

# Âmbito, funções e gestão de memória

**Linguagens de Programação  
2019.2020**

*Teresa Gonçalves*  
[tcg@uevora.pt](mailto:tcg@uevora.pt)

Departamento de Informática, ECT-UÉ

# Sumário

**Linguagens estruturadas em blocos**

**Blocos em linha**

**Funções**

**Funções de ordem superior**

# Linguagens estruturadas em blocos

# Bloco

## O que é?

Região de texto do programa

## Como se identifica?

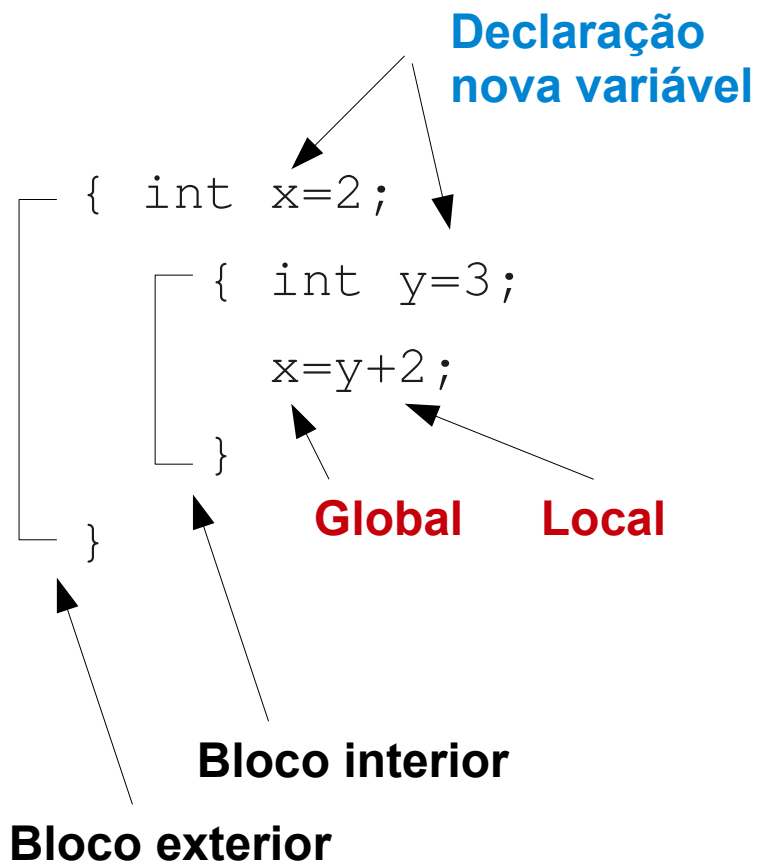
Marcadores de início e fim

## O que contém?

Declarações locais à região

Instruções

## Exemplo



# Linguagens estruturadas em blocos

## Blocos em linha

C

{ ... }

Pascal

begin ... end

ML

let ... in ... end

## Funções / procedimentos

Associados à execução

# Propriedades

## Declaração de variáveis em diferentes pontos do programa

## Declaração visível numa certa região - bloco

Os blocos podem ser aninhados, mas não se sobrepõem parcialmente

## Execução das instruções dum bloco

No início é **alocada** memória para as variáveis declaradas nesse bloco

No final (alguma) essa memória pode ser libertada

## Um identificador não declarado no bloco é global

refere a entidade declarada no bloco mais próximo

# Conceitos básicos

## Âmbito

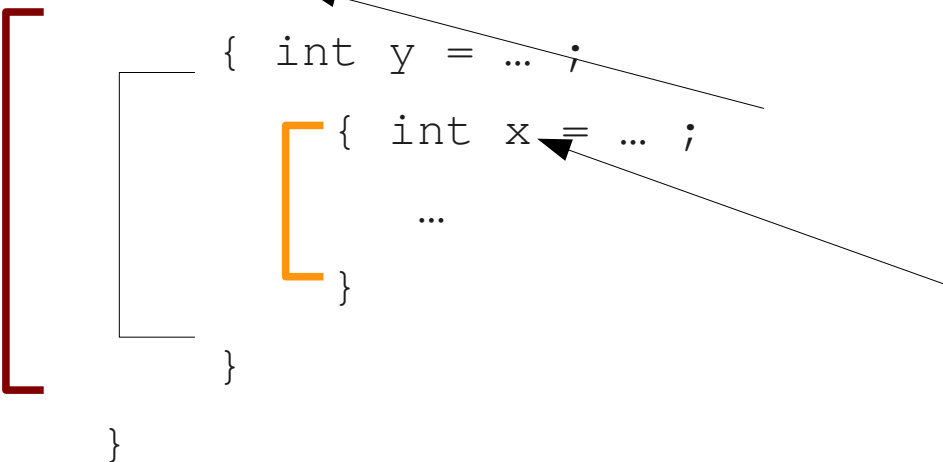
Região do texto do programa onde a declaração é visível

## Tempo de vida

Período de tempo onde a localização está acessível ao programa

## Exemplo

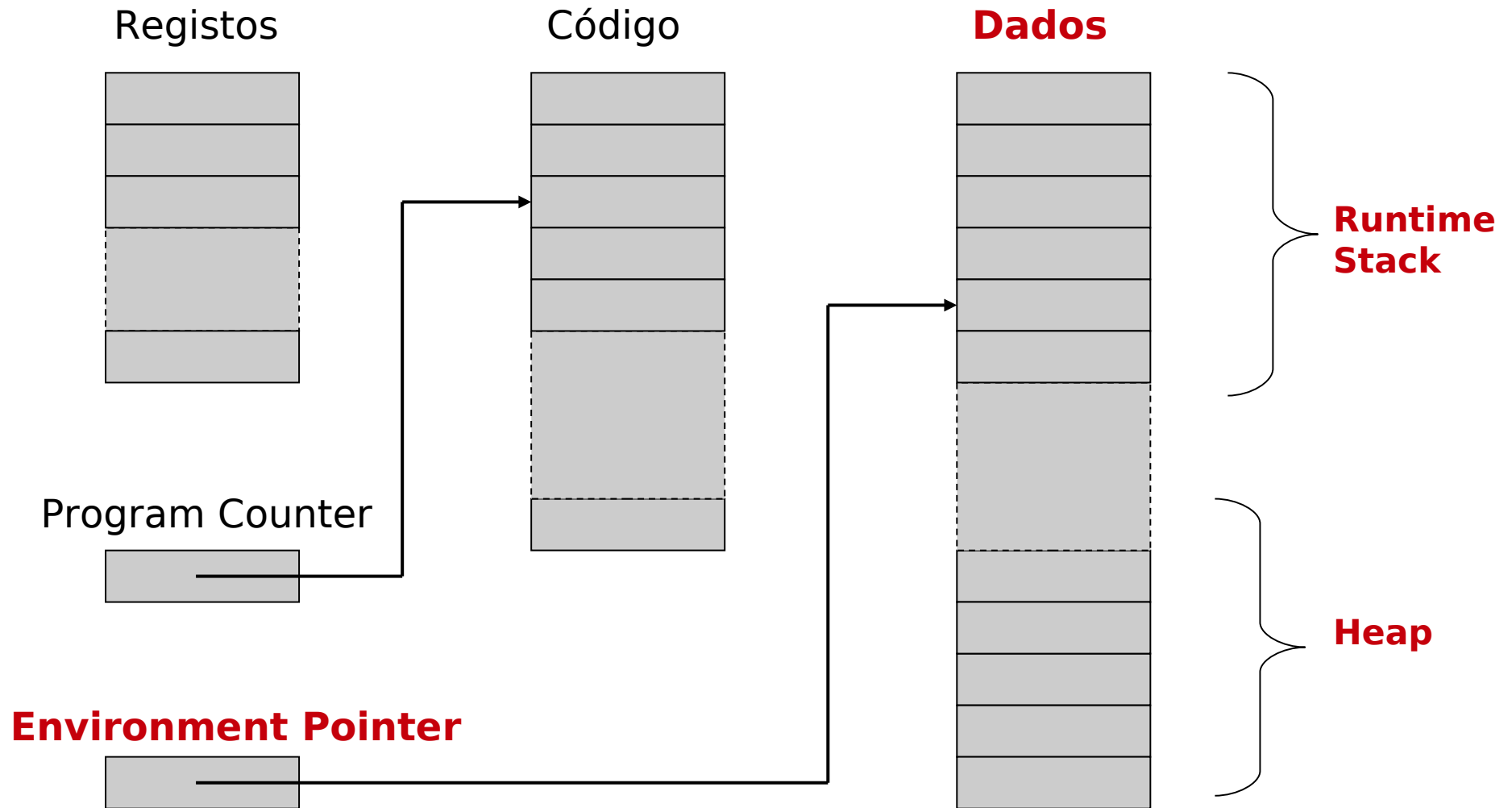
```
{ int x = ... ;  
  { int y = ... ;  
    { int x = ... ;  
      ...  
    }  
  }  
}
```



