```
      }
    else if (x > lista[m])
      {
        /* Si ripete la ricerca nella parte superiore.               */
        return ricercabin (lista, x, m+1, z);
      }
    else
      {
        /* m rappresenta l'indice dell'elemento cercato.             */
        return m;
      }
}

/* ================================================================= */
/* Inizio del programma.                                             */
/* ----------------------------------------------------------------- */
int main (int argc, char *argv[])
{
    /* int lista[argc-2]; */
    int *lista = (int *) malloc ((argc - 2) * sizeof (int));
    int x;
    int i;

    /* Acquisisce il primo argomento come valore da cercare.         */
    sscanf (argv[1], "%d", &x);

    /* Considera gli argomenti successivi come gli elementi          */
    /* dell'array da scandire.                                       */
    for (i = 2; i < argc; i++)
      {
        sscanf (argv[i], "%d", &lista[i-2]);
      }

    /* Esegue la ricerca.                                            */
    i = ricercabin (lista, x, 0, argc-2);

    /* Emette il risultato.                                          */
    printf ("%d si trova nella posizione %d\n", x, i);

    return 0;
}
```

289.3 Algoritmi tradizionali

In questa sezione vengono mostrati alcuni algoritmi tradizionali portati in C. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo 282.

289.3.1 Bubblesort

Il problema del Bubblesort è stato descritto nella sezione 282.4.1. Viene mostrata prima una soluzione iterativa e successivamente la funzione '**`bsort`**' in versione ricorsiva.

```
/* ================================================================= */
/* bsort <valore>...                                                 */
/* BubbleSort.                                                       */
/* ================================================================= */

#include <stdio.h>
#include <stdlib.h>

/* ================================================================= */
/* bsort (<lista>, <inizio>, <fine>)                                 */
/* ----------------------------------------------------------------- */
```

```
void bsort (int lista[], int a, int z)
{
    int scambio;
    int j;
    int k;

    /* Inizia il ciclo di scansione dell'array.                      */
    for (j = a; j < z; j++)
      {
        /* Scansione interna dell'array per collocare nella          */
        /* posizione j l'elemento giusto.                            */
        for (k = j+1; k <= z; k++)
          {
            if (lista[k] < lista[j])
              {
                /* Scambia i valori.                                 */
                scambio = lista[k];
                lista[k] = lista[j];
                lista[j] = scambio;
              }
          }
      }
}

/* ================================================================= */
/* Inizio del programma.                                             */
/* ----------------------------------------------------------------- */
int main (int argc, char *argv[])
{
    /* int lista[argc-1]; */
    int *lista = (int *) malloc ((argc - 1) * sizeof (int));
    int i;

    /* Considera gli argomenti come gli elementi                     */
    /* dell'array da ordinare.                                       */
    for (i = 1; i < argc; i++)
      {
        sscanf (argv[i], "%d", &lista[i-1]);
      }

    /* Esegue il riordino.                                           */
    bsort (lista, 0, argc-2);

    /* Emette il risultato.                                          */
    for (i = 0; i < (argc-1); i++)
      {
        printf ("%d ", lista[i]);
      }
    printf ("\n");

    return 0;
}
```

Segue la funzione '**`bsort`**' in versione ricorsiva.

```
void bsort (int lista[], int a, int z)
{
    int scambio;
    int k;

    if (a < z)
      {
        /* Scansione interna dell'array per collocare nella          */
        /* posizione a l'elemento giusto.                            */
        for (k = a+1; k <= z; k++)
          {
            if (lista[k] < lista[a])
```

```
            {
              /* Scambia i valori.                                  */
              scambio = lista[k];
              lista[k] = lista[a];
              lista[a] = scambio;
            }
        }

      bsort (lista, a+1, z);
    }
}
```

289.3.2 Torre di Hanoi

Il problema della torre di Hanoi è stato descritto nella sezione 282.4.2.

```
/* ================================================================ */
/* hanoi <n-anelli> <piolo-iniziale> <piolo-finale>                 */
/* Torre di Hanoi.                                                  */
/* ================================================================ */

#include <stdio.h>

/* ================================================================ */
/* hanoi (<n-anelli>, <piolo-iniziale>, <piolo-finale>)             */
/* ---------------------------------------------------------------- */
void hanoi (int n, int p1, int p2)
{
    if (n > 0)
      {
        hanoi (n-1, p1, 6-p1-p2);
        printf ("Muovi l'anello %d dal piolo %d al piolo %d\n", n, p1, p2);
        hanoi (n-1, 6-p1-p2, p2);
      }
}

/* ================================================================ */
/* Inizio del programma.                                            */
/* ---------------------------------------------------------------- */
int main (int argc, char *argv[])
{
    int n;
    int p1;
    int p2;

    sscanf (argv[1], "%d", &n);
    sscanf (argv[2], "%d", &p1);
    sscanf (argv[3], "%d", &p2);

    hanoi (n, p1, p2);

    return 0;
}
```

289.3.3 Quicksort

L'algoritmo del Quicksort è stato descritto nella sezione 282.4.3.

```
/* ================================================================ */
/* qsort <valore>...                                                */
/* QuickSort.                                                       */
/* ================================================================ */

#include <stdio.h>
```

```
#include <stdlib.h>

/* ================================================================ */
/* part (<lista>, <inizio>, <fine>)                                 */
/* ---------------------------------------------------------------- */
int part (int lista[], int a, int z)
{
    /* Viene preparata una variabile per lo scambio di valori.      */
    int scambio = 0;

    /* Si assume che a sia inferiore a z.                           */
    int i = a + 1;
    int cf = z;

    /* Inizia il ciclo di scansione dell'array.                     */
    while (1)
      {
        while (1)
          {
            /* Sposta i a destra.                                   */
            if ((lista[i] > lista[a]) || (i >= cf))
              {
                break;
              }
            else
              {
                i += 1;
              }
          }
        while (1)
          {
            /* Sposta cf a sinistra.                                */
            if (lista[cf] <= lista[a])
              {
                break;
              }
            else
             {
                cf -= 1;
             }
          }
        if (cf <= i)
          {
            /* È avvenuto l'incontro tra i e cf.                    */
            break;
          }
        else
          {
            /* Vengono scambiati i valori.                          */
            scambio = lista[cf];
            lista[cf] = lista[i];
            lista[i] = scambio;

            i += 1;
            cf -= 1;
          }
      }

    /* A questo punto lista[a..z] è stata ripartita e cf è la       */
    /* collocazione di lista[a].                                    */
    scambio = lista[cf];
    lista[cf] = lista[a];
    lista[a] = scambio;

    /* A questo punto, lista[cf] è un elemento (un valore) nella    */
    /* giusta posizione.                                            */
```

```
    return cf;
}

/* ================================================================ */
/* quicksort (<lista>, <inizio>, <fine>)                            */
/* ---------------------------------------------------------------- */
void quicksort (int lista[], int a, int z)
{
    /* Viene preparata la variabile cf.                             */
    int (cf) = 0;

    if (z > a)
      {
        cf = part (lista, a, z);
        quicksort (lista, a, cf-1);
        quicksort (lista, cf+1, z);
      }
}

/* ================================================================ */
/* Inizio del programma.                                            */
/* ---------------------------------------------------------------- */
int main (int argc, char *argv[])
{
    /* int lista[argc-1]; */
    int *lista = (int *) malloc ((argc - 1) * sizeof (int));
    int i;

    /* Considera gli argomenti come gli elementi                    */
    /* dell'array da ordinare.                                      */
    for (i = 1; i < argc; i++)
      {
        sscanf (argv[i], "%d", &lista[i-1]);
      }

    /* Esegue il riordino.                                          */
    quicksort (lista, 0, argc-2);

    /* Emette il risultato.                                         */
    for (i = 0; i < (argc-1); i++)
      {
        printf ("%d ", lista[i]);
      }
    printf ("\n");

    return 0;
}
```

289.3.4 Permutazioni

L'algoritmo ricorsivo delle permutazioni è stato descritto nella sezione 282.4.4.

```
/* ================================================================ */
/* permuta <valore>...                                              */
/* Permutazioni.                                                    */
/* ================================================================ */

#include <stdio.h>
#include <stdlib.h>

/* Variabile globale.                                               */
int iDimArray;

/* ================================================================ */
/* visualizza (<array>, <dimensione>)                               */
```

```
/* ---------------------------------------------------------------- */
void visualizza (int lista[], int dimensione)
{
    int i;

    for (i = 0; i < dimensione; i++)
      {
        printf ("%d ", lista[i]);
      }
    printf ("\n");
}

/* ================================================================ */
/* permuta (<lista>, <inizio>, <fine>)                              */
/* ---------------------------------------------------------------- */
void permuta (int lista[], int a, int z)
{
    int scambio;
    int k;

    /* Se il segmento di array contiene almeno due elementi, si     */
    /* procede.                                                     */
    if ((z - a) >= 1)
      {
        /* Inizia un ciclo di scambi tra l'ultimo elemento e uno    */
        /* degli altri contenuti nel segmento di array.             */
        for (k = z; k >= a; k--)
          {
            /* Scambia i valori.                                    */
            scambio = lista[k];
            lista[k] = lista[z];
            lista[z] = scambio;

            /* Esegue una chiamata ricorsiva per permutare un       */
            /* segmento più piccolo dell'array.                     */
            permuta (lista, a, z-1);

            /* Scambia i valori.                                    */
            scambio = lista[k];
            lista[k] = lista[z];
            lista[z] = scambio;
          }
      }
    else
      {
        /* Visualizza l'array e utilizza una variabile dichiarata   */
        /* globalmente.                                             */
        visualizza (lista, iDimArray);
      }
}

/* ================================================================ */
/* Inizio del programma.                                            */
/* ---------------------------------------------------------------- */
int main (int argc, char *argv[])
{
    /* int lista[argc-1]; */
    int *lista = (int *) malloc ((argc - 1) * sizeof (int));
    int i;

    /* Considera gli argomenti come gli elementi                    */
    /* dell'array da permutare.                                     */
    for (i = 1; i < argc; i++)
      {
        sscanf (argv[i], "%d", &lista[i-1]);
      }
```

```
    /* Salva la dimensione dell'array nella variabile globale.       */
    iDimArray = argc-1;

    /* Esegue le permutazioni.                                       */
    permuta (lista, 0, argc-2);

    return 0;
}
```

Automazione della compilazione: Make e file-make

La compilazione di un programma, in qualunque linguaggio sia scritto, può essere un'operazione molto laboriosa, soprattutto se si tratta di aggregare un sorgente suddiviso in più parti. Una soluzione potrebbe essere quella di predisporre uno script che esegue sequenzialmente tutte le operazioni necessarie, ma la tradizione impone di utilizzare il programma Make.

Uno dei vantaggi più appariscenti sta nella possibilità di evitare che vengano ricompilati i file sorgenti che non sono stati modificati, abbreviando quindi il tempo di compilazione necessario quando si procede a una serie di modifiche limitate.

290.1 Make

Make, per la precisione l'eseguibile '**`make`**', viene utilizzato normalmente assieme a un file, il file-make (o *makefile*), il cui nome può essere generalmente '`makefile`' o '`Makefile`', dove tra i due si tende a preferire l'ultimo con l'iniziale maiuscola. Il file-make serve a elencare a Make le operazioni da compiere e le interdipendenze che ci sono tra le varie fasi.

Make può anche essere usato da solo, senza file-make, per compilare un solo sorgente; in questo caso, tenta di determinare l'operazione da compiere più adatta, in base all'estensione del sorgente stesso. Per esempio, se esiste il file '`prova.c`' nella directory corrente, il comando

```
$ make prova
```

fa sì che '**`make`**' avvii in pratica il comando seguente:

```
$ cc -o prova prova.c
```

Se invece esistesse un file-make, lo stesso comando, '**`make prova`**', avrebbe un significato diverso, corrispondendo alla ricerca di un *obiettivo* con il nome '**`prova`**' all'interno del file-make stesso.

290.2 File-make

Un file-make è uno script specializzato per l'automazione della compilazione attraverso Make. Contiene la definizione di macro, simili alle variabili di ambiente di uno script di shell, e di *obiettivi* che rappresentano le varie operazioni da compiere.

All'interno di questi file, il simbolo '`#`' rappresenta l'inizio di un commento, cioè di una parte di testo che non viene interpretata da Make.

290.2.1 Macro

La definizione di una macro avviene in modo molto semplice, indicando l'assegnamento di una stringa a un nome che da quel momento la rappresenterà.

nome `=` *stringa*

La stringa non deve essere delimitata. Il funzionamento è molto simile alle variabili di ambiente dichiarate all'interno di uno script di shell. Per esempio,

```
prefix=/usr/local
```

definisce la macro '**prefix**' che da quel punto in poi equivale a '**/usr/local**'. La sostituzione di una macro si indica attraverso due modi possibili:

```
$(nome)
```

oppure

```
${nome}
```

come nell'esempio seguente, dove la macro '**exec_prefix**' viene generata a partire dal contenuto di '**prefix**'.

```
prefix=/usr/local
exec_prefix=$(prefix)
```

Esistono alcune macro predefinite il cui contenuto può anche essere modificato. Le più importanti sono elencate nella tabella 290.1.

Tabella 290.1. Elenco di alcune macro predefinite di Make.

Nome	Contenuto
MAKE	make
AR	ar
ARFLAGS	rw
YACC	yacc
YFLAGS	
LEX	lex
LFLAGS	
LDFLAGS	
CC	cc
CFLAGS	
FC	f77
FFLAGS	

Per verificare il contenuto delle macro predefinite, si può predisporre un file-make simile a quello seguente, eseguendo poi semplicemente '**make**' (i vari comandi '**echo**' sono rientrati con un carattere di tabulazione).

```
all:
        @echo MAKE $(MAKE) ; \
        echo AR $(AR) ; \
        echo ARFLAGS $(ARFLAGS) ; \
        echo YACC $(YACC) ; \
        echo YFLAGS $(YFLAGS) ; \
        echo LEX $(LEX) ; \
        echo LFLAGS $(LFLAGS) ; \
        echo LDFLAGS $(LDFLAGS) ; \
        echo CC $(CC) ; \
        echo CFLAGS $(CFLAGS) ; \
        echo FC $(FC) ; \
        echo FFLAGS $(FFLAGS)
```

Oltre alle macro predefinite ne esistono altre, la cui utilità si vedrà in seguito.

Tabella 290.2. Elenco di alcune macro interne.

Macro	Significato
$<	Il nome del file per il quale è stato scelto l'obiettivo per deduzione.
$*	Il nome dell'obiettivo senza suffisso.
$@	L'obiettivo della regola specificata.

290.2.2 Regole

Le regole sono il fondamento dei file-make. Attraverso di esse si stabiliscono degli *obiettivi* abbinati ai comandi necessari per ottenerli.

obiettivo... : [*dipendenza* ...]
<HT>*comando* [; *comando*] ...

La sintassi indica un comando che deve essere eseguito per raggiungere uno degli obiettivi nominati all'inizio, con le dipendenze che devono essere soddisfatte. In pratica, non si può eseguire il comando se prima non esistono i file indicati nelle dipendenze.

La dichiarazione inizia a partire dalla prima colonna, con il nome del primo obiettivo, mentre i comandi **devono** iniziare dopo un carattere di tabulazione.

L'esempio seguente mostra una regola attraverso cui si dichiara il comando necessario a eseguire il link di un programma oggetto, specificando che questo può essere eseguito solo quando esiste già il file oggetto in questione.

```
mio_prog: prova.o
        cc -o prova prova.o
```

Il comando indicato in una regola, può proseguire su più righe successive, basta concludere la riga, prima del codice di interruzione di riga, con una barra obliqua inversa (nella sezione precedente è già stato mostrato un esempio di questo tipo). Quello che conta è che le righe aggiuntive inizino sempre dopo un carattere di tabulazione.

Il comando di una regola può iniziare con un prefisso particolare:

- '**-**' fa in modo che gli errori vengano ignorati;
- '**+**' fa in modo che il comando venga eseguito sempre;
- '**@**' fa in modo che il testo del comando non venga mostrato.

290.3 Regole deduttive

Make prevede alcune regole predefinite, o deduttive, riferite ai suffissi dei file indicati come obiettivo. Si distingue tra due tipi di regole deduttive: a suffisso singolo e a suffisso doppio. La tabella 290.3 ne riporta alcune per chiarire il concetto.

Tabella 290.3. Elenco di regole deduttive a singolo e a doppio suffisso.

Obiettivo	Comando corrispondente
.c	$(CC) $(CFLAGS) $(LDFLAGS) -o $@ $<
.f	$(FC) $(FFLAGS) $(LDFLAGS) -o $@ $<
.c.o	$(CC) $(CFLAGS) -o $<
.f.o	$(FC) $(FFLAGS) -o $<

290.4 File-make tipico

Il file-make tipico, permette di automatizzare tutte le fasi legate alla ricompilazione di un programma e alla sua installazione. Si distinguono alcuni obiettivi comuni, usati di frequente:

- '**`all`**'
  utile per definire l'azione da compiere quando non si indica alcun obiettivo;
- '**`clean`**'
  per eliminare i file oggetto e i binari già compilati;
- '**`install`**'
  per installare il programma eseguibile dopo la compilazione.

Si ricorderà che le fasi tipiche di un'installazione di un programma distribuito in forma sorgente sono appunto:

```
# make
```

che richiama automaticamente l'obiettivo '**`all`**' del file-make, coincidente con i comandi necessari per la compilazione del programma, e

```
# make install
```

che provvede a installare gli eseguibili compilati nella loro destinazione prevista.

Supponendo di avere realizzato un programma, denominato '`mio_prog.c`', il cui eseguibile debba essere installato nella directory '`/usr/local/bin/`', si potrebbe utilizzare un file-make composto come l'esempio seguente:

```
prefix=/usr/local
bindir=${prefix}/bin

all:
        cc -o mio_prog mio_prog.c

clean:
        rm -f core *.o mio_prog

install:
        cp mio_prog $(bindir)
```

Come si può osservare, sono state definire le macro '**`prefix`**' e '**`bindir`**' in modo da facilitare la modifica della destinazione del programma installato, senza intervenire sui comandi.

L'obiettivo '**`clean`**' elimina un eventuale file '`core`', generato da un errore irreversibile durante l'esecuzione del programma, probabilmente mentre lo si prova tra una compilazione e l'altra, quindi elimina gli eventuali file oggetto e infine l'eseguibile generato dalla compilazione.

Parte lvi

Pascal

Pascal: preparazione di Pascal-to-C

Pascal-to-C, [1] è una sorta di compilatore che permette di convertire un sorgente Pascal in un sorgente C. I problemi che possono sorgere da questo tipo di conversione sono nella definizione precisa del tipo di dialetto Pascal e del tipo di dialetto C. Utilizzando Pascal-to-C con GNU/Linux, non si dovrebbero avere difficoltà con il compilatore C. Quello che resta da sistemare è la definizione del dialetto Pascal che si vuole usare, dal momento che ne esistono di diversi, che alle volte sono incompatibili.

Questi dettagli possono essere controllati e configurati; quello che conta è esserne consapevoli e approfondire l'uso di Pascal-to-C attraverso lo studio della documentazione originale, quando se ne presenta la necessità, ovvero quando si intende programmare seriamente attraverso questo strumento.

Il nome di Pascal-to-C è indicato dal suo autore come P2c. Tuttavia, P2C è anche il nome di un altro compilatore analogo, realizzato per sistemi speciali: <*http://www.geocities.com/SiliconValley/Network/3656/rocket/*>. In questo secondo caso, oltre alla particolarità del compilatore stesso, c'è da considerare il fatto che non si tratta di software libero.

291.1 Librerie e compilazione

Il codice C generato da Pascal-to-C contiene sempre l'inclusione del file '`p2c/p2c.h`', che poi, a sua volta, provvede a includere il solito '`stdio.h`'.

Il link del file generato dalla compilazione del sorgente C che si ottiene, deve essere fatto includendo la libreria '`libp2c.a`', cosa che si traduce generalmente nell'uso dell'opzione '**`-lp2c`**'.

In pratica, le fasi necessarie a ottenere un programma eseguibile si riassumono nei due comandi seguenti.

```
p2c sorgente_pascal
cc -lp2c sorgente_c
```

L'eseguibile che si ottiene, richiede la presenza della libreria dinamica '`libp2c.so`'.

291.2 Configurazione

Il funzionamento predefinito di '**`p2c`**' può essere configurato attraverso una serie di file di configurazione:

1. '`/usr/lib/p2c/p2crc`', '**`$P2CRC`**'
2. '`~/p2crc`'
3. '`~/.p2crc`'

Il primo file dell'elenco è quello usato per definire la configurazione generale. Eventualmente, si può usare la variabile di ambiente '**`P2CRC`**', contenente il percorso assoluto per raggiungere un file analogo, sostituendosi in tal modo a quello generale.

Dopo il file di configurazione generale, viene cercato il file '`p2crc`' nella directory personale dell'utente, oppure, in sua mancanza, il file '`.p2crc`'. Questo file serve a definire una personalizzazione della configurazione di '**`p2c`**'.

[1] **Pascal-to-C** GNU GPL

291.2.1 Direttive dei file

Le direttive di questo file di configurazione sono rappresentate da assegnamenti, espressi in una delle due forme seguenti.

nome = *valore*
nome *valore*

I commenti si rappresentano come di consueto facendoli precedere dal simbolo '`#`', dove le righe vuote o bianche vengono semplicemente ignorate.

Il file di configurazione che accompagna Pascal-to-C, cioè '`/usr/lib/p2c/p2crc`', contiene l'elenco completo di tutte le direttive utilizzabili, tutte impostate nel modo più conveniente per l'uso normale e tutte debitamente commentate in modo da sapere come può essere modificato ogni valore.

Esempi

`Language Turbo`

Definisce l'utilizzo di un sorgente TURBO Pascal.

291.2.2 Direttive incorporate nel sorgente Pascal

Le direttive di configurazione possono anche essere incorporate all'interno dello stesso sorgente Pascal, permettendo così una definizione dinamica, riferita a porzioni di codice. Per farlo, si utilizza una forma speciale dei commenti Pascal (le parentesi graffe fanno parte della direttiva).

{*nome*=*valore*}

In tal caso, come si può vedere, il simbolo '`=`' è obbligatorio e l'uso di spazi bianchi è generalmente inammissibile. È possibile l'utilizzo di commenti anche all'interno di direttive espresse in questo modo. Per farlo, occorre usare la sequenza '`##`'.

La configurazione dinamica all'interno del sorgente, permette di utilizzare anche altre modalità di assegnamento e di eliminazione automatica delle definizioni alla fine del sorgente. Per approfondirle, conviene consultare la documentazione originale, cosa che si riduce in pratica alla lettura di *p2c(1)*.

Esempi

`{Language=Turbo}`

Definisce l'utilizzo di un sorgente TURBO Pascal.

`{Language=Turbo ## utilizza una codifica TURBO Pascal}`

Definisce l'utilizzo di un sorgente TURBO Pascal e vi aggiunge un commento interno.

291.2.3 Alcune direttive importanti

Le direttive della configurazione di Pascal-to-C sono numerose; anche se l'impostazione predefinita si adatta alle situazioni più comuni, potrebbe essere conveniente modificarne alcune, già le prime volte che si utilizza Pascal-to-C.

`AnsiC` [`0`|`1`]

Permette di definire il tipo di dialetto C da utilizzare. Se si attiva la modalità, utilizzando il valore uno, si fa in modo di generare codice C ANSI; se invece non si inserisce, o si utilizza il valore zero, si ottiene un codice compatibile con il C K&R originale.

Come accennato, se non si definisce diversamente, si ottiene un codice C tradizionale, mentre potrebbe essere desiderabile di generare codice C ANSI.

```
Language [HP|HP-UX|Turbo|UCSD|VAX|Oregon|Berk|Modula]
```

Permette di definire il dialetto Pascal utilizzato come sorgente per la conversione. Le varie parole chiave usate per distinguere i dialetti hanno il valore seguente:

- '**HP**'

  È la codifica usata in modo predefinito e si riferisce precisamente al Pascal HP, compatibile con il Pascal dello standard ISO.

- '**HP-UX**'

  È il Pascal HP del sistema HP-UX, praticamente identico al Pascal HP normale.

- '**Turbo**'

  TURBO Pascal 5.0, quello usato con il Dos. La differenza rispetto al tipo HP è minima, tanto che generalmente non è necessario richiedere esplicitamente questo tipo di codifica, quando si usano sorgenti TURBO Pascal.

- '**UCSD**'

  UCSD Pascal. Si tratta di un dialetto molto simile al TURBO Pascal.

- '**MPW**'

  Macintosh Programmer's Workshop Pascal 2.0, senza le estensioni Object Pascal.

- '**VAX**'

  VAX/VMS Pascal 3.5. Non tutte le funzionalità sono disponibili.

- '**Oregon**'

  Oregon Software Pascal/2.

- '**Berk**'

  Berkeley Pascal con le estensioni Sun.

- '**Modula**'

  Modula-2. Basato sul libro *Programming in Modula-2* di Wirth, terza edizione. La conversione in C a partire da questo formato non è ancora completa.

```
ShortOpt [0|1]
```

Permette di definire il modo con cui devono essere valutate le espressioni logiche: uno abilita il «cortocircuito» attraverso cui si valutano effettivamente solo le condizioni strettamente necessarie a determinare il risultato finale; zero lo disabilita, in modo che tutte le condizioni vengano valutate in ogni caso.

291.3 Uso di Pascal-to-C

La conversione del sorgente Pascal in linguaggio C avviene per mezzo del programma '**p2c**', configurato come descritto nelle sezioni precedenti.

'**p2c**' è effettivamente un compilatore, il cui risultato è un programma C. Questo significa che genera da solo la segnalazione di errori di sintassi nel sorgente Pascal e, alla fine, il sorgente C che si ottiene dovrebbe essere corretto (dal punto di vista del C).

291.3.1 $ p2c

`p2c` [*opzioni*] [*file*]

'**p2c**' legge il file indicato come argomento, oppure lo standard input in sua mancanza. In base alle opzioni e alla configurazione definita, genera da quel file una trasformazione in linguaggio C.

Il nome del file generato si ottiene togliendo l'eventuale estensione precedente e aggiungendo '`.c`'.

Alcune opzioni

`-o` *file*

Definisce esplicitamente il nome del file del sorgente C da generare.

`-c` *file_di_configurazione*

Definisce il nome di un file di configurazione da utilizzare al posto di quelli standard.

`-a`

Genera codice C ANSI. Questa opzione permette di sostituirsi agevolmente alla configurazione standard secondo cui il sorgente generato dovrebbe essere di tipo tradizionale (K&R).

`-l` {`HP`|`HP-UX`|`Turbo`|`UCSD`|`VAX`|`Oregon`|`Berk`|`Modula`}

Permette di definire il tipo di Pascal nel sorgente. Le caratteristiche abbinate alle varie parole chiave sono state descritte in occasione della descrizione dei file di configurazione.

Esempi

$ **p2c mio_programma.pas**

Genera il file '`mio_programma.c`' convertendo il contenuto di '`mio_programma.pas`'.

$ **p2c -a mio_programma.pas**

Come nell'esempio precedente, ma genere un programma C secondo lo standard ANSI.

$ **p2c -a -o mio.c mio_programma.pas**

Come nell'esempio precedente, ma il file generato è '`mio.c`'.

291.3.2 Esempio di compilazione

Si suppone di volere compilare il programma seguente:

```
{
    CiaoMondo.pas
    Programma elementare di visualizzazione di un messaggio
    attraverso lo standard output.
}

program CiaoMondo;

begin
    Writeln('Ciao Mondo!');
end.
```

Se il file si chiama '`CiaoMondo.pas`', si può trasformare in C con il comando seguente:

```
$ p2c CiaoMondo.pas[ Invio ]

CiaoMondo

Translation completed
```

Si ottiene così il file '`CiaoMondo.c`', mostrato di seguito.

```
/* Output from p2c, the Pascal-to-C translator */
/* From input file "CiaoMondo.pas" */


/*
    CiaoMondo.pas
    Programma elementare di visualizzazione di un messaggio
    attraverso lo standard output.
*/


#include <p2c/p2c.h>


main(argc, argv)
int argc;
Char *argv[];
{
  PASCAL_MAIN(argc, argv);
  printf("Ciao Mondo!\n");
  exit(EXIT_SUCCESS);
}



/* End. */
```

Questo file può essere compilato a sua volta.

```
$ cc -lp2c -o CiaoMondo CiaoMondo.c[ Invio ]
```

Se tutto funziona correttamente, si ottiene il file '`CiaoMondo`' eseguibile.

```
$ ./CiaoMondo[ Invio ]

Ciao Mondo!
```

Se si desidera generare un sorgente C ANSI, si può usare l'opzione '**`-a`**' di '**`p2c`**'. Nel caso dell'esempio, il corpo del programma C sarebbe stato il seguente:

```
main(int argc, Char *argv[])
{
  PASCAL_MAIN(argc, argv);
  printf("Ciao Mondo!\n");
  exit(EXIT_SUCCESS);
}
```

Pascal: introduzione

Il linguaggio Pascal è nato come strumento puramente didattico, che poi si è esteso fino a raggiungere potenzialità vicine a quelle del linguaggio C.

La caratteristica più appariscente di questo linguaggio è che tutto deve essere dichiarato prima del suo utilizzo. Il vantaggio di questo tipo di approccio sta nella possibilità di escludere errori di programmazione dovuti a digitazione errata dei nomi delle variabili, perché il compilatore le rifiuta se non sono state dichiarate preventivamente.

Dal momento che di dialetti Pascal ne esistono molti, in questo capitolo si cerca di fare riferimento allo standard ANSI, anche se potrebbe essere particolarmente riduttivo. Gli esempi che vengono proposti sono stati verificati con Pascal-to-C, nella sua configurazione predefinita.

292.1 Struttura fondamentale

Il Pascal impone una struttura nella preparazione dei sorgenti. L'esempio seguente è un programma che non fa alcunché.

```
program Nulla;

begin
end.
```

Nella prima riga dell'esempio, si può osservare la definizione del nome del programma, attraverso la direttiva '**`program`**'. Il nome, in questo caso è '**`Nulla`**', non deve corrispondere necessariamente al nome del file.

Le parole chiave '**`begin`**' e '**`end`**' delimitano lo spazio utilizzato per le istruzioni del programma, che in questo caso non esistono.

Il punto finale, dopo la parola chiave '**`end`**', serve a indicare al compilatore la conclusione del programma, che può apparire solo alla fine del sorgente.

292.1.1 Istruzioni Pascal

Le istruzioni Pascal terminano con un punto e virgola ('**`;`**'), così un'istruzione può impiegare più righe senza bisogno di utilizzare simboli di continuazione, oppure, su una riga possono apparire più istruzioni (sempre separate con il punto e virgola).

È possibile raggruppare più istruzioni attraverso i delimitatori '**`begin`**' e '**`end`**': il primo dei due viene seguito dalle istruzioni senza l'uso del punto e virgola, mentre il secondo termina normalmente con un punto e virgola, oppure un punto se si tratta del delimitatore che conclude il programma.

istruzione `;`
`begin` *istruzione* `;` *istruzione* `;` *istruzione* `;` `end;`

L'istruzione nulla può essere rappresentata da un punto e virgola isolato.

292.1.2 Nomi

Secondo il Pascal standard, i nomi che servono per identificare ciò che si utilizza, come variabili, procedure o funzioni, sono composti da una lettera alfabetica, seguita da una combinazione libera di altre lettere e cifre numeriche. Secondo lo standard originale non è ammissibile l'uso del trattino basso, ma la maggior parte dei compilatori ammette anche questo carattere.

La lunghezza dei nomi dovrebbe essere libera, con la limitazione che ogni compilatore è in grado di distinguere i nomi solo in base a un numero massimo di caratteri. Il valore minimo definito dallo standard è di otto caratteri.

Per quanto riguarda i nomi, il Pascal non distingue tra maiuscole e minuscole, come invece avviene nel linguaggio C.

292.1.3 Commenti

Il Pascal consente l'utilizzo di due tipi di delimitatore per circoscrivere i commenti: le parentesi graffe ('**{**' e '**}**') e la coppia '**(***' '***)**'. Generalmente non sono ammissibili i commenti annidati, cioè quelli a più livelli.

Quello che segue è l'esempio del programma che non fa alcunché, con qualche commento.

```
{
    Ecco un programma che non fa proprio nulla.
}
program Nulla;

begin
    (* è qui che ha luogo il «nulla» *)
end.
```

Esistono due tipi di delimitatori per i commenti solo perché i primi, cioè le parentesi graffe, potevano essere difficili da ottenere nelle prime tastiere di alcuni paesi europei.

292.1.4 Suddivisione di un programma Pascal

Il linguaggio Pascal è un po' rigido per ciò che riguarda la sequenza con cui possono essere descritte le varie parti che lo compongono. Si distinguono tre parti fondamentali nel file sorgente:

1. intestazione del programma -- si tratta della dichiarazione '**program**' seguita dal nome;
2. dichiarazioni -- è lo spazio in cui si dichiara tutto ciò che viene usato nel programma, per esempio le variabili, le procedure e le funzioni;
3. istruzioni -- è lo spazio, delimitato dalle parole chiave '**begin**' '**end**', in cui si inseriscono le istruzioni del programma, ovvero è quello che in altri linguaggi di programmazione è la funzione o la procedura principale.

È il caso di osservare che i commenti possono essere collocati in ogni punto del file sorgente.

292.1.5 Output elementare

Quasi tutti gli esempi di programmazione elementare, in qualunque linguaggio di programmazione, utilizzano un'istruzione per l'output elementare.

Negli esempi che verranno mostrati inizialmente, si farà spesso uso della procedura '**`Writeln()`**', la quale si occupa semplicemente di emettere attraverso lo standard output tutti gli argomenti forniti. L'esempio seguente serve a emettere la frase «1000 volte ciao mondo!», utilizzando due parametri: la costante numerica 1000 e la stringa « volte ciao mondo!».

```
program CiaoMondo1000;

begin
    Writeln(1000, ' volte ciao mondo!');
end.
```

Si tenga presente, in ogni caso, che '**`Writeln`**' e '**`writeln`**' sono la stessa cosa.

292.2 Variabili e tipi

I tipi di dati elementari del linguaggio Pascal dipendono dal compilatore utilizzato e dall'architettura dell'elaboratore sottostante. I tipi standard del Pascal ANSI sono elencati nella tabella 292.1. Il tipo '**`char`**', non fa parte dello standard ANSI, ma è molto diffuso e così appare incluso in quella tabella.

Tabella 292.1. Elenco dei tipi di dati primitivi fondamentali in Pascal.

Tipo	Descrizione
int	Numeri interi positivi e negativi.
byte	Interi positivi di un solo byte (da 0 a 255).
real	Numeri a virgola mobile.
boolean	Valori logici booleani.
char	Carattere (generalmente di 8 bit).

292.2.1 Valori contenibili e costanti letterali

Ogni tipo di variabile può contenere un solo tipo di dati, esprimibile eventualmente attraverso una costante letterale scritta secondo una forma adatta.

I valori numerici vengono espressi da costanti letterali senza simboli di delimitazione.

- Gli interi ('**`integer`**') vanno espressi con numeri normali, senza punti di separazione di un'ipotetica parte decimale, prefissati eventualmente dal segno meno ('**`-`**') nel caso di valori negativi.
- I valori '**`byte`**' vanno espressi come gli interi positivi, con la limitazione della dimensione massima.
- I numeri reali ('**`real`**') possono essere espressi come numeri aventi una parte decimale, segnalata dalla presenza di un punto decimale.

  Se si vuole indicare un numero reale corrispondente a un numero intero, si deve aggiungere un decimale finto, per esempio, il numero 10 si può rappresentare come 10.0.

  Naturalmente è ammissibile anche la notazione esponenziale, come per esempio '**`7e-2`**' che corrisponde in pratica a $7*(10^{-2})$, pari a 0.07.

I valori logici vengono espressi dalle costanti letterali '**TRUE**' e '**FALSE**'.

I valori carattere e stringa, vengono delimitati da coppie di apici singoli, come '**'A'**', '**'B'**', ... '**'Ciao Mondo!'**'.

292.2.2 Dichiarazione delle variabili

La dichiarazione delle variabili può essere fatta esclusivamente prima di un blocco '**begin**' '**end**' di un programma, di una funzione o di una procedura.

```
var nome : tipo;
```

Dalla sintassi si vede l'utilizzo della parola chiave '**var**', seguita dal nome della variabile da definire, quindi da due punti ('**:**'), infine dalla definizione del tipo di variabile.

In realtà, è possibile anche indicare un elenco di nomi, separati da virgole, quando questi devono essere tutti dello stesso tipo; inoltre, è possibile dichiarare più variabili differenti, utilizzando la parola chiave '**var**' una sola volta.

Esempi

```
var     conta   :       integer;
```

Dichiara la variabile '**conta**' di tipo intero.

```
var     conta,canta     :       integer;
```

Dichiara le variabili '**conta**' e '**canta**' di tipo intero.

```
var     conta   :       integer;
        canta   :       integer;
```

Esattamente uguale all'esempio precedente.

```
var
        conta   :       integer;
        lettera :       char;
```

Dichiara la variabile '**conta**' di tipo intero e la variabile '**lettera**' di tipo carattere.

292.3 Operatori ed espressioni

Gli operatori sono qualcosa che esegue un qualche tipo di funzione, su uno o due operandi, restituendo un valore. Il tipo di valore restituito varia a seconda dell'operatore e degli operandi utilizzati. Per esempio, la somma di due interi genera un intero, mentre una divisione di un valore intero per un altro numero intero, genera un numero reale.

292.3.1 Operatori aritmetici

Gli operatori che intervengono su valori numerici sono elencati nella tabella 292.2.

Tabella 292.2. Elenco degli operatori aritmetici e di quelli di assegnamento relativi a valori numerici.

Operatore e operandi	Descrizione
op1 + *op2*	Somma i due operandi.
op1 - *op2*	Sottrae dal primo il secondo operando.
op1 * *op2*	Moltiplica i due operandi.
op1 / *op2*	Divide il primo operando per il secondo, il risultato è in virgola mobile.
op1 div *op2*	Divide il primo operando per il secondo generando un risultato intero.
op1 mod *op2*	Modulo: il resto della divisione tra il primo e il secondo operando.
var := *valore*	Assegna alla variabile il valore alla destra.

Una caratteristica fondamentale del Pascal è la sua attenzione nella coerenza dei tipi di dati utilizzati nelle espressioni e nelle assegnazioni. Tanto per comprendere il problema con un esempio, un compilatore non dovrebbe consentire l'assegnamento di un valore in virgola mobile in una variabile intera. Naturalmente, ogni compilatore può utilizzare una politica differente, consentendo una conversione di tipo automatica in situazioni particolari.

In ogni caso, è necessario conoscere l'uso di alcune funzioni essenziali, utili per prevenire conflitti nel tipo dei dati.

`Round(` *numero_reale* `)`
`Trunc(` *numero_reale* `)`

Le due funzioni, usate in questo modo, restituiscono un valore intero a partire da un valore a virgola mobile. Nel primo caso il numero viene arrotondato, mentre nel secondo viene semplicemente troncato al valore intero.

292.3.2 Operatori di confronto e operatori logici

Gli operatori di confronto determinano la relazione tra due operandi. Il risultato dell'espressione composta da due operandi messi a confronto è di tipo booleano, rappresentabile in Pascal con le costanti '**TRUE**' e '**FALSE**'. Gli operatori di confronto sono elencati nella tabella 292.3.

Tabella 292.3. Elenco degli operatori di confronto. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
op1 = *op2*	*Vero* se gli operandi si equivalgono.
op1 != *op2*	*Vero* se gli operandi sono differenti.
op1 < *op2*	*Vero* se il primo operando è minore del secondo.
op1 > *op2*	*Vero* se il primo operando è maggiore del secondo.
op1 <= *op2*	*Vero* se il primo operando è minore o uguale al secondo.
op1 >= *op2*	*Vero* se il primo operando è maggiore o uguale al secondo.

Quando si vogliono combinare assieme diverse espressioni logiche, comprendendo in queste anche delle variabili che contengono un valore booleano, si utilizzano gli operatori logici. Gli operatori logici sono elencati nella tabella 292.4.

Tabella 292.4. Elenco degli operatori logici. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
not *op*	Inverte il risultato logico dell'operando.
op1 and *op2*	*Vero* se entrambi gli operandi restituiscono il valore *Vero*.
op1 or *op2*	*Vero* se uno o entrambi gli operandi restituiscono il valore *Vero*.

Nel Pascal tradizionale, le espressioni logiche vengono valutate in ogni parte, prima di definire il risultato finale di un operatore AND o di un operatore OR. Dal momento che questo metodo di risoluzione è inutilmente dispersivo, spesso i compilatori Pascal consentono di ottenere il «cortocircuito», attraverso cui si valutano solo le parti dell'espressione che sono indispensabili per arrivare al risultato finale.

292.4 Strutture di controllo del flusso

Il linguaggio Pascal gestisce un buon numero di strutture di controllo di flusso, compreso il salto *go-to* che comunque è sempre meglio non utilizzare e qui, volutamente, non viene presentato.

Le strutture di controllo permettono di sottoporre l'esecuzione di una parte di codice alla verifica di una condizione, oppure permettono di eseguire dei cicli, sempre sotto il controllo di una condizione. La parte di codice che viene sottoposta a questo controllo, può essere un'istruzione singola, oppure un gruppo di istruzioni. Nel secondo caso, quasi sempre, è necessario delimitare questo gruppo attraverso l'uso di '**`begin`**' e '**`end`**'.

Dal momento che è comunque consentito di realizzare un gruppo di istruzioni che in realtà ne contiene una sola, probabilmente è meglio utilizzare sempre i delimitatori '**`begin`**' '**`end`**', a vantaggio dello stile e della leggibilità del codice.

292.4.1 if

`if` *condizione* `then` *istruzione*
`if` *condizione* `then` *istruzione* `else` *istruzione*

Se la condizione si verifica, viene eseguita l'istruzione (o il gruppo di istruzioni) seguente; quindi il controllo passa alle istruzioni successive alla struttura. Se viene utilizzato '**`else`**', nel caso non si verifichi la condizione, viene eseguita l'istruzione che ne segue.

Vengono mostrati alcuni esempi.

```
...
var     importo : integer;
...
if importo > 10000000 then Writeln( 'offerta vantaggiosa' );
```

```
...
var     importo         : integer;
        memorizza       : integer;
...
if importo > 10000000 then
begin
    memorizza := importo;
    Writeln( 'offerta vantaggiosa' );
end
else
    Writeln( 'meglio lasciar perdere' );
```

```
...
var     importo         : integer;
        memorizza       : integer;
...
if importo > 10000000 then
begin
    memorizza := importo;
    Writeln( 'offerta vantaggiosa' );
end
else if importo > 5000000 then
begin
    memorizza := importo;
    Writeln( 'offerta accettabile' );
end
else
    Writeln( 'meglio lasciar perdere' );
```

Il blocco *if-then-else* rappresenta un'unica istruzione in Pascal. In questo senso, dovrebbe apparire un punto e virgola alla fine del blocco, a terminare l'istruzione. Se si utilizzano raggruppamenti di istruzioni attraverso i delimitatori '**begin**' '**end**', le istruzioni contenute terminano con il punto e virgola, mentre il blocco, dopo la parola chiave '**end**', no, a meno che si tratti della fine dell'istruzione '**if**'.

Per osservare meglio questo particolare, si potrebbero riscrivere gli stessi esempi nel modo seguente, in cui il punto e virgola finale serve a concludere visivamente la dentellatura delle istruzioni '**if**'.

```
...
var     importo : integer;
...
if importo > 10000000 then
    Writeln( 'offerta vantaggiosa' )
;
```

```
...
var     importo         : integer;
        memorizza       : integer;
...
if importo > 10000000 then
    begin
        memorizza := importo;
        Writeln( 'offerta vantaggiosa' );
    end
else
    Writeln( 'meglio lasciar perdere' )
;
```

```
...
var     importo         : integer;
        memorizza       : integer;
...
if importo > 10000000 then
    begin
        memorizza := importo;
        Writeln( 'offerta vantaggiosa' );
    end
else
    if importo > 5000000 then
        begin
            memorizza := importo;
            Writeln( 'offerta accettabile' );
        end
    else
        Writeln( 'meglio lasciar perdere' )
;
```

292.4.2 case

La struttura di selezione si ottiene con l'istruzione '**case**'. Si tratta di una struttura un po' troppo complessa per essere rappresentata facilmente attraverso uno schema sintattico. In generale, l'istruzione '**case**' permette di eseguire una o più istruzioni in base al risultato di un'espressione. L'esempio seguente mostra la visualizzazione del nome del mese, in base al valore di un intero.

```
...
var     mese    : integer;
...
```

```
case mese of
    1  : Writeln( 'gennaio' );
    2  : Writeln( 'febbraio' );
    3  : Writeln( 'marzo' );
    4  : Writeln( 'aprile' );
    5  : Writeln( 'maggio' );
    6  : Writeln( 'giugno' );
    7  : Writeln( 'luglio' );
    8  : Writeln( 'agosto' );
    9  : Writeln( 'settembre' );
    10 : Writeln( 'ottobre' );
    11 : Writeln( 'novembre' );
    12 : Writeln( 'dicembre' );
end;
```

È importante osservare l'uso del punto e virgola, che conclude ogni istruzione richiamata dai vari casi. La parola chiave '**end**' finale, conclude la struttura.

Un gruppo di casi può essere raggruppato assieme, quando si vuole che ognuno di questi esegua lo stesso gruppo di istruzioni.

```
...
var     anno    : integer;
        mese    : integer;
        giorni  : integer;
...
case mese of
    1,3,5,7,8,10,12 :
        giorni := 31;
    4,6,9,11 :
        giorni := 30;
    2   :
        if ((anno mod 4 = 0) and not (anno mod 100 = 0)) or
                (iAnno mod 400 = 0) then
            giorni := 29
        else
            giorni := 28
        ;
end;
```

È anche possibile definire un caso predefinito che si verifichi quando nessuno degli altri si avvera.

```
...
var     mese    : integer;
...
case mese of
    1  : Writeln( 'gennaio' );
    2  : Writeln( 'febbraio' );
...
    11 : Writeln( 'novembre' );
    12 : Writeln( 'dicembre' );
else
    Writeln( 'mese non corretto' );
end;
```

Un intervallo di casi può essere indicato facilmente come nell'esempio seguente:

```
...
var     mese    : integer;
...
case mese of
    6..9 : Writeln( 'mesi caldi' );
    ...
end;
```

292.4.3 while

```
while condizione do istruzione
```

‘**while**’ esegue un’istruzione finché la condizione restituisce il valore *Vero*. La condizione viene valutata prima di eseguire l’istruzione e poi ogni volta che termina un ciclo, prima dell’esecuzione del successivo.

Come sempre, al posto della singola istruzione se ne può inserire un raggruppamento delimitato dalle parole chiave ‘**begin**’ e ‘**end**’.

L’esempio seguente fa apparire per 10 volte la lettera «x».

```
program DieciX;

var contatore   : integer;

begin
    contatore := 0;
    while contatore < 10 do
    begin
        contatore := contatore + 1;
        Writeln( 'x' );
    end;
end.
```

La struttura ‘**while**’ è un’istruzione singola in Pascal. Per sottolinearlo, si potrebbe cambiare la dentellatura dell’esempio appena mostrato per fare in modo che il punto e virgola finale, che chiude l’istruzione, inizi sulla stessa colonna della parola chiave ‘**while**’.

```
...
    contatore := 0;
    while contatore < 10 do
        begin
            contatore := contatore + 1;
            Writeln( 'x' );
        end
    ;
...
```

292.4.4 repeat-until

```
repeat istruzione ;... until condizione ;
```

La struttura ‘**repeat**’ ‘**until**’ permette di eseguire un gruppo di istruzioni una volta e poi di ripeterne l’esecuzione fino a quando la condizione posta alla fine continua a non verificarsi.

Ci sono quindi due diversità fondamentali, rispetto alla struttura ‘**while**’: il gruppo di istruzioni viene eseguito sicuramente almeno una volta; il verificarsi della condizione implica l’interruzione del ciclo.

Per quanto riguarda la sintassi usata dal Pascal, c’è da osservare che dopo la parola chiave ‘**repeat**’ possono essere collocate una serie di istruzioni, senza bisogno di un raggruppamento ‘**begin**’ ‘**end**’. In questo senso, ogni istruzione termina con il suo punto e virgola.

L’esempio seguente è solo un pretesto per mostrare il funzionamento di questa struttura: visualizza 10 volte la lettera «x».

```
program DieciX;

var contatore   : integer;

begin
```

```
    contatore := 0;
    repeat
        contatore := contatore + 1;
        Writeln( 'x' );
    until contatore = 10;
end.
```

292.4.5 for

```
for variabile; := inizio to fine do istruzione
```

L'istruzione '**for**' permette di definire un ciclo enumerativo, in cui una variabile intera viene inizializzata, quindi viene eseguita ripetitivamente l'istruzione controllata, incrementando alla fine di ogni esecuzione tale variabile e interrompendo il ciclo quando questa raggiunge il valore finale (quando la variabile ha raggiunto il valore finale, si esegue l'istruzione per l'ultima volta). L'incremento è di un'unità quando il valore finale è maggiore di quello iniziale, oppure di un'unità negativa quando il valore finale è minore di quello iniziale.

L'esempio già visto, in cui veniva visualizzata per 10 volte una «x», potrebbe tradursi nel modo seguente, attraverso l'uso di un ciclo '**for**'.

```
program DieciX;

var contatore   : integer;

begin
    for contatore := 1 to 10 do
        Writeln( 'x' )
    ;
end.
```

Come sempre, al posto di controllare una singola istruzione, se ne può gestire un gruppo, attraverso l'uso dei delimitatori '**begin**' e '**end**'. L'esempio già visto, potrebbe eventualmente tradursi nel modo seguente:

```
...
    for contatore := 1 to 10 do
        begin
            Writeln( 'x' );
        end
    ;
...
```

292.5 Procedure e funzioni

Il linguaggio Pascal distingue due tipi di subroutine: procedure e funzioni. In pratica, le procedure sono funzioni che non restituiscono alcun valore.

La dichiarazione e descrizione delle procedure e delle funzioni deve essere fatta all'interno della parte iniziale del programma, dedicata alle dichiarazioni. Procedure e funzioni possono chiamarsi a vicenda e, in ogni caso, perché la chiamata possa essere valida, occorre che la procedura o la funzione sia stata dichiarata precedentemente.

Ci sono situazioni in cui non è possibile descrivere una funzione o una procedura prima di quella chiamante. In tali casi, è possibile dichiarare una funzione senza descriverla immediatamente.

292.5.1 Struttura

Per il linguaggio Pascal, le procedure e le funzioni sono dei sottoprogrammi veri e propri, tanto che anche in questo caso si distinguono tre parti: intestazione, dichiarazioni e istruzioni. In particolare, l'intestazione può includere anche la dichiarazione, a meno che questa non sia separata per renderla visibile ad altre procedure e funzioni precedenti.

```
procedure nome [(parametro_formale [...])];
```

```
function nome [(parametro_formale [...])] : tipo;
```

La sintassi che appare sopra rappresenta la dichiarazione di una procedura e di una funzione. Come si può osservare, a parte la parola chiave iniziale, la funzione ha alla fine l'indicazione del tipo di dati che restituisce.

Se la procedura o la funzione non richiede l'indicazione di parametri, allora non è necessario specificare alcun *parametro formale* , quindi non sono necessarie nemmeno le parentesi tonde.

Dopo la dichiarazione della funzione o della procedura, vanno indicate le dichiarazioni, per esempio le variabili utilizzate, nello stesso modo già visto per il programma.

Infine vanno poste le istruzioni, all'interno di un raggruppamento '**begin**' '**end**'. A differenza del raggruppamento analogo che riguarda il blocco principale del programma, la parola chiave '**end**' è conclusa con un punto e virgola invece che con il punto.

La funzione restituisce un valore, attraverso l'assegnamento a una variabile ipotetica che ha lo stesso nome della funzione.

Esempi

```
procedure CiaoCiao;
begin
    Writeln('Ciao a tutti');
    Writeln('ciao ciao ciao');
end;
```

Si tratta di una procedura elementare che non utilizza alcun parametro e si limita a emettere un messaggio di saluto.

```
function CiaoCiao : boolean;
begin
    Writeln('Ciao a tutti');
    Writeln('ciao ciao ciao');
    CiaoCiao := TRUE;
end;
```

Si tratta di una funzione elementare che non utilizza alcun parametro e si limita a emettere un messaggio di saluto, restituendo sempre il valore booleano *Vero*.

292.5.2 Campo di azione

Sia le variabili che le procedure e le funzioni, hanno un campo di azione. Le variabili dichiarate nella parte introduttiva di un programma, prima della dichiarazione di procedure e funzioni, sono accessibili al corpo del programma e a tutte le procedure e funzioni. Le variabili dichiarate nella parte introduttiva di una procedura o di una funzione, hanno effetto locale, non essendo visibili all'esterno; se queste hanno nomi già utilizzati per le variabili globali, di fatto ne impediscono l'accesso.

Le procedure e le funzioni, in qualità di sottoprogrammi, possono contenere anche la dichiarazione di sottoprocedure e sottofunzioni. In tal caso, tali subroutine sono accessibili solo dal

codice contenuto nella procedura o funzione in cui sono dichiarate. Nello stesso modo, le variabili locali delle procedure o delle funzioni sono accessibili anche alle rispettive sottoprocedure e sottofunzioni.

292.5.3 Forward

Si è accennato al fatto che, perché una chiamata possa essere valida, occorre che la procedura o la funzione in questione sia stata dichiarata prima, cioè in una posizione precedente all'interno del sorgente.

In presenza di chiamate ricorsive tra più procedure o funzioni, diviene impossibile che ogni chiamata si riferisca sempre a qualcosa di definito e descritto in precedenza.

Per risolvere il problema, si può dichiarare una procedura o una funzione prima della sua descrizione effettiva, attraverso l'uso della parola chiave '**forward**', come nell'esempio seguente:

```
...
procedure MiaProcedura(...);
forward;
...
...
procedure MiaProcedura;
begin
    ...
end;
...
```

La dichiarazione della procedura o della funzione deve contenere la dichiarazione di tutti i parametri formali, mentre la descrizione è assente.

292.5.4 Parametri formali e chiamata per valore o per riferimento

La descrizione dei parametri formali, all'interno della dichiarazione di una procedura o di una funzione, richiede la definizione del nome delle variabili e del tipo relativo. Il campo di azione di queste variabili è locale.

```
...
procedure MiaProcedura( primo,secondo : integer;
                        terzo         : char);
begin
    ...
end;
...
```

L'esempio mostra la dichiarazione di una procedura che utilizza tre parametri formali, denominati casualmente proprio: '**primo**', '**secondo**' e '**terzo**'. I primi due sono di tipo '**integer**', mentre l'ultimo è di tipo '**char**'.

Come si può osservare, la dichiarazione dei parametri formali è molto simile alla dichiarazione delle variabili, con la differenza che ciò avviene all'interno di parentesi tonde, oltre al fatto che (per il momento) manca la parola chiave '**var**'.

Una procedura o una funzione in cui i parametri formali siano stati dichiarati in questo modo, riceve una copia dei dati nel momento della chiamata, senza poter riflettere all'indietro le modifiche che a questi dovesse applicare. Si ha in pratica una chiamata per valore.

È possibile dichiarare una procedura o una funzione in cui la chiamata sia per riferimento, in modo da riflettere all'indietro le modifiche, utilizzando la parola chiave '**var**'.

```
...
procedure MiaProcedura( primo       : integer;
                        var secondo : integer;
                        terzo       : char);
begin
    ...
end;
...
```

L'esempio mostra una variante in cui si dichiara che il secondo parametro formale, '**secondo**', riflette all'indietro le modifiche che dovessero essergli apportate all'interno della procedura.

292.5.5 Chiamata e parametri attuali

La chiamata di una procedura o di una funzione, avviene semplicemente nominandola e facendola seguire dall'indicazione dei *parametri attuali*, cioè dei valori che si vuole siano passati per l'elaborazione.

La differenza fondamentale tra procedure e funzioni sta nel fatto che le chiamate alle prime vengono utilizzate come istruzioni pure e semplici, mentre le seconde, vanno inserite all'interno di espressioni.

Merita un minimo di attenzione anche il tipo di chiamata: per valore o per riferimento. Nel primo caso, non si pongono problemi di alcun tipo, dal momento che la funzione o la procedura chiamata non può alterarli; se invece si tratta di una chiamata per riferimento, occorre fare attenzione che il parametro attuale, usato nella chiamata, non sia una costante, perché questo genererebbe un errore irreversibile.

```
...
var     MioNumero : integer;
...
procedure MiaProcedura( primo       : integer;
                        var secondo : integer;
                        terzo       : char);
begin
    ...
    secondo := 777;
    ...
end;
...
{ inizio del programma }
begin
    MiaProcedura( 123, MioNumero, 'C' );
    Writeln( MioNumero );
end.
```

L'esempio mostra una chiamata a una procedura in cui uno dei parametri deve essere chiamato per riferimento. In tal caso, il parametro attuale corrispondente utilizzato nella chiamata, è necessariamente una variabile.

292.6 I/O elementare

Per le operazioni di I/O elementare, cioè per l'utilizzo di standard output e standard error, si hanno a disposizione due coppie di procedure: '**Write()**' e '**Writeln()**'; '**Read()**' e '**Readln()**'. La prima coppia per emettere qualcosa attraverso lo standard output, la seconda per leggere qualcosa dallo standard input.

Anche se non è ancora stato affrontato l'argomento stringhe, è opportuno anticipare che per inserire un apice singolo all'interno di una costante stringa, basta indicarne due consecutivi. Per esempio, la stringa seguente,

```
'questa è la ''vera'' verità'
```

Si traduce in:

```
questa è la 'vera' verità
```

292.6.1 Write(), Writeln()

`Write(`*elemento_da_visualizzare* [`:`*dimensione* [`:`*decimali*]] [`,` ...] `)`
`Writeln(`*elemento_da_visualizzare* [`:`*dimensione* [`:`*decimali*]] [`,` ...] `)`

Le procedure '**`Write()`**' e '**`Writeln()`**' permettono di emettere attraverso lo standard output il contenuto di tutti i parametri che gli vengono forniti. A seconda dei tipi di dati utilizzati, vengono effettuate tutte le conversioni necessarie a ottenere un risultato stringa.

Se un parametro attuale, fornito nella chiamata, viene indicato seguito da due punti ('`:`') e quindi da un numero, si stabilisce lo spazio (espresso in colonne) che questo utilizzerà nell'output. Se si specifica tale dimensione, l'informazione verrà rappresentata allineandola a destra. Questa possibilità di definire la dimensione viene utilizzata prevalentemente per i dati numerici e in questo senso sta la logica dell'allineamento a destra.

Se si vuole rappresentare un valore numerico con decimali, è abbastanza importante fissare la dimensione della visualizzazione, aggiungendo anche l'indicazione delle colonne da riservare alla parte decimale. Diversamente, la rappresentazione risulterebbe in notazione esponenziale.

