Parte xxix

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 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.

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
```
[*Invio*]

```
# cd dist-7.5
```
[*Invio*]

```
# ./install.sh
```
[*Invio*]

Nel sito in cui viene distribuito questo interprete, si trova anche la documentazione per il suo utilizzo. Qui si intende mostrare solo l'essenziale.

Utilizzo interattivo

«

Per avviare l'interprete Scheme del MIT, basta il comando seguente:

```
$ scheme
```
[*Invio*]

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 ]=> (exit) [Invio]

Kill Scheme (y or n)? y

  Happy Happy Joy Joy.
```

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'esem-

pio 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.

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.

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 [Invio]
```

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") [Invio]
```

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") [Invio]

1 ]=> (cf "ciao_mondo.scm" "ciao_mondo.com") [Invio]
```

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
```

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 u122). 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/java/kawa/repl.class`'. Eventualmente, dovendo fare a meno di questo script, basterebbe il comando seguente:

```
$ java kawa.repl [Invio]
```

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
```

Utilizzo interattivo

«

Per avviare l'interprete Scheme di Kawa, basta il comando seguente:

```
$ kawa [Invio]
```

Oppure, in mancanza di questo script:

```
$ java kawa.repl [Invio]
```

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]
```

Lo script '**kawa**', ovvero il comando '**java kawa.repl**', permette l'utilizzo di alcune opzioni che possono rivelarsi importanti.

```
kawa [opzioni]
```

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.

Tabella u126.11. Alcune opzioni.

Opzione	Descrizione
`-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**'.

Opzione	Descrizione
`--main`	Assieme all'opzione '**`-C`**', consente di generare un binario Java, autonomo.

Segue la descrizione di alcuni esempi.

- `$` **`kawa -c '(display "Ciao mondo!") (newline)'`** [*Invio*]

  Visualizza il messaggio «Ciao mondo!», senza prendere in considerazione il file di configurazione.

- `$` **`kawa -f ciao_mondo.scm`**[*Invio*]

  Esegue il contenuto del file '`ciao_mondo.scm`', che si presume essere un sorgente Scheme.

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 seguente:

```
(compile-file nome_sorgente radice_destinazione)
```

Con questa si può fare qualcosa del genere:

`#|kawa:`***`m`***`|#` **`(compile-file "ciao_mondo.scm" "ciao")`** [*Invio*]

In tal 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`**[*Invio*]

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[Invio]
```

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
```

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/

[1] **MIT Scheme** GNU GPL

[2] **Kawa** GNU GPL, ma con meno restrizioni

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.

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 cerca 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 utilizza il simbolo '**===>**' per indicare il valore restituito da una funzione (che appare alla sua destra).

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.

```
(+ 3 4)
```

Come è già stato fatto osservare, l'esempio mostra 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
```

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.

Tipi di dati

«

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])
```

L'esempio seguente, alloca l'area di memoria necessaria a contenere un numero intero, quindi abbina all'identificatore '**x**' il puntatore a questa area.

```
(define x 1)
```

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.

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 u127.9. 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.

Predicato	Descrizione
`(port?` ***espressione***`)`	*Vero* se l'espressione dà come risultato una «porta».
`(procedure?` ***espressione***`)`	*Vero* se l'espressione dà come risultato una funzione.

Costanti letterali

«

Scheme ha una gestione particolare delle espressioni, dove al loro interno è speciale la gestione dei valori costanti. Questo fatto viene chiarito nel seguito. Tuttavia, è necessario conoscere subito in che modo possono essere indicati i valori più comuni in un sorgente Scheme.

Costanti booleane

«

I valori booleani possono essere rappresentati attraverso la sigla '**`#t`**' per *Vero* e '**`#f`**' per *Falso*.

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 ('***`xey`***').

```
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
```

In questo caso, ***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.

Stringhe

Scheme ha una gestione speciale delle espressioni costanti, cosa che viene 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)
```

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.

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.

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.

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; 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.

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.

```
(* 3 (+ 2 4))
```

L'esempio appena mostrato 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`’

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.

Nella sezione dedicata ai numeri, è assente la spiegazione riguardo al tipo numerico «complesso». Eventualmente si può consultare il documento R^5RS in cui questo argomento è affrontato.

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;
- 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 u127.21. In generale, queste funzioni restituiscono il valore *Vero* ('`#t`') nel caso in cui sia valida l'appartenenza presunta.

Tabella u127.21. 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.

Predicato	Descrizione
`(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 u127.25.

Tabella u127.25. 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 u127.27. 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.

Funzione	Descrizione
`(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.

Funzione	Descrizione
`(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.
`(denominator` ***n_razionale*** `)`	Restituisce il denominatore di un numero razionale.

La tabella u127.27 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^5RS).

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)
```

L'esempio appena mostrato equivale a:

```
(- (- 5 3) 2)
```

Nello stesso modo, si osservi l'esempio seguente:

```
(/ 5 3 2)
```

Questo equivale a:

```
(/ (/ 5 3) 2)
```

Infine, la tabella u127.32 riporta alcuni predicati utili per classificare in qualche modo un valore numerico.

Tabella u127.32. 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^5RS, già citato più volte, del quale si trova un riferimento alla fine del capitolo.

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 vengo-

no 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 seguente:

```
(boolean? oggetto)
```

Questo 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 u127.33 elenca queste funzioni.

Tabella u127.33. 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 tabel-

le u127.34, u127.36, u127.38 e u127.40, riepilogano le funzioni in grado di eseguire tali confronti.

Tabella u127.34. 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 u127.36. 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.

Funzione	Descrizione
`(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 u127.38. Elenco delle funzioni per il confronto tra stringhe.

Funzione	Descrizione
`(string=?` ***str1*** ***str2***`)`	*Vero* se le due stringhe sono uguali.

Funzione	Descrizione
`(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 u127.40. 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 u127.40. 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
```

Caratteri

Alcune funzioni specifiche per i caratteri sono elencate nella tabella u127.42. 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 u127.42. Elenco di alcune funzioni specifiche per la gestione dei caratteri.

Funzione	Descrizione
`(char?` ***oggetto***`)`	*Vero* se l'oggetto è un carattere.

Funzione	Descrizione
`(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 dipende dalla codifica dell'insieme di caratteri utilizzato.

Stringhe

«

Alcune funzioni specifiche per i caratteri sono elencate nella tabella u127.43. 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 u127.43. 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.

Funzione	Descrizione
`(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"
```

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.

Funzione «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 valutate 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.

Struttura condizionale: «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
)
```

Struttura di selezione: «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 viene chiarito quando si passa alla descrizione delle 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))
)
```

Struttura di selezione: «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)
        )
    )
)
```

Iterazione: «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 espressio-

ni 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)
```

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.

Riferimenti

«

- A. Aaby, *Scheme Tutorial*, 1996

  http://cs.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

Scheme: struttura del programma e campo di azione

Nel capitolo introduttivo, sono state elencate le strutture elementari per il controllo e il raggruppamento delle istruzioni (espressioni) di Scheme. In questo capitolo, si vuole mostrare in che modo possano essere definite delle funzioni, o comunque dei raggruppamenti di istruzioni all'interno dei quali si possa individuare un campo di azione locale per le variabili che vi vengono dichiarate.

Le funzioni del linguaggio Scheme prevedono il passaggio di parametri solo **per valore**; questo significa che gli argomenti di una funzione vengono valutati prima di tutto. Al posto del passaggio dei parametri per riferimento, Scheme consente l'indicazione di espressioni costanti, concetto a cui si è accennato nel capitolo precedente.

Definizione e campo di azione

La definizione e inizializzazione di un oggetto avviene normalmente attraverso la funzione '**`define`**'. Questa può servire per dichiarare una variabile normale, o anche per dichiarare una funzione.

```
(define nome_variabile espressione_di_inizializzazione)
```

Quello che si vede sopra è appunto lo schema sintattico per la dichiarazione e inizializzazione di una variabile, cosa che è stata vista più volte nel capitolo precedentemente. Sotto, si vede lo schema sintattico per la dichiarazione di una funzione:

```
(define (nome_funzione elenco_parametri_formali)
    corpo
)
```

In questo caso, i parametri formali sono dei nomi che rappresentano i parametri della funzione che viene dichiarata, mentre il corpo è costituito da una serie di espressioni, che rappresentano il contenuto della funzione che si dichiara. Il valore che viene restituito dall'ultima espressione che viene eseguita all'interno della funzione, è ciò che restituisce la funzione stessa. L'esempio seguente, serve a definire la funzione '**moltiplica**' con due parametri, '**x**' e '**y**', che restituisce il prodotto dei suoi due argomenti:

```
(define (moltiplica x y)
    ; il corpo di questa funzione è molto breve
    (* x y)
)
```

Per chiamare questa funzione, basta semplicemente un'istruzione come quella seguente:

```
(moltiplica 10 11)              ===> 110
```

