INTRODUZIONE AL LINGUAGGIO ADA95

DAVIDE TAMBUCHI

1. Primi passi con Ada

Il primo programma in Ada é composto da poche righe di codice, e si limita a visualizzare la parola `Ciao` sullo schermo. Chi conosce giá un linguaggio di programmazione, noterá la grande leggibilitá delle righe di codice di questo linguaggio. Per chi inizia ora a programmare, consiglio di aprire un editor di testi qualsiasi e di scrivere il codice seguente:

```
WITH Ada.text_io; -- la libreria text_io contiene le funzioni
USE Ada.text_io;  -- e le procedure. utilizzate per leggere dati
                  -- da tastiera e scriverli su video

PROCEDURE saluto IS
   -- la procedura ha un nome: saluto; conviene
   -- scegliere nomi che indicano il compito
   -- svolto dalla procedura (in questo caso un
   -- saluto)
BEGIN
   -- la serie di istruzioni e' compresa tra
   -- un BEGIN ed un END
   put("Ciao");   -- Scrive su video la parola "Ciao"
   new_line;      -- Va a capo
END saluto;
   -- Viene indicato, a fianco dell'END, per
   -- maggiore leggibilita' del programma, anche
   -- il nome della procedura
```

Il programma é costituito da una procedura, che ha un nome, e nella quale, tra le istruzioni `BEGIN` ed `END` vengono elencate le istruzioni del programma: in questo caso abbiamo l'istruzione `put("Ciao");` che scrive sul video (il dispositivo standard di output) la parola `Ciao`. Abbiamo poi l'istruzione: `new_line`; che va a capo. I doppi trattini (`--`) servono per poter inserire dei commenti, ignorati dalla macchina, ma utili al programmatore per descrivere il significato delle istruzioni e delle righe di codice scrittte. Dopo aver scritto il codice, salvarlo con il nome: `saluto.adb` (il nome del file deve essere uguale a quello della procedura, e l'estensione deve essere `.adb`). Per poterlo compilare, aprire una finestra terminale, e digitare il comando `gcc -c saluto.adb`. Se ci sono errori, vengono segnalati, a partire dal primo, con l'indicazione della riga e della colonna dove tali errori sono stati commessi. Se non ci sono errori, non compare alcun messaggio; e si é ottenuto un file dal nome `saluto.ali`. Occorre adesso utilizzare un programma detto `linker`, e per far ció digitiamo il comando `gnatbl saluto.ali`

A questo punto, si ottiene un file eseguibile, di nome `saluto` (senza estensione). Digitando il comando `./saluto` si esegue il programma, e sul video apparirá, come si desiderava, la parola `Ciao`. Sul significato di compilazione e di collegamento torneremo in seguito. A partire da qui, é possibile scrivere un programma leggermente piú complesso, che visualizza per cinque volte la parola `Ciao`, ad intervalli di 2 secondi. Questo programma, dal nome `cinque_ciao.adb` utilizza, per visualizzare la

parola `Ciao`, la procedura `saluto`, scritta in precedenza. Occorre aprire un nuovo file con un editor di testi, e scrivere il seguente codice:

```
WITH saluto;
   -- la procedura cinque_ciao utilizza, al suo interno,
   -- la procedura saluto scritta in precedenza. Con questa
   -- dichiarazione il compilatore viene avvisato che si
   -- intende utilizzare la procedura saluto.

PROCEDURE cinque_ciao IS
   -- dichiarazione della procedura cinque_ciao
BEGIN
   FOR numero_volte IN 1..5 LOOP
      -- la variabile numero_volte serve
      -- come contatore del ciclo
      -- che viene eseguito per 5 volte
      saluto;
      -- viene chiamata qui la procedura saluto
      delay 2.0;
      -- ritardo in secondi (occorre esprimere almeno
      -- una cifra dopo la virgola)
   END LOOP;
   -- il ciclo, da eseguire 5 volte, inizia con le
   -- istruzioni FOR ... LOOP e termina con
   -- l'istruzione END LOOP;
END cinque_ciao;
```

Notiamo cone non sia necessario indicare la libreria `Ada.text_io` in quanto questa procedura non utilizza istruzioni per scrivere o leggere dei dati, ma chiama la procedura `saluto` che si occupa di ció. Salviamo ora il programma con il nome `cinque_ciao.adb`, e compiliamolo e linkiamolo con le istruzioni:

```
gcc -c cinque_ciao.adb
gnatbl cinque_ciao.ali
```

Adesso possiamo eseguire il programma, digitando `./cinque_ciao`, otteniamo cosí sul video, per cinque volte, la parola `Ciao`.

Passiamo ora ad un altro programma piú complesso, che chiede l'etá di una persona da tastiera, e visualizza per cinque volte la scritta `Ciao` solo se l'eta é inferiore o uguale a 18 anni. In caso contrario, visualizza una sola volta, (con maggiore rispetto), la frase `Buon giorno, signore`. Se l'etá introdotta non é compresa tra 0 e 120 anni, viene visualizzato il messaggio `Come fai ad essere in vita?`. Scriviamo dunque le seguenti righe di codice, in un nuovo file di nome `anni.adb`:

```
WITH cinque_ciao;        -- questo programma utilizza la
                         -- procedura cinque_ciao
WITH Ada.text_io;        -- libreria standard per leggere (input)
USE Ada.text_io;         -- e scrivere (output) testi
WITH Ada.integer_text_io;
USE Ada.integer_text_io; -- libreria standard per leggere e
                         -- scrivere numeri interi

PROCEDURE anni IS
   eta: integer RANGE 0..120; -- eta e' una variabile intera compresa
                              -- tra 0 e 120
BEGIN
   put("Introduci la tua eta', per favore : ");
   -- messaggio video per chiedere l'introduzione dell'eta'
   get(eta);
   -- l'istruzione get legge l'eta' da tastiera, e la
```

```
   -- assegna alla variabile eta. Se il dato introdotto
   -- non e' un numero compreso tra 0 e 120, lo considera
   -- come errore, ed esegue l'istruzione che segue la parola
   -- EXCEPTION (eccezione)
   IF eta <= 18 THEN
      -- IF significa 'se', THEN significa 'allora'
      cinque_ciao;
      -- se la persona ha meno di 19 anni, visualizza per cinque
      -- volte la parola "Ciao" (chiama, a tal scopo, la procedura
      -- cinque_ciao)
   ELSE
      -- ELSE significa 'altrimenti'
      put("Buon giorno, signore");
      -- se l'eta' supera i 18 anni visualizza un messaggio piu'
      -- opportuno
   END IF;
   -- la struttura condizionale IF...THEN...ELSE (se, allora,
   -- altrimenti) viene chiusa con il comando END IF

   -- le righe seguenti servono a gestire l'errore di
   -- introduzione dell'eta': se si introduce un numero
   -- non valido, viene visualizzato un opportuno messaggio
   EXCEPTION
      WHEN constraint_error => put_line("Come fai ad essere in vita?");
      -- errore: il dato introdotto non e' compreso tra 0 e 120 anni
END anni;
```

Osserviamo l'utilizzo della libreria `Ada.integer_text_io` necessaria per leggere l'etá (come numero intero) da tastiera. Osserviamo altresí l'uso della procedura

```
put_line("Come fai ad essere in vita?");
```

essa é equivalente ai comandi

```
put("Come fai ad essere in vita?");
new_line;
```

ovvero vá a capo automaticamente, dopo aver scritto la frase racchiusa tra virgolette. Ora, compiliamolo e linkiamolo ora con i comandi

```
gcc -c anni.adb
gnatbl anni.ali
```

ed eseguiamolo, digitando `./anni`

La morale di tutto questo capitolo é che si possono costruire programmi via via sempre piú complessi, a partire (e utilizzando) programmi piú semplici, che possono essere provati di volta in volta. Nei prossimi capitoli studieremo il linguaggio in maggiore dettaglio.

2. Costanti e variabili (introduzione)

Il nome di una variabile deve sempre iniziare con una lettera, seguita eventualmente da una o piú lettere e numeri, separati dal carattere di sottolineatura '`\_`'. Dichiarando una variabile si riserva per essa un opportuno spazio in memoria destinato a contenere il valore della variabile stessa. Ad esempio, con la dichiarazione

```
altezza_in_centimetri: integer;
```

si riserva spazio in memoria per memorizzare l'altezza in centimetri di un oggetto. Analogamente, con la dichiarazione di variabile

```
raggio: float;
```

si riserva spazio in memoria per contenere il raggio di una circonferenza. Nel primo di questi due casi, la variabile é stata dichiarata di tipo intero (`integer`), ovvero il dato in essa contenuto é un numero intero. Nel secondo caso, il dato é un numero reale (`float`). Consideriamo adesso le seguenti istruzioni; alcuni assegnamenti sono validi, altri no, come mostrano i commenti a fianco

```
altezza_in_centimetri := 178;   -- valido: 178 e' un numero intero
altezza_in_centimetri := 177.2; -- non valido: 177.2 non e' un numero
                                -- intero
raggio := 3.25;                 -- valido: 3.25 e' un numero reale
raggio := 4.0;                  -- valido: scrivere 4.0 anziche' 4
                                -- mette in mostra che la variabile
                                -- raggio assume valori reali
raggio := 4.;                   -- errore, occorre scrivere almeno una
                                -- cifra decimale, es. 4.0
raggio := 4;                    -- errore: il compilatore si aspetta un
                                -- numero reale, non un intero
raggio := .2;                   -- errore: occorre scrivere 0.2, come
                                -- nell'esempo seguente
raggio := 0.2;                  -- valido
raggio := -4.5;                 -- valido
```

Per quanto riguarda il tipo piú semplice di variabile alfanumerica, Ada dispone del tipo character; una variabile dichiarata di tipo character serve a memorizzare un qualsiasi carattere `ASCII` (lettere, numeri, simboli speciali). Ad esempio, si possonno avere le seguente dichiarazioni e assegnazioni:

```
lettera : character; -- dichiara una variabile destinata a contenere
                     -- un carattere ASCII
lettera := 'a';      -- viene assegnato alla variabile lettera il
                     -- carattere 'a' (i caratteri devono essere
                     -- sempre racchiusi) tra apici singoli
lettera := ''';      -- viene assegnato alla variabile lettera il
                     -- carattere apostrofo
lettera := '7';      -- viene assegnato il carattere 7 (non il numero 7,
                     -- fare attenzione agli apici)
lettera := 9;        -- errore: non posso assegnare un numero ad una
                     -- variabile di tipo character
lettera := 'Laura';  -- errore; posso assegnare un solo carattere ad una
                     -- variabile di tipo character
```

Passiamo ora alla dichiarazione delle costanti. Quando in una procedura alcuni parametri sono costanti, conviene dichiararli come tali. Ad esempio, consideriamo la seguente procedura, che converte una temperatura espressa in gradi Celsius in Kelvin (scala di temperatura assoluta). La formula di conversione é:

```
temperatura_kelvin := temperatura_celsius + 273.15;
```

conviene allora dichiarare come costante il termine `273.15`, come illustrato qui sotto:

```
WITH Ada.text_io;
USE Ada.text_io;       -- libreria per input/output testi
WITH Ada.float_text_io;
USE Ada.float_text_io; -- libreria per input/output numeri reali
                       -- (le due temperature)

PROCEDURE celsius_to_kelvin IS
   temperatura_celsius : float;
   temperatura_kelvin : float;
   addendo_conversione: CONSTANT float := 273.15;
```

```
      -- dichiarazione della costante di conversione
   BEGIN
      put ("Introduci la temperatura in gradi Celsius : ");
      get (temperatura_celsius);
      temperatura_kelvin := temperatura_celsius + addendo_conversione;
      put ("La temperatura in kelvin vale : ");
      put (temperatura_kelvin, EXP=>0, FORE=>4, AFT=>2);
      -- il comando EXP=>0 visualizza la temperatura in notazione non
      -- scientifica
      -- il comando FORE=>4 visualizza lascia quattro spazi (segno
      -- compreso) prima della virgola
      -- (FORE e' abbreviazione di before, prima)
      -- il comando AFT=>2 visualizza due cifre dopo la virgola
      -- (AFT e' abbreviazione di after, dopo)
END celsius_to_kelvin;
```

La parola riservata per la dichiarazione delle costanti é `CONSTANT`, che viene messa prima del tipo (in questo caso prima di `float`); dopo il tipo si assegna (con l'istruzione `:=`) il valore alla costante. Nella procedura, sono state messe in evidenza alcuni comandi per una formattazione (impaginazione) del risultato da visualizzare; questi comandi (`EXP`, `FORE`, `AFT`) sono tuttavia facoltativi. E' comunque utile adoperarli sin dall'inizio, per abituarsi ad impaginare il risultato in modo elegante. Cosí come é importante fare uso abbondante di commenti; essi favoriscono la leggibilitá e le modifiche ad un programma anche dopo molti anni. Un altro tipo di variabile é il tipo logico (`boolean`); una variabile logica puó assumere due soli valori: vero (`TRUE`) o falso (`FALSE`). Il programma seguente mostra l'uso di una variabile logica per controllare l'uscita da un ciclo. Esso consiste in un semplice gioco: il giocatore 1 introduce un numero intero compreso tra 1 e 5 ed il giocatore 2 tenta di indovinarlo. Puó provarci piú volte, in quanto il programma é costituito da un ciclo in cui viene continuamente chiesto il numero da indovinare, sino a quando il giocatore non introduce la soluzione esatta.

```
WITH Ada.text_io; USE Ada.text_io;
WITH Ada.integer_text_io; USE Ada.integer_text_io;

PROCEDURE indovina_un_numero IS
   numero_minimo : CONSTANT integer := 1;
   numero_massimo : CONSTANT integer := 5;
   -- il numero massimo e minimo da introdurre vengono dichiarati
   -- come costanti, al fine di poterli cambiare facilmente
   -- modificando questi valori scritti in cima alla procedura
   numero_da_indovinare : integer RANGE numero_minimo..numero_massimo;
   -- numero da indovinare, introdotto dal giocatore 1
   -- e' un numero intero compreso tra un valore minimo e massimo
   numero : integer RANGE numero_minimo .. numero_massimo;
   -- numero introdotto, per tentativi, dal giocatore 2
   fine_partita : boolean := FALSE;
   -- variabile logica utilizzata per individuare la fine
   -- della partita; conviene inizializzarla qui a FALSE
   -- (tutte le volte che e' possibile assegnare un valore ad
   -- una variabile, conviene inizializzarla subito dopo la
   -- dichiarazione di tipo, per una maggiore leggibilita'
BEGIN
   -- Viene chiesto ora al giocatore 1 di introdurre il numero
   -- da indovinare. Notiamo come la specifica WIDTH => 1 visualizza
   -- sia il numero_minimo che il numero_massimo con 1 cifra decimale
   -- La parola WIDTH e' l'equivalente, per i numeri interi, delle
```

```
-- parole EXP, FORE, AFT utilizzate per i numeri reali

put("Giocatore 1, introduci un numero compreso tra ");
put(numero_minimo, WIDTH => 1);
put(" e ");
put(numero_massimo, WIDTH => 1);
put(" : ");
get(numero_da_indovinare);
new_line;

-- viene ora visualizzato un messaggio per il giocatore 2

put("Tocca ora al giocatore 2 indovinare un numero compreso tra ");
put(numero_minimo, WIDTH => 1);
put(" e ");
put(numero_massimo, WIDTH => 1);
new_line;

-- inizio del ciclo di gioco, in cui viene continuamente chiesto
-- al giocatore 2 di introdurre un numero, sino a quando non lo
-- indovina

WHILE NOT fine_partita LOOP
   -- si tratta di un ciclo indefinito, ovvero questo ciclo
   -- puo' essere eseguito per un numero non definito di volte;
   -- se il giocatore 2 continua a non indovinare, non si esce dal
   -- ciclo (e la partita non ha termine). La parola WHILE
   -- significa finche', e si resta in ciclo (delimitato dalle
   -- parole LOOP ... END LOOP) se fine_partita vale FALSE
   -- infatti in questo caso l'espressione logica NOT fine_partita
   -- vale TRUE (negazione di FALSE)
   -- Se l'espressione logica compresa tra WHILE e LOOP vale
   -- TRUE il ciclo prosegue, altrimenti si esce dal ciclo.
   put("Giocatore 2, prova a indovinare il numero : ");
   get(numero);
   -- viene chiesto il numero al giocatore 2
   IF numero < numero_da_indovinare THEN
      -- se (IF) il numero introdotto e' minore del numero da
      -- indovinare allora (THEN) viene visualizzato un messaggio
      -- di aiuto per il giocatore 2
      put_line("Il numero introdotto e' troppo piccolo");
   ELSIF numero > numero_da_indovinare THEN
      -- altrimenti, se (ELSIF) il numero introdotto e' maggiore
      -- del numero da indovinare, viene visualizzato un altro
      -- messaggio che aiuta opportunamente il giocatore 2
      put_line("Il numero introdotto e' troppo grande");
   ELSE
      -- altrimenti (ELSE) viene visualizzato un messaggio che
      -- indica al giocatore 2 che ha indovinato, e viene assegnato
      -- il valore TRUE alla variabile logica fine_partita, per
      -- poter uscire dal ciclo
      put_line("Bravo, hai indovinato");
      fine_partita := TRUE;
   END IF;
END LOOP;

-- trattamento delle eccezioni: se in una delle istruzioni get
```

```
   -- viene introdotto dato errato (esempio, un numero fuori dal campo
   -- di valori richiesto), viene visualizzato un messaggio di errore
   EXCEPTION
      WHEN constraint_error => put_line("introduzione errata del dato");

END indovina_un_numero;
```

3. Istruzioni condizionali

Giá nei programmi illustrati nei capitoli precedenti, abbiamo visto come l'esecuzione di alcune istruzioni fosse condizionata al verificarsi o no di una certa condizione, ovvero alla veritá o falsitá di una data espressione logica. Nel linguaggio naturale, ció si verifica tutti i giorni, ad esempio nel ragionamento:

```
SE piove
   ALLORA prendo l'ombrello
```

Un altro esempio puó essere dato dalla frase:

```
SE il tempo e' sereno
   ALLORA vado in bicicletta
   ALTRIMENTI vado in piscina
```

Infine si possono verificare piú possibilitá; ad esempio:

```
SE piove
   ALLORA vado in piscina
   ALTRIMENTI SE nevica
      ALLORA vado a sciare
      ALTRIMENTI vado in bicicletta
```

Il primo caso corrisponde in Ada ad una struttura condizionale del tipo

```
IF espressione logica THEN
   istruzioni da eseguire
END IF;
```

Il programma seguente utilizza questa struttura condizionale per visualizzare la media di due voti; e se la media risulta insufficiente, visualizza altresí un messaggio di avvertimento.

```
WITH Ada.text_io; USE Ada.text_io;
WITH Ada.integer_text_io; USE Ada.integer_text_io;

PROCEDURE media_voti IS  -- calcolo della media di due voti
                         -- (come numero intero)

   -- dichiarazione delle costanti
   voto_minimo : CONSTANT integer := 1; -- voto minimo in Italia
   voto_massimo : CONSTANT integer := 10; -- voto massimo in Italia
   sufficienza : CONSTANT integer := 6; -- soglia della sufficienza

   -- dichiarazione delle variabili
   primo_voto, secondo_voto: integer RANGE voto_minimo..voto_massimo;
   media: integer RANGE voto_minimo..voto_massimo;

BEGIN

   -- introduzione voti da tastiera
   put("Introduci il primo voto : ");
   get(primo_voto);
   put("Introduci il secondo voto : ");
   get(secondo_voto);
```

```
   -- calcolo della media (divisione tra interi)
   media := (primo_voto + secondo_voto) / 2;

   -- visualizzazione della media
   put("La media dei due voti e' : ");
   put(media, WIDTH => 2);
   new_line;

   -- se la media e' insufficiente, visualizza un messaggio che
   -- invita a studiare.
   -- Qui si puo' notare l'uso della struttura condizionale
   -- IF...THEN...ELSE; se la condizione media < sufficienza
   -- e' vera (TRUE), vengono eseguite le istruzioni comprese tra
   -- THEN ed END IF

   IF media < sufficienza THEN
      put("Caro baby, devi studiare di piu'");
      new_line;
   END IF;

   EXCEPTION
      WHEN constraint_error => put_line("Voto non valido");
      -- trattamento delle eccezioni; voto introdotto non corretto

END media_voti;
```

Il secondo caso si verifica quando occorre scegliere tra due condizioni; ad esempio possiamo modificare la struttura condizionale del precedente programma per visualizzare anche un messaggio di complimenti se la media dei voti é sufficiente. In questo caso, la struttura condizionale assume la forma:

```
IF condizione THEN
   instruzioni da eseguire (se la condizione e' vera)
ELSE
   istruzioni da eseguire (se la condizione e' falsa)
ENDIF
```

Per quanto detto, possiamo riscrivere la struttura condizionale del programma `media_voti` come segue:

```
IF media < sufficienza THEN
   put("Caro baby, devi studiare di piu'");
   new_line;
ELSE
   put("Caro baby, sei bravo");
   new_line;
END IF;
```

Infine, possiamo avere un terzo caso, in cui si vuol scegliere tra piú di una possibilitá; ad esempio il programma seguente legge i coefficienti di una equazione di secondo grado del tipo:

$$ax^2 + bx + c = 0$$

e determina se esistono due soluzioni reali distinte, due soluzioni coincidenti o nessuna soluzione reale. In questo caso si utilizza la struttura condizionale seguente:

```
IF condizione1 THEN
   istruzioni (da eseguire se la condizione 1 e' vera)
ELSIF condizione2 THEN
   istruzioni (da eseguire se la condizione 1 era falsa e la 2 e' vera)
ELSE
```

```
   istruzioni (da eseguire in tutti gli altri casi
ENDIF;
```

Scriviamo dunque il programma:

```
WITH Ada.text_io;
USE Ada.text_io;
WITH Ada.float_text_io;
USE Ada.float_text_io;

PROCEDURE equazione_secondo_grado IS

   -- dichiarazione delle variabili
   a, b, c : float; -- coefficienti dell'equazione
   discrim : float; -- discriminante

BEGIN

   -- introduzione dei dati da tastiera
   put("Introduci il coefficiente di x^2 : "); get(a);
   put("Introduci il coefficiente di x : "); get(b);
   put("Introduci il termine noto : "); get(c);

   -- calcolo del discriminante (si eleva a potenza con **)
   -- notiamo che con l'operatore ** si puo' elevare un numero
   -- solo a potenze intere positive (esempio, non si puo'
   -- scrivere b**2.0 oppure b**2.5
   discrim := b**2 - 4.0*a*c;

   -- visualizzazione di un opportuno messaggio, a seconda dei casi
   IF discrim > 0.0 THEN
      put_line("esistono due soluzioni reali distinte");
   ELSIF discrim = 0.0 THEN
      put_line("esistono due soluzioni reali coincidenti");
   ELSE
      put_line("non esistono soluzioni reali");
   END IF;

END equazione_secondo_grado;
```

