Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II Laboratorio

Procedure annidate e ricorsive

Procedure «foglia»

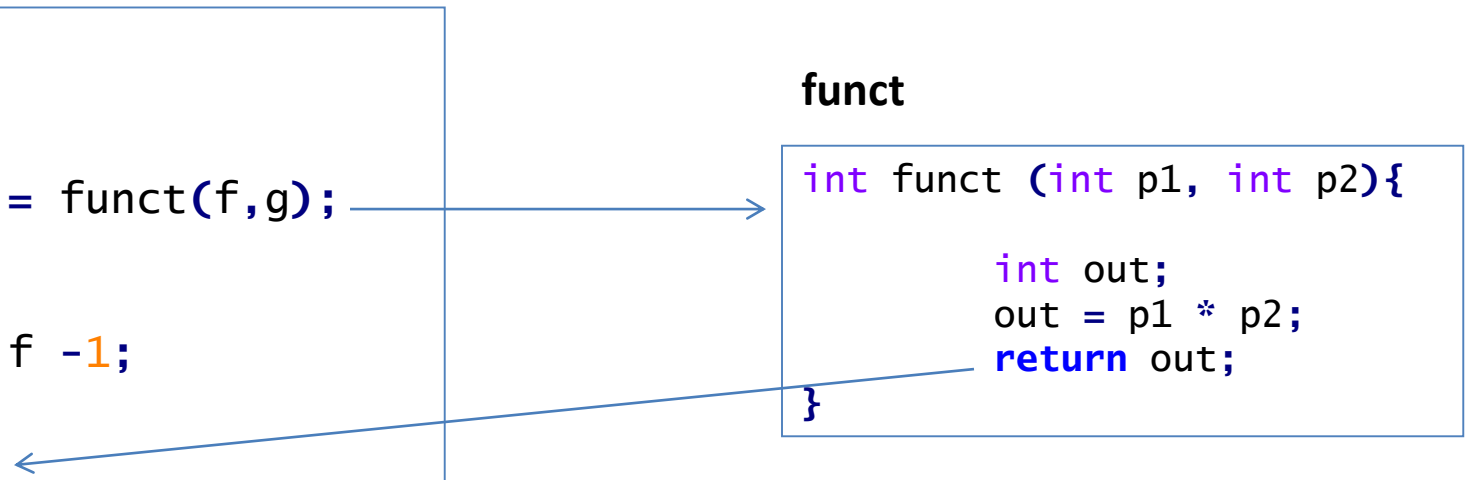
- Scenario più semplice: `main` chiama la procedura `funct` che, senza chiamare a sua volta altre procedure, termina e restituisce il controllo al `main`

main

```
f = f + 1;

if (f == g)
        res = funct(f,g);

else
        f = f -1;

print(res)
```

funct

```
int funct (int p1, int p2){

        int out;
        out = p1 * p2;
        return out;
}
```

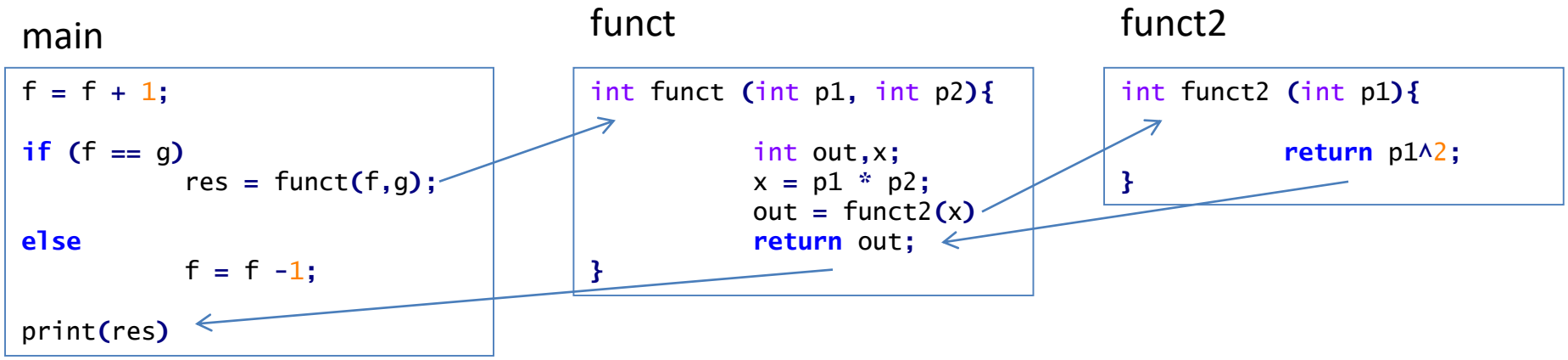


- Una procedura che non ne chiama un'altra al suo interno è detta procedura *foglia*

  Perché? Rappresentiamo le nostre procedure con un albero: le procedure diventano nodi e un arco tra due nodi x e y indica che x contiene almeno una chiamata a y

Procedure non «foglia»

- Una procedura che può invocarne un'altra durante la sua esecuzione non è una procedura foglia, ha annidata al suo interno un'altra procedura:



- Se una procedura contiene una chiamata ad un'altra procedura dovrà effettuare delle operazioni per (1) garantire la non-alterazione dei registri opportuni (2) consentire una restituzione del controllo consistente con l'annidamento delle chiamate.