Le dichiarazioni di questo tipo, cioè di variabili e di funzioni, possono avvenire solo nella parte più esterna di un programma Scheme,

oppure all'interno della dichiarazione di altre funzioni e delle altre strutture descritte in questo capitolo, ma in tal caso devono apparire all'inizio del «corpo» delle espressioni che queste strutture contengono. Si osservi l'esempio seguente, in cui viene dichiarata una funzione e al suo interno si dichiarano altre variabili locali:

```
(define (moltiplica x y)
    ; dichiara le variabili locali
    (define z 0)

    ; definisce un ciclo enumerativo, per il calcolo del prodotto
    ; attraverso la somma, in cui viene dichiarata implicitamente
    ; la variabile «i».
    (do ((i 1 (+ i 1)))
        ; condizione di uscita
        ((> i y))
        ; istruzioni del ciclo
        (set! z (+ z x))
    )
    ; al termine restituisce il valore contenuto nella variabile «z»
    z
)
```

Dovrebbe essere intuitivo, quindi, che il campo di azione delle variabili dichiarate all'interno di una funzione '**`define`**' è limitato alla funzione stessa. La stessa cosa varrebbe per le funzioni, dichiarate all'interno di un ambiente del genere. Si osservi l'esempio seguente, in cui si calcola il prodotto tra due numeri, a partire dalla somma di questi, ma dove la somma si ottiene da un'altra funzione, locale, che a sua volta la calcola con incrementi di una sola unità alla volta.

```
(define (moltiplica x y)
    ; dichiara la funzione «somma», locale nell'ambito della
    ; funzione «moltiplica»
    (define (somma x y)
        ; dichiara una variabile locale per la funzione «somma»,
        ; che comunque non serve a nulla :-)
        (define z 2000)
        ; definisce un ciclo enumerativo, per il calcolo della
        ; somma, sommando un'unità alla volta
```

```
        (do ()
            ; condizione di uscita
            ((<= y 0))
            ; istruzioni del ciclo
            (set! x (+ x 1))
            ; decrementa «y»
            (set! y (- y 1))
        )
        ; al termine restituisce il valore contenuto nella variabile «x»
        x
    ; fine della funzione locale «somma»
    )
    ; dichiara le variabili locali della funzione «moltiplica»
    (define z 0)
    ; definisce un ciclo enumerativo, per il calcolo del prodotto
    ; attraverso la somma, in cui viene dichiarata implicitamente
    ; la variabile «i».
    (do ((i 1 (+ i 1)))
        ; condizione di uscita
        ((> i y))
        ; istruzioni del ciclo
        (set! z (somma z x))
    )
    ; al termine restituisce il valore contenuto nella variabile «z»
    z
)
```

Questo esempio è solo un pretesto per mostrare che le variabili locali '**x**', '**y**' e '**z**', della funzione '**somma**' hanno effetto solo nell'ambito di questa funzione; inoltre, la funzione '**somma**' e le variabili locali '**x**', '**y**' e '**z**', della funzione '**moltiplica**', hanno effetto solo nell'ambito della funzione '**moltiplica**' stessa.

Ridefinizione

«

Nel capitolo introduttivo si è accennato al fatto che la ridefinizione di una variabile, o di una funzione, implica una nuova allocazione di memoria, senza liberare quella utilizzata precedentemente. Pertanto, i riferimenti fatti in precedenza a quell'oggetto, continuano

a utilizzare in pratica la vecchia allocazione. Si osservi l'esempio seguente:

```
(define x 20)
x                       ===> 20
(define y (* 2 x))
y                       ===> 40
(define x 100)
x                       ===> 100
y                       ===> 40
```

Quanto mostrato con questo esempio, non ha nulla di eccezionale, rispetto ai linguaggi di programmazione tradizionali. Tuttavia, potrebbe risultare strano da un punto di vista strettamente matematico. Se invece lo scopo fosse quello di definire un sistema di equazioni, '**y**' dovrebbe essere trasformato in una funzione, come nell'esempio seguente:

```
(define x 20)
x                       ===> 20
(define (f y) (* 2 x))
(f)                     ===> 40
(define x 100)
x                       ===> 100
(f)                     ===> 200
```

Qualunque oggetto con un identificatore può essere ridefinito. Si osservi l'esempio seguente, in cui si imbrogliano le carte e si fa in modo che l'identificatore '*****' corrisponda a una funzione che esegue la somma, mentre prima valeva per una moltiplicazione:

```
(define (* x y) (+ x y)
(* 3 5)                 ===> 8
```

Si ricorda che per modificare il contenuto di una variabile allocata, senza allocare un'altra area di memoria, si utilizza generalmente la funzione '**set!**'.

Definizione «lambda»

«

Scheme tratta gli identificatori delle funzioni (i loro nomi), nello stesso modo di quelli delle variabili. In altri termini, le funzioni sono variabili che contengono un riferimento a un blocco di codice. È possibile dichiarare una funzione attraverso la funzione '**lambda**', che restituisce la funzione stessa. In questo modo, una funzione può essere dichiarata anche attraverso l'assegnamento di una variabile, che poi diventa una funzione a tutti gli effetti.

Prima di vedere come si usa la dichiarazione di una funzione attraverso la funzione '**lambda**', è bene ribadire che, attraverso questo meccanismo, è possibile dichiarare una funzione in tutte quelle situazioni in cui è possibile inizializzare o assegnare una variabile.

```
(lambda (elenco_parametri_formali)
    corpo
)
```

Come si vede dal modello sintattico, la funzione '**lambda**' è relativamente semplice: il primo argomento è un blocco contenente l'elenco dei nomi (locali) dei parametri formali; gli argomenti successivi sono le espressioni che costituiscono il corpo della funzione. Non si dichiara il nome della funzione, dal momento che '**lambda**' restituisce la funzione stessa, che viene poi identificata (ammesso che lo si voglia fare) dalla variabile a cui questa viene assegnata.

All'inizio del «corpo» delle espressioni che descrivono il contenuto della funzione che si dichiara, si possono inserire delle dichiarazioni ulteriori attraverso la funzione '**define**'.

Sotto vengono proposti alcuni esempi che dovrebbero lasciare intendere in quante situazioni si può utilizzare una dichiarazione di funzione attraverso '**lambda**'.

```
; dichiara la variabile «f» e la inizializza temporaneamente al valore zero
(define f 0)
; assegna a «f» una funzione che esegue la somma dei suoi due argomenti
(set! f
    (lambda (x y)
        (+ x y)
    )
)
; calcola la somma tra 4 e 5, restituendo 9
(f 4 5)
```

L'esempio che appare sopra mostra in che modo si possa dichiarare una funzione in qualunque situazione in cui si può assegnare un valore a una variabile.

```
; dichiara direttamente la funzione «f»
(define f
    ; inizializza «f» con una funzione che esegue la somma
    ; dei suoi due argomenti
    (lambda (x y)
        ; corpo della dichiarazione della funzione
        (+ x y)
    )
)
; calcola la somma tra 4 e 5, restituendo 9
(f 4 5)
```

In questo caso, l'assegnamento della funzione alla variabile '**f**' è avvenuto contestualmente alla dichiarazione della variabile stessa.

```
(define moltiplica
    (lambda (x y)
        ; dichiara le variabili locali
        (define z 0)
        ; definisce un ciclo enumerativo, per il calcolo del prodotto
        ; attraverso la somma, in cui viene dichiarata implicitamente
        ; la variabile «i».
        (do ((i 1 (+ i 1)))
            ; condizione di uscita
            ((> i y))
            ; istruzioni del ciclo
            (set! z (+ z x))
        )
        ; al termine restituisce il valore contenuto nella variabile «z»
        z
    )
)
```

Questo esempio, mostra in che modo possano avvenire delle dichiarazioni locali nel corpo di una dichiarazione '**`lambda`**'.

L'esempio successivo è un po' un estremo, nel senso che viene mostrata la dichiarazione di una funzione «anonima», che viene usata immediatamente per calcolare il prodotto tra tre e quattro. Successivamente al suo utilizzo istantaneo, non c'è modo di riutilizzare tale funzione.

```
(
    ; dichiarazione della funzione anonima
    (lambda (x y)
        ; dichiara le variabili locali
        (define z 0)
        ; definisce un ciclo enumerativo, per il calcolo del prodotto
        ; attraverso la somma, in cui viene dichiarata implicitamente
        ; la variabile «i».
        (do ((i 1 (+ i 1)))
            ; condizione di uscita
            ((> i y))
            ; istruzioni del ciclo
            (set! z (+ z x))
        )
        ; al termine restituisce il valore contenuto nella variabile «z»
        z
    )
    ; indicazione del primo argomento
    3
    ; indicazione del secondo argomento
    4
)
```

Ricorsione

Si intuisce la possibilità di Scheme di scrivere funzioni ricorsive. Non dovrebbe essere difficile arrivare a questo risultato senza spiegazioni particolari. L'esempio seguente mostra il calcolo del fattoriale attraverso una funzione ricorsiva:

```
(define (fattoriale n)
    (if (= n 0)
        ; then
        1
        ; else
        (* n (fattoriale (- n 1)))
    )
)
```

Si intuisce che una funzione senza nome, come nel caso di quella dichiarata con '**`lambda`**', senza assegnarla a una variabile, non

può essere resa ricorsiva, a meno di definire una sotto-funzione ricorsiva al suo interno. L'esempio seguente è una variante di quello precedente, in cui viene utilizzata una dichiarazione '**lambda**'.

```
(define fattoriale
    (lambda (n)
        (if (= n 0)
            ; then
            1
            ; else
            (* n (fattoriale (- n 1)))
        )
    )
)
```

Funzioni «let», «let*» e «letrec»

«

Le funzioni '**let**', '**let***' e '**letrec**', hanno lo scopo di circoscrivere un ambiente, all'interno del quale può essere inserita una serie indefinita di espressioni (istruzioni), prima delle quali vengono dichiarate delle variabili il cui campo di azione è locale rispetto a quell'ambito.

```
(let ((variabile inizializzazione)...)
     corpo
)
```

```
(let* ((variabile inizializzazione)...)
     corpo
)
```

```
(letrec ((variabile inizializzazione)...)
     corpo
)
```

In tutti e tre le forme, le variabili vengono inizializzate e quindi si passa alla valutazione delle espressioni successive (le istruzioni). Alla fine, la funzione restituisce il valore dell'ultima espressione a essere stata eseguita al suo interno.

