Compitino 3

20–21 gennaio 2000

Correzione del compitino del Turno 1

Esercizio 1

Si considerino le seguenti definizioni di tipo:

```
TYPE
   tipolista = ^nodolista;
   nodolista = RECORD
                  info: integer;
                  pros: tipolista
               END;
```

Scrivere in Pascal una `PROCEDURE` che riceva, tramite un parametro di tipo `tipolista`, un puntatore a una lista di numeri interi, e modifichi tale lista inserendo al suo inizio due nuovi nodi: il primo contenente il numero di valori pari presenti nella lista data, il secondo contenente il numero di valori dispari presenti nella lista data. Ad esempio, se la lista contiene inizialmente i valori `9 7 6 13`, dopo l'esecuzione della `PROCEDURE` dovrà contenere `1 3 9 7 6 13`.

Scriviamo prima di tutto l'intestazione della `PROCEDURE` che, come stabilito dal testo, ha un unico parametro di tipo `tipolista`. Per effettuare gli inserimenti all'inizio della lista, la procedura dovrà essere in grado di modificare il puntatore al primo elemento. Pertanto il parametro dovrà essere passato *per riferimento*. L'intestazione sarà dunque:

```
PROCEDURE modifica (VAR l: tipolista);
```

La procedura può essere suddivisa in due fasi: nella prima si scandisce la lista contando il numero di valori pari e di valori dispari presenti, nella seconda si effettua l'inserimento all'inizio della lista dei nuovi nodi richiesti.

La fase di scansione può essere realizzata con un ciclo in cui, mediante un puntatore ausiliario `p`, si attraversa l'intera lista, utilizzando due variabili `npari` e `ndispari` per contare il numero di elementi pari e dispari presenti:

```
p := l;
npari := 0;
ndispari := 0;
WHILE p <> NIL DO
   BEGIN
      IF p^.info MOD 2 = 0
         THEN npari := npari + 1
         ELSE ndispari := ndispari + 1;
      p := p^.pros
   END
```

A questo punto è sufficiente effettuare l'inserimento dei due nuovi nodi all'inizio della lista: prima si inserisce il nodo contenente il numero di valori dispari, poi quello contenente il numero di valori pari (dunque quest'ultimo risulterà essere il primo nodo della lista).

```
new(p);
p^.info := ndispari;
p^.pros := l;
l := p;
new(p);
p^.info := npari;
p^.pros := l;
l := p
```

Si osservi che se la lista inizialmente era vuota, in questo modo vengono inseriti due nodi contenenti il valore 0. La procedura completa è:

```
PROCEDURE modifica (VAR l: tipolista);

   VAR
      p: tipolista;
      npari, ndispari: integer;

BEGIN {modifica}

   {scansione della lista e conteggio del numero di elementi pari e
    dispari}
   p := l;
   npari := 0;
   ndispari := 0;
   WHILE p <> NIL DO
      BEGIN
         IF p^.info MOD 2 = 0
            THEN npari := npari + 1
            ELSE ndispari := ndispari + 1;
         p := p^.pros
      END;

   {creazione e inserimento dei nuovi nodi}
   new(p);
   p^.info := ndispari;
   p^.pros := l;
   l := p;
   new(p);
```

```
   p^.info := npari;
   p^.pros := l;
   l := p

END; {modifica}
```

Esercizio 2

Si considerino le seguenti definizioni di tipo:

```
TYPE
   tipoalbero = ^nodoalbero;
   nodoalbero = RECORD
                  info: integer;     {informazione contenuta nel nodo}
                  sx, dx: tipoalbero {puntatori ai sottoalberi sinistro e destro}
                END;
```

Scrivere in Pascal una `FUNCTION` che riceva, tramite un parametro di tipo `tipoalbero`, il puntatore a un albero di numeri interi, e restituisca come risultato la somma di tutti i numeri primi memorizzati nell'albero.
Per verificare se un numero è primo, si supponga di disporre di una `FUNCTION primo (n: integer): boolean` (di cui non è richiesta la codifica), che restituisca `true` se il parametro utilizzato nella chiamata è un numero primo, `false` altrimenti.

