

Strutturare il controllo

Espressioni, comandi, iterazione, ricorsione

Differenza tra comandi ed espressioni ?

ESPRESSIONE: costrutto sintattico la cui (sua) valutazione produce un valore
(se termina, poiché la valutazione potrebbe non terminare)

M. Gabbielli, 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

DEF:

- Un'espressione è 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 → OPERANDO OPERATORE OPERANDO $a + b$

più
semplici
da valutare
dalla
macchina
(rispetto
all'infissso, più facile per noi)

- prefissa (polacca)
→ OPERATORE
- postfissa (polacca)
→ OPERANDO

Semantica delle espressioni: notazione infissa

- Precedenza fra gli operatori:

a + b * c ** d ** e / f ??
precedenza su >, <

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

→ Gli operatori aritmetici sfruttano di solito le regole matematiche per stabilire le precedenze Es. $5 + (3+2)$ precedenza

più
precedenza

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

Meno precedenza

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 **VALUTAZIONE delle espressioni infisse** tiene conto di:
 - 1) – Regole di precedenza
 - 2) – Regole di associatività
 - 3) – Necessità di usare comunque le parentesi in alcuni casi: ad esempio in

$$(15 - 4) * 3 \rightarrow \text{Se no senza fa } 15 - 12$$

le parentesi sono essenziali

- ⇒ – La valutazione di un'espressione infissa non è semplice ...

Semantica delle espressioni: notazione postfissa

NOTAZIONE POLACCA INVERSA

- Molto più semplice della infissa:

Vantaggi (per la macchina)

- non servono regole di precedenza
- non servono regole di associatività
- non servono le parentesi
- valutazione semplice usando una pila

Esempio:
postfissa:
(OPERANDI OPERATORI)

15 4 - 3 *

Se dovessi fare $15 - (4 * 3)$ faresti invece:

15 4 3 * -

E se invece volessi
valutare
 $3 * (15 - 4)$
compilatore vede le
parentesi
va avanti e
indietro nella
valutazione

Semantica delle espressioni: notazione postfissa

- Valutazione usando una pila

Algoritmo di valutazione

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).

Occore conoscere l'arietà di ogni operando !

Con la notazione polacca ho un algoritmo
semplice di valutazione delle espressioni
che sfrutta una PILA

1) Ho un operando o un'operazione sulla
pila?

→ Se ho un operando, vado avanti

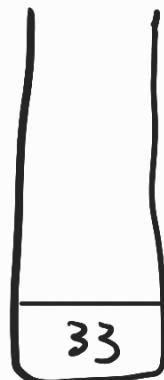
→ Se ho un'operazione, faccio l'
operazione indicata sui due operandi

(o uno è seconda dell'operando) sotto l'
operazione, cancello tutti e tre (operatori e operandi)
e metto il risultato sulla cima della
pila

--- (vedi sopra)

Es.

15 4 - 3 *

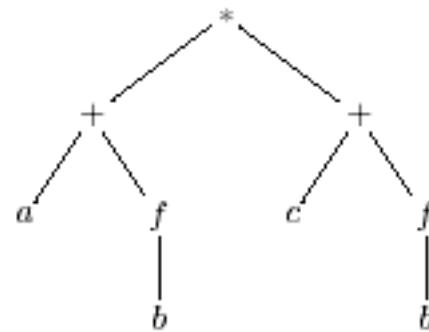


Semantica delle espressioni: notazione prefissa

NOTAZIONE POLACCA

- 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 nei linguaggi di programmazione
- Visite diverse dell'albero producono le varie notazioni lineari:

– Simmetrica -> infissa

$\xrightarrow{\text{OUT}}$ (SOTTOALB. SX RAD SOTTOALB. DX)

– Anticipata -> prefissa

$\xrightarrow{\text{OUT}}$ (RAD SOTTOALB. SX SOTTOALB. DX)

– Differita -> postfissa

$\xrightarrow{\text{OUT}}$ (SOTTOALB. SX SOTTOALB. DX RAD)

