Lezione 14
16–17 novembre 1999

Argomenti trattati
- Tipi strutturati: i record.
- Regole di visibilità del linguaggio Pascal.
- Gestione della symbol table da parte del compilatore.

14.1 I record

Un *record* è una collezione di variabili (semplici o strutturate), ciascuna delle quali è accessibile mediante un identificatore. Le variabili che costituiscono un record vengono dette *campi* o *componenti* e possono essere di tipi differenti.

Ad esempio, una data è costituita da tre informazioni: il giorno, il mese e l'anno. È opportuno organizzare queste informazioni, tra loro correlate, in un'unica struttura, e non trattarle come singole variabili. Ad esempio, possiamo definire:

```
TYPE
   mesi = (gen, feb, mar, apr, mag, giu, lug, ago, sep, ott, nov, dic);
   giorni = 1..31;
   anni = 1900..9999;
   data = RECORD
             giorno: giorni;
             mese: mesi;
             anno: anni
          END;

VAR
   x, y: data;
```

La definizione del tipo `data` si apre con la parola `RECORD` e si chiude con la parola `END`. All'interno vengono definiti gli identificatori dei campi (`giorno`, `mese` e `anno`) con i rispettivi tipi. Le variabili `x` e `y`, di tipo `data`, sono quindi costituite da tre campi, di nomi rispettivamente `giorno`, `mese` e `anno`.

È possibile accedere ai singoli campi di un record mediante l'uso di un *selettore*, che specifichi il nome del campo desiderato. Ad esempio, per accedere al campo `giorno` della variabile `x`, si scrive `x.giorno`. Consideriamo i seguenti assegnamenti:

```
x.giorno := 10;
x.mese := nov;
x.anno := 1999;
y := x;
y.giorno := y.giorno + 1;
```

Le prime tre istruzioni assegnano valori ai tre campi del record `x`. La quarta istruzione, `y := x`, copia l'intero contenuto del record `x` nel record `y`. Infine, l'ultima istruzione incrementa di `1` il valore del campo `giorno` del record `y`.

Quando vi siano piú istruzioni vicine che accedono ai campi dello stesso record, si può evitare di ripetere ogni volta il nome del record, utilizzando l'istruzione `WITH`. Ad esempio, i primi tre assegnamenti precedenti possono essere riscritti come:

```
WITH x DO
BEGIN
   giorno := 10;
   mese := nov;
   anno := 1999
END {with}
```

I campi dei record possono essere, a loro volta, di tipo strutturato, ad esempio record o array.

14.2 Tipi di dati impaccati e stringhe

Nella rappresentazione dei dati in memoria, spesso alle variabili strutturate viene riservato piú spazio di quello strettamente necessario, al fine di rendere piú veloci le operazioni di accesso ai singoli elementi. È possibile chiedere che un tipo di dati venga rappresentato occupando solo lo spazio necessario. A tale scopo si specifica, nella dichiarazione, la parola riservata `PACKED` (es. `PACKED ARRAY`, `PACKED RECORD`, ecc.). Non ci addentriamo nei dettagli relativi alla rappresentazione, che dipendono dalla singola implementazione. Tuttavia, vogliamo evidenziare alcune caratteristiche dei `PACKED ARRAY` costituiti da elementi di tipo `char`, che permettono di rappresentare le stringhe di caratteri.

Abbiamo già utilizzato nei programmi le stringhe di caratteri, come costanti racchiuse tra apici, per scrivere messaggi in `output`. Vediamo ora una possibile rappresentazione delle stringhe mediante variabili. Supponiamo di definire una lunghezza per le stringhe, ad esempio di 15 caratteri. Possiamo definire una stringa come un `PACKED ARRAY` costituito da 15 caratteri:

```
CONST
   maxlung = 15;

TYPE
   stringa = PACKED ARRAY [1..maxlung] OF char;

VAR
   x, y: stringa;
```

Sulle variabili `x` e `y` è possibile effettuare tutte le operazioni che abbiamo visto per gli array, come ad esempio la selezione di un elemento (`x[10]`) o l'assegnamento dell'intero array `y` a `x` (`x := y`). Sono inoltre possibili le seguenti operazioni:

- Assegnamento di una costante stringa all'array.
  Alle variabili di tipo `stringa` è possibile assegnare il valore di una costante, scritta tra apici, *di lunghezza uguale* a quella di `stringa`. Pertanto è possibile l'assegnamento

```
x := 'ecco la stringa'
```

ma non l'assegnamento

```
x := 'ecco la strega'
```

in quanto, nel secondo caso, la costante scritta tra apici ha una lunghezza differente da quella di `x`.

- Confronto tra due stringhe della stessa lunghezza, mediante gli usuali operatori `<`, `>`, `<=`, `>=`, `<>` e `=`.
  Il confronto avviene in base all'ordinamento lessicografico: la stringa `x` è minore della stringa `y` se il contenuto di `x` precede, in odine "alfabetico", quello di `y`. Ad esempio `'ecco la stringa'` è minore di `'ecco la zattera'`. Infatti, esaminando le due stringhe da sinistra verso destra, osserviamo che i caratteri fino al secondo spazio coincidono, mentre il carattere successivo della prima stringa, cioè la `'s'`, precede alfabeticamente il carattere corrispondente della seconda stringa, che è una `'z'`.

  Si ricordi che le costanti di tipo `char` non sono solo le lettere, ma anche le cifre, i caratteri di punteggiatura, i caratteri speciali, ecc. Una stringa può contenere qualunque di questi caratteri. Per determinare l'esatto ordinamento, occorre riferirsi alla tabella ASCII.

- Scrittura dell'intero contenuto della stringa. L'istruzione

  ```
  writeln(x)
  ```

  scrive in `output` l'intero contenuto della stringa `x` (pertanto 15 caratteri). Al contrario, per la lettura il Pascal Standard non permette l'uso di un'unica istruzione, e dunque è necessario utilizzare un ciclo.

Una grossa limitazione del tipo `stringa` basato sui `PACKED ARRAY`, che abbiamo definito sopra, è il fatto che la lunghezza della stringa è fissata, ed è possibile fare assegnamenti o confronti solo tra stringhe della stessa lunghezza. Ridefiniamo ora il tipo `stringa`, utilizzando un record costituito da due campi: un campo `contenuto` in cui viene memorizzata la stringa, e un campo `lung`, che indica quanti caratteri della stringa siano effettivamente significativi. Ad esempio, se nel campo `contenuto` abbiamo i 15 caratteri `'toponomastica01'` e nel campo `lung` abbiamo il valore 4, la stringa rappresentata è `'topo'`.

```
CONST
   maxlung = 15;

TYPE
   stringa = RECORD
                contenuto: PACKED ARRAY [1..maxlung] OF char;
                lung: 0..maxlung
             END;
```

Per `lung` abbiamo considerato anche il valore `0` in modo che si possa rappresentare la stringa vuota, costituita da zero caratteri.

Esempio: stringhe palindrome

Vogliamo ora scrivere un programma che riceva in ingresso una sequenza di caratteri, e decida se essa è *palindroma*, cioè se letta al contrario coincide con se stessa. Esempi di stringhe palindrome sono `radar`, `anna` e `ailatiditalia`.

