Scope, Function Calls and Storage Management

Angelo Gargantini

M1 visibilità e funzioni

Introduzione

- Quando dichiariamo una variabile, il computer dove la memorizza?
- Quali sono le regole per accedere ad una variabile?
- Come vengono passate ai sottoprogrammi i dati?
- Principali feature:
  - Divisione di un programma in sottoprogrammi
    - Non come il BASIC
      - Non unica sequenza di istruzioni (con GOTO)
    - Non si sanno tutte le variabili prima dell'esecuzione e l'allocazione della memoria avviene dinamicamente
      - Non costringo al programmatore di dichiarare tutte le varibili fin dall'inizio
  - Uso della ricorsione

Topics

- Block-structured languages and stack storage
- In-line Blocks
  - activation records
  - storage for local, global variables
- First-order functions
  - parameter passing
  - tail recursion and iteration
- NO - Higher-order functions
  - deviations from stack discipline
  - language expressiveness => implementation complexity

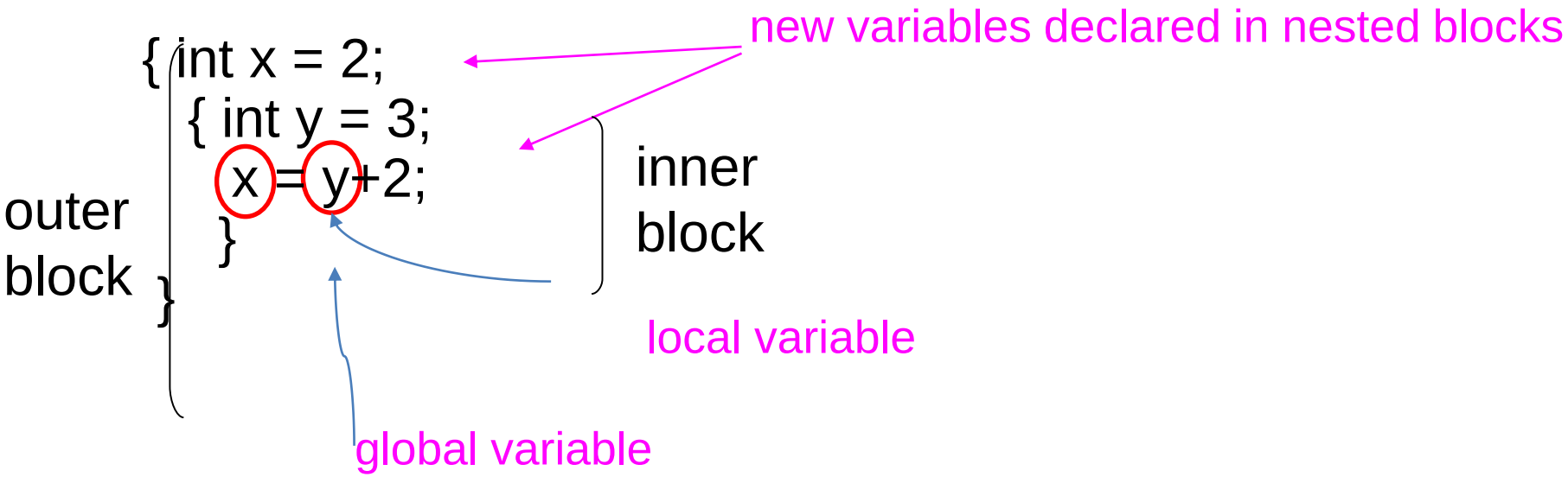
Block-Structured Languages

Storage management

- Enter block: allocate space for variables
- Exits block: some or all space may be deallocated

Nested blocks, local variables

Example



Examples

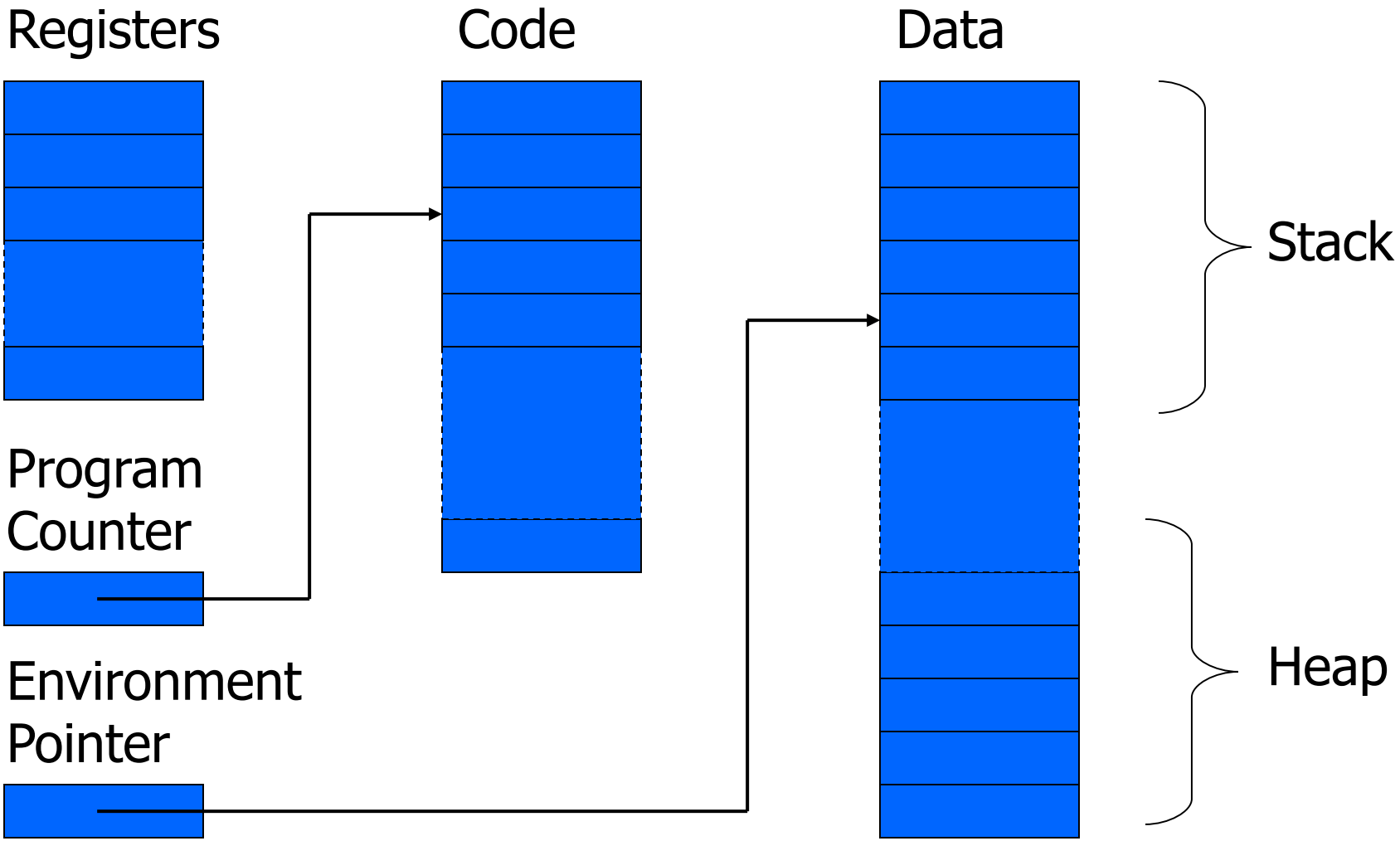
- Blocks in common languages
  - C/c++/Java { … }
  - Algol begin … end
  - ML let … in … end
- Two forms of blocks
  - In-line blocks
    - Blocks for control structure like if, for and so on.. similar to block inline
  - Blocks associated with functions or procedures