Valutazione delle espressioni

- A partire dall'albero il compilatore produce il codice oggetto oppure l'interprete valuta l'espressione
 - ↳ compilazione o interprete
- In entrambi i casi l'ordine di valutazione delle sottoespressioni è importante per vari motivi:

PROBLEMI:

- 1)– Effetti collaterali
- 2)– Aritmetica finita
- 3)– Operandi non definiti
- 4)– Ottimizzazione

Problema 1)

$$\underbrace{(a + f(b))}_{1} * \underbrace{(c + f(b))}_{2}$$

La valutazione di questa espressione come cambia
 da $1 \rightarrow 2$ e da (valutaz. sx \rightarrow dx)
 $1 \leftarrow 2$ (valutaz. dx \rightarrow sx)

$\Rightarrow f()$ potrebbe produrre EFFETTI COLLATERALI:

$1 \rightarrow 2$	$a = 10$	$f(b) = 5$	$c = 2$	(risultato 105)
$2 \rightarrow 1$	<u>$a = 0$</u>	$f(b) = 5$	$c = 2$	(risultato 35)

in $f(b)$ potrebbe esserci una riga che dice $a=0$

Problema 2)

$\text{MAXINT} = \text{massimo valore INTERO rappresentabile}$
 in memoria

$\text{MAXINT} + 1$ OVERFLOW

$\text{MAXINT} + 1 - 2$ produce OVERFLOW?

→ DIPENDE dall' ORDINE di valutazione

Se fosse $\text{MAXINT} + (1 - 2) \neq (\text{MAXINT} + 1) - 2$
 Allora sapendo l'ordine posso stabilire
 (se c'è OVERFLOW o meno)

In arithmetica finita non è
 importante l'ordine di
 valutazione, in informatica sì!

Problema 3)

In linguaggio C:

`if a=0 then b else b/a`

OPERANDI:

`if [a=0] then [b] else [b/a]`

$$\begin{array}{l} a=0 \\ b=0 \end{array}$$

Se una valutazione è **EAGER** vengono valutati TUTTI gli OPERANDI
 → ERRORE! (in questo caso)

`if [a=0] then [b] else [b/a]` → valutati TUTTI → ERRORE!

Invece una valutazione **LAZY** (**conto-circuito**) che valuta solo gli operandi strettamente necessari

→ Non dà errore, non valuta b/a

`if [a=0] then [b] else [b/a]`

↑ non valutata
Seconda valutazione
LAZY

Ese. con LAZY

$$\frac{(A \vee \neg A)}{\text{VALUTATA}} \vee \frac{(\beta/0 > 0)}{\text{NON VALUTATA}}$$

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 ammesso 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 \rightarrow LAZY

- 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

→ costituito principale dei linguaggi imperativi

- 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)

Esempio COMANDI:

Ho un linguaggio iperattivo, C per esempio:

$$x = 15$$

Dal punto di vista formale (del significato del programma)
la valutazione di questa espressione modifica lo
STORE / STATO del programma → **EFFETTI COLLATERALI**

STATO: Insieme di tutti i nomi di variabili
di tutto il programma

→ di fatto lo STATO è una "FOTOGRAFIA
dello STATO del programma"

La modifica dello STATO del PROGRAMMA dà
effetti collaterali (side effects)

→ Lo STATO non c'è nei linguaggi funzionali

Variabili → scatola / contenitore di un valore

- Per modificare il contenuto, basta fare un ASSEGNAZIONE
- 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 e' un contenitore di valori che ha un nome
 - il valore nel contenitore può essere modificato mediante il comando di assegnamento.

x 2

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 Opass exp2

Esempio ASSEGNA MENTO

$x \leftarrow 10$ e sia l'espressione $x = x + 1$:

$x = x + 1$

↑
prendi il "contenitore" "x"
↓
valore
prendi il valore del contenitore

→ USO LA x in due modi completamente diversi

$$\begin{array}{l} L \text{ valore} = R \text{ valore} \\ (\text{a sx di } =) \quad (\text{a dx di } =) \end{array}$$