Notiamo come, per il confronto tra la variabile `discrim` e `0.0` non si usa l'operatore di assegnazione `:=`, ma l'operatore di confronto `=`. Notiamo altresí come il confronto venga fatto tra numeri reali (`float`); di conseguenza occorre scrivere `0.0` e non `0` (che rappresenterebbe un intero). Inoltre, non é possibile chiamare il discriminante con il nome delta utilizzato di solito nei testi di Matematica in lingua italiana, perché `DELTA` é una parola riservata del linguaggio Ada (cosí come lo sono `IF`, `ELSE`, `WHEN`, e tutte le parole scritte fin qui in maiuscolo). Osserviamo inoltre che Ada non fa distinzione tra maiuscole e minuscole (cosí ad esempio `THEN` é equivalente a `then` ed anche a `Then`; tuttavia per una migliore leggibilitá dei programmi, é buona norma scrivere le parole riservate in maiuscolo) La struttura condizionale potrebbe essere scritta anche evitando l'utilizzo dell'istruzione `ELSIF`, semplicemente utilizzando le istruzioni `IF`, `THEN`, `ELSE`, ma questo sarebbe stato un cattivo esempio di programmazione, per la diminuzione di leggibilitá del programma. Ecco un esempio di come NON si dovrebbe mai programmare:

```
-- cattivo esempio di programmazione

IF discrim > 0.0 THEN
   put_line("esistono due soluzioni reali distinte");
```

```
ELSE
   IF discrim = 0.0 THEN
      put_line("esistono due soluzioni reali coincidenti");
   ELSE
      put_line("non esistono soluzioni reali");
   ENDIF;
ENDIF; -- che brutto dover utilizzare
       -- due END IF, uno per ciascun IF
```

Si possono mettere quante condizioni si vogliono: ad esempio:

```
IF condizione 1 THEN
   istruzioni
ELSIF condizione 2 THEN
   istruzioni
ELSIF condizione 3 THEN
   istruzioni
...
ELSIF condizione n THEN
   istruzioni
ELSE -- l'ultimo else e' facoltativo
   istruzioni -- facoltative
ENDIF
```

Quando occorre scegliere tra piú casi, conviene utilizzare a volte la struttura `CASE`, corrisponente, in linguaggio naturale, ad una frase del tipo:

```
NEI CASI IN CUI il tuo voto VALE
   QUANDO e' tra 1 e 3 ALLORA studia di notte
   QUANDO e' 4         ALLORA studia anche di sabato
   QUANDO e' 5         ALLORA studia un poco di piu'
   NEGLI ALTRI CASI    ALLORA sono contento
```

In Ada ció puó essere realizzato con una struttura del tipo:

```
CASE voto IS
   WHEN 1      => put_line("Studia di notte");
   WHEN 2      => put_line("Studia di notte");
   WHEN 3      => put_line("Studia di notte");
   WHEN 4      => put_line("Studia anche di sabato");
   WHEN 5      => put_line("Studia un poco di piu'");
   WHEN OTHERS => put_line("Sono contento");
END CASE;
```

ove voto é una variabile di tipo intero. La freccia (`=>`) equivale alla parola ALLORA; viene evitato l'uso del termine `THEN` per evitare confuzione con la struttura `IF...THEN...ENDIF`. I casi in cui il voto é compreso tra 1 e 3 potevano essere scritto in modo raggruppato elencando il campo di valori, come nel seguente frammento di codice, del tutto equivalente al precedente, ma decisamente migliore per sinteticitá e leggibilitá:

```
CASE voto IS
   WHEN 1..3   => put_line("Studia di notte");
   WHEN 4      => put_line("Studia anche di sabato");
   WHEN 5      => put_line("Studia un poco di piu'");
   WHEN OTHERS => put_line("Sono contento");
END CASE;
```

In alcuni casi, occorre anche elencare diversi casi che non sono in ordine; ad esempio il seguente programma chiede di introdurre il valore di una moneta in euro, e visualizza se la moneta é costuita da un solo metallo rossastro (pezzi da 1, 2, 5 centesimi); da un solo metallo di colore giallo-oro (pezzi da 10, 20, 50 centesimi) o se é costituita da due metalli (pezzi da 1 e 2 euro). Il valore viene

chiesto in centesimi; se si introduce un valore errato (esempio 62 centesimi) viene visualizzato un messaggio di errore. Se il valore é inferiore a 1 centesimo o superiore a 2 euro viene visualizzato un messaggio che indica che l'importo introdotto é fuori dal campo di valori ammesso (mediante il trattamento di una eccezione).

```
WITH Ada.text_io; USE Ada.text_io;
WITH Ada.integer_text_io; USE Ada.integer_text_io;

PROCEDURE colore_della_moneta IS

  -- dichiarazione delle costanti
  valore_minimo : CONSTANT integer := 1;
                  -- 1 centesimo
  valore_massimo: CONSTANT integer := 200;
                  -- 200 centesimi = 2 euro

  -- dichiarazione delle variabili: importo della moneta
  valore : integer RANGE valore_minimo..valore_massimo;

BEGIN

  -- introduzione importo della moneta
  put("Introduci l'importo della moneta in centesimi di euro: ");
  get(valore);

  -- struttura condizionale di tipo CASE per la visualizzazione
  -- del tipo di metallo della moneta. Notiamo che il simbolo |
  -- significa OPPURE

  CASE valore IS
    WHEN 1 | 2 | 5    => put_line("La moneta e' rossastra");
                         put_line("E ci compri ben poco");
    WHEN 10 | 20 | 50 => put_line("La moneta e' giallo-oro");
    WHEN 100 | 200    => put_line("La moneta e' di due colori");
    WHEN OTHERS       => put_line("La moneta non esiste");
                         put_line("Dove l'hai vista?");
  END CASE;

  EXCEPTION
     -- eccezione: valore di moneta fuori campo ammesso
     WHEN constraint_error
          => put_line("Valore fuori campo ammesso");

END colore_della_moneta;
```

4. Cicli e ripetizioni

Puó capitare che una certa istruzione (o serie di istruzioni) debbano essere eseguite piú di una volta. Come al solito, facciamo riferimento al linguaggio naturale per una migliore comprensione; ad esempio quante volte avete sentito frasi del tipo:

```
PER 10 volte RIPETI
   scrivi la frase "Ti amo" (su un bigliettino di auguri)
FINE
```

In questo caso abbiamo a che fare con una RIPETIZIONE, o CICLO (in inglese ed in Ada: `LOOP`); inoltre sono a conoscenza di quante volte occorre eseguire il ciclo (10 volte). Per quest'ultimo motivo si parla di un ciclo DEFINITO. Trasportiamo

ora il tutto in un programma Ada, introducendo la struttura `FOR...LOOP...END LOOP`, destinata a realizzare questo tipo di ciclo.

```
WITH Ada.text_io; USE Ada.text_io;

PROCEDURE ti_amo IS
BEGIN
   FOR i IN 1..10 LOOP
      put_line("Ti amo");
   END LOOP;
END ti_amo;
```

Osserviamo due cose: innanzitutto che NON é necessario dichiarare le variabili di conteggio del ciclo (cio la variabile `i`, che serve per contare da 1 a 10, ovvero per contare il numero di esecuzioni del ciclo stesso). La seconda, che non é possibile alterare il valore della variabile di conteggio (con operazioni di assegnamento) all'interno di un ciclo, ovvero non si puó scrivere istruzioni tipo `i := i + 2;` all'interno del ciclo delimitato dalle parole `LOOP...END LOOP`. Questo ai fini della sicurezza, onde essere sicuri che il numero di esecuzioni del ciclo sia chiaramente specificato dal campo di valori assegnato tra la parola `IN` e la parola `LOOP`. Questo programma puó essere scritto in modo migliore dichiarando come costante il numero di esecuzioni:

```
WITH Ada.text_io; USE Ada.text_io;

PROCEDURE i_love_you IS
   numero_esecuzioni : CONSTANT integer := 10;
BEGIN
   FOR i IN 1..numero_esecuzioni LOOP
      put_line("Ti amo");
   END LOOP;
END i_love_you;
```

Osserviamo che il campo di valori deve essere costituito da due numeri interi[1]. Ad esempio, il ciclo

```
FOR i IN -2..2 LOOP
   -- istruzioni da eseguire
END LOOP;
```

é ammesso (e viene eseguito cinque volte, in cui i assume tutti i valori compresi tra -2 e 2), mentre non lo é il seguente:

```
FOR i IN 1 .. 10.5 LOOP
   -- istruzioni da eseguire
END LOOP;
```

in quanto `10.5` non é un numero intero. Osserviamo inoltre che il valore piú piccolo del campo precede sempre quello piú grande. Nel caso in cui occorra contare all'indietro, si utilizza la parola riservata `REVERSE`, come nell'esempio seguente, che simula un conto alla rovescia da 5 ad 1, ad intervalli di 1 secondo.

```
WITH Ada.text_io; USE Ada.text_io;
WITH Ada.integer_text_io; USE Ada.integer_text_io;

PROCEDURE conteggio_alla_rovescia IS
   numero_iniziale : CONSTANT integer := 5;
BEGIN
   FOR i IN REVERSE 1..numero_iniziale LOOP
      put(i); new_line; -- visualizza il numero i
      delay 1.0; -- ritardo di 1 secondo
   END LOOP;
```

[1] o da qualsiasi altro tipo di variabile *enumerativo*

```
   put_line("Fine conteggio");
END conteggio_alla_rovescia;
```

In alcuni casi, é necessario ripetere piú di una volta una o piú operazioni, ma non é noto il numero di esecuzioni del ciclo di ripetizioni. Parliamo in questo caso di un ciclo INDEFINITO. Ad esempio, consideriamo la seguente frase:

```
FINCHE' la scatola di libri non e' vuota RIPETI
   prendi un libro dalla scatola
   numeralo con una etichetta
   metti il libro numerato sullo scaffale della libreria
FINE CICLO
```

In Ada questa struttura puó essere realizzata con un ciclo `WHILE ... LOOP ... END LOOP`.

```
WHILE condizione LOOP
   -- istruzioni da eseguire;
END LOOP;
```

Se la condizione logica descritta tra la parola riservata `WHILE` e la parola riservata `LOOP` é vera (`TRUE`), vengono eseguite le istruzioni comprese tra `LOOP` ed `END LOOP`; il termine dell'esecuzione avviene quando la condizione logica risulta falsa (`FALSE`). Se giá inizialmente tale condizione risulta falsa, il ciclo non viene eseguito nemmeno una volta. Se é sempre vera, il ciclo viene eseguito infinite volte, e non termina mai. Di conseguenza, il programma non potrá mai terminare. Quest'ultima situazione é evidentemente da evitare. Nel seguente programma il ciclo indefinito viene utilizzato per controllare la corretta introduzione di un dato: l'utente deve introdurre un numero `n` intero maggiore di zero; nel caso in cui egli introduca un numero minore o uguale a zero, viene visualizzato un messaggio che invita alla ripetizione dell'introduzione del numero. Il programma visualizza poi se tale numero é primo. Il controllo viene fatto calcolando il resto della divisione del numero n introdotto per tutti i suoi divisori minori o uguali ad `n/2`. A tale scopo viene utilizzato un secondo ciclo `WHILE...LOOP...END`. Il ciclo termina quando si é effettuata la divisione per `n/2` o quando si é trovato che `n` non é primo.

```
WITH Ada.text_io; USE Ada.text_io;
WITH Ada.integer_text_io; USE Ada.integer_text_io;

PROCEDURE test_primo IS

   n : integer;
   -- numero introdotto dall'utente
   divisore : integer := 2;
   -- per verificare se n e' primo, lo si divide
   -- per tutti i suoi divisori compresi tra 2 ed n/2
   primo: boolean := TRUE;
   -- FALSE se il numero n non e' primo, TRUE se lo e'.
   -- Lo si suppone primo, e si cambia il valore della
   -- variabile primo se si scopre che lo e'.

BEGIN

   -- introduzione del numero da tastiera
   put("Introduci un numero intero maggiore di zero : ");
   get(n);

   -- se n <= 0 occorre reintrodurre il numero
   WHILE n <= 0 LOOP
      put("Il numero non e' maggiore di zero, reintroducilo : ");
```

```
      get(n);
   END LOOP;

   -- test per vedere se il numero e' primo
   -- l'istruzione REM calcola il resto della divisione tra le
   -- variabili n e divisore; se tale resto e' zero il numero non
   -- e' primo (e si esce dal ciclo).
   WHILE (divisore <= n/2) AND primo LOOP
      IF n REM divisore = 0 THEN
         primo := FALSE;
         -- se il resto della divisione e' zero, il numero non e' primo
         -- si assegna percio' il valore FALSE alla variabile logica
         -- primo, e cio' permette di uscire dal ciclo perche' la
         -- condizione compresa tra WHILE e LOOP e' ora falsa.
      ELSE
         divisore := divisore + 1;
         -- se il resto della divisione e' diverso da zero, si riprova
         -- il test con il divisore successivo
      END IF;
   END LOOP;

   -- visualizzazione dei risultato
   IF primo THEN
      put_line("Il numero e' primo");
   ELSE
      put_line("Il numero non e' primo");
   END IF;

END test_primo;
```

Osserviamo come, nella dichiarazione delle variabili, primo e divisore siano state inizializzate, rispettivamente a `TRUE` (si fa l'ipotesi che il numero sia primo, e poi si controlla se non lo é), ed a 2 (il primo divisore). Se si conosce il valore iniziale di una variabile, é buona norma assegnarlo alla variabile stessa giá nella fase di dichiarazione. Osserviamo ora che l'introduzione del numero `n` avviene sicuramente almeno una volta. Per questo motivo, ci chiediamo se esiste una struttura in grado di realizzare un ciclo (indefinito) eseguito almeno una volta. Ció é possibile in Ada, mediante la struttura

```
LOOP
   -- istruzioni;
   EXIT WHEN condizione;
   -- altre eventuali istruzioni;
END LOOP;
```

Questo corrisponde, nel linguaggio naturale, ad una frase del tipo:

```
RIPETI
   fai un esercizio di matematica
   controlla il risultato
   TERMINA QUANDO l'hai svolto esattamente
   segnati sul quaderno le difficolta' che hai incontrato
FINE
```

L'istruzione di uscita dal ciclo (`EXIT WHEN condizione`) permette di uscire dal ciclo quando la condizione logica specificata risulta vera (`TRUE`). Il comando `EXIT` puó essere inserito in un punto qualsiasi all'interno del ciclo. Possono esserci anche piú istruzioni di uscita, come ad esempio nel seguente frammento di programma:

```
-- i, j, k sono numeri interi (integer)
...
```

```
i := 10;
j := 20;
LOOP
   put("Introduci 0 per uscire dal ciclo, o un numero compreso tra ");
   put(i, WIDTH => 2); put(" e "); put(j, WIDTH => 2); new_line;
   get(k);
   EXIT WHEN (k = 0);
   -- esce se e' stato introdotto il numero 0
   EXIT WHEN (i <= k) AND (k <= j);
   -- esce se e' stato introdotto un numero compreso tra i e j
   put_line("il numero introdotto e' errato ");
END LOOP;
```

Per una migliore leggibilitá del programma, é meglio tuttavia utilizzare una sola istruzione di uscita, come nel seguente frammento:

```
-- i, j, k sono numeri interi (integer)
...
i := 10;
j := 20;
LOOP
   put("Introduci 0 per uscire dal ciclo, o un numero compreso tra ");
   put(i, WIDTH => 2); put(" e "); put(j, WIDTH => 2); new_line;
   get(k);
   EXIT WHEN (k = 0) OR ((i <= k) AND (k <= j));
   -- esce se e' stato introdotto il numero 0 oppure se
   -- e' stato introdotto un numero compreso tra i e j
   -- notiamo inoltre come siano necessarie le parentesi per
   -- evitare ambiguita' nell'interpretazione dell'espressione
   -- logica.
   put_line("il numero introdotto e' errato ");
END LOOP;
```

In alcuni linguaggi, come il `C`, non servono le parentesi nell'espressione logica, in quando l'operatore `AND` (indicato in `C` con `&&`) ha la precedenza sull'operatore `OR` (indicato con `||`). Ada, per evitare interpretazioni ambigue dell'espressione logica, non permette di scrivere, come in `C`:

```
if (k==0 || i<=k && k<=j) break; /* break: equivalente di EXIT */
                                 /* ==: equivalente all'operatore = */
                                 /* ||, &&: equivalenti ad OR ed AND */
```

Il programma seguente legge due numeri interi da tastiera, facendoli reintrodurre se sono negativi o nulli, e ne calcola il massimo comun divisore ed il minimo comune multiplo, utilizza diversi tipi di cicli indefiniti, a scopo didattico. I commenti illustreranno in maggior dettaglio suggerimenti sulle tecniche di programmazione.

```
WITH Ada.text_io; USE Ada.text_io;
WITH Ada.integer_text_io; USE Ada.integer_text_io;

PROCEDURE mcm_mcd IS

   n1, n2 : integer;
   -- numeri introdotto dall'utente
   minimo : integer;
   -- minimo tra n1 ed n2
   mcd, mcm : integer;
   -- massimo comun divisore e minimo comune multiplo di n1 ed n2

BEGIN
```

```
-- introduzione dei numeri da tastiera
-- il ciclo indefinito con istruzione di uscita dopo l'istruzione
-- get viene sicuramente eseguito almeno una volta, e cio' per
-- poter introdurre il numero n1
-- questo ciclo e' un buon esempio di programmazione
LOOP
   put("Introduci un numero intero maggiore di zero : ");
   get(n1);
   EXIT WHEN n1 > 0;
   -- se n1 >= 0 esce dal ciclo
   put_line("Il numero introdotto e' errato: reintroducilo ");
   -- altrimenti visualizza un messaggio
   -- che invita a reintrodurlo
END LOOP;

-- stesso ciclo per l'introduzione di n2; solo che si e' scelto
-- (a scopo didattico) uno stile differente: il test di uscita
-- e' posto alla fine.
-- Personalmente, trovo migliore questo secondo modo di scrivere,
-- in quanto piu' leggibile. Entrambi i metodi sono comunque validi,
-- l'importante e' usarne uno solo per entrambi i numeri, al fine
-- di una maggiore leggibilita' del programma.
LOOP
   put("Introduci il secondo intero maggiore di zero : ");
   get(n2);
   IF n2 <= 0 THEN
      put_line("Il numero introdotto e' errato: reintroducilo");
   END IF;
   EXIT WHEN n2 >= 0;
END LOOP;

-- calcolo del mcd
IF n1 < n2 THEN
   -- ricerca del minimo
   minimo := n1;
ELSE
   minimo := n2;
END IF;

-- il ciclo seguente ricerca il mcd, calcolando i resti della
-- divisione di n1 ed n2 per un numero compreso tra il minimo di n1
-- ed n2 e 1, e puo' essere sostituito con un ciclo LOOP...END LOOP
-- in quanto viene sicuramente eseguito almeno una volta.
mcd := minimo;
WHILE (n1 REM mcd /= 0) OR (n2 REM mcd /= 0) LOOP
   -- resta in ciclo se una delle due divisioni ha resto diverso
   -- da zero (l'operatore /= significa 'diverso da')
   mcd := mcd - 1;
   -- e decrementa mcd, per effettarre la divisione successiva
   -- (nel caso peggiore, di n1 ed n2 primi tra loro, esce dal
   -- ciclo quando mcd = 1)
END LOOP;

-- calcolo del minimo comune multiplo (prodotto dei due numeri
-- diviso per il loro massimo comun divisore)
mcm := n1 * n2 / mcd;
```

```
   -- visualizzazione dei risultato (mcd e mcm)
   put("Il massimo comun divisore e' : ");
   put(mcd); new_line;
   put("Il minimo comune multiplo e' : ");
   put(mcm); new_line;

END mcm_mcd;
```

Il ciclo di ricerca del minimo comune multiplo, puó essere sostituito da un ciclo LOOP...END LOOP, che puó essere scritto come segue:

```
mcd := minimo;
LOOP
   EXIT WHEN (n1 REM mcd = 0) AND (n2 REM mcd = 0);
   -- esce dal ciclo se entrambi le divisioni hanno resto zero
   mcd := mcd - 1;
   -- altrimenti decrementa mcd e resta nel ciclo
END LOOP;
```

La scelta é lasciata al programmatore. I cicli possono anche essere annidati, ovvero all'interno di un ciclo (definito o indefinito) é possibile implementare altri cicli, come mostra l'esempio seguente, in cui viene visualizzata una tabellina delle moltiplicazioni.

```
WITH Ada.text_io; USE Ada.text_io;
WITH Ada.integer_text_io; USE Ada.integer_text_io;

PROCEDURE tabellina IS

   num_righe: CONSTANT integer := 10;
   -- numero di righe della tabellina
   num_col: CONSTANT integer := 10;
   -- numero di colonne della tabellina

BEGIN

   -- visualizzazione dei fattori moltiplicativi da 1 a num_col
   -- corrispondenti alle colonne
   put(" ");
   -- sei spazi per allineamento dati della tabella

   -- ciclo per la visualizzazione dei fattori moltiplicativi
   -- rappresentati sulle colonne
   FOR c IN 1..num_col LOOP
      put(c, WIDTH => 4);
   END LOOP;
   new_line;
   new_line;
   -- va a capo e lascia una riga vuota

   -- ciclo esterno, per contare le righe della tabellina,
   -- da 1 a num_righe
   FOR r IN 1..num_righe LOOP
      put(r, WIDTH => 4);
      -- visualizza il numero della riga
      put(" ");
      -- lascia due spazi per maggior leggibilita'

      -- ciclo interno, per contare le colonne della tabellina,
      -- in cui vengono effettuate e visualizzate le moltiplicazioni
```

```
      -- di r (numero riga) per c (numero colonna)
      FOR c IN 1..num_col LOOP
         put(r * c, WIDTH => 4);
         -- il prodotto r * c viene calcolato e visualizzato
         -- direttamente senza essere memorizzato in alcuna variabile
      END LOOP;

      new_line;
      -- va a capo, per iniziare la riga successiva

   END LOOP;

END tabellina;
```

Osserviamo innanzitutto, che potevamo memorizzare il prodotto dichiarando una variabile destinata a tal scopo:

```
prodotto: integer RANGE 1..num_righe*num_col;
```

e sostituendo l'istruzione

```
put(r * c, WIDTH => 4);
```

con la coppia di istruzioni:

```
prodotto := r * c;         -- calcolo del prodotto riga per colonna
put(prodotto, WIDTH => 4); -- visualizzazione del prodotto
```

la scelta, anche in questo caso, é lasciata al programmatore. L'importante é far uso di commenti per garantire la leggibilitá di un programma, anche a distanza di molti anni. Notiamo inoltre, che per distinguere i due cicli annidati, é possibile ETICHETTARLI; a tal scopo prima dei cicli si pone una etichetta (label in inglese) costituita da un opportuno nome del ciclo seguito dal simbolo `:`; inoltre il termine del ciclo viene indicato, con `END LOOP` seguito dall'etichetta data al ciclo:

```
ciclo_su_righe:               -- ETICHETTA CICLO ESTERNO
FOR r IN 1..num_righe LOOP
   put(r, WIDTH => 4);
   put(" ");
   ciclo_su_colonne:          -- ETICHETTA CICLO INTERNO
   FOR c IN 1..num_col LOOP
      put(r * c, WIDTH => 4);
   END LOOP ciclo_su_colonne; -- notare la presenza dell'etichetta
   new_line;
END LOOP ciclo_su_righe;      -- notare la presenza dell'etichetta
```