Il programma può essere strutturato come segue:

```
lettura della stringa
verifica che la stringa sia palindroma
scrittura del risultato
```

In realtà, poiché il messaggio da scrivere in `output` dipende da una condizione booleana (cioè dal fatto che la stringa sia o non sia palindroma), possiamo riscrivere la struttura del programma come:

```
lettura della stringa
IF la stringa e' palindroma
   THEN scrivi "palindroma"
   ELSE scrivi "non palindroma"
```

La valutazione della condizione `la stringa e' palindroma` può essere demandata ad un'apposita `FUNCTION` che, ricevendo mediante un parametro la stringa, restituisca un valore di tipo `boolean`.

Rappresentiamo la stringa mediante una variabile di nome `parola` del tipo `stringa` definito sopra. La `FUNCTION` che determina se la stringa è palindroma avrà la seguente intestazione:

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

Sviluppiamo prima di tutto la fase di lettura e la fase di scrittura, che verranno inserite direttamente nel programma principale.

La fase di lettura è costituita essenzialmente da un ciclo:

```
WHILE la riga non e' finita DO
   leggi un carattere e ponilo nella prima posizione libera dell'array
   parola.contenuto
```

Piú precisamente, il primo carattere letto andrà posto nella posizione `1`, il secondo nella posizione `2`, e così via. Inoltre, la posizione in cui viene posto l'ultimo carattere letto fornisce la lunghezza della stringa, che va assegnata al campo `lung`.

Utilizziamo una variabile `i` di tipo `integer` per indicare, di volta in volta, la posizione in cui è stato inserito l'ultimo carattere letto. Poiché, inizialmente, non vi sono caratteri letti, tale variabile viene inizializzata a zero.

```
i := 0;
WHILE NOT eoln DO
   BEGIN
      i := i + 1;
      read(parola.contenuto[i])
   END; {while}
readln;
parola.lung := i
```

Nel caso l'utente inserisca un numero di caratteri superiore a `maxlung`, nel precedente ciclo vi sarà un tentativo di accedere a un elemento non esistente dell'array `parola.contenuto`, che provocherà un errore in esecuzione. Decidiamo di troncare le stringhe di lunghezza superiore a `maxlung`: quando la stringa contiene piú di `maxlung` caratteri, consideriamo solo i primi `maxlung`. A tale scopo, modifichiamo la condizione del ciclo, controllando anche che la posizione `maxlung` non sia già stata occupata, cioè che il valore della variabile `i` sia inferiore a `maxlung`:

```
WHILE NOT eoln AND (i < maxlung) DO
   ...
```

Nel caso la stringa in ingresso sia stata troncata, è opportuno dare una segnalazione all'utente. All'uscita dal ciclo quindi effettuiamo un controllo del tipo

```
IF la stringa e' stata troncata THEN
   informa l'utente con un messaggio
```

Dobbiamo ora esprimere una condizione che sia vera, in uscita dal ciclo, quando la stringa è stata troncata. Osserviamo che nel caso in cui la stringa in ingresso venga troncata, si esce dal ciclo senza avere raggiunto la fine della riga, cioè con `eoln` che vale `false`. Pertanto la condizione dell'`IF` è semplicemente `NOT eoln`.

La fase di lettura può dunque essere riscritta come:

```
write('Inserire la stringa da esaminare ');
i := 0;
WHILE NOT eoln AND (i < maxlung) DO
   BEGIN
      i := i + 1;
      read(parola.contenuto[i])
   END; {while}
IF NOT eoln THEN
   BEGIN
      writeln('Attenzione: stringa troppo lunga');
      writeln('Verranno considerati solo i primi ', maxlung : 1, ' caratteri')
   END;
readln;
parola.lung := i
```

Analizziamo ora la fase di verifica e scrittura, che abbiamo schematizzato sopra come:

```
IF la stringa e' palindroma
   THEN scrivi "palindroma"
   ELSE scrivi "non palindroma"
```

La verifica della condizione `la stringa e' palindroma` viene effettuata richiamando la funzione palindroma, sul valore contenuto in `parola`. È inoltre opportuno riportare in `output` in contenuto della stringa esaminata. A tale scopo, utilizziamo un ciclo `FOR` immediatamente prima dell'istruzione di selezione. La fase di verifica e scrittura può dunque essere codificata come:

```
write('La parola ');
FOR i := 1 TO parola.lung DO
   write(parola.contenuto[i]);
IF palindroma(parola) THEN
   writeln(' e'' palindroma')
ELSE
   writeln(' non e'' palindroma')
```

Il codice del completo del programma principale è dunque:

```
BEGIN {palindromi}

   {leggi parola}
   write('Inserire la stringa da esaminare ');
   i := 0;
   WHILE NOT eoln AND (i < maxlung) DO
      BEGIN
         i := i + 1;
         read(parola.contenuto[i])
      END; {while}
```

```
   IF NOT eoln THEN
      BEGIN
         writeln('Attenzione: stringa troppo lunga');
         writeln('Verranno considerati solo i primi ', maxlung : 1, ' caratteri')
      END;
   readln;
   parola.lung := i;

   {calcola e scrivi il risultato}
   write('La parola ');
   FOR i := 1 TO parola.lung DO
      write(parola.contenuto[i]);
   IF palindroma(parola) THEN
      writeln(' e'' palindroma')
   ELSE
      writeln(' non e'' palindroma')

END. {palindromi}
```

Sviluppiamo ora la `FUNCTION palindroma` che, come abbiamo detto, ha la seguente intestazione:

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

Per la verifica, occorre confrontare il primo e l'ultimo carattere, il secondo e il penultimo carattere, e così via. Se in tutte le coppie i caratteri coincidono, la parola rappresentata nel parametro `s` è palindroma, altrimenti no.

Possiamo dunque utilizzare il seguente ciclo

```
FOR i := 1 TO s.lung DO
   IF il carattere in posizione i e' diverso da quello
      in posizione s.lung - i + 1 THEN la parola non e' palindroma
```

Utilizziamo una variabile `p` di tipo `boolean` per ricordare se la parola è palindroma: inizialmente, ponendoci nell'ipotesi che la parola sia palindroma, assegnamo a `p` valore `true`; quando due dei caratteri confrontati risultano differenti, l'ipotesi iniziale risulta falsificata, pertanto si assegna alla variabile `p` il valore `false`. Per non ripetere ogni volta il nome del record, utilizziamo anche l'istruzione `WITH`. La fase di controllo può dunque essere scritta come:

```
p := true;
WITH s DO
   FOR i := 1 TO lung DO
      IF contenuto[i] <> contenuto[lung - i + 1] THEN
         p := false
```

Terminato il controllo, si può restituire il valore contenuto in `p` all'ambiente chiamante, scrivendo

```
palindroma := p
```

Il codice completo della `FUNCTION` ottenuta è:

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

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

```
   VAR
      i: 1..maxlung;
      p: boolean;

BEGIN {palindroma}
   p := true;
   WITH s DO
      FOR i := 1 TO lung DO
         IF contenuto[i] <> contenuto[lung - i + 1] THEN
            p := false;
   palindroma := p
END; {palindroma}
```

Nella soluzione appena sviluppata, ogni confronto viene effettuato due volte (ad esempio all'inizio del ciclo si confronta il primo carattere con l'ultimo, mentre alla fine dei ciclo si confronta l'ultimo carattere con il primo). Inoltre, quando si trovano due caratteri differenti, vengono effettuati anche i confronti successivi, sebbene siano del tutto inutili. Possiamo riscrivere la `FUNCTION` servendoci di un ciclo `WHILE`. Per comodità utilizziamo due variabili, `sx` e `dx` di tipo `integer`, che di volta in volta indicano le posizioni della stringa da esaminare. All'inizio i due indici puntano alle due estremità della stringa. Ad ogni passo si confrontano i caratteri puntati dai due indici, e si spostano gli indici, terminando quando si sia esaminata tutta la stringa o si siano trovati caratteri differenti:

```
sx := 1;
dx := lunghezza della stringa;
WHILE non hai esaminato tutta la stringa AND il carattere in posizione
      sx coincide con quello in posizione dx DO
   sposta gli indici
```

Osserviamo che quando i due indici `sx` e `dx` si incrociano, tutta la stringa è stata esaminata. Pertanto il ciclo può essere riscritto come:

```
sx := 1;
dx := s.lung;
WHILE (sx < dx) AND (s.contenuto[sx] = s.contenuto[dx]) DO
   BEGIN
      sx := sx + 1;
      dx := dx - 1
   END; {while}
```

In uscita dal ciclo occorre stabilire se la stringa sia palindroma o no. A tale scopo è necessario capire quale delle due condizioni sia stata falsificata, provocando l'uscita dal ciclo. Possiamo notare che quando la stringa è palindroma vengono esaminati tutti i caratteri, e dunque i due indici si incrociano. Viceversa, quando la stringa non è palindroma all'uscita dal ciclo il valore di `sx` risulta ancora minore del valore di `dx`. Pertanto, in uscita dal ciclo è sufficiente restituire il risultato scrivendo:

```
palindroma := sx >= dx
```

Ecco il listato completo del programma, con la seconda versione della `FUNCTION palindroma`:

```
PROGRAM palindromi (input, output);

{decide se la stringa in input e' palindroma o no}

   CONST
```

```
      maxlung = 15;

   TYPE
      stringa = RECORD
            contenuto: PACKED ARRAY[1..maxlung] OF char;
            lung: 0..maxlung
         END;

   VAR
      parola: stringa;
      i: integer;


   FUNCTION palindroma (s: stringa): boolean;

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

      VAR
         sx, dx: integer;

   BEGIN {palindroma}
      sx := 1;
      dx := s.lung;
      WHILE (sx < dx) AND (s.contenuto[sx] = s.contenuto[dx]) DO
         BEGIN
            sx := sx + 1;
            dx := dx - 1
         END; {while}
      palindroma := sx >= dx
   END; {palindroma}


BEGIN {palindromi}

   {leggi parola}
   write('Inserire la stringa da esaminare ');
   i := 0;
   WHILE NOT eoln AND (i < maxlung) DO
      BEGIN
         i := i + 1;
         read(parola.contenuto[i])
      END; {while}
   IF NOT eoln THEN
      BEGIN
         writeln('Attenzione: stringa troppo lunga');
         writeln('Verranno considerati solo i primi ', maxlung : 1, ' caratteri')
      END;
   readln;
   parola.lung := i;

   {calcola e scrivi il risultato}
   write('La parola ');
```

```
   FOR i := 1 TO parola.lung DO
      write(parola.contenuto[i]);
   IF palindroma(parola) THEN
      writeln(' e'' palindroma')
   ELSE
      writeln(' non e'' palindroma')

END. {palindromi}
```

14.3 Struttura di un blocco Pascal

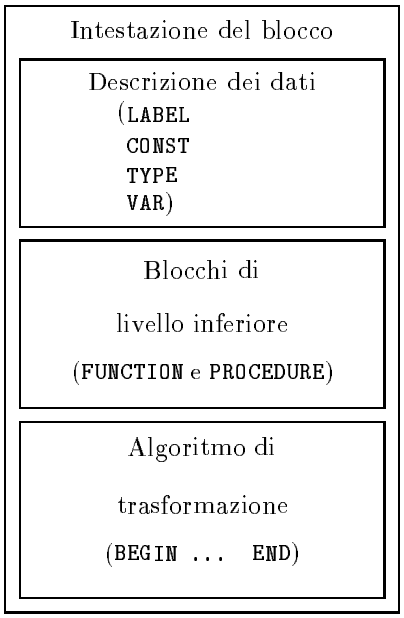
In Pascal, il programma principale e i sottoprogrammi hanno la medesima struttura, che chiamiamo *Blocco Pascal*. Il blocco si apre con una *intestazione*, che specifica la natura del blocco (`PROGRAM`, `PROCEDURE` o `FUNCTION`), il nome del blocco e i canali di comunicazione con l'esterno (che dipendono dalla natura del blocco considerato). Il blocco è poi costituito dalle seguenti tre parti:

Descrizione dei dati Contiene le definizioni e dichiarazioni di etichette (`LABEL`, che non abbiamo ancora esaminato), costanti (`CONST`), tipi (`TYPE`) e variabili (`VAR`) proprie del blocco. Queste dichiarazioni sono visibili all'interno del blocco e in tutti i blocchi di livello inferiore (salvo, come vedremo piú avanti, in caso di "adombramento"), mentre non sono visibili all'esterno del blocco.

Blocchi di livello inferiore Contiene altri blocchi, definiti all'interno del blocco.

Algoritmo di trasformazione Contiene il codice del blocco, racchiuso tra le parole riservate `BEGIN` ed `END`.

La struttura di un blocco Pascal è schematizzata nella seguente figura.



Da quanto detto, in ogni blocco è possibile definire risorse (etichette, costanti, tipi, variabili, sottoprogrammi). Le risorse definite all'interno di un blocco sono utilizzate per l'implementazione del blocco e non possono essere utilizzate all'esterno di esso.

Introdurremo ora le *regole di visibilità* (o *scope rules*) del linguaggio Pascal, che specificano esattamente la visibilità di ciascuna risorsa definita all'interno di un programma.

14.4 Regole di visibilità del linguaggio Pascal

Una risorsa è accessibile o visibile (e dunque può essere utilizzata):

- nella porzione di programma che segue la definizione o dichiarazione,
- limitatamente al blocco che contiene la definizione o dichiarazione (compresi i sottoblocchi in esso descritti),
- ad eccezione delle porzioni di programma in cui l'identificatore della risorsa sia ridefinito.

In altre parole, le regole precedenti stabiliscono che lo "scope" di una risorsa, cioè la porzione di programma nella quale essa è accessibile, è il blocco che contiene la definizione o dichiarazione, a partire dal punto in cui si trova la definizione o dichiarazione stessa. Ogni risorsa è privata per il blocco che la definisce: pertanto i blocchi esterni non possono utilizzarla, mentre i blocchi interni, successivi alla dichiarazione della risorsa, la ereditano, a meno che contengano a loro volta la dichiarazione o definizione di una risorsa con il medesimo identificatore. In questo caso la risorsa risulta *adombrata*.