Tempo de vida de **x** **inclui** o tempo de execução do bloco interior

Esta declaração de **x** **esconde** a variável x exterior → **buraco de âmbito**

# Modelo simplificado da máquina





# Gestão de memória

## Segmento de dados

*Stack* contém informação relativa à entrada/saída dos blocos

Para cada bloco existe um registo de ativação

*Heap* contém informação com tempo de vida variável

## Apontador de ambiente

Aponta para o registo de ativação corrente

Entrada no bloco → adição de um novo registo de ativação

Saída do bloco → remoção do registo de ativação mais recente

# Variáveis e parâmetros

## **Variáveis locais**

Mantidas no registo de ativação associado ao bloco

## **Variáveis globais**

São declaradas noutro bloco → estão num registo de ativação criado anteriormente ao bloco corrente

## **Parâmetros de funções e procedimentos**

Mantidos no registo de ativação associado ao bloco

# Registo de ativação

## Estrutura de dados guardada no stack de execução

Conj. de localizações de memória para guardar informação local ao bloco

**Também designado por *stack-frame***

**A informação mantida depende do tipo de bloco**

Blocos em linha

Funções e procedimentos

Funções de ordem superior

# Variáveis locais e globais

## Variável local

Variável declarada no bloco atual

## Variável global

Variável declarada fora do bloco atual

O acesso envolve encontrar o RA “certo” no stack

## Exemplo

x e y locais no bloco exterior

z local no bloco interior

x e y globais no bloco interior

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

**Blocos em linha**

# Blocos em linha

## Registo de ativação

Espaço para variáveis locais

Espaço adicional para valores intermédios (se necessário)

## Exemplo

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

**Push** registo com espaço para x, y  
Atribui valores a x e y

**Push** registo para bloco interior  
Atribui valor a z

**Pop** registo do bloco interior

**Pop** registo do bloco exterior

# RA para blocos em linha

## Control link

Apontador para o RA anterior no stack

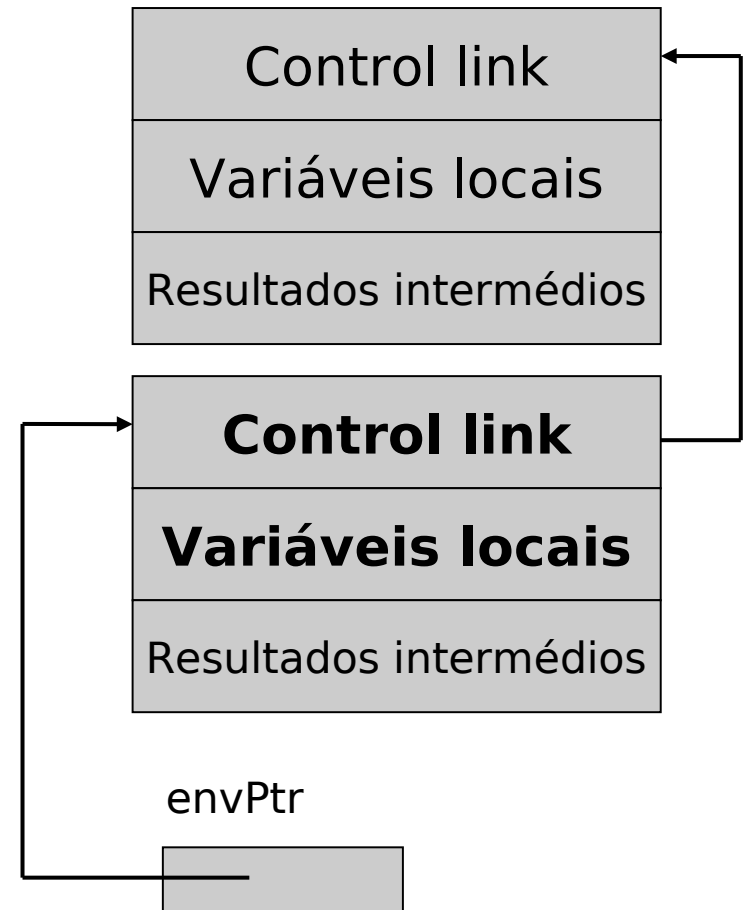
## Push

Control link corrente = EnvPtr

EnvPtr = novo registo

## Pop

EnvPtr é restabelecido utilizando o control link corrente



# Exemplo

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

**Push registo com espaço para x, y**

**Atribui valores a x e y**

Push registo para bloco interior

Atribui valor a z

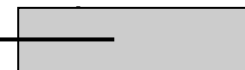
Pop registo do bloco interior

**Pop registo do bloco exterior**

Control link	
x	0
y	1

Control link	
z	-1
x+y	1
x-y	-1

Environment





# Funções e procedimentos

# Funções e procedimentos

## Sintaxe

```
<tipo> function f(<pars>)  
{  
    <vars locais>;  
    <corpo função>;  
}
```

## Registro de ativação

Endereço de retorno

Endereço para valor de retorno da função

Parâmetros

Variáveis locais

Resultados intermédios (+ valor de retorno)

