



Università degli Studi di Milano
Corso di Laurea in Informatica, A.A. 2017-2018

Procedure annidate

 [Homepage del corso](#)

Turno A

Nicola Basilico

Dipartimento di Informatica

Via Comelico 39/41 - 20135 Milano (MI)

Ufficio S242

nicola.basilico@unimi.it

+39 02.503.16294

Turno B

Jacopo Essenziale

Dipartimento di Informatica

Via Celoria 20 - 20133 Milano (MI)

AILab

jacopo.essenziale@unimi.it

+39 02.503.14010

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

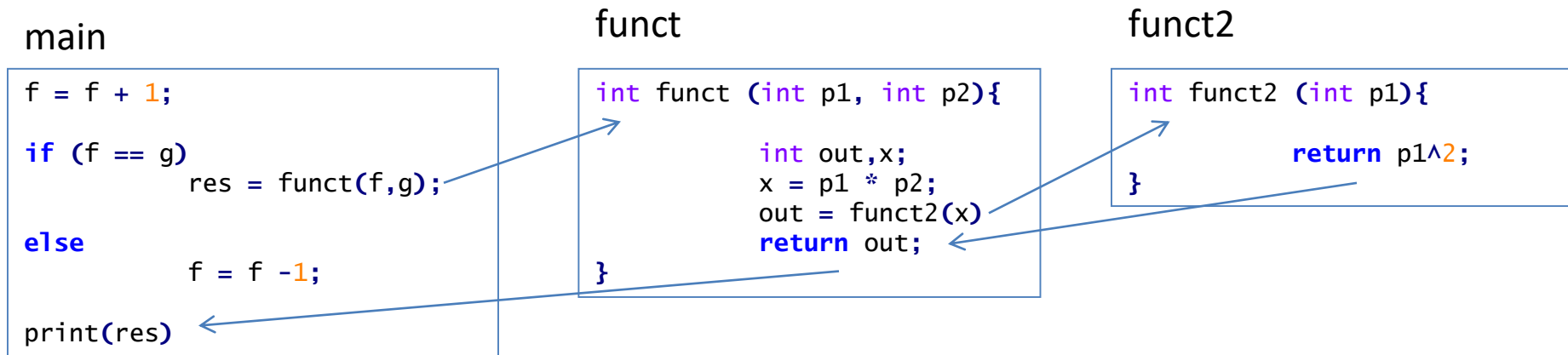
```
int funct (int p1, int p2){  
  
    int out;  
    out = p1 * p2;  
    return out;  
}
```

- 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

Esempio

- Calcolo dell'espressione $(a+b)-(c+d)$ utilizzando due chiamate annidate (codice pseudo-C):

```
main(){
    ...
    res = f1(a,b,c,d)
    ...
}

int f1(int a, int b, int c, int d){
    int t0 = a + b;
    int t1 = c + d;
    int result = f2(t0, t1);
    return(result);
}

int f2(int t0, int t1){
    retrun (t0 - t1);
}
```

Esempio

- Calcolo l'espressione $(a+b) - (c+d)$ utilizzando due procedure annidate (codice pseudo-C):

main:

```
li $s0, 2      # $s0=a=2
li $s1, 25     # $s1=b=25
li $s2, 31     # $s2=c=31
li $s3, 4      # $s3=d=4
```

```
move $a0, $s0
move $a1, $s1
move $a2, $s2
move $a3, $s3
```

```
addi $sp, $sp, -4
sw $a0, 0($sp)
```

```
jal f1
```

```
move $s5, $v0
```

f1:

```
addi $sp, $sp, -4
sw $ra, 0($sp)
```

```
add $a0, $a0, $a1
add $a1, $a2, $a3
jal f2
```

```
lw $ra, 0($sp)
addi $sp, $sp, 4
jr $ra
```