Gli identificatori predefiniti, come ad esempio `integer` e `write`, sono risorse dell'ambiente esterno, cioè dell'ambiente che contiene l'intero programma.

Per chiarire meglio il significato delle regole di visibilità presentiamo ora alcuni esempi.

Esempio 1

```
PROGRAM prog (output);
   VAR
      i, j: integer;


   PROCEDURE a;
      VAR
         k: integer;

   BEGIN {a}
      ...
   END; {a}


   PROCEDURE b;
      VAR
         j, h: char;

   BEGIN {b}
      ...
   END; {b}


BEGIN {prog}
   ...
END. {prog}
```

Evidenziamo lo "scope" di ciascuna risorsa definita nel programma.

- Risorse dichiarate in `PROGRAM prog`

  - `VAR i: integer`
    Accessibile in tutto `prog` (e dunque anche nelle procedure `a` e `b`).
  - `VAR j: integer`
    Accessibile in tutto `prog`, compresa la procedura `a`, ma non nella procedura `b`, dove è adombrata.
  - `PROCEDURE a`
    Accessibile in `a`, `b` e nel codice del programma principale.
  - `PROCEDURE b`
    Accessibile in `b` e nel codice del programma principale (non accessibile in `a` in quanto dichiarata dopo).

- Risorse dichiarate in `PROCEDURE a`
  - `VAR k: integer`
    Accessibile in `a`.

- Risorse dichiarate in `PROCEDURE b`
  - `VAR j, h: char`
    Accessibili in `b`.

Esempio 2

```
PROGRAM alfa (output);
   VAR
      i, j: integer;

   PROCEDURE beta;
      VAR
         i, k: integer;

      PROCEDURE gamma;
         VAR
            i, h: integer;

      BEGIN {gamma}
         h := 1;
         i := 20;
         writeln('gamma', i)
      END; {gamma}

   BEGIN {beta}
      k := 1;
      i := 10;
      gamma;
      writeln('beta', i)
   END; {beta}

BEGIN {alfa}
   j := 1;
   i := 5;
   beta;
   writeln('alfa', i)
```

```
END. {alfa}
```

Indichiamo, per ogni risorsa, il suo "scope", cioè la porzione di programma in cui essa è accessibile.

- Risorse dichiarate nel blocco `PROGRAM alfa`
  - `VAR i: integer`
    Accessibile solo nella porzione di codice di `alfa`; infatti nel sottoblocco `beta` l'identificatore `i` è ridefinito.
  - `VAR j: integer`
    Accessibile nel programma principale e in tutti i sottoprogrammi.
  - `PROCEDURE beta`
    Accessibile nel programma principale e in tutti i sottoprogrammi.
- Risorse dichiarate nel blocco `PROCEDURE beta`
  - `VAR i: integer`
    Accessibile solo nella porzione di codice di `beta`; infatti nel sottoblocco `gamma`, l'identificatore `i` è ridefinito.
  - `VAR k: integer`
    Accessibile nei sottoprogrammi `beta` e `gamma`.
  - `PROCEDURE gamma`
    Accessibile nei sottoprogrammi `beta` e `gamma`.
- Risorse dichiarate nel blocco `PROCEDURE gamma`
  - `VAR i: integer`
    Accessibile nel sottoprogramma `gamma`.
  - `VAR h: integer`
    Accessibile nel sottoprogramma `gamma`.

Il fatto che la procedura `gamma` sia accessibile nei sottoprogrammi `beta` e `gamma`, significa che entrambi i sottoprogrammi possono richiamare la procedura `gamma`. La possibilità che ogni procedura Pascal ha di richiamare se stessa, viene detta *ricorsione* e verrà approfondita in seguito.

Sottolineamo il fatto che le tre variabili di nome `i` utilizzate nei vari blocchi del programma sono tra loro distinte. Ogni istruzione di assegnamento si riferisce alla variabile accessibile nelle porzione di codice in cui l'istruzione si trova e non modifica il valore delle altre variabili, anche se hanno lo stesso nome. Pertanto l'output prodotto dal programma è:

```
gamma      20
beta       10
alfa        5
```

Evidenziamo ora, per ogni blocco di codice, le risorse visibili. Prima di tutto osserviamo che nell'ambiente esterno, cioè nell'ambiente che contiene il programma, sono definite le risorse corrispondenti agli identificatori standard (ad esempio la costante `maxint`, i tipi `integer`, `real`, ecc., i file `input` e `output`, le procedure `read`, `write`, ecc., le funzioni `ord`, `succ`, ecc.). Queste risorse sono accessibili ovunque, a meno che i relativi identificatori vengano ridefiniti (ad esempio, in un blocco si potrebbe dichiarare una variabile di nome `write`, adombrando la procedura standard). Tra le risorse dell'ambiente esterno c'è anche l'identificatore `alfa` del programma.

Esaminiamo la struttura del blocco `PROGRAM alfa`. Le risorse dichiarate al suo interno sono le variabili `i` e `j`, di tipo `integer`, e il blocco `PROCEDURE beta`. Quindi, queste risorse sono accessibili nella porzione di codice del blocco `alfa` (cioè nella porzione di codice compresa tra le

linee `BEGIN {alfa}` ed `END. {alfa}`), insieme alle risorse ereditate dall'ambiente esterno, cioè gli identificatori standard.

Esaminiamo ora il blocco `PROCEDURE beta`. Al suo interno sono definite due variabili `i` e `k` di tipo `integer` e il blocco `PROCEDURE gamma`. Pertanto il codice del blocco `PROCEDURE beta` può accedere alle variabili `i` e `k` e al blocco `PROCEDURE gamma`, dichiarati nel blocco stesso. Inoltre, il blocco può accedere alle risorse del blocco esterno ad esso, cioè `PROGRAM alfa`. Tali risorse sono la variabile `j`, la `PROCEDURE beta` stessa e gli identificatori standard. D'altra parte, la variabile `i` del `PROGRAM alfa` non è accessibile nel blocco `PROCEDURE beta` in quanto adombrata; infatti, l'identificatore `i` è stato utilizzato per definire una variabile all'interno del blocco `PROCEDURE beta` stesso.

Consideriamo infine il blocco `PROCEDURE gamma`. In esso sono accessibili le due variabili locali `i` e `j` di tipo `integer`, oltre a tutte le risorse accessibili nel blocco `PROCEDURE beta`, eccettuata la variabile `i`, dichiarata in `PROCEDURE beta`, che risulta a sua volta adombrata.

Riepilogando, ed escludendo per brevità di menzionare le risorse corrispondenti agli identificatori standard, riportiamo l'elenco delle risorse accessibili in ciascun blocco.