5. Funzioni

Iniziamo il capitolo ponendoci uno scopo ben preciso: vogliamo scrivere un programma che determini posizione e velocitá di un punto materiale in moto rettilineo uniformemente accelerato in un dato intervallo di tempo, nota la sua velocitá iniziale V_i (individuata dalla variabile `v_iniz`), la sua posizione iniziale x_i (variabile `x_iniz`), e la sua accelerazione `a` (dati introdotti da tastiera). Vengono introdotti da tastiera anche l'incremento `deltat` da dare alla variabile tempo, e il numero `n` di intervalli di tempo da considerare. Sappiamo che la velocitá all'istante t del mobile puó essere calcolata come

$$v = v_i + at$$

e che la sua posizione é individuata dal punto

$$x = x_i + v_i t + \frac{at^2}{2}$$

A tal scopo, creiamo due file, dal nome `velocita.adb` e `posizione.adb` in cui inserire due funzioni che calcolano tali valori. Ricordiamo che in matematica una funzione é una corrispondenza che associa all'insieme dei valori delle variabili indipendenti $x_1, x_2, \ldots, x_n$ uno e un sol valore di una variabile y (detta variabile dipendente). Ad esempio, se b_1, b_2, h rappresentano la base maggiore, la base minore e l'altezza di un trapezio, possiamo definire una funzione $y = f(b_1, b_2, h)$ che determina l'area y del trapezio come:

$$y = \frac{(b_1 + b_2)h}{2}$$

In Ada una funzione é una corrispondenza che calcola il valore di una variabile (dipendente) `y` noti i valori delle variabili dipendenti `x1, x2, ..., xn` (detti parametri della funzione). Il concetto di funzione in Ada é pertanto simile (e generalizza) quello di funzione matematica. La dichiarazione di funzione, ed il suo corpo possono essere cosí definite:

```
FUNCTION nome della funzione (lista e tipo dei parametri)
         RETURN tipo della variabile indipendente IS

-- eventuale dichiarazione di costanti e variabili locali
-- (cioe' ad uso proprio della funzione)

BEGIN

   -- istruzioni per il calcolo della funzione
   RETURN valore della variabile indipendente

END nome della funzione;
```

Ad esempio, la funzione per il calcolo dell'area del trapezio puó essere scritta come:

```
FUNCTION area_trapezio (b1: float; b2: float; h: float) RETURN float IS
   area: float;
BEGIN
   area := (b1 + b2) * h / 2.0;
   RETURN area;
END area_trapezio;
```

La funzione ha un nome: `area_trapezio` (cosí come ogni procedura ha un nome). Per provarla, occorre innanzitutto salvarla in un file (con lo stesso nome, ed estensione `.adb`, come per le procedure): in questo caso salviamola come `area_trapezio.adb` Poi, occorre creare una procedura che utilizzi questa funzione (che cioe' chiami questa funzione al suo interno); a tal scopo creiamo la seguente procedura in un nuovo file, e salviamola con il nome `trapezio.adb`

```
WITH Ada.text_io; USE Ada.text_io;
WITH Ada.float_text_io; USE Ada.float_text_io;
WITH area_trapezio;

PROCEDURE trapezio IS
   base_maggiore, base_minore, altezza : float;
   superficie: float;
BEGIN
   -- introduzione dei dati
   put("Introduci la base maggiore del trapezio : ");
   get(base_maggiore);
   put("Introduci la base minore del trapezio : ");
   get(base_minore);
```

```
   put("Introduci l'altezza del trapezio : ");
   get(altezza);

   -- calcolo della superficie utilizzando la funzione area_trapezio
   superficie := area_trapezio(base_maggiore, base_minore, altezza);

   -- visualizzazione dei risultato
   put("L'area del trapezio vale : ");
   put(superficie, EXP=>0, FORE=>5, AFT=>3);
END trapezio;
```

Possiamo poi compilare sia la funzione che la procedura con i comandi

```
gcc -c area_del_trapezio.adb
gcc -c trapezio.adb
```

e poi collegare assieme i due file (link) con il comando `gnatbl trapezio.ali` e provare il programma con il comando `./trapezio`

Notiamo come la chiamata della funzione non sia molto diversa dalla chiamata di una procedura (confrontare la chiamata della procedura saluto nel programma `cinque_ciao` con la chiamata della funzione `area_trapezio` nel presente programma). La differenza fondamentale consiste nel fatto che la funzione ritorna un valore alla procedura chiamante (in questo caso ritorna l'area del trapezio), che deve essere assegnato ad una variabile utilizzando l'operatore di assegnazione `:=`. Commentiamo ora, passo dopo passo, come avviene l'esecuzione della procedura `trapezio` e della funzione `area_trapezio`; innanzitutto nel programma trapezio viene chiesta l'introduzione della base maggiore, della base minore, e dell'altezza, tramite tastiera, mediante la terna di istruzioni `put` e `get`. Successivamente, la procedura `trapezio` chiama la funzione `area_trapezio` mediante l'istruzione:

```
superficie := area_trapezio(base_maggiore, base_minore, altezza);
```

confrontiamo questa riga con l'intestazione della funzione:

```
FUNCTION area_trapezio (b1: float; b2: float; h: float) RETURN float IS
```

Se confrontiamo i termini racchiusi tra parentesi

```
(base\_maggiore,base\_minore, altezza)
```

nell'istruzione di chiamata della funzione, con i termini

```
(b1, b2, h)
```

presenti nell'intestazione della funzione, osserviamo che il numero di termini é lo stesso (tre), e dello stesso tipo (`float`). In pratica, avviene che i valori delle variabili racchiuse tra parentesi nel'istruzione di chiamata della funzione vengono copiati nei parametri indicati nell'intestazione della funzione stessa, nell'ordine in cui sono specificati nella chiamata e nell'intestazione della funzione. Cioé:

- il valore della variabile `base_maggiore` viene copiato nel parametro `b1`
- il valore di `base_minore` viene copiato in `b2`
- il valore dell'altezza viene copiato in `h`

A questo punto, vengono eseguite le istruzioni della funzione; cioé il calcolo dell'area:

```
area := (b1 + b2) * h / 2.0;
```

che poi viene ritornata alla procedura chiamante mediante l'istruzione RETURN. Il valore dell'area viene cosí assegnato alla variabile superficie, mediante l'operatore `:=` ; infine la procedura visualizza il valore della superficie del trapezio. Notiamo che parametri e variabili hanno nome differente; infatti non é assolutamente necessario chiamare variabili e parametri con lo stesso nome. Anzi, utilizzare nomi differenti (specialmente il altri linguaggi, come il Pascal ed il C) aiuta ad evitare la confusione tra variabili e parametri. La scelta é lasciata al programmatore. Quindi i nomi dei

parametri della funzione possono anche essere diversi dai nomi delle variabili indicati nella chiamata della funzione stessa; ma devono essere dello stesso tipo. Inoltre anche la variabile a cui viene assegnato il risultato ritornato dalla funzione deve essere dello stesso tipo del valore ritornato, indicato nell'intestazione, tra `RETURN` ed `IS`. Osserviamo anche la presenza di una variabile area, dichiarata nella funzione, subito dopo l'intestazione. Questa variabile non puó essere utilizzata all'esterno della funzione; cioé nella procedura chiamante, ma solo all'interno della funzione stessa (serve per memorizzare temporaneamente un dato, in questo caso il valore dell'area). Come ultima cosa, occorre dire che i valori dei parametri `b1`, `b2`, `h` non possono essere modificati all'interno della funzione. Se ad esempio, all'interno della funzione, volessimo cambiare il valore dell'altezza `h`, con l'istruzione `h := h + 1.0;` (o altre istruzioni simili), il compilatore segnalerebbe un errore. Questa restrizione é importante; infatti come in matematica una funzione non altera mai il valore delle sue variabili indipendenti, cosí in Ada una funzione non dovrebbe mai alterare i valori dei suoi parametri. Passiamo ora alla realizzazione del nostro programma, ovvero il calcolo e la visualizzazione della posizione e della velocitá di un punto materiale in moto rettilineo uniforme. La prima funzione che scriviamo, e salviamo nel file `velocita.adb`, calcola la velocitá del mobile nell'instante `t`, note la sua velocitá iniziale `v_i` e la sua accelerazione `acc`.

```
FUNCTION velocita (v_i: float; acc: float; t: float)
         RETURN float IS
   v : float;
BEGIN
   v := v_i + acc * t;
   RETURN v;
END velocita;
```

Analogamente, possiamo scrivere la funzione posizione che determina la posizione del mobile all'istante `t`, nota la sua posizione iniziale `x_i`, la sua velocitá iniziale `v_i` e la sua accelerazione `acc` (e salvarla nel file di nome `posizione.adb`)

```
FUNCTION posizione (x_i:float; v_i:float; acc:float; t:float)
         RETURN float IS
BEGIN
   RETURN x_i + v_i * t + 0.5 * acc * t ** 2;
END;
```

A scopo didattico, abbiamo raggruppato due istruzioni (calcolo della posizione e ritorno del valore) in una sola; potevamo infatti scrivere, dopo aver dichiarato una variabile locale `x` di tipo `float`, la coppia di istruzioni:

```
x := x_i + v_i * t + 0.5 * a * t ** 2;
RETURN x;
```

la scelta é lasciata al programmatore. Non ci resta altro che scrivere una procedura per poter provare le due funzioni; eccola qui:

```
WITH Ada.text_io; USE Ada.text_io;
WITH Ada.integer_text_io; USE Ada.integer_text_io;
WITH Ada.float_text_io; USE Ada.float_text_io;
WITH posizione;
WITH velocita;

PROCEDURE moto_unif_acc IS
   x_iniz : float;
   -- posizione iniziale del mobile (in metri)
   v_iniz : float;
   -- velocita' iniziale del mobile (in metri/secondo)
   a : float;
```

```
   -- accelerazione (in metri/secondo^2)
   deltat : float;
   -- intervallo di tempo (in secondi)
   n : integer;
   -- numero di intervalli di tempo
   t: float;
   -- istante di tempo (in secondi) in cui calcolare
   -- posizione e velocita'
   x : float;
   -- posizione all'istante t
   v : float;
   -- velocita' all'istante t
BEGIN
   -- introduzione dati
   put("Posizione iniziale in metri ? "); get(x_iniz);
   put("Velocita' iniziale in m/s ? "); get(v_iniz);
   put("Accelerazione in m/s^2 ? "); get(a);
   put("Intervallo di tempo in secondi ? "); get(deltat);
   put("Numero intervalli ? "); get(n);

   -- riga di intestazione della tabella dei risultati
   put(" t (s) v (m/s) x (m)"); new_line;

   -- ciclo di calcolo e visualizzazione
   FOR i IN 1 .. n LOOP
      t := float(i) * deltat;
      -- per poter moltiplicare un integer con un float e' necessario
      -- convertire il numero intero n in un reale; cio' viene fatto
      -- con l'istruzione di conversione (cast in inglese) float(n)
      -- ed e' importante per non mischiare (per errore) tra loro tipi
      -- diversi di variabile nelle operazioni matematiche. Quando e'
      -- necessario e giusto farlo, occorre convertire le variabili
      -- tutte nello stesso tipo

      -- chiamata delle funzioni velocita e posizione
      v := velocita(v_iniz, a, t);
      x := posizione(x_iniz, v_iniz, a, t);

      -- visualizzazione dei valori calcolati all'istante t
      put(t, EXP=>0, FORE=>5, AFT=>3);
      put(v, EXP=>0, FORE=>5, AFT=>3);
      put(x, EXP=>0, FORE=>5, AFT=>3);
      new_line;
   END LOOP;
END moto_unif_acc;
```

Proviamo il programma, compilando innanzitutto le funzioni e la procedura con i comandi:

```
gcc -c moto_unif_acc.adb
gcc -c velocita.adb
gcc -c posizione.adb
```

e poi colleghiamoli insieme, con il comando `gnatbl moto_unif_acc.ali` ed eseguiamo il programma, come al solito, con il comando `./moto_unif_acc`

6. Procedure e passaggio dei parametri

Il seguente programma legge un numero intero da tastiera, e ne visualizza il valore assoluto (modulo). Il programma si compone di due file: il primo é una

procedura, put_modulo.adb, che calcola il modulo e lo visualizza; il secondo é il programma chiamante val_assoluto.adb

```
WITH Ada.text_io; USE Ada.text_io;
WITH Ada.integer_text_io; USE Ada.integer_text_io;

PROCEDURE put_modulo (num : IN integer) IS
   -- numero e' il parametro della procedura. E' di tipo IN in quanto
   -- procedura non puo' alterarne il valore. Ad esempio, non puo'
   -- cambiarlo di segno.

   m : integer;
   -- variabile locale per contenere il modulo del numero num

BEGIN
   -- calcolo del modulo di num
   IF num < 0 THEN
      m := -1 * num;
   ELSE
      m := num;
   END IF;

   -- visualizzazione
   put("Il modulo del numero vale : ");
   put(m, WIDTH => 4);
END put_modulo;
```

Salviamo la procedura con il nome put_modulo.adb e compiliamola con il comando gcc -c put_modulo.adb. Scriviamo ora il programma chiamante, e salviamolo con il nome val_assoluto.adb

```
WITH Ada.text_io; USE Ada.text_io;
WITH Ada.integer_text_io; USE Ada.integer_text_io;
WITH put_modulo;

PROCEDURE val_assoluto IS
   n: integer;
   -- numero introdotto da tastiera di cui si vuol calcolare
   -- e visualizzare il modulo
BEGIN
   -- introduzione del numero n da tastiera
   put("Introduci un numero intero : ");
   get(n);

   -- chiamata della procedura per il calcolo e la visualizzazione
   -- del modulo di n
   put_modulo(n);
END val_assoluto;
```

Per provarlo, compiliamolo con il comando gcc -c val_assoluto.adb e colleghiamolo alla procedura con il comando gnatbl val_assoluto.ali; eseguiamolo poi con il comando ./val_assoluto Il programma, dopo l'introduzione del numero n da tastiera, chiama la procedura con il comando

```
put_modulo(n);
```

Il valore della variabile intera n viene copiato nel parametro num della procedura, e non puó essere alterato dalla procedura stessa. Questo é il significato della parola riservata IN presente nell'intestazione:

```
PROCEDURE modulo (num : IN integer) IS
```

La procedura utilizza pertanto un altra variabile (`m`) per calcolare il modulo e visualizzarlo. Se la parola `IN` non viene specificata, il parametro é per convenzione considerato ugualmente (per convenzione) di tipo `IN`. (Infatti non é stato messo nelle intestazioni delle funzioni del precedente capitolo). Potevamo infatti scrivere, come intestazione

```
PROCEDURE put_modulo (num : integer) IS
```

La prossima procedura, che salveremo in un file dal nome `scambia.adb` permette di scambiare tra di loro i valori di due parametri `a`, `b` (di tipo intero)

```
PROCEDURE scambia (a: IN OUT integer; b: IN OUT integer) IS
   -- questa procedura scambia tra di loro i valori di a e b
   temp : integer;
   -- variabile locale usata per memorizzare il valore di a
   -- durante le operazioni di scambio
BEGIN
   temp := a; -- copia a in temp
   a := b;    -- copia b in a
   b := temp; -- copia temp in b
END scambia;
```

Dopo averla compilata, scriviamo un programma in grado di provarla: questo programma legge due numeri `n1`, `n2` da tastiera, che vengono passati alla procedura scambia, che effettua la permutazione dei valori di `n1` ed `n2` che verranno copiati nei parametri `a`, `b` della procedura di scambio al momento della chiamata. Ecco il programma, che salveremo con il nome `swap.adb`:

```
WITH Ada.text_io; USE Ada.text_io;
WITH Ada.integer_text_io; USE Ada.integer_text_io;
WITH scambia;

PROCEDURE swap IS
   -- questa procedura legge due numeri interi n1 ed n2 da
   -- tastiera, chiama una procedura per scambiarli tra di
   -- loro e visualizza i nuovi valori di n1 ed n2
   n1, n2 : integer;
BEGIN
   -- introduzione di n1 ed n2
   put("Introduci n1 : "); get(n1);
   put("Introduci n2 : "); get(n2);

   -- chiamata della procedura di scambio
   scambia (n1, n2);

   -- visualizzazione del risultato
   put("Adesso n1 vale : "); put(n1, WIDTH => 4);
   put(" ed n2 vale : "); put (n2, WIDTH => 4);
END swap;
```

Compiliamolo, linkiamolo con `gnatbl swap.ali`, e proviamolo con `./swap`

Osserviamo due cose: la prima che, nella chiamata, le variabili `n1`, `n2` vengono copiati nei parametri `a`, `b` della procedura (e questi parametri sono di tipo `IN OUT`), e quindi la procedura puó alterarne il valore, scambiandoli tra di loro. Dopo che la procedura é stata eseguita, i valori dei parametri `a`, `b` vengono ricopiati in `n1` ed `n2` cui valori sono stati cosí scambiati tra loro. La seconda é che per non possiamo eseguire direttamente una procedura in cui compare una lista di parametri (perché il loro valore non sarebbe noto alla procedura, e perché essa non potrebbe associarli a delle variabili); ad esempio digitando il comando `gnatbl scambia.ali` compare un messaggio di errore. Occorre, per provare la procedura, una seconda procedura che

funge da programma principale (main), in cui non compaiono parametri. Vogliamo ora scrivere un programma che legge da tastiera due numeri interi maggiori di zero, e che determini se sono primi tra loro (cioé se il loro m.c.d. vale 1 o no). Il programma deve controllare che i due numeri siano maggiori di zero, altrimenti deve richiederne di nuovo l'introduzione. Siccome questo controllo e l'eventuale reintroduzione deve essere ripetuta per i due numeri, conviene scrivere una procedura adatta allo scopo:

```
WITH Ada.text_io; USE Ada.text_io;
WITH Ada.integer_text_io; USE Ada.integer_text_io;

PROCEDURE get_intero_magg_zero (numero : OUT integer) IS
   -- questa procedura permette l'introduzione da tastiera di un numero
   -- intero; se tale numero e' minore o uguale a zero ne chiede la
   -- reintroduzione
BEGIN
   get(numero);
   WHILE numero <= 0 LOOP
      put("Il numero e' minore o uguale a zero: reintroducilo : ");
      get(numero);
   END LOOP;
END get_intero_magg_zero;
```

Osserviamo che l'intestazione della procedura contiene un parametro: numero, dichiarato di tipo `integer`, con la parola riservata `OUT`. Questa parola sta a significare che il valore del parametro numero NON é noto alla procedura al momento in cui viene chiamata, ma che la procedura puó modificare questo parametro e passarlo alla variabile corrispondente indicato nella sua chiamata. Scriviamo adesso una funzione che calcola il massimo comun divisore tra due numeri rappresentati dai suoi parametri `n1`, `n2`:

```
FUNCTION max_com_div (n1: IN integer; n2: IN integer) RETURN integer IS
   -- la parola riservata IN e' facoltativa

   minimo : integer;
   -- variabile locale usata per contenere il minimo tra n1 ed n2
   mcd : integer;
   -- variabile locale per contenere il massimo comun divisore

BEGIN
   -- calcolo del mcd
   -- vedi programma mcm_mcd del capitolo 4

   IF n1 < n2 THEN
      -- ricerca del minimo
      minimo := n1;
   ELSE
     minimo := n2;
   END IF;

   -- il ciclo seguente ricerca il mcd, calcolando i resti della
   -- divisione di n1 ed n2 per un numero compreso tra il minimo di n1
   -- ed n2 e 1, e puo' essere sostituito con un ciclo LOOP...END LOOP
   -- in quanto viene sicuramente eseguito almeno una volta.
   mcd := minimo;
   WHILE (n1 REM mcd /= 0) OR (n2 REM mcd /= 0) LOOP
      -- resta in ciclo se una delle due divisioni ha resto diverso
      -- da zero (l'operatore /= significa 'diverso da')
      mcd := mcd - 1;
```

```
      -- e decrementa mcd, per effettarre la divisione successiva
      -- (nel caso peggiore, di n1 ed n2 primi tra loro, esce dal
      -- ciclo quando mcd = 1)
   END LOOP;

   RETURN mcd;
   -- ritorna alla procedura chiamante il m.c.d.

END max_com_div;
```

Scriviamo ora, una funzione che ritorna un valore booleano (logico), vero o falso, a seconda che il massimo comun divisore tra i parametri `n` ed `m` sia 1 oppure diverso da 1. Questa funzione chiama la precedente (`max_com_div`) per calcolare il massimo comun divisore tra i due parametri `n` ed `m`, (che pertanto vengono copiati nei parametri `n1`, `n2` della funzione `max_com_div`).

```
WITH max_com_div;

FUNCTION sono_primi_tra_loro (n: IN integer; m: IN integer)
         RETURN boolean IS
BEGIN
   IF max_com_div(n, m) = 1 THEN
      RETURN TRUE;  -- i due numeri n, m sono primi tra loro
   ELSE
      RETURN FALSE; -- n ed m non sono primi tra loro
   END IF;
END sono_primi_tra_loro;
```

Scriviamo infine il programma:

```
WITH Ada.text_io; USE Ada.text_io;
WITH get_intero_magg_zero;
WITH sono_primi_tra_loro;

-- non serve Ada.integer_text_io perche' in questa procedura si
-- visualizzano solo testi, non si visualizzano e nemmeno si
-- introducono numeri interi (questo e' compito della procedura
-- get_intero_magg_zero)

PROCEDURE numeri_primi_tra_loro IS
   num1, num2 : integer;
   -- numeri introdotti da tastiera
BEGIN
   -- introduzione dei dati
   put("Introduci il primo numero : ");
   get_intero_magg_zero(num1);
      -- introduzione di num1, ripetuta se non e' maggiore di 0
   put("Introduci il secondo numero : ");
   get_intero_magg_zero(num2);
      -- introduzione di num2, come per num1

   -- controlla se sono primi tra loro e visualizza un
   -- opportuno messaggio
   IF sono_primi_tra_loro(num1, num2) THEN
      put_line("I due numeri sono primi tra loro");
   ELSE
      put_line("I due numeri non sono primi tra loro");
   END IF;
END numeri_primi_tra_loro;
```

