Lezione 16
23–24 novembre 1999

Argomenti trattati
- Sottoprogrammi ricorsivi.
- Direttiva `forward`.

16.1 Sottoprogrammi ricorsivi

Studiando la visibilità delle risorse, abbiamo osservato che un sottoprogramma è visibile a se stesso. Dunque, un sottoprogramma può richiamare se stesso. Questa possibilità viene detta *ricorsione*; i sottoprogrammi che la utilizzano sono detti *ricorsivi*. Presenteremo alcuni esempi di sottoprogrammi ricorsivi e ne studieremo il comportamento dinamico.

In generale, un oggetto si dice *ricorsivo* se viene definito in termini di se stesso. In matematica si incontrano tipici esempi di oggetti definiti ricorsivamente, tramite le *definizioni per ricorrenza* o *definizioni induttive*. Ad esempio, il *fattoriale* di un numero intero non negativo n, indicato con $n!$, può essere definito come:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{altrimenti} \end{cases}$$

La definizione precedente contiene un *caso base* (corrispondente a $n = 0$) e un *caso induttivo* (corrispondente alla parola "altrimenti") in cui la definizione di $n!$ viene data in termini di $(n-1)!$. Ad esempio, utilizzando il caso induttivo, otteniamo la catena di uguaglianze:

$$3! = 3 \cdot 2! = 3 \cdot (2 \cdot 1!) = 3 \cdot (2 \cdot (1 \cdot 0!)).$$

A questo punto, possiamo applicare il caso base, sostituendo 1 al posto di 0!, e calcolare il risultato:

$$3 \cdot (2 \cdot (1 \cdot 1)) = 3 \cdot (2 \cdot 1) = 3 \cdot 2 = 6.$$

Data la definizione induttiva, è immediato scrivere in Pascal una `FUNCTION fatt` per il calcolo del fattoriale di `n`, che ricalchi la definizione stessa:

- se il valore di `n` è zero, la funzione restituisce immediatamente `1` come risultato;
- negli altri casi la funzione chiama se stessa per il calcolo del fattoriale di `n-1` e, una volta ottenuto il risultato, lo moltiplica per `n`, restituendolo all'esterno (si suppone che la funzione venga chiamata con un valore per il parametro `n` non negativo).

La funzione è dunque:

```
FUNCTION fatt (n: integer): integer;

BEGIN {fatt}
   IF n = 0 THEN
      fatt := 1
   ELSE
      fatt := n * fatt(n - 1)
END; {fatt}
```

Inseriamo ora la funzione all'interno di un programma e ne effettuiamo la simulazione dinamica con input `3`.

```
PROGRAM fattoriale (input, output);

   VAR
      x, f: integer;


   FUNCTION fatt (n: integer): integer;

   {restituisce il fattoriale del parametro dato}

   BEGIN {fatt}
      IF n = 0 THEN
         fatt := 1
      ELSE
         fatt := n * fatt(n - 1) {*}
   END; {fatt}


BEGIN {fattoriale}
   {lettura dato}
   write('Inserire un numero non negativo ');
   readln(x);
   WHILE x < 0 DO
   BEGIN
      write('L''input non puo'' essere negativo. Ripetere l''inserimento ');
      readln(x)
   END; {while}

   {calcolo fattoriale}
   f := fatt(x) {+};

   {scrittura risultato}
   writeln('Il fattoriale di ', x : 1, ' e'' uguale a ', f : 1)
END. {fattoriale}
```

L'esecuzione inizia creando sullo stack il record di attivazione del programma principale. Dopo la lettura del dato in `input`, lo stack conterrà:

PROG. fattor.	
x	3
f	?
ris.fz.	?
ritorno	?

L'istruzione successiva è un assegnamento, sul cui lato destro si trova una chiamata alla funzione `fatt`, con parametro attuale `x`. Pertanto viene creato sullo stack un record di attivazione per `fatt` in cui, nello spazio per il parametro formale `n`, viene copiato il valore di `x` (passaggio per valore). Inoltre, nel record di attivazione del programma principale viene memorizzato il punto di ritorno, indicato nel testo da {+}:

PROG. fattor.	
x	3
f	?
ris.fz.	?
ritorno	{+}
FUNCT. fatt	
n	3
ris.fz.	?
ritorno	?