- `PROGRAM alfa`
  - Risorse dichiarate nel blocco stesso:
    `VAR i, j: integer`
    `PROCEDURE beta`
- `PROCEDURE beta`
  - Risorse dichiarate nel blocco stesso:
    `VAR i, k: integer`
    `PROCEDURE gamma`
  - Risorse ereditate dal blocco esterno:
    `VAR j: integer`
    `PROCEDURE beta` (tutte dichiarate in `PROGRAM alfa`).
- `PROCEDURE gamma`
  - Risorse dichiarate nel blocco stesso:
    `VAR i, h: integer`
  - Risorse ereditate dal blocco esterno:
    `VAR k: integer` (dichiarata in `PROCEDURE beta`)
    `PROCEDURE gamma` (dichiarata in `PROCEDURE beta`)
    `VAR j: integer` (dichiarata in `PROGRAM alfa`)
    `PROCEDURE beta` (dichiarata in `PROGRAM alfa`).

Esempio 3

```
PROGRAM p;

   VAR
      x, y, z: char;

   PROCEDURE a;

      VAR
         y: integer;

      PROCEDURE b;
```

```
      VAR
         z, w: integer;

   BEGIN {b}
      ...
   END; {b}

BEGIN {a}
   ...
END; {a}


PROCEDURE c;

   VAR
      z, t: integer;

BEGIN {c}
   ...
END; {c}

BEGIN {p}
   ...
END. {p}
```

Evidenziamo lo "scope" di ciascuna risorsa. Questo può essere individuato applicando direttamente le regole di visibilità: la risorsa è accessibile all'interno del blocco in cui viene dichiarata, a partire dal punto della dichiarazione e in tutti i blocchi interni, a meno che il suo identificatore sia ridefinito.

- Risorse dichiarate nel blocco `PROGRAM p`
  - `VAR x: char`
    Accessibile ovunque.
  - `VAR y: char`
    Accessibile nel blocco `PROCEDURE c` e nel programma principale; non è accessibile nel blocco `PROCEDURE a` in quanto l'identificatore `y` è ridefinito.
  - `VAR z: char`
    Accessibile nel programma principale, nel blocco `a`, ma non all'interno del blocco `b` in cui l'identificatore `z` è ridefinitito, e non nel blocco `c`.
  - `PROCEDURE a`
    Accessibile ovunque.
  - `PROCEDURE c`
    Accessibile nel programma principale e nel blocco `c` stesso (non nel blocco `a` in quanto è dichiarata dopo).
- Risorse dichiarate nel blocco `PROCEDURE a`
  - `VAR y: integer`
    Accessibile ovunque all'interno del blocco `a`, cioè sia nel blocco `a` che nel blocco `b`.
  - `PROCEDURE b`
    Accessibile ovunque all'interno del blocco `a`, cioè sia nel blocco `a` che nel blocco `b`.
- Risorse dichiarate nel blocco `PROCEDURE b`

  - `VAR z: integer`
    Accessibile nel blocco `b`.
  - `VAR w: integer`
    Accessibile nel blocco `b`.

- Risorse dichiarate nel blocco `PROCEDURE c`
  - `VAR z: integer`
    Accessibile nel blocco `c`.
  - `VAR t: integer`
    Accessibile nel blocco `c`.

Evidenziamo ora, per ogni blocco, quali sono le risorse accessibili. Per individuarle si può procedere come segue: partendo dal blocco considerato, che può accedere a tutte le proprie risorse, si procede all'indietro nel testo del programma, considerando le risorse definite in blocchi esterni: sono visibili tutte le risorse il cui identificatore viene incontrato per la prima volta.

Ad esempio, per individuare le risorse accessibili nel codice del blocco `PROCEDURE b`, si procede come segue: prima di tutto si considerano le risorse dichiarate in `b`, cioè le due variabili `z` e `w`; si esaminano poi le risorse definite nel blocco che contiene `b`, cioè il blocco `PROCEDURE a`; tali risorse sono la variabile `y` e la `PROCEDURE b` stessa; si passa dunque al blocco che contiene `a` in cui è definita `a` stessa e, prima di `a`, le variabile `x`, `y` e `z`. Poiché gli identificatori `y` e `z` sono già stati incontrati nelle dichiarazioni di altre risorse, le variabili `y` e `z` di `p` risultano adombrate.

In questo modo possiamo individuare le risorse accessibili da ogni blocco (come al solito omettiamo le risorse corrispondenti agli identificatori predefiniti):

- `PROGRAM p`
  - Risorse dichiarate nel blocco stesso:
    `VAR x, y, z: char`
    `PROCEDURE a`
    `PROCEDURE c`

- `PROCEDURE a`
  - Risorse dichiarate nel blocco stesso:
    `VAR y: integer`
    `PROCEDURE b`
  - Risorse ereditate dal blocco esterno:
    `VAR x, z: char`
    `PROCEDURE a`
    La variabile `y` del programma principale non viene ereditata, in quanto all'interno di `a` l'identificatore `y` viene ridefinito. La risorsa `PROCEDURE c` del blocco esterno *non* viene ereditata, in quanto la sua definizione è successiva a quella della `PROCEDURE a`. In altre parole, al momento della definizione della procedura `a`, il programma principale non possiede ancora la risorsa `c` e quindi non può renderla disponibile ad `a`.

- `PROCEDURE b`
  - Risorse dichiarate nel blocco stesso:
    `VAR z, w: integer`
  - Risorse ereditate dal blocco esterno:
    `VAR y: integer` (dichiarata in `PROCEDURE a`)
    `PROCEDURE b` (dichiarata in `PROCEDURE a`)
    `VAR x: char` (dichiarata nel programma principale)
    `PROCEDURE a` (dichiarata nel programma principale)

- `PROCEDURE c`
  - Risorse dichiarate nel blocco stesso:
    `VAR z, t: integer`
  - Risorse ereditate dal blocco esterno (tutte dichiarate nel programma principale):
    `VAR x, y: char`
    `PROCEDURE a`
    `PROCEDURE c`
    Si osservi che tutte le risorse interne alla `PROCEDURE a`, tra cui ad esempio la `PROCEDURE b`, sono visibili solo all'interno del blocco `a` stesso, e quindi non nel blocco `c`.

Il procedimento che abbiamo utilizzato riflette ciò che fa il compilatore per individuare, di volta in volta, le risorse accessibili. Notiamo, prima di tutto, che la visibilità degli identificatori dipende dalla struttura statica del programma, cioè da come è organizzato il suo testo (negli esercizi è presentato un esempio in cui si mostra come lo stesso codice, organizzato in due maniere differenti, possa produrre risultati differenti). Il compilatore Pascal esamina il programma in un'unica passata. Gli identificatori definiti nel programma vengono memorizzati in una tabella, detta *symbol table*, gestita dinamicamente. Quando si esamina un nuovo modulo, si costruisce una nuova porzione della tabella (destinata a contenere gli identificatori definiti all'interno del modulo). Quando termina l'esame del modulo, tale porzione di tabella viene distrutta, in quanto gli identificatori ivi definiti non sono piú accessibili. Quando il compilatore, esaminando il codice di un programma o di un sottoprogramma, incontra un identificatore, lo ricerca nella symbol table, a partire dal fondo. In questo modo gli identificatori piú interni adombrano gli omonimi piú esterni.

