

Strutture di controllo

Espressioni, assegnazione, costrutti per il controllo di flusso,
ricorsione

- Codice macchina: sequenza di istruzioni elementari, istruzioni di salto
- Linguaggi di programmazione: si vuole astrarre sul controllo
- definizioni più:
 - strutturate
 - compatte
 - leggibili

Strutture per il controllo del flusso

- Espressioni
 - Notazioni (sintassi)
 - Meccanismi di valutazione (semantica)
- Comandi
 - Assegnamento
- Sequenzializzazione di comandi
- Test, condizionali
- Comandi iterativi
- Ricorsione

che presenteremo ora

Altri meccanismi di controllo

- Chiamate di funzioni
- Gestione delle eccezioni
- Esecuzione concorrente
- Scelta non deterministica - probabilistica

presi in considerazione nel resto del corso

I paradigmi di programmazione (imperativo, dichiarativo) differiscono principalmente nei meccanismi di controllo adottati

- imperativo: assegnazione, sequenzializzazione, iterazione
- dichiarativo: valutazione di espressioni, ricorsione

Espressioni contenenti: identificatori, letterali, operatori (+, - ...),
funzioni

valutate dalla macchina producono:

- un valore
- un possibile effetto collaterale
- possono divergere, generare errori

Principali differenze:

- Posizione dell'operatore: infissa, prefissa, postfissa
- Uso delle parentesi

Diverse notazioni possibili	Esempi
infissa	$a + b * c$
funzione matematica	<code>add(a, mult(b, c))</code>
linguaggi funzionali (Cambridge polish)	<code>(+ a (* b c))</code>
omissione di alcune parentesi ML, Haskell	<code>add a (mult b c)</code>

Notazione

Parentesi:

- Scheme, Lisp: (Cambridge polish)
parentesi necessarie per forzare la valutazione
non possono essere aggiunte arbitrariamente
- ML, Haskell, C,
parentesi usate per definire un ordine di valutazione
possono essere aggiunte arbitrariamente

Zucchero sintattico:

scritture alternative di un'espressione (comando) per migliorare la leggibilità

- Haskell, Ada: si può usare notazione infissa in funzioni definite dal programmatore $a \text{ +- } b$ al posto di $'+-' \ a \ b$ $'+-' \ (a, \ b)$
- Ruby, C++: $a \ + \ b$ al posto di $a.operator+ \ (b)$

Notazione polacca

Esistono notazioni che non necessitano parentesi:

- prefissa (polacca diretta) $+ a * b c$
- postfissa (polacca inversa) $a b c * +$

Ottenute tramite una visita anticipata, o differita, dell'albero sintattico

Le parentesi possono essere omesse solo se l'arietà delle funzioni è fissa e nota a priori

Esempi di arità variabile:

- Scheme: $(+ 1 2 3)$
- Erlang: funzioni diverse, con stesso nome, distinte per l'arietà

Funzioni di arità arbitraria, parentesi indispensabili.

Poco leggibili e poco usate nei linguaggi di programmazione: Forth, calcolatrici tascabili

- polacca diretta, giustificazione: nella notazione a funzione argomenti $f(x, y)$ possiamo omettere $(,)$ se conosciamo l'arietà di ogni funzione
- polacca inversa: descrive la valutazione di un'espressione con lo stack degli operandi

Stack operandi

Processori basati su registri o basati su stack operandi__ processori virtuali di java bytecode, e in altri linguaggi intermedi: CIL

2 + 3 * 5 diventa:

2 3 5 * +

push 2;

push 3;

push 5;

mult;

add;

la compilazione Java riscrive un'espressione nella notazione polacca

Sintassi delle espressioni: notazione infissa

I linguaggi di programmazione tendono a usare le notazioni della scrittura matematica:

- notazione infissa
- regole di precedenza tra gli operatori per risparmiare parentesi ma non sempre ovvio il risultato della valutazione:

- $a + b * c ** d ** e / f$??

- $A < B \text{ and } C < D$??

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

Regole di precedenza

Ogni linguaggio di programmazione fissa le sue regole di precedenza tra operatori

- di solito operatori aritmetici hanno precedenza su quelli di confronto che hanno precedenza su quelli logici (non in Pascal)
 - Numerose regole e 15 livelli di precedenza in C e suoi derivati (C++, Java, C#)
 - 4 livelli di precedenza in Pascal
 - APL, Smalltalk: quasi tutti gli operatori hanno eguale precedenza: si devono usare le parentesi
 - Haskell (Prolog, Swift) permette di definire nuove funzioni con notazione infissa, e specificarne precedenza e associatività `infixr`
- 8 ^

Tabella delle precedenze

Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), &, * (address, contents of), !, ~ (logical, bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*, /, mod, rem
+, - (unary and binary)	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)	<, <=, >, >=, =, <>, IN	<, <=, >, >= (inequality tests)	=, /=, <, <=, >, >=
.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)	

Regole di associatività

Oltre al livello di precedenza, bisogna specificare in che ordine eseguire le operazioni di uno stesso livello

- Normalmente a sinistra $15 - 4 + 3$ $(15 - 4) + 3$
- In alcuni casi a destra: $5 ** 2 ** 3$ $5 ** (2 ** 3)$

Non sempre ovvie: in APL, tutto associa a destra, ad esempio,

$15 - 4 + 3$

è interpretato come

$15 - (4 + 3)$

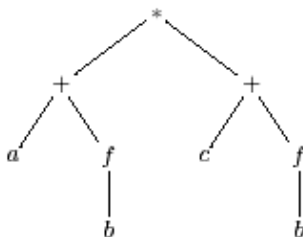
- regole di precedenza e associatività
- poco uniformi tra i vari linguaggi
- in alcuni casi piuttosto complesse

Nella pratica: se non si conoscono bene le regole, si è insicuri, si vuole esplicitare l'ordine, meglio inserire parentesi.