L'esecuzione di `fatt` ha inizio valutando la condizione dell'`IF`. Poiché risulta falsa, si esegue il ramo `ELSE`, che contiene una chiamata alla funzione `fatt` stessa.

Questa chiamata viene trattata nel solito modo: si memorizza il punto di ritorno nell'ambiente chiamante, si crea un nuovo record attivazione in cima allo stack, si effettua il passaggio dei parametri. In particolare, il parametro attuale della chiamata è l'espressione `n-1`. Ne viene calcolato il valore, cioè `2`, nell'ambiente chiamante. Tale valore viene copiato nel parametro formale del nuovo ambiente, cioè nella variabile `n` del nuovo record di attivazione. Il contenuto dello stack diventa quindi:

PROG. fattor.	
x	3
f	?
ris.fz.	?
ritorno	{+}
FUNCT. fatt	
n	3
ris.fz.	?
ritorno	{*}
FUNCT. fatt	
n	2
ris.fz.	?
ritorno	?

Analogamente al caso precedente, l'esecuzione inizia valutando la condizione. Poiché è falsa si procede ad effettuare un'ulteriore chiamata ricorsiva, ottenendo sullo stack:

PROG. fattor.	
x	3
f	?
ris.fz.	?
ritorno	{+}
FUNCT. fatt	
n	3
ris.fz.	?
ritorno	{*}
FUNCT. fatt	
n	2
ris.fz.	?
ritorno	{*}
FUNCT. fatt	
n	1
ris.fz.	?
ritorno	?

Di nuovo, eseguendo il codice della funzione, si deve effettuare una chiamata ricorsiva. Dopo la chiamata il contenuto dello stack è:

PROG. fattor.	
x	3
f	?
ris.fz.	?
ritorno	{+}
FUNCT. fatt	
n	3
ris.fz.	?
ritorno	{*}
FUNCT. fatt	
n	2
ris.fz.	?
ritorno	{*}
FUNCT. fatt	
n	1
ris.fz.	?
ritorno	{*}
FUNCT. fatt	
n	0
ris.fz.	?
ritorno	?

Di nuovo, si comincia ad eseguire il codice della funzione. In questo caso la condizione `n = 0` risulta vera. Pertanto viene eseguito il ramo **THEN**, in cui la funzione restituisce valore `1` nell'ambiente chiamante. Inoltre, questa è l'ultima istruzione, pertanto si rientra dalla funzione, distruggendo il record di attivazione che si trova in cima allo stack. A questo punto, il contenuto dello stack è:

PROG. fattor.	
x	3
f	?
ris.fz.	?
ritorno	{+}
FUNCT. fatt	
n	3
ris.fz.	?
ritorno	{*}
FUNCT. fatt	
n	2
ris.fz.	?
ritorno	{*}
FUNCT. fatt	
n	1
ris.fz.	1
ritorno	{*}

Al rientro dalla funzione, l'esecuzione va ripresa dal punto memorizzato nel campo `ritorno` del record di attivazione che si trova ora in cima allo stack. Usando il risultato ottenuto dalla chiamata ricorsiva della funzione, possiamo ora calcolare il valore dell'espressione `n * fatt(n - 1)`, moltiplicando appunto il contenuto di `n` nel record di attivazione corrente, per il risultato che ha fornito la funzione, memorizzato nel campo `ris.fz.` Il risultato dell'espressione è `1`, che a sua volta viene restituito, mediante l'istruzione `fatt := n * fatt(n - 1)` all'ambiente esterno.

Dopo il rientro dalla funzione, il contenuto dello stack è dunque:

PROG. fattor.	
x	3
f	?
ris.fz.	?
ritorno	{+}
FUNCT. fatt	
n	3
ris.fz.	?
ritorno	{*}
FUNCT. fatt	
n	2
ris.fz.	1
ritorno	{*}

Di nuovo, si riprende dal punto {*} e si procede eseguendo l'istruzione `fatt := n * fatt(n - 1)`, mediante la quale si restituisce `2` all'ambiente esterno. Dopo il rientro dalla funzione, il contenuto dello stack è:

PROG. fattor.	
x	3
f	?
ris.fz.	?
ritorno	{+}
FUNCT. fatt	
n	3
ris.fz.	2
ritorno	{*}