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
→ La costante ha un valore assegnato dal compilatore
- Linguaggi logici: modello analogo a quello dei funzionali, ma con la possibilità di modificare (entro certi limiti) il valore associato alla variabile → ho un meccanismo di istanziazione
- 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

meccanismo per cui una variabile si associa ad un valore
 x
 $x \rightarrow z$

variabile modificabile
variabile a riferimento

Modello a riferimento

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



- 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

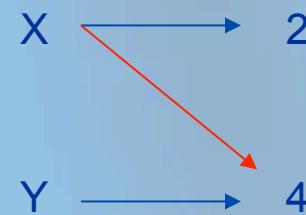
↳ utile per evitare di copiare grandi quantità di dati

- Var modificabile

X = 2; X 4

Y = 4; Y 4

Modello a riferimento



X = Y

↓
X diventa un riferimento al
valore di Y → si crea una
situazione di ALIASING

- 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 → gli *array* in Java sono oggetti con modello di riferimento

→ Si può realizzare questo modello a riferimento
tramite i puntatori (in alcuni linguaggi di programmazione)

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

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

No

($\text{index}(i)$ potrebbe causare un side effect)

\nearrow è per esempio
modifico $\text{index}(i)$

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

}

Meglio

→ ANCORA MEGLIO: avere un' operazione di INCREMENTO ESPUCITO ($A[i]++$)

- Per evitare questi problemi 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:
 - $++e$ ($--e$): 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)

$a,b,c := pippo (d,e,f)$

Espressioni e comandi (l. imperativi)

→ Ci sono dei linguaggi dove

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

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

espressioni sono netamente separate dai
comandi
alcune con i espressioni comandi coincidono

- 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

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

Ambiente e memoria

→ Se voglio dare un significato ad un programma devo daragli uno stato

- 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 l'ambiente

(Esecuzione di un
programma insieme
dei suoi stati)

Ogni programma può essere espresso da una funzione
variabile:

P L

$$f_p : D_1 \longrightarrow D_2$$

Nel linguaggi imperativi ne servono z , perché?
Però dall'aliasing



Usa una funzione che dà un nome
(di variabile) mi dice il valore di quel nome

$$f(x) = ?$$
$$f(y) = ?$$

deve essere possibile
l'ALIASING nei linguaggi
imperativi \rightarrow uso due funzioni

AMBIENTE e MEMORIA

funzione AMBIENTE: associa ad un nome un valore denotato

Es.

int $x = 2$

Qual' è l'oggetto denotato da x ?

\Rightarrow E' la locazione di memoria contenente 2

Se rispondi 2! ↗ Gorrieri ti boccia (ti picchia / ti schiaccia)
↘ Gabbielli ci ride su e ti dà 0,2 punti,
apprezzza che sei simpatico :D

funzione ambiente locaz. di memoria
 \uparrow \uparrow
AMBIENTE(x) = 1000 MEMORIA(1000) = 2

AMBIENTE(y) = 1000

N.B. Nei linguaggi funzionali basta la funzione ambiente perché una volta assegnato un valore ad una variabile non posso più modificarlo

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

Es. if $x > 0$ then
{
 $x = 1$
 $y = 1$
}
] comando composto
(cio le gaffe)

GOTO

→ comando che, dal posto di farsi eseguire l'istruzione successiva
me segue un'altra istruzione
Esiste perché c'è un comando ASSEMBLY della macchina → l'unico che altera il flusso

