

conceptos contruídos sobre los más fundamentales

Paradigmas de la Programación

FaMAF 2020

capítulo 7.

(adicionales: 4.4. y 5.)

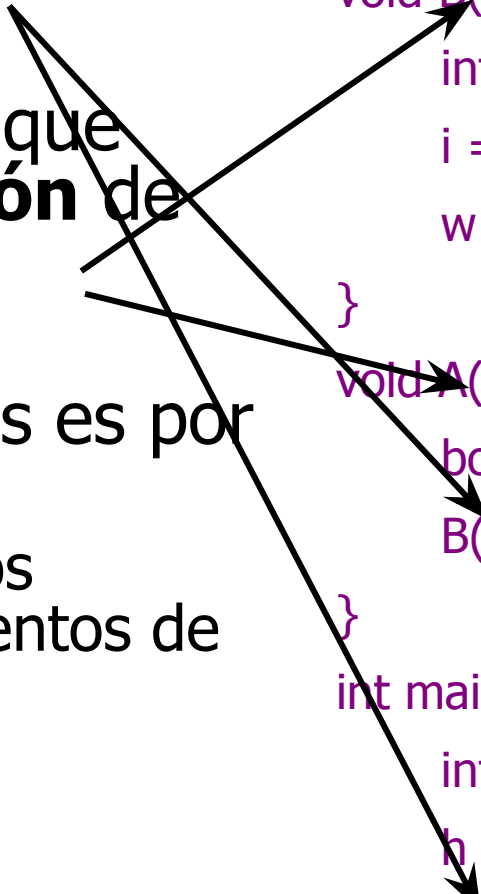
basado en filminas de [John Mitchell](#) y [Vitaly Shmatikov](#)

- pasaje de parámetros
- alcance y clausuras
- recursión a la cola

argumentos y parámetros

- **argumento**: expresión que aparece en una **llamada** a función
- **parámetro**: identificador que aparece en la **declaración** de una función
- la correspondencia entre parámetros y argumentos es por número y posición
 - excepto en Perl, donde los parámetros son los elementos de un arreglo especial @_

```
int h, i;  
void B(int w) {  
    int j, k;  
    i = 2*w;  
    w = w+1;  
}  
void A(int x, int y) {  
    bool i, j;  
    B(h);  
}  
int main() {  
    int a, b;  
    h = 5; a = 3; b = 2;  
    A(a, b);  
}
```



mecanismos de pasaje de parámetros

- por valor
- por referencia
- por valor-resultado
- por nombre
- por necesidad

pasaje por valor

- la función que llama pasa el **r-valor** del argumento a la función que es llamada
 - se computa el valor del argumento en la llamada
 - no hay “aliasing” (dos identificadores para una sola ubicación en memoria)
- la función no puede cambiar el valor de la variable de la función que llama
- C, Java, Scheme
 - se pueden pasar punteros si queremos que se pueda modificar el valor de la variable de la función que llama

```
void swap(int *a, int *b) { ... }
```

pasaje por referencia

- la función que llama pasa el **l-valor** del argumento a la función que es llamada
 - se asigna la dirección de memoria del argumento al parámetro
 - causa “aliasing” (dos identificadores para una sola ubicación en memoria)
- la función puede modificar la variable de la función que llama
- C++, PHP

comparación valor - referencia

- en el caso de trabajar con estructuras de datos grandes, cuál es la opción más económica?
- cuál es la opción que puede tener efectos secundarios?
- en lenguajes funcionales no hay diferencia entre pasaje por referencia y pasaje por valor, por qué?

ejemplo en C

```
void Modify(int p, int * q, int * o)
{
    p = 27; // passed by value
    *q = 27; // passed by value or reference, check call site
    *o = 27; // passed by value or reference, check call site
}


int main()
{
    int a = 1;
    int b = 1;
    int x = 1;
    int * c = &x;
    Modify(a, &b, c);    // a is passed by value, b is passed by
reference by creating a pointer,
                        // c is a pointer passed by value
    // b and x are changed
    return(0);
}
```


pasaje por referencia en C++

- el “tipo referencia” indica que el l-valor se pasa como argumento

```
void swap ((int& a, int& b))  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

los l-valores para los tipos referencia en C++ se determinan totalmente en tiempo de compilación



- el operador **&** está sobrecargado en C++
 - cuando lo aplicamos a una variable, nos da su l-valor
 - cuando lo aplicamos a un tipo en una lista de parámetros, significa que queremos pasar el argumento por referencia

dos formas de pasar por referencia

C o C++

```
void swap (int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

```
int x=3, y=4;  
swap(&x, &y);
```

solamente C++

```
void swap (int& a, int& b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
int x=3, y=4;  
swap(x, y);
```

pasaje por valor-resultado

- Intenta tener beneficios de llamada por referencia (efectos secundarios en los argumentos) sin los problemas de aliasing.
- Hace una copia en los argumentos al principio, copia las variables locales a los propios argumentos al final del procedimiento, de forma que se modifican los argumentos.
- Cuidado: el comportamiento depende del orden en que se copian las variables locales.
- Usado por BBC BASIC V

pasaje por nombre

- en el cuerpo de la función se sustituye textualmente el argumento para cada instancia de su parámetro
 - se implementó para Algol 60 pero sus sucesores no lo incorporaron
- es un ejemplo de ligado tardío
 - la evaluación del argumento se posterga hasta que efectivamente se ejecuta en el cuerpo de la función
 - asociado a evaluación perezosa en lenguajes funcionales (e.g., Haskell)