In base al testo del problema, la `FUNCTION` deve ricevere come parametro un puntatore di tipo `tipoalbero` e restituire un valore di tipo `integer`. Pertanto l'intestazione sarà:

```
FUNCTION SommaPrimi (a: tipoalbero): integer;
```

Per sviluppare la funzione consideriamo la definizione ricorsiva di albero.

- Nel caso di *albero vuoto* la somma è zero.
- Nel caso di *albero costituito da una radice piú due sottoalberi sinistro e destro*, supponendo di conoscere la somma dei numeri primi contenuti nel sottoalbero sinistro, che indichiamo con `sommasx`, e la somma dei numeri primi contenuti nel sottoalbero destro, che indichiamo con `sommadx`, per calcolare la somma dei numeri primi contenuti nell'intero albero dobbiamo considerare i seguenti due casi:
  - Se *il valore contenuto nella radice è primo* allora la somma richiesta si ottiene sommando `sommasx`, `sommadx` e il valore contenuto nella radice;
  - se *il valore contenuto nella radice non è primo* allora la somma richiesta si ottiene aggiungendo a `sommasx` il valore di `sommadx`.

Le somme dei numeri primi contenuti nei sottoalberi sinistro e destro possono essere calcolate mediante chiamate ricorsive. Per decidere se un numero è primo, richiamiamo la `FUNCTION primo` che, in base al testo dell'esercizio, possiamo considerare già definita. Il codice completo della `FUNCTION` sviluppata secondo questo schema è:

```
FUNCTION SommaPrimi (a: tipoalbero): integer;

{restituisce la somma dei numeri primi presenti nell'albero puntato dal parametro a}

   VAR
      sommasx, sommadx: integer;
```

```
BEGIN {SommaPrimi}
   IF a = NIL THEN
      SommaPrimi := 0
   ELSE
      BEGIN
         sommasx := SommaPrimi(a^.sx);
         sommadx := SommaPrimi(a^.dx);
         IF primo(a^.info) THEN
            SommaPrimi := sommasx + sommadx + a^.info
         ELSE
            SommaPrimi := sommasx + sommadx
      END
END; {SommaPrimi}
```

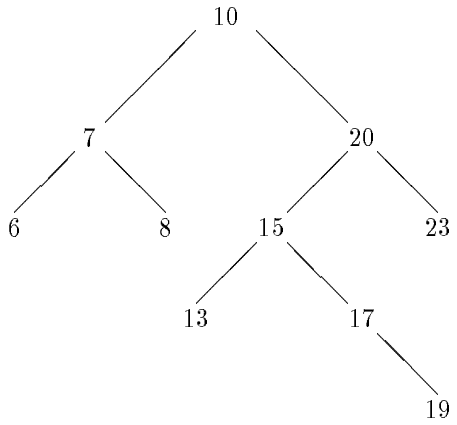
La `FUNCTION` può essere scritta in forma piú compatta come segue:

```
FUNCTION SommaPrimi (a: tipoalbero): integer;

{restituisce la somma dei numeri primi presenti nell'albero puntato dal parametro a}

BEGIN {SommaPrimi}
   IF a = NIL THEN
      SommaPrimi := 0
   ELSE IF primo(a^.info) THEN
      SommaPrimi := SommaPrimi(a^.sx) + SommaPrimi(a^.dx) + a^.info
   ELSE
      SommaPrimi := SommaPrimi(a^.sx) + SommaPrimi(a^.dx)
END; {SommaPrimi}
```

Esercizio 3

Disegnare l'albero di ricerca ottenuto inserendo, uno dopo l'altro, i numeri `10 20 7 6 8 15 17 19 23 13` in un albero inizialmente vuoto. Scrivere gli output prodotti visitando tale albero nei tre ordini anticipato, simmetrico e posticipato.

Gli alberi di ricerca vengono costruiti secondo la seguente regola: per ogni nodo, i valori minori del valore contenuto nel nodo vengono inseriti a sinistra, gli altri a destra. Secondo questa regola, inserendo i valori indicati a partire da un albero vuoto, si ottiene l'albero rappresentato nella seguente figura.



Gli output prodotti dalle tre visite all'albero sono:

- *Visita in ordine anticipato:* `10 7 6 8 20 15 13 17 19 23`.
- *Visita in ordine simmetrico:* `6 7 8 10 13 15 17 19 20 23`.
- *Visita in ordine posticipato:* `6 8 7 13 19 17 15 23 20 10`.

