EBook Gratuito

APPRENDIMENTO scheme

Free unaffiliated eBook created from
Stack Overflow contributors.

#scheme

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: scheme

It is an unofficial and free scheme ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official scheme.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con lo schema

Osservazioni

Questa sezione fornisce una panoramica su quale schema è e perché uno sviluppatore potrebbe volerlo utilizzare.

Dovrebbe anche menzionare qualsiasi grande argomento all'interno dello schema e collegarsi agli argomenti correlati. Poiché la documentazione per lo schema è nuova, potrebbe essere necessario creare versioni iniziali di tali argomenti correlati.

Examples

Installare Schema di pollo

CHICKEN è un interprete di Scheme e un compilatore con un proprio sistema di moduli di estensione chiamato "uova". È in grado di compilare Scheme in codice nativo compilando prima Scheme in C.

Installazione

Debian o Ubuntu o altre distribuzioni derivate:

```
sudo apt-get install chicken-bin
```

Fedora / RHEL / CentOS:

```
sudo yum install chicken-bin
```

Arch Linux:

```
sudo pacman -S chicken
```

Gentoo:

```
sudo emerge -av dev-scheme/chicken
```

OS X con Homebrew:

```
brew install chicken
```

OpenBSD

```
doas pkg_add -vi chicken
```

Microsoft Windows

- Installa MSYS2
- Eseguire la shell MSYS2 MinGW-w64
- Installa alcuni prerequisiti eseguendo:

  ```
  pacman -S mingw-w64-cross-toolchain base-devel mingw-w64-x86_64-gcc winpty wget
  ```

- Scarica l' ultimo tarball della versione digitando:

  ```
  wget https://code.call-cc.org/releases/current/chicken.tar.gz
  ```

- Estrai il tarball eseguendo `tar xvf chicken.tar.gz`
- Immettere la directory estratta, ad esempio digitando `cd chicken-4.11.0`
- Esegui `make PLATFORM=mingw-msys install`

Se hai problemi nell'esecuzione di `csi` , prova invece a eseguire `winpty csi`

Usando POLLO

Per utilizzare il REPL di schema di pollo, digitare `csi` nella riga di comando.

Per compilare un programma Scheme utilizzando CHICKEN, eseguire `csc program.scm` , che creerà un `program` denominato eseguibile nella directory corrente.

Installazione dei moduli

Chicken Scheme ha molti moduli che possono essere sfogliati nell'indice delle uova . Le uova sono moduli di schema che verranno scaricati e quindi compilati per schema di pollame. In alcuni casi, potrebbe essere necessario installare dipendenze esterne usando il solito gestore di pacchetti.

Si installa le uova scelte con questo comando:

```
sudo chicken-install [name of egg]
```

Fare uso della REPL

Si potrebbe desiderare di aggiungere il supporto `readline` al proprio REPL per fare in modo che la modifica della riga in `csi` comporti più come ci si potrebbe aspettare.

Per fare ciò, eseguire `sudo chicken-install readline` , e quindi creare un file chiamato `~/.csirc` con i seguenti contenuti:

```
(use readline)
(current-input-port (make-readline-port))
(install-history-file #f "/.csi.history")
```

Installare mit-scheme

Di seguito sono riportati alcuni esempi su come installare Schema MIT / GNU :

Installazione di Debian / Ubuntu:

```
sudo apt-get install mit-scheme
```

Installazione manuale:

Scarica il binario Unix direttamente dal Progetto GNU , quindi segui le istruzioni dalla pagina web ufficiale :

```
# Unpack the tar file
tar xzf mit-scheme.tar.gz

# move into the directory
cd mit-scheme/src

# configure the software
./configure
```

Per impostazione predefinita, il software verrà installato in `/usr/local` , nelle cartelle sottodirectory bin e lib. Se vuoi installarlo da qualche altra parte, ad esempio `/opt/mit-scheme` , passa l'opzione `--prefix` allo script configure, come in `./configure --prefix=/opt/mit-scheme` .

Lo script configure accetta tutti gli argomenti normali per tali script e accetta anche alcuni specifici per Schema MIT / GNU. Per vedere tutti gli argomenti possibili e i loro significati, eseguire il comando `./configure --help` .

```
# build
make compile-microcode

# compile
make install # may require super-user permissions (Depending on configuration)
```

Windows 7 :

L' eseguibile autoinstallante può essere trovato nel sito web ufficiale .