- Topic: block-based memory management, access to *local variables*, *parameters*, *global vars*
- It allows **recursive functions**

Alcune note

- Alcuni linguaggi (come Fortran) allocavano in modo fisso le variabili
  - Svantaggi …
    - Ricorsione?
- Block-structured languages:
  - New variables may be declared at various points in a program
  - Each declaration is visible within a block
  - When a program begins executing the instructions contained in a block, the memory is allocated
  - When a program exits, the memory is freed
  - An identifier that is not declared in the current block is considered global to the block

Simplified Machine Model

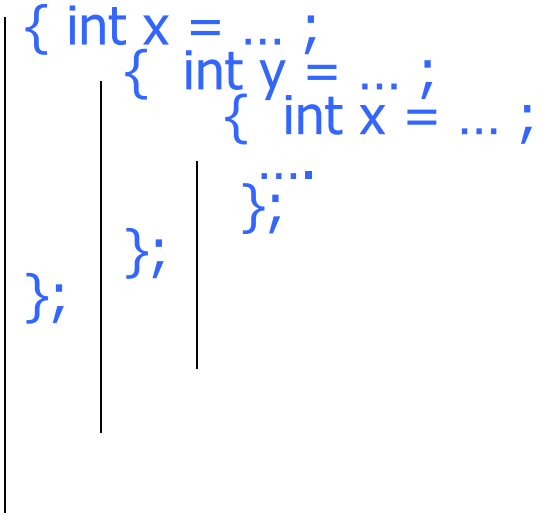


Interested in Memory Mgmt Only

- Registers, Code segment, Program counter
  - Ignore registers
  - Details of instruction set will not matter
- Data Segment
  - Stack contains data related to block entry/exit
  - Heap contains data of varying lifetime
  - **Environment pointer** points to current stack position
    - Block entry: add new activation record to stack
    - Block exit: remove most recent activation record

Some basic concepts

- Scope
  - Region of program text where declaration is visible
- Lifetime
  - Period of time when location is allocated to program



- Inner declaration of x hides outer one.
- Called "hole in scope"
- Lifetime of outer x includes time when inner block is executed
- Lifetime ≠ scope
- Lines indicate "contour model" of scope.

In-line Blocks

- Activation record
  - Data structure stored on run-time stack
  - Contains space for local variables (if any)
- Example

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

```
Push record with space for x, y
Set values of x, y
      Push record for inner block
      Set value of z
      Pop record for inner block
Pop record for outer block
```

Intermediate results on the stack

May need space for variables and intermediate results like (x+y), (x-y)

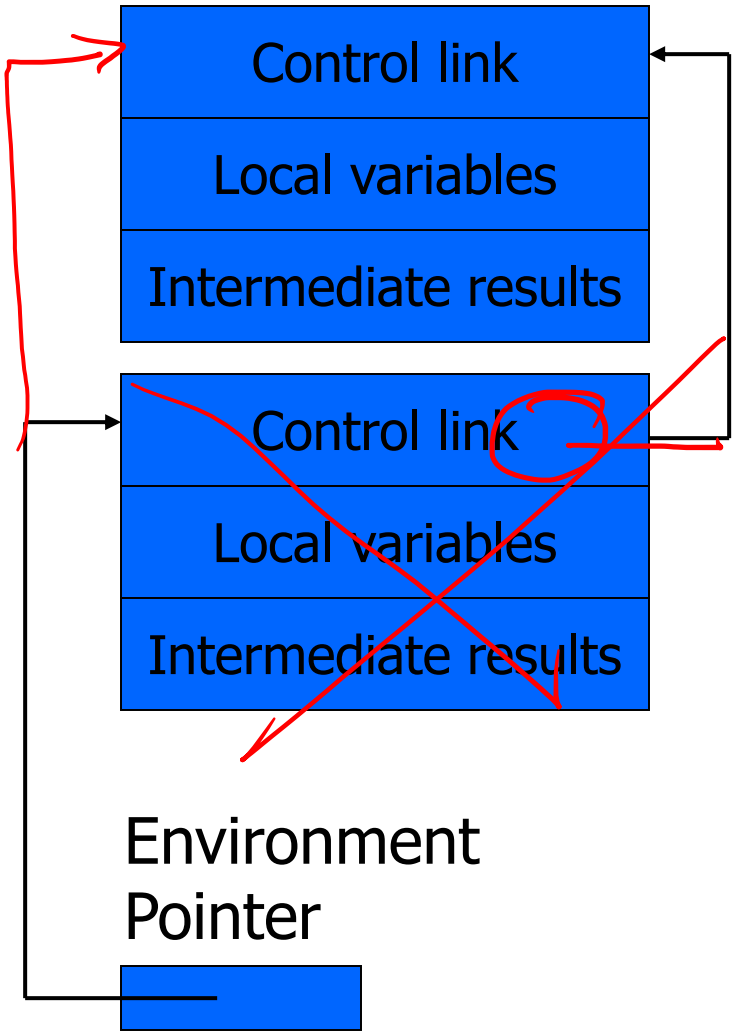
Example:
int z = (x+y) *(x-y)

push x+y
push x-y
a1= pop a2 =pop push a1*a2

Control Link

- EP punta alla cima del record di attivazione corrente
- Record di attivazione ha dimensione variabile
- Come faccio a ripristinare EP quando faccio il pop del record di attivazione che non serve più?
- Uso il control link:
  - Puntatore alla cima del record di attivazione precedente
  - Viene salvato quando creo il record di attivazione
  - Viene ripristinato quando faccio il pop
-