L'unica differenza tra le due procedure, sta nel fatto che '**`Writeln()`**' aggiunge automaticamente, alla fine della stringa visualizzata, il codice di interruzione di riga, in modo da riportare il cursore all'inizio della riga successiva.

Esempi

```
var totale : integer;
...
totale := 1950000;
...
Write('Totale:', totale:11);
```

Emette la stringa seguente, senza portare a capo il cursore alla fine.

```
Totale:    1950000
```

```
var totale : real;
...
real := 1234.5678;
...
Writeln('Totale:', totale:11:5);
```

Emette la stringa seguente, portando a capo il cursore alla fine.

```
Totale: 1234.56780
```

292.6.2 Read(), Readln()

`Read(`*variabile* [`,` ...] `)`
`Readln(`*variabile* [`,` ...] `)`

Le procedure '**`Read()`**' e '**`Readln()`**' permettono di leggere dallo standard input dei valori per le variabili che vengono indicate come parametri della chiamata. I dati inseriti, vengono distinti in

base all'inserimento di spaziature, così come avviene di solito con gli argomenti di un comando del sistema operativo.

È importante che i dati inseriti siano compatibili con il tipo delle variabili utilizzate, altrimenti si rischia di ottenere un errore irreversibile durante il funzionamento del programma.

La differenza tra le due procedure sta nel fatto che '**`Readln()`**' dovrebbe restituire l'eco del codice di interruzione di riga, quando si preme [*Invio*] per concludere l'inserimento dei dati, mentre '**`Read()`**' no. In pratica, può darsi che il compilatore non riesca a distinguere tra le due procedure, comportandosi sempre nello stesso modo.

Esempi

```
var totale : integer;
...
Write('Inserisci il totale: ');
Read(totale);
...
```

Emette l'invito a inserire un valore e quindi lo attende dallo standard input.

```
var capitale : integer;
var tasso    : real;
...
Write('Inserisci di seguito il capitale e il tasso: ');
Read(capitale,tasso);
...
```

Emette l'invito a inserire due valori consecutivi: un intero e un valore decimale.

292.7 Struttura del sorgente: le dichiarazioni

È già stato accennato alla struttura di un sorgente Pascal: del programma, delle procedure e delle funzioni. Si tratta di tre parti fondamentali:

1. intestazione del programma, dichiarazione della procedura o della funzione;
2. dichiarazioni;
3. istruzioni.

Il punto più delicato è la definizione della parte delle dichiarazioni, dato che nel Pascal originale esiste un ordine preciso nel tipo di istruzioni che possono esservi inserite. Si tratta di dichiarazioni:

1. '**`label`**'
2. '**`const`**'
3. '**`type`**'
4. '**`var`**'
5. '**`procedure`**'
6. '**`function`**'

La maggior parte di queste dichiarazioni non è ancora stata descritta. In particolare, '**`label`**', dal momento che serve a realizzare dei salti incondizionati senza ritorno (*go-to*), non viene descritta in questi capitoli sul Pascal.

292.8 Riferimenti

- Gordon Dodrill, *Pascal Language Tutorial*
  <*http://www8.silversand.net/techdoc/pascal/paslist.htm*>

Pascal: tipi di dati derivati

Nel capitolo introduttivo è stato visto l'uso di variabili identificabili semplicemente con il loro nome. La programmazione elementare richiede anche l'utilizzo di strutture di dati più complesse; le stesse stringhe sono degli array di caratteri e come tali vanno trattate.

293.1 Array

Gli array in Pascal sono una sequenza ordinata, in una quantità prestabilita, di elementi dello stesso tipo. Gli elementi possono essere composti da qualunque tipo di dati, nativo o derivato.

Una caratteristica importante del linguaggio Pascal sta nel fatto che nel momento della dichiarazione di un array, viene definito anche il valore iniziale dell'indice da utilizzare per la scansione dei vari elementi.

293.1.1 Dichiarazione e accesso

```
var nome : array[inizio..fine] of tipo
```

La sintassi indicata, dove le parentesi quadre fanno parte dell'istruzione, mostra in breve in che modo si possa dichiarare un array, a una sola dimensione, di elementi di un certo tipo di dati.

È importante osservare che vengono stabiliti in modo esplicito sia l'indice iniziale del primo elemento che quello finale dell'ultimo, stabilendo implicitamente la quantità di questi.

L'esempio seguente mostra la dichiarazione di tre array simili, composti tutti da sette interi, dove, rispettivamente, il primo elemento si raggiunge con l'indice iniziale 1, 0 e 2.

```
var elenco  : array[1..7] of integer;
    elenco2 : array[0..6] of integer;
    elenco3 : array[2..8] of integer;
```

Per accedere agli elementi di un array si usa la sintassi seguente e anche qui le parentesi quadre fanno parte dell'istruzione.

```
nome [indice]
```

Quello che conta è che l'indice indicato sia valido, in funzione della dichiarazione fatta in origine. L'esempio seguente assegna al primo elemento il valore 10.

```
elenco[1] := 10;
```

Gli array multidimensionali non sono altro che array di array. Il modo più semplice per dichiarare un array multidimensionale è quello di indicare due o più intervalli di valori per gli indici, secondo la sintassi seguente:

```
var nome : array[inizio..fine,inizio..fine...] of tipo
```

Per esempio, l'istruzione seguente dichiara un array a due dimensioni di tre elementi per otto, di tipo intero. Si osservi in particolare il secondo intervallo di indici, dove il primo elemento verrà raggiunto con l'indice zero.

```
var elenco : array[1..3,0..7] of integer;
```

In modo analogo, si raggiunge un elemento di un array multidimensionale utilizzando due o più indici, secondo la sintassi seguente:

```
nome [indice,indice...]
```

L'esempio seguente assegna un valore all'elemento «1,0».

```
elenco[1,0] := 10;
```

293.1.2 Scansione di un array

La scansione di un array avviene generalmente con un ciclo enumerativo, '**for**', come nell'esempio seguente:

```
...
var indice : integer;
var elenco : array[1..7] of integer;
...
begin
    ...
    for indice := 1 to 7 do begin
        ...
        elenco[indice] := ...
        ...
    end;
    ...
end.
```

La scansione di array multidimensionali avviene generalmente attraverso una serie di cicli enumerativi, uno per ogni dimensione, annidati opportunamente. L'esempio seguente mostra la scansione di un array a tre dimensioni.

```
...
var i,j,k : integer;
var elenco : array[1..7,0..8,2..10] of integer;
...
begin
    ...
    for i := 1 to 7 do begin
        ...
        for j := 0 to 8 do begin
            ...
            for k := 2 to 10 do begin
                ...
                elenco[i,j,k] := ...
                ...
            end;
            ...
        end;
        ...
    end;
    ...
end.
```

293.2 Stringhe

Nel linguaggio Pascal, così come in molti altri, le stringhe sono semplicemente degli array di caratteri, con qualche piccola differenza per facilitarne l'utilizzo.

La dichiarazione di una variabile stringa è quindi la dichiarazione di un array composto da una quantità predefinita di caratteri. Nell'esempio seguente, viene creato una variabile stringa di 20 caratteri.

```
var cognome : array[1..20] of char;
```

La variabile dichiarata in questo modo può essere usata come un array, cioè accedendo alle informazioni carattere per carattere, oppure nel suo insieme. Nell'esempio seguente si assegna un nome alla variabile stringa mostrata sopra.

```
cognome := 'Rossi';
```

Se si utilizza un assegnamento di questo tipo, vengono ricoperti anche gli elementi successivi alla lunghezza della stringa letterale assegnata. Quindi, seguendo l'esempio, l'array riceverà il nome «Rossi» nei suoi primi cinque elementi, mentre negli altri verrà comunque inserito uno spazio.

293.3 Tipi

Il linguaggio Pascal permette di definire dei tipi di dati derivati, a partire da quelli elementari, o a partire da altri tipi composti dichiarati precedentemente.

```
type tipo_nuovo = definizione_del_tipo
```

La definizione di un nuovo tipo va posta nella zona dichiarativa del programma, della procedura o della funzione. L'esempio seguente serve a dichiarare il tipo «Numero» come equivalente al tipo intero standard.

```
type Numero = integer;
```

Naturalmente, la definizione di un nuovo tipo è sensata quando serve a individuare qualcosa di più complesso dei dati elementari, come nel caso di un array. L'esempio seguente dichiara il tipo «Stringa» come un array di 80 caratteri, quindi dichiara il tipo «Nominativo» come array composto da due elementi «Stringa» (probabilmente uno per il nome e l'altro per il cognome).

```
type Stringa = array[1..80] of char;
type Nominativo = array[1..2] of Stringa;
```

A questo punto, per seguire l'esempio, se si generasse una variabile di tipo «Nominativo», si otterrebbe un array di due elementi, che in realtà sono array di 80 caratteri.

```
...
var Nome : Nominativo;
...
begin
    ...
    Nome[1] := 'Pinco';
    Nome[2] := 'Pallino';
    ...
end.
```

L'esempio mostra in che modo si potrebbe usare una variabile del genere. Tuttavia, si poteva accedere anche al singolo elemento carattere, utilizzando due indici.

```
    ...
    Nome[1,1] := 'P';
    Nome[1,2] := 'i';
    Nome[1,3] := 'n';
    Nome[1,4] := 'c';
    Nome[1,5] := 'o';
    ...
end.
```

Convenzionalmente, quando si dichiara un nuovo tipo di dati, si usa l'iniziale maiuscola, per distinguerlo facilmente dagli altri tipi nativi.

293.4 Costanti

Il linguaggio Pascal offre qualcosa di simile alle costanti macro di altri linguaggi come il C. Non si tratta di un linguaggio di precompilazione, ma proprio del Pascal, anche se si tratta comunque di costanti letterali, senza la definizione di un tipo a priori.

```
const nome_della_costante = valore_letterale
```

La dichiarazione di queste costanti va fatta, come prevedibile, nella zona dichiarativa del programma, della procedura o della funzione. L'esempio seguente dichiara la costante «DIMENSIONE», che poi viene usata per definire la dimensione di una serie di array.

```
...
const DIMENSIONE = 11;
...
var elenco  : array[1..DIMENSIONE] of integer;
    elenco2 : array[1..DIMENSIONE] of integer;
    elenco3 : array[1..DIMENSIONE] of integer;
```

Il vantaggio di utilizzare le costanti sta nel facilitare la lettura del sorgente, nel riconoscere il significato di determinate costanti, e nel facilitare la modifica di tali valori, senza dover rileggere tutto il sorgente alla loro ricerca.

293.5 Tipo enumerativo, sottointervallo e insieme

Il linguaggio Pascal offre dei tipi di dati particolari, che non sono ancora stati descritti, il cui scopo è solo quello di facilitare il compito del programmatore.

293.5.1 Tipo enumerativo

Il tipo enumerativo, o scalare, secondo la terminologia del Pascal, è una forma di rappresentazione di un intero attraverso costanti mnemoniche. In pratica, si definisce una variabile che può assumere un elenco di valori simbolici possibili, valori che in realtà sono solo delle costanti e non hanno alcun valore verbale.

```
( costante , costante [ , ... ] )
```

La sintassi indicata mostra il modo in cui si definisce un tipo del genere: all'interno di parentesi tonde si elencano i nomi delle costanti che possono essere assegnate a una variabile di questo tipo.

L'esempio seguente mostra la dichiarazione di una variabile scalare che può assumere i valori «VERDE», «BLU» e «ROSSO».

```
var colore : (VERDE, BLU, ROSSO);
```

L'esempio stesso dovrebbe chiarire l'utilità di questo tipo di dati: si lascia al compilatore il compito di stabilire i valori più appropriati per i simboli che possono essere associati a una variabile. Tuttavia, è importante chiarire che non è possibile visualizzare il contenuto di una variabile del genere, in quanto questo non è prevedibile.

```
if colore = VERDE then
    begin
        ...
        Writeln( "Il colore è verde" );
    end;
else
    ...
;
```

Naturalmente, questo tipo di dati si presta particolarmente per la definizione di tipi derivati, come nell'esempio seguente, dove prima si dichiara un tipo e più avanti si utilizza nella dichiarazione di una nuova variabile.

```
type Sapore = (INSIPIDO, DOLCE, SALATO, ACIDO, PICCANTE, AMARO);
...
var pietanza : Sapore;
...
```

293.5.2 Sottointervallo

Il sottointervallo è la definizione di un tipo derivato che può utilizzare solo un intervallo stabilito di valori. Questo intervallo si definisce solo con l'indicazione di due costanti dello stesso tipo, separate da due punti in sequenza.

Per esempio, per indicare la serie di numeri interi che va da uno a sette, si può utilizzare la notazione '**`1..7`**', mentre per indicare la serie delle lettere alfabetiche minuscole, si può utilizzare la notazione '**`'a'..'z'`**'.

Naturalmente, si possono indicare anche degli intervalli di un tipo enumerativo dichiarato in precedenza. Seguono alcuni esempi.

```
type Settimana = (LUNEDÌ, MARTEDÌ, MERCOLEDÌ,
                  GIOVEDÌ, VENERDÌ, SABATO, DOMENICA);

type Feriale   = LUNEDÌ..VENERDÌ;
...
var lavoro    : Feriale;
    minuscola : 'a'..'z';
...
```

Le variabili dichiarate in questo modo, ottengono dal compilatore il tipo più adatto a contenere l'informazione indicata, senza la necessità di doverlo indicare in modo esplicito.

293.5.3 Insieme

Una variabile può contenere un'informazione riferita a un insieme di elementi enumerativi. In pratica, si tratta di un tipo simile a quello enumerativo, dove ogni elemento può essere presente o meno. Si dichiara questo tipo di dati con le parole chiave '**`set of`**'. Si osservi l'esempio seguente:

```
type Settimana = (LUNEDÌ, MARTEDÌ, MERCOLEDÌ,
                  GIOVEDÌ, VENERDÌ, SABATO, DOMENICA);
...
type Lavoro    = set of Settimana;
...
var tutti     : Lavoro;
    presenze  : Lavoro;
    assenze   : Lavoro;
    altri     : Lavoro;
...
```

Le variabili '**`tutti`**', '**`presenze`**' e '**`assenze`**', definite del tipo '**`Lavoro`**', il quale a sua volta è definito come insieme di tutti i simboli del tipo '**`Settimana`**', possono contenere un sottoinsieme di tali simboli.

```
...
begin
    ...
    presenze := (LUNEDÌ, MERCOLEDÌ, VENERDÌ,
                 DOMENICA);
```

```
    ...
    tutti := (LUNEDÌ..DOMENICA);
    ...
    assenze := tutti - presenze;
    ...
    altri := assenze;
    ...
    tutti := assenze + presenze;
    ...
end.
```

L'esempio mostra alcuni modi in cui possono essere utilizzate le variabili contenenti insiemi e quali espressioni si possono realizzare. In pratica:

- due variabili dello stesso tipo di insieme possono essere assegnate l'una nell'altra;
- due variabili dello stesso tipo di insieme possono essere sommate, generando un insieme risultato dell'unione dei due;
- tra due variabili dello stesso tipo di insieme può essere indicata una sottrazione, con la quale si genera un insieme risultato dall'eliminazione degli elementi presenti nella seconda variabile.

A parte gli assegnamenti che possono essere fatti alle variabili contenenti un insieme, è poi necessario poter verificare il contenuto di tali variabili, con istruzioni apposite. Per questo si usa la parola chiave '**in**'. L'esempio seguente dovrebbe essere autoesplicativo.

```
    if LUNEDÌ in presenze then begin
        ...
        ...
    end;
    if MARTEDÌ in presenze then begin
        ...
        ...
    end;
```

Un insieme può essere definito anche come gruppo di valori di un intervallo, come nell'esempio seguente in cui si definisce un tipo nuovo che rappresenta l'insieme delle lettere minuscole.

```
type Lettere    = set of 'a'..'z';
```

Nello stesso modo, si può utilizzare la parola chiave '**in**' per verificare che un valore appartenga a un insieme definito in forma di intervallo.

```
if iniziale in 'a'..'z' then begin
    ...
end;
```

293.6 Record

Il record è un tipo di dati composto dall'insieme di altri tipi, ognuno con una sua denominazione. L'esempio seguente mostra in che modo possano essere creati tipi nuovi definiti come record.

```
type Datario =
    record
        anno    : integer;
        mese    : integer;
        giorno  : integer;
    end;

type Anagrafico =
    record
```

```
        cognome : array[1..40] of char;
        nome    : array[1..40] of char;
        luogo   : array[1..40] of char;
        data    : Datario;
    end;
```

L'esempio vuole mostrare la creazione di un record anagrafico con tutti i dati (riferiti alla nascita) che permettono di identificare una persona. Si può osservare che la data (di nascita) è stata definita come tipo '**`Datario`**', che a sua volta è un altro record.

Quando si dichiara una variabile come tipo record, si pone il problema di accedere ai vari elementi di questo. Per farlo si usa l'operatore punto ('**`.`**'). Si osservi l'esempio seguente, in cui si dichiara un array di dati anagrafici e quindi si assegnano i valori per il primo elemento di questo array.

```
...
var anagrafe : array[1..10] of Anagrafico;
...
begin
    ...
    anagrafe[1].cognome         := 'Pallino';
    anagrafe[1].nome            := 'Pinco';
    anagrafe[1].luogo           := 'Sferopoli';
    anagrafe[1].data.anno       := 1990;
    anagrafe[1].data.mese       := 1;
    anagrafe[1].data.giorno     := 31;
    ...
end;
```

Come si può osservare, per inserire le informazioni sulla data di nascita, è stato necessario usare due volte il punto per accedere agli elementi del sottorecord '**`data`**'.

Una variabile definita come record può ricevere l'assegnamento in blocco di un'altra variabile record, purché dello stesso tipo.

293.6.1 With

Quando si utilizzano frequentemente i record, potrebbe essere conveniente specificare che in una porzione di codice sorgente si vuole fare riferimento a elementi di una variabile determinata. Si osservi l'esempio seguente, che è una variante di quanto già visto in precedenza.

```
...
var anagrafe : array[1..10] of Anagrafico;
...
begin
    ...
    with anagrafe[1] do begin
        cognome         := 'Pallino';
        nome            := 'Pinco';
        luogo           := 'Sferopoli';
        data.anno       := 1990;
        data.mese       := 1;
        data.giorno     := 31;
    end;
    ...
end;
```

Il significato dovrebbe essere evidente: nell'intervallo delimitato dalle parole chiave '**`begin`**' '**`end`**', tutti i nomi si riferiscono a elementi di '**`anagrafe[1]`**'.

293.7 Riferimenti

- Gordon Dodrill, *Pascal Language Tutorial*
  <*http://www8.silversand.net/techdoc/pascal/paslist.htm*>

Pascal: esempi di programmazione

Questo capitolo raccoglie solo alcuni esempi di programmazione, in parte già descritti in altri capitoli. Lo scopo di questi esempi è solo didattico, utilizzando forme non ottimizzate per la velocità di esecuzione.

294.1 Problemi elementari di programmazione

In questa sezione vengono mostrati alcuni algoritmi elementari portati in Pascal. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo 282.

294.1.1 Somma tra due numeri positivi

Il problema della somma tra due numeri positivi, attraverso l'incremento unitario, è stato descritto nella sezione 282.2.1.

```
(* ============================================================ *)
(* Somma.pas                                                    *)
(* Somma esclusivamente valori positivi.                        *)
(* ============================================================ *)
program Sommare;

var     x       : integer;
        y       : integer;
        z       : integer;
```

```
(* ================================================================ *)
(* somma( <x>, <y> )                                                *)
(* ---------------------------------------------------------------- *)
function somma( x : integer; y : integer ) : integer;

var     z       : integer;
        i       : integer;

begin

    z := x;

    for i := 1 to y do begin
        z := z+1;
    end;

    somma := z;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci il primo numero intero positivo: ' );
    Readln( x );
    Write( 'Inserisci il secondo numero intero positivo: ' );
    Readln( y );

    z := somma( x, y );

    Write( x, ' + ', y, ' = ', z );

end.

(* ================================================================ *)
```

In alternativa si può tradurre il ciclo '**for**' in un ciclo '**while**'.

```
function somma( x : integer; y : integer ) : integer;

var     z       : integer;
        i       : integer;

begin

    z := x;
    i := 1;

    while i <= y do begin
        z := z+1;
        i := i+1;
    end;

    somma := z;

end;
```

294.1.2 Moltiplicazione di due numeri positivi attraverso la somma

Il problema della moltiplicazione tra due numeri positivi, attraverso la somma, è stato descritto nella sezione 282.2.2.

```
(* ================================================================ *)
(* Moltiplica.pas                                                   *)
(* Moltiplica esclusivamente valori positivi.                       *)
(* ================================================================ *)
program Moltiplicare;

var     x       : integer;
        y       : integer;
        z       : integer;

(* ================================================================ *)
(* moltiplica( <x>, <y> )                                           *)
(* ---------------------------------------------------------------- *)
function moltiplica( x : integer; y : integer ) : integer;

var     z       : integer;
        i       : integer;

begin

    z := 0;

    for i := 1 to y do begin
        z := z+x;
    end;

    moltiplica := z;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci il primo numero intero positivo: ' );
    Readln( x );
    Write( 'Inserisci il secondo numero intero positivo: ' );
    Readln( y );

    z := moltiplica( x, y );

    Write( x, ' * ', y, ' = ', z );

end.

(* ================================================================ *)
```

In alternativa si può tradurre il ciclo '**for**' in un ciclo '**while**'.

```
function moltiplica( x : integer; y : integer ) : integer;

var     z       : integer;
        i       : integer;

begin

    z := 0;
    i := 1;
```

```
    while i <= y do begin
        z := z+x;
        i := i+1;
    end;

    moltiplica := z;

end;
```

294.1.3 Divisione intera tra due numeri positivi

Il problema della divisione tra due numeri positivi, attraverso la sottrazione, è stato descritto nella sezione 282.2.3.

```
(* ================================================================== *)
(* Dividi.pas                                                         *)
(* Divide esclusivamente valori positivi.                             *)
(* ================================================================== *)
program Dividere;

var     x       : integer;
        y       : integer;
        z       : integer;

(* ================================================================== *)
(* dividi( <x>, <y> )                                                 *)
(* ------------------------------------------------------------------ *)
function dividi( x : integer; y : integer ) : integer;

var     z       : integer;
        i       : integer;

begin

    z := 0;
    i := x;

    while i >= y do begin
        i := i - y;
        z := z+1;
    end;

    dividi := z;

end;

(* ================================================================== *)
(* Inizio del programma.                                              *)
(* ------------------------------------------------------------------ *)
begin

    Writeln;
    Write( 'Inserisci il primo numero intero positivo: ' );
    Readln( x );
    Write( 'Inserisci il secondo numero intero positivo: ' );
    Readln( y );

    z := dividi( x, y );

    Write( x, ' / ', y, ' = ', z );

end.
(* ================================================================== *)
```

294.1.4 Elevamento a potenza

Il problema dell'elevamento a potenza tra due numeri positivi, attraverso la moltiplicazione, è stato descritto nella sezione 282.2.4.

```
(* ================================================================ *)
(* Exp.pas                                                          *)
(* Eleva a potenza.                                                 *)
(* ================================================================ *)
program Potenza;

var     x       : integer;
        y       : integer;
        z       : integer;

(* ================================================================ *)
(* exp( <x>, <y> )                                                  *)
(* ---------------------------------------------------------------- *)
function exp( x : integer; y : integer ) : integer;

var     z       : integer;
        i       : integer;

begin

    z := 1;

    for i := 1 to y do begin
        z := z * x;
    end;

    exp := z;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci il primo numero intero positivo: ' );
    Readln( x );
    Write( 'Inserisci il secondo numero intero positivo: ' );
    Readln( y );

    z := exp( x, y );

    Write( x, ' ** ', y, ' = ', z );

end.
(* ================================================================ *)
```

In alternativa si può tradurre il ciclo '**`for`**' in un ciclo '**`while`**'.

```
(* ================================================================ *)
(* exp( <x>, <y> )                                                  *)
(* ---------------------------------------------------------------- *)
function exp( x : integer; y : integer ) : integer;

var     z       : integer;
        i       : integer;

begin

    z := 1;
```

```
    i := 1;

    while i <= y do begin
        z := z * x;
        i := i+1;
    end;

    exp := z;

end;
```

È possibile usare anche un algoritmo ricorsivo.

```
function exp( x : integer; y : integer ) : integer;

begin

    if x = 0 then
        begin
            exp := 0;
        end
    else if y = 0 then
        begin
            exp := 1;
        end
    else
        begin
            exp := ( x * exp(x, y-1) );
        end
    ;

end;
```

294.1.5 Radice quadrata

Il problema della radice quadrata è stato descritto nella sezione 282.2.5.

```
(* ================================================================ *)
(* Radice.pas                                                       *)
(* Radice quadrata.                                                 *)
(* ================================================================ *)
program RadiceQuadrata;

var     x       : integer;
        z       : integer;

(* ================================================================ *)
(* radice( <x> )                                                    *)
(* ---------------------------------------------------------------- *)
function radice( x : integer; ) : integer;

var     z       : integer;
        t       : integer;
        ciclo   : boolean;

begin

    z := 0;
    t := 0;
    ciclo := TRUE;

    while ciclo do begin

        t := z * z;

        if t > x then
```

```
            begin
                z := z-1;
                radice := z;
                ciclo := FALSE;
            end
        ;

        z := z+1;

    end;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci il numero intero positivo: ' );
    Readln( x );

    z := radice( x );

    Writeln( 'La radice di ', x, ' e'' ', z );

end.
(* ================================================================ *)
```

294.1.6 Fattoriale

Il problema del fattoriale è stato descritto nella sezione 282.2.6.

```
(* ================================================================ *)
(* Fact.pas                                                         *)
(* Fattoriale.                                                      *)
(* ================================================================ *)
program Fattoriale;

var     x       : integer;
        z       : integer;

(* ================================================================ *)
(* fact( <x> )                                                      *)
(* ---------------------------------------------------------------- *)
function fact( x : integer ) : integer;

var     i       : integer;

begin

    i := x - 1;

    while i > 0 do begin

        x := x * i;
        i := i-1;

    end;

    fact := x;

end;

(* ================================================================ *)
```

```
(* Inizio del programma.                                              *)
(* ------------------------------------------------------------------ *)
begin

    Writeln;
    Write( 'Inserisci il numero intero positivo: ' );
    Readln( x );

    z := fact( x );

    Writeln( 'Il fattoriale di ', x, ' e'' ', z );

end.

(* ================================================================== *)
```

In alternativa, l'algoritmo si può tradurre in modo ricorsivo.

```
function fact( x : integer ) : integer;

begin

    if x > 1 then
        begin
            fact := ( x * fact( x - 1 ) )
        end
    else
        begin
            fact := 1
        end
    ;

end;
```

294.1.7 Massimo comune divisore

Il problema del massimo comune divisore, tra due numeri positivi, è stato descritto nella sezione 282.2.7.

```
(* ================================================================== *)
(* MCD.pas                                                            *)
(* Massimo Comune Divisore.                                           *)
(* ================================================================== *)
program MassimoComuneDivisore;

var     x       : integer;
        y       : integer;
        z       : integer;

(* ================================================================== *)
(* mcd( <x>, <y> )                                                    *)
(* ------------------------------------------------------------------ *)
function mcd( x : integer; y : integer ) : integer;

begin

    while x <> y do begin

        if x > y then
            begin
                x := x - y;
            end
        else
            begin
                y := y - x;
            end
```

```
        ;

    end;

    mcd := x;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci il primo numero intero positivo: ' );
    Readln( x );
    Write( 'Inserisci il secondo numero intero positivo: ' );
    Readln( y );

    z := mcd( x, y );

    Write( 'Il massimo comune divisore tra ', x, ' e ', y, ' e'' ', z );

end.

(* ================================================================ *)
```

294.1.8 Numero primo

Il problema della determinazione se un numero sia primo o meno, è stato descritto nella sezione 282.2.8.

```
(* ================================================================ *)
(* Primo.pas                                                        *)
(* ================================================================ *)
program NumeroPrimo;

var     x       : integer;

(* ================================================================ *)
(* primo( <x> )                                                     *)
(* ---------------------------------------------------------------- *)
function primo( x : integer ) : boolean;

var     np      : boolean;
        i       : integer;
        j       : integer;

begin

    np := TRUE;
    i := 2;

    while (i < x) AND np do begin

        j := x / i;
        j := x - (j * i);

        if j = 0  then
            begin
                np := FALSE;
            end
        else
            begin
                i := i+1;
```

```
            end
        ;

    end;

    primo := np;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci un numero intero positivo: ' );
    Readln( x );

    if primo( x ) then
        begin
            Writeln( 'E'' un numero primo' );
        end
    else
        begin
            Writeln( 'Non e'' un numero primo' );
        end
    ;

end.
(* ================================================================ *)
```

294.2 Scansione di array

In questa sezione vengono mostrati alcuni algoritmi, legati alla scansione degli array, portati in Pascal. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto descritto nel capitolo 282.

Per semplicità, gli esempi mostrati fanno uso di array dichiarati globalmente, che come tali sono accessibili alle procedure e alle funzioni senza necessità di farne riferimento all'interno delle chiamate.

294.2.1 Ricerca sequenziale

Il problema della ricerca sequenziale all'interno di un array, è stato descritto nella sezione 282.3.1.

```
(* ================================================================ *)
(* RicercaSeq.pas                                                   *)
(* Ricerca sequenziale.                                             *)
(* ================================================================ *)
program RicercaSequenziale;

const   DIM     = 100;

var     lista   : array[1..DIM] of integer;
        x       : integer;
        i       : integer;
        z       : integer;

(* ================================================================ *)
(* ricercaseq( <x>, <ele-inf>, <ele-sup> )                          *)
(* ---------------------------------------------------------------- *)
```

```
function ricercaseq( x : integer; a : integer; z : integer ) : integer;

var     i       : integer;

begin

    (* ------------------------------------------------------------ *)
    (* Se l'elemento non viene trovato, il valore -1 segnala        *)
    (* l'errore.                                                    *)
    (* ------------------------------------------------------------ *)
    ricercaseq := -1;

    (* ------------------------------------------------------------ *)
    (* Scandisce l'array alla ricerca dell'elemento.                *)
    (* ------------------------------------------------------------ *)
    for i := a to z do begin

        if x = lista[i] then
            begin
                ricercaseq := i;
            end
        ;

    end;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln( 'Inserire il numero di elementi.' );
    Writeln( DIM, ' al massimo.' );
    Readln( z );

    if z > DIM then
        begin
            z := DIM;
        end
    ;

    Writeln( 'Inserire i valori dell''array' );

    for i := 1 to z do begin
        Write( 'elemento ', i:2, ': ' );
        Readln( lista[i] );
    end;

    Writeln( 'Inserire il valore da cercare' );
    Readln( x );

    i := ricercaseq( x, 1, z );

    Writeln( 'Il valore cercato si trova nell''elemento', i );

end.
(* ================================================================ *)
```

Esiste anche una soluzione ricorsiva che viene mostrata nella subroutine seguente:

```
function ricercaseq( x : integer; a : integer; z : integer ) : integer;

begin

    if a > z then
        begin
```

```
            (* ------------------------------------------------- *)
            (* La corrispondenza non è stata trovata.            *)
            (* ------------------------------------------------- *)
            ricercaseq := -1;
        end
    else if x = lista[a] then
        begin
            ricercaseq := a;
        end
    else
        begin
            ricercaseq := ricercaseq( x, a+1, z);
        end
    ;

end;
```

294.2.2 Ricerca binaria

Il problema della ricerca binaria all'interno di un array, è stato descritto nella sezione 282.3.2.

```
(* ================================================================ *)
(* RicercaBin.pas                                                   *)
(* Ricerca binaria.                                                 *)
(* ================================================================ *)
program RicercaBinaria;

const   DIM     = 100;

var     lista   : array[1..DIM] of integer;
        x       : integer;
        i       : integer;
        z       : integer;

(* ================================================================ *)
(* ricercabin( <x>, <ele-inf>, <ele-sup> )                          *)
(* ---------------------------------------------------------------- *)
function ricercabin( x : integer; a : integer; z : integer ) : integer;

var     m       : integer;

begin

    (* ------------------------------------------------------------ *)
    (* Determina l'elemento centrale.                               *)
    (* ------------------------------------------------------------ *)
    m := ( a + z ) / 2;

    if m < a then
        begin

            (* ------------------------------------------------- *)
            (* Non restano elementi da controllare.              *)
            (* ------------------------------------------------- *)
            ricercabin := -1;
        end
    else if x < lista[m] then
        begin

            (* ------------------------------------------------- *)
            (* Si ripete la ricerca nella parte inferiore.       *)
            (* ------------------------------------------------- *)
            ricercabin := ricercabin( x, a, m-1 );
        end
    else if x > lista[m] then
```

```
        begin

            (* ------------------------------------------------- *)
            (* Si ripete la ricerca nella parte superiore.       *)
            (* ------------------------------------------------- *)
            ricercabin := ricercabin( x, m+1, z );
        end
    else
        begin

            (* ------------------------------------------------- *)
            (* m rappresenta l'indice dell'elemento cercato.     *)
            (* ------------------------------------------------- *)
            ricercabin := m;
        end
    ;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln( 'Inserire il numero di elementi.' );
    Writeln( DIM, ' al massimo.' );
    Readln( z );

    if z > DIM then
        begin
            z := DIM;
        end
    ;

    Writeln( 'Inserire i valori dell''array' );

    for i := 1 to z do begin
        Write( 'elemento ', i:2, ': ' );
        Readln( lista[i] );
    end;

    Writeln( 'Inserire il valore da cercare' );
    Readln( x );

    i := ricercabin( x, 1, z );

    Writeln( 'Il valore cercato si trova nell''elemento', i );

end.
(* ================================================================ *)
```

294.3 Algoritmi tradizionali

In questa sezione vengono mostrati alcuni algoritmi tradizionali portati in Pascal. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo 282.

294.3.1 Bubblesort

Il problema del Bubblesort è stato descritto nella sezione 282.4.1. Viene mostrata prima una soluzione iterativa e successivamente la funzione '**bsort**' in versione ricorsiva.

```
(* ================================================================ *)
(* BSort.pas                                                        *)
(* ================================================================ *)
program BubbleSort;

const   DIM     = 100;

var     lista   : array[1..DIM] of integer;
        i       : integer;
        z       : integer;

(* ================================================================ *)
(* bsort( <ele-inf>, <ele-sup> )                                    *)
(* ---------------------------------------------------------------- *)
procedure bsort( a : integer; z : integer );

var     scambio : integer;
        j       : integer;
        k       : integer;

begin

    (* ------------------------------------------------------------ *)
    (* Inizia il ciclo di scansione dell'array.                     *)
    (* ------------------------------------------------------------ *)
    for j := a to ( z-1 ) do begin

        (* -------------------------------------------------------- *)
        (* Scansione interna dell'array per collocare nella         *)
        (* posizione j l'elemento giusto.                           *)
        (* -------------------------------------------------------- *)
        for k := ( j+1 ) to z do begin

            if lista[k] < lista[j] then
                begin

                    (* ------------------------------------------- *)
                    (* Scambia i valori.                           *)
                    (* ------------------------------------------- *)
                    scambio := lista[k];
                    lista[k] := lista[j];
                    lista[j] := scambio;
                end
            ;

        end;
    end;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln( 'Inserire il numero di elementi.' );
    Writeln( DIM, ' al massimo.' );
    Readln( z );

    if z > DIM then
        begin
```

```
            z := DIM;
        end
    ;

    Writeln( 'Inserire i valori dell''array' );

    for i := 1 to z do begin
        Write( 'elemento ', i:2, ': ' );
        Readln( lista[i] );
    end;

    bsort( 1, z );

    Writeln( 'Array ordinato:' );

    for i := 1 to z do begin
        Write( lista[i] );
    end;

end.
(* ================================================================= *)
```

Segue la procedura '**bsort**' in versione ricorsiva.

```
procedure bsort( a : integer; z : integer );

var     scambio : integer;
        k       : integer;

begin

    if a < z then
        begin

            (* ------------------------------------------------- *)
            (* Scansione interna dell'array per collocare nella  *)
            (* posizione j l'elemento giusto.                    *)
            (* ------------------------------------------------- *)
            for k := ( a+1 ) to z do begin

                if lista[k] < lista[a] then
                    begin

                        (* ------------------------------------- *)
                        (* Scambia i valori.                     *)
                        (* ------------------------------------- *)
                        scambio := lista[k];
                        lista[k] := lista[a];
                        lista[a] := scambio;
                    end
                ;

            end;

            bsort( a+1, z );

        end
    ;

end;
```

294.3.2 Torre di Hanoi

Il problema della torre di Hanoi è stato descritto nella sezione 282.4.2.

```
(* ================================================================ *)
(* Hanoi.pas                                                        *)
(* Torre di Hanoi.                                                  *)
(* ================================================================ *)
program TorreHanoi;

var     n       : integer;
        p1      : integer;
        p2      : integer;

(* ================================================================ *)
(* hanoi( <n>, <p1>, <p2> )                                         *)
(* ---------------------------------------------------------------- *)
procedure hanoi( n : integer; p1 : integer; p2 : integer );

begin

    if n > 0 then
        begin
            hanoi( n-1, p1, 6-p1-p2 );

            Writeln(
                'Muovi l''anello ', n:1,
                ' dal piolo ', p1:1,
                ' al piolo ', p2:1
            );

            hanoi( n-1, 6-p1-p2, p2 );
        end
    ;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci il numero di anelli: ' );
    Readln( n );
    Write( 'Inserisci il piolo iniziale: ' );
    Readln( p1 );
    Write( 'Inserisci il piolo finale: ' );
    Readln( p2 );

    hanoi( n, p1, p2 );

end.
(* ================================================================ *)
```

294.3.3 Quicksort

L'algoritmo del Quicksort è stato descritto nella sezione 282.4.3.

```
(* ================================================================ *)
(* QSort.pas                                                        *)
(* ================================================================ *)
program QuickSort;

const   DIM     = 100;
```

```
var     lista   : array[1..DIM] of integer;
        i       : integer;
        z       : integer;

(* ================================================================ *)
(* part( <ele-inf>, <ele-sup> )                                     *)
(* ---------------------------------------------------------------- *)
function part( a : integer; z : integer ) : integer;

var     scambio : integer;
        i       : integer;
        cf      : integer;
        loop1   : boolean;
        loop2   : boolean;
        loop3   : boolean;

begin

    (* ------------------------------------------------------------ *)
    (* Si assume che a sia inferiore a z.                           *)
    (* ------------------------------------------------------------ *)
    i := a+1;
    cf := z;

    (* ------------------------------------------------------------ *)
    (* Inizia il ciclo di scansione dell'array.                     *)
    (* ------------------------------------------------------------ *)
    loop1 := TRUE;
    while loop1 do begin

        loop2 := TRUE;
        while loop2 do begin

            (* ---------------------------------------------------- *)
            (* Sposta i a destra.                                   *)
            (* ---------------------------------------------------- *)
            if ( lista[i] > lista[a] ) OR ( i >= cf ) then
                begin
                    loop2 := FALSE;
                end
            else
                begin
                    i := i+1;
                end
            ;

        end;

        loop3 := TRUE;
        while loop3 do begin

            (* ---------------------------------------------------- *)
            (* Sposta cf a sinistra.                                *)
            (* ---------------------------------------------------- *)
            if lista[cf] <= lista[a] then
                begin
                    loop3 := FALSE;
                end
            else
                begin
                    cf := cf-1;
                end
            ;

        end;
```

```
        if cf <= i then
            begin

                (* ------------------------------------------------ *)
                (* è avvenuto l'incontro tra i e cf.                *)
                (* ------------------------------------------------ *)
                loop1 := FALSE;
            end
        else
            begin

                (* ------------------------------------------------ *)
                (* Vengono scambiati i valori.                      *)
                (* ------------------------------------------------ *)
                scambio := lista[cf];
                lista[cf] := lista[i];
                lista[i] := scambio;

                i := i+1;
                cf := cf-1;
            end
        ;
    end;

    (* ---------------------------------------------------------------- *)
    (* A questo punto, lista[a..z] è stata ripartita e cf è la          *)
    (* collocazione finale.                                             *)
    (* ---------------------------------------------------------------- *)
    scambio := lista[cf];
    lista[cf] := lista[a];
    lista[a] := scambio;

    (* ---------------------------------------------------------------- *)
    (* In questo momento, lista[cf] è un elemento (un valore) nella     *)
    (* posizione giusta.                                                *)
    (* ---------------------------------------------------------------- *)
    part := cf

end;

(* ================================================================== *)
(* quicksort( <ele-inf>, <ele-sup> )                                  *)
(* ------------------------------------------------------------------ *)
procedure quicksort( a : integer; z : integer );

var     cf      : integer;

begin

    if z > a then
        begin
            cf := part( a, z );
            quicksort( a, cf-1 );
            quicksort( cf+1, z );
        end
    ;

end;

(* ================================================================== *)
(* Inizio del programma.                                              *)
(* ------------------------------------------------------------------ *)
begin

    Writeln( 'Inserire il numero di elementi.' );
    Writeln( DIM, ' al massimo.' );
```

```
    Readln( z );

    if z > DIM then
        begin
            z := DIM;
        end
    ;

    Writeln( 'Inserire i valori dell''array' );

    for i := 1 to z do begin
        Write( 'elemento ', i:2, ': ' );
        Readln( lista[i] );
    end;

    quicksort( 1, z );

    Writeln( 'Array ordinato:' );

    for i := 1 to z do begin
        Write( lista[i] );
    end;

end.
(* ================================================================= *)
```

294.3.4 Permutazioni

L'algoritmo ricorsivo delle permutazioni è stato descritto nella sezione 282.4.4.

```
(* ================================================================= *)
(* Permuta.pas                                                       *)
(* ================================================================= *)
program Permutazioni;

const   DIM     = 100;

var     lista   : array[1..DIM] of integer;
        i       : integer;
        z       : integer;

(* ================================================================= *)
(* permuta( <ele-inf>, <ele-sup>, <elementi-totali> )                *)
(* ----------------------------------------------------------------- *)
function permuta( a : integer; z : integer; elementi : integer ) : integer;

var     scambio : integer;
        k       : integer;
        i       : integer;

begin

    (* ------------------------------------------------------------- *)
    (* Se il segmento di array contiene almeno due elementi,         *)
    (* si procede.                                                   *)
    (* ------------------------------------------------------------- *)
    if ( z-a ) >= 1 then
        begin

            (* --------------------------------------------------- *)
            (* Inizia il ciclo di scambi tra l'ultimo elemento e     *)
            (* uno degli altri contenuti nel segmento di array.      *)
            (* --------------------------------------------------- *)
            k := z;
            while k >= a do begin
```

```
                (* ------------------------------------------------ *)
                (* Scambia i valori.                                *)
                (* ------------------------------------------------ *)
                scambio := lista[k];
                lista[k] := lista[z];
                lista[z] := scambio;

                (* ------------------------------------------------ *)
                (* Esegue una chiamata ricorsiva per permutare un   *)
                (* segmento più piccolo dell'array.
                (* ------------------------------------------------ *)
                permuta( a, z-1, elementi );

                (* ------------------------------------------------ *)
                (* Scambia i valori.                                *)
                (* ------------------------------------------------ *)
                scambio := lista[k];
                lista[k] := lista[z];
                lista[z] := scambio;

                k := k-1;

            end;
        end
    else
        begin

            (* ------------------------------------------------------ *)
            (* Visualizza la situazione attuale dell'array.           *)
            (* ------------------------------------------------------ *)
            for i := 1 to elementi do begin
                Write( lista[i]:4 );
            end;
            Writeln;

        end
    ;

end;

(* ===================================================================== *)
(* Inizio del programma.                                                 *)
(* --------------------------------------------------------------------- *)
begin

    Writeln( 'Inserire il numero di elementi.' );
    Writeln( DIM, ' al massimo.' );
    Readln( z );

    if z > DIM then
        begin
            z := DIM;
        end
    ;

    Writeln( 'Inserire i valori dell''array' );

    for i := 1 to z do begin
        Write( 'elemento ', i:2, ': ' );
        Readln( lista[i] );
    end;

    permuta( 1, z, z );

end.
```

```
(* ============================================================== *)
```

Parte lvii

Perl

Perl: introduzione

Perl è un linguaggio di programmazione *interpretato* (o quasi) che quindi viene eseguito da un interprete senza bisogno di generare un eseguibile binario. In questo senso, i programmi Perl sono degli script eseguiti dal programma '**`perl`**' che per convenzione dovrebbe essere collocato in '`/usr/bin/`'.

Perl è molto importante in tutti gli ambienti Unix e per questo è molto utile conoscerne almeno i rudimenti. Volendo fare una scala di importanza, subito dopo la programmazione con le shell Bourne e derivate, viene la programmazione in Perl.

I capitoli, che a partire da questo, sono dedicati a Perl, introducono solamente il linguaggio, che per essere studiato seriamente richiederebbe invece molto tempo e la lettura di molta documentazione.

295.1 Struttura fondamentale

Dal momento che i programmi Perl vengono realizzati in forma di script, per convenzione occorre indicare il nome del programma interprete nella prima riga.

```
#!/usr/bin/perl
```

Per l'esecuzione di script da parte di un interprete non si può fare affidamento sul percorso di ricerca degli eseguibili (la variabile di ambiente '**`PATH`**'), è quindi importante che il binario '**`perl`**' si trovi dove previsto. Questa posizione ('`/usr/bin/perl`') è quella standard ed è opportuno che sia rispettata tale consuetudine, altrimenti i programmi in Perl di altri autori non potrebbero funzionare nel proprio sistema senza una variazione di tutti i sorgenti.

> Il buon amministratore di sistema farebbe bene a collocare dei collegamenti simbolici in tutte le posizioni in cui sarebbe possibile che venisse cercato l'eseguibile '**`perl`**': '`/bin/perl`', '`/usr/bin/perl`' e '`/usr/local/bin/perl`'.

Come si può intuire, il simbolo '`#`' rappresenta l'inizio di un commento.

```
#!/usr/bin/perl
#
# Esempio di intestazione e di commenti in Perl.
...
```

Un'altra convenzione che riguarda gli script Perl è l'estensione: '`.pl`', anche se l'utilizzo o meno di questa non costituisce un problema.

295.1.1 Istruzioni

Le istruzioni seguono la convenzione del linguaggio C, per cui terminano con un punto e virgola ('`;`') e i raggruppamenti di queste, detti anche blocchi, si fanno utilizzando le parentesi graffe ('**`{ }`**').

istruzione `;`
`{`*istruzione* `;` *istruzione* `;` *istruzione* `;}`

Generalmente, un'istruzione può essere interrotta e ripresa nella riga successiva, dal momento che la sua conclusione è dichiarata chiaramente dal punto e virgola finale.

295.1.2 Nomi

I nomi che servono per identificare ciò che si utilizza all'interno del programma seguono regole determinate. In particolare:

- un nome può iniziare con un trattino basso o una lettera e può continuare con lettere, numeri e trattini bassi;
- i nomi sono sensibili alla differenza tra lettere maiuscole e minuscole.

Spesso i nomi sono preceduti da un simbolo che ne definisce il contesto:

- $ il dollaro precede i nomi delle variabili scalari e degli elementi scalari di un array;
- @ il simbolo *at* precede i nomi degli array normali o di raggruppamenti di elementi in essi contenuti;
- % il simbolo di percentuale precede i nomi degli array associativi, detti anche hash;
- & il simbolo e-commerciale precede i nomi delle funzioni quando queste vengono chiamate.

295.1.3 Contesto operativo

Perl è un linguaggio di programmazione con cui gli elementi che si indicano hanno un valore riferito al contesto in cui ci si trova. Questo significa, per esempio, che un array può essere visto come: una lista di elementi, il numero degli elementi contenuti, o una stringa contenente tutti i valori degli elementi contenuti.

In pratica, ciò serve a garantire che i dati siano trasformati nel modo più adatto al contesto, al quale è importante fare attenzione.

295.1.4 Tipi di dati

I tipi di dati più importanti che si possono gestire con Perl sono:

- stringhe;
- valori numerici;
- riferimenti;
- liste.

Le variabili di Perl vengono create semplicemente con l'assegnamento di un valore, senza la necessità di dichiarare il tipo o la dimensione. Le conversioni dei valori numerici sono fatte automaticamente in base al contesto.

In Perl non esiste un tipo di dati logico (nel senso di *Vero* o *Falso*); solo il risultato di una condizione lo è, ma non equivale a un valore gestibile in una variabile. Da un punto di vista logico-booleano, i valori seguenti vengono considerati equivalenti a *Falso*:

- indefinito -- equivalente a una variabile non dichiarata;

- "" -- la stringa nulla;
- 0 -- il valore numerico zero;
- "0" -- la stringa corrispondente al numero zero.

Qualunque altro valore viene trattato come equivalente a *Vero*.

295.1.5 Esecuzione dei programmi Perl

Per poter eseguire un programma Perl, così come accade per qualunque altro tipo di script, occorre attivare il permesso di esecuzione per il file che lo contiene.

```
chmod +x programma_perl
```

Sembra banale o evidente, ma spesso ci si dimentica di farlo e quello che si ottiene è il classico *permesso negato*: *Permission denied*.

295.2 Variabili e costanti scalari

La gestione delle variabili e delle costanti in Perl è molto simile a quella delle shell comuni. Una variabile scalare è quella che contiene un valore unico, contrapponendosi generalmente all'array che in Perl viene definito come variabile contenente una lista di valori.

295.2.1 Variabili

Le variabili scalari di Perl possono essere dichiarate in qualunque punto del programma e la loro dichiarazione coincide con l'inizializzazione, cioè l'assegnamento di un valore. I nomi delle variabili scalari iniziano sempre con il simbolo dollaro ('**$**').[1]

```
$variabile_scalare = valore
```

L'assegnamento di un valore a una variabile scalare implica l'utilizzo di quanto si trova alla destra del simbolo di assegnamento ('**=**') come valore scalare: una stringa, un numero o un riferimento. È il contesto a decidere il risultato dell'assegnamento.

295.2.2 Variabili predefinite

Perl fornisce automaticamente alcune variabili scalari che normalmente non devono essere modificate dai programmi. Tali variabili servono per comunicare al programma alcune informazioni legate al sistema, oppure l'esito dell'esecuzione di una funzione, esattamente come accade con i parametri delle shell comuni. La tabella 295.1 mostra un elenco di alcune di queste variabili standard. Si può osservare che i nomi di tali variabili non seguono la regola per cui il primo carattere deve essere un trattino basso o una lettera. Questa eccezione consente di evitare di utilizzare inavvertitamente nomi corrispondenti a variabili predefinite.

[1] L'utilizzo del dollaro come prefisso dei nomi delle variabili assomiglia a quanto si fa con le shell derivate da quella di Bourne, con la differenza che con Perl il dollaro si lascia sempre, mentre con queste shell si utilizza solo quando si deve leggere il loro contenuto.

Tabella 295.1. Elenco di alcune variabili standard di Perl.

Nome	Descrizione
$$	Numero PID del programma.
$<	Numero UID reale dell'utente che esegue il programma.
$>	Numero UID efficace dell'utente che esegue il programma.
$?	Lo stato dell'ultima chiamata di sistema.
$_	Argomento predefinito di molte funzioni.
$0	Il nome del programma.
$"	Separatore di lista.
$/	Separatore di righe per l'input (*input record separator*).

295.2.3 Costanti

Le costanti scalari più importanti sono di tipo stringa o numeriche. Le prime richiedono la delimitazione con apici doppi o singoli, mentre quelle numeriche non richiedono alcuna delimitazione.

Perl gestisce le stringhe racchiuse tra apici doppi in maniera simile a quanto fanno le shell tradizionali:

- le variabili indicate al loro interno vengono espanse, o meglio, *interpolate* (secondo la terminologia di Perl);
- la barra obliqua inversa ('\') può essere utilizzata come prefisso di escape quando si vogliono includere nella stringa simboli che altrimenti sarebbero interpretati in modo diverso e quando si vogliono indicare codici per cui non esiste un simbolo della tastiera.[2]

Anche le stringhe racchiuse tra apici singoli sono gestite in modo simile alle shell tradizionali:

- al loro interno non vengono effettuate interpolazioni di variabili;
- il carattere di escape, rappresentato dalla barra obliqua inversa, può essere utilizzato solo per inserire un apice letterale e la barra obliqua inversa stessa ('\'' e '\\').

Inoltre, davanti all'apice di inizio di una tale stringa, è necessario sia presente uno spazio.

La tabella 295.2 mostra un elenco di alcune di queste sequenze di escape utilizzabili nelle stringhe.

Tabella 295.2. Elenco di alcune sequenze di escape utilizzabili nelle stringhe delimitate con gli apici doppi.

Escape	Corrispondenza
\\	\
\"	"
\$	$
\@	@
\'	'
\t	<*HT*>
\n	<*LF*>
\r	<*CR*>

[2] Se una stringa viene interrotta e ripresa nella riga successiva, quello che si ottiene, nel punto dell'interruzione, è l'inserimento di un codice di interruzione di riga. In pratica, lo stesso codice di interruzione di riga utilizzato per andare a capo, viene inserito nella stringa e trattato esattamente per quello che è.

Escape	Corrispondenza
\f	*<FF>*
\b	*<BS>*
\a	*<BELL>*
\e	*<ESC>*
\0*n*	Numero ottale rappresentato da *n*.
\x*h*	Numero esadecimale rappresentato da *h*.

Quando all'interno di stringhe tra apici doppi si indicano delle variabili (scalari e non), potrebbe porsi un problema di ambiguità causato dalla necessità di distinguere il nome delle variabili dal resto della stringa. Quando dopo il nome della variabile segue un carattere o un simbolo che non può fare parte del nome (come uno spazio o un simbolo di punteggiatura), Perl non ha difficoltà a individuare la fine del nome della variabile e la continuazione della stringa. Quando ciò non è sufficiente, si può delimitare il nome della variabile tra parentesi graffe, così come si fa con le shell tradizionali.

`${`*variabile*`}`
`@{`*variabile*`}`

295.2.3.1 Costanti numeriche

Le costanti numeriche possono essere indicate nel modo consueto, quando si usa la numerazione a base decimale, oppure anche in esadecimale e in ottale.

Con la numerazione a base 10, si possono indicare interi nel modo normale e valori decimali utilizzando il punto come separazione tra la parte intera e la parte decimale. Si può utilizzare anche la notazione esponenziale.

- numero intero: 123456
- numero intero leggibile più facilmente: 1_234_567
- numero reale: 123456.789
- notazione esponenziale: 2.3E-10

Un numero viene trattato come esadecimale quando è preceduto dal prefisso '**`0x`**' e come ottale quando inizia con uno zero.

- numero esadecimale: '**`0xFFFF`**'
- numero ottale: '**`0377`**'

Quando un numero ottale o esadecimale è contenuto in una stringa, l'eventuale conversione in numero non avviene automaticamente, come invece accade in presenza di notazioni in base 10.

295.2.4 Esempi

L'esempio seguente è il più banale, emette semplicemente la stringa '**`"Ciao Mondo!\n"`**' attraverso lo standard output. È da osservare la parte finale, '**`\n`**', che completa la stringa con un codice di interruzione di riga in modo da portare a capo il cursore in una nuova riga dello schermo.

```
#!/usr/bin/perl

print "Ciao Mondo!\n";
```

Se il file si chiama '**1.pl**', lo si deve rendere eseguibile e quindi si può provare il suo funzionamento.

```
$ chmod +x 1.pl[ Invio ]

$ 1.pl[ Invio ]

Ciao Mondo!
```

L'esempio seguente genera lo stesso risultato di quello precedente, ma con l'uso di variabili. Si può osservare che solo alla fine viene emesso il codice di interruzione di riga.

```
#!/usr/bin/perl

$primo = "Ciao";
$secondo = "Mondo";
print $primo;
print " ";
print $secondo;
print "!\n";
```

L'esempio seguente genera lo stesso risultato di quello precedente, ma con l'uso dell'interpolazione delle variabili all'interno di stringhe racchiuse tra apici doppi.

```
#!/usr/bin/perl

$primo = "Ciao";
$secondo = "Mondo";
print "$primo $secondo!\n";
```

L'esempio seguente emette la parola '**CiaoMondo**' senza spazi intermedi utilizzando la tecnica delle parentesi graffe.

```
#!/usr/bin/perl

$primo = "Ciao";
print "${primo}Mondo!\n";
```

L'esempio seguente mostra il comportamento degli apici singoli per delimitare le stringhe. Non si ottiene più l'interpolazione delle variabili.

```
#!/usr/bin/perl

$primo = "Ciao";
$secondo = "Mondo";
print '$primo $secondo!\n';
```

Se il file si chiama '**5.pl**', si può verificare il suo funzionamento nel modo seguente:

```
$ chmod +x 5.pl[ Invio ]

$ 5.pl[ Invio ]

$primo $secondo!\n
```

Inoltre, mancando il codice di interruzione di riga finale, l'invito della shell riappare subito alla destra di quanto visualizzato.

L'esempio seguente mostra l'uso di una costante e di una variabile numerica. Il valore numerico viene convertito automaticamente in stringa al momento dell'interpolazione.

```
#!/usr/bin/perl

$volte = 1000;
$primo = "Ciao";
$secondo = "Mondo";
print "$volte volte $primo $secondo!\n";
```

Se il file si chiama '**6.pl**', si può verificare il suo funzionamento nel modo seguente:

```
$ chmod +x 6.pl[ Invio ]

$ 6.pl[ Invio ]

1000 volte Ciao Mondo!
```

L'esempio seguente permette di prendere confidenza con le variabili predefinite descritte in precedenza.

```
#!/usr/bin/perl

print "Nome del programma: $0\n";
print "PID del programma: $$\n";
print "UID dell'utente: $<\n";
print "Ultima chiamata di sistema: $?\n";
```

Se il file si chiama '**7.pl**', si può verificare il suo funzionamento nel modo seguente:

```
$ chmod +x 7.pl[ Invio ]

$ 7.pl[ Invio ]
```

Il risultato potrebbe essere simile a quello seguente:

```
Nome del programma: ./7.pl
PID del programma: 717
UID dell'utente: 500
Ultima chiamata di sistema: 0
```

295.3 Array e liste

Perl gestisce gli array in modo dinamico, nel senso che possono essere allungati e accorciati a piacimento. Quando si parla di array si pensa generalmente a una variabile che abbia questa forma; ma Perl permette di gestire delle costanti array, definite liste.

Generalmente, il primo elemento di un array o di una lista ha indice zero. Questo assunto può essere cambiato agendo su una particolare variabile predefinita, ma ciò è sconsigliabile.

295.3.1 Liste

Le liste sono una sequenza di elementi scalari, di qualunque tipo, separati da virgole, racchiusi tra parentesi tonde. L'ultimo elemento può essere seguito o meno da una virgola, prima della parentesi chiusa.

