

detalles de manejo de memoria en la abstracción procedural

Paradigmas de la Programación

FaMAF 2015

capítulo 7.

(adicionales: 4.4. y 5.)

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

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

reglas de alcance

variables locales y globales

x, y son locales al bloque exterior

z es local al bloque interior

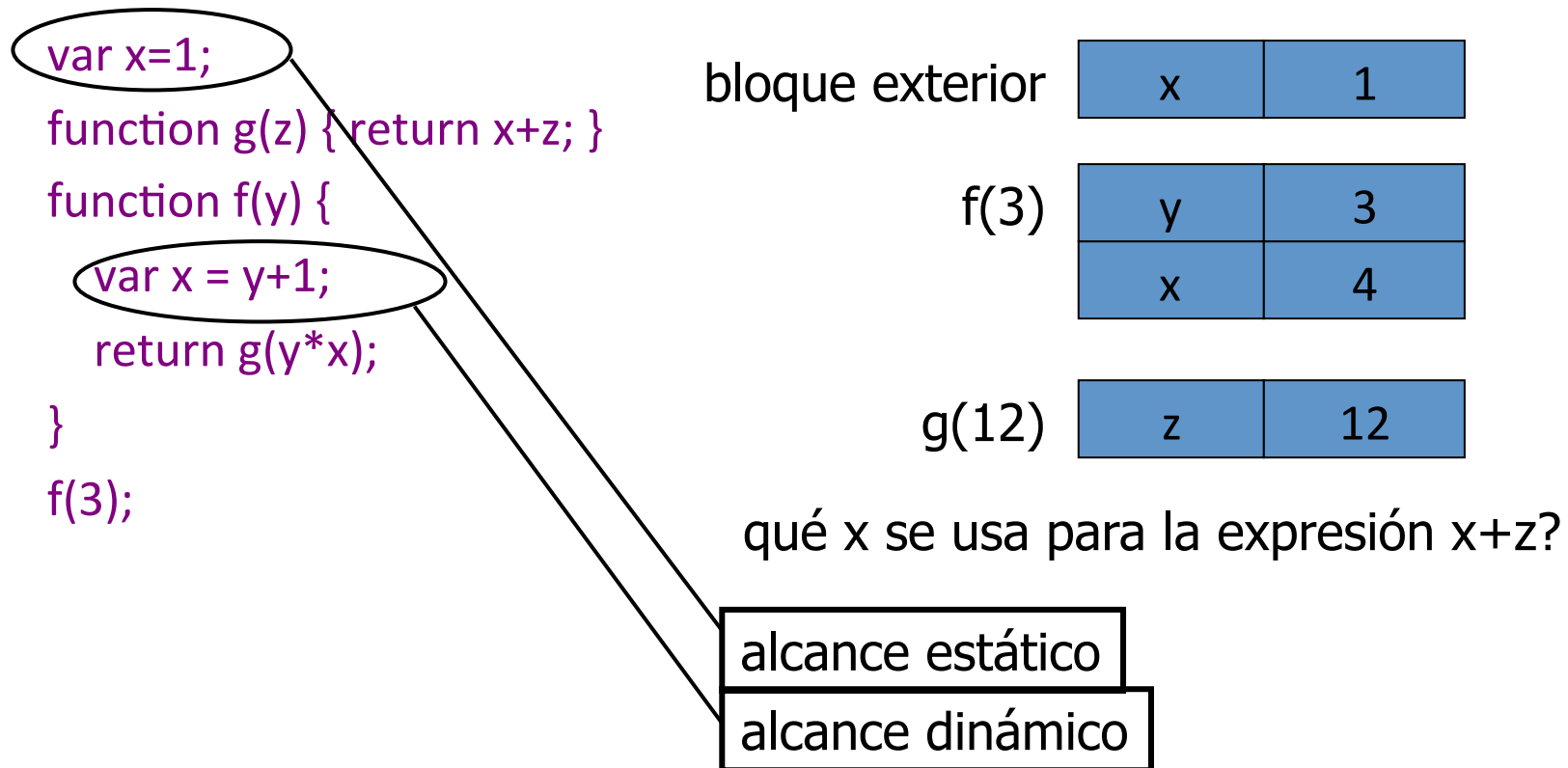
x, y son globales al bloque interior

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

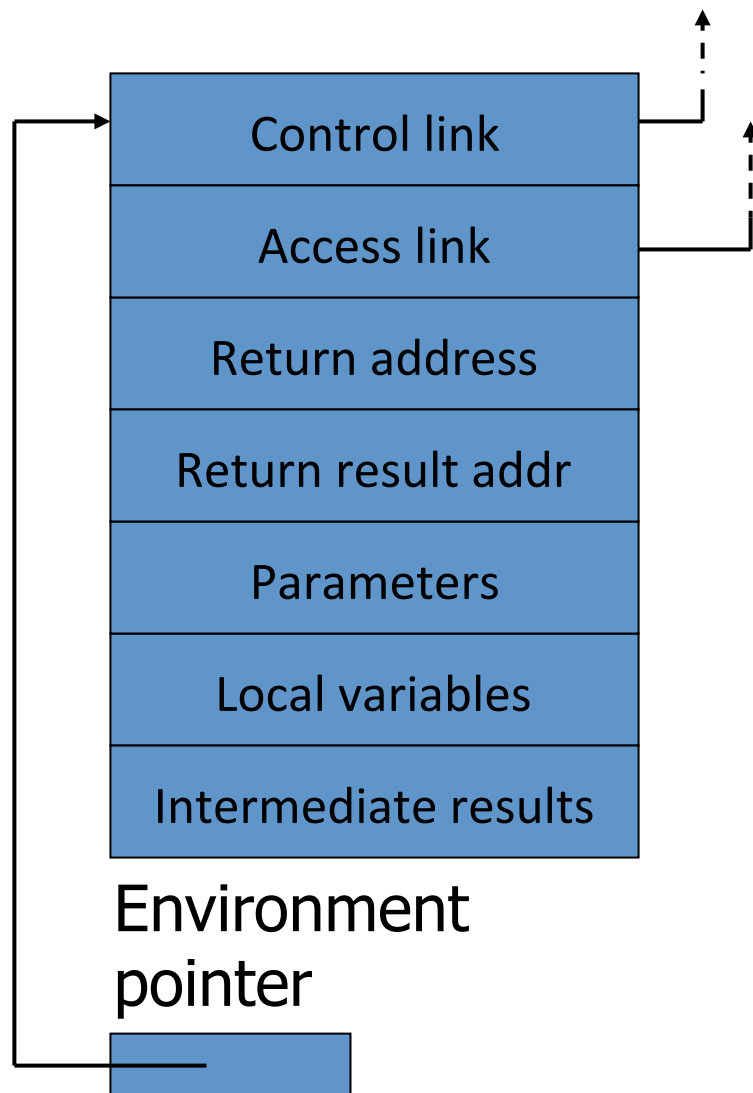
alcance estático: el valor de las variables globales se obtiene del bloque inmediatamente contenedor

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