> Schema MIT / GNU è distribuito come eseguibile autoinstallante. L'installazione del software è semplice. Basta eseguire il file scaricato e rispondere alle domande dell'installatore. Il programma di installazione ti consentirà di scegliere la directory in cui deve essere installato MIT / GNU Scheme e il nome della cartella in cui devono essere posizionate le scorciatoie.

Leggi Iniziare con lo schema online: https://riptutorial.com/it/scheme/topic/851/iniziare-con-lo-schema

Capitolo 2: Implementazione di diversi algoritmi di ordinamento

Examples

quicksort

Quicksort è un algoritmo di ordinamento comune con una complessità del caso medio di `O(n log n)` e una complessità del caso peggiore di `O(n^2)` . Il suo vantaggio rispetto ad altri metodi `O(n log n)` è che può essere eseguito sul posto.

Quicksort divide l'input su un valore pivot scelto, separando l'elenco in quei valori che sono inferiori a e quei valori che sono maggiori di (o uguali a) il pivot. Dividere la lista è facile con il `filter` .

Usando questo, l'implementazione di Schema di Quicksort potrebbe essere simile alla seguente:

```
(define (quicksort lst)
  (cond
    ((or (null? lst) ; empty list is sorted
         (null? (cdr lst))) ; single-element list is sorted
     lst)
    (else
      (let ((pivot (car lst)) ; Select the first element as the pivot
            (rest (cdr lst)))
        (append
          (quicksort ; Recursively sort the list of smaller values
            (filter (lambda (x) (< x pivot)) rest)) ; Select the smaller values
          (list pivot) ; Add the pivot in the middle
          (quicksort ; Recursively sort the list of larger values
            (filter (lambda (x) (>= x pivot)) rest)))))))) ; Select the larger and equal values
```

Unisci Ordina

Unisci ordinamento è un algoritmo di ordinamento comune con una complessità del caso medio di `O(n log n)` e una complessità del caso peggiore di `O(n log n)` . Sebbene non possa essere eseguito sul posto, garantisce la complessità di `O(n log n)` in tutti i casi.

Unisci Ordina divide ripetutamente l'input in due, finché non viene raggiunto un elenco vuoto o un elenco di elementi singoli. Avendo raggiunto la parte inferiore dell'albero di divisione, viene ripristinato, unendo le due suddivisioni ordinate l'una nell'altra, finché non viene lasciato un singolo elenco ordinato.

Utilizzando questo, un'implementazione Scheme di Merge Sort può essere simile alla seguente:

```
;; Merge two sorted lists into a single sorted list
(define (merge list1 list2)
  (cond
```

```
    ((null? list1)
     list2)
    ((null? list2)
     list1)
    (else
      (let ((head1 (car list1))
            (head2 (car list2)))
        ; Add the smaller element to the front of the merge list
        (if (<= head1 head2)
          (cons
            head1
            ; Recursively merge
            (merge (cdr list1) list2))
          (cons
            head2
            ; Recursively merge
            (merge list1 (cdr list2)))))))))

(define (split-list lst)
  (let ((half (quotient (length lst) 2)))
    ; Create a pair of the first and second halves of the list
    (cons
      (take lst half)
      (drop lst half))))

(define (merge-sort lst)
  (cond
    ((or (null? lst) ; empty list is sorted, so merge up
         (null? (cdr lst))) ; single-element list is sorted, so merge up
     lst)
    (else
      (let ((halves (split-list lst)))
        ; Recursively split until the bottom, then merge back up to sort
        (merge (merge-sort (car halves))
               (merge-sort (cdr halves)))))))
```

Leggi Implementazione di diversi algoritmi di ordinamento online:
https://riptutorial.com/it/scheme/topic/3191/implementazione-di-diversi-algoritmi-di-ordinamento

Capitolo 3: Input Output in Scheme

introduzione

Input e Output nello schema vengono generalmente gestiti attraverso le porte. Una porta è una struttura dati che viene utilizzata per interagire con il mondo esterno a Scheme. Una porta non è limitata ai file, ma può essere utilizzata per leggere / scrivere sui socket. In qualche modo, l'oggetto port è una sorta di oggetto universale che non può solo manipolare file e socket, ma qualsiasi tipo di operazione di lettura / scrittura con il sistema operativo. Ad esempio, si potrebbe implementare una porta che può scrivere su una stampante o persino controllare una macchina CNC da Scheme usando una porta.