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

```
if a <b  goto 10
```

...

```
10: ...
```

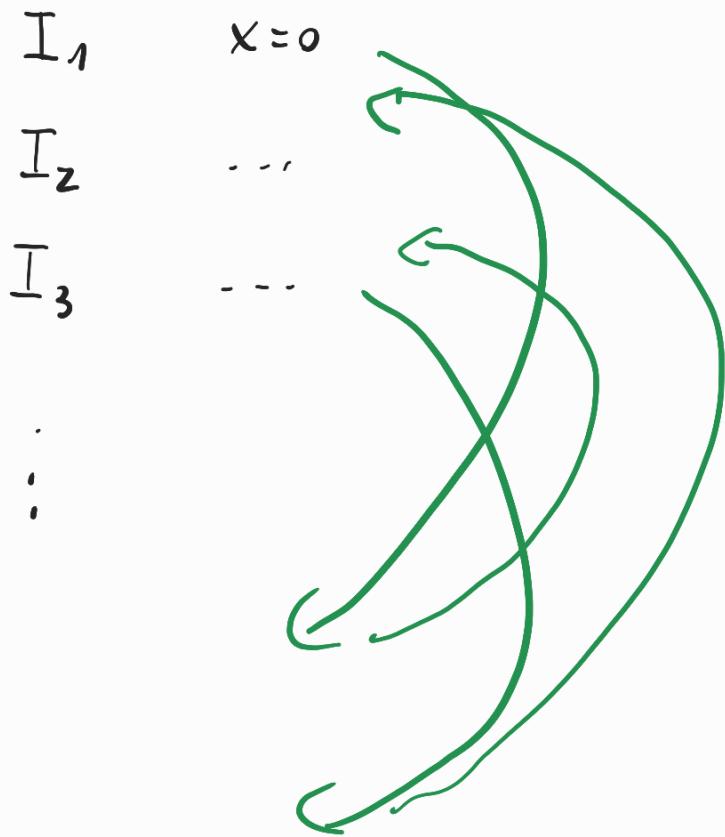
- Considerato utile essenzialmente per
 - uscita dal centro di un loop
 - ritorno da sottoprogramma
 - gestire eccezioni
- Alla fine considerato dannoso
- I moderni linguaggi
 - usano altri costrutti per gestire il controllo dei loop e dei sottoprogrammi (while, for, if then else, procedure ...vedi Algol 60)
 - usano un meccanismo strutturato di gestione eccezioni (Clu, Ada, C++, Lisp, Haskell, Java, Modula 3)
 - Goto non è presente in Java

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

Esempi GOTO:

Istruzioni in ASSEMBLY

GOTO:



Il flusso è descritto da una serie di salti
che lo rendono simile a "spaghetti"

→ Per la macchina è un problema? **No!**
ma lo è per noi, un codice con molti Goto
diventa un programma difficile da leggere
(praticamente illeggibile)

Programmazione strutturata

- Goto ``sconfitto'' perche' considerato contrario ai principi della programmazione strutturata
→ obiettivo: creare programmi meno verbosi, più facili da comprendere per noi umani
- 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

→ Il programma deve essere strutturato in blocchi:

blocchi
com

→ 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

Allora ho
PROGRAMMA
STRUUTURATO

- 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
```

in alcuni linguaggi esiste
il "terminatore dell'if" introdotto
per evitare ambiguità se ci sono if
annidati

- 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 → evoluzione dell'if con più possibilità

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

→ valuto \exp e cerco l'etichetta con valore corrispondente
Discendente del goto di Fortran e del switch di Algol 60

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

→ posso fare quello che faccio con il case con l'if? Sì, if annidati

- 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

→ le CASE si usa perché è più comodo e leggibile (mispetto a tanti if annidati)
e offre vantaggi di compilazione (mispetto a if annidati)

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 → **tabella dei salti**

↓
NON devo scomparire tutti i case

Compilazione del case

com if anni dati avrei bisogno di m accessi
in memoria