Rappresentazione ad albero

- La rappresentazione naturale di un'espressione è il suo albero sintattico
nell'albero sintattico l'ordine è evidente
- Le espressioni vengono linearizzate per necessità di scrittura
nell'espressione lineare bisogna definire le precedenze

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



- la rappresentazione ad albero generata dall'analizzatore sintattico, per poter poi lavorare sull'espressione

L'ordine di valutazione delle sottoespressioni

Le regole di precedenza, parentesi, o rappresentazione ad albero:

- definiscono precedenza e associatività,
- non definiscono un ordine temporale di valutazione delle sottoespressioni

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

Ordine importante per:

- effetti collaterali
- ottimizzazione

La valutazione di un'espressione restituisce un valore ma modifica lo **stato** del programma

Esempio tipico: la valutazione di un'espressione

- porta a chiamate di funzioni
- le funzioni modificano la memoria

Nell'esempio:

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

il risultato della valutazione da sinistra a destra può essere diverso da quello da destra a sinistra

Ordine di valutazione

- In Java è specificato chiaramente l'ordine (da sinistra a destra)
- C non specifica l'ordine di valutazione, compilatore sceglie ordine di valutazione
programmatore deve considerare tutti i possibili ordini
compilatori diversi si comportano in modo diverso.

```
int x=1;  
printf("%d \n", (x++) + (++x));  
x=1;  
printf("%d \n", (++x) + (x++));
```

- L'ordine di valutazione ha influenza sul tempo di esecuzione, specie nei processori attuali (computazione parallela, accesso lento alla memoria)
 - C preferisce l'efficienza alla chiarezza, affidabilità
 - Java il contrario.

Ottimizzazione e ordine di valutazione.

I compilatori possono modificare l'ordine di valutazione per ragioni di efficienza

$$a = b + c$$

$$d = c + e + b$$

può essere riarrangiato in

$$a = b + c$$

$$d = b + c + e$$

ed eseguito come

$$a = b + c$$

$$d = a + e$$

in alcuni casi, queste modifiche portano a modifiche nel risultato finale.

Evitare le ambiguità dovute all'ordine di valutazione

- In alcuni linguaggi non sono ammesse funzioni con effetti collaterali nelle espressioni (Haskell)
- altri linguaggi specificano l'ordine di valutazione (Java)
- in altri linguaggi, per evitare che il risultato dipenda da scelte del compilatore, forzando un ordine di valutazione, posso spezzare l'espressione

```
y = (a+f(b)) * (c+f(d))
```

riscritta come

```
x = a+f(b);
```

```
y = x * (c+f(d))
```

Svantaggi: senza effetti collaterali la valutazione delle espressioni diventa:

- indipendente dall'ordine di valutazione
- più vicina all'intuizione matematica
- più chiara, facile da capire
- più facile verificare, provare, correttezza
- più facile da ottimizzare per il compilatore (preservando il significato originale)

Stato (memoria) (effetti collaterali) utili per:

- gestire strutture dati di grandi dimensioni, funzioni che operano su matrici, modificandole in parte
- definire funzione che generano numeri casuali `rand()`

In linguaggi funzionali puri, la computazione si riduce a:

- la sola valutazione di espressioni
- senza quasi effetti collaterali

Aritmetica finita

- Numeri interi: **limitati** (aritmetica modulo 2^{32} , 2^{64})
- Numeri floating point: **valori limitati** e **precisione finita**

Conseguenze: errori di overflow, errori di arrotondamento ma anche le usuali identità matematiche non sempre valgono

$$a + (b + c) \neq (a + b) + c$$

- interi: la prima espressione genera errore di overflow la seconda no
con

```
a = -2; b = maxint; c = 2;
```

- floating point: l'errore nelle due valutazioni è differente con

```
a = 10**15; b = -10**15; c = 10**(-15);
```


Swap tra due variabili intere,

```
% x=a, y=b
x = x+y;    % x= a+b (mod 232)
y = x-y;    % y= a+b-b = a (mod 232)
x = x-y;    % x= a+b-a = b (mod 232)
```

programma corretto anche in caso di overflow

Valutazione eager - lazy. Operandi non definiti

Non sempre tutte le sottoespressioni sono valutate

esempio tipico, espressioni `if then else`

C, Java	<code>a == 0 ? b : b/a</code>
Scheme	<code>(if (= a 0) b (/ b a))</code>
Python	<code>b if a == 0 else b/a</code>

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

Valutazione corto circuito

Alcuni operatori booleani (and, or) usano una la valutazione lazy

- detta corto-circuito:
- se la valutazione del primo argomento è sufficiente a determinare il risultato, non valuto il secondo
- ordine di valutazione fondamentale per determinare il risultato, di solito da sinistra a destra

Esempio: con a uguale a 0:

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

- con valutazione corto circuito restituisce true
- valutazione eager genera errore
- anche la valutazione corto circuito da destra a sinistra genera errore

Valutazione corto-circuito

Restituisce immediatamente il risultato

- se il primo argomento di un `or (||)` è `true` restituisce `true`
- se il primo argomento di un `and (&&)` è `false` restituisce `false`