Esercizio 4

Scrivere l'output prodotto da ciascuno dei seguenti programmi.

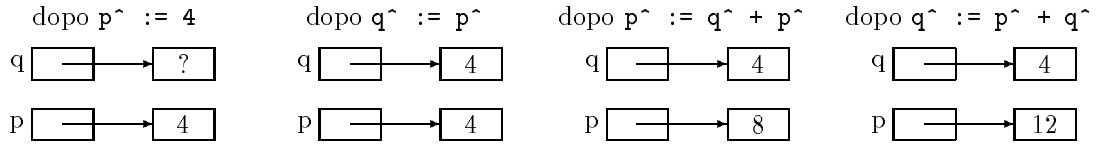
```
PROGRAM p1 (output);

   VAR
      p, q: ^integer;

BEGIN {p1}
   new(p);
   new(q);
   p^ := 4;
   q^ := p^;
   p^ := q^ + p^;
   q^ := p^ + q^;
   writeln(q^)
END. {p1}
```

```
PROGRAM p2 (output);

   VAR
      p, q: ^integer;

BEGIN {p2}
   new(p);
   new(q);
   p^ := 4;
   q := p;
   p^ := q^ + p^;
   q^ := p^ + q^;
   writeln(q^)
END. {p2}
```

```
PROGRAM p3 (output);

   VAR
      p, q: ^integer;

BEGIN {p3}
   new(p);
   new(q);
   p^ := 4;
   q^ := -p^;
   p^ := q^ + p^;
   q^ := p^ + q^;
```

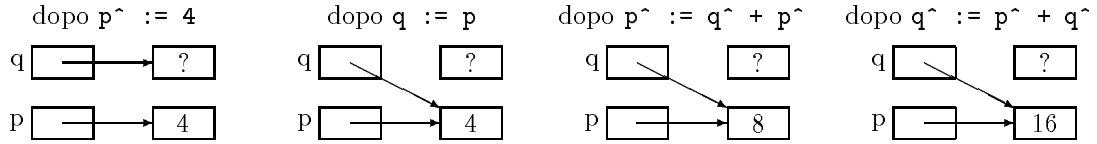
```
   writeln(q^)
END. {p3}
```

La seguente figura rappresenta il contenuto della memoria durante l'esecuzione del programma `p1` nei punti piú significativi:



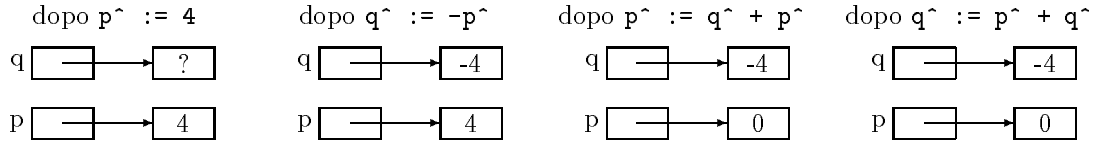
L'output prodotto dal programma è `12`.

Nel caso di `p2`, il contenuto della memoria nei punti piú significativi è rappresentato dalla seguente figura:



Pertanto in output viene stampato `16`.

Consideriamo infine l'esecuzione del programma `p3`:



L'output prodotto dal programma è `-4`.

Esercizio 5

Per ognuna delle seguenti linee di codice individuare delle dichiarazioni di variabile ed eventualemente di tipo, in modo che le istruzioni che vi appaiono risultino corrette dal punto di vista della compatibilità dei tipi. Se ciò non fosse possibile, spiegare il motivo.

- `a^[b^]^ := ord(NOT(b^))`
- `a^.b := a^.c <> ord(a^.b)`
- `a.b^ := a.c <> ord(a.b = NIL)`