Nel caso di '**let**', le variabili vengono dichiarate e inizializzate senza un ordine preciso, ma semplicemente prima di passare alla valutazione delle espressioni successive:

```
(let ((x 1) (y 2))
    (+ x y)
)                          ===> 3
```

L'esempio non ha un grande significato da un punto di vista pratico, ma si limita a mostrare intuitivamente come si comporta la funzione '**let**'. In questo caso, vengono dichiarate le variabili locali '**x**' e '**y**', inizializzandole rispettivamente a uno e due, infine viene calcolata semplicemente la somma tra le due variabili, cosa che restituisce il valore tre.

Nel caso di '**let***', le variabili vengono dichiarate e inizializzate nell'ordine in cui sono (da sinistra a destra); pertanto, ogni inizializzazione può fare riferimento alle variabili dichiarate precedentemente nella stessa sequenza:

```
(let* ((x 1) (y (+ x 1)))
    (+ x y)
)                          ===> 3
```

L'esempio mostra che la variabile locale '**y**' viene inizializzata par-

tendo dal valore della variabile locale '**x**', incrementando il valore di questa di un'unità.

La funzione '**letrec**' è più sofisticata; il nome sta per *let recursive*. È un po' difficile spiegare il senso di questa; si tenta almeno di mostrare la cosa in modo intuitivo.

Nello stesso modo in cui si può dichiarare una variabile, si può dichiarare una funzione. In questo senso, tali dichiarazioni possono anche essere ricorsive all'interno di una funzione '**letrec**'. Viene mostrato un esempio tratto da *R*^{5}*RS*:

```
(letrec
    ; dichiara le «variabili», che in realtà sono funzioni (predicati)
    (
        ; dichiara la funzione «pari?»
        (pari?
            (lambda (n)
                (if (zero? n)
                    ; il numero è pari
                    #t
                    ; altrimenti si prova a vedere se è dispari
                    (dispari? (- n 1))
                )
            )
        )
        ; dichiara la funzione «dispari?»
        (dispari?
            (lambda (n)
                (if (zero? n)
                    ; il numero è dispari
                    #f
                    ; altrimenti si prova a vedere se è pari
                    (pari? (- n 1))
                )
            )
        )
    )
    ; fine della dichiarazione delle variabili

    ; verifica che il numero 88 è pari, chiamando la funzione
    ; «pari?» dichiarata all'inizio
```

```
    (pari? 88)
    ; la chiamata restituisce il valore #t e, di conseguenza,
    ; è questo il valore restituiti da tutto
)
```

Le variabili '**pari?**' e '**dispari?**' vengono inizializzate assegnando loro una funzione dichiarata con '**lambda**' e il loro scopo è quello di verificare che l'argomento sia rispettivamente un numero pari o dispari.

```
(pari? 2)                ===> #t
(dispari? 2)             ===> #f
```

Tali variabili e di conseguenza queste funzioni, hanno effetto solo nell'ambito della dichiarazione '**letrec**', al termine della quale diventano semplicemente irraggiungibili. Il principio di funzionamento di queste funzioni, sta nel fatto che lo zero sia pari, di conseguenza:

```
(pari? 0)                ===> #t
(dispari? 0)             ===> #f
```

Per tutti i numeri superiori, invece, è sufficiente verificare in modo ricorsivo di che tipo è il valore $n-1$. Per la precisione, se si sta verificando il fatto che un numero sia pari, se questo è superiore a zero, si può verificare che quel numero, meno uno, sia dispari, continuando così, di seguito.

Queste tre strutture sono importanti soprattutto perché consentono di inserire delle dichiarazioni di variabili o di funzioni, oltre al fatto che così circoscrivono un ambito locale per queste. Come si è visto, queste dichiarazioni possono essere fatte anche prima (anche con '**let**' e '**let***'), tenendo conto dell'ordine di valutazione che ognuna di queste strutture garantisce.

```
(let ((x 1) (y 2))
    (define messaggio "sto calcolando la somma...")
    (display messaggio)
    (newline)
    (+ x y)
)                       ===> 3
```

L'esempio che si vede sopra, è solo un'estensione di quanto già visto sopra, allo scopo di mostrare la possibilità di utilizzare la funzione '**`define`**' all'inizio del corpo di espressioni che contiene. L'esempio successivo è una variante ulteriore, in cui il messaggio viene dichiarato tra le variabili iniziali di '**`let`**'.

```
(let ((x 1) (y 2) (messaggio "sto calcolando la somma..."))
    (display messaggio)
    (newline)
    (+ x y)
)                       ===> 3
```

Scheme: liste e vettori

Scheme dispone di due strutture di dati particolari: liste e vettori. Le liste sono una sequenza di elementi a cui si accede con una certa difficoltà, senza la possibilità di utilizzare un indice, mentre i vettori sono l'equivalente degli array degli altri linguaggi.

Liste e coppie

La lista è la struttura di dati fondamentale di Scheme. In questo linguaggio, le stesse istruzioni (le chiamate delle funzioni) sono espresse in forma di lista:

```
(elemento...)
```

La lista è un elenco di elementi ordinati. Gli elementi di una lista possono essere oggetti di qualunque tipo, comprese altre liste. Ci sono molte situazioni in cui i parametri di una funzione di Scheme sono delle liste; per esempio la dichiarazione di una funzione, attraverso '**`define`**':

```
(define (nome_funzione elenco_parametri_formali)
    corpo
)
```

Come si vede, il primo parametro della funzione '**`define`**' è una lista, in cui il primo elemento è il nome della funzione che si crea, mentre gli elementi successivi sono la descrizione dei parametri formali.

Le liste vuote, sono rappresentate da una coppia di parentesi aperta e chiusa, '**`()`**', rappresentando degli oggetti speciali nella filosofia di Scheme.

Tabella u129.1. Elenco di alcune funzioni specifiche per la gestione delle stringhe.

Funzione	Descrizione
`(list?` ***oggetto*** `)`	*Vero* se l'oggetto è una lista.
`(pair?` ***oggetto*** `)`	*Vero* se l'oggetto è una coppia (una lista non vuota).
`(null?` ***lista*** `)`	*Vero* se la lista è vuota.
`(length` ***lista*** `)`	Restituisce il numero di elementi della lista.
`(car` ***lista*** `)`	Restituisce il primo elemento di una lista.

Funzione	Descrizione
`(cdr` ***lista*** `)`	Restituisce una lista da cui è stato tolto il primo elemento.
`(cadr` ***lista*** `)`	`(car (cdr` ***lista*** `))`
`(cddr` ***lista*** `)`	`(cdr (cdr` ***lista*** `))`
`(caadr` ***lista*** `)`	`(car (car (cdr` ***lista*** `)))`
`(caddr` ***lista*** `)`	`(car (cdr (cdr` ***lista*** `)))`
`(cons` ***elemento lista*** `)`	Restituisce una lista in cui inserisce al primo posto l'elemento indicato.
`(list` ***elemento*** `…)`	Restituisce una lista composta dagli elementi indicati.
`(append` ***lista lista*** `)`	Restituisce una lista composta dagli elementi delle due liste indicate.
`(reverse` ***lista*** `)`	Restituisce una lista con gli elementi in ordine inverso.
`(set-car!` ***lista oggetto*** `)`	Memorizza nella prima posizione della lista l'oggetto indicato.
`(set-cdr!` ***lista oggetto*** `)`	Memorizza nella parte successiva al primo elemento l'oggetto indicato.
`(list-tail` ***lista k*** `)`	Restituisce una lista in cui mancano i primi ***k*** elementi.
`(list-ref` ***lista k*** `)`	Restituisce l'elemento (***k*** + 1)-esimo della lista.
`(vector->list` ***vettore*** `)`	Converte il vettore in lista.
`(list->vector` ***list*** `)`	Converte la lista in vettore.

