar el orden de los datos.

- Vamos a ir poniendo los datos de una pila en otra auxiliar hasta que no quede ninguno en la primera, y entonces se devuelve la pila auxiliar.

```
invertir_aux: pila pila → pila
invertir_aux(pvacía, p2) = p2
invertir_aux(apilar(x,p1), p2) =
                    invertir_aux(p1, apilar(x,p2))
```

- La operación que invierte una pila usa ***invertir_aux*** usando una pila vacía como pila auxiliar.

```
invertir: pila → pila
invertir(p) = invertir_aux(p, pvacía)
```

# EJEMPLO 3. PSEUDOCÓDIGO

Ejemplo: Obtener la inversa de una pila, es decir, la pila resultante al cambiar el orden de los datos.

```
fun invertir(p:pila) dev q:pila
var e: elemento
q ← pila_vacía()
   Mientras ¡(es_pila_vacía(p)) Hacer
      e ← cima(p)
      apilar(e,q)
      desapilar(p)
   fmientras
ffun
```

IMPORTANTE: ¡La pila de entrada `p` se queda vacía, puede ser un problema si no es una copia local sino un enlace directo a memoria!

# EJEMPLO 3. PSEUDOCÓDIGO

**func** invertir_aux (p1, p2: pila) **dev** pila {recursiva}

**si** vacia?(p1) **entonces devolver** p2

**si no**

e←cima(p1)

**devolver** invertir_aux(desapilar(p1), apilar (e, p2))

**finsi**

**finfunc**

***Como no sabemos la implementación de "desapilar" guardamos la cima en una variable por si se modifica la variable p1***

**func** invertir(p:pila) **dev** pila

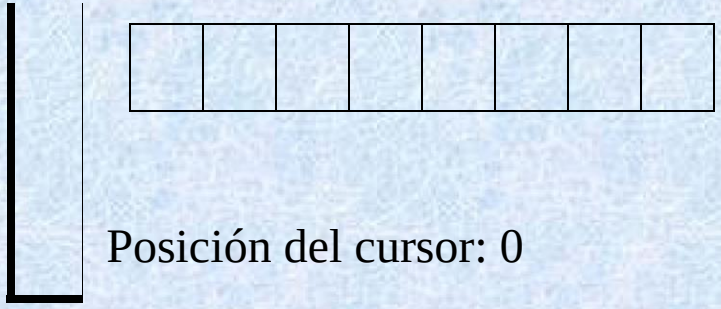
Invertir_aux (p, pvacia)

**finfunc**

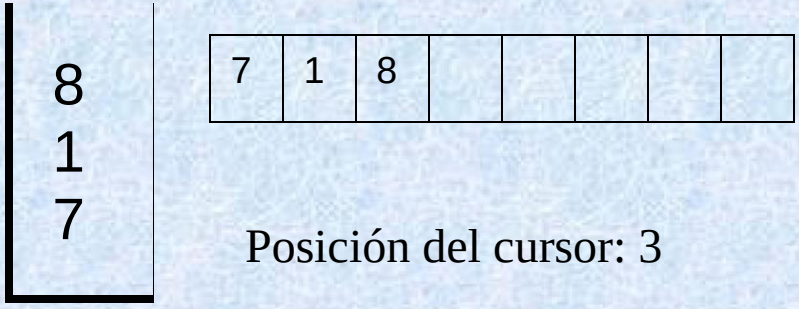
# IMPLEMENTACIÓN DE PILAS

Una pila puede implementarse mediante un vector (o array), aprovechando que las inserciones y borrados se hacen en un único punto de la estructura. Al usar memoria estática, la pila tendrá una capacidad máxima (es necesario crear una operación *está_llena?: pila* → *bool*).

La parte inferior de la pila se corresponde con la primera posición del array, y se utiliza un índice (denominado *cursor*) para acceder a la parte superior de la pila. Así, la pila crece hasta alcanzar el tamaño definido del array.