alcance estático vs. dinámico



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
 - **en C no se usa!**
 - depende del texto estático del programa

Complex Nesting Structure

```
function m(...) {  
  var x=1;  
  ...  
  function n( ... ){  
    function g(z) { return x+z; }  
    ...  
    { ...  
      function f(y) {  
        var x = y+1;  
        return g(y*x); }  
      ...  
      f(3); ... }  
    ... n( ... ) ...}  
  ... m(...)
```



Simplify to

```
var x=1;  
function g(z) { return x+z; }  
function f(y)  
{ var x = y+1;  
  return g(y*x); }  
f(3);
```

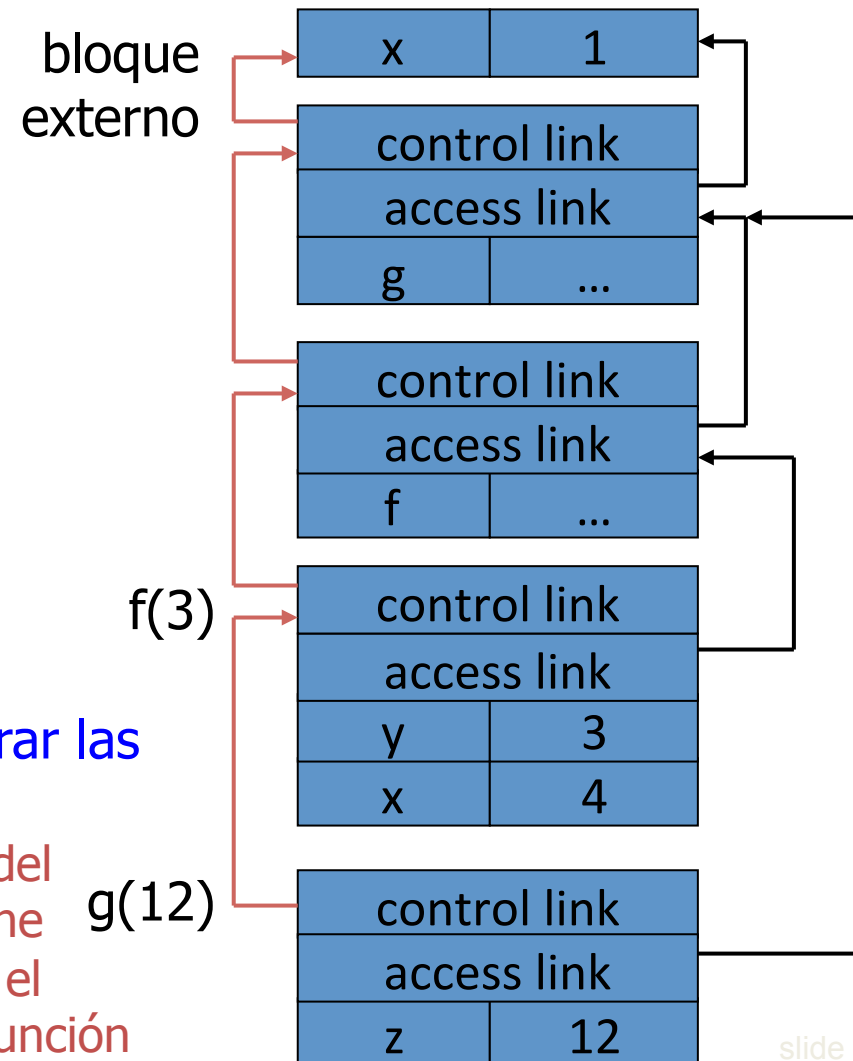
Simplified code has same block nesting,
if we follow convention that each
declaration begins a new block

