Lección 4: Entornos de Ejecución

1. Introducción

2. Asignación estática

3. Asignación en pila
   a. Bloques de activación
   b. Variables locales
   c. Variables no locales
   d. Invocación de procedimientos

4. Asignación dinámica
   a. Administración del *heap*
   b. Asignación de memoria
   c. Desasignación de memoria
   d. Invocación de procedimientos

5. Disposición del programa en memoria

6. Perspectiva

7. Máquina P

Lecturas:

Cooper, caps. 5, 6, 7
Scott, caps. 3, 5
Munchnick, cap. 5
Aho, cap. 7
Fischer, cap. 9
Holub, cap. 6 (6.2)
Bennett, cap. 2

1. Introducción

- El ***modelo de ejecución*** de un lenguaje de programación en una máquina, describe la administración de memoria relacionada con los objetos del programa durante su ejecución.
  - asignación
  - recuperación
  - direccionado

Tienen influencia:

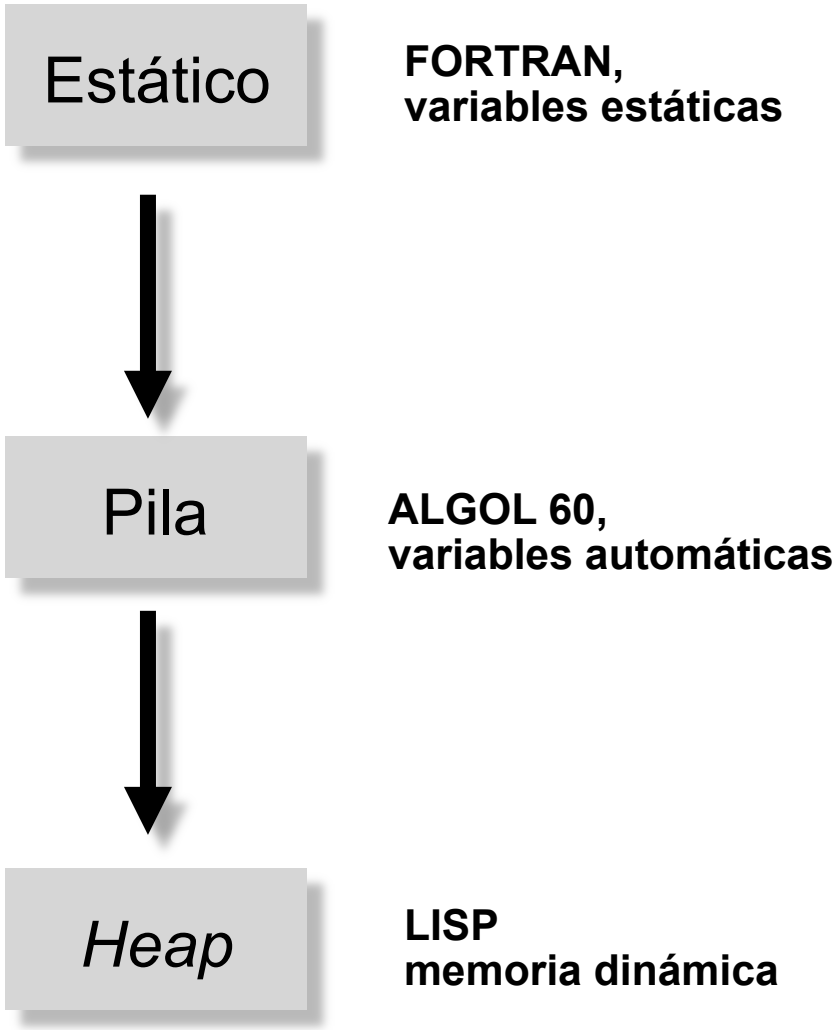
1. **La semántica del lenguaje**
2. **La arquitectura destino**
3. **El sistema operativo**

- ¿Cómo se ***asigna*** espacio para las variables declaradas en procedimientos, funciones, y en otras unidades del programa?
- ¿Este espacio puede asignarse en ***compilación***, o debe hacerse a medida que el programa se ***ejecuta***?
- ¿Cómo se ***direccionan*** las variables locales a una unidad, y cómo se direccionan variables visibles de otras?
- Cuando el espacio asignado para las variables de una unidad ya no se necesita, ¿cómo puede ***reutilizarse***?

Influencia de la semántica del lenguaje

- Algunos lenguajes (FORTRAN) han sido diseñados de tal forma que todos los requerimientos de almacenamiento de un programa puedan ser **determinados en compilación** (una sola instancia de c/variable).
- PASCAL, C, ADA, requieren una pila para objetos declarados dentro de procedimientos (recursividad, **varias instancias** de c/variable).
- Algunos lenguajes permiten un **cierto control** sobre la cantidad de memoria utilizada. El programador es responsable de su control.
- Si el lenguaje de programación permite la **concurrencia**, como ADA, el modelo debe permitir a procesos concurrentes compartir memoria común.

- Modelo de ejecución



Influencia de...

... la arquitectura destino

- Tienen influencia en la facilidad de implementación del modelo.
  - **Modos de direccionamiento** concebidos como soporte para la implementación de lenguajes de alto nivel.
  - **Registros dedicados** para implementar con facilidad las estructuras de administración de la memoria (p.e. SP).
- Dos filosofías de diseño:
  - **CISC:** instrucciones de alto nivel semejantes a instrucciones disponibles en lenguajes de programación modernos.
  - **RISC:** número reducido de instrucciones simples que permiten una implementación rápida en el hardware. Relegate Important Stuff to Compilers!

... el sistema operativo

- La ***asignación dinámica*** de memoria suele ser manejada por el sistema operativo, proporcionando soporte a los requerimientos de memoria dinámica de la mayoría de los lenguajes de programación (**`malloc`** en C, **`new`** en PASCAL, etc.)
- Puede ofrecer un modelo que incluye una ***pila***, con el propósito de ser utilizada como pila de ejecución.

Semántica del lenguaje

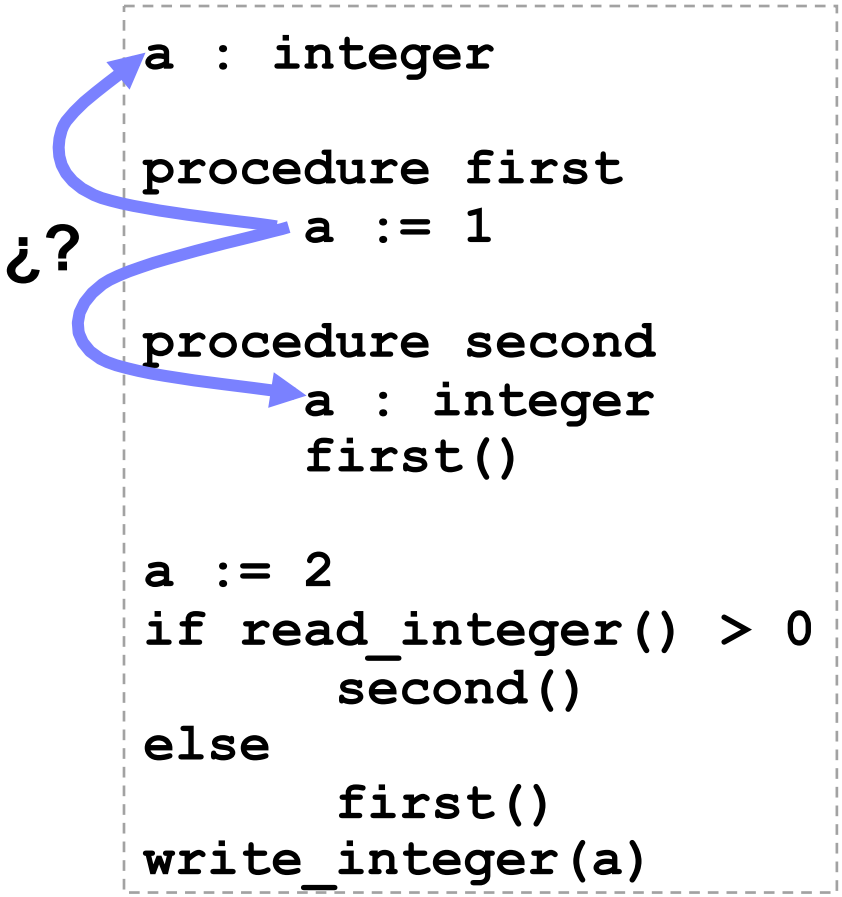
- **Ámbito *(scope):*** región textual del programa en la que un objeto es visible.

- **Tiempo de vida *(lifetime, extent):*** período de tiempo en el que la instancia del objeto con-serva su valor.

  - **Globales:** el programa es el ámbito y tiempo de vida
  - **Locales:** el proc/función es el ámbito y tiempo de vida.
  - **Estáticas:** el proc/función es el ámbito y el programa el tiempo de vida.
  - **Dinámicas:** el ámbito es el del puntero y el tiempo de vida controlable.

```
int i;
int *p;

void f()
{
   int j;
   static int k;
   int *q;

   q = malloc(sizeof(i));
   ...
   p = q;
   ...
}

main()
{
   i = *p;
   ...

   free(p);
   ...
}
```

Ámbito estático vs. dinámico



- **Ámbito estático:** un proc. tiene acceso a los bloques en los que está **declarado**. Puede determinarse durante la compilación (Ada, C, Fortran, Pascal).

```
entrada: 0, salida: 1
entrada: 1, salida: 1
```

- **Ámbito dinámico:** un proc. tiene acceso a los bloques desde los que ha sido **invocado**. Sólo puede determinarse en el momento de la ejecución (APL, Snobol, Perl).

```
entrada: 0, salida:
entrada: 1, salida:
```

- Conceptualmente más complejo.

2. Asignación Estática

- ***Datos estáticos:*** solo puede aparecer ***una*** instancia durante la ejecución de un programa.
  - FORTRAN: Todos (excepto los parámetros).
  - ADA: Unidades de biblioteca.
  - C: constantes, variables globales y estáticas.

- En compilación se puede determinar la cantidad de espacio requerido por el programa, y así a cada dato se le puede asignar una posición fija en memoria.
- Es consecuencia de la imposibilidad de hacer ***invocaciones recursivas*** a procedimientos y de la ausencia de punteros.

- Caso relativamente trivial (se utilizan direcciones contiguas), aunque puede desperdiciarse espacio.

```
struct {                    struct {
   char c1;                    short s;
   short s;                    char c1;
   char c2;                    char c2;
   } r;                        } r;
main ()
{
  printf("r.c1: %d, "
         "r.s %d, "
         "r %d\n",
         sizeof(r.c1),
         sizeof(r.s),
         sizeof(r));
}
```

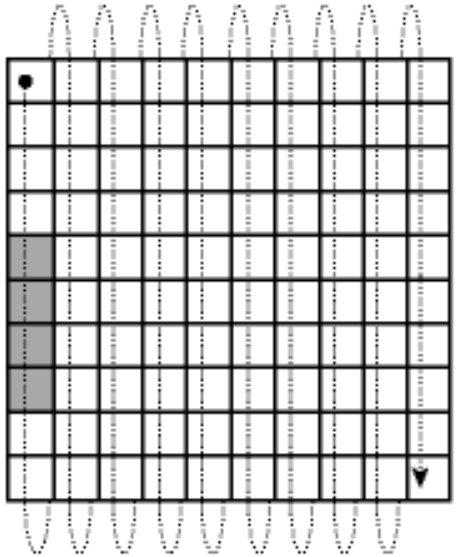
¿?

¿qué escribe en pantalla?

- Algunas arquitecturas requieren que objetos de tamaño > 1 byte se coloquen en una dirección que sea múltiplo de una potencia de 2.
- **Motorola 68000:** objetos > 1 byte a direcciones pares.

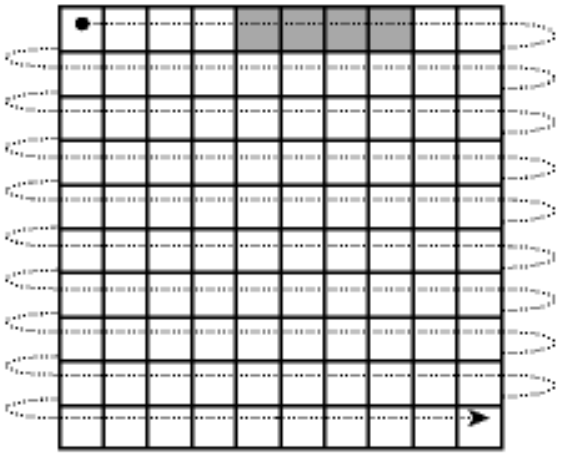
Matrices

1. Contigua por **columnas** (FORTRAN): el primer índice cambia más rápidamente



`A[2,4]` es seguido por `A[3,4]`

2. Contigua por **filas** (PASCAL, C, ADA): el segundo índice cambia más rápidamente



`A[2,4]` es seguido por `A[2,5]`

Por filas:

```
var v : array[l1..u1, l2..u2] of integer;
```

es equivalente a

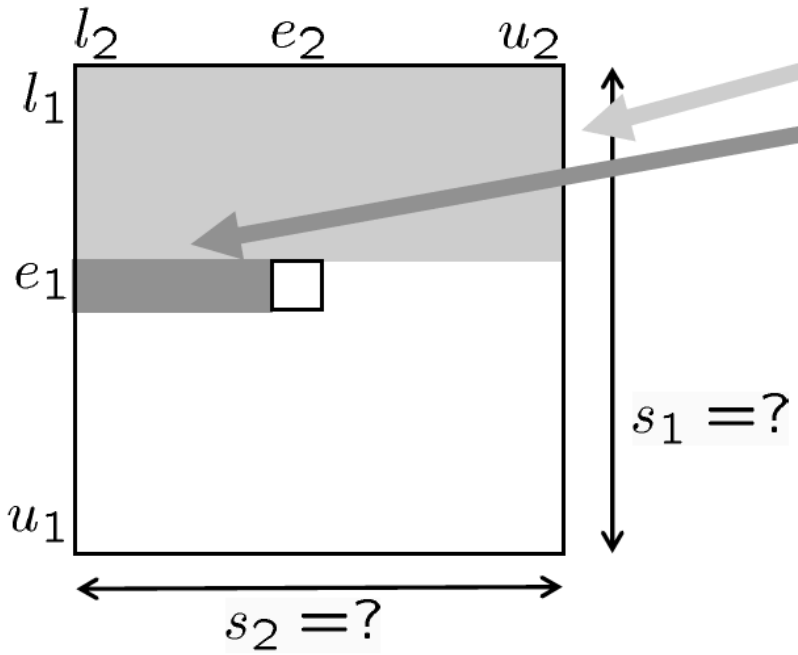
```
var v : array[l1..u1] of
        array[l2..u2] of integer;
```

Matrices contiguas por filas

- Dada la siguiente declaración:

$$\texttt{tipo v}[l_1..u_1, l_2..u_2];$$

- ¿cuál es la posición de la componente **v[e_1 , e_2]**?



