Lezione 8
25 ottobre 1999

Argomenti trattati
- Cicli `FOR` innestati.
- Alcuni esempi.

8.1 Ancora sull'uso dei cicli `FOR`

Presentiamo ora un semplice programma che legge due numeri non negativi w e z e calcola la potenza w^z. Il programma è basato su tre fasi: la prima di acquisizione dall'esterno dei dati, la seconda di calcolo, e la terza di comunicazione all'esterno del risultato.

Supponiamo che alla fine della prima fase i dati si trovino in due variabili di tipo `integer` di nomi `w` e `z`, rispettivamente. Decidiamo inoltre che la seconda fase dovrà porre il risultato in una variabile di nome `potenza`, anch'essa di tipo `integer`. Fissate queste convenzioni, è possibile sviluppare le tre frasi indipendentemente l'una dall'altra.

Concentriamoci dunque sulla fase di calcolo. Ricordando che w^z non è altro che w moltiplicato z volte per se stesso, non è difficile vedere che tale fase può essere realizzata mediante un ciclo `FOR`:

```
potenza := 1;
FOR k := 1 TO z DO
   potenza := potenza * w
```

Si osservi che l'istruzione interna al ciclo viene ripetuta esattamente `z` volte (se `z` contiene zero non viene mai eseguita).

Riportiamo il testo del programma completo:

```
PROGRAM power (input, output);

{calcola l'elevamento a potenza di due numeri non negativi}

   VAR
      z, w, k, potenza: integer;

BEGIN {power}
   {fase di lettura}
```

```
   write('Inserire due interi non negativi ');
   readln(w, z);
   WHILE (z < 0) OR (w < 0) DO
      BEGIN
         write('Errore nei dati: ripetere l''inserimento ');
         readln(w, z)
      END; {while}

   {fase di calcolo}
   potenza := 1;
   FOR k := 1 TO z DO
      potenza := potenza * w;

   {fase di scrittura}
   writeln('Il risultato e'' ', potenza : 1)
END. {power}
```

Vogliamo ora riscrivere il programma precedente, senza l'utilizzo diretto dell'operatore `*` per il calcolo del prodotto, ma facendo uso esclusivamente di operazioni di somma. In altre parole, dobbiamo riscrivere l'assegnamento `potenza := potenza * w`.

Cominciamo col dividere l'operazione di assegnamento nelle due parti fondamentali che la costituiscono, cioè il calcolo e l'assegnamento del risultato:

1. calcola il valore dell'espressione `potenza * w`;
2. assegna il risultato alla variabile `potenza`.

Per la prima fase, cioè il calcolo del prodotto `potenza * w`, possiamo utilizzare la tecnica vista nella lezione precedente, in cui abbiamo calcolato il prodotto di due interi non negativi utilizzando un ciclo `FOR`. A tale scopo introduciamo una nuova variabile di nome `prodotto`, in cui verrà memorizzato il risultato, e una variabile di nome `j` per il ciclo:

```
   prodotto := 0;
   FOR j := 1 TO w DO
      prodotto := prodotto + potenza
```

Per assegnare il risultato alla variabile `potenza`, sarà sufficiente scrivere `potenza := prodotto`. Pertanto, l'intera fase di calcolo, risulta riscritta come:

```
   potenza := 1;
   FOR k := 1 TO z DO
      BEGIN
         prodotto := 0;
         FOR j := 1 TO w DO
            prodotto := prodotto + potenza;
         potenza := prodotto
      END
```

Si osservi che il codice appena scritto contiene due costrutti `FOR` innestati. Per ogni esecuzione del costrutto esterno, cioè per ogni valore della variabile `k` da `1` fino a `z`, viene eseguito l'intero costrutto interno, con `j` che va da `1` a `w`. Pertanto, l'istruzione `prodotto := prodotto + potenza` verrà eseguita `w * z` volte: prima con `k`=`j`=`1`, poi con `k`=`1` e `j`=`2`, ..., poi con `k`=`1` e `j`=`w`, poi con `k`=`2` e `j`=`1`, e così via.

Le istruzioni interne al `FOR` piú esterno, come ad esempio l'assegnamento `potenza := prodotto`, vengono invece eseguite `z` volte.

