Lezione 3

7–8 ottobre 1999

Argomenti trattati

- Programmazione strutturata.
- Strutture di controllo fondamentali: sequenza, selezione, iterazione.
- Variabili e istruzioni di assegnamento.

3.1 La Programmazione Strutturata

Abbiamo visto che un programma è costituito da istruzioni elementari. Per potere descrivere algoritmi significativi, deve essere possibile associare ad un programma piú sequenze d'esecuzione che dipendano da situazioni che si creano durante l'esecuzione. Ad esempio, un programma per il calcolo della divisione tra due numeri dovrà effettuare la divisione solo nel caso in cui il divisore sia diverso da zero, e dovrà stampare un messaggio d'errore in caso contrario. Nella *Programmazione Strutturata*, proposta come metodologia per la stesura di programmi agli inizi degli anni '70, l'esecutore viene guidato alla sequenza di esecuzione opportuna, tra tutte quelle possibili, mediante tre strutture di controllo fondamentali:

- la *sequenza*, che permette di eseguire le istruzioni secondo l'ordine in cui sono scritte;
- la *selezione*, che permette di scegliere, in base al valore di una condizione, l'esecuzione di un blocco di istruzioni tra due possibili;
- l'*iterazione*, che permette di ripetere, per un certo numero di volte, l'esecuzione di un blocco di istruzioni.

È stato dimostrato che i programmi esprimibili mediante l'uso di istruzioni di salto (goto) o diagrammi di flusso (flow–chart), possono essere riscritti utilizzando esclusivamente le tre strutture di controllo fondamentali. Inoltre, l'uso di queste strutture migliora la qualità dei programmi prodotti, di cui risulta piú facilmente comprensibile l'architettura, e allo stesso tempo riduce la possibilità di errori. Pertanto la programmazione strutturata si è imposta come metodologia nella stesura dei programmi. Il linguaggio Pascal è un linguaggio strutturato, contiene cioè i costrutti che permettono di programmare utilizzando direttamente le strutture di controllo fondamentali.

Sequenza

In assenza di indicazioni differenti, le istruzioni vengono eseguite nello stesso ordine in cui compaiono nel programma, cioè secondo la *sequenza* in cui sono scritte.

Ad esempio la somma di due numeri può essere calcolata, utilizzando esclusivamente la sequenza, mediante le seguenti operazioni:

leggi i numeri a, b
calcola $a + b$
scrivi il risultato

Selezione

La *selezione* permette di scegliere un'istruzione o un blocco di istruzioni da eseguire, tra due alternative. In ogni caso, terminata l'esecuzione del blocco selezionato, si prosegue dall'istruzione successiva al costrutto di selezione.

Possiamo schematizzare la selezione come segue:

```
SE condizione
      ALLORA
         blocco1
      ALTRIMENTI
         blocco2
FINESE
```

L'esecuzione avviene come segue. Prima di tutto viene valutata la `condizione`. Se questa risulta vera, vengono eseguite le istruzioni del `blocco1`, se risulta falsa, vengono eseguite quelle del `blocco2`. In entrambi i casi, l'esecuzione prosegue con l'istruzione che segue immediatamente la fine del costrutto di selezione, cioè con l'istruzione che segue la parola `FINESE`.

Nel caso in cui il `blocco2` non contenga alcuna istruzione, si può utilizzare il seguente schema semplificato:

```
SE condizione
      ALLORA
         blocco1
FINESE
```

Ad esempio, il calcolo della divisione tra due numeri può essere effettuato, controllando che il divisore sia diverso da zero, come segue:

```
leggi il dividendo e il divisore
SE il divisore è diverso da zero
    ALLORA
        calcola dividendo/divisore
        scrivi il risultato
    ALTRIMENTI
        scrivi "errore: divisione per zero"
FINESE
```

Notiamo che, in base al valore del divisore, è possibile selezionare una tra due possibili sequenze di esecuzione. Dopo l'esecuzione del ramo selezionato, l'elaborazione riprende da ciò che segue la parola `FINESE` (in questo caso, non essendoci nulla, l'esecuzione termina).

Esempio: calcolo delle soluzioni di un'equazione di secondo grado

Descriviamo, utilizzando le strutture fondamentali di sequenza e selezione, un algoritmo per il calcolo delle radici di un'equazione di secondo grado della forma $ax^2 + bx + c = 0$.