- Espacio total ocupado:

- Componentes antes de la componente **v[e_1 , e_2]**:

$$c_e = [(e_1 - l_1) \cdot s_2 + \quad]$$

¿s_1?

- Si la dirección de la primera componente es **d**, la dirección de la componente **v[e_1 , e_2]** será:

$$\texttt{dir} = \texttt{d} + t_{\texttt{tipo}} \cdot c_e$$

Matrices contiguas por filas

- Generalizando a n dimensiones:

$$\texttt{tipo v}[l_1..u_1,\ldots,l_n..u_n];$$

- La dirección del elemento v[e_1,...,e_n] será:

$$\texttt{dir} = \texttt{d} + t_{\texttt{tipo}} \cdot c_e$$

$$c_e = \sum_{i=1}^{n} (e_i - l_i) m_i$$

$$m_i = \prod_{j=i+1}^{n} s_j$$

$$s_j = (u_j - l_j + 1)$$

- Dos cuestiones importantes para el acceso eficiente a componentes de matrices:

$$c_e = \sum_{i=1}^{n} (e_i - l_i) m_i$$

$$= \sum_{i=1}^{n} (e_i m_i - l_i m_i)$$

$$= \sum_{i=1}^{n} e_i m_i - \sum_{i=1}^{n} l_i m_i$$

constante

¿en C?

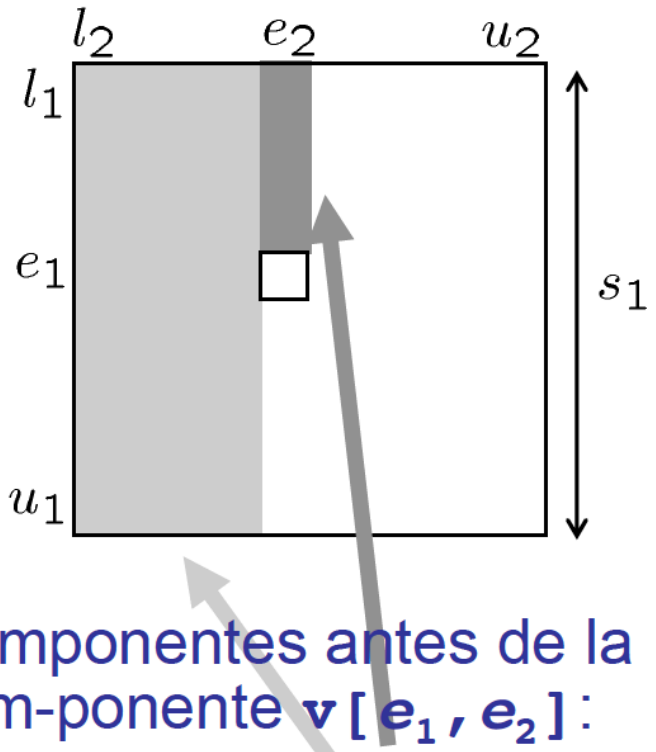
$$m_i = s_{i+1} m_{i+1}$$

recursivo

Matrices contiguas por columnas

- Dada la declaración:

$$\texttt{tipo v}[l_1..u_1, l_2..u_2];$$



- Componentes antes de la com-ponente **v[e_1, e_2]**:

$$c_e = [(e_1 - l_1) + (e_2 - l_2) \cdot s_1]$$

- Dada la declaración:

$$\texttt{tipo v}[l_1..u_1, \ldots, l_n..u_n];$$

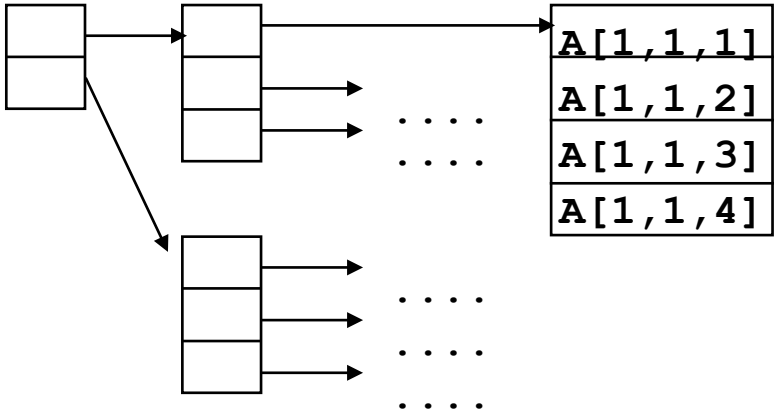
- La dirección del elemento **v[e_1, . . . , e_n]** será:

¿?

Matrices

3. Vectores de **punteros**:

```
var A: array[1..2,
             1..3,
             1..4] of integer;
```



- Los vectores no tienen que ser contiguos, ni siquiera ser residentes en memoria (útil para matrices grandes o en máquinas con segmentos de memoria pequeños).
- No se hacen multiplicaciones (útil si esta operación es costosa en la máquina).

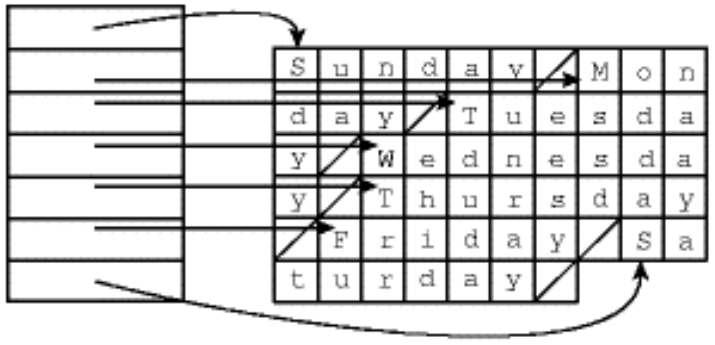
```
char days[][10] = {
     "Sunday", "Monday", "Tuesday",
     "Wednesday", "Thursday",
     "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

contigua por filas

S	u	n	d	a	y	/			
M	o	n	d	a	y	/			
T	u	e	s	d	a	y	/		
W	e	d	n	e	s	d	a	y	/
T	h	u	r	s	d	a	y	/	
F	r	i	d	a	y	/			
S	a	t	u	r	d	a	y	/	

```
char *days[] = {
     "Sunday", "Monday", "Tuesday",
     "Wednesday", "Thursday",
     "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

vector de punteros



Conjuntos

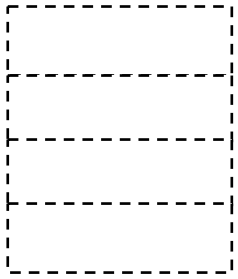
```
type color = (red, orange,
              yellow, green,
              blue, indigo,
              violet);

var primary: set of color;
...
primary := [red, yellow, blue];
```

- **Operaciones**:

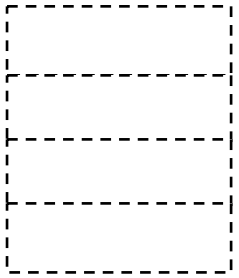
–pertenencia –unión –comparación
–inclusión –intersecció –complemento
–eliminación –diferencia –cardinalidad

$$S \subseteq U = \{0 \cdots u-1\}$$
$$|S| = n$$

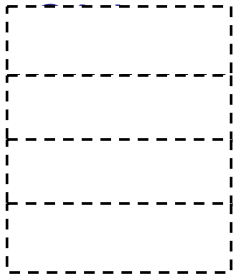
- **Vectores**:
  - espacio
  - pertenencia
  - inclusión
  - vacío



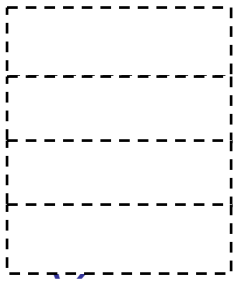
- **Árboles binarios**:
  - espacio
  - pertenencia
  - inclusión
  - vacío



- **Vectores de bits**:
  - espacio
  - pertenencia
  - inclusión
  - vacío



- **Tablas hash de tamaño *t***:
  - espacio
  - pertenencia
  - inclusión
  - vacío



Conjuntos

- **Briggs y Torczon, 1994:**

$$S = \{2, 5, 7, 4\}, \; u = 8$$

$d[i]$	i	$s[i]$
2	0	
5	1	
7	2	0
4	3	
	4	3
	5	1
	6	
	7	2

$c = 4$

- ***c*** es la cardinalidad de ***S***.
- ***d*[ ]** contiene los elementos de ***S*** en cualquier orden.
- pertenencia de ***v***:

$$0 \leq s[v] < c \;\wedge\; d[s[v]] = v$$

- inclusión y eliminación de ***v*** :

$$\begin{aligned} d[c] &= v \\ s[v] &= c \\ c &= c + 1 \end{aligned}$$

$$\begin{aligned} c &= c - 1 \\ d[s[v]] &= d[c] \\ s[d[c]] &= s[v] \end{aligned}$$

Operation	*Bit Vector*	*Sparse*
member	$O(1)$	$O(1)$
add-member	$O(1)$	$O(1)$
delete-member	$O(1)$	$O(1)$
clear-set	$O(u)$	$O(1)$
choose-one	$O(u)$	$O(1)$
cardinality	$O(u)$	$O(1)$
forall	$O(u)$	$O(n)$
copy	$O(u)$	$O(n)$
compare	$O(u)$	$O(n)$
union	$O(u)$	$O(n)$
intersect	$O(u)$	$O(n)$
difference	$O(u)$	$O(n)$
complement	$O(u)$	$O(u)$

set of char: u = 255
set of integer (32): u = 4.294.967.296
set of integer (64): u = 1.84467E+19

3. Asignación en pila

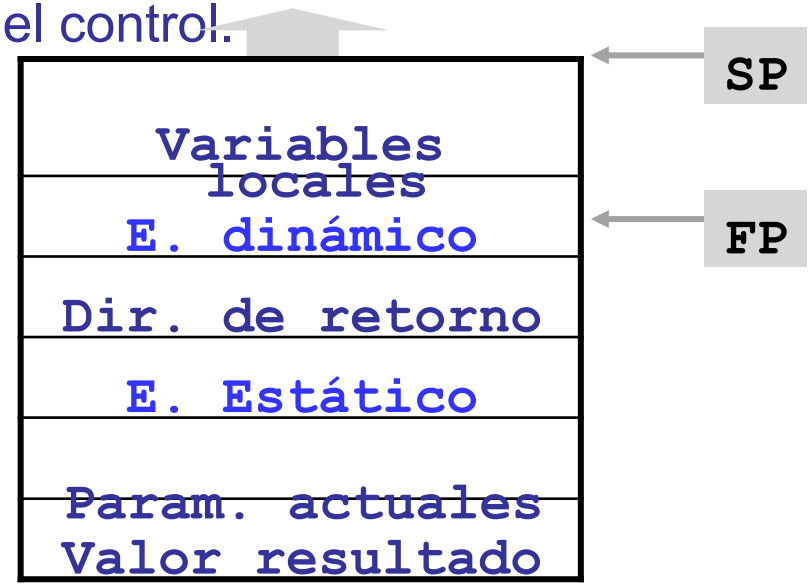
- Lenguajes como PASCAL, C, ADA, permiten la invocación ***recursiva*** de procedimientos y funciones.
- Puede haber ***más de una instancia*** de una variable en algún momento de la ejecución.
- La cantidad de objetos que se debe crear ***no se conoce*** en compilación.
- Hace falta una estructura de tipo ***pila*** (el último procedimiento invocado es el primero que termina), normalmente llamada *pila de ejecución*.

- Se organiza en ***bloques de activación*** *(BAs),* c/u mantiene la información relevante a la ejecución de un procedimiento. En particular, sus variables locales.
- Cada ***invocación*** a un procedimiento implica la creación de un nuevo bloque de activación en la pila.
- Administración, dos punteros:
  - El ***stack pointer*** (SP) señala el tope de la pila de ejecución.
  - El ***frame pointer*** (FP) señala el bloque de activación del procedimiento actualmente en ejecución.

En algunos casos el esquema de pila no es válido (no se cumple la condición *LIFO*).

Bloques de Activación

- BA: datos locales + información del control.



- ¿creación y destrucción?
- ¿tamaño?
- ¿direccionar variables locales?
- ¿... variables no locales?
- Múltiples instancias de un procedimiento en ejecución: ¿cómo localizar el BA correcto?
- ¿paso de parámetros?