(*elemento* , …)

La lista vuota, o nulla, si rappresenta con le semplici parentesi aperta e chiusa:

```
()
```

Seguono alcuni esempi in cui si mostrano diversi modi di indicare la stessa lista.

```
("uno", "due", "tre", "quattro", "ciao")
```

```
("uno", "due", "tre", "quattro", "ciao",)
```

```
("uno",
 "due",
 "tre",
 "quattro",
 "ciao",)
```

```
(
    "uno",
    "due",
    "tre",
    "quattro",
    "ciao",
)
```

Una lista può essere utilizzata per inizializzare un array, ma se si pretende di assegnare una lista a un variabile scalare, si ottiene in pratica che la variabile scalare contenga solo il valore dell'ultimo elemento della lista (alla variabile vengono assegnati, in sequenza, tutti gli elementi della lista, per cui, quello che resta è l'ultimo). Per esempio:

```
$miavar = ("uno", "due", "tre", "quattro", "ciao");
```

assegna a '**`$miavar`**' solo la stringa '**`"ciao"`**'.

Una lista di valori può essere utilizzata con un indice, per fare riferimento solo a uno di tali valori. Naturalmente ciò è utile quando l'indice è rappresentato da una variabile. L'esempio seguente mostra la trasformazione di un indice ('**`$ind`**'), che abbia un valore numerico compreso tra zero e nove, in un termine verbale.

```
$numverb = (
        "zero",
        "uno",
        "due",
        "tre",
        "quattro",
        "cinque",
        "sei",
        "sette",
        "otto",
        "nove",
)[$ind];
```

Gli elementi contenuti in una lista che non sono scalari, vengono interpolati, incorporando in quel punto tutti gli elementi che questi rappresentano. Gli eventuali elementi non scalari nulli, non rappresentano alcun elemento e vengono semplicemente ignorati. Per esempio, la lista

```
("uno", "due", (), ("tre", "quattro", "cinque"), "sei")
```

è perfettamente identica a quella seguente:

```
("uno", "due", "tre", "quattro", "cinque", "sei")
```

Naturalmente ciò ha maggiore significato quando non si tratta semplicemente di liste annidate, ma di array collocati all'interno di liste.

295.3.2 Array

L'array è una variabile contenente una lista di valori di qualunque tipo, purché scalari. Il nome di un array inizia con il simbolo '**@**' quando si fa riferimento a tutto l'insieme dei suoi elementi o anche solo a parte di questi. Quando ci si riferisce a un solo elemento di questo si utilizza il dollaro.[3]

Un array può essere dichiarato vuoto, con la sintassi seguente,

```
@array = ()
```

oppure assegnandogli una lista di elementi.

```
@array = (elemento, ...)
```

Il riferimento a un solo elemento di un array viene indicato con la notazione seguente (le parentesi quadre fanno parte della notazione),

```
$array[indice]
```

mentre il riferimento a un raggruppamento può essere indicato in vari modi.

```
@array[indice1, indice2, ...]
```

In tal caso ci si riferisce a un sottoinsieme composto dagli elementi indicati dagli indici contenuti all'interno delle parentesi quadre.

```
@array[indice_iniziale..indice_finale]
```

In questo modo ci si riferisce a un sottoinsieme composto dagli elementi contenuti nell'intervallo espresso dagli indici iniziale e finale.

Nella gestione degli array sono importanti due variabili predefinite:

- '**$[**' -- rappresenta l'indice del primo elemento di un array e si usa azzerata convenzionalmente, in modo che per identificare il primo elemento serva l'indice zero;[4]
- '**$#***array*' -- rappresenta l'ultimo indice dell'array identificato dal nome posto dopo il simbolo '**$#**'.

Assegnare un array o parte di esso a una variabile scalare, significa in pratica assegnare un numero intero esprimente il numero di elementi in esso contenuti. Per esempio,

```
@mioarray = ("uno", "due");
$mioscalare = @mioarray;
```

significa assegnare a '**$mioscalare**' il valore due.

Inserire un array o parte di esso in una stringa delimitata con gli apici doppi, implica l'interpolazione degli elementi, separati con quanto contenuto nella variabile '**$"**' (il separatore di lista). La variabile predefinita '**$"**' contiene normalmente uno spazio singolo. Per esempio,

```
@mioarray = ("uno", "due");
$mioscalare = "@mioarray";
```

significa assegnare a '**$mioscalare**' la stringa '**"uno due"**'.

Perl fornisce degli array predefiniti, di cui il più importante è '**@ARGV**' che contiene l'elenco degli argomenti ricevuti dalla riga di comando.

[3] In pratica, quando si fa riferimento a un solo elemento di un array si può immaginare che si tratti di un gruppo di elementi composto da un solo elemento, per cui si può utilizzare il prefisso '**@**' anche in questo caso.

[4] Meglio non modificare questa variabile.

295.3.3 Esempi

L’esempio seguente permette di verificare quanto espresso sugli array di Perl.

```
#!/usr/bin/perl

# Dichiara l’array assegnandogli sia stringhe che numeri
@elenco = ("primo", "secondo", 3, 4, "quinto");

# Attraverso l’assegnamento seguente, $elementi riceve il numero di
# elementi contenuti nell’array.
$elementi = @elenco;

# Emette tutte le informazioni legate all’array.
print "L’array contiene $elementi elementi.\n";
print "L’indice iniziale è $[.\n";
print "L’ultimo elemento si raggiunge con l’indice $#elenco.\n";

# Emette in ordine tutti gli elementi dell’array.
print "L’array contiene: $elenco[0] $elenco[1] $elenco[2] $elenco[3] $elenco[4].\n";

# Idem
print "Anche in questo modo si legge il contenuto dell’array: @elenco.\n";
```

Se il file si chiama ‘**`11.pl`**’, si può verificare il suo funzionamento nel modo seguente:

```
$ chmod +x 11.pl[ Invio ]

$ 11.pl[ Invio ]

L’array contiene 5 elementi.
L’indice iniziale è 0.
L’ultimo elemento si raggiunge con l’indice 4.
L’array contiene: primo secondo 3 4 quinto.
Anche in questo modo si legge il contenuto dell’array: primo secondo 3 4 quinto.
```

L’esempio seguente mostra il funzionamento dell’array predefinito ‘**`@ARGV`**’.

```
#!/usr/bin/perl

print "Il programma $0 è stato avviato con gli argomenti seguenti:\n";
print "@ARGV\n";
print "Il primo argomento era $ARGV[0]\n";
print "e l’ultimo era $ARGV[$#ARGV].\n";
```

Se il file si chiama ‘**`12.pl`**’, si può verificare il suo funzionamento nel modo seguente:

```
$ chmod +x 12.pl[ Invio ]

$ 12.pl carbonio idrogeno ossigeno[ Invio ]

Il programma ./12.pl è stato avviato con gli argomenti seguenti:
carbonio idrogeno ossigeno
Il primo argomento era carbonio
e l’ultimo era ossigeno.
```

295.4 Array associativi o hash

L'array associativo, o hash, è un tipo speciale di array che normalmente non si trova negli altri linguaggi di programmazione. Gli elementi sono inseriti a coppie, dove il primo elemento della coppia è la chiave di accesso per il secondo.

Il nome di un hash inizia con il segno di percentuale ('**%**'), mentre il riferimento a un elemento scalare di questo si fa utilizzando il dollaro, mentre l'indicazione di un sottoinsieme avviene con il simbolo '**@**', come per gli array.

La dichiarazione, ovvero l'assegnamento di un array associativo, si esegue in uno dei due modi seguenti.

```
%array_associativo = (chiave, elemento, ...)
```

```
%array_associativo = (chiave => elemento, ...)
```

La seconda notazione esprime meglio la dipendenza tra la chiave e l'elemento che con essa viene raggiunto. L'elemento che funge da chiave viene trattato sempre come stringa, mentre gli elementi abbinati alle chiavi possono essere di qualunque tipo scalare. In particolare, nel caso si utilizzi l'abbinamento tra chiave e valore attraverso il simbolo '**=>**', ciò che sta alla sinistra di questo viene interpretato come stringa in ogni caso, permettendo di eliminare la normale delimitazione attraverso apici.

Un elemento singolo di un hash viene indicato con la notazione seguente, dove le parentesi graffe fanno parte dell'istruzione.

```
$array_associativo{chiave}
```

La chiave può essere una costante stringa o un'espressione che restituisce una stringa. La costante stringa può anche essere indicata senza apici.

Un sottoinsieme di un hash è un'entità equivalente a un array e viene indicato con la notazione seguente:

```
@array_associativo{chiave1,chiave2,...}
```

Perl fornisce alcuni array associativi predefiniti. Il più importante è '**%ENV**' che contiene le variabili di ambiente, cui si accede indicando il nome della variabile come chiave.

295.4.1 Esempi

L'esempio seguente mostra un semplice array associativo e il modo di accedere ai suoi elementi in base alla chiave.

```
#!/usr/bin/perl

# Dichiarazione dell'array: attenzione a non fare confusione!
#             -------------    ----------------    ---------
%deposito = ("primo", "alfa", "secondo", "bravo", "terzo", 3);


# Emette il contenuto dei vari elementi.
print "$deposito{primo}\n";
print "$deposito{secondo}\n";
print "$deposito{terzo}\n";
```

Se il file si chiama '**21.pl**', si può verificare il suo funzionamento nel modo seguente:

```
$ chmod +x 21.pl[ Invio ]

$ 21.pl[ Invio ]
```

```
alfa
bravo
3
```

L'esempio seguente è identico al precedente, ma l'hash viene dichiarato in modo più facile da interpretare visivamente.

```
#!/usr/bin/perl

# Dichiarazione dell'array.
%deposito = (
        "primo", "alfa",
        "secondo", "bravo",
        "terzo", 3,
);

# Emette il contenuto dei vari elementi.
print "$deposito{primo}\n";
print "$deposito{secondo}\n";
print "$deposito{terzo}\n";
```

L'esempio seguente è identico al precedente, ma l'hash viene dichiarato in modo ancora più leggibile.

```
#!/usr/bin/perl

# Dichiarazione dell'array.
%deposito = (
        primo   => "alfa",
        secondo => "bravo",
        terzo   => 3,
);

# Emette il contenuto dei vari elementi.
print "$deposito{primo}\n";
print "$deposito{secondo}\n";
print "$deposito{terzo}\n";
```

L'esempio seguente mostra l'uso dell'array '**%ENV**' per la lettura delle variabili di ambiente.

```
#!/usr/bin/perl

print "PATH: $ENV{PATH}\n";
print "TERM: $ENV{TERM}\n";
```

Se il file si chiama '**24.pl**', si può verificare il suo funzionamento nel modo seguente:

```
$ chmod +x 24.pl[ Invio ]

$ 24.pl[ Invio ]

PATH: /usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
TERM: linux
```

295.5 Operatori ed espressioni

Il sistema di operatori e delle relative espressioni che possono essere create con Perl è piuttosto complesso. La parte più consistente di questa gestione riguarda il trattamento delle stringhe, che qui verrà descritto particolarmente in un altro capitolo. Alcuni tipi di espressioni e i relativi operatori non vengono mostrati, data la loro complessità per chi non conosca già il linguaggio C. In particolare viene saltata la gestione dei dati a livello di singoli bit.

Il senso e il risultato di un'espressione dipende dal contesto. La valutazione di un'espressione dipende dalle precedenze che esistono tra i vari tipi di operatori. Si parla di precedenza superiore quando qualcosa viene valutato prima di qualcos'altro, mentre la precedenza è inferiore quando qualcosa viene valutato dopo qualcos'altro.

295.5.1 Operatori che intervengono su valori numerici, stringhe e liste

Gli operatori che intervengono su valori numerici sono elencati nella tabella 295.3.

Tabella 295.3. Elenco degli operatori utilizzabili in presenza di valori numerici. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
++op	Incrementa di un'unità l'operando prima che venga restituito il suo valore.
op++	Incrementa di un'unità l'operando dopo averne restituito il suo valore.
--op	Decrementa di un'unità l'operando prima che venga restituito il suo valore.
op--	Decrementa di un'unità l'operando dopo averne restituito il suo valore.
+op	Non ha alcun effetto.
-op	Inverte il segno dell'operando.
op1 + op2	Somma i due operandi.
op1 - op2	Sottrae dal primo il secondo operando.
*op1 * op2*	Moltiplica i due operandi.
op1 / op2	Divide il primo operando per il secondo.
op1 % op2	Modulo: il resto della divisione tra il primo e il secondo operando.
*op1 ** op2*	Eleva il primo operando alla potenza del secondo.
var = valore	Assegna alla variabile il valore alla destra.
op1 += op2	*op1 = op1 + op2*
op1 -= op2	*op1 = op1 - op2*
*op1 *= op2*	*op1 = op1 * op2*
op1 /= op2	*op1 = op1 / op2*
op1 %= op2	*op1 = op1 % op2*
*op1 **= op2*	*op1 = op1 ** op2*
op1 == op2	*Vero* se gli operandi si equivalgono.
op1 != op2	*Vero* se gli operandi sono differenti.
op1 < op2	*Vero* se il primo operando è minore del secondo.
op1 > op2	*Vero* se il primo operando è maggiore del secondo.
op1 <= op2	*Vero* se il primo operando è minore o uguale al secondo.
op1 >= op2	*Vero* se il primo operando è maggiore o uguale al secondo.

La gestione da parte di Perl delle stringhe è molto sofisticata e questa si attua principalmente attraverso gli operatori di delimitazione. In questa sezione si vuole solo accennare agli operatori che hanno effetto sulle stringhe, sorvolando su raffinatezze che si possono ottenere in casi particolari. La tabella 295.4 elenca tali operatori.

Tabella 295.4. Elenco degli operatori utilizzabili in presenza di valori alfanumerici, o stringa. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
str1 . *str2*	Concatena le due stringhe.
str x *num*	Restituisce la stringa ripetuta consecutivamente *num* volte.
str =~ *modello*	Collega il modello alla stringa. Il risultato dipende dal contesto.
str !~ *modello*	Come '**=~**', ma restituisce un valore inverso.
var = *valore*	Assegna alla variabile il valore alla destra.
op1 x= *op2*	*op1* = *op1* x *op2*
op1 .= *op2*	*op1* = *op1* . *op2*
str1 eq *str2*	*Vero* se le due stringhe sono uguali.
str1 ne *str2*	*Vero* se le due stringhe sono differenti.
str1 lt *str2*	*Vero* se la prima stringa è lessicograficamente inferiore alla seconda.
str1 gt *str2*	*Vero* se la prima stringa è lessicograficamente superiore alla seconda.
str1 le *str2*	*Vero* se la prima stringa è lessicograficamente inferiore o uguale alla seconda.
str1 ge *str2*	*Vero* se la prima stringa è lessicograficamente superiore o uguale alla seconda.

Gli operatori che intervengono sulle liste sono elencati nella tabella 295.5.

Tabella 295.5. Elenco degli operatori utilizzabili in presenza di liste. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
lista x *num*	Restituisce la lista composta ripetendo quella indicata per *num* volte.
array = *lista*	Crea l'array assegnandogli la lista indicata alla destra.
elem1 , *elem2*	La virgola è l'operatore di separazione degli elementi di una lista.
elem1 => *elem2*	Sinonimo della virgola.
elem1 .. *elem2*	Rappresenta una lista di valori da *elem1* a *elem2*.

295.5.2 Operatori logici

È il caso di ricordare che con Perl tutti i tipi di dati possono essere valutati in modo logico: lo zero numerico o letterale, la stringa nulla e un valore indefinito corrispondono a *Falso*, in tutti gli altri casi si considera equivalente a *Vero*. Gli operatori logici sono elencati nella tabella 295.6.

Tabella 295.6. Elenco degli operatori logici. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
! *op*	Inverte il risultato logico dell'operando.
op1 && *op2*	Se il risultato del primo operando è *Falso* non valuta il secondo.
op1 || *op2*	Se il risultato del primo operando è *Vero* non valuta il secondo.
op1 and *op2*	Come '**&&**', ma con un livello di precedenza molto basso.
op1 or *op2*	Come '**||**', ma con un livello di precedenza molto basso.

Il risultato di un'espressione logica complessa è quello dell'ultima espressione elementare a essere valutata. Questo particolare è importante, anche se si tratta di un comportamento comune di diversi linguaggi, perché viene usato spesso per condizionare l'esecuzione di istruzioni, senza usare le strutture tradizionali, come *if-else*, o simili.[5]

[5]Questo tipo di approccio da parte del programmatore è sconsigliabile in generale, dato che serve a complicare la lettura e l'interpretazione umana del sorgente; tuttavia è importante conoscere esempi di questo tipo, perché sono sempre molti i programmi fatti alla svelta senza pensare alla leggibilità.

L'esempio seguente dovrebbe dare l'idea di come si può utilizzare l'operatore logico '**||**' (OR). Il risultato logico finale non viene preso in considerazione, quello che conta è solo il risultato della condizione '**$valore > 90**', che se non si avvera fa sì che venga eseguita l'istruzione '**print**' posta come secondo operando.

```
#!/usr/bin/perl
...
$valore = 100;
...
$valore > 90 || print "Il valore è insufficiente\n";
...
```

In pratica, se il valore contenuto nella variabile '**$valore**' supera 90, si ottiene l'emissione del messaggio attraverso lo standard output. In questi casi, si usano preferibilmente gli operatori '**and**' e '**or**', che si distinguono perché hanno una precedenza molto bassa, adattandosi meglio alla circostanza.

```
$valore > 90 or print "Il valore è insufficiente\n";
```

Come si vede dalla variante dell'esempio proposta, l'espressione diventa quasi simpatica, perché sembra una frase inglese più comprensibile. La cosa può diventare ancora più «divertente» se si utilizza la funzione interna '**die()**', che serve a visualizzare un messaggio attraverso lo standard error e a concludere il funzionamento del programma Perl.

```
$valore > 90 or die "Il valore è insufficiente\n";
```

> A parte la simpatia o il divertimento nello scrivere codice del genere, è bene ricordare che poi si tratta di qualcosa che un altro programmatore può trovare difficile da interpretare.

295.5.3 Operatori particolari

Tra gli operatori che non sono stati indicati nelle categorie descritte precedentemente, il più interessante è il seguente:

condizione ? *espressione1* : *espressione2*

Se la condizione restituisce il valore *Vero*, allora l'operatore restituisce il valore della prima espressione, altrimenti quello della seconda.

295.5.4 Raggruppamenti di espressioni

Le espressioni, di qualunque genere siano, possono essere raggruppate in modo che la loro valutazione avvenga in un ordine differente da quanto previsto dalle precedenze legate agli operatori utilizzati. Per questo si usano le parentesi tonde, come avviene di solito anche negli altri linguaggi.

Le parentesi tonde sono anche i delimitatori delle liste, per cui è anche possibile immaginare che esistano delle liste contenenti delle espressioni. Se si valuta una lista di espressioni, si ottiene il risultato della valutazione dell'ultima di queste.

295.6 Strutture di controllo del flusso

Perl gestisce praticamente tutte le strutture di controllo di flusso degli altri linguaggi di programmazione, compreso *go-to* che comunque è sempre meglio non utilizzare e qui non viene presentato volutamente.

Quando una struttura particolare controlla un gruppo di istruzioni, queste vengono necessariamente delimitate attraverso le parentesi graffe, come avviene in C, ma a differenza di quel linguaggio, non è possibile farne a meno quando ci si limita a indicare una sola istruzione.

Le strutture di controllo del flusso basano normalmente questo controllo sulla verifica di una condizione espressa all'interno di parentesi tonde.

Nei modelli sintattici indicati, le parentesi graffe fanno parte delle istruzioni, essendo i delimitatori dei blocchi di istruzioni di Perl.

295.6.1 if | unless

`if (`*condizione*`) {` *istruzione*`;…}`
`if (`*condizione*`) {` *istruzione*`;…} else {` *istruzione*`;…}`
`if (`*cond*`) {` *istr*`;…} elsif (`*cond*`) {` *istr*`;…}… else {` *istr*`;…}`

Se la condizione si verifica viene eseguito il gruppo di istruzioni seguente, racchiuso tra parentesi graffe, quindi il controllo passa alle istruzioni successive alla struttura. Se viene utilizzato '**`elsif`**', nel caso non si verifichino altre condizioni precedenti, viene verificata la condizione successiva; se questa si avvera, viene eseguito il gruppo di istruzioni che ne segue. Al termine il controllo riprende dopo la struttura. Se viene utilizzato '**`else`**', quando non si verifica alcuna condizione di quelle poste, viene eseguito il gruppo di istruzioni finale. Seguono alcuni esempi.

```
if ($importo > 10000000) { print "L'offerta è vantaggiosa"; }
```

```
if ($importo > 10000000)
  {
    $memorizza = $importo;
    print "L'offerta è vantaggiosa.\n";
  }
else
  {
    print "Lascia perdere.\n";
  }
```

```
if ($importo > 10000000)
  {
    $memorizza = $importo;
    print "L'offerta è vantaggiosa.\n";
  }
elsif ($importo > 5000000)
  {
    $memorizza = $importo;
    print "L'offerta è accettabile.\n";
  }
else
  {
    print "Lascia perdere.\n";
  }
```

‘**unless**’ può essere utilizzato come ‘**if**’, con la differenza che la condizione viene valutata in modo opposto, cioè viene eseguito il gruppo di istruzioni che segue ‘**unless**’ solo se **non** si verifica la condizione.

295.6.2 while | until

`while (`*condizione*`) { `*istruzione*`;…}`
`while (`*condizione*`) { `*istruzione*`;…} continue { `*istruzione*`;…;}`

La struttura ‘**while**’ esegue un gruppo di istruzioni finché la condizione restituisce il valore *Vero*. La condizione viene valutata prima di eseguire il gruppo di istruzioni e poi ogni volta che termina un ciclo, prima dell’esecuzione del successivo.

Il blocco di istruzioni che segue ‘**continue**’ viene eseguito semplicemente di seguito al gruppo normale. Ci sono situazioni in cui viene saltato. Segue l’esempio del calcolo del fattoriale.

```
#!/usr/bin/perl

# Il numero di partenza viene fornito come argomento nella riga di comando.
$numero = $ARGV[0];
$cont = $numero -1;
while ($cont > 0)
  {
    $numero = $numero * $cont;
    $cont = $cont -1;
  }
print "Il fattoriale è $numero.\n";
```

La stessa cosa si poteva semplificare nel modo seguente:

```
#!/usr/bin/perl

# Il numero di partenza viene fornito come argomento nella riga di comando.
$numero = $ARGV[0];
$cont = $numero -1;
while ($cont)
  {
    $numero *= $cont;
    $cont--;
  }
print "Il fattoriale è $numero.\n";
```

All’interno delle istruzioni di un ciclo ‘**while**’ possono apparire alcune istruzioni particolari:

- ‘**next**’

  interrompe l’esecuzione del gruppo di istruzioni e riprende dalla valutazione della condizione (se esiste il gruppo ‘**continue**’, ‘**next**’ rinvia all’esecuzione di questo e quindi alla valutazione della condizione);

- ‘**last**’

  esce definitivamente dal ciclo ‘**while**’ senza curarsi del gruppo di istruzioni ‘**continue**’;

- ‘**redo**’

  ripete il ciclo, senza valutare e verificare nuovamente l’espressione della condizione e senza curarsi del gruppo di istruzioni ‘**continue**’;

L’esempio seguente è una variante del calcolo del fattoriale in modo da vedere il funzionamento di ‘**last**’. ‘**while (1){…}**’ equivale a un ciclo senza fine perché la condizione (cioè il valore 1) è sempre vera.

```
#!/usr/bin/perl

# Il numero di partenza viene fornito come argomento nella riga di comando.
$numero = $ARGV[0];
$cont = $numero -1;
# Il ciclo seguente è senza fine.
while (1)
  {
    $numero *= $cont;
    $cont--;
    if (!$cont)
      {
        last;
      }
  }
print "Il fattoriale è $numero.\n";
```

'**until**' può essere utilizzato come '**while**', con la differenza che la condizione viene valutata in modo opposto, cioè viene eseguito il gruppo di istruzioni che segue '**until**' solo se **non** si verifica la condizione. In pratica, al verificarsi della condizione, il ciclo termina.

295.6.3 do ... while | do ... until

```
do { istruzione ;...} while (condizione )
```

'**do**...**while**' esegue un gruppo di istruzioni almeno una volta, quindi ne ripete l'esecuzione finché la condizione restituisce il valore *Vero*. Segue il solito esempio del calcolo del fattoriale.

```
#!/usr/bin/perl

# Il numero di partenza viene fornito come argomento nella riga di comando.
$cont       = $ARGV[0];
$fattoriale = 1;
do
  {
    $fattoriale *= $cont;
    $cont--;
  }
while ($cont);
print "Il fattoriale è $fattoriale.\n";
```

'**until**', al posto di '**while**', verifica che la condizione non si avveri, in pratica inverte il senso della condizione che controlla il ciclo.

295.6.4 for

```
for (espressione1 ; espressione2 ; espressione3 ) { istruzione ;...}
```

Questa è la forma tipica di un'istruzione '**for**', in cui la prima espressione corrisponde all'assegnamento iniziale di una variabile, la seconda a una condizione che deve verificarsi fino a che si vuole che sia eseguito il gruppo di istruzioni e la terza all'incremento o decremento della variabile inizializzata con la prima espressione. In pratica, potrebbe esprimersi nella sintassi seguente:

```
for ($var = n ; condizione ; $var++) { istruzione ;...}
```

In realtà la forma del ciclo '**for**' potrebbe essere diversa, ma in tal caso si preferisce utilizzare il nome '**foreach**' che è comunque un sinonimo.

In breve: la prima espressione viene eseguita una volta sola all'inizio del ciclo; la seconda viene valutata all'inizio di ogni ciclo e il gruppo di istruzioni viene eseguito solo se il risultato è *Vero*. L'ultima espressione viene eseguita alla fine dell'esecuzione del gruppo di istruzioni, prima che si ricominci con l'analisi della condizione.

Segue il solito esempio del calcolo del fattoriale.

```
#!/usr/bin/perl

# Il numero di partenza viene fornito come argomento nella riga di comando.
$numero = $ARGV[0];
for ($cont = 1; $cont < $ARGV[0]; $cont++)
  {
    $numero *= $cont;
  }
print "Il fattoriale è $numero.\n";
```

295.6.5 foreach

`foreach` *var_scalare lista* `{` *istruzione* `;…}`

'**`foreach`**' è un sinonimo di '**`for`**', per cui si tratta della stessa cosa, solo che si preferisce utilizzare due termini differenti per una struttura che può articolarsi in due modi diversi.

La variabile scalare iniziale, viene posta di volta in volta ai valori contenuti nella lista, eseguendo ogni volta il gruppo di istruzioni. Il ciclo finisce quando non ci sono più elementi nella lista.

Segue il solito esempio del calcolo del fattoriale.

```
#!/usr/bin/perl

# Il numero di partenza viene fornito come argomento nella riga di comando.
$numero = $ARGV[0];
foreach $cont (1 .. ($ARGV[0] -1))
  {
    $numero *= $cont;
  }
print "Il fattoriale è $numero.\n";
```

295.6.6 Istruzioni condizionate

Una brutta tradizione di Perl consente la scrittura di istruzioni condizionate secondo le sintassi seguenti:

espressione1 `if` *espressione2*
espressione1 `unless` *espressione2*
espressione1 `while` *espressione2*
espressione1 `until` *espressione2*

Si tratta di forme abbreviate e sconsigliabili (secondo il parere di chi scrive) delle sintassi seguenti.

`if (`*espressione2*`) {` *espressione1* `}`
`unless (`*espressione2*`) {` *espressione1* `}`
`while (`*espressione2*`) {` *espressione1* `}`
`until (`*espressione2*`) {` *espressione1* `}`

Come si vede, lo sforzo necessario a scrivere le istruzioni nel modo normale, è minimo. Evidentemente, l'idea che sta alla base della possibilità di usare sintassi così strane delle strutture '**`if`**', '**`while`**' e simili, è quella di permettere la scrittura di codice che assomigli alla lingua inglese.

295.7 Funzioni interne

Perl fornisce una serie di funzioni già pronte. In realtà, più che di funzioni vere e proprie, si tratta di operatori unari che intervengono sull'argomento posto alla loro destra. Questa precisazione è importante perché serve a comprendere meglio il meccanismo con cui Perl interpreta le chiamate di tali funzioni od operatori.

Finora si è visto il funzionamento di una funzione molto semplice, '**`print`**'. Questa emette il risultato dell'operando che si trova alla sua destra, ma solo del primo. Se ciò che appare alla destra di '**`print`**' è un'espressione, la valutazione dell'insieme '**`print`** *espressione*', dipende dalle precedenze tra gli operandi. Infatti:

```
print 1+2+4;
```

restituisce sette;

```
print (1+2)+4;
```

restituisce tre;

```
print (1+2+4);
```

restituisce sette.

Utilizzando le funzioni di Perl nello stesso modo in cui si fa negli altri linguaggi, racchiudendo l'argomento tra parentesi, si evitano ambiguità; soprattutto, in questo modo, sembrano essere veramente funzioni anche se si tratta di operatori.

L'argomento di queste funzioni di Perl (ovvero l'operando) può essere uno scalare o una lista. In questo caso quindi, così come lo scalare non ha la necessità di essere racchiuso tra parentesi, anche la lista non lo ha. Resta in ogni caso il fatto che ciò sia almeno consigliabile per migliorare la leggibilità del programma. Il capitolo 298 elenca e descrive alcune di queste funzioni.

295.8 Input/Output dei dati

L'I/O può avvenire sia attraverso l'uso dei flussi standard di dati (standard input, standard output e standard error) che utilizzando file differenti. I flussi di dati standard sono trattati come file normali, con la differenza che generalmente non devono essere aperti o chiusi.

Assieme alla gestione dei file si affianca la possibilità di eseguire comandi del sistema operativo, in parte descritta nella sezione dedicata agli operatori di delimitazione di stringhe.

295.8.1 Esecuzione di comandi di sistema

Una stringa racchiusa tra apici inversi, oppure indicata attraverso l'operatore di stringa '**`qx`**', viene interpolata e il risultato viene fatto eseguire dal sistema operativo.

L'output del comando è il risultato della valutazione della stringa e il valore restituito dal comando può essere letto dalla variabile predefinita '**`$?`**'. È importante ricordare che generalmente i comandi del sistema operativo restituiscono un valore pari a zero quando l'operazione ha avuto successo. Dal punto di vista di Perl, quando '**`$?`**' contiene il valore *Falso* significa che il comando ha avuto successo.

L'esempio seguente dovrebbe rendere l'idea.

```
#!/usr/bin/perl

# $elenco riceve l'elenco di file in forma di un'unica stringa.
```

```
$elenco = ‘ls *.pl‘;

if ($? == 0)
  {
    # L’operazione ha avuto successo e viene visualizzato l’elenco.
    print "$elenco\n";
  }
else
  {
    # L’operazione è fallita.
    print "Non ci sono programmi Perl\n";
  }
```

295.8.2 Gestione dei file

Perl, come molti altri linguaggi, gestisce i file come flussi, o *file handle*, che sono un riferimento interno a un file aperto. I flussi di file vengono indicati attraverso un nome, che per convenzione è espresso quasi sempre attraverso lettere maiuscole.

Perl mette a disposizione tre flussi di file predefiniti: ‘**STDIN**’, ‘**STDOUT**’ e ‘**STDERR**’. Questi corrispondono rispettivamente ai flussi di standard input, standard output e standard error. Altri file possono essere utilizzati aprendoli attraverso la funzione ‘**open()**’, con cui si abbina un flusso al file reale.

Perl è predisposto per gestire agevolmente i file di testo, cioè quelli organizzati convenzionalmente in righe terminanti con il codice di interruzione di riga. Si valuta un flusso di file, come se si trattasse di una variabile, racchiudendone il nome tra parentesi angolari, ottenendo la lettura e la restituzione di una riga, ogni volta che avviene tale valutazione. Per esempio,

```
#!/usr/bin/perl

while (defined ($riga = <STDIN>))
 {
    print $riga;
 }
```

emette attraverso lo standard output ciò che riceve dallo standard input. Quindi, la lettura del flusso di file attraverso la semplice valutazione dell’espressione, restituisce una riga fino al codice di interruzione di riga incluso. In questo modo, nell’esempio non è necessario aggiungere il codice ‘**\n**’ nell’argomento della funzione ‘**print**’.

Se un flusso di file è l’unica cosa che appare nella condizione di un ciclo ‘**while**’ o ‘**for**’, la sua valutazione genera la lettura della riga e il suo inserimento all’interno della variabile predefinita ‘**$**_’. Questo fatto può essere usato convenientemente considerando che quando si raggiunge la fine, la valutazione del flusso di file genera un valore indefinito, pari a *Falso* in una condizione. I due esempi seguenti sono identici al quello mostrato poco sopra.

```
#!/usr/bin/perl

while (<STDIN>)
  {
    print $_;
  }
```

```
#!/usr/bin/perl

for ( ; <STDIN>; )
  {
    print $_;
  }
```

Un flusso di file può essere valutato in un contesto lista. In tal caso restituisce tutto il file in una lista in cui ogni elemento è una riga. Naturalmente ciò viene fatto a spese della memoria di elaborazione.

```
#!/usr/bin/perl

@mio_file = <STDIN>;
print @mio_file;
```

L'esempio appena mostrato si comporta come gli altri visti finora: restituisce lo standard input attraverso lo standard output.[6]

295.8.3 File globbing

Perl, se non riconosce ciò che trova all'interno di parentesi angolari come un flusso di file, tratta questo come un modello per indicare nomi di file, che viene valutato ottenendo l'elenco dei nomi corrispondenti. In pratica, la valutazione di '**<*.pl>**' restituisce l'elenco dei nomi dei file che terminano con l'estensione '.pl' nella directory corrente. Generalmente è preferibile eseguire un tipo di valutazione del genere in un contesto lista, come nell'esempio seguente:

```
#!/usr/bin/perl

@mioelenco = <*.pl>;
print "@mioelenco\n";
```

In alternativa si può utilizzare la funzione interna '**glob()**', come nell'esempio seguente:

```
#!/usr/bin/perl

@mioelenco = glob ("*.pl");
print "@mioelenco\n";
```

295.9 Funzioni definite dall'utente

Le funzioni definite dall'utente, o subroutine se si preferisce il termine, possono essere collocate in qualunque parte del sorgente Perl. Eventualmente possono anche essere caricate da file esterni. I parametri delle funzioni vengono passati nello stesso modo in cui si fa per le funzioni predefinite, interne a Perl: attraverso una lista di elementi scalari. Le funzioni ottengono i parametri dall'array predefinito '@_'. Il valore restituito dalle funzioni è quello dell'ultima istruzione eseguita all'interno della funzione: solitamente si tratta di '**return**' che permette di controllare meglio la cosa.

La sintassi normale per la dichiarazione di una funzione è la seguente. Le parentesi graffe vanno intese in modo letterale e non fanno parte della descrizione del modello sintattico.

```
sub nome { istruzione… }
```

Per la chiamata di una funzione si deve usare la forma seguente:

```
&nome (parametro,…)
```

L'uso della e-commerciale ('**&**') all'inizio del nome è opportuno anche se non è strettamente obbligatorio: permette di evitare ambiguità se il nome della funzione è stato usato per altri tipi di entità all'interno del programma Perl.

```
#!/usr/bin/perl

sub somma
```

[6] La funzione '**print**' ha l'argomento senza virgolette perché altrimenti inserirebbe uno spazio indesiderato tra un elemento e l'altro.

```
{
    return ($_[0] + $_[1]);
}

# I valori da sommare vengono indicati nella riga di comando.
$totale = &somma ($ARGV[0], $ARGV[1]);

print "$ARGV[0] + $ARGV[1] = $totale\n";
```

L'esempio mostrato sopra dovrebbe chiarire il ruolo dell'array '`@_`' all'interno della funzione, come mezzo per il trasporto dei parametri di chiamata.

295.9.1 Chiamata per riferimento e chiamata per valore

L'array '`@_`' è costruito attraverso riferimenti ai parametri utilizzati originariamente nella chiamata. Ciò è sufficiente a fare in modo che modificando il contenuto dei suoi elementi, queste modifiche si riflettano sui parametri di chiamata. Si ha in tal modo quello che si definisce *chiamata per riferimento* , in cui la funzione è in grado di modificare le variabili utilizzate come parametri.

> Naturalmente ciò ha senso solo se i parametri utilizzati sono espressi in forma di variabile e come tali possono essere modificati. Tentare di modificare una costante produce un errore irreversibile.

Dal momento che l'array '`@_`' contiene riferimenti ai dati originali, assegnando all'array un'altra lista di valori non si alterano i dati originali, ma si perde il contatto con quelli. Quindi, non si può assegnare a tale array una lista come modo rapido di variare tutti i parametri della chiamata.

Per gestire elegantemente una funzione che utilizzi il sistema della chiamata per valore, si può fare come nell'esempio seguente:

```
sub miasub
{
        local ($primo, $secondo, $terzo) = @_;
        ...
        return ...;
}
```

In tal modo, agendo successivamente solo sulle variabili scalari ottenute non si modifica l'array '`@_`', e lo stesso codice diventa più leggibile.

295.9.2 Campo di azione delle variabili

Perl gestisce tre tipi di campi di azione per le variabili (di solito si usa il termine *scope* per fare riferimento a questo concetto). Si tratta di variabili *pubbliche*, *private* e *locali*.

Le variabili pubbliche sono accessibili in ogni punto del programma, senza alcuna limitazione, a meno che vengano oscurate localmente. Si ottiene una variabile pubblica quando questa viene creata senza specificare nulla di particolare.

```
# Inizializzazione di una variabile pubblica.
$pubblica = "ciao";
```

Una variabile privata è visibile solo all'interno del blocco di istruzioni in cui viene creata e dichiarata come tale. Le funzioni chiamate eventualmente all'interno del blocco, non possono accedere alle variabili private dichiarate nel blocco chiamante. Si dichiara una variabile privata attraverso l'istruzione '**`my`**'.

`my` *variabile*
`my` *variabile* `=` *valore*
`my (`*variabile1*`,` *variabile2*`, …)`

Una variabile locale è visibile solo all'interno del blocco di istruzioni in cui viene creata e dichiarata come tale. Le funzioni chiamate eventualmente all'interno del blocco, possono accedere alle variabili locali dichiarate nel blocco chiamante. Si dichiara una variabile locale attraverso l'istruzione '**`local`**'.

`local` *variabile*
`local` *variabile* `=` *valore*
`local (`*variabile1*`,` *variabile2*`, …)`

Sia le variabili private che quelle locali permettono di utilizzare un nome già esistente a livello globale, sovrapponendosi temporaneamente a esso. Quelle locali, in particolare, hanno valore anche per le funzioni chiamate all'interno dei blocchi in cui queste variabili sono state dichiarate.

Si dice anche che le variabili private abbiano un campo di azione definito in modo lessicale, mentre quelle locali in modo dinamico: terminata la zona di influenza, le variabili locali vengono rilasciate, mentre quelle private no.

Seguono due esempi di calcolo del fattoriale in modo ricorsivo. In un caso si utilizza una variabile privata, nell'altro una locale. Funzionano entrambi correttamente.

```
#!/usr/bin/perl

sub fattoriale
{
    my $valore = $_[0];
    if ($valore > 1)
      {
        return ($valore * &fattoriale ($valore -1));
      }
    else
      {
        return 1;
      }
}

$miofatt = &fattoriale ($ARGV[0]);

print "$ARGV[0]! = $miofatt\n";
```

```
#!/usr/bin/perl

sub fattoriale
{
    local $valore = $_[0];
    if ($valore > 1)
      {
        return ($valore * &fattoriale ($valore -1));
      }
    else
      {
        return 1;
      }
}

$miofatt = &fattoriale ($ARGV[0]);

print "$ARGV[0]! = $miofatt\n";
```

295.10 Variabili contenenti riferimenti

Si è accennato al fatto che una variabile scalare può contenere anche *riferimenti* , oltre a valori stringa o numerici. Il riferimento è un modo alternativo per puntare a un'entità determinata del programma. La gestione di questi riferimenti da parte di Perl è piuttosto complessa. Qui vengono analizzate solo alcune caratteristiche e possibilità.

Perl gestisce due tipi di riferimenti: diretti (*hard*) e simbolici. Volendo fare un'analogia con quello che accade con i collegamenti dei file system Unix, i primi sono paragonabili ai collegamenti fisici (gli *hard link*), mentre i secondi sono simili ai collegamenti simbolici.

295.10.1 Riferimenti diretti

I riferimenti diretti vengono creati utilizzando l'operatore barra obliqua inversa ('\'), come negli esempi seguenti.

```
$rifscalare     = \$mioscalare;

$rifarray       = \@mioarray;

$rifhash        = \%miohash;

$rifcodice      = \&miafunzione;

$rifflusso      = \*MIO_FILE;
```

Esiste anche una forma sintattica alternativa di esprimere i riferimenti: si tratta di indicare il nome dell'entità per la quale si vuole creare il riferimento, preceduto da un asterisco e seguito dalla definizione del tipo a cui questa entità appartiene, tra parentesi graffe.

```
$rifscalare     = *mioscalare{SCALAR};

$rifarray       = *mioarray{ARRAY};

$rifhash        = *miohash{HASH};

$rifcodice      = *miafunzione{CODE};

$rifflusso      = *MIO_FILE{IO};
```

> Perl riconosce anche il tipo '**FILEHANDLE**' equivalente a '**IO**', per motivi di compatibilità con il passato.

295.10.2 Riferimenti simbolici

I riferimenti simbolici sono basati sul nome dell'entità a cui si riferiscono, per cui, una variabile scalare contenente il nome dell'oggetto può essere gestita come un riferimento simbolico. Seguono alcuni degli esempi visti nel caso dei riferimenti diretti, in quanto con questo tipo di riferimenti non si possono gestire tutte le situazioni.

```
$rifscalare = 'mioscalare';

$rifarray   = 'mioarray';

$rifhash    = 'miohash';

$rifcodice  = 'miafunzione';
```

Generalmente, l'utilizzo di riferimenti simbolici è sconsigliabile, a meno che ci sia una buona ragione.

295.10.3 Dereferenziazione

Restando in questi termini, a parte il caso dei flussi di file, il modo per *dereferenziare* le variabili che contengono i riferimenti è uguale per entrambi i tipi. La forma normale richiede l'utilizzo delle parentesi graffe per delimitare lo scalare. In precedenza si era visto che una variabile scalare poteva essere indicata attraverso la forma '**${***nome***}**'. Estendendo questo concetto, racchiudendo tra parentesi graffe un riferimento, si ottiene l'oggetto stesso. Per cui:

```
${$rifscalare}
```

equivale a utilizzare '**$mioscalare**';

```
${$rifscalare}[0]
```

equivale a utilizzare '**$mioarray[0]**';

```
${$rifhash}{primo}
```

equivale a utilizzare '**$miohash{primo}**';

```
&{$rifcodice} (1, 7)
```

equivale a utilizzare '**&miafunzione (1, 7)**'.

Sono anche ammissibili altre forme, più espressive o più semplici. La tabella 295.7 riporta alcuni esempi con le forme possibili per dereferenziare gli scalari contenenti dei riferimenti.

Tabella 295.7. Esempi attraverso cui dereferenziare le variabili scalari contenenti dei riferimenti.

${$rifscalare}	$$rifscalare	
${$rifscalare}[0]	$$rifscalare[0]	$rifscalare->[0]
${$rifhash}{primo}	$$rifhash{primo}	$rifhash->{primo}
&{$rifcodice} (1, 7)	&$rifcodice (1, 7)	$rifcodice-> (1, 7)

Il caso dei flussi di file è più semplice, in quanto è sufficiente valutare il riferimento, invece del flusso di file vero e proprio. L'esempio seguente dovrebbe chiarire il meccanismo.

```
$rifstdio  = \*STDIO;
$riga = <$rifstdio>;
```

295.10.4 Array multidimensionali

Gli array di Perl hanno una sola dimensione. Per ovviare a questo inconveniente si possono utilizzare elementi che fanno riferimento ad altri array. In pratica, si potrebbe fare qualcosa di simile all'esempio seguente:

```
@primo   = (1, 2);
@secondo = (3, 4);

@mioarray = (\@primo, \@secondo);
```

Qui, l'array '**mioarray**' rappresenta una matrice a due dimensioni rappresentabile nel modo seguente:

```
/ 1  2 \
|      |
\ 3  4 /
```

Per accedere a un elemento singolo di questo array, per esempio al primo elemento della seconda riga (il numero tre), si può usare intuitivamente una di queste due forme.

```
${$mioarray[1]}[0]

$mioarray[1]->[0]
```

In alternativa è concessa anche la forma seguente, più semplice e simile a quella di altri linguaggi.

```
$mioarray[1][0]
```

Una particolarità di Perl sta nella possibilità di definire delle entità anonime. Solitamente si tratta di variabili che non hanno un nome e a cui si accede attraverso uno scalare contenente un riferimento diretto al loro contenuto. Il caso più interessante è dato dagli array, perché questa possibilità permette di definire istantaneamente un array multidimensionale. L'array dell'esempio precedente poteva essere dichiarato nel modo seguente:

```
@mioarray = ([1, 2], [3, 4]);
```

La gestione pratica di un array multidimensionale secondo Perl, potrebbe sembrare un po' complessa a prima vista. Tuttavia, basta ricordare che si tratta di array dinamici, per cui, basta assegnare un elemento per dichiararlo implicitamente:

```
@mio_array = ();
...
$mio_array[0] = "ciao";
$mio_array[1] = "come";
$mio_array[2] = "stai";
...
```

Come si vede, viene dichiarato l'array senza elementi, al quale questi vengono inseriti solo successivamente. Così facendo, la dimensione dell'array varia in base all'uso che se ne fa. Con questo criterio si possono gestire anche gli array multidimensionali:

```
@mio_array = ();
...
$mio_array[0] = ();
...
$mio_array[0][0] = "ciao";
$mio_array[0][1] = "come";
$mio_array[0][2] = "stai";
...
```

In questo caso, dopo aver dichiarato l'array '**`@mio_array`**', senza elementi, viene dichiarato il primo elemento come contenente un altro array vuoto; infine, vengono dichiarati i primi tre elementi di questo sotto-array. Il funzionamento dovrebbe essere intuitivo, anche se si tratta effettivamente di un meccanismo molto complesso e potente.

Di fronte a array multidimensionali di questo tipo, potenzialmente irregolari, si può porre il problema di conoscere la lunghezza di un sotto-array. Volendo usare la tecnica del prefisso '**`$#`**', si potrebbe fare come nell'esempio seguente, per determinare la lunghezza dell'array contenuto in '**`$mio_array[0]`**'.

```
$ultimo = $#{$mio_array[0]};
```

295.10.5 Alias

Attraverso l'uso dei riferimenti, è possibile creare un alias di una variabile. Per comprendere questo è necessario introdurre l'uso dell'asterisco. Si osservi questo esempio: se '**`$variabile`**' rappresenta una variabile scalare, '**`*variabile`**' rappresenta il puntatore alla variabile omonima. In un certo senso, '**`*variabile`**' è equivalente a '**`\$variabile`**', ma non è proprio la stessa cosa. Si osservino gli assegnamenti seguenti, supponendo che esista già la variabile '**`$tua`**' e si tratti di uno scalare.

```
*mia = \$tua;
*mia = *tua;
```

I due assegnamenti sono identici, perché in entrambi i casi si assegna a '***mia**' il riferimento alla variabile scalare '**$tua**'. Il risultato di questo è che si può usare la variabile scalare '**$mia**' come alias di '**$tua**'. L'esempio seguente dovrebbe chiarire meglio la cosa.

```
#!/usr/bin/perl
$tua = "ciao";
*mia = \$tua;
print "$mia\n";
```

Quello che si ottiene è l'emissione della stringa '**ciao**', cioè il contenuto della variabile '**$tua**', ottenuto attraverso l'alias '**$mia**'.

Attraverso gli alias è possibile gestire agevolmente il passaggio di parametri per riferimento nelle chiamate delle funzioni. Si osservi l'esempio seguente, in cui una funzione altera il contenuto di un array, senza che questo debba essere dichiarato come variabile globale.

```
#!/usr/bin/perl
sub alterazione_array
{
  local (*a) = $_[0];
  $a[0] = 1;
  $a[1] = 2;
}

local ($b) = ();

$b[0] = 9;
$b[1] = 8;
$b[2] = 7;

&alterazione_array (\@b);

print STDOUT ($a1[0] . " " . $a1[1] . " " . $a1[2] . "\n");
```

Eseguendo questo programmino molto semplice, si ottiene la stringa seguente:

```
1 2 7
```

Questo serve a dimostrare che i primi due elementi dell'array sono stati modificati dalla funzione.

295.11 Avvio di Perl

Normalmente è sufficiente rendere eseguibile uno script Perl per fare in modo che il programma '`/usr/bin/perl`' venga eseguito automaticamente per la sua interpretazione. Il programma '`/usr/bin/perl`' permette di utilizzare alcune opzioni, principalmente utili per individuare errori sintattici e problemi di altro tipo.

Alcune opzioni

`-c`

Analizza sintatticamente lo script e termina senza eseguirlo.

`-w`

Viene usato assieme a '**-c**' e permette di avere informazioni più dettagliate su problemi eventuali che non sono necessariamente considerabili come errori sintattici.

`-d`

Esegue lo script all'interno di un sistema diagnostico di *debug*.

Esempi

```
$ perl mio.pl
```

Avvia il programma Perl '**mio.pl**'. Generalmente si avvia direttamente lo script, ma se questo non è stato reso eseguibile attraverso i permessi, si può ovviare in questo modo.

```
$ perl -c mio.pl
```

Analizza lo script '**mio.pl**' senza eseguirlo. Se tutto va bene si ottiene l'output seguente:

```
mio.pl syntax OK
```

```
$ perl -c -w mio.pl
```

Come nell'esempio precedente, con l'aggiunta dell'opzione '**-w**', con la quale si ottengono maggiori indicazioni e suggerimenti per migliorare il programma.

```
$ perl -d mio.pl
```

Avvia il sistema diagnostico per il programma '**mio.pl**'.

Perl: gestione delle stringhe

La gestione delle stringhe da parte di Perl è fatta attraverso gli operatori di delimitazione delle stringhe stesse e le espressioni regolari. È questo insieme di cose che rende Perl uno strumento valido per la gestione dei file di testo.

296.1 Operatori di delimitazione di stringhe

Nella sezione dedicata agli operatori e alle espressioni erano rimasti in sospeso gli operatori di delimitazione di stringhe. Nei linguaggi di programmazione tradizionale esiste normalmente il problema di delimitare le stringhe, ovvero le costanti alfanumeriche. Sono già stati mostrati due tipi di delimitatori, gli apici doppi e singoli che hanno un comportamento simile a quello delle shell comuni. In realtà Perl ha una gestione molto più raffinata e generalizzata delle stringhe. Quando il tipo di delimitazione, ovvero il tipo di stringa, lo consente, sono validi alcuni codici di escape. La tabella 296.1 mostra l'elenco di queste sequenze di escape utilizzabili nelle stringhe.

Tabella 296.1. Elenco delle sequenze di escape utilizzabili nelle stringhe delimitate con gli apici doppi.

Escape	Corrispondenza
\\	\
\"	"
\$	$
\@	@
\'	'
\t	*<HT>*
\n	*<LF>*
\r	*<CR>*
\f	*<FF>*
\a	*<BEL>*
\e	*<ESC>*
\0*n*	Numero ottale rappresentato da *n*.
\x*h*	Numero esadecimale rappresentato da *h*.
\[	Carattere di controllo.
\l	Il carattere successivo in minuscolo.
\u	Il carattere successivo in maiuscolo.
\L	Minuscolo fino al codice '**\E**'.
\U	Maiuscolo fino al codice '**\E**'.
\E	Conclusione di un modificatore.
\Q	Evita l'interpretazione come espressione regolare fino al codice '**\E**'.

La delimitazione dei vari tipi di stringa avviene in una forma tradizionale, attraverso delimitatori che esprimono di per sé il tipo di stringa, oppure attraverso una forma che consente di cambiare tipo di delimitatore:

x delim_sinistro stringa delim_destro eventuali_opzioni

La sigla che appare inizialmente, *x* in questo caso, definisce il tipo di stringa; il delimitatore sinistro e quello destro possono essere parentesi aperte e chiuse di qualunque tipo: tonde, quadre, graffe e angolari, ma si possono utilizzare anche altri simboli, solo che in tal caso, il delimitatore sinistro e quello destro saranno uguali.

La tabella 296.5, alla fine di questo gruppo di sezioni, riassume i vari tipi di operatori di delimitazione delle stringhe.

296.1.1 q | ' ' -- stringa letterale non interpolata

Si tratta della stringa racchiusa normalmente tra apici singoli. È già stata descritta in precedenza. In particolare, restituisce la stringa racchiusa senza effettuare l'interpolazione delle eventuali variabili e dei simboli di escape che dovesse incorporare, a eccezione di '`\'`' e '`\\`'. Si può esprimere in due modi.

'*stringa*'
q*delim_sinistro stringa delim_destro*

Seguono alcuni esempi.

```
$miavar = 'Stringa tradizionale che non interpola';

$miavar = q|Una stringa che "contiene 'apici' di ogni tipo".|;

$miavar = q(Sembra una funzione, ma non lo è);

$miavar = q{Le variabili non vengono interpolate. $ciao};
```

296.1.2 qq | " " -- stringa letterale interpolata

Si tratta della stringa racchiusa normalmente tra apici doppi. È già stata descritta in precedenza. In particolare, restituisce la stringa racchiusa interpolando le variabili e i simboli di escape che dovesse incorporare. Si può esprimere in due modi.

"*stringa*"
qq*delim_sinistro stringa delim_destro*

Seguono alcuni esempi.

```
$miavar = "Stringa tradizionale che interpola";

$miavar = qq|Una stringa che \"contiene 'apici' di ogni tipo\".|;

$miavar = qq(Sembra una funzione, ma non lo è);

$ciao = "Saluti!";
$miavar = qq{Le variabili vengono interpolate. $ciao};
```

296.1.3 qx | ` ` -- comando di sistema

Si tratta di stringhe il cui contenuto deve essere valutato e successivamente eseguito come comando dal sistema operativo. Questo tipo di stringa è racchiuso normalmente tra apici singoli inversi, come avviene nelle shell comuni. Il contenuto della stringa viene interpolato prima dell'esecuzione del comando. La valutazione della stringa si traduce nell'output emesso attraverso lo standard output dal comando stesso. Si può esprimere in due modi.

`*stringa*`
qx*delim_sinistro stringa delim_destro*

Seguono alcuni esempi.

```
$miadata = `date`;

$mioelenco = qx(ls);

$opzioni = '-l'
$mioelenco = qx{ls $opzioni};
```

296.1.4 qw -- lista di parole

La stringa racchiusa in questo tipo di delimitazione, non viene interpolata, ma semplicemente restituita in forma di *lista di parole*. In pratica, tutto ciò che risulta separato da spazi (spazi veri e propri, caratteri di tabulazione e codici di interruzione di riga) viene estratto e inserito in una lista di elementi. Si può esprimere solo nel modo seguente:

qw*delim_sinistro stringa delim_destro*

Seguono alcuni esempi validi.

```
@mialista = qw/ciao come stai/;

@mialista = qw(uno      due     tre);

@mialista = qw(alfa     bravo   charlie
delta   echo    foxtrot golf    hotel
india   kilo    lima);
```

296.1.5 m | // -- modello di confronto

Definisce un modello di confronto con una stringa. Non restituisce alcunché; serve per essere paragonato a un'altra stringa. Può essere usato in un contesto scalare o lista. Nel primo caso serve a determinare se esiste una corrispondenza con il modello o meno. Nel secondo caso, viene sempre paragonato a un'altra stringa, ma il risultato di questo abbinamento è una lista di elementi.

Il modello si esprime in forma di espressione regolare, con delle particolarità che derivano dal tipo di delimitatori utilizzati e dal fatto che prima di valutare l'espressione regolare viene eseguita un'interpolazione. Si può esprimere in due modi.

/ stringa / opzioni
m*delim_sinistro stringa delim_destro modificatori*

I modificatori si esprimono con una serie di lettere, o nulla se non è necessario. La tabella 296.2 ne riporta l'elenco.

Tabella 296.2. Elenco dei modificatori utilizzabili con l'operatore di delimitazione '`m`'.

Modificatore	Descrizione
i	Il confronto avviene ignorando la differenza tra maiuscole e minuscole.
m	Le stringhe vengono trattate come righe multiple (riguarda '`^`' e '`$`').
s	Tratta le stringhe come una riga singola (riguarda '`.`').
x	Permette l'inserzione di spazi e commenti che non vengono interpretati.
g	Confronta in modo globale, cioè trova tutte le occorrenze.
o	Interpreta il modello (e di conseguenza lo interpola) solo la prima volta.

L'utilizzo delle espressioni regolari nelle istruzioni Perl è ciò che generalmente rende il sorgente di un programma piuttosto confuso. Se si devono utilizzare intensivamente le espressioni regolari sarebbe opportuno approfondirne il funzionamento e l'utilizzo di questo tipo di delimitatori, per trovare un modo meno complicato del solito di scrivere queste espressioni. Il primo punto su cui si può intervenire è la scelta dei simboli di delimitazione. La forma tradizionale prevede l'uso della barra obliqua normale, cosa però che crea problemi quando si vuole utilizzare questo simbolo all'interno dell'espressione stessa. Infatti, i simboli usati come delimitazione non possono essere utilizzati nell'espressione regolare senza la tecnica della protezione per mezzo del prefisso '\'.

Esempi

```
#!/usr/bin/perl

$miafrase = 'Ciao, come stai?';
if ($miafrase =~ /ciao/i)
  {
    print "Ciao!\n";
  }
```

In questo esempio, il modello '`/ciao/i`' combacia con una parte della frase, facendo sì che la condizione si avveri.

```
#!/usr/bin/perl

$mioelenco = `ls`;
if ($mioelenco =~ /.*\.pl/)
  {
    print "Ci sono programmi Perl in questa directory.\n";
  }
```

In questo esempio, viene letto il contenuto della directory corrente e posto nella variabile '**`$mioelenco`**'. Successivamente viene verificato se in quell'elenco si trova qualcosa che termina con '`.pl`'. Dal momento che il punto ha un significato nelle espressioni regolari, per poterlo includere si è posta anteriormente una barra obliqua inversa.

296.1.6 s -- modello di sostituzione

Definisce un modello di confronto con una stringa, assieme a una stringa di sostituzione per la parte che corrisponde al modello. Se il confronto non viene fatto attraverso gli operatori '**=~**' oppure '**!~**', si intende che l'abbinamento avvenga con il contenuto della variabile '**$**_'. Ha luogo l'interpolazione.

L'abbinamento per la sostituzione può avvenire solo in un contesto scalare. Il modello si esprime in forma di espressione regolare. La sintassi può essere espressa in due modi, a seconda del tipo di delimitatori utilizzati.

s*delim_sx stringa delim_dx delim_sx rimpiazzo delim_dx modificatori*
s*delim stringa delim rimpiazzo delim modificatori*

Il primo tipo di sintassi si adatta al caso in cui si usino parentesi per delimitare le stringhe del modello e del rimpiazzo, il secondo tipo si riferisce all'uso di altri simboli che non sono utilizzati in coppia.

I modificatori si esprimono con una serie di lettere, o nulla se ciò non è necessario. La tabella 296.3 ne riporta l'elenco.