Compiliamo, un file alla volta, ciascuno contenente una procedura o una funzione, e poi linkiamo il tutto con `gnatbl numeri_primi_tra_loro`; possiamo cosí eseguire il programma, come al solito. Siccome la variabile (`num1` o `num2`) passata dal programma principale (main) alla procedura che si occupa dell'introduzione del dato non é noto al momento della chiamata della procedura stessa, possiamo in alternativa utilizzare una funzione,senza parametri, che ritorna alla procedura chiamante il dato introdotto. Scriviamo dunque questa funzione, e salviamola con il nome `input_int_magg_zero.adb` La scelta é lasciata al programmatore.

```
WITH Ada.text_io; USE Ada.text_io;
WITH Ada.integer_text_io; USE Ada.integer_text_io;

FUNCTION input_intero_magg_zero RETURN integer IS
   -- questa funzione permette l'introduzione da tastiera di un numero
   -- intero; se tale numero e' minore o uguale a zero ne chiede la
   -- reintroduzione

   numero : integer;
   -- variabile locale per memorizzare il dato introdotto
BEGIN
   get(numero);
   WHILE numero <= 0 LOOP
      put("Il numero e' minore o uguale a zero: reintroducilo : ");
      get(numero);
   END LOOP;
   RETURN numero;
END input_intero_magg_zero;
```

Possiamo ora riscrivere il programma principale, utilizzando questa funzione per l'introduzione dei due numeri interi:

```
WITH Ada.text_io; USE Ada.text_io;
WITH input_intero_magg_zero;
WITH sono_primi_tra_loro;

-- non serve Ada.integer_text_io perche' in questa procedura si
-- visualizzano solo testi, non si visualizzano e nemmeno si
-- introducono numeri interi (questo e' compito della procedura
-- get_intero_magg_zero)

PROCEDURE numeri_primi_fra_loro IS
   num1, num2 : integer;
   -- numeri introdotti da tastiera
BEGIN
   -- introduzione dei dati
   put("Introduci il primo numero : ");
   num1 := input_intero_magg_zero;
      -- introduzione di num1, ripetuta se non e' maggiore di 0
      -- avendo a che fare con una funzione, si ha un assegnamento
   put("Introduci il secondo numero : ");
   num2 := input_intero_magg_zero;
      -- introduzione di num2, come per num1

   -- controlla se sono primi tra loro e visualizza un
   -- opportuno messaggio
   IF sono_primi_tra_loro(num1, num2) THEN
      put_line("I due numeri sono primi tra loro");
   ELSE
      put_line("I due numeri non sono primi tra loro");
```

```
   END IF;
END numeri_primi_fra_loro;
```

7. Tabelle e matrici

Consideriamo un problema pratico: vogliamo scrivere una procedura che sia in grado di leggere e di ordinare una tabella di 10 numeri interi, in ordine crescente. Possiamo pensare di dichiarare 10 variabili di tipo `integer` nel programma principale. Ma non vi paiono un po' troppe dieci variabili intere? E se poi i numeri da ordinare fossero cinquanta o cento o ancor di piú? Per questo motivo, possiamo introdurre il concetto di TABELLA (`ARRAY` in inglese ed in Ada); con la dichiarazione:

```
TYPE tabella_di_interi IS ARRAY (1..10) OF integer;
numeri : tabella_di interi;
```

possiamo dichiarare un nuovo TIPO di variabile (una tabella di 10 numeri interi), e successivamente utilizzare tale tipo per allocare spazio in memoria per la variabile numeri, capace di contenere 10 numeri interi. Ogni numero della tabella é individuato da un indice, che va dal valore minimo 1 al valore massimo 10; cosí `numeri(1)` é il primo numero della tabella, `numeri(2)` il secondo, ..., `numeri(10)` il decimo. Ecco allora come si presenta il programma:

```
WITH Ada.text_io; USE Ada.text_io;
WITH Ada.integer_text_io; USE Ada.integer_text_io;

PROCEDURE ordinamento_numeri IS
   dimensione : CONSTANT integer := 10;
   -- numero di elementi della tabella
   TYPE tabella_di_interi IS ARRAY (1..dimensione) OF integer;
   -- dichiarazione di un tipo di tabella capace di
   -- contenere 10 numeri interi
   numeri: tabella_di_interi;
   -- dichiarazione della variabile che conterra' i 10 numeri interi
   -- introdotti da tastiera
   num_ordinati: tabella_di_interi;
   -- tabella che conterra' gli stessi numeri, ma in ordine crescente
   minimo, massimo: integer;
   -- minimo e massimo tra tutti i numeri introdotti
   indice_ord: integer;
   -- indice per individuare la posizione in cui introdurre i numeri
   -- nella tabella num_ordinati durante la fase di ordinamento
BEGIN
   -- introduzione dei numeri interi
   -- viene utilizzato a tal scopo un ciclo definito

   FOR i IN 1..dimensione LOOP
      put("Introduci l'intero di posto ");
      put(i, WIDTH => 2);
      put(":");
      get(numeri(i));
      -- notare l'indice i che individua il posto nella tabella
      -- dove collocare il numero letto da tastiera
   END LOOP;

   -- ordinamento dei numeri: innanzitutto vengono cercati il minimo
   -- ed il massimo dei numeri introdotti

   minimo := numeri(1);
```

```
   massimo := numeri(1);
   -- si suppone innanzitutto che il minimo ed il massimo coincidano
   -- con il primo elemento della tabella
   FOR i IN 2..dimensione LOOP
      -- poi con un ciclo definito si confrontano il minimo ed il
      -- massimo con il secondo, terzo, ..., decimo elemento della
      -- tabella; se si trova un numero maggiore di massimo lo si
      -- prende come nuovo massimo, e se si trova un numero piu'
      -- piccolo del minimo, lo si prende come nuovo minimo
      IF numeri(i) > massimo THEN
         massimo := numeri(i);
      END IF;
      IF numeri(i) < minimo THEN
         minimo := numeri(i);
      END IF;
   END LOOP;

   -- a questo punto, si riempie la tabella num_ordinati con un
   -- ciclo, che incomincia a controllare quali numeri coincidono
   -- con il numero da introdurre (inizialmente uguale al minimo),
   -- per introdurli nella tabella num_ordinati. Poi incremenda di
   -- uno il numero da introdurre, e inserisce (se esistono) i numeri
   -- che hanno tale valore nella tabella. E cosi' via, sino al massimo.

   indice_ord := 1;
   -- all'inizio introdurremo nella tabella num_ordinati tutti i
   -- numeri che coincidono con il minimo
   -- partendo dalla posizione 1 della tabella num_ordinati

   -- il ciclo esterno serve per scorrere tutti gli interi tra il
   -- minimo e il massimo
   ciclo_min_max:
   FOR numero_da_cercare IN minimo..massimo LOOP

      -- il ciclo interno scorre la tabella contenente i numeri
      -- introdotti da tastiera; quando trova un numero coincidente
      -- con numero_da_introdurre lo copia nella tabella num_ordinati
      -- e incrementa la posizione in cui copiare il numero successivo
      ciclo_scorri_tabella:
      FOR i IN 1..dimensione LOOP
         IF numeri(i) = numero_da_cercare THEN
            num_ordinati(indice_ord) := numero_da_cercare;
            indice_ord := indice_ord + 1;
         END IF;
      END LOOP ciclo_scorri_tabella;
   END LOOP ciclo_min_max;
  -- visualizzazione dei numeri ordinati, con un ciclo definito

   FOR i IN 1..dimensione LOOP
      put(num_ordinati(i), WIDTH => 4);
      new_line;
   END LOOP;

END ordinamento_numeri;
```

E' importante notare che esistono metodi piú rapidi per l'ordinamento; quello che abbiamo presentato é assai lento, e quindi inefficiente per ordinare grandi quantitá di dati. Sui metodi di ordinamento torneremo in un'altro capitolo. Le tabelle

possono essere confrontate tra di loro utilizzando gli operatori `=` (uguale) e `/=` (diverso); due tabelle sono uguali se hanno le stesse dimensioni e se il loro contenuto é indentico, altrimenti sono diverse.

8. Tabelle multidimensionali

Riprendiamo in esame l'uso delle tabelle, con un programma che consente la numerazione di un cruciverba. L'utente introduce da tastiera il numero di righe ed il numero di colonne che compongono il cruciverba e la posizione delle caselle nere. Il programma provvede automaticamente alla numerazione ed alla generazione dell'elenco numerato delle definizioni orizzontali e verticali. Cominciamo, innanzitutto, con una osservazione: supponiamo di voler fissare una dimensione *massima* del cruciverba, ad esempio di 6 righe ed 8 colonne. Possiamo allora pensare ad una tabella bidimensionale, che puó essere dichiarata con il seguente frammento di programma:

```
TYPE cruciv IS ARRAY (1..6, 1..8) OF integer;
cruciverba : cruciv;
```

Abbiamo creato un nuovo *tipo* di variabile (una tabella 6x8 di interi), di nome `cruciv`, ed abbiamo dichiarato una nuova variabile, `cruciverba`, di quel tipo. Da questa definizione appare evidente che il primo indice dell'array `1..6` individua le righe del cruciverba, il secondo indice `1..8` le sue colonne. Ogni elemento della tabella rappresenta una casella del cruciverba. Ad esempio `cruciverba(2,4)` individua la casella posta sulla seconda riga e sulla quarta colonna. Quando definiamo il cruciverba, possiamo riempire la tabella con

- Il numero 0, per indicare una casella bianca
- Il numero -1, per indicare una casella nera

Ad esempio, la tabella seguente rappresenta il cruciverba (non ancora numerato) di figura 1:

$$\begin{bmatrix} 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$

Osserviamo che, avendo utilizzato il -1 per indicare le caselle nere (anziché il numero 1), possiamo utilizzare i numeri positivi per indicare le caselle bianche numerate.

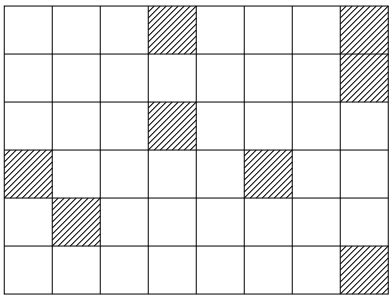

Figura 1. Cruciverba non numerato

A numerazione terminata (vedi figura 2), la tabella apparirá nel modo seguente:

$$\begin{bmatrix} 1 & 2 & 3 & -1 & 4 & 5 & 6 & -1 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 8 & 0 & 0 & -1 & 9 & 0 & 0 & 10 \\ -1 & 11 & 0 & 12 & 0 & -1 & 13 & 0 \\ 14 & -1 & 15 & 0 & 0 & 16 & 0 & 0 \\ 17 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$

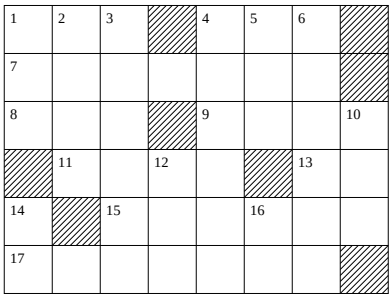

FIGURA 2. Cruciverba numerato

Passiamo ora a scrivere il programma, dividendolo in tante piccole procedure, ciascuna delle quali si occuperá di un compito particolare (inizializzare il cruciverba, inserire le dimensioni, inserire le caselle nere e controllarne la corretta posizione, procedere alla numerazione, visualizzare il risultato). La procedura principale, `parole_crociate.adb` contiene la dichiarazione delle variabili che servono a contenere il cruciverba, le sue dimensioni, e l'elenco delle definizioni orizzontali e verticali. Questa procedura chiamerá poi altre procedure per svolgere i singoli compiti sopra indicati. Prima di scrivere il programma, osserviamo alcune cose:

- Le dimensioni massime del cruciverba, sono le stesse per tutte le procedure
- Le dimensioni dell'elenco delle definizioni, sono le stesse per tutte le procedure
- Tutte le procedure condividono le costanti che indicano il numero massimo di definizioni, il simbolo che indica una casella bianca e quello che indica una casella nera

Tutte queste sono *risorse comuni* che NON serve dichiarare di volta in volta in ciascuna procedura, ma che possono essere poste in un PACCHETTO comune (detto `PACKAGE`) condiviso da tutte le procedure dichiarando questo pacchetto con le istruzioni`WITH` ed `USE`. Notiamo come, a differenza delle procedure chiamate (che appaiono solo al fianco di `WITH`), i `PACKAGE` devono comparire anche a fianco di `USE`. Ció é dovuto al fatto che questi ultimi mettono a disposizione delle *risorse* (le procedure e le funzioni svolgono un compito, ma non mettono a disposizione delle risorse comuni). Osserviamo inoltre che i `PACKAGE` devono essere salvati con estensione `.ads` (e NON `.adb`), e che vanno normalmente compilati, come le procedure. Scriviamo dunque il nostro pacchetto di risorse:

```
-- Package contenente tutte le definizioni di costanti e dei tipi di
-- variabile utilizzati nel programma per la numerazione automatica del
-- criciverba. Deve essere salvato come definizioni.ads e compilato
-- con il comando gcc -c definizioni.ads
-- Includendo delle RISORSE (le definizioni di costanti e di tipi) va
-- dichiarato nelle liste WITH e USE di tutte le procedure e funzioni
-- che utilizzano queste risorse.
```

```
PACKAGE definizioni IS
   max_row : CONSTANT positive := 6;
   max_col : CONSTANT positive := 8;
     -- massimo numero di righe e di colonne
     -- il tipo positive e' un intero maggiore o uguale ad 1
     -- dato che le dimensioni minime sono 2x1 o 1x2 e' piu'
     -- espressivo usarlo rispetto al tipo integer

   TYPE cruciv IS ARRAY (1..max_row, 1..max_col) OF integer;
     -- definizione del tipo cruciverba come tabella bidimensionale

   max_orizz : CONSTANT positive := ((max_col + 1) / 3) * max_row;
   max_vert : CONSTANT positive := ((max_row + 1) / 3) * max_col;
     -- numero massimo di definizioni orizzontali e verticali

   TYPE defin_or IS ARRAY (1..max_orizz) OF natural;
   TYPE defin_vert IS ARRAY (1..max_vert) OF natural;
     -- tipi di tabelle contenenti le numerazioni orizzontali e verticali
     -- il numero zero indica l'assenza di definizioni
     -- il tipo natural e' un intero maggiore o uguale a zero.
     -- dato che nella tabella non si usano numeri negativi e' piu'
     -- espressivo usarlo rispetto al tipo integer
   max_def_video : CONSTANT positive := 10;
     -- numero massimo di numeri di definizioni orizzontali e verticali
     -- visualizzate su una riga video senza andare a capo

   casella_bianca : CONSTANT integer := 0;
   casella_nera : CONSTANT integer := -1;
     -- rappresentazione caselle bianche e nere nella tabella
END definizioni;
```

Salviamolo con il nome `definizioni.ads` e compiliamolo con il comando `gcc -c definizioni.ads`. In questo pacchetto sono stati introdotti due nuovi tipi, `natural` e `positive`, rispettivamente utilizzati per rappresentare numeri interi naturali (zero compreso) e positivi (cioé maggiori o uguali ad uno). Piú precisamente, il tipo `natural` comprende l'insieme dei numeri non negativi, ovvero l'insieme $\{0, 1, 2, 3, \dots\}$, mentre il tipo `positive` é utilizzato per rappresentare l'insieme dei numeri maggiori di zero, ovvero per rappresentare l'insieme $\{1, 2, 3, ...\}$. Se analizziamo il package, osserviamo che il calcolo del numero massimo di definizioni orizzontali e verticali é stato calcolato in funzione delle dimensioni massime del cruciverba. Cio' perche' nel caso limite in cui tutte le definizioni orizzontali del cruciverba siano di due lettere, occorrono due lettere ed una casella nera per rappresentare una definizione e la separazione dalla definizione successiva. Pensando al margine destro del cruciverba come ad una casella nera, segue che su ciascuna riga del cruciverba ci stanno `((max_col + 1) / 3)` definizioni. Il numero totale di definizioni si ottiene moltiplicando tale numero per il numero di righe che compongono il cruciverba. La figura 3 mette in evidenta questo caso estremo. Una considerazione analoga vale per il numero massimo di definizioni verticali.

Passiamo ora all'esame della procedura principale, che chiama a sua volta le altre procedure che si occupano dei vari compiti sopra citati.

```
WITH Ada.text_io, Ada.integer_text_io, definizioni,
     dimensioni, inizializza, caselle_nere, numera,
     visualizza_cruciverba, visualizza_definizioni;
USE Ada.text_io, Ada.integer_text_io, definizioni;

PROCEDURE parole_crociate IS
```

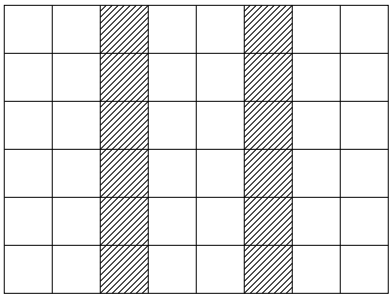

FIGURA 3. Cruciverba con numero massimo di definizioni orizzontali

```
   cruciverba : cruciv;
       -- definizione del cruciverba come tabella bidimensionale
   righe : positive;
   colonne : positive;
       -- dimensioni del cruciverba
   def_or : defin_or := (others => 0);
   def_vert : defin_vert := (others => 0);
       -- tabelle contenenti le numerazioni orizzontali e verticali
       -- l'istruzione others => 0 inizializza a zero tutte le componenti
       -- delle tabelle senza utilizzare cicli (confrontare con la
       -- procedura inizializza)
   successo : boolean;
       -- questa variabile viene passata alle procedure di introduzione
       -- dei dati: se ci sono problemi (EXCEPTION) nella loro
       -- introduzione le procedure chiamate gli assegnano il valore
       -- FALSE, per consentire la reintroduzione dei dati stessi
BEGIN
   LOOP
      dimensioni(righe, colonne, successo);
         -- introduzione dimensioni del cruciverba
      EXIT WHEN successo;
         -- esce se i dati sono stati introdotti correttamente
      put("Errore nell'introduzione delle dimensioni.");
      put_line("Prego reintrodurle");
         -- altrimenti visualizza un messaggio di errore
   END LOOP;

   LOOP
      inizializza(cruciverba, righe, colonne);
          -- inizializza con caselle vuote (bianche) il cruciverba
      caselle_nere(cruciverba, righe, colonne, successo);
          -- introduzione delle caselle nere
      EXIT WHEN successo;
      put("Errore nell'introduzione delle caselle nere.");
      put_line("Prego reintrodurle");
      -- se l'introduzione ha avuto un qualsiasi insuccesso,
      -- chiede di ripeterla dopo aver inizializzato di nuovo
      -- il cruciverba
   END LOOP;

   put_line("Cruciverba non numerato");
   visualizza_cruciverba(cruciverba, righe, colonne);
      -- visualizzazione del cruciverba non numerato
```

```
   numera(cruciverba, righe, colonne, def_or, def_vert);
      -- numerazione del cruciverba e generazione delle definizioni

   put_line("Cruciverba numerato");
   visualizza_cruciverba(cruciverba, righe, colonne);
   visualizza_definizioni(def_or, def_vert);
      -- visualizzazione del cruciverba numerato e dell'elenco
      -- delle definizioni

END parole_crociate;
```

Questa procedura, chiede innanzitutto le dimensioni del cruciverba, che saranno introdotte mediante la chiamata della procedura `dimensioni`, la quale controlla anche che i dati introdotti siano corretti. Nel caso in cui nell'introduzione dei dati si verifici una eccezione (`EXCEPTION`), la procedura `dimensioni` restituirá il valore `FALSE` al parametro `successo`, e questo consentirá di ripetere l'introduzione senza arrestare il programma. Ecco la procedura `dimensioni`:

```
WITH Ada.text_io, Ada.integer_text_io, definizioni;
USE Ada.text_io, Ada.integer_text_io, definizioni;

PROCEDURE dimensioni(rows : IN OUT positive;
                     columns : IN OUT positive;
                     success : OUT boolean) IS

-- la procedura chiede le dimensioni del cruciverba, controlla che siano
-- comprese tra quelle minime e massime ammesse, e nel caso di insuccesso
-- di introduzione dati assegna il valore FALSE a success per permettere
-- alla procedura chiamante di ripetere l'introduzione dei dati, mentre
-- assegna al parametro success il valore TRUE in caso di successo

BEGIN

  ciclo_controllo_dim_minime:
  LOOP
     -- questo ciclo piu' esterno serve per controllare che il cruciverba
     -- abbia una dimensione minima di 1 riga e 2 colonne, o di 1 colonna
     -- e 2 righe (cioe' sia costituito da una sola definizione)

     ciclo_introduzione_righe:
     LOOP
        -- introduzione del numero di righe controllando che non superi
        -- il numero massimo ammesso
        put("Introduci il numero di righe, minore di ");
        put(max_row, WIDTH => 2);
        put(" : ");
        get(rows);
        EXIT WHEN rows <= max_row;
        put_line("Dato errato");
     END LOOP ciclo_introduzione_righe;

     ciclo_introduzione_colonne:
     LOOP
        -- introduzione del numero di colonne controllando che non
        -- superi il numero massimo ammesso
        put("Introduci il numero di colonne, minore di ");
        put(max_col, WIDTH => 2);
```

```
        put(" : ");
        get(columns);
        EXIT WHEN columns <= max_col;
        put_line("Dato errato");
     END LOOP ciclo_introduzione_colonne;

     EXIT WHEN (rows + columns >= 3);
     -- esce se il cruciverba ha dimensioni minime 1x2 o 2x1

     put("Il cruciverba deve essere almeno 1x2 o 2x1");
     -- in caso contrario visualizza un messaggio di errore

  END LOOP ciclo_controllo_dim_minime;

  success := TRUE;
  -- introduzione effettuata con successo

  EXCEPTION
     -- introduzione errata
     WHEN others => skip_line;
                    success := FALSE;
     -- notare l'uso di others per TUTTE le eccezioni
     -- (non solo constraint_error) e l'uso di skip_line per
     -- eliminare il dato introdotto erroneamente

END dimensioni;
```

Un'altra procedura utilizzata, di nome `inizializza`, permette di riempire il cruciverba con tutte caselle bianche (inizializzazione del cruciverba, prima dell'inserimento delle caselle nere). Eccola:

```
WITH definizioni; USE definizioni;

PROCEDURE inizializza (cross : IN OUT cruciv;  -- cruciverba
                       rg : IN positive;       -- numero di righe
                       cl : IN positive) IS    -- e di colonne

-- questa procedura inizializza il cruciverba riempiendolo
-- di caselle bianche

BEGIN
   riempi_righe:
   FOR riga IN 1..rg LOOP
      riempi_colonne:
      FOR colonna IN 1..cl LOOP
          cross(riga, colonna) := casella_bianca;
      END LOOP riempi_colonne;
   END LOOP riempi_righe;
END inizializza;
```