La parte di codice appena presentata, è equivalente al seguente codice, in cui si utilizzano esclusivamente cicli `WHILE`:

```
   potenza := 1;
   k := 1;
   WHILE k <= z DO
      BEGIN
         prodotto := 0;
         j := 1;
         WHILE j <= w DO
            BEGIN
               prodotto := prodotto + potenza;
               j := j + 1
            END; {while j}
         potenza := prodotto;
         k := k + 1
      END {while k}
```

Si noti in particolare che ad ogni ripetizione delle istruzioni del ciclo esterno è necessario eseguire tutto il ciclo interno. Pertanto, il valore della variabile `j` viene fatto ripartire da `1` prima di ogni esecuzione del ciclo interno. Per esercizio, si consiglia di esaminare come sarebbe eseguito il ciclo, se l'assegnamento `j := 1` venisse spostato immediatamente dopo l'assegnamento `k := 1`.

Riportiamo ora il testo completo del programma nella seconda versione, cioè utilizzando due cicli `FOR` innestati per il calcolo della potenza. Sia in questa versione, che nella prima, il programma non gestisce correttamente il calcolo di zero elevato alla zero. Si introducano per esercizio le modifiche necessarie per operare correttamente anche questo calcolo.

```
PROGRAM power (input, output);

{calcola l'elevamento a potenza di due numeri non negativi}
{nel calcolo non viene utilizzata la moltiplicazione}

   VAR
      z, w, k, j, potenza, prodotto: integer;

BEGIN {power}
   {fase di lettura}
   write('Inserire due interi non negativi ');
   readln(w, z);
   WHILE (z < 0) OR (w < 0) DO
      BEGIN
         write('Errore nei dati: ripetere l''inserimento ');
         readln(w, z)
      END; {while}

   {fase di calcolo}
   potenza := 1;
   FOR k := 1 TO z DO
      BEGIN
         {simulazione dell'assegnamento potenza := potenza * w}
         {1 - calcolo di potenza * w}
         prodotto := 0;
         FOR j := 1 TO w DO
            prodotto := prodotto + potenza;
         {2 - assegnamento del risultato alla variabile potenza}
         potenza := prodotto
```

```
      END; {for k}

   {fase di scrittura}
   writeln('Il risultato e'' ', potenza : 1)
END. {power}
```

8.2 Alcuni suggerimenti per la scrittura dei programmi

Per migliorare la leggibilità dei programmi e rendere quindi piú agevoli le fasi di modifica e individuazione di errori, è indispensabile:

- Inserire dei *commenti* che descrivano a parole le varie fasi del programma. I commenti in Pascal vengono racchiusi da parentesi graffe `{...}`, o da parentesi tonde con asterischi `(*...*)`.
- Scegliere come *identificatori* dei nomi che ricordino il significato dell'oggetto (variabile, tipo...) a cui sono associati. Ad esempio, `primo` è un nome appropriato per una variabile di tipo `boolean` utilizzata per indicare se un numero è primo; il nome `flag` non ha invece alcun significato.
- Scrivere i programmi *indentando* (cioè facendo rientrare) opportunamente le varie istruzioni. Ogni `END` deve essere allineato sotto il `BEGIN` al quale si riferisce e le istruzioni racchiuse tra le due parole vanno scritte un po' piú a destra. Inoltre, quando vi siano piú blocchi innestati, è sempre opportuno scrivere dopo la parola `END` un breve commento che ricordi cosa c'era all'inizio del blocco che si sta chiudendo. Ad esempio, se le parole `BEGIN` ed `END` racchiudono un blocco di istruzioni da ripetere in un ciclo `WHILE`, dopo la parola `END` sarà bene aggiungere il commento `{while}`.
- L'uso di uno stile differente per le parole riservate e per gli identificatori aumenta notevolmente la leggibilità del programma, facilitando la comprensione della struttura a blocchi. Ad esempio, si possono scrivere le parole chiave con lettere maiuscole e gli identificatori con lettere minuscole. Se un identificatore è formato da piú parole, è bene utilizzare le maiuscole per le iniziali di ciascuna parola, come ad esempio `SommaPari` e `SommaDispari`. Esistono programmi che indentano automaticamente il testo di programmi Pascal, e che evidenziano in maiuscolo le parole riservate.

8.3 Esempio: decidere se un numero è primo

Sviluppiamo ora un programma che legge un numero e stabilisce se esso sia primo o no.

Prima di tutto, ricordiamo che un numero p è *primo* se valgono le seguenti condizioni:

1. $p > 1$;
2. p non ha divisori positivi, eccetto 1 e p stesso.

