Procedure annidate

Università degli studi di Milano
matteo.re@unimi.it
http://homes.di.unimi.it/re/

Procedura foglia

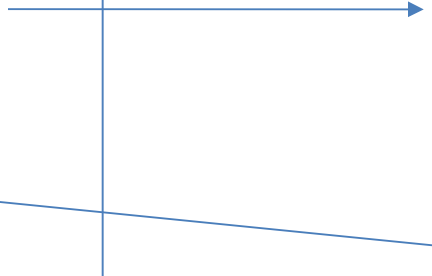
- Scenario più semplice: il main chiama una procedura, la procedura termina senza chiamare a sua volta un'altra procedura e restituisce il controllo al main

main

```
f = f + 1;

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

else
      f = f -1;

print(res)
```

funct

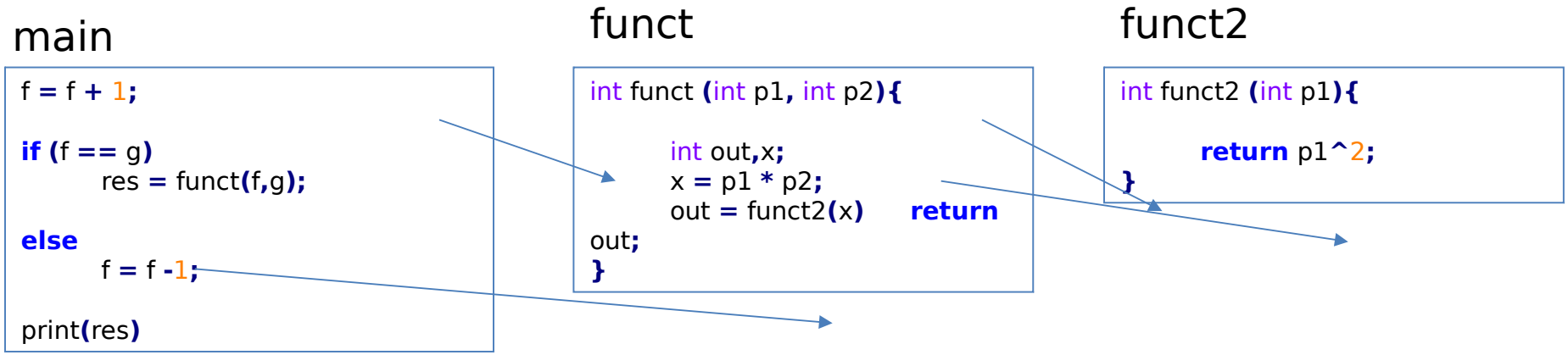
```
int funct (int p1, int p2){

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



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

Procedura annidata

- Procedure annidate: una procedura può al suo interno chiamare 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.*

Procedura annidata

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

- Ci sono potenziali conflitti su $a0 e $ra!
  1. A potrebbe necessitare di $a0 (il suo parametro in input) anche dopo la chiamata a B
  2. invocando B, A perde il suo return address e non sa più a chi restituire il controllo

- Come prevenirli? Uso dello stack:
  1. A salva sullo stack i registri di cui avrà ancora bisogno dopo la chiamata, ad esempio $a0
  2. A salva sullo stack il registro $ra
  –. Una volta che B è terminata, A potrà ripristinare le informazioni per terminare correttamente leggendo dallo stack

Esempio

- Calcolo l'espressione (a+b) – (c+d) utilizzando due procedure 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

mov $a0, $s0
mov $a1, $s1
mov $a2, $s2
mov $a3, $s3
jal f1

mov $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

mov $a0, $s0
mov $a1, $s1
mov $a2, $s2
mov $a3, $s3
jal f1

mov $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 di cui avrà bisogno per 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

mov $a0, $s0
mov $a1, $s1
mov $a2, $s2
mov $a3, $s3
jal f1

mov $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

Esempio

- Stesso esercizio con set del frame pointer:

```
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

mov $a0, $s0
mov $a1, $s1
mov $a2, $s2
mov $a3, $s3
jal f1

mov $s5, $v0
```

```
f1:
subu $sp, $sp, 8
sw $fp, 4($sp)
sw $ra, 0($sp)
addiu $fp, $sp, 4


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

lw $fp, 4($sp)
lw $ra, 0($sp)
addi $sp,$sp,8
jr $ra
```

```
f2:
subu $sp, $sp, 4
sw $fp, 0($sp)

sub $v0, $a0, $a1

lw $fp, 0($sp)

addi $sp,$sp,8

jr $ra
```

Esempio : teorema di Pitagora

Teorema Pitagora:

c = sqrt(a^2 + b^2) **c** è l'ipotenusa, **a** e **b** i cateti

Obiettivo: Scrivere programma che chiami una procedura per calcolare i lati **c** (ipotenuse) di una **serie** di triangoli rettangoli. Avremo due array aSides[] e bSides[] e i risultati saranno salvati in un array **cSides[]** . Il programma dovra' inoltre calcolare, per il set delle ipotenuse **cSides[]**, minimo, massimo, somma e media.

Esempio di:

- Utilizzo array
- Utilizzo cicli
- Chiamata a procedura che, a sua volta, chiama una terza procedura

Esempio : teorema di Pitagora

```
# Data declarations
.data

aSides:     .word   19, 17, 15, 13, 11, 19, 17, 15, 13, 11
            .word   12, 14, 16, 18, 10
bSides:     .word   34, 32, 31, 35, 34, 33, 32, 37, 38, 39
            .word   32, 30, 36, 38, 30
cSides:     .space  60

length:     .word   15
min:        .word   0
max:        .word   0
sum:        .word   0
ave:        .word   0
```

(continua...)

Esempio : teorema di Pitagora

```
# text / code section
.text
.globl   main
.ent     main
main:

# la chiamata alla funzione cSidesStats ha il seguente formato:
# cSidesStats(aSides,bSides,cSides,length,min,max,sum,ave)

la   $a0,    aSides          # passaggio per indirizzo
la   $a1,    bSides          #   “                  “
la   $a2,    cSides          #   “                  “
lw   $a3,    length          # valore length
la   $t0,    min             # indirizzo   min
la   $t1,    max             #   “         max
la   $t2,    sum             #   “         sum
la   $t3,    ave             #   “         ave
```

(continua ...)

Esempio : teorema di Pitagora

```
# preparazione chiamata (caricamento dati sullo stack)

subu    $sp,  $sp, 16
sw      $t0,     ($sp)              # push indirizzi
sw      $t1,     4($sp)
sw      $t2,     8($sp)
sw      $t3,     12($sp)

jal     cSidesStats                 # call function

addu    $sp, $sp, 16                # clear arguments

# done
li      $v0, 10
syscall                             # exit

.end main
```

Esempio : teorema di Pitagora

Funzione per il calcolo di **cSides[]** (ipotenuse di un set di triangoli
rettangoli)
Usa la funzione iSqrt() per trovare la radice quadrata intera di un
intero
Argomenti: ----------
$a0 - address of aSides[]
$a1 - address of bSides[]
$a2 - address of cSides[]
$a3 - list length
($fp) - addr of min
4($fp) - addr of max
8($fp) - addr of sum
12($fp) - addr of ave
Ritorna (passando indirizzi): ----------
cSides[]
min
max
sum
ave

Esempio : teorema di Pitagora

```
.globl   cSidesStats
.ent     cSidesStats
cSidesStats:
     subu $sp, $sp, 20   # preserve registers
     sw   $s0, 0($sp)
     sw   $s1, 4($sp)
     sw   $s2, 8($sp)
     sw   $s3, 12($sp)
     sw   $fp, 16($sp)
     sw   $ra, 20($sp)
     addu $fp, $sp, 20   # set frame pointer
```

(continua ...)

Esempio : teorema di Pitagora

