

Strutturare il controllo

Espressioni, comandi, iterazione, ricorsione

M. Gabrielli, S. Martini

Linguaggi di programmazione:

principi e paradigmi

McGraw-Hill Italia, 2005

Controllo del flusso

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

Espressioni

- Un'espressione un'entità sintattica la cui valutazione produce un valore oppure non termina, nel qual caso l'espressione è indefinita.
- Sintassi delle espressioni: tre notazioni principali

- 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

Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), & (address of), * (contents of), ! (logical not), ~ (bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*, /, mod, rem
+, -	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)		<, >, <=, >= (inequality tests)	=, /=, <=, >, >= (comparisons)
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	
		(bit-wise inclusive or)	
.and.		&& (logical and)	and, or, xor (logical operators)
.or.		(logical or)	
.eqv., .neqv. (logical comparisons)		?: (if...then...else)	
		=, +=, -=, *=, /=, %= >>=, <<=, &=, ^=, = (assignment)	
		, (sequencing)	

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

$$(15 - 4) * 3$$

le parentesi sono essenziali

- 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

Semantica delle espressioni: notazione postfissa

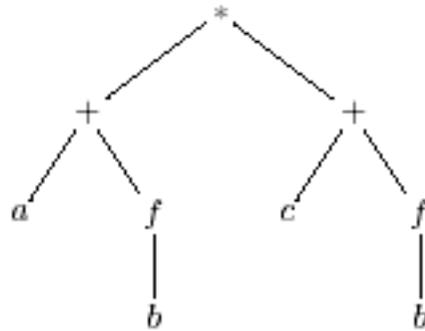
- Valutazione usando una pila
 1. Leggi il prossimo simbolo dell'exp. e mettilo sulla pila
 2. Se il simbolo letto è un operatore:
 - applica a operandi immediatamente precedenti sulla pila,
 - memorizza il risultato in R,
 - elimina operatore ed operandi dalla pila
 - memorizza il valore di R sulla pila.
 3. Se la sequenza da leggere non è vuota torna a (1).
 4. Se il simbolo letto un operando torna a (1).

Occorre conoscere l'arietà di ogni operando !

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:
 - Simmetrica -> infissa
 - Anticipata -> prefissa
 - Differita -> postfissa

Valutazione delle espressioni

- A partire dall'albero il compilatore produce il codice oggetto oppure l'interprete valuta l'espressione
- In entrambi i casi l'ordine di valutazione delle sottoespressioni è importante per vari motivi:
 - Effetti collaterali
 - Aritmetica finita
 - Operandi non definiti
 - Ottimizzazione

Effetti collaterali

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

Se f modifica b il risultato da sinistra a destra è diverso di quello da destra a sinistra

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

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

- Un comando è un' entità sintattica la cui valutazione non necessariamente restituisce un valore, ma può avere un effetto collaterale.
 - 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)

Variabili

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

X

- il valore nel contenitore può essere modificato mediante il comando di assegnamento.

Assegnamento

- Comando che modifica il valore di una variabile (modificabile)

$X := 2$

$X = X + 1$

Si noti il diverso ruolo di X e X

- X è un **l-value**, ossia un valore che denota una locazione (e può comparire a sinistra di un assegnamento)
- X è un **r-value** valore, ossia un valore può essere contenuto in una locazione (e può comparire a destra di un assegnamento)

- In generale

$exp1 \text{ Opass } exp2$

Assegnamento

- Normalmente la valutazione di un assegnamento non restituisce un valore ma produce un ``side effect'' (effetto collaterale)
 - In alcuni linguaggi l'assegnamento restituisce anche un valore. In C $x = 2$ restituisce 2 quindi possiamo scrivere

$Y = X = 2$

- Nei linguaggi imperativi la computazione avviene mediante effetti collaterali

Modelli di variabile diversi

- Linguaggi funzionali (Lisp, ML, Haskell, Smalltalk): modello analogo a quella della matematica. Una variabile denota un valore e non è modificabile
- Linguaggi logici: modello analogo a quello dei funzionali, ma con la possibilità di modificare (entro certi limiti) il valore associato alla variabile
- Clu: modello a oggetti, chiamato anche modello a riferimento
- Java:
 - variabile modificabile per i tipi primitivi (interi, booleani ecc.)
 - modello a riferimento per i tipi classe