Alcuni linguaggi, Ada, Pascal, hanno due versioni degli operatori booleani

- short circuit: `and then` `or else`
- eager: `and` `or`

retrocompatibilità

utili se la valutazione delle espressioni ha un effetto collaterale necessario alla computazione

Valutazione corto-circuito

Stesso codice (ricerca valore 3 in una lista)

si comporta in maniera diversa a seconda del linguaggio

- C, valutazione corto circuito: corretto

```
p = lista
while (p && p->valore != 3)
    p = p->next
```

- Pascal (implementazione standard) valutazione eager: genera errore

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

`p->next` abbreviazione per `*p.next`

Parti del codice la cui esecuzione **tipicamente**:

- non restituisce un valore
- ha un effetto collaterale (modifica dello stato)

I comandi

- sono tipici del paradigma imperativo
- non sono presenti (o quasi mai usati) nei linguaggi funzionali e logici
- in alcuni casi restituiscono un valore (es. = in C)

Assegnamento: l-value, r-value

Comando base dei linguaggi imperativi

Inserisce in una locazione, cella, un valore ottenuto valutando un'espressione

```
x = 2  
y = x+1
```

Notare il diverso ruolo svolto da x nei due assegnamenti:

- nel primo, x denota una locazione, è un **l-value**
- nel secondo, x denota il contenuto della locazione precedente, è un **r-value**

Assegnamento

In generale

$\text{exp1} = \text{exp2}$

- valuto exp1 per ottenere un l-value (locazione)
- valuto exp2 per ottenere un r-value, valore memorizzabile
- inserisco il valore nella locazione

l-value può essere definito da un'espressione complessa

- esempio (in C)
 $(f(a)+3) \rightarrow b[c] = 2$
 - $f(a)$ puntatore ad un elemento in un array di puntatori a strutture A
 - la struttura A ha un campo b che è un array
 - inserisco 2 nel campo c -esimo dell'array

Diversi significati del termine **variabile**

La parte sinistra di un assegnazione è tipicamente una variabile

In contesti diversi il termine “variabile” ha significati differenti:

- linguaggi imperativi: identificatore a cui è associata una locazione, dove troviamo il valore modificabile.
- linguaggi funzionali (Lisp, ML, Haskell, Smalltalk): un identificatore a cui è associato un valore, non modificabile
coincidente con la nozione di costante per linguaggi imperativi
- Linguaggi logici: una variabile rappresenta un valore indefinito, la computazione cerca le istanziazioni delle variabili che rendano vero un certo predicato

Due diversi modi per:

- implementare le variabili
- implementare l'assegnamento
- definire cosa denotano le variabili

Quello discusso sinora,

- alle variabili l'ambiente (il compilatore) associa una locazione di memoria
 - il valore contenuto nella locazione è il valore associato alla variabile

L'assegnazione modifica il valore associato, dopo

$$y = z$$

y e z denotano due copie distinte dello stesso valore

Modello a riferimento:

- l'ambiente associa ad una variabile una locazione di memoria
- nella locazione troviamo un riferimento (una seconda locazione)
- contenente il valore,

Per accedere al valore:

- devo dereferenziare la variabile
- dereferenziazione implicita

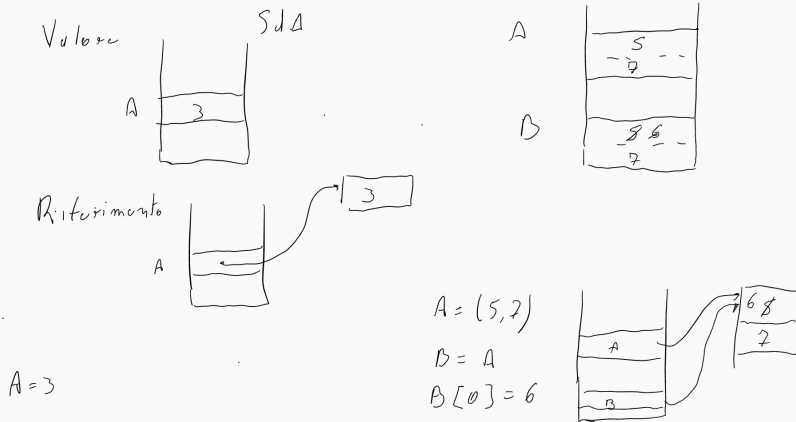
L'assegnazione modifica:

- il riferimento
- non il contenuto dello store

Dopo l'assegnamento $y = z$, y e z fanno riferimento alla stessa locazione di memoria, contenente un valore condiviso

- ogni variabile è, in certo senso, un puntatore
- usata con una sintassi diversa dai puntatori

Differenze dal punto di vista implementativo



In alcuni linguaggi i due modelli si miscolano

A seconda del tipo della variabile

Java:

- **tipi primitivi** (interi, booleani ecc.) :
 - modello a valore,
 - assegnamento copia un valore nella memoria
- **tipi riferimento** (tipi classe, array):
 - modello a riferimento
 - assegnamento crea una condivisione dell'oggetto.