La procedura `caselle_nere` permette invece di introdurre da tastiera il numero di caselle nere e la loro posizione; assegnando anch'essa il valore `FALSE` al parametro `successo` in caso di una eccezione nell'introduzione dei dati, permette di poterli reintrodurre senza fermare il programma. Eccola:

```
WITH Ada.text_io, Ada.integer_text_io, definizioni;
USE Ada.text_io, Ada.integer_text_io, definizioni;

PROCEDURE caselle_nere (crosswords: IN OUT cruciv;
                                    -- cruciverba
```

```
                        r: IN positive;
                                   -- numero righe
                        c: IN positive;
                                   -- numero colonne
                        success: OUT boolean) IS
                                   -- come per dimensioni.adb
   numero_nere : natural;
               -- numero di caselle nere
   riga, colonna : positive;
               -- riga e colonna in cui si vuol piazzare la casella nera
BEGIN
   -- ciclo di introduzione del numero di caselle nere
   -- questa versione della procedura non controlla la posizione delle
   -- caselle nere introdotte in modo da evitare che nel cruciverba
   -- appaiano zone isolate, o non sia possibile inserire alcuna
   -- definizione
   LOOP
      put("Numero di caselle nere ");
      put("Minimo = 0, Massimo = "); put(r*c - 2);
      put(" ? ");
      get(numero_nere);
      EXIT WHEN numero_nere >= 0 AND numero_nere <= r*c - 2;
      -- esce se il numero introdotto e' corretto.
      put_line("Numero di caselle nere errato: reintroducilo");
      -- altrimenti visualizza un messaggio di errore
   END LOOP;

   -- ciclo di introduzione delle caselle nere
   ciclo_introduzione_nere:
   WHILE numero_nere > 0 LOOP

      ciclo_introduzione_riga:
      LOOP
         put("Introduci la riga della casella nera : ");
         get(riga);
         EXIT WHEN riga <= max_row;
         -- controllo che la riga non esca dalle dimensioni
         -- del cruciverba
         put("Valore di riga errato: reintroducilo");
      END LOOP ciclo_introduzione_riga;

      ciclo_introduzione_colonna:
      LOOP
         put("Introduci la colonna della casella nera : ");
         get(colonna);
         EXIT WHEN colonna <= max_col;
         -- controllo che la colonna non esca dalle dimensioni
         -- del cruciverba
         put("Valore di colonna errato: reintroducilo");
      END LOOP ciclo_introduzione_colonna;

      IF crosswords(riga,colonna) = casella_bianca THEN
         crosswords(riga,colonna) := casella_nera;
         numero_nere := numero_nere - 1;
         -- se la casella non e' gia' nera, la annerisce e decrementa
         -- l numero di caselle nere ancora da inserire
      ELSE
```

```
         -- la casella e' gia' nera: non la annerisce e lascia invariato
         -- il numero di caselle nere ancora da inserire per
         -- permettere di reintrodurre il dato
         put("La casella e' gia' nera: reintroduci le coordinate");
         new_line;
      END IF;
   END LOOP ciclo_introduzione_nere;

   -- introduzione effettuata con successo
   success := TRUE;

   EXCEPTION
   -- errore nell'introduzione dei dati
      WHEN others => skip_line;
                     success := FALSE;

END caselle_nere;
```

La procedura `visualizza_cruciverba` viene invece chiamata due volte: la prima per visualizzare il cruciverba NON numerato (una tabella di 0 e -1) e poi per visualizzarle il cruciverba numerato. Eccola:

```
WITH Ada.text_io, Ada.integer_text_io, definizioni;
USE Ada.text_io, Ada.integer_text_io, definizioni;

PROCEDURE visualizza_cruciverba (crosswd : IN cruciv;
                                           -- cruciverba
                                 r : IN positive;
                                           -- numero di righe
                                 c : IN positive) IS
                                           -- numero di colonne

-- visualizzazione del cruciverba

BEGIN
   visualizza_riga:
   FOR riga IN 1..r LOOP
      visualizza_colonna:
      FOR colonna IN 1..c LOOP
         put(crosswd(riga, colonna), WIDTH => 3);
      END LOOP visualizza_colonna;
      new_line (SPACING => 2);
      -- il parametro SPACING indica il numero di righe da saltare,
      -- e se viene omesso allora new_line si limita ad andare a capo
   END LOOP visualizza_riga;
END visualizza_cruciverba;
```

Analogamente, la procedura `visualizza_definizioni` permette di visualizzare l'elenco numerico delle definizioni orizzontali e verticali. Eccola:

```
WITH Ada.text_io, Ada.integer_text_io, definizioni;
USE Ada.text_io, Ada.integer_text_io, definizioni;

PROCEDURE visualizza_definizioni (d_or : IN defin_or;
                                       -- tabella definizioni orizzontali
                                  d_vt : IN defin_vert) IS
                                       -- tabella definizioni verticali
BEGIN

   -- visualizza l'elenco delle definizioni orizzontali, per un totale
```

```
   -- di max_def_video per riga
   put_line("Elenco numerazione delle definizioni orizzontali");
   FOR i IN 1..max_orizz LOOP
      EXIT WHEN d_or(i) = 0;
         -- esce se sono terminate le definizioni
      put(d_or(i), WIDTH => 3);
         -- visualizza il numero della la definizione
      IF i MOD max_def_video = 0 THEN
         -- va a capo ogno max_def_video definizioni visualizzate
         new_line;
      END IF;
   END LOOP;

   new_line;

   -- visualizza l'elenco delle definizioni verticali, per un totale
   -- di max_def_video per riga
   put_line("Elenco numerazione delle definizioni verticali");
   FOR i IN 1..max_vert LOOP
      EXIT WHEN d_vt(i) = 0;
          -- esce se sono terminate le definizioni
      put(d_vt(i), WIDTH => 3);
         -- visualizza il numero della definizione
      IF i MOD max_def_video = 0 THEN
         -- va a capo ogno max_def_video definizioni visualizzate
         new_line;
      END IF;
   END LOOP;

END visualizza_definizioni;
```

Infine, analizziamo la procedura piú interessante, quella che si occupa della numerazione. Per poter dire che una casella deve essere numerata perché in essa inizia una definizione orizzontale, occorre innanzitutto che essa non deve trovarsi sull'ultima colonna. Inoltre, se essa si trova sulla prima colonna, deve essere bianca ed avere alla sua sinistra un'altra casella bianca. Se invece la casella si trova in una colonna diversa dalla prima, deve altresí avere a destra una casella nera, Analoghe considerazioni valgono per le definizioni verticali. La procedura analizza casella per casella, con un doppio ciclo `FOR`, e se la casella é numerabile (orizzontalmente e/o verticalmente) pone a `TRUE` rispettivamente le variabili booleane `numerata_orizz` e/o `numerata_vert`, per procedere poi alla numerazione della casella, all'inserimento del numero della definizione nell'elenco, e all'incremento del contatore delle definizioni.

```
WITH definizioni;
USE definizioni;

PROCEDURE numera (cruc : IN OUT cruciv;
                             -- cruciverba
                  n_righe : IN positive;
                             -- numero di righe del cruciverba
                  n_colon : IN positive;
                             -- numero di colonne del cruciverba
                  defi_or : IN OUT defin_or;
                             -- elenco definizioni orizzontali
                  defi_vt : IN OUT defin_vert) IS
                             -- elenco definizioni verticali
```

```
  ir : positive := 1;
  ic : positive := 1;
      -- variabili locali usate come indici delle tabelle
      -- dell'elencon delle definizioni
  numero_definizione : positive := 1;
      -- progressivo per numerazione delle definizioni
  numerata_orizz, numerata_vert : boolean;
      -- vengono poste a TRUE quando una definizione e' stata numerata
      -- rispettivamente in orizzontale o in verticale
BEGIN
  -- la numerazione viene effettuata analizzando casella per casella con
  -- un doppio ciclo FOR. Una casella e' l'inizio di una definizione
  -- orizzontale se si trova sulla prima colonna e ha sulla sua sinistra
  -- una casella bianca, oppure se si trova all'interno del cruciverba
  -- ed ha una casella nera alla sua destra ed una bianca alla sua sinistra
  -- Ovviamente, le caselle nell'ultima colonna non potranno mai costituire
  -- delle caselle numerate per le definizioni orizzontali. Analoghe
  -- considerazioni valgono per le definizioni verticali, se si scambia la
  -- parola riga con colonna e destra/sinistra con sopra/sotto.
  -- La variabile locale numero_definizione serve per numerare tutte le
  -- definizioni, in ordine progressivo, e tiene conto delle caselle gia'
  -- numerate. Le variabili ir ed ic servono invece come indici delle
  -- tabelle delle definizioni, per inserirvi dentro i numeri delle
  -- definizioni

  ciclo_sulle_righe:
  FOR r IN 1..n_righe LOOP

    ciclo_sulle_colonne:
    FOR c IN 1..n_colon LOOP
       numerata_orizz := FALSE;
       numerata_vert := FALSE;
       -- la casella e' inizialmente non numerata

       IF c = 1 THEN
           -- controlla se la casella nella prima colonna e' numerabile
           -- come orizzontale, e se non lo e', controlla se e' numerabile
           -- come verticale

           IF r = 1 THEN
              -- controlla la numerabilita' sulla riga 1 (orizzontale e
              -- verticale
              IF cruc(r,c)=casella_bianca AND cruc(r,c+1)=casella_bianca
                 -- la casella puo' essere numerata in orizzontale
                 THEN numerata_orizz := TRUE;
              END IF;
              IF cruc(r,c)=casella_bianca AND cruc(r+1,c) = casella_bianca
                 -- come sopra, ma in verticale
                 THEN numerata_vert := TRUE;
              END IF;
           ELSIF r =  n_righe THEN
              -- controlla la numerabilita' (solo orizzontale) di una
              -- casella posta sull'ultima riga
              IF cruc(r,c)=casella_bianca AND cruc(r,c+1) = casella_bianca
                 THEN numerata_orizz := TRUE;
              END IF;
           ELSE
```

```
      -- controlla la numerabilita' (orizzontale e verticale) su una
      -- riga interna
      IF cruc(r,c)=casella_bianca AND cruc(r,c+1)=casella_bianca
         THEN numerata_orizz := TRUE;
      END IF;
      IF cruc(r-1,c)=casella_nera AND cruc(r, c)=casella_bianca
         AND cruc(r+1,c)=casella_bianca
         THEN numerata_vert := TRUE;
      END IF;
   END IF;

ELSIF c =  n_colon THEN
    -- l'ultima casella di una riga non e' numerabile come
    -- orizzontale, controlla solo se lo e' come verticale
    IF r = 1 THEN
      -- controlla la numerabilita' verticale della casella posta
      -- sulla prima riga e sull'ultima colonna
      IF cruc(r,c)=casella_bianca AND cruc(r+1,c)=casella_bianca
         THEN numerata_vert := TRUE;
      END IF;
    ELSIF r < n_righe THEN
       -- controlla la numerabilita' verticale di una casella posta
       -- su una riga interna e sull'ultima colonna
       IF cruc(r-1,c)=casella_nera AND cruc(r, c)=casella_bianca
          AND cruc(r+1,c)=casella_bianca
          THEN numerata_vert := TRUE;
       END IF;
    END IF;

ELSE
   -- tutte le altre caselle sono potenzialmente numerabili sia come
   -- orizzontali che come verticali
   IF r = 1 THEN
     -- controllo numerabilita' orizzontale e verticale di una
     -- casella posta su una colonna interna e sulla prima riga
     IF cruc(r,c-1)=casella_nera AND cruc(r, c)=casella_bianca
        AND cruc(r,c+1)=casella_bianca
        THEN numerata_orizz := TRUE;
     END IF;
     IF cruc(r,c)=casella_bianca AND cruc(r+1,c)=casella_bianca
        THEN numerata_vert := TRUE;
     END IF;
   ELSIF r =  n_righe THEN
      -- controllo numerabilita' (solo orizzontale) di una casella
      -- posta su una colonna interna e sull'ultima riga
      IF cruc(r,c-1)=casella_nera AND cruc(r,c)=casella_bianca
         AND cruc(r,c+1)=casella_bianca
         THEN numerata_orizz := TRUE;
      END IF;
   ELSE
      -- controllo numerabilita' orizzontale di una casella posta
      -- su una riga interna e su una colonna interna
      IF cruc(r,c-1)=casella_nera AND cruc(r,c)=casella_bianca
         AND cruc(r,c+1)=casella_bianca
         THEN numerata_orizz := TRUE;
      END IF;
      IF cruc(r-1,c)=casella_nera AND cruc(r,c)=casella_bianca
```

```
                AND cruc(r+1,c)=casella_bianca
                THEN numerata_vert := TRUE;
             END IF;
          END IF;

       END IF;

       IF numerata_orizz THEN
          defi_or(ir) := numero_definizione;
          ir := ir + 1;
          -- se la casella e' numerabile orizzontalmente inserisce il numero
          -- della definizione nell'elenco e incrementa l'indice ir relativo
          -- alla tabella delle definizioni orizzontali
       END IF;
       IF numerata_vert THEN
          defi_vt(ic) := numero_definizione;
          ic := ic + 1;
          -- come sopra, per una definizione verticale
       END IF;

       IF numerata_orizz OR numerata_vert THEN
          -- se la casella e' numerabile ne inserisce il numero nella
          -- tabella cruc e poi incrementa il numero progressivo delle
          -- definizioni
          cruc(r, c) := numero_definizione;
          numero_definizione := numero_definizione + 1;
       END IF;

    END LOOP ciclo_sulle_colonne;
  END LOOP ciclo_sulle_righe;

END numera;
```

Possiamo ora provare il nostro programma, compilando dapprima il `PACKAGE definizioni.ads`, poi compilando le procedure chiamate, ed infine la procedura chiamante. Poi basta collegare il tutto con il comando `gnatbl`. Ecco l'elenco dei comandi:

```
gcc -c definizioni.ads
gcc -c dimensioni.adb
gcc -c inizializza.adb
gcc -c caselle_nere.adb
gcc -c numera.adb
gcc -c visualizza_cruciverba.adb
gcc -c visualizza_definizioni.adb
gcc -c parole_crociate.adb
gnatbl parole_crociate.ali
./parole_crociate
```

Se non vogliamo digitare tutti questi comandi, possiamo utilizzare il comando `gnatmake` che si occupa di compilare tutte le procedure e tutte le funzioni del programma, e di collegarle insieme. Questo comando é particolarmente utile quanto le procedure sono gia' state compilate senza errori una prima volta, e sono state operate solo delle piccole modifiche al programma. Non é consigliabile se si prova a compilare il programma per la prima volta; l'elenco degli errori visualizzato potrebbe infatti essere lunghissimo, dato che riguarda gli errori presenti in ciascuna procedura. Possiamo dunque in alternativa digitare il comando:

```
gnatmake parole_crociate.adb
```

in alternativa all'elenco sopra citato, e poi provare il programma, come al solito, digitando `./parole_crociate`. Notiamo come la procedura indicata a fianco del comando `gnatmake` é quella principale (`parole_crociate.adb`), ovvero quella che chiama tutte le altre. Osserviamo che questo programma presenta la limitazione che il numero massimo di righe e di colonne della tabella é giá fissato nella dichiarazione della tabella stessa. In un'altro capitolo, impareremo l'uso delle *tabelle non vincolate* per superare questa limitazione. Dato lo scopo introduttivo di questo capitolo, ci basta per ora aver imparato i primi rudimenti sull'uso delle tabelle, che ci saranno utili nei capitoli successivi.

9. Tipi enumerativi

Per arricchire maggiormente le nostre conoscenze di Ada95, e' giunto il momento di illustrare la definizione e l'utilizzo di un nuovo tipo di variabile, detto *tipo enumerativo*. Questo tipo puó essere pensato come una dichiarazione di un insieme finito di oggetti. Ad esempio, dichiarando:

```
TYPE stato_civile IS (libero, sposato, divorziato, vedovo);
```

abbiamo creato un nuovo *tipo* di oggetto che puó assumere tutti e soli i valori indicati nella lista. Giá da tempo utilizziamo un tipo enumerativo *predefinito*, e precisamente il tipo `boolean`, che é in effetti definito come:

```
TYPE boolean IS (FALSE, TRUE);
  -- questo tipo e' gia' definito e pertanto questa dichiarazione
  -- non deve essere inserita nei programmi
```

In pratica, un tipo enumerativo consiste in un elenco finito di valori, specificati dopo la parola riservata `IS`. Gli elementi elencati devono essere considerati ordinati, in modo che la relazione d'ordine dipende dalla successione dei valori elencati; ad esempio nel tipo `stato_civile` sopra dichiarato, l'elemento successivo a `libero` é `sposato`, e l'elemento precedente a `vedovo` é `divorziato`. Non esiste tuttavia alcun elemento precedente al primo (cioé all'elemento `sposato`), e neppure alcun elemento che segue l'ultimo (`vedovo`). Le variabili di tipo enumerativo possono essere inizializzate, come tutte le variabili; ad esempio possiamo dichiarare una variabile di tipo `stato_civile` e inizializzarla, con l'istruzione:

```
stato : stato_civile := sposato;
```

Per individuare il precedente ed il successivo di un elemento, si utilizzano le parole riservate `PRED` (precedente) e `SUCC` (successivo). Il seguente programma legge una lunghezza in una unitá di misura (centimetri, metri, pollici o piedi) e la converte nell'unitá successiva (e se questa non esiste, non viene effettuata nessuna conversione). Per far ció, viene dichiarato un tipo `unita_di_misura` di tipo enumerativo, e viene dichiarato un nuovo `PACKAGE` per permettere l'introduzione da tastiera di questo nuovo tipo.

```
WITH Ada.text_io, Ada.float_text_io;
  -- volutamente non mettiamo qui le dichiarazioni di uso delle risorse
  -- dei pacchetti Ada.text_io, Ada.float_text_io per mostrare come in
  -- alternativa questi possono essere specificati a fianco delle
  -- procedure put e get

PROCEDURE converti IS

  TYPE unita_di_misura IS (centimetri, metri, pollici, piedi);
     -- dichiarazione del tipo enumerativo
  PACKAGE unita_io IS NEW Ada.text_io.enumeration_io(unita_di_misura);
     -- per poter introdurre da tastiera o per poter visualizzare un
     -- valore del nuovo tipo sopra dichiarato, occorre dichiarare un
```

```
    -- nuovo pacchetto che metta a disposizione tali risorse.
    -- Tra parentesi viene indicato il tipo per il quale occorre
    -- mettere a disposizione le risorse di input/output, mentre
    -- la specificazione enumeration_io permette di specificare che
    -- tali risorse si riferiscono ad un tipo enumerativo.

  lunghezza : float;
    -- lunghezza da convertire nell'unita' successiva
  unita : unita_di_misura;
    -- questa variabile conterra' l'unita' di misura introdotta
    -- dall'utente

  cm_to_m : CONSTANT float := 0.01;
    -- fattore di conversione da centimetri a metri
  m_to_poll : CONSTANT float := 39.37;
    -- fattore di conversione da metri a pollici
  poll_to_pd : CONSTANT float := 0.083;
    -- fattore di conversione da pollici a piedi

BEGIN

  Ada.text_io.put("Introduci la lunghezza : ");
  Ada.float_text_io.get(lunghezza);
    -- introduzione della lunghezza
    -- se i pacchetti Ada.text_io ed Ada.float_text_io non vengono
    -- indicati a fianco della parola USE, e' possibile utilizzare
    -- ancora le istruzioni put e get a patto di specificare, al
    -- loro fianco, i pacchetti di appartenenza

  Ada.text_io.put("Introduci l'unita' di misura : ");
  unita_io.get(unita);
    -- introduzione dell'unita' di misura
    -- dato che utilizziamo le risorse del nuovo package unita_io
    -- occorre specificarlo prima della parola get, per indicare
    -- che questa istruzione appartiene al nuovo pacchetto

  CASE unita IS
    WHEN centimetri => lunghezza := lunghezza * cm_to_m;
         -- conversione da centimetri in metri
    WHEN metri      => lunghezza := lunghezza * m_to_poll;
         -- conversione da metri in pollici
    WHEN pollici    => lunghezza := lunghezza * poll_to_pd;
         -- conversione da pollici in piedi
    WHEN piedi      => NULL;
         -- l'istruzione NULL e' detta l'istruzione nulla
         -- perche' non esegue nessun compito; e' messa
         -- in questo caso proprio perche' non si deve
         -- esegure nulla
  END CASE;

  -- visualizzazione del risultato
  IF unita = piedi THEN
    Ada.text_io.put_line ("Non e' stata eseguita nessuna conversione");
  ELSE
    unita := unita_di_misura'SUCC(unita);
      -- per ottenere l'unita' di misura successiva si utilizza
      -- l'attributo SUCC (successivo), applicato alla variabile
```

```
      -- unita; occorre precisamente specificare anche il
      -- tipo di variabile, nella forma:
      --    tipo_di_variabile'SUCC(nome_della_variabile)
    Ada.text_io.put("La nuova unita' di misura e' : ");
    unita_io.put(unita);
    Ada.text_io.new_line;
       -- anche l'istruzione new_line fa parte del pacchetto Ada.text_io
    Ada.text_io.put("La nuova lunghezza e' : ");
    Ada.float_text_io.put(lunghezza, EXP=>0, FORE=>4, AFT=>3);
  END IF;

  EXCEPTION
     WHEN others => Ada.text_io.put("Errore di introduzione dei dati");

END converti;
```

Dal programma si puó notare come sia necessario specificare il pacchetto a fianco di ciascuna risorsa (ad esempio a fianco di put e get) se tale pacchetto non é stato dichiarato a fianco dell'istruzione `USE`. Occorre notare che molti programmatori Ada95 preferiscono lavorare in questo modo, per una maggiore chiarezza. Ad esempio, nel nostro caso in cui abbiamo dovuto dichiarare un nuovo pacchetto (`unita_io`) per introdurre e visualizzare i valori del nuovo tipo enumerativo `unita_di_misura`, l'utilizzo della specificazione del pacchetto mette meglio in evidenza la risorsa utilizzata (che e' differente a seconda del tipo di variabile che si vuol leggere o visualizzare). Notiamo anche come, per utilizzare l'attributo `SUCC` abbiamo dovuto specificare anche il tipo di variabile e la variabile stessa. La stessa cosa vale se avessimo dovuto usare l'attributo `PREC`. Osserviamo infine, che le istruzioni:

```
unita : unita_di_misura := piedi;
unita := unita_di_misura'SUCC(unita);
```

generano un errore non esistendo il successivo del valore `piedi`. Lo stesso discorso vale se avessimo inizializzato la variabile `unita` al valore `centimetri` e avessimo cercato il precedente di tale valore. Vediamo ora come un qualsiasi tipo enumerativo possa essere utilizzato per definire i limiti di esecuzione di un ciclo definito in cui la variabile di conteggio assume i valori dichiarati nel tipo stesso (vedi nota 1 a pié di pagina nel capitolo 4). A tal scopo, consideriamo il seguente programma, che visualizza, per tutta la settimana, un menu variato composto di cinque piatti che vengono presentati a rotazione (il piatto iniziale viene scelto dall'utente)

```
WITH Ada.text_io;

PROCEDURE pranzi IS
  TYPE giorno IS (lunedi, martedi, mercoledi, giovedi, venerdi, sabato);
  TYPE momento IS (colazione, mezzogiorno, cena);
  TYPE piatto IS (patatine, prosciutto, bistecca, sogliola, pastasciutta);
     -- dichiarazione di tre tipi enumerativi
  PACKAGE giorno_io IS NEW Ada.text_io.enumeration_io(giorno);
  PACKAGE momento_io IS NEW Ada.text_io.enumeration_io(momento);
  PACKAGE piatto_io IS NEW Ada.text_io.enumeration_io(piatto);
     -- pacchetti di input/output del piatto, del giorno e del momento
     -- della giornata
  portata : piatto; -- tipo di portata
BEGIN
  -- introduzione del piatto iniziale
  Ada.text_io.put("Con che piatto inizi la settimana ? ");
```

```
  piatto_io.get(portata);

  giorno_della_settimana:
  FOR giornata IN lunedi..sabato LOOP
    giorno_io.put(giornata); Ada.text_io.put(" => ");
    -- visualizza il giorno della settimana
    momento_della_giornata:
    FOR pranzo IN colazione..cena LOOP
      momento_io.put(pranzo); Ada.text_io.put(" : ");
      -- visualizza il momento della giornata
      piatto_io.put(portata); Ada.text_io.put("   ");
      -- visualizza il piatto
      IF portata = piatto'LAST THEN
         portata := piatto'FIRST;
         -- se il piatto e' l'ultimo della lista (pastasciutta) prende
         -- come piatto successivo il primo della lista nell'elenco
         -- del tipo enumerativo piatto. FIRST e LAST sono attributi
         -- che individuano il primo ed ultimo elemento di un tipo
         -- di variabile
      ELSE
         portata := piatto'SUCC(portata);
         -- passa al piatto successivo della lista
      END IF;
    END LOOP momento_della_giornata;
    Ada.text_io.new_line(SPACING => 2);
  END LOOP giorno_della_settimana;

  EXCEPTION
    WHEN others => Ada.text_io.put("Errore nell'introduzione dei dati");

END pranzi;
```