Modello a riferimento

- Una variabile è un riferimento ad un valore, che ha un nome

X \longrightarrow 2

- Analogo alla nozione di puntatore, ma senza le possibilità di manipolazione delle locazioni dei puntatori: le locazioni qui possono essere manipolate solo implicitamente

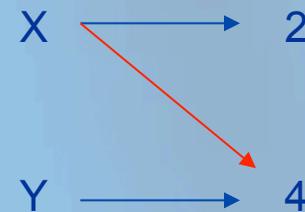
Assegnamento con il modello a riferimento

- Var modificabile

X = 2; X 4

Y = 4; Y 4

Modello a riferimento



X = Y

- A destra, se gli oggetti riferiti da X e Y sono modificabili (es. oggetti Java) modifiche fatte attraverso la X si riflettono sull'oggetto riferito da Y

Operatori di assegnamento

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

`A[index(i)] := A[index(i)] +1` No

(`index(i)` potrebbe causare un side effect)

$\left\{ \begin{array}{l} j := \text{index}(i) \\ A[j] := A[j] + 1; \end{array} \right.$ Meglio

- Per evitare questi problemi alcuni linguaggi usano opportuni operatori di assegnamento

Operatori di assegnamento

- $X := X + 1$ \longrightarrow $X += 1$ (Pascal)
 $X += 1$ (C)
- In C 10 diversi operatori di assegnamento, incremento/decremento prefissi e postfissi:
 - $++e$ ($--e$): incrementa (decrementa) e prima di fornire il valore al contesto
 - $e++$ ($e--$): incrementa (decrementa) e dopo aver fornito il valore al contesto
- L'incremento di un puntatore tiene conto della dimensione degli oggetti puntati
 $p += 3$ incrementa il puntatore p di $3n$ bytes, n dimensione oggetto puntato

Associativita' assegnamento

- In generale:

$a = b = c$

- Clu, ML, Perl

$a, b := c, d$ (oppure $a, b = c, d$)

$a, b := b, a$ (non servono variabili ausiliarie)

$a, b, c := \text{pippo}(d, e, f)$

Espressioni e comandi (I. imperativi)

- Algol 68: expression oriented
 - non c'e' nozione separata di comando
 - ogni procedura restituisce un valore

```
begin
  a:= if b< c then d else e;
  a:= begin f(b); g(c) end;
  g(d);
  2+3
end
```

- Pascal: comandi separati da espressioni
 - un comando non puo' comparire dove e' richiesta un'espressione
 - e viceversa
- C: comandi separati da espressioni
 - espressioni possono comparire dove ci si aspetta un comando
 - assegnamento (=) permesso nelle espressioni

```
if (a == b){ ...          if (a = b) { ....
/* se a = b fai il resto ..   assegna b ad a e se il risultato non
                             e' zero fai il resto ...
```

Ambiente e memoria

- Due variabili diverse possono denotare lo stesso oggetto (aliasing)
 - come si rappresenta questa situazione in termini di stato ?
 - la semplice funzione Stato: Nomi ---> Valori non basta
- 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
- 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 ";" più che un comando sequenziale 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
 - gestire eccezioni
- Alla fine considerato dannoso
- I moderni linguaggi
 - usano altri costrutti per gestire il controllo dei loop e dei sottoprogrammi (while, for, if then else, procedure ...vedi Algol 60)
 - usano un meccanismo strutturato di gestione eccezioni (Clu, Ada, C++, Lisp, Haskell, Java, Modula 3)
 - Goto non e' presente in Java

[1] E. Dijkstra. Go To statements considered Harmul. Communications of the ACM, 11(3): 147-148. 1968.

Programmazione strutturata

- Goto ``sconfitto’’ perche’ considerato contrario ai principi della programmazione strutturata
- Programmazione strutturata: anni 70, antesignana della programmazione object oriented
 - design top-down (raffinamenti successivi) o bottom-up
 - codice modulare
 - nomi identificatori significativi
 - uso esteso commenti
 - tipi di dato strutturati (array, record ..)
 - comandi per il controllo strutturati
 - ...