Ricordiamo che il numero di soluzioni dipende dal valore della quantità $b^2 - 4ac$ detta anche *discriminante*. In particolare, si possono avere le seguenti situazioni:

- il discriminante è negativo: l'equazione non ammette soluzioni reali;
- il discriminante è nullo: l'equazione ammette due soluzioni reali coincidenti, di valore $\frac{-b}{2a}$;
- il discriminante è positivo: l'equazione ammette due soluzioni reali distinte, date dalla formula $\frac{-b \pm \sqrt{b^2-4ac}}{2a}$.

Il procedimento risolutivo dell'equazione può essere dunque descritto dal seguente algoritmo:

```
leggi i valori dei parametri a, b, c
calcola il discriminante
SE il discriminante è minore di zero
    ALLORA
        scrivi "nessuna soluzione reale"
    ALTRIMENTI
        SE il discriminante è uguale a zero
            ALLORA
                calcola -b/2a
                scrivi "Due soluzioni coincidenti: ", il risultato
            ALTRIMENTI
                scrivi "Due soluzioni: ",
                calcola (-b-√(b²-4ac))/2a
                scrivi il risultato
                calcola (-b+√(b²-4ac))/2a
                scrivi il risultato
        FINESE
FINESE
```

Si osservi che, nell'algoritmo appena descritto, sono possibili tre differenti sequenze d'esecuzione. Nell'algorimo vengono utilizzate istruzioni di tre tipi:

- di *lettura*, per leggere dall'ambiente esterno i valori da elaborare;
- di *scrittura*, per comunicare all'ambiente esterno i risultati;
- di *calcolo*, per calcolare il valore di espressioni aritmetiche.

Iterazione

L'*iterazione* permette di ripetere l'esecuzione di una o piú istruzioni, in base al valore di una condizione.

Lo schema dell'iterazione è il seguente:

```
ESEGUI
   blocco
FINO A QUANDO condizione
```

Viene eseguito prima di tutto il `blocco` di istruzioni. Quindi si valuta la `condizione`. Se questa risulta vera si prosegue con l'istruzione scritta dopo il costrutto iterativo; se la `condizione` risulta falsa si ripete, eseguendo il `blocco`, e valutando nuovamente la `condizione`. Si noti in particolare che:

- il `blocco` viene eseguito sempre *almeno una volta*, in quanto la `condizione` viene valutata in coda;
- l'esecuzione del costrutto iterativo termina quando la `condizione` *diventa vera*.

Un altro schema iterativo è il seguente:

```
QUANDO condizione ESEGUI
   blocco
RIPETI
```

In questo caso viene valutata prima di tutto la `condizione`. Se essa risulta vera, allora si esegue il `blocco` e si valuta nuovamente la `condizione`. Se la `condizione` risulta falsa si passa ad eseguire l'istruzione che segue il costrutto iterativo, cioè l'istruzione che segue la parola `RIPETI`. Dunque:

- il `blocco` può essere eseguito anche zero volte, in quanto la `condizione` viene valutata in testa;
- l'esecuzione del costrutto iterativo termina quando la `condizione` *diventa falsa*.

Osserviamo che il comportamento dello schema `QUANDO ... RIPETI` può essere simulato combinando lo schema della selezione con lo schema `ESEGUI ... FINO A QUANDO...`, come segue

```
SE condizione
      ALLORA
         ESEGUI
            blocco
         FINO A QUANDO (condizione e' falsa)
FINESE
```

Esempio: calcolo della somma dei primi 100 numeri interi

Vogliamo calcolare la somma dei primi 100 numeri interi. Questo problema potrebbe essere risolto utilizzando semplicemente la formula di Gauss ($\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$). Tuttavia, al fine di mostrare l'uso dei costrutti iterativi, scriviamo un algoritmo che calcola iterativamente tale somma.

```
poni il valore della somma a zero
inizia a considerare il numero 1
ESEGUI
     aggiungi alla somma il numero che stai contando
     considera il numero successivo
FINO A QUANDO raggiungi un numero maggiore di 100
scrivi la somma
```