# RA para funções

## End. retorno

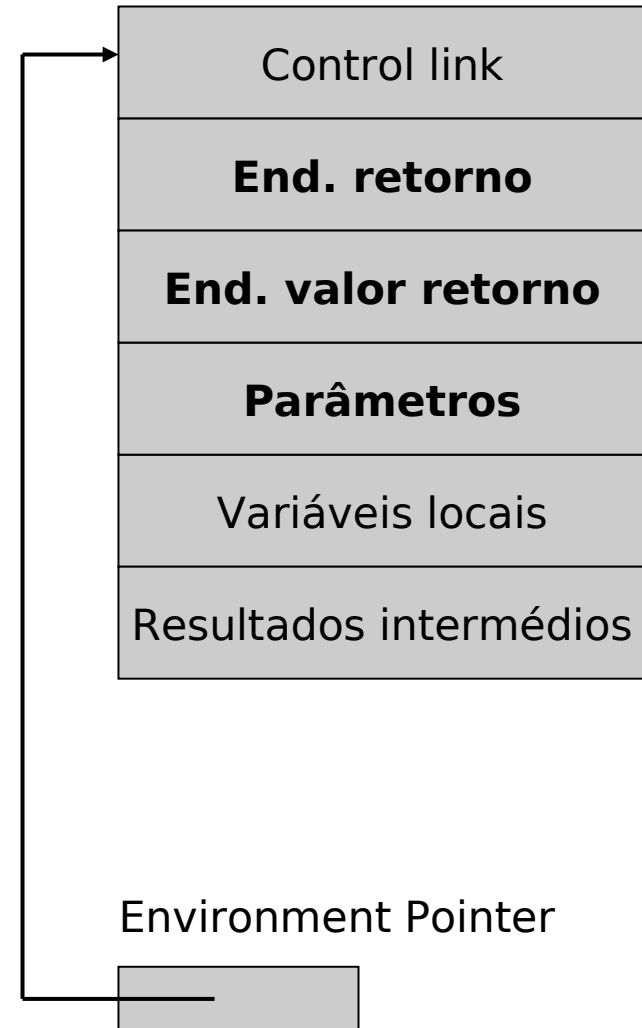
Endereço do código a executar depois do retorno da função

## End. valor retorno

Endereço no registo ativação do bloco chamador para receber valor de retorno

## Parâmetros

Posições para dados do bloco chamador



# Exemplo

## Função

```
fact(n) = if n<=1  
         then 1  
         else n*fact(n-1)
```

## End. valor retorno

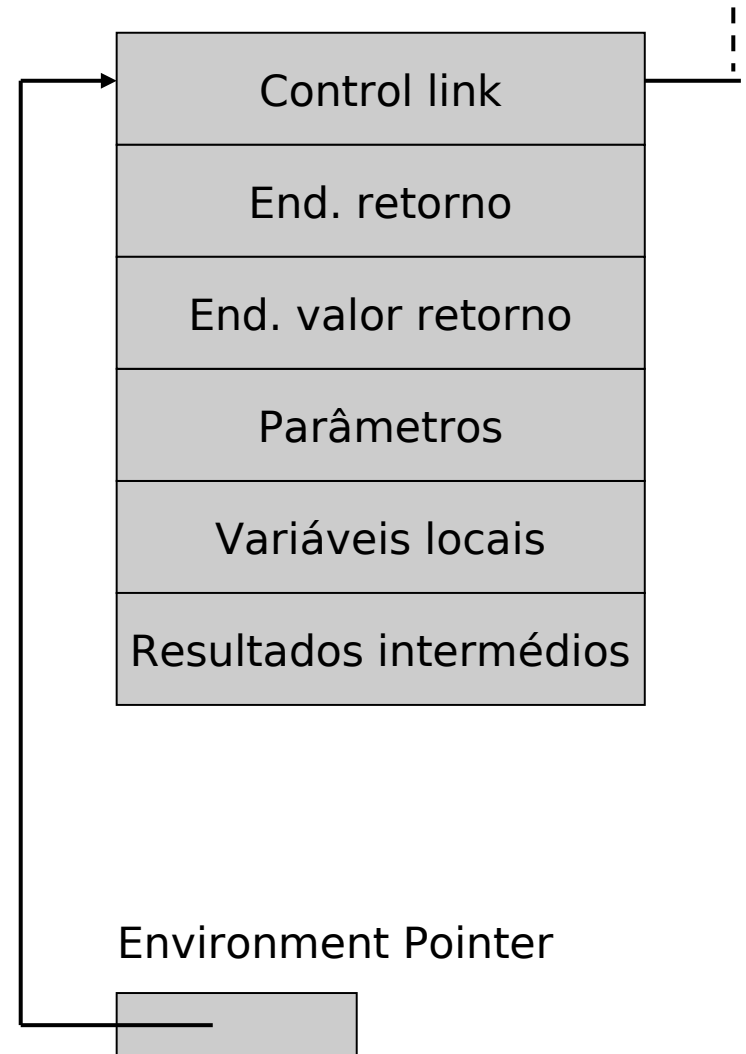
Localização para colocar fact(n)

## Parâmetros

Valor de n

## Resultado intermédio

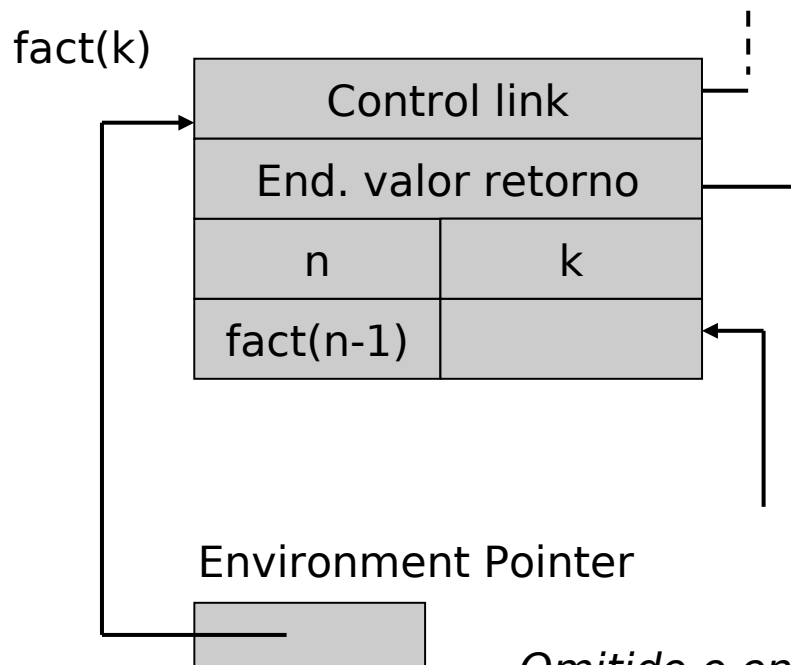
Local para fact(n-1)



# Chamada de função

1 + fact(3)

```
fact(n) = if n<=1  
then 1  
else n*fact(n-1)
```



*Omitido o endereço de retorno  
Aponta para segmento de código!*

fact(3)

Control link	
End. valor retorno	
n	3
fact(n-1)	