Examples

Crea una porta di input

Una porta di input può essere creata in molti modi, ma di solito il metodo inizia con `open-input-` .

Porta stringa

Puoi usare una stringa come porta usando `open-input-string` . Creerà una porta che sarà in grado di leggere dalla stringa.

```
(define p
  (open-input-string "(a . (b . (c . ()))) 34"))
```

Porta file

È possibile aprire un file per la lettura con `open-input-file` .

```
(define p
  (open-input-file "path/to/file"))
```

Leggi da una porta di input

La lettura da una porta di input può essere eseguita in molti modi. Possiamo usare il metodo di `read` usato dalla REPL. Legge e interpreta le espressioni separate dallo spazio.

Prendendo l'esempio dalla porta delle stringhe sopra. Possiamo leggere dal porto in questo modo:

```
(define p
  (open-input-string "(a . (b . (c . ()))) 34"))
(read p) -> (a b c)
```

```
(read p) -> 34
```

Possiamo leggere da una porta come `char` usando il metodo speciale `read-char` . Ciò restituirà un singolo carattere dalla porta da cui stiamo leggendo.

```
(define p (open-input-string "hello"))
(read-char p) -> #\h
```

Leggi Input Output in Scheme online: https://riptutorial.com/it/scheme/topic/8188/input-output-in-scheme

Capitolo 4: Macro di schema

Examples

Macro igieniche e referenzialmente trasparenti con regole di sintassi

Il maggior vantaggio di LISP e Scheme rispetto ad altri linguaggi di programmazione tradizionali è il loro sistema macro. A differenza del preprocessore C e di altri linguaggi macro, le macro Scheme prendono il codice analizzato come input e restituiscono il codice espanso come output. Questa è una delle applicazioni della frase "codice è dati" di Scheme, ed è ciò che rende il linguaggio così potente.

Le macro in Scheme sono create con la `define-syntax` , che può definire una macro in un certo numero di modi. Il metodo più semplice consiste nell'utilizzare le `syntax-rules` , che utilizzano la corrispondenza del modello per trasformare il codice di input nel codice di output.

Questo esempio crea un `for item in list` semplice `for item in list` e `for list as item` sintassi `for list as item` per il loop sugli elementi di un elenco:

```
(define-syntax for
  (syntax-rules (in as) ; 'in' and 'as' keywords must match in the pattern
    ; When the 'for' macro is called, try matching this pattern
    ((for element in list
          body ...) ; Match one or more body expressions
     ; Transform the input code
     (for-each (lambda (element)
                 body ...)
               list))
    ; Try matching another pattern if the first fails
    ((for list as element
          body ...)
     ; Use the existing macro for the transform
     (for element in list
          body ...))))
```

Questi due macro possono quindi essere utilizzati come segue, fornendo uno stile più imperativo:

```
(let ((names '(Alice Bob Eve)))
  (for name in names
    (display "Hello ")
    (display name)
    (newline))
  (for names as name
    (display "name: ")
    (display name)
    (newline)))
```

L'esecuzione del codice fornirà l'output previsto:

```
Hello Alice
Hello Bob
```

```
Hello Eve
name: Alice
name: Bob
name: Eve
```

L'errore più comune a cui prestare attenzione non è il passaggio dei valori corretti a una macro, che spesso si tradurrà in un messaggio di errore inutile che si applica al modulo espanso anziché alla chiamata macro.

Il `for` definizione della sintassi sopra non controllare se sono passati un identificatore e una lista, in modo da passare qualsiasi altro tipo si tradurrà in un errore che punta al `for-each` chiamata invece della `for` chiamata. Il debugging elimina lo scopo della macro, quindi spetta all'utente inserire i controlli e segnalare errori di utilizzo, che possono essere rilevati in fase di compilazione.

Leggi Macro di schema online: https://riptutorial.com/it/scheme/topic/3024/macro-di-schema

Capitolo 5: Pairs

introduzione

Una coppia è uno dei tipi di dati più basilari nello schema. Di solito si chiama anche contro le cellule.

Examples

Crea una coppia

Una coppia può essere creata con la funzione `cons` . Il nome della funzione sta per *costruttore* . In Scheme, tutto è praticamente basato su coppie.

```
(cons a b)
```

La funzione restituisce una coppia contenente l'elemento `a` e `b` . Il primo parametro di `cons` è chiamato `car` (Content Address Register) e il secondo argomento è il `cdr` (Content Decrement Register).