Python:

Due categorie di tipi:

immutabili tipi semplici: interi, booleani, tuple

mutabili tipi complessi: vettori, liste, insiemi

Assegnamento:

- **immutabili**: viene creata una nuova istanza dell'oggetto, non si modifica la memoria, ma analogo effetto.
- **mutabili**: viene condivisa, eventualmente modificata, l'istanza esistente

Esempio in Python

```
tuple1 = (1,2,3) # tuples are immutable
list1 = [1,2,3] # lists are mutable

tuple2 = tuple1
list2 = list1
list3 = list1[.]

tuple2 += (9,9,9)
list2 += [9,9,9]

print 'tuple1 = ', tuple1 # outputs (1, 2, 3)
print 'tuple2 = ', tuple2 # outputs (1, 2, 3, 9, 9, 9)
print 'list1 = ', list1 # outputs [1, 2, 3, 9, 9, 9]
print 'list2 = ', list2 # outputs [1, 2, 3, 9, 9, 9]
print 'list3 = ', list3 # outputs [1, 2, 3]
```


Vantaggi - svantaggi modello a riferimento

Vantaggi:

- non duplico strutture dati complesse
- tutte le variabili sono puntatori
 - utile nelle funzioni polimorfe
stessa funzione agisce su una varietà di tipi di dato
esempio: funzione che ordina un vettore

Svantaggi:

- si creano aliasing che oscurano il comportamento del programma
- accesso ai dati indiretto

Assegnamento in linguaggi funzionali non puri (ML)

- posso dichiarare una variabile come locazione di memoria
- accedo al contenuto esplicitamente
- distinguo chiaramente locazione e contenuto
- l'assegnamento visto come una funzione con effetto collaterale

```
val y = 2
val x = ref 2      (* x denota una locazione contenente 2 *)
val x2 = x         (* x e x2 denotano la stessa locazione *)
val y2 = !x        (* y2 denota 2 *)
val _ = x := (!x) + 7  (* il contenuto di x, x2 è ora 9, *)
                      (* y2 denota sempre 2 *)
```

Assegnamento in Scheme

```
(define multiplier 2) ; Definiamo una variabile globale
```

```
(define (multiply x) ; Definiamo una funzione che usa multiplier  
  (* x multiplier))
```

```
(display (multiply 5)) ; Output: 10
```

```
(set! multiplier 3) ; Modifichiamo multiplier
```

```
(display (multiply 5)) ; Output: 15
```

Nei linguaggi imperativi distinguiamo tra:

- **Ambiente**: Nomi \rightarrow Valori Denotabili
 - definito, modificato dalle dichiarazioni
- **Memoria**: Locazioni \rightarrow Valori Memorizzabili
 - modificato dalle istruzioni di assegnamento

Distinguiamo tra tre classi di valori:

- Valori **Denotabili** (quelli a cui si può associare un nome)
- Valori **Memorizzabili** (si possono inserire nello store esplicitamente con assegnamento)
- Valori **Esprimibili** (risultato della valutazione di una espressione)

Valori denotabili, memorizzabili, esprimibili

Le tre classi si sovrappongono ma non coincidono

- procedure: denotabili, a volte esprimibili, quasi mai memorizzabili,
- locazioni: denotabili, esprimibili, memorizzabili con l'uso esplicito dei puntatori

Linguaggi imperativi, i valori denotabili includono le locazioni:

- variabili nomi che denotano locazioni.

Linguaggi funzionali puri:

- non esistono valori memorizzabili
- le locazioni non sono denotabili o esprimibili

Linguaggi funzionali:

- le funzioni sono valori esprimibili (funzioni anonime)
- Java, C#, Python, Ruby permettono la programmazione funzionale

Operazioni di assegnamento

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

Realizzate in maniera standard pongono i seguenti problemi:

- scarsa leggibilità
- efficienza: doppia computazione dell'indice $\text{index}(i)$, doppio accesso alla locazione
- side-effect: se $\text{index}(i)$ causa un effetto collaterale, questo viene ripetuto

Si definiscono degli operatori di assegnamento, più sintetici

$X = X + 1$	diventa	$X += 1$	(C, Java, ...)
$X := X + 1$	diventa	$X += 1$	(Algol, Pascal ...)

Operazioni di assegnamento

In C, Java . . . una pletora di operatori di assegnamento, incremento/decremento

`+=` `-=` `*=` `/=` `%=` `&=` `|=`

- somma, sottrazione, moltiplicazione, divisione, resto, bit-wise and, bit-wise or

Incremento, decremento di una unità

`x++` `x--`

nel caso la variabile incrementata sia un puntatore o un array C
l'incremento viene moltiplicato per la dimensione degli oggetti puntati

- `int *a`
- `a++` incrementa il valore `a` di 4 (la dimensione di un `int`)

Sintatticamente si distingue tra comandi e espressioni

- Comandi: è importante l'effetto collaterale
- Espressioni: è importante il valore restituito.

In alcuni linguaggi la distinzione tra comando ed espressione risulta sfumata:

- i due aspetti, effetto collaterale e risultato coesistono,
- dove è previsto un comando posso inserire un'espressione e viceversa.

C, Java, C#:

Comandi separati da espressioni:

```
if (a==b) {x=1} else {x=0};  
x = (a==b) ? 1 : 0 ;
```

ma

- espressioni possono comparire dove ci si aspetta un comando
- assegnamento (=) permesso nelle espressioni
 - l'assegnamento restituisce il valore assegnato, posso scrivere:
 - `a = b = 5` interpretato come `a = (b = 5)`
 - `if (a == b) { ... }` naturalmente, ma anche
 - `if (a = b) { ... }` che può generare errore di tipo

<code>{ (a==b) ? (x=1) : (x=0); y=0 }</code>	<code>\\ lecito</code>
<code>(a==b) ? {x=1} : {x=0};</code>	<code>\\ genera errore</code>

un singolo comando può essere visto come un espressione, un blocco
no

pre e post incremento

Il comando di incremento, `++`, può essere visto come un'espressione, con effetti collaterali,