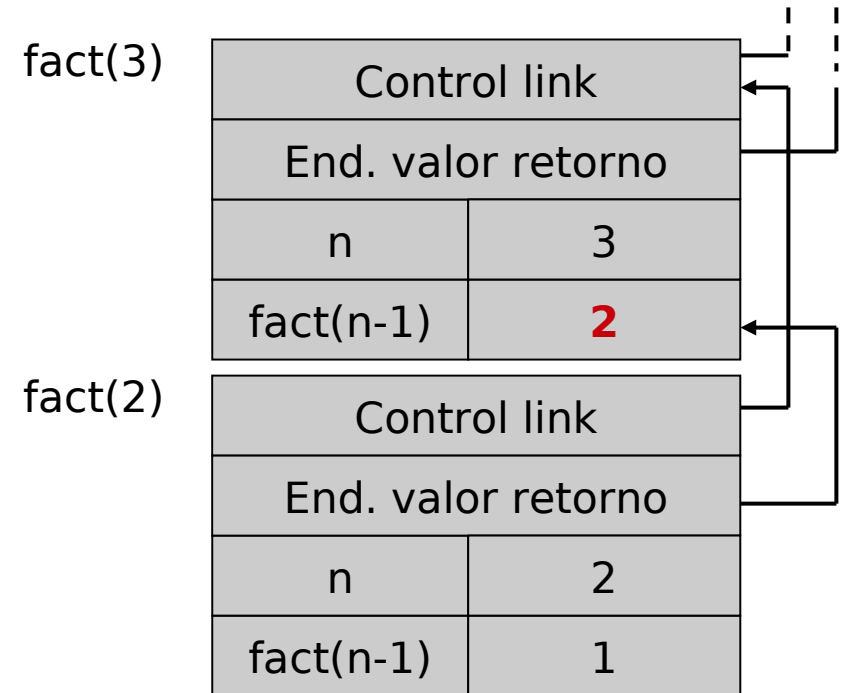
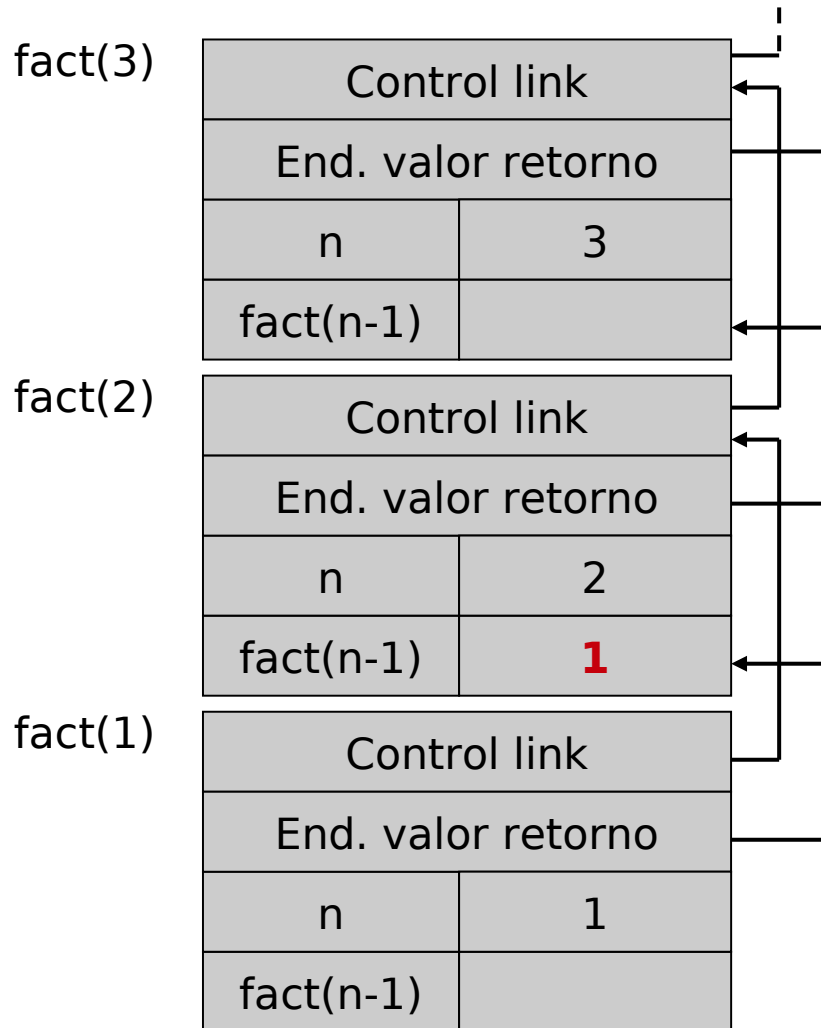
fact(2)

Control link	
End. valor retorno	
n	2
fact(n-1)	

fact(1)

Control link	
End. valor retorno	
n	1
fact(n-1)	

# Retorno da função



```
fact(n) = if n <= 1
          then 1
          else n * fact(n-1)
```

# Parâmetro formal e atual

## Parâmetro formal

Nome utilizado na declaração da função

## Parâmetro atual

Valor do parâmetro numa chamada de função

## Exemplo

```
proc p ( int x, int y )  
  if (x>y) then ... else ...;  
  ...  
  x := y*2+3;  
  ...  
}  
p ( z, 4*z+1 ) ;
```

The diagram illustrates the relationship between formal parameters and actual parameters. In the function declaration, the formal parameters are `x` and `y`, which are circled in red. In the function call, the actual parameters are `z` and `4*z+1`, which are also circled in red. Arrows point from the actual parameters to the formal parameters: one arrow from `z` to `x`, and another from `4*z+1` to `y`.

Parâmetros formais

Parâmetros atuais

# R-value e L-value

## Atribuição

$x := y + 3$

## L-value

Refere **localização** da variável

## R-value

Refere **conteúdo** da variável



# Passagem por referência e por valor

## Passagem por valor

Coloca o R-value do parâmetro atual no RA

Características

- Função não pode alterar valor da variável passada

- Reduz a criação de pseudónimos

- Pode ser menos eficiente para grandes estruturas

## Passagem por referência

Coloca o L-value do parâmetro atual no RA

Características

- Função pode alterar valor à variável que é passada

- Podem existir vários nomes para a mesma localização de memória

- Pode ser menos eficiente para pequenas estruturas

# Exemplo

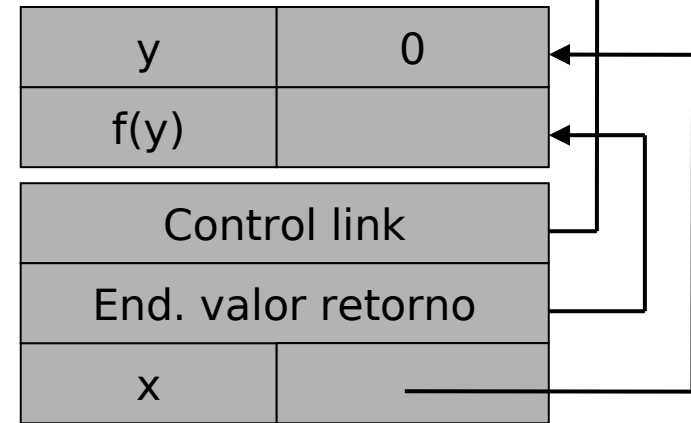
```
function f (x) =  
  { x = x+1;  
    return x;  
  }
```

```
var y = 0;
```

```
print (f(y)+y);
```

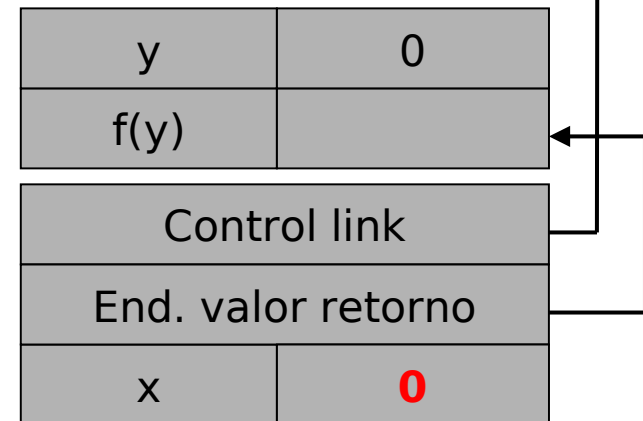
Passagem por  
referência

f(y)