pasaje por necesidad

- Variación de *call-by-name* donde se guarda la evaluación del parámetro después del primer uso
- Idéntico resultado a *call-by-name* (y más eficiente!) si no hay efectos secundarios
- El mismo concepto que lazy evaluation

resumen de pasaje de parámetros

método	qué se pasa	lenguajes	comentarios
por valor (by value)	valor	C, C++	simple, los parámetros que se pasan no cambian, pero puede ser costoso
por referencia (by reference)	dirección	FORTRAN, C++	económico, pero los parámetros pueden cambiar!
por valor-resultado (by value-result)	valor + dirección	FORTRAN, Ada	más seguro que por referencia, pero más costoso
por nombre (by name)	texto	Algol	complicado, ya no se usa

Cuál es la salida?

```
begin
  integer n;
  procedure p(k: integer);
    begin
      k := k+2;
      print(n);
      n := n+(2*k);
    end;
  n := 4;
  p(n);
  print(n);
end;
```

- ☐ Call by value?
- ☐ Call by reference?
- ☐ Call by value-result?

Cuál es la salida?

```
begin
  integer n; integer m; integer r;
  procedure p(i j k);
    begin
      r := 5;
      i := k;
      r := 4;
      j := k+n
    end;
  n := 3; m := 4; r := 1
  p(n m (n+r));
  print(n);
end;
```

- ☐ Call by value?
- ☐ Call by reference?
- ☐ Call by value-result?
- ☐ Call by name?
- ☐ Call by need?

- pasaje de parámetros
- **alcance y clausuras**
- recursión a la cola

reglas de alcance

variables locales y globales

x, y son locales al bloque exterior

z es local al bloque interior

x, y son globales al bloque interior

```
{ int x=0;  
  int y=x+1;  
    { int z=(x+y)*(x-y);  
      };  
};
```

alcance estático: el valor de las variables globales se obtiene del bloque inmediatamente contenedor

alcance dinámico: el valor de las variables globales se obtiene del activation record más reciente

ambigüedad en alcance

```
val x = 4;  
fun f(y) = x*y;  
fun g(h) =  
  let  
    val x=7  
  in  
    h(3) + x;  
g(f);
```

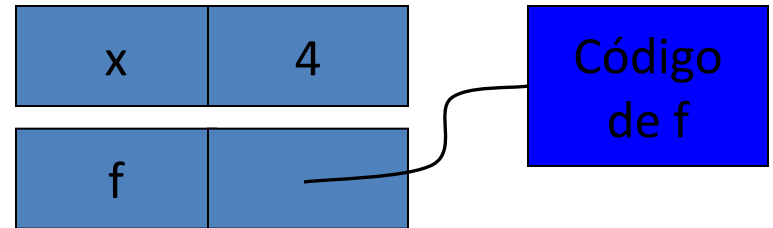
ambigüedad en alcance

```
val x = 4;  
fun f(y) = x*y;  
fun g(h) =  
  let  
    val x=7  
  in  
    h(3) + x;  
g(f);
```

x	4
---	---

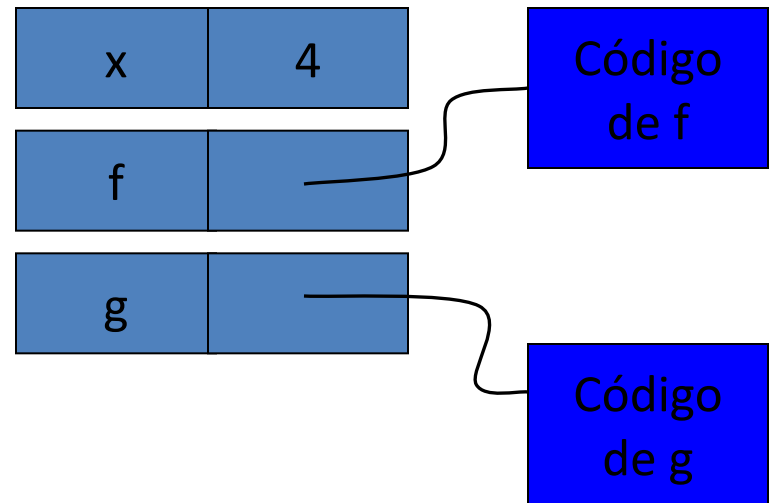
ambigüedad en alcance

```
val x = 4;  
fun f(y) = x*y;  
fun g(h) =  
  let  
    val x=7  
  in  
    h(3) + x;  
  g(f);
```