Accedi alla macchina della coppia.

È possibile accedere ai dati nella coppia con le funzioni di utilità. Per accedere alla `car` , dobbiamo usare la funzione `car` .

```
(car (cons a b))
> a
```

Inoltre possiamo verificare la seguente uguaglianza:

```
(eq? a (car (cons a b)))
> #t
```

Accedi al cdr della coppia

Per accedere al cdr, dobbiamo usare la funzione cdr.

(cdr (cons ab))

    B

Inoltre possiamo verificare la seguente uguaglianza:

(eq? b (cdr (cons ab)))

    #t

Crea una lista con coppie

L'elenco nello schema non è altro che una serie di coppie annidate l'una nell'altra nel `cdr` di una `cons` . E l'ultimo `cdr` di una lista corretta è la lista vuota `'()` .

Per creare la lista `(1 2 3 4)` , avremmo qualcosa di simile a questo:

```
(cons 4 '())
> (4)
(cons 3 (cons 4 '()))
> (3 4)
(cons 2 (cons 3 (cons 4 '())))
> (2 3 4)
(cons 1 (cons 2 (cons 3 (cons 4 '()))))
> (1 2 3 4)
```

Come puoi vedere, un elenco in schema è un elenco collegato fatto di coppie. Per questo motivo, l'aggiunta di un oggetto in primo piano non richiede quasi tempo, ma l'aggiunta di un elemento alla fine dell'elenco costringe l'interprete a camminare sull'intero elenco.

Leggi Pairs online: https://riptutorial.com/it/scheme/topic/8190/pairs

Capitolo 6: Sintassi

Examples

S-Expression

Un'espressione in Scheme è ciò che verrà eseguito. Un'espressione S, come di solito viene chiamata inizia con a `(` e termina con a `)`. Il primo membro dell'espressione è ciò che verrà eseguito. Il seguente membro dell'espressione sono i parametri che verranno inviati all'espressione durante la valutazione dell'espressione.

Ad esempio aggiungendo numeri:

```
(+ 1 2 3)
```

In questo caso, `+` è un simbolo per una funzione di *aggiunta* che richiede più parametri. `1`, `2` e `3` vengono inviati alla funzione `+`.

S-Expression può contenere S-Expressions come parametri come mostrato nel seguente esempio:

```
(if (< x y)
  x
  y)
```

Che può essere letto come se `x` fosse minore di `y` restituisca `x` altrimenti ritorna `y`. In questo esempio valutiamo l'espressione della condizione, a seconda del valore risolto, verrà restituito x o y. Potrebbe essere valutato a questo

```
(if #t x y)
x
(if #f x y)
y
```

Un esempio meno ovvio per i principianti è di avere un'espressione S come parte del primo membro di una S-Expression. In questo modo, possiamo cambiare il comportamento di un metodo cambiando la funzione che verrà chiamata senza dover creare rami con gli stessi parametri. Ecco un rapido esempio di un'espressione che può aggiungere o sottrarre numeri se x è sotto y.

```
((if (< x y) + -)
  1 2 3)
```

Se `x` è sotto `y`, l'espressione sarà valutata come:

```
(+ 1 2 3)
6
```

altrimenti

```
(- 1 2 3)
-4
```

Come potete vedere, Scheme consente al programmatore di creare una parte di codice complessa mentre fornisce al programmatore gli strumenti per prevenire la duplicazione del codice. In altre lingue potremmo vedere lo stesso esempio scritto come tale:

(if (<xy) (+ 1 2 3) (- 1 2 3))

Il problema con questo metodo è che duplichiamo molto codice mentre l'unica cosa che cambia è il metodo che viene chiamato. Questo esempio è abbastanza semplice ma con più condizioni potremmo vedere molte linee simili duplicate.

Lasciare macro semplice

Le espressioni let in scheme sono in effetti macro. Possono essere espressi con lambda. Un semplice let potrebbe assomigliare a questo:

```
(let ((x 1) (y 2))
  (+ x y))
```

Restituirà 3 come viene restituito il valore dell'ultima espressione del corpo let. Come puoi vedere, un'espressione let è in realtà in esecuzione qualcosa. Se traduciamo questa parte di codice con lambdas, otterremmo qualcosa del genere:

```
((lambda (x y) (+ x y)) 1 2)
```

Qui possiamo vedere che stiamo chiamando l'anonymous lambda con 1 e 2 direttamente. Quindi il risultato in questo caso è anche 3.