Anche qui, il punto di rientro memorizzato è {*}. Eseguendo l'istruzione `fatt := n * fatt(n - 1)`, si restituisce `6` all'ambiente esterno. Dopo il rientro dalla funzione, il contenuto dello stack è:

PROG. fattor.	
x	3
f	?
ris.fz.	6
ritorno	{+}

Si rientra ora al punto {+}, dove si completa l'esecuzione dell'assegnamento `f := fatt(x)`, copiando il valore restituito dalla funzione (che si trova nel campo `ris.fz`), nella variabile `f`. Il contenuto dello stack diventa dunque:

PROG. fattor.	
x	3
f	6
ris.fz.	6
ritorno	{+}

Esempio: stringhe palindrome

Riprendiamo l'esempio presentato nella Lezione 14, in cui abbiamo costruito un programma per controllare se una stringa letta da `input` sia palindroma. Tale programma è basato su una `FUNCTION` con la seguente intestazione:

```
FUNCTION palindroma (s: stringa): boolean;
```

La `FUNCTION palindroma`, ricevendo come parametro la stringa da esaminare, deve restituire `true` se essa è palindroma, `false` altrimenti. Nella Lezione 14 la `FUNCTION palindroma` è stata sviluppata facendo uso di un algoritmo *iterativo*. Svilupperemo ora una soluzione differente, facendo uso della ricorsione.

A tale scopo, cerchiamo di esprimere in maniera induttiva cos'è una parola palindroma. Possiamo osservare che se il primo e l'ultimo simbolo di una parola sono differenti allora la parola non è palindroma; se il primo e l'ultimo simbolo sono uguali, la parola è palindroma se e solo se la parola che si ottiene cancellando il primo e l'ultimo simbolo è ancora palindroma. Inoltre, la parola vuota e le parole costituite da un solo simbolo sono palindrome. In altri termini possiamo dare la seguente definizione:

Una parola p si dice palindroma quando:

- p è formata da zero simboli o da un unico simbolo, oppure
- il primo e l'ultimo simbolo di p coincidono e la parola che si ottiene cancellando da p il primo e l'ultimo simbolo è a sua volta palindroma.

Scriveremo ora una `FUNCTION` basata sulla definizione precedente. Ricordiamo che, nella rappresentazione che abbiamo scelto, una parola è rappresentata da un record `s` con un campo `contenuto` (un `PACKED ARRAY` che contiene i caratteri che costituiscono la parola) e un campo `lung` che indica quanti sono i caratteri significativi. In pratica, la parola è costituita dai caratteri di `contenuto` che si trovano memorizzati nelle posizioni da `1` a `lung`.

Nella nuova versione della `FUNCTION palindroma` creiamo un'ulteriore `FUNCTION pal` con la seguente intestazione:

```
FUNCTION pal (sx, dx: integer): boolean;
```

Compito di questa `FUNCTION`, che verrà definita all'interno di `palindroma`, è di verificare che la porzione di stringa rappresentata nella parte di `contenuto` che va dalla posizione `sx` sino alla posizione `dx` sia palindroma. Utilizzando la funzione `pal`, il risultato di `palindroma` si ottiene semplicemente chiamando `pal` sull'intera stringa. In altra parole, la `FUNCTION palindroma` può essere scritta come:

```
FUNCTION palindroma (s: stringa): boolean;

{restituisce true se e solo se s rappresenta una parola palindroma}

   ...blocco FUNCTION pal...

BEGIN {palindroma}
   palindroma := pal(1, s.lung)
END; {palindroma}
```

Sviluppiamo ora la `FUNCTION pal`, basandoci sulla definizione induttiva di parola palindroma che abbiamo scritto sopra.

Compito della funzione è quello di verificare che la parola p costituita dai caratteri dell'array `s.contenuto` che vanno dalla posizione `sx` sino alla posizione `dx` sia palindroma. Un primo caso evidenziato nella definizione è quello in cui la parola è formata da zero o un solo simbolo. In questo caso possiamo restituire direttamente `true`. Per verificare che la parola sia costituita da zero o un simbolo è sufficiente controllare che il valore di `sx` sia maggiore o uguale al valore di `dx`.