Passagem por  
valor

f(y)



# Outro exemplo

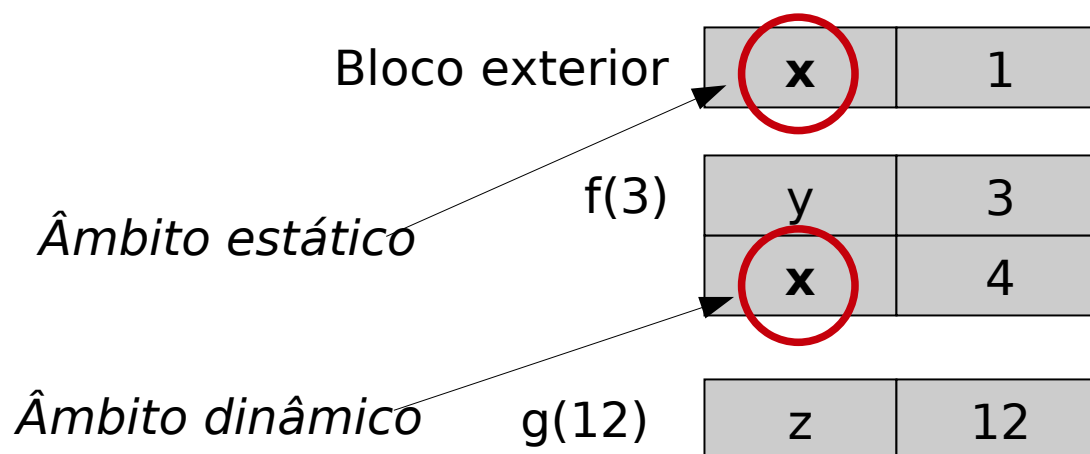
```
function f (pass-by-ref x, pass-by-value y)
begin
    x:=2;
    y:=1;
    if x=1 then return 1 else return 2;
end;
var z: int;
z:=0;
print f(z, z);
```

**Qual o output?**

# Acesso a variáveis globais

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

**Qual o x usado na expressão x+z?**



# Regras de âmbito

## Âmbito estático

Global refere o identificador declarado no bloco envolvente mais próximo do texto do programa

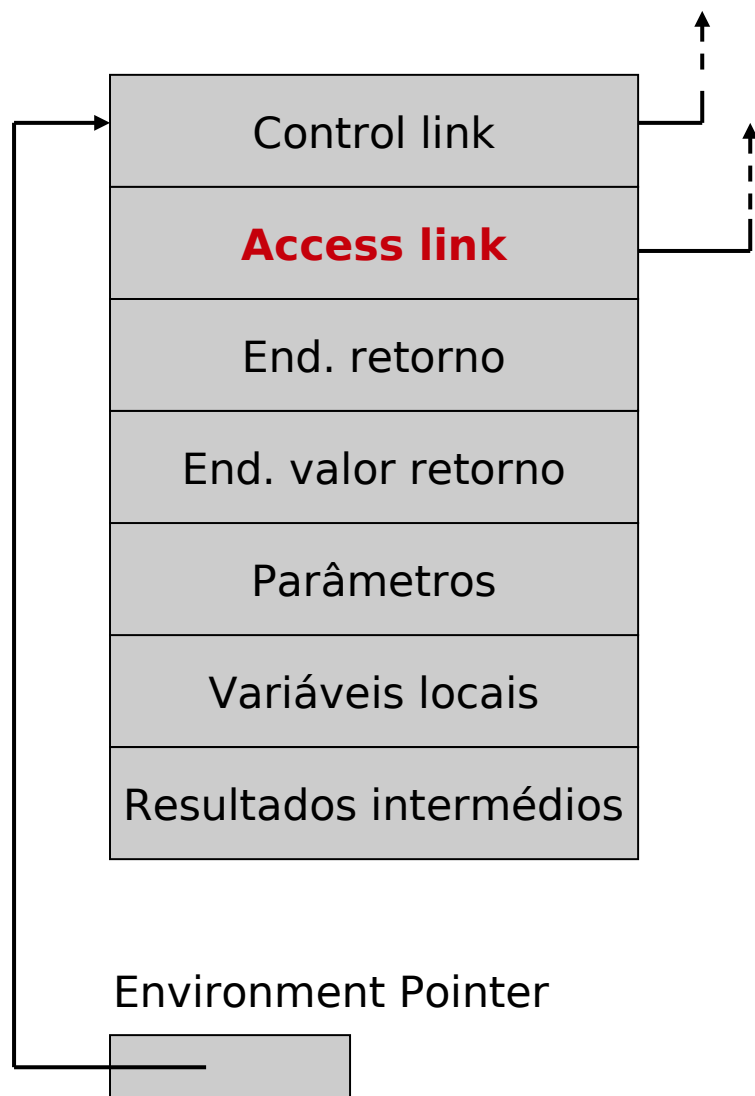
Utiliza a relação estática entre blocos

## Âmbito dinâmico

Global refere o identificador associado ao registo de ativação mais recente

Utiliza a sequência dinâmica de chamadas na execução

# RA para âmbito estático



## Control link

Ligação ao RA do bloco anterior (chamador)

Depende do comportamento **dinâmico** do programa

## Access link

Ligação ao RA do bloco envolvente mais próximo (no texto do programa)

Depende da forma estática do texto do programa

Também designado por *static link*

# Âmbito estático

## Bloco em linha

O bloco envolvente mais próximo é o bloco mais recente

control link = access link

## Função

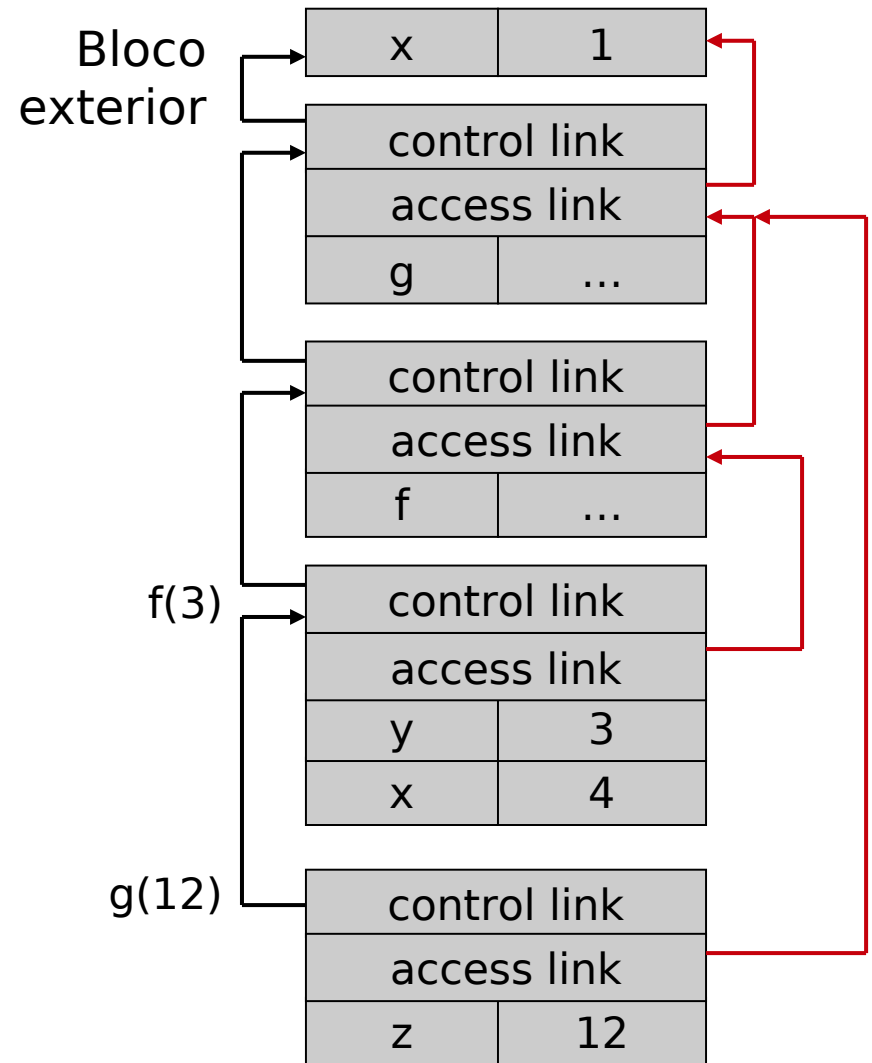
O bloco envolvente mais próximo é determinado pelo sítio onde a função é declarada, que usualmente é diferente do sítio onde é chamada