- Los BAs se crean (destruyen) durante la ***ejecución*** cuando los procedimientos son invocados (terminan).

- Cada BA se crea al invocar un procedimiento mediante un proceso llamado ***secuencia de invocacion***.

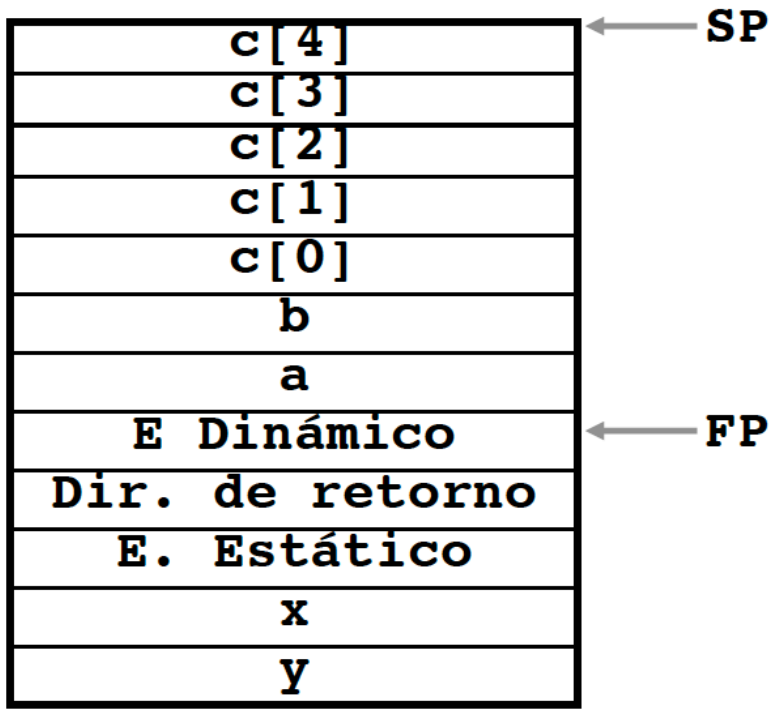
- El espacio ocupado por el BA es devuelto a la pila cuando el procedimiento termina, como parte de un proceso llamado ***secuencia de retorno***.

El tamaño de la pila de ejecución NO puede determinarse en el momento de la compilación.

Acceso a variables locales

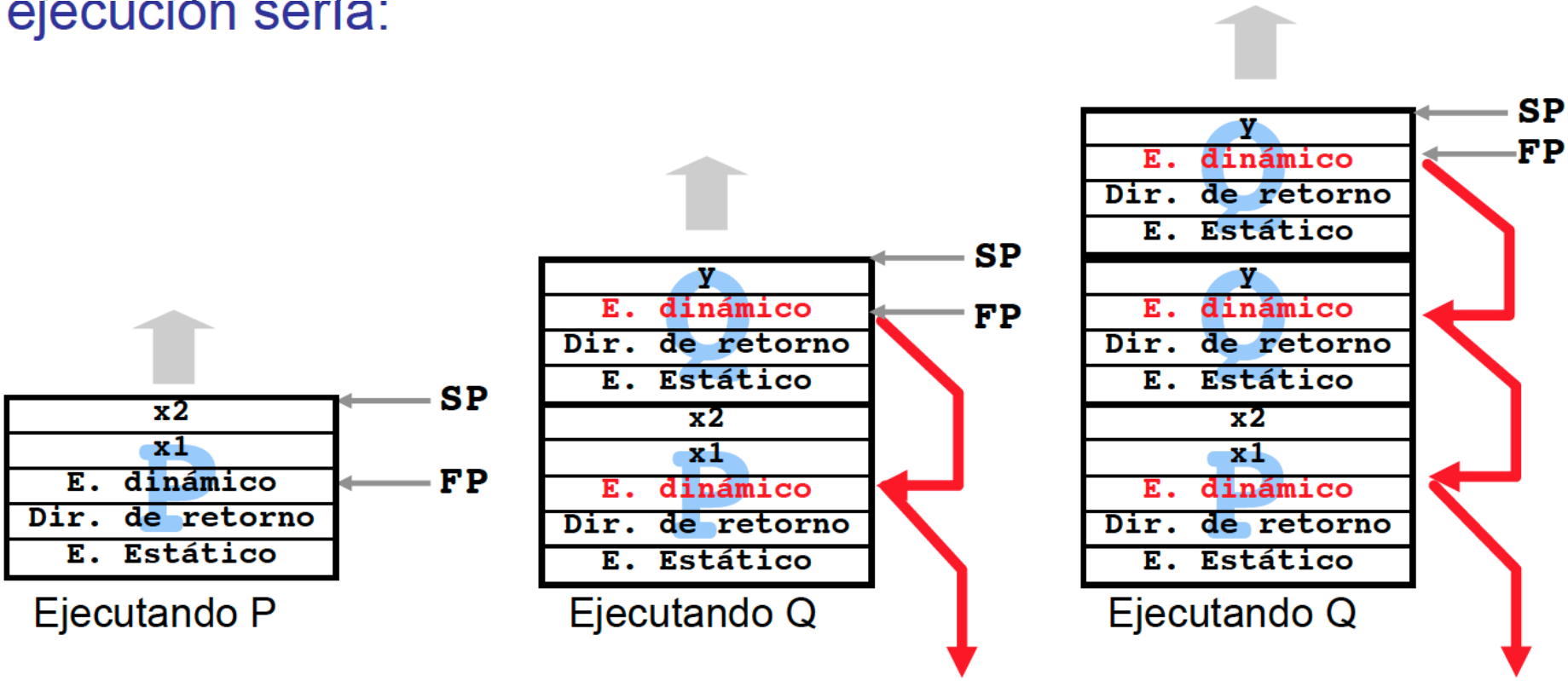
- Para permitir el acceso a variables locales, el compilador genera código ***relativo*** al FP.
- La localización de cada variable local es ***conocida en compilación***, (a cada una se le asigna un desplazamiento relativo al FP).
- Algunas arquitecturas proveen registros especiales para su utilización como FP.
- Las ***variables locales estáticas*** NO forman parte de este bloque (se almacenan en el bloque del programa principal, o estático).

```
void test (int x,
           int y)
{
  int a, b, c[5];
  static int d;
  ...
}
```



Encadenamiento dinámico

Considera el procedimiento **P** , con dos variables locales **x1** y **x2**, y el procedimiento **Q**, con una variable local **y**. Suponiendo que una invocación a **P** es seguida por una invocación a **Q**, y luego por otra invocación a **Q**, el estado de la pila de ejecución sería:

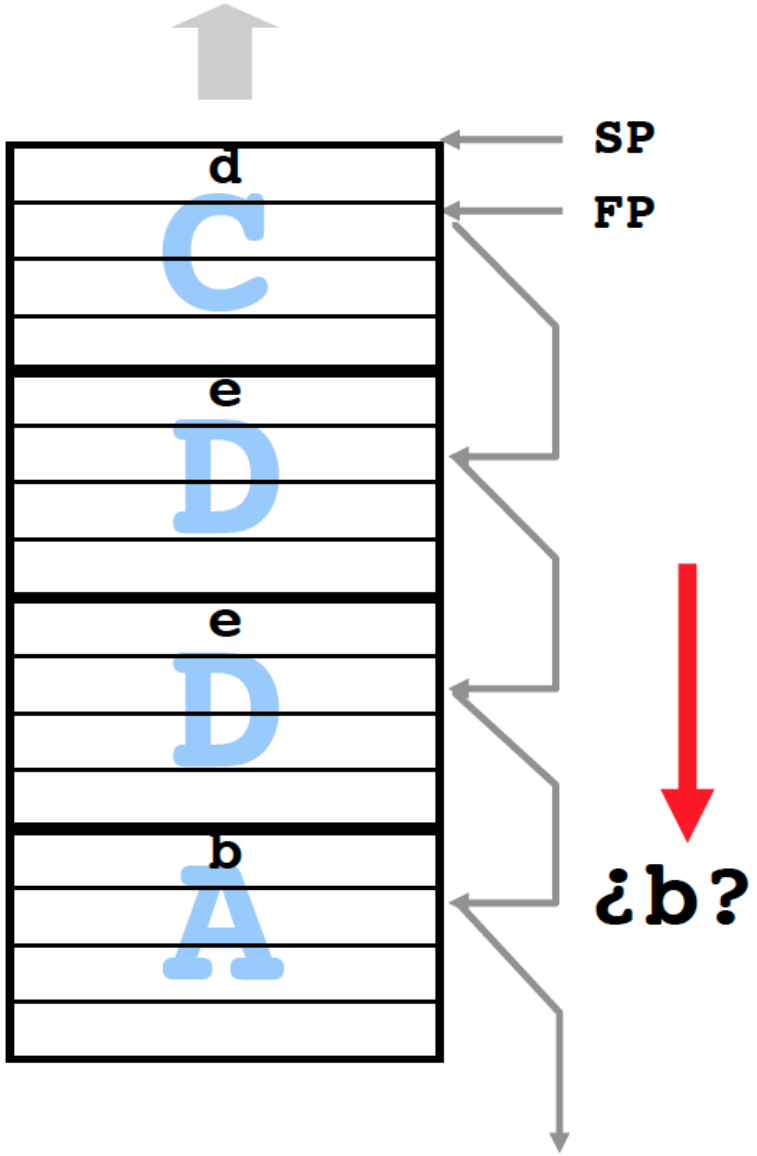


E. Dinámico = FP anterior

Acceso a variables no locales

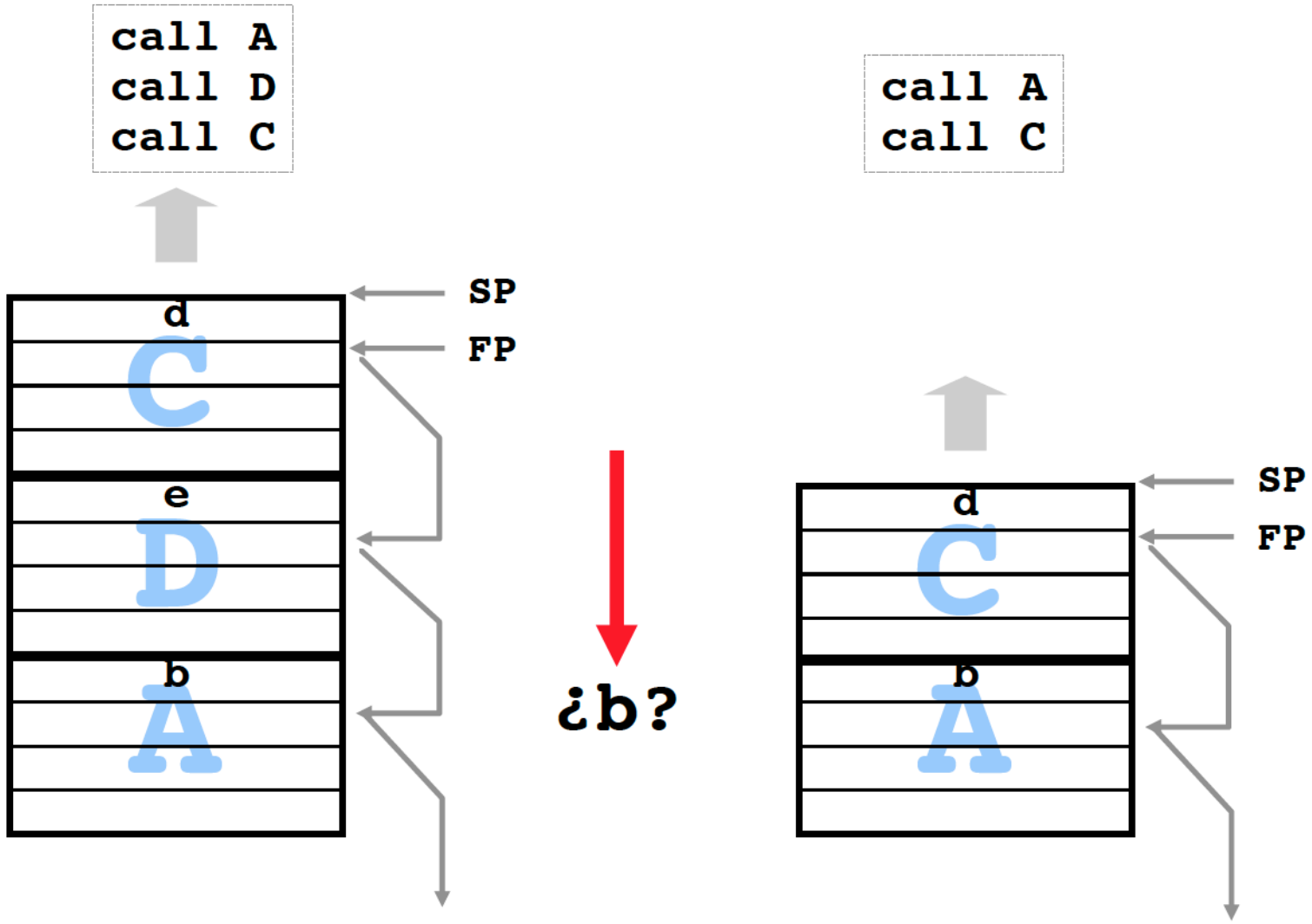
- Algunos lenguajes (ADA) permiten procedimientos ***anidados***.
- C no tiene variables no locales (no permite el anidamiento).

```
procedure A;
var b;
      procedure C;
      var d;
      begin
      ...
          b :=  .....
      end;

      procedure D;
      var e;
      begin
      ...
      end;
begin
...
end;
```