Il metodo immediato per verificare se un numero p è primo consiste quindi nel provare a dividere p per tutti i numeri compresi tra 2 e $p - 1$.
Possiamo quindi descrivere un algoritmo, basato su questo metodo, formato dalle seguenti fasi:

1. lettura del numero;
2. divisioni del numero per i numeri compresi tra 2 e il numero meno uno, per deciderne la primalità;

3. scrittura del risultato.

Per la scrittura del programma ci serviranno due variabili di tipo `integer`, che chiameremo `numero` e `divisore`, per memorizzare rispettivamente il numero di cui si deve testare la primalità e il divisore con cui si tenta di dividere il numero. Utilizzeremo inoltre una variabile di nome `primo` e di tipo `boolean`. Alla fine della seconda fase, la variabile `primo` dovrà contenere `true` se e solo se il numero dato è primo. Con questa convenzione, è immediato scrivere la fase finale del programma:

```
{scrittura del risultato}
IF primo THEN
   writeln('Il numero ', numero : 1, ' e'' primo')
ELSE
   writeln('Il numero ', numero : 1, ' non e'' primo')
```

Nella fase iniziale di acquisizione dei dati si richiede l'inserimento del numero da esaminare, fino a quando non venga inserito un numero maggiore di 1. Tale fase può dunque essere scritta utilizzando un ciclo `REPEAT` o un ciclo `WHILE`. Ecco una possibile soluzione:

```
{lettura dati}
write('Inserisci un intero maggiore di 1 ');
readln(numero);
WHILE numero <= 1 DO
   BEGIN
      write('Input non valido. Inserisci un intero maggiore di 1 ');
      readln(numero)
   END;{while}
```

Consideriamo ora la fase centrale. Abbiamo stabilito che alla fine di questa fase, la variabile `primo` dovrà contenere `true` se e solo se `numero` è primo. Il metodo piú semplice è quello di assegnare a `primo` inizialmente il valore `true`, di dividere poi `numero` iterativamente per tutti i valori tra `2` e `numero-1`, assegnando a `primo` valore `false` nel caso in cui una delle divisioni dia resto uguale a zero. Questa fase può dunque essere codificata utilizzando un ciclo `FOR`:

```
{verifica se numero e' primo, dividendolo per tutti i numeri tra 2 e numero-1}
primo := true;
FOR divisore := 2 TO numero - 1 DO
   IF numero MOD divisore = 0 THEN
      primo := false;
```

Ecco il testo del programma completo:

```
PROGRAM primi (input, output);

{Dato in ingresso un intero maggiore di 1, verifica se questo e' primo}

   VAR
      numero, divisore: integer;
      primo: boolean;

BEGIN {primi}

   {lettura dati}
   write('Inserisci un intero maggiore di 1 ');
   readln(numero);
   WHILE numero <= 1 DO
```

```
      BEGIN
         write('Input non valido. Inserisci un intero maggiore di 1 ');
         readln(numero)
      END;{while}

   {verifica se numero e' primo, dividendolo per tutti i numeri tra 2 e numero-1}
   primo := true;
   FOR divisore := 2 TO numero - 1 DO
      IF numero MOD divisore = 0 THEN
         primo := false;

   {scrittura del risultato}
   IF primo THEN
      writeln('Il numero ', numero : 1, ' e'' primo')
   ELSE
      writeln('Il numero ', numero : 1, ' non e'' primo')

END. {primi}
```

È possibile apportare al programma alcuni miglioramenti. Prima di tutto possiamo osservare che si potrebbe uscire dal ciclo non appena viene individuato un divisore di `numero`. A tale fine, possiamo sostituire il ciclo `FOR` con un ciclo `WHILE`, aggiungendo un controllo sulla variabile `primo`.

```
   {verifica se numero e' primo, dividendolo per i numeri tra 2 e numero-1}
   primo := true;
   divisore := 2;
   WHILE primo AND (divisore < numero) DO
      BEGIN
         primo := numero MOD divisore <> 0;
         divisore := divisore + 1
      END; {while}
```

Osserviamo poi che se `numero` ammette un divisore, allora ammette sicuramente anche un divisore minore o uguale alla propria radice quadrata. Questo permette di ridurre il numero di ripetizioni del ciclo:

```
   {verifica se numero e' primo, dividendolo per i numeri tra 2 e
    la radice quadrata di numero}
   primo := true;
   divisore := 2;
   WHILE primo AND (divisore * divisore <= numero) DO
      BEGIN
         primo := numero MOD divisore <> 0;
         divisore := divisore + 1
      END; {while}
```