Activation record for in-line block

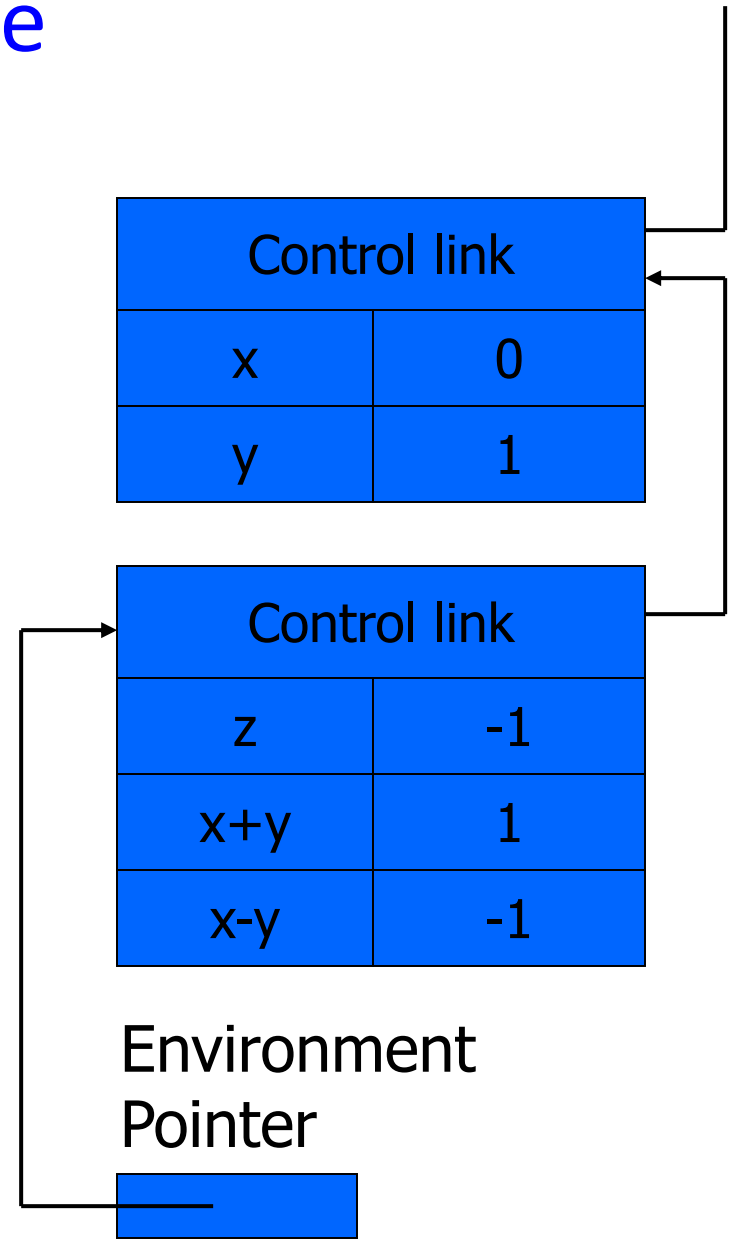


- Control link
  - pointer to previous record on stack
- Push record on stack:
  - Set new control link to point to old env ptr
  - Set env ptr to new record
- Pop record off stack
  - Follow control link of current record to reset environment pointer

Example

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

Push record with space for x, y (set control link = old env pointer, set env pointer)
Set values of x, y
    Push record for inner block
    Set value of z
    Pop record for inner block (set env pointer to control link)
Pop record for outer block



Scoping rules

- Global and local variables
  - x, y are local to outer block
  - z is local to inner bock
  - x, y are global to inner block

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

- Static scope
  - global refers to declaration in closest enclosing block
- Dynamic scope
  - global refers to most recent activation record

These are same until we consider function calls.

Esercizi

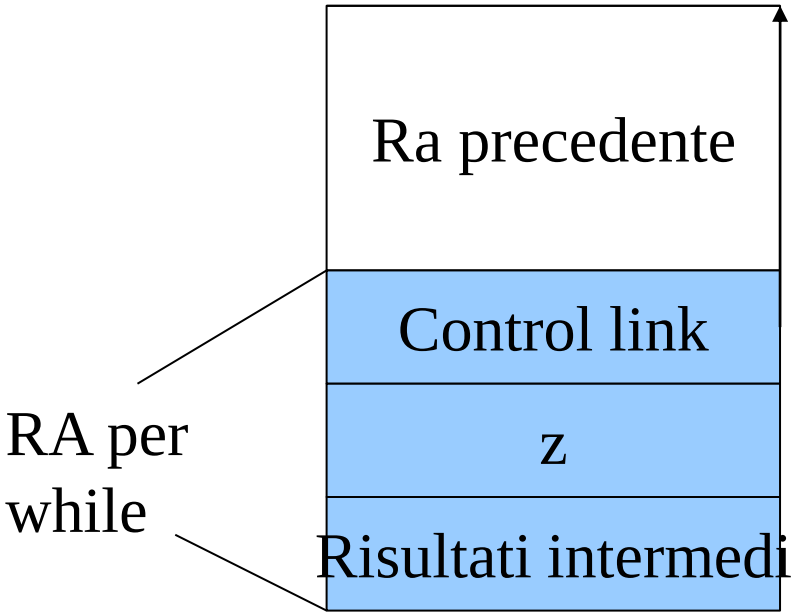
- Qualche esercizio sullo stack ...
- proviamo a stampare alcune info dello stack

Per blocchi in-line di condizioni if e cicli for

- Del tutto simile come i blocchi inline
- Nel caso di cicli il RA viene messo solo una volta e usato per tutta la durata del ciclo (fino alla fine di tutti i cicli)
- Esempio
-

```
while(..){
   int z;
  z = ....

}
```



Se non ci sono var locali al blocco non c'è neanche RA

Promozione variabili globali

- nota: non sempre il compilatore crea un record di attivazione di un blocco, la maggior parte delle volte nei compilatori moderni promuove la variabile a variabile globale al blocco.