Dichiarazione di una lista

La dichiarazione di una lista avviene nello stesso modo in cui si dichiara una variabile normale:

```
(define variabile lista_costante)
```

Tuttavia, occorre tenere presente che una lista può essere interpretata come la chiamata di una funzione e come tale verrebbe intesa in questa situazione. Per evitare che ciò avvenga, la si indica attraverso un'espressione costante, cioè la si fa precedere da un apostrofo, o la si inserisce in una funzione '**quote**'. L'esempio seguente dichiara la lista '**lis**' composta dall'elenco dei numeri interi da uno a sei:

```
(define lis '(1 2 3 4 5 6))
```

In questo caso, se la lista non venisse indicata con l'apostrofo, si otterrebbe la valutazione della lista stessa, prima dell'inizializzazione della variabile '**lis**', provocando un errore, dal momento che l'oggetto '**1**' (uno) non esiste.

Caratteristiche esteriori di una lista

Le caratteristiche esteriori di una lista sono semplicemente la lunghezza, espressa in numero di elementi, e il fatto che contengano o meno qualcosa. Per verificare queste caratteristiche sono disponibili due funzioni, '**null?**' e '**length**', che richiedono come argomento una lista. Si osservino gli esempi seguenti.

```
; dichiara la lista «lis»
(define lis (1 2 3 4 5 6)

; verifica se la lista «lis» è vuota
(null? lis)                                    ===> #f

; calcola la lunghezza della lista
(length lis)                                   ===> 6
```

Se fosse stata fornita la lista in modo letterale, senza la variabile '**lis**', la stessa cosa avrebbe dovuto essere scritta nel modo seguente:

```
; verifica se la lista è vuota
(null? '(1 2 3 4 5 6))                         ===> #f

; calcola la lunghezza della lista
(length '(1 2 3 4 5 6))                        ===> 6
```

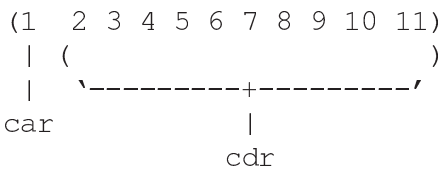
Operazioni fondamentali con le liste

L'accesso agli elementi singoli di una lista è un'impresa piuttosto complessa che si attua fondamentalmente con le funzioni '**car**' e '**cdr**'. A queste due si affianca anche '**cons**', il cui scopo è quello di «costruire» una lista.

Per comprendere il senso di queste funzioni, occorre tenere presente che per Scheme una lista è una ***coppia*** composta dal primo elemento, ovvero l'elemento '**car**', e dalla parte restante, ovvero la parte '**cdr**'.

Per la precisione, una coppia è una lista, mentre la lista vuota non è una coppia. La lista contenente un solo elemento, è la composizione dell'unico elemento a disposizione e della lista vuota.

Figura u129.5. La parte «car» e la parte «cdr» che compongono le liste di Scheme.




```
(car lista)
```

```
(cdr lista)
```

Le due funzioni ‘**car**’ e ‘**cdr**’ hanno come argomento una lista, della quale restituiscono, rispettivamente, il primo elemento e la lista restante quando si elimina il primo elemento. Si osservino gli esempi seguenti.[1]

```
(car '(1 2 3 4 5 6))            ===> 1
(cdr '(1 2 3 4 5 6))            ===> (2 3 4 5 6)
```

Data l’idea che ha Scheme sulle liste, la funzione ‘**cons**’ crea una lista a partire dalle sue parti ‘**car**’ e ‘**cdr**’:

```
(cons elemento_car lista_cdr)
```

In altri termini, ‘**cons**’ aggiunge un elemento all’inizio della lista indicata come secondo argomento. Si osservi l’esempio.

```
(cons 0 '(1 2 3 4 5 6))         ===> (0 1 2 3 4 5 6)
```

Le tre funzioni ‘**car**’, ‘**cdr**’ e ‘**cons**’ si completano a vicenda, in base alla relazione schematizzata dalla figura u129.9.

Se viene fornita una lista come primo argomento della funzione '**car**', questa viene inserita come primo elemento della lista risultante.

```
(cons '(0 1 2) '(1 2 3 4 5 6))          ===> ((0 1 2) 1 2 3 4 5 6)
```

Figura u129.9. Relazione che lega le funzioni '**car**', '**cdr**' e '**cons**'. In particolare, «x» e «y» sono liste non vuote; «a» è un elemento ipotetico di una lista.

```
(cons (car x) (cdr x)) = x
(car (cons a y)) = a
(cdr (cons a y)) = y
```

Altri modi per creare una lista sono dati dalle funzioni '**list**' e '**append**'.

(list *elemento*...)

(append *lista* *lista*)

La funzione '**list**' restituisce una lista composta dai suoi argomenti (se non si vuole che questi siano valutati prima, occorre ricordare di usare l'apostrofo); la funzione '**append**' restituisce una lista composta dagli elementi delle due liste indicate come argomento (se le liste vengono fornite in modo letterale, occorre ricordare di usare l'apostrofo, per evitare che vengano valutate come funzioni).

```
(list 1 2 3 4 5 6)                      ===> (1 2 3 4 5 6)
(append '(1 2 3 4 5 6) '(7 8 9))        ===> (1 2 3 4 5 6 7 8 9)
```

Per verificare che un oggetto sia una lista, è disponibile il predicato '**list?**'. Si osservi l'esempio seguente, con il quale si inten-

de ribadire il significato dell'apostrofo per evitare che una lista sia interpretata come funzione:

```
(define a (+ 1 2))
a                                ===> 3

(define b '(+ 1 2))
b                                ===> (+ 1 2)

(list? a)                        ===> #f
(list? b)                        ===> #t
```

Funzioni tipiche sulle liste

«

Dal momento che con le liste di Scheme non è disponibile un accesso diretto all'elemento ***n***-esimo, se non attraverso la funzione di libreria '**`list-ref`**', è importante imparare a gestire le funzioni elementari già mostrate nella sezione precedente.

- Calcolo della lunghezza di una lista:

```
(define (lunghezza x)
    (if (null? x)
        ; se la lista è vuota, restituisce zero
        0
        ; altrimenti esegue una chiamata ricorsiva
        (+ 1 (lunghezza (cdr x)))
    )
)
```

- Ricerca dell'elemento ***i***-esimo, dove il primo è il numero uno (si veda anche la funzione di libreria '**`list-ref`**', descritta più avanti in questa serie di esempi):

```
(define (i-esimo-elemento i x)
    ; «i» è l'indice, «x» è la lista
    (if (null? x)
        ; la lista è più corta di «i» elementi
        "errore: la lista è troppo corta"
        ; altrimenti procede
        (if (= i 1)
            ; se si tratta del primo elemento, basta la funzione
            ; car per prelevarlo
            (car x)
            ; altrimenti, si utilizza una chiamata ricorsiva
            (i-esimo-elemento (- i 1) (cdr x))
        )
    )
)
```

- Estrae l'ultimo elemento:

```
(define (ultimo x)
    (if (null? x)
        ; la lista è vuota e questo è un errore
        "errore: la lista è vuota"
        ; altrimenti si occupa di estrarre l'ultimo elemento
        (if (null? (cdr x))
            ; se si tratta di una lista contenente un solo elemento,
            ; restituisce il primo e unico di questa
            (car x)
            ; altrimenti utilizza una chiamata ricorsiva
            (ultimo (cdr x))
        )
    )
)
```

- Elimina l'ultimo elemento:

```
(define (elimina-ultimo x)
    (if (null? x)
        ; la lista è vuota e questo è un errore
        "errore: la lista è vuota"
        ; altrimenti si occupa di eliminare l'ultimo elemento
        (if (null? (cdr x))
            ; se si tratta di una lista contenente un solo elemento,
            ; restituisce la lista vuota
            '()
            ; altrimenti utilizza una chiamata ricorsiva per comporre
            ; una lista senza l'ultimo elemento
            (cons (car x) (elimina-ultimo (cdr x)))
        )
    )
)
```

- Restituisce la parte finale della lista, escludendo alcuni elementi iniziali. Si tratta precisamente di una funzione di libreria di Scheme, denominata '**`list-tail`**':

```
(define (list-tail x k)
    (if (zero? k)
        ; se «k» è pari a zero, viene restituita tutta la lista
        x
        ; altrimenti occorre eliminare k-1 elementi iniziali
        ; da (cdr x)
        (list-tail (cdr x) (- k 1))
    )
)
```

- Ricerca del (***k***+1)-esimo elemento di una lista. Si tratta di una funzione di libreria di Scheme, denominata '**`list-ref`**' (in pratica, l'indice ***k*** viene usato in modo da indicare il primo elemento con il numero zero):

```
(define (list-ref x k)
    ; si limita a restituire il primo elemento ottenuto
    ; dalla funzione list-tail
    (car (list-tail x k))
)
```