Tabella 296.3. Elenco dei modificatori utilizzabili con l'operatore di delimitazione '**s**'.

Modificatore	Descrizione
i	Il confronto avviene ignorando la differenza tra maiuscole e minuscole.
m	Le stringhe vengono trattate come righe multiple (riguarda '**^**' e '**$**').
s	Tratta le stringhe come una singola riga (riguarda '**.**').
x	Permette l'inserzione di spazi e commenti che non vengono interpretati.
g	Confronta in modo globale, cioè trova tutte le occorrenze.
o	Interpreta il modello (e di conseguenza lo interpola) solo la prima volta.
e	Valuta la parte destra come un'espressione.

Esempi

```
$path =~ s|/usr/bin|/usr/local/bin|
```

Sostituisce la prima occorrenza di '`/usr/bin`' nella variabile '**`$path`**' con '`/usr/local/bin`'. Per delimitare il modello e la stringa di sostituzione sono state usate le barre verticali, per evitare ambiguità con le barre oblique delle directory.

```
$path =~ s{/usr/bin}{/usr/local/bin}
```

Esattamente come nell'esempio precedente, ma questa volta sono state usate le parentesi graffe.

296.1.7 tr | y -- traslazione di caratteri

Definisce un modello di sostituzione di una serie di caratteri in un'altra. Si applica al contenuto di una variabile scalare utilizzando l'operatore '**`=~`**' oppure '**`!~`**', altrimenti si intende la variabile '**`$_`**'. Restituisce il numero di trasformazioni eseguite. Non ha luogo l'interpolazione.

L'abbinamento per la sostituzione può avvenire solo in un contesto scalare. Il modello si esprime in forma di espressione regolare. La sintassi può essere espressa nei modi seguenti, a seconda che si voglia utilizzare l'identificatore '**`tr`**' o '**`y`**' e a seconda del tipo di delimitatori utilizzati.

`tr`*delim_sx car_da_sost delim_dx delim_sx rimpiazzo delim_dx modificatori*
`tr`*delim car_da_sostituire delim rimpiazzo delim modificatori*
`y`*delim_sx car_da_sost delim_dx delim_sx rimpiazzo delim_dx modificatori*
`y`*delim car_da_sostituire delim rimpiazzo delim modificatori*

I modificatori si esprimono con una serie di lettere, o nulla se ciò non è necessario. La tabella 296.4 ne riporta l'elenco.

Tabella 296.4. Elenco dei modificatori utilizzabili con l'operatore di delimitazione '**`tr`**'.

Modificatore	Descrizione
c	Cerca gli elementi che **non** sono elencati nel gruppo da sostituire.
d	Cancella i caratteri trovati e non rimpiazzati.
s	Fonde insieme i caratteri doppi che sono stati ritrovati.

Tabella 296.5. Elenco riassuntivo dei tipi di operatori di stringa. Le parentesi graffe rappresentano la posizione dei delimitatori.

Formato normale	Formato generico	Significato	Interpolazione
' '	q{}	Stringa letterale.	NO
" "	qq{}	Stringa letterale.	SÌ
` `	qx{}	Comando di sistema.	SÌ
	qw{}	Lista di parole.	NO
/ /	m{}	Modello di confronto.	SÌ
	s{}{}	Modello di sostituzione.	SÌ
	tr{}{}	Traslazione di caratteri.	SÌ

Esempi

```
$miavar =~ tr/A-Z/a-z/;
```

Converte in minuscolo il contenuto della variabile (a parte le vocali accentate).

```
$contatore = ($miavar =~ tr/0-9//);
```

Conta i caratteri numerici contenuti nella variabile '**`$miavar`**'.

296.2 Espressioni regolari

Le espressioni regolari possono essere considerate l'elemento più potente e più difficile di Perl. Purtroppo non esiste una definizione e uno standard universale delle espressioni regolari, così, per ogni applicazione che ne fa uso occorre studiarne le particolarità.

In questa sezione si descrive solo parte delle potenzialità di Perl con le espressioni regolari. Per conoscerne i dettagli è necessario consultare la pagina di manuale *perlre(1)*. Può essere conveniente anche la lettura della sezione 77.1 e del capitolo 316.

296.2.1 Modificatori

Perl utilizza le espressioni regolari con gli operatori di stringa '**`m{}`**' e '**`s{}{}`**'. Con questi è possibile utilizzare delle opzioni finali, ovvero dei modificatori, che alterano le regole delle espressioni regolari. La tabella 296.6 mostra l'elenco dei modificatori più comuni.

Tabella 296.6. Elenco dei modificatori utilizzabili in generale in coda alle espressioni regolari di Perl.

Modificatore	Descrizione
i	Il confronto avviene ignorando la differenza tra maiuscole e minuscole.
m	Le stringhe vengono trattate come righe multiple (riguarda '**`^`**' e '**`$`**').
s	Tratta le stringhe come una singola riga (riguarda '**`.`**').
x	Permette l'inserzione di spazi e commenti che non vengono interpretati.

296.2.2 Metacaratteri

In generale, i caratteri utilizzati in un'espressione regolare, che non abbiano un significato speciale, corrispondono a loro stessi nella stringa di comparazione. Ciò è come dire che la comparazione seguente è valida.

```
'Ciao' =~ /Ciao/
```

I *metacaratteri* di un'espressione regolare sono dei simboli che hanno un significato diverso rispetto ai caratteri utilizzati per rappresentarli. La tabella 296.7 mostra l'elenco dei metacaratteri più comuni.

Tabella 296.7. Elenco dei metacaratteri standard utilizzati in Perl.

Metacarattere	Descrizione
\	Protegge il carattere seguente da un'interpretazione diversa da quella letterale.
^	Corrisponde all'inizio di una riga.
.	Corrisponde a un carattere qualunque.
$	Corrisponde alla fine di una riga.
|	Indica due possibilità alternative alla sua sinistra e alla sua destra.
( )	Definiscono un raggruppamento.
[ ]	Definiscono una classe di caratteri.

La barra obliqua inversa protegge il carattere successivo da un'interpretazione diversa da quella letterale, quando la sequenza '*x*' (*x* rappresenta qui un carattere qualunque) non rappresenta già un metacarattere. In pratica, se '*x*' non ha un significato particolare, rappresenta semplicemente '*x*' in modo letterale.

L'accento circonflesso ('**`^`**') corrisponde generalmente all'inizio di una riga; nello stesso modo, il

simbolo dollaro (‘**$**’) rappresenta la fine di una riga. Questi metacaratteri rappresentano in pratica la stringa nulla di inizio e di fine di una riga. Se la stringa da analizzare è composta da più righe terminate dal codice di interruzione di riga, è possibile fare in modo che ‘**^**’ e ‘**$**’ corrispondano all’inizio e alla fine di queste righe virtuali utilizzando il modificatore ‘**m**’.

Il punto rappresenta un carattere singolo, con l’esclusione del codice di interruzione di riga a meno che sia stato utilizzato il modificatore ‘**s**’.

Perl aggiunge a quelli standard una serie di metacaratteri rappresentati dalla tabella 296.8.

Tabella 296.8. Elenco dei metacaratteri speciali di Perl.

Metacarattere	Corrispondenza
\w	Un carattere alfanumerico (lettere e numeri) compreso il trattino basso.
\W	Un carattere non alfanumerico (l’opposto di ‘**\w**’).
\s	Uno spazio lineare (spazio o tabulazione).
\S	Qualunque carattere che non sia uno spazio lineare.
\d	Un carattere numerico.
\D	Un carattere non numerico.
\b	La stringa nulla prima o dopo una sequenza di caratteri corrispondenti a ‘**\w**’.
\B	La stringa nulla interna a una sequenza di caratteri corrispondenti a ‘**\w**’.
\A	L’inizio di una stringa.
\Z	La fine di una stringa (eventualmente prima di un *newline* finale).

Inoltre, per complicare ulteriormente le cose, le espressioni regolari di Perl vengono trattate come se fossero racchiuse tra apici doppi, cioè vengono interpolate prima di essere valutate come espressioni regolari. Questo significa che le variabili vengono espanse e vengono riconosciuti anche altri simboli che in pratica potrebbero essere considerati come dei metacaratteri aggiuntivi. Si tratta di ‘**\n**’, ‘**\t**’ e altri come già indicato nella tabella 296.1 all’inizio del capitolo.

296.2.3 Classi di caratteri

Un modello racchiuso tra parentesi quadre rappresenta un solo carattere in base a quanto indicato nelle parentesi.

Una fila di caratteri racchiusa tra parentesi quadre corrisponde a un carattere qualunque tra quelli indicati; se all’inizio di questa fila c’è l’accento circonflesso, si ottiene una corrispondenza con un carattere qualunque diverso da quelli della fila. Per esempio, l’espressione regolare ‘**[0123456789]**’ corrisponde a una cifra numerica qualunque, mentre ‘**[^0123456789]**’ corrisponde a un carattere qualunque purché non sia una cifra numerica.

All’interno delle parentesi quadre, invece che indicare un insieme di caratteri, è possibile indicarne un intervallo mettendo il carattere iniziale e finale separati da un trattino (‘**-**’). I caratteri che vengono rappresentati in questo modo dipendono dalla codifica che ne determina la sequenza. Per esempio, l’espressione regolare ‘**[9-A]**’ rappresenta un carattere qualsiasi tra: ‘**9**’, ‘**:**’, ‘**;**’, ‘**<**’, ‘**=**’, ‘**>**’, ‘**?**’, ‘**@**’ e ‘**A**’, perché così è la sequenza ASCII.

Questa definizione corrisponde in parte a quella di ‘**grep**’ GNU, in particolare si deve tenere presente che all’interno delle parentesi quadre, ‘**\b**’ corrisponde al carattere <*BS*>.

Un’altra diversità rispetto alle espressioni regolari di altri programmi sta nella mancanza di classi di caratteri espresse attraverso una denominazione. Ciò giustifica la presenza di metacaratteri che non ci sono normalmente. La tabella 296.9 mostra l’abbinamento tra le classi delle espressioni regolari comuni e i metacaratteri corrispondenti di Perl.

Tabella 296.9. Comparazione tra alcuni metacaratteri di Perl e le classi di caratteri equivalenti delle espressioni regolari POSIX.

Perl	Espressioni regolari POSIX
\w	[[:alnum:]]
\W	[^[:alnum:]]
\s	[[:space:]]
\S	[^[:space:]]
\d	[[:digit:]]
\D	[^[:digit:]]

296.2.4 Qualificatori -- operatori di ripetizione

Attraverso altri simboli è possibile indicare la ripetizione di un carattere determinato o di un raggruppamento. La tabella 296.10 mostra l'elenco di queste notazioni e il loro significato.

Tabella 296.10. Operatori di ripetizione, o qualificatori, nelle espressioni regolari di Perl.

Codifica	Corrispondenza.
*x**	Nessuna o più volte *x*. Equivalente a '*x***{0,}**'.
x?	Nessuna o al massimo una volta *x*. Equivalente a '*x***{0,1}**'.
x+	Una o più volte *x*. Equivalente a '*x***{1,}**'.
x{*n*}	Esattamente *n* volte *x*.
x{*n*,}	Almeno *n* volte *x*.
x{*n*,*m*}	Da *n* a *m* volte *x*.
*x**?	Equivale al minimo di '*x******'.
x??	Equivale al minimo di '*x***?**'.
x+?	Equivale al minimo di '*x***+**'.
x{*n*}?	Equivale al minimo di '*x***{***n***}**', ovvero allo stesso '*x***{***n***}**'.
x{*n*,}?	Equivale al minimo di '*x***{***n***,}**'.
x{*n*,*m*}?	Equivale al minimo di '*x***{***n***,***m***}**'.

Dalla tabella si può osservare la presenza di qualificatori insoliti che terminano con un punto interrogativo. Un modello espresso in forma di espressione regolare può corrispondere a una stringa in diversi modi. Generalmente, la corrispondenza dei qualificatori avviene nel modo più ampio possibile. Se è necessario che la corrispondenza avvenga nel modo più ristretto possibile, occorre utilizzare i qualificatori che terminano con il punto interrogativo. Per esempio, di seguito si vedono alcune corrispondenze valide e le zone delle stringhe originali in cui i modelli combaciano.

```
"CIAO" =~ /\w+/
 ^--^

"Ciao, come stai?" =~ /\s/
      ^

"Ciao, come stai? Io sto bene." =~ /\s.*\s/
      ^-----------------^

"Ciao, come stai? Io sto bene." =~ /\s.*?\s/
      ^----^
```

296.2.5 Raggruppamenti

Una o più parti di un'espressione regolare possono essere raggruppate attraverso l'uso delle parentesi tonde. Ciò permette di abbinare tali raggruppamenti ai qualificatori (gli operatori di ripetizione), oppure permette di estrarre ciò che corrisponde al segmento racchiuso tra parentesi, o di potervi fare riferimento. Per esempio, l'espressione '**\s(come\s)+.*\s**' è valida per tutte le stringhe seguenti.

```
"Ciao, come stai? Io sto bene."
"Ciao, come come stai? Io sto bene."
"Ciao, come come come stai? Io sto bene."
...
```

All'interno della stessa espressione regolare è possibile fare riferimento a una corrispondenza parziale contenuta in un raggruppamento. Per farlo si utilizza il metacarattere '*****n*', dove *n* è una sola cifra numerica. In pratica, '**\1**' corrisponde al primo raggruppamento, '**\2**' corrisponde al secondo, proseguendo così, di seguito, fino al nono.

Per esempio, '**(0|0x0)\d*\s\1\d***' è valida per '**0x0123 0x0456**', ma non per '**0x0123 0456**'. Infatti, si fa riferimento alla corrispondenza, non al modello che potrebbe essere ripetuto agevolmente.

Perl permette di utilizzare queste corrispondenze anche al di fuori delle espressioni regolari. Per questo però non si può più utilizzare la notazione '*****n*', ma occorre invece '**$***n*'. In pratica si tratta di variabili predefinite che vengono generate per l'occasione. Per esempio,

```
s/^(\w+)\s+(\w+)/$2 $1/
```

inverte le prime due parole ed elimina gli spazi superflui tra le due. Un altro esempio interessante è il seguente, in cui si estrae la data da una stringa, per gestirla all'interno del programma.

```
if ($miadata =~ m|Data:\s+(\d\d)/(\d\d)/(\d{2,4})|)
  {
    $giorno = $1;
    $mese = $2;
    $anno = $3;
  }
```

Come si può vedere, i delimitatori dell'espressione regolare sono stati sostituiti con le barre verticali, in modo da poter utilizzare le barre oblique per l'espressione stessa senza troppi problemi.

Perl: gestione dei file

La gestione dei file è uno dei punti di forza di Perl. Perl permette di gestire in modo molto semplice i file di testo e i file DBM. Sono presenti ugualmente gli strumenti per la gestione di file di qualunque altro tipo, attraverso l'accesso al singolo byte, ma questo aspetto passa in secondo piano rispetto al resto e qui verrà trascurato.

297.1 Organizzazione generale

Prima di poter accedere in qualunque modo a un file, occorre che questo sia stato aperto all'interno del programma, il quale, da quel punto in poi, vi farà riferimento attraverso il flusso di file.

Per una convenzione diffusa, i nomi attribuiti ai flussi di file sono sempre composti da lettere maiuscole, cosa che facilita il loro riconoscimento all'interno di un sorgente Perl.

Oltre ai file su disco, esistono tre file particolari: standard input, standard output e standard error. Questi risultano sempre già aperti e ai flussi di file corrispondenti si fa riferimento attraverso tre nomi predefiniti: '**`STDIN`**', '**`STDOUT`**' e '**`STDERR`**'.

297.1.1 Apertura

Quando è necessario aprire un file, cioè quando non si tratta dei flussi predefiniti, si utilizza la funzione '**`open()`**'.

`open` *flusso*`,` *file*

La funzione utilizza quindi solo due argomenti: il nome del flusso di file e il nome effettivo del file, eventualmente con l'indicazione del percorso necessario a raggiungerlo. Per esempio,

```
open MIO_FILE, 'mio_file';
```

apre il file '`mio_file`' che si trova nella directory corrente e gli abbina il flusso di file '**`MIO_FILE`**'. Con l'apertura del file si deve definire anche in che modo si intende accedervi. Fondamentalmente si distingue tra lettura e scrittura, ma in realtà si presentano anche altre sfumature. Per poter informare la funzione del modo in cui si intende aprire il file, la stringa che viene utilizzata per indicare il nome del file su disco può contenere dei simboli aggiuntivi che servono proprio per questo. Tali simboli vanno posti quasi sempre di fronte al nome e possono essere spaziati da questo in modo da facilitarne la lettura:

- se non si utilizza alcun simbolo, oppure se si pone '**`<`**' davanti al nome del file, si ottiene l'apertura in lettura (input);
- se si utilizza il simbolo '**`>`**' si intende aprire il file in scrittura (output), troncando inizialmente il file;
- se si utilizza il simbolo '**`>>`**' si intende aprire il file in scrittura in aggiunta (*append*).

A questa simbologia si può aggiungere il segno '**`+`**' in modo da permettere anche l'altro tipo di accesso non dichiarato, per cui:

- '**`+<`**'

  rappresenta un accesso in lettura e scrittura;

- ‘**+>**’

  rappresenta un accesso in scrittura e lettura, ma la prima azione è quella di troncare il file annullando il suo contenuto precedente;

- ‘**+>>**’

  rappresenta un accesso in aggiunta e lettura.

In generale, un file aperto in lettura e scrittura attraverso il simbolo ‘**+<**’ permette anche l’allungamento del file stesso. Il pezzo di codice seguente mostra l’apertura di un file in aggiunta e l’inserimento al suo interno di una riga contenente una frase di saluto.

```
open MIO_FILE, ">> /home/tizio/mio_file";
...
print MIO_FILE ("ciao a tutti\n");
```

Nello stesso modo in cui si possono gestire i file su disco, si può accedere a una pipeline, cioè una sequenza di programmi che ricevono dati dal loro standard input e ne emettono attraverso lo standard output. Per ottenere questo, al posto di indicare un file su disco si mette una riga di comando che si vuole sia eseguita, preceduta o terminata con la consueta barra verticale: se si trova all’inizio, significa che si vuole scrivere inviando dati attraverso lo standard input della pipeline; se si trova alla fine, significa che si vuole leggere attingendo dati dallo standard output della pipeline.

```
open MIAPIPE, "| sort > /home/tizio/mio_file";
```

L’esempio appena mostrato apre una pipeline in scrittura. Ciò che verrà ricevuto dalla pipeline sarà ordinato e registrato nel file ‘`/home/tizio/mio_file`’.

```
open MIAPIPE, "ls -l |";
```

L’esempio precedente apre una pipeline in lettura in modo da poter elaborare il risultato del comando ‘**`ls -l`**’.

297.1.2 Chiusura

Un file aperto che non serve più deve essere chiuso. Ciò si ottiene attraverso la funzione ‘**`close()`**’ indicando semplicemente il flusso di file da chiudere.

```
close flusso
```

L’apertura di un file può essere fatta anche se questo risulta già aperto, per cui non è strettamente necessario chiudere un file prima di riaprirlo.

297.2 Condivisione

In presenza di un sistema operativo in multiprogrammazione, tanto più se anche multiutente, si pone il problema della gestione degli accessi simultanei ai file. In pratica occorre gestire un sistema di blocchi, o di semafori, che impediscano le operazioni di scrittura simultanea da parte di processi indipendenti.

Infatti, la lettura simultanea di un file da parte di più programmi non ha alcun effetto collaterale, mentre la modifica simultanea può tradursi anche in un danneggiamento dei dati. Per questo, quando un file deve essere modificato, è importante che venga impedito ad altri programmi di fare altrettanto, almeno per il tempo necessario a concludere l’operazione.

297.2.1 Blocco dei file

Il modo più semplice per impedire che un file possa essere modificato da un altro processo, è quello di bloccarlo (*lock*), per il tempo necessario a compiere le operazioni che si vogliono fare in modo esclusivo.

Teoricamente, il blocco potrebbe limitarsi solo a una porzione del file, ma questo implica un'organizzazione condivisa anche dagli altri processi, in modo che sia ben definita l'estensione di questo blocco. In pratica, ci si limita quasi sempre a eseguire un blocco totale del file, rilasciando il blocco subito dopo la modifica che si vuole effettuare.

Il blocco e lo sblocco del file si ottiene generalmente con la funzione '**`flock()`**' su un file già aperto. La funzione richiede l'indicazione del flusso di file e del tipo di operazione che si vuole compiere.

`flock` *flusso* , *operazione*

Per la precisione, il tipo di operazione si esprime attraverso un numero il cui valore dipende dal sistema operativo utilizzato effettivamente. Per evitare di doversi accertare di quale valore sia corretto per il proprio sistema, è possibile acquisire alcune macro attraverso l'istruzione seguente:

```
use Fcntl ':flock';
```

In questo modo, l'operazione può poi essere indicata attraverso i nomi: '**`LOCK_SH`**', '**`LOCK_EX`**', '**`LOCK_NB`**' e '**`LOCK_UN`**'.

Il blocco del file può essere richiesto in modo da mettere in pausa il programma fino a quando si riesce a ottenere il blocco, oppure no. Nel secondo caso, il programma deve essere in grado di riconoscere il fallimento dell'operazione e di comportarsi di conseguenza. Il blocco con attesa deve essere utilizzato con prudenza, perché può generare una situazione di stallo generale: il processo A apre e blocca il file X, il processo B apre e blocca il file Y e successivamente tenta anche con il file X che però è occupato; a questo punto anche il processo A tenta di aprire il file Y senza avere rilasciato il file X; infine i due processi si sono bloccati a vicenda.

Il blocco esclusivo di un file si ottiene con il tipo di operazione '**`LOCK_EX`**'; se si vuole evitare l'attesa dello sblocco da parte di un altro processo si deve aggiungere il valore di '**`LOCK_NB`**'. Lo sblocco di un file si ottiene con il tipo di operazione '**`LOCK_UN`**'.

Esempi

```
use Fcntl ':flock';   # importa le costanti LOCK_...
...
open (ELENCO, ">> /home/tizio/mioelenco");
flock (ELENCO, LOCK_EX);
...
flock (ELENCO, LOCK_UN);
```

Vengono eseguite le operazioni seguenti:

- si caricano le costanti di definizione dei tipi di blocco attraverso l'istruzione '**`use Fcntl ':flock';`**';
- si apre il file '`/home/tizio/mioelenco`' in aggiunta;
- si blocca il file in modo esclusivo;
- si compiono alcune operazioni che non sono indicate;
- si rilascia il blocco.

```
use Fcntl ':flock';   # importa le costanti LOCK_...
...
open (ELENCO, ">> /home/tizio/mioelenco");
```

```
if (flock (ELENCO, (LOCK_EX)+(LOCK_NB)))
  {
    ...
    flock (ELENCO, LOCK_UN);
  }
else
  {
    print STDOUT "Il file è impegnato.\n";
  }
```

Si tratta di una variante dell'esempio precedente, in cui si richiede un blocco esclusivo senza attesa. Se il blocco ha successo, si procede, altrimenti viene segnalata la presenza del blocco da parte di un altro processo.[1]

297.3 I/O con i file

Le operazioni di I/O con i file richiedono la conoscenza del modo in cui si esegue la lettura, la scrittura e lo spostamento, del puntatore interno a un flusso di file. Fortunatamente, Perl gestisce tutto in modo piuttosto trasparente, soprattutto per ciò che riguarda la lettura. È il caso di ricordare che queste operazioni si compiono su file già aperti, di conseguenza si fa riferimento a loro tramite il flusso corrispondente.

297.3.1 Lettura

La lettura di un flusso di file riferito a un file di testo è un'operazione molto semplice, basta utilizzare le parentesi angolari per ottenere la valutazione dello stesso che si traduce nella restituzione di una riga, nel caso di contesto scalare, o di tutto il file, nel caso di un contesto lista. Per esempio:

```
$riga = <MIOHANDLE>;
```

restituisce una riga, a partire dalla posizione del puntatore del file fino al codice di interruzione di riga incluso, spostando in avanti il puntatore del file. Per questo, dopo un'operazione di questo tipo, si esegue un '**`chop()`**' o un '**`chomp()`**', in modo da eliminare il codice di interruzione di riga finale.

```
chomp $riga;
```

In alternativa,

```
@file = <MIOHANDLE>;
```

restituisce tutto il file suddiviso in righe terminanti con il codice di interruzione di riga. In pratica, l'array conterrà tanti elementi quante sono le righe del file. Anche in questo caso si può eseguire un '**`chop()`**' o un '**`chomp()`**', che interverrà su ogni elemento dell'array.

```
chomp (@file);
```

La valutazione di un flusso di file in questo modo, quando il puntatore del file ha superato la fine del file, restituisce un valore indefinito che può essere utilizzato per controllare un ciclo di lettura. L'esempio seguente mostra in modo molto semplice come un ciclo '**`while`**' possa controllare la lettura di un flusso di file terminando quando questo ha raggiunto la conclusione.

[1] Per qualche motivo oscuro, se si vuole sommare il valore della macro '**`LOCK_EX`**' assieme a quello di qualche altra, è necessario racchiuderla tra parentesi, come si vede nell'esempio. Probabilmente questo dipende dal modo in cui il valore viene generato. Per uniformità, nell'esempio si mostra racchiusa tra parentesi anche la macro '**`LOCK_NB`**'. Volendo verificare questa anomalia, basta provare ad assegnare a una variabile la somma di queste o di altre macro, visualizzando poi il risultato; se si prova una cosa del tipo '**`$pippo = LOCK_EX+LOCK_NB;`**', senza parentesi, e poi si visualizza il contenuto di '**`$pippo`**', si ottiene solo il valore due, mentre dovrebbe essere un sei!

```
while ($riga = <MIOHANDLE>)
  {
    ...
  }
```

297.3.2 Scrittura

La scrittura di un file avviene generalmente attraverso la funzione '**`print()`**' che inizia a scrivere a partire dalla posizione attuale del puntatore del file stesso.

`print` *flusso lista*
`print` *lista*

Se non viene specificato un flusso di file, tutto viene emesso attraverso lo standard output, oppure attraverso quanto specificato con la funzione '**`select()`**'.

È il caso di osservare che l'argomento che specifica il flusso è separato dalla lista di stringhe da emettere solo attraverso uno o più spazi (non si usa la virgola). Per lo stesso motivo, se il flusso di file è contenuto in un elemento di un array, oppure è il risultato di un'espressione, ciò deve essere indicato in un blocco.

Esempi

```
print MIOHANDLE "Ciao, come stai?\n";
```

Scrive nel flusso di file indicato, a partire dalla posizione attuale del puntatore, il messaggio indicato come argomento.

```
print {$elenco_file[$i]} "Bla bla bla\n";
```

Inserisce il messaggio nel file indicato da '**`$elenco_file[$i]`**'.

```
use Fcntl ':flock';   # importa le costanti LOCK_...
...
open (ELENCO, ">> /home/tizio/mioelenco");
flock (ELENCO, LOCK_EX);
print ELENCO $daelencare,"\n";
flock (ELENCO, LOCK_UN);
```

Vengono eseguite le seguenti operazioni:

- si caricano le costanti di definizione dei tipi di blocco attraverso l'istruzione '**`use Fcntl ':flock';`**';
- si apre il file '`/home/tizio/mioelenco`' in aggiunta;
- si blocca il file in modo esclusivo;
- si inserisce una riga nel file;
- si rilascia il blocco.

297.3.3 Spostamento del puntatore

Lo spostamento del puntatore interno a un flusso di file avviene generalmente in modo automatico, sia in lettura che in scrittura. Si possono porre dei problemi, o dei dubbi, quando si accede simultaneamente a un file sia in lettura che in scrittura. Lo spostamento del puntatore può essere fatto attraverso la funzione '**`seek()`**'.

`seek` *flusso*`,` *posizione*`,` *partenza*

La posizione effettiva nel file dipende dal valore del secondo e del terzo argomento. Precisamente, il terzo argomento può essere zero, uno o due, in base al significato seguente:

- 0 -- la nuova posizione corrisponde esattamente a quanto indicato dal secondo argomento;
- 1 -- la nuova posizione corrisponde alla posizione corrente più quanto indicato nel secondo argomento;
- 2 -- la nuova posizione corrisponde alla posizione successiva alla fine del file più il valore del secondo argomento (solitamente negativo).

Esempi

```
seek (MIO_FILE, 0, 2);
```

Posiziona alla fine del file in modo da poter, successivamente, aggiungere qualcosa a questo.

```
seek (MIO_FILE, 0, 0);
```

Posiziona all'inizio del file.

```
use Fcntl ':flock';   # importa le costanti LOCK_...
...
open (ELENCO, ">> /home/tizio/mioelenco");
flock (ELENCO, LOCK_EX);
seek (ORDINI, 0, 2);
print ELENCO $daelencare,"\n";
flock (ELENCO, LOCK_UN);
```

Vengono eseguite le seguenti operazioni:

- si caricano le costanti di definizione dei tipi di blocco attraverso l'istruzione '**`use Fcntl ':flock';`**';
- si apre il file '`/home/tizio/mioelenco`' in aggiunta;
- si blocca il file in modo esclusivo;
- per sicurezza si posiziona il puntatore alla fine del file;
- si inserisce una riga nel file;
- si rilascia il blocco.

297.3.4 Identificazione dei flussi di file

Nel momento in cui si apre un file, si deve attribuire il nome del flusso relativo. Fino a questo punto è stato visto l'uso di nomi dichiarati nell'istante dell'apertura, come nell'esempio seguente:

```
open (MIO_FLUSSO, "< pippo.txt");
```

Da quel punto, il simbolo '**`MIO_FLUSSO`**' diviene ciò che identifica il flusso. È già stato mostrato anche il modo in cui è possibile trasferire il riferimento a questi simboli, come nell'esempio seguente:

```
$mio_flusso = \*MIO_FLUSSO;
```

Successivamente è possibile fare riferimento in modo indifferente al simbolo originale o alla variabile che vi punta:

```
$riga = <$mio_flusso>;
```

In realtà, il simbolo che rappresenta un flusso, può anche essere una variable, contenente una stringa qualunque: ciò che conta diviene il contenuto della variabile per identificare effettivamente il flusso. Si osservi l'esempio seguente:

```
#!/usr/bin/perl
$a = "tizio";
open ($a, "< prova_1");
$a = "caio";
```

```
open ($a, "> prova_2");
$a = "tizio";
$riga = <$a>;
print STDOUT "$riga";
$riga = <"tizio">;
print STDOUT "$riga";
$a = "caio";
print $a "ciao\n";
print caio "come stai\n";
$a = "tizio";
close ($a);
$a = "caio";
close ($a);
```

Si vede la variabile '**$a**' che inizialmente riceve la stringa '**tizio**' e in questa situazione viene usata per aprire in lettura il file '`prova_1`'. Subito dopo, la stessa variable riceve la stringa '**caio**' e in questo modo viene usata per aprire in scrittura il file '`prova_2`'. I due flussi sono identificati rispettivamente dalle stringhe '**tizio**' e '**caio**'; non ha importanza se queste stringhe sono contenute in una variabile o se sono usate direttamente come sono.

Più avanti, si può vedere che, quando '**$a**' contiene la stringa '**tizio**', scrivere

```
$riga = <$a>;
```

oppure

```
$riga = <"tizio">;
```

dà lo stesso risultato: la lettura del flusso abbinato al file '`prova_1`'. Nello stesso modo si può fare per il flusso in scrittura, con la differenza che il non si può usare la stringa in modo delimitato:

```
$a = "caio";
print $a "ciao\n";
print caio "come stai\n";
```

Questa possibilità di gestire i flussi identificandoli subito attraverso delle variabili, facilita il trasferimento dell'indicazione dei flussi nelle chiamate di funzione, senza più il bisogno di creare dei riferimenti.

Si noti che non basta dichiarare un flusso indicando semplicemente una variabile, perché questa variabile deve essere inizializzata in qualche modo. Utilizzando una variabile non inizializzata sarebbe come volere identificare il flusso con la stringa nulla.

Perl: funzioni interne

Nelle sezioni seguenti viene descritto brevemente il funzionamento di alcune funzioni interne di Perl. La sintassi viene mostrata secondo lo stile della documentazione di Perl, per cui, '*blocco*' rappresenta un gruppo di istruzioni nella forma consueta di Perl, e '*lista*' rappresenta un elenco di espressioni separate da virgole.

'*blocco*' equivale a:

```
{ istruzione... }
```

'*lista*' equivale a:

```
espressione1, espressione2, ...
```

Le funzioni descritte sono raggruppate in base al tipo di situazione in cui vengono utilizzate normalmente.

298.1 File

Nelle sezioni seguenti vengono elencate alcune funzioni che riguardano la gestione dei file, nel senso globale, esterno. Le funzioni per la gestione del contenuto dei file vengono descritte più avanti.

Tabella 298.1. Elenco di alcune funzioni riferite alle operazioni sui file.

Nome	Descrizione.
-*x*	Test sui file.
chmod()	Cambia i permessi.
chown()	Cambia l'utente e il gruppo proprietari.
link()	Crea un collegamento fisico.
lstat()	Restituisce le informazioni sui collegamenti simbolici.
readlink()	Restituisce il valore di un collegamento simbolico.
rename()	Cambia il nome di un file.
stat()	Restituisce le informazioni su un file.
symlink()	Crea un collegamento simbolico.
unlink()	Cancella i file.
utime()	Modifica la data di accesso e di modifica dei file.

298.1.1 Test sui file

Perl permette di effettuare una serie di test sui file in modo analogo a quanto si fa con le shell tradizionali. La sintassi è esprimibile nei due modi seguenti.

```
-x nome_file
-x flusso
```

Nel primo caso si fa riferimento a un file indicato per nome, nel secondo il riferimento è a un flusso di file. La lettera *x* cambia a seconda del tipo di test da verificare. La tabella 298.2 mostra l'elenco di questi test.

Tabella 298.2. Elenco dei test '`-x`'.

Test	Significato.
`-r`	Il file è accessibile in lettura dal numero UID/GID efficace.
`-w`	Il file è accessibile in scrittura dal numero UID/GID efficace.
`-x`	Il file è accessibile in esecuzione dal numero UID/GID efficace.
`-o`	Il file appartiene al numero UID efficace.
`-R`	Il file è accessibile in lettura dal numero UID/GID reale.
`-W`	Il file è accessibile in scrittura dal numero UID/GID reale.
`-X`	Il file è accessibile in esecuzione dal numero UID/GID reale.
`-O`	Il file appartiene al numero UID reale.
`-e`	Il file esiste.
`-z`	Il file ha dimensione zero.
`-s`	Il file ha una dimensione maggiore di zero (restituisce la dimensione).
`-f`	Si tratta di un file normale.
`-d`	Si tratta di una directory.
`-l`	Si tratta di un collegamento simbolico.
`-p`	Si tratta di una pipe con nome.
`-S`	Si tratta di un socket.
`-b`	Si tratta di file di dispositivo a blocchi.
`-c`	Si tratta di file di dispositivo a caratteri.
`-t`	Si tratta di un flusso di file aperto su un terminale.
`-u`	Il file ha il bit SUID attivo.
`-g`	Il file ha il bit SGID attivo.
`-k`	Il file ha il bit Sticky attivo.
`-T`	Si tratta di un file di testo.
`-B`	Si tratta di un file binario.
`-M`	Restituisce quanto tempo ha il file in base alla data di modifica.
`-A`	Restituisce quanto tempo ha il file in base alla data di accesso.
`-C`	Restituisce quanto tempo ha il file in base alla data di creazione.

I vari test restituiscono il valore uno se si verificano, oppure la stringa nulla in caso contrario. A questo ci sono delle eccezioni che sono indicate nella tabella.

Esempi

```
if (-x "esempio.pl")
  {
    print "Il file è eseguibile\n";
  }
```

Restituisce il messaggio se il file '`esempio.pl`' è eseguibile.

298.1.2 chmod()

`chmod` *permessi*`,` *file*`,` ...

'**`chmod()`**' cambia i permessi dei file indicati come argomento. In particolare, l'argomento è una lista, in cui il primo elemento è costituito dai permessi espressi in forma numerica ottale. Dal momento che si tratta di un numero ottale, è bene che non sia fornito in forma di stringa perché la conversione da stringa a numero ottale non è automatica. Restituisce il numero di file su cui ha potuto intervenire con successo.

Esempi

```
chmod 0755, 'mio_file', 'tuo_file', 'suo_file';
```

Cambia i permessi ai file indicati dopo la modalità.

```
@elenco = ('mio_file', 'tuo_file', 'suo_file');
chmod 0755, @elenco;
```

Esattamente come nell'esempio precedente.

```
@elenco = ('mio_file', 'tuo_file', 'suo_file');
chmod (0755, @elenco);
```

Esattamente come nell'esempio precedente, ma più simile alle chiamate di funzione degli altri linguaggi.

298.1.3 chown()

```
chown uid, gid, file, ...
```

'**chown()**' cambia i permessi dei file indicati nella lista di argomenti. I primi due elementi della lista sono rispettivamente il numero UID e GID. Gli elementi restanti sono i file su cui si vuole intervenire. Restituisce il numero di file su cui ha potuto intervenire con successo.

Esempi

```
chown 1001, 100, 'mio_file', 'tuo_file', 'suo_file';
```

Cambia l'utente e il gruppo proprietari dei file '`mio_file`', '`tuo_file`' e '`suo_file`'.

```
chown (1001, 100, 'mio_file', 'tuo_file', 'suo_file');
```

Esattamente come nell'esempio precedente.

298.1.4 link()

```
link file_di_origine , collegamento_di_destinazione
```

'**link()**' genera un collegamento fisico a partire da un file esistente. Restituisce *Vero* se la creazione ha successo.

298.1.5 lstat()

```
lstat file
```

```
lstat flusso
```

'**lstat()**' funziona esattamente come '**stat()**', con la differenza che restituisce le informazioni relative a un collegamento simbolico, invece di quelle del file a cui questo punta. Se non viene indicato l'argomento, '**lstat()**' utilizza il contenuto della variabile predefinite '**$_**'.

298.1.6 readlink()

```
readlink file
```

'**readlink()**' restituisce il valore di un collegamento simbolico. Se non viene indicato l'argomento, '**readlink()**' utilizza il contenuto della variabile predefinita '**$_**'.

Esempi

```
$prova = readlink '/bin/sh';
```

Assegna alla variabile '**$prova**' il percorso contenuto nel collegamento simbolico '`/bin/sh`'. Probabilmente, alla fine, la variabile conterrà la stringa '**bash**'.

298.1.7 rename()

```
rename nome_vecchio , nome_nuovo
```

‘**rename()**’ cambia il nome di un file, o lo sposta. Tuttavia, lo spostamento non può avvenire al di fuori del file system di partenza. Restituisce uno se l’operazione riesce, altrimenti zero.

298.1.8 stat()

```
stat file
stat flusso
```

‘**stat()**’ restituisce un array di tredici elementi contenenti tutte le informazioni sul file indicato per nome o attraverso un flusso di file. Se non viene indicato l’argomento, ‘**stat()**’ utilizza il contenuto della variabile predefinita ‘**$**_’.

Gli elementi dell’array restituito sono riportati nella tabella 298.3 in cui appare anche il nome suggerito per la trasformazione in variabili scalari.

Tabella 298.3. Elenco degli elementi componenti l’array restituito da ‘**stat()**’.

Elemento	Nome consueto	Descrizione.
0	$dev	Numero del dispositivo del file system.
1	$ino	Numero dell’inode.
2	$mode	Permessi del file.
3	$nlink	Numero di collegamenti fisici al file.
4	$uid	UID dell’utente proprietario del file.
5	$gid	GID del gruppo proprietario del file.
6	$rdev	Identificatore di dispositivo, per i file speciali.
7	$size	Dimensione in byte.
8	$atime	Data dell’ultimo accesso.
9	$mtime	Data dell’ultima modifica.
10	$ctime	Data di cambiamento di inode.
11	$blksize	Dimensione preferita dei blocchi per le operazioni di I/O del sistema.
12	$blocks	Numero di blocchi allocati attualmente.

Va osservato che le informazioni data-orario sui file sono espresse in forma numerica che esprime il tempo trascorso a partire dalla data di riferimento del sistema operativo. Nel caso dei sistemi derivati da Unix si tratta dell’ora zero del 1/1/1970. Nello stesso modo, è evidente che tutte queste informazioni possono essere ottenute solo da un file system che può gestirle.

Esempi

```
($dev, $ino, $mode, $nlink,
 $uid, $gid, $rdev, $size,
 $atime, $mtime, $ctime,
 $blksize, $blocks) = stat (’/home/tizio/mio_file’);
```

Preleva tutte le informazioni sul file ‘`/home/tizio/mio_file`’ e le scompone in diverse variabili scalari.

298.1.9 symlink()

`symlink` *file_di_origine* , *collegamento_di_destinazione*

‘**`symlink()`**’ genera un collegamento simbolico a partire da un file esistente. Restituisce *Vero* se la creazione ha successo.

298.1.10 unlink()

`unlink` *lista_di_file*

‘**`unlink()`**’ cancella i file indicati per nome tra gli argomenti. Generalmente non possono essere cancellate le directory (e comunque sarebbe inopportuno dato il tipo di cancellazione che si fa). Restituisce il numero di file cancellati con successo. Se non viene indicato l’argomento, ‘**`unlink()`**’ utilizza il contenuto della variabile predefinita ‘**`$_`**’.

298.1.11 utime()

`utime` *data_di_accesso* , *data_di_modifica* , *lista_di_file*

‘**`utime()`**’ cambia la data di modifica e di accesso di una serie di file. Le date, indicate come argomenti iniziali, sono espresse nella forma numerica gestita dal sistema operativo. La data di modifica dell’inode viene cambiata automaticamente in modo che corrisponda al momento in cui questa modifica viene effettuata.

Esempi

```
$momento = time;
utime $momento, $momento, 'mio_file';
```

Cambia la data di accesso e modifica in modo da farle coincidere con quella riportata dall’orologio dell’elaboratore nel momento in cui si eseguono queste istruzioni.

298.2 Directory

Nelle sezioni seguenti vengono elencate alcune funzioni che riguardano la gestione delle directory e di raggruppamenti di file. Vengono ignorate volutamente le funzioni specifiche di Perl per la lettura delle directory.

Tabella 298.4. Elenco di alcune funzioni riferite alle operazioni sulle directory.

Nome	Descrizione.
chdir()	Cambia la directory di lavoro.
glob()	Espande un modello fatto attraverso l’uso di caratteri jolly.
mkdir()	Crea una directory.
rmdir()	Cancella una directory vuota.

298.2.1 chdir()

```
chdir directory
```

'**chdir()**' cambia la directory di lavoro posizionandosi in corrispondenza di quanto indicato come argomento. Se l'argomento viene omesso, lo spostamento avviene nella directory personale, attraverso quanto determinato dal contenuto di '**$ENV{"HOME"}**'. Restituisce *Vero* se l'operazione ha successo, *Falso* in tutti gli altri casi.

298.2.2 glob()

```
glob espressione
```

'**glob()**' restituisce quanto indicato nell'argomento dopo un'operazione di espansione, come farebbe una shell. Se l'argomento non viene indicato, l'espansione viene effettuata sul contenuto della variabile '**$_**'.

Esempi

```
$primo = glob ("/bin/*");
```

Assegna alla variabile '**$primo**' il percorso assoluto del primo file che viene trovato attraverso l'espansione del modello '`/bin/*`'.

```
@elenco = glob ("/bin/*");
```

Assegna all'array '**@elenco**' i percorsi assoluti dei file che vengono trovati attraverso l'espansione del modello '`/bin/*`'.

298.2.3 mkdir()

```
mkdir directory, permessi
```

'**mkdir()**' crea la directory indicata come primo argomento. I permessi della directory sono indicati come secondo argomento, devono essere espressi con un numero ottale e risulteranno filtrati ulteriormente dalla maschera dei permessi. Restituisce uno se l'operazione riesce, altrimenti zero, impostando anche la variabile '**$!**'.

> In generale, non dovrebbe essere possibile assegnare dei permessi negli S-bit. In pratica dovrebbe essere consentito di operare solo con i soliti permessi di lettura, scrittura ed esecuzione (attraversamento).

Esempi

```
mkdir ("/tmp/prova");
```

Crea la directory '`/tmp/prova/`' con i permessi normali dell'utente.[1]

```
mkdir ("/tmp/prova", 0755);
```

Crea la directory '`/tmp/prova/`' con i permessi 0755_8 (si osservi che si tratta di un numero ottale), che vengono comunque filtrati dalla maschera dei permessi.

[1] Anche se la documentazione fa esplicito riferimento a questa possibilità, può darsi che non sia possibile evitare di indicare i permessi. Nello stesso modo, anche se si indicano i permessi non è garantito che questi vengano rispettati fedelmente dal sistema operativo sottostante, come descritto nell'esempio successivo.

298.2.4 rmdir()

```
rmdir directory
```

‘**rmdir()**’ elimina la directory indicata come argomento. Se l’argomento non viene fornito, si utilizza la variabile predefinita ‘**$_**’. Restituisce uno se l’operazione riesce, altrimenti zero, impostando anche la variabile ‘**$!**’.

298.3 I/O

Nelle sezioni seguenti vengono elencate alcune funzioni che riguardano la gestione dei dati contenuti nei file.

Tabella 298.5. Elenco di alcune funzioni riferite alle operazioni di I/O.

Nome	Descrizione.
binmode()	Attiva la modalità binaria di lettura e scrittura.
chomp()	Elimina il codice di interruzione di riga finale.
chop()	Elimina l’ultimo carattere di una stringa.
close()	Chiude un flusso di file.
eof()	Verifica la conclusione del file.
fcntl()	Esegue la chiamata della funzione di sistema omonima.
fileno()	Restituisce il descrittore di un file aperto.
flock()	Esegue la chiamata di sistema omonima.
getc()	Legge un carattere alla volta.
ioctl()	Esegue la chiamata di sistema omonima.
open()	Apre un file e gli associa un flusso di file.
pipe()	Esegue la chiamata di sistema omonima.
print()	Scrive all’interno di un flusso di file.
printf()	Scrive all’interno di un flusso di file utilizzando ‘**sprintf()**’.
read()	Legge un file.
seek()	Sposta il puntatore interno a un file aperto.
select()	Definisce il flusso di file attuale.
sprintf()	Restituisce una stringa formattata.
tell()	Restituisce la posizione del puntatore interno di un flusso di file.

298.3.1 binmode()

```
binmode flusso
```

‘**binmode()**’ attiva la modalità binaria per il file corrispondente al flusso di file indicato come argomento. Generalmente, non è necessario utilizzare questa istruzione con GNU/Linux, mentre potrebbe essere necessario in altri ambienti. Si può dire che questa istruzione serva solo quando il sistema operativo sottostante utilizza un codice di interruzione di riga diverso dal semplice <*LF*>.

298.3.2 chomp()

```
chomp espressione_stringa
```

```
chomp lista
```

‘**chomp()**’ riceve come argomento un’espressione che restituisce una stringa o una lista di stringhe. Il suo scopo è eliminare dalla parte finale il codice di interruzione di riga, che coincide normalmente con il carattere <*LF*>. Precisamente si tratta di quanto contenuto nella variabile

predefinita '**$/**'. Se non viene indicato l'argomento, interviene sul contenuto della variabile '**$_**'. Restituisce il numero di caratteri eliminati.

Esempi

```
#!/usr/bin/perl
$/ = "\r\n";
while ($riga = <STDIN>)
  {
    chomp ($riga);
    print STDOUT ("$riga\n");
  }
```

Quello che si vede è un esempio molto semplice di un filtro che trasforma un file di testo in stile Dos a uno in stile Unix. In pratica, viene definito che l'interruzione di riga è indicata attraverso la sequenza dei caratteri *<CR><LF>* ('**\r\n**'), che attraverso la funzione '**chomp()**' viene eliminata dalle righe lette. Infine, le righe vengono emesse attraverso lo standard output, con l'aggiunta del codice *<LF>* finale.

298.3.3 chop()

chop *espressione_stringa*
chop *lista*

'**chop()**' riceve come argomento un'espressione che restituisce una stringa o una lista di stringhe. Il suo scopo è eliminare l'ultimo carattere della stringa, o delle stringhe della lista. In questo senso differisce da '**chomp()**' che invece elimina la parte finale solo se necessario. Restituisce l'ultimo carattere eliminato.

298.3.4 close()

close *flusso*

'**close()**' chiude un flusso di file aperto precedentemente. Restituisce *Vero* se l'operazione ha successo e non si sono prodotti errori di alcun tipo. È opportuno osservare che non è necessario chiudere un file se poi si deve riaprire immediatamente con la funzione '**open()**': lo si può semplicemente riaprire.

Esempi

```
close (MIO_FILE);
```

Chiude il flusso di file '**MIO_FILE**'.

298.3.5 eof()

eof *flusso*

'**eof()**' verifica se la prossima lettura del flusso di file supera la fine del file. Restituisce uno se ciò si verifica. Questa funzione è generalmente di scarsa utilità dal momento che la lettura di una riga oltre la fine del file genera un risultato indefinito che può essere verificato tranquillamente in un'espressione condizionale. Oltre a ciò, '**eof()**' si verifica prima che il tentativo di lettura sia stato fatto veramente, contrariamente a quanto avviene di solito in altri linguaggi di programmazione.

298.3.6 fcntl()

```
fcntl flusso , funzione , scalare
```

‘**fcntl()**’ esegue la chiamata di sistema omonima e per questo può essere utilizzata solo con un sistema operativo che la gestisce. Prima di poter utilizzare questa funzione occorre richiamare una serie di valori corrispondenti a macro del proprio sistema:

```
use Fcntl;
```

298.3.7 fileno()

```
fileno flusso
```

‘**fileno()**’ restituisce il descrittore corrispondente a un flusso di file.

298.3.8 flock()

```
flock flusso , operazione
```

‘**flock()**’ esegue la chiamata di sistema omonima, oppure una sua emulazione, per il file identificato tramite il flusso di file. ‘**flock()**’ permette di eseguire il blocco di un file nel suo complesso e non record per record. Restituisce *Vero* se l’operazione ha successo.

L’operazione, cioè il tipo di blocco, viene indicata attraverso una sorta di macro che viene inserita nel sorgente di Perl attraverso la dichiarazione seguente:

```
use Fcntl ':flock';
```

Operazioni

```
LOCK_SH
```

Corrisponde normalmente al valore numerico uno. Richiede un blocco condiviso (*shared*).

```
LOCK_EX
```

Corrisponde normalmente al valore numerico due. Richiede un blocco esclusivo.

```
LOCK_UN
```

Corrisponde normalmente al valore numerico otto. Rilascia il blocco.

```
LOCK_NB
```

Corrisponde normalmente al valore numerico quattro. Viene sommato a ‘**LOCK_SH**’ o a ‘**LOCK_EX**’ in modo da non attendere lo sblocco del file nel caso che questo risulti già bloccato.

Esempi

```
use Fcntl ':flock';   # importa le costanti LOCK_...
...
open (ELENCO, ">> /home/tizio/mioelenco");
flock (ELENCO, LOCK_EX);
seek (ELENCO, 0, 2);
print ELENCO $daelencare,"\n";
flock (ELENCO, LOCK_UN);
```

Vengono eseguite le operazioni seguenti:

- si caricano le costanti di definizione dei tipi di blocco attraverso l’istruzione ‘**use Fcntl ':flock';**’;

- si apre il file '`/home/tizio/mioelenco`' in aggiunta;
- si blocca il file in modo esclusivo;
- per sicurezza si posiziona il puntatore del file alla fine dello stesso;
- si inserisce una riga nel file;
- si rilascia il blocco.

```
use Fcntl ':flock';   # importa le costanti LOCK_...
...
open (ELENCO, ">> /home/tizio/mioelenco");
if (flock (ELENCO, (LOCK_EX)+(LOCK_NB)))
  {
    seek (ELENCO, 0, 2);
    print ELENCO $daelencare,"\n";
    flock (ELENCO, LOCK_UN);
  }
else
  {
    print STDOUT "Il file è impegnato.\n";
  }
```

Si tratta di una variante dell'esempio precedente in cui si richiede un blocco esclusivo senza attesa. Se il blocco ha successo, si procede, altrimenti viene segnalata la presenza del blocco eseguito da un altro processo (si osservi il fatto che le macro sono state racchiuse tra parentesi tonde prima di sommarle assieme).

298.3.9 getc()

`getc` *flusso*

'**`getc()`**' legge il file indicato dal flusso di file, o dallo standard input se viene omesso l'argomento, restituendo il prossimo carattere. Se si supera la fine del file restituisce la stringa nulla.

298.3.10 ioctl()

`ioctl` *flusso* , *funzione* , *scalare*

'**`ioctl()`**' esegue la chiamata di sistema omonima e per questo può essere utilizzata solo con un sistema operativo che la gestisce. Per poterla utilizzare occorre consultare la documentazione interna di Perl.

298.3.11 open()

`open` *flusso* , *file*

'**`open()`**' apre il file indicato come secondo argomento utilizzando il flusso di file indicato come primo argomento. Il nome del file è composto normalmente da un prefisso simbolico che ne rappresenta la modalità di utilizzo:

- '**`<`**'

  questo simbolo o l'assenza di ogni altro prefisso rappresenta l'apertura del file in lettura, o in input;

- '**`>`**'

  il file viene troncato (viene ridotto a un file vuoto) e aperto in scrittura, o in output;

- ‘**>>**’

  il file viene aperto in scrittura in aggiunta;

- ‘**+<**’

  il file viene aperto in lettura e scrittura, senza il troncamento iniziale;

- ‘**+>**’

  il file viene aperto in scrittura e lettura, a cominciare dal troncamento iniziale;

- ‘**|**’

  il file viene interpretato come un comando a cui inviare i dati in scrittura attraverso una pipeline. Eccezionalmente, questo simbolo può apparire anche, o soltanto, alla fine di un comando del quale si vuole leggere lo standard output.

Il prefisso può essere staccato dal nome del file attraverso spazi. L’apertura del file rappresentato da un trattino (‘**-**’) è equivalente all’apertura dello standard input, mentre l’apertura del file ‘**>-**’ è equivalente all’apertura dello standard output.

Restituisce *Vero* se l’apertura ha successo.

Esempi

```
if (open (ORDINI, ">> /var/log/ordini"))
  {
    if (flock (ORDINI, LOCK_EX))
      {
        seek (ORDINI, 0, 2);
        print ORDINI ("$ordine\n");
      }
    close (ORDINI);
  }
```

Tenta di aprire il file ‘`/var/log/ordini`’ in aggiunta, quindi tenta di bloccarlo in modo esclusivo. Se ci riesce sposta il puntatore alla fine del file, per sicurezza, quindi inserisce un nuovo ordine. Infine chiude il file.

```
if (open (MAN, "man $DATI{sezione} $DATI{man} | col -bx |"))
  {
    print "Content-type: text/html\n";
    print "\n";
    print "<HTML>\n";
    print "<HEAD>\n";
    print "<TITLE>man $DATI{sezione} $DATI{man}</TITLE>\n";
    print "</HEAD>\n";
    print "<BODY>\n";
    print "<H1>man $DATI{sezione} $DATI{man}</H1>\n";
    print "<PRE>\n";

    while ($risposta = <MAN>)
      {
        print $risposta;
      }

    print "</PRE>\n";
    print "</BODY>\n";
    print "</HTML>\n";
}
```

Genera una pagina HTML a partire da un comando ‘**man**’.

298.3.12 pipe()

```
pipe flusso_in_lettura , flusso_in_scrittura
```

‘**pipe()**’ esegue la chiamata di sistema omonima, aprendo due flussi di file, uno in lettura e l’altro in scrittura. Per poterla utilizzare occorre consultare la documentazione interna di Perl.

298.3.13 print()

```
print flusso lista
print lista
```

‘**print()**’ emette attraverso il flusso di file indicato la lista di argomenti successiva. Se non viene specificato un flusso di file, tutto viene emesso attraverso lo standard output, oppure attraverso quanto specificato con la funzione ‘**select()**’. Se non viene specificato alcun argomento, viene emesso il contenuto della variabile ‘**$_**’.

È il caso di osservare che l’argomento che specifica il flusso è separato dalla lista di stringhe da emettere solo attraverso uno o più spazi (non si usa la virgola). Per lo stesso motivo, se il flusso di file è contenuto in un elemento di un array, oppure è il risultato di un’espressione, ciò deve essere indicato in un blocco.

Restituisce *Vero* se l’operazione di scrittura ha successo.

Esempi

```
print "Ciao, come stai?\n";
```

Emette attraverso lo standard output il messaggio indicato come argomento.

```
print STDERR "Errore $errore\n";
```

Emette attraverso lo standard error il messaggio indicato come argomento.

```
print { $elenco_file[$i] } "Bla bla bla\n";
```

Inserisce il messaggio nel flusso di file indicato da ‘**$elenco_file[$i]**’.

```
print { $ok ? STDOUT : STDERR } ("Bla bla bla\n");
```

Emette il messaggio attraverso lo standard output, oppure lo standard error, a seconda del valore contenuto in ‘**$ok**’.

298.3.14 printf()

```
printf flusso formato , lista
printf formato , lista
```

È equivalente all’uso di ‘**sprintf()**’ nel modo seguente:

```
print flusso sprintf formato , lista
```

298.3.15 read()

```
read flusso , scalare , lunghezza , scostamento
read flusso , scalare , lunghezza
```

‘**read()**’ tenta di leggere il flusso di file specificato e di ottenere la quantità di byte espressa nel terzo argomento, inserendo quanto letto nella variabile scalare indicata come secondo. Se

viene indicato anche il quarto argomento, lo scostamento, il contenuto della variabile non viene rimpiazzato completamente, ma è sovrascritto a partire dalla posizione indicata dallo scostamento stesso. La funzione restituisce il numero di byte letti effettivamente, oppure il valore indefinito se si è verificato un errore.

298.3.16 seek()

`seek` *flusso* , *posizione* , *partenza*

'**`seek()`**' modifica la posizione del puntatore riferito al flusso di file. La posizione effettiva nel file dipende dal valore del secondo e del terzo argomento. Precisamente, il terzo argomento può essere zero, uno o due, in base al significato seguente:

- 0 -- la nuova posizione corrisponde esattamente a quanto indicato dal secondo argomento;
- 1 -- la nuova posizione corrisponde alla posizione corrente più quanto indicato nel secondo argomento;
- 2 -- la nuova posizione corrisponde alla posizione successiva alla fine del file più il valore del secondo argomento (solitamente negativo).

Esempi

```
seek (MIO_FILE, 0, 2);
```

Posiziona alla fine del file in modo da poter aggiungere successivamente qualcosa a questo.

```
seek (MIO_FILE, 0, 0);
```

Posiziona all'inizio del file.

298.3.17 select()

`select` *flusso*

'**`select()`**' permette di definire il flusso di file in scrittura predefinito, per tutte quelle situazioni in cui questo concetto ha significato.

Esempi

```
select (MIO_FILE);
...
print ("ciao a tutti\n");
```

Aggiunge al file identificato dal flusso di file '**`MIO_FILE`**' il messaggio '**`ciao a tutti`**'.