alcance estático con Access Links

```
var x=1;  
function g(z) = { return x+z; }  
function f(y) =  
  { var x = y+1;  
    return g(y*x); }  
f(3);
```

se usan los access links para encontrar las variables globales:

- el access link siempre se fija al marco del bloque léxico más cercano que lo contiene
- para el cuerpo de una función, este es el bloque que contiene la definición de la función



Variable Arguments (Redux)

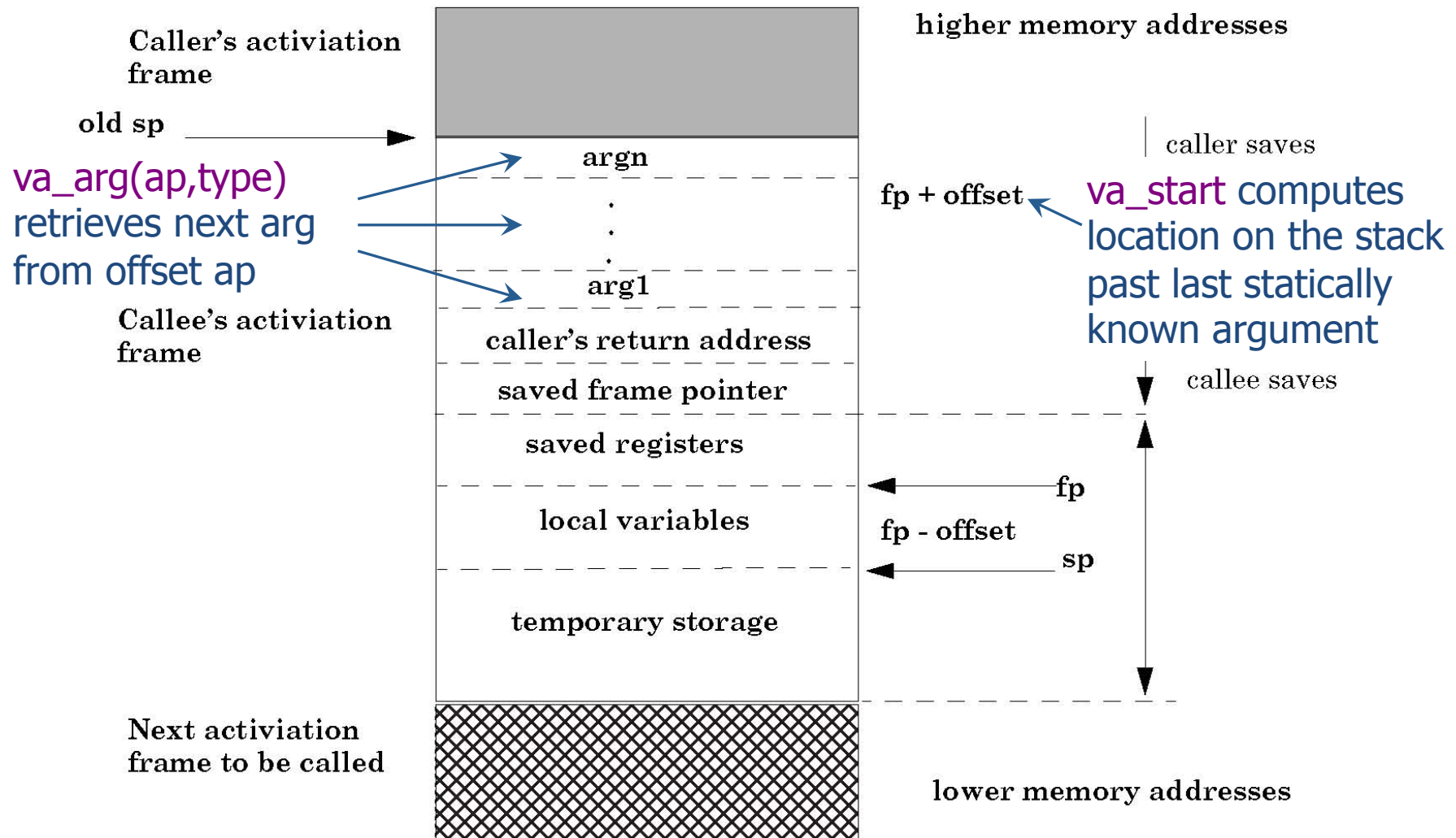
- Special functions `va_start`, `va_arg`, `va_end` compute arguments at run-time (how?)

```
void printf(const char* format, ...)
{
    int i; char c; char* s; double d;
    va_list ap; /* declare an "argument pointer" to a variable arg list */
    va_start(ap, format); /* initialize arg pointer using last known arg */

    for (char* p = format; *p != '\\0'; p++) {
        if (*p == '%') {
            switch (*++p) {
                case 'd':
                    i = va_arg(ap, int); break;
                case 's':
                    s = va_arg(ap, char*); break;
                case 'c':
                    c = va_arg(ap, char); break;
            }
            ... /* etc. for each % specification */
        }
    }
    ...

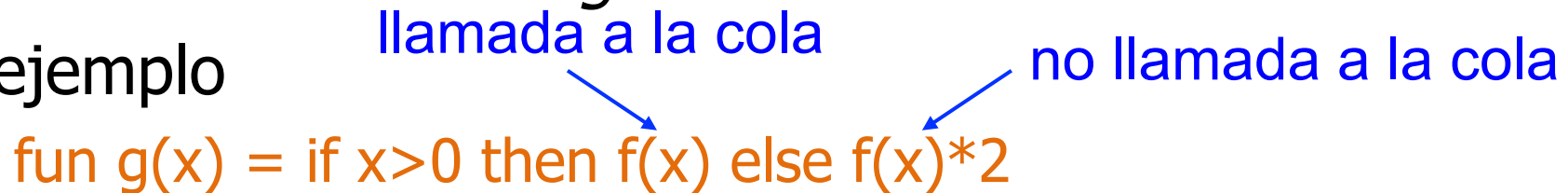
    va_end(ap); /* restore any special stack manipulations */
}
```