Acceso a variables no locales

- No es un problema debido solamente a la recursividad

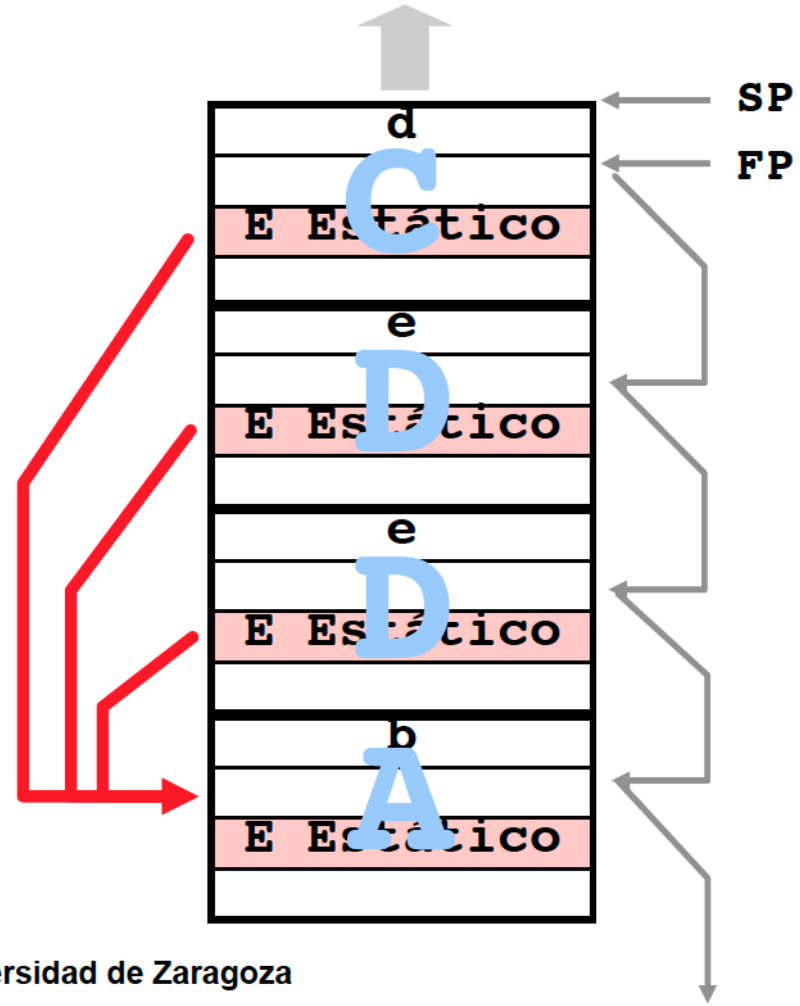


Solución: encadenamiento estático

- Puntero almacenado en el BA de un procedimiento (el encapsulado) que apunta a la instancia más reciente del procedimiento que lo encapsula.

Encad. Estático

Refleja la estructura estática de bloques.

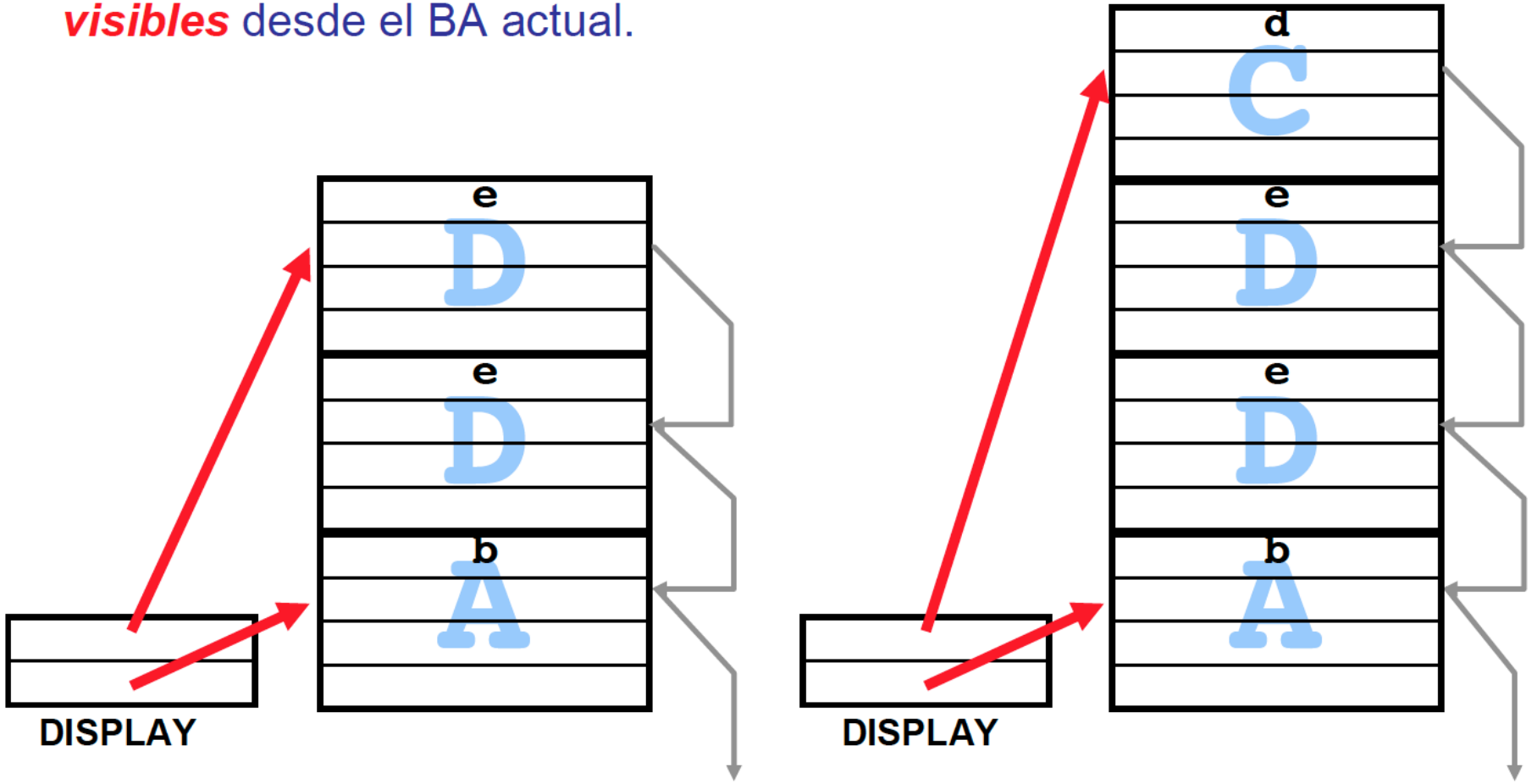


Encad. Dinámico

Refleja la secuencia de invocaciones.

Displays

- **Displays**: pila de punteros que contienen la dirección de todos los BAs ***visibles*** desde el BA actual.



- El acceso a variables no locales es poco frecuente:
  - 80 % de accesos a globales o locales; 17% a no locales del bloque inmediatamente anterior; 3% restante a no locales de bloques no inmediatamente anteriores.

Parámetros de tamaño variable

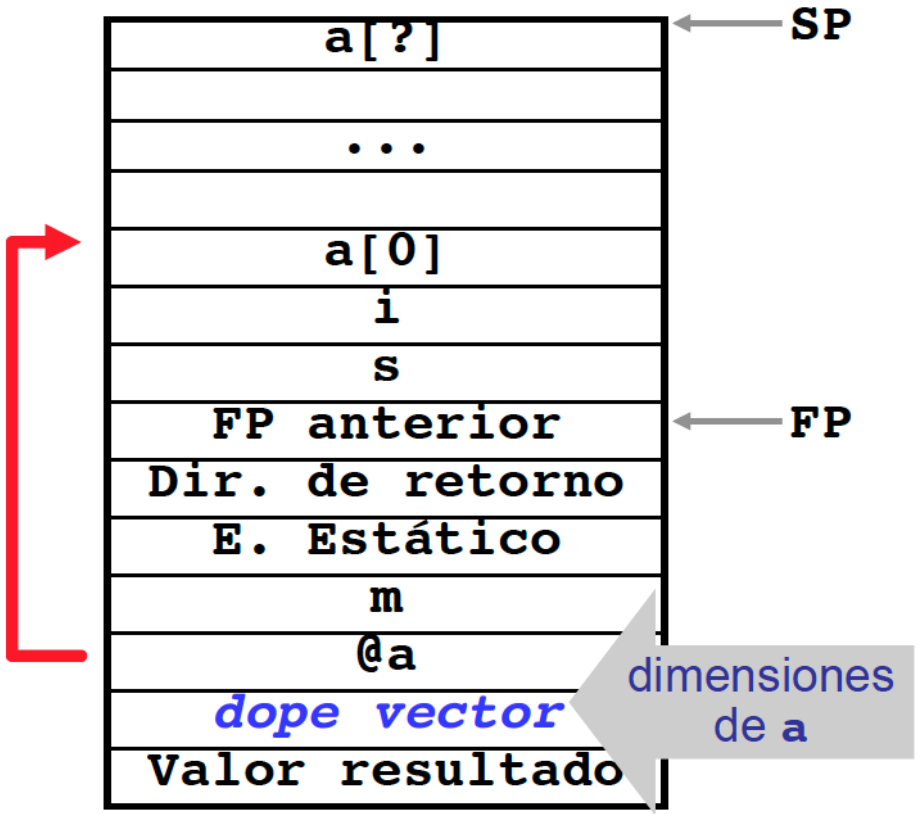
- Algunas veces no es posible determinar en compilación el tamaño del BA de un procedimiento.

```
type vector is array
     (integer range <>) of
       real;
function suma(a: vector;
                m: real)
       return real is
       s : real := m;
begin
       for i in a'range
      loop
             s := s + a(i);
       end loop;
       return s;
end suma;
```

¿m?

El tamaño de **suma** depende del tamaño de **a**.

- Una posible solución consiste en definir el parámetro como un puntero al comienzo del vector, que al elaborar se coloca con las variables locales (en el tope de la pila).



Más...

- En las arquitecturas sin registros todas las operaciones se hacen en la **pila de ejecución** (*operand stack)*.

- Algunos lenguajes permiten declarar variables locales a bloques (Algol, C , Ada).

  – Crear un BA para cada procedimiento y cada bloque. Es costoso en tiempo.

  – Crear un BA solamente para procedimientos. Los bloques se administran con ***overlays*** al BA del procedimiento.

```
int f (int x, int y)
{
    int z, w;
    ...
    {
        int v, i;
        ...
    }
    ...
    {
        int a, b;
        ...
    }
}
```

Invocación de procedimientos

- La ***secuencia de invocación*** está constituida por una serie de *acciones* para crear un bloque de activación, salvar cualquier información requerida, e inicia-lizar datos que lo requieran.

1. Copiar los argumentos
2. Salvar dirección de retorno
3. Establecer e. dinámico
4. Establecer e. estático
5. FP <- SP
6. Saltar al comienzo de la rutina

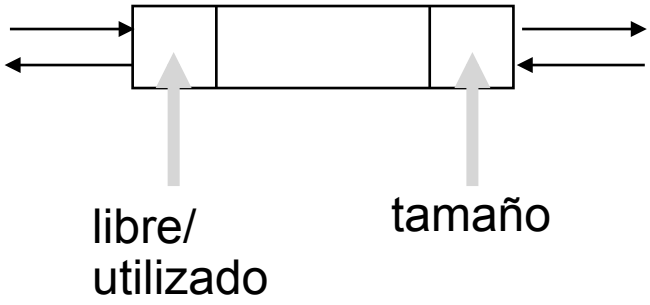
- La ***secuencia de retorno*** es el conjunto de instrucciones que se ejecutan cuando el procedimien-to invocado termina.

1. Obtener la dirección de retorno.
2. Devolver el valor resultados de los argumentos.
3. Restaurar el FP del e. dinámico
4. Salvar el resultado de la función en el BA restaurado.
5. Saltar a la dirección de retorno

4. Asignación dinámica

- ***HEAP:*** Permite asignar y desasignar memoria para un objeto del programa en cualquier momento. El mecanismo más flexible, pero más costoso.
  - **Explícita:** instrucciones específicas (**`new`** en PASCAL).
  - **Implícita:** se hace automáticamente (LISP)
  - **Inexistente:** no hay instrucción específica del lenguaje, se hace a través de funciones de biblioteca (**`malloc`** en C). El sistema operativo no puede intervenir.
  - **Intermedia:** el lenguaje proporciona operadores, pero pueden ser sobrecargados (**`new`** y **`delete`** de C++).

- **Estructura***:* lista doblemente encadenada



- **Inicialización de punteros:** Es importante verificar que apuntan a zonas válidas del *heap*.

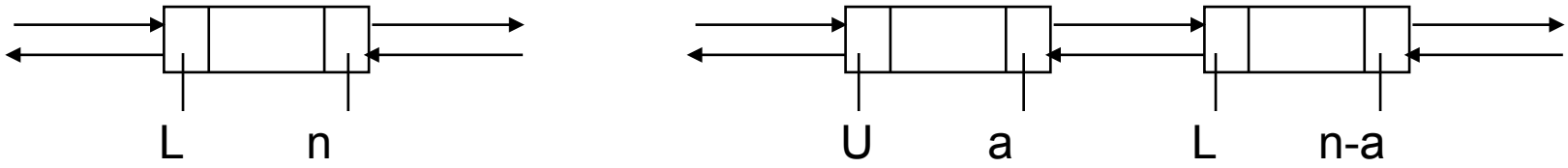
```
var p : ^integer;
begin
   ...
   p^ := 100;
   ...
end;
```

¡puede alterar otras zonas de la memoria!