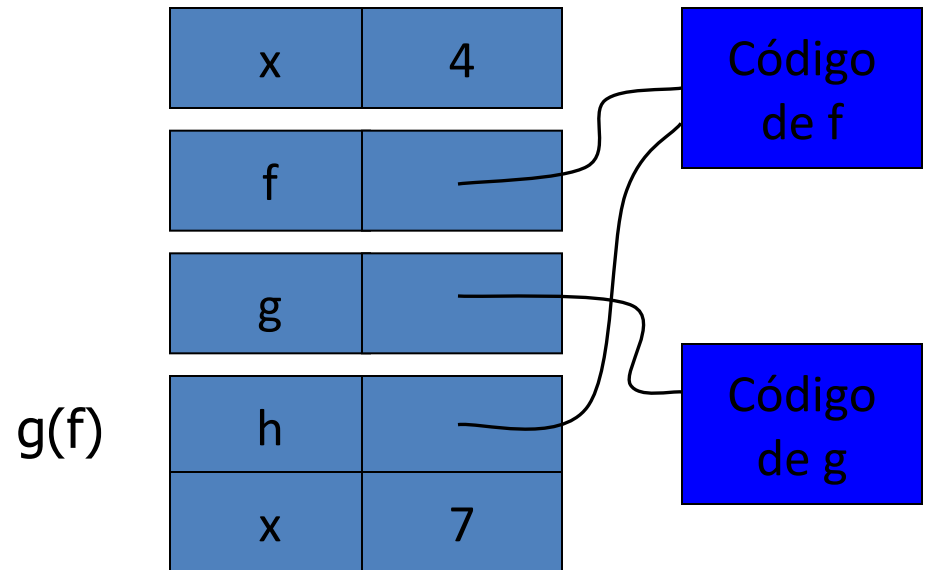
ambigüedad en alcance

```
val x = 4;  
fun f(y) = x*y;  
fun g(h) =  
  let  
    val x=7  
  in  
    h(3) + x;  
  g(f);
```



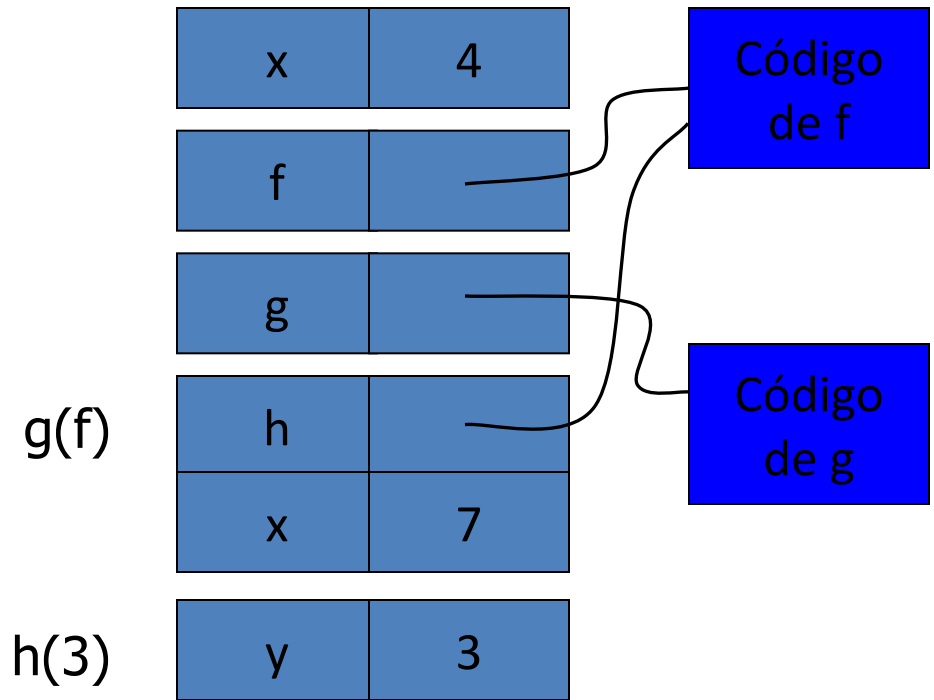
ambigüedad en alcance

```
val x = 4;  
fun f(y) = x*y;  
fun g(h) =  
  let  
    val x=7  
  in  
    h(3) + x;  
  g(f);
```



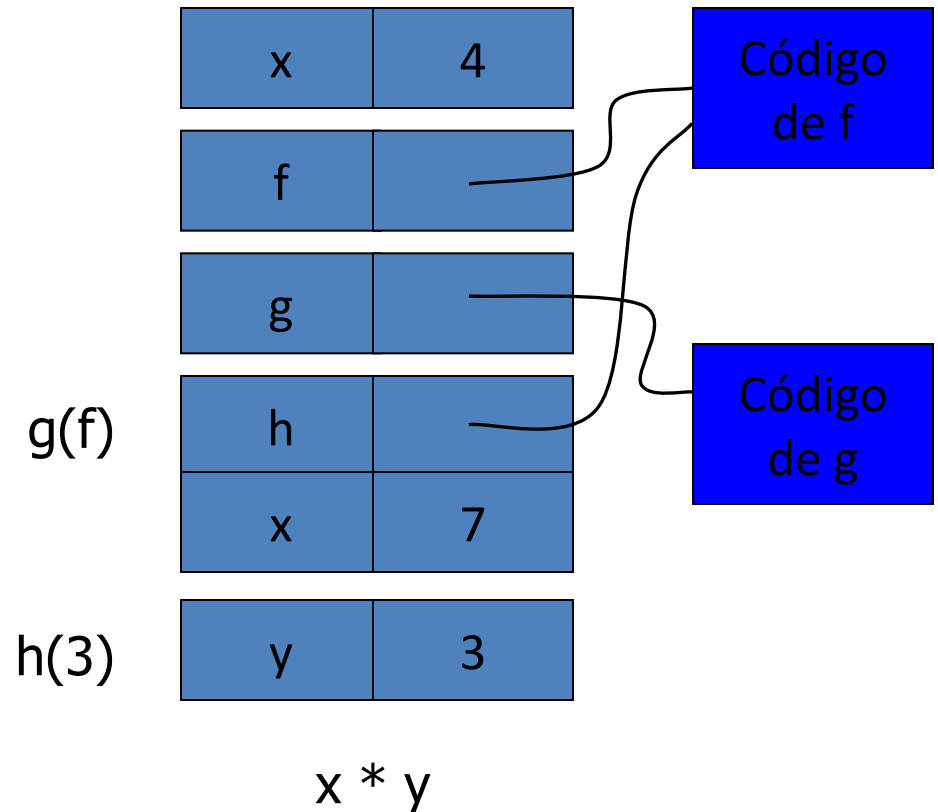
ambigüedad en alcance

```
val x = 4;  
fun f(y) = x*y;  
fun g(h) =  
  let  
    val x=7  
  in  
    h(3) + x;  
  g(f);
```