Descriviamo, a titolo d'esempio, il contenuto della symbol table durante la compilazione del programma `p` considerato sopra. Omettiamo di rappresentare la parte esterna della tabella che contiene gli identificatori predefiniti e il nome del programma principale.

Dopo la lettura della riga in cui sono dichiarate le variabili `x`, `y` e `z`, la symbol table conterrà:

risorse blocco `PROGRAM p`		
identificatore	risorsa	attributi
`x`	`var`	`char`
`y`	`var`	`char`
`z`	`var`	`char`

Dopo la lettura dell'intestazione del blocco `PROCEDURE a`, il contenuto della tabella diventa:

risorse blocco `PROGRAM p`		
identificatore	risorsa	attributi
`x`	`var`	`char`
`y`	`var`	`char`
`z`	`var`	`char`
`a`	`procedure`	`...`
risorse blocco `PROCEDURE a`		
identificatore	risorsa	attributi

Si noti che è stata creata una zona della tabella destinata a contenere le risorse di `a`.

Proseguendo l'analisi del testo, il compilatore aggiunge alla tabella le informazioni relative alle risorse dichiarate in `a` e poi in `b`. In particolare, dopo la lettura delle dichiarazioni di variabili in `b` e immediatamente prima della lettura della riga `BEGIN {b}`, il contenuto della tabella sarà:

risorse blocco `PROGRAM p`		
identificatore	risorsa	attributi
`x`	`var`	`char`
`y`	`var`	`char`
`z`	`var`	`char`
`a`	`procedure`	`...`

risorse blocco `PROCEDURE a`		
identificatore	risorsa	attributi
`y`	`var`	`integer`
`b`	`procedure`	`...`

risorse blocco `PROCEDURE b`		
identificatore	risorsa	attributi
`z`	`var`	`integer`
`w`	`var`	`integer`

Gli identificatori visibili in questo punto del programma possono essere determinati leggendo nella tabella, a partire dal fondo, tutti i nomi, con eccezione di quelli già incontrati. In altre parole, il compilatore, quando incontra un identificatore, non fa altro che effettuare una ricerca del nome, partendo dal fondo della tabella. Se ad esempio il codice di `b` contiene l'istruzione `y := 20`, il compilatore associerà l'identificatore `y` alla variabile omonima definita della `PROCEDURE a`, che adombra la variabile con il medesimo nome definita in `p`.

La riga `END; {b}` conclude il blocco di programma `b`. Pertanto, leggendo tale linea, il compilatore elimina dalla symbol table tutte le risorse dichiarate in `b`. La tabella diventa quindi:

risorse blocco `PROGRAM p`		
identificatore	risorsa	attributi
`x`	`var`	`char`
`y`	`var`	`char`
`z`	`var`	`char`
`a`	`procedure`	`...`

risorse blocco `PROCEDURE a`		
identificatore	risorsa	attributi
`y`	`var`	`integer`
`b`	`procedure`	`...`

Nella parte di programma compresa tra `BEGIN {a}` ed `END; {a}`, non essendo presenti dichiarazioni, il contenuto della symbol table resta invariato. Alla fine della lettura del codice di `a` anche la parte di tabella relativa alle risorse dichiarate in `a` viene distutta. Pertanto dopo la lettura dell'intestazione di `c`, il contenuto della tabella è:

risorse blocco `PROGRAM p`		
identificatore	risorsa	attributi
`x`	`var`	`char`
`y`	`var`	`char`
`z`	`var`	`char`
`a`	`procedure`	`...`
`c`	`procedure`	`...`

risorse blocco `PROCEDURE c`		
identificatore	risorsa	attributi

Dopo le dichiarazioni di variabile di `c`, il contenuto della tabella è:

risorse blocco `PROGRAM p`		
identificatore	risorsa	attributi
`x`	`var`	`char`
`y`	`var`	`char`
`z`	`var`	`char`
`a`	`procedure`	`...`
`c`	`procedure`	`...`

risorse blocco `PROCEDURE c`		
identificatore	risorsa	attributi
`z`	`var`	`integer`
`t`	`var`	`integer`

Dalla tabella possiamo notare che all'interno di `c` è possibile chiamare le procedure `a` e `c`, utilizzare le variabili locali `t` e `z`, e le variabili `x` e `y` del programma principale.

Infine, dopo la lettura della linea `END; {c}`, il compilatore elimina dalla symbol table anche le risorse dichiarate in `c`. Restano pertanto le sole risorse utilizzabili dal programma principale:

risorse blocco `PROGRAM p`		
identificatore	risorsa	attributi
`x`	`var`	`char`
`y`	`var`	`char`
`z`	`var`	`char`
`a`	`procedure`	`...`
`c`	`procedure`	`...`

Esercizi

1. Scrivere le definizioni utili per introdurre un tipo record che descriva i dati anagrafici di una persona (nome, cognome, data di nascita, ecc.). È utile fare uso di record di record, ad esempio la data di nascita sarà un campo di tipo `data`, che è a sua volta un record.

2. Si consideri il tipo `stringa` presentato nella lezione. Si codifichino i seguenti sottoprogrammi, di cui si fornisce l'intestazione e la specifica della funzionalità richiesta.

   - `PROCEDURE concatena (VAR s1, s2, s3: stringa);`
     Restituisce nel parametro `s3` la stringa ottenuta concatenando la stringa rappresentata in `s1` con la stringa rappresentata in `s2`. Nel caso la stringa risultante sia piú lunga di `maxlen` caratteri, essa viene troncata ignorando i caratteri in piú.
     Esempi: se `s1` e `s2` rappresentano rispettivamente le stringhe `'topo'` e `'gatto'`, `s3` dovrà rappresentare, al termine dell'esecuzione, la stringa `'topogatto'`; se `s1` e `s2` rappresentano rispettivamente le stringhe `'topolino'` e `'gattaccio'`, `s3` dovrà rappresentare, al termine dell'esecuzione, la stringa `'topolinogattacc'`.
   - `PROCEDURE appendi (VAR s1, s2: stringa);`
     Restituisce nel parametro `s1` la stringa ottenuta concatenando la stringa rappresentata inizialmente in `s1` con la stringa rappresentata in `s2`.
   - `PROCEDURE estrai (VAR s1: stringa; VAR i,j: integer; VAR s2: stringa);`
     Restituisce nel parametro `s2` la stringa composta dai caratteri di `s1` compresi tra la posizione `i` e la posizione `j`, se presenti in `s1`.
     Esempi: se `s1` rappresenta la stringa `'topolino'`, e i parametri `i` e `j` valgono rispettivamente `3` e `6`, in `s2` dovrà rappresentare la stringa `'poli'`; se invece `i` e `j` valgono rispettivamente `5` e `13`, in `s2` dovrà rappresentare la stringa `'lino'`.