```
case exp of
```

```
| label_1 : C_1
```

```
| label_2 : C_2
```

```
....
```

```
| label_n : C_n
```

```
else C_n+1
```

ho solo bisogno
di 2 accessi:

Verde
se il
valore si
< o >
di una
certa label

L0

	Istruzioni precedenti al case
	Calcola il valore v di Exp
	Se $v < (\text{Label } 1)$, allora Jump L(n+1)
	Se $v > (\text{Label } n)$, allora Jump L(n+1)
	Jump L0 + v → offset per selezionare l'istruzione di salto corretta
L1	Jump L1
L2	Jump L2
	⋮
Ln	Jump Ln
L(n+1)	Comando C_1
	Jump FINE
	Comando C_2
	Jump FINE
	⋮
	Comando C_n
	Jump FINE
	Comando C_{n+1}
Fine	Istruzione successiva al case

Valutazione case

Controllo limiti

per selezionare l'istruzione
di salto corretta

Tabella di salto

Rami alternativi

Ramo else

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)

In genere NON posso creare programmi che fanno calcoli su dati parametrici (in input dall'utente)

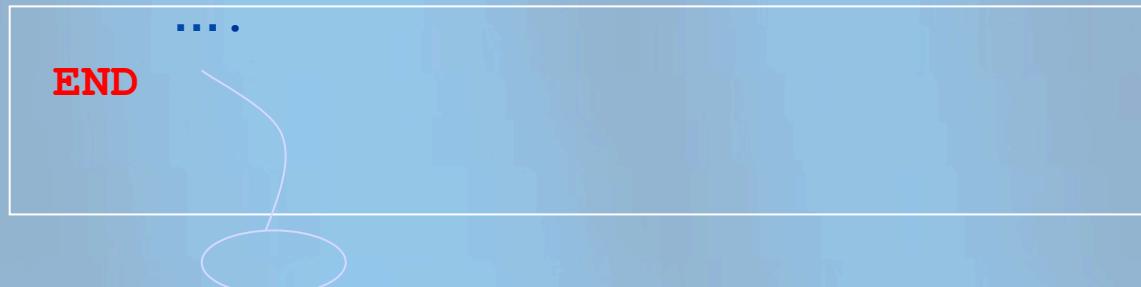
Iterazione

- 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

→ N.B. Il for non ha
GOTO

Iterazione determinata → è un ciclo for TRADIZIONALE
MA NON il FOR del C!

FOR tradizionale: **FOR** indice : = inizio TO fine BY passo DO



NON posso modificare i valori interni del for

- non si possono modificare indice, inizio, fine, passo all'interno del loop

→ cioè so che prima di eseguire il for so quante volte viene eseguito il for → PRIMA: indica al momento numero di ripetizioni del ciclo

K=100
for i=1 to K^{100 volte}
: esec. del for

quindi non è turing completo

- 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

LIMITE: I programmi scritti con un ciclo for TERMINANO!
NON POSSO SCRIVERE UN PROGRAMMA CHE NON TERMINA

→ con un for c'è la funzione di Ackermann non si può realizzare

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 > fine** termina l'esecuzione del **for**
altrimenti
 - si esegue corpo e si incrementa **I** del valore di **passo**;
 - si torna a (3).

In alcune implementazioni il for è implementato con il calcolo dell' **ITERATION COUNT**

Passo negativo

- Comando esplicito, come `downto` (Pascal) e `reverse` (Ada) → poiché così il controllo viene fatto con $I < \text{fine}$
– 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