- Scansione di una lista in modo da restituire un'altra lista, contenente i valori restituiti dalla chiamata di una funzione data per

ogni elemento della lista. Si tratta di una semplificazione della funzione di libreria '**map**', in questo caso con la possibilità di indicare una sola lista di valori di partenza:

```
(define (map1 f x)
    ; «f» è la funzione da applicare agli elementi della lista «x»
    (if (null? x)
        ; la lista è vuota e restituisce un'altra lista vuota
        '()
        ; altrimenti compone la lista da restituire
        (cons (f (car x)) (map1 f (cdr x)))
    )
)
```

- Descrizione della funzione di libreria '**append**':

```
(define (append x y)
    (if (null? x)
        ; se la lista «x» è vuota, restituisce la lista «y»
        y
        ; altrimenti costruisce la lista in modo ricorsivo
        (cons
            (car x)
            (append (cdr x) y)
        )
    )
)
```

- Descrizione della funzione di libreria '**reverse**':

```
(define (reverse x)
    (if (null? x)
        ; se la lista «x» è vuota, non c'è nulla da invertire
        '()
        ; altrimenti compone l'inversione con una chiamata ricorsiva
        (append (reverse (cdr x)) (list (car x)))
    )
)
```

Vettori

«

Scheme gestisce anche i vettori, che sono in pratica gli array dei linguaggi di programmazione normali. Un vettore viene rappresentato in forma costante come una lista preceduta dal simbolo '**#**':

```
#(elemento_1... elemento_n)
```

L'indice dei vettori di Scheme parte da zero. Il funzionamento dei vettori di Scheme non richiede spiegazioni particolari. La tabella u129.21 riassume le funzioni utili con questo tipo di dati.

Tabella u129.21. Elenco di alcune funzioni specifiche per la gestione dei vettori.

Funzione	Descrizione
`(vector?` ***oggetto***`)`	*Vero* se l'oggetto è un vettore.
`(make-vector` ***k***`)`	Restituisce un vettore di ***k*** elementi indefiniti.
`(make-vector` ***k valore***`)`	Restituisce un vettore di ***k*** elementi inizializzati al valore specificato.
`(vector` ***elemento***`...)`	Restituisce un vettore degli elementi indicati.
`(vector-length` ***vettore***`)`	Restituisce il numero di elementi del vettore.
`(vector-ref` ***vettore k***`)`	Restituisce l'elemento nella posizione ***k***, partendo da zero.
`(vector-set!` ***vettore k oggetto***`)`	Assegna all'elemento ***k***-esimo l'oggetto indicato.
`(vector->list` ***vettore***`)`	Converte il vettore in lista.

Funzione	Descrizione
`(list->vector` ***lista***`)`	Converte la lista in vettore.

Strutture di controllo applicate alle liste

Alcune funzioni tipiche di Scheme servono ad applicare una funzione a un gruppo di valori contenuto in una lista.

Tabella u129.22. Elenco di alcune funzioni specifiche per la scansione degli elementi di una lista, allo scopo di applicarvi una funzione.

Funzione	Descrizione
`(apply` ***funzione lista***`)`	Esegue la funzione utilizzando gli elementi della lista come argomenti.
`(map` ***funzione lista***`...)`	Esegue la funzione iterativamente per gli elementi delle liste.
`(for-each` ***funzione lista***`...)`	Esegue la funzione iterativamente per gli elementi delle liste.

Funzione «apply»

```
(apply funzione lista)
```

La funzione '**`apply`**' esegue una funzione a cui affida gli elementi di una lista come altrettanti argomenti. Si osservi il modello seguente:

```
(apply funzione '(elem_1 elem_2... elem_n))
```

Questo equivale in pratica a:

```
(funzione elem_1 elem_2... elem_n)
```

Per esempio:

```
(apply + '(1 2))                    ===> 3
```

Funzione «map»

```
(map funzione lista...)
```

La funzione '**map**' scandisce una o più liste, tutte con la stessa quantità di elementi, in modo tale che, a ogni ciclo, viene passato alla funzione l'insieme ordinato dell'***i***-esimo elemento di ognuna di queste liste. La funzione restituisce una lista contenente i valori restituiti dalla funzione a ogni ciclo.

Anche se viene rispettato l'ordine delle varie liste, '**dat**' non garantisce che la scansione avvenga dal primo elemento all'ultimo.

L'esempio seguente esegue la somma di una serie di coppie di valori, restituendo la lista dei risultati:

```
(map + '(1 2 3) '(4 5 6))                    ===> (5 7 9)
```

Funzione «for-each»

```
(for-each funzione lista...)
```

La funzione '**`for-each`**' è molto simile a '**`map`**', nel senso che avvia una funzione ripetutamente, quanti sono gli elementi delle liste successive, garantendo di eseguire l'operazione in ordine, secondo la sequenza degli elementi nelle liste. Tuttavia, non restituisce nulla.

Riferimenti

«

- A. Aaby, *Scheme Tutorial*, 1996
  http://cs.wwc.edu/~cs_dept/KU/PR/Scheme.html
- *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

[1] A questo punto si intende ormai chiarito il significato dell'apostrofo posto di fronte a una lista, quando questa non deve essere valutata, prima di essere fornita come argomento di una funzione.

Scheme: I/O

Scheme ha una gestione particolare dei file. Per prima cosa, i flussi di file, che negli altri linguaggi sono dei *file handle* o semplicemente *stream*, in Scheme prendono il nome di *port*: ***porte***. Scheme distingue quindi tra porte in ingresso, in grado di «consegnare» dei caratteri, e porte in uscita, in grado di «accettare» caratteri.

Apertura e chiusura

Scheme distingue tra flussi di file in ingresso e in uscita, per cui le funzioni per aprire i file e trasformarli in porte, sono due, uno per l'apertura in lettura (ingresso) e l'altra per l'apertura in scrittura (uscita). La tabella u130.1 riassume le funzioni utili per aprire, controllare e chiudere i file. Gli esempi successivi, dovrebbero aiutare a comprenderne l'utilizzo.

Tabella u130.1. Elenco di alcune funzioni per l'apertura e la chiusura dei file, oltre che per il controllo dei flussi di file predefiniti.

Funzione	Descrizione
`(open-input-file` ***str_nome_file***`)`	Apre il file nominato e restituisce la porta in ingresso.

Funzione	Descrizione
`(open-output-file` ***str_nome_file***`)`	Apre il file nominato e restituisce la porta in uscita.
`(port?` ***oggetto***`)`	*Vero* se si tratta di una porta.
`(input-port?` ***oggetto***`)`	*Vero* se si tratta di una porta in ingresso.
`(output-port?` ***oggetto***`)`	*Vero* se si tratta di una porta in uscita.
`(close-input-port` ***porta***`)`	Chiude la porta in ingresso.
`(close-output-port` ***porta***`)`	Chiude la porta in uscita.

```
(define porta-i (open-input-file "mio_file"))

(port? porta-i)                                  ===> #t
(output-port? porta-i)                           ===> #f
(input-port? porta-i)                            ===> #t

(close-input-port porta-i)
```

In condizioni normali, sono sempre disponibili una porta in ingresso e una in uscita, in modo predefinito. Si tratta generalmente di standard input e standard output. Questi flussi di file predefiniti potrebbero essere diretti verso altri file. Tuttavia questo non viene mostrato; eventualmente si può approfondire il problema leggendo *R*^{5}*RS*.

Ingresso dei dati

L'ingresso dei dati, ovvero la lettura, avviene attraverso due funzioni fondamentali: '**`read-char`**' e '**`read`**'. La prima legge un carattere alla volta, la seconda interpreta ciò che legge in forma di dati Scheme. In pratica, '**`read`**' legge ogni volta ciò che riesce a interpretare come un oggetto per Scheme.

Tabella u130.3. Elenco di alcune funzioni per la gestione dei dati in ingresso.

Funzione	Descrizione
`(read-char)`	Legge e restituisce il carattere successivo dalla porta predefinita.
`(read-char` ***porta***`)`	Legge e restituisce il carattere successivo dalla porta indicata.
`(peek-char)`	Restituisce una copia del carattere successivo dalla porta predefinita.
`(peek-char` ***porta***`)`	Restituisce una copia del carattere successivo dalla porta indicata.
`(read)`	Legge un oggetto dalla porta predefinita.
`(read` ***porta***`)`	Legge un oggetto dalla porta indicata.
`(eof-object` ***porta***`)`	*Vero* la lettura dalla porta ha raggiunto la fine.