Infine, ricordando che l'unico numero primo pari è 2, possiamo evitare le divisioni per numeri pari:

```
   {verifica se numero e' primo, controllando prima di tutto che non sia pari, e
    poi dividendolo per i numeri dispari tra 3 e la radice quadrata di numero}
   IF numero = 2 THEN
      primo := true
   ELSE
```

```
      BEGIN
         {controlla se il numero e' pari}
         primo := numero MOD 2 <> 0;
         divisore := 3;
         WHILE primo AND (divisore * divisore <= numero) DO
            BEGIN
               primo := numero MOD divisore <> 0;
               divisore := divisore + 2
            END {while}
      END; {else}
```

Ecco il programma completo, dopo queste modifiche:

```
PROGRAM primi4 (input, output);

{Dato in ingresso un intero maggiore di 1, verifica se questo e' primo}
{In questa versione si esce dal ciclo di controllo non appena si trova un
 divisore o quando si sono esaminati tutti i numeri fino alla radice
 quadrata del numero dato. Inoltre non si esaminano divisori pari (eccetto 2)}

   VAR
      numero, divisore: integer;
      primo: boolean;

BEGIN {primi4}

   {lettura dati}
   write('Inserisci un intero maggiore di 1 ');
   readln(numero);
   WHILE numero <= 1 DO
      BEGIN
         write('Input non valido. Inserisci un intero maggiore di 1 ');
         readln(numero)
      END;{while}

   {verifica se numero e' primo, controllando prima di tutto che non sia pari, e
    poi dividendolo per i numeri dispari tra 3 e la radice quadrata di numero}
   IF numero = 2 THEN
      primo := true
   ELSE
      BEGIN
         {controlla se il numero e' pari}
         primo := numero MOD 2 <> 0;
         divisore := 3;
         WHILE primo AND (divisore * divisore <= numero) DO
            BEGIN
               primo := numero MOD divisore <> 0;
               divisore := divisore + 2
            END {while}
      END; {else}

   {scrittura del risultato}
   IF primo THEN
```

```
      writeln('Il numero ', numero : 1, ' e'' primo')
   ELSE
      writeln('Il numero ', numero : 1, ' non e'' primo')

END. {primi4}
```

Esercizi

1. Il programma che verifica se un numero è primo, dopo avere esaminato un numero fornisce la risposta e termina la propria esecuzione. In questo modo, l'utente che voglia esaminare piú numeri è costretto a lanciare piú volte il programma. Modificare il programma in modo tale che, dopo avere esaminato un numero e fornito la risposta, chieda all'utente se vuole continuare, esaminando altri numeri, o se vuole terminare l'esecuzione.

2. Scrivere un programma che riceva in ingresso un numero intero n e stampi un elenco di tutti i numeri primi compresi tra 1 e n.

3. Scrivere un programma che riceva in ingresso un numero intero n e stampi un elenco dei primi n numeri primi.

4. Per ognuno dei seguenti frammenti di codice, individuare delle dichiarazioni di variabile in modo che le istruzioni che vi appaiono risultino corrette dal punto di vista della compatibilità dei tipi. Indicare inoltre, in funzione dei valori iniziali delle variabili, l'istruzione indicata con il commento `{*}`. Riscrivere infine il frammento sostituendo i cicli `FOR` con cicli `WHILE`.

   - ```
     FOR x := succ(x) DOWNTO pred(x) DO
        writeln(x) {*}
     ```
   - ```
     FOR i := 0 TO n DO
        FOR j := m DOWNTO n DO
           writeln(i, j) {*}
     ```
   - ```
     FOR i := 1 TO n DO
        n := n + i {*}
     ```
   - ```
     FOR c := 'A' TO 'Z' DO
        FOR j := 'z' DOWNTO 'a' DO
           writeln(c, j) {*}
     ```
   - ```
     FOR i := 1 TO n DO
        FOR j := 1 TO i DO
           writeln(i, j) {*}
     ```
   - ```
     FOR i := -m TO n DO
        FOR j := -10 TO 10 DO
           writeln(i, j) {*}
     ```
   - ```
     FOR x := 1 TO n DO
        FOR b := false TO true DO
           writeln(i, j) {*}
     ```
   - ```
     FOR x := 1 TO n DO
        FOR b := true TO false DO
           writeln(i, j) {*}
     ```
   - ```
     FOR i := 1 TO n DO
        FOR j := n DOWNTO 1 DO
           FOR k:= -n TO n DO
              writeln(i, j, k) {*}
     ```
     ```