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
```