

Capitolo 8

Strutturare il controllo



Controllo del flusso

- Espressioni
 - Notazioni
 - Valutazione
 - Problemi
- Comandi
 - Assegnamento
 - Sequenziale
 - Condizionale
- Comandi iterativi
- Ricorsione

Espressioni

- Entità sintattiche la cui valutazione produce un valore oppure non termina (espressione indefinita).
- Tre notazioni principali per la sintassi:
 - infissa $a + b$
 - prefissa (polacca) $+ a b$
 - postfissa (polacca inversa) $a b +$

Semantica delle espressioni: notazione infissa

- Precedenza fra gli operatori:

`a + b * c ** d ** e / f ??`

`if A < B and C < D then ??`

(in Pascal Errore se A,B,C,D non sono tutti booleani)

- Di solito operatori aritmetici precedenza su quelli di confronto che hanno precedenza su quelli logici (non in Pascal)
- APL, Smalltalk: tutti gli operatori hanno eguale precedenza: si devono usare le parentesi

Semantica delle espressioni: notazione infissa

- Associatività

15 - 4 - 3 ??

(15 - 4) - 3

5 ** 2 ** 3 ??

5 ** (2 ** 3)

- Non sempre ovvio: in APL, ad esempio,

15 - 4 - 3

è interpretato come

15 - (4 - 3) !

Semantica delle espressioni: notazione infissa

- Ricapitolando
 - Regole di precedenza
 - Regole di associatività
 - Necessità di usare comunque le parentesi in alcuni casi: ad esempio in
$$(15 - 4) * 3$$

La valutazione di un'espressione infissa non è semplice ...

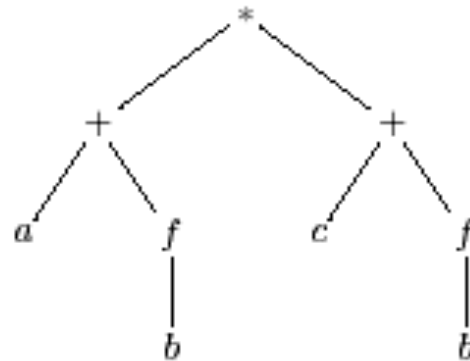
Semantica delle espressioni: notazione postfissa

- Molto più semplice della infissa:
 - non servono regole di precedenza
 - non servono regole di associatività
 - non servono le parentesi
 - valutazione semplice usando una pila:
 1. Leggi il prossimo simbolo dell'exp. e mettilo sulla pila
 2. Se il simbolo letto è un operatore:
 - applica a operandi immediatamente precedenti sulla pila,
 - memorizza il risultato in R,
 - elimina operatore ed operandi dalla pila
 - memorizza il valore di R sulla pila.
 3. Se la sequenza da leggere non è vuota torna a (1).
 4. Se il simbolo letto è un operando torna a (1).

Semantica delle espressioni: notazione prefissa

- Molto più semplice della infissa:
 - non servono regole di precedenza
 - non servono regole di associatività
 - non servono le parentesi
 - valutazione semplice usando una pila (ma più complicata di quella della postfissa: dobbiamo contare gli operandi che vengono letti)

Valutazione delle espressioni



- Le espressioni internamente sono rappresentate da alberi (visite diverse dell'albero producono le varie notazione lineari)
- A partire dall'albero il compilatore produce il codice oggetto oppure l'interprete valuta l'espressione. L'ordine di valutazione delle sottoespressioni è importante per vari motivi:
 - Effetti collaterali
 - Aritmetica finita
 - Operandi non definiti
 - Ottimizzazione

Effetti collaterali

- $(a + f(b)) * (c + f(b))$

Il risultato della valutazione da sinistra a destra può essere diverso di quello da destra a sinistra (perché)?

- In Java è specificato chiaramente l'ordine (da sinistra a destra)
- In alcuni linguaggi non sono ammesse funzioni con effetti collaterali nelle espressioni

Operandi non definiti

- In C l'espressione

`a == 0 ? b : b/a`

presuppone una valutazione lazy: si valutano solo gli operandi strettamente necessari.