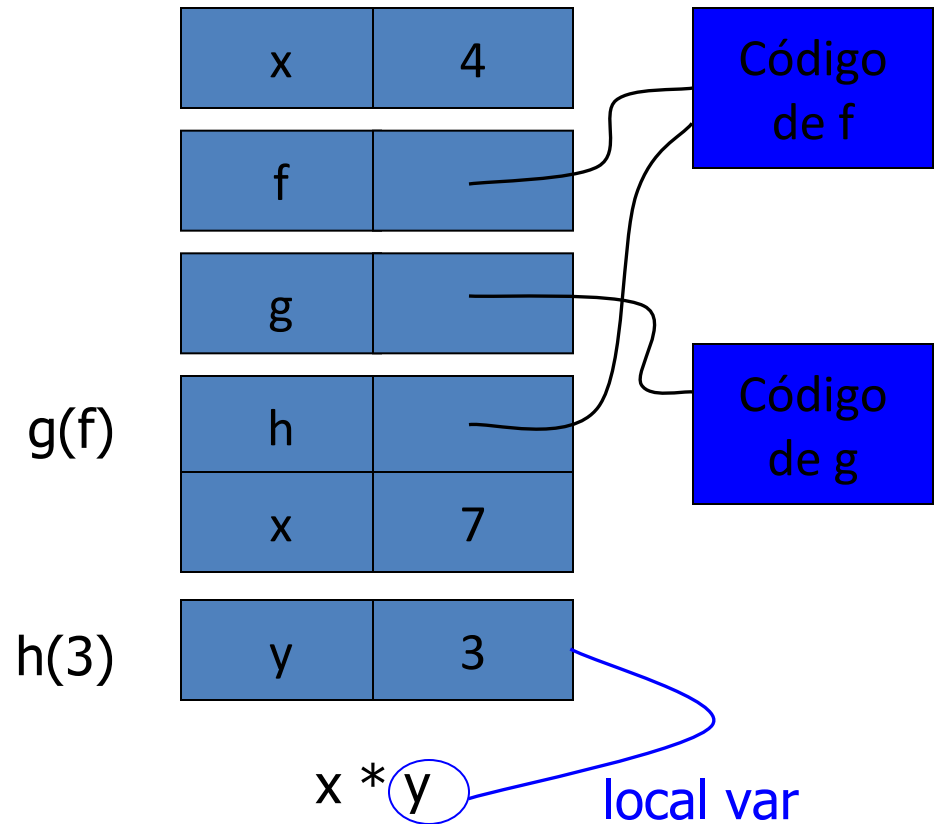
ambigüedad en alcance

```
val x = 4;  
fun f(y) = x*y;  
fun g(h) =  
  let  
    val x=7  
  in  
    h(3) + x;  
  g(f);
```