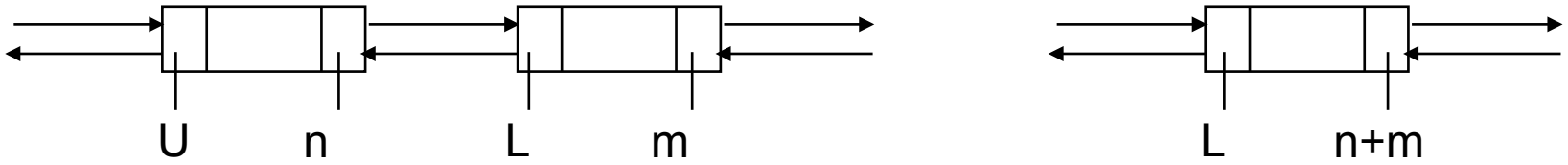
¡puede ser desasignado incorrectamente!

Administración del Heap

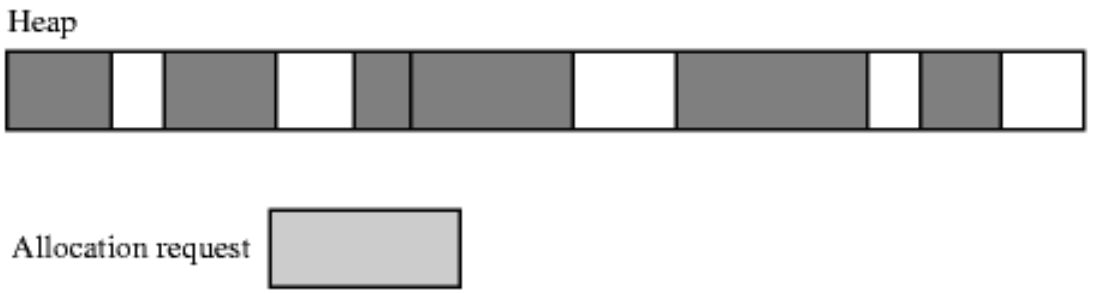
- **Asignación:** al escoger el bloque libre, lo divide en uno utilizado con el espacio pedido y uno libre con el espacio restante.



- **Desasignación:** al liberar un bloque determina si los contiguos también están libres para fusionarlos.



- **Fragmentación:** no hay bloques de memoria suficientemente grandes, pero en total hay suficiente memoria.



Asignación

¿qué hacer si hay varios bloques de tamaño suficiente?

- ***Best-fit:*** se escoge el bloque que desperdicie el mínimo espacio.
  - Ineficiente.
  - Fragmenta en trozos pequeños e inútiles.
- ***First-fit:*** se asigna el primero de la lista que sea adecuado.
  - Inicialmente más eficiente.
  - Los trozos pequeños tienden a juntarse al comienzo de la lista.
- ***Circular-first-fit:*** buscar desde el punto donde se encontró el último bloque.
  - Estadísticamente más eficiente
- ***Bit map:*** a menudo un programa requiere bloques del mismo tamaño (p.e. registros de un tipo definido).
  - *heap* específico para los tamaños frecuentes.
  - Cada elemento de un *bit map* puede indicar si un bloque determinado está libre o está siendo utilizado.
  - Evita la fragmentación.
- ***Buddy system:*** *heaps* específicos para bloques de tamaño 2^k.
  - Si no hay un bloque disponible de tamaño 2^k, se divide uno de tamaño 2^{k+1}.

Desasignación

- **Ignorarla:** cuando la memoria se acaba, el programa de detiene.
  - Útil si los objetos tienen permanencia, o hay una gran cantidad de memoria virtual.
  - Puede ser la única alternativa viable (sistemas tiempo real).
- **Explícita:** a través de funciones como **`free()`**, **`dispose`** **`unchecked_deallocation`**.
  - Responsabilidad del programador
  - Errores MUY frecuentes

```
var p,q : ^real;
    ....
    new (p);
    q := p;
    dispose (p);
    q^ := 1.0;
```

error

- **Implícita:** Se lleva cuenta de los espacios ocupados y libres.
  - **Referencia única:** sólo un puntero apuntando a c/bloque. Cuando el ámbito del puntero se cierra, se libera la memoria.

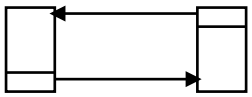
```
var p : ^real;
begin
    ... new (p); ...
end;
```

liberar `p^`

  - **Referencias múltiples:** se cuentan. Referencias = 0? liberar memoria.

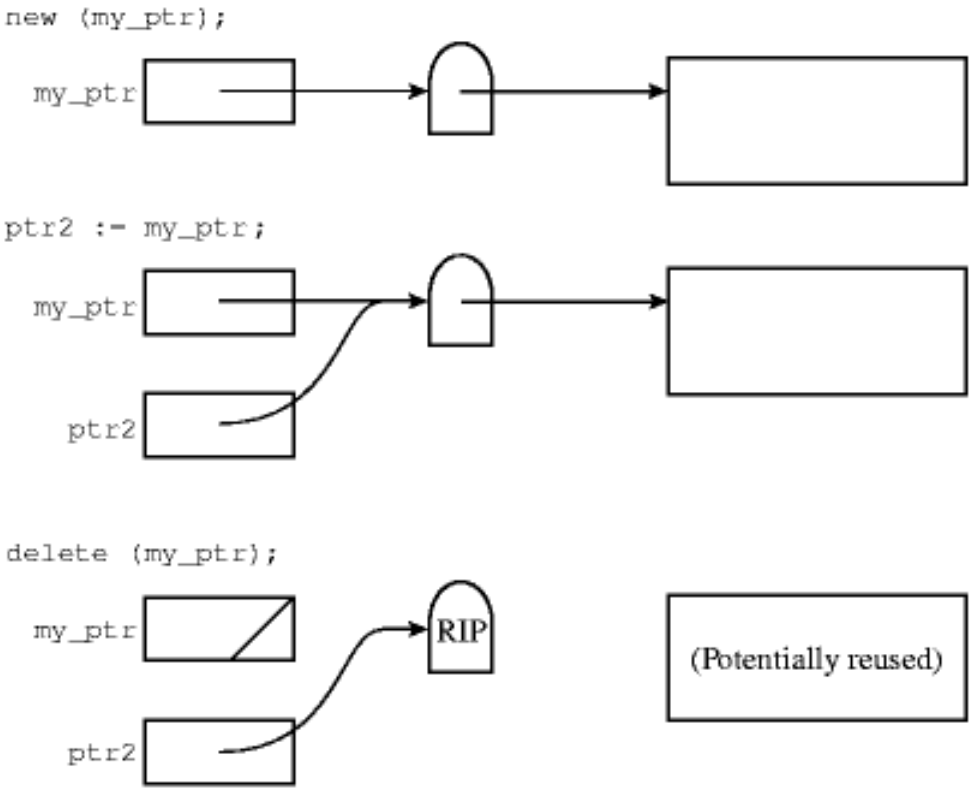
```
var p,q : ^real;
begin
  1 new (p);
    ....
  2 q := p;
    q^ := 1.0;
0 end;
```

» Excepción:



Desasignación

- Lápidas (*Tombstones*):

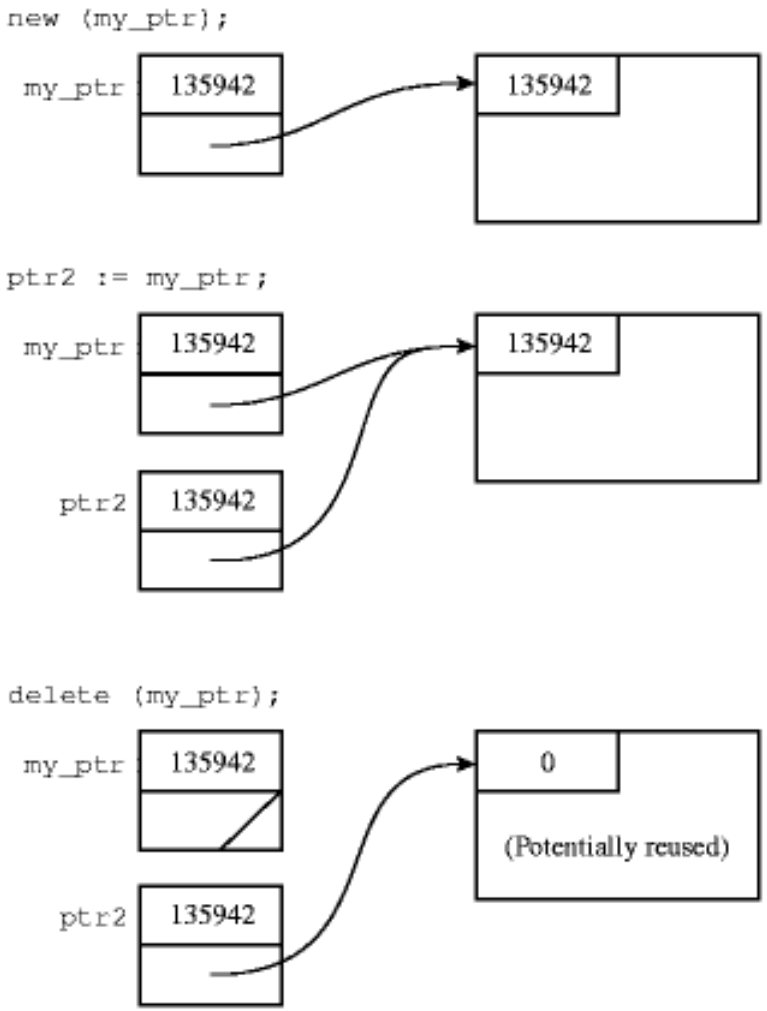


- Nivel adicional de indirección
- El puntero apunta a una lápida
- La lápida apunta al objeto
- Cuando el puntero desaparece, la lápida se invalida.
- Autoconsumo:
  - Creación (las lápidas son objetos en el heap).
  - Verificación de la validez del acceso a través de punteros
  - Eliminación de las lápidas (puede hacerse por referencias múltiples).

Permite mantener la consistencia en las referencias al heap

Desasignación

- Cerraduras y llaves (*locks and keys*):



- C/puntero está compuesto por una dirección y una llave.
- C/objeto en el *heap* tiene un cerradura.
- Un puntero es válido cuando la cerradura de la dirección apuntada coincide con la llave.
- Al asignar espacio en el *heap*, se crea una nueva cerradura.
- Al liberar el espacio, se invalida la cerradura (con llave 0, p.e.)
- Es poco ***probable*** que esa posición de memoria llegue a contener el código de una cerradura por casualidad.

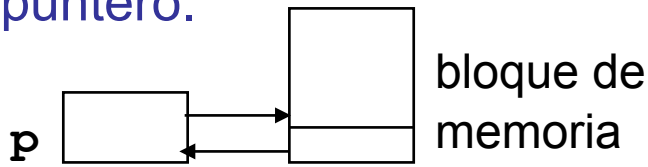
Recolección de basura

- Cuando la memoria se termina, el administrador del *heap* debe determinar qué bloques de memoria no se están siendo utilizados, y liberarlos.

- ***Mark-and-sweep:*** seguimiento (recursivo) de todos los punteros; se marcan las regiones del *heap* en utilización. Lo no marcado es inaccesible, se libera.

  – Hace falta una estructura de datos adicional para saber donde están los punteros.

- ***Hand-shaking:***
  c/bloque de memoria solo está apuntado por un puntero. El bloque memoria a su vez apunta a su puntero.



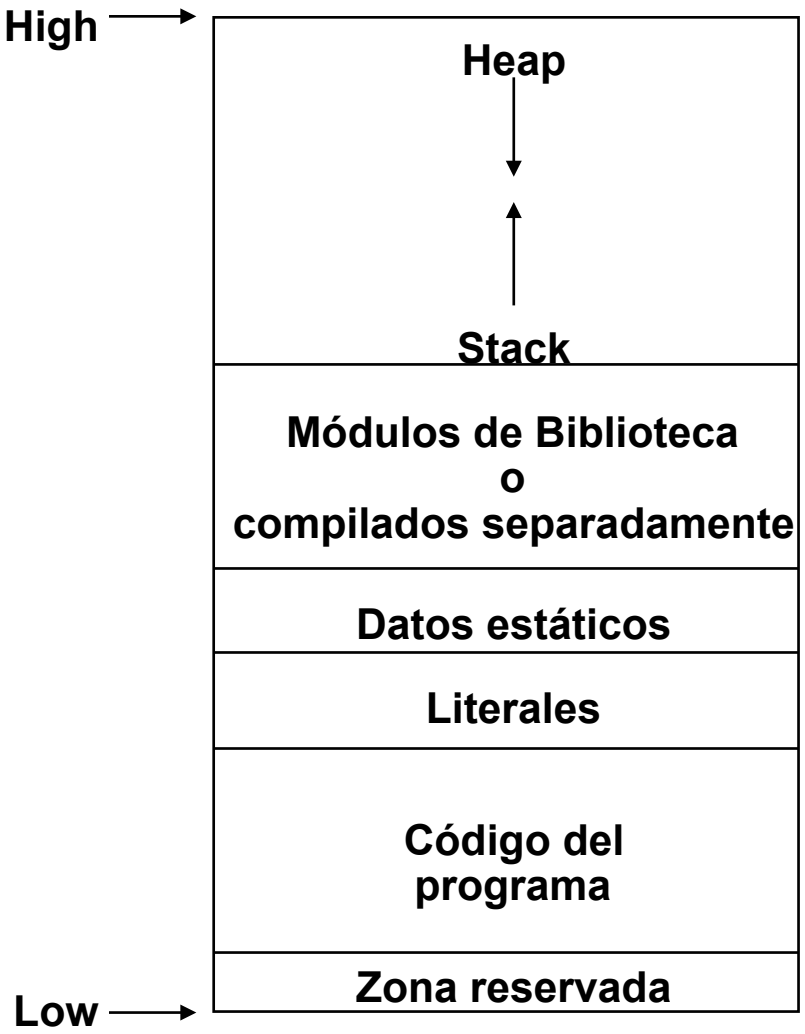
  – Si **p** no está apuntando a su bloque de memoria, la memoria se libera.

```
new (p);
p^ := 1.0;
...
new (p);
```

hand-shake diferente

- ***Generational scavenging:***
  se basa en el principio de que los objetos más antiguos tienden a tener un tiempo de vida mayor y viceversa.