- distinguo tra due versioni dell'espressione di incremento, pre e post incremento
 - `++x`: pre-incremento, esegue `x = x+1` restituisce il valore incrementato
 - `x++`: post-incremento, esegue `x = x+1` restituisce il valore originario

eseguono la stessa assegnazione ma restituiscono un diverso valore

- uguali come comandi
 - diversi come espressioni
- similmente esistono
 - `(x--)` `(--x)`

Algol68: expression oriented

- in Algol68 tutto è un'espressione:
 - non c'è nozione separata di comando
 - ogni procedura restituisce un valore

begin

a := begin f(b); g(c) end;

g(d);

2 + 3

end

Pascal:

- comandi separati da espressioni
- un comando non può comparire dove è richiesta un'espressione e viceversa

Python:

- maggiore separazione rispetto a C
operatore di assegnazione distinto

```
if (n := 4) > 3:      # `:=` assegna e restituisce un valore
    print("n:", n)    # Output: n: 4
```

Esercizi

Mostrare l'evoluzione dello store nei seguenti comandi:

- valutazione delle espressioni da sinistra a destra
- nell'assegnazione, si valuta prima r-value, poi l-value
- indice base del vettore: 0

```
int V[5] = { 1, 2, 3, 4, 5};
```

```
int i = 3;
```

```
V[--i] += i;
```

```
V[i--] = i + i++;
```

```
(V[i++])++;
```

```
i = 0;
```

```
i = V[i++] = V[i++] = (V[i++])++;
```

Comandi per il controllo sequenza

Comandi per il controllo sequenza esplicito

- ;
- blocchi
- goto

Comandi condizionali

- if
- case

Comandi iterativi

- iterazione indeterminata (while)
- iterazione determinata (for, foreach)

Comando sequenziale

Composizione sequenziale

C1 ; C2 ;

- è il costrutto di base dei linguaggi imperativi
- sintassi possibili scelte:
 - ; **terminatore** di comandi, devo inserirlo anche dopo l'ultimo comando
C, C++, Java, C#, JavaScript, Kotlin, PHP
 - ; **separatore** di comandi, non serve inserirlo dopo l'ultimo comando
Perl, Ruby, Bash, OCaml, Ada
 - ; può essere sostituito dall'indentazione
Python, Haskell

Algol 68: quando un comando composto è visto come espressione
il valore restituito è quello dell'ultimo comando

Sintassi

{	begin	for x in ..
...	...	y=y+x
}	end	z=z-x

Trasformo una sequenza di comandi in un singolo comando,
raggruppo una sequenza di comandi

Posso usarlo per introdurre variabili locali

GOTO - istruzioni di salto

```
if a < b goto 10
...
10: ...
```

- costruito base in assembly
- permette una notevole flessibilità
- ma rende programmi poco leggibili, e nasconde gli errori
- interazione complessa con chiamate di funzioni e stack di attivazione

Accesso dibattito negli anni 60/70 sulla utilità del goto

Alla fine considerato dannoso, contrario ai principi della
programmazione strutturata

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

sostituibilità del GOTO

Teoria

- teorema di Boehm-Jacopini (1966)
 - GOTO sostituibile da costrutti cicli while - test,

Pratica:

- la rimozione del GOTO non porta a una grossa perdita di flessibilità - espressività
- istruzioni di salto giustificabili e utili solo in particolari contesti, con costrutti appositi:
 - uscita alternativa da un loop: `break`
 - uscita da una procedura: `return`
 - gestione eccezioni: `raise exception`

Nei linguaggi moderni, per sicurezza, chiarezza, il Goto non è presente:
Python, Java, Rust, Swift

Presente, ma scoraggiato in: C, C++, Fortran, Ada, Go

Programmazione strutturata

Metodologia introdotta negli anni 70, per gestire la complessità del software

- Progettazione gerarchica, top-down
- Modularizzare il codice
- Uso di nomi significativi
- Uso estensivo dei commenti
- Tipi di dati strutturati
- Uso dei costrutti strutturati per il controllo
 - ogni costrutto, pezzo di codice, un unico punto di ingresso e di uscita
 - le singole parti della procedura modularizzate
 - diagrammi di flusso non necessari

Comando condizionale

```
if (B) {C_1} ;  
if (B) {C_1} else {C_2} ;
```

- Introdotto in Algol 60
- possibili ambiguità in presenza di if annidati:

```
if b1 then if b2 then c1 else c2  
  
if (i != 2) if (i == 1) printf("%d, 3);  
                                else printf("%d, i);
```

- varie opzioni per risolvere l'ambiguità
 - Pascal, C: else associa con il then non chiuso più vicino
 - Algol 68, Fortran 77: parole chiave `endif` o `fi` marcano la fine del comando

Scelte multiple

gestite, in maniera canonica, con sequenze di comandi `if else`

```
if (Bexp1) {C1}  
  else if (Bexp2) {C2}  
  ...  
  else if (Bexpn) {Cn}  
  else {Cn+1}
```

Espressione condizionale in Scheme.

```
(if test-expr then-expr else-expr)
```

- Valuta test-expr.
- Se il risultato un valore diverso da #f,
 - allora viene valutata then-expr
 - altrimenti si valuta else-expr

Esempi:

```
> (if (positive? -5) (error "doesn't get here") 2)
2
> (if (positive? 5) 1 (error "doesn't get here"))
1
> (if 'we-have-no-bananas "yes" "no")
"yes"
```

Comando condizionale in Scheme.