Comandi di controllo strutturati

- Un solo punto di ingresso e un solo punto di uscita
 - la scansione lineare del testo corrisponde al flusso di esecuzione
 - fondamentale per la comprensione del codice
- Comandi strutturati
 - `for`, `if`, `while`, `case` ...
 - non è il caso del `goto`
- Permette codice strutturato e non ``spaghetti code''

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

```
if B then C_1 else C_2 endif
```

- 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

Short Circuit

Pascal

```
if ((A > B) and (C > D)) or (E <> F) then
    then_clause
else
    else_clause
```

Codice non ottimizzato

```
r1 := A          -- load
r2 := B
r1 := r1 > r2
r2 := C
r3 := D
r2 := r2 > r3
r1 := r1 & r2
r2 := E
r3 := F
r2 := r2 <> r3
r1 := r1 | r2
if r1 = 0 goto L2
L1: then_clause  -- (label not actually used)
    goto L3
L2: else_clause
L3:
```

Codice ottimizzato

```
r1 := A
r2 := B
if r1 <= r2 goto L4
r1 := C
r2 := D
if r1 > r2 goto L1
L4: r1 := E
    r2 := F
    if r1 = r2 goto L2
L1: then_clause
    goto L3
L2: else_clause
L3:
```

Case

```
case exp of
| label_1 : C_1
| label_2 : C_2
| label_n : C_n
else      C_n+1
```

Discendente del goto di Fortran e del switch di Algol 60

exp: espressione a valori discreti

etichette: valori costanti, disgiunti, di tipo compatibile con exp

- Molte versioni nei vari linguaggi
 - Modula: possibili piu' valori (in or o range) nello stesso ramo;
 - Pascal,C: no range nella lista delle etichette;
 - Pascal: ogni ramo contiene un comando singolo, no ramo default (a meno di clausola `else`);
 - 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

If o case ?

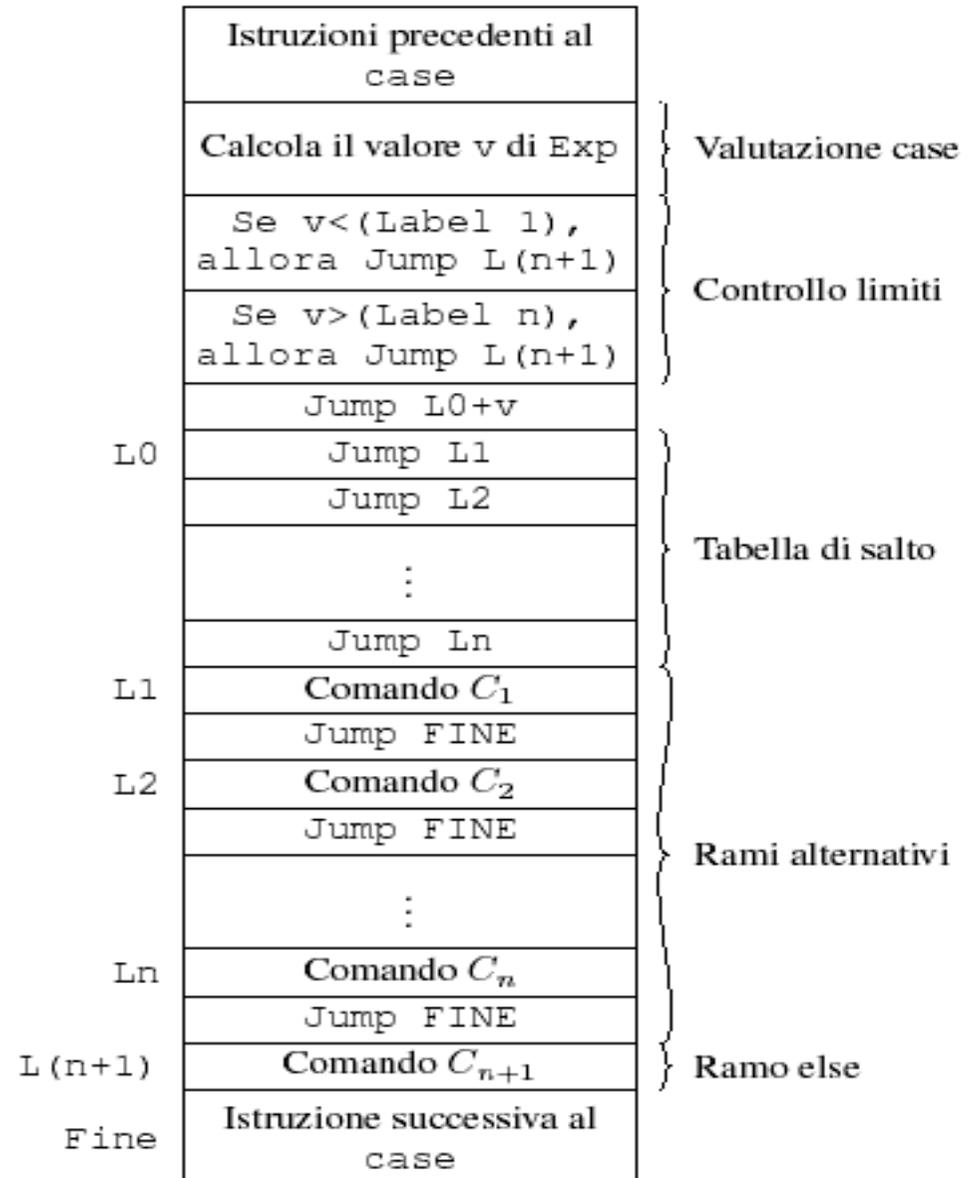
- Rispetto all'uso di `if ... then ... else` il `case exp ...` offre
 - maggiore leggibilità
 - maggiore efficienza codice prodotto, se compilato in modo astuto:
 - invece di test sequenziali come nella valutazione di `if ... then ... else`
 - calcolo indirizzo dato da `exp` e salto diretto al ramo corrispondente