Activation Record for Variable Args



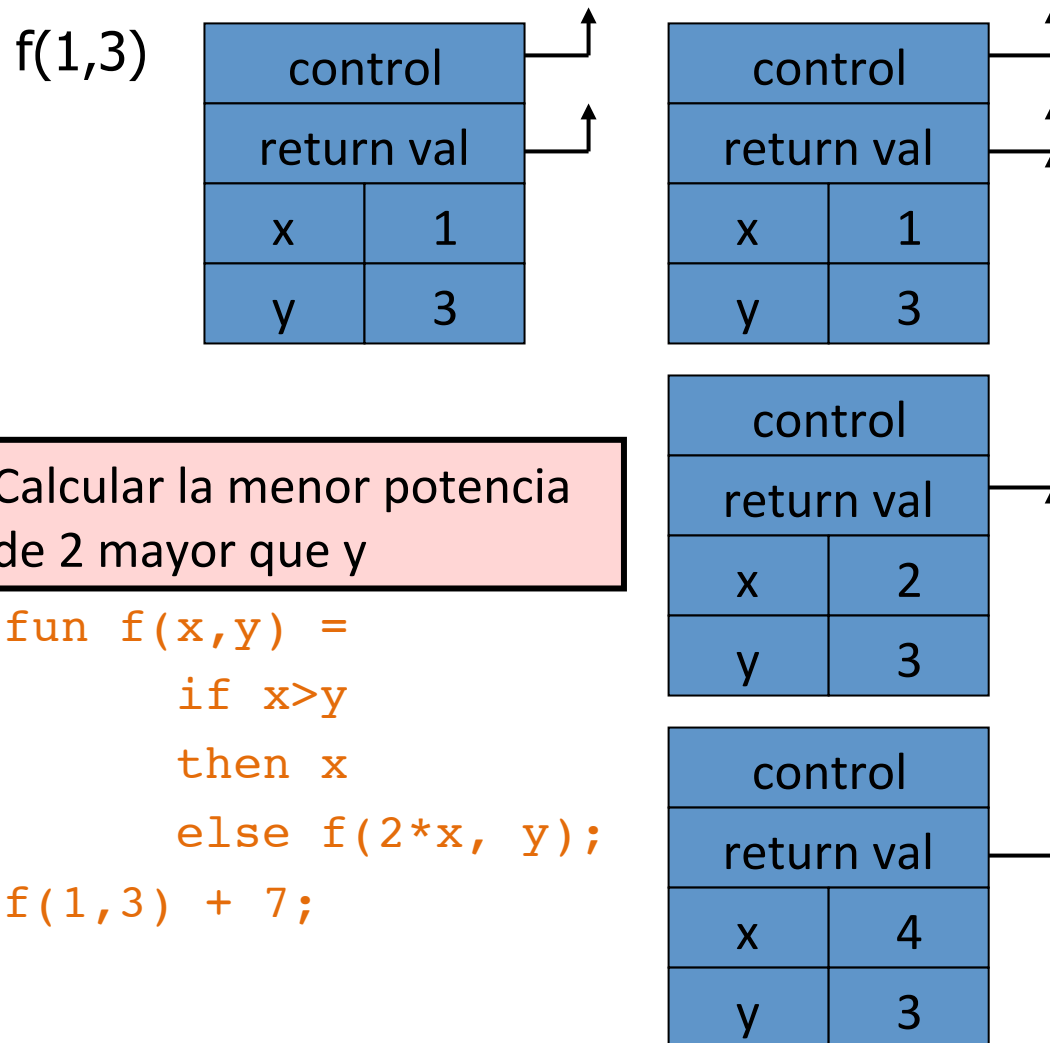
- alcance
- recursión a la cola
- clausuras
- pasaje de parámetros