Functions and procedures

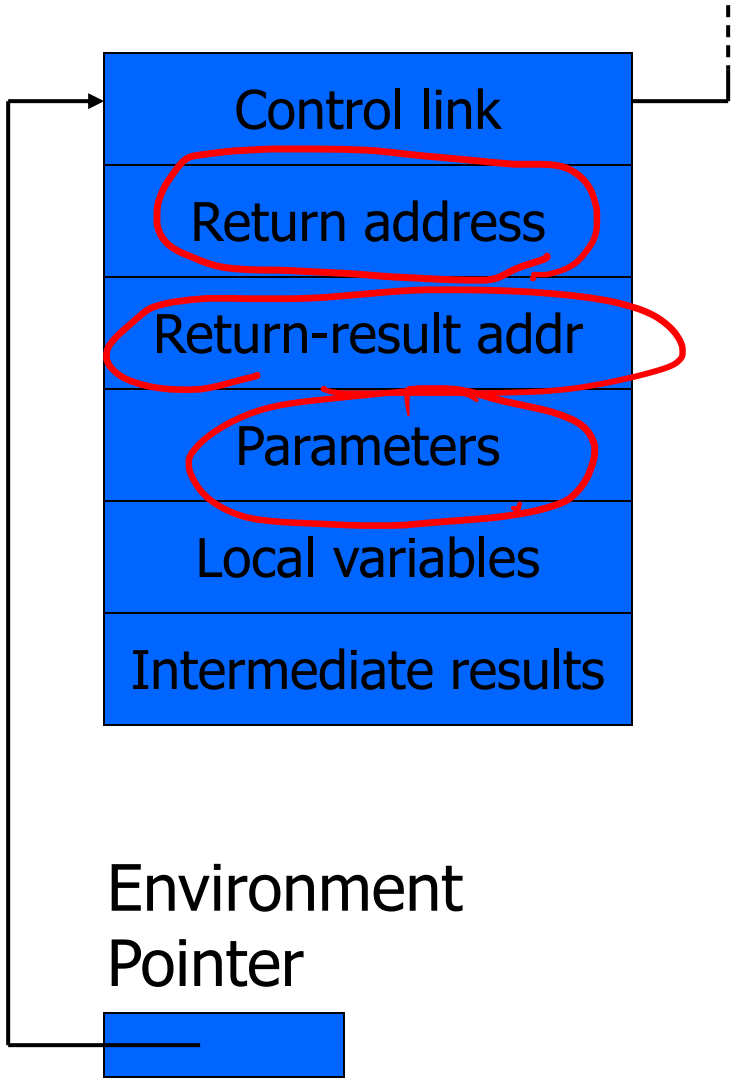
- Syntax of procedures (Algol) and functions (C)

```
procedure P (<pars>)          <type> function f(<pars>)
    begin                      {
        <local vars>                  <local vars>
        <proc body>                   <function body>
    end;                      };
```

- Activation record must include space for

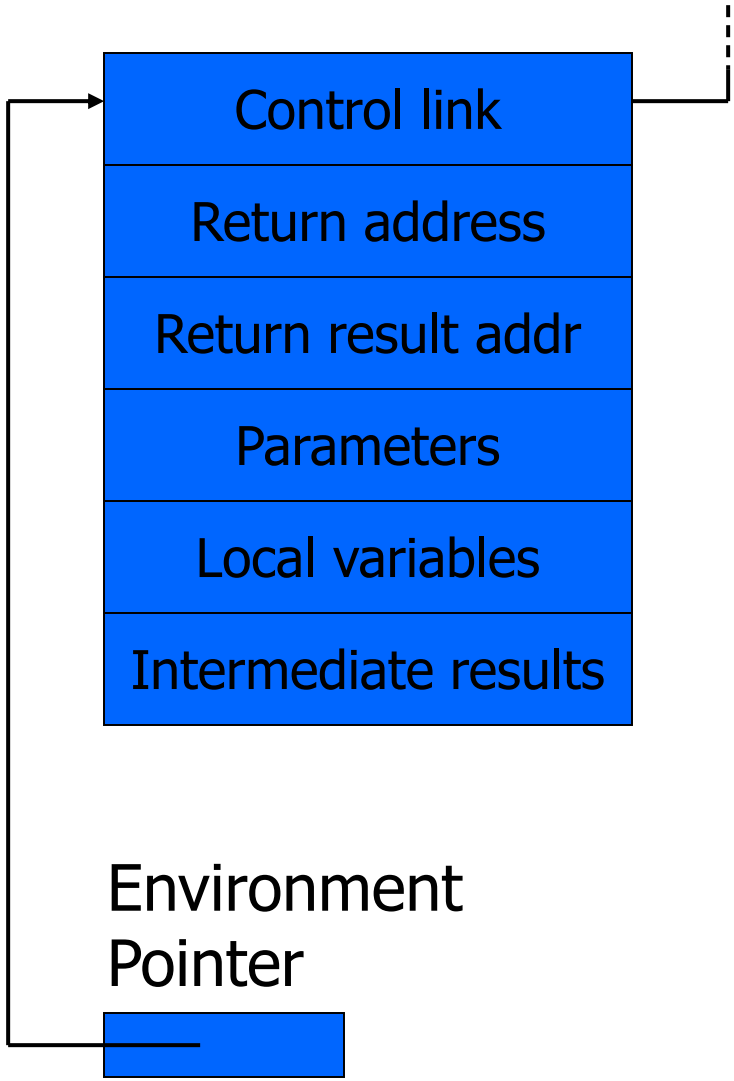
  - parameters
  - return address
  - Local variables
    (and intermediate result)
  - location to put return value on function exit

Activation record for function



- Return address
  - Location of code to execute on function return
- Return-result address
  - Address in activation record of calling block to receive return address
- Parameters
  - Locations to contain data from calling block