- E' importante sapere se il linguaggio adotta una valutazione lazy oppure eager (tutti gli operandi sono comunque valutati)

Valutazione corto-circuito

- Nel caso delle espressioni booleane spesso la valutazione lazy è detta corto-circuito:

```
a == 0 || b/a > 2
```

- Con valutazione lazy (corto circuito, come in C) => VERO
- Con valutazione eager => possibile errore

```
p := lista;  
while (p <> nil ) and (p^.valore <> 3) do  
    p := p^.prossimo;
```

- Con valutazione eager (come in Pascal) => ERRORE

Comandi

- Entità sintattiche la cui valutazione non necessariamente restituisce un valore
 - può avere un effetto collaterale (modifica dello stato della computazione senza restituzione di un valore)
- I comandi
 - sono tipici del paradigma imperativo
 - non sono presenti nei paradigmi funzionale e logico
 - in alcuni casi restituiscono un valore (es. = in C)

Variabile

- In matematica un'incognita che può assumere i valori di un insieme predefinito
 - non è modificabile !
- Nei linguaggi imperativi: (Pascal, C, Ada, ...):
variabile modificabile
 - un contenitore di valori che ha un nome

X 2

- il valore nel contenitore può essere modificato

Assegnamento

- Comando che modifica il valore di una variabile (modificabile)
- Normalmente la valutazione di un assegnamento non restituisce un valore ma produce un effetto collaterale (in C restituisce valore)

$X := 2$

$X = X + 1$

Diverso ruolo di X e X :

- X è un l-value (denota una locazione)
- X è un r-value (può essere contenuto in una locazione)

- In generale

exp1 Opass exp2

Modelli di variabile diversi

- Linguaggi funzionali (Lisp, ML, Haskell, Smalltalk): una variabile denota un valore e non è modificabile
- Linguaggi logici: una variabile è modificabile solo entro certi limiti (istanziamento)
- Clu: modello a oggetti, chiamato anche modello a riferimento
 - Una variabile è un riferimento ad un valore, che ha un nome
 - Analogo al puntatore ma senza le possibilità di manipolazione esplicita delle locazioni
- Java:
 - variabile modificabile per i tipi primitivi (interi, booleani ecc.)
 - modello a riferimento per i tipi classe

Operatori di assegnamento

- $X := X+1$
 - doppio accesso alla locazione di a (a meno di ottimizzazione del compilatore)
 - poco chiaro; in alcuni casi può causare errori

$A[\text{index}(i)] := A[\text{index}(i)] + 1$
($\text{index}(i)$ potrebbe causare un side effect)


No

$j := \text{index}(i)$
 $A[j] := A[j] + 1;$

Meglio

- { Alcuni linguaggi usano opportuni operatori di assegnamento }

Operatori di assegnamento

- $X := X + 1$  $X += 1$ (Pascal)
 $X += 1$ (C)
- In C 10 diversi operatori di assegnamento, incremento/decremento prefissi e postfissi:
++e --e e++ e--
- L'incremento di un puntatore in C tiene conto della dimensione degli oggetti puntati

Espressioni e comandi (l. imperativi)

- Algol 68: expression oriented
 - non c'è nozione separata di comando
 - ogni procedura restituisce un valore
- Pascal: comandi separati da espressioni
 - un comando non può comparire dove è richiesta un'espressione
 - e viceversa
- C: comandi separati da espressioni
 - espressioni possono comparire dove ci si aspetta un comando
 - assegnamento (=) permesso nelle espressioni

Ambiente e memoria

- Nei linguaggi imperativi sono presenti tre importanti domini semantici:
 - Valori Denotabili (quelli a cui si può dare un nome)
 - Valori Memorizzabili (si possono memorizzare)
 - Valori Esprimibili (risultato della valutazione di una exp.)
- La semantica dei linguaggi imperativi usa
 - Ambiente: Nomi ----> Valori Denotabili
 - Memoria: Locazioni ---> Valori Memorizzabili
 - Permettono di rappresentare l'aliasing
- I linguaggi funzionali usano solo l'ambiente

Comandi per il controllo sequenza

- Comandi per il controllo sequenza esplicito
 - ;
 - blocchi
 - goto
- Comandi condizionali
 - if
 - case
- Comandi iterativi
 - iterazione determinata (for)
 - iterazione indeterminata (while)

Comando sequenziale e blocchi

- $C1 ; C2$
 - E' il costrutto di base dei linguaggi imperativi
 - Ha senso solo se ci sono side-effects
 - in alcuni linguaggi il ``;'`' è un terminatore
- Algol 68, C: Il valore di un comando composto e' quello dell'ultimo comando.
- Comando composto
 - può essere usato al posto di un comando semplice
 - Algol 68, C (no distinzione espressione-comando): il valore di un comando composto e' quello dell'ultimo comando

{
...
}

begin

...

end

GOTO

- Accesso dibattito negli anni 60/70 sulla utilità del goto

```
if a < b goto 10
...
10: ...
```
 - Considerato utile essenzialmente per
 - uscita dal centro di un loop
 - ritorno da sottoprogramma
 - gestione eccezioni
 - Alla fine considerato dannoso (contrario ai principi della programmazione strutturata)
 - I moderni linguaggi
 - usano altri costrutti per gestire il controllo dei loop e dei sottoprogrammi
 - usano un meccanismo strutturato di gestione eccezioni
 - Goto non e' presente in Java
- [1] E. Dijkstra. Go To statements considered Harmful. Communications of the ACM, 11(3): 147-148. 1968.