L'esempio seguente mostra in che modo potrebbe essere utilizzata la funzione '**`read-char`**'. Si inizia aprendo il file '`/etc/passwd`', dal quale vengono letti i primi caratteri. Si suppone che il primo record a essere letto sia quello di definizione dell'utente '**`root`**':

```
; apre il file e gli associa la porta «utenti»
(define utenti (open-input-file "/etc/passwd"))

; legge un carattere alla volta
(read-char utenti)                                  ===> #\r
(read-char utenti)                                  ===> #\o
(read-char utenti)                                  ===> #\o
(read-char utenti)                                  ===> #\t
(read-char utenti)                                  ===> #\:
;…

; chiude il file
(close-input-file utenti)
```

Nell'esempio seguente si vuole mostrare l'uso della funzione '**`read`**'. Prima si suppone di avere preparato il file seguente:

```
; prova_lettura.scm

; somma
(+ 1 2)

; moltiplicazione
(* 2 5)

; stringa
"ciao"

; valore numerico
123

; fine
```

Supponendo che il file si chiami '`prova_lettura.scm`', si osservi la sequenza di istruzioni Scheme seguente, assieme a ciò che si ottiene dalla lettura del file:

```
; apre il file e gli associa la porta «prova»
(define prova (open-input-file "prova_lettura.scm"))

; legge il primo oggetto
(read utenti)                                    ===> (+ 1 2)

; legge il secondo oggetto
(read utenti)                                    ===> (* 2 5)

; legge il terzo oggetto
(read utenti)                                    ===> "ciao"

; legge il quarto oggetto
(read utenti)                                    ===> 123

; chiude il file
(close-input-file prova)
```

Si intende l'importanza della funzione '**read**' per facilitare l'inserimento di dati nei programmi in modo interattivo.

Uscita dei dati

L'emissione dei dati, ovvero la scrittura, avviene in maniera simile alla lettura, con la stessa distinzione tra le funzioni '**write-char**' e '**write**'. Anche in questo caso, la prima scrive un carattere alla volta, mentre la seconda emette la rappresentazione di un oggetto alla volta. Tuttavia, si aggiunte un'altra funzione fondamentale: '**output**'. Questa funzione viene usata preferibilmente per mostrare dei messaggi senza codici di escape, soprattutto per non lasciare le virgolette di delimitazione delle stringhe.

Tabella u130.7. Elenco di alcune funzioni per la gestione dei dati in ingresso.

Funzione	Descrizione
`(write-char` ***carattere*** `)`	Scrive il carattere indicato attraverso la porta predefinita.
`(write-char` ***carattere porta*** `)`	Scrive il carattere indicato attraverso la porta indicata.
`(write` ***oggetto*** `)`	Scrive la rappresentazione dell'oggetto attraverso la porta predefinita.
`(write` ***oggetto porta*** `)`	Scrive la rappresentazione dell'oggetto attraverso la porta indicata.
`(display` ***oggetto*** `)`	Mostra l'oggetto attraverso la porta predefinita.
`(display` ***oggetto porta*** `)`	Mostra l'oggetto attraverso la porta indicata.
`(newline)`	Emette un codice di interruzione di riga attraverso la porta predefinita.
`(newline` ***porta*** `)`	Emette un codice di interruzione di riga attraverso la porta indicata.

L'esempio seguente dovrebbe chiarire la differenza tra la funzione '**`write`**' e '**`display`**'. Gli oggetti vengono emessi attraverso lo standard output, ovvero la porta predefinita:

```
(write (+ 1 2))                          ; visualizza «3»
(write "ciao")                           ; visualizza «"ciao"»
(write "ciao, come \"stai\"?")           ; visualizza «"ciao, come \"stai\""»
(write #\A)                              ; visualizza «#\A»
(display (+ 1 2))                        ; visualizza «3»
(display "ciao")                         ; visualizza «ciao»
(display "ciao, come \"stai\"?")         ; visualizza «ciao, come "stai"»
(display #\A)                            ; visualizza «A»
```

È già stato descritto l'uso di '**`newline`**', che è indispensabile per ottenere l'avanzamento alla riga successiva. In linea di principio, non è possibile inserire un carattere di controllo nella stringa emessa

da ‘**`write`**’ o da ‘**`display`**’.

Scheme: 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.

Problemi elementari di programmazione

«

In questa sezione vengono mostrati alcuni algoritmi elementari portati in Scheme.

Somma tra due numeri positivi

«

Il problema della somma tra due numeri positivi, attraverso l'incremento unitario, è descritto nella sezione 62.3.1.

```
; ========================================================================
; somma1.scm
; Somma esclusivamente valori positivi.
; ========================================================================

; ========================================================================
; (somma <x> <y>)
; ------------------------------------------------------------------------
(define (somma x y)
    (define z x)
    (define i 1)

    (do ()
        ((> i y))

        (set! z (+ z 1))
        (set! i (+ i 1))
    )

    z
)

; ========================================================================
; Inizio del programma.
; ------------------------------------------------------------------------
(define x 0)
(define y 0)
(define z 0)

(display "Inserisci il primo numero intero positivo: ")
(set! x (read))
(newline)
(display "Inserisci il secondo numero intero positivo: ")
(set! y (read))
(newline)
(set! z (somma x y))
(display x) (display " + ") (display y) (display " = ") (display z)
(newline)
```

```
; ========================================================================
```

In alternativa, si può modificare la funzione '**somma**', in modo che il ciclo '**do**' gestisca la dichiarazione e l'incremento delle variabili che usa. Tuttavia, in questo caso, la variabile '**z**' deve essere «copiata» in modo da poter trasmettere il risultato all'esterno del ciclo '**do**':

```
(define (somma x y)
    (define risultato 0)

    (do ((z x (+ z 1)) (i 1 (+ i 1)))
        ((> i y))
        (set! risultato z)
    )

    risultato
)
```

Volendo gestire la cosa in modo un po' più elegante, occorre togliere la variabile '**z**' dalla gestione del ciclo '**do**':

```
(define (somma x y)
    (define z x)

    (do ((i 1 (+ i 1)))
        ((> i y))
        (set! z (+ z 1))
    )

    z
)
```

Moltiplicazione di due numeri positivi attraverso la somma

«

Il problema della moltiplicazione tra due numeri positivi, attraverso la somma, è descritto nella sezione 62.3.2.

```
; ========================================================================
; moltiplica1.scm
; Moltiplica esclusivamente valori positivi.
; ========================================================================
```

```
; ============================================================================
; (moltiplica <x> <y>)
; ----------------------------------------------------------------------------
(define (moltiplica x y)
    (define z 0)
    (define i 1)

    (do ()
        ((> i y))

        (set! z (+ z x))
        (set! i (+ i 1))
    )

    z
)

; ============================================================================
; Inizio del programma.
; ----------------------------------------------------------------------------
(define x 0)
(define y 0)
(define z 0)

(display "Inserisci il primo numero intero positivo: ")
(set! x (read))
(newline)
(display "Inserisci il secondo numero intero positivo: ")
(set! y (read))
(newline)
(set! z (moltiplica x y))
(display x) (display " * ") (display y) (display " = ") (display z)
(newline)

; ============================================================================
```

In alternativa, si può modificare la funzione '**`moltiplica`**', in modo che il ciclo '**`do`**' gestisca la dichiarazione e l'incremento dell'indice '**`i`**':

```
(define (moltiplica x y)
    (define z 0)

    (do ((i 1 (+ i 1)))
        ((> i y))

        (set! z (+ z x))
    )

    z
)
```

Divisione intera tra due numeri positivi

«

Il problema della divisione tra due numeri positivi, attraverso la sottrazione, è descritto nella sezione 62.3.3.

```
; ================================================================
; dividi1.scm
; Divide esclusivamente valori positivi.
; ================================================================

; ================================================================
; (dividi <x> <y>)
; ----------------------------------------------------------------
(define (dividi x y)
    (define z 0)
    (define i x)

    (do ()
        ((< i y))

        (set! i (- i y))
        (set! z (+ z 1))
    )

    z
)

; ================================================================
; Inizio del programma.
; ----------------------------------------------------------------
(define x 0)
(define y 0)
```

```
(define z 0)

(display "Inserisci il primo numero intero positivo: ")
(set! x (read))
(newline)
(display "Inserisci il secondo numero intero positivo: ")
(set! y (read))
(newline)
(set! z (dividi x y))
(display x) (display " / ") (display y) (display " = ") (display z)
(newline)

; ============================================================================
```

In alternativa, si può modificare la funzione '**dividi**', in modo che il ciclo '**do**' gestisca la dichiarazione e il decremento della variabile '**i**'. Per la precisione, la variabile '**z**' non può essere dichiarata nello stesso modo, perché serve anche al di fuori del ciclo:

```
(define (dividi x y)
    (define z 0)

    (do ((i x (- i y)))
        ((< i y))

        (set! z (+ z 1))
    )

    z
)
```

Elevamento a potenza

Il problema dell'elevamento a potenza tra due numeri positivi, attraverso la moltiplicazione, è descritto nella sezione 62.3.4.

```
; ============================================================================
; potenza1.scm
; Eleva a potenza.
; ============================================================================
```

```
; ============================================================================
; (potenza <x> <y>)
; ----------------------------------------------------------------------------
(define (potenza x y)
    (define z 1)
    (define i 1)

    (do ()
        ((> i y))

        (set! z (* z x))
        (set! i (+ i 1))
    )

    z
)

; ============================================================================
; Inizio del programma.
; ----------------------------------------------------------------------------
(define x 0)
(define y 0)
(define z 0)

(display "Inserisci il primo numero intero positivo: ")
(set! x (read))
(newline)
(display "Inserisci il secondo numero intero positivo: ")
(set! y (read))
(newline)
(set! z (potenza x y))
(display x) (display " ** ") (display y) (display " = ") (display z)
(newline)

; ============================================================================
```