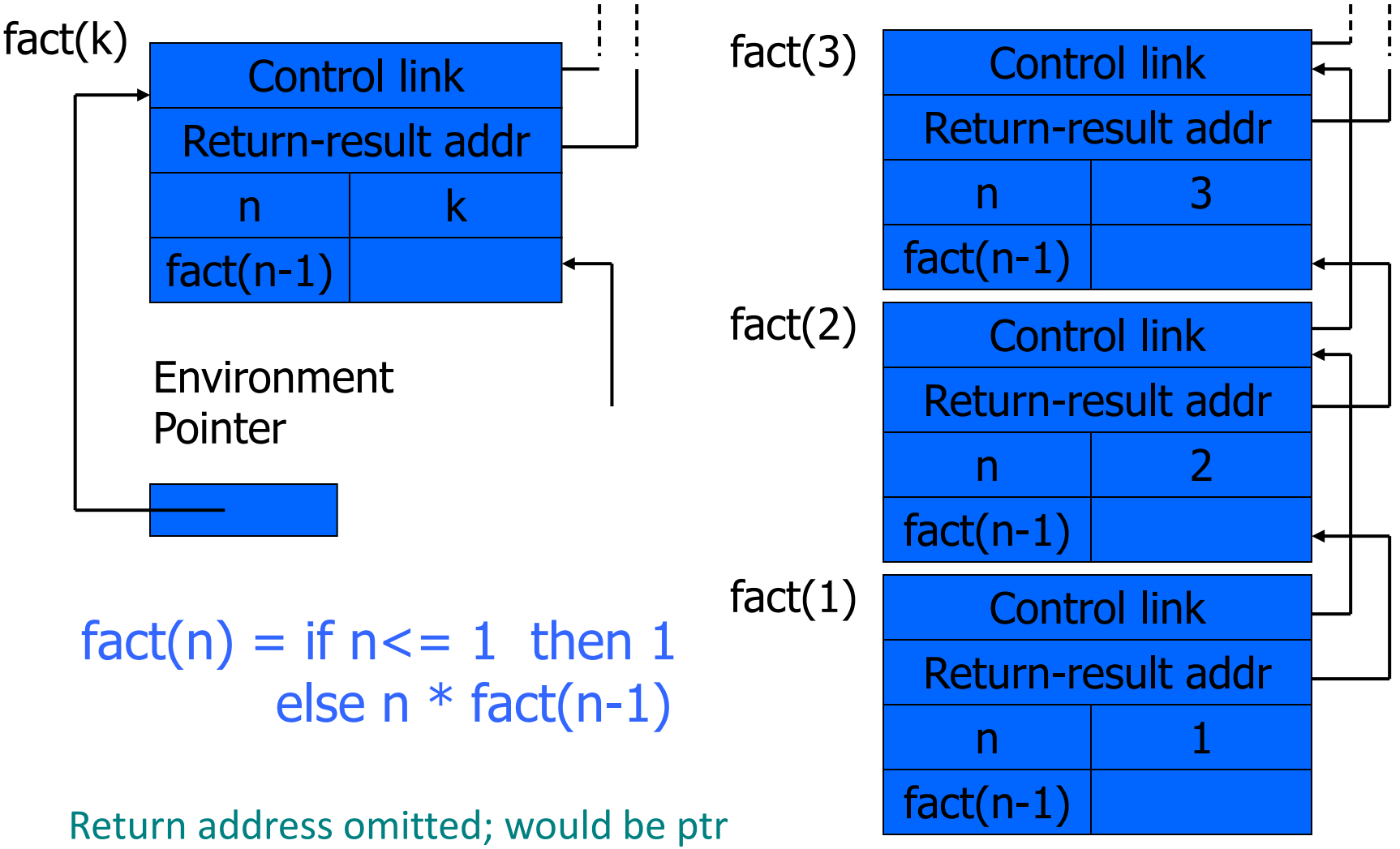
Example



- Function

  fact(n) = if n<= 1 then 1
  else n * fact(n-1)
- Return result address
  - location to put fact(n)
- Parameter
  - set to value of n by calling sequence
- Intermediate result
  - locations to contain value of fact(n-1)

Function call

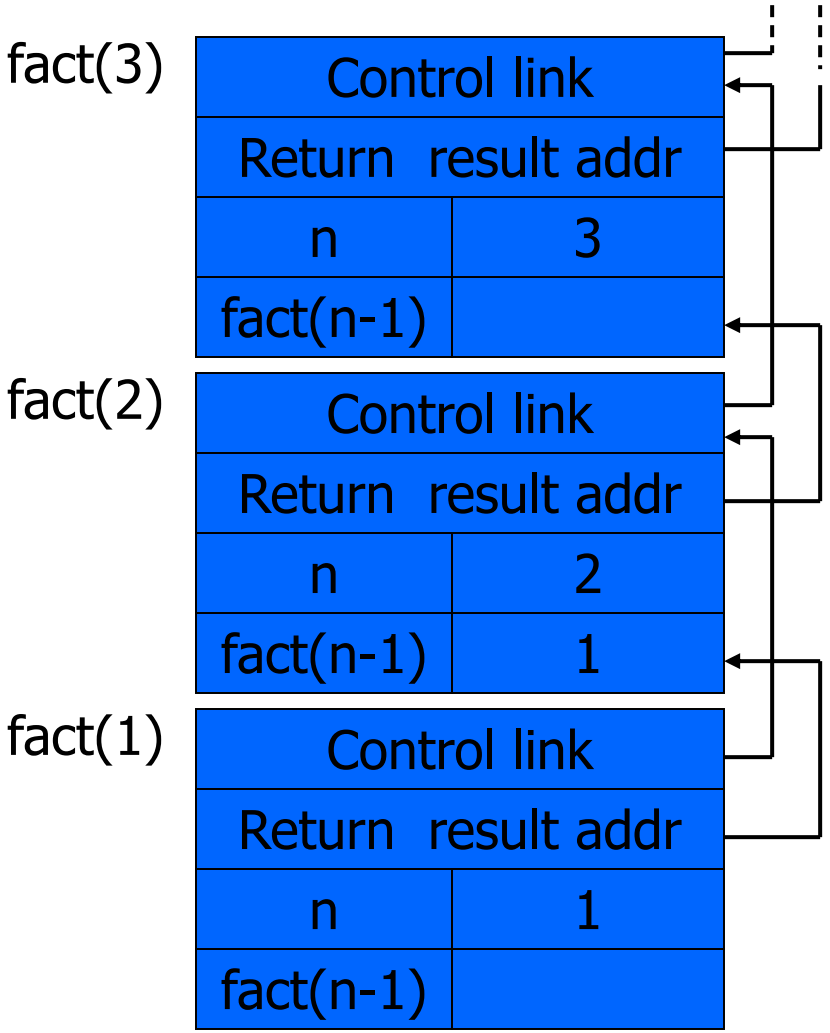


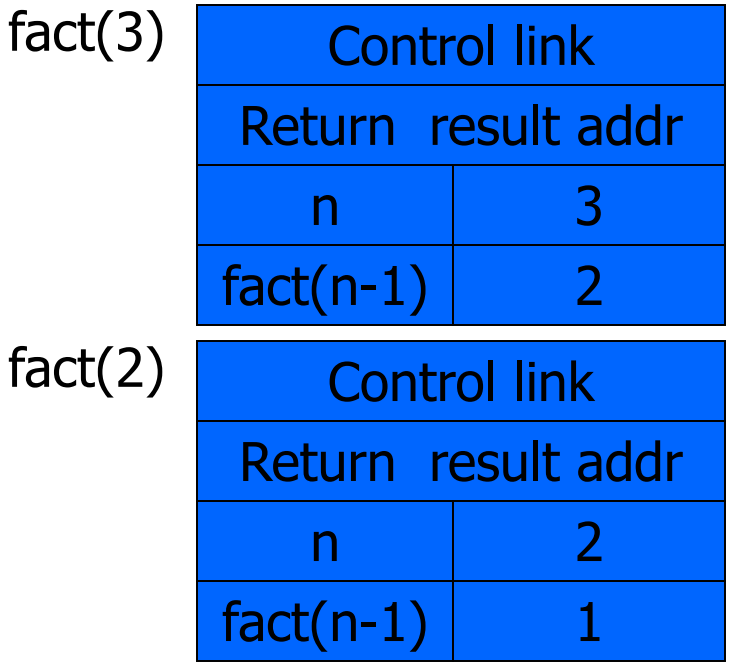
fact(n) = if n<= 1 then 1
else n * fact(n-1)

Return address omitted; would be ptr into code segment

Function return next slide →

Function return





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

Topics for first-order functions

- Parameter passing
  - use ML reference cells to describe pass-by-value, pass-by-reference
- Access to global variables
  - global variables are contained in an activation record higher “up” the stack
- Tail recursion
  - an optimization for certain recursive functions

See this yourself: write factorial and run under debugger

ML imperative features (review)