→ Vantaggio dell' **ITERATION COUNT**: posso avere il passo con valore negativo e l'indice con valore positivo o negativo

→ è indeterminata perché immediatamente prima di eseguire il while non posso conoscere il numero di iterazioni

Iterazione indeterminata

condizione
↑ ↗ true → esecuzione while
 ↗ false → termina il while

while condizione do comando

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

repeat comando until condizione

equivalente a

comando;

while not condizione do comando;

→ Il while è un costrutto che rende il linguaggio Turing-completo

Se ho un ciclo for che ripete n volte il comando c

for i=1 to n

{

c

}

Se voglio fare lo stesso con un while

while B do

{ c // al comando c ho associato la funzione f_c

}

Quante volte viene chiamata f?

=> Per rispondere a questa domanda si usa il metodo del punto fisso (dal una certa iterazione il valore di f non cambia più)

Questo per evidenziare che la semantica del while è complicata e quella del for è più semplice

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 no

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 → ha il goto
(a differenza del for)

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++) il 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 *quindi se un linguaggio ha ricorsione è Turing-completo*
- 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)

Servono per spiegare la ricorsione

Definizioni induttive

*non finire mai → voglio una definizione
FINITA per qualcosa che è infinito*

- 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 $\underline{n + 1}$ è X ;
successore di n
anch'esso in

- 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 $P(n + 1)$.

→ $P(0)$ vale, $P(1)$ vale perché è $P(0+1)$ e so che $P(0)$ vale, ...

*Su questo principio
posso dare delle
definizioni*

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.

Esempio definizione induttiva: FATTORIALE

fatt() : funzione FATTORIALE

$$\text{fatt}(0) = 1$$

$$\text{fatt}(m+1) = (m+1) * \text{fatt}(m)$$

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

1) fie(1) = fie(1)

non definisco nulla

2) foo(0) = foo(0)
foo(n) = foo(n+1)