298.3.18 sprintf()

`sprintf` *formato* , *lista*

'**`sprintf()`**' restituisce una stringa formattata in modo analogo a quanto fa la funzione omonima del linguaggio C. Il primo argomento è la stringa da formattare, quelli successivi sono i valori da inserire. Perl utilizza una propria gestione della conversione secondo quanto riportato nelle tabelle 298.6 e 298.7.

Tabella 298.6. Elenco dei simboli utilizzabili in una stringa formattata per l'utilizzo con '**sprintf()**'.

Simbolo	Corrispondenza
%%	Segno di percentuale.
%c	Un carattere con il numero dato.
%s	Una stringa.
%d	Un intero con segno a base 10.
%u	Un intero senza segno a base 10.
%o	Un intero senza segno in ottale.
%x	Un intero senza segno in esadecimale.
%e	Un numero a virgola mobile, in notazione scientifica.
%f	Un numero a virgola mobile, in notazione decimale fissa.
%g	Un numero a virgola mobile, secondo la notazione di '**%e**' o '**%f**'.
%X	Come '**%x**', ma con l'uso di lettere maiuscole.
%E	Come '**%e**', ma con l'uso della lettera '**E**' maiuscola.
%G	Come '**%g**', ma con l'uso della lettera '**E**' maiuscola (se applicabile).
%p	Un puntatore (l'indirizzo utilizzato da Perl in esadecimale).
%n	Immagazzina, nella prossima variabile, il numero di caratteri già emessi.
%i	Sinonimo di '**%d**'.
%D	Sinonimo di '**%ld**'.
%U	Sinonimo di '**%lu**'.
%O	Sinonimo di '**%lo**'.
%F	Sinonimo di '**%f**'.

Tabella 298.7. Elenco dei simboli utilizzabili tra il segno di percentuale e la lettera di conversione.

Simbolo	Corrispondenza
spazio	Il prefisso di un numero positivo è uno spazio.
+	Il prefisso di un numero positivo è il segno '**+**'.
-	Allinea a sinistra rispetto al campo.
0	Utilizza zeri, invece di spazi, per allineare a destra.
#	Prefissa un numero ottale con uno zero e un numero esadecimale con 0x...
n	Un numero definisce la dimensione minima del campo.
.*n*	Per i numeri a virgola mobile esprime la precisione, ovvero il numero di decimali.
.*n*	Per le stringhe definisce la lunghezza massima.
.*n*	Per gli interi definisce la lunghezza minima.
l	Interpreta un intero come il tipo C '**long**' o '**unsigned long**'.
h	Interpreta un intero come il tipo C '**short**' o '**unsigned short**'.
V	Interpreta un intero secondo il tipo standard di Perl.

Quando il simbolo è formato da un numero, al posto di tale numero può essere utilizzato l'asterisco ('*****') intendendo in questo modo di utilizzare il valore inserito nell'elemento successivo.

'**sprintf()**' è sensibile all'attivazione della localizzazione, nel qual caso, il carattere utilizzato per separare le cifre intere da quelle decimali, dipende dalla variabile di ambiente '**LC_NUMERIC**'.

298.3.19 tell()

```
tell flusso
```

'**tell()**' restituisce la posizione corrente del puntatore interno riferito al flusso di file indicato come argomento, oppure a quello dell'ultima operazione di lettura eseguita.

298.4 Interazione con il sistema

Nel gruppo di sezioni seguenti vengono descritte alcune funzioni per l'interazione con il sistema.

Tabella 298.8. Elenco di alcune funzioni riferite all'interazione con il sistema.

Nome	Descrizione.
exec()	Esegue il comando senza ritornare.
kill()	Invia un segnale ai processi.
sleep()	Pausa.
system()	Esegue il comando e attende la sua conclusione.
time()	Restituisce la data e l'ora del sistema espressa in secondi.
times()	Restituisce la data e l'ora del sistema in modo dettagliato.
umask()	Definisce la maschera dei permessi.

298.4.1 exec()

```
exec elenco
```

'**exec()**' avvia l'esecuzione del comando indicato negli argomenti, senza riprendere l'esecuzione del programma al termine. Si comporta quindi in modo analogo al comando interno omonimo delle shell comuni.

Esempi

```
...
exec ("ls");
...
```

Esegue il comando '**ls**' e conclude il funzionamento del programma. In pratica, le istruzioni successive a '**exec()**', non vengono eseguite.

298.4.2 kill()

```
kill segnale, elenco_di_processi
```

'**kill()**' invia un segnale a una serie di processi. Il primo argomento deve essere il segnale. Restituisce il numero di processi che hanno ricevuto il segnale.

Esempi

```
kill ("TERM", 588);
```

Invia il segnale '**SIGTERM**' al processo numero 588.

```
kill (15, 588);
```

Esattamente come nell'esempio precedente.

```
kill (-15, 588);
```

Come nell'esempio precedente, ma il segnale viene inviato anche a tutti i processi discendenti da quello indicato.

298.4.3 sleep()

```
sleep secondi
```

'**sleep()**' mette in pausa l'esecuzione del programma, per il numero di secondi indicato come argomento, eventualmente attraverso un'espressione. Se l'argomento non viene indicato, la pausa non ha fine. L'attesa può essere interrotta inviando un segnale '**SIGALRM**' al processo. Restituisce il numero di secondi trascorsi effettivamente.

Esempi

```
sleep;
```

Mette il programma in pausa senza specificare la fine di questa.

```
sleep (10);
```

Mette il programma in pausa per 10 secondi.

```
sleep ($pausa);
```

Mette il programma in pausa per la quantità di secondi indicata dalla variabile '**$pausa**'.

298.4.4 system()

```
system elenco
```

'**system()**' avvia l'esecuzione del comando indicato negli argomenti, attende la sua conclusione e restituisce il valore generato dal comando stesso.

Esempi

```
system ("ls");
```

Esegue il comando '**ls**' e poi riprende con il programma.

```
if (system ("mkdir ciao")
  {
    die("La creazione della directory è fallita\n");
  }
else
  {
    print ("La directory è stata creata\n");
  }
```

L'esempio mostra il caso in cui si voglia controllare l'esito di un comando di sistema avviato attraverso la funzione '**system()**'. Se il comando '**mkdir ciao**' viene eseguito con successo, restituisce il valore zero, che per Perl equivale a *Falso*. Quindi, se la condizione si avvera, significa che l'operazione è fallita, altrimenti, tutto è andato bene.

298.4.5 time()

```
time
```

'**time()**' restituisce la data e l'ora attuale espressa in secondi trascorsi dalla data iniziale gestita dal sistema. Nel caso della maggior parte dei sistemi Unix si tratta dell'ora zero del 1/1/1970. Il valore ottenuto da '**time()**' può essere utilizzato dalle funzioni '**gmtime()**' e '**localtime()**'

298.4.6 times()

```
times
```

‘**times()**’ restituisce un array di quattro elementi che indicano rispettivamente:

1. orario dell’utente;
2. orario di sistema;
3. orario dell’utente del processo figlio;
4. orario di sistema del processo figlio.

Esempi

```
($user, $system, $cuser, $csystem) = times;
```

Scompone l’array restituito da ‘**times()**’ in quattro variabili scalari.

298.4.7 umask()

```
umask maschera_numerica
```

‘**umask()**’ permette di definire la maschera dei permessi per il processo elaborativo del programma. Restituisce il valore precedente.

La maschera è espressa in forma numerica; ciò significa che se la maschera da indicare come argomento è una stringa, potrebbe essere necessario l’utilizzo della funzione ‘**oct()**’ per garantire l’interpretazione ottale e non a base 10.

Esempi

```
$maschera = '644';
umask (oct ($maschera));
```

Modifica la maschera dei permessi in modo che sia pari a 0644_8. Dal momento che l’informazione è contenuta in una stringa, che per di più non ha lo zero iniziale della rappresentazione ottale convenzionale, occorre convertire prima la stringa in numero nel modo corretto.

298.5 Funzioni matematiche

Perl fornisce una serie di funzioni matematiche tipiche della maggior parte dei linguaggi di programmazione.

Tabella 298.9. Elenco di alcune funzioni matematiche.

Nome	Descrizione.
abs()	Calcola il valore assoluto.
atan2()	Calcola l’arcotangente dell’intervallo da -PI a +PI.
cos()	Calcola il coseno.
exp()	Calcola l’esponente.
int()	Estrae la parte intera.
log()	Calcola il logaritmo naturale.
sin()	Calcola il seno.
sqrt()	Calcola la radice quadrata.

298.5.1 abs()

```
abs x
```

‘**abs()**’ restituisce il valore assoluto del suo argomento. Se l’argomento non viene indicato, si utilizza la variabile predefinita ‘**$_**’.

298.5.2 atan2()

```
atan2 x,y
```

‘**atan2()**’ restituisce l’arcotangente nell’intervallo da -PI a +PI.

298.5.3 cos()

```
cos x
```

‘**cos()**’ restituisce il coseno. Se l’argomento non viene indicato, si utilizza la variabile predefinita ‘**$_**’.

298.5.4 exp()

```
exp x
```

‘**exp()**’ restituisce *e* (la base del logaritmo naturale) elevato al valore di *x*, cioè dell’argomento. Se l’argomento non viene indicato, si utilizza la variabile predefinita ‘**$_**’.

298.5.5 int()

```
int x
```

‘**int()**’ restituisce la parte intera del numero (o dell’espressione) fornito come argomento. Se l’argomento non viene indicato, si utilizza la variabile predefinita ‘**$_**’.

298.5.6 log()

```
log x
```

‘**log()**’ restituisce il logaritmo naturale del valore fornito come argomento. Se l’argomento non viene indicato, si utilizza la variabile predefinita ‘**$_**’.

298.5.7 sin()

```
sin x
```

‘**sin()**’ restituisce il seno. Se l’argomento non viene indicato, si utilizza la variabile predefinita ‘**$_**’.

298.5.8 sqrt()

```
sqrt x
```

‘**sqrt()**’ restituisce la radice quadrata. Se l’argomento non viene indicato, si utilizza la variabile predefinita ‘**$_**’.

298.6 Funzioni di conversione

Nelle sezioni seguenti sono elencate le funzioni che si occupano di convertire dati in formati differenti.

Tabella 298.10. Elenco di alcune funzioni di conversione.

Nome	Descrizione.
chr()	Converte un numero nel carattere corrispondente.
hex()	Converte una stringa esadecimale nel numero corrispondente.
oct()	Converte una stringa ottale nel numero corrispondente.
ord()	Converte un carattere nel numero corrispondente.

298.6.1 chr()

```
chr n
```

‘**chr()**’ restituisce il carattere corrispondente al numero indicato come argomento. Se non viene specificato l’argomento, il numero viene letto dalla variabile ‘**$_**’.

Esempi

```
chr (65);
```

Restituisce la lettera ‘**A**’ maiuscola.

298.6.2 hex()

```
hex stringa
```

‘**hex()**’ interpreta il proprio argomento come una stringa contenente un numero esadecimale. Restituisce il numero (decimale) corrispondente. Se non viene specificato l’argomento, il dato viene letto dalla variabile ‘**$_**’.

Esempi

```
hex ("0xAf");
```

Restituisce il numero 175.

```
hex ("af");
```

Restituisce il numero 175.

298.6.3 oct()

```
oct stringa
```

‘**oct()**’ interpreta il proprio argomento come una stringa contenente un numero ottale. Restituisce il numero (decimale) corrispondente. Se non viene specificato l’argomento, il dato viene letto dalla variabile ‘**$_**’.

Esempi

```
$permessi = '0755';
mkdir ("/tmp/prova", oct ($permessi));
```

Crea la directory ‘`/tmp/prova/`’ con i permessi 0755_8. Dal momento che questi permessi sono contenuti in una variabile, in forma di stringa, devono essere convertiti in ottale prima dell’uso, altrimenti verrebbero interpretati in forma decimale.

298.6.4 ord()

```
ord stringa
```

‘**ord()**’ restituisce il valore numerico corrispondente al codice ASCII del primo carattere della stringa fornita come argomento. Se non viene specificato l’argomento, il dato viene letto dalla variabile ‘**$_**’.

298.7 Gestione delle espressioni

Sono elencate nelle sezioni seguenti le funzioni che si occupano di gestire l’esecuzione delle espressioni (quando necessario) e di conoscerne alcune caratteristiche.

298.7.1 defined()

```
defined espressione
```

‘**defined()**’ restituisce *Vero* se l’espressione (o la variabile) restituisce un valore diverso da indefinito. Il valore indefinito può essere restituito in particolare nelle seguenti situazioni:

- la lettura oltre la fine del file;
- un errore di sistema;
- una variabile non ancora inizializzata.

È importante non confondere il valore indefinito con lo zero o la stringa nulla: si tratta di tre cose differenti, in particolare, zero e stringa nulla sono valori definiti.

298.7.2 scalar()

```
scalar espressione
```

‘**scalar()**’ restituisce il risultato dell’espressione valutato in un contesto espressamente scalare.

298.8 Array e hash

Nelle sezioni seguenti sono elencate le funzioni che sono particolarmente dedicate alla gestione di array e hash.

Tabella 298.11. Elenco di alcune funzioni utili nella gestione degli array.

Nome	Descrizione.
delete()	Elimina elementi da un hash.
exists()	Verifica la presenza di una chiave all'interno di un hash.
keys()	Restituisce un array con le chiavi di un hash.
pop()	Estrae l'ultimo elemento di un array.
push()	Aggiunge un elemento in coda a un array.
splice()	Elimina o inserisce degli elementi in un array, in posizioni arbitrarie.

298.8.1 delete()

```
delete espressione
```

'**`delete()`**' elimina uno o più elementi da un hash. L'espressione che rappresenta l'argomento della funzione deve rappresentare uno o più elementi dell'hash. Restituisce i valori cancellati, cioè quelli abbinati alle chiavi indicate per la cancellazione.

Esempi

```
delete $miohash{ $miachiave };
```

Elimina dall'hash '**`%miohash`**' l'elemento rappresentato dalla chiave contenuta nella variabile '**`$miachiave`**'.

298.8.2 exists()

```
exists espressione
```

'**`exists()`**' verifica l'esistenza di una chiave all'interno di un hash. Se esiste, anche se il valore corrispondente dovesse risultare indefinito, restituisce *Vero*. L'espressione che rappresenta l'argomento della funzione deve rappresentare un solo elemento dell'hash.

Esempi

```
if (exists $miohash{ $miachiave })
  {
    ...
  }
```

Verifica l'esistenza dell'elemento rappresentato dalla chiave contenuta nella variabile '**`$miachiave`**', all'interno dell'hash '**`%miohash`**'. In caso affermativo esegue alcune istruzioni.

298.8.3 keys()

```
keys hash
```

‘**keys()**’ restituisce un array composto da tutte le chiavi dell’hash posto come argomento.

298.8.4 pop()

```
pop array
```

‘**pop()**’ restituisce l’ultimo elemento dell’array eliminandolo dall’array stesso (accorciandolo). In pratica tratta l’array come una pila (stack) ed esegue un’azione di *pop*.

298.8.5 push()

```
push array, lista
```

‘**push()**’ aggiunge all’array indicato come primo argomento gli elementi della lista successiva. In pratica tratta l’array come una pila (stack) ed esegue un’azione di *push*.

298.8.6 splice()

```
splice array, posizione_iniziale , lunghezza , lista
splice array, posizione_iniziale , lunghezza
splice array, posizione_iniziale
```

‘**splice()**’ elimina dall’array, indicato come primo argomento, gli elementi collocati a partire dalla posizione iniziale, indicata come secondo argomento, per una quantità definita dal terzo argomento. Se il terzo argomento (la quantità di elementi da eliminare) viene omesso, vengono eliminati tutti gli elementi a partire dalla posizione iniziale.

Se dopo il numero di argomenti da eliminare appaiono altri argomenti, vengono interpretati come una lista da inserire in sostituzione degli elementi cancellati. In tal modo, attraverso questa funzione, si può accorciare e allungare un array a piacimento, intervenendo in qualunque punto dello stesso.

298.9 Controllo dell’esecuzione del programma

Nelle sezioni seguenti sono elencate le funzioni che sono utili per controllare l’esecuzione di un programma Perl. In particolare ciò che permette di gestire le situazioni di errore.

Tabella 298.12. Elenco di alcune funzioni per il controllo dell’esecuzione del programma.

Nome	Descrizione.
die()	Emette un messaggio e termina l’esecuzione del programma.
do()	Esegue un sottoprogramma.
eval()	Controlla un gruppo di istruzioni.
exit()	Termina l’esecuzione del programma restituendo un valore.
require()	Richiede qualcosa per proseguire con il programma.
warn()	Emette un messaggio di avvertimento attraverso lo standard error.

298.9.1 die()

`die` *lista*

‘**`die()`**’ emette il contenuto degli elementi della lista fornita come argomento attraverso lo standard error e quindi termina l’esecuzione del programma.

Il programma Perl terminato in questo modo restituisce generalmente il valore contenuto dalla variabile ‘**`$!`**’.

Esempi

```
if (chdir '/var/spool/lpd')
  {
        ...
  }
else
  {
    die "L'operazione non è consentita.\n";
  }
```

Se lo spostamento nella directory ‘`/var/spool/lpd/`’ fallisce, visualizza il messaggio attraverso lo standard error e termina.

298.9.2 do()

`do` *file*

‘**`do()`**’ permette di includere il file indicato come argomento. In generale viene usato per inserire delle subroutine esterne.

Esempi

```
do 'prova.pl';
```

Esegue il contenuto del file ‘`prova.pl`’.

298.9.3 eval()

`eval` *blocco*
`eval` *espressione*

‘**`eval()`**’ permette di controllare l’esecuzione di un blocco di istruzioni, in modo da limitare i danni in caso di interruzione. In pratica, se all’interno del blocco si manifesta un errore di sintassi o di esecuzione, o ancora se viene incontrata un’istruzione ‘**`die()`**’, ‘**`eval()`**’ restituisce un valore indefinito e l’esecuzione del programma continua.

Se si manifesta un errore, questo viene riportato dalla variabile ‘**`$@`**’.

Nel caso non si verifichino errori, ‘**`eval()`**’ restituisce il valore dell’ultima espressione del blocco di istruzioni controllato.

298.9.4 exit()

`exit` *espressione*

‘**`exit()`**’ valuta l’espressione posta come argomento e termina l’esecuzione del programma restituendo all’esterno quel valore.

È importante ricordare che dal punto di vista dei programmi, la restituzione del valore zero corrisponde a una conclusione con successo, mentre un valore pari a uno o superiore, rappresenta una conclusione anomala.

Esempi

```
if (chdir '/var/spool/lpd')
  {
    ...
  }
else
  {
    print "L'operazione non è consentita.\n";
    exit 1;
  }
```

Se lo spostamento nella directory ‘`/var/spool/lpd/`’ fallisce, visualizza il messaggio e termina restituendo il valore uno.

298.9.5 require()

`require` *espressione*
`require` *file*

‘**`require()`**’ permette di specificare nel programma l’esigenza di qualcosa. Se si tratta di un’espressione il cui risultato è numerico, si vuole indicare che il programma richiede un interprete ‘**`perl`**’ di versione maggiore o uguale a quel numero. Se si tratta di una stringa si intende che il programma richiede l’inclusione del file corrispondente come libreria.

L’inclusione del file si ottiene solo se ciò non è già avvenuto.

298.9.6 warn()

`warn` *lista*

‘**`warn()`**’ emette il contenuto degli elementi della lista fornita come argomento attraverso lo standard error. Solitamente, ‘**`warn()`**’ viene utilizzato come ‘**`die()`**’ nelle situazioni in cui non è necessario interrompere l’esecuzione del programma.

298.10 Riferimenti

- Johan Vromans, *Perl 5 Desktop Guide*, O’Reilly & Associates

  <ftp://ftp.perl.org/pub/CPAN/authors/Johan_Vromans/>

Perl: esempi di programmazione

Questo capitolo raccoglie solo alcuni esempi di programmazione, in parte già descritti in altri capitoli. Lo scopo di questi esempi è solo didattico, utilizzando forme non ottimizzate per la velocità di esecuzione.

299.1 Problemi elementari di programmazione

In questa sezione vengono mostrati alcuni algoritmi elementari portati in Perl. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo 282.

299.1.1 Somma tra due numeri positivi

Il problema della somma tra due numeri positivi, attraverso l'incremento unitario, è stato descritto nella sezione 282.2.1.

```
#!/usr/bin/perl
#=====================================================================
# somma.pl <x> <y>
# Somma esclusivamente valori positivi.
#=====================================================================

#=====================================================================
# &somma (<x>, <y>)
#---------------------------------------------------------------------
sub somma
{
    local ($x) = $_[0];
```

```
    local ($y) = $_[1];

    local ($z) = $x;
    local ($i);

    for ($i = 1; $i <= $y; $i++)
      {
        $z++;
      }

    return $z;
}

#====================================================================
# Inizio del programma.
#--------------------------------------------------------------------
$x = $ARGV[0];
$y = $ARGV[1];

$z = &somma ($x, $y);

print "$x + $y = $z\n";

#====================================================================
```

In alternativa si può tradurre il ciclo '**`for`**' in un ciclo '**`while`**'.

```
sub somma
{
    local ($x) = $_[0];
    local ($y) = $_[1];

    local ($z) = $x;
    local ($i) = 1;

    while ($i <= $y)
      {
        $z++;
        $i++;
      }

    return $z;
}
```

299.1.2 Moltiplicazione di due numeri positivi attraverso la somma

Il problema della moltiplicazione tra due numeri positivi, attraverso la somma, è stato descritto nella sezione 282.2.2.

```
#!/usr/bin/perl
#====================================================================
# moltiplica.pl <x> <y>
#====================================================================

#====================================================================
# &moltiplica (<x>, <y>)
#--------------------------------------------------------------------
sub moltiplica
{
    local ($x) = $_[0];
    local ($y) = $_[1];

    local ($z) = 0;
    local ($i);

    for ($i = 1; $i <= $y; $i++)
```

```
      {
        $z = $z + $x;
      }

    return $z;
}

#=====================================================================
# Inizio del programma.
#---------------------------------------------------------------------
$x = $ARGV[0];
$y = $ARGV[1];

$z = &moltiplica ($x, $y);

print "$x * $y = $z\n";
#=====================================================================
```

In alternativa si può tradurre il ciclo '**for**' in un ciclo '**while**'.

```
sub moltiplica {
    local ($x) = $_[0];
    local ($y) = $_[1];

    local ($z) = 0;
    local ($i) = 1;

    while ($i <= $y)
      {
        $z = $z + $x;
        $i++;
      }

    return $z;
}
```

299.1.3 Divisione intera tra due numeri positivi

Il problema della divisione tra due numeri positivi, attraverso la sottrazione, è stato descritto nella sezione 282.2.3.

```
#!/usr/bin/perl
#=====================================================================
# dividi.pl <x> <y>
# Divide esclusivamente valori positivi.
#=====================================================================

#=====================================================================
# &dividi (<x>, <y>)
#---------------------------------------------------------------------
sub dividi
{
    local ($x) = $_[0];
    local ($y) = $_[1];

    local ($z) = 0;
    local ($i) = $x;

    while ($i >= $y)
      {
        $i = $i - $y;
        $z++;
      }

    return $z;
}
```

```
#=====================================================================
# Inizio del programma.
#---------------------------------------------------------------------
$x = $ARGV[0];
$y = $ARGV[1];

$z = &dividi ($x, $y);

print "Divisione intera - $x:$y = $z\n";
#=====================================================================
```

299.1.4 Elevamento a potenza

Il problema dell'elevamento a potenza tra due numeri positivi, attraverso la moltiplicazione, è stato descritto nella sezione 282.2.4.

```
#!/usr/bin/perl
#=====================================================================
# exp.pl <x> <y>
# Eleva a potenza.
#=====================================================================

#=====================================================================
# &exp (<x>, <y>)
#---------------------------------------------------------------------
sub exp
{
    local ($x) = $_[0];
    local ($y) = $_[1];

    local ($z) = 1;
    local ($i);

    for ($i = 1; $i <= $y; $i++)
      {
        $z = $z * $x;
      }

    return $z;
}

#=====================================================================
# Inizio del programma.
#---------------------------------------------------------------------
$x = $ARGV[0];
$y = $ARGV[1];

$z = &exp ($x, $y);

print "$x ** $y = $z\n";
#=====================================================================
```

In alternativa si può tradurre il ciclo '**for**' in un ciclo '**while**'.

```
sub exp
{
    local ($x) = $_[0];
    local ($y) = $_[1];

    local ($z) = 1;
    local ($i) = 1;

    while ($i <= $y)
      {
        $z = $z * $x;
```

```
        $i++;
      }

    return $z;
}
```

È possibile usare anche un algoritmo ricorsivo.

```
sub exp
{
    local ($x) = $_[0];
    local ($y) = $_[1];

    if ($x == 0)
      {
        return 0;
      }
    elsif ($y == 0)
      {
        return 1;
      }
    else
      {
        return ($x * &exp ($x, $y-1));
      }
}
```

299.1.5 Radice quadrata

Il problema della radice quadrata è stato descritto nella sezione 282.2.5.

```
#!/usr/bin/perl
#============================================================================
# radice.pl <x>
# Radice quadrata.
#============================================================================

#============================================================================
# &radice (<x>)
#----------------------------------------------------------------------------
sub radice
{
    local ($x) = $_[0];

    local ($z) = 0;
    local ($t) = 0;

    while (1)
      {
        $t = $z * $z;

        if ($t > $x)
          {
            # È stato superato il valore massimo.
            $z--;
            return $z;
          }

        $z++;
      }
    # Teoricamente, non dovrebbe mai arrivare qui.
}

#============================================================================
# Inizio del programma.
#----------------------------------------------------------------------------
```

```
$x = $ARGV[0];

$z = &radice ($x);

print "radq ($x) = $z\n";
#=====================================================================
```

299.1.6 Fattoriale

Il problema del fattoriale è stato descritto nella sezione 282.2.6.

```
#!/usr/bin/perl
#=====================================================================
# fatt.pl <x>
#=====================================================================

#=====================================================================
# &fatt (<x>)
#---------------------------------------------------------------------
sub fatt
{
    local ($x) = $_[0];

    local ($i) = ($x - 1);

    while ($i > 0)
      {
        $x = $x * $i;
        $i--;
      }

    return $x;
}

#=====================================================================
# Inizio del programma.
#---------------------------------------------------------------------
$x = $ARGV[0];

$fatt = &fatt ($x);

print "$x! = $fatt\n";
#=====================================================================
```

In alternativa, l'algoritmo si può tradurre in modo ricorsivo.

```
sub fatt {
    local ($x) = $_[0];

    if ($x > 1)
      {
        return ($x * &fatt ($x - 1));
      }
    else
      {
        return 1;
      }
}
```

299.1.7 Massimo comune divisore

Il problema del massimo comune divisore, tra due numeri positivi, è stato descritto nella sezione 282.2.7.

```
#!/usr/bin/perl
#=========================================================================
# mcd.pl <x> <y>
#=========================================================================

#=========================================================================
# &mcd (<x>, <y>)
#-------------------------------------------------------------------------
sub mcd
{
    local ($x) = $_[0];
    local ($y) = $_[1];

    while ($x != $y)
      {
        if ($x > $y)
          {
            $x = $x - $y;
          }
        else
          {
            $y = $y - $x;
          }
      }

    return $x;
}

#=========================================================================
# Inizio del programma.
#-------------------------------------------------------------------------
$x = $ARGV[0];
$y = $ARGV[1];

$z = &mcd ($x, $y);

print "Il massimo comune divisore di $x e $y è $z\n";
#=========================================================================
```

299.1.8 Numero primo

Il problema della determinazione se un numero sia primo o meno, è stato descritto nella sezione 282.2.8.

```
#!/usr/bin/perl
#=========================================================================
# primo.pl <x>
#=========================================================================

#=========================================================================
# &primo (<x>)
#-------------------------------------------------------------------------
sub primo
{
    local ($x) = $_[0];

    local ($primo) = 1;
    local ($i) = 2;
    local ($j);
```

```
    while (($i < $x) && $primo)
      {
        $j = int ($x / $i);
        $j = $x - ($j * $i);

        if ($j == 0)
          {
            $primo = 0;
          }
        else
          {
            $i++;
          }
      }

    return $primo;
}

#=====================================================================
# Inizio del programma.
#---------------------------------------------------------------------
$x = $ARGV[0];

if (&primo ($x))
  {
    print "$x è un numero primo\n";
  }
else
  {
    print "$x non è un numero primo\n";
  }
#=====================================================================
```

299.2 Scansione di array

In questa sezione vengono mostrati alcuni algoritmi, legati alla scansione degli array, portati in Perl. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo 282.

299.2.1 Ricerca sequenziale

Il problema della ricerca sequenziale all'interno di un array, è stato descritto nella sezione 282.3.1.

```
#!/usr/bin/perl
#=====================================================================
# ricercaseq.pl <elemento-cercato> <valore>...
#=====================================================================

#=====================================================================
# &ricercaseq (<lista>, <elemento>, <inizio>, <fine>)
#---------------------------------------------------------------------
sub ricercaseq
{
    #-----------------------------------------------------------------
    # Il primo argomento è un riferimento all'array, per cui
    # lo scalare $lista diventa il nuovo riferimento locale
    # all'array.
    # Per leggerlo come array occorrerà la forma @$lista, mentre
    # per leggerne un elemento occorrerà la forma ${$lista}[n].
    #-----------------------------------------------------------------
    local ($lista) = $_[0];
```

```
    local ($x) = $_[1];
    local ($a) = $_[2];
    local ($z) = $_[3];

    local ($i);

    for ($i = $a; $i <= $z; $i++)
      {
        if ($x == ${$lista}[$i])
          {
            return $i;
          }
      }

    # La corrispondenza non è stata trovata.
    return -1;
}

#=====================================================================
# Inizio del programma.
#---------------------------------------------------------------------

$x = $ARGV[0];
@lista = @ARGV[1 .. $#ARGV];

$i = &ricercaseq (\@lista, $x, 0, $#lista);

print "L'elemento $x si trova nella posizione $i\n";
#=====================================================================
```

Esiste anche una soluzione ricorsiva che viene mostrata nella subroutine seguente:

```
sub ricercaseq {
    local ($lista) = $_[0];
    local ($x) = $_[1];
    local ($a) = $_[2];
    local ($z) = $_[3];

    if ($a > $z)
      {
        return -1;
      }
    elsif ($x == ${$lista}[$a])
      {
        return $a;
      }
    else
      {
        return &ricercaseq ($lista, $x, $a+1, $z);
      }
}
```

299.2.2 Ricerca binaria

Il problema della ricerca binaria all'interno di un array, è stato descritto nella sezione 282.3.2.

```
#!/usr/bin/perl
#=====================================================================
# ricercabin.pl <elemento-cercato> <valore>...
#=====================================================================

#=====================================================================
# &ricercabin (<lista>, <elemento>, <inizio>, <fine>)
#---------------------------------------------------------------------
sub ricercabin
{
```

```
    #-------------------------------------------------------------------
    # Il primo argomento è un riferimento all'array, per cui
    # lo scalare $lista diventa il nuovo riferimento locale
    # all'array.
    # Per leggerlo come array occorrerà la forma @$lista, mentre
    # per leggerne un elemento occorrerà la forma ${$lista}[n].
    #-------------------------------------------------------------------
    local ($lista) = $_[0];
    local ($x) = $_[1];
    local ($a) = $_[2];
    local ($z) = $_[3];

    local ($m);

    # Determina l'elemento centrale.
    $m = int (($a + $z) / 2);

    if ($m < $a)
      {
        # Non restano elementi da controllare: l'elemento cercato non c'è.
        return -1;
      }
    elsif ($x < ${$lista}[$m])
      {
        # Si ripete la ricerca nella parte inferiore.
        return &ricercabin ($lista, $x, $a, $m-1);
      }
    elsif ($x > ${$lista}[$m])
      {
        # Si ripete la ricerca nella parte superiore.
        return &ricercabin ($lista, $x, $m+1, $z);
      }
    else
      {
        # $m rappresenta l'indice dell'elemento cercato.
        return $m;
      }
}

#=====================================================================
# Inizio del programma.
#---------------------------------------------------------------------

$x = $ARGV[0];
@lista = @ARGV[1 .. $#ARGV];

$i = &ricercabin (\@lista, $x, 0, $#lista);

print "L'elemento $x si trova nella posizione $i\n";
#=====================================================================
```

299.3 Algoritmi tradizionali

In questa sezione vengono mostrati alcuni algoritmi tradizionali portati in Perl. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo 282.

299.3.1 Bubblesort

Il problema del Bubblesort è stato descritto nella sezione 282.4.1. Viene mostrato prima una soluzione iterativa e successivamente la funzione '**bsort**' in versione ricorsiva.

```
#!/usr/bin/perl
#=========================================================================
# bsort.pl <valore>...
#=========================================================================

#=========================================================================
# &bsort (<lista>, <inizio>, <fine>)
#-------------------------------------------------------------------------
sub bsort
{
    #---------------------------------------------------------------------
    # Il primo argomento è un riferimento all'array, per cui
    # lo scalare $lista diventa il nuovo riferimento locale
    # all'array.
    # Per leggerlo come array occorrerà la forma @$lista, mentre
    # per leggerne un elemento occorrerà la forma ${$lista}[n].
    #---------------------------------------------------------------------
    local ($lista) = $_[0];
    local ($a) = $_[1];
    local ($z) = $_[2];

    local ($scambio);

    local ($j);
    local ($k);

    #---------------------------------------------------------------------
    # Inizia il ciclo di scansione dell'array.
    #---------------------------------------------------------------------
    for ($j = $a; $j < $z; $j++)
      {
        #-----------------------------------------------------------------
        # Scansione interna dell'array per collocare nella posizione
        # $j l'elemento giusto.
        #-----------------------------------------------------------------
        for ($k = $j+1; $k <= $z; $k++)
          {
            if (${$lista}[$k] < ${$lista}[$j])
              {
                #-------------------------------------------------------
                # Scambia i valori
                #-------------------------------------------------------
                $scambio = ${$lista}[$k];
                ${$lista}[$k] = ${$lista}[$j];
                ${$lista}[$j] = $scambio;
              }
          }
      }
}

#=========================================================================
# Inizio del programma.
#-------------------------------------------------------------------------

@lista = @ARGV;

&bsort (\@lista, 0, $#lista);

print "@lista\n";

#=========================================================================
```

Segue la funzione '**bsort**' in versione ricorsiva.

```
sub bsort
{
    local ($lista) = $_[0];
    local ($a) = $_[1];
    local ($z) = $_[2];

    local ($k);
    local ($scambio);

    if ($a < $z)
      {
        #-----------------------------------------------------------------
        # Scansione interna dell'array per collocare nella posizione
        # $a l'elemento giusto.
        #-----------------------------------------------------------------
        for ($k = $a+1; $k <= $z; $k++)
          {
            if (${$lista}[$k] < ${$lista}[$a])
              {
                #-------------------------------------------------------
                # Scambia i valori
                #-------------------------------------------------------
                $scambio = ${$lista}[$k];
                ${$lista}[$k] = ${$lista}[$a];
                ${$lista}[$a] = $scambio;
              }
          }

        &bsort ($lista, $a+1, $z);
      }
}
```

299.3.2 Torre di Hanoi

Il problema della torre di Hanoi è stato descritto nella sezione 282.4.2.

```
#!/usr/bin/perl
#============================================================================
# hanoi.pl <n-anelli> <piolo-iniziale> <piolo-finale>
#============================================================================

#============================================================================
# &hanoi (<n-anelli>, <piolo-iniziale>, <piolo-finale>)
#----------------------------------------------------------------------------
sub hanoi {
    local ($n) = $_[0];
    local ($p1) = $_[1];
    local ($p2) = $_[2];

    if ($n > 0)
      {
        &hanoi ($n-1, $p1, 6-$p1-$p2);
        print "Muovi l'anello $n dal piolo $p1 al piolo $p2\n";
        &hanoi ($n-1, 6-$p1-$p2, $p2);
      }
}
#============================================================================

#============================================================================
# Inizio del programma.
#----------------------------------------------------------------------------
$n = $ARGV[0];
$p1 = $ARGV[1];
$p2 = $ARGV[2];
```

```
&hanoi ($n, $p1, $p2);

#=============================================================================
```

299.3.3 Quicksort

L'algoritmo del Quicksort è stato descritto nella sezione 282.4.3.

```
#!/usr/bin/perl
#=============================================================================
# qsort.pl <valore>...
#=============================================================================

#=============================================================================
# &part (<lista>, <inizio>, <fine>)
#-----------------------------------------------------------------------------
sub part
{
    #-------------------------------------------------------------------
    # Il primo argomento è un riferimento all'array, per cui
    # lo scalare $lista diventa il nuovo riferimento locale
    # all'array.
    # Per leggerlo come array occorrerà la forma @$lista, mentre
    # per leggerne un elemento occorrerà la forma ${$lista}[n].
    #-------------------------------------------------------------------
    local ($lista) = $_[0];
    local ($a) = $_[1];
    local ($z) = $_[2];

    #-------------------------------------------------------------------
    # Viene preparata una variabile che servirà per scambiare due
    # valori.
    #-------------------------------------------------------------------
    local ($scambio) = 0;

    #-------------------------------------------------------------------
    # Si assume che $a sia inferiore a $z.
    #-------------------------------------------------------------------
    local ($i) = $a + 1;
    local ($cf) = $z;

    #-------------------------------------------------------------------
    # Inizia il ciclo di scansione dell'array.
    #-------------------------------------------------------------------
    while (1)
      {
        while (1)
          {
            #-----------------------------------------------------
            # Sposta $i a destra.
            #-----------------------------------------------------
            if ((${$lista}[$i] > ${$lista}[$a]) || ($i >= $cf))
              {
                last;
              }
            else
              {
                $i += 1;
              }

          }

        while (1)
          {
            #-----------------------------------------------------
```

```
            # Sposta $cf a sinistra.
            #-------------------------------------------------------
            if (${$lista}[$cf] <= ${$lista}[$a])
              {
                last;
              }
            else
             {
                $cf -= 1;
             }
          }

        if ($cf <= $i)
          {
            #-------------------------------------------------------
            # È avvenuto l'incontro tra $i e $cf.
            #-------------------------------------------------------
            last;
          }
        else
          {
            #-------------------------------------------------------
            # Vengono scambiati i valori.
            #-------------------------------------------------------
            $scambio = ${$lista}[$cf];
            ${$lista}[$cf] = ${$lista}[$i];
            ${$lista}[$i] = $scambio;

            $i += 1;
            $cf -= 1;
          }
      }

    #-----------------------------------------------------------------
    # A questo punto @$lista[$a..$z] è stata ripartita e $cf è la
    # collocazione di @$lista[$a].
    #-----------------------------------------------------------------
    $scambio = ${$lista}[$cf];
    ${$lista}[$cf] = ${$lista}[$a];
    ${$lista}[$a] = $scambio;

    #-----------------------------------------------------------------
    # A questo punto, @$lista[$cf] è un elemento (un valore) nella
    # giusta posizione.
    #-----------------------------------------------------------------

    return $cf;

}

#=====================================================================
# &quicksort (<lista>, <inizio>, <fine>)
#---------------------------------------------------------------------
sub quicksort
{
    #-----------------------------------------------------------------
    # Il primo argomento è un riferimento all'array, per cui
    # lo scalare $lista diventa il nuovo riferimento locale
    # all'array.
    #-----------------------------------------------------------------
    local ($lista) = $_[0];
    local ($a) = $_[1];
    local ($z) = $_[2];

    #-----------------------------------------------------------------
    # Viene preparata la variabile $cf.
```

```
    #-------------------------------------------------------------
    local ($cf) = 0;

    if ($z > $a)
      {
        $cf = &part ($lista, $a, $z);
        &quicksort ($lista, $a, $cf-1);
        &quicksort ($lista, $cf+1, $z);
      }
}

#=============================================================================
# Inizio del programma.
#-----------------------------------------------------------------------------

@lista = @ARGV;

quicksort (\@lista, 0, $#lista);

print "@lista\n";

#=============================================================================
```

299.3.4 Permutazioni

L'algoritmo ricorsivo delle permutazioni è stato descritto nella sezione 282.4.4.

```
#!/usr/bin/perl
#=============================================================================
# permuta.pl <valore>...
#=============================================================================

#=============================================================================
# &permuta (<lista>, <inizio>, <fine>)
#-----------------------------------------------------------------------------
sub permuta
{
    #-------------------------------------------------------------------
    # Il primo argomento è un riferimento all'array, per cui
    # lo scalare $lista diventa il nuovo riferimento locale
    # all'array.
    # Per leggerlo come array occorrerà la forma @$lista, mentre
    # per leggerne un elemento occorrerà la forma ${$lista}[n].
    #-------------------------------------------------------------------
    local ($lista) = $_[0];
    local ($a) = $_[1];
    local ($z) = $_[2];

    local ($scambio);

    local ($k);

    #-------------------------------------------------------------------
    # Se il segmento di array contiene almeno due elementi, si
    # procede.
    #-------------------------------------------------------------------
    if (($z - $a) >= 1)
      {
        #---------------------------------------------------------------
        # Inizia un ciclo di scambi tra l'ultimo elemento e uno degli
        # altri contenuti nel segmento di array.
        #---------------------------------------------------------------
        for ($k = $z; $k >= $a; $k--)
          {
            #-------------------------------------------------------
```

```
            # Scambia i valori
            #-------------------------------------------------------
            $scambio = ${$lista}[$k];
            ${$lista}[$k] = ${$lista}[$z];
            ${$lista}[$z] = $scambio;

            #-------------------------------------------------------
            # Esegue una chiamata ricorsiva per permutare un segmento
            # più piccolo dell'array.
            #-------------------------------------------------------
            permuta ($lista, $a, $z-1);

            #-------------------------------------------------------
            # Scambia i valori
            #-------------------------------------------------------
            $scambio = ${$lista}[$k];
            ${$lista}[$k] = ${$lista}[$z];
            ${$lista}[$z] = $scambio;
          }
      }
    else
      {
        #-----------------------------------------------------------
        # Visualizza la situazione attuale dell'array.
        #-----------------------------------------------------------
        print "@$lista\n";
      }
}

#============================================================================
# Inizio del programma.
#----------------------------------------------------------------------------

@lista = @ARGV;

&permuta (\@lista, 0, $#lista);
#============================================================================
```

Perl: esercizi di programmazione

Questo capitolo raccoglie una sequenza di esercizi didattici di programmazione realizzati in Perl. Il contenuto e la gradualità degli esercizi è orientato verso studenti di scuola media. Come si può osservare dagli esempi, in tutti i programmi viene aggiunto inizialmente l'opzione '**-w**' per assicurare l'emissione di informazioni diagnostiche da parte dell'interprete.[1]

300.1 Area del rettangolo

Con il pretesto di calcolare l'area di un rettangolo, si vuole introdurre all'uso dell'istruzione '**print**', alla gestione delle stringhe, con la relativa espansione delle variabili e l'eliminazione del codice di interruzione di riga.

1. La prima soluzione proposta ha lo scopo di mostrare l'uso dei flussi di file standard nel linguaggio Perl.

```
#!/usr/bin/perl -w
#
# Programma area-01.pl
# Scritto da ...
#
# Programma per trovare l'area di un rettangolo.

print ("Inserisci la base: ");
$base = <STDIN>;
print ("Inserisci l'altezza: ");
$altezza = <STDIN>;
$area = $base * $altezza;
print ("Il rettangolo con una base di ");
print $base;
print (" e un'altezza di ");
print $altezza;
print (" ha un'area di ");
print $area;
print ("\n");
```

   Inserendo rispettivamente i valori 10 e 20, il risultato che si ottiene è quello dell'interazione seguente:

```
Inserisci la base: 10[ Invio ]

Inserisci l'altezza: 20[ Invio ]

Il rettangolo con una base di 10
 e un'altezza di 20
 ha un'area di 200
```

2. La seconda soluzione serve a mostrare la possibilità di concatenare le stringhe nella composizione della frase finale, attraverso l'operatore '**.**'.

```
#!/usr/bin/perl -w
#
# Programma area-02.pl
# Scritto da ...
#
# Programma per trovare l'area di un rettangolo.
# Rispetto alla versione 01 si introduce il concatenamento
# di stringa.

print ("Inserisci la base: ");
$base = <STDIN>;
print ("Inserisci l'altezza: ");
```

[1] Questo capitolo è ispirato da un lavoro didattico del prof. Antonio Bernardi, *brngb @ tin.it*.

```
$altezza = <STDIN>;
$area = $base * $altezza;
print ("Il rettangolo con una base di "
       . $base
       . " e un'altezza di "
       . $altezza
       . " ha un'area di "
       . $area
       . "\n");
```

3. La terza soluzione serve a mostrare l'opportunità di fare riferimento allo standard output in modo esplicito, indicare espressamente il nome '**STDOUT**' nell'istruzione '**print**'; inoltre, si mostra la possibilità di espandere le variabili scalari all'interno delle stringhe letterali.

```
#!/usr/bin/perl -w
#
# Programma area-03.pl
# Scritto da ...
#
# Programma per trovare l'area di un rettangolo.
#
# Rispetto alla versione 02 si introduce il riferimento allo
# standard output in modo esplicito e si mostra l'espansione
# delle variabili scalari all'interno delle stringhe.

print STDOUT ("Inserisci la base: ");
$base = <STDIN>;
print STDOUT ("Inserisci l'altezza: ");
$altezza = <STDIN>;
$area = $base * $altezza;
print STDOUT ("Il rettangolo con una base di $base e un'altezza di "
              . "$altezza ha un'area di $area\n");
```

4. Come sarà stato possibile osservare, l'inserimento dei valori attraverso istruzioni del tipo '*variabile* **= <STDIN>**', include anche il codice di interruzione di riga, con il quale in effetti si termina l'inserimento. Per ovviare a questo inconveniente, nella quarta variante dell'esercizio si utilizza l'istruzione '**chomp**'.

```
#!/usr/bin/perl -w
#
# Programma area-04.pl
# Scritto da ...
#
# Programma per trovare l'area di un rettangolo.
#
# Rispetto alla versione 03 si utilizza «chomp» per eliminare
# il codice di interruzione di riga finale che accompagna i valori
# inseriti.

print STDOUT ("Inserisci la base: ");
$base = <STDIN>;
chomp ($base);
print STDOUT ("Inserisci l'altezza: ");
$altezza = <STDIN>;
chomp ($altezza);
$area = $base * $altezza;
print STDOUT ("Il rettangolo con una base di $base e un'altezza di "
              . "$altezza ha un'area di $area\n");
```

L'utilizzo di '**chomp**' risolve il problema di visualizzazione del risultato che avevano tutti gli esempi precedenti:

```
Inserisci la base: 10[ Invio ]

Inserisci l'altezza: 20[ Invio ]

Il rettangolo con una base di 10 e un'altezza di 20 ha un'area di 200
```

5. La quinta soluzione mostra l'opportunità di dichiarare le variabili prima dell'uso, inizializ-

zandole secondo il tipo di dati per le quali verranno adoperate. Questo serve a migliorare la leggibilità del programma, anche se il linguaggio non richiede tale accortezza.

```
#!/usr/bin/perl -w
#
# Programma area-05.pl
# Scritto da ...
#
# Programma per trovare l'area di un rettangolo.
#
# Rispetto alla versione 04 si dichiarano e si inizializzano le
# variabili prima del loro uso.

$base=0;
$altezza=0;
$area=0;

print STDOUT ("Inserisci la base: ");
$base = <STDIN>;
chomp ($base);
print STDOUT ("Inserisci l'altezza: ");
$altezza = <STDIN>;
chomp ($altezza);
$area = $base * $altezza;
print STDOUT ("Il rettangolo con una base di $base e un'altezza di "
              . "$altezza ha un'area di $area\n");
```

6. La sesta soluzione mostra l'opportunità di aggiungere delle descrizioni (commenti) all'interno del sorgente, per spiegare il significato di ciò che viene fatto, facilitando così l'interpretazione dello stesso per la lettura umana. Inoltre, l'istruzione iniziale '**`system ("clear")`**' serve a introdurre l'uso di comandi del sistema operativo sottostante.

```
#!/usr/bin/perl -w
#
# Programma area-06.pl
# Scritto da ...
#
# Programma per trovare l'area di un rettangolo.
#
# Rispetto alla versione 05 si aggiungono dei commenti descrittivi.

#-----------------------------------------------------------------------
# DICHIARAZIONE DELLE VARIABILI
#-----------------------------------------------------------------------

# Si creano le variabili $base, $altezza, $area, inizializzandole
# nello stesso contesto.

$base=0;                # crea la variabile $base    e la inizializza
$altezza=0;             # crea la variabile $altezza e la inizializza
$area=0;                # crea la variabile $area    e la inizializza

#-----------------------------------------------------------------------
# INSERIMENTO DEI DATI (INPUT)
#-----------------------------------------------------------------------

# Si ripulisce lo schermo con il comando «clear» del sistema operativo.
system ("clear");

# Si emette un messaggio di invito a inserire il valore della base.
print STDOUT ("Inserisci la base: ");

# Si assegna alla variabile $base una riga proveniente dallo standard
# input, ovvero ciò che viene inserito presumibilmente dalla tastiera.
$base = <STDIN>;

# Si toglie il codice di interruzione di riga che si trova alla fine
# della variabile $base.
chomp ($base);
```

```
# Si emette un messaggio di invito a inserire il valore dell'altezza.
print STDOUT ("Inserisci l'altezza: ");

# Si assegna alla variabile $altezza una riga proveniente dallo standard
# input, ovvero ciò che viene inserito presumibilmente dalla tastiera.
$altezza = <STDIN>;

# Si toglie il codice di interruzione di riga che si trova alla fine
# della variabile $altezza.
chomp ($altezza);

#-----------------------------------------------------------------------
# ELABORAZIONE DEI DATI
#-----------------------------------------------------------------------

# Assegna alla variabile $area il prodotto del contenuto di $base e di
# $altezza; in altri termini si calcola l'area e la si assegna alla
# variabile $area.
$area = $base * $altezza;

#-----------------------------------------------------------------------
# EMISSIONE DEL RISULTATO DELL'ELABORAZIONE (OUTPUT)
#-----------------------------------------------------------------------

# Si emette un messaggio con il quale si mostra il risultato del calcolo
# dell'area del rettangolo.
print STDOUT ("Il rettangolo con una base di $base e un'altezza di "
              . "$altezza ha un'area di $area\n");
```

7. La settima soluzione mostra la possibilità di utilizzare la riga di comando per fornire i dati da elaborare. In pratica, la base viene ottenuta dal primo argomento, mentre l'altezza si ottiene dal secondo argomento.

```
#!/usr/bin/perl -w
#
# Programma area-07.pl
# Scritto da ...
#
# Programma per trovare l'area di un rettangolo.
#
# Rispetto alla versione 06 si ottengono i dati in ingresso dalla
# riga di comando (inoltre mancano i commenti).

$base=0;
$altezza=0;
$area=0;

$base    = $ARGV[0];
$altezza = $ARGV[1];

$area = $base * $altezza;
print STDOUT ("Il rettangolo con una base di $base e un'altezza di "
              . "$altezza ha un'area di $area\n");
```

L'esempio seguente mostra l'avvio del programma per calcolare l'area di un rettangolo con base 10 e altezza 20:

```
$ ./area-07.pl 10 20[ Invio ]

Il rettangolo con una base di 10 e un'altezza di 20 ha un'area di 200
```

300.2 Ricerca del valore scalare più alto

Lo scopo di questi esercizi è quello di far prendere confidenza con le espressioni di confronto numerico e confronto tra stringhe, che in Perl usano operatori differenti nei due casi. Inoltre, si presenta la struttura condizionale.

1. Il primo caso mostra la ricerca del valore più altro tra tre valori numerici. In questo caso si usano gli operatori '**>**', '**<**' e gli altri di questa serie.

```
#!/usr/bin/perl -w
#
# Programma massimo-01.pl
# Scritto da ...
#
# Programma per trovare il valore massimo tra tre numeri.
#

$num1 = 0;
$num2 = 0;
$num3 = 0;
$max  = 0;

print STDOUT ("inserisci il primo numero: ");
$num1 = <STDIN>;
chomp ($num1);
print STDOUT ("inserisci il secondo numero: ");
$num2 = <STDIN>;
chomp ($num2);
print STDOUT ("inserisci il terzo numero: ");
$num3 = <STDIN>;
chomp ($num3);

if ($num1 > $num2)
  {
    $max = $num1;
  }
else
  {
    $max = $num2;
  }
if ($num3 > $max)
  {
    $max = $num3;
  }

print STDOUT ("Il massimo tra $num1, $num2 e $num3, è $max\n");
```

2. Il secondo caso mostra la ricerca della stringa lessicograficamente superiore tra tre valori stringa. In questo caso si usano gli operatori '**gt**', '**lt**' e gli altri di questa serie.

```
#!/usr/bin/perl -w
#
# Programma massimo-02.pl
# Scritto da ...
#
# Programma per trovare il valore lessicograficamente superiore
# tra tre stringhe.
#

$str1 = "";
$str2 = "";
$str3 = "";
$max  = "";

print STDOUT ("inserisci la prima stringa: ");
$str1 = <STDIN>;
chomp ($str1);
print STDOUT ("inserisci la seconda stringa: ");
```

```
$str2 = <STDIN>;
chomp ($str2);
print STDOUT ("inserisci la terza stringa: ");
$str3 = <STDIN>;
chomp ($str3);

if ($str1 gt $str2)
  {
    $max = $str1;
  }
else
  {
    $max = $str2;
  }
if ($str3 gt $max)
  {
    $max = $str3;
  }

print STDOUT ("Il massimo tra \"$str1\", \"$str2\" e \"$str3\", è \"$max\"\n");
```

300.3 Equazione di primo e di secondo grado

1. L'equazione di primo grado ax+b=0 si risolve come x=-b/a, dove la soluzione è indeterminata se «a» e «b» hanno valore zero, oppure è impossibile se «a» vale zero e «b» ha un valore diverso da zero.

```
#!/usr/bin/perl -w
#
# Programma equazione-01.pl
# Scritto da ...
#
# Programma per risolvere un'equazione di primo grado.
#

$a         = 0;
$b         = 0;
$soluzione = 0;

print STDOUT ("inserisci a: ");
$a = <STDIN>;
print STDOUT ("inserisci b: ");
$b = <STDIN>;

if ($a == 0 && $b == 0)
  {
    print STDOUT ("L'equazione è indeterminata\n");
  }
elsif ($a == 0 && $b != 0)
  {
    print STDOUT ("l'equazione è impossibile\n");
  }
elsif ($a != 0)
  {
    $soluzione = -$b/$a;
    print STDOUT ("la soluzione è $soluzione\n");
  }
```

2. Equazione di secondo grado.

```
#!/usr/bin/perl -w
#
# Programma equazione-02.pl
# Scritto da ...
#
# Programma per risolvere un'equazione di secondo grado.
#
```

```
$a         = 0;
$b         = 0;
$c         = 0;
$discri    = 0;
$x1        = 0;
$x2        = 0;

print STDOUT ("inserisci a: ");
$a = <STDIN>;
print STDOUT ("inserisci b: ");
$b = <STDIN>;
print STDOUT ("inserisci c: ");
$c = <STDIN>;

$discri = $b ** 2 - 4*$a*$c;

if ($a == 0)
  {
    print STDOUT ("L'equazione è di primo grado\n");
  }
elsif ($discri < 0)
  {
    print STDOUT ("La soluzione è imposibile: il discriminante è $discri\n");
  }
elsif ($discri == 0)
  {
    $x1 = -$b/(2*$a);
    print STDOUT ("Le soluzioni sono reali e coincidenti x1=x2= $x1\n");
  }
elsif ($discri > 0)
  {
    $x1 = (-$b - sqrt ($discri))/2*$a;
    $x2 = (-$b + sqrt ($discri))/2*$a;
    print STDOUT ("Le soluzioni sono x1= $x1 e x2= $x2 \n");
  }
```

300.4 Somma ciclica

Lo scopo di questi esercizi è quello di far prendere confidenza con le strutture iterative ed enumerative.

1. Ciclo iterativo.

```
#!/usr/bin/perl -w
#
# Programma somma-01.pl
# Scritto da ...
#
# Programma per sommare i primi k numeri naturali.
# Utilizza la struttura while.
#

$k     = 0;
$n     = 0;
$somma = 0;

print STDOUT ("inserisci il valore per k: ");
$k = <STDIN>;
chomp ($k);
$n = 1;
$somma = 0;
while ($n <= $k)
  {
    $somma = $somma +$n;
    $n = $n + 1;
  }

print STDOUT ("La somma dei primi $k numeri è $somma\n");
```

2. Ciclo iterativo con una struttura differente.

```
#!/usr/bin/perl -w
#
# Programma somma-02.pl
# Scritto da ...
#
# Programma per sommare i primi k numeri naturali.
# Utilizza la struttura until.
#

$k     = 0;
$n     = 0;
$somma = 0;

print STDOUT ("inserisci il valore per k: ");
$k = <STDIN>;
chomp ($k);
$n = 1;
$somma = 0;
until ($n > $k)
  {
    $somma = $somma +$n;
    $n = $n + 1;
  }

print STDOUT ("La somma dei primi $k numeri è $somma\n");
```

3. Ciclo enumerativo.

```
#!/usr/bin/perl -w
#
# Programma somma-03.pl
# Scritto da ...
#
# Programma per sommare i primi k numeri naturali.
# Utilizza la struttura for.
#

$k     = 0;
$n     = 0;
$somma = 0;

print STDOUT ("inserisci il valore per k: ");
$k = <STDIN>;
chomp ($k);

$somma = 0;
for ($n = 1; $n <= $k; $n++)
  {
    $somma += $n;
  }

print STDOUT ("La somma dei primi $k numeri è $somma\n");
```

4. Ciclo enumerativo più sofisticato.

```
#!/usr/bin/perl -w
#
# Programma somma-04.pl
# Scritto da ...
#
# Programma per sommare i primi k numeri naturali.
# Utilizza la struttura for in modo più sofisticato.
#

$k     = 0;
$n     = 0;
$somma = 0;

print STDOUT ("inserisci il valore per k: ");
$k = <STDIN>;
```

```
chomp ($k);

for ($somma = 0, $n = 1; $n <= $k; $somma += $n, $n++)
  {
    ;
  }

print STDOUT ("La somma dei primi $k numeri è $somma\n");
```

300.5 Prodotto ciclico

Lo scopo di questi esercizi è quello di far prendere confidenza con le strutture iterative ed enumerative.