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;
```

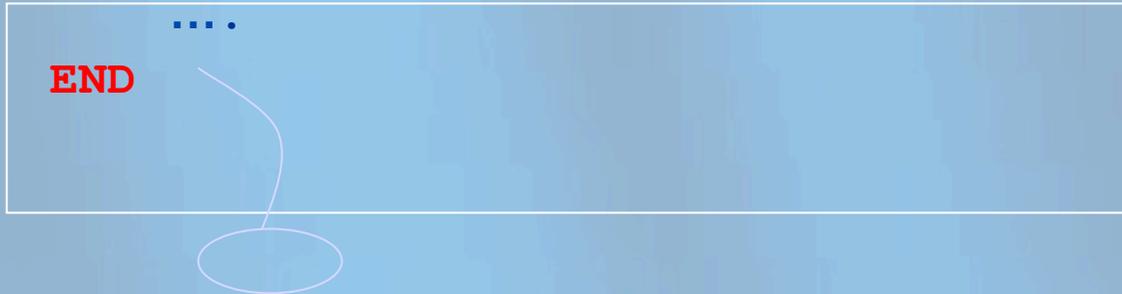
```
while not condizione do comando;
```

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
```



- 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 del punto (3) verifica, che `I` sia strettamente minore di **fine**
- Nessuna sintassi speciale: si usa `iteration count` (Fortran 77 e 90):

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

`ic` è il numero di ripetizioni del ciclo (se > 0). Si decrementa `ic` fino a raggiungere il valore 0

Cicli controllati 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 si, 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
- Il costrutto **do** di Fortran permette quasi tutto, con conseguenti problemi di leggibilità e correttezza. I linguaggi moderni moderni no

Indici del ciclo

- Nella maggior parte dei linguaggi moderni (Algol, Pascal, Ada, Fortran 77 e 90, Modula 3)
 - non possibili cambiamenti all'interno del ciclo
 - valori valutati una sola volta prima dell'inizio del ciclo
 - spesso devono essere variabili dichiarate nel blocco esterno più vicino (ISO Pascal)
 - `se inizio > fine` ciclo non eseguito
 - limiti controllati prima dell'inizio del ciclo
 - in caso di
 - passo negativo: Pascal, Ada comando esplicito (`downto` e `reverse`);
 - Fortran 77 e 90 iteration count usato dal compilatore;

Valore di `indice`. Salti

- Il valore di `indice` alla fine del ciclo
 - è l'ultimo assegnato, normalmente il primo valore che eccede il limite `fine`
 - può causare problemi di overflow non controllabili
 - è l'ultimo valore valido: codice più lento (un test in più);
 - indefinito (Pascal, Fortran IV ...);
 - elimina il problema di cui sopra
- In alcuni casi (Algol W, Algol 68, Ada, Modula-3, C++) `indice` è una variabile locale del loop, dichiarata implicitamente dal loop stesso e non visibile al di fuori di esso
- In Algol 60, Fortran 77 e molti linguaggi moderni non si può saltare all'interno di un loop usando il goto (ma se ne può uscire)

Loop in C

- In C il costrutto for è controllato logicamente:

```
FOR i := primo TO ultimo BY passo DO  
... END
```

in C diventa

```
for ( i = first; i <= last ; i += step )  
{ ... }
```

che e' equivalente a

```
i = first;  
while ( i <= last )  
{ ... ; i += step }
```