A destra del primo assegnamento viene applicato l'operatore `NOT` alla variabile `b^`, che pertanto sarà di tipo `boolean`. Dunque `b` dovrà essere un puntatore a `boolean`. Poichè `ord` si applica a tutti i tipi scalari (tra cui anche `boolean`) e restituisce un `integer`, il risultato dell'espressione scritta a destra dell'assegnamento è di tipo `integer`. Consideriamo ora il lato sinistro dell'assegnamento. Dopo l'identificatore `a` compare il simbolo di puntatore; pertanto `a` è un puntatore. Dopo il nome `a^` dell'oggetto puntato da `a`, compaiono le parentesi quadre che indicano il selettore di un array. Dunque `a` punta a un tipo array. Come indice dell'array viene utilizzato `b^`, che abbiamo detto essere di tipo `boolean`. Subito dopo il selettore dell'array, cioè dopo la parentesi quadra chiusa, vi è di nuovo il simbolo di puntatore. Ciò significa che l'elemento dell'array, cioè `a^[b^]` è un puntatore. Alla variabile puntata da esso viene assegnato il risultato dell'espressione scritta sul lato destro, che abbiamo visto essere di tipo `integer`. Dunque, possiamo concludere che gli elementi dell'array sono puntatori a `integer`. Riassumendo, possiamo fornire le seguenti dichiarazioni:

```
TYPE
   ar = ARRAY [boolean] OF ^integer;
VAR
   b : ^boolean;
   a : ^ar;
```

Nel secondo assegnamento compare l'identificatore `a` seguito dal simbolo di puntatore, seguito dal selettore di record. Pertanto `a` sarà un puntatore a un record. Osserviamo che tale record deve avere due campi di nomi `b` e `c`. Nel lato destro dell'assegnamento, il campo `c` viene confrontato con il risultato della funzione `ord`; pertanto tale campo deve essere di tipo `integer`. Inoltre al campo `b` viene applicata la funzione `ord`; pertanto tale campo deve essere di un tipo scalare. L'espressione scritta a destra dell'assegnamento è un espressione di confronto, che ha dunque un risultato di tipo `boolean`. Tale risultato viene assegnato al campo `b` che deve quindi essere di tipo `boolean`. Possiamo dunque fornire le seguenti dichiarazioni:

```
TYPE
   r = RECORD
          b: boolean;
          c: integer
       END;
VAR
   a : ^r;
```

Nel terzo assegnamento, l'identificatore `a` è seguito dal selettore di record, e dagli identificatori `b` oppure `c`. Pertanto `a` sarà un record con due campi `b` e `c`. Il campo `c` viene confrontato con il risultato della funzione `ord` e deve dunque essere di tipo `integer`. Il campo `b` viene confrontato con la costante `NIL` e deve dunque essere un puntatore. Notiamo inoltre che al valore puntato dal campo `b` viene assegnato il risultato del confronto scritto a destra del simbolo di assegnamento. Quindi `b` deve essere un puntatore a `boolean`. Forniamo dunque seguenti dichiarazioni:

```
TYPE
   r = RECORD
          b: ^boolean;
          c: integer
       END;
VAR
   a : r;
```

Correzione del compitino del Turno serale

Esercizio 1

Si considerino le seguenti definizioni di tipo:

```
TYPE
   tipolista = ^nodolista;
   nodolista = RECORD
                  info: integer;
                  pros: tipolista
               END;
```

Scrivere in Pascal una `PROCEDURE` che riceva, tramite un parametro di tipo `tipolista`, un puntatore a una lista di numeri interi, e modifichi tale lista inserendo al suo inizio un nuovo nodo contenente il massimo tra i valori presenti nella lista data. Ad esempio, se la lista contiene inizialmente i valori

`9 7 16 13`, dopo l'esecuzione della `PROCEDURE` dovrà contenere `16 9 7 16 13`. Se la lista data è vuota, la `PROCEDURE` dovrà lasciarla immutata.

Scriviamo prima di tutto l'intestazione della `PROCEDURE` che, come stabilito dal testo, ha un unico parametro di tipo `tipolista`. Per effettuare l'inserimento all'inizio della lista, la procedura dovrà essere in grado di modificare il puntatore al primo elemento. Pertanto il parametro dovrà essere passato *per riferimento*. L'intestazione sarà dunque:

```
PROCEDURE modifica (VAR l: tipolista);
```

La procedura può essere suddivisa in due fasi: nella prima si determina il valore massimo, nella seconda si effettua l'inserimento all'inizio della lista del nuovo nodo. Nel caso di lista vuota, la procedura non effettua alcuna operazione. Pertanto le due fasi precedenti andranno inserite nel ramo `THEN` di un'istruzione di selezione, nella quale si controlla che la lista non sia vuota.

