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