5. Disposición en memoria



1. Se asignan los segmentos estáticos
2. Los módulos de biblioteca contendrán sus propios segmentos estáticos.
3. La memoria restante se comparte entre el *stack* y el *heap*.
4. Cuando hay colisión, se hace recolección de basura.
5. Cuando hay memoria segmentada (8086) el *stack* y el *heap* quedan en segmentos diferentes por lo que no puede haber colisión.
6. Algunas arquitecturas reservan zonas de memoria para operaciones específicas

6. Perspectiva

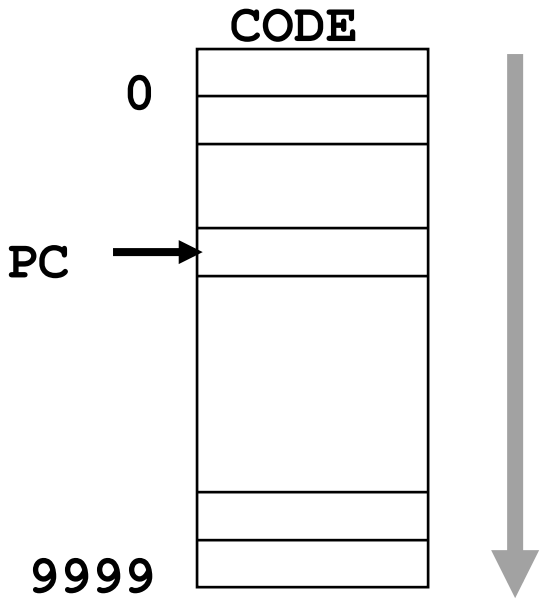
- Algunos aspectos de la semántica del lenguaje son difíciles de entender (y más de implementar):

```
B0: begin
      procedure  A(F); procedure  F;
      begin
            F(1)
      end;

      procedure  B;
      begin
            procedure G(X) : integer X;
            begin
            ....
            end;
            A(G);
      end;
    end;
```

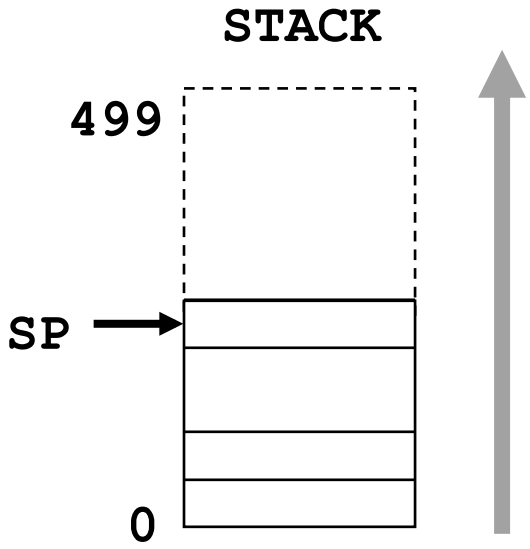
¿Cómo saber el encadenamiento estático de F en el procedimiento A?

7. Máquina P

- Elementos de la máquina virtual:



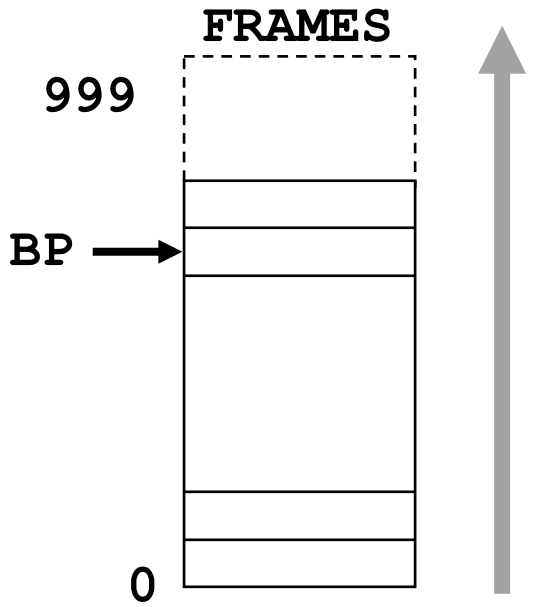
Vector de Instrucciones

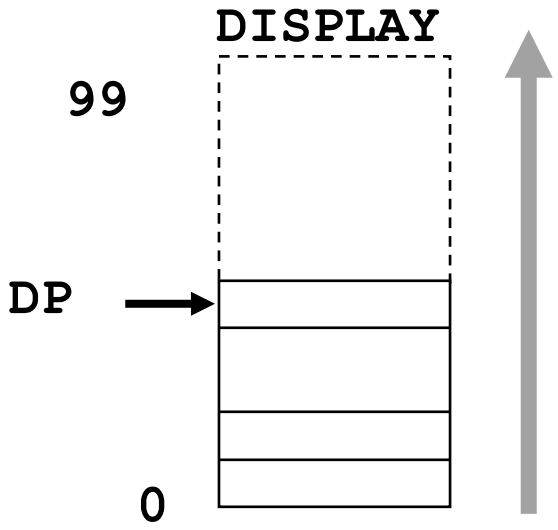


Pila de evaluación (16 bits)
(valores temporales)

Máquina P

- Elementos de la máquina virtual:





Pila de Ejecución (16 bits) (Bloques de Activación, Almacenamiento de variables)

Encadenamiento Estático (16 bits)

Máquina P

- Operadores aritméticos:

```
PLUS  push (pop2() + pop1())
SBT   push (pop2() - pop1())
TMS   push (pop2() * pop1())
MOD   push (pop2() mod pop1())
DIV   push (pop2() div pop1())
NGI   push (-pop())
```

- Operadores lógicos:

```
AND   push (pop2() and pop1())
OR    push (pop2() or pop1())
EQ    push (pop2() = pop1())
NEQ   push (pop2() <> pop1())
LT    push (pop2() < pop1())
LTE   push (pop2() <= pop1())
GT    push (pop2() > pop1())
GTE   push (pop2() >= pop1())
NGB   push (not pop())
```

- Entrada/salida:

```
RD n   if n = 0 read char and
        store at frames[pop()]
        else as int
WRT n  if n = 0 write pop()
                  as char
        else as int
```

- Miscelánea

```
NOP
SWP   push (pop1(); pop2())
DUP   push (pop1(); pop1())
```

- Comentarios

```
; <linea>
```

Máquina P

- Referencia a datos:

```
STC n       push(n)
SRF f o     push (display[DP - f] + o)
DRF         push (frames[pop()])
ASG         frames[pop2()] = pop1()
ASGI        frames[pop1()] = pop2()
```

- Control de flujo:

```
ENP   n         enter program at n
LVP             leave program
JMP   n         PC = n
JMT   n         pop()=1? PC = n else PC = PC+1
JMF   n         pop()=0? PC = n else PC = PC+1
OSF   s l a      open stack frame (crear BA)
CSF             close stack frame (destruir BA)
```

- Etiquetas:

```
ENP   L
JMP   L
JMT   L
JMF   L
```



```
L: <instr>
```

Ejemplo 1

```
programa max;
entero i, j;

principio
   leer (i);
   leer (j);
   si i > j ent
      escribir (i) ;
   si_no
      escribir (j) ;
   fsi
fin
```

```
; Programa max.
    ENP L0
; Comienzo del programa max.
; Leer.
; Direccion de variable i.
L0:
    SRF  0  3
    RD  1
; Leer.
; Direccion de variable j.
    SRF  0  4
    RD  1
; SI.
; Acceso a variable i.
    SRF  0  3
    DRF
; Acceso a variable j.
    SRF  0  4
    DRF
    GT
    JMF L1
; ENT.
; Escribir.
; Acceso a variable i.
    SRF  0  3
    DRF
    WRT  1
    JMP L2
; SI NO.
; Escribir.
; Acceso a variable j.
L1:
    SRF  0  4
    DRF
    WRT  1
; Fin SI.
L2:
    LVP
```

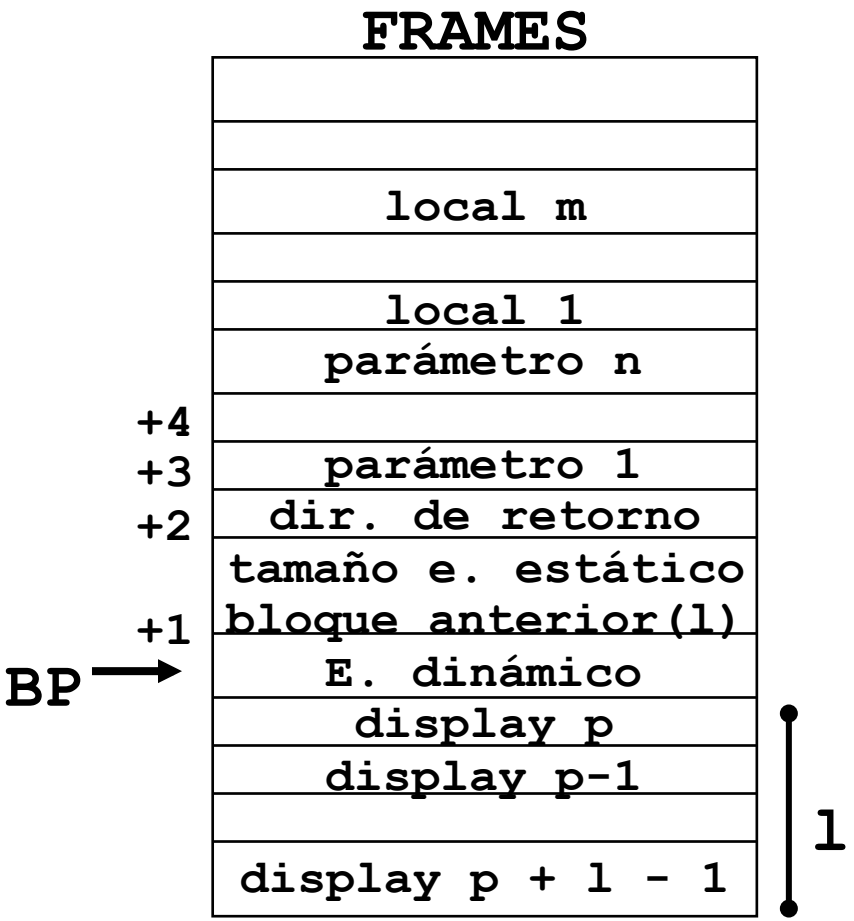
Ejemplo 2

```
programa fact;
entero n, f, i;

principio
   leer (n);
   f := 1;
   i := 2;
   mq i <= n
         f := f * i;
         i := i + 1;
   fmq
   escribir (f)
fin
```

```
; Programa fact.
    ENP L0
; Comienzo de fact.
; Leer.
; Direccion de var n.
L0:
    SRF  0  3
    RD  1
; Direccion de var f.
    SRF  0  4
    STC  1
; Asignacion.
    ASG
; Direccion de var i.
    SRF  0  5
    STC  2
; Asignacion.
    ASG
```

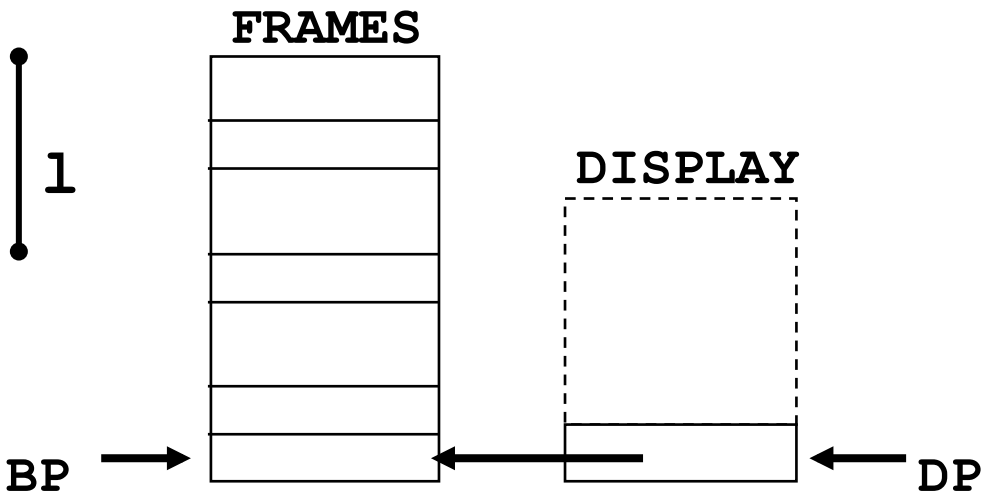
```
L1:
; MQ
; Acceso a variable i.
    SRF  0  5
    DRF
; Acceso a variable n.
    SRF  0  3
    DRF
    LTE
    JMF L2
; Direccion de var f.
    SRF  0  4
; Acceso a var f.
    SRF  0  4
    DRF
; Acceso a var i.
    SRF  0  5
    DRF
    TMS
; Asignacion.
    ASG
; Direccion de var i.
    SRF  0  5
; Acceso a var i.
    SRF  0  5
    DRF
    STC  1
    PLUS
; Asignacion.
    ASG
    JMP L1
; Fin MQ.
; Escribir.
; Acceso a var f.
L2:
    SRF  0  4
    DRF
    WRT  1
    LVP
```