- *Ricordiamo: in assembly la modularizzazione in procedure è un'assunzione concettuale sulla struttura e sul significato del codice. Nella pratica, ogni «blocco» di istruzioni condivide lo stesso register file e aree di memoria*

Procedure non «foglia»

- Supponiamo che:
  - il `main` invochi una procedura `A` passandogli `3` come parametro e cioè copiando `3` nel registro `$a0` ed eseguendo `jal A`
  - la procedura `A` chiami a sua volta una procedura foglia `B` passandogli `5` come parametro e cioè copiando `5` nel registro `$a0` e eseguendo `jal B`

- Nell'esempio sopra ci potrebbero essere problemi con `$a0` e `$ra`:
  1. `main` potrebbe necessitare di `$a0` (il suo parametro in input) anche dopo la chiamata ad `A`
  2. invocando `B`, `A` perde il suo return address (`$ra`) e non sa più a chi restituire il controllo

- Cosa previene questo problema: **la convenzione di chiamata a procedure**! Perché?

Procedure non «foglia»

- La convenzione di chiamata a procedure ci dà già il meccanismo con cui prevenire il problema attraverso la classificazione dei registri tra caller-saved and callee-saved

- `$ra` è un registro **callee-saved**: **deve** essere preservato tra chiamate a procedura

- `$a0` è un registro **caller-saved**: **non deve** essere preservato tra chiamate

- Procedimento:
  1. `main` salva sullo stack i registri di cui avrà ancora bisogno dopo la chiamata, ad esempio `$a0`
  2. `A` salva sullo stack il registro `$ra` in modo da poter restituire il controllo:
  - Una volta che `B` è terminata, `A` potrà terminare correttamente ripristinando le informazioni dallo stack

Ricorsione

- La risoluzione di un problema P è costruita sulla base della risoluzione di un sottoproblema di P

- Esempio classico: il fattoriale di n

$$n! = \prod_{k=1}^{n} k = n \prod_{k=1}^{n-1} k = n \times (n-1)!$$

- il fattoriale di n è uguale a n moltiplicato per il fattoriale di n-1, non vero se n=0! Serve una regola aggiuntiva

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

Ricorsione

- Applico la regola in cascata

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

$$4! = 4 \times (3)!$$

Ricorsione

- Applico la regola in cascata

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

$$4! = 4 \times (3)!$$

$$3! = 3 \times (2)!$$

Ricorsione

- Applico la regola in cascata

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

$$4! = 4 \times \underbrace{(3)!}$$

$$3! = 3 \times \underbrace{(2)!}$$

$$2! = 2 \times (1)!$$

Ricorsione

- Applico la regola in cascata

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

$$4! = 4 \times \underbrace{(3)!}$$
$$3! = 3 \times \underbrace{(2)!}$$
$$2! = 2 \times \underbrace{(1)!}$$
$$1! = 1 \times (0)!$$

Ricorsione

- Applico la regola in cascata

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

$$4! = 4 \times \underbrace{(3)!}$$
$$3! = 3 \times \underbrace{(2)!}$$
$$2! = 2 \times \underbrace{(1)!}$$
$$1! = 1 \times \underbrace{(0)!}$$
$$0! = 1$$

Ricorsione

- Applico la regola in cascata

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

$$4! = 4 \times \underbrace{(3)!}$$
$$3! = 3 \times \underbrace{(2)!}$$
$$2! = 2 \times \underbrace{(1)!}$$
$$1! = 1 \times \underbrace{(0)!}$$
$$0! = 1$$
$$= 4 \times 3 \times 2 \times 1 \times 1$$

Procedure ricorsive

- Procedura ricorsiva: è una procedura che per risolvere il problema P invoca se stessa per risolvere un sotto-problema di P

- In generale una procedura ricorsiva non è una procedura foglia: invoca se stessa per sua definizione

- Una procedura ricorsiva è:
  - Un callee: deve salvare i registri callee-saved ($s0, …, $ra, $fp)
  - Un caller: deve salvare i registri caller-saved ($t0, …, $a0, …, $v0, $v1)

- Al momento del ritorno dalla chiamata ricorsiva è necessario ripristinare il valore di `$ra`

Procedure ricorsive

Possono essere strutturate in diversi blocchi funzionali:

- Punto di ingresso
- Push sullo stack dei registri usati
- Check caso base / step ricorsivo
  - Caso base
  - Step ricorsivo
- Ripristino dei registri usati
- `jr $ra`

Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II Laboratorio