In alternativa:

```
(cond
  [(positive? -5) (error "doesn't get here")]
  [(zero? -5) (error "doesn't get here, either")]
  [(positive? 5) 'here])
```

Argomenti (in numero arbitrario

- coppie [guardia_booleana valore_restituito]

Case

Estensione del comando di test a tipi non booleani

```
case exp of                                {* exp: espressione a valori discreti *
|   const_1 : C_1
|   const_2 : C_2
...                                         {* const valori costanti, disgiunti *
|   const_n : C_n                         {* di tipo compatibile con exp *
else      C_{n+1}
```

- Molte versioni nei vari linguaggi
- Possibilità di definire **range** case 0 ... 9: C2
 - non presente in: Pascal, C (vecchie versioni)
 - presente C, Visual Basic ammettono range:
- **ramo di default** : C, Modula, Ada, Fortran,
 - senza default, e con nessuna opzione valida: non si esegue nulla.

```
int i    ...
switch (i)
{
    case 3:
        printf("Case3 ");
        break;
    case 5:
        printf("Case5 ");
        break;
    default:
        printf("Default ");
}
```

Ogni sotto-comando termina break,

- devo uscire esplicitamente dal caso con break
 - altrimenti si continua col comando successivo, posso evitare di scrivere un comando due volte
 - facile causa di errori
- in C# devo esplicitare l'azione finale: break, continue

```
switch (i)
{
    case 1:
        printf("Case1 ");
        break;
    case 2:
    case 7:
        printf("Case2 or Case 7");
        break;
    default:
        printf("Default ");
}
```

Estensioni con range

```
switch (arr[i])
{
    case 1 ... 6:
        printf("%d in range 1 to 6\n", arr[i]);
        break;
    case 19 ... 20:
        printf("%d in range 19 to 20\n", arr[i]);
        break;
    default:
        printf("%d not in range\n", arr[i]);
        break;
```

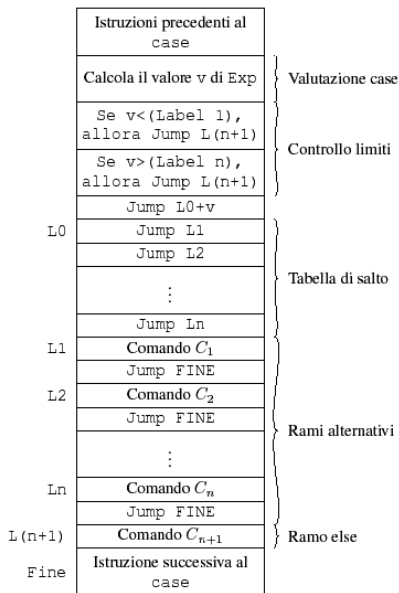
Compilazione del case

Più efficiente di if multiplo se compilato in modo astuto ...

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

- con il valore di `exp` accedo a
- una tabella di istruzioni di salto
- che porta al codice macchina del ramo corrispondente al valore

struttura del codice generato:



Versione per range ampi

- Lo schema precedente funziona bene,
 - tempo di esecuzione costante e non lineare sul numero di possibilità
 - occupazione di memoria limitata se i range di valori sono limitati
- Con range ampi, troppa occupazione di memoria devo ripetere la stessa istruzione di salto per ogni valore nel range
- È possibile ridurre l'occupazione di memoria con
 - ricerca binaria
 - tabella hash
- codice assembly più complesso, tempo di esecuzione: logaritmico o costante

Pattern matching

- ML, Haskell, Rust: possibilità di usare pattern complessi all'interno di case

```
case (m, xs) of
  (0, _) => []
| (_, []) => []
| (n, (y:ys)) => y : take (n-1) ys
```

- case su più espressioni
- istanziamento di variabili, meccanismo piuttosto sofisticato
 - casi non mutuamente esclusivi, si sceglie il primo,
 - casi non esaustivi, si genera errore

Pattern matching - Rust

```
enum Message {  
    Quit,  
    Move { x: i32, y: i32 },  
    Write(String),  
}  
  
fn process_message(message: Message) {  
    match message {  
        Message::Quit => {  
            println!("Quit variant");  
        }  
        Message::Move { x, y } => {  
            println!("Move to x: {}, y: {}", x, y);  
        }  
        Message::Write(text) => {  
            println!("Text message: {}", text);  
        }  
    }  
}
```

- Iterazione e ricorsione sono i due meccanismi che permettono di ottenere tutte le funzioni computabili
 - formalismi di calcolo Turing completi

Senza iterazione: nessuna istruzione ripetuta, tutto termina in un numero limitato di passi

- Iterazione
 - indeterminata: cicli controllati da un'espressione booleana
`while`, `repeat`, ...
 - determinata: cicli su un range di valori
`do`, `for`, `foreach` ... con numero di ripetizioni del ciclo determinate al momento dell'inizio del ciclo

Iterazione indeterminata

`while` condizione `do` comando

Sintassi più usata Java, ...

```
while (counter > 1)
{ factorial *= counter--;
};
```

in altri linguaggi: Pascal,

```
while Counter > 0 do
begin
    Factorial := Factorial * Counter;
    Counter := Counter - 1
end
```

- Introdotta in Algol-W,

Versione post-test, ripetuto almeno una volta

- tipicamente C, C++, Java;

```
do {  
    factorial *= counter--; // Multiply, then decrement.  
} while (counter > 0);
```

- Ruby

```
begin  
    factorial *= counter  
    counter -= 1  
end while counter > 1
```

- Pascal

```
repeat  
    Factorial := Factorial * Counter;  
    Counter -= 1  
until Counter = 0
```