Comando condizionale

```
if B then c_1 else c_2
```

- Introdotto in Algol 60
- Varie regole per evitare ambiguità in presenza di if annidati:
 - Pascal, Java: else associa con il then non chiuso più vicino
 - Algol 68, Fortran 77: parola chiave alla fine del comando
- Rami multipli espliciti

```
if Bexp1 then C1
    elseif Bexp2 then C2
    ...
    elseif Bexpn then Cn
    else Cn+1
endif
```

- La valutazione dell'espressione booleana di controllo può essere ottimizzata dal compilatore: Short-circuit

Case

```
case exp of
```

```
| label_1 : c_1
```

```
| label_2 : c_2
```

```
...
```

```
| label_n : c_n
```

```
else          c_{n+1}
```

exp: espressione a valori discreti

etichette: valori costanti, disgiunti

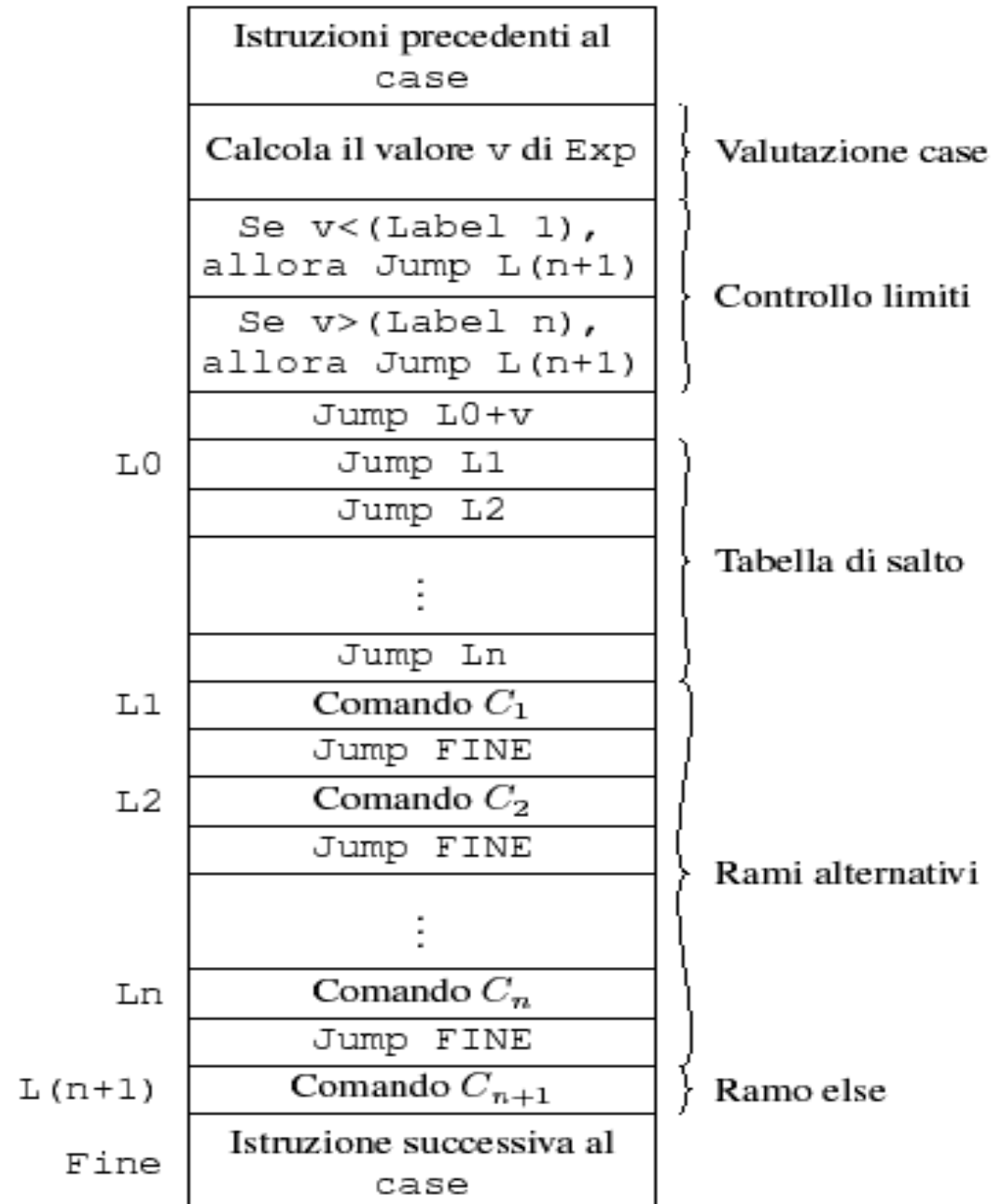
di tipo compatibile con exp

- Molte versioni nei vari linguaggi
 - Pascal, C: no range nella lista delle etichette;
 - Pascal: ogni ramo contiene comando singolo, no ramo default;
 - Modula, Ada, Fortran: ramo di default;
 - Ada: etichette coprono tutti i possibili valori nel dominio del tipo exp;
 - C, Fortran90: se valore exp non in val_i intero comando = null
 - ...
- Più efficiente di `if` multiplo se compilato in modo astuto ...

Compilazione del case

```

case   exp of
|   label_1 : C_1
|   label_2 : C_2
...
|   label_n : C_n
else C_{n+1}
    
```



Sintassi di C, C++ e Java

- **switch** *exp* {
 - case** 1: *C*₁
 break;
 - case** 2: *C*₂;
 break; ...
 - case** *k*: *C*_{*k*};
 break;
 - default**: *C*_{*k*+1};
 break;
 - }
- Range e liste di etichette non ammesse
- Si possono ottenere usando rami con corpo vuoto (senza break)

Iterazione

- **Iterazione** e **ricorsione** sono i due meccanismi che permettono di ottenere formalismi di calcolo Turing completi. Senza di essi avremmo automi a stati finiti