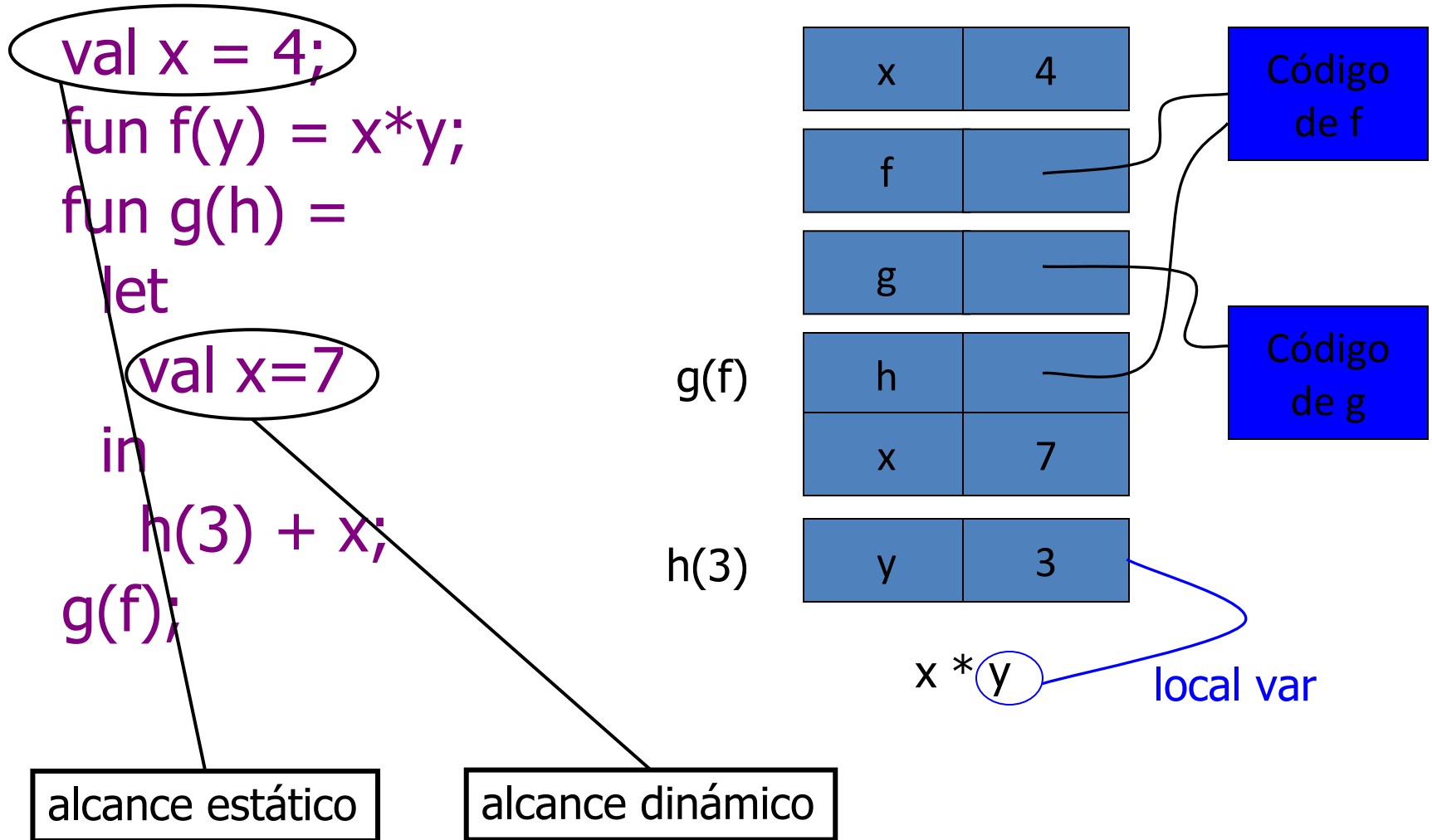


ambigüedad en alcance

```
val x = 4;  
fun f(y) = x*y;  
fun g(h) =  
  let  
    val x=7  
  in  
    h(3) + x;  
  g(f);
```