Nel programma abbiamo anche visto come gli attributi `FIRST` e `LAST` possono essere utilizzati per conoscere il valore del primo e dell'ultimo elemento di un tipo di variabile. Notiamo inoltre che i valori limiti dei cicli non devono necessariamente coincidere con il primo e l'ultimo valore di un tipo enumerativo; ad esempio se vogliamo visualizzare soltanto i piatti portati a pranzo e a cena possiamo cambiare il ciclo interno sostituendolo con l'istruzione: `FOR pranzo IN mezzogiorno..cena LOOP`. Osserviamo infine che anche il tipo carattere é un *tipo enumerativo predefinito*; esso é definito come:

```
TYPE character IS ('nul', ..., 'a', 'b', ..., 'z', ... 'del');
```

ove all'interno delle parentesi, sono elencati, in ordine, i 256 valori possibili che una variabile carattere puó assumere (non tutti questi caratteri sono stampabili; alcuni sono caratteri di controllo, e i primi 128 caratteri corrispondono a quelli dello standard ASCII, in cui sono numerati da 0 a 127). Per il tipo `character` esistono due attributi, `VAL` e `POS`; il primo fornisce il carattere avente il codice ASCII indicato a fianco, ed il secondo fornisce il codice ASCII del carattere specificato. Ad esempio, il seguente programma chiede un codice ASCII (da 0 a 127 e visualizza il corrispondente carattere, e poi chiede un carattere e ne visualizza il codice ASCII):

```
WITH Ada.text_io, Ada.integer_text_io;

PROCEDURE codici IS
  TYPE codici_ascii IS RANGE 0..127;
    -- definizione di un nuovo tipo naturale, limitato tra 0 e 127
    -- utilizzato per memorizzare i codici ASCII
  PACKAGE code_io IS NEW Ada.text_io.integer_io(codici_ascii);
```

```
    -- e corrispondente pacchetto per risorse di i/o
  ascii : codici_ascii;
  caratt : character;

BEGIN
  -- introduzione del codice ASCII e visualizzazione del carattere
  -- corrispondente
  Ada.text_io.put("Introduci un codice ASCII : ");
  code_io.get(ascii);
  caratt := character'VAL(ascii);
  Ada.text_io.put("Il carattere corrispondente e' : ");
  Ada.text_io.put(caratt);
  Ada.text_io.new_line(SPACING => 2);

  -- introduzione del carattere e visualizzazione del suo codice ASCII
  Ada.text_io.put("Introduci un carattere : ");
  Ada.text_io.get(caratt);
  ascii := character'POS(caratt);
  Ada.text_io.put("Il codice ASCII corrispondente e' : ");
  code_io.put(ascii);

  EXCEPTION
    WHEN others => Ada.text_io.put("Errore introduzione dati");

END codici;
```

10. Inizializzazione delle tabelle

Nei capitoli precedenti, abbiamo accennato ai vari modi di inizializzare le tabelle. É giunto ora il momento di analizzare piú in dettaglio i vari metodi di inizializzazione delle tabelle, in modo da evitare il piú possibile il ricorso a cicli definiti per tale operazione (che causerebbero un rallentamento del programma). Consideriamo, innanzitutto, tabelle monodimensionali; sia ad esempio `primi` una tabella contenente i primi 16 numeri primi maggiori di 1; possiamo dichiararla e inizializzarla in modo rapido scrivendo, all'interno delle parentesi tonde, l'elenco dei numeri primi:

```
primi : ARRAY (1..10) OF positive := (2, 3, 5, 7, 11, 13, 17, 19, 23, 29);
```

In questo modo, il primo elemento della tabella (cioé `primi(1)`) é il numero 2, il secondo (`primi(2)`) é il numero 3, e cosí via. Questa procedura di inizializzazione é sicuramente piú rapida e leggibile rispetto all'assegnare in una successione di 10 istruzioni, un valore a ciascun elemento della tabella. Il seguente frammento di programma rappresenta dunque un *cattivo* esempio di programmazione:

```
primi : ARRAY (1..10) OF positive;
...
primi(1) = 2;  -- cattivo esempio di inizializzazione
primi(2) = 3;  -- occorrono ben 10 instruzioni anzich\'e una sola
...
primi(10) = 29;
```

Puó anche capitare di dover inizializzare una tabella in modo che ciascun elemento sia uguale all'altro, ad esempio, riempire una tabella di 1. In questo caso, possiamo utilizzare la parola riservata `others` che permette di inizializzare tutti gli elementi non specificati della tabella al valore desiderato:

```
tabella : ARRAY (1..10) OF natural := (others => 1);
```

Oppure ad esempio puó essere necessario inizializzare i primi quattro inserendo in essi i primi numeri pari $(2, 4, 6, 8)$, ed il resto della tabella inserendo una successione di 0. In questo caso, va bene la seguente dichiarazione:

```
tabella : ARRAY (1..10) OF natural := (2, 4, 6, 8, others => 0);
   -- gli elementi dal quinto in poi sono inizializzati a 0
```

Notiamo che la fase di dichiarazione puó essere separata dalla fase di inizializzazione; ad esempio é possibile scrivere:

```
tabella : ARRAY (1..10) OF natural;
   -- dichiarazione
...
tabella := (2, 4, 6, 8, others => 0);
   -- inizializzazione
```

Possiamo anche indicare il posto a fianco del numero, ad esempio per indicare che il primo elemento deve contenere il numero 2, possiamo specificarlo con la scrittura `1 => 2`; in questo modo l'inizializzazione appare anche piú leggibile, come mostra il seguente frammento, equivalente al precedente:

```
tabella : ARRAY (1..10) OF natural;
   -- dichiarazione
...
tabella := (1 => 2, 2 => 4, 3 => 6, 4 => 8, others => 0);
   -- inizializzazione
```

É anche possibile elencare gli elementi che devono essere inizializzati; ad esempio supponiamo di avere una tabella di 20 numeri reali che deve essere inizializzata come segue:

- Il primo elemento deve essere uguale a $\pi \approx 3.14$
- Il secondo elemento deve essere uguale a $\sqrt{2} \approx 1.41$
- Gli elementi di posto $3, 6, 7$ devono essere uguali a 2.54
- Gli elementi dal posto 15 al posto 18 devono essere uguali a $\frac{1}{\sqrt{2}} \approx 0.71$
- Tutti gli altri elementi devono essere uguali a $\frac{1}{3} \approx 0.33$

Possiamo allora procedere con la seguente dichiarazione e inizializzazione:

```
n_reali : ARRAY (1..20) OF float;
  -- dichiarazione
...
n_reali := (1 => 3.14, 2 => 1.41,
               -- inizializza primo e secondo elemento
            3 | 6 | 7 => 2.54,
               -- inizializza gli elementi di posto 3, 6, 7
               -- (ricordare il significato di | che \'e OPPURE,
               -- vedi suo uso nelle strutture CASE)
            15..18 => 0.71,
               -- inizializza gli elementi di posto dal 15 al 18
            others => 0.33);
               -- inizializza tutti gli altri elementi
```

Per una tabella bidimensionale, é possibile specificare gli elementi, riga per colonna, utilizzando le parentesi tonde per indicare gli elementi che appartengono alla stessa riga. Volendo ad esempio inizializzare una tabella con i numeri:

$$\begin{bmatrix} -1 & 1 & 2 & -1 \\ 3 & 0 & 0 & 4 \\ 1 & 5 & 0 & 0 \\ -1 & 6 & 0 & 0 \end{bmatrix}$$

é possibile farlo con il seguente codice:

```
table : ARRAY (1..4, 1..4) OF integer := ((-1, 1, 2, -1),
                                            -- prima riga
                                          (3, 0, 0, 4),
                                            -- seconda riga
```

```
                                    (1, 5, 0, 0),
                                      -- terza riga
                                    (-1, 6, 0, 0));
                                      -- quarta riga
```

Anche per le tabelle bidimensionali, é possibile utilizzare la parola riservata `others`; ad esempio se vogliamo riempire una tabella di 6 righe e 6 colonne nel modo seguente:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & -1 & -1 & 0 & 1 \\ 1 & 0 & -1 & -1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

possiamo farlo nel modo seguente:

```
table : ARRAY (1..6, 1..6) OF integer :=
              (1 | 6 => (others => 1),
                -- prima e sesta riga inizializzare con tutti 1
               2 | 5 => (1 | 6 => 1, 3..4 => -1, others => 0),
                -- nella seconda e quinta riga riempie gli elementi
                -- sulle colonne 1 e 6 con il numero 1, quelli sulle
                -- colonne 3 e 4 con il numero -1, gli altri con 0
               3 | 4 => (1 | 6 => 1, others => 0));
                -- sulle righe 3 e 4 inizializza il primo ed il sesto
                -- elemento ad uno, gli altri a zero.
```

Se tutti gli elementi di una tabella bidimensionale devono essere inizializzati allo stesso valore, ad esempio al valore `'A'` in una tabella di caratteri, é possibile farlo specificando sia gli elementi di riga che di colonna con `others`:

```
caratteri : ARRAY (1..4, 1..5) OF character := (others => (others => 'A'));
```

Infine, osserviamo che il campo degli indici delle tabelle puó essere anche un tipo *enumerativo*, ad esempio se vogliamo costruire una tabella che contiene il numero di alunni per ciascun tipo di scuola presente in una cittá, possiamo procedere con la seguente dichiarazione, in cui procediamo anche ad inizializzare il numero di alunni:

```
TYPE tipo_di_scuola IS (classico, scientifico, professionale, tecnico);
scuola : ARRAY (classico..tecnico) OF natural;
scuola := (classico => 100, scientifico => 200, others => 300);
```

11. Stringhe di caratteri

Un tipo particolare di tabelle é costituito dalle *stringhe* di caratteri. Una stringa di caratteri é una tabella capace di contenere una successione di caratteri (numeri, lettere, simboli di punteggiatura ed altro), ad esempio un nome, un cognome, il codice fiscale di una persona, una password. Il tipo stringa viene dichiarato con la parola `string`; ad esempio, con la seguente dichiarazione:

```
nome : string(1..20);
cognome : string(1..20);
codice_fiscale : string(1..16)
```

abbiamo allocato spazio in memoria per contenere nome, cognome e codice fiscale di una persona. Nella nostra dichiarazione, il nome e il cognome possono essere composti da un massimo di 20 caratteri, il codice fiscale invece da 16 caratteri. Per introdurre una stringa, si utilizza la procedura `get_line`, che legge tutti i caratteri inseriti da tastiera sino al termine della linea (segnalata dalla pressione del tasto INVIO), e li inserisce nella stringa indicata come primo parametro. Il secondo

parametro é invece un numero naturale che indica il numero di caratteri letti da tastiera. Ad esempio, il seguente programma legge da tastiera una stringa di *al piú* otto caratteri e la visualizza alla rovescia (esempio: se da tastiera si introduce la stringa `Miao` la procedura visualizza `oaiM`).

```
WITH Ada.text_io, Ada.integer_text_io;
USE Ada.text_io, Ada.integer_text_io;

PROCEDURE stringa IS
  s : string (1..8);
      -- la variabile s rappresenta la stringa di caratteri
      -- introdotta da tastiera (massimo otto caratteri)
  l : natural;
      -- l rappresenta la lunghezza della stringa, ovvero il
      -- numero di caratteri introdotti da tastiera
BEGIN
  put("Introduci la stringa :");
  get_line(s,l);
  -- il primo parametro di get_line e' la stringa, il secondo
  -- il numero di caratteri introdotti. La procedura get_line
  -- legge una linea di caratteri da tastiera e la inserisce
  -- nella stringa s; assegna inoltre ad l il numero di caratteri
  -- letti. Se i caratteri digitati sono piu' di otto considera
  -- solo i primi otto.
  FOR i IN REVERSE 1..l LOOP
     -- questo ciclo visualizza gli l caratteri letti, alla rovescia
     put(s(i));
     -- l'istruzione put(s(i)) visualizza l'iesimo carattere della
     -- stringa s. E' importante notare come l'iesimo componente di
     -- una stringa sia un carattere singolo.
  END LOOP;
  new_line;
  put("Numero di caratteri letti : ");
  put(l, WIDTH => 1);
END stringa;
```

Una qualsiasi stringa viene invece indicata con i suoi caratteri racchiusi tra doppi apici (esempio: `"Ada"`). Ricordiamo che la variabile di tipo `character` contiene *sempre* uno ed un solo carattere, che viene rappresentato tra apici singoli. Pertanto, il simbolo `'a'` indica un *carattere*, mentre il simbolo `"a"` indica una stringa (composta in questo caso da un solo carattere). Le stringhe possono essere inizializzate giá al momento della loro dichiarazione, ad esempio:

```
nome_prog : string(1..12) = "Ada Lovelace";
   -- il nome della prima donna programmatrice
   -- anche lo spazio vuoto costituisce un carattere della stringa
linguaggio_preferito : string(1..5) = "Ada95";
comando_di_compilazione : string(1..6) = "gcc -c";
```

Se si vuole visualizzare sono una parte della stringa, e' possibile specificarlo all'interno della procedura `put_line`. Ad esempio, osservando che `nome_prog(1..3)` é `"Ada"` e che `nome_prog(5..12)` é `"Lovelace"`, é possibile scrivere separatamente nome e cognome della persona con i comandi:

```
put_line(nome_prog(1..3));  -- visualizza "Ada"
put_line(nome_prog(5..12)); -- visualizza "Lovelace"
```

É inoltre possibile concatenare due stringhe per ottenerne una terza. Per far ció, si utilizza l'operatore di concatenazione `&`. Ad esempio, siano:

```
nome_maschietto : string(1..6) = "Davide";
```

```
fidanzata : string(1..8) = "Beatrice";
frase : string(1..19);
```

In questo caso, con i comandi

```
frase := nome_maschietto & " ama " & fidanzata
put_line(frase);
```

viene visualizzata la frase "Davide ama Beatrice". Occorre notare che se la lunghezza totale della stringa supera la dimensione dell'array, ad esempio nel caso in cui frase fosse dichiarata come una tabella di 18 caratteri o meno, si ha un constraint_error che puó essere trattato come una eccezione. Nel caso sia necessario introdurre piú dati dello stesso tipo (ad esempio i cognomi degli alunni di una classe), é possibile dichiarare una tabella di stringhe. Ad esempio il seguente PACKAGE permette di dichiarare una tabella contenente i nomi di una classe di 10 alunni, in cui la lunghezza massima di ciascun nome é limitata a 20 caratteri.

```
PACKAGE alunni IS
  max_caratteri : CONSTANT positive := 20;
  max_alunni : CONSTANT positive := 10;
    -- costanti per definire il numero massimo di caratteri
    -- di un nome ed il numero massimo di alunni
  TYPE num_caratteri IS ARRAY (1..max_alunni) OF positive;
    -- tabella contenente la lunghezza di ciascun nome
  TYPE names IS ARRAY (1..max_alunni) OF string(1..max_caratteri);
    -- dichiarazione della tabella dei nomi (per 10 persone)
END alunni;
```

Scriviamolo e salviamolo come alunni.ads. Scriviamo ora un programma che legge da tastiera i nomi dei dieci alunni, e poi chiede una lettera, e visualizza tutti i nomi che cominciano con tale lettera. Esso si compone di una procedura chiamante (allievi.adb) che chiama le procedure introduci_alunni.adb (per introdurre i nomi degli alunni), introduci_lettera.adb (per introdurre la lettera iniziale del nome, e visualizza_nomi.adb (per visualizzare i nomi che hanno tale iniziale). Cominciamo dalla procedura principale:

```
WITH alunni, introduci_alunni, introduci_lettera, visualizza_alunni;
USE alunni;

PROCEDURE allievi IS
   lettera : character;  -- carattere iniziale dei cognomi
   nomi : names;         -- tabella dei nomi degli allievi
   numero_alunni : positive;
                         -- numero di alunni da introdurre
   tabella_lunghezze : num_caratteri;
                         -- tabella contenente le lunghezze dei
                         -- nomi introdotti
   successo : boolean;   -- FALSE quando si ha una eccezione
                         -- nell'introduzione dei dati (vedi
                         -- programma numerazione cruciverba)
BEGIN
  LOOP
    introduci_alunni(nomi, tabella_lunghezze, numero_alunni, successo);
    -- introduzione del numero degli alunni e dei loro nomi
    EXIT WHEN successo;
    -- esce dal ciclo se l'introduzione e' avvenuta senza eccezioni
  END LOOP;

  LOOP
    introduci_lettera(lettera, successo);
```

```
    -- introduzione della lettera iniziale
    EXIT WHEN successo;
    -- come sopra
  END LOOP;

  visualizza_alunni(nomi, tabella_lunghezze, numero_alunni, lettera);
    -- visualizzazione dell'elenco degli alunni con tale iniziale
END allievi;
```

La procedura `introduci_alunni` chiede innanzitutto il numero di alunni di cui si vuol introdurre i nomi, e poi dopo aver controllato che tale numero é compreso tra 1 e il numero massimo di alunni ammesso (`max_alunni`), provvede a far introdurre i nomi. Notiamo come l'uso di una variabile booleana permette di far ripetere l'introduzione qualora si verificasse una eccezione (vedi programma di numerazione del cruciverba). Per evitare errori di digitazione, dopo l'introduzione di ciascun nome chiede conferma della corretta introduzione; ed eventualmente permette di reintrodurlo. Eccola:

```
WITH Ada.text_io, Ada.integer_text_io, alunni;
USE Ada.text_io, Ada.integer_text_io, alunni;

PROCEDURE introduci_alunni (nomi_alunni : OUT names;
                                -- tabella dei nomi degli alunni
                            numero_car : OUT num_caratteri;
                                -- tabella lunghezze cognomi
                            numero : OUT positive;
                                -- numero di alunni
                            esito : OUT boolean) IS
                                -- FALSE se vi e' una eccezione
                                -- nell'introduzione dati

conferma : character;
   -- carattere di conferma ('S' / 'N') del nome introdotto
n_char : natural;
   -- numero di caratteri letti per ciascun alunno

BEGIN

   esito := TRUE;

   LOOP
     -- introduzione del numero degli alunni
     -- con un ciclo di controllo
     put("Numero di alunni  - massimo = ");
     put(max_alunni, WIDTH => 2);
     put(" ? ");
     get(numero);
     skip_line;
     -- se dopo aver letto un numero si deve leggere una
     -- stringa, occorre mettere il comando skip_line per
     -- evitare che il carattere INVIO venga considerato
     -- come una stringa, con il risultato di leggere una
     -- stringa di zero caratteri.

     EXIT WHEN numero >= 1 AND numero <= max_alunni;
     put_line("Numero di alunni errato");
   END LOOP;
```

```
-- ciclo di introduzione degli alunni
introduci_i_nomi:
FOR i IN 1..numero LOOP

   -- ciclo interno per l'introduzione del singolo nome
   -- e la conferma del dato introdotto
   introduzione_nome_con_conferma:
   LOOP
      put("Nome dell'alunno numero : ");
      put(i, WIDTH => 2);
      put(" ? ");
      get_line(nomi_alunni(i), n_char);
      -- legge il nome dell'iesimo alunno

      IF n_char = 0 THEN
         -- non e' stato letto nessun carattere
         -- visualizza un messaggio di errore
         put_line("Nessun carattere letto");
      ELSE
        -- chiede di digitare il carattere 'S' maiuscolo o minuscolo
        -- oppure il carattere 'N' (od 'n') per sapere se confermare
        -- tale nome; rifiuta altri caratteri e in questo caso rimane
        -- in ciclo a richiederne la reintroduzione
        ciclo_chiedi_conferma:
        LOOP
          put("Confermi il nome introdotto (S/N) ? ");
          get(conferma);
          skip_line;
          -- evita che il carattere INVIO sia considerato come la
          -- prossima stringa da leggere
          EXIT WHEN conferma = 'S' OR conferma = 's' OR
                    conferma = 'N' OR conferma = 'n';
          put_line("Carattere errato");
        END LOOP ciclo_chiedi_conferma;

        IF conferma = 'S' OR conferma = 's' THEN
           numero_car(i) := n_char;
           -- nel caso in cui il nome e' stato confermato
           -- salva il numero di caratteri letti nella tabella
           put("Nome correttamente introdotto di ");
           put(n_char, WIDTH => 2);
           put_line(" caratteri");
        ELSE
           put_line("Nome non confermato: reintroducilo");
        END IF;

      END IF;

      EXIT WHEN (conferma = 'S' OR conferma = 's') AND n_char >= 1;
        -- esce se il nome introdotto e' stato confermato e se e'
        -- composto di almeno un carattere in modo da procedere
        -- all'introduzione del nome successivo

   END LOOP introduzione_nome_con_conferma;

END LOOP introduci_i_nomi;
```

```
   EXCEPTION
      WHEN others => esito := FALSE;
                     skip_line;
                     put_line("Errore di introduzione dei dati");

END introduci_alunni;
```

La procedura seguente é molto semplice, limitandosi a chiedere l'introduzione del carattere iniziale.

```
WITH Ada.text_io; USE Ada.text_io;

PROCEDURE introduci_lettera (caratt : OUT character;
                             esito : OUT boolean) IS
BEGIN

   esito := TRUE;
   -- vedi procedura introduci_alunni

   LOOP
     put("Introduci la lettera iniziale : ");
     get(caratt);
     skip_line;
     -- salta gli altri caratteri introdotti, compreso il
     -- carattere invio
     EXIT WHEN caratt IN 'a' .. 'z' OR
               caratt IN 'A' .. 'Z';
     -- il ciclo permette di controllare che il carattere
     -- iniziale sia una lettera dell'alfabeto compresa
     -- tra la 'a' e la 'z' (minuscoli e maiuscoli).
     put("Carattere errato");
   END LOOP;

   EXCEPTION
     WHEN others => esito := FALSE;
                    skip_line;
                    put("Errore introduzione lettera");

END introduci_lettera;
```