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

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
```

```
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
```

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;
}
```

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,[sp+data1] ; foo:

push reg ; poner data1 del parámetro (sp) del stack a un reg

call B ; B usa data1 ; poner data1 en el stack, donde B lo espera

pop ; eliminar data1 del stack ; B usa data1

mov reg,[sp+data2] ; eliminar data1 del stack

push reg ; poner data2 del parámetro (sp) del stack a un reg

call A ; A usa data2 ; poner data2 en el stack, donde A lo espera

pop ; eliminar data2 del stack ; A usa data2 y retorna a la función que llama

ret

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

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...

pasar una función como argumento

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

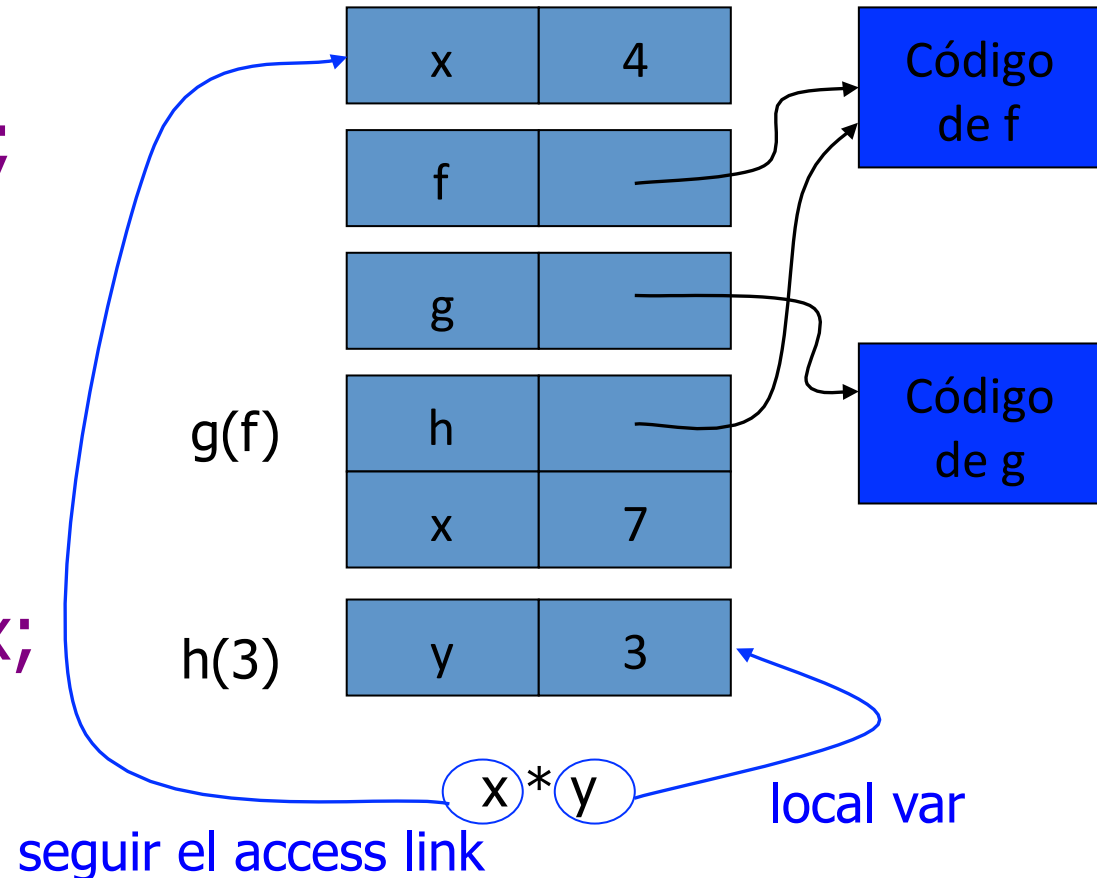
qué declaración de x se usa en cada ocurrencia de x?