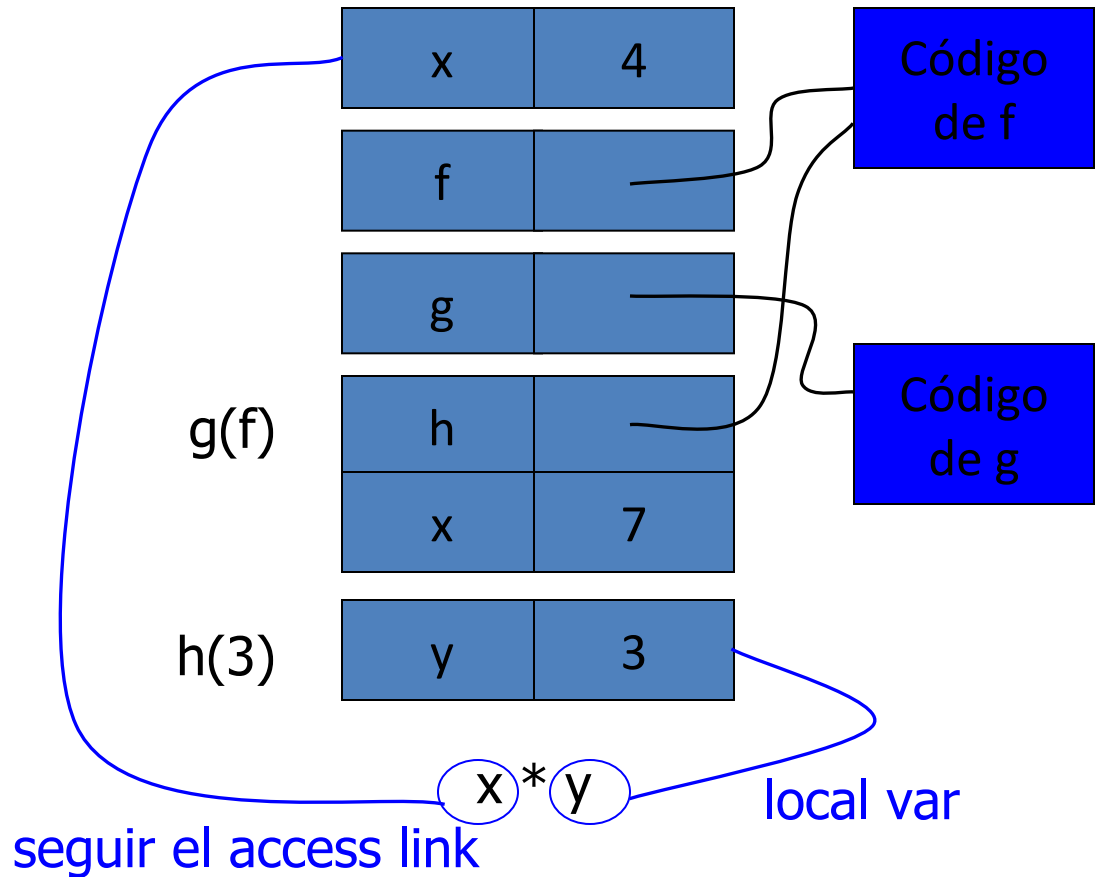


ambigüedad en alcance

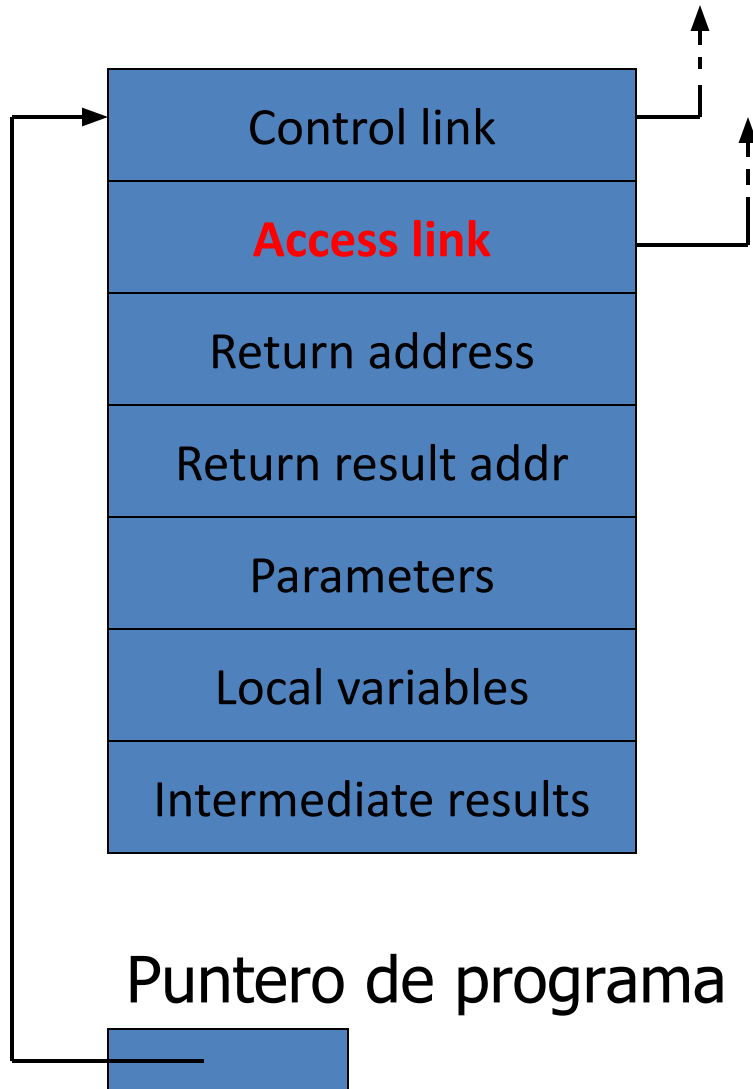


ambigüedad en alcance

```
val x = 4;  
fun f(y) = x*y;  
fun g(h) =  
  let  
    val x=7  
  in  
    h(3) + x;  
  g(f);
```



activation record para alcance estático



- Control link
 - link al activation record del bloque anterior (el que llama al actual)
 - depende del comportamiento dinámico del programa
- **Access link**
 - link al activation record del bloque que incluye de más cerca al actual, léxicamente, en el texto del programa
 - depende del texto estático del programa

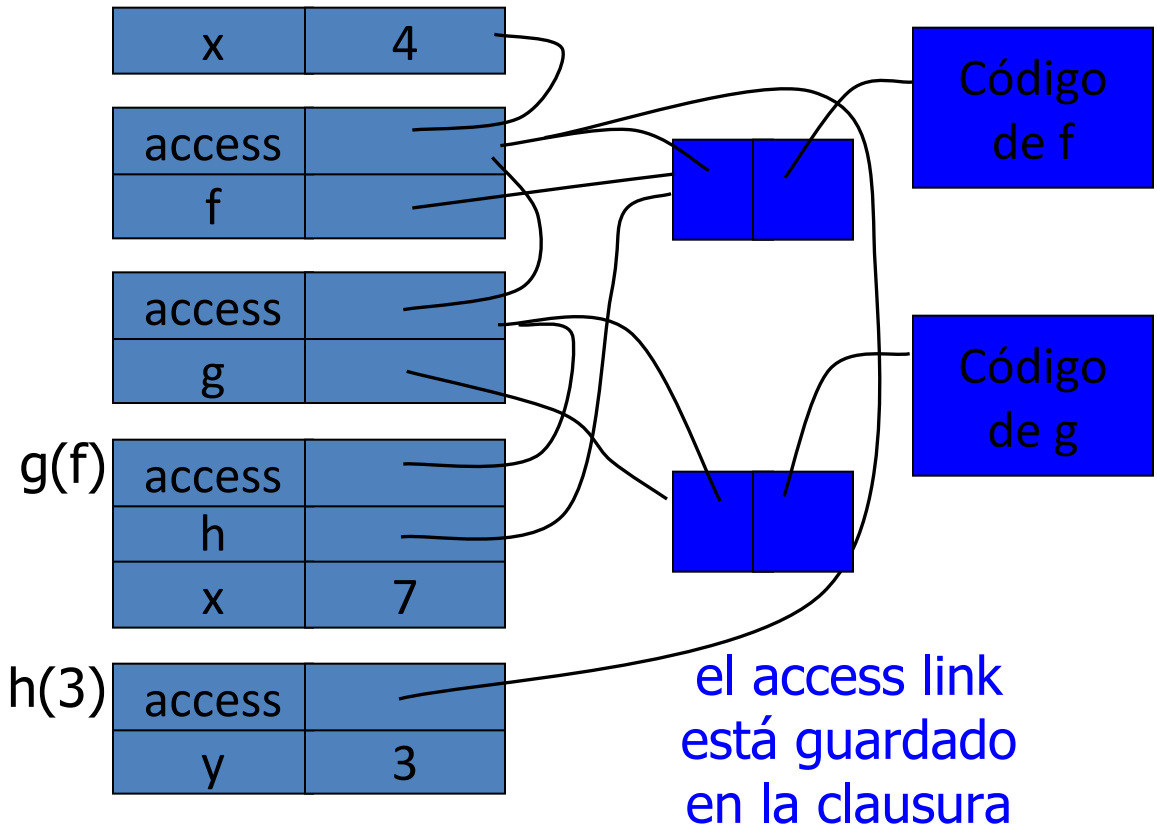
Clausuras (estáticas)