```
#  Loop to calculate cSides[]
#  Note, must use $s<n> registers due to iSqrt() call
      move  $s0, $a0          # address of aSides
      move  $s1, $a1          # address of bSides
      move  $s2, $a2          # address of cSides
      li    $s3, 0            # index = 0

cSidesLoop:
      lw  $t0, ($s0)          # get aSides[n]
      mul $t0, $t0, $t0       # aSides[n]^2
      lw $t1, ($s1)           # get bSides[n]
      mul $t1, $t1, $t1       # bSides[n]^2
      add $a0, $t0, $t1
      jal iSqrt               # call iSqrt

      sw   $v0, ($s2)         # save to cSides[n]
      addu $s0, $s0, 4        # update aSides addr
      addu $s1, $s1, 4        # update bSides addr
      addu $s2, $s2, 4        # update cSides addr
      addu $s3, $s3, 1        # index++

      blt $s3, $a3, cSidesLoop # if indx<len, loop
```

Esempio : teorema di Pitagora

```
#  Loop to find minimum, maximum, and sum.
      move $s2, $a2              # start addr of cSides
      li   $t0, 0                # index = 0
      lw   $t1, ($s2)            # min = cSides[0]
      lw   $t2, ($s2)            # max = cSides[0]
      li   $t3, 0                # sum = 0

statsLoop:
      lw   $t4, ($s2)            # get cSides[n]
      bge  $t4, $t1, notNewMin   # if cSides[n] >= item -> skip
      move $t1, $t4              # set new min value

notNewMin:
      ble  $t4, $t2, notNewMax   # if cSides[n] <= item -> skip
      move $t2, $t4              # set new max value

notNewMax:
      add  $t3, $t3, $t4         # sum += cSides[n]
      addu $s2, $s2, 4           # update cSides addr

      addu $t0, $t0, 1           # index++
      blt $t0, $a3, statsLoop    # if index<len -> loop
```

Esempio : teorema di Pitagora

```
        lw $t5, ($fp)               # get address of min
        sw $t1, ($t5)               # save min

        lw $t5, 4($fp)              # get address of max
        sw $t2, ($t5)               # save max

        lw $t5, 8($fp)              # get address of sum
        sw $t3, ($t5)               # save sum

        div $t0, $t3, $a3           # ave = sum / len

        lw $t5, 12($fp)             # get address of ave
        sw $t0, ($t5)               # save ave
# Done, restore registers and return to calling routine.
        lw  $s0, 0($sp)
        lw  $s1, 4($sp)
        lw  $s2, 8($sp)
        lw  $s3, 12($sp)
        lw  $fp, 16($sp)
        lw  $ra, 20($sp)
        addu $sp, $sp, 20
        jr $ra
.end cSidesStats
```

Esempio : teorema di Pitagora

```
#  Function to computer integer square root for
#  an integer value.
#  Uses a simplified version of Newtons method.
#  x = N
#  iterate 20 times:
#  x' = (x + N/x) / 2
#  x = x'
# -----
#  Arguments
#  $a0 - N
#  Returns
#  $v0 - integer square root of N
```

(continua ...)

Esempio : teorema di Pitagora

```
.globl     iSqrt
.ent iSqrt
iSqrt:
      move     $v0, $a0       # St0 = x = N
      li     $t0, 0           # counter
sqrLoop:
      div  $t1, $a0, $v0          # N/x
      add $v0, $t1, $v0           # x + N/x
      div  $v0, $v0, 2         # (x + N/x)/2

      add $t0, $t0, 1
      blt   $t0, 20, sqrLoop

      jr     $ra
.end iSqrt
```

Esercizio 7.1

- Si implementino le seguenti 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*

- *Ricordiamo: i parametri in input vengono passati attraverso i registri $a0-$a3 (e lo stack). I valori in output vengono restituiti attraverso i registri $v0 e $v1.*

Esercizio 7.2

- Si implementi la procedura subseteq così definita:
  - Input: due array di interi A1 e A2
  - Output: 1 se ogni elemento di A1 è presente in A2, 0 altrimenti

- La procedura subseteq dovrà a sua volta utilizzare un'altra procedura belongs così definita
  - Input: un array A e un intero i
  - Output: 1 se i è contenuto in A, 0 altrimenti

- Si implementi infine il main che acquisisca i dati, chiami subseteq e raccolga i risultati

Esercizio 7.3

- 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 raccolga i risultati.

- *Suggerimento: una matrice di N x N interi può essere rappresentata con un array di dimensione N, dove ogni elemento è il base address di un array di dimensione N*