Infine la seguente procedura permette di visualizzare i nomi che hanno come iniziale la lettera inserita:

```
WITH Ada.text_io, Ada.integer_text_io, alunni;
USE Ada.text_io, Ada.integer_text_io, alunni;

PROCEDURE visualizza_alunni (nomi_al : IN names;
                                -- tabella con i nomi degli alunni
                             lungh : IN num_caratteri;
                                -- tabella con lunghezza nomi
                             num : IN positive;
                                -- numero di alunni
                             lett : character) IS
                                -- lettera iniziale
BEGIN
   put("Visualizzo i nomi che iniziano con la lettera : ");
   put(lett);
   new_line;

   -- ciclo che permette di esaminare un alunno alla volta
```

```
   -- di quelli presenti nella tabella e di visualizzarlo
   -- se ha come iniziale la lettera specificata
   FOR i IN 1..num LOOP
      IF nomi_al(i)(1) = lett THEN
         -- il primo indice (i) serve per individuare
         -- l'alunno nella tabella, mentre il
         -- numero 1 serve per individuare il primo
         -- carattere dell'alunno in esame
         put_line(nomi_al(i)(1..lungh(i)));
          -- scrive il nome dell'iesimo alunno, la cui
          -- lunghezza e' contenuta all'iesimo posto
          -- della tabella lungh
      END IF;
   END LOOP;
END visualizza_alunni;
```

Possiamo, come al solito, provare il tutto con i comandi `gnatmake allievi.adb` e `./allievi`. Le stringhe possono essere confrontate tra di loro, cosí come i caratteri e i numeri, utilizzando gli operatori di confronto `<` (minore), `<=` (minore o uguale), `>` (maggiore), `>=` (maggiore o uguale), `=` (uguale), `/=` (diverso). L'ordine dei caratteri che rappresentano le lettere é quello alfabetico, sia per le maiuscole che per le minuscole. Come esempio, il programma seguente legge due nomi da tastiera di al massimo 10 caratteri e li visualizza in ordine alfabetico, visualizzandone uno solo nel caso in cui essi siano uguali.

```
WITH Ada.text_io;
USE Ada.text_io;

PROCEDURE due_nomi IS
   nome_1 : string(1..10) := (others => ' ');  -- primo nome
   nome_2 : string(1..10) := (others => ' ');  -- secondo nome
            -- le due variabili stringa vengono inizializzate
            -- come stringe di spazi, al fine di evitare che
            -- i caratteri presenti nella stringa oltre a
            -- quelli letti da tastiera creino dei problemi
            -- nell'ordinamento
   lun_1, lun_2 : natural;  -- numero caratteri introdotti
BEGIN
   put("Introduci il primo nome : ");
   get_line(nome_1, lun_1);
   put("Introduci il secondo nome : ");
   get_line(nome_2, lun_2);
   IF nome_1 = nome_2 THEN
      put("I due nomi sono uguali e sono : ");
      put_line(nome_1(1..lun_1));
   ELSIF nome_1 < nome_2 THEN
      put("Primo nome in ordine alfabetico : ");
      put_line(nome_1(1..lun_1));
      -- nota: dato che i rimanenti caratteri sono
      -- degli spazi, e che si vuole andare a capo,
      -- e' anche possibile scrivere, piu' semplicemente,
      -- put_line(nome_1). Stessa considerazione per nome_2.
      put("Secondo nome : ");
      put_line(nome_2(1..lun_2));
   ELSE
      put("Primo nome in ordine alfabetico : ");
      put_line(nome_2(1..lun_2));
      put("Secondo nome : ");
```

```
      put_line(nome_1(1..lun_1));
   END IF;
END due_nomi;
```

L'inizializzazione delle variabili contenenti i nomi serve ad evitare comportamenti anomali dovuti alla presenza nelle stringhe di caratter indesiderati. Infatti, quando viene riservato spazio in memoria per una stringa (o per qualsiasi altra variabile) il suo contenuto é sconosciuto e del tutto casuale, dipendente solo dai vecchi dati contenuti in memoria. Ad esempio, i contenuti di `nome_1` e `nome_2`, se non inizializzati potrebbe essere `"Y7-PQ+zs.d"` per `nome_1` e `"de12-z;s9q"` per `nome_2`. Se da tastiera vengono introdotti i nomi `Davide` e `David` i nuovi valori delle due variabili saranno rispettivamente `"Davidezs.d"` e `"Davidz;s9q"`. Confrontanto le due stringhe, si avrebbe che la prima risulta precedere la seconda (i primi cinque caratteri, `"David"` coincidono, mentre il sesto carattere é `'e'` per la prima e `'z'` per la seconda (e la `'e'` precede la `'z'` in ordine alfabetico). Di conseguenza, la visualizzazione sarebbe `Davide` come primo nome e `David` come secondo. Tuttavia sappiamo che nel confronto lessicografico dei nomi, `David` dovrebbe essere stampato prima di `Davide` (perché i primi cinque caratteri coincidono, e la prima stringa é piú corta della seconda). Per ottenere questo risultato, basta procedere all'inizializzazione delle stringhe, come abbiamo fatto nel nostro programma.

12. Funzioni e procedure ricorsive

Passiamo ora ad analizzare un potente metodo di risoluzione dei problemi: la *ricorsione*. Diremo che una funzione (o una procedura) é ricorsiva se é definita solo in termini di se stessa. Imparare a scrivere funzioni e procedure ricorsive é un esercizio molto importante: esistono problemi che possono essere risolti solo mediante l'uso della ricorsione. Ma c'é di piú: se un problema non puó essere risolto in modo ricorsivo, non esiste alcun altro modo di risolverlo per mezzo di un elaboratore. Anziché concentrarsi sulla teoria, vediamo un semplice esempio: il calcolo del fattoriale di un numero. Ricordiamo che, dato un intero $n \geq 0$, il fattoriale di n, indicato con $n!$, é cosí definito:

$$0! = 1$$

$$1! = 1$$

$$2! = 2 \cdot 1$$

$$3! = 3 \cdot 2 \cdot 1$$

$$\cdots$$

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1$$

La funzione seguente (che non fa ancora uso della ricorsione), ritorna il fattoriale di `n`, ove `n` é il parametro intero della funzione.

```
FUNCTION fattor (n : natural) RETURN natural IS
   f : natural := 1;
BEGIN
   f := 1;
   FOR j IN 2..n LOOP
       f := f * j;
   END LOOP;
   RETURN f;
END fattor;
```

In questo programma, se `n` é maggiore di 1, il numero `f`, inizializzato a 1, viene moltiplicato (all'interno del ciclo `FOR`) per 2, per 3, ..., per `n` sino ad ottenere `n!` Il ciclo non viene eseguito se `n` é minore o uguale ad 1, e quindi se n vale 0 oppure 1, la variabile locale `f` rimane, come desiderato, al valore 1. Notiamo come anche in questo caso la variabile `f` ed il parametro `n`, nonché il valore ritornato siano dichiarati di tipo `natural` (e non `integer`); il tipo `natural` indica l'insieme dei numeri naturali (zero compreso), ovvero l'insieme degli interi non negativi. Essendo nel nostro caso $n >= 0$, ed $n! > 0$, conviene pertanto utilizzare il tipo `natural` per mettere meglio in evidenza che le grandezze in gioco non sono mai negative.

Riscriviamo ora la tabellina dei fattoriali, mettendo in evidenza alcune cose:

$$0! = 1$$
$$1! = 1 \cdot 0!$$
$$2! = 2 \cdot 1!$$
$$3! = 3 \cdot (2 \cdot 1) = 3 \cdot 2!$$
$$4! = 4 \cdot (3 \cdot 2 \cdot 1) = 4 \cdot 3!$$
$$\dots$$
$$n! = n \cdot [(n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1] = n \cdot (n-1)!$$

Possiamo immediatamente notare che:

(1) Il *fattoriale* di 0 é 1
(2) Se $n > 0$, il *fattoriale* di n é uguale al prodotto di n per il *fattoriale* di $n-1$

Queste osservazioni, costituiscono la definizione di $n!$ in termini *ricorsivi* (infatti l'evidenziazione in corsivo nella seconda proposizione mostra come il fattoriale di un numero possa essere definito in termini del fattoriale del numero precedente, ovvero abbiamo una funzione definita in termini di se stessa) La prima osservazione, che contempla il caso piú semplice ($0! = 1$) é detta la *base* della ricorsione. La seconda osservazione, che definisce il fattoriale di n in termini del fattoriale di $n-1$, é la *ricorsione* stessa. In questo modo abbiamo potuto definire il calcolo del fattoriale di un numero n in termini del fattoriale del suo predecessore, ovvero del fattoriale di $n-1$. Scriviamo ora la funzione, in modo *ricorsivo*: non si tratta di far altro che tradurre in Ada quanto abbiamo detto in italiano:

```
FUNCTION fattoriale (n : natural) RETURN natural IS
BEGIN
   IF n = 0 THEN RETURN 1;
      -- il fattoriale di 0 e' 1, base della ricorsione
   ELSE RETURN n * fattoriale(n - 1);
      -- ricorsione: per calcolare il fattoriale di n > 0 la funzione
      -- richiama se stessa per calcolare il fattoriale di n - 1,
      -- e a sua volta la funzione richiamera' se stessa per il
      -- calcolo del fattoriale di n - 2, ..., e cosi' sino all'ultima
      -- chiamata, il fattoriale di 0 (la base della ricorsione)
   END IF;
END fattoriale;
```

Osserviamo che la funzione scritta in modo ricorsivo NON fa uso di cicli (il ciclo é sostituito dalla ricorsione stessa), e che la scrittura ricorsiva ci permette di ridurre il numero di righe di codice, e di scrivere la funzione in modo piú semplice e naturale. E' importante notare come la soluzione ricorsiva sia, di solito, piú semplice e piú naturale (cioé piú vicino al linguaggio umano) della soluzione non ricorsiva. Tuttavia, dato che la funzione ricorsiva continua a chiamare se stessa, la soluzione ricorsiva risulta in genere piú lenta e richiede un maggior utilizzo di memoria rispetto alla soluzione non ricorsiva (quando quest'ultima esiste). Per

provarla, scriviamo una procedura che chiede l'introduzione del numero di cui si vuol calcolare il fattoriale, che chiama la funzione, e che visualizza il risultato:

```
WITH Ada.text_io, Ada.integer_text_io, fattoriale;
USE Ada.text_io, Ada.integer_text_io;

PROCEDURE prova_fatt IS
   numero : natural;
   fatt : natural;
BEGIN
   put("Introduci un numero maggiore o uguale a zero : ");
   get(numero);
   fatt := fattoriale(numero);
   put("Il suo fattoriale e' : ");
   put(fatt, WIDTH => 5);
END prova_fatt;
```

Passiamo ora ad un nuovo problema, noto come l'*allevamento di conigli di Fibonacci*. Il problema consiste nel calcolare, anno dopo anno, il numero di coppie di conigli presenti in un allevamento, tenendo presente le seguenti regole:

(1) Il primo hanno l'allevamento consiste in una sola coppia di conigli giovani
(2) Dopo un anno una coppia di conigli giovani diventa adulta
(3) I conigli giovani non si accoppiano
(4) Ogni anno, ogni coppia di conigli adulti genera una coppia di conigli giovani
(5) I conigli sono immortali

Costruiamo, innanzitutto, una tabella, in cui rappresentiamo, anno per anno, la situazione dell'allevamento

Anno	Coppie giovani	Coppie adulte	Totale coppie
1	1	0	1
2	0	1	1
3	1	1	2
4	1	2	3
5	2	3	5
6	3	5	8

TABELLA 1. Andamento dell'allevamento di conigli di Fibonacci

Il primo anno abbiamo una coppia di conigli giovani, che diventa adulta (senza procreare) in un anno, cosí che al secondo anno abbiamo una coppia di adulti. Questa genera una coppia di giovani, e quindi, al terzo anno, disponiamo di una coppia di giovani e di una coppia di adulti. Passa un anno, e la coppia giovane diventa adulta, mentre la coppia di adulta genera un'altra coppia di giovani. In questo modo, al terzo anno, abbiamo tre coppie di conigli, e cosí via. Cerchiamo di descrivere matematicamente l'andamento dell'allevamento: indichiamo con t l'anno, con $g(t)$ il numero di coppie giovani nell'anno t, con $g(t-1)$ il numero di coppie giovani nell'anno precedente $(t-1)$, con $a(t)$ il numero di coppie adulte nell'anno t, e con $a(t-1)$ il numero di coppie adulte nell'anno precedente, e cosí via. Infine, indichiamo con $n(t)$ il numero totale di coppie nell'anno t. Si ha allora:

- $g(1) = 1 \qquad a(1) = 0$
- $g(2) = a(1) = 1 \qquad a(2) = a(1) + g(1) = n(1) = 1$
- $g(3) = a(2) \qquad a(3) = a(2) + g(2) = n(2)$
- ...
- $g(t-1) = a(t-2) \qquad a(t-1) = a(t-2) + g(t-2) = n(t-2)$

- $g(t) = a(t-1)$ $\quad a(t) = a(t-1) + g(t-1) = n(t-1)$

Inoltre, il numero totale di coppie nell'anno t é la somma del numero di coppie giovani ed adulte:

$$n(t) = g(t) + a(t)$$

Ma prendendo le equazioni seguenti:
$g(t) = a(t-1)$ e $a(t) = n(t-1)$ possiamo esprimere $n(t)$ come:

$$n(t) = a(t-1) + n(t-1)$$

A sua volta, avendosi $a(t-1) = n(t-2)$ ricaviamo:

$$n(t) = n(t-1) + n(t-2)$$

Questa equazione esprime il numero totale di coppie di conigli nell'anno $t \geq 2$ in funzione del numero di coppie presenti nell'allevamento nei due anni precedenti. Dato che é definita in funzione di se stessa, costituirá la nostra ricorsione. Queste equazioni sono note anche come *equazioni di ricorrenza*, ed hanno una importanza fondamentale nella Teoria dei Sistemi. Possiamo ora abbandonare la matematica, e dedicarci all'informatica; da quanto visto possiamo dire che come base della ricorsione possiamo prendere lo stato dell'allevamento negli anni 1 e 2, e come ricorsione stessa quella appena trovata. Possiamo pertanto risolvere il problema nel modo seguente:

```
SE t = 1 OPPURE t = 2 ALLORA numero coppie = 1  (base della ricorsione)
ALTRIMENTI il numero di coppie nell'anno t si ottiene dalla
           somma del numero di coppie degli anni t-1 e t-2
```

Possiamo ora tradurre tutto questo in una funzione:

```
FUNCTION num_coppie (anno : positive) RETURN positive IS
BEGIN
   IF anno = 1 OR anno = 2
      THEN RETURN 1; -- base della ricorsione
      ELSE RETURN num_coppie(anno - 1) + num_coppie(anno - 2);
   END IF;
END num_coppie;
```

Per provarla, scriviamo una procedura che visualizza il numero di coppie di conigli nei primi dieci anni di allevamento.

```
WITH Ada.text_io, Ada.integer_text_io, num_coppie;
USE Ada.text_io, Ada.integer_text_io;

PROCEDURE conigli IS
   coppie : positive;   -- numero coppie di conigli
BEGIN
   FOR year IN 1..10 LOOP
      coppie := num_coppie(year);
      put (" Anno = "); put(year, WIDTH => 3);
      put (" Coppie = "); put(coppie, WIDTH => 3);
      new_line;
   END LOOP;
END conigli;
```

Un altro problema che illustra l'uso della ricorsivitá é dato dal calcolo del massimo comun divisore tra due numeri interi. Siano a, b due numeri interi maggiori o uguali a zero, ma non entrambi nulli. Il massimo comun divisore tra a e b puó essere calcolato osservando che:

- Se $a = 0$ e $b \neq 0$ allora il massimo comun divisore é b

- Altrimenti, se $a \neq 0$ e $b = 0$ allora il massimo comun divisore é a
- Altrimenti, se $a \geq b$ il massimo comun divisore tra a e b é uguale al massimo comun divisore tra $a - b$ e b
- Altrimenti (se $a < b$) il massimo comun divisore tra a e b é uguale al massimo comun divisore tra a e $b - a$

I primi due casi (i piú semplici) costituiscono la *base* della ricorsione, i rimanenti due la *ricorsione* stessa. Se indichiamo con $mcd(a, b)$ la funzione che calcola il massimo comun divisore tra i due numeri, possiamo facilmente tradurre quanto detto in un esempio pratico. Ad esempio, sia $a = 40$, $b = 24$. Si ha allora:

$$mcd(40, 24) = mcd(16, 24) = mcd(16, 8) = mcd(8, 8) = mcd(0, 8) = 8$$

Possiamo ora tradurre quanto esposto in una funzione Ada:

```
FUNCTION mcd_ric (a : natural; b: natural) RETURN natural IS
BEGIN
   IF a = 0
      THEN RETURN b; -- base della ricorsione
   ELSIF b = 0
      THEN RETURN a; -- base della ricorsione
   ELSIF a >= b
      THEN RETURN mcd_ric(a-b, b); -- ricorsione
      ELSE RETURN mcd_ric(a, b-a); -- ricorsione
   END IF;
END mcd_ric;
```

Ecco infine la procedura per provare la nostra funzione:

```
WITH Ada.text_io, Ada.integer_text_io, mcd_ric;
USE Ada.text_io, Ada.integer_text_io;

PROCEDURE prova_mcd IS
   n1, n2 : natural;  -- numeri introdotti da tastiera di cui si vuol
                      -- calcolare il massimo comun divisore
   maxcd : natural;   -- massimo comun divisore
BEGIN
   -- si tralasciano, per brevita', i controlli sull'esattezza dei
   -- numeri introdotti, che devono essere maggiori o uguali a zero,
   -- e non entrambi nulli
   put("Introduci il primo numero : "); get(n1);
   put("Introduci il secondo numero : "); get(n2);
   maxcd := mcd_ric(n1, n2);                 -- chiamata della funzione
   put("Il m.c.d. vale: "); put(maxcd, WIDTH => 4); -- visualizzazione
END prova_mcd;
```

Passiamo ora ad un nuovo problema, ovvero determinare il *massimo* numero di parti di piano ottenute dividendo un piano con piú rette. Piú precisamente, consideriamo la figura 4, in cui una retta divide un piano in due parti.

Se tracciamo una nuova retta, non parallela alla prima, il piano resta diviso in quattro parti[2] (vdi figura 5).

[2]Se la seconda retta é parallela alla prima, il piano sarebbe diviso in tre parti, pertanto se vogliamo determinare il numero *massimo* di parti di piano divise da piú rette, dobbiamo ipotizzare che nessuna retta sia parallela alle altre.

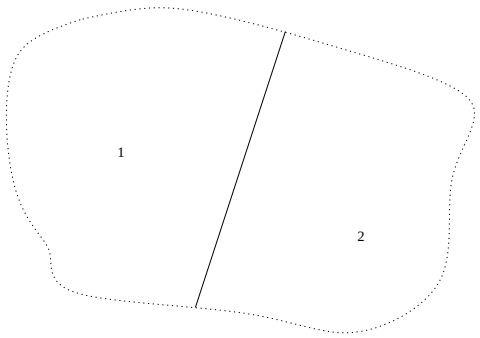

FIGURA 4. Un piano é diviso in due parti da una retta

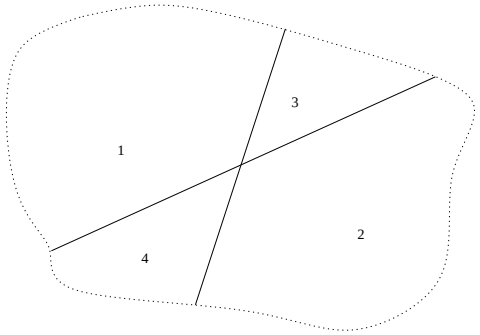

FIGURA 5. Un piano é diviso in al piú quattro parti da due rette

Tracciamo adesso una terza retta, non parallela alle altre due, e che non passi per il punto di intersezione delle prime due[3]. In questo caso, il piano risulta diviso in sette parti (figura 6).

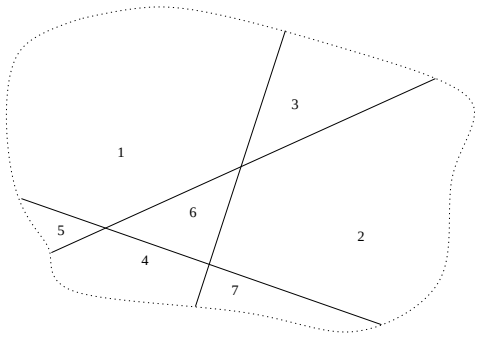

FIGURA 6. Un piano é diviso in sette parti da tre rette

Osserviamo ora la figura 7, ove sono tracciate $n-1$ rette (disegnate con linea continua). Se aggiungiamo una retta (tratteggiata), otteniamo che quest'ultima divide il piano in altre n parti.

Possiamo pertanto dire che:

[3]Se cosí fosse, il piano risulterebbe diviso in sei parti, anziché in sette; pertanto per determinare il massimo numero di parti di piano, occorre ipotizzare che nessuna retta passi per i punti di intersezione di quelle precedentemente tracciate sul piano.

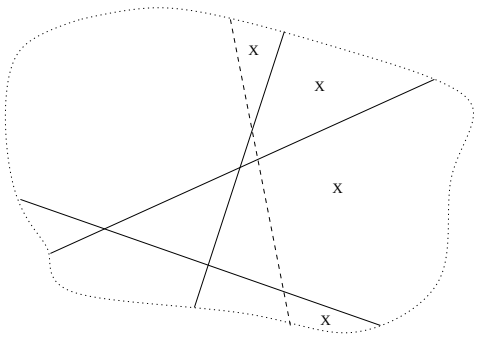

FIGURA 7. Una nuova retta aggiunge al piú le parti di piano indicate con una crocetta

- Una retta divide il piano in due parti (la base della ricorsione)
- Se $n > 1$, il numero di parti di piano separate da n rette si ottiene sommando n (le nuove parti di piano) al numero di rette divise dalle precedenti $n - 1$ rette (la ricorsione)

Possiamo adesso scrivere una funzione Ada, ove n é il numero di rette (passato come parametro), e che ritorna il numero di parti di piano:

```
FUNCTION parti_di_piano (n : positive) RETURN positive IS
BEGIN
   IF n = 1
      THEN RETURN 2; -- base della ricorsione
      ELSE RETURN n + parti_di_piano(n-1); -- ricorsione
   END IF;
END parti_di_piano;
```

La ricorsivitá puó essere utilizzata anche per il famoso gioco della *Torre di Hanoi*. Il gioco consiste in tre pali, sui quali sono infilati dei dischi forati, di dimensioni differenti, e ci si deve cimentare nello spostare una pila di n dischi dal palo di destra al palo di sinistra (utilizzando il palo centrale come palo ausiliario per gli spostamenti), e tenendo presente le seguenti regole:

- Si puó spostare, da un palo all'altro, solo un disco alla volta
- Un disco grande non puó stare sopra un disco piú piccolo

In figura 8 é rappresentato la soluzione del gioco, nel caso semplice di $n = 3$ dischi. Il palo 1 é il palo di partenza, il palo 3 quello di arrivo, ed il 2 quello ausiliario.