- el valor de una función es el par **clausura** = $\langle \text{entorno, código} \rangle$
 - la idea es que una función con alcance estático lleva un link a su environment estático
- llamada a una función con clausura
 - alojar el activation record para la llamada
 - fijar el access link del activation record usando el puntero de entorno de la clausura

alcance estático con clausura

pila de ejecución con clausuras

```
- val x = 4;  
- fun f(y) = x*y;  
- fun g(h) =  
  let  
    val x=7  
  in  
    h(3) + x;  
- g(f);
```



funciones de alto orden

- una función puede ser argumento o resultado de otra función
 - se necesita un puntero al registro de activación más arriba en la pila
 - pueden surgir problemas especialmente al pasar una función como argumento...

devolver funciones como resultado

- no todos los lenguajes tienen esta posibilidad
- funciones que devuelven nuevas funciones

```
fun compose(f,g) = (fn x => g(f x));
```

- se pueden crear funciones de forma dinámica, con valores instanciados en tiempo de ejecución (lo que ustedes conocían como generalizaciones)
- el valor de una función es la clausura = $\langle \text{env}, \text{código} \rangle$
- el código no se compila dinámicamente en casi ningún lenguaje
- necesitamos mantener el entorno de la función que generó la función dinámica

clausuras en Programación Web

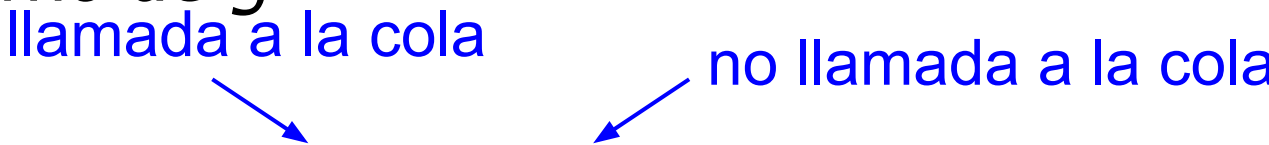
- Útil para manejar eventos

```
function AppendButton(container, name, message) {  
    var btn = document.createElement('button');  
    btn.innerHTML = name;  
    btn.onclick = function(evt) { alert(message); }  
    container.appendChild(btn);  
}
```

- El puntero de entorno le dice al handler del clic del botón qué mensaje mostrar

- pasaje de parámetros
- alcance
- clausuras
- recursión a la cola

recursión a la cola (caso de primer orden)

- la función g hace una **llamada a la cola** a la función f si *el valor de retorno de la función f es el valor de retorno de g*
- ejemplo


```
fun g(x) = if x>0 then f(x) else f(x)*2
```
- optimización: se puede desapilar el activation record actual en una llamada a la cola
 - especialmente útil para llamadas a la cola recursivas porque el siguiente activation record tiene exactamente la misma forma

cómo convertir una función recursiva en iterativa

```
def fact(n):  
    if n == 0:  
        return 1  
    return n * fact(n-1)
```

```
def fact_h(n, acc):  
    if n == 0:  
        return acc  
    return fact_h(n-1, acc*n)
```

```
def fact(n):  
    return fact_h(n, 1)
```

cómo convertir una función recursiva en iterativa

```
function foo(x) is:  
  if predicate(x) then  
    return foo(bar(x))  
  else  
    return baz(x)
```