In alternativa, si può modificare la funzione '**potenza**', in modo che il ciclo '**do**' gestisca la dichiarazione e l'incremento della variabile '**i**':

```
(define (potenza x y)
    (define z 1)

    (do ((i 1 (+ i 1)))
        ((> i y))

        (set! z (* z x))
    )

    z
)
```

È possibile usare anche un algoritmo ricorsivo:

```
(define (potenza x y)
    (if (= x 0)
        0
        (if (= y 0)
            1
            (* x (potenza x (- y 1)))
        )
    )
)
```

Radice quadrata

«

Il problema della radice quadrata è descritto nella sezione 62.3.5.

```
; ================================================================
; radice1.scm
; Radice quadrata.
; ================================================================

; ================================================================
; (radice <x>)
; ----------------------------------------------------------------
(define (radice x)
    (define z -1)
    (define t 0)
    (define uscita #f)

    (do ()
        (uscita)
```

```
        (set! z (+ z 1))
        (set! t (* z z))
        (if (> t x)
            ; È stato superato il valore massimo
            (begin
                (set! z (- z 1))
                (set! uscita #t)
            )
        )
    )

    z
)

; ============================================================================
; Inizio del programma.
; ----------------------------------------------------------------------------
(define x 0)
(define z 0)

(display "Inserisci il numero intero positivo: ")
(set! x (read))
(newline)
(set! z (radice x))
(display "La radice quadrata di ") (display x) (display " è ") (display z)
(newline)

; ============================================================================
```

Fattoriale

«

Il problema del fattoriale è descritto nella sezione 62.3.6.

```
; ============================================================================
; fattoriale1.scm
; Fattoriale.
; ============================================================================

; ============================================================================
; (fattoriale <x>)
; ----------------------------------------------------------------------------
(define (fattoriale x)
    (define i (- x 1))
```

```
    (do ()
        ((<= i 0))

        (set! x (* x i))
        (set! i (- i 1))
    )

    x
)

; ============================================================================
; Inizio del programma.
; ----------------------------------------------------------------------------
(define x 0)
(define z 0)

(display "Inserisci il numero intero positivo: ")
(set! x (read))
(newline)
(set! z (fattoriale x))
(display x) (display "! = ") (display z)
(newline)

; ============================================================================
```

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

```
(define (fattoriale x)
    (if (> x 1)
        (* x (fattoriale (- x 1)))
        1
    )
)
```

Massimo comune divisore

«

Il problema del massimo comune divisore, tra due numeri positivi, è descritto nella sezione 62.3.7.

```
; ============================================================================
; mcd1.scm
; Massimo Comune Divisore.
```

```
; ============================================================================

; ============================================================================
; (moltiplica <x> <y>)
; ----------------------------------------------------------------------------
(define (mcd x y)
    (do ()
        ((= x y))

        (if (> x y)
            (set! x (- x y))
            (set! y (- y x))
        )
    )

    x
)

; ============================================================================
; Inizio del programma.
; ----------------------------------------------------------------------------
(define x 0)
(define y 0)
(define z 0)

(display "Inserisci il primo numero intero positivo: ")
(set! x (read))
(newline)
(display "Inserisci il secondo numero intero positivo: ")
(set! y (read))
(newline)
(set! z (mcd x y))
(display "MCD di ") (display x) (display " e ") (display y)
(display " è ") (display z)
(newline)

; ============================================================================
```

Numero primo

Il problema della determinazione se un numero sia primo o meno, è descritto nella sezione 62.3.8.

```
; ============================================================================
; primo1.scm
; Numero primo.
; ============================================================================

; ============================================================================
; (primo <x>)
; ----------------------------------------------------------------------------
(define (primo x)
    (define np #t)
    (define i 2)
    (define j 0)

    (do ()
        ((or (>= i x) (not np)))

        (set! j (truncate (/ x i)))
        (set! j (- x (* j i)))
        (if (= j 0)
            (set! np #f)
            (set! i (+ i 1))
        )
    )

    np
)

; ============================================================================
; Inizio del programma.
; ----------------------------------------------------------------------------
(define x 0)

(display "Inserisci un numero intero positivo: ")
(set! x (read))
(newline)
(if (primo x)
    (display "È un numero primo")
    (display "Non è un numero primo")
)
```

```
(newline)

; ========================================================================
```

Scansione di array

«

Ricerca sequenziale

«

Il problema della ricerca sequenziale all'interno di un array, è descritto nella sezione 62.4.1.

```
; ========================================================================
; ricerca_sequenziale1.scm
; Ricerca Sequenziale.
; ========================================================================

; ========================================================================
; (ricerca <vettore> <x> <ele-inf> <ele-sup>)
; ------------------------------------------------------------------------
(define (ricerca vettore x a z)
    (define risultato -1)

    (do ((i a (+ i 1)))
        ((> i z))

        (if (= x (vector-ref vettore i))
            (set! risultato i)
        )
    )
    risultato
)

; ========================================================================
; Inizio del programma.
; ------------------------------------------------------------------------

(define DIM 100)
(define vettore (make-vector DIM))
(define x 0)
(define i 0)
(define z 0)
```

```
(display "Inserire la quantità di elementi; ")
(display DIM)
(display " al massimo: ")
(set! z (read))
(newline)

(if (> z DIM)
    (set! z DIM)
)

(display "Inserire i valori del vettore.")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display "elemento ")
    (display i)
    (display " ")
    (vector-set! vettore i (read))
    (newline)
)

(display "Inserire il valore da cercare: ")
(set! x (read))
(newline)

(set! i (ricerca vettore x 0 (- z 1)))

(display "Il valore cercato si trova nell'elemento ")
(display i)
(newline)

; ================================================================
```

Esiste anche una soluzione ricorsiva che viene mostrata di seguito:

```
(define (ricerca vettore x a z)
    (if (> a z)
        ; La corrispondenza non è stata trovata.
        1
        (if (= x (vector-ref vettore a))
            a
            (ricerca vettore x (+ a 1) z)
        )
    )
)
```

Ricerca binaria

«

Il problema della ricerca binaria all'interno di un array, è descritto nella sezione 62.4.2.

```
; ================================================================
; ricerca_binaria1.scm
; Ricerca Binaria.
; ================================================================

; ================================================================
; (ricerca <vettore> <x> <ele-inf> <ele-sup>)
; ----------------------------------------------------------------
(define (ricerca vettore x a z)
    (define m (truncate (/ (+ a z) 2)))

    (if (or (< m a) (> m z))
        ; Non restano elementi da controllare: l'elemento cercato
        ; non c'è.
        -1

        (if (< x (vector-ref vettore m))
            ; Si ripete la ricerca nella parte inferiore.
            (ricerca vettore x a (- m 1))

            (if (> x (vector-ref vettore m))
                ; Si ripete la ricerca nella parte superiore.
                (ricerca vettore x (+ m 1) z)

                ; Se x è uguale a vettore[m], l'obiettivo è
                ; stato trovato.
                m
            )
```

```
        )
    )
)

; ============================================================================
; Inizio del programma.
; ----------------------------------------------------------------------------

(define DIM 100)
(define vettore (make-vector DIM))
(define x 0)
(define i 0)
(define z 0)

(display "Inserire la quantità di elementi; ")
(display DIM)
(display " al massimo: ")
(set! z (read))
(newline)

(if (> z DIM)
    (set! z DIM)
)

(display "Inserire i valori del vettore (in modo ordinato).")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display "elemento ")
    (display i)
    (display " ")
    (vector-set! vettore i (read))
    (newline)
)

(display "Inserire il valore da cercare: ")
(set! x (read))
(newline)

(set! i (ricerca vettore x 0 (- z 1)))

(display "Il valore cercato si trova nell'elemento ")
(display i)
```

```
(newline)

; =============================================================================
```

Algoritmi tradizionali

«

In questa sezione vengono mostrati alcuni algoritmi tradizionali portati in Scheme.