- `FUNCTION cerca (VAR s: stringa; c: char): integer;`
  Restituisce la posizione della prima occorrenza del carattere `c` nella stringa `s`, o `0` nel caso il carattere non sia presente.
  Esempi: se `s` rappresenta la stringa `'topolino'` e `c` contiene il carattere `'o'`, dovrà essere restituito il valore `2`.
- `FUNCTION minore (VAR s1, s2: stringa): boolean;`
  Restituisce `true` quando la stringa rappresentata in `s1` è lessicograficamente minore della stringa rappresentata in `s2`.

Scrivere poi dei brevi programmi per testare il funzionamento dei sottoprogrammi costruiti.

3. Nelle intestazioni dei sottoprogrammi dell'esercizio precedente, i parametri di tipo `stringa` sono sempre stati passati per indirizzo. Individuare quali potrebbero essere passati per valore e spiegare perché sia preferibile, per parametri di questo tipo, il passaggio per indirizzo.

4. Per ognuno dei seguenti frammenti di codice individuare delle dichiarazioni di variabile ed eventualmente 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.

   - `m.d := m.r > (m.v + q)`
   - `x.s := chr(x.t[x.s])`
   - `p := a[p] > ord(p)`
   - ```
     b := a <= c;
     c := (d + 1) - c
     ```
   - ```
     b := a <= c;
     c := [d + 1] - c
     ```
   - ```
     b := a <= c;
     c := {d + 1} - c
     ```
   - ```
     z := 1;
     y := 2;
     x := z - y
     ```
   - ```
     z := [1];
     y := [2];
     x := z - y
     ```
   - ```
     z := [1];
     y := 2;
     x := z - [y]
     ```
   - ```
     x := x + y;
     y := y * z;
     z := []
     ```
   - `a := a IN p`

5. Si considerino i seguenti programmi. Per ciascuno di essi indicare se ci saranno errori in fase di compilazione, se ci saranno errori in fase d'esecuzione (in tal caso indicare dei valori di ingresso che causino errore), se il programma può entrare in un ciclo infinito (in tal caso indicare dei valori d'ingresso per cui ciò avviene). Non si tenga conto degli errori d'esecuzione che si potrebbero avere durante la lettura dei dati.

   - ```
     PROGRAM p1 (input, output);

         CONST
     ```

```
      max = 10;

   TYPE
      ar = ARRAY[1..max] OF char;

   VAR
      v: ar;
      n, i: integer;

BEGIN
   REPEAT
      readln(n)
   UNTIL n <= max;
   FOR i := 1 TO n DO
      BEGIN
         v[i] := chr(ord('A') + i);
         writeln(v[i])
      END
END.
```

- ```
  PROGRAM p2 (input, output);

     CONST
        max = 10;

     TYPE
        ar = ARRAY[1..max] OF char;

     VAR
        v: ar;
        n, i: integer;

  BEGIN
     REPEAT
        readln(n)
     UNTIL n <= max;
     i := 1;
     WHILE i <= n DO
        BEGIN
           v[i] := chr(ord('A') + i);
           writeln(v[i]);
           i := i + 1
        END
  END.
  ```

- ```
  PROGRAM p3 (input, output);

  CONST max = 10;

  TYPE ar = ARRAY [1..max] OF char;

  VAR v: ar;
      n : integer;
  ```

```
BEGIN
   REPEAT
      readln(n)
   UNTIL n <= max;
   i := 1;
   REPEAT
      v[i] := chr(ord('A') + i);
      writeln(v[i]);
      i := i + 1
   UNTIL i > n
END.
```

-
```
PROGRAM p4 (input, output);

   CONST
      max = 10;

   TYPE
      ar = ARRAY[1..max] OF char;

   VAR
      v: ar;
      n, i: integer;

BEGIN
   REPEAT
      readln(n)
   UNTIL n <= max;
   i := 1;
   REPEAT
      v[i] := chr(ord('A') + i);
      writeln(v[i]);
      i := i + 1
   UNTIL i = n + 1
END.
```

6. Per ognuno dei seguenti programmi indicare:
   - lo "scope" di ciascuna risorsa dichiarata;
   - le risorse visibili all'interno di ogni blocco di codice.

   -
```
PROGRAM p1;

   VAR
      x, y, z: char;


   PROCEDURE c;

      VAR
         z, t: integer;
```

```
   BEGIN {c}
      ...
   END; {c}


   PROCEDURE a;

      VAR
         y: integer;

      PROCEDURE b;
         VAR
            z, w: integer;

      BEGIN {b}
         ...
      END; {b}

   BEGIN {a}
      ...
   END; {a}


BEGIN {p1}
   ...
END. {p1}
```

-
```
PROGRAM p2;

   VAR
      x, y, z: char;


   PROCEDURE b;
      VAR
         z, w: integer;

   BEGIN {b}
      ...
   END; {b}


   PROCEDURE c;

      VAR
         z, t: integer;

   BEGIN {c}
      ...
   END; {c}


   PROCEDURE a;
```

```
      VAR
         y: integer;

   BEGIN {a}
      ...
   END; {a}


BEGIN {p2}
   ...
END. {p2}
```

-
```
PROGRAM p3;

   VAR
      x, y, z: char;


   PROCEDURE b;
      VAR
         z, w: integer;


      PROCEDURE c;

         VAR
            z, t: integer;


         PROCEDURE a;

            VAR
               y: integer;

         BEGIN {a}
            ...
         END; {a}


      BEGIN {c}
         ...
      END; {c}


   BEGIN {b}
      ...
   END; {b}


BEGIN {p3}
   ...
END. {p3}
```

7. Indicare le risorse utilizzabili in ciascun blocco di codice del seguente programma.

```
PROGRAM p (input, output);

   VAR
      x: integer;

   PROCEDURE a;

      PROCEDURE b;

      BEGIN {b}
         writeln('Nella procedura b, la variabile x contiene ', x : 1)
      END; {b}


      PROCEDURE x;

      BEGIN {x}
         b
      END; {x}

   BEGIN {a}
      x
   END; {a}

BEGIN {p}
   x := 10;
   a
END. {p}
```

8. La compilazione di questo programma fornirà un errore. Individuare, senza ricorrere all'aiuto della macchina, il motivo.

```
PROGRAM p (input, output);

   VAR
      x: integer;

   PROCEDURE a;

      PROCEDURE x;

         PROCEDURE b;

         BEGIN {b}
            writeln('Nella procedura b, la variabile x contiene ', x : 1)
         END; {b}

      BEGIN {x}
         b
      END; {x}
```

```
   BEGIN {a}
      x
   END; {a}

BEGIN {p}
   x := 10;
   a
END. {p}
```

9. Si simuli "manualmente" l'esecuzione di ognuno dei seguenti programmi al fine di dedurre l'output prodotto. Si verifichino poi le risposte mandandoli in esecuzione sul proprio computer.