Bloques de Activación



- Inicialización:

```
/* contruir el bloque de   */
/* activación del programa */
FRAMES[0] = -1;
BP = 0;

/* inicializar pilas       */
DP = -1;
SP = -1;

/* establecer encad. estat.*/
dpush (BP);

/* comenzar ejecución      */
PC = 0;
```



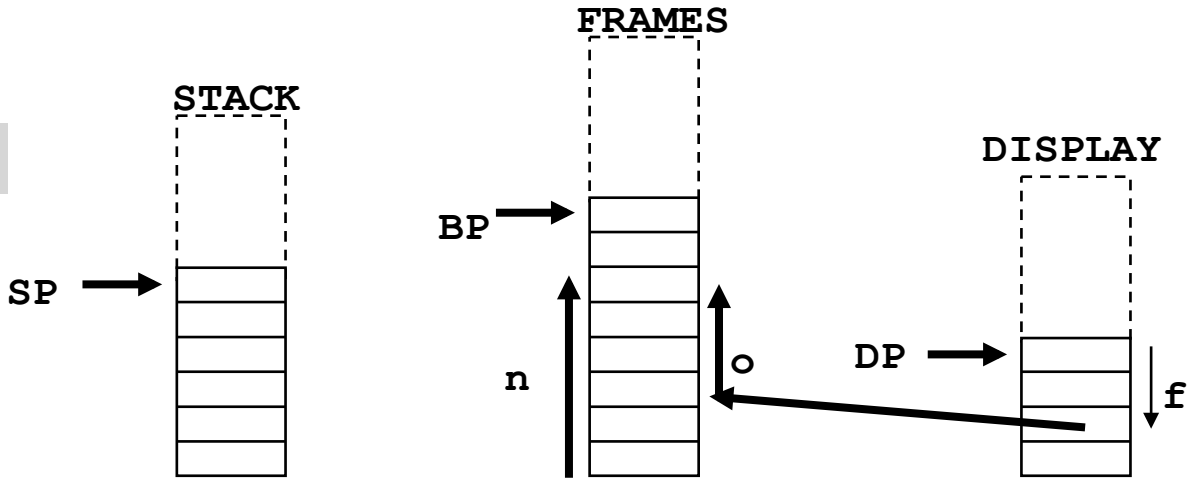
ENP ¿addr?

```
programa fact;
entero i;

funcion f (val entero n)
         dev entero;
principio
  si n > 0 ent
     dev (n * f ( n - 1));
  si_no dev (1);
  fsi
fin

principio
 leer (i);
 escribir (f(i));
fin
```

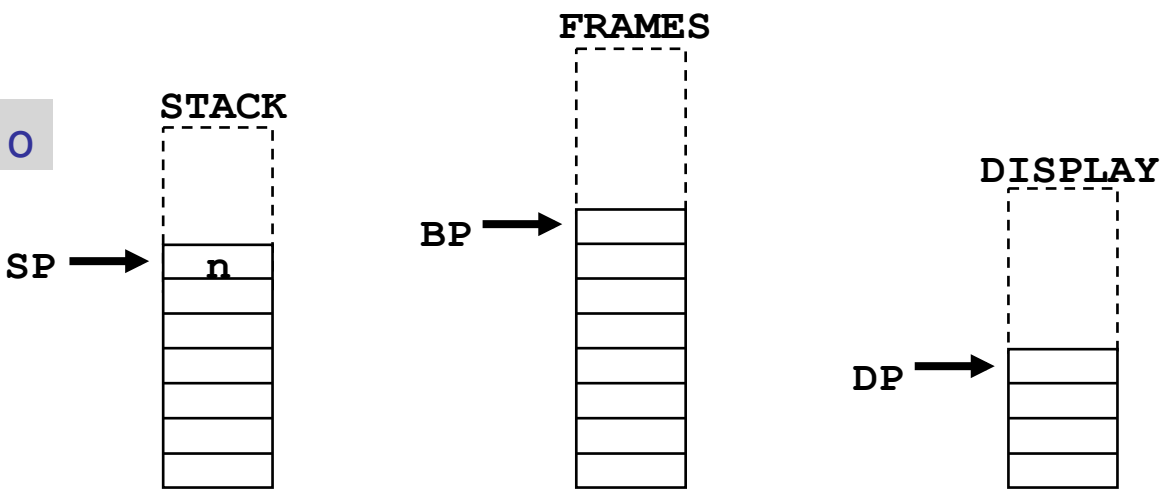
```
; fact;
          ENP   L0
; f
          SRF   0   3
          ASGI
          JMP   L1
; comienzo f
    L1:   SRF   0   3
          DRF
          STC   0
          GT
          JMF   L2
          SRF   0   3
          DRF
          SRF   0   3
          DRF
          STC   1
          SBT
          OSF   4   1   1
          TMS
          CSF
          JMP   L3
    L2:   STC   1
          CSF
    L3:   CSF
; comienzo fact
    L0:   SRF   0   3
          RD    1
          SRF   0   3
          DRF
          OSF   4   0   1
          WRT   1
          LVP
```

SRF f o

Antes de ejecutar SRF f o



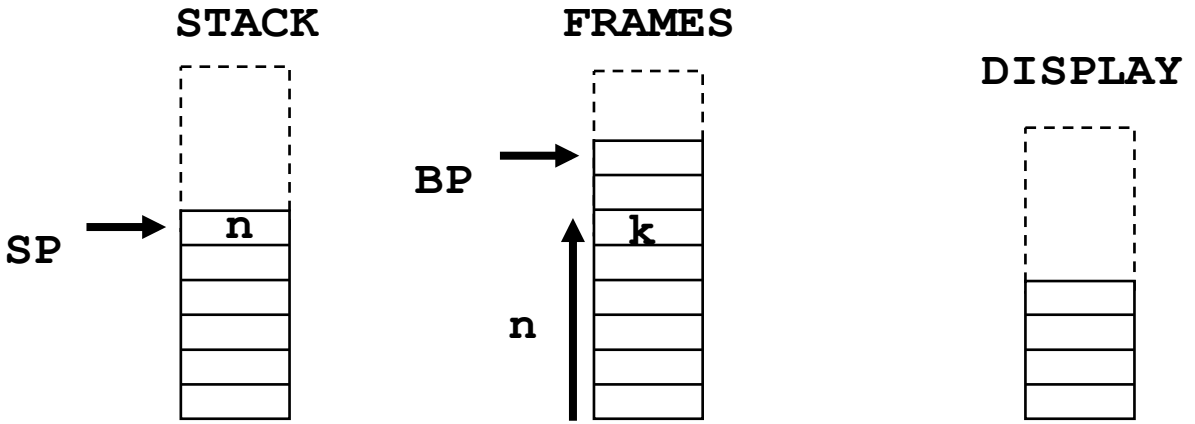
Después de ejecutar SRF f o



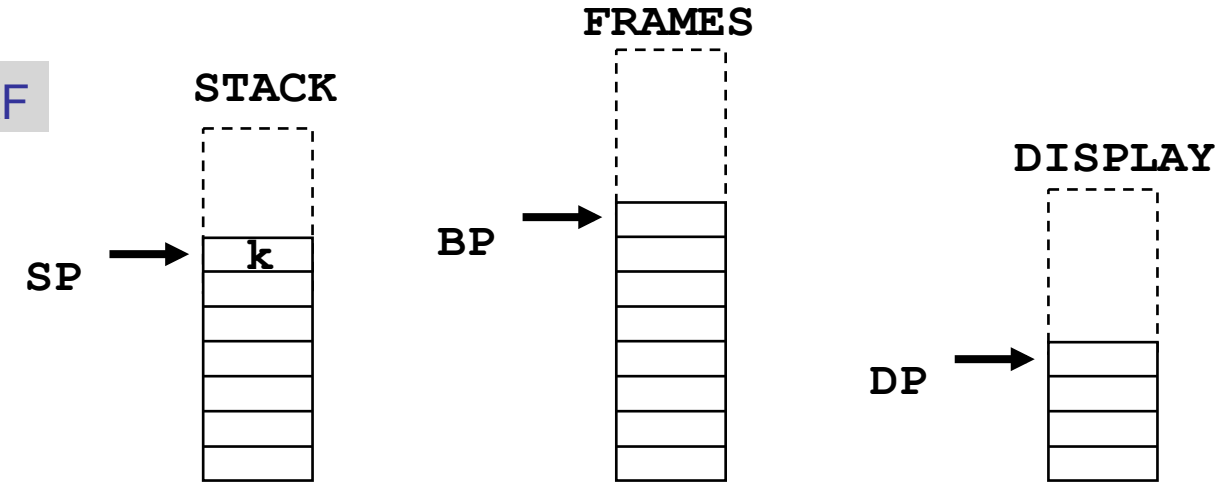
Almacena en la pila de ejecución *la dirección absoluta* de la variable en el offset `o, f` bloques más abajo

DRF

Antes de ejecutar DRF



Después de ejecutar DRF

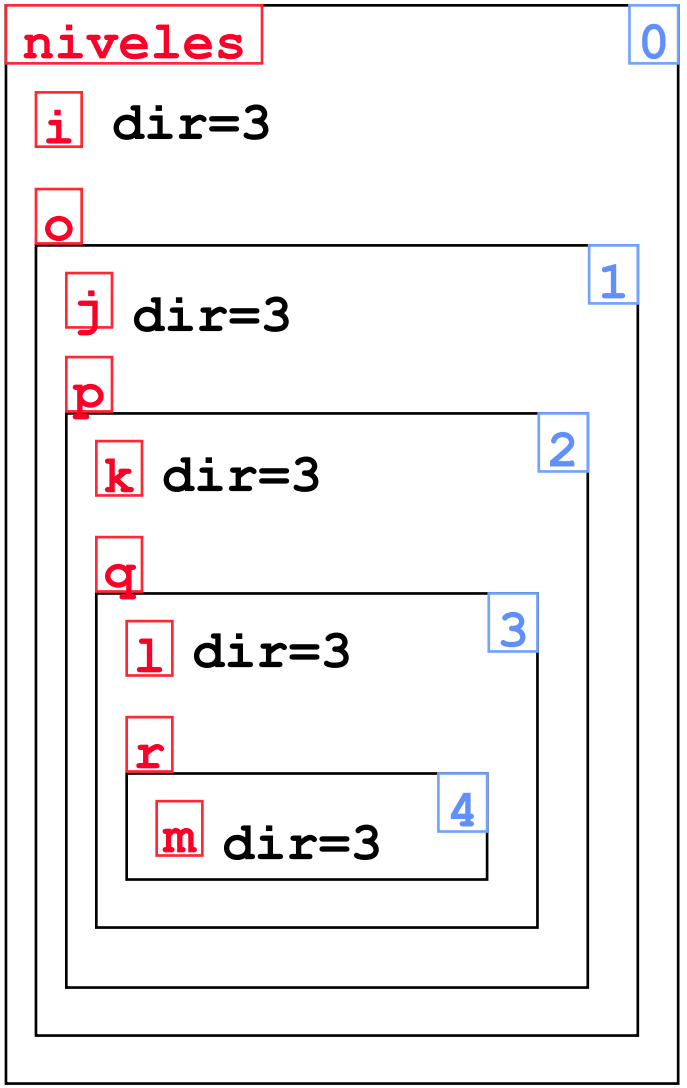


Almacena en la pila de ejecución *el valor* de la variable cuya dirección estaba en la pila.

Bloques de Activación

```
programa niveles;
0  entero i;
   accion o;
      entero j;
1     accion p;
         entero k;
2        accion q;
            entero l;
3           accion r;
               entero m;
4              principio
               i := i+1; j := j+1; k := k+1;
               l := l+1; m := 5;
               escribir (i, j, k, l, m);
               r; q; p; o
               fin
            principio
            i := i+1; j := j+1; k := k+1;
            l := 4;
            r; q; p; o;
            fin
         principio
         i := i+1; j := j+1; k := 3;
         q; p; o;
         fin
   principio
   i := i+1; j := 2; p; o;
   fin
principio
i := 1;
o
fin
```

Estructura estática:



Código Generado

```
  ; comienzo r
  ;
  ; i := i + 1
  ;
  6: SRF    4   3
  7: SRF    4   3
  8: DRF
  9: STC    1
 10: PLUS
 11: ASG
  ;
  ; j := j + 1
  ;
 12: SRF    3   3
 13: SRF    3   3
 14: DRF
 15: STC    1
 16: PLUS
 17: ASG
```