Bubblesort

«

Il problema del Bubblesort è stato descritto nella sezione 62.5.1. Viene mostrato prima una soluzione iterativa e successivamente la funzione '**bsort**' in versione ricorsiva.

```
; =============================================================================
; bsort1.scm
; Bubblesort.
; =============================================================================

; =============================================================================
; (ordina <vettore> <ele-inf> <ele-sup>)
; -----------------------------------------------------------------------------
(define (ordina vettore a z)
    (define scambio 0)

    (do ((j a (+ j 1)))
        ((>= j z))

        (do ((k (+ j 1) (+ k 1)))
            ((> k z))

            (if (< (vector-ref vettore k) (vector-ref vettore j))
                ; Scambia i valori.
                (begin
                    (set! scambio (vector-ref vettore k))
                    (vector-set! vettore k (vector-ref vettore j))
                    (vector-set! vettore j scambio)
                )
            )
```

```
        )
    )

    vettore
)

; ============================================================================
; Inizio del programma.
; ----------------------------------------------------------------------------

(define DIM 100)
(define vettore (make-vector DIM))
(define x 0)
(define i 0)
(define z 0)

(display "Inserire la quantità di elementi; ")
(display DIM)
(display " al massimo: ")
(set! z (read))
(newline)

(if (> z DIM)
    (set! z DIM)
)

(display "Inserire i valori del vettore.")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display "elemento ")
    (display i)
    (display " ")
    (vector-set! vettore i (read))
    (newline)
)

(set! vettore (ordina vettore 0 (- z 1)))

(display "Il vettore ordinato è il seguente: ")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))
```

```
    (display (vector-ref vettore i))
    (display " ")
)
(newline)

; ============================================================================
```

Segue la funzione '**ordina**' in versione ricorsiva:

```
(define (ordina vettore a z)
    (define scambio 0)

    (if (< a z)
        (begin
            ; Scansione interna dell'array per collocare nella
            ; posizione a l'elemento giusto.
            (do ((k (+ a 1) (+ k 1)))
                ((> k z))

                (if (< (vector-ref vettore k) (vector-ref vettore a))
                    ; Scambia i valori.
                    (begin
                        (set! scambio (vector-ref vettore k))
                        (vector-set! vettore k (vector-ref vettore a))
                        (vector-set! vettore a scambio)
                    )
                )
            )

            (set! vettore (ordina vettore (+ a 1) z))
        )
    )
    vettore
)
```

Torre di Hanoi

«

Il problema della torre di Hanoi è descritto nella sezione 62.5.3.

```
; ============================================================================
; hanoi1.scm
```

```
; Torre di Hanoi.
; ============================================================================

; ============================================================================
; (hanoi <n-anelli> <piolo-iniziale> <piolo-finale>)
; ----------------------------------------------------------------------------
(define (hanoi n p1 p2)
    (if (> n 0)
        (begin
            (hanoi (- n 1) p1 (- 6 (+ p1 p2)))
            (begin
                (display "Muovi l'anello ")
                (display n)
                (display " dal piolo ")
                (display p1)
                (display " ")
                (display p2)
                (newline)
            )
            (hanoi (- n 1) (- 6 (+ p1 p2)) p2)
        )
    )
)


; ============================================================================
; Inizio del programma.
; ----------------------------------------------------------------------------
(define n 0)
(define p1 0)
(define p2 0)

(display "Inserisci il numero di pioli: ")
(set! n (read))
(newline)
(display "Inserisci il numero del piolo iniziale (da 1 a 3): ")
(set! p1 (read))
(newline)
(display "Inserisci il numero del piolo finale (da 1 a 3): ")
(set! p2 (read))
(newline)
(hanoi n p1 p2)
```

```
; ========================================================================
```

Quicksort

«

L’algoritmo del Quicksort è stato descritto nella sezione 62.5.4.

```
; ========================================================================
; qsort1.scm
; Quicksort.
; ========================================================================

; ------------------------------------------------------------------------
; Dichiara il vettore a cui successivamente fanno riferimento tutte le
; funzioni.
; Il vettore non viene passato alle funzioni tra gli argomenti, per
; semplificare le funzioni, soprattutto nel caso di «part», che
; deve restituire anche un altro valore.
; ------------------------------------------------------------------------
(define DIM 100)
(define vettore (make-vector DIM))

; ========================================================================
; (inverti-elementi <indice-1> <indice-2>)
; ------------------------------------------------------------------------
(define (inverti-elementi a z)
    (define scambio 0)
    (set! scambio (vector-ref vettore a))
    (vector-set! vettore a (vector-ref vettore z))
    (vector-set! vettore z scambio)
)

; ========================================================================
; (part <ele-inf> <ele-sup>)
; ------------------------------------------------------------------------
(define (part a z)
    ; Si assume che «a» sia inferiore a «z».
    (define i (+ a 1))
    (define cf z)
    ; Vengono preparate delle variabili per controllare l’uscita dai cicli.
    (define uscita1 #f)
    (define uscita2 #f)
    (define uscita3 #f)
```

```
    ; Inizia il ciclo di scansione dell'array.
    (set! uscita1 #f)
    (do ()
        (uscita1)
        (set! uscita2 #f)
        (do ()
            (uscita2)

            ; Sposta «i» a destra.
            (if (or
                    (> (vector-ref vettore i) (vector-ref vettore a))
                    (>= i cf)
                )
                ; Interrompe il ciclo interno.
                (set! uscita2 #t)
                ; Altrimenti incrementa l'indice
                (set! i (+ i 1))
            )
        )
        (set! uscita3 #f)
        (do ()
            (uscita3)

            ; Sposta «cf» a sinistra.
            (if (<= (vector-ref vettore cf) (vector-ref vettore a))
                ; Interrompe il ciclo interno.
                (set! uscita3 #t)
                ; Altrimenti decrementa l'indice
                (set! cf (- cf 1))
            )
        )

        (if (<= cf i)
            ; È avvenuto l'incontro tra «i» e «cf».
            (set! uscita1 #t)
            ; Altrimenti vengono scambiati i valori.
            (begin
                (inverti-elementi i cf)
                (set! i (+ i 1))
                (set! cf (- cf 1))
            )
        )
    )
```

```
    ; A questo punto vettore[a..z] è stato ripartito e «cf» è la
    ; collocazione di vettore[a].
    (inverti-elementi a cf)

    ; A questo punto, vettore[cf] è un elemento (un valore) nella
    ; posizione giusta, e «cf» è ciò che viene restituito.
    cf
)

; ===========================================================================
; (ordina <ele-inf> <ele-sup>)
; ---------------------------------------------------------------------------
(define (ordina a z)
    ; Viene preparata la variabile «cf».
    (define cf 0)

    (if (> z a)
        (begin
            (set! cf (part a z))
            (ordina a (- cf 1))
            (ordina (+ cf 1) z)
        )
    )
)

; ===========================================================================
; Inizio del programma.
; ---------------------------------------------------------------------------

(define x 0)
(define i 0)
(define z 0)

(display "Inserire la quantità di elementi; ")
(display DIM)
(display " al massimo: ")
(set! z (read))
(newline)

(if (> z DIM)
    (set! z DIM)
)

(display "Inserire i valori del vettore.")
```

```
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display "elemento ")
    (display i)
    (display " ")
    (vector-set! vettore i (read))
    (newline)
)

; Il vettore non viene trasferito come argomento della funzione,
; ma risulta accessibile esternamente.
(ordina 0 (- z 1))

(display "Il vettore ordinato è il seguente: ")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display (vector-ref vettore i))
    (display " ")
)
(newline)

; ================================================================
```

Permutazioni

L'algoritmo ricorsivo delle permutazioni è descritto nella sezione 62.5.5.

```
; ================================================================
; permuta1.scm
; Permutazioni.
; ================================================================

; ----------------------------------------------------------------
; Dichiara il vettore a cui successivamente fanno riferimento tutte le
; funzioni.
; ----------------------------------------------------------------
(define DIM 100)
```

```
(define vettore (make-vector DIM))

; ---------------------------------------------------------------------
; Sempre per motivi pratici, rende disponibile la dimensione utilizzata
; effettivamente.
; ---------------------------------------------------------------------
(define n-elementi 0)

; =====================================================================
; (inverti-elementi <indice-1> <indice-2>)
; ---------------------------------------------------------------------
(define (inverti-elementi a z)
    (define scambio 0)
    (set! scambio (vector-ref vettore a))
    (vector-set! vettore a (vector-ref vettore z))
    (vector-set! vettore z scambio)
)

; =====================================================================
; (visualizza)
; ---------------------------------------------------------------------
(define (visualizza)
    (do ((i 0 (+ i 1)))
        ((>= i n-elementi))

        (display (vector-ref vettore i))
        (display " ")
    )
    (newline)
)

; =====================================================================
; (permuta <inizio> <fine>)
; ---------------------------------------------------------------------
(define (permuta a z)
    (define k 0)

    ; 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.
        (do ((k z (- k 1)))
            ((< k a))
```

```
            ; Scambia i valori.
            (inverti-elementi k z)

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

            ; Scambia i valori.
            (inverti-elementi k z)
        )

        ; Altrimenti, visualizza l'array e utilizza una variabile
        ; dichiarata globalmente.
        (visualizza)
    )
)

; ========================================================================
; Inizio del programma.
; ------------------------------------------------------------------------
(display "Inserire la quantità di elementi; ")
(display DIM)
(display " al massimo: ")
(set! n-elementi (read))
(newline)

(if (> n-elementi DIM)
    (set! n-elementi DIM)
)

(display "Inserire i valori del vettore.")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i n-elementi))

    (display "elemento ")
    (display i)
    (display " ")
    (vector-set! vettore i (read))
    (newline)
)

; Il vettore non viene trasferito come argomento della funzione,
```

```
; ma risulta accessibile esternamente.
(permuta 0 (- n-elementi 1))

; ========================================================================
```