   - ```
     PROGRAM p1 (input,output);
     VAR v: integer;

        PROCEDURE d;
        VAR v: integer;
        BEGIN {v}
           v := 10;
           writeln('Alla fine dell''esecuzione di d la variabile v vale ',v:1)
        END; {v}

     BEGIN {p1}
        v := 5;
        writeln('Prima dell''esecuzione di d la variabile v vale ',v:1);
        d;
        writeln('Dopo l''esecuzione di d la variabile v vale ',v:1)
     END. {p1}
     ```
   - ```
     PROGRAM p2 (input,output);
     VAR v: integer;

        PROCEDURE c;
        BEGIN {c}
           writeln('All''inizio dell''esecuzione di c la variabile v vale ',v:1);
           v := 8;
           writeln('Alla fine dell''esecuzione di c la variabile v vale ',v:1)
        END; {c}

        PROCEDURE d;
        VAR v: integer;
        BEGIN {d}
           v := 10;
           writeln('In d, prima dell''esecuzione di d la variabile v vale ',v:1);
           c;
           writeln('Alla fine dell''esecuzione di d la variabile v vale ',v:1)
        END; {d}

     BEGIN {p2}
        v := 5;
        writeln('Prima dell''esecuzione di d la variabile v vale ',v:1);
     ```

```
   d;
   writeln('Dopo l''esecuzione di d la variabile v vale ',v:1)
END. {p2}
```

- ```
  PROGRAM p3 (input,output);
  VAR v: integer;

     PROCEDURE d;
     VAR v: integer;

        PROCEDURE c;
        BEGIN {c}
           writeln('All''inizio dell''esecuzione di c la variabile v vale ',v:1);
           v := 8;
           writeln('Alla fine dell''esecuzione di c la variabile v vale ',v:1)
        END; {c}


     BEGIN {d}
        v := 10;
        writeln('In d, prima dell''esecuzione di c la variabile v vale ',v:1);
        c;
        writeln('Alla fine dell''esecuzione di d la variabile v vale ',v:1)
     END; {d}

  BEGIN {p3}
     v := 5;
     writeln('Prima dell''esecuzione di d la variabile v vale ',v:1);
     d;
     writeln('Dopo l''esecuzione di d la variabile v vale ',v:1)
  END. {p3}
  ```

Si presti particolare attenzione al fatto che sebbene le istruzioni dei programmi `p2` e `p3` e dei relativi sottoprogrammi siano identiche, gli output prodotti risultano differenti. Ciò è dovuto alla diversa nidificazione delle procedure. Si individui l'ambiente in cui è stata dichiarata la variabile `v` utilizzata dalla procedura `c` di `p2`, e l'ambiente in cui è stata dichiarata la variabile `v` utilizzata dalla procedura `c` di `p3`.

10. Si consideri un programma Pascal con la seguente struttura:

```
PROGRAM p (input,output);
VAR x,y,z: integer;

   PROCEDURE a (VAR x: integer);
   VAR w,z: char;
   BEGIN {a}
      ...
   END; {a}

   PROCEDURE b (VAR y,z: integer);
   VAR w: char;

      PROCEDURE c (y: integer);
      VAR z: char;
```
```

```
      BEGIN {c}
         ...
      END; {c}

      PROCEDURE d (w: char);
      VAR x: integer;
      BEGIN {d}
         ...
      END; {d}

   BEGIN {b}
      ...
   END; {b}

BEGIN {p}
   ...
END. {p}
```

Per ogni procedura presente nel programma si indichino i nomi della variabili che possono essere utilizzate, il loro tipo, e il punto in cui sono state dichiarate.

11. Per ognuna delle procedure presenti nei seguenti programmi si individuino le variabili che possono essere utilizzate e il punto in cui sono state dichiarate. Si simuli "manualmente" l'esecuzione dei programmi al fine di dedurre l'output prodotto. Si verifichino poi le risposte mandandoli in esecuzione sul proprio computer.

- ```
  PROGRAM p1 (input,output);

  VAR x: integer;

     PROCEDURE a;
     BEGIN {a}
        x:= 2*x;
        writeln('Alla fine di a il valore di x e'' ',x:1)
     END; {a}

     PROCEDURE b;
     VAR x: integer;
     BEGIN {b}
        x:= 10;
        a;
        x:=x+2
     END; {b}

  BEGIN {p1}
     x:= 100;
     b;
     writeln('Alla fine del programma il valore di x e'' ',x:1)
  END. {p1}
  ```
- ```
  PROGRAM p2 (input,output);

  VAR x: integer;
  ```

```
      PROCEDURE b;
      VAR x: integer;

         PROCEDURE a;
         BEGIN {a}
            x:= 2*x;
            writeln('Alla fine di a il valore di x e'' ',x:1)
         END; {a}

      BEGIN {b}
         x:= 10;
         a;
         x:=x+2
      END; {b}

   BEGIN {p2}
      x:= 100;
      b;
      writeln('Alla fine del programma il valore di x e'' ',x:1)
   END. {p2}
```

12. Si consideri un programma Pascal con la seguente struttura:

```
PROGRAM p;
VAR x,y,z: integer;

   PROCEDURE a (VAR x: integer);
   VAR y: char;

      PROCEDURE b;
      VAR z: integer;
      BEGIN {b}
         ...
      END; {b}

   BEGIN {a}
      ...
   END; {a}

   PROCEDURE b;
   VAR x,y: real;
   BEGIN {b}
      ...
   END; {b}

BEGIN {p}
   ...
END. {p}
```

Per ogni procedura indicare i nomi delle procedure che essa può chiamare e i nomi delle variabili che possono essere utilizzate, indicando dove state dichiarate.

Per ognuna delle seguenti istruzioni, indicare in quali procedure è consentita e in quali no:

```
x := y + z
x := x + z
z := z + x
z := x + y
```

13. Si consideri un programma Pascal con la seguente struttura:

```
PROGRAM P;
VAR x,y: integer;

   PROCEDURE A;
   VAR u,v: integer;

      PROCEDURE B;
      VAR x,z: integer;
      BEGIN {B}
         ...
      END; {B}

      PROCEDURE C;
      VAR v,y: char;
      BEGIN {C}
         ...
      END; {C}

   BEGIN {A}
      ...
   END; {A}

   PROCEDURE D;
   VAR x: integer;
   BEGIN {D}
      ...
   END; {D}

BEGIN {P}
   ...
END. {P}
```

Per ogni procedura indicare i nomi delle procedure che essa può chiamare, e i nomi delle variabili che possono essere utilizzate, indicando dove state dichiarate.

14. Si consideri un programma Pascal con la seguente struttura:

```
PROGRAM p;
VAR x,y,z: integer;

   PROCEDURE a (VAR x: integer);
   VAR y: char;

      PROCEDURE b;
      VAR z: integer;
      BEGIN {b} {*}
```

```
        ...
      END; {b}

   BEGIN {a} {*}
      ...
   END; {a}

   PROCEDURE c (VAR g: integer);
   VAR y: char;

      PROCEDURE d;
      VAR z: integer;
      BEGIN {d} {*}
         ...
      END; {d}

      PROCEDURE e (VAR a: real);
      VAR y: integer;

          PROCEDURE f;
          VAR a: integer;
          BEGIN {f} {*}
            ...
          END; {f}


      BEGIN {e} {*}
         ...
      END; {e}

   BEGIN {c} {*}
      ...
   END; {c}

BEGIN {p} {*}
   ...
END. {p}
```

Indicare cosa contiene la symbol table, quando il compilatore Pascal legge le linee indicate con `{*}`.