La prima fase deve porre in una variabile `max` il valore massimo trovato. Inizialmente a `max` viene assegnato il valore contenuto nel primo nodo della lista. Successivamente, in un ciclo, si confrontano gli altri elementi della lista con `max`, al fine di determinare il piú grande:

```
max := l^.info;
p := l^.pros;
WHILE p <> NIL DO
   BEGIN
      IF p^.info > max THEN max := p^.info;
      p := p^.pros
   END
```

A questo punto è sufficiente effettuare l'inserimento del nuovo nodi all'inizio della lista:

```
new(p);
p^.info := max;
p^.pros := l;
l := p
```

La procedura completa è:

```
PROCEDURE modifica (VAR l: tipolista);

   VAR
      p: tipolista;
      max: integer;

BEGIN {modifica}
   IF l <> NIL THEN
      BEGIN

         {scansione della lista per determinare il valore massimo}
         max := l^.info;
         p := l^.pros;
         WHILE p <> NIL DO
            BEGIN
               IF p^.info > max THEN max := p^.info;
               p := p^.pros
            END;

         {creazione e inserimento del nuovo nodo}
```

```
        new(p);
        p^.info := max;
        p^.pros := l;
        l := p
      END
END; {modifica}
```

Esercizio 2

Si considerino le seguenti definizioni di tipo:

```
TYPE
   tipoalbero = ^nodoalbero;
   nodoalbero = RECORD
                  info: integer;     {informazione contenuta nel nodo}
                  sx, dx: tipoalbero {puntatori ai sottoalberi sinistro e destro}
                END;
```

Scrivere in Pascal una `FUNCTION` che riceva, tramite un parametro di tipo `tipoalbero`, il puntatore a un albero di numeri interi, e restituisca come risultato la somma di tutti i quadrati perfetti memorizzati nell'albero.
Per verificare se un numero è un quadrato perfetto, si supponga di disporre di una `FUNCTION quadrato (n: integer): boolean` (di cui non è richiesta la codifica), che restituisca `true` se il parametro utilizzato nella chiamata è un quadrato perfetto, `false` altrimenti.

In base al testo del problema, la `FUNCTION` deve ricevere come parametro un puntatore di tipo `tipoalbero` e restituire un valore di tipo `integer`. Pertanto l'intestazione sarà:

```
FUNCTION SommaQuadrati (a: tipoalbero): integer;
```

Per sviluppare la funzione consideriamo la definizione ricorsiva di albero.

- Nel caso di *albero vuoto* la somma è zero.
- Nel caso di *albero costituito da una radice piú due sottoalberi sinistro e destro*, supponendo di conoscere la somma dei quadrati perfetti contenuti nel sottoalbero sinistro, che indichiamo con `sommasx`, e la somma dei quadrati perfetti contenuti nel sottoalbero destro, che indichiamo con `sommadx`, per calcolare la somma dei quadrati perfetti contenuti nell'intero albero dobbiamo considerare i seguenti due casi:
  - Se *il valore contenuto nella radice è un quadrato perfetto* allora la somma richiesta si ottiene sommando `sommasx`, `sommadx` e il valore contenuto nella radice;
  - se *il valore contenuto nella radice non è un quadrato perfetto* allora la somma richiesta si ottiene aggiungendo a `sommasx` il valore di `sommadx`.

Le somme dei quadrati perfetti contenuti nei sottoalberi sinistro e destro possono essere calcolate mediante chiamate ricorsive. Per decidere se un numero è un quadrato perfetto, richiamiamo la `FUNCTION quadrato` che, in base al testo dell'esercizio, possiamo considerare già definita. Il codice completo della `FUNCTION` sviluppata secondo questo schema è:

```
FUNCTION SommaQuadrati (a: tipoalbero): integer;

{restituisce la somma dei quadrati perfetti presenti nell'albero puntato dal parametro a}

   VAR
      sommasx, sommadx: integer;
```

```
BEGIN {SommaQuadrati}
   IF a = NIL THEN
      SommaQuadrati := 0
   ELSE
      BEGIN
         sommasx := SommaQuadrati(a^.sx);
         sommadx := SommaQuadrati(a^.dx);
         IF quadrato(a^.info) THEN
            SommaQuadrati := sommasx + sommadx + a^.info
         ELSE
            SommaQuadrati := sommasx + sommadx
      END
END; {SommaQuadrati}
```