non definisco nulla

non definisco nulla

→ 1) e 2) Non sono
DEFINIZIONI INDUTTIVE
] non definiscono
nulla

1) e 2) → Dal punto di vista matematico non hanno senso (non sono funzioni)
– invece i seguenti programmi sono possibili

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

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

che non fanno
niente o non
terminano

→ N.B. Il fatto che la ricorsione assomigli ("analogo") alle definizioni

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 ...

Può essere implementata usando l'allocazione statica

Ricorsione in coda (tail recursion)

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

FUNZIONE CON RICORSIONE IN CODA:

- 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

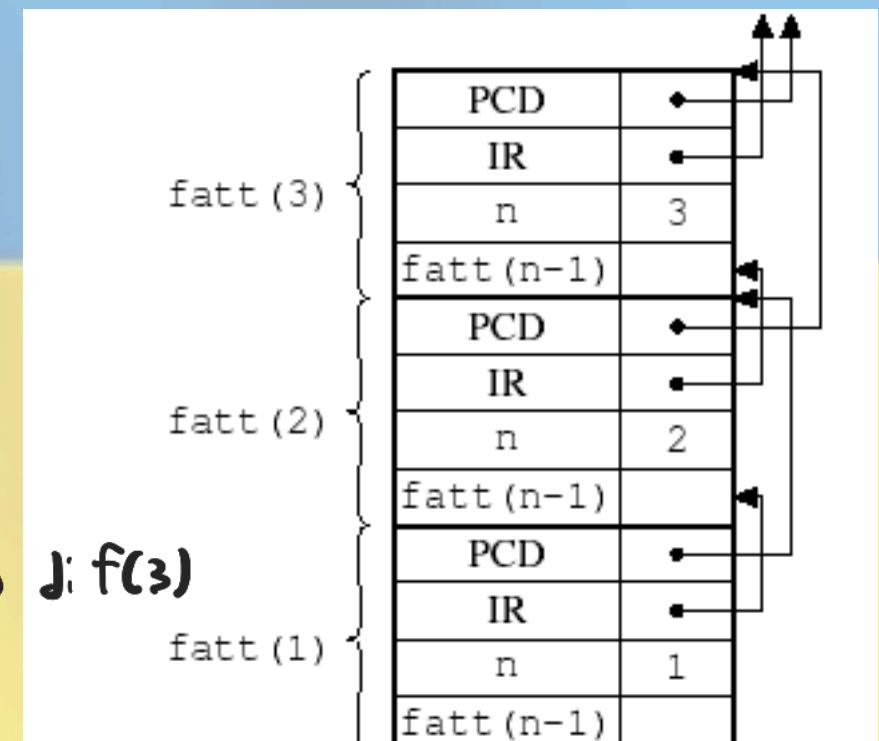
Esempio: il caso del fattoriale

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

Situazione dei RdA

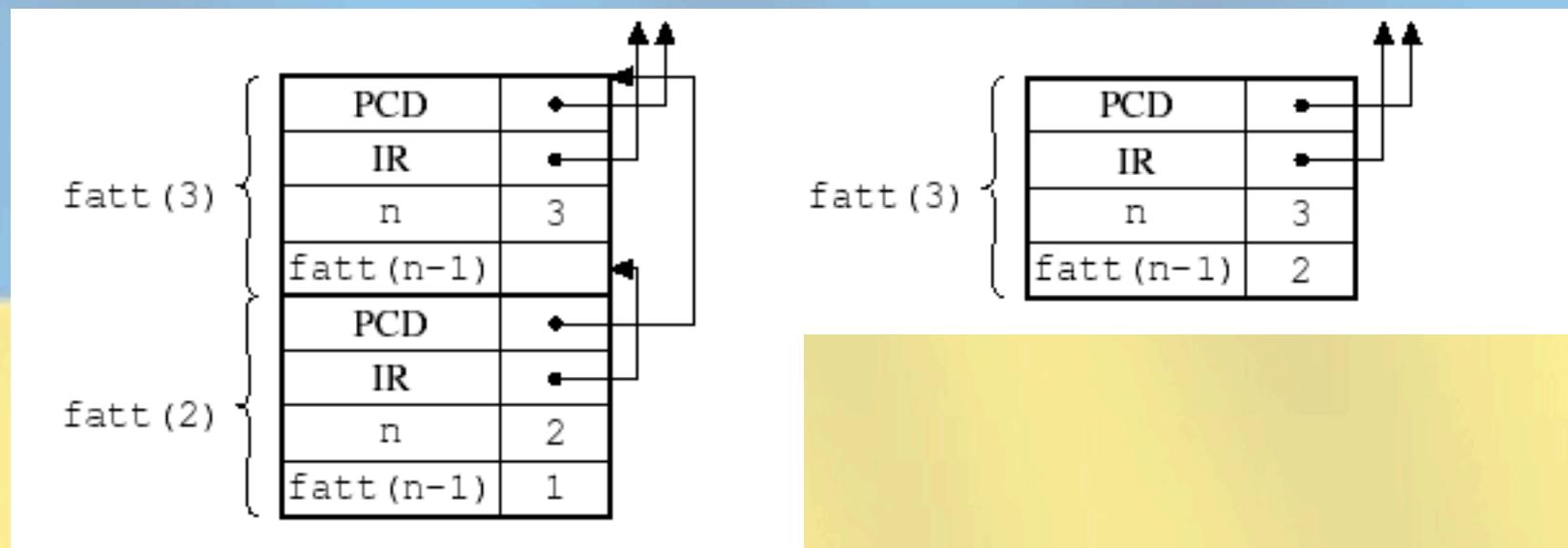
Dopo la chiamata di $f(3)$ e le successive chiamate ricorsive

Se dopo la chiamata ricorsiva non dovessi fare nulla,
basterebbe un solo RdA
perché il risultato non cambia più nelle chiamate ricorsive
successive.



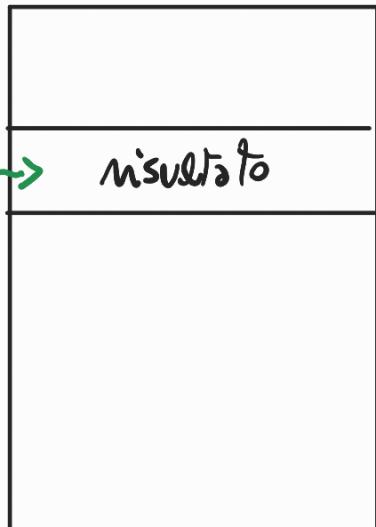
Esempio: il caso del fattoriale

- Evlouzione della computazione



$f(n)$ // prima chiamata ricorsiva \rightarrow RDA di $f(n)$

: $f(n - 1)$



Una versione tail-recursive del fattoriale

→ è efficiente
→ usa un solo RdA

- Cosa accade con la seguente funzione ?

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

risultato (res l'avevo calcolato in precedenza)

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

L'idea è che ciò che si fa durante la
"continuazione" viene fatto all'interno della funzione

Un altro esempio: numeri di Fibonacci

→ COMPLESSITÀ ESPOENZIALE → il numero di chiamate
cresce esponenzialmente

- Definizione.

Fib(1) = 0;

Fib(1) = 1;

Fib(n) = Fib(n-1)+Fib(n-2)

```
int fib (int n){  
    if (n == 0)  
        return 1;  
    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 → uso due parametri (res1 e res2) per memorizzare i risultati precedenti

```
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 (complessità lineare)
 - in spazio costante (un soloRdA)