- General terminology: L-values and R-values
  - Assignment y := x+3
    - Identifier on left refers to location, called its L-value
    - Identifier on right refers to contents, called R-value
- ML reference cells and assignment (anche in C++)
  - Different types for location and contents

    x : int non-assignable integer value

    y : int ref location whose contents must be integer

    !y the contents

    ref x expression creating new cell initialized to x
  - ML form of assignment

    y := x+3 place value of x+3 in location (cell) y

    y := !y + 3 add 3 to contents of y and store in location y

(in C++)

- Anche in C++ esistono i riferimenti:

```
int y;
int& x = y;
```

- Nota:
- References cannot be uninitialized. Because it is impossible to reinitialize a reference, they must be initialized as soon as they are created. In particular, local and global variables must be initialized where they are defined...
- Vedremo più avanti nel modulo C++

Parameter passing

- Pass-by-reference
  - Caller places L-value (address) of actual parameter in activation record
  - Function can assign to variable that is passed
  - In some language is also call by variable
    - PASCAL:
    - `procedure Name(a,b : integer;VAR c,d: integer);`
    - a and b are passed by value, c and d by reference
- Pass-by-value
  - Caller places R-value (contents) of actual parameter in activation record
  - Function cannot change value of caller's variable
  - Reduces aliasing (alias: two names refer to same loc)

Example

pseudo-code

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

pass-by-ref

Standard ML

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

pass-by-value

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

Example

pseudo-code

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

pass-by-ref

```
int   f (int & x) {
      x = x+1;
      return x;
}
int y = 0;
cout<< f(y) + y;
```

pass-by-value

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

int y = 0;
cout<< f(y) + y;
```

C++

Passaggio di puntatori

- Il passaggio di puntatori è un passaggio per valore, ma si usa (in C) per ottenere lo stesso effetto del passaggio per riferimento.
- Es.:

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

int y = 0;
printf(f(&y) + y;)
```

Se si vuole, si può evitare la modifica del parametro attuale mediante copia:

```
int   f (int*  x) {
     int z =  *x
     return z+1;
}

int y = 0;
printf(f(&y) + y);
```

Passaggio degli array in C

- Come si passano gli array in C
- Si possono passare come array:
  - void foo(int arr[5])
  - ATTENZIONE: When an array is passed as a parameter, only the memory address of the array is passed (not all the values). An array as a parameter is declared similarly to an array as a variable, but no bounds are specified. The function doesn't know how much space is allocated for an array.
  - Ma arr è semplicemente un puntatore di interi, non c'è alcuna informazione sulla dimensione dell'array !!!
  - Attenzione quindi all'uso di **sizeof**
  - Vedi esempio !!!
- Soluzione:
  1. Passare anche la dimensione void foo(int arr[], int n)
  2. Mettere un terminatore (come le stringhe)

Passaggio stringhe

- Caso particolare le stringhe
  - Zero terminated (0 o ‘\0’ NON ‘0’)
  - Qualche problema (vedremo in futuro)

- Esercizio. Scrivi una funzione che prende una stringa in ingresso e restituisce quante ‘a’ ci sono
- Oppure palindroma (lab)

Passaggio struct

- Quando passo una struct passo l'intero record.
- Esempio:

```
struct student{
      char firstname[30];
      char surname[30];
   };
```

- void addStudent(struct student s) {}

- Tutta la struct è copiata sullo stack
- Lo stesso anche per il return value.
- Per questo si preferisce spesso usare il puntatore o riferimento.

Passaggio di puntatori a puntatori

- Esercizio di passaggio di puntatore a puntatore
- Uso più frequente per modificare un puntatore.

PUNTATORE

```
int y = 10;

void styp(int* p) {
        p = &y;
}

int main(void) {
        int m = 0;
        int * q = &m;
        styp(q);
        printf("%d", *q);
        return EXIT_SUCCESS;
}
>> 0
```

– q punta ancora ad m

PUNTATORE a PUNTATORE

```
int y = 10;

void stypp(int** p) {
        *p = &y;
}

int main(void) {
        int m = 0;
        int * q = &m;
        stypp(&q);
        printf("%d", *q);
        return EXIT_SUCCESS;
}
>> 10
```

– q punta a y

Esercizio

- Una funzione che toglie il primo carattere da una stringa.
- Due alternative:
  - Usa stringhe
  - Usa puntatore a stringhe (puntatore a puntatore)
- Disegnamo lo stack

Parameter passing & activation record

- pass by value: the value of the actual parameter is copied in the activation record as value of the formal parameter
  - Pass by pointer is a particular case
- pass by ref: the address of the actual parameter is copied in the activation record

Osservazioni

- Il passaggio per riferimento ha alcuni vantaggi
  - Meno memoria (pensa ad un oggetto)
- però alcuni svantaggi:
  - Indirezione ulteriore sullo stack
  - Side effect non desiderati – vedi esercizio sul libro
  - Vedi es 7.4
  - Come passare le costanti??
    - Ad esempio un numero
    - Solo L-values, non posso passare Rvalue
  - Posso modificare il dato passato
- Passaggio per nome: il nome del par. formale viene sostituito con il par. attuale
  - Vedi esercizio 5.2
- Fate esercizi 7.3, 7.5, 7.6. 7.7 7.8

Passing parameters in Java

- Classically
  - Pass by value for primitive types
  - Pass “by reference” for reference types (Objects, arrays, …)
  - NOT possible by value for Objects as in C++
- However
  - If you consider: Pass-by-reference
    - The formal parameter merely acts as an alias for the actual parameter. Anytime the method/function uses the formal parameter (for reading or writing), it is actually using the actual parameter.
  - Java is not a real pass by reference but it is pass by value of the pointer

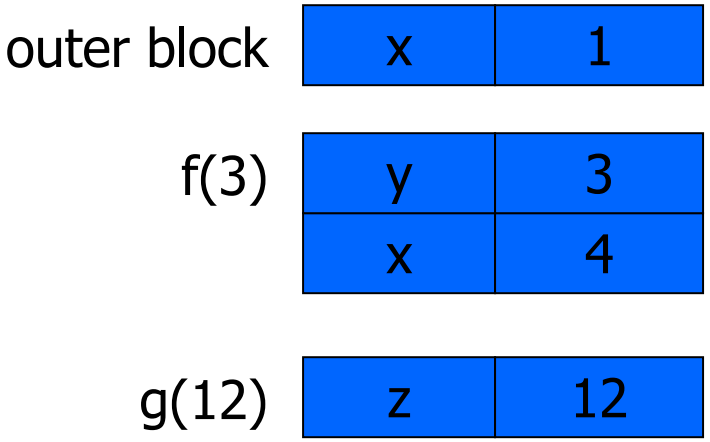
Swap in C++/Java

- Swap: cambio dei valori tra due variabili
  - Primitivi: semplice
  - Oggetti:???
- The Litmus Test
  - There's a simple "litmus test" for whether a language supports pass-by-reference semantics:
  - Can you write a traditional swap(a,b) method/function in the language?
- SWAP in C++
  - esercizio
- SWAP in JAVA????