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
 - sólo se necesita si la función se define inline, en un bloque anidado (por qué?)
- 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 para los argumentos

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

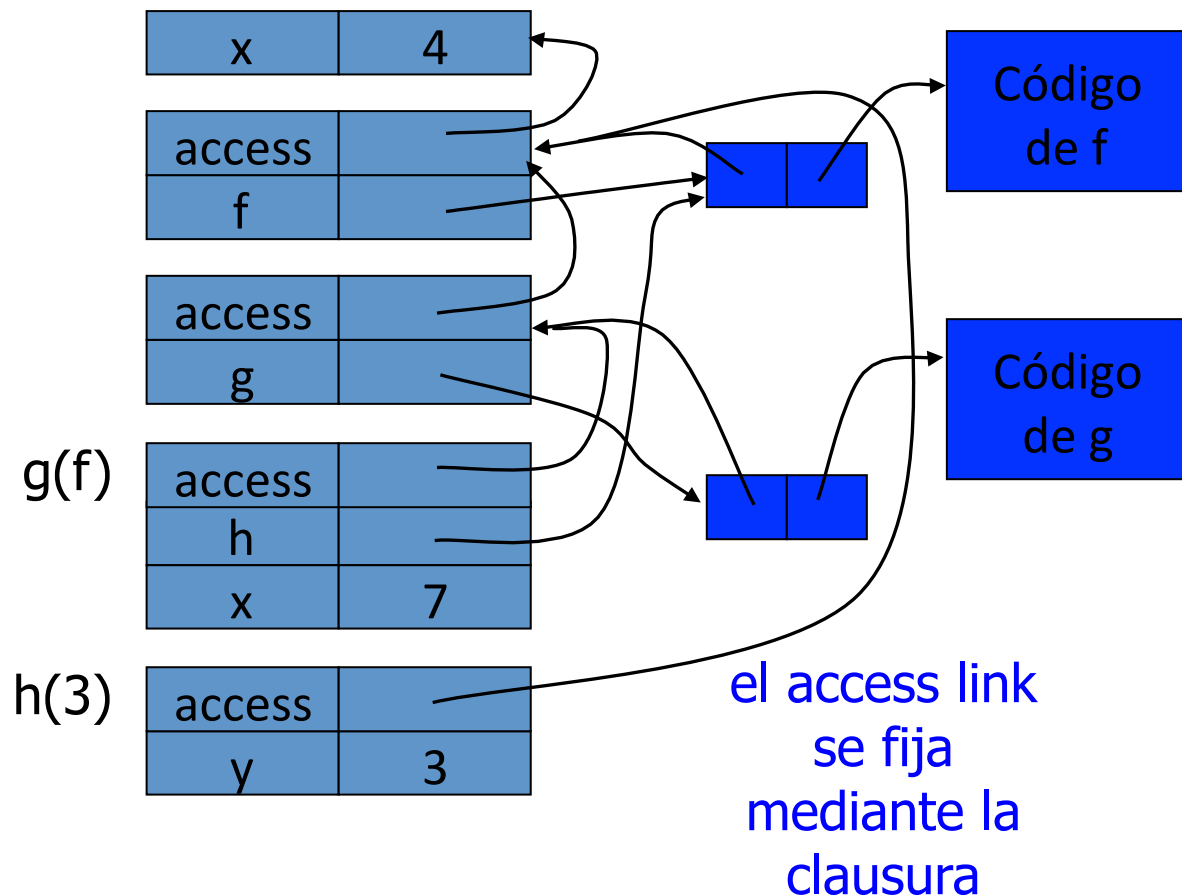


cómo se fija el access link para $h(3)$?