Equivalenza tra i due tipi di ciclo

Facile sostituire un tipo di ciclo con l'altro

```
do {  
    do_work();  
} while (condition);
```

equivalente a:

```
do_work();  
while (condition) {  
    do_work();  
}
```

A seconda dei casi, una versione risulta più sintetica dell'altra

Iterazione indeterminata

- Indeterminata perché il numero di iterazioni non è noto a priori
- l'iterazione indeterminata permette di ottenere tutte le funzioni calcolabili,
- facile da tradurre in codice assembly

Iterazione determinata

```
for indice := inizio to fine step passo do  
    comandi  
end
```

- al momento dell'inizio dell'esecuzione del ciclo, è determinato il numero di ripetizioni del ciclo
 - all'interno del loop, non si possono modificare: `indice`,
 - `fine`, `passo` valutati e salvati a inizio esecuzione
- il potere espressivo è minore rispetto all'iterazione indeterminata: non si possono esprimere computazioni che non terminano
- da preferire perché:
 - garantisce la terminazione,
 - ha una scrittura più leggibile e compatta
- in C, e suoi derivati, il `for` non è un costrutto di iterazione determinata, posso modificare l'indice

Semantica del For

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

nell'ipotesi di passo positivo:

- ➊ valuta le espressioni inizio e fine e salva i valori ottenuti
- ➋ inizializza indice con il valore di inizio;
- ➌ se $\text{indice} > \text{fine}$ termina l'esecuzione del for altrimenti:
 - si esegue corpo
 - si incrementa indice del valore di passo;
 - si torna a (3).

Diverse realizzazioni

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

I vari linguaggi differiscono nei seguenti aspetti:

- possibilità di modificare indice, valore finale, passo nel loop (se sì, non si tratta di vera iterazione determinata)
- possibilità incremento negativo
- valore `indice` al termine del ciclo: indeterminato, `fine`, `fine + 1`.

Iterazione determinata in C, C++, Java

```
for (initialization; condition; increment/decrement)  
    statement
```

Dove statement è spesso un blocco

```
int sum = 0;  
for (int i = 1; i < 6; ++i) {  
    sum += i;  
}
```

Altri linguaggi

Python:

```
for counter in range(1, 6):  # range(1, 6) gives values from 1 to 5
    # statements
```

Ruby:

```
for counter in 1..5
    # statements
end
```

Foreach

Ripeto il ciclo su tutti gli elementi di un oggetto enumerabile: array, lista

- Presente in vari linguaggi sotto diverse forme
- Limitato potere espressivo, ma utile per
 - chiarezza
 - compattezza
 - prevedibilità

Java dalla versione 5

```
int [] numbers = {10, 20, 30, 40, 50};  
  
for(int x : numbers ) {  
    System.out.print( x + "," );  
}
```

Foreach

Vengono separati,

- algoritmo di scansione della struttura:
 - implicito nel `foreach`,
 - generato automaticamente nel compilatore
- operazioni da svolgere sul singolo elemento
 - definito esplicitamente nel codice

Può essere svolto su un tipo di dato strutturato che metta a disposizione funzioni implicite per determinare

- primo elemento
- passo ad elemento successivo
- test di terminazioni

Esempi

- liste, array, insiemi
- alberi

Foreach altri esempi

Python:

```
pets = ['cat', 'dog', 'fish']  
for f in pets:  
    print f
```

- Ciclo **for** per Python: un caso particolare di questo meccanismo.

Ruby

```
pets = ['cat', 'dog', 'fish']  
pets.each do |p|  
    p.print  
end  
  
for p in pets  
    p.print  
end
```

Foreach altri esempi

JavaScript

```
var numbers = [4, 9, 16, 25];  
function myFunction(item, index) {  
    ....; }  
numbers.forEach(myFunction)
```

.forEach metodo che riceve una funzione e la applica a tutti gli elementi di una struttura

Ricorsione

- Modo alternativo all'iterazione per ottenere la Turing completezza
- scelta obbligata nei linguaggi puramente funzionali
- Intuizione: una funzione (procedura) è ricorsiva se definita in termini di se stessa.
- Riflette la natura induttiva di alcune funzioni.

fattoriale (0) = 1.

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

diventa

```
int fatt (int n){  
    if (n <= 0)  
        return 1;  
    else  
        return n * fatt ( n-1 );  
}
```


Ricorsione e induzione, considerazioni generali

Numeri naturali $0, 1, 2, 3, \dots$. Minimo insieme X che soddisfa i due assiomi seguenti (Peano):

- 0 è in X ;
- 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

- $P(0)$ è vera;
- Per ogni n , se $P(n)$ è vera allora è vera anche $P(n + 1)$.

Definizioni induttive (primitive ricorsive). Se $g: (\text{Nat} \times A) \rightarrow A$ totale allora esiste una unica funzione totale $f: \text{Nat} \rightarrow A$ tale che

- $f(0) = a$;
- $f(n + 1) = g(n, f(n))$.

Fattoriale segue lo schema di sopra.