control link  $\neq$  access link!

# Exemplo

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

**Cada declaração é tratada como um novo bloco**





## **Utilizado para encontrar variáveis globais**

Aponta sempre para o registo do bloco envolvente mais próximo  
Para chamadas de função é o bloco que contém a declaração da função!

**Necessário para linguagens onde as funções podem ser declaradas dentro de funções ou outros blocos aninhados**

# Âmbito nas linguagens de programação

## Âmbito dinâmico

Primeiros Lisp

Tex/Latex

Exceções

Macros

## Âmbito estático

Novos Lisp, Scheme

Algol e Pascal

C

ML

Outras lings. atuais

# Recursividade terminal

## Chamada terminal

A chamada da função  $f$  no corpo de  $g$  é **terminal** se  $g$  retornar o resultado da chamada de  $f$  sem mais computação

```
fun g(x) = if x>0 then f(x) else f(x)*2
```

## Recursividade terminal

A função  $f$  é recursiva terminal se **todas** as chamadas recursivas em  $f$  forem terminais

```
fun fact(n, a) =  
if n<=1 then a else fact(n-1, n*a)
```

# Recursividade terminal

## Chamada terminal

A chamada da função  $f$  no corpo de  $g$  é terminal se  $g$  retornar o resultado da chamada de  $f$  sem mais computação

```
fun g(x) = if x>0 then f(x) else f(x) * 2
```

Chamada terminal

Chamada não terminal

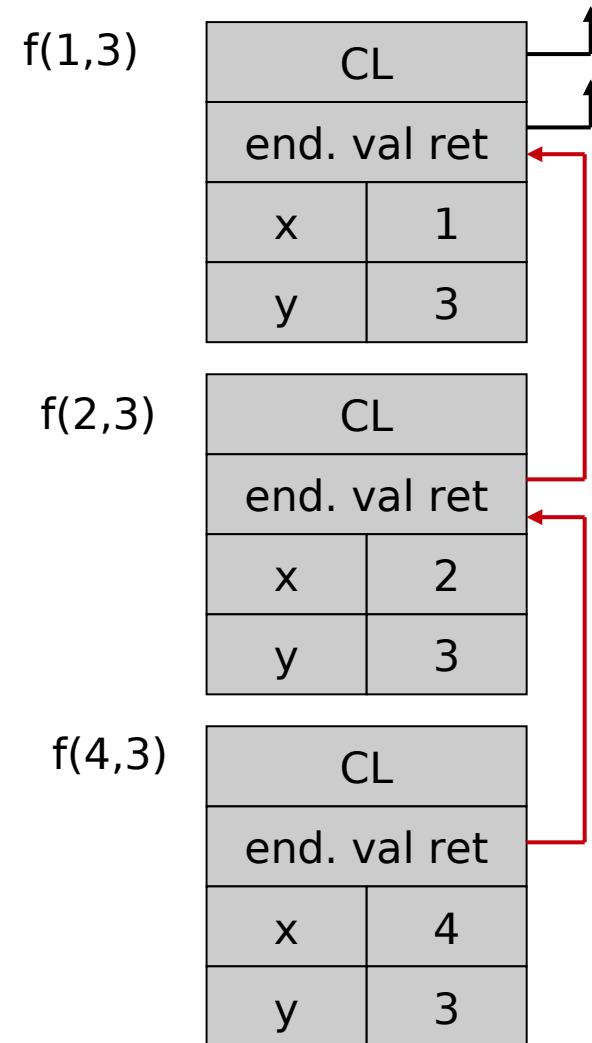
## Recursividade terminal

A função  $f$  é recursiva terminal se **todas** as chamadas recursivas em  $f$  forem terminais

```
fun fact(n, a) =  
if n<=1 then a else fact(n-1, n*a)
```

# Exemplo

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



# Otimização

## Como não há cálculo após a chamada

Coloca-se o end. valor retorno igual ao do chamador

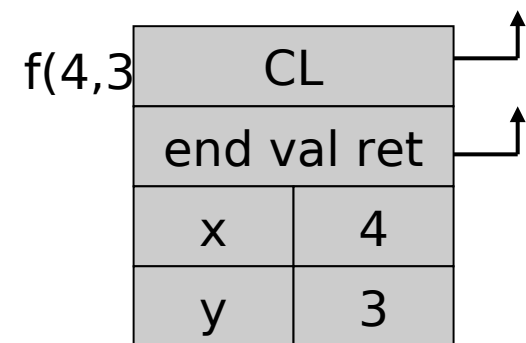
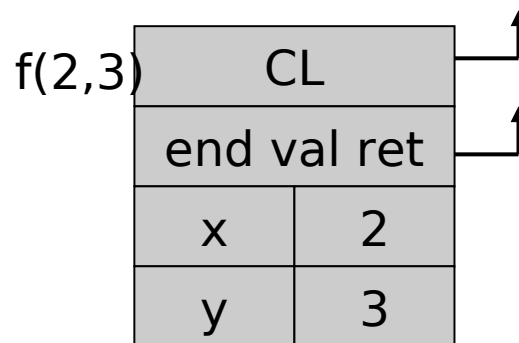
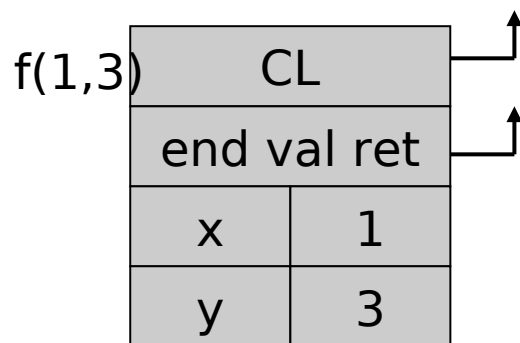
É desnecessário voltar ao registo do chamador