f2:

```
sub $v0, $a0, $a1
jr $ra
```

Esempio

- Calcolo l'espressione $(a+b) - (c+d)$ utilizzando due procedure annidate (codice pseudo-C):

```
main:
    li $s0, 2      # $s0=a=2
    li $s1, 25     # $s1=b=25
    li $s2, 31     # $s2=c=31
    li $s3, 4      # $s3=d=4

    move $a0, $s0
    move $a1, $s1
    move $a2, $s2
    move $a3, $s3

    addi $sp, $sp, -4
    sw $a0, 0($sp)

    jal f1

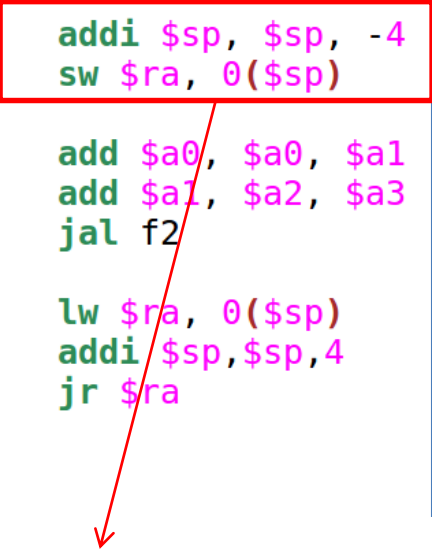
    move $s5, $v0

f1:
    addi $sp, $sp, -4
    sw $ra, 0($sp)

    add $a0, $a0, $a1
    add $a1, $a2, $a3
    jal f2

    lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra

f2:
    sub $v0, $a0, $a1
    jr $ra
```



f1 salva sullo stack tutto ciò che spetta a lei (in qualità di chiamante) così da poter terminare correttamente **ogni** sua esecuzione

Esempio

- Calcolo l'espressione $(a+b) - (c+d)$ utilizzando due procedure annidate (codice pseudo-C):

```
main:
    li $s0, 2      # $s0=a=2
    li $s1, 25     # $s1=b=25
    li $s2, 31     # $s2=c=31
    li $s3, 4      # $s3=d=4

    move $a0, $s0
    move $a1, $s1
    move $a2, $s2
    move $a3, $s3

    addi $sp, $sp, -4
    sw $a0, 0($sp)

    jal f1

    move $s5, $v0
```

```
f1:
    addi $sp, $sp, -4
    sw $ra, 0($sp)

    add $a0, $a0, $a1
    add $a1, $a2, $a3
    jal f2

    lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
```

```
f2:
    sub $v0, $a0, $a1
    jr $ra
```

dopo la chiamata ad `f2`, `f1` ripristina quanto salvato in precedenza (in questo esempio `$ra`) e poi riporta lo stack allo stato iniziale

Esercizio 7.1

- Nome del file sorgente: *prodottoConSomme.asm*

Si implementino queste procedure

1. *somma*:

- input: due interi a e b
- output: a+b

2. *prodotto_s*

- input: due interi a e b
 - output: a*b
-
- la procedura *prodotto_s* **NON** utilizzi istruzioni di moltiplicazione (*mult* et simila), ma calcoli il prodotto effettuando chiamate multiple alla procedura *somma*
 - *Esempio: il prodotto 3x2 è svolto come 3+3 oppure 2+2+2*

Esercizio 7.2

- Nome del file sorgente: *maxmin.asm*
- Si implementi la procedura *max* così definita:
 - Input: un intero *N* e un array *A* di *N* interi
 - Output: il valore massimo in *A*
- Si implementi la procedura *min* così definita:
 - Input: un intero *N* e un array *A* di *N* interi
 - Output: il valore minimo in *A*
- Si implementi la procedura *maxmin* così definita:
 - Input: un intero *N* e una matrice di interi *M* di dimensione *N* x *N*
 - Output: il massimo valore tra i minimi di riga in *M*
- Si implementi la procedura *minmax* così definita:
 - Input: un intero *N* e una matrice di interi *M* di dimensione *N* x *N*
 - Output: il minimo valore tra i massimi di riga in *M*
- Si implementi infine il *main* che acquisisca i dati, chiami *maxmin* e *minmax* e stampi i risultati



Università degli Studi di Milano
Laboratorio di Architettura degli Elaboratori II
Corso di Laurea in Informatica, A.A. 2017-2018

Nicola Basilico

Dipartimento di Informatica

Via Comelico 39/41 - 20135 Milano (MI)

Ufficio S242

nicola.basilico@unimi.it

+39 02.503.16294

Hanno contribuito alla realizzazione di queste slides:

- Iuri Frosio
- Jacopo Essenziale