1. Ciclo iterativo.

```
#!/usr/bin/perl -w
#
# Programma prodotto-01.pl
# Scritto da ...
#
# Programma per moltiplicare i primi k numeri naturali.
# Utilizza la struttura while.
#

$k        = 0;
$n        = 0;
$prodotto = 0;

print STDOUT ("inserisci il valore per k: ");
$k = <STDIN>;
chomp ($k);
$n = 1;
$prodotto = 1;
while ($n <= $k)
  {
    $prodotto = $prodotto * $n;
    $n = $n + 1;
  }
print STDOUT ("il prodotto dei primi $k numeri è $prodotto\n");
```

2. Ciclo iterativo con una struttura differente.

```
#!/usr/bin/perl -w
#
# Programma prodotto-02.pl
# Scritto da ...
#
# Programma per moltiplicare i primi k numeri naturali.
# Utilizza la struttura until.
#

$k        = 0;
$n        = 0;
$prodotto = 0;

print STDOUT ("inserisci il valore per k: ");
$k = <STDIN>;
chomp ($k);
$n = 1;
$prodotto = 1;
until ($n > $k)
  {
    $prodotto = $prodotto * $n;
    $n = $n + 1;
  }

print STDOUT ("il prodotto dei primi $k numeri è $prodotto\n");
```

3. Ciclo enumerativo.

```
#!/usr/bin/perl -w
#
# Programma prodotto-03.pl
# Scritto da ...
#
# Programma per moltiplicare i primi k numeri naturali.
# Utilizza la struttura for.
#

$k        = 0;
$n        = 0;
$prodotto = 0;

print STDOUT ("inserisci il valore per k: ");
$k = <STDIN>;
chomp ($k);

$prodotto = 1;
for ($n = 1; $n <= $k; $n++)
  {
    $prodotto *= $n;
  }

print STDOUT ("il prodotto dei primi $k numeri è $prodotto\n");
```

4. Ciclo enumerativo più sofisticato.

```
#!/usr/bin/perl -w
#
# Programma prodotto-04.pl
# Scritto da ...
#
# Programma per moltiplicare i primi k numeri naturali.
# Utilizza la struttura for in modo più sofisticato.
#

$k        = 0;
$n        = 0;
$prodotto = 0;

print STDOUT ("inserisci il valore per k: ");
$k = <STDIN>;
chomp ($k);

for ($prodotto = 1, $n = 1; $n <= $k; $prodotto *= $n, $n++)
  {
    ;
  }

print STDOUT ("il prodotto dei primi $k numeri è $prodotto\n");
```

300.6 Scansione di array

Lo scopo di questi esercizi è quello di far prendere confidenza con la scansione degli array.

1. Il primo esercizio mostra l'inserimento di dati all'interno di un vettore (in forma di array) e la sua scansione allo scopo di mostrarne il contenuto.

```
#!/usr/bin/perl -w
#
# Programma vettore-01.pl
# Scritto da ...
#
# Programma per inserire dati all'interno di un vettore e per visualizzarli.
#
```

```
@vettore = ();
$k       = 0;
$i       = 0;

print STDOUT ("inserisci l'indice massimo del vettore: ");
$k = <STDIN>;
chomp ($k);

# Inserimento.
for ($i = 0; $i <= $k; $i++)
  {
    print STDOUT ("inserisci l'elemento $i:");
    $vettore[$i] = <STDIN>;
    chomp ($vettore[$i]);
  }

# Visualizzazione.
print STDOUT ("Gli elementi del vettore sono: ");
for ($i = 0; $i <= $k; $i++ )
  {
    print STDOUT ($vettore[$i]);
    print STDOUT (" ");
  }
print STDOUT ("\n");
```

2. Il secondo esercizio mostra la ricerca del valore massimo all'interno di un vettore non ordinato.

```
#!/usr/bin/perl -w
#
# Programma vettore-02.pl
# Scritto da ...
#
# Programma per cercare il valore massimo all'interno di un vettore
# non ordinato.
#

@vettore = ();
$k       = 0;
$i       = 0;
$max     = 0;

print STDOUT ("inserisci l'indice massimo del vettore: ");
$k = <STDIN>;
chomp ($k);

# Inserimento.
for ($i = 0; $i <= $k; $i++)
  {
    print STDOUT ("inserisci l'elemento $i:");
    $vettore[$i] = <STDIN>;
    chomp ($vettore[$i]);
  }

# Scansione di ricerca del massimo.
for ($i = 0; $i <= $k; $i++ )
  {
    if ($vettore[$i] > $max)
      {
        $max = $vettore[$i];
      }
  }

# Risultato.
print STDOUT ("Il massimo è $max\n");
```

3. Nel terzo esercizio si aggiunge un controllo nel caso in cui il vettore da scandire sia vuoto.

```
#!/usr/bin/perl -w
#
# Programma vettore-03.pl
```

```
# Scritto da ...
#
# Programma per cercare il valore massimo all'interno di un vettore
# non ordinato. Se il vettore è vuoto non si esegue alcuna scansione.
#

@vettore = ();
$k       = 0;
$i       = 0;
$max     = 0;

print STDOUT ("inserisci l'indice massimo del vettore: ");
$k = <STDIN>;
chomp ($k);

# Inserimento.
for ($i = 0; $i <= $k; $i++)
  {
    print STDOUT ("inserisci l'elemento $i:");
    $vettore[$i] = <STDIN>;
    chomp ($vettore[$i]);
  }

# Scansione di ricerca del massimo.
if ($k < 0)
  {
    print STDOUT ("Non ci sono elementi nel vettore\n");
  }
else
  {
    for ($i = 0; $i <= $k; $i++ )
      {
        if ($vettore[$i] > $max)
          {
            $max = $vettore[$i];
          }
      }

    # Risultato.
    print STDOUT ("Il massimo è $max\n");
  }
```

4. Ordinamento di un vettore con l'algoritmo Bubblesort. L'elaborazione avviene con cicli iterativi.

```
#!/usr/bin/perl -w
#
# Programma vettore-04.pl
# Scritto da ...
#
# Programma per riordinare gli elementi di un vettore.
# Si utilizza la struttura while.
#

@vettore = ();
$k       = 0;
$i       = 0;
$j       = 0;
$scambio = 0;

print STDOUT ("Inserisci l'indice massimo del vettore: ");
$k = <STDIN>;
chomp ($k);

# Inserimento.
for ($i = 0; $i <= $k; $i++)
  {
    print STDOUT ("inserisci l'elemento $i:");
    $vettore[$i] = <STDIN>;
    chomp ($vettore[$i]);
  }
```

```
# Riordino.
$j = $k;
while ($j >= 1)
  {
    $i = 0;
    while ($i <= $j-1)
      {
        if ($vettore[$i] > $vettore[$i+1])
          {
            $scambio = $vettore[$i];
            $vettore[$i] = $vettore[$i+1];
            $vettore[$i+1] = $scambio;
          }
        $i++;
      }
    $j--
  }

# Visualizzazione.
print STDOUT ("Il vettore ordinato è: ");
for ($i = 0; $i <= $k; $i++ )
  {
    print STDOUT ($vettore[$i]);
    print STDOUT (" ");
  }
print STDOUT ("\n");
```

5. Ordinamento di un vettore con l'algoritmo Bubblesort. Questa volta si usano cicli enumerativi.

```
#!/usr/bin/perl -w
#
# Programma vettore-05.pl
# Scritto da ...
#
# Programma per riordinare gli elementi di un vettore.
# Si utilizza la struttura for.
#

@vettore = ();
$k       = 0;
$i       = 0;
$j       = 0;
$scambio = 0;

print STDOUT ("Inserisci l'indice massimo del vettore: ");
$k = <STDIN>;
chomp ($k);

# Inserimento.
for ($i = 0; $i <= $k; $i++)
  {
    print STDOUT ("inserisci l'elemento $i:");
    $vettore[$i] = <STDIN>;
    chomp ($vettore[$i]);
  }

# Riordino.
$j = $k;

for ($j = $k; $j >= 1; $j--)
  {
    for ($i = 0; $i <= $j-1; $i++)
      {
        if ($vettore[$i] > $vettore[$i+1])
          {
            $scambio = $vettore[$i];
            $vettore[$i] = $vettore[$i+1];
            $vettore[$i+1] = $scambio;
          }
```

```
      }
  }

# Visualizzazione.
print STDOUT ("Il vettore ordinato è: ");
for ($i = 0; $i <= $k; $i++ )
  {
    print STDOUT ($vettore[$i]);
    print STDOUT (" ");
  }
print STDOUT ("\n");
```

300.7 Elaborazione con in file

Questi esercizi servono a prendere un po' di confidenza con la lettura e la scrittura dei file.

1. I flussi di file standard risultano già aperti. Il primo esercizio mostra la lettura e la scrittura con questi flussi.

```
#!/usr/bin/perl -w
#
# Programma file-01.pl
# Scritto da ...
#
# Programma per il trasferimento dello standard input nello
# standard output.
#

$riga = "";

while ($riga = <STDIN>)
  {
    print STDOUT ($riga);
  }
```

2. Nel caso si vogliano gestire altri file, è necessario aprire il flusso di file relativo. Nel prossimo esercizio si visualizza il contenuto di un file, il cui nome viene specificato in modo interattivo.

```
#!/usr/bin/perl -w
#
# Programma file-02.pl
# Scritto da ...
#
# Programma per la visualizzazione del contenuto di un file.
#

$mio_file = "";
$riga     = "";

print STDOUT ("Inserisci il nome del file da leggere: ");
$mio_file = <STDIN>;
chomp ($mio_file);

open (PRIMOFILE, "< $mio_file");

while ($riga = <PRIMOFILE>)
  {
    print STDOUT ($riga);
  }

close (PRIMOFILE);
```

3. La stringa che identifica il flusso di file può anche essere contenuta in una variabile.

```
#!/usr/bin/perl -w
#
```

```
# Programma file-03.pl
# Scritto da ...
#
# Programma per la visualizzazione del contenuto di un file.
#

$mio_file = "";
$riga     = "";
$flusso   = "";

print STDOUT ("Inserisci il nome del file da leggere: ");
$mio_file = <STDIN>;
chomp ($mio_file);

$flusso = "PRIMOFILE";
open ($flusso, "< $mio_file");

while ($riga = <$flusso>)
  {
    print STDOUT ($riga);
  }

close ($flusso);
```

4. Copia di un file.

```
#!/usr/bin/perl -w
#
# Programma file-04.pl
# Scritto da ...
#
# Programma per copiare un file.
#

$file_origine        = "";
$file_destinazione   = "";
$flusso_origine      = "";
$flusso_destinazione = "";
$riga                = "";

print STDOUT ("Inserisci il nome del file di origine: ");
$file_origine = <STDIN>;
chomp ($file_origine);
print STDOUT ("Inserisci il nome del file di destinazione: ");
$file_destinazione = <STDIN>;
chomp ($file_destinazione);

$flusso_origine      = "ORIGINE";
$flusso_destinazione = "DESTINAZIONE";

open ($flusso_origine, "< $file_origine");
open ($flusso_destinazione, "> $file_destinazione");

while ($riga = <$flusso_origine>)
  {
    print $flusso_destinazione ($riga);
  }

close ($flusso_destinazione);
close ($flusso_origine);
```

5. Copia di un file, utilizzando i nomi forniti come argomenti della riga di comando.

```
#!/usr/bin/perl -w
#
# Programma file-05.pl
# Scritto da ...
#
# Programma per copiare un file. Utilizza i nomi indicati nella
# riga di comando.
#
```

```
$file_origine        = "";
$file_destinazione   = "";
$flusso_origine      = "";
$flusso_destinazione = "";
$riga                = "";

$file_origine        = $ARGV[0];
$file_destinazione   = $ARGV[1];

$flusso_origine      = "ORIGINE";
$flusso_destinazione = "DESTINAZIONE";

open ($flusso_origine, "< $file_origine");
open ($flusso_destinazione, "> $file_destinazione");

while ($riga = <$flusso_origine>)
  {
    print $flusso_destinazione ($riga);
  }

close ($flusso_destinazione);
close ($flusso_origine);
```

Parte lviii

Java

Java: preparazione

Java è un linguaggio di programmazione realizzato da Sun Microsystems. Il suo scopo principale è l'inserzione di programmi all'interno di pagine HTML (applet), un po' come si fa con le immagini. Per questo motivo, il risultato della compilazione di un sorgente Java è una codifica intermedia, indipendente dalla piattaforma, che deve poi essere interpretata localmente dal navigatore *web* o da un altro programma indipendente.

In questo senso, Java potrebbe essere molto utile anche al di fuori della programmazione legata ai serventi HTTP, proprio per la portabilità dei suoi programmi.

Per programmare in Java occorre un compilatore, generalmente noto come '**`javac`**', che sia in grado di generare il formato binario Java, il cosiddetto Java bytecode. Il file che si ottiene non è propriamente un eseguibile, in quanto necessita di un interprete che generalmente è il programma '**`java`**'.

Esiste una versione ufficiale di questi strumenti, definita JDK (*Java development kit*), e almeno una versione indipendente per la maggior parte degli ambienti Unix (GNU/Linux incluso): Kaffe.

Nelle sezioni seguenti viene descritto in particolare come utilizzare Kaffe. Alla fine del capitolo si trova la descrizione dell'installazione e della configurazione di JDK originale, oltre a una sezione sull'uso di GCJ per la compilazione di sorgenti o binari Java nel formato eseguibile adatto alla propria architettura.

301.1 Kaffe

Kaffe [1] è un compilatore di sorgenti Java e un interprete di compilati in formato Java (Java bytecode). Attualmente, si tratta di un pacchetto standard delle distribuzioni GNU/Linux, per cui non ci dovrebbero essere problemi nella sua installazione. Attualmente, assieme al compilatore e all'interprete, dovrebbero essere disponibili anche le *classi*, ovvero le librerie Java.[2]

301.1.1 Classi

Le classi di Kaffe, che ormai accompagnano questo applicativo, dovrebbero essere contenute in un solo file compresso, che deve rimanere tale. Potrebbe trattarsi di '`/usr/share/kaffe/Klasses.jar`'.

301.1.2 Configurazione

Se si installa Kaffe autonomamente, senza affidarsi a un pacchetto già predisposto per la propria distribuzione GNU/Linux, potrebbe essere necessario definire alcune variabili di ambiente. Nell'esempio seguente si fa riferimento a uno script per una shell Bourne o derivata.

```
CLASSPATH=.:/usr/share/kaffe/Klasses.jar
KAFFEHOME=/usr/share/kaffe
LD_LIBRARY_PATH=/usr/lib:/usr/local/lib
export CLASSPATH
export KAFFEHOME
export LD_LIBRARY_PATH
```

[1] **Kaffe** licenza speciale

[2] In passato era necessario procurarsele a parte, dal momento che la versione libera realizzata appositamente per Kaffe non era stata ancora completata.

Se Kaffe fosse stato installato a partire dalla directory '`/usr/local/`', si dovrebbe usare la definizione seguente:

```
CLASSPATH=.:/usr/local/share/kaffe/Klasses.jar
KAFFEHOME=/usr/local/share/kaffe
LD_LIBRARY_PATH=/usr/lib:/usr/local/lib
export CLASSPATH
export KAFFEHOME
export LD_LIBRARY_PATH
```

Merita un po' di attenzione la variabile '**`LD_LIBRARY_PATH`**' che potrebbe essere utilizzata anche da altri programmi. '**`LD_LIBRARY_PATH`**' deve contenere i percorsi in cui si trovano i file di libreria; se il proprio sistema utilizza applicazioni che collocano le proprie librerie all'interno di directory inconsuete, queste devono essere aggiunte all'elenco. Segue un esempio esplicativo.

```
LD_LIBRARY_PATH=/usr/lib:/usr/local/lib:/opt/mio_prog/lib:/opt/tuo_prog/lib
```

301.1.3 Compilazione

Per verificare che la compilazione funzioni correttamente, basta preparare il solito programma banale che visualizza un messaggio attraverso lo standard output e poi termina.

```
class CiaoMondoApp
{
    public static void main (String[] args)
      {
        System.out.println ("Ciao Mondo!");
      }
}
```

Il file deve essere salvato con il nome '`CiaoMondoApp.java`'. Kaffe, tra le altre cose, fornisce un collegamento simbolico, denominato '**`javac`**', attraverso cui avviare la compilazione. Così la compilazione avviene nello stesso modo in cui si fa utilizzando gli strumenti del JDK originale.

```
$ javac CiaoMondoApp.java[ Invio ]
```

Se la sintassi del sorgente Java è corretta, si ottiene un file in formato binario Java, denominato '`CiaoMondoApp.class`'.

301.1.4 Esecuzione

Per eseguire il binario Java generato, ovvero il file '`.class`', occorre un interprete. In questo senso, il binario Java non ha bisogno necessariamente dei permessi di esecuzione, perché verrà solo letto dall'interprete.

```
$ kaffe CiaoMondoApp[ Invio ]

Ciao Mondo!
```

Come si può osservare dalla riga di comando, il file binario Java deve essere indicato senza l'estensione, che di conseguenza è obbligatoriamente '`.class`'. Kaffe si compone anche dello script '**`java`**', il cui scopo è quello di rendere il comando di interpretazione conforme al JDK; in pratica, '**`java`**' si limita ad avviare il comando '**`kaffe`**'.

```
$ java CiaoMondoApp
```

Tuttavia, questo script potrebbe essere modificato in modo da permettere l'avvio di un eseguibile Java anche se è stato fornito il nome del file corrispondente, completo di estensione '`.class`'. L'esempio seguente rappresenta le modifiche che potrebbero essere apportate in tal senso.

```
#! /bin/sh
#
# /usr/bin/java

CLASSE=‘/bin/basename $1 .class‘
shift
kaffe $CLASSE $@
```

301.2 Kernel

Come è noto, uno script viene interpretato automaticamente in base alla convenzione per cui la prima riga inizia con l’indicazione del programma adatto. Per esempio: ‘**#!/bin/sh**’, ‘**#!/bin/bash**’ e ‘**#!/usr/bin/perl**’. Con i binari Java ciò non è possibile, quindi, per ottenere l’avvio automatico dell’interprete ‘**java**’, occorre che il kernel ne sia informato. Per la precisione, occorre attivare la funzionalità generica di riconoscimento dei binari (sezione 29.2.4).

Questo comporta poi una configurazione per definire quali file devono essere riconosciuti e quali interpreti devono essere avviati di conseguenza. Nel caso dei binari Java normali, si tratta di eseguire il comando seguente (il percorso dell’interprete, ‘`/usr/bin/java`’ può essere cambiato a seconda delle proprie necessità).

```
# echo ':Java:M::\xca\xfe\xba\xbe::/usr/bin/java:' >
/proc/sys/fs/binfmt_misc/register
```

In alternativa, se si è sicuri dell’estensione ‘`.class`’, si può utilizzare il comando seguente:

```
# echo ':Java:E::class::/usr/bin/java:' > /proc/sys/fs/binfmt_misc/register
```

Per verificare che la definizione sia stata recepita correttamente dal kernel, si può leggere il contenuto del file virtuale ‘`/proc/sys/fs/binfmt_misc/Java`’, creato a seguito di uno dei due comandi mostrati sopra.

Quando il kernel è predisposto nel modo appena visto, si possono rendere eseguibili i file binari Java; così, quando si tenta di avviarli, il kernel stesso avvia invece il comando seguente:

```
java file_binario_java argomenti
```

Lo svantaggio di questo sistema sta nel fatto che il nome del file binario Java viene indicato con tutta l’estensione, cosa che normalmente crea dei problemi, sia a Kaffe che al JDK. Per questo, conviene che ‘`/usr/bin/java`’ sia uno script predisposto per risolvere il problema, come già mostrato nella sezione precedente.

Se invece di usare Kaffe si usa il JDK originale, conviene modificare il nome dell’interprete Java, per esempio in ‘**java1**’, realizzando poi un file script analogo a quello già visto.

```
#! /bin/sh
#
# /usr/bin/java

CLASSE=‘/bin/basename $1 .class‘
shift
java1 $CLASSE $@
```

C’è però una cosa che occorre tenere a mente. Con GNU/Linux, così come con altri sistemi, non è possibile avviare un eseguibile se il nome non viene indicato per esteso. In pratica, non è pensabile che succeda quanto accade in Dos in cui i file che finiscono per ‘`.COM`’ o ‘`.EXE`’ sono avviati semplicemente nominandoli senza estensione.

Per chi ha usato GNU/Linux da un po’ di tempo ciò dovrebbe essere logico, ma con Java si rischia ancora di essere ingannati: il fatto che, sia l’interprete ‘**java**’ originale, sia ‘**kaffe**’, vogliano il

nome dell'eseguibile Java senza l'estensione '`.class`', non deve fare supporre che ciò valga anche per il kernel. Per cui, se si avvia '`CiaoMondoApp.class`' nel modo seguente,

```
$ java CiaoMondoApp
```

quando si vuole che sia il kernel a fare tutto questo per noi, il comando sarà il seguente:

```
$ CiaoMondoApp.class
```

Se si tentasse si eseguire il comando seguente,

```
$ CiaoMondoApp
```

si otterrebbe una segnalazione di errore del tipo: '**`command not found`**'.

301.3 Applet

Un'applet Java è un programma particolare che può essere incorporato in un documento HTML. Il meccanismo è simile all'inserzione di immagini; l'effetto è quello di un programma grafico che, invece di utilizzare una finestra si inserisce in un'area prestabilita del documento HTML. Un'applet Java non può quindi vivere da solo, richiede sempre l'abbinamento a una pagina HTML.

Il modo migliore per vedere il funzionamento di un programma del genere è attraverso l'utilizzo di un navigatore in grado di eseguire tali applet, per esempio Netscape.

301.3.1 Verifica del funzionamento

Per verificare il funzionamento di un'applet si può provare il solito programma banale. In questo caso si comincia con la realizzazione di una pagina HTML che incorpori l'applet che si vuole realizzare.

```
<!-- CiaoMondo.html -->
<HTML>
<HEAD>
        <TITLE>La mia prima applet</TITLE>
</HEAD>

<BODY>

<OBJECT CODETYPE="application/java"
    CLASSID="java:CiaoMondo.class"
    WIDTH=150
    HEIGHT=25>
Applet Java
</OBJECT>

</BODY>
</HTML>
```

Come si vede, l'elemento '**`OBJECT`**' dichiara l'utilizzo dell'applet '`CiaoMondo.class`' che si collocherà nello spazio di un rettangolo di 150 per 25 punti grafici (pixel). Segue il sorgente dell'applet.

```
// CiaoMondo.java

import java.applet.Applet;
import java.awt.Graphics;

public class CiaoMondo extends Applet
```

```
{
    public void paint (Graphics g)
    {
        g.drawString ("Ciao Mondo!", 50, 25);
    }
}
```

Si compila il sorgente ‘`CiaoMondo.java`’ nel solito modo, ottenendo il binario Java ‘`CiaoMondo.class`’

```
$ javac CiaoMondo.java
```

Quando si carica il file ‘`CiaoMondo.html`’ attraverso un navigatore adatto, incontrando l’elemento ‘**`OBJECT`**’ che fa riferimento al binario Java ‘**`CiaoMondo.class`**’, viene caricato il programma ‘`CiaoMondo.class`’ nell’area stabilita.

All’interno di quell’area, a partire dall’angolo superiore sinistro, vengono calcolate le coordinate (x=50, y=25) dell’istruzione ‘**`g.drawString("Ciao mondo!", 50, 25)`**’ vista nell’applet.

301.4 JDK

JDK [3] è il pacchetto originale per la compilazione e l’esecuzione di applicativi Java. Viene distribuito in forma binaria, già compilata. Per ottenerlo, si può consultare *<http://www.blackdown.org/>* o eventualmente si può fare una ricerca attraverso *<http://www.alltheweb.com/?c=ftp>* per i file contenenti la stringa ‘`linux-jdk`’ (si potrebbero trovare nomi come ‘`linux-jdk.1.1.3-v2.tar.gz`’). Se si desidera installare il JDK è importante verificare di non avere tracce di Kaffe.

Il JDK può essere installato a partire da qualunque punto del proprio file system. Qui viene proposta l’installazione a partire da ‘`/opt/`’.

Se nel proprio sistema non è presente, la si può creare, quindi al suo interno si può espandere il contenuto del pacchetto JDK. Si ottiene così la directory ‘`jdk`*versione*`/`’, per esempio ‘`jdk1.1.3/`’. Per motivi pratici è opportuno modificare il nome della directory, o creare un collegamento simbolico, in modo che vi si possa accedere utilizzando il nome ‘`/opt/java/`’.

Prima di poter funzionare, il JDK deve essere configurato attraverso delle variabili di ambiente opportune. Nell’esempio seguente si mostra un pezzo di script per una shell Bourne o derivata, in grado di predisporre le variabili necessarie.

```
PATH="/opt/java/bin:$PATH"
CLASSPATH=.:/opt/java/lib/classes.zip:/opt/java/lib/classes
JAVA_HOME=/opt/java
export PATH
export CLASSPATH
export JAVA_HOME
```

Per il funzionamento si può rivedere quanto già indicato per Kaffe. In questo caso, utilizzando il JDK originale, il compilatore è proprio ‘**`javac`**’ e l’esecutore (o interprete) è ‘**`java`**’.

301.5 GCJ

GCJ [4] è un programma frontale per il controllo del compilatore GCC e di altri programmi accessori, il cui scopo è quello di compilare sorgenti Java.

[3] **JDK** software non libero

[4] **GCJ** GNU GPL

La compilazione può avvenire a diversi livelli: da sorgenti Java (‘`.java`’) o da binari Java (‘`.class`’) si può arrivare a un file eseguibile per il proprio sistema operativo; in alternativa si possono semplicemente compilare dei sorgenti Java per generare i binari Java corrispondenti (‘`.class`’). Semplificando le cose, si possono distinguere questi tre tipi di comandi per la compilazione:

- `gcj -C` *file_sorgente_java …*

  per generare binari Java (file ‘`.class`’);

- `gcj --main=`*classe_principale* `-o` *file_da_generare file_sorgente_java …*

  per generare un eseguibile a partire da dei sorgenti Java (file ‘`.java`’);

- `gcj --main=`*classe_principale* `-o` *file_da_generare binario_java …*

  per generare un eseguibile a partire da binari Java (file ‘`.class`’).

Supponendo di avere il solito esempio già visto in precedenza,

```
class CiaoMondoApp
{
    public static void main (String[] args)
    {
        System.out.println ("Ciao Mondo!");
    }
}
```

supponendo questa volta che sia contenuto nel file ‘`ciao_mondo.java`’, si può generare il binario Java ‘`CiaoMondoApp.class`’ con il comando seguente:

```
$ gcj -C ciao_mondo.java
```

Per compilare il binario Java in modo da ottenere un binario adatto al sistema operativo e all’architettura del proprio elaboratore, si può usare il comando seguente, generando quindi l’eseguibile ‘**`ciao`**’:

```
$ gcj --main=CiaoMondoApp -o ciao CiaoMondoApp.class
```

Infine, per compilare direttamente il sorgente Java, si può agire nello stesso modo:

```
$ gcj --main=CiaoMondoApp -o ciao ciao_mondo.java
```

GCJ riconosce la variabile di ambiente ‘**`CLASSPATH`**’, per la ricerca delle classi, fornendo anche la possibilità di indicare tale informazione attraverso la riga di comando, con delle opzioni che qui non vengono mostrate.

Alcune opzioni

`-C`

In questo caso, i file in ingresso sono sorgenti Java e vengono compilati generando le classi in forma di binari Java.

`--main=`*classe*

Questa opzione permette di stabilire quale sia la classe da utilizzare come principale, in modo che il programma che si genera inizi da lì il suo funzionamento.

`-o` *file*

Definisce il nome dell’eseguibile da generare, quando la compilazione non è destinata a ottenere soltanto un binario Java.

301.6 Riferimenti

- *TransVirtual Technologies Inc.*

  <http://www.transvirtual.com>

- Riferimenti per ottenere il JDK dalla rete

  <http://www.blackdown.org/>

- *The source for Java, Documentation*

  <http://java.sun.com/docs/index.html>

- *The source for Java, Tutorial*

  <http://java.sun.com/docs/books/tutorial/index.html>

Java: introduzione

Il capitolo precedente (301) ha già descritto cosa sia Java e in che modo si possa utilizzare in un sistema GNU/Linux. Questo capitolo vuole introdurre alla programmazione in Java, in modo superficiale, per dare un'idea più chiara delle potenzialità di questo linguaggio.

302.1 Struttura fondamentale

Java è un linguaggio di programmazione strettamente OO (*Object oriented*), cioè a dire che qualunque cosa si faccia, anche un semplice programma che emette un messaggio attraverso lo standard output, va trattato secondo la programmazione a oggetti.

Ciò significa anche che i componenti di questo linguaggio hanno nomi diversi da quelli consueti. Volendo fare un abbinamento approssimativo con un linguaggio di programmazione normale, si potrebbe dire che in Java i programmi sono *classi* e le funzioni sono *metodi*. Naturalmente ci sono anche tante altre cose nuove.

Fatta questa premessa, si può dare un'occhiata alla solita classe banale: quella che visualizza un messaggio e termina.

```
/**
 *  CiaoMondoApp.java
 *  La solita classe banale.
 */

import java.lang.*; // predefinita

class CiaoMondoApp
{
    public static void main (String[] args)
    {
        System.out.println ("Ciao Mondo!"); // visualizza il messaggio
    }
}
```

Il sorgente Java ha molte somiglianze con quello del linguaggio C e qui si intendono segnalare le particolarità rispetto a quel linguaggio.

302.1.1 Commenti

Java ammette l'uso di commenti in stile C, nella solita forma '**`/*`**…**`*/`**', ma ne introduce altri due tipi: uno per la creazione automatica di documentazione, nella forma '**`/**`**…**`*/`**', e uno per fare ignorare tutto ciò che appare a partire dal simbolo di commento fino alla fine della riga, nella forma '**`//`***commento* '.

`/*` *commento_generico* `*/`
`/**` *documentazione* `*/`
`//` *commento_fino_alla_fine_della_riga*

Tutti e tre questi tipi di commenti servono a fare ignorare al compilatore una parte del sorgente e questo dovrebbe bastare al principiante. Convenzionalmente, è conveniente usare il commento di documentazione per la spiegazione di ciò che fa la classe, all'inizio del sorgente.

302.1.2 Nomi ed estensioni

Le estensioni dei file Java sono definite in modo obbligatorio: '`.java`' per i sorgenti e '`.class`' per le classi (i binari Java).

Generalmente, nel sorgente, il nome della classe deve corrispondere alla radice del nome del sorgente e, di conseguenza, anche del binario Java. Per lo stile convenzionale di Java, questo nome inizia con una lettera maiuscola e non contiene simboli strani; se è composto dall'unione di più parole, ognuna di queste inizia con una lettera maiuscola.

302.1.3 Istruzioni

Le istruzioni seguono la convenzione del linguaggio C, per cui terminano con un punto e virgola ('**`;`**') e i raggruppamenti di queste, detti anche blocchi, si fanno utilizzando le parentesi graffe ('**`{ }`**').

istruzione `;`
`{`*istruzione* `;` *istruzione* `;` *istruzione* `; }`

Generalmente, un'istruzione può essere interrotta e ripresa nella riga successiva, dal momento che la sua conclusione è dichiarata chiaramente dal punto e virgola finale.

302.1.4 Librerie di classi

Ogni programma in Java deve fare affidamento sull'utilizzo di classi fondamentali che compongono il linguaggio stesso. L'importazione delle classi necessarie viene fatta attraverso l'istruzione '**`import`**', indicando una classe particolare o un gruppo (nel secondo caso si usa un asterisco).

Nell'esempio introduttivo vengono importate tutte le classi del pacchetto '**`java.lang`**', anche se non sarebbe stato necessario dichiararlo, dato che queste classi vengono sempre importate in modo predefinito (senza di queste, nessuna classe potrebbe funzionare).

Le classi standard di Java (cioè queste librerie fondamentali), sono contenute normalmente in un archivio compresso '`.zip`', oppure '`.jar`'. Si è visto nel capitolo 301 che è importante indicare il percorso in cui si trovano, nella variabile di ambiente '**`CLASSPATH`**'.

Osservando il contenuto di questo file, si può comprendere meglio il concetto di pacchetto di classi. Segue solo un breve estratto.

```
Archive:  classes.zip
 Length    Date    Time    Name
 ------    ----    ----    ----
      0  05-19-97  22:46   java/
      0  05-19-97  22:24   java/lang/
   1322  05-19-97  22:24   java/lang/Object.class
   4202  05-19-97  22:24   java/lang/Class.class
...
   3450  05-19-97  22:24   java/lang/System.class
...
      0  05-19-97  22:26   java/util/
...
      0  05-19-97  22:26   java/io/
...
      0  05-19-97  22:42   java/awt/
...
```

Ecco che così può diventare più chiaro il fatto che, importare tutte le classi del pacchetto '**`java.lang`**' significa in pratica includere tutte le classi contenute nella directory '`java/lang/`', anche se qui si tratta solo di un file compresso.

302.1.5 Dichiarazione della classe

Generalmente, un file sorgente Java contiene la dichiarazione di una sola classe, il cui nome corrisponde alla radice del file sorgente. La dichiarazione della classe delimita in pratica il contenuto del sorgente, definendo eventuali *ereditarietà* da altre classi esistenti.

Quando una classe non eredita da un'altra, si parla convenzionalmente di *applicazione*, mentre quando eredita dalla classe '**`java.applet.Applet`**' (cioè da '`java/applet/Applet.class`') si usa la definizione *applet*.

302.1.6 Contenuto della classe

La classe contiene essenzialmente dichiarazioni di variabili e metodi. L'esecuzione di un metodo dipende da una chiamata, detta anche *messaggio*. Perché una classe si traduca in un programma autonomo, occorre che al suo interno ci sia un metodo che viene eseguito in modo automatico all'avvio.

Nel caso delle classi che non ereditano nulla da altre, come nell'esempio, ci deve essere il metodo '**`main`**' che viene eseguito all'avvio del binario Java contenente la classe stessa. Quando una classe eredita da un'altra, queste regole sono stabilite dalla classe ereditata.

Il metodo '**`main`**' è formato necessariamente come nell'esempio: '**`public static void main(String[] args) {...}`**'.

302.1.7 Variabili e tipi di dati

In Java si distinguono fondamentalmente due tipi di rappresentazione dei dati: primitivi e riferimenti a oggetti. I tipi di dati primitivi sono per esempio i soliti tipi numerici (intero, a virgola mobile, ecc.); gli altri sono *oggetti*. Un oggetto è quindi una variabile contenente un riferimento a una struttura, più o meno complessa. In Java, gli array e le stringhe sono oggetti; pertanto non esistono tipi di dati primitivi equivalenti.

I nomi delle variabili possono essere composti utilizzando caratteri Unicode, tuttavia, si può anche utilizzare semplicemente la codifica ISO 8859-1, essendo questa compatibile con Unicode, così da poter utilizzare anche le lettere accentate, se ciò può essere utile per la leggibilità del sorgente stesso. Naturalmente, non è possibile utilizzare nomi coincidenti con parole chiave già utilizzate dal linguaggio stesso. La convenzione stilistica di Java richiede che il nome delle variabili inizi con la lettera minuscola; inoltre, se si tratta di un nome composto, la convenzione richiede di segnalare l'inizio di ogni nuova parola con una lettera maiuscola. Per esempio: '**`miaVariabile`**', '**`dataOdierna`**', '**`elencoNomiFemminili`**'.

302.1.8 Chiamata per valore

In Java, le chiamate dei metodi avvengono trasferendo il valore degli argomenti indicati nella chiamata stessa. Ciò significa che le modifiche che si dovessero apportare all'interno dei metodi non si riflettono all'indietro. Tuttavia, questo ragionamento vale solo per i tipi di dati primitivi, dal momento che quando si utilizzano degli oggetti, essendo questi dei riferimenti, le variazioni fatte al loro interno rimangono anche dopo la chiamata.

302.2 Variabili e tipi di dati

Si è già accennato al fatto che Java distingue tra due tipi di dati, primitivi e riferimenti a oggetti (o più semplicemente solo oggetti). L'esempio seguente mostra la dichiarazione di un intero all'interno di un metodo e il suo incremento fino a raggiungere un valore predefinito.

```
/**
 *  DieciXApp.java
 *  Un esempio di utilizzo delle variabili.
 */

import java.lang.*; // predefinita

class DieciXApp
{
    public static void main (String[] args)
    {
        int contatore = 0;

        // Inizia un ciclo in cui si emettono 10 «x» attraverso lo
        // standard output.
        while (contatore < 10)
          {
            contatore++;
            System.out.println ("x"); // emette una «x»
          }
    }
}
```

302.2.1 Tipi

I tipi di dati primitivi rappresentano un valore singolo. Il loro elenco si trova nella tabella 302.1.

Tabella 302.1. Elenco dei tipi di dati primitivi in Java.

Tipo	Dimensione	Descrizione
byte	8 bit, complemento a due.	Intero a 8 bit.
short	16 bit, complemento a due.	Intero ridotto.
int	32 bit, complemento a due.	Intero normale.
long	64 bit, complemento a due.	Intero molto grande.
float	32 bit	Virgola mobile, singola precisione.
double	64 bit	Virgola mobile, doppia precisione.
char	16 bit, carattere Unicode.	Carattere.
boolean	*Vero* o *Falso*.	Valore booleano.

Nell'esempio mostrato precedentemente, viene dichiarato un intero normale, '**`contatore`**', inizializzato al valore zero, che poi viene incrementato all'interno di un ciclo.

```
int contatore = 0;

// Inizia un ciclo in cui si emettono 10 «x» attraverso lo
// standard output.
while (contatore < 10)
{
    contatore++;
    System.out.println ("x"); // emette una «x»
}
```

302.2.2 Costanti

Ogni tipo primitivo ha la possibilità di essere rappresentato in forma di costante letterale. La tabella 302.2 mostra l'elenco dei tipi di dati abbinati alla rappresentazione in forma di costante letterale.

Tabella 302.2. Elenco dei tipi di dati primitivi abbinati a una possibile rappresentazione in forma di costante letterale.

Tipo	Esempio di costante	Descrizione o intervallo
byte	123	-128..+127
short	12345	-32768..+32767
int	1234567890	$-(2^{31})..+((2^{31})-1)$
long	12345678901234567890	$-(2^{63})..+((2^{63})-1)$
float	(float)123.456	La costante con virgola è sempre a doppia precisione.
double	123.456	
char	'A'	Si usano gli apici semplici.
boolean	true	Si usano le parole chiave '**`true`**' e '**`false`**'.

È importante osservare che una costante numerica a virgola mobile è sempre a doppia precisione, per cui, se si vuole assegnare a una variabile a singola precisione ('**`float`**') una costante letterale, occorre una conversione di tipo, per mezzo di un cast. Si vedrà in seguito che le stringhe si delimitano utilizzando gli apici doppi. Per ora è solo il caso ti tenere in considerazione che in Java le stringhe non sono tipi di dati primitivi, ma oggetti veri e propri.

302.2.3 Campo di azione

Il campo di azione delle variabili in Java viene determinato dalla posizione in cui queste vengono dichiarate. Ciò determina il momento della loro creazione e distruzione. A fianco del concetto del campo di azione, si pone quello della *protezione*, che può limitare l'accessibilità di una variabile. La protezione verrà analizzata in seguito.

A seconda del loro campo di azione, si distinguono in particolare tre categorie più importanti di variabili: variabili appartenenti alla classe (*member variable*), variabili locali e parametri dei metodi.

Variabili appartenenti alla classe

Queste variabili appartengono alle classi e come tali sono dichiarate all'interno delle classi stesse, ma all'esterno dei metodi. L'esempio seguente mostra la dichiarazione della variabile '**`serveAQualcosa`**' come parte della classe '**`FaQualcosa`**'.

```
class FaQualcosa
{
    int serveAQualcosa = 0;

    // Dichiarazione dei metodi
    ...
}
```

Variabili locali

Sono variabili dichiarate all'interno dei metodi. Vengono create alla chiamata del metodo e distrutte alla sua conclusione. Per questo sono visibili solo all'interno del metodo che le dichiara.

Nell'esempio visto in precedenza, quello che visualizza 10 «x», la variabile '**`contatore`**' veniva dichiarata all'interno del metodo '**`main`**'.

Parametri dei metodi

Le variabili indicate in concomitanza con la dichiarazione di un metodo (quelle che appaiono tra parentesi tonde), vengono create nel momento della chiamata del metodo stesso e distrutte alla sua conclusione. Queste variabili contengono la copia degli argomenti utilizzati per la chiamata; in questo senso si dice che le chiamate ai metodi avvengono per valore.

302.2.4 Operatori

Gli operatori sono qualcosa che esegue un qualche tipo di funzione, su uno o due operandi, restituendo un valore. Il valore restituito è di tipo diverso a seconda degli operandi utilizzati. Per esempio, la somma di due interi genera un risultato intero.

Gli operandi descritti nelle sezioni seguenti sono solo quelli più comuni e importanti. In particolare, sono stati omessi quelli necessari al trattamento delle variabili in modo binario.

302.2.4.1 Operatori aritmetici

Gli operatori che intervengono su valori numerici sono elencati nella tabella 302.3.

Tabella 302.3. Elenco degli operatori aritmetici e di quelli di assegnamento relativi a valori numerici.

Operatore e operandi	Descrizione
++*op*	Incrementa di un'unità l'operando prima che venga restituito il suo valore.
op++	Incrementa di un'unità l'operando dopo averne restituito il suo valore.
--*op*	Decrementa di un'unità l'operando prima che venga restituito il suo valore.
op--	Decrementa di un'unità l'operando dopo averne restituito il suo valore.
+*op*	Non ha alcun effetto.
-*op*	Inverte il segno dell'operando.
op1 + *op2*	Somma i due operandi.
op1 - *op2*	Sottrae dal primo il secondo operando.
op1 * *op2*	Moltiplica i due operandi.
op1 / *op2*	Divide il primo operando per il secondo.
op1 % *op2*	Modulo -- il resto della divisione tra il primo e il secondo operando.
var = *valore*	Assegna alla variabile il valore alla destra.
op1 += *op2*	*op1* = *op1* + *op2*
op1 -= *op2*	*op1* = *op1* - *op2*
op1 *= *op2*	*op1* = *op1* * *op2*
op1 /= *op2*	*op1* = *op1* / *op2*
op1 %= *op2*	*op1* = *op1* % *op2*

302.2.4.2 Operatori di confronto e operatori logici

Gli operatori di confronto determinano la relazione tra due operandi. Il risultato dell'espressione composta da due operandi posti a confronto è di tipo booleano, rappresentabile in Java dalle costanti letterali '**`true`**' e '**`false`**'. Gli operatori di confronto sono elencati nella tabella 302.4.

Tabella 302.4. Elenco degli operatori di confronto. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
op1 == *op2*	*Vero* se gli operandi si equivalgono.
op1 != *op2*	*Vero* se gli operandi sono differenti.
op1 < *op2*	*Vero* se il primo operando è minore del secondo.
op1 > *op2*	*Vero* se il primo operando è maggiore del secondo.
op1 <= *op2*	*Vero* se il primo operando è minore o uguale al secondo.
op1 >= *op2*	*Vero* se il primo operando è maggiore o uguale al secondo.

Quando si vogliono combinare assieme diverse espressioni logiche, comprendendo in queste anche delle variabili che contengono un valore booleano, si utilizzano gli operatori logici (noti normalmente come: AND, OR, NOT, ecc.). Il risultato di un'espressione logica complessa è quello dell'ultima espressione elementare che sia stata valutata effettivamente. Gli operatori logici sono elencati nella tabella 302.5.

Tabella 302.5. Elenco degli operatori logici. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
! *op*	Inverte il risultato logico dell'operando.
op1 && *op2*	Se il risultato del primo operando è *Falso* non valuta il secondo.
op1 || *op2*	Se il risultato del primo operando è *Vero* non valuta il secondo.

302.2.4.3 Concatenamento di stringhe

Si è accennato al fatto che in Java, le stringhe siano oggetti e non tipi di dati primitivi. Esiste tuttavia la possibilità di indicare stringhe letterali nel modo consueto, attraverso la delimitazione con gli apici doppi.

Diverse stringhe possono essere concatenate, in modo da formare una stringa unica, attraverso l'operatore '**+**'.

```
...
    public static void main (String[] args)
    {
        int contatore = 0;

        while (contatore < 10)
          {
            contatore++;
            System.out.println ("Ciclo n. " + contatore);
          }
    }
```

Nel pezzo di codice appena mostrato, appare in particolare l'istruzione

```
            System.out.println ("Ciclo n. " + contatore);
```

in cui l'espressione '**`"Ciclo n. " + contatore`**' si traduce nel risultato seguente:

```
Ciclo n. 1
Ciclo n. 2
...
Ciclo n. 10
```

In pratica, il contenuto della variabile '**`contatore`**' viene convertito automaticamente in stringa e unito alla costante letterale precedente.

302.3 Strutture di controllo del flusso

Le strutture di controllo del flusso delle istruzioni sono molto simili a quelle del linguaggio C. In particolare, dove può essere messa un'istruzione si può mettere anche un gruppo di istruzioni delimitate dalle parentesi graffe.

Normalmente, le strutture di controllo del flusso basano questo controllo sulla verifica di una condizione espressa all'interno di parentesi tonde.

302.3.1 if

`if (`*condizione*`)` *istruzione*
`if (`*condizione*`)` *istruzione* `else` *istruzione*

Se la condizione si verifica, viene eseguita l'istruzione (o il gruppo di istruzioni) seguente; quindi il controllo passa alle istruzioni successive alla struttura. Se viene utilizzato '**`else`**', nel caso non si verifichi la condizione, viene eseguita l'istruzione che ne segue. Vengono mostrati alcuni esempi.

```
int importo;
...
if (importo > 10000000) System.out.println ("L'offerta è vantaggiosa");
```

```
int importo;
int memorizza;
...
if (importo > 10000000)
  {
    memorizza = importo;
    System.out.println ("L'offerta è vantaggiosa");
  }
else
  {
    System.out.println ("Lascia perdere");
  }
```

```
int importo;
int memorizza;
...
if (importo > 10000000)
  {
    memorizza = importo;
    System.out.println ("L'offerta è vantaggiosa");
  }
else if (importo > 5000000)
  {
    memorizza = importo;
    System.out.println ("L'offerta è accettabile");
  }
else
  {
    System.out.println ("Lascia perdere");
  }
```

302.3.2 switch

L'istruzione '**switch**' è un po' troppo complessa per essere rappresentata in modo chiaro attraverso uno schema sintattico. In generale, l'istruzione '**switch**' permette di eseguire una o più istruzioni in base al risultato di un'espressione. L'esempio seguente mostra la visualizzazione del nome del mese, in base al valore di un intero.

```
int mese;
...
switch (mese)
  {
    case 1: System.out.println ("gennaio"); break;
    case 2: System.out.println ("febbraio"); break;
    case 3: System.out.println ("marzo"); break;
    case 4: System.out.println ("aprile"); break;
    case 5: System.out.println ("maggio"); break;
    case 6: System.out.println ("giugno"); break;
    case 7: System.out.println ("luglio"); break;
    case 8: System.out.println ("agosto"); break;
    case 9: System.out.println ("settembre"); break;
    case 10: System.out.println ("ottobre"); break;
    case 11: System.out.println ("novembre"); break;
    case 12: System.out.println ("dicembre"); break;
  }
```

Come si vede, dopo l'istruzione con cui si emette il nome del mese attraverso lo standard output, viene aggiunta un'istruzione di salto '**break**', che serve a evitare la verifica degli altri casi. Un gruppo di casi può essere raggruppato assieme, quando si vuole che questi eseguano lo stesso gruppo di istruzioni.

```
int mese;
int giorni;
...
switch (mese)
  {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        giorni = 31;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        giorni = 30;
        break;
    case 2:
        if (((anno % 4 == 0) && !(anno % 100 = 0))
            || (anno % 400 == 0))
            giorni = 29;
        else
            giorni = 28;
        break;
  }
```

È anche possibile definire un caso predefinito che si verifichi quando nessuno degli altri si avvera.

```
int mese;
...
switch (mese)
  {
```

```
      case 1: System.out.println ("gennaio"); break;
      case 2: System.out.println ("febbraio"); break;
      ...
      case 11: System.out.println ("novembre"); break;
      case 12: System.out.println ("dicembre"); break;
      default: System.out.println ("mese non corretto"); break;
  }
```

302.3.3 while

`while (`*condizione*`)` *istruzione*

'**`while`**' esegue un'istruzione, o un gruppo di queste, finché la condizione restituisce il valore *Vero*. La condizione viene valutata prima di eseguire il gruppo di istruzioni e poi ogni volta che termina un ciclo, prima dell'esecuzione del successivo. Segue il pezzo dell'esempio già visto, di quella classe che visualizza 10 volte la lettera «x».

```
int contatore = 0;

while (contatore < 10)
  {
    contatore++;
    System.out.println ("x");
  }
```

Nel blocco di istruzioni di un ciclo '**`while`**', ne possono apparire alcune particolari:

- '**`break`**'

  esce definitivamente dal ciclo '**`while`**';

- '**`continue`**'

  interrompe l'esecuzione del gruppo di istruzioni e riprende dalla valutazione della condizione.

L'esempio seguente è una variante del ciclo di visualizzazione mostrato sopra, modificato in modo da vedere il funzionamento di '**`break`**'. '**`while (true)`**' equivale a un ciclo senza fine, perché la condizione è sempre vera.

```
int contatore = 0;

while (true)
  {
    if (contatore >= 10)
      {
        break;
      }
    contatore++;
    System.out.println ("x");
  }
```

302.3.4 do-while

`do` *blocco_di_istruzioni* `while (`*condizione*`);`

'**`do`**' esegue un gruppo di istruzioni una volta e poi ne ripete l'esecuzione finché la condizione restituisce il valore *Vero*.

302.3.5 for

```
for (espressione1; espressione2; espressione3) istruzione
```

Questa è la forma tipica di un'istruzione '**for**', in cui la prima espressione corrisponde all'assegnamento iniziale di una variabile, la seconda a una condizione che deve verificarsi fino a che si vuole che sia eseguita l'istruzione (o il gruppo di istruzioni), mentre la terza serve per l'incremento o decremento della variabile inizializzata con la prima espressione. In pratica, potrebbe esprimersi nella sintassi seguente:

```
for (var = n; condizione; var++) istruzione
```

Il ciclo '**for**' potrebbe essere definito anche in maniera differente, più generale: la prima espressione viene eseguita una volta sola all'inizio del ciclo; la seconda viene valutata all'inizio di ogni ciclo e il gruppo di istruzioni viene eseguito solo se il risultato è *Vero*; l'ultima viene eseguita alla fine dell'esecuzione del gruppo di istruzioni, prima che si ricominci con l'analisi della condizione.

Il vecchio esempio banale, in cui veniva visualizzata per 10 volte una «x», potrebbe tradursi nel modo seguente, attraverso l'uso di un ciclo '**for**'.

```
int contatore;

for (contatore = 0; contatore < 10; contatore++)
  {
    System.out.println ("x");
  }
```

302.4 Array e stringhe

In Java, array e stringhe sono oggetti. In pratica, la variabile che contiene un array o una stringa è in realtà un riferimento alla struttura di dati rispettiva.

302.4.1 Array

La dichiarazione di un array avviene in Java in modo molto semplice, senza l'indicazione esplicita del numero di elementi. La dichiarazione avviene come se si trattasse di un tipo di dati normale, con la differenza che si aggiungono una coppia di parentesi quadre a sottolineare che si tratta di un array di elementi di quel tipo. Per esempio,

```
int[] arrayDiInteri;
```

dichiara che '**arrayDiInteri**' è un array in cui gli elementi sono di tipo intero ('**int**'), senza specificare quanti siano.

Per fare in modo che l'array esista effettivamente, occorre che questo sia inizializzato, fornendogli gli elementi. Si usa per questo l'operatore '**new**' seguito dal tipo di dati con il numero di elementi racchiuso tra parentesi quadre. Per esempio,

```
arrayDiInteri = new int[7];
```

assegna alla variabile '**arrayDiInteri**' il riferimento a un array composto da sette interi. Nella pratica, è normale inizializzare l'array quando lo si dichiara; per cui, quanto già visto si può ridurre all'esempio seguente:

```
int[] arrayDiInteri = new int[7];
```

Il riferimento a un elemento di un array avviene aggiungendo al nome della variabile che rappresenta l'array stesso, il numero dell'elemento, racchiuso tra parentesi quadre. Come nel linguaggio

C, il primo elemento si raggiunge con l'indice zero, mentre l'ultimo corrisponde alla dimensione meno uno.

Si è detto che gli array sono oggetti. In particolare, è possibile determinare la dimensione di un array, espressa in numero di elementi, leggendo il contenuto della variabile '**`length`**' dell'oggetto array. Nel caso dell'esempio già visto, si tratta di leggere il contenuto di '**`arrayDiInteri.length`**'.

L'esempio seguente mostra una scansione di un array, indicando una condizione di interruzione del ciclo indipendente dalla conoscenza anticipata della dimensione dell'array stesso. In particolare, la variabile '**`i`**' viene dichiarata contestualmente con la sua inizializzazione, nella prima espressione di controllo del ciclo '**`for`**'.

```
for (int i = 0; i < arrayDiInteri.length; i++) {
    arrayDiInteri[i] = i;
}
```

Un array può contenere sia elementi primitivi che riferimenti a oggetti. In questo modo si possono avere gli array multidimensionali. L'esempio seguente rappresenta il modo in cui può essere definito un array 3x2 di interi e anche come scandirne i vari elementi.

```
/**
 *      Matrice3x2App.java
 *      Esempio di uso di array multidimensionali.
 */

import java.lang.*; // predefinita

class Matrice3x2App
{
    public static void main (String[] args)
    {
        int[][] matrice = new int[3][2];

        for (int i = 0; i < matrice.length; i++)
          {
            for (int j = 0; j < matrice[i].length; j++)
              {
                matrice[i][j] = 1000 + j + i * 10;
                System.out.println ("matrice[" + i + "][" + j + "] = "
                                    + matrice[i][j]);
              }
          }
    }
}
```

L'esecuzione di questo piccolo programma, genera il risultato seguente:

```
matrice[0][0] = 1000
matrice[0][1] = 1001
matrice[1][0] = 1010
matrice[1][1] = 1011
matrice[2][0] = 1020
matrice[2][1] = 1021
```

302.4.2 Stringhe

Le stringhe in Java sono oggetti e se ne distinguono due tipi: stringhe costanti e stringhe variabili. La distinzione è utile perché questi due tipi di oggetti hanno bisogno di una forma di rappresentazione diversa. Così, ciò porta a un'ottimizzazione del programma, che per una stringa costante richiede meno risorse rispetto a una stringa che deve essere variabile, oltre a migliorare altri aspetti legati alla sicurezza.

La dichiarazione di una variabile che possa contente un riferimento a un oggetto stringa-costante, si ottiene con la dichiarazione seguente:

```
String variabile;
```

In pratica, si dichiara che la variabile può contenere un riferimento a un oggetto di tipo '**`String`**'. La creazione di questo oggetto '**`String`**' si ottiene come nel caso degli array, utilizzando l'operatore '**`new`**'.

```
new String (stringa);
```

L'esempio seguente crea la variabile '**`stringaCostante`**' di tipo '**`String`**' e la inizializza assegnandoci il riferimento a una stringa.

```
String stringaCostante = new String ("Ciao ciao.");
```

Fortunatamente, si possono utilizzare anche delle costanti letterali pure e semplici. Per cui la stringa '**`"Ciao ciao."`**' è già di per sé un oggetto stringa-costante.

Si è già accennato al fatto che le stringhe-costanti possono essere concatenate facilmente utilizzando l'operatore '**`+`**'. Per esempio,

```
"Ciao " + "come " + "stai?"
```

restituisce un'unica stringa-costante, come la seguente:

```
"Ciao come stai?"
```

Inoltre, in questi concatenamenti, entro certi limiti, possono essere inseriti elementi diversi da stringhe, come nell'esempio seguente, dove il contenuto numerico intero della variabile '**`contatore`**' viene convertito automaticamente in stringa prima di essere emesso attraverso lo standard output.

```
int contatore = 0;

while (contatore < 10)
{
    contatore++;
    System.out.println ("Ciclo n. " + contatore);
}
```

Le stringhe variabili sono oggetti di tipo '**`StringBuffer`**' e verranno descritte più avanti.

302.5 Metodo main()

Si è accennato al fatto che una classe che non eredita esplicitamente da un'altra, richiede l'esistenza del metodo '**`main()`**' e viene detta applicazione. Questo metodo deve avere una forma precisa e si tratta di quello che viene chiamato automaticamente quando si avvia il binario Java corrispondente alla classe stessa. Senza questa convenzione, non ci sarebbe un modo per avviare un programma Java.

```
public static void main (String[] args) { istruzioni }
```

Nella sintassi indicata, le parentesi graffe fanno parte della dichiarazione del metodo e delimitano un gruppo di istruzioni.

302.5.1 args

È Importante osservare l'unico parametro del metodo '**`main()`**': l'array '**`args`**' composto da elementi di tipo '**`String`**'. Questo array contiene gli argomenti passati al programma Java attraverso la riga di comando.

L'esempio seguente, mostra come si può leggere il contenuto di questo array, tenendo presente che non si conosce inizialmente la sua dimensione. L'esempio emette separatamente, attraverso lo standard output, l'elenco degli argomenti ricevuti.

```
/**
 *  LeggiArgomentiApp.java
 *  Legge gli argomenti e gli emette attraverso lo standard output.
 */

import java.lang.*; // predefinita

class LeggiArgomentiApp
{
    public static void main (String[] args)
    {
        int i;

        for (i = 0; i < args.length; i++)
          {
            System.out.println (args[i]);
          }
    }
}
```

Java: programmazione a oggetti

Il capitolo precedente ha introdotto l'uso del linguaggio Java per arrivare a scrivere programmi elementari, utilizzando i metodi come se fossero delle funzioni pure e semplici. In questo capitolo si introducono gli oggetti secondo Java.

303.1 Creazione e distruzione di un oggetto

Un oggetto è un'*istanza* di una classe, come una copia ottenuta da uno stampo. Come nel caso della creazione di una variabile contenente un tipo di dati primitivo, si distinguono due fasi: la dichiarazione e l'inizializzazione. Trattandosi di un oggetto, l'inizializzazione richiede prima la creazione dell'oggetto stesso, in modo da poter assegnare alla variabile il riferimento di questo.

303.1.1 Dichiarazione dell'oggetto

La dichiarazione di un oggetto è precisamente la dichiarazione di una variabile atta a contenere un riferimento a un particolare tipo di oggetto, specificato dalla classe che può generarlo.

classe variabile

La sintassi appena mostrata dovrebbe essere sufficientemente chiara. Nell'esempio seguente si dichiara la variabile '**`miaStringa`**' predisposta a contenere un riferimento a un oggetto di tipo '**`String`**'.

```
String miaStringa;
```

La semplice dichiarazione della variabile non basta a creare l'oggetto, in quanto così si crea solo il contenitore adatto.

303.1.2 Instanza di un oggetto

L'istanza di un oggetto si ottiene utilizzando l'operatore '**`new`**' seguito da una chiamata a un metodo particolare il cui scopo è quello di inizializzare opportunamente il nuovo oggetto che viene creato. In pratica, '**`new`**' alloca memoria per il nuovo oggetto, mentre il metodo chiamato lo prepara. Alla fine, viene restituito un riferimento all'oggetto appena creato.