Questo gioco, si é diffuso in Europa alla fine del diciannovesimo secolo, anche per una efficace campagna pubblicitaria promozionale che sosteneva che alcuni sacerdoti di un tempio di Brahma si cimentavano quotidianamente con esso (spostando un disco della torre al giorno sui 64 totali), e che il termine del gioco sarebbe coinciso con la fine del mondo. Vogliamo scrivere una procedura, ricorsiva, capace di risolvere il problema noto il numero n di dischi che compongono la torre[4]. Per far ció, osserviamo che, se abbiamo un solo disco, la soluzione é semplice: basta spostarlo dal palo 1 al palo 3. Questo caso costituirá la *base* della nostra ricorsione. Se invece abbiamo n dischi, il problema puó essere risolto riconducendolo allo spostamento dei primi $n - 1$ dischi dal palo 1 al palo ausiliario 2 (e qui abbiamo *ricorsivamente* semplificato il problema, passando da un problema con n dischi ad uno con $n - 1$

[4]Si puó dimostrare che il tempo richiesto per spostare una pila di n dischi é all'incirca proporzionale a 2^n. Considerando che un computer, per quanto veloce, é in grado di *calcolare* (ma non di stampare o visualizzare) una mossa in circa un nanosecondo (10^{-9} secondi), occorrerebbero *almeno* mille anni per risolvere il problema con una torre di 64 dischi)

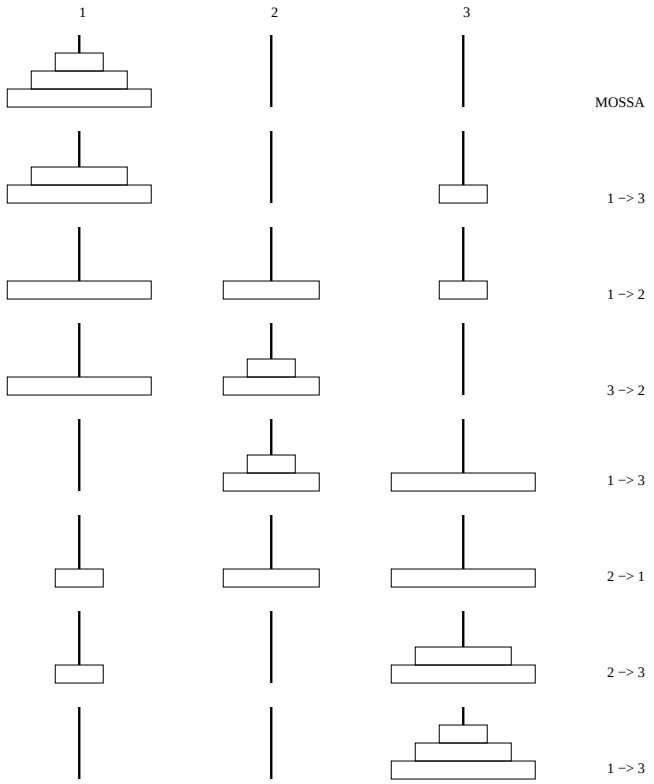

FIGURA 8. Il gioco della Torre di Hanoi con tre dischi

dischi), nello spostare l'ultimo disco rimasto sul palo 1 verso il palo 3 (*la base della ricorsione*), nello spostare gli $n-1$ dischi dal palo 2 al palo 3 (*ancora la ricorsione*). Possiamo pertanto risolvere il problema come segue:

- Sposta $n-1$ dischi dal palo di partenza (1) a quello intermedio (2)
- Sposta il disco piú grande dal palo di partenza (1) a quello di arrivo (3)
- Sposta gli $n-1$ dischi dal palo intermedio (2) al palo di arrivo (3)

La figura 9 mostra questo procedimento, per $n = 4$ dischi. Notiamo come NON ci interessa analizzare lo spostamento del gruppo di $n-1 = 3$ dischi, in quanto questo é *ricorsivamente* riconducibile allo spostamento di 2 dischi, che a sua volta é riconducibile alla *base della ricorsione* (lo spostamento di un disco).

Scriviamo ora la procedura che implementa la nostra ricorsione:

```
WITH Ada.text_io, Ada.integer_text_io;
USE Ada.text_io, Ada.integer_text_io;

PROCEDURE spostatorre (numdischi: IN natural; -- numero dischi
                       part: IN natural;      -- numero palo di partenza
                       arrivo: IN natural;    -- numero palo di arrivo
                       ausil: IN natural) IS  -- numero palo ausiliario
BEGIN
   -- posizione iniziale (esempio con 4 dischi)
   --    (palo: part)      (palo: ausil)        (palo: arrivo)
   --
   --         +
   --        ooo
   --       xxxxx
```

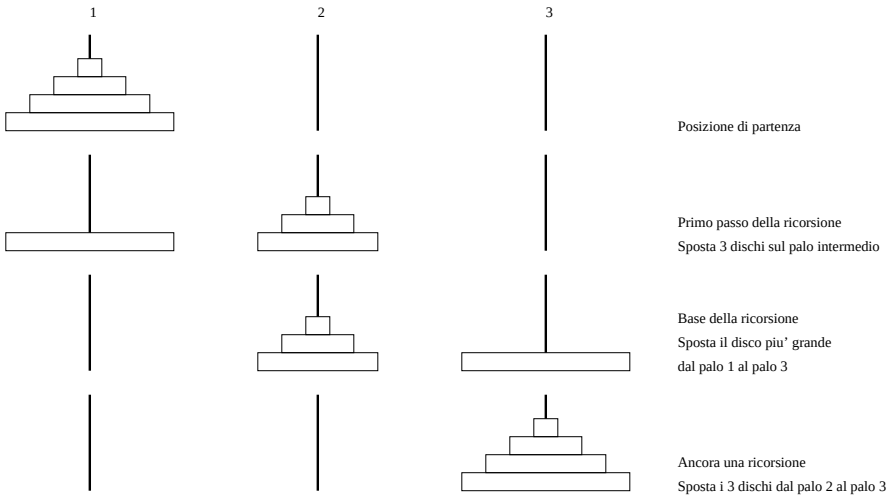

FIGURA 9. Soluzione ricorsiva del gioco

```
--      *******

IF numdischi > 0 THEN
      spostatorre(numdischi-1, part, ausil, arrivo);
          -- chiamata ricorsiva per spostare tutti i dischi, tranne
          -- l'ultimo (il piu' grande) dal palo di partenza al
          -- palo ausiliario, utilizzando come palo intermedio quello
          -- di arrivo

--    (palo: part)      (palo: ausil)        (palo: arrivo)
--
--                            +
--                           ooo
--      *******             xxxxx

      put(part, WIDTH =>2); put(" => "); put(arrivo, WIDTH=>2);
      new_line;
          -- base della ricorsione: sposta il disco piu' grande
          -- dal palo di partenza a quello di arrivo

--    (palo: part)      (palo: ausil)        (palo: arrivo)
--
--                            +
--                           ooo
--                          xxxxx                *******

      spostatorre(numdischi-1, ausil, arrivo, part);
          -- seconda chiamata ricorsiva per spostare la pila formata
          -- da numdischi-1 dischi dal palo ausiliario a quello di
          -- arrivo (sopra il disco piu' grande), utilizzando come
          -- palo intermedio quello di partenza

--    (palo: part)      (palo: ausil)        (palo: arrivo)
--
--                                                 +
--                                                ooo
```

```
   --                                                    xxxxx
   --                                                   *******

      END IF;
END spostatorre;
```

Per porterla provare, occorre una procedura che chieda il numero di dischi che compongono la torre, e che chiama la procedura `spostatorre` per risolvere il problema. Eccola:

```
-- Torre di Hanoi

WITH Ada.text_io, Ada.integer_text_io, spostatorre;
USE Ada.text_io, Ada.integer_text_io;

PROCEDURE hanoi IS
   paloiniz: CONSTANT natural := 1;   -- numero palo di partenza
   palointerm: CONSTANT natural := 2; -- numero palo ausiliario
   palofinale: CONSTANT natural := 3; -- numero palo finale
   numero_dischi: natural;  -- numero totale di dischi della torre
   -- il tipo natural rappresenta interi positivi (numeri naturali)

BEGIN
   -- introduzione numero dischi della torre

   put("Quanti sono i dischi della torre ? ");
   get(numero_dischi);

   -- chiamata della procedura per la risoluzione del problema
   spostatorre(numero_dischi, paloiniz, palofinale, palointerm);

   EXCEPTION
      WHEN constraint_error => put_line("Dati errati");

END hanoi;
```

Vediamo ora una applicazione matematica della ricorsione: la generazione dei polinomi di Hermite, di Legendre e della funzione di Ackermann. La definizione del polinomio di Hermite $H_n(x)$ di grado n nella variabile reale x, fa uso della ricorsione:

- Se $n = 0$ allora $H_0(x) = 1$
- Se $n = 1$ allora $H_1(x) = 2x$ (i primi due casi costituiscono la *base* della ricorsione
- Se $n > 1$ allora $H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x)$ (questa formula di *ricorsione* definisce il polinomio di grado $n > 1$ in funzione dei polinomi di grado $n - 1$ e di grado $n - 2$.

Prima di procedere nella scrittura di un programma, osserviamo che $H_0(x)$ é rappresentabile, nel piano cartesiano, con una retta di ordinata $y = 1$, che $H_1(x)$ é una retta con pendenza uguale a 2 (infatti per incrementando x di 1, $y = H_n(x)$ si incrementa di 2), che $H_2(x)$ é rappresentato da una parabola (é un polinomio di secondo grado), e cosí via. Piú precisamente, applicando le regole della ricorsione, possiamo facilmente ottenere le espressioni dei polinomi: lo facciamo qui sino a quello di grado tre:

$$H_0(x) = 1$$
$$H_1(x) = 2x$$
$$H_2(x) = 2xH_1(x) - 2(2-1)H_0(x) = 2x \cdot 2x - 2 \cdot 1 = 4x^2 - 2$$

$$H_3(x) = 2xH_2(s) - 2(3-1)H_1(x) = 2x(4x^2 - 2) - 4 \cdot 2x = 8x^3 - 12x$$

Scriviamo ora una funzione che ha come parametri il grado `n` del polinomio (`natural`) ed il valore `x` della variabile indipendente (`float`) e che calcoli e ritorni il valore della variabile indipendente $y = H_n(x)$.

```
FUNCTION hermite (n : natural; x : float) RETURN float IS
BEGIN
   IF n = 0
      THEN RETURN 1.0;   -- polinomio di grado 0, base della ricorsione
   ELSIF n = 1
      THEN RETURN 2.0*x; -- polinomio di grado 1, base della ricorsione
   ELSE
      RETURN 2.0*x*hermite(n-1, x) - 2.0*float(n - 1)*hermite(n-2, x);
                  -- grado maggiore di 1, ricorsione
                  -- notiamo che NON possiamo moltiplicare un numero
                  -- reali (2.0) per un intero (n-1); per questo motivo
                  -- occorre convertire il numero intero (n-1) in un
                  -- reale mediante la FUNZIONE di conversione float(n-1)
   END IF;
END hermite;
```

Per provarla, scriviamo una procedura che chiede di introdurre il grado del polinomio ed il valore della variabile indipendente, che chiama la funzione `hermite` e che visualizza il risultato:

```
WITH Ada.text_io, Ada.integer_text_io, Ada.float_text_io, hermite;
USE Ada.text_io, Ada.integer_text_io, Ada.float_text_io;

PROCEDURE polinomio IS
  x : float;   -- variabile indipendente
  n : natural; -- grado del polinomio
  y : float;   -- variabile dipendente
BEGIN
  put("Introduci il grado del polinomio : "); get(n);
  put("Introduci il valore di x : "); get(x);
     -- n.b. non sono stati messi, per brevita', i controlli
     -- sull'esattezza dei valori introdotti (il grado deve
     -- essere intero e maggiore o uguale a zero)
  y := hermite(n, x);
     -- chiamata della funzione
  put("y = "); put(y, EXP => 0, FORE => 4, AFT => 4);
     -- visualizzazione del risultato
END polinomio;
```

Passiamo ora alla scrittura di una funzione ricorsiva per il calcolo dei polinomi di Legendre (che confronteremo poi con una versione NON ricorsiva). Questi polinomi sono definiti (in modo molto simile ai polinomi di Hermite) come segue:

- Se $n = 0$ allora $P_0(x) = 1$
- Se $n = 1$ allora $P_1(x) = x$
- Se $n > 1$ allora $P_n(x) = [(2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x)]/n$

Anche qui, i primi due casi costituiscono la *base* della ricorsione, mentre il terzo rappresenta la *ricorsione* in grado di definire i polinomi di grado maggiore o uguale a 2 a partire da quelli di grado inferiore. Non é difficile scrivere la funzione, del tutto simile alla precedente:

```
FUNCTION legendre (n : natural; x : float) RETURN float IS
BEGIN
  IF n = 0
    THEN RETURN 1.0;   -- polinomio di grado 0, base della ricorsione
```

```
  ELSIF n = 1
    THEN RETURN x;     -- polinomio di grado 1, base della ricorsione
  ELSE RETURN
    (float(2*n-1)*x*legendre(n-1,x)-float(n-1)*legendre(n-2,x))/float(n);
                  -- grado maggiore di 1, ricorsione
                  -- notiamo che NON possiamo moltiplicare un numero
                  -- reali (2.0) per un intero (n-1); per questo motivo
                  -- occorre convertire il numero intero (n-1) in un
                  -- reale mediante la FUNZIONE di conversione float(n-1)
  END IF;
END legendre;
```

Passiamo ora alla scrittura della stessa funzione, in forma NON ricorsiva (lo stesso lavoro lo possiamo fare per i polinomi di Hermite)

```
FUNCTION legendre_nr (n : natural; x : float) RETURN float IS
  -- versione NON ricorsiva
  preced, attuale, prossimo : float;
BEGIN
  IF n = 0
    THEN RETURN 1.0; -- polinomio di grado zero
  ELSIF n = 1
    THEN RETURN x;   -- polinomio di grado 1
  ELSE               -- se il grado e' maggiore di 1 ricorriamo ad un
    preced := 1.0;   -- ad un CICLO al posto della ricorsione (come
    attuale := x;    -- abbiamo gia' fatto per il calcolo del fattoriale)
    FOR c IN 2 .. n LOOP
      prossimo:=(float(2*c-1)*x*attuale-float(c-1)*preced)/float(c);
      preced:=attuale;
      attuale:=prossimo;
    END LOOP;
    RETURN prossimo;
  END IF;
END legendre_nr;
```

Notiamo come la soluzione ricorsiva sia piú semplice (Il numero di righe di codice é decisamente minore, non ci sono variabili locali, e risulta piú chiaro il significato delle istruzioni e delle variabili). Occorre tuttavia notare, che quando sono richieste prestazioni di velocitá e di ridotta occupazione di memoria, é da preferirsi la soluzione non ricorsiva, sebbene il suo codice sia piú complesso, piú lungo e meno vicino al linguaggio naturale. Infine, per esercizio, scriviamo una definizione ricorsiva della funzione di Ackermann $Ack(m, n)$, ove m ed n sono due interi non negativi, cosí definita:

- Se $m = 0$ allora $Ack(0, n) = n + 1$ (base della ricorsione)
- altrimenti se $n = 0$ allora$Ack(m, 0) = Ack(m - 1, 1)$ (ricorsione)
- altrimenti $Ack(m, n) = Ack(m - 1, Ack(m, n - 1))$ (ricorsione)

Ecco la funzione:

```
FUNCTION ackermann (m : natural; n : natural) RETURN natural IS
BEGIN
  IF m = 0
     THEN RETURN n + 1;  -- base della ricorsione
  ELSIF n = 0
     THEN RETURN ackermann(m - 1, 1); -- ricorsione, caso in cui n = 0
  ELSE
     RETURN ackermann(m - 1, ackermann(m, n - 1));
                                      -- ricorsione, negli altri casi
  END IF;
END ackermann;
```

Proviamola con la seguente procedura:

```
WITH Ada.text_io, Ada.integer_text_io, ackermann;
USE Ada.text_io, Ada.integer_text_io;

PROCEDURE prova_ack IS
   n : natural;
   m : natural;
   risultato : natural;
BEGIN
   put("Introduci m : "); get(m);
   put("Introduci n : "); get(n);
      -- n.b.: nessun controllo sui dati inseriti
   risultato := ackermann(m, n);
   put("Il risultato vale : "); put(risultato, WIDTH => 5);
END prova_ack;
```

É istruttivo provarla: cominciando con valori bassi di m ed n, ad esempio con $m = 0$, $n = 1$, con $m = 1$, $n = 1$, ..., con $m = 2$, $n = 2$, e poi, magari con $m = 3$, $n = 5$ (E se si ha pazienza, aumentando ancora un poco i valori di m e di n (ad esempio, con $m = 3$ ed $n = 6$). Cosa succede al crescere di m e di n? Perché? Se avete fatto questo esperimento, vi accorgerete quanto tempo é necessario per il calcolo della funzione di Ackermann, al crescere dei due parametri. Ció é dovuto alle molteplici chiamate ricorsive della funzione stessa. Osserviamo che la funzione di Ackermann cresce molto velocemente, ad esempio si ha:

$$Ack(0,0) = 1$$

$$Ack(1,1) = 3$$

$$Ack(2,2) = 7$$

$$Ack(3,3) = 61$$

$$Ack(4,4) = 2^{2^{2^{2^{16}}}}$$

e per lo stesso motivo il tempo di calcolo cresce assai rapidamente. Riprenderemo in esame la ricorsione in altri capitoli, analizzando gli algoritmi di ordinamento e di backtracking.

13. I flussi

L'utilizzo della tastiera e del video come unici dispositivi di ingresso e di uscita è limitante, in quanto molte volte si desidera leggere dei dati da un archivio (ad esempio, da un hard-disk), e memorizzare l'output sempre su una memoria permanente, oppure scriverlo su carta mediante una stampante. Per questo motivo, risulta utile introdurre il concetto di *flusso*, traduzione corretta dall'inglese *file* (erroneamente tradotto in italiano come *archivio*). Si tratta in pratica di considerare l'input di un elaboratore come un flusso di dati da un dispositivo (la tastiera, il mouse, uno scanner, un disco) verso l'elaboratore, e come output un altro flusso di dati dall'elaboratore verso un apposito dispositivo (il video, la stampante, un disco, un plotter). I dispositivi più comuni, la tastiera ed il video (ai quali si accede con le funzioni `get` e `put`, sono detti i dispositivi *standard* di ingresso e di uscita; gli altri dispositivi sono detti *periferici* o *ausiliari*.

13.1. **I tipi di flusso.** Un flusso è una *struttura organizzata* contenente delle informazioni, memorizzate sotto forma di dati, che possono essere lette, oppure scritte mediante le operazioni che saranno illustrate nei prossimi paragrafi.

Una tipica memorizzazione di dati può avvenire su hard-disk, su nastro magnetico, su CD o su DVD. L'accesso ai dati avviene in modalità differenti a seconda di come è organizzato il flusso, e dal tipo di supporto fisico impiegato.

SI ha un *accesso sequenziale* quando il flusso viene letto (oppure scritto) sequenzialmente, partendo dal primo elemento, sino a raggiungere (o scrivere) l'elemento desiderato. Si pensi ad esempio ad una cassetta musicale o del videoregistratore; per poter accedere ad una data canzone, oppure ad una particolare scena di un film, occorre scorrere sequenzialmente il nastro sino a raggiungere la posizione desiderata. Sebbene i dati informatici siano di solito memorizzati in forma numerica (come 0 ed 1), mentre i dati presenti su cassette musicali e nastri del videoregistratore sono solitamente in formato analogico, il concetto di scansione sequenziale rimane invariato.

Si ha invece un *accesso diretto* quando è possibile accedere alla posizione desiderata, per leggere l'elemento che ci interessa, o scriverlo nella posizione opportuna. Si pensi ad esempio ad un CD musicale; scegliendo sul lettore CD il numero della canzone desiderata, è possibile ascoltarla senza dover scorrere su CD tutti i dati relativi alle altre canzoni.

Infine, si ha un *accesso mediante chiave* od *indice* quando si ha una chiave che permette di accedere all'elemento desiderato. Si pensi ad esempio all'indice analitico di un libro. Cercando nell'indice la parola chiave desiderata, si puó risalire al numero di pagina in cui questa parola compare (la chiave), e successivamente leggere la pagina desiderata.

I flussi si dividono poi in *flussi di testo*, che contengono caratteri e simboli leggibili e stampabili, e *binari*, che contengono i dati codificati mediante una successione di 0 ed 1; quest'ultimi sono normalmente non leggibili, e vengono manipolati da programmi adatti allo scopo. Ad esempio, un sorgente Ada con estensione `.adb` è un flusso di testo, che può essere letto, scritto, modificato, visualizzato e stampato in forma leggibile, mentre il programma eseguibile ottenuto dopo la compilazione è un flusso binario (che puó essere eseguito dal sistema operativo). Allo stesso modo, una immagine digitale è un flusso binario, in quanto ciascun punto dell'immagine è codificato con un numero binario. Per poter visualizzare un'immagine, occorrono infatti degli appositi programmi che siano in grado di interpretare i dati codificati come successione di 0 e di 1.

13.2. **La dichiarazione di un flusso.** In preparazione.

13.3. **Lettura e scrittura con i flussi.** In preparazione.

Riferimenti bibliografici

[1] Do-While Jones. *Ada in Action*, Mc-Graw Hill, New York, (1989).
[2] S. Johnston. *Ada-95. A guide for C and C++ programmers*, Intellectual Ammunition Department, (1995).
[3] C. Marsan. *Programmazione in Ada95*, Liceo Cantonale di Mendrisio (Svizzera), Lavoro di Maturità, (2002).
[4] R. Riehle. *Ada Distilled*, AdaWorks Software Engineering, `http://www.adaworks.com`, (2002).
[5] M. A. Smith. *Object-Oriented Software in Ada95, second edition*, Brighton, (2001).

14. Avvertenza

© Davide Tambuchi. Il presente documento puó essere distribuito e copiato liberamente, purché integralmente, gratuitamente, senza scopo di lucro, senza modifiche e citando questa avvertenza. Ogni cura è stata posta nella realizzazione di

questo documento. Tuttavia l'autore non puó assumersi alcuna responsabilitá per l'utilizzo di questa opera[5].

[5]Ultimo aggiornamento: 27 marzo 2004. Per informazioni, o per la segnalazione di errori e bugs, contattare l'autore all'indirizzo e-mail: davide.tambuchi@tin.it