- Iterazione
 - **indeterminata**: cicli controllati logicamente
(`while`, `repeat`, ...)
 - **determinata** cicli controllati numericamente
(`do`, `for` ...) con numero di ripetizioni del ciclo determinate al momento dell'inizio del ciclo

Iterazione indeterminata

```
while condizione do comando
```

- Introdotto in Algol-W, rimasto in Pascal e in molti altri linguaggi, piu' semplice semanticamente del `for`
- In Pascal anche versione post-test:

```
repeat comando untill condizione
```

equivalente a

```
comando;
```

```
while not condizione do comando;
```

Iterazione indeterminata

- Indeterminata perché il numero di iterazioni non è noto a priori
- L'iterazione indeterminata permette il potere espressivo delle MdT
- È di facile implementazione usando l'istruzione di salto condizionato della macchina fisica

Iterazione determinata

```
FOR indice : = inizio TO fine BY passo DO  
    ...  
END
```

- non si possono modificare `indice`, `inizio`, `fine`, `passo` all'interno del loop
- è **determinato** (al momento dell'inizio dell'esecuzione del ciclo) il numero di ripetizioni del ciclo
- il potere espressivo è minore rispetto all'iterazione indeterminata: non si possono esprimere computazioni che non terminano
- in molti linguaggi (ad esempio C) il `for` non è un costrutto di iterazione determinata

Semantica del for

- Supponendo **passo** positivo:
- 1. valuta le espressioni **inizio** e **fine** e ``congela" i valori ottenuti
- 2. inizializza **I** con il valore di **inizio**;
- 3. se $I > \mathbf{fine}$ termina l'esecuzione del **for**
altrimenti
 - si esegue corpo e si incrementa **I** del valore di **passo**;
 - si torna a (3).