Access to global variables

- Two possible scoping conventions
  - Static scope: refer to closest enclosing block
  - Dynamic scope: most recent activation record on stack
- Example

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

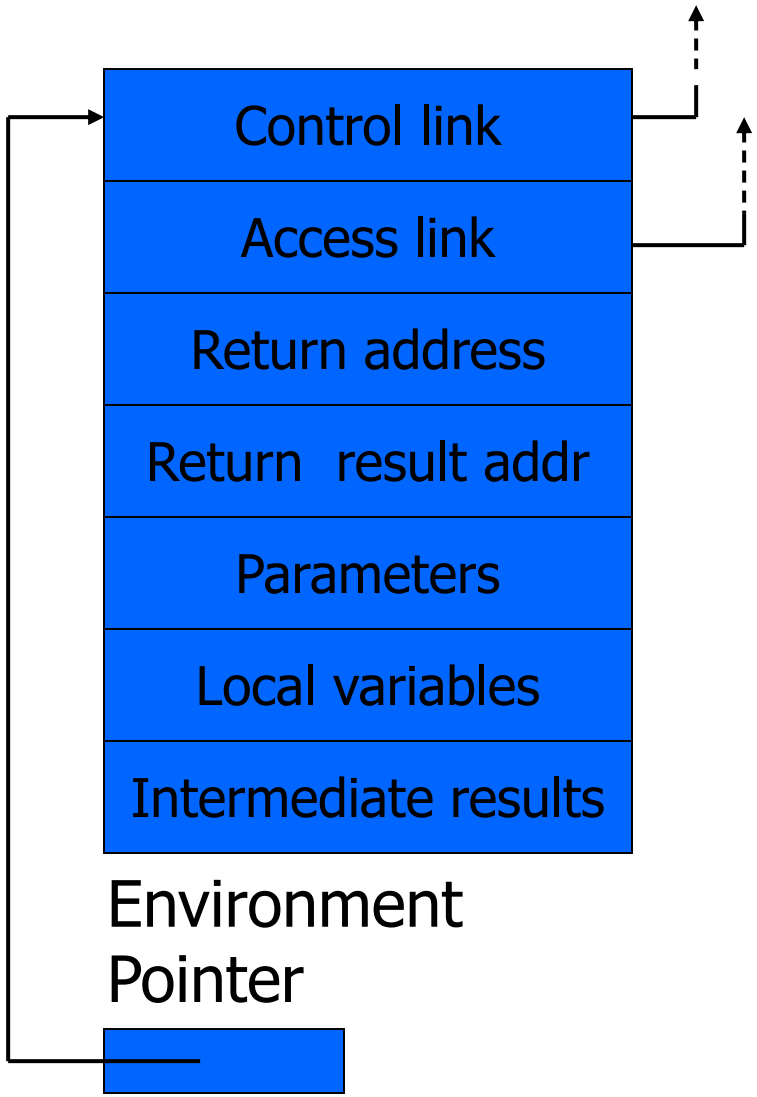


Which x is used for expression x+z ?

Static: prende x = 1 guardando il codice

Dinamico: prende il primo x sullo stack, x = 4

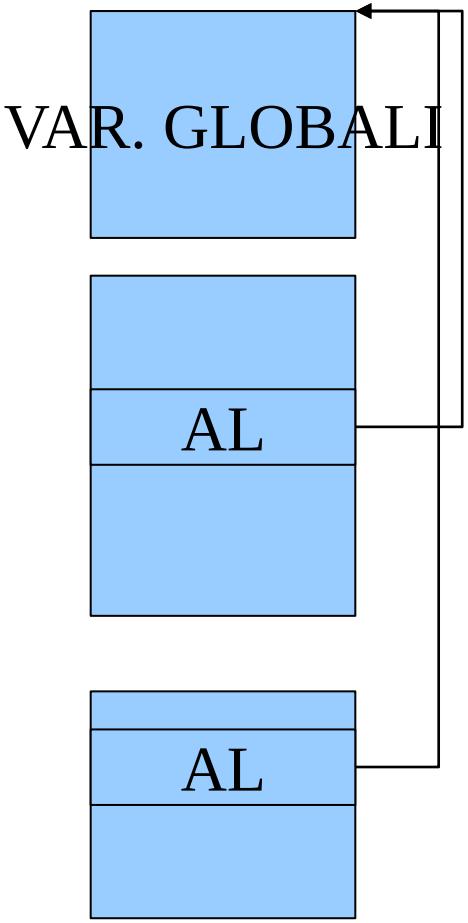
Activation record for static scope



- Control (dynamic) link
  - Link to activation record of previous (calling) block
- Access (static) link
  - Link to activation record of closest enclosing block in program text
- Difference
  - Control link depends on dynamic behavior of prog
  - Access link depends on static form of program text

Access link

La maggior parte dei linguaggi usa il primo (statico)
Per tener traccia si usa lo static link mediante l'access link
Nei linguaggi che usiamo noi (tipo C) in cui ci sono solo due livelli (globale e locale) gli access link non sarebbero necessari – li segnamo per rendere chiaro dove è il record di attivazione globale
In C quindi l'access link punta sempre al RA delle variabili globali



Ricorsione

- Una funzione matematica è definita ricorsivamente quando nella sua definizione compare un riferimento (chiamata) a se stessa.
- Esempio: Funzione fattoriale su interi non negativi:

  fatt(n) = n!
- definita ricorsivamente come segue:
  - 1 se n=0
  - fatt(n)= n*fatt(n-1) se n>0

Esempi di problemi ricorsivi:

1) Somma dei primi n numeri naturali:
   - somma(n)= 0 se n=0
   - n+somma(n-1) altrimenti
2) Ricerca di un elemento el in una sequenza di interi:
   - falso se sequenza terminata, altrimenti
   - ricerca(el,sequenza)=vero se el=primo(sequenza), altrimenti
   - ricerca(el,resto(sequenza))=

Programmi ricorsivi

- Molti linguaggi di programmazione offrono la possibilità di definire funzioni/procedure ricorsive.
- Calcolo del fattoriale di un numero:

```
int fattoriale(unsigned int n) {
      if (n <= 1)
            return 1;
      else
            return n * fattoriale(n - 1);
}
```

Esempi (2)

- Alcune volte è necessario “complicare” la segnatura del metodo per renderelo ricorsivo:
- Ricerca di un elemento in un array (Java)

// cerca x in array a a partire dalla posizione pos

```
boolean search(int x, int[] a, int pos)
```

```
boolean search(int x, int[] a, int pos) {
        if (pos >= a.length)
                return false;
        if (a[pos] == x)
                return true;
        // non trovato nella posizione
        //pos vai alla prossima
        return search(x, a, pos + 1);
}
boolean search(int x, int[] a) {
        return search(x, a, 0) {
}
```

Esempi (2 in C)