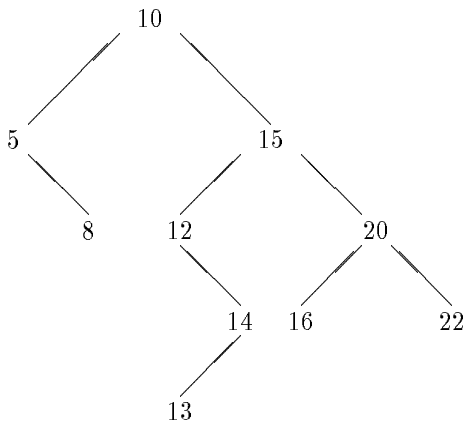
La `FUNCTION` può essere scritta in forma piú compatta come segue:

```
FUNCTION SommaQuadrati (a: tipoalbero): integer;

{restituisce la somma dei quadrati perfetti presenti nell'albero puntato dal parametro a}

BEGIN {SommaQuadrati}
   IF a = NIL THEN
      SommaQuadrati := 0
   ELSE IF primo(a^.info) THEN
      SommaQuadrati := SommaQuadrati(a^.sx) + SommaQuadrati(a^.dx) + a^.info
   ELSE
      SommaQuadrati := SommaQuadrati(a^.sx) + SommaQuadrati(a^.dx)
END; {SommaQuadrati}
```

Esercizio 3

Disegnare l'albero di ricerca ottenuto inserendo, uno dopo l'altro, i numeri `10 15 20 5 8 12 14 13 22 16` in un albero inizialmente vuoto. Scrivere gli output prodotti visitando tale albero nei tre ordini anticipato, simmetrico e posticipato.

L'albero richiesto è rappresentato nella seguente figura.



Gli output prodotti dalle tre visite all'albero sono:

- *Visita in ordine anticipato:* `10 5 8 15 12 14 13 20 16 22`.
- *Visita in ordine simmetrico:* `5 8 10 12 13 14 15 16 20 22`.
- *Visita in ordine posticipato:* `8 5 13 14 12 16 22 20 15 10`.

Esercizio 4

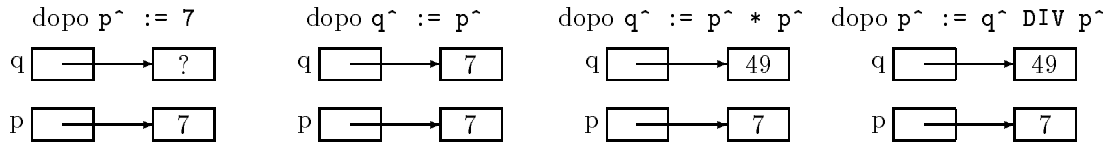
Scrivere l'output prodotto da ciascuno dei seguenti programmi.

```
PROGRAM p1 (output);

   VAR
      p, q: ^integer;

BEGIN {p1}
   new(p);
   new(q);
   p^ := 7;
   q^ := p^;
   q^ := p^ * p^;
   p^ := q^ DIV p^;
   writeln(p^, q^)
END. {p1}
```

```
PROGRAM p2 (output);

   VAR
      p, q: ^integer;

BEGIN {p2}
   new(p);
   new(q);
   p^ := 7;
   q := p;
   q^ := p^ * p^;
   p^ := q^ DIV p^;
   writeln(p^, q^)
END. {p2}
```

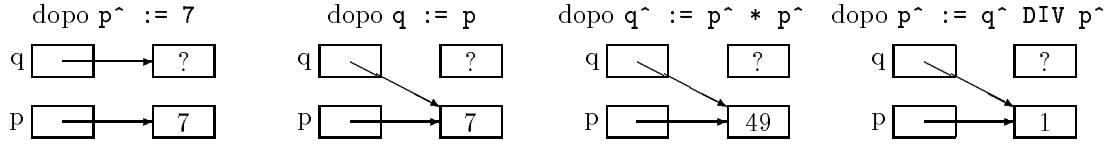
```
PROGRAM p3 (output);

   VAR
      p, q: ^integer;

BEGIN {p3}
   new(p);
   new(q);
   p^ := 7;
   q^ := -p^;
   q^ := p^ * p^;
   p^ := q^ DIV p^;
   writeln(p^, q^)
END. {p3}
```