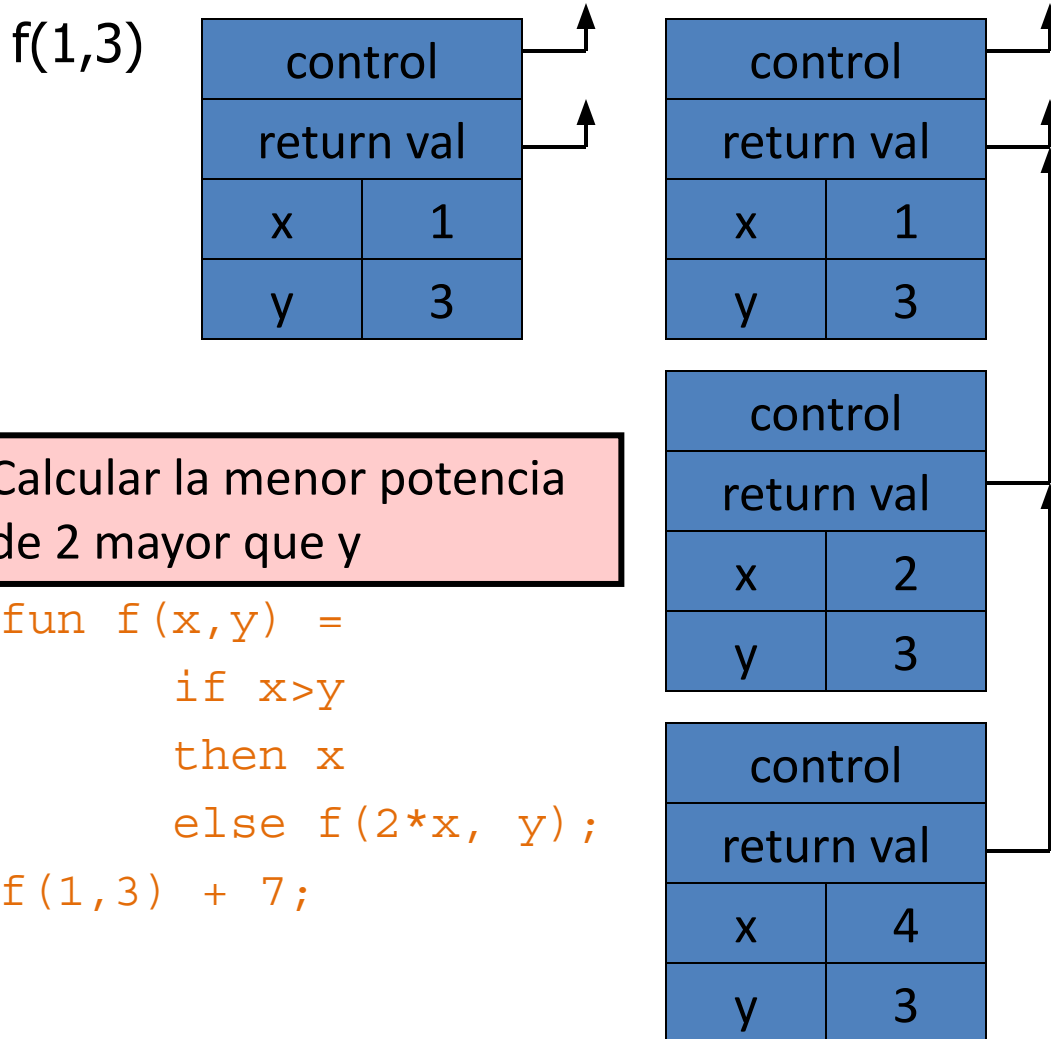
```
function foo(x) is:  
  while predicate(x) do:  
    x ← bar(x)  
  return baz(x)
```

optimización del compilador

```
foo:
  mov reg
  push reg
  call B
  pop
  mov reg
  push reg
  call A
  pop
  ret
```

<pre>foo: mov reg,[sp+data1] ; poner data1 del parámetro (sp) del stack a un registro push reg ; poner data1 en el stack, donde B lo espera call B ; B usa data1 pop ; eliminar data1 del stack mov reg,[sp+data2] ; poner data2 del parámetro (sp) del stack a un registro mov [sp+data1],reg ; poner data2 en el stack, donde A lo espera jmp A ; A usa data2 y retorna a la función que llama</pre>
--

ejemplo de recursión



Optimización: fijar la dirección de retorno a la de la función de llamada

- se puede hacer lo mismo con el control link?

Optimización: evitar el retorno al que llama

- funciona con el alcance dinámico?

el problema de la recursión

```
call factorial (3)
  call fact (3 1)
    call fact (2 3)
      call fact (1 6)
        call fact (0 6)
          return 6
        return 6
      return 6
    return 6
  return 6
return 6
```

```
call factorial (3)
  call fact (3 1)
    reemplazar argumentos por (2 3)
    reemplazar argumentos por (1 6)
    reemplazar argumentos por (0 6)
    return 6
  return 6
return 6
```

eliminación de recursión a la cola

f(1,3)

control		↑
return val		↑
x	1	
y	3	

f(2,3)

control		↑
return val		↑
x	2	
y	3	

f(4,3)

control		↑
return val		↑
x	4	
y	3	

Optimización:

```
fun f(x,y) =  
  if x>y  
  then x  
  else f(2*x, y);  
f(1,3) + 7;
```

- pop seguido de push – se ocupa el mismo lugar de activation record en la pila
- se convierte la función recursiva en un ciclo iterativo

recursión a la cola e iteración

$f(1,3)$

control		↑
return val		↑
x	1	
y	3	

$f(2,3)$

control		↑
return val		↑
x	2	
y	3	

$f(4,3)$

control		↑
return val		↑
x	4	
y	3	

```
fun f(x,y) = if x>y test
  then x
  else f(2*x, y);
f(1,y);
```

valor inicial

cuerpo del ciclo

```
function g(y) {
  var x = 1;
  while (!x>y)
    x = 2*x;
  return x;
}
```