funciones como argumentos y clausuras

pila de ejecución con access links

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



Activation of a Scheme Function (access links): Example 4

```
(define M (lambda (j k)

  (define P (lambda (x y z)

    (define Q (lambda ()

      (define R (lambda ()
        (P j k z)))          ; end R

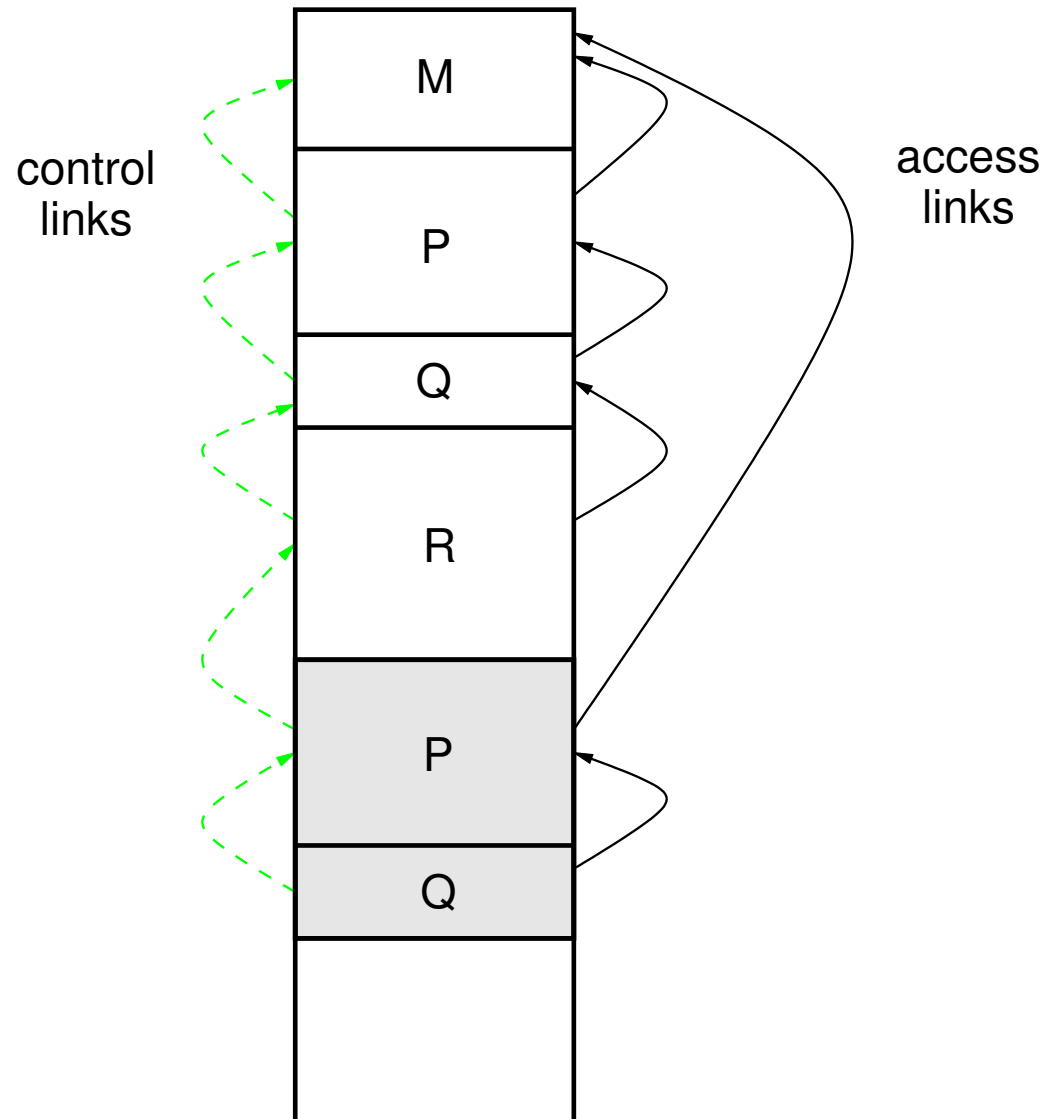
      (* (R) y)))          ; end Q

      (+ (Q) x)))          ; end P

    (P j k 2)))          ; end M
```

Activation of a Scheme Function (access links): Example 4

..



resumen de funciones como argumentos

- usar la clausura para mantener un puntero al entorno estático del cuerpo de una función
- en la llamada a la función, el access link se determina desde la clausura
- todos los access links apuntan para “arriba” en la pila
 - pueden saltar muy arriba, saltando varios activation records, para encontrar variables globales
 - sigue siendo válida la estrategia de desalojar activation records usando orden de pila (primero adentro – último afuera)

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 (por qué?)

devolver funciones con estado privado

```
fun mk_counter (init : int) =  
  let val count = ref init  
      fun counter(inc:int) =  
        (count := !count + inc; !count)  
      in  
        counter  
      end;  
  val c = mk_counter(1);  
  c(2) + c(2);
```

- Function to “make counter” returns a closure
- How is correct value of `count` determined in `c(2)` ?

Function Results and Closures

```
fun mk_counter (init : int) =  
  let val count = ref init  
      fun counter(inc:int) =  
        (count := !count + inc; !count)  
      in  
        counter  
      end;
```

```
val c = mk_counter(1);  
c(2) + c(2);
```

Call changes cell
value from 1 to 3

mk_counter(1)

c(2)

mk_c	
access	
c	

access	
init	1
count	
counter	

access	
inc	2

Code for
mk_counter

3

Code for
counter

Closures in Web Programming

- Useful for event handlers

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

- Environment pointer lets the button's click handler find the message to display

Managing Closures

- Closures as used to maintain static environment of functions as they are passed around
- May need to keep activation records after function returns (why?)
 - Stack (last-in-first-out) order fails! (why?)
- Possible “stack” implementation:
 - Put activation records on heap
 - Instead of explicit deallocation, invoke garbage collector as needed
 - Not as totally crazy as it sounds (may only need to search reachable data)