In caso contrario, la parola è palindroma se il primo e l'ultimo simbolo, cioè `s.contenuto[sx]` e `s.contenuto[dx]` coincidono e la parte di parola che sta in mezzo, cioè che va dalla posizione `sx + 1` sino alla posizione `dx - 1`, è palindroma. Per effettuare questa verifica chiamiamo ricorsivamente la funzione `pal` con parametri `sx + 1` e `dx - 1`. Viceversa, se i simboli in posizione `sx` e in posizione `dx` sono differenti, la parola sicuramente non è palindroma. Dunque, possiamo restituire direttamente `false`.

Il codice completo della `FUNCTION palindroma`, con la `FUNCTION pal` al suo interno, è:

```
FUNCTION palindroma (s: stringa): boolean;

{restituisce true se e solo se s rappresenta una parola palindroma}

   FUNCTION pal (sx, dx: integer): boolean;

   {restituisce true se e solo se la porzione di s compresa}
   {tra le posizioni i e j e' palindroma}

   BEGIN {pal}
      IF (sx >= dx) THEN
         pal := true
      ELSE IF s.contenuto[sx] = s.contenuto[dx] THEN
         pal := pal(sx + 1, dx - 1)
      ELSE
         pal := false
   END; {pal}

BEGIN {palindroma}
   palindroma := pal(1, s.lung)
END; {palindroma}
```

16.2 Osservazioni sul passaggio di array ai sottoprogrammi

La `FUNCTION palindroma` ha un parametro di tipo `stringa` che, ricordiamo, è un record contenente un campo array. Utilizzando il passaggio per valore, come abbiamo fatto, è necessario ricopiare l'intero record nel record di attivazione di `palindroma`. Se l'array è di grandi dimensioni, questa

operazione è dispendiosa sia in termini di tempo che in termini di spazio. In generale, quando si utilizzino come parametri `array` o strutture che contengano array, o anche altre strutture di grandi dimensioni, è preferibile sempre l'utilizzo del passaggio per riferimento, in cui l'unica informazione che va scambiata è il riferimento all'inizio della struttura.

16.3 Direttiva `forward`

È possibile spostare in avanti la dichiarazione di un sottoprogramma facendo uso della direttiva `forward`, evitando in questo modo la nidificazione. Ad esempio si potrebbe scrivere:

```
PROCEDURE p1 (VAR x: integer);
forward;

PROCEDURE p2 (VAR y: char);
BEGIN {p2}
   ...
END; {p2}

PROCEDURE p1;
BEGIN {p1}
   ...
END; {p1}
```

In questo caso, si dichiara l'intestazione di `p1`, con i relativi parametri, rimandando, mediante la direttiva `forward`, il resto della procedura a un punto piú avanti nel testo. In questo modo, dal codice di `p2` è possibile richiamare il codice di `p1`, perché l'intestazione di `p1` è dichiarata prima del codice di `p2`, e dal codice di `p1` è possibile richiamare `p2`.

Due procedure che si richiamino a vicenda sono dette *mutuamente ricorsive*. La direttiva `forward` va utilizzata solo se strettamente indispensabile. Nella maggior parte dei casi si può ottenere la ricorsione mutua utilizzando la nidificazione. Ad esempio, per scrivere un programma `p` contenente due procedure `p1` e `p2` mutuamente ricorsive, e tale che `p` possa richiamare `p1`, è sufficiente definire `p1` all'interno di `p` e `p2` all'interno di `p1`. La direttiva forward è necessaria solo se le procedure `p1` e `p2`, oltre a essere mutuamente ricorsive, devono essere richiamabili entrambe dal programma principale (e quindi devo essere definite all'interno di esso, allo stesso livello).

Esercizi

1. La successione dei numeri di *Fibonacci* è data da $1, 1, 2, 3, 5, 8, 13, 21$, ecc. In pratica, il primo e il secondo numero della successione sono uguali a 1, mentre ognuno degli altri numeri è uguale alla somma dei due numeri che lo precedono.

   Si costruisca, sia in versione iterativa che in versione ricorsiva, una funzione che, ricevuto come parametro un numero k, restituisca il k–esimo termine della successione di Fibonacci. Si scriva poi un programma che utilizzi tale funzione e lo si simuli "manualmente" evidenziando il contenuto dello stack.