Pode reutilizar-se o mesmo registo de activação

## Exemplo

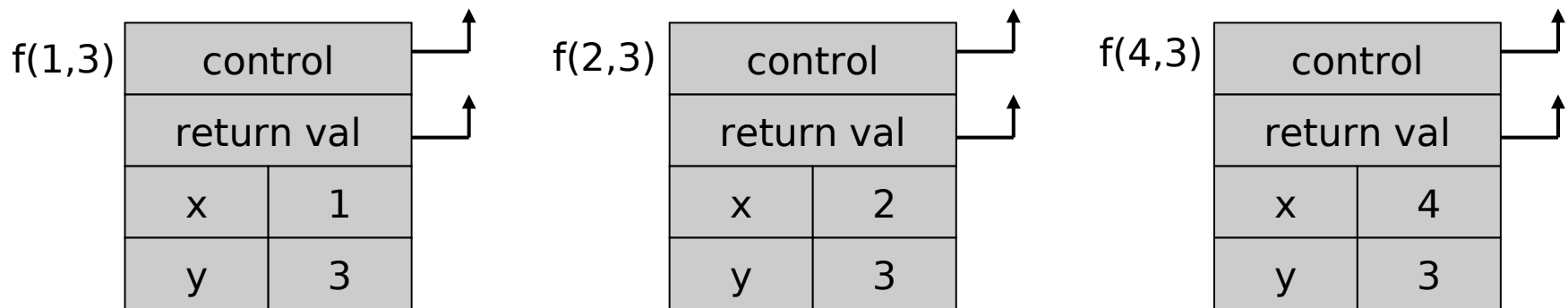
```
fun f(x,y) = if x>y then x else f(2*x, y);
```

```
f(1,3) + 7;
```



# Recursividade terminal e iteração

São equivalentes!



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

fun g(y) {  
 x := 1;  
 while !(x > y)  
 x := 2 \* x;  
 return x;  
}

Diagram illustrating the equivalence between recursive and iterative implementations of a function. The recursive function  $f(x, y)$  is shown on the left, and the iterative function  $g(y)$  is shown on the right. The diagram highlights the correspondence between the recursive call and the loop in the iterative version.

Annotations:

- teste** (green): Points to the condition  $x > y$  in the recursive function.
- valor inicial** (red): Points to the initial value  $x := 1$  in the iterative function.
- corpo do ciclo** (orange): Points to the loop body  $x := 2 * x;$  in the iterative function.

# Funções de ordem superior



# Função de ordem superior

## Caraterísticas

- declarada em qualquer âmbito
- passada como argumento de outras funções
- devolve função como resultado

**Também designada por função de 1ª classe**

## Exemplo

```
fun map (f, nil) = nil
| map (f, x::xs) = f (x) :: map (f, xs)
```

# Linguagens com funções de 1ª classe

**É necessário manter o ambiente da função para determinar o âmbito estático das variáveis globais**

## **Como fazê-lo?**

Função como argumento

É necessário um apontador para um RA “mais acima” na stack

Função como resultado

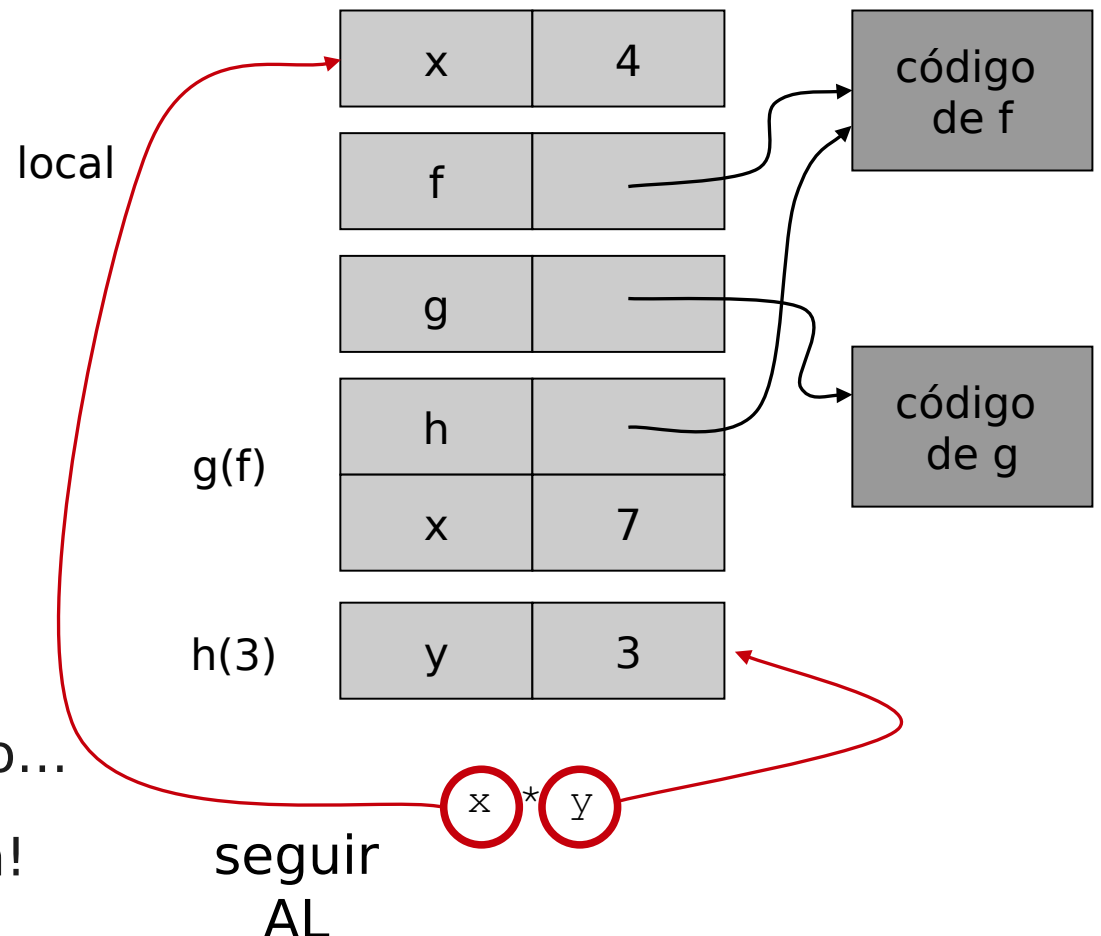
É necessário manter o RA da função devolvida (embora o âmbito da função que a devolve termine)

# Função como argumento

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

global

local



**Qual o AL de h(3)?**

bloco envolvente mais próximo...

... mas não há declaração de h!