3.2 Variabili e assegnamenti

Nell'esempio relativo alla soluzione delle equazioni di secondo grado, sono stati utilizzati i nomi a, b e c per indicare i valori dei coefficienti, forniti dall'esterno. È stato inoltre utilizzato il nome discriminante per indicare il valore del discriminante, calcolato subito dopo la lettura dei dati. Possiamo immaginare che queste quantità vengano immagazzinate in appositi contenitori (detti *variabili*), che chiameremo proprio `a`, `b`, `c` e `discriminante`. È possibile assegnare un valore ad una variabile mediante un'istruzione di lettura, come

leggi a, b, c

in cui si ricevono dall'ambiente esterno tre valori, che vengono posti, rispettivamente, nei contenitori di nomi `a`, `b` e `c`, oppure mediante un'istruzione di *assegnamento*. Ad esempio, l'istruzione calcola il discriminante dell'algoritmo precedente, può essere rappresentata mediante il seguente assegnamento alla variabile `discriminante`:

`discriminante` $\leftarrow$ `b`^{2} `- 4 * a * c`

La *semantica operazionale* (cioè il significato dato in termini di operazioni da eseguire) di un'istruzione di assegnamento della forma

variabile $\leftarrow$ espressione

è il seguente:

1. viene calcolato il valore dell'espressione scritta a destra del simbolo $\leftarrow$;
2. il risultato ottenuto viene assegnato alla variabile (quindi posto nel contenitore) il cui nome è scritto a sinistra del simbolo $\leftarrow$, eliminando l'eventuale valore precedentemente presente.

Ad esempio per eseguire l'istruzione

`x` $\leftarrow$ `y + z`

si recuperano prima di tutto i valori contenuti nelle due variabili `y` e `z`, si calcola la loro somma e si pone il risultato nella variabile `x`, eliminando il valore precedentemente presente.

Se prima dell'assegnamento

`k` $\leftarrow$ `k + 1`

la variabile `k` conteneva il valore `7`, dopo l'esecuzione di tale assegnamento `k` conterrà il valore `8`. Si osservi che il nome `k` scritto a destra del simbolo $\leftarrow$ denota il valore contenuto nella variabile `k`, mentre il nome `k` scritto a sinistra denota la variabile a cui assegnare il risultato.

Ad ogni variabile viene associato un *tipo* che specifica la classe di valori che essa può assumere (e dunque anche l'insieme di operazioni che possono essere effettuate su di essa). Ad esempio una variabile `x` di tipo intero può assumere come valori solo numeri interi, e su di essa possono essere effettuate sole le operazioni permesse per i numeri interi. I valori e le operazioni possibili su una variabile `y` di tipo carattere saranno differenti da quelli permessi su `x`.

Si osservi che il concetto di *variabile* è un'*astrazione* del concetto di locazione di memoria; l'assegnamento di un valore a una variabile è un'astrazione dell'operazione `STORE`. Sebbene tutte le variabili siano rappresentate nella memoria come sequenze di bit, tali sequenze hanno una diversa interpretazione in base ai tipi. Pertanto la nozione di tipo fornisce un'astrazione rispetto alla rappresentazione effettiva dei dati: il programmatore può utilizzare variabili di tipi differenti, senza necessità di conoscerne l'effettiva rappresentazione.

Molti linguaggi, come il Pascal, richiedono di *dichiarare* le variabili all'inizio del programma, indicandone il tipo. Questo, oltre ad aumentare la leggibilità dei programmi, facilita la realizzazione di compilatori efficienti. Pertanto, nei prossimi esempi, indicheremo all'inizio i nomi e i tipi delle variabili utilizzate.

Riscriviamo ora l'algoritmo per la soluzione delle equazioni di secondo grado, in base a quanto abbiamo appena visto. In particolare introduciamo alcune variabili, esplicitiamo alcuni passi mediante assegnamenti ed esprimiamo le condizioni utilizzando gli *operatori di confronto* `<` e `=`:

```
variabili a, b, c, discriminante, x, x1, x2: numeri reali

leggi a, b, c
discriminante ← b^2 - 4 * a * c
SE discriminante < 0
    ALLORA
        scrivi "nessuna soluzione reale"
    ALTRIMENTI
        SE discriminante = 0
            ALLORA
                x ← - b / (2 * a)
                scrivi "Due soluzioni coincidenti: ", x
            ALTRIMENTI
                x1 ← (- b - √discriminante) / (2 * a)
                x2 ← (- b + √discriminante) / (2 * a)
                scrivi " Due soluzioni: ", x1, x2
        FINESE
FINESE
```

Rivediamo ora l'esempio relativo al calcolo della somma dei primi 100 numeri interi, introducendo esplicitamente delle variabili. È necessaria una variabile in cui memorizzare via via la somma. Chiameremo questa variabile `somma`. Serve inoltre un'altra variabile per sapere di volta in volta quale numero stiamo considerando. Poiché questa variabile funge da "contantore", in quanto ci permette di contare da 1 a 100, utilizzeremo il nome `cont`. Al fine di aumentare la leggibilità dei programmi, è opportuno infatti scegliere dei nomi che ricordino cosa rappresentano le singole variabili. In questo caso, entrambe le variabili sono destinate a contenere numeri interi.