2. Indicare come potrebbe essere strutturato un programma `p` contenente tre procedure `a`, `b` e `c`, per ognuno dei seguenti casi:

   - il programma principale può richiamare la procedura `a`, che a sua volta può richiamare `b`, che può richiamare `c`, che può richiamare `a`;
   - il programma principale può richiamare le procedure `a` e `b`, la procedura `a` può richiamare `b` e `c`, la procedura `b` può richiamare `c`;

- il programma principale può richiamare la procedura `a`, che a sua volta può richiamare le procedure `b` e `c`, la procedura `b` può richiamare `c`, la procedura `c` può richiamare `a`.

Nota: si risolva il problema utilizzando la nidificazione di procedure e non la direttiva `forward`.

3. Indicare come potrebbe essere strutturato un programma `p` contenente tre procedure `a`, `b` e `c`, per ognuno dei seguenti casi:

   - il programma principale può richiamare la procedura `a`, che a sua volta può richiamare `b` e `c`; la procedura `b` può richiamare `c`;
   - il programma principale può richiamare le procedure `a` e `c`, la procedura `a` può richiamare `b` e `c`, la procedura `b` può richiamare `a`;
   - il programma principale può richiamare la procedura `a`, che a sua volta può richiamare le procedure `b`, che a sua volta può richiamare `c`, che a sua volta può richiamare `a`.

   Nota: si risolva il problema utilizzando la nidificazione di procedure e non la direttiva `forward`.

4. Per ognuno dei seguenti tre casi, indicare come potrebbe essere strutturato un programma `p` che contenga tre procedure `a`, `b` e `c`, e nel quale non sia utilizzata la direttiva `forward`, in modo tale che:

   - il programma principale possa richiamare la procedura `a` e la procedura `b`; la procedura `a` possa richiamare la procedura `c`, la procedura `b` possa richiamare sia la procedura `a` che la procedura `c`.
   - il programma principale possa richiamare la procedura `a` e la procedura `b`, ma *non* la procedura `c`; la procedura `a` possa richiamare la procedura `c`; la procedura `b` possa richiamare la procedura `a`;
   - il programma principale possa richiamare la procedura `b` e la procedura `c`; la procedura `a` possa richiamare la procedura `b`; la procedura `b` possa richiamare sia la procedura `a` che la procedura `c`.

5. Si scrivano tre funzioni ricorsive che dati due parametri interi non negativi `x` e `y` restituiscano il valore di `x` elevato alla `y`. La prima funzione deve essere basata sulla seguente uguaglianza:

$$x^y = \begin{cases} x \cdot x^{y-1} & \text{se } y > 0, \\ 1 & \text{se } y = 0. \end{cases}$$

   La seconda e la terza funzione trattano differentemente il caso di potenze pari e potenze dispari:

$$x^y = \begin{cases} x \cdot x^{y-1} & \text{se } y > 0 \text{ e } y \text{ è dispari}, \\ \left(x^{\frac{y}{2}}\right)^2 & \text{se } y > 0 \text{ e } y \text{ è pari}, \\ 1 & \text{se } y = 0. \end{cases}$$

   Per il calcolo di $\left(x^{\frac{y}{2}}\right)^2$, nella seconda funzione si utilizzino due chiamate ricorsive che calcolano entrambe $x^{\frac{y}{2}}$ e si calcoli quindi il prodotto dei risultati. Nella terza funzione, invece, si utilizzi una sola chiamata ricorsiva, che calcoli $x^{\frac{y}{2}}$, pondendo il risultato in una variabile, che viene poi moltiplicata per se stessa per ottenere il quadrato.

   Si esaminino le tre funzioni, al fine di identificare, al variare di `y`, il numero totale di chiamate ricorsive (si consiglia di considerare prima il caso in cui `y` è potenza di `2`). Si verifichi sperimentalmente il risultato ottenuto, scrivendo un programma per la prova delle funzioni, in cui, ad ogni chiamata ricorsiva, venga incrementato un contatore, da stampare alla fine dell'esecuzione allo scopo di conoscere il numero totale di chiamate effettuate.