Passo negativo

- Comando esplicito, come `downto` (Pascal) e `reverse` (Ada)
 - il test in (3) diventa `l < fine`
- Nessuna sintassi speciale: si usa `ic` (iteration count, Fortran 77 e 90):

$$ic = \left\lfloor \frac{fine - inizio + passo}{passo} \right\rfloor$$

- Si decrementa `ic` fino a raggiungere il valore 0

Iterazione controllata numericamente

```
FOR indice : = inizio TO fine  
                BY passo DO ... END
```

I vari linguaggi differiscono nei seguenti aspetti:

1. Possibilità di **modificare gli indici** primo, ultimo, passo nel loop (se sì, non si tratta di iterazione determinata)
2. **Numero di iterazioni** (dove avviene il controllo `indice < fine`)
3. Incremento **negativo**
4. **Valore** di `indice` al termine del ciclo
5. Possibilità di **salto** dall'esterno all'interno

Ricorsione

- Modo alternativo all'iterazione per ottenere il potere espressivo delle MdT
- Intuizione: una funzione (procedura) è ricorsiva se definita in termini di se stessa.
- Esempio (abusato): il fattoriale

- Corrisponde alla definizione

fattoriale
fattoriale

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

Definizioni induttive (intermezzo)

- Numeri naturali $0, 1, 2, 3, \dots$. Minimo insieme X che soddisfa le due regole seguenti (Peano):
 1. 0 è in X ;
 2. Se n è in X allora $n + 1$ è X ;
- Principio di induzione. Una proprietà $P(n)$ è vera su tutti i numeri naturali se
 1. $P(0)$ è vera;
 2. Per ogni n , se $P(n)$ è vera allora è vera anche $P(n + 1)$.
- Definizioni induttive. Se $g: (\text{Nat} \times A) \rightarrow A$ totale allora esiste una unica funzione totale $f: \text{Nat} \rightarrow A$ tale che
 1. $f(0) = a$;
 2. $f(n + 1) = g(n, f(n))$.
- Si può generalizzare: well founded induction.

Ricorsione e definizioni induttive

- Funzione ricorsiva F analoga alla definizione induttiva di F:
 - il valore di F su n è definito in termini dei valori di F su $m < n$
- Tuttavia nei programmi sono possibili definizioni non ``corrette``:
 - le seguenti scritture non definiscono alcuna funzione

`fie(1) = fie(1)`

`foo(0) = foo(0)`

`foo(n) = foo(n+1)`

- invece i seguenti programmi sono possibili

```
int fie1 (int n){  
    if (n == 1) return fie1(1);  
}
```

```
int fool (int n){  
    if (n == 0)  
        return 1;  
    else  
        return fool(n) + 1;  
}
```

Ricorsione e iterazione

- La ricorsione è possibile in ogni linguaggio che permetta
 - funzioni (o procedure) che possono chiamare se stesse
 - gestione dinamica della memoria (pila)
- Ogni programma ricorsivo (iterativo) può essere tradotto in uno equivalente iterativo (ricorsivo)
 - ricorsione più naturale con linguaggi funzionali e logici
 - iterazione più naturale con linguaggi imperativi
- In caso di implementazioni naif ricorsione meno efficiente di iterazione tuttavia
 - optimizing compiler può produrre codice efficiente
 - tail-recursion ...

Ricorsione in coda (tail recursion)

- Una chiamata di g in f si dice “in coda” (o tail call) se f restituisce il valore restituito da g senza ulteriore computazione.

- f è tail recursive se contiene solo chiamate in coda

```
function tail_rec (n: integer): integer  
begin ... ; x:= tail_rec(n-1) end
```

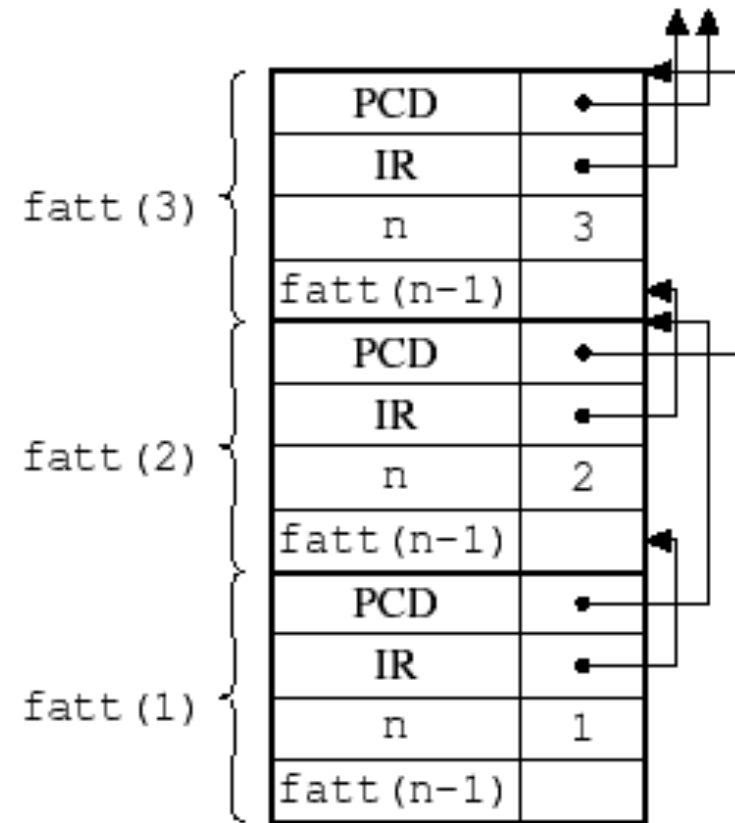
```
function non_tail_rec (n: integer): integer  
begin ... ; x:= non_tail_rec(n-1); y:= g(x) end
```