Definizioni primitive ricorsive

- Garanzia di buona definizione (definisco univocamente una funzione totale)
- Schema piuttosto rigido:
 - la definizione primitiva ricorsiva della divisione $\text{div}(n, m)$ è non ovvia
 - si può generalizzare: well founded induction, per avere
 - schema più flessibile
 - garanzia di buona definizione

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”:
- la seguente scrittura non definisce alcuna funzione

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

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

- invece i seguenti programmi sono possibili

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

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 può essere tradotto in uno equivalente iterativo

- e viceversa,

Confronto:

- ricorsione più naturale:
 - su strutture dati ricorsive (alberi),
 - quando la natura del problema è ricorsiva
- iterazione più efficiente su matrici e vettori
- ricorsione scelta obbligata nei linguaggi funzionali
iterazione scelta più usata nei linguaggi imperativi

In caso di implementazioni naif ricorsione molto meno efficiente di iterazione tuttavia

- un compilatore ottimizzato può produrre codice efficiente
- tail-recursion

Ricorsione in coda (tail recursion)

Una chiamata di g in f di si dice **in coda** (o **tail call**) se f restituisce il valore restituito da g senza nessuna ulteriore computazione

f è **tail recursive** se contiene solo chiamate in coda a se stessa

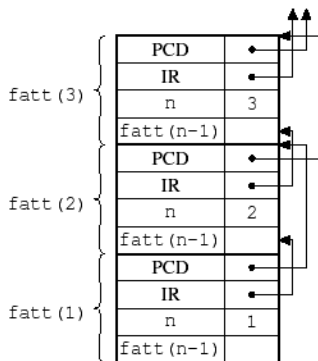
```
function tail_rec (n: integer, m): integer {  
    ... ; return tail_rec(n-1, m1)}
```

```
function non_tail_rec (n: integer): integer {  
    ... ; return (non_tail_rec(n-1) + 1)}
```

- Non serve allocazione dinamica della memoria con pila: basta un unico RdA,
 - dopo la chiamata ricorsiva, il chiamante non deve fare nulla, attende il risultato, e lo passa al suo rispettivo chiamante
 - record di attivazione del chiamante inutile, spazio riutilizzato dal chiamato

Più efficiente, esempio: il caso del fattoriale

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



Versione tail-recursive di fattoriale

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

```
int fatt (int n){  
    fattTR (n, 1);} 
```

- viene aggiunto un parametro `res` che rappresenta il valore da passare al **resto della computazione**
- `fattTC (n, res)` restituisce $n! * res$, uguale a $(n-1)! * (n * res)$

Basta un unico RdA

- dopo ogni chiamata il RdA della funzione chiamante può essere riutilizzato
- come RdA della funzione chiamata

Altro esempio: numeri di Fibonacci

Definizione:

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

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

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

in Scheme diventa

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Complessità in tempo e spazio esponenziale in n

- ad ogni chiamata due nuove chiamate
- più precisamente il numero della chiamate ha una crescita alla Fibonacci.

Una versione efficiente per Fibonacci

Calcolo una coppia di valori nella sequenza,

```
(define (fib n)
  (if (= n 0)
      (cons 0 1)
      (let ((p (fib (- n 1))))
        (cons (cdr p) (+ (car p) (cdr p))))))
```

- basta una chiamata ricorsiva
- tempo lineare

Ricorsione di coda per Fibonacci

`fibHelper(n, a, b)` una funzione che nell'ipotesi:

- $a = \text{Fib}(i)$
- $b = \text{Fib}(i+1)$

`fibHelper(n, a, b) = Fib(n + i)`

in Scheme:

```
(define (fibHelper n a b)
  (if (= n 0)
      a
      (fibHelper (- n 1) b (+ a b))))
(define (fib n)
  (fibHelper n 0 1))
```

Invariante:

- se a e b sono l' $(m-1)$ -esimo e l' m -esimo elementi nella serie di Fibonacci,
- allora `(fibHelper n a b)` è $(m+n)$ -esimo elemento nella serie

Complessità:

- in tempo, lineare in n
- in spazio, costante (un solo RdA)

Schema per la ricorsione di coda

- Uso dei parametri aggiuntivi alla funzione per memorizzare risultati intermedi
- Simulo l'esecuzione di un ciclo (`while`) dentro un linguaggio funzionale:
 - per ogni funzione `f` che in un linguaggio imperativo avrei implementato tramite un ciclo
 - definisco una funzione `f-helper` avente parametri extra
 - questi parametri extra svolgono il ruolo delle variabili modificabili nel ciclo
 - `f-helper` chiama se stessa aggiornando i parametri extra, come avviene in un ciclo
 - `f` chiama `f-helper` inizializzando i parametri extra (come la funzione imperativa)
- Simulo uno stato in maniera locale e controllata, senza introdurre uno stato globale

Esempio: esponenziale (efficiente) su interi

```
exp(int a, n)
  if (n == 0) {return 1;}
  if (n % 2) {return (exp(a*a, n/2));}
  else      {return (exp(a*a, n/2) * a);}
```

Esempio: esponenziale (efficiente) su interi

```
exp(int a, n)
    if (n == 0) {return 1;}
    if (n % 2) {return (exp(a*a, n/2));}
    else      {return (exp(a*a, n/2) * a);}

exp = 1;
while(0 < n){
    if (n % 2 == 1){exp = exp * a;}
    a = a * a;
    n = n/2;}
return exp;
```

Esempio: esponenziale (efficiente) su interi

```
exp(int a, n)
    if (n == 0) {return 1;}
    if (n % 2) {return (exp(a*a, n/2));}
    else      {return (exp(a*a, n/2) * a);}

exp = 1;
while(0 < n){
    if (n % 2 == 1){exp = exp * a;}
    a = a * a;
    n = n/2;}
return exp;

expHelper(int a, n, exp)
    if (n == 0) {return exp;}
    if (n % 2) {return (expHelper(a*a, n/2, exp*a));}
    else      {return (expHelper(a*a, n/2, exp));}

expTR(int a,n) = {return (expHelper(a,n,1));}
```


- Ottimizzazione: solo le funzioni ricorsive più critiche (per la velocità d'esecuzione globale del programma) vengono riscritte usando la ricorsione di coda