# Fecho

## O que é?

A representação do valor de uma função em linguagens com funções de 1ª classe e âmbito estático

## Como é constituído?

Par <ambiente, código>

Ambiente → aponta o RA da declaração da função

Código → aponta o código da função

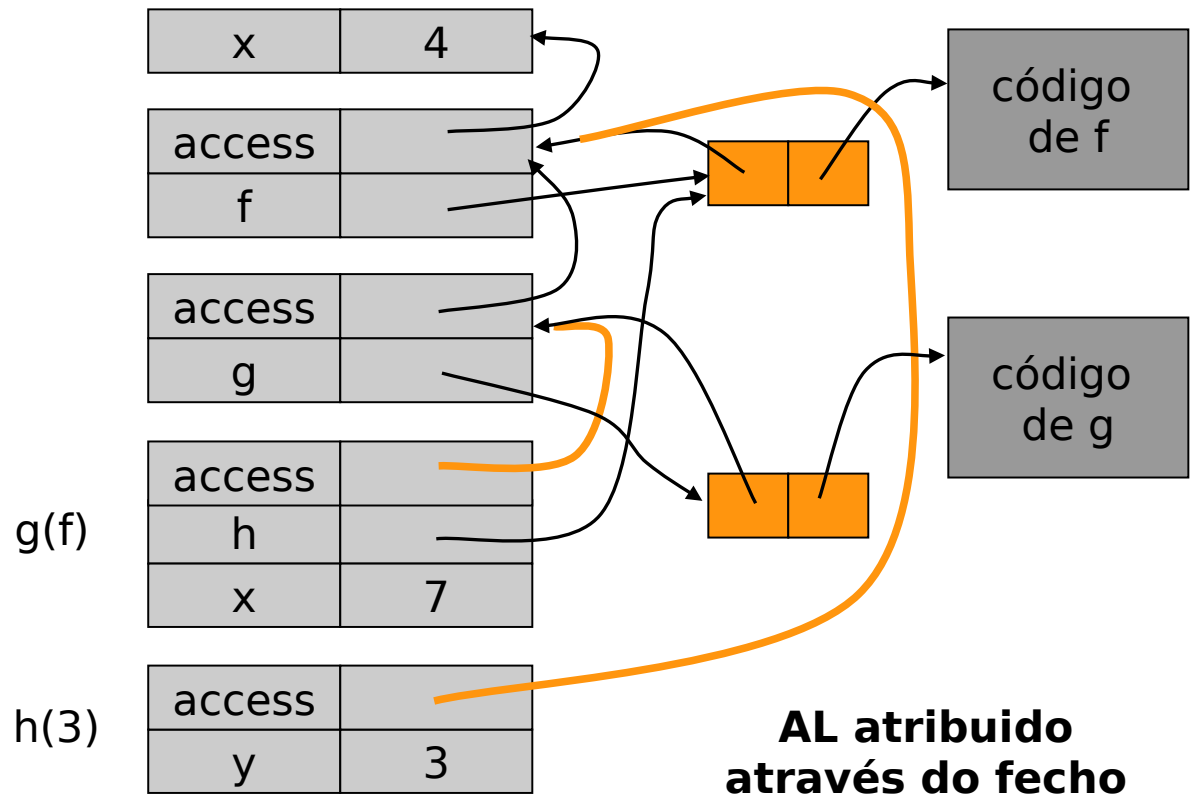
## Como funciona?

Quando uma função é chamada é criado o RA e

O access link fica com o valor do ambiente do fecho da função

# Função como argumento e fecho

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



# Resumo

**O fecho mantém o ambiente estático da função**

**Na chamada da função o fecho é usado para determinar o access link**

**Os AL apontam “para cima” na stack**

Pode ser necessário “saltar” RA anteriores para encontrar variáveis globais

Os RA são libertados usando a ordem normal de stack → lifo

# Função como resultado de função

## Exemplo

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

## A função é criada dinamicamente

Expressão com variáveis livres

Os valores são determinados em tempo de execução

O valor da função é o fecho `<ambiente, código>`

## Mas o código não é compilado dinamicamente (na maioria das linguagens)

# Exemplo

```
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);
```

**A função `mk_counter` devolve uma função**

**Como é determinado o valor de `count` em `c(2)`?**



# Função como resultado e fecho

```
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);
```

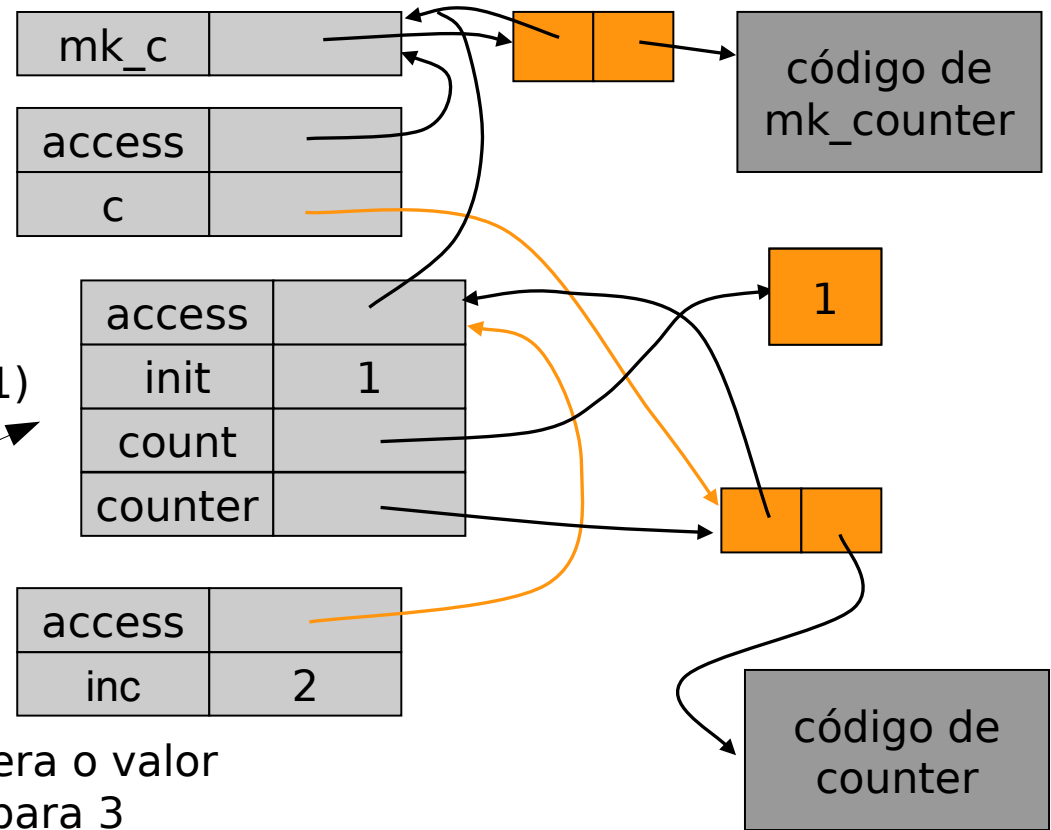
3      5

RA necessário após o retorno de `mk_counter` porque a função `count` devolvida necessita das vars `init` e `count`

`mk_counter(1)`

`c(2)`

A chamada altera o valor de `count` de 1 para 3



# Resumo

**O fecho mantém o âmbito estático da função**  
**Pode ser necessário manter o RA depois do retorno da função**

A política de stack (lifo) não funciona!

## **Como resolver?**

Esquecer a libertação explícita

Colocar os RA na heap

Invocar o *garbage collector* quando necessário

# Conceitos importantes

## **Lings estruturadas em blocos usam stack de RA**

RA contêm parâmetros, vars locais, ...  
e apontam para o âmbito envolvente

## **Diversos mecanismos de passagem parâmetros**

## **Recursividade terminal pode ser otimizada**

## **Funções de 1ª classe necessitam de fecho**

Apontador de ambiente utilizado na chamada da função  
Política de stack pode falhar se uma chamada devolver função

## **Desnecessário se funções não puderem estar em blocos aninhados**