L'esempio seguente, definisce la variabile '**`miaStringa`**' predisposta a contenere un riferimento a un oggetto di tipo '**`String`**', creando contestualmente un nuovo oggetto '**`String`**' inizializzato in modo da contenere un messaggio di saluto.

```
String miaStringa = new String ("Ciao ciao.");
```

303.1.3 Metodo costruttore

L'inizializzazione di un oggetto viene svolta da un metodo specializzato per questo scopo: il *costruttore*. Una classe può fornire diversi metodi costruttori che possono servire a inizializzare in modo diverso l'oggetto che si ottiene. Tuttavia, convenzionalmente, ogni classe fornisce sempre un metodo il cui nome corrisponde a quello della classe stessa, ed è senza argomenti. Questo metodo esiste anche se non viene indicato espressamente all'interno della classe.

Java consente di utilizzare lo stesso nome per metodi che accettano argomenti in quantità o tipi diversi, perché è in grado di distinguere il metodo chiamato effettivamente in base agli argomenti forniti. Questo meccanismo permette di avere classi con diversi metodi costruttori, che richiedono una serie differente di argomenti.

303.1.4 Utilizzo degli oggetti

Finché non si utilizza in pratica un oggetto non si può apprezzare, né comprendere, la programmazione a oggetti. Un oggetto è una sorta di scatola nera a cui si accede attraverso variabili e metodi dell'oggetto stesso.

Si indica una variabile o un metodo di un oggetto aggiungendo un punto ('`.`') al riferimento dell'oggetto, seguito dal nome della variabile o del metodo da raggiungere. Variabili e metodi si distinguono perché questi ultimi possono avere una serie di argomenti racchiusi tra parentesi (se non hanno argomenti, vengono usate le parentesi senza nulla all'interno).

riferimento_all'oggetto . *variabile*
riferimento_all'oggetto . *metodo* ( )

Prima di proseguire, è bene soffermarsi sul significato si tutto questo. Indicare una cosa come '**`oggetto.variabile`**', significa raggiungere una variabile appartenente a una particolare struttura di dati, che è appunto l'oggetto. In un certo senso, ciò si avvicina all'accesso a un elemento di un array.

Un po' più difficile è comprendere il senso di un metodo di un oggetto. Indicare '**`oggetto.metodo()`**' significa chiamare una funzione che interviene in un ambiente particolare: quello dell'oggetto.

A questo punto, è necessario chiarire che il riferimento all'oggetto è qualunque cosa in grado di restituire un riferimento a questo. Normalmente si tratta di una variabile, ma questa potrebbe appartenere a sua volta a un altro oggetto. È evidente che sta poi al programmatore cercare si scrivere un programma leggibile.

Nella programmazione a oggetti si insegna comunemente che si dovrebbe evitare di accedere direttamente alle variabili, cercando di utilizzare il più possibile i metodi. Si immagini l'esempio seguente che è solo ipotetico.

```
class Divisione
{
    public int x;
    public int y;
    public calcola ()
    {
        return x/y;
    }
}
```

Se venisse creato un oggetto a partire da questa classe, si potrebbe modificare il contenuto delle variabili e quindi richiamare il calcolo, come nell'esempio seguente:

```
Divisione div = new Divisione ();
div.x = 10;
div.y = 5;
System.out.println ("Il risultato è " + div.calcola ());
```

Però, se si tenta di dividere per zero si ottiene un errore irreversibile. Se invece esistesse un metodo che si occupa di ricevere i dati da inserire nelle variabili, verificando prima che siano validi, si potrebbe evitare di dover prevedere questi inconvenienti.

L'esempio mostrato è volutamente banale, ma gli oggetti (ovvero le classi che li generano) possono essere molto complessi; pertanto, la loro utilità sta proprio nel fatto di poter inserire al loro interno tutti i meccanismi di filtro e controllo necessari al loro buon funzionamento.

In conclusione, in Java è considerato un buon approccio di programmazione l'utilizzo delle variabili solo in lettura, senza poterle modificarle direttamente dall'esterno dell'oggetto.

La chiamata di un metodo di un oggetto viene anche detta *messaggio*, per sottolineare il fatto che si invia un'informazione (eventualmente composta dagli argomenti del metodo) all'oggetto stesso.

303.1.5 Distruzione di un oggetto

In Java, un oggetto viene eliminato automaticamente quando non esistono più riferimenti alla sua struttura. In pratica, se viene creato un oggetto assegnando il suo riferimento a una variabile, quando questa viene eliminata perché è terminato il suo campo di azione, anche l'oggetto viene eliminato.

Tuttavia, l'eliminazione di un oggetto non può essere presa tanto alla leggera. Un oggetto potrebbe avere in carico la gestione di un file che deve essere chiuso prima dell'eliminazione dell'oggetto stesso. Per questo, esiste un sistema di eliminazione degli oggetti, definito *garbage collector*, o più semplicemente *spazzino*, che prima di eliminare un oggetto gli permette di eseguire un metodo conclusivo: '**`finalize()`**'. Questo metodo potrebbe occuparsi di chiudere i file rimasti aperti e di concludere ogni altra cosa necessaria.

303.2 Classi

Le classi sono lo stampo, o il prototipo, da cui si ottengono gli oggetti. La sintassi per la creazione di una classe è la seguente. Le parentesi graffe fanno parte dell'istruzione necessaria a creare la classe e ne delimitano il contenuto, ovvero il corpo, costituito dalla dichiarazione di variabili e metodi. Convenzionalmente, il nome di una classe inizia con una lettera maiuscola.

[*modificatore*] `class` *classe* [`extends` *classe_superiore*] [`implements` *elenco_interfacce*] `{…}`

Il modificatore può essere costituito da uno dei nomi seguenti, a cui corrisponde un valore differente della classe.

- '**`public`**'

  Quando la classe è accessibile anche al di fuori del pacchetto di classi cui appartiene, si utilizza il modificatore '**`public`**'. Se questo non viene indicato, la classe è accessibile solo all'interno del pacchetto cui appartiene.

- '**`abstract`**'

  Quando una classe serve solo come modello astratto per generare altre sottoclassi si utilizza il modificatore '**`abstract`**'.

- '**`final`**'

  Quando si vuole evitare che una classe possa generare altre sottoclassi si indica il modificatore '**`final`**'.

Tutte le classi ereditano automaticamente dalla classe '**`java.lang.Object`**', quando non viene dichiarano espressamente di ereditare da un'altra. La dichiarazione esplicita di volere ereditare da una classe particolare, si ottiene attraverso la parola chiave '**`extends`**' seguita dal nome della classe stessa.

A fianco dell'eredità da un'altra classe, si abbina il concetto di interfaccia, che rappresenta solo un'impostazione a cui si vuole fare riferimento. Questa impostazione non è un'eredità, ma solo un modo per definire una struttura standard che si vuole sia attuata nella classe che si va a creare.

L'eredità avviene sempre solo da una classe, mentre le interfacce che si vogliono utilizzare nella classe possono essere diverse. Se si vogliono specificare più interfacce, i nomi di queste vanno separati con la virgola.

Nel corpo di una classe possono apparire dichiarazioni di variabili e metodi, definiti anche *membri* della classe.

303.2.1 Variabili

Le variabili dichiarate all'interno di una classe, ma all'esterno dei metodi, fanno parte dei cosiddetti membri, sottintendendo con questo che si tratta di componenti delle classi (anche i metodi sono definiti membri). La dichiarazione di una variabile di questo tipo, può essere espressa in forma piuttosto articolata. La sintassi seguente mostra solo gli aspetti più importanti.

```
[specificatore_di_accesso] [static] [final] tipo variabile [= valore_iniziale]
```

Lo specificatore di accesso rappresenta la visibilità della variabile ed è qualcosa di diverso dal campo di azione, che al contrario rappresenta il ciclo vitale di questa. Per definire questa visibilità si utilizza una parola chiave il cui elenco e significato è descritto nella sezione 303.2.3.

La parola chiave '**`static`**' indica che si tratta di una variabile appartenente strettamente alla classe, mentre la mancanza di questa indicazione farebbe sì che si tratti di una variabile di istanza. Quando si dichiarano variabili statiche, si intende che ogni istanza (ogni oggetto generato) della classe che le contiene faccia riferimento alle stesse variabili. Al contrario, in presenza di variabili non statiche, ogni istanza della classe genera una nuova copia indipendente di queste variabili.

La parola chiave '**`final`**' indica che si tratta di una variabile che non può essere modificata, in pratica si tratta di una costante. In tal caso, la variabile deve essere inizializzata contemporaneamente alla sua creazione.

Il nome di una variabile inizia convenzionalmente con una lettera minuscola, ma quando si tratta di una costante, si preferisce usare solo lettere maiuscole.

303.2.2 Metodi

I metodi, assieme alle variabili dichiarate all'esterno dei metodi, fanno parte dei cosiddetti membri delle classi. La sintassi seguente mostra solo gli aspetti più importanti della dichiarazione di un metodo. Le parentesi graffe fanno parte dell'istruzione necessaria a creare il metodo e ne delimitano il contenuto, ovvero il corpo.

```
[specificatore_di_accesso] [static] [abstract] [final] tipo_restituito metodo ( [
elenco_parametri ] ) [throws elenco_eccezioni ] {…}
```

Lo specificatore di accesso rappresenta la visibilità del metodo. Per definire questa visibilità si utilizza una parola chiave il cui elenco e significato è descritto nella sezione 303.2.3.

La parola chiave '**`static`**' indica che si tratta di un metodo appartenente strettamente alla classe, mentre la mancanza di questa indicazione farebbe sì che si tratti di un metodo di istanza. I metodi statici possono accedere solo a variabili statiche; di conseguenza, per essere chiamati non c'è bisogno di creare un'istanza della classe che li contiene. Il metodo normale, non statico, richiede la creazione di un'istanza della classe che lo contiene per poter essere eseguito.

La parola chiave '**`abstract`**' indica che si tratta della struttura di un metodo, del quale vengono indicate solo le caratteristiche esterne, senza definirne il contenuto.

La parola chiave '**`final`**' indica che si tratta di un metodo che non può essere dichiarato nuovamente, nel senso che non può essere modificato in una sottoclasse eventuale.

Il tipo di dati restituito viene indicato prima del nome, utilizzando la stessa definizione che si darebbe a una variabile normale. Nel caso si tratti di un metodo che non restituisce alcunché, si utilizza la parola chiave '**`void`**'.

Il nome di un metodo inizia convenzionalmente con una lettera minuscola, come nel caso delle variabili.

L'elenco di parametri è composto da nessuno o più nomi di variabili precedute dal tipo. Questa elencazione corrisponde implicitamente alla creazione di altrettante variabili locali contenenti il valore corrispondente (in base alla posizione) utilizzato nella chiamata.

La parola chiave '**`throws`**' introduce un elenco di oggetti utili per superare gli errori generati durante l'esecuzione del programma. Tale gestione non viene analizzata in questa documentazione su Java.

303.2.2.1 Sovraccarico

Java ammette il *sovraccarico* dei metodi. Questo significa che, all'interno della stessa classe, si possono dichiarare metodi differenti con lo stesso nome, purché sia diverso il numero o il tipo di parametri che possono accettare. In pratica, il metodo giusto viene riconosciuto alla chiamata in base agli argomenti che vengono forniti.

303.2.2.2 Chiamata di un metodo

La chiamata di un metodo avviene in modo simile a quanto si fa con le chiamate di funzione negli altri linguaggi. La differenza fondamentale sta nella necessità di indicare l'oggetto a cui si riferisce la chiamata.

Java consente anche di eseguire chiamate di metodi riferiti a una classe, quando si tratta di metodi statici.

303.2.3 Specificatore di accesso

Lo specificatore di accesso di variabili e metodi permette di limitare o estendere l'accessibilità di questi, sia per una questione di ordine (nascondendo i nomi di variabili e metodi cui non ha senso accedere da una posizione determinata), sia per motivi di sicurezza.

La tabella 303.1 mostra in modo sintetico e chiaro l'accessibilità dei componenti in base al tipo di specificatore indicato.

Tabella 303.1. Accessibilità di variabili e metodi in base all'uso di specificatori di accesso.

Specificatore	Classe	Sottoclasse	Pacchetto di classi	Altri
package	X		X	
private	X			
protected	X	X	X	
public	X	X	X	X

Se le variabili o i metodi vengono dichiarati senza l'indicazione esplicita di uno specificatore di accesso, viene utilizzato il tipo '**`package`**' in modo predefinito.

303.3 Sottoclassi

Una sottoclasse è una classe che eredita esplicitamente da un'altra. A questo proposito, è il caso di ripetere che tutte le classi ereditano in modo predefinito da '**`java.lang.Object`**', se non viene specificato diversamente attraverso la parola chiave '**`extends`**'.

Quando si crea una sottoclasse, si ereditano tutte le variabili e i metodi che compongono la classe, salvo quei componenti che risultano oscurati dallo specificatore di accesso. Tuttavia, la classe può dichiarare nuovamente alcuni di quei componenti e si può ancora accedere a quelli della classe precedente, nonostante tutto.

303.3.1 super

La parola chiave '**`super`**' rappresenta un oggetto contenente esclusivamente componenti provenienti dalla classe di livello gerarchico precedente. Questo permette di accedere a variabili e metodi che la classe dell'oggetto in questione ha ridefinito. L'esempio seguente mostra la dichiarazione di due classi: la seconda estende la prima.

```
class MiaClasse
{
    int intero;
    void mioMetodo ()
    {
        intero = 100;
    }
}

class MiaSottoclasse extends MiaClasse
{
    int intero;
    void mioMetodo ()
    {
        intero = 0;
        super.mioMetodo ();
        System.out.println (intero);
        System.out.println (super.intero);
    }
}
```

La coppia di classi mostrata sopra è fatta per generare un oggetto a partire dalla seconda, quindi per eseguire il metodo '**`mioMetodo()`**' su questo oggetto. Il metodo a essere eseguito effettivamente è quello della sottoclasse.

Quando ci si comporta in questo modo, ridefinendo un metodo in una sottoclasse, è normale che questo richiami il metodo della classe superiore, in modo da aggiungere solo il codice sorgente che serve in più. In questo caso, viene richiamato il metodo omonimo della classe superiore utilizzando '**`super`**' come riferimento.

Nello stesso modo, è possibile accedere alla variabile '**`intero`**' della classe superiore, anche se in quella attuale tale variabile viene ridefinita.

> È il caso di osservare che la parola chiave '**`super`**' ha senso solo quando dalla classe si genera un oggetto. Quando si utilizzano metodi e variabili statici per evitare di dover generare l'istanza di un oggetto, non è possibile utilizzare questa tecnica per raggiungere metodi e variabili di una classe superiore.

303.3.2 this

La parola chiave '**this**' permette di fare riferimento esplicitamente all'oggetto stesso. Ciò può essere utile in alcune circostanze, come nell'esempio seguente:

```
class MiaClasse
{
    int imponibile;
    int imposta;
    void datiFiscali (int imponibile, int imposta)
    {
        this.imponibile = imponibile;
        this.imposta = imposta;
    }
    ...
}
```

La classe appena mostrata dichiara due variabili che servono a conservare le informazioni su imponibile e imposta. Il metodo '**datiFiscali()**' permette di modificare questi dati in base agli argomenti con cui viene chiamato.

Per comodità, il metodo indica con gli stessi nomi le variabili utilizzate per ricevere i valori delle chiamate. Tali variabili diventano locali e oscurano le variabili di istanza omonime. Per poter accedere alle variabili di istanza si utilizza quindi la parola chiave '**this**'.

> Anche in questa situazione, la parola chiave '**this**' ha senso solo quando dalla classe si genera un oggetto.

303.4 Interfacce

In Java, l'interfaccia è una raccolta di costanti e di definizioni di metodi senza attuazione. In un certo senso, si tratta di una sorta di prototipo di classe. Le interfacce non seguono la gerarchia delle classi perché rappresentano una struttura indipendente: un'interfaccia può ereditare da una o più interfacce definite precedentemente (al contrario delle classi che possono ereditare da una sola classe superiore), ma non può ereditare da una classe.[1]

La sintassi per la definizione di un'interfaccia, è la seguente:

[`public`] `interface` *interfaccia* [`extends` *elenco_interfacce_superiori*] {…}

Il modificatore '**public**' fa in modo che l'interfaccia sia accessibile a qualunque classe, indipendentemente dal pacchetto di classi cui questa possa appartenere. Al contrario, se non viene utilizzato, l'interfaccia risulta accessibile solo alle classi dello stesso pacchetto.

La parola chiave '**extends**' permette di indicare una o più interfacce superiori da cui ereditare. Un'interfaccia non può ereditare da una classe.

303.4.1 Contenuto di un'interfaccia

Un'interfaccia può contenere solo la dichiarazione di costanti e di metodi astratti (senza attuazione). In pratica, non viene indicato alcuno specificatore di accesso e nessun'altra definizione che non sia il tipo, come nell'esempio seguente:

[1] Nel caso di interfacce, non è corretto parlare di ereditarietà, ma questo concetto rende l'idea di ciò che succede effettivamente.

```
interface Raccoltina
{
    int LIMITEMASSIMO = 1000;

    void aggiungi (Object, obj);
    int conteggio ();
    ...
}
```

Si intende implicitamente che le variabili siano '**`public`**', '**`static`**' e '**`final`**', inoltre si intende che i metodi siano '**`public`**' e '**`abstract`**'.

Come si può osservare dall'esempio, la definizione dei metodi termina con l'indicazione dei parametri. Il corpo dei metodi, ovvero la loro attuazione, non viene indicato, perché non è questo il compito di un'interfaccia.

303.4.2 Utilizzo di un'interfaccia

Un'interfaccia viene utilizzata in pratica quando una classe dichiara di attuare (realizzare) una o più interfacce. L'esempio seguente mostra l'utilizzo della parola chiave '**`implements`**' per dichiarare il legame con l'interfaccia vista nella sezione precedente.

```
class MiaClasse implements Raccoltina
{
    ...
    void aggiungi (Object, obj)
    {
        ...
    }
    int conteggio ()
    {
        ...
    }
    ...
}
```

In pratica, la classe che attua un'interfaccia, è obbligata a definire i metodi che l'interfaccia si limita a dichiarare in modo astratto. Si tratta quindi solo di una forma di standardizzazione e di controllo attraverso la stessa compilazione.

303.5 Pacchetti di classi

In Java si realizzano delle librerie di classi e interfacce attraverso la costruzione di pacchetti, come già accennato in precedenza. L'esempio seguente mostra due sorgenti Java, '**`Uno.java`**' e '**`Due.java`**' rispettivamente, appartenenti allo stesso pacchetto denominato '**`PaccoDono`**'. La dichiarazione dell'appartenenza al pacchetto viene fatta all'inizio, con l'istruzione '**`package`**'.

```
/**
 *  Uno.java
 *  Classe pubblica appartenente al pacchetto «PaccoDono».
 */

package PaccoDono;

public class Uno
{
    public void Visualizza ()
    {
        System.out.println ("Ciao Mondo - Uno");
    }
}
```

```
/**
 *  Due.java
 *  Classe pubblica appartenente al pacchetto «PaccoDono».
 */

package PaccoDono;

public class Due
{
    public void Visualizza ()
    {
        System.out.println ("Ciao Mondo - Due");
    }
}
```

303.5.1 Collocazione dei pacchetti

Quando si dichiara in un sorgente che una classe appartiene a un certo pacchetto, si intende che il binario Java corrispondente (il file '`.class`') sia collocato in una directory con il nome di quel pacchetto. Nell'esempio visto in precedenza si utilizzava la dichiarazione seguente:

```
package PaccoDono;
```

In tal modo, la classe (o le classi) di quel sorgente deve poi essere collocata nella directory '`PaccoDono/`'. Questa directory, a sua volta, deve trovarsi all'interno dei percorsi definiti nella variabile di ambiente '**`CLASSPATH`**'.

La variabile '**`CLASSPATH`**' è già stata vista in riferimento al file '`classes.zip`' o '`Klasses.jar`' (a seconda del tipo di compilatore e interprete Java), che si è detto contenere le librerie standard di Java. Tali librerie sono in effetti dei pacchetti di classi.[2]

Se per ipotesi si decidesse di collocare la directory '`PaccoDono/`' a partire dalla propria directory personale, si potrebbe aggiungere nello script di configurazione della propria shell, qualcosa come l'istruzione seguente (adatta a una shell derivata da quella di Bourne).

```
CLASSPATH="$HOME:$CLASSPATH"
export CLASSPATH
```

Generalmente, per permettere l'accesso a pacchetti installati a partire dalla stessa directory di lavoro (nel caso del nostro esempio si tratterebbe di '`./PaccoDono/`'), si può aggiungere anche questa ai percorsi di '**`CLASSPATH`**'.

```
CLASSPATH=".:$HOME:$CLASSPATH"
export CLASSPATH
```

303.5.2 Utilizzo di classi di un pacchetto

L'utilizzo di classi da un pacchetto è già stato visto nei primi esempi, dove si faceva riferimento al fatto che ogni classe importa implicitamente le classi del pacchetto '**`java.lang`**'. Si importa una classe con un'istruzione simile all'esempio seguente:

```
import MioPacchetto.MiaClasse;
```

Per importare tutte le classi di un pacchetto, si utilizza un'istruzione simile all'esempio seguente:

```
import MioPacchetto.*;
```

[2] Il file '`classes.zip`' (o il file '`Klasses.jar`') potrebbe essere decompresso a partire dalla posizione in cui si trova, ma generalmente questo non si fa.

In realtà, la dichiarazione dell'importazione di una o più classi, non è indispensabile, perché si potrebbe fare riferimento a quelle classi utilizzando un nome che comprende anche il pacchetto, separato attraverso un punto.

L'esempio seguente rappresenta un programma banale che utilizza le due classi mostrate negli esempi all'inizio di queste sezioni dedicate ai pacchetti.

```
/**
 *  MiaProva.java
 *  Classe che accede alle classi del pacchetto «PaccoDono».
 */

import PaccoDono.*;

class MiaProva
{
    public static void main (String[] args)
    {
        // Dichiara due oggetti dalle classi del pacchetto PaccoDono.
        Uno primo = new Uno ();
        Due secondo = new Due ();

        // Utilizza i metodi degli oggetti.
        primo.Visualizza ();
        secondo.Visualizza ();
    }
}
```

L'effetto che si ottiene è la sola emissione dei messaggi seguenti attraverso lo standard output.

```
Ciao Mondo - Uno
Ciao Mondo - Due
```

Se nel file non fosse stato dichiarato esplicitamente l'utilizzo di tutte le classi del pacchetto, sarebbe stato possibile accedere ugualmente alle sue classi utilizzando una notazione completa, che comprende anche il nome del pacchetto stesso. In pratica, l'esempio si modificherebbe come segue:

```
/**
 *  MiaProva.java
 *  Classe che accede alle classi del pacchetto «PaccoDono».
 */

class MiaProva
{
    public static void main (String[] args)
    {
        // Dichiara due oggetti dalle classi del pacchetto PaccoDono.
        PaccoDono.Uno primo = new PaccoDono.Uno ();
        PaccoDono.Due secondo = new PaccoDono.Due ();

        // Utilizza i metodi degli oggetti.
        primo.Visualizza ();
        secondo.Visualizza ();
    }
}
```

303.6 Esempi

Gli esempi mostrati nelle sezioni seguenti sono molto semplici, nel senso che si limitano a mostrare messaggi attraverso lo standard output. Si tratta quindi di pretesti per vedere come utilizzare quanto spiegato in questo capitolo. Viene usata in particolare la classe seguente per ottenere degli oggetti e delle sottoclassi.

```
/**
 *      SuperApp.java
 */

class SuperApp
{
    static int variabileStatica = 0; // variabile statica o di classe
    int variabileDiIstanza = 0;  // variabile di istanza

    // Nelle applicazioni è obbligatoria la presenza di questo metodo.
    public static void main (String[] args)
    {
        // Se viene avviata questa classe da sola, viene visualizzato
        // il messaggio seguente.
        System.out.println ("Ciao!");
    }

    // Metodo statico. Può essere usato per accedere solo alla
    // variabile statica.
    public static void metodoStatico ()
    {
        variabileStatica++;
        System.out.println
          ("La variabile statica ha raggiunto il valore "
           + variabileStatica);
    }

    // Metodo di istanza. Può essere usato per accedere sia alla
    // variabile statica che a quella di istanza.
    public void metodoDiIstanza ()
    {
        variabileStatica++;
        variabileDiIstanza++;
        System.out.println
          ("La variabile statica ha raggiunto il valore "
           + variabileStatica);
        System.out.println
          ("La variabile di istanza ha raggiunto il valore "
           + variabileDiIstanza);
    }
}
```

303.6.1 Oggetti e messaggi

Si crea un oggetto a partire da una classe, contenuta generalmente in un pacchetto. Nella sezione precedente è stata presentata una classe che si intende non appartenga ad alcun pacchetto di classi. Ugualmente può essere utilizzata per creare degli oggetti.

L'esempio seguente crea un oggetto a partire da quella classe e quindi esegue la chiamata del metodo '**`metodoDiIstanza`**', che emette due messaggi, per ora senza significato.

```
/**
 *      EsempioOggetti1App.java
 */

class EsempioOggetti1App
```

```
{
    public static void main (String[] args)
    {
        SuperApp oSuperApp = new SuperApp ();
        oSuperApp.metodoDiIstanza ();
    }
}
```

303.6.2 Variabili di istanza e variabili statiche

Le variabili di istanza appartengono all'oggetto, per cui, ogni volta che si crea un oggetto a partire da una classe si crea una nuova copia di queste variabili. Le variabili statiche, al contrario, appartengono a tutti gli oggetti della classe, per cui, quando si crea un nuovo oggetto, per queste variabili viene creato un riferimento all'unica copia esistente.

L'esempio seguente è una variante di quello precedente in cui si creano due oggetti dalla stessa classe, quindi viene chiamato lo stesso metodo, prima da un oggetto, poi dall'altro. Il metodo '**`metodoDiIstanza()`**' incrementa due variabili: una di istanza e l'altra statica.

```
/**
 *      EsempioOggetti2App.java
 */

class EsempioOggetti2App
{
    public static void main (String[] args)
    {
        SuperApp oSuperApp = new SuperApp ();
        SuperApp oSuperAppBis = new SuperApp ();

        oSuperApp.metodoDiIstanza ();
        oSuperAppBis.metodoDiIstanza ();
    }
}
```

Avviando l'eseguibile Java che deriva da questa classe, si ottiene la visualizzazione del testo seguente:

```
La variabile statica ha raggiunto il valore 1
La variabile di istanza ha raggiunto il valore 1
La variabile statica ha raggiunto il valore 2
La variabile di istanza ha raggiunto il valore 1
```

Le prime due righe sono generate dalla chiamata '**`oSuperApp.metodoDiIstanza()`**', mentre le ultime due da '**`oSuperAppBis.metodoDiIstanza()`**'. Si può osservare che l'incremento della variabile statica avvenuto nella prima chiamata riferita all'oggetto '**`oSuperApp`**' si riflette anche nel secondo oggetto, '**`oSuperAppBis`**', che mostra un valore più grande rispetto alla variabile di istanza corrispondente.

303.6.3 Ereditarietà

Nella programmazione a oggetti, il modo più naturale di acquisire variabili e metodi è quello di ereditare da una classe superiore che fornisca ciò che serve. L'esempio seguente mostra una classe che estende quella dell'esempio introduttivo ('**`SuperApp`**'), aggiungendo due metodi.

```
/**
 *      SottoclasseApp.java
 */

class SottoclasseApp extends SuperApp
{
```

```
    public static void decrementaStatico ()
    {
        variabileStatica--;
        System.out.println
          ("La variabile statica ha raggiunto il valore "
           + variabileStatica);
    }

    public void decrementaDiIstanza ()
    {
        variabileStatica--;
        variabileDiIstanza--;
        System.out.println
          ("La variabile statica ha raggiunto il valore "
           + variabileStatica);
        System.out.println
          ("La variabile di istanza ha raggiunto il valore "
           + variabileDiIstanza);
    }
}
```

Se dopo la compilazione si esegue questa classe, si ottiene l'esecuzione del metodo '**`main()`**' che è stato definito nella classe superiore. In pratica, si ottiene la visualizzazione di un semplice messaggio di saluto e nulla altro.

303.6.4 Metodi di istanza e metodi statici

Il metodo di istanza può accedere sia a variabili di istanza che a variabili statiche. Questo è stato visto nell'esempio del sorgente '`EsempioOggetti2App.java`', in cui il metodo '**`metodoDiIstanza()`**' incrementava e visualizzava il contenuto di due variabili, una di istanza e una statica.

I metodi statici possono accedere solo a variabili statiche, che come tali possono essere chiamati anche senza la necessità di creare un oggetto: basta fare riferimento direttamente alla classe. L'esempio mostra in che modo si possa chiamare il metodo '**`metodoStatico()`**' della classe '**`SuperApp`**', senza fare riferimento a un oggetto.

```
/**
 *      EsempioOggetti3App.java
 */

class EsempioOggetti3App
{
    public static void main (String[] args)
    {
        SuperApp.metodoStatico ();
    }
}
```

Nello stesso modo, quando in una classe si vuole chiamare un metodo senza dovere prima creare un oggetto, è necessario che i metodi in questione siano statici.

Java: esempi di programmazione

Questo capitolo raccoglie solo alcuni esempi di programmazione, in parte già descritti in altri capitoli. Lo scopo di questi esempi è solo didattico, utilizzando forme non ottimizzate per la velocità di esecuzione.

304.1 Problemi elementari di programmazione

In questa sezione vengono mostrati alcuni algoritmi elementari portati in Java. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo 282.

304.1.1 Somma tra due numeri positivi

Il problema della somma tra due numeri positivi, attraverso l'incremento unitario, è stato descritto nella sezione 282.2.1.

```
//=====================================================================
// java SommaApp <x> <y>
// Somma esclusivamente valori positivi.
//=====================================================================

import java.lang.*; // predefinita

//---------------------------------------------------------------------
class SommaApp
{
    //-----------------------------------------------------------------
    static int somma (int x, int y)
```

```
    {
        int i;
        int z = x;

        for (i = 1; i <= y; i++)
          {
            z++;
          }
        return z;
    }

    //=================================================================
    // Inizio del programma.
    //-----------------------------------------------------------------
    public static void main (String[] args)
    {
        int x;
        int y;

        x = Integer.valueOf(args[0]).intValue ();
        y = Integer.valueOf(args[1]).intValue ();

        System.out.println (x + "+" + y + "=" + somma (x, y));
    }
}
//=================================================================
```

In alternativa si può tradurre il ciclo '**`for`**' in un ciclo '**`while`**'.

```
    static int somma (int x, int y)
    {
        int z = x;
        int i = 1;

        while (i <= y)
          {
            z++;
            i++;
          }
        return z;
    }
```

304.1.2 Moltiplicazione di due numeri positivi attraverso la somma

Il problema della moltiplicazione tra due numeri positivi, attraverso la somma, è stato descritto nella sezione 282.2.2.

```
//=====================================================================
// java MoltiplicaApp <x> <y>
// Moltiplica esclusivamente valori positivi.
//=====================================================================

import java.lang.*; // predefinita

//-------------------------------------------------------------------
class MoltiplicaApp
{
    //---------------------------------------------------------------
    static int moltiplica (int x, int y)
    {
        int i;
        int z = 0;

        for (i = 1; i <= y; i++)
          {
            z = z + x;
```

```
          }
        return z;
    }

    //================================================================
    // Inizio del programma.
    //----------------------------------------------------------------
    public static void main (String[] args)
    {
        int x;
        int y;

        x = Integer.valueOf(args[0]).intValue ();
        y = Integer.valueOf(args[1]).intValue ();

        System.out.println (x + "*" + y + "=" + moltiplica (x, y));
    }
}
//================================================================
```

In alternativa si può tradurre il ciclo '**`for`**' in un ciclo '**`while`**'.

```
    static int moltiplica (int x, int y)
    {
        int z = 0;
        int i = 1;

        while (i <= y)
          {
            z = z + x;
            i++;
          }
        return z;
    }
```

304.1.3 Divisione intera tra due numeri positivi

Il problema della divisione tra due numeri positivi, attraverso la sottrazione, è stato descritto nella sezione 282.2.3.

```
//========================================================================
// java DividiApp <x> <y>
// Divide esclusivamente valori positivi.
//========================================================================

import java.lang.*; // predefinita

//------------------------------------------------------------------------
class DividiApp
{
    //--------------------------------------------------------------------
    static int dividi (int x, int y)
    {
        int z = 0;
        int i = x;

        while (i >= y)
          {
            i = i - y;
            z++;
          }
        return z;
    }

    //================================================================
    // Inizio del programma.
```

```
    //-------------------------------------------------------------
    public static void main (String[] args)
    {
        int x;
        int y;

        x = Integer.valueOf(args[0]).intValue ();
        y = Integer.valueOf(args[1]).intValue ();

        System.out.println (x + ":" + y + "=" + dividi (x, y));
    }
}
//=============================================================================
```

304.1.4 Elevamento a potenza

Il problema dell'elevamento a potenza tra due numeri positivi, attraverso la moltiplicazione, è stato descritto nella sezione 282.2.4.

```
//=============================================================================
// java ExpApp <x> <y>
// Elevamento a potenza di valori positivi interi.
//=============================================================================

import java.lang.*; // predefinita

//-----------------------------------------------------------------------------
class ExpApp
{
    //-------------------------------------------------------------
    static int exp (int x, int y)
    {
        int z = 1;
        int i;

        for (i = 1; i <= y; i++)
          {
            z = z * x;
          }
        return z;
    }

    //=============================================================
    // Inizio del programma.
    //-------------------------------------------------------------
    public static void main (String[] args)
    {
        int x;
        int y;

        x = Integer.valueOf(args[0]).intValue ();
        y = Integer.valueOf(args[1]).intValue ();

        System.out.println (x + "**" + y + "=" + exp (x, y));
    }
}
//=============================================================================
```

In alternativa si può tradurre il ciclo '**`for`**' in un ciclo '**`while`**'.

```
    static int exp (int x, int y)
    {
        int z = 1;
        int i = 1;

        while (i <= y)
```

```
          {
            z = z * x;
            i++;
          }
        return z;
    }
```

Infine, si può usare anche un algoritmo ricorsivo.

```
    static int exp (int x, int y)
    {
        if (x == 0)
          {
            return 0;
          }
        else if (y == 0)
          {
            return 1;
          }
        else
          {
            return (x * exp (x, y-1));
          }
    }
```

304.1.5 Radice quadrata

Il problema della radice quadrata è stato descritto nella sezione 282.2.5.

```
//=========================================================================
// java RadiceApp <x>
// Estrazione della parte intera della radice quadrata.
//=========================================================================

import java.lang.*; // predefinita

//-------------------------------------------------------------------------
class RadiceApp
{
    //---------------------------------------------------------------------
    static int radice (int x)
    {
        int z = 0;
        int t = 0;

        while (true)
          {
            t = z * z;

            if (t > x)
              {
                // È stato superato il valore massimo.
                z--;
                return z;
              }
            z++;
          }
        // Teoricamente, non dovrebbe mai arrivare qui.
    }

    //=====================================================================
    // Inizio del programma.
    //---------------------------------------------------------------------
    public static void main (String[] args)
    {
        int x;
```

```
        x = Integer.valueOf(args[0]).intValue ();

        System.out.println ("radq(" + x + ")=" + radice (x));
    }
}
//=========================================================================
```

304.1.6 Fattoriale

Il problema del fattoriale è stato descritto nella sezione 282.2.6.

```
//=========================================================================
// java FattorialeApp <x>
// Calcola il fattoriale di un valore intero.
//=========================================================================

import java.lang.*; // predefinita

//-------------------------------------------------------------------------
class FattorialeApp
{
    //---------------------------------------------------------------------
    static int fattoriale (int x)
    {
        int i = x - 1;

        while (i > 0)
          {
            x = x * i;
            i--;
          }
        return x;
    }

    //=====================================================================
    // Inizio del programma.
    //---------------------------------------------------------------------
    public static void main (String[] args)
    {
        int x;

        x = Integer.valueOf(args[0]).intValue ();

        System.out.println (x + "! = " + fattoriale (x));
    }
}
//=========================================================================
```

In alternativa, l'algoritmo si può tradurre in modo ricorsivo.

```
    static int fattoriale (int x)
    {
        if (x > 1)
          {
            return (x * fattoriale (x - 1));
          }
        else
          {
            return 1;
          }
        // Teoricamente non dovrebbe arrivare qui.
    }
```

304.1.7 Massimo comune divisore

Il problema del massimo comune divisore, tra due numeri positivi, è stato descritto nella sezione 282.2.7.

```
//=====================================================================
// java MCDApp <x> <y>
// Determina il massimo comune divisore tra due numeri interi positivi.
//=====================================================================

import java.lang.*; // predefinita

//---------------------------------------------------------------------
class MCDApp
{
    //-----------------------------------------------------------------
    static int mcd (int x, int y)
    {
        int i;
        int z = 0;

        while (x != y)
          {
            if (x > y)
              {
                x = x - y;
              }
            else
              {
                y = y - x;
              }
          }
        return x;
    }

    //=================================================================
    // Inizio del programma.
    //-----------------------------------------------------------------
    public static void main (String[] args)
    {
        int x;
        int y;

        x = Integer.valueOf(args[0]).intValue ();
        y = Integer.valueOf(args[1]).intValue ();

        System.out.println ("Il massimo comune divisore tra " + x
                            + " e " + y + " è " + mcd (x, y));
    }
}
//=====================================================================
```

304.1.8 Numero primo

Il problema della determinazione se un numero sia primo o meno, è stato descritto nella sezione 282.2.8.

```
//=====================================================================
// java PrimoApp <x>
// Determina se un numero sia primo o meno.
//=====================================================================

import java.lang.*; // predefinita
```

```
//-------------------------------------------------------------------
class PrimoApp
{
    //-------------------------------------------------------------------
    static boolean primo (int x)
    {
        boolean primo = true;
        int i = 2;
        int j;

        while ((i < x) && primo)
          {
            j = x / i;
            j = x - (j * i);

            if (j == 0)
              {
                primo = false;
              }
            else
              {
                i++;
              }
          }
        return primo;
    }

    //===================================================================
    // Inizio del programma.
    //-------------------------------------------------------------------
    public static void main (String[] args)
    {
        int x;

        x = Integer.valueOf(args[0]).intValue ();

        if (primo (x))
          {
            System.out.println (x + " è un numero primo");
          }
        else
          {
            System.out.println (x + " non è un numero primo");
          }
    }
}
//===================================================================
```

304.2 Scansione di array

In questa sezione vengono mostrati alcuni algoritmi, legati alla scansione degli array, portati in Java. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo 282.

304.2.1 Ricerca sequenziale

Il problema della ricerca sequenziale all'interno di un array, è stato descritto nella sezione 282.3.1.

```
//===================================================================
// java RicercaSeqApp.java
//===================================================================
```

```
import java.lang.*; // predefinita

//---------------------------------------------------------------------
class RicercaSeqApp
{
    //-----------------------------------------------------------------
    static int ricercaseq (int[] lista, int x, int a, int z)
    {
        int i;

        //-------------------------------------------------------------
        // Scandisce l'array alla ricerca dell'elemento.
        //-------------------------------------------------------------
        for (i = a; i <= z; i++)
          {
            if (x == lista[i])
              {
                return i;
              }
          }

        //-------------------------------------------------------------
        // La corrispondenza non è stata trovata.
        //-------------------------------------------------------------
        return -1;
    }

    //=================================================================
    // Inizio del programma.
    //-----------------------------------------------------------------
    public static void main (String[] args)
    {
        int[] lista = new int[args.length-1];
        int x;
        int i;

        //-------------------------------------------------------------
        // Conversione degli argomenti della riga di comando in
        // numeri.
        //-------------------------------------------------------------
        x = Integer.valueOf(args[0]).intValue ();

        for (i = 1; i < args.length; i++)
          {
            lista[i-1] = Integer.valueOf(args[i]).intValue ();
          }

        //-------------------------------------------------------------
        // Esegue la ricerca.
        // In Java, gli array sono oggetti e come tali vengono passati
        // per riferimento.
        //-------------------------------------------------------------
        i = ricercaseq (lista, x, 0, lista.length-1);

        //-------------------------------------------------------------
        // Visualizza il risultato.
        //-------------------------------------------------------------
        System.out.println (x + " si trova nella posizione "
                            + i + ".");
    }
}
//=====================================================================
```

Esiste anche una soluzione ricorsiva che viene mostrata nella subroutine seguente:

```
    static int ricercaseq (int[] lista, int x, int a, int z)
    {
```

```
        if (a > z)
          {
            //-----------------------------------------------------
            // La corrispondenza non è stata trovata.
            //-----------------------------------------------------
            return -1;

          }
        else if (x == lista[a])
          {
            return a;
          }
        else
          {
            return ricercaseq (lista, x, a+1, z);
          }
    }
```

304.2.2 Ricerca binaria

Il problema della ricerca binaria all'interno di un array, è stato descritto nella sezione 282.3.2.

```
//=========================================================================
// java RicercaBinApp.java
//=========================================================================

import java.lang.*; // predefinita

//-------------------------------------------------------------------------
class RicercaBinApp
{
    //---------------------------------------------------------------
    static int ricercabin (int[] lista, int x, int a, int z)
    {
        int m;

        //-----------------------------------------------------------
        // Determina l'elemento centrale.
        //-----------------------------------------------------------
        m = (a + z) / 2;

        if (m < a)
          {
            //-----------------------------------------------------
            // Non restano elementi da controllare: l'elemento cercato
            // non c'è.
            //-----------------------------------------------------
            return -1;

          }
        else if (x < lista[m])
          {
            //-----------------------------------------------------
            // Si ripete la ricerca nella parte inferiore.
            //-----------------------------------------------------
            return ricercabin (lista, x, a, m-1);

          }
        else if (x > lista[m])
          {
            //-----------------------------------------------------
            // Si ripete la ricerca nella parte superiore.
            //-----------------------------------------------------
            return ricercabin (lista, x, m+1, z);
          }
```

```
        else
          {
            //-------------------------------------------------------
            // m rappresenta l'indice dell'elemento cercato.
            //-------------------------------------------------------
            return m;
          }
    }

    //===================================================================
    // Inizio del programma.
    //-------------------------------------------------------------------
    public static void main (String[] args)
    {
        int[] lista = new int[args.length-1];
        int x;
        int i;

        //-------------------------------------------------------------
        // Conversione degli argomenti della riga di comando in
        // numeri.
        //-------------------------------------------------------------
        x = Integer.valueOf(args[0]).intValue ();

        for (i = 1; i < args.length; i++)
          {
            lista[i-1] = Integer.valueOf(args[i]).intValue ();
          }

        //-------------------------------------------------------------
        // Esegue la ricerca.
        // In Java, gli array sono oggetti e come tali vengono passati
        // per riferimento.
        //-------------------------------------------------------------
        i = ricercabin (lista, x, 0, lista.length-1);

        //-------------------------------------------------------------
        // Visualizza il risultato.
        //-------------------------------------------------------------
        System.out.println (x + " si trova nella posizione "
                            + i + ".");
    }
}
//=====================================================================
```

304.3 Algoritmi tradizionali

In questa sezione vengono mostrati alcuni algoritmi tradizionali portati in Java. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo 282.

304.3.1 Bubblesort

Il problema del Bubblesort è stato descritto nella sezione 282.4.1. Viene mostrata prima una soluzione iterativa e successivamente il metodo '**`bsort`**' in versione ricorsiva.

```
//=====================================================================
// java BSortApp.java
//=====================================================================

import java.lang.*; // predefinita

//---------------------------------------------------------------------
class BSortApp
```

```
{
    //-------------------------------------------------------------
    static int[] bsort (int[] lista, int a, int z)
    {
        int scambio;
        int j;
        int k;

        //-------------------------------------------------------------
        // Inizia il ciclo di scansione dell'array.
        //-------------------------------------------------------------
        for (j = a; j < z; j++)
          {
            //-------------------------------------------------------------
            // Scansione interna dell'array per collocare nella
            // posizione j l'elemento giusto.
            //-------------------------------------------------------------
            for (k = j+1; k <= z; k++)
              {
                if (lista[k] < lista[j])
                  {
                    //-------------------------------------------------
                    // Scambia i valori
                    //-------------------------------------------------
                    scambio = lista[k];
                    lista[k] = lista[j];
                    lista[j] = scambio;
                  }
              }
          }
        //-------------------------------------------------------------
        // In Java, gli array sono oggetti e come tali vengono passati
        // per riferimento. Qui si restituisce ugualmente un
        // riferimento all'array ordinato.
        //-------------------------------------------------------------
        return lista;
    }

    //=============================================================================
    // Inizio del programma.
    //-----------------------------------------------------------------------------
    public static void main (String[] args)
    {
        int[] lista = new int[args.length];
        int i;

        //-------------------------------------------------------------
        // Conversione degli argomenti della riga di comando in
        // numeri.
        //-------------------------------------------------------------
        for (i = 0; i < args.length; i++)
          {
            lista[i] = Integer.valueOf(args[i]).intValue ();
          }

        //-------------------------------------------------------------
        // Ordina l'array.
        // In Java, gli array sono oggetti e come tali vengono passati
        // per riferimento.
        //-------------------------------------------------------------
        bsort (lista, 0, args.length-1);

        //-------------------------------------------------------------
        // Visualizza il risultato.
        //-------------------------------------------------------------
        for (i = 0; i < lista.length; i++)
```

```
          {
            System.out.println ("lista[" + i + "] = "
                                + lista[i]);
          }
    }

}
//=========================================================================
```

Segue il metodo '**`bsort`**' in versione ricorsiva.

```
    static int[] bsort (int[] lista, int a, int z)
    {
        int scambio;
        int k;

        if (a < z)
          {
            //-----------------------------------------------------
            // Scansione interna dell'array per collocare nella
            // posizione a l'elemento giusto.
            //-----------------------------------------------------
            for (k = a+1; k <= z; k++)
              {
                if (lista[k] < lista[a])
                  {
                    //-------------------------------------------------
                    // Scambia i valori
                    //-------------------------------------------------
                    scambio = lista[k];
                    lista[k] = lista[a];
                    lista[a] = scambio;
                  }
              }
            bsort (lista, a+1, z);
          }
        return lista;
    }
```

304.3.2 Torre di Hanoi

Il problema della torre di Hanoi è stato descritto nella sezione 282.4.2.

```
//=========================================================================
// java HanoiApp <n-anelli> <piolo-iniziale> <piolo-finale>
//=========================================================================

import java.lang.*; // predefinita

//-------------------------------------------------------------------------
class HanoiApp
{
    //---------------------------------------------------------------
    static void hanoi (int n, int p1, int p2)
    {
        if (n > 0)
          {
            hanoi (n-1, p1, 6-p1-p2);
            System.out.println ("Muovi l'anello " + n
                                + " dal piolo " + p1
                                + " al piolo " + p2 + ".");
            hanoi (n-1, 6-p1-p2, p2);
          }
    }

    //===============================================================
```

```
    // Inizio del programma.
    //-------------------------------------------------------------
    public static void main (String[] args)
    {
        int n;
        int p1;
        int p2;

        n = Integer.valueOf(args[0]).intValue ();
        p1 = Integer.valueOf(args[1]).intValue ();
        p2 = Integer.valueOf(args[2]).intValue ();

        hanoi (n, p1, p2);
    }
}
//=============================================================================
```

304.3.3 Quicksort

L'algoritmo del Quicksort è stato descritto nella sezione 282.4.3.

```
//=============================================================================
// java QSortApp.java
//=============================================================================

import java.lang.*; // predefinita

//-----------------------------------------------------------------------------
class QSortApp
{
    //-------------------------------------------------------------
    static int part (int[] lista, int a, int z)
    {
        int scambio;

        //---------------------------------------------------------
        // Si assume che a sia inferiore a z.
        //---------------------------------------------------------
        int i = a + 1;
        int cf = z;

        //---------------------------------------------------------
        // Inizia il ciclo di scansione dell'array.
        //---------------------------------------------------------
        while (true)
          {
            while (true)
              {
                //-------------------------------------------------
                // Sposta i a destra.
                //-------------------------------------------------
                if ((lista[i] > lista[a]) || (i >= cf))
                  {
                    break;
                  }
                else
                  {
                    i++;
                  }
              }
            while (true)
              {
                //-------------------------------------------------
                // Sposta cf a sinistra.
                //-------------------------------------------------
```

```
            if (lista[cf] <= lista[a])
              {
                break;
              }
            else
              {
                cf--;
              }
          }

        if (cf <= i)
          {
            //-------------------------------------------------------
            // è avvenuto l'incontro tra i e cf.
            //-------------------------------------------------------
            break;
          }
        else
          {
            //-------------------------------------------------------
            // Vengono scambiati i valori.
            //-------------------------------------------------------
            scambio = lista[cf];
            lista[cf] = lista[i];
            lista[i] = scambio;

            i++;
            cf--;
          }
      }
    //-----------------------------------------------------------------
    // A questo punto lista[a..z] è stata ripartita e cf è la
    // collocazione di lista[a].
    //-----------------------------------------------------------------
    scambio = lista[cf];
    lista[cf] = lista[a];
    lista[a] = scambio;

    //-----------------------------------------------------------------
    // A questo punto, lista[cf] è un elemento (un valore) nella
    // giusta posizione.
    //-----------------------------------------------------------------
    return cf;
}

//---------------------------------------------------------------------
static int[] quicksort (int[] lista, int a, int z)
{
    int cf;

    if (z > a)
      {
        cf = part (lista, a, z);
        quicksort (lista, a, cf-1);
        quicksort (lista, cf+1, z);
      }

    //-----------------------------------------------------------------
    // In Java, gli array sono oggetti e come tali vengono passati
    // per riferimento. Qui si restituisce ugualmente un
    // riferimento all'array ordinato.
    //-----------------------------------------------------------------
    return lista;
}

//=====================================================================
```

```
    // Inizio del programma.
    //-----------------------------------------------------------------
    public static void main (String[] args)
    {
        int[] lista = new int[args.length];
        int i;

        //-----------------------------------------------------------------
        // Conversione degli argomenti della riga di comando in
        // numeri.
        //-----------------------------------------------------------------
        for (i = 0; i < args.length; i++)
          {
            lista[i] = Integer.valueOf(args[i]).intValue ();
          }

        //-----------------------------------------------------------------
        // Ordina l'array.
        // In Java, gli array sono oggetti e come tali vengono passati
        // per riferimento.
        //-----------------------------------------------------------------
        quicksort (lista, 0, args.length-1);

        //-----------------------------------------------------------------
        // Visualizza il risultato.
        //-----------------------------------------------------------------
        for (i = 0; i < lista.length; i++)
          {
            System.out.println ("lista[" + i + "] = "
                                + lista[i]);
          }
    }
}
//=========================================================================
```

304.3.4 Permutazioni

L'algoritmo ricorsivo delle permutazioni è stato descritto nella sezione 282.4.4.

```
//=========================================================================
// java PermutaApp.java
//=========================================================================

import java.lang.*; // predefinita

//-----------------------------------------------------------------
class PermutaApp
{
    //-----------------------------------------------------------------
    static void permuta (int[] lista, int a, int z)
    {
        int scambio;
        int k;
        int i;

        //-----------------------------------------------------------
        // Se il segmento di array contiene almeno due elementi, si
        // procede.
        //-----------------------------------------------------------
        if ((z - a) >= 1)
          {
            //-------------------------------------------------------
            // Inizia un ciclo di scambi tra l'ultimo elemento e uno
            // degli altri contenuti nel segmento di array.
            //-------------------------------------------------------
```

```
            for (k = z; k >= a; k--)
              {
                //-------------------------------------------------
                // Scambia i valori
                //-------------------------------------------------
                scambio = lista[k];
                lista[k] = lista[z];
                lista[z] = scambio;

                //-------------------------------------------------
                // Esegue una chiamata ricorsiva per permutare un
                // segmento più piccolo dell'array.
                //-------------------------------------------------
                permuta (lista, a, z-1);

                //-------------------------------------------------
                // Scambia i valori
                //-------------------------------------------------
                scambio = lista[k];
                lista[k] = lista[z];
                lista[z] = scambio;

              }
          }
        else
          {
            //-----------------------------------------------------
            // Visualizza la situazione attuale dell'array.
            //-----------------------------------------------------
            for (i = 0; i < lista.length; i++)
              {
                System.out.print (" " + lista[i]);
              }
            System.out.println ("");
          }
    }

    //=================================================================
    // Inizio del programma.
    //-----------------------------------------------------------------
    public static void main (String[] args)
    {
        int[] lista = new int[args.length];
        int i;

        //-------------------------------------------------------------
        // Conversione degli argomenti della riga di comando in
        // numeri.
        //-------------------------------------------------------------
        for (i = 0; i < args.length; i++)
          {
            lista[i] = Integer.valueOf(args[i]).intValue ();
          }

        //-------------------------------------------------------------
        // Esegue le permutazioni.
        //-------------------------------------------------------------
        permuta (lista, 0, args.length-1);
    }
}
//=====================================================================
```

Parte lix

Scheme

Scheme: preparazione

Scheme è un linguaggio di programmazione discendente dal LISP, inventato da Guy Lewis Steele Jr. e Gerald Jay Sussman nel 1995 presso il MIT. Scheme è importante soprattutto in quanto lo si ritrova utilizzato in situazioni estranee alla realizzazione di programmi veri e propri; in particolare, i fogli di stile DSSSL (capitolo 254) utilizzano il linguaggio Scheme.

In questo capitolo vengono mostrati gli strumenti più comuni che possono essere utilizzati per fare pratica con questo linguaggio di programmazione: MIT Scheme e Kawa, entrambi interpreti Scheme.

305.1 MIT Scheme

L'interprete Scheme del MIT [1] è disponibile per varie piattaforme. La versione per GNU/Linux può essere ottenuta dal MIT, a partire all'indirizzo <*http://www.swiss.ai.mit.edu/projects/scheme/*>. Il pacchetto può essere estratto a partire da una directory temporanea, da dove poi viene avviato uno script che provvede all'installazione, solitamente a partire dalla gerarchia '`/usr/local/`':

```
# tar xzvf scheme-7.5/scheme-7.5.12-ix86-gnu-linux.tar.gz

# cd dist-7.5

# ./install.sh
```

Nel sito in cui viene distribuito questo interprete, si trova anche la documentazione per il suo utilizzo. Qui si intende mostrare solo l'essenziale.

305.1.1 Utilizzo interattivo

Per avviare l'interprete Scheme del MIT, basta il comando seguente:

```
$ scheme
```

Quello che si vede dopo è una presentazione, seguita dall'invito a inserire comandi secondo il linguaggio Scheme.

```
Scheme saved on Sunday October 18, 1998 at 11:02:46 PM
  Release 7.4.7
  Microcode 11.151
  Runtime 14.168

1 ]=>
```

Per verificare rapidamente il funzionamento, basta utilizzare un'istruzione Scheme elementare che permette la visualizzazione di un messaggio:

```
1 ]=> (display "Ciao mondo!")[ Invio ]

Ciao mondo!
;Unspecified return value
```

Quello che si ottiene, come si vede, è l'emissione del messaggio, seguito dalla conferma che l'istruzione non ha restituito alcun valore.

La conclusione di una sessione di lavoro con l'interprete, si ottiene con l'istruzione '`(exit)`', dopo la quale viene richiesta una conferma, a cui si risponde con la lettera '`y`':

[1] **MIT Scheme** GNU GPL

```
1 ]=> (exit)[ Invio ]

Kill Scheme (y or n)? y

Happy Happy Joy Joy.
```

305.1.2 REPL: l'ambito di funzionamento

REPL sta per *Read-eval-print loop* e rappresenta una struttura di sottosessioni di lavoro all'interno dell'interprete. Il testo che appare come invito a inserire delle istruzioni, indica un numero intero positivo che rappresenta il livello REPL:

```
1 ]=>
```

Inizialmente questo livello è il primo, cioè il numero uno, ma può aumentare quando per qualche motivo c'è bisogno di passare a una sottosessione. La situazione tipica per la quale si passa a un livello successivo è l'inserimento di un'istruzione errata. Si osservi l'esempio seguente, in cui per errore non è stata delimitata la stringa che si vuole visualizzare:

```
1 ]=> (display Ciao mondo!)[ Invio ]

;Unbound variable: mondo!
;To continue, call RESTART with an option number:
; (RESTART 3) => Specify a value to use instead of mondo!.
; (RESTART 2) => Define mondo! to a given value.
; (RESTART 1) => Return to read-eval-print level 1.
```

A seguito di questo, si osserva che la stringa di invito è cambiata, indicando il passaggio a un secondo livello, a causa di un errore. Generalmente, per tornare al primo livello basta l'istruzione '**`(restart 1)`**', come si vede chiaramente nella spiegazione.

```
2 error> (restart 1)[ Invio ]
```

Se si fanno altri errori, si passa a livelli successivi, dai quali è possibile tornare sempre al primo livello nel modo appena mostrato.

305.1.3 Utilizzo non interattivo

Un programma Scheme può essere interpretato direttamente avviando '**`scheme`**' nel modo seguente:

```
scheme < sorgente_scheme
```

In pratica, si fornisce il sorgente attraverso lo standard input, come se venisse digitato attraverso la tastiera.

305.1.4 Compilazione e caricamento di file

L'interprete Scheme del MIT, consente anche una sorta di compilazione, con la quale si genera un formato intermedio, più pratico per l'esecuzione. Per ottenere questo, occorre avviare l'eseguibile '**`scheme`**' con l'opzione '**`-compiler`**'.

```
$ scheme -compiler
```

Una volta predisposto un sorgente Scheme, lo si può compilare attraverso l'interprete con l'istruzione seguente:

```
(cf file_sorgente [file_destinazione])
```

Come si vede, l'indicazione di un file di destinazione è facoltativa, dal momento che in mancanza di questa, si utilizza un nome con la stessa radice di quello del sorgente.

```
1 ]=> (cf "ciao_mondo.scm")
```

L'esempio mostra la compilazione del sorgente '`ciao_mondo.scm`', per generare il file '`ciao_mondo.com`'. La stessa cosa avrebbe potuto essere ottenuta con una delle due istruzioni seguenti:

```
1 ]=> (cf "ciao_mondo.scm" "ciao_mondo")

1 ]=> (cf "ciao_mondo.scm" "ciao_mondo.com")
```

La compilazione di questo tipo può essere utile per memorizzare dei sottoprogrammi da caricare durante una sessione interattiva. L'esempio seguente è la dichiarazione della funzione '**`fattoriale`**', il cui scopo è quello di calcolare il fattoriale di un numero intero.

```
(define (fattoriale n)
    (if (= n 0)
        1
        (* n (fattoriale (- n 1)))))
```

Il sorgente contenente esclusivamente queste righe, potrebbe chiamarsi '`fattoriale.scm`' ed essere stato compilato generando il file '`fattoriale.com`'.

L'interprete consente di caricare un file sorgente, o uno compilato, attraverso l'istruzione seguente:

`(load` *file*`)`

Se il nome del file viene indicato per intero, viene caricato in modo preciso quel file, mentre se si omette l'estensione, l'interprete cerca prima di trovare un file con l'estensione '`.com`', preferendo così una versione compilata eventuale.