La seguente figura rappresenta il contenuto della memoria durante l'esecuzione del programma `p1` nei punti piú significativi:



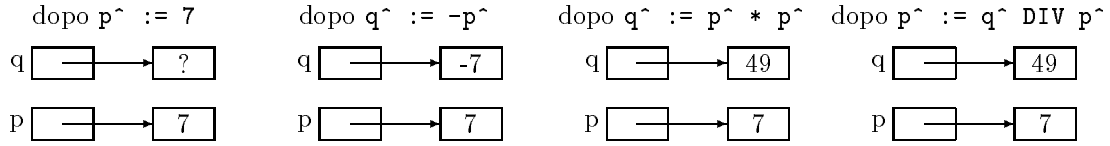
L'output prodotto dal programma è `7 49`.

Nel caso di `p2`, il contenuto della memoria nei punti piú significativi è rappresentato dalla seguente figura:



Pertanto in output viene stampato `1 1`.

Consideriamo infine l'esecuzione del programma `p3`:



L'output prodotto dal programma è `7 49`.

Esercizio 5

Per ognuna delle seguenti linee di codice individuare delle dichiarazioni di variabile ed eventualemente di tipo, in modo che le istruzioni che vi appaiono risultino corrette dal punto di vista della compatibilità dei tipi. Se ciò non fosse possibile, spiegare il motivo.

- `a^[b^] := chr(ord(NOT(b^)))`
- `a^.b := a^.c <> ord(a^.b > 'A')`
- `a.b^ := trunc(a.c) <> ord(a.b = NIL)`

A destra del primo assegnamento viene applicato l'operatore `NOT` alla variabile `b^`, che pertanto sarà di tipo `boolean`. Dunque `b` dovrà essere un puntatore a `boolean`. Al risultato di `NOT(b^)` viene applicata la funzione `ord`, al cui risultato, di tipo `integer`, viene applicata la funzione `chr`. Pertanto, il risultato dell'espressione scritta a destra dell'assegnamento è di tipo `integer`. Consideriamo ora il lato sinistro dell'assegnamento. Dopo l'identificatore `a` compare il simbolo di puntatore; pertanto `a` è un puntatore. Dopo il nome `a^` dell'oggetto puntato da `a`, compaiono le parentesi quadre che indicano il selettore di un array. Dunque `a` punta a un tipo array. Come indice dell'array viene utilizzato `b^`, che abbiamo detto essere di tipo `boolean`. Alla variabile `a^[b^]` viene assegnato il risultato dell'espressione scritto sul lato destro, di tipo `char`. Dunque, possiamo concludere che gli elementi dell'array sono di tipo `char`. Riassumendo, possiamo fornire le seguenti dichiarazioni:

```
TYPE
   ar = ARRAY [boolean] OF char;
VAR
   b : ^boolean;
   a : ^ar;
```

Nel secondo assegnamento compare l'identificatore `a` seguito dal simbolo di puntatore, seguito dal selettore di record. Pertanto `a` sarà un puntatore a un record. Osserviamo che tale record deve avere due campi di nome `b` e `c`. Il lato destro dell'assegnamento è un'espressione di confronto. Dunque il suo risultato, assegnato al campo `b`, è di tipo `boolean`. D'altra parte, nel lato destro compare un confronto tra il campo `b` e un valore di tipo `char`, questo implica che `b` sia di tipo `char`. Dunque non è possibile determinare dichiarazioni di tipo per cui l'assegnamento sia corretto.

Nel terzo assegnamento, l'identificatore `a` è seguito dal selettore di record, e dagli identificatori `b` oppure `c`. Pertanto `a` sarà un record con due campi `b` e `c`. Al campo `c` viene applicata la funzione `trunc`, dunque sarà di tipo `real` o `integer`. Il campo `b` viene confrontato con la costante `NIL` e deve dunque essere un puntatore. Notiamo inoltre che al valore puntato dal campo `b` viene assegnato il risultato del confronto tra `integer` (i risultati di `trunc` e di `ord`) scritto a destra del simbolo di assegnamento. Quindi `b` deve essere un puntatore a `boolean`. Forniamo dunque seguenti dichiarazioni:

```
TYPE
   r = RECORD
          b: ^boolean;
          c: real
       END;
VAR
   a : r;
```