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

System Calls

- OS procedures often return status codes
 - Not the result of computing some function, but an indicator of whether the procedure succeeded or failed to cause a certain side effect

```
int open(const char* file, int mode)
{
    if (file == NULL) {
        return -1; // invalid file name

    if (open(file, mode) < 0)
        return -2; // system open failed
    ...
}
```

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

pasaje por valor

- la función que llama pasa el **r-valor** del argumento a la función
 - es necesario computar el valor del argumento en el momento de la llamada
 - Reduce el “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
 - se asigna la dirección de memoria del argumento al parámetro
 - aumenta aliasing
- 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);
}
```


ejemplo en ML

pseudo-código

```
function f (x) =  
  { x = x+1; return x; }  
var y = 0;  
print (f(y)+y);
```

*pasaje por
referencia*



Standard ML

```
fun f (x : int ref) =  
  ( x := !x+1; !x );  
y = ref 0 : int ref;  
f(y) + !y;
```

pasaje por valor



```
fun f (z : int) =  
  let x = ref z in  
    x := !x+1; !x  
  end;  
y = ref 0 : int ref;  
f(!y) + !y;
```

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
(por qué es importante?)

- 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);
```

cuál es mejor? por qué?

pasaje por valor-resultado

- Intenta tener los 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 argumentos actuales al final del procedimiento. Así los argumentos son modificados.
- Se comporta como llamada por referencia sin la presencia de aliasing
- Cuidado: el comportamiento depende del orden en que las variables locales se copian.
- 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?

```
declare A=3 B=4 C=1
```

```
proc {P X Y Z}
```

```
  C=5
```

```
  X=Z
```

```
  C=4
```

```
  Y=Z+A
```

```
end
```

```
in {P A B A+C}
```

```
  {Browse B}
```

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

2
4
6
7
8
10
12
16
20

Cuál es la salida?

```
declare A=3 B=4 C=1
```

```
proc {P X Y Z}
```

```
  C=5
```

```
  X=Z
```

```
  C=4
```

```
  Y=Z+A
```

```
end
```

```
in {P A B A+C}
```

```
  {Browse B}
```

- Call by value? 4
- Call by reference? 8
- Call by value-result? 7
- Call by name? 20
- Call by need? 16

Jensen's Device

- Computes $\sum_{i=1}^{100} \frac{1}{i}$ in Algol 60

```
begin
  integer i;
  real procedure sum (i, lo, hi, term);
    value lo, hi;
    integer i, lo, hi;
    real term;
    begin
      real temp;
      temp := 0;
      for i := lo step 1 until hi do
        temp := temp + term;
      sum := temp
    end;
  print (sum (i, 1, 100, 1/i))
end
```

passed by name

becomes 1/i when sum is executed

Macro

- Textual substitution

```
#define swap(a,b) temp=a; a=b; b=temp;
```

```
...
```

```
int x=3, y=4;
```

```
int temp;
```

```
swap(x,y);
```

Textually expands to
temp=x; x=y; y=temp;



- Looks like a function definition, but ...
 - Does not obey the lexical scope rules (i.e., visibility of variable declarations)
 - No type information for arguments or result

Problems with Macro Expansion

```
#define swap(a,b) temp=a; a=b; b=temp;
```

... Textually expands to

```
if (x<y)      if (x<y)
    swap(x,y);      temp=x;
                   x=y;
                   y=temp;
```

Why not `#define swap(a,b) { int temp=a; a=b; b=temp; }`?

```
Instead #define swap(a,b) do { Fixes type of swapped variables  
    int temp=a; a=b; b=temp;  
} while(false);
```

Variable Arguments

- In C, can define a function with a variable number of arguments

– Example: `void printf(const char* format, ...)`

Part of
syntax!

- Examples of usage:

```
printf("hello, world");  
printf("length of '%s' = %d\n", str, str.length());  
printf("unable to open file descriptor %d\n", fd);
```

Format specification encoded by
special %-encoded characters

`%d,%i,%o,%u,%x,%X` – integer argument
`%s` – string argument
`%p` – pointer argument (void *)
Several others (see C Reference Manual!)

Implementation of Variable Args

- Special functions `va_start`, `va_arg`, `va_end` compute arguments at run-time (how?)

```
void printf(const char* format, ...)
{
    int i; char c; char* s; double d;
    va_list ap; /* declare an "argument pointer" to a variable arg list */
    va_start(ap, format); /* initialize arg pointer using last known arg */

    for (char* p = format; *p != '\\0'; p++) {
        if (*p == '%') {
            switch (*++p) {
                case 'd':
                    i = va_arg(ap, int); break;
                case 's':
                    s = va_arg(ap, char*); break;
                case 'c':
                    c = va_arg(ap, char); break;
            }
            ... /* etc. for each % specification */
        }
    }
    ...

    va_end(ap); /* restore any special stack manipulations */
}
```