Il caricamento di un file coincide anche con la sua esecuzione; se si tratta di dichiarazioni di variabili o di funzioni, si può avere la sensazione che non sia accaduto nulla. In questo caso, caricando il file '`fattoriale.com`', oppure '`fattoriale.scm`', si ottiene la disponibilità della funzione '**`fattoriale`**':

```
1 ]=> (load "fattoriale.scm")[ Invio ]

;Loading "fattoriale.scm" -- done
;Value: fattoriale

1 ]=> (fattoriale 3)[ Invio ]

;Value: 6
```

305.2 Kawa

Kawa [2] è un sistema Scheme, scritto in Java, in grado di funzionare come interprete e anche come compilatore, per trasformare un sorgente Scheme in un binario Java.

Come si può intendere, per poter utilizzare Kawa occorre avere installato Java (il JDK o Kaffe, come descritto nel capitolo 301). Di solito, per utilizzare Kawa come interprete, è sufficiente il comando '**`kawa`**', che dovrebbe corrispondere a uno script in grado di avviare l'interpretazione Java di '`repl.class`', che a sua volta dovrebbe trovarsi nella directory '`/usr/share/`

[2] **Kawa** modified GNU Public License

`java/kawa/repl.class`'. Eventualmente, dovendo fare a meno di questo script, basterebbe il comando seguente:

```
$ java kawa.repl
```

A ogni modo, questo non basta, dal momento che Kawa dispone di una propria libreria di classi che va aggiunta tra i percorsi della variabile di ambiente '**`CLASSPATH`**'. Lo script a cui si faceva riferimento, dovrebbe essere già predisposto in modo tale da includere in questa variabile di ambiente anche il percorso per la libreria di classi di Kawa, tuttavia, volendo realizzare dei binari Java indipendenti, partendo da programmi Scheme, è necessario pubblicizzare tale libreria anche all'esterno dell'interprete Kawa.

Le istruzioni seguenti sono adatte a una shell Bourne, o a una sua derivata e fanno riferimento all'ipotesi che la libreria di classi di Kawa sia stata installata a partire dalla directory '`/usr/share/java/`' (in pratica, si intende che in questo caso la libreria sia stata estratta dal solito archivio compresso):

```
CLASSPATH="$CLASSPATH:/usr/share/java/"
export CLASSPATH
```

305.2.1 Utilizzo interattivo

Per avviare l'interprete Scheme di Kawa, basta il comando seguente:

```
$ kawa
```

oppure, in mancanza di questo script,

```
$ java kawa.repl
```

ricordando che in questo secondo caso, è indispensabile la predisposizione della variabile di ambiente '**`CLASSPATH`**'. Quello che si vede dopo è un invito a inserire delle istruzioni Scheme:

```
#|kawa:1|#
```

Anche con l'interprete Kawa, si può fare una verifica rapida del funzionamento, utilizzando l'istruzione '**`display`**':

```
#|kawa:1|# (display "Ciao mondo!") (newline)[ Invio ]

Ciao mondo!
#|kawa:2|#
```

Mano a mano che si inseriscono delle istruzioni, il numero che compone il testo dell'invito si incrementa progressivamente, indipendentemente dal fatto che siano stati fatti degli errori.

Anche con Kawa, la conclusione di una sessione di lavoro con l'interprete si ottiene con l'istruzione '**`(exit)`**':

```
#|kawa:2|# (exit)[ Invio ]
```

305.2.2 $ kawa e ~/.kawarc.scm

`kawa` [*opzioni*]

Lo script '**`kawa`**', ovvero il comando '**`java kawa.repl`**', permette l'utilizzo di alcune opzioni che possono rivelarsi importanti. In particolare, l'interprete Kawa può leggere ed eseguire le istruzioni contenute in un file apposito, '`~/.kawarc.scm`', prima di procedere con le attività normali. Il file in questione è semplicemente un sorgente Scheme.

Alcune opzioni

`-e` *espressione*

Fa sì che Kawa valuti l'espressione, interpretandola come una serie di istruzioni Scheme, senza leggere il file '`~/.kawarc.scm`'.

`-c` *espressione*

Fa sì che Kawa valuti l'espressione, interpretandola come una serie di istruzioni Scheme, dopo aver letto ed eseguito il contenuto del file '`~/.kawarc.scm`'.

`-f` *file_sorgente_scheme*

Fa in modo che Kawa legga ed esegua il contenuto del file indicato come argomento, ignorando il file di configurazione. Se al posto del nome si indica un trattino orizzontale ('**`-`**'), si intende specificare lo standard input.

`-C` *file_sorgente_scheme*

Compila il sorgente indicato in una classe Java. Se si vuole generare un programma autonomo, occorre utilizzare anche l'opzione '**`--main`**'.

`--main`

Assieme all'opzione '**`-C`**', consente di generare un binario Java, autonomo.

Esempi

```
$ kawa -c '(display "Ciao mondo!") (newline)'
```

Visualizza il messaggio «Ciao mondo!», senza prendere in considerazione il file di configurazione.

```
$ kawa -f ciao_mondo.scm
```

Esegue il contenuto del file '`ciao_mondo.scm`', che si presume essere un sorgente Scheme.

305.2.3 Compilazione

Con Kawa è possibile compilare sia all'interno della sessione di lavoro dell'interprete, sia all'esterno. Nel primo caso, si utilizza l'istruzione

`(compile-file` *nome_sorgente radice_destinazione* `)`

con la quale si può fare qualcosa del genere:

`#|kawa:`*m*`|#` **`(compile-file "ciao_mondo.scm" "ciao")`**[*Invio*]

In questo modo, dal file sorgente '`ciao_mondo.scm`' si ottiene il file '`ciao.zip`', contenente una classe non meglio precisata, il quale può essere ricaricato successivamente con l'istruzione

`(load` *radice_file_compilato* `)`

In pratica, volendo caricare ed eseguire il contenuto del file '`ciao.zip`', basta l'istruzione seguente:

`#|kawa:`*m*`|#` **`(load "ciao")`**[*Invio*]

La compilazione al di fuori dell'ambiente interattivo, si ottiene utilizzando l'opzione '**`-C`**', con la quale si ottengono delle classi Java non compresse. Si distinguono due situazioni:

`kawa` [*altre_opzioni*] `-C` *sorgente_scheme*

`kawa` [*altre_opzioni*] `--main -C` *sorgente_scheme*

Nel primo caso si ottiene un file con estensione '`.class`' che può essere caricato all'interno di una sessione di lavoro dell'interprete, con la funzione '**`load`**' già mostrata; nel secondo si ottiene un programma indipendente.

A titolo di esempio, si può utilizzare il sorgente di prova che viene mostrato di seguito:

```
;
; fattoriale.scm
;
(define (fattoriale n)
    (if (= n 0)
        1
        (* n (fattoriale (- n 1)))))
```

Questo può essere compilato in modo da poterlo ricaricare successivamente:

```
$ kawa -C fattoriale.scm
```

Si ottiene il file '`fattoriale.class`'. All'interno dell'interprete, si può caricare quanto compilato con la funzione '**`load`**' (con la quale si potrebbe caricare anche un sorgente non compilato, indicando il nome completo del file).

```
#|kawa:m|# (load "fattoriale")[ Invio ]
```

Quindi si potrebbe sfruttare la funzione '**`fattoriale`**' dichiarata all'interno del file appena caricato:

```
#|kawa:n|# (display (fattoriale 3)) (newline)[ Invio ]

6
```

Volendo rendere autonomo il programma del calcolo del fattoriale, occorrerebbe aggiungere le istruzioni necessarie per gestire l'inserimento e l'emissione dei dati:

```
;
; fattoriale.scm
;
(define (fattoriale n)
    (if (= n 0)
        1
        (* n (fattoriale (- n 1)))))

(define valore 0)
(display "Inserisci un numero intero: ")
(set! valore (read))
(display "Il fattoriale di ")
(display valore)
(display " è ")
(display (fattoriale valore))
(newline)
```

Per la sua compilazione si procede nel modo già descritto, utilizzando l'opzione '**`--main`**':

```
$ kawa --main -C fattoriale.scm
```

Anche in questo caso si genera il file '`fattoriale.class`', che però può essere avviato direttamente da Java:

```
$ java fattoriale[ Invio ]

Inserisci un numero intero: 3[ Invio ]

Il fattoriale di 3 è 6
```

305.3 Riferimenti

- *MIT Scheme*

  <http://www.swiss.ai.mit.edu/projects/scheme/>

  <ftp://swiss-ftp.ai.mit.edu/pub/>

- Per Bothner, *Kawa, the Java-based Scheme system*, 1999

  <http://www.gnu.org/software/kawa/>

  <ftp://ftp.gnu.org/pub/gnu/kawa/>

Scheme: introduzione

Il linguaggio Scheme ha una filosofia che si basa fondamentalmente sul suo tipo di notazione. Scheme è un linguaggio utile per rappresentare un problema, più che per realizzare un programma completo. La standardizzazione di questo linguaggio è riferita fondamentalmente a un documento che viene aggiornato periodicamente: R^{n}RS, ovvero *Revised-n Report on the Algorithmic Language Scheme*, dove *n* è il numero di questa revisione (attualmente dovrebbe trattarsi della quinta). Tuttavia, la standardizzazione riguarda gli aspetti fondamentali del linguaggio, mentre ogni realizzazione che utilizza Scheme introduce le estensioni necessarie alle circostanze.

In questo capitolo si vogliono descrivere solo alcuni degli aspetti più importanti di questo linguaggio. Il documento di riferimento è quello citato, ovvero *R^{5}RS*; alla fine del capitolo si possono trovare anche altri riferimenti per guide più o meno dettagliate su Scheme.

306.1 Aspetto generale

Il linguaggio Scheme prevede dei commenti, che vengono ignorati regolarmente: si distinguono perché iniziano con un punto e virgola (‘**;**’) e terminano alla fine della riga. Generalmente, le righe vuote e quelle bianche sono ignorate nello stesso modo. In generale, le istruzioni Scheme hanno l’aspetto di qualcosa che è racchiuso tra parentesi tonde.

```
(display "Ciao")
```

Per comprenderne il senso, l’esempio precedente potrebbe essere espresso come si vede sotto, se lo si volesse rappresentare in un linguaggio ipotetico, basato sulle funzioni:

```
display ("Ciao")
```

Tutto quello che si fa con Scheme viene ottenuto attraverso chiamate di funzione, ovvero, secondo la terminologia utilizzata da *R^{5}RS*, *procedure*, che possono restituire o meno un valore. Le chiamate di queste procedure, o di queste funzioni, iniziano con un nome, posto subito dopo la parentesi tonda di apertura, continuando eventualmente con l’elenco dei parametri che gli vengono passati, separati semplicemente da uno o più spazi, anche verticali (non si utilizzano virgole o altri simboli di interpunzione), terminando con la parentesi tonda di chiusura.

(*nome* [*parametro_1* [*parametro_2*]… [*parametro_n*]])

Da quanto affermato, si intende anche che un’istruzione può essere interrotta in qualunque punto in cui potrebbe essere inserito uno spazio, per riprenderla nella riga successiva, incolonnandola in base allo stile preferito. Si osservi l’esempio seguente:

```
(+ 3 4)
```

si tratta di una chiamata a una funzione denominata ‘**+**’, a cui vengono passati i parametri ‘**3**’ e ‘**4**’. Si intende, intuitivamente, che questa funzione restituisca la somma dei parametri.

> Le istruzioni non hanno bisogno di essere terminate da un qualche simbolo di interpunzione, dal momento che le parentesi tonde esprimono chiaramente l’estensione di queste e l’ambito relativo all’interno dei vari annidamenti.

Questo tipo di notazione ha diversi pregi, ma ha il difetto fondamentale di essere un po’ difficile da seguire visivamente, soprattutto a causa dell’affollarsi delle parentesi tonde.

> In questi capitoli si cercherà di utilizzare un allineamento di queste parentesi che renda più facile la lettura delle istruzioni, anche se si tratta di uno stile che di solito non si applica.

Per facilitare la comprensione degli esempi, in questi capitoli dedicati a Scheme, si utilizzerà il simbolo '**===>**' per indicare il valore restituito da una funzione (che appare alla sua destra).

306.1.1 Identificatori e convenzioni nei nomi

I nomi utilizzati per «identificare» qualunque cosa in Scheme, possono essere scritti utilizzando le lettere dell'alfabeto, le cifre numeriche e una serie di caratteri particolari che vengono considerati come un'estensione ai caratteri alfabetici:

```
!   $   %   &   *   +   -   .   /   :   <   =   >   ?   @   ^   _   ~
```

Non tutte le combinazioni sono possibili: in generale non è ammissibile che tali nomi inizino con una cifra numerica.

In generale, Scheme non dovrebbe fare differenza tra lettere maiuscole e minuscole nei nomi che identificano qualcosa.

È importante osservare che, a differenza di altri linguaggi di programmazione, caratteri come '**+**', '**-**', '*****' e '**/**', possono essere (e in pratica sono) dei nomi. Come è già stato fatto osservare,

```
(+ 3 4)
```

è la chiamata della funzione (procedura) '**+**', a cui vengono passati i valori tre e quattro come parametri.

Anche se si possono usare caratteri insoliti nei nomi degli identificatori, quando si dichiara qualcosa, come il nome di una variabile, o di una funzione, è bene astenersi dalle cose troppo stravaganti, a meno che ci sia un buon motivo per le scelte che si fanno. In generale, sono già stabilite delle convenzioni per i nomi delle funzioni, almeno quelle che fanno già parte del linguaggio standard:

- le funzioni il cui nome termina con un punto interrogativo ('**?**') sono intese essere dei «predicati», ovvero delle funzioni che verificano l'avverarsi di una condizione (la verità di un'affermazione) e restituiscono un valore booleano;
- le funzioni il cui scopo è quello di modificare il valore di una variabile, senza cambiarne l'allocazione (per la precisione si tratta di modificare un valore in un'area di memoria già allocata), terminano con un punto esclamativo ('**!**');
- Le funzioni il cui scopo è quello di convertire un «oggetto» di un tipo, in un altro di tipo differente, contengono un '**->**' all'interno del nome.

Per permettere di comprendere meglio come possa essere formato un identificatore, si osservi l'elenco seguente di nomi, che rappresentano tutti delle possibilità valide:

```
ciao            a               b               +               -
*               list->vector    ABCdef123       A123b124        <=?
ciao-come-stai-io-sto-bene-grazie
```

306.1.2 Funzioni o procedure

Scheme è un linguaggio basato sulle funzioni, per quanto queste vengano chiamate «procedure» nella sua terminologia specifica. Questo significa, per esempio, che tutte le espressioni che si possono scrivere con Scheme sono dei valori costanti, oppure delle chiamate di funzione, più o meno annidate. Anche le strutture di controllo sono realizzate in forma di funzione.

È importante osservare che in Scheme non esiste una funzione principale che debba essere eseguita prima delle altre; si segue semplicemente l'ordine sequenziale in cui appaiono le istruzioni. In generale, con lo stesso criterio, le funzioni che si utilizzano devono essere state dichiarate prima del loro utilizzo.

306.2 Allocazione dei dati, espressioni, costanti, oggetti

Scheme utilizza una gestione speciale per le variabili. La dichiarazione di una variabile implica l'allocazione di uno spazio di memoria adatto e l'abbinamento del puntatore relativo a una variabile.

`(define` *variabile* [*valore_iniziale*] `)`

Per esempio,

```
(define x 1)
```

alloca l'area di memoria necessaria a contenere un numero intero, quindi abbina all'identificatore '**`x`**' il puntatore a questa area. In pratica, l'identificatore '**`x`**' si comporta come una variabile di un linguaggio di programmazione «normale», dal momento che quando viene valutato in un'espressione restituisce esattamente il valore a cui punta.

Questa distinzione, non è soltanto una questione di pignoleria, ma si tratta di un punto fondamentale della filosofia di Scheme: la dichiarazione successiva dello stesso identificatore, non va a modificare l'informazione precedente, ma alloca una nuova area di memoria. L'allocazione precedente non viene recuperata e potrebbe continuare a essere utilizzata da ciò che è stato dichiarato prima del cambiamento. In questo senso, a livello teorico, il linguaggio Scheme non prevede un sistema di eliminazione degli oggetti inutilizzati (lo spazzino, ovvero il *garbage collector*), benché le realizzazioni possano attuare in pratica queste forme di ottimizzazione quando sono in grado di provare che un'area di memoria allocata non può più essere presa in considerazione nel programma.

Proprio a causa di questa particolarità di Scheme, per assegnare un valore a un'area di memoria già allocata, attraverso l'identificatore relativo, si utilizza la funzione '**`set!`**':

`(set!` *variabile espressione_del_valore_da_assegnare* `)`

Il punto esclamativo finale che compone il nome della funzione, serve a sottolineare il fatto che si ottiene la modifica di un valore già allocato, senza allocare un'altra area di memoria.

306.2.1 Oggetti, tipi di dati e rappresentazione esterna

I dati secondo Scheme sono organizzati in *oggetti*, ma non nel senso che viene attribuito dai linguaggi di programmazione a oggetti (*object oriented*). I tipi di dati di Scheme sono precisamente:

- booleano -- inteso come il risultato di un'espressione logica, o una costante booleana;
- coppia (lista non vuota);

- simbolico -- che fa riferimento a costanti simili alle stringhe, ma che sono trattate diversamente in Scheme;
- numerico;
- carattere -- un carattere singolo che non è una stringa;
- stringa;
- vettore -- quello che per gli altri linguaggi è un array;
- porta, o flusso -- ovvero un file aperto;
- procedura -- le funzioni di Scheme.

I dati hanno una loro essenza e una loro rappresentazione esterna, che corrisponde al modo in cui vengono espressi a livello umano. Questa rappresentazione può consentire a volte l'uso di forme diverse ed equivalenti; per esempio, il numero 16 può essere espresso con la sequenza dei caratteri '**16**', oppure '**#d16**', '**#x10**' e in altri modi ancora.

Tuttavia, è bene osservare che un oggetto per Scheme può essere di un tipo solo. Si parla in questo senso di «tipi disgiunti».

Scheme fornisce alcuni predicati, ovvero alcune funzioni, per il controllo del tipo a cui appartiene un oggetto. Nello stesso ordine in cui sono stati elencati i tipi di dati, si tratta di: '**boolean?**', '**pair?**', '**symbol?**', '**number?**', '**char?**', '**string?**', '**vector?**', '**port?**', '**procedure?**'. Per esempio, l'istruzione seguente restituisce *Vero* se l'identificatore '**x**' fa riferimento a un numero:

```
(number? x)
```

Tra tutti i tipi di dati visti, ne esiste uno speciale: la lista vuota, che non appartiene alle coppie. Per identificare una lista di qualunque tipo, includendo anche quelle vuote, si usa il predicato '**list?**'.

Tabella 306.1. Elenco dei predicati utili per verificare l'appartenenza ai vari tipi di dati.

Predicato	Descrizione
(boolean? *espressione*)	*Vero* se l'espressione dà come risultato un valore logico booleano.
(pair? *espressione*)	*Vero* se l'espressione dà come risultato una «coppia» (lista non vuota).
(list? *espressione*)	*Vero* se l'espressione dà come risultato una lista (anche vuota).
(symbol? *espressione*)	*Vero* se l'espressione dà come risultato un simbolo.
(number? *espressione*)	*Vero* se l'espressione dà un risultato numerico di qualunque tipo.
(char? *espressione*)	*Vero* se l'espressione dà come risultato un carattere.
(string? *espressione*)	*Vero* se l'espressione dà come risultato una stringa.
(vector? *espressione*)	*Vero* se l'espressione dà come risultato un vettore.
(port? *espressione*)	*Vero* se l'espressione dà come risultato una «porta».
(procedure? *espressione*)	*Vero* se l'espressione dà come risultato una funzione.

306.2.2 Costanti letterali

Scheme ha una gestione particolare delle espressioni, dove al loro interno è speciale la gestione dei valori costanti. Questo fatto verrà chiarito nel seguito. Tuttavia, è necessario conoscere subito in che modo possono essere indicati i valori più comuni in un sorgente Scheme.

306.2.2.1 Costanti booleane

I valori booleani possono essere rappresentati attraverso la sigla '**#t**' per *Vero* e '**#f**' per *Falso*.

306.2.2.2 Costanti numeriche

I valori numerici possono essere usati nel modo consueto, quando si tratta di valori interi (positivi o negativi), quando si vogliono indicare numeri che hanno una quantità fissa di decimali e quando si usa la notazione scientifica comune ('*x***e***y*').

```
67
+67
-67
678.67
+678.67
-678.67
6.7867e2
67867e-3
```

In aggiunta a quello che si può vedere dagli esempi mostrati sopra, si possono indicare dei valori specificando la base di numerazione. Per ottenere questo, si utilizza un prefisso del tipo

```
#x
```

dove *x* è una lettera che esprime la base di numerazione. Segue l'elenco di questi prefissi:

- '**#b**' -- numero binario;
- '**#o**' -- numero ottale;
- '**#d**' -- numero decimale;
- '**#x**' -- numero esadecimale.

Per esempio, '**#x10**' è equivalente a '**#d16**', ovvero a 16 senza prefissi.

Scheme consente di utilizzare anche altri tipi di notazioni, per indicare alcuni tipi particolari di numeri. Questa caratteristica di Scheme viene descritta più avanti.

306.2.2.3 Stringhe

Scheme ha una gestione speciale delle espressioni costanti, cosa che verrà descritta in seguito. Ugualmente, è prevista la presenza delle stringhe, rappresentate attraverso una sequenza di caratteri delimitata da una coppia di apici doppi: '**"**…**"**'.

All'interno delle stringhe è previsto l'uso di sequenze di escape composte dalla barra obliqua inversa ('\') seguita da un carattere. Secondo lo standard *R*^{5}*RS* è prevista solo la sequenza '**\"**', per inserire un apice doppio, e '\\', per poter inserire una barra obliqua inversa. Le varie realizzazioni di Scheme, possono prevedere l'utilizzazione di altre sequenze di escape, per esempio come avviene nel linguaggio C.

> Potrebbe venire spontaneo l'utilizzo della sequenza '**\n**' per inserire il codice di interruzione di riga all'interno di una stringa; tuttavia, anche se potrebbe funzionare, Scheme dispone della funzione '**newline**', che non prevede l'uso di parametri, il cui scopo è quello di fare ciò che serve per ottenere un avanzamento all'inizio della riga successiva.

```
(display "ciao a tutti, sì, proprio a \"tutti\"")
(newline)
```

306.2.2.4 Costanti carattere

In Scheme, i caratteri sono qualcosa di diverso dalle stringhe, ma questo vale anche per altri linguaggi di programmazione. Tuttavia, la rappresentazione di una costante carattere è molto diversa rispetto alle stringhe:

#\carattere	*#\nome_carattere*

Questi caratteri, sempre secondo Scheme, sono oggetti singoli e non possono essere uniti assieme a formare una stringa, a meno di utilizzare delle funzioni apposite di conversione in stringa. Segue un elenco che mostra alcuni esempi di rappresentazione di questi oggetti carattere.

- '**#\a**' -- la lettera «a» minuscola;
- '**#\A**' -- la lettera «A» maiuscola;
- '**#\(**' -- la parentesi tonda aperta;
- '**#\ **' -- lo spazio (dopo la barra obliqua inversa c'è esattamente un carattere *<SP>*;
- '**#\space**' -- lo spazio, espresso per nome;
- '**#\newline**' -- il codice di interruzione di riga.

306.2.3 Espressioni

Un'espressione è qualcosa che, per mezzo di una valutazione, fa qualcosa, oppure restituisce un qualche valore, o fa tutte e due le cose. Le espressioni sono cose che riguardano praticamente tutti i linguaggi di programmazione, ma Scheme ha una gestione particolare quando si vuole evitare che qualcosa venga trasformato da una valutazione.

In pratica, in Scheme si distinguono le *espressioni letterali*, che sono delle espressioni che per qualche ragione, non devono essere elaborate nel modo consueto, ma passate così come sono in modo letterale.

306.2.3.1 Riferimenti variabili

Nella filosofia di Scheme non si hanno delle variabili vere e proprie, ma degli identificatori che fanno riferimento a delle zone di memoria allocate. Tuttavia, si può usare ugualmente il termine «variabile», se si fa attenzione a ricordare la particolarità di Scheme.

La valutazione di una variabile in Scheme genera la restituzione del valore contenuto nell'area di memoria a cui questa punta. Se si usa un interprete Scheme, come quelli descritti nel capitolo introduttivo di questa parte, si può osservare quanto descritto in modo molto semplice:

```
(define x 195)
x                       ===> 195
```

In pratica, l'espressione banale che consiste nell'indicare semplicemente l'identificatore di una variabile, genera la restituzione del valore che in precedenza gli è stato assegnato.

306.2.3.2 Espressioni letterali

In un linguaggio di programmazione qualunque, le espressioni letterali corrispondono alle costanti letterali, come i numeri, le stringhe e oggetti simili. In Scheme si aggiungono anche altri oggetti.

costante
' *dato*
(quote *dato*)

A parte le costanti letterali normali, le altre espressioni letterali si distinguono per essere precedute da un apostrofo iniziale (‘**'**’), oppure (ed è la stessa cosa), per essere indicate come argomento della funzione ‘**quote**’.

Inizialmente è difficile comprendere il senso di questa notazione. Tuttavia, è importante riconoscere subito che non si tratta di stringhe, in quanto lo scopo per il quale esistono queste espressioni letterali, è proprio quello di evitare che vengano valutate prima del necessario. Si osservino gli esempi seguenti, divisi su tre colonne, allo scopo di facilitarne il confronto. In particolare, si suppone che esista una variabile ‘**a**’ che faccia riferimento a una zona di memoria contenente il valore uno.

```
(quote a)               ===> a «simbolo»
'a                      ===> a «simbolo»
a                       ===> 1

(quote (+ 1 2))         ===> (+ 1 2)
'(+ 1 2)                ===> (+ 1 2)
(+ 1 2)                 ===> 3

(quote (quote a))       ===> (quote a)
''a                     ===> (quote a)
'a                      ===> a «simbolo»

(quote "a")             ===> "a" «stringa»
'"a"                    ===> "a" «stringa»
"a"                     ===> "a" «stringa»

(quote 1)               ===> 1
'1                      ===> 1
1                       ===> 1

(quote #t)              ===> #t
'#t                     ===> #t
#t                      ===> #t

(quote #\a)             ===> #\a «carattere»
'#\a                    ===> #\a «carattere»
#\a                     ===> #\a «carattere»
```

Nei primi esempi si fa riferimento a qualcosa che si identifica attraverso la lettera «a». ‘**(quote a)**’, ovvero ‘**'a**’, non è un carattere e non è una stringa: è un simbolo non meglio identificato; dipende dal programmatore il significato che questo può avere. Per semplificare le cose, si è immaginato che si trattasse di una variabile.

Tra gli esempi si vede la possibilità di indicare una funzione per la somma, ‘**(+ 1 2)**’, come espressione costante. Ci sono situazioni in cui questo è necessario, per esempio quando una funzione deve essere passata come argomento di un’altra, mentre lo scopo non è quello di passare il risultato della valutazione della prima.

Le costanti letterali, come le stringhe, i numeri, i caratteri e i valori booleani, possono essere indicati come espressioni letterali; in tal modo il risultato non cambia, dal momento che la valutazione di tali costanti restituisce le costanti stesse.

Ci sono altri tipi di dati che possono essere indicati in forma di espressioni letterali, ma non sono stati mostrati gli esempi relativi perché questi tipi non sono ancora stati descritti. Tuttavia, il senso non cambia: si usano le espressioni letterali quando non si può lasciare che queste siano valutate.

306.2.3.3 Ordine nella valutazione di un'espressione

L'ordine in cui viene valutata un'espressione è relativamente semplice in Scheme, dal momento che non si utilizzano operatori simbolici e tutto è espresso in forma di funzioni. In generale, si valuta prima ciò che sta nella posizione più «interna», venendo mano a mano verso l'esterno. Per esempio,

```
(* 3 (+ 2 4))
```

si risolve secondo la sequenza di operazioni elencate di seguito:

- '**3**' ===> '**3**'
- valutazione di '**(+ 2 4)**'
  - '**2**' ===> '**2**'
  - '**4**' ===> '**4**'
  - '**2+4**' ===> '**6**'
- '**3*6**' ===> '**18**'

306.3 Funzioni comuni nelle espressioni e particolarità di alcuni tipi di dati elementari

Nei linguaggi di programmazione comuni, le espressioni si avvalgono prevalentemente di operatori di vario tipo, tanto che gli operatori sono di per sé delle funzioni, più o meno celate. Con Scheme, questa ambiguità viene eliminata, dal momento che tutte le operazioni di un'espressione si svolgono per mezzo di funzioni. Le funzioni che vengono descritte in queste sezioni, sono quelle che vengono utilizzate più frequentemente nelle espressioni di Scheme.

Il valore restituito da una funzione può essere di tipo diverso a seconda degli operandi utilizzati. Di solito si fa l'esempio della somma di due interi che genera un risultato intero. Scheme ha una gestione particolare dei numeri, almeno a livello teorico, per cui questi vengono classificati in modo molto più sofisticato di quanto facciano i linguaggi di programmazione normali.[1]

306.3.1 Numeri

Con Scheme, i numeri sono gestiti a due livelli differenti: l'astrazione matematica e la realizzazione pratica. Dal punto di vista dell'astrazione matematica, si distinguono i livelli seguenti:

- numero generico;
- numero complesso;

[1] Nella sezione dedicata ai numeri, è assente la spiegazione riguardo al tipo numerico «complesso». Questo dipende dalla mancanza di preparazione dell'autore al riguardo. Eventualmente si può consultare il documento *R^{5}RS* in cui questo argomento è affrontato.

- numero reale;
- numero razionale;
- numero intero.

In generale, un numero che appartiene a una classe inferiore, è anche un numero che può essere considerato appartenente a tutti i livelli superiori. Per esempio, un numero razionale è anche un numero reale ed è anche un numero complesso, ecc.

Scheme fornisce una serie di predicati (funzioni), per la verifica dell'appartenenza di un valore a un tipo di numero. L'elenco si vede nella tabella 306.2. In generale, queste funzioni restituiscono il valore *Vero* (‘`#t`’) nel caso in cui sia valida l'appartenenza presunta.

Tabella 306.2. Elenco dei predicati utili per verificare l'appartenenza ai vari tipi numerici.

Predicato	Descrizione
(number? *espressione*)	*Vero* se l'espressione dà un risultato numerico di qualunque tipo.
(complex? *espressione*)	*Vero* se l'espressione dà come risultato un numero complesso.
(real? *espressione*)	*Vero* se l'espressione dà come risultato un numero reale.
(rational? *espressione*)	*Vero* se l'espressione dà come risultato un numero razionale.
(integer? *espressione*)	*Vero* se l'espressione dà come risultato un numero intero.

Nel modo in cui si rappresenta un numero si indica implicitamente il tipo di questo. Tuttavia, se Scheme è in grado di conoscere una semplificazione nel modo di rappresentarne il valore, lo classifica automaticamente nella fascia inferiore relativa. Per esempio, se 4/2 viene mostrato come numero razionale, dal momento che è equivalente a due, è anche un intero puro e semplice. Gli esempi seguenti mostrano in che modo possono reagire i predicati per la verifica del tipo numerico. Si osservi in particolare la disponibilità della notazione m/n, che permette di indicare agevolmente i numeri razionali.

```
(integer? 3)            ===> #t
(rational? 3)           ===> #t
(real? 3)               ===> #t
(complex? 3)            ===> #t
(number? 3)             ===> #t

(integer? 6/2)          ===> #t
(integer? 3/2)          ===> #f
(rational? 6/2)         ===> #t
(rational? 3/2)         ===> #t

(integer? 1.1)          ===> #f
(rational? 1.1)         ===> #t (dipende dalla realizzazione di Scheme)
(real? 1.1)             ===> #t
```

Secondo Scheme, i numeri sono *esatti* o *inesatti*, a seconda di varie circostanze, che possono dipendere anche dalla realizzazione che si utilizza. In generale, un numero è esatto se è stato fornito attraverso una costante che di per sé è esatta (come un numero intero o un numero razionale), oppure se deriva da numeri esatti utilizzati in operazioni esatte. Si comprende intuitivamente che nel momento in cui si introducono approssimazioni di qualche tipo, per qualche ragione, i valori che si ottengono dai calcoli che si fanno, non sono precisi, ma sono, appunto, inesatti. Nonostante sia molto facile generare risultati inesatti, anche quando si parte da valori esatti, ci sono alcune situazioni in cui i risultati sono esatti anche se i valori di partenza sono inesatti; per esempio, la moltiplicazione per uno zero esatto, genera uno zero esatto, qualunque sia l'altro valore. A proposito dell'esattezza o meno dei valori, sono disponibili alcune funzioni che sono elencate nella tabella 306.3.

Tabella 306.3. Elenco dei predicati e delle altre funzioni riferite ai valori esatti e inesatti.

Funzione	Descrizione
(exact? *espressione*)	*Vero* se l'espressione dà un risultato numerico esatto.
(inexact? *espressione*)	*Vero* se l'espressione dà un risultato numerico inesatto.
(exact->inexact *espressione*)	Converte il risultato dell'espressione in un valore numerico inesatto.
(inexact->exact *espressione*)	Converte il risultato dell'espressione in un valore numerico esatto.

Seguono alcuni esempi sull'uso di queste funzioni.

```
(exact? 3)              ===> #t
(exact? 3/2)            ===> #t
(exact? 1.5)            ===> #f
(exact->inexact 3)      ===> 3.0
(inexact->exact 1.5)    ===> 3/2
```

Come accennato all'inizio, oltre all'astrazione matematica si pone il problema della precisione dei valori inesatti (quelli che per altri linguaggi di programmazione sono semplicemente dei valori a virgola mobile). Ammesso che la realizzazione di Scheme permetta di distinguere tra diversi livelli di precisione, si possono rappresentare delle costanti numeriche «reali» (a virgola mobile), utilizzando la notazione esponenziale, dove al posto della lettera «e» consueta, si utilizzano rispettivamente le lettere, '**s**', '**f**', '**d**' e '**l**', che indicano valori a precisione ridotta (*short*), a singola precisione (*float*), a doppia precisione (*double*) e a precisione ancora maggiore (*long*).

Tabella 306.4. Elenco delle funzioni matematiche comuni.

Funzione	Descrizione
(+ *op*...)	Somma gli argomenti.
(* *op*...)	Moltiplica gli argomenti.
(- *op*)	Moltiplica il valore dell'operando per -1.
(- *op1 op2*...)	Sottrae dal primo la somma degli operandi successivi.
(/ *op*)	Divide il primo operando per 1.
(/ *op1 op2*...)	Divide il primo operando per il secondo, divide il risultato per il terzo...
(log *op*)	Calcola il logaritmo naturale.
(exp *op*)	Calcola l'esponente.
(sin *op*)	Calcola il seno.
(cos *op*)	Calcola il coseno.
(tan *op*)	Calcola la tangente.
(asin *op*)	Calcola l'arco-seno.
(acos *op*)	Calcola l'arco-coseno.
(atan *op*)	Calcola l'arco-tangente.
(sqrt *op*)	Calcola la radice quadrata.
(expt *op1 op2*)	Eleva il primo operando alla potenza del secondo.
(abs *op*)	Calcola il valore assoluto.
(quotient *op1 op2*)	Divide il primo operando per il secondo e restituisce il valore intero.
(remainder *op1 op2*)	Resto della divisione del primo operando per il secondo.
(modulo *op1 op2*)	Calcola il modulo (vedere nota).
(ceiling *op*)	Calcola la parte intera per eccesso.
(floor *op*)	Calcola la parte intera per difetto.
(round *op*)	Calcola la parte intera più vicina.
(truncate *op*)	Calcola la parte intera eliminando semplicemente la parte decimale.
(max *op*...)	Restituisce il valore massimo dei suoi operandi.
(min *op*...)	Restituisce il valore minimo dei suoi operandi.
(gcd *n_intero* ...)	Calcola il massimo comune divisore dei vari operandi.
(lcd *n_intero* ...)	Calcola il minimo comune multiplo dei vari operandi.
(numerator *n_razionale*)	Restituisce il numeratore di un numero razionale.
(denomiator *n_razionale*)	Restituisce il denominatore di un numero razionale.

La tabella 306.4 riporta l'elenco delle funzioni più comuni che possono essere usate nelle espressioni aritmetiche e matematiche. In particolare si deve osservare che '**remainder**' e '**modulo**' si comportano nello stesso modo, tranne quando si utilizzano valori negativi (per approfondire la differenza si può leggere il documento di riferimento su Scheme, ovvero *R*^{5}*RS*).

Scheme permette di utilizzare più di due operandi per le funzioni che sommano, sottraggono, dividono e moltiplicano. A parte la spiegazione sintetica data nella tabella in cui sono state presentate, si può intendere il senso del loro funzionamento immaginando che le operazioni avvengono in modo progressivo, da sinistra a destra:

```
(- 5 3 2)
```

equivale a:

```
(- (- 5 3) 2)
```

Nello stesso modo,

```
(/ 5 3 2)
```

equivale a:

```
(/ (/ 5 3) 2)
```

Infine, la tabella 306.5 riporta alcuni predicati utili per classificare in qualche modo un valore numerico.

Tabella 306.5. Elenco di altri predicati utili per classificare i valori numerici.

Funzione	Descrizione
(zero? *op*)	*Vero* se l'operando equivale a zero.
(positive? *op*)	*Vero* se l'operando è un numero positivo.
(negative? *op*)	*Vero* se l'operando è un numero negativo.
(odd? *op*)	*Vero* se l'operando è un numero dispari.
(even? *op*)	*Vero* se l'operando è un numero pari.

Scheme dispone di altre risorse per la gestione dei valori numerici; inoltre, ciò che è stato presentato qui è descritto in modo approssimativo. Se si vogliono sfruttare bene tali possibilità di questo linguaggio, è indispensabile studiare bene il documento *R*^{5}*RS*, già citato più volte, del quale si trova un riferimento alla fine del capitolo.

306.3.2 Valori logici, funzioni di confronto e funzioni logiche

Sono già state presentate le costanti booleane '**#t**' e '**#f**', che valgono per *Vero* e *Falso* rispettivamente. Per Scheme, da un punto di vista logico-booleano, valgono come *Vero* anche le liste (che verranno descritte in seguito), compresa la lista vuota, i simboli, i numeri, le stringhe, i vettori e le funzioni. In pratica, qualsiasi oggetto diverso dal tipo booleano, assieme al valore booleano '**#t**', vale come *Vero*, mentre solo '**#f**' vale per *Falso*. Tuttavia, per verificare che un oggetto corrisponda effettivamente a un valore booleano, si può usare il predicato

```
(boolean? oggetto)
```

che restituisce *Vero* in caso affermativo.

Alcune realizzazioni più vecchie di Scheme trattano la lista vuota, che si rappresenta con '**()**', come equivalente al valore booleano *Falso*.

Gli operatori logici sono realizzati in Scheme attraverso funzioni. La tabella 306.6 elenca queste funzioni.

Tabella 306.6. Elenco delle funzioni logiche.

Funzione	Descrizione
(not *op*)	Inverte il risultato logico dell'operando.
(and *op1 op2...*)	*Vero* se tutti gli operandi restituiscono *Vero*.
(or *op1 op2...*)	*Vero* se anche solo un operando restituisce *Vero*.

Per quanto riguarda il confronto, si distinguono situazioni diverse, a seconda che si vogliano confrontare dei valori numerici, carattere, stringa, oppure che si vogliano confrontare gli «oggetti». Le tabelle 306.7, 306.8, 306.9 e 306.10, riepilogano le funzioni in grado di eseguire tali confronti.

Tabella 306.7. Elenco delle funzioni per il confronto numerico.

Funzione	Descrizione
(= *op1 op2...*)	*Vero* se gli operandi si equivalgono.
(< *op1 op2...*)	*Vero* se gli operandi sono in ordine crescente.
(> *op1 op2...*)	*Vero* se gli operandi sono in ordine decrescente.
(<= *op1 op2...*)	*Vero* se gli operandi sono in ordine non decrescente.
(>= *op1 op2...*)	*Vero* se gli operandi sono in ordine non crescente.

È interessante notare che le funzioni per il confronto ammettono l'uso di più di due argomenti. Si osservino gli esempi seguenti, con i risultati che restituiscono.

```
(= 2 2)                 ===> #t
(= 2 2 2)               ===> #t
(= 2 2 2 1)             ===> #f
(< 1 2)                 ===> #t
(< 1 2 3)               ===> #t
(< 1 2 3 2)             ===> #f
```

Tabella 306.8. Elenco delle funzioni per il confronto tra caratteri.

Funzione	Descrizione
(char=? *car1 car2*)	*Vero* se i due caratteri sono uguali.
(char<? *car1 car2*)	*Vero* se il primo carattere è lessicograficamente inferiore al secondo.
(char>? *car1 car2*)	*Vero* se il primo carattere è lessicograficamente superiore al secondo.
(char<=? *car1 car2*)	*Vero* se il primo carattere è lessicograficamente non superiore al secondo.
(char>=? *car1 car2*)	*Vero* se il primo carattere è lessicograficamente non inferiore al secondo.
(char-ci=? *car1 car2*)	Come '**`char=?`**', senza distinguere tra maiuscole e minuscole.
(char-ci<? *car1 car2*)	Come '**`char<?`**', senza distinguere tra maiuscole e minuscole.
(char-ci>? *car1 car2*)	Come '**`char>?`**', senza distinguere tra maiuscole e minuscole.
(char-ci<=? *car1 car2*)	Come '**`char<=?`**', senza distinguere tra maiuscole e minuscole.
(char-ci>=? *car1 car2*)	Come '**`char>=?`**', senza distinguere tra maiuscole e minuscole.

Per quanto riguarda il confronto tra caratteri e tra stringhe, non è stabilita la possibilità di inserire più di due argomenti, anche se è possibile che una realizzazione Scheme lo consenta.

```
(char<? #\a #\b)        ===> #t
(char<? #\A #\B)        ===> #t
(char-ci<=? #\A #\b)    ===> #t
(char-ci<=? #\a #\B)    ===> #t
(char-ci=? #\a #\A)     ===> #t
```

Tabella 306.9. Elenco delle funzioni per il confronto tra stringhe.

Funzione	Descrizione
(string=? *str1 str2*)	*Vero* se le due stringhe sono uguali.
(string<? *str1 str2*)	*Vero* se la prima stringa è lessicograficamente inferiore alla seconda.
(string>? *str1 str2*)	*Vero* se la prima stringa è lessicograficamente superiore alla seconda.
(string<=? *str1 str2*)	*Vero* se la prima stringa è lessicograficamente non superiore alla seconda.
(string>=? *str1 str2*)	*Vero* se la prima stringa è lessicograficamente non inferiore alla seconda.
(string-ci=? *str1 str2*)	Come '**`string=?`**', senza distinguere tra maiuscole e minuscole.
(string-ci<? *str1 str2*)	Come '**`string<?`**', senza distinguere tra maiuscole e minuscole.
(string-ci>? *str1 str2*)	Come '**`string>?`**', senza distinguere tra maiuscole e minuscole.
(string-ci<=? *str1 str2*)	Come '**`string<=?`**', senza distinguere tra maiuscole e minuscole.
(string-ci>=? *str1 str2*)	Come '**`string>=?`**', senza distinguere tra maiuscole e minuscole.

```
(string<? "ab" "aba")           ===> #t
(string<? "AB" "ABA")           ===> #t
(string-ci<? "AB" "aba")        ===> #t
(string-ci<? "ab" "ABA")        ===> #t
(string-ci=? "ciao" "CIAO")     ===> #t
```

Scheme offre dei predicati particolari per il confronto tra due oggetti, come mostrato nella tabella 306.10. È difficile definire in modo chiaro la differenza che c'è tra questi tre predicati. In generale si può affermare che '**`equal?`**' sia il predicato che è più permissivo, mentre '**`eq?`**' è quello più restrittivo.

Tabella 306.10. Elenco delle funzioni per il confronto tra gli oggetti.

Funzione	Descrizione
(eq? *op1 op2*)	*Vero* se i due operandi sono identici.
(eqv? *op1 op2*)	*Vero* se i due operandi sono equivalenti dal punto di vista operativo.
(equal? *op1 op2*)	*Vero* se i due operandi hanno la stessa struttura e lo stesso contenuto.

```
(equal? "abc" "abc")            ===> #t
(eqv? "abc" "abc")              ===> #f
(eq? "abc" "abc")               ===> #f

(equal? 2 2)                    ===> #t
(eqv? 2 2)                      ===> #t
(eq? 2 2)                       ===> (non specificato)

(equal? 'a 'a)                  ===> #t
(eqv? 'a 'a)                    ===> #t
(eq? 'a 'a)                     ===> #t
```

306.3.3 Caratteri

Alcune funzioni specifiche per i caratteri sono elencate nella tabella 306.11. Per quanto riguarda il caso particolare del predicato '**`char-whitespace?`**', questo si avvera nel caso in cui si tratti di <*SP*>, <*HT*>, <*LF*>, <*FF*> e <*CR*>.

Tabella 306.11. Elenco di alcune funzioni specifiche per la gestione dei caratteri.

Funzione	Descrizione
(char? *oggetto*)	*Vero* se l'oggetto è un carattere.
(char-alphabetic? *carattere*)	*Vero* se il carattere è alfabetico.
(char-numeric? *carattere*)	*Vero* se il carattere è numerico.
(char-whitespace? *carattere*)	*Vero* se si tratta di uno spazio orizzontale o verticale.
(char-upper-case? *carattere*)	*Vero* se si tratta di un carattere alfabetico maiuscolo.
(char-lower-case? *carattere*)	*Vero* se si tratta di un carattere alfabetico minuscolo.
(char->integer *carattere*)	Restituisce un numero corrispondente al carattere.
(integer->char *numero_intero*)	Restituisce un carattere corrispondente al numero.
(char-upcase *carattere*)	Se possibile, converte il carattere in maiuscolo.
(char-downcase *carattere*)	Se possibile, converte il carattere in minuscolo.

Nella conversione attraverso le funzioni '**`char->integer`**' e '**`integer->char`**', l'equivalenza tra carattere e numero dipende dalla realizzazione di Scheme; molto probabilmente dipenderà dalla codifica dell'insieme di caratteri utilizzato.

306.3.4 Stringhe

Alcune funzioni specifiche per i caratteri sono elencate nella tabella 306.12. Quando le funzioni fanno riferimento a un indice per indicare un carattere all'interno di una stringa, si deve ricordare che il primo corrisponde alla posizione zero. Quando si fa riferimento a due indici, uno per indicare il carattere iniziale e uno per fare riferimento al carattere finale, il secondo indice deve puntare alla posizione successiva all'ultimo carattere da prendere in considerazione. Questo permette di individuare una stringa nulla quando l'indice iniziale e l'indice finale sono uguali.

Tabella 306.12. Elenco di alcune funzioni specifiche per la gestione delle stringhe.

Funzione	Descrizione
(string? *oggetto*)	*Vero* se l'oggetto è una stringa.
(make-string *numero_caratteri*)	Restituisce una stringa della lunghezza indicata.
(make-string *numero_caratteri carattere*)	Restituisce una stringa composta con il carattere indicato.
(string *carattere*...)	Restituisce una stringa composta dai caratteri indicati.
(string-length *stringa*)	Restituisce il numero di caratteri contenuto.
(string-ref *stringa indice*)	Restituisce il carattere nella posizione dell'indice.
(string-set! *stringa indice carattere*)	Modifica il carattere che si trova nella posizione dell'indice.
(substring *stringa inizio fine*)	Estrae la sottostringa compresa tra i due indici.
(string-append *stringa*...)	Restituisce una stringa unica complessiva.
(string-copy *stringa*)	Restituisce una copia della stringa.
(string-fill! *stringa carattere*)	Sostituisce gli elementi della stringa con il carattere indicato.
(string->list *stringa*)	Restituisce una lista composta dai caratteri della stringa.
(list->string *lista_di_caratteri*)	Restituisce una stringa a partire da una lista di caratteri.

```
(make-string 10 #\A)            ===> "AAAAAAAAAA"
(string-length "ciao")          ===> 4

(define a "ciao")
(string-set! a 0 #\C)
a                               ===> "Ciao"
(substring a 2 4)               ===> "ao"
```

306.4 Strutture di controllo

Anche con Scheme sono disponibili le strutture di controllo comuni nei linguaggi di programmazione. Evidentemente, queste sono realizzate attraverso delle funzioni. In base a tale impostazione, per sottoporre una parte di codice alla verifica di una condizione, o per metterla in un ciclo, occorre che questa sia inserita in una funzione che possa essere chiamata all'interno di un'espressione.

Per intendere il problema, si osservi l'esempio seguente, che mostra la scelta tra la chiamata della funzione '**`display`**' per visualizzare il messaggio «bello», o «brutto», in funzione di una condizione (che in questo caso si avvera necessariamente):

```
(if (> 3 2) (display "bello") (display "brutto"))
```

Per ovviare a questo inconveniente si può utilizzare la funzione '**`begin`**', che permette di incorporare più espressioni dove invece se ne potrebbe inserire una sola.

306.4.1 begin

Per tutte le situazioni in cui è possibile indicare una sola espressione, mentre invece se ne vorrebbero inserire diverse, esiste la funzione '**`begin`**':

```
(begin
     espressione
     espressione
     ...
)
```

Il senso si comprende intuitivamente: le espressioni che costituiscono gli argomenti di '**`begin`**' vengono valutati in ordine, da sinistra a destra (in questo caso dall'alto in basso). L'esempio seguente è molto banale: visualizza un messaggio e termina.

```
(begin
    (display "ciao ")
    (display "a ")
    (display "tutti!")
    (newline)
)
```

> È importante osservare che all'interno della funzione '**`begin`**' non è possibile dichiarare delle variabili locali, a meno che per questo si inseriscano delle altre funzioni che creano un loro ambiente, come '**`let`**' e le altre simili.

306.4.2 if

La struttura condizionale è il sistema di controllo fondamentale dell'andamento del flusso delle istruzioni.

```
(if condizione espressione_se_vero [espressione_se_falso])
```

La funzione '**`if`**' valuta i suoi argomenti in un ordine preciso: per prima cosa viene valutato il primo argomento; se il risultato è *Vero*, o comunque se si ottiene un risultato equiparabile a *Vero*, valuta il secondo argomento; in alternativa, valuta il terzo argomento, se è stato fornito. Alla fine restituisce il valore dell'ultima espressione a essere stata valutata (ammesso che questa restituisca qualcosa). Sotto vengono mostrati alcuni esempi in cui alcune parti del programma sono state saltate per non distrarre l'attenzione del lettore.

```
(define Importo 0)
...
(if (> Importo 10000000) (display "L'offerta è vantaggiosa"))
```

```
(define Importo 0)
...
(if (> Importo 10000000)
    (display "L'offerta è vantaggiosa")
    (display "Lascia perdere")
)
```

```
(define Importo 0)
...
(if (> Importo 10000000)
    (display "L'offerta è vantaggiosa")
    (if (> Importo 5000000)
        (display "L'offerta è accettabile")
        (display "Lascia perdere")
    )
)
```

Come accennato, potrebbe essere conveniente l'utilizzo della funzione '**`begin`**' per facilitare la descrizione di gruppi di istruzioni (espressioni). Si osservi l'esempio seguente, in cui viene salvato il valore dell'importo nella variabile '**`Offerta`**':

```
(define Importo 0)
(define Offerta 0)
...
(if (> Importo 10000000)
    ; then
    (begin
        (display "L'offerta è vantaggiosa")
        (set! Offerta Importo)
        ; eventualmente fa anche qualcosa in più
        ;...
    )
    ; else
    (if (> Importo 5000000)
        ; then
        (begin
            (display "L'offerta è accettabile")
            (set! Offerta Importo)
            ; eventualmente fa anche qualcosa in più
            ;...
        )
        ; else
        (display "Lascia perdere")
    ; end if
    )
; end if
)
```

306.4.3 cond

Scheme fornisce due strutture di selezione. In questo caso, la funzione '**`cond`**' si basa sulla verifica di condizioni distinte per ogni blocco di espressioni.

```
(cond
    (condizione  espressione...)
    (condizione  espressione...)
    ...
    [(else  espressione...)]
)
```

Lo schema sintattico dovrebbe essere chiaro a sufficienza: la funzione '**cond**' ha come argomenti una serie di «blocchi» (si tratta di liste, ma questo verrà chiarito quando verranno mostrate le liste), contenenti ognuno un'espressione iniziale che deve essere valutata per determinare se le espressioni successive devono essere valutate o meno. Nel momento in cui si incontra una condizione che si avvera, i blocchi successivi vengono ignorati. Se non si incontra alcuna condizione che si avvera, se esiste l'ultimo blocco, corrispondente alla funzione '**else**', le espressioni relative vengono eseguite.

A differenza della funzione '**if**', in questo caso si possono indicare più espressioni per ogni condizione della selezione; in questo senso, la funzione '**cond**' può diventare un sostituto opportuno di quella. Segue un esempio tipico di selezione.

```
(define Mese 0)
...
(cond
    ((= Mese 1) (display "gennaio") (newline))
    ((= Mese 2) (display "febbraio") (newline))
    ((= Mese 3) (display "marzo") (newline))
    ((= Mese 4) (display "aprile") (newline))
    ((= Mese 5) (display "maggio") (newline))
    ((= Mese 6) (display "giugno") (newline))
    ((= Mese 7) (display "luglio") (newline))
    ((= Mese 8) (display "agosto") (newline))
    ((= Mese 9) (display "settembre") (newline))
    ((= Mese 10) (display "ottobre") (newline))
    ((= Mese 11) (display "novembre") (newline))
    ((= Mese 12) (display "dicembre") (newline))
    (else (display "mese errato!") (newline))
)
```

306.4.4 case

Scheme fornisce anche la struttura di selezione tradizionale, ovvero la funzione '**case**', che si basa sulla verifica del valore di una sola «chiave». Anche '**case**' permette l'indicazione di più espressioni per ogni elemento della selezione.

```
(case  espressione_di_selezione
    ((dato...)  espressione...)
    ((dato...)  espressione...)
    ...
    [(else  espressione...)]
)
```

La prima espressione a essere valutata è quella che costituisce il primo argomento della funzione '**case**'. Successivamente, il suo risultato viene comparato con quello dei «dati» elencati all'inizio di ogni gruppo di espressioni (si vedano gli esempi). Se la comparazione ha successo, allora vengono valutate le espressioni successive (all'interno del blocco), nell'ordine in cui si trovano. Se il confronto non ha successo, se esiste un blocco finale costituito dalla funzione '**else**', vengono eseguite le espressioni relative. Seguono alcuni esempi.

```
(define Mese 0)
...
(case Mese
    ((1) (display "gennaio") (newline))
    ((2) (display "febbraio") (newline))
```

```
    ((3) (display "marzo") (newline))
    ((4) (display "aprile") (newline))
    ((5) (display "maggio") (newline))
    ((6) (display "giugno") (newline))
    ((7) (display "luglio") (newline))
    ((8) (display "agosto") (newline))
    ((9) (display "settembre") (newline))
    ((10) (display "ottobre") (newline))
    ((11) (display "novembre") (newline))
    ((12) (display "dicembre") (newline))
    (else (display "mese errato!") (newline))
)
```

```
(define Anno 0)
(define Mese 0)
(define Giorni 0)
...
(case Mese
    ((1 3 5 7 8 10 12) (set! Giorni 31))
    ((4 6 9 11) (set! Giorni 30))
    ((2)
        (if
            (or
                (and (= (modulo Anno 4) 0) (not (= (modulo Anno 100) 0)))
                (= (modulo Anno 400) 0)
            )
            (set! Giorni 29)
            (set! Giorni 28)
        )
    )
)
```

306.4.5 do

Scheme dispone di una funzione unica per realizzare i cicli iterativi e quelli enumerativi. Si tratta di '**do**', il cui funzionamento è, a prima vista, un po' strano. Come ciclo iterativo la sintassi si riduce al modello seguente:

```
(do ()
    (condizione_di_uscita [espressione_pre_uscita...])
    espressione_del_ciclo...
)
```

In questa forma, viene valutata prima la condizione; se si avvera, vengono valutate le espressioni successive, quelle contenute nello spazio delle parentesi (la lista della condizione), quindi il ciclo termina. Se la condizione non si avvera, vengono eseguite le espressioni esterne al blocco della condizione, al termine delle quali riprende il ciclo.

Quando si vuole usare la funzione '**do**' per realizzare un ciclo enumerativo, si definiscono una o più variabili da inizializzare e modificare in qualche modo a ogni ciclo:

```
(do ((variabile inizializzazione passo)...)
    (condizione_di_uscita [espressione_pre_uscita...])
    espressione_del_ciclo...
)
```

Le variabili vengono dichiarate (allocate) dalla funzione '**do**' stessa, avendo effetto solo in ambito locale, all'interno della funzione che le dichiara (in pratica, mascherano temporaneamente altre variabili esterne con lo stesso nome). Le variabili vengono inizializzate immediatamente con il valore ottenuto dall'espressione di inizializzazione, quindi inizia il primo ciclo. Alla fine di ogni ciclo, prima dell'inizio del successivo, vengono valutate le espressioni del passo, assegnando alle variabili relative i valori che si ottengono.

L'esempio seguente fa apparire per 10 volte la lettera «x». Si osservi l'uso di una variabile esterna per scandire i cicli.

```
(define Contatore 0)

(do () ((>= Contatore 10))
    ; incrementa il contatore di un'unità
    (set! Contatore (+ Contatore 1))
    (display "x")
)

(newline)
```

La stessa cosa avrebbe potuto essere ottenuta dichiarando la variabile all'interno della funzione '**do**':

```
(do ((Contatore 0 Contatore))
    ; condizione di uscita
    ((>= Contatore 10))
    ; incrementa il contatore di un'unità
    (set! Contatore (+ Contatore 1))
    (display "x")
)

(newline)
```

Infine, si può trasferire l'incremento del contatore nel blocco in cui si dichiara e si inizializza la variabile '**Contatore**'.

```
(do ((Contatore 0 (+ Contatore 1)))
    ; condizione di uscita
    ((>= Contatore 10))
    ; istruzioni del ciclo
    (display "x")
)

(newline)
```

306.5 Conclusione di un programma Scheme

Un programma Scheme termina quando si esauriscono le istruzioni, oppure quando viene incontrata e valutata la funzione '**exit**'.

(exit [*valore_di_uscita*])

Come si vede dallo schema sintattico, è possibile indicare un numero che si traduce poi nel valore di uscita del programma stesso.

L'utilizzo di questa funzione all'interno di un ambiente di interpretazione Scheme, serve normalmente a concludere il funzionamento del programma relativo.

306.6 Riferimenti

- A. Aaby, *Scheme Tutorial*, 1996

  <http://cs-sa1.wwc.edu/~cs_dept/KU/PR/Scheme.html>

- Pierre Castéran, Robert Cori, *Passeport pour Scheme*

  Il documento citato sembra essere scomparso dalla rete, probabilmente in vista di una sua pubblicazione. In origine, si trovava presso *<http://dept-info.labri.u-bordeaux.fr/~cori/Bouquins/scheme.ps>*.

- *R^{5}RS -- Revised-5 Report on the Algorithmic Language Scheme*, 1998

  <*http://www.swiss.ai.mit.edu/~jaffer/r5rs_toc.html*>

  <*http://www.swiss.ai.mit.edu/ftpdir/scheme-reports/r5rs.ps.gz*>