- Gli indici possono essere modificati nel corpo del ciclo
- Il controllo sul possibile overflow nella condizione di terminazione deve essere gestito a programma

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

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

- Corrisponde alla definizione induttiva

fattoriale (0) = 1.

fattoriale (n) = n*fattoriale(n-1)

Definizioni induttive

- **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

- La definizione di una funzione ricorsiva è analoga alla definizione induttiva di una funzione:
 - il valore di F su un argomento è definito in termini dei valori di F su argomenti più piccoli:
- Nei programmi tuttavia sono possibili anche 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)
- Modi alternativi per ottenere lo stesso potere espressivo:
 - 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 “chiamata 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
```

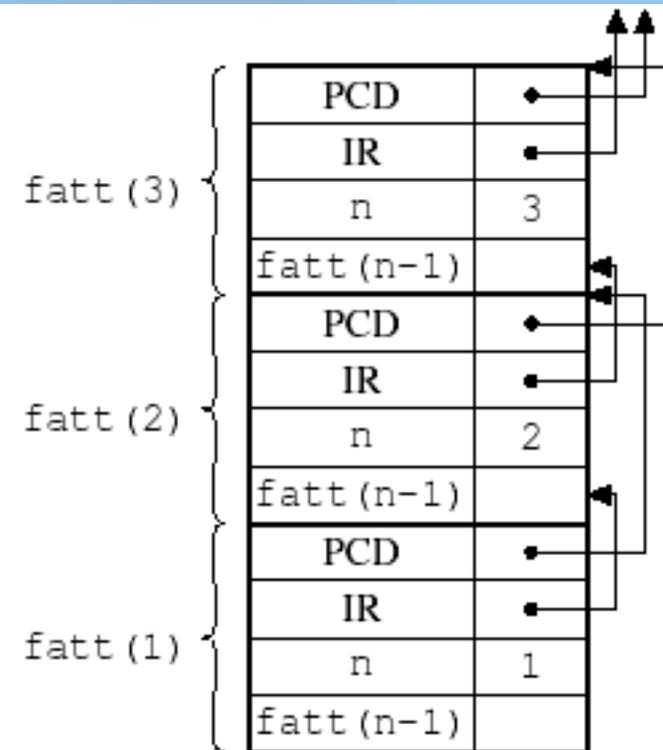
```
function non_tail_rec (n: integer): integer  
begin ... ; x:= non_tail_rec(n-1); y:= g(x) end
```

- 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

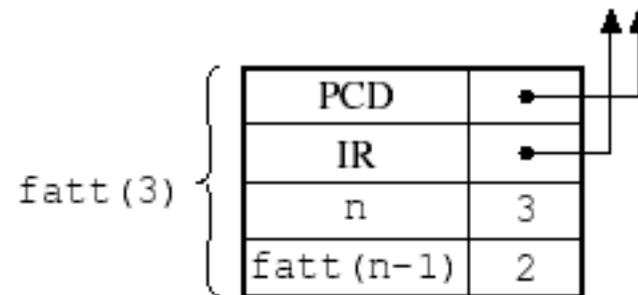
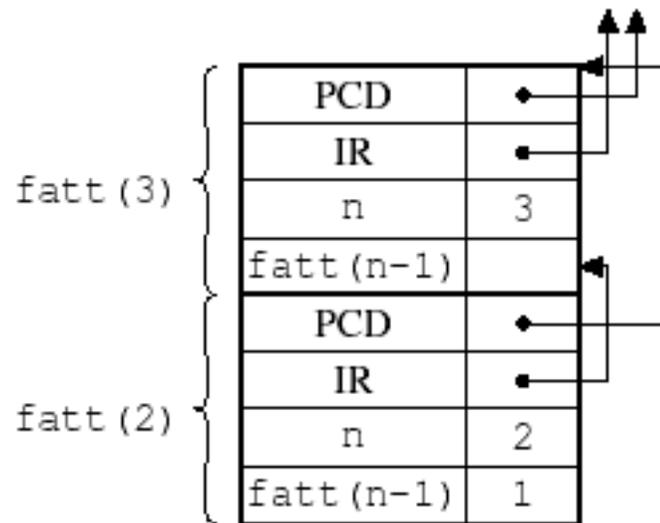
```
int fatt (int n){  
    if (n <= 1)  
        return 1;  
    else  
        return n * fatt(n-1);  
}
```

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



Esempio: il caso del fattoriale

- Evoluzione della computazione



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 in n (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)