- Non serve allocazione dinamica della memoria con pila: basta un
unico RdA !
- Più efficiente
- Possibile la generazione di codice tail-recursive usando continuation passing style

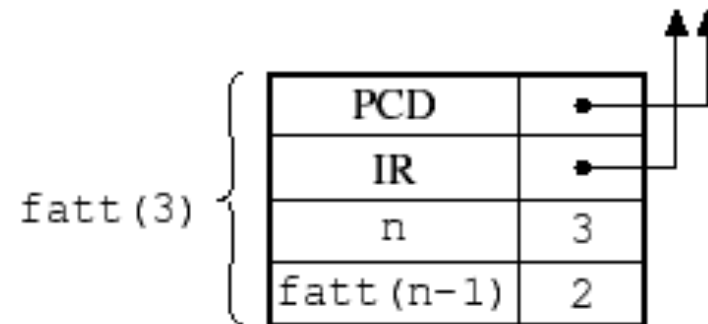
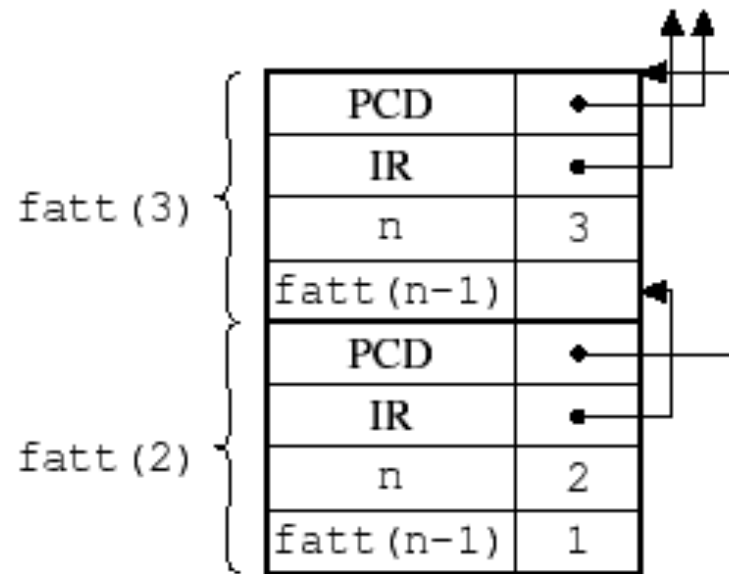
Esempio: il caso del fattoriale

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

Situazione dei RdA
Dopo la chiamata di f(3) e le
successive chiamate ricorsive



Esempio: il caso del fattoriale



Una versione tail-recursive del fattoriale

- Cosa accade con la seguente funzione ?

```
int fattrc (int n, int res){  
    if (n <= 1)  
        return res;  
    else  
        return fattrc(n-1, n * res)  
}
```

- Abbiamo aggiunto un parametro per memorizzare ``il resto della computazione’’
- Basta un unico RdA
 - Dopo ogni chiamata il RdA può essere eliminato

Un altro esempio: numeri di Fibonacci

- Definizione:

$\text{Fib}(0) = 0;$

$\text{Fib}(1) = 1;$

$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

```
int fib (int n){  
    if (n == 0)  
        return 0;  
    else  
        if (n == 1)  
            return 1;  
        else  
            return fib(n-1) + fib(n-2);  
}
```

- Complessità in tempo e spazio esponenziale (ad ogni chiamata due nuove chiamate)

Una versione più efficiente per Fibonacci

- La versione tail-recursive

```
int fibrc (int n, int res1, int res2){  
    if (n == 0)  
        return res2;  
    else  
        if (n == 1)  
            return res2;  
        else  
            return fibrc(n-1, res2, res1+res2);  
}
```

- Complessità
 - in tempo lineare in n
 - in spazio costante (un soloRdA)