Nivel actual: 4
Nivel de i: 0
f = 4 - 0
o = 3

Nivel actual: 4
Nivel de j: 1
f = 4 - 1
o = 3

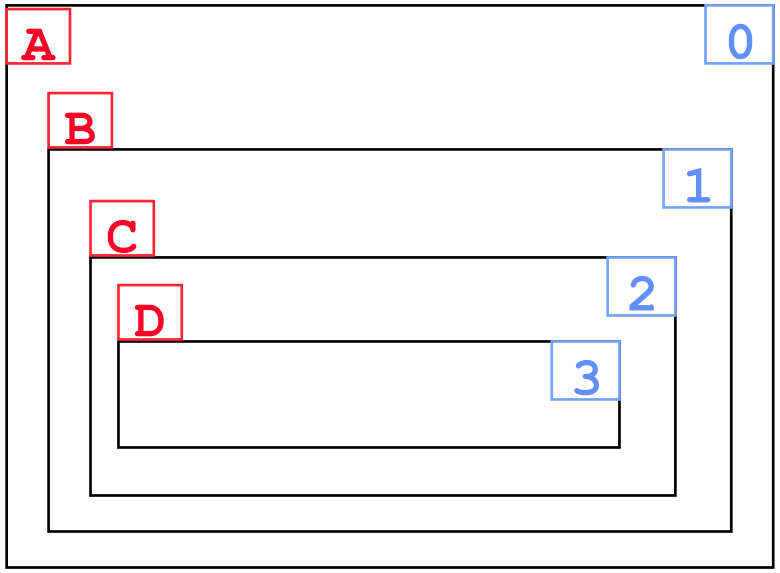
```
  ;
  ; k := k + 1
  ;
 18: SRF    2   3
 19: SRF    2   3
 20: DRF
 21: STC    1
 22: PLUS
 23: ASG
  ;
  ; l := l + 1
  ;
 24: SRF    1   3
 25: SRF    1   3
 26: DRF
 27: STC    1
 28: PLUS
 29: ASG
  ;
  ; m := 5
  ;
 30: SRF    0   3
 31: STC    5
 32: ASG
```

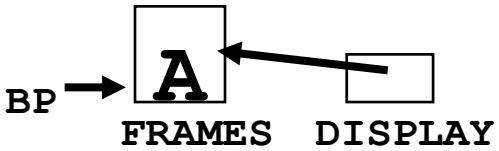
Nivel actual: 4
Nivel de k: 2
f = 4 - 2
o = 3

Nivel actual: 4
Nivel de l: 3
f = 4 - 3
o = 3

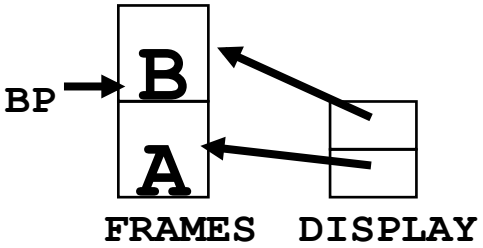
Nivel actual: 4
Nivel de m: 4
f = 4 - 4
o = 3

Encad. Estático .vs. Dinámico



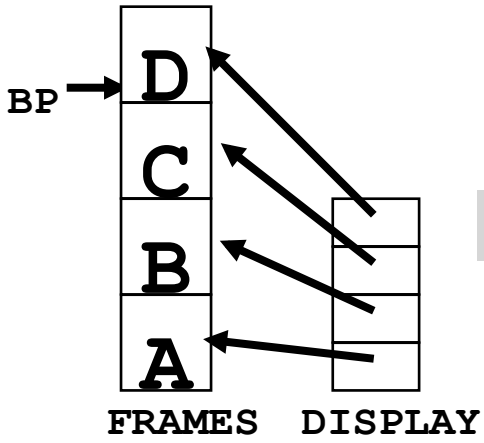


A invoca a B



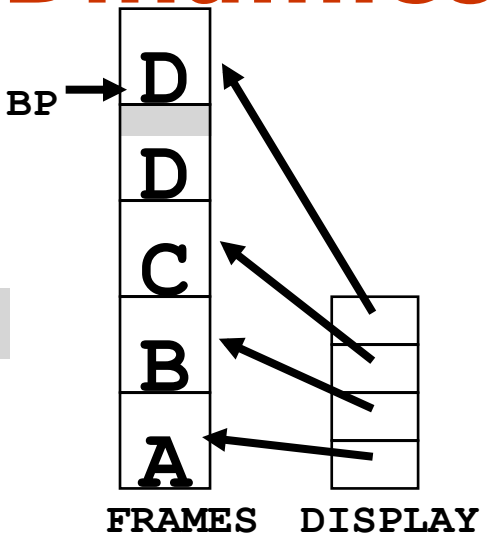
Ocultar **nivel_actual - nivel_invocado** (0 - 0) encadenamientos estáticos

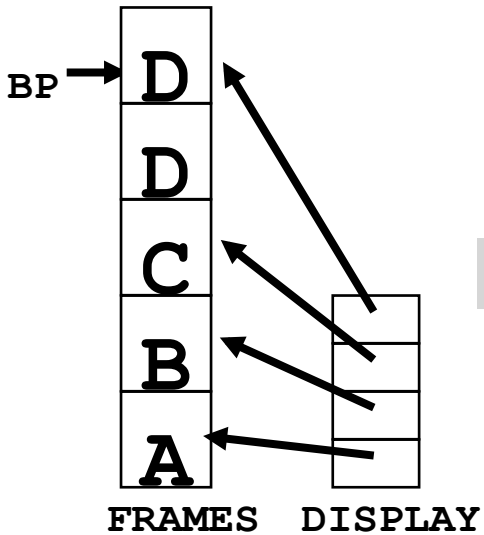
Encad. Estático .vs. Dinámico



D invoca a D

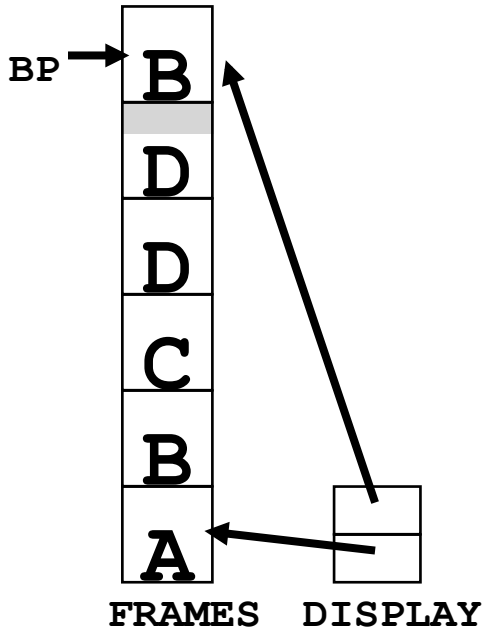
Ocultar (3 - 2) e. estáticos



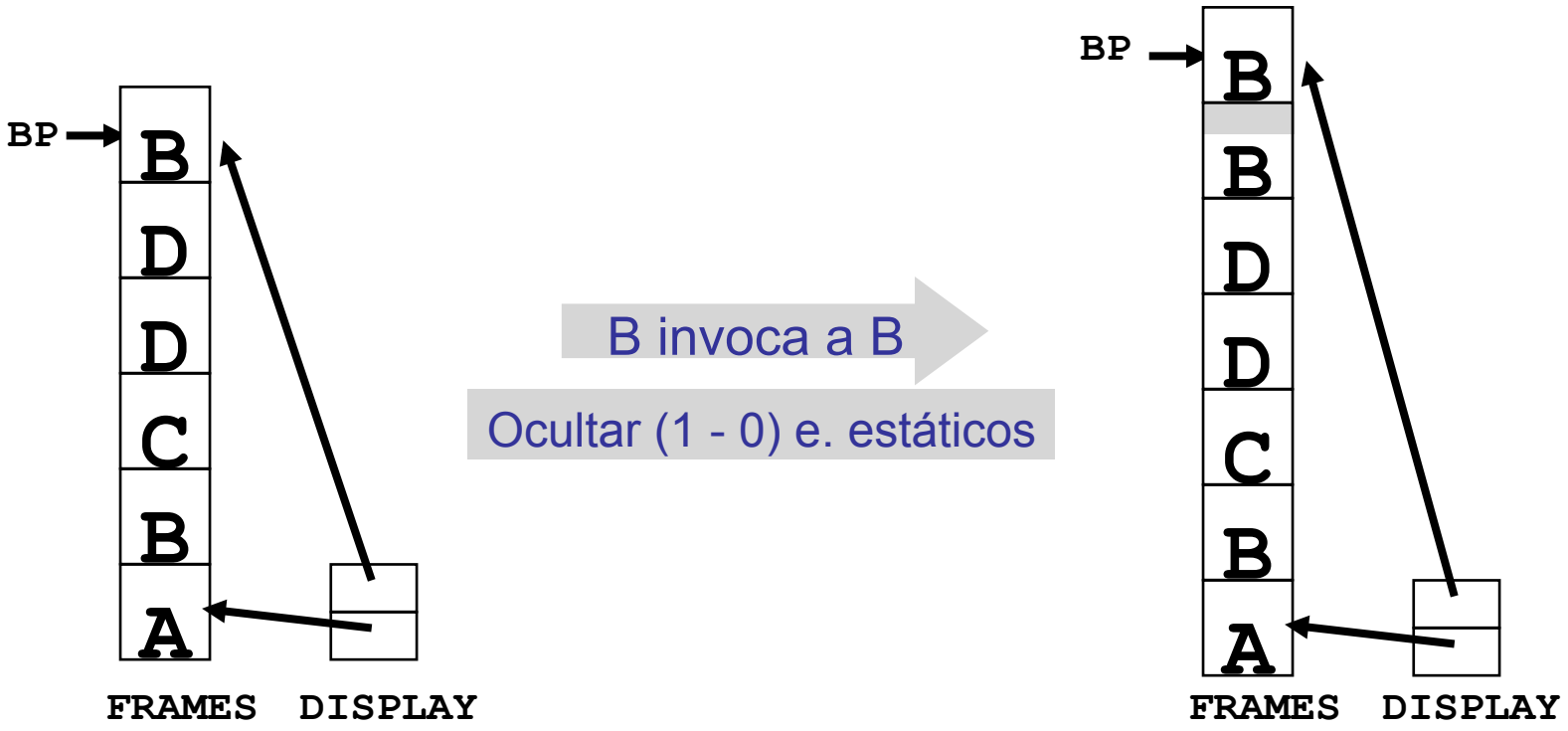


D invoca a B

Ocultar (3 - 0) e. estáticos



Encad. Estático .vs. Dinámico

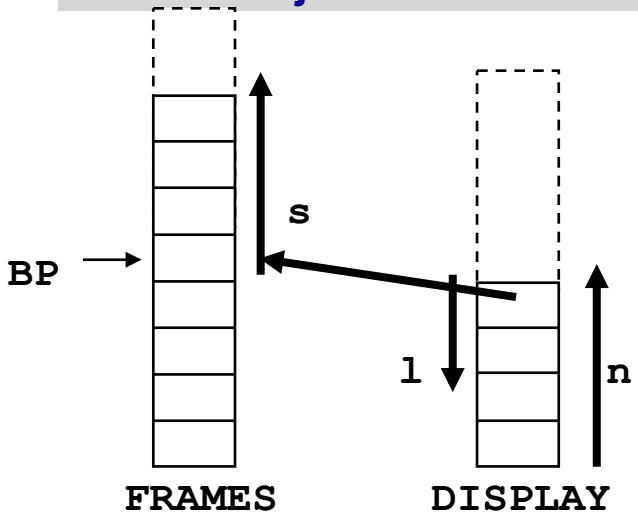


i : OSF (s)ize (l)evels (a)ddr

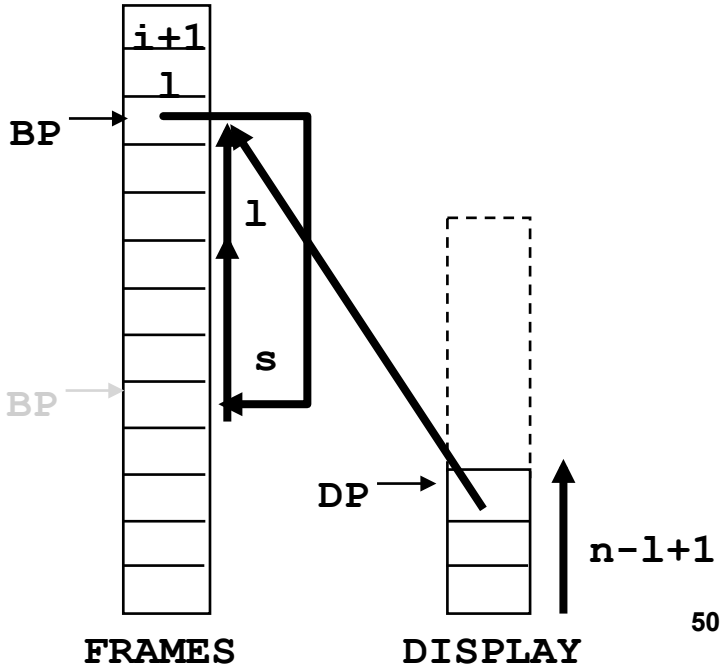
- Crear un bloque de activación:
  - respetando `s` componentes del bloque anterior.
  - ocultando `l` bloques de activación del DISPLAY.
  - comenzando la ejecución en `a`.

1) Salvar l componentes de DISPLAY en FRAMES
2) Eliminar l componentes de DISPLAY
3) Salvar el BP en FRAMES
4) BP <- nuevo tope de FRAMES
5) Salvar l en FRAMES
6) Salvar dirección retorno en FRAMES
7) Salvar el BP en DISPLAY
8) PC <- a

Antes de ejecutar OSF s l a



Después de ejecutar OSF s l a



Código Generado

```
;
; invocar r
;
    48   OSF     4    1    4
;
; invocar q
;
    49   OSF     4    2    3
;
; invocar p
;
    50   OSF     4    3    2
;
; invocar o
;
    51   OSF     4    4    1
;
; fin r
;
    52   CSF
```

Nivel actual: 4
Nivel de r: 3
l = 4 - 3

Nivel actual: 4
Nivel de q: 2
l = 4 - 2

Nivel actual: 4
Nivel de p: 1
l = 4 - 1

Nivel actual: 4
Nivel de o: 0
l = 4 - 0

CSF

- Destruir un bloque de activación:
  - eliminando el encadenamiento estático actual.
  - restaurando el encadenamiento estático anterior.
  - recuperando la dirección de retorno.
  - restaurando el bloque de activación anterior (a través del encadenamiento dinámico).