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