L'algoritmo dato precedentemente può essere facilmente riscritto come segue:

```
variabili somma, cont: numeri interi

somma ← 0
cont ← 1
ESEGUI
    somma ← somma + cont
    cont ← cont + 1
FINO A QUANDO cont > 100
scrivi somma
```

Si osservi che l'assegnamento `somma ←somma + cont` ha come effetto quello di aggiungere alla variabile `somma` il valore della variabile `cont`.

Estendiamo ora l'esempio appena presentato, al fine di calcolare la somma dei numeri da 1 a n, dove n viene fornito da input. Possiamo introdurre una nuova variabile che riceve dall'esterno il valore n, sino al quale calcolare la somma, e modificare la condizione del ciclo:

```
variabili somma, cont, n: numeri interi

somma ← 0
cont ← 1
leggi n
ESEGUI
    somma ← somma + cont
    cont ← cont + 1
```

```
FINO A QUANDO cont > n
scrivi somma
```

L'algoritmo precedente opera correttamente nel caso il valore fornito in ingresso sia positivo. Se invece viene inserito in ingresso zero o un numero negativo, il risultato è scorretto. Infatti, se n è minore o uguale a zero, la somma dei primi n numeri è zero, mentre l'algoritmo fornirà come risultato 1.

Questo problema è dovuto al fatto che le istruzioni `somma` ←`somma + cont` e `cont` ←`cont + 1`, che si trovano all'interno del ciclo, vengono sempre eseguite almeno una volta, anche quando `cont` inizialmente è già maggiore di `n`. Il problema può essere facilmente risolto controllando la condizione all'inizio, e non alla fine del ciclo o, in altre parole, utilizzando un ciclo della forma `QUANDO ... RIPETI`:

```
variabili somma, cont, n: numeri interi

somma ← 0
cont ← 1
leggi n
QUANDO cont <= n ESEGUI
    somma ← somma + cont
    cont ← cont + 1
RIPETI
scrivi somma
```

Esercizi

1. Abbiamo visto come sia possibile simulare un ciclo della forma `QUANDO ... RIPETI` con un ciclo del forma `ESEGUI ... FINO A QUANDO ...`. Si mostri che è possibile anche la simulazione inversa, utilizzando un ciclo della forma `QUANDO ... RIPETI` in sostituzione di un ciclo della forma `ESEGUI ... FINO A QUANDO...`.

2. Descrivere, mediante le strutture di controllo fondamentali, un algoritmo che leggendo un numero intero, stabilisca se esso è pari o dispari.

3. Si consideri l'algoritmo per il calcolo delle radici di un'equazione di secondo grado. Cosa succede se il coefficiente a vale zero? Si modifichi l'algoritmo per gestire anche questa situazione.

4. Descrivere, mediante le strutture di controllo fondamentali, un algoritmo che calcoli il prodotto dei numeri da 1 a n, dove il valore n viene letto dall'esterno.

5. Descrivere, mediante le strutture di controllo fondamentali, un algoritmo che legga una sequenza di numeri interi, e scriva quanti di questi sono pari e quanti sono dispari (utilizzare il numero `0` per indicare la fine della sequenza in ingresso).

6. Supponendo che l'esecutore conosca solo le operazioni di somma e sottrazione, scrivere un algoritmo per il calcolo del prodotto di due numeri interi non negativi x e y ricevuti in ingresso, e un algoritmo per il calcolo del quoziente e del resto della divisione di x per y.

7. Supponendo che l'esecutore conosca le quattro operazioni, scrivere un algoritmo per il calcolo della potenza x^y di due interi non negativi x e y ricevuti in ingresso. Riscrivere poi l'algoritmo nel caso in cui l'esecutore conosca solo le operazioni di somma e sottrazione.

8. Si supponga che le variabili `x` e `y` contengano rispettivamente gli interi `4` e `10`. Quali valori conterranno dopo avere eseguito l'assegnamento `x` ← `y`? E se invece venisse eseguito l'assegnamento `y` ← `x`?