6. Scrivere in Pascal una `FUNCTION` per il calcolo del massimo comun divisore tra due numeri non negativi, basata sulla seguente uguaglianza:

$$mcd(x, y) = \begin{cases} x & \text{se } y = 0 \\ mcd(y, x \bmod y) & \text{altrimenti.} \end{cases}$$

7. Scrivere una `FUNCTION` Pascal che riceva un parametro intero n e restituisca il valore $f(n)$, dove f è la funzione definita come

$$f(n) = \begin{cases} 1 & \text{se } n = 0 \\ n * (f(n-1) + 1) & \text{se } n > 0. \end{cases}$$

8. Scrivere una `FUNCTION` Pascal che riceva un parametro intero n e restituisca il valore $f(n)$, dove f è la funzione definita come

$$f(n) = \begin{cases} 0 & \text{se } n = 0 \\ n + f(n-1) & \text{se } n > 0. \end{cases}$$

9. Scrivere una `FUNCTION` Pascal che riceva due parametri interi x e y e restituisca il valore $f(x, y)$, dove f è la funzione definita come

$$f(x, y) = \begin{cases} 1 & \text{se } y = 0 \\ 3 + f(y-1, x) & \text{se } y > 0. \end{cases}$$

10. Scrivere una `FUNCTION` Pascal che riceva due parametri interi x e y e restituisca il valore $f(x, y)$, dove f è la funzione definita come

$$f(x, y) = \begin{cases} 0 & \text{se } y = 0 \\ x + f(x, y-1) & \text{se } y > 0. \end{cases}$$

11. Scrivere l'output prodotto dal seguente programma su input `4`

```
PROGRAM p (input, output);

   VAR
      x, y: integer;

   FUNCTION f (n: integer): integer;

   BEGIN {f}
      IF n = 0 THEN
         f := 1
      ELSE
         f := n * (f(n - 1) + 1)
   END; {f}

BEGIN {p}
   readln(x);
   y := f(x);
   writeln(y)
END. {p}
```

12. Scrivere l'output prodotto dal seguente programma su input `3 2`.

```
PROGRAM p (input, output);

   VAR
      x, y: integer;

   FUNCTION f (m, n: integer): integer;

   BEGIN {f}
      IF n = 0 THEN
         f := 1
      ELSE
         f := 3 + f(n - 1, m)
   END; {f}

BEGIN {p}
   readln(x);
   readln(y);
   y := f(x, y);
   writeln(y)
END. {p}
```

13. Scrivere l'output prodotto dal seguente programma su input `5 3`.

```
PROGRAM p (input, output);

   VAR
      x, y: integer;

   FUNCTION f (m, n: integer): integer;

   BEGIN {f}
      IF n = 0 THEN
         f := 0
      ELSE
         f := m + f(m, n - 1)
   END; {f}

BEGIN {p}
   readln(x);
   readln(y);
   y := f(x, y);
   writeln(y)
END. {p}
```

14. Scrivere l'output prodotto dal seguente programma su input `8`.

```
PROGRAM p (input, output);

   VAR
      x, y: integer;

   FUNCTION f (n: integer): integer;
```

```
   BEGIN {f}
      IF n = 1 THEN
         f := 1
      ELSE
         f := n + f(n DIV 2)
   END; {f}

BEGIN {p}
   readln(x);
   y := f(x);
   writeln(y)
END. {p}
```

15. Scrivere l'output prodotto dal seguente programma su input `3`.

```
PROGRAM p (input, output);
   VAR
      x, y: integer;
   FUNCTION f (n, m: integer): integer;
   BEGIN
      IF n + m <= 4 THEN
         f := n
      ELSE IF n > m THEN
         f := 2 + f(m + 1, n - 2)
      ELSE
         f := 3 * f(n + 1, m - 2)
   END;
BEGIN
   readln(x);
   y := f(x, x);
   writeln(y)
END.
```

16. Scrivere l'output prodotto dal seguente programma su input `7`.

```
PROGRAM p1 (input, output);
   VAR
      x, y: integer;
   FUNCTION f (n: integer): integer;
   BEGIN
      IF n <= 1 THEN f := 1
      ELSE f := 3 * f(n DIV 2) + 4 * f(n MOD 2)
   END;
BEGIN
   readln(x);
   y := f(x);
   writeln(y)
END.
```