Posición del cursor: 0




Posición del cursor: 3

# IMPLEMENTACIÓN DE PILAS

La implementación más habitual es la de **celdas enlazadas**:

- La pila se representa mediante un puntero a una celda
  - Si la pila está vacía, el puntero es “NULL”.
  - Si no, la celda contiene el elemento que se encuentra en la cima de la pila, y un puntero a la celda que contiene lo que está *debajo* de la cima (que a su vez es una pila).

IMPORTANTE: ¡Solo se accede a la cima de la pila!
aunque a la hora de la implementación se pueda “recorrer” la pila.

# PILAS. TIPOS

```
tipos
enlace-pila = puntero a nodo-pila
nodo-pila = reg     {un nodo debe tener, como poco,…}
        valor: elemento     {puede ser otra estructura}
        sig: enlace-pila
 freg
pila = enlace-pila                  {acceso a la pila}
ftipos
```

# PILAS. CONSTRUCTORAS

*{ Crear una pila vacía **pvacía** y poner un elemento **apilar** }*

```
fun pila_vacía() dev p:pila


proc apilar(E e:elemento, p:pila)
```

# PILAS. CONSTRUCTORAS

*{ Crear una pila vacía **pvacía** y poner un elemento **apilar** }*

```
fun pila_vacía() dev p:pila
    p ← null
ffun

proc apilar(E e:elemento, p:pila)
var q: enlace-pila
reservar(q)
q^.valor ← e
q^.sig ← p
p ← q
fproc
```

IMPORTANTE: Da igual que “p” sea de E o E/S: es un puntero, cualquier cambio afecta al exterior del procedimiento.

# PILAS. OBSERVADORAS

*{ Ver si una pila está vacía **vacía?** y obtener la cima **cima** }*

```
fun es_pila_vacía(p:pila) dev b:bool

fun cima(p:pila) dev e:elemento
```

# PILAS. OBSERVADORAS

*{ Ver si una pila está vacía **vacía?** y obtener la cima **cima** }*

```
fun es_pila_vacía(p:pila) dev b:bool
   b ← (p = null)
ffun

fun cima(p:pila) dev e:elemento
si es_pila_vacía(p) entonces
   error(Pila vacía)
si no
   e ← p^.valor
fsi
ffun
```

# PILAS. MODIFICADORAS

*{ Quitar la cima de una pila **desapilar** }*

```
proc desapilar(p:pila)
```

# PILAS. MODIFICADORAS

*{ Quitar la cima de una pila **desapilar** }*

```
proc desapilar(p:pila)
var q: enlace-pila
   si es_pila_vacía(p) entonces
      error(Pila vacía)
   si no
      q ← p
      p ← p^.sig
      q^.sig ← null   {por seguridad}
      liberar(q)
   fsi
fproc
```

# EJEMPLO: EQUILIBRADO DE SÍMBOLOS

Objetivo: Comprobar si en una expresión los símbolos abiertos de llaves, corchetes y paréntesis se corresponden con los cerrados (no pueden mezclarse símbolos de distinto tipo al cerrar).

{ [ ] ( { } ) }  sí se corresponden.

[ ( { ) } ]  no se corresponden.

Idea: Utilizar una pila para introducir símbolos abiertos. Cuando se recibe un símbolo cerrado, se compara con el que está en la cima de la pila (si es que hay alguno), si son de distinto tipo es que no es correcto, y si son del mismo tipo se elimina y se sigue leyendo.

# EJEMPLO: EQUILIBRADO DE SÍMBOLOS

```
fun expresión_correcta() dev bool
var p: pila; c,d: carácter
p ← pila_vacía()
mientras queden_símbolos hacer
    leer(c)
    si (c es símbolo_abierto) entonces apilar(c,p) fsi
    si (c es símbolo_cerrado) entonces
        si (es_pila_vacía(p)) entonces Devolver Falso
        si no
            d ←cima(p)
            si (c y d se corresponden) entonces desapilar(p)
                                       si no Devolver Falso
            fsi
        fsi
    fsi
 fmientras
 Devolver es_pila_vacía(p) ffun
```