- In C spesso si passa anche la dimensione dell'array
- Ricerca di un elemento in un array (C)
- Array passato come puntatore

// cerca x in array a con lunghezza n

```
bool search(int x, int* a, int n)
```

```
#include <stdbool.h>
bool search(int x, int* a, int n) {
        if (n == 0) return 0;
        if (a[0] == x) return 1;
        // non trovato nella posizione a[0]
        //vai alla prossima
        return search(x, a + 1, n - 1);
}
```

Tail recursion (first-order case)

- Function g makes a *tail call* to function f if
  - Return value of function f is return value of g
- Example

tail call

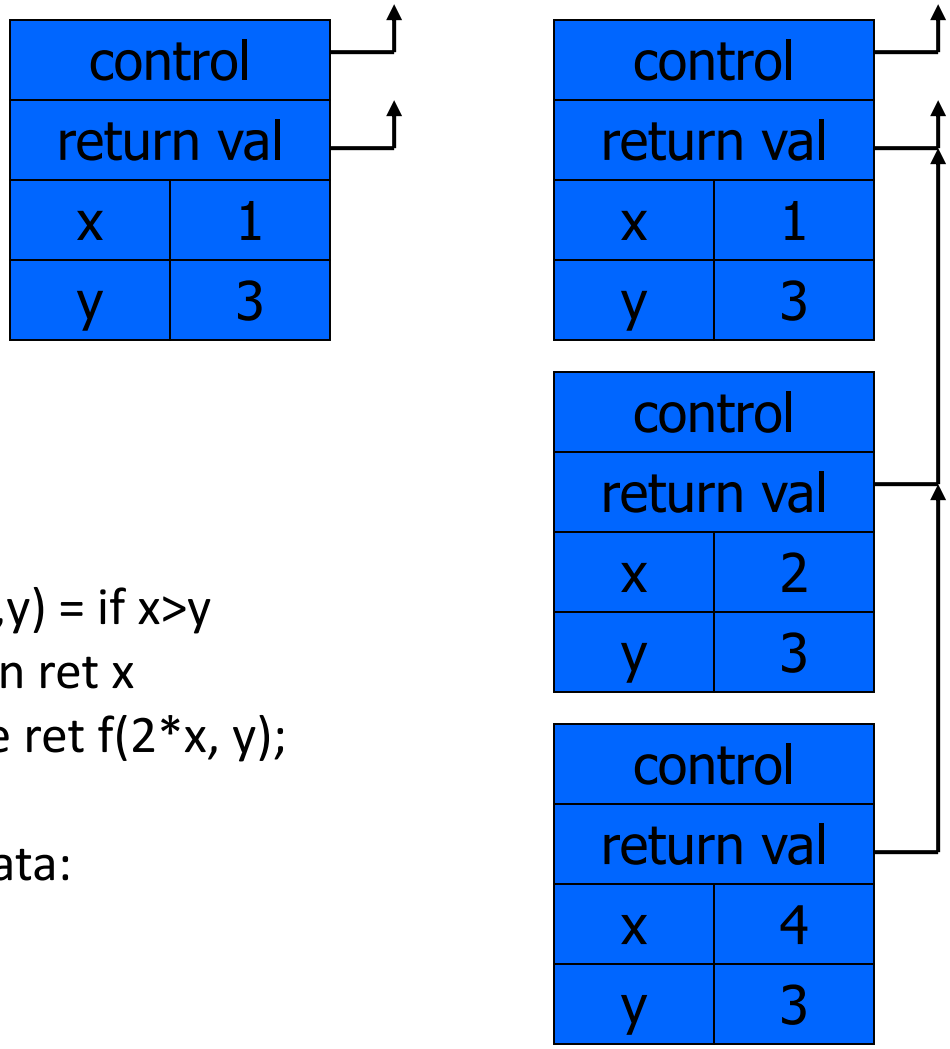
not a tail call

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

- Optimization
  - Can pop activation record on a tail call
    - Non al ritorno come teoricamente dovrebbe fare
  - Especially useful for recursive tail call
    - next activation record has exactly same form

Example Calculate least power of 2 greater than y

f(1,3)




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

Chiamata:
f(1,3)

Optimization 1

- Set return value address to that of caller

Question

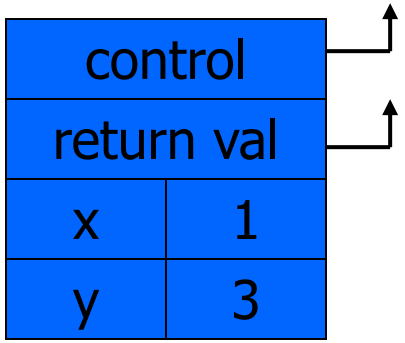
- Can we do the same with control link?

Optimization

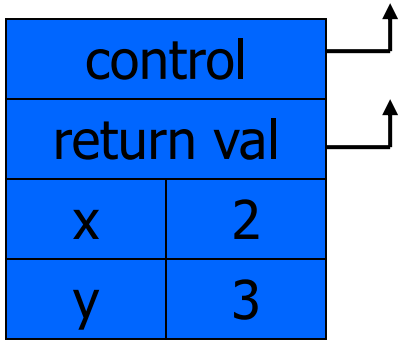
- avoid return to caller

Tail recursion elimination

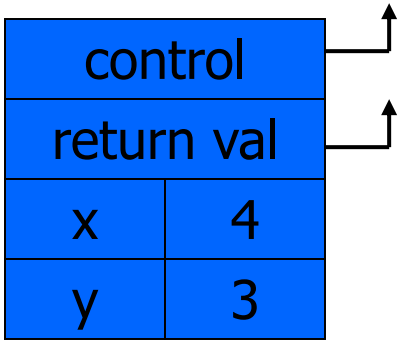
f(1,3)



f(2,3)



f(4,3)




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

Optimization

- pop followed by push = reuse activation record in place

Conclusion

- Tail recursive function equiv to iterative loop

Tail recursion and iteration

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
    then x
    else f(2*x, y);
f(1,y);
```

test

loop body

initial value

```
fun g(y)  = {
    x := 1;
    while not(x>y) do
        x := 2*x;
    return x;
};
```

Higher-Order Functions

- Language features
  - Functions passed as arguments
  - Functions that return functions from nested blocks
  - Need to maintain environment of function
- Simpler case
  - Function passed as argument
  - Need pointer to activation record “higher up” in stack
- More complicated second case
  - Function returned as result of function call
  - Need to keep activation record of returning function

Summary of scope issues

- Block-structured lang uses stack of activ records
  - Activation records contain parameters, local vars, …
  - Also pointers to enclosing scope
- Several different parameter passing mechanisms
- Tail calls may be optimized
- Function parameters/results require closures
  - Closure environment pointer used on function call
  - Stack deallocation may fail if function returned from call
  - Closures *not* needed if functions not in nested blocks