Con questo in mente, capiamo che un'espressione let è composta da 2 parti. Ha dei parametri e un corpo come un lambda, ma la differenza è che lascia che le espressioni vengano chiamate dopo la loro valutazione.

Per spiegare come un'espressione let funziona da una vista astratta a una vista concreta, sarebbe simile a questa.

```
(let params body ...)
(let (param1 param2 ...) body ...)
(let ((p1 val1) (p2 val2) ...) body ...)
```

I parametri sono una lista di coppie di `(name value)` da utilizzare nel corpo del `let` .

Perché usare lasciare espressione?

Le espressioni sono particolarmente utili per memorizzare le variabili in un metodo proprio come le inizializzazioni della variabile in c come i linguaggi. È favorevole all'uso di `define` perché fuori

dall'espressione let, le variabili sono sparite ... L'uso di un define è in realtà l'aggiunta di una variabile all'ambiente di esecuzione corrente. Le variabili aggiunte all'ambiente globale non possono essere rimosse. Lascia che l'espressione sia sicura da usare ovunque. Può anche essere usato per ghostare le variabili senza toccare gli ambiti genitore.

Per esempio:

```
(let ((x 1))
  (let ((x 2) (y x))
    (display x)
    (display y))
  (display x))
```

Stamperà:

```
2
1
1
```

In questo caso, `x` è definito con 1, quindi ghostato dalla `x` nella seconda `let` con il valore `2` . La variabile `y` viene iniziata con il valore `x` dell'ambito principale. Dopo che l'espressione `let` interna è stata eseguita, mostra il valore iniziale di `x` con 1. L'espressione inner `let` non ha modificato il valore dell'ambito genitore.

Ogni volta che hai bisogno di inizializzare le variabili, dovresti usare let espressioni come questa:

```
(let (
  (user (get-current-user session))
  (db (get-current-db session))
  (ids (get-active-ids session))
  )
  (mark-task-completed db user ids)
  (change-last-logged db user)
  (db-commit db))
```

Qui in questo esempio, le variabili vengono inizializzate e utilizzate più volte nel blocco di codice. E quando l'espressione let è terminata, le variabili vengono automaticamente liberate poiché non sono più necessarie.

Sintassi punteggiata per coppie

Esiste una sintassi particolare che ci consente di scrivere `cons` cella in un modo più compatto rispetto all'uso del `cons` constr.

Una coppia può essere scritta come tale:

```
'(1 . 2) == (cons 1 2)
```

La grande differenza è che possiamo creare `pairs` usando la citazione. In caso contrario, Scheme creerebbe un elenco appropriato `(1 . (2 . '()))` .

La sintassi del punto forza l'espressione ad avere solo 2 membri. Ogni membro può essere di qualsiasi tipo, incluse le coppie.

```
'(1 . (2 . (3 . 4)))
> (1 2 3 . 4)
```

Si noti che l'elenco improprio dovrebbe essere visualizzato con un punto alla fine per mostrare che il `cdr` dell'ultima coppia della lista non è la lista vuota `'()` .

Questo modo di mostrare le liste è un po 'confuso poiché la seguente espressione non sarebbe espressa come ci si aspetterebbe.

```
'((1 . 2) . ( 3 . 4))
> ((1 . 2) 3 . 4)
```

Poiché la lista di solito salta il `.` , il primo argomento della lista sarebbe `(1 . 2)` , il secondo argomento sarebbe `3` ma poiché l'elenco è errato, l'ultimo `.` viene mostrato per mostrare che l'ultimo elemento della lista non è `'()` . Anche se si pensa, i dati sono mostrati in un modo diverso, i dati interni sono come è stato creato.

Leggi Sintassi online: https://riptutorial.com/it/scheme/topic/5989/sintassi

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con lo schema	Candy Gumdrop, Community, eyqs, Loïc Faure-Lacroix, Reut Sharabani, Will Ness
2	Implementazione di diversi algoritmi di ordinamento	Billy Brown
3	Input Output in Scheme	Loïc Faure-Lacroix
4	Macro di schema	Billy Brown
5	Pairs	Loïc Faure-Lacroix
